The Julia Language

The Julia Project

January 15, 2018

Contents

Part I

Home

Part II

Julia Documentation

Chapter 1

Manual

Chapter 2

Biblioteca Estándar

- Esenciales
- Colecciones y Estructuras de Datos
- Matemáticas
- Números
- Cadenas
- Arrays
- Tareas y Computación Paralela
- Álgebra Lineal
- Constantes
- Sistema de Ficheros
- E/S y Redes
- Puntuación
- Ordenación y Funciones Relacionadas
- Funciones del Gestor de Paquetes
- Fechas y Hora
- Utilidades de Iteración
- Realizando Pruebas Unitarias
- Interfaz C
- Librería Estándar C
- Enlazador Dinámico
- Realización de Perfiles
- *StackTraces*
- Soporte SIMD

Chapter 3

Documentación para Desarrolladores

- Reflexión e Introspección
- Documentación de los Interiores de Julia
  - Initialization of the Julia runtime
  - ASTs de Julia
  - Más sobre Tipos
  - Memory layout of Julia Objects
  - Evaluación de Código Julia
  - Convenios de Llamada
  - High-level Overview of the Native-Code Generation Process
  - Funciones Julia
  - Base.Cartesian
  - Hablando al Compilador (El Mecanismo `:meta`)
  - SubArrays
  - Construcción de Imagen del Sistema
  - Trabajando con LLVM
  - printf() and stdio in the Julia runtime
  - Comprobación de Límites
  - Proper maintenance and care of multi-threading locks
  - Arrays with custom indices
  - Base.LibGit2
  - Module loading
- Desarrollo/Depuración de Código C de Julia
  - Reporting and analyzing crashes (segfaults)
  - gdb debugging tips
  - Using Valgrind with Julia
  - Sanitizer support

Part III

Manual

Chapter 4

Introducción

La computación científica ha requerido tradicionalmente el máximo rendimiento, aunque los expertos de los distintos dominios se hayan movido en gran parte a los lenguajes dinámicos, más lentos, para el trabajo diario. Creemos que hay muchas buenas razones para preferir lenguajes dinámicos para estas aplicaciones, y no esperamos que su uso disminuya. Afortunadamente, el diseño de lenguajes y las técnicas de compilación modernos permiten casi eliminar el compromiso del rendimiento y proporcionar un solo entorno suficientemente productivo para la creación de prototipos y suficientemente eficiente para implementar aplicaciones de alto rendimiento. El lenguaje de programación de Julia cumple este papel: es un lenguaje dinámico y flexible, apropiado para la computación científica y numérica, con un rendimiento comparable al de los lenguajes tradicionales de tipo estático.

Debido a que el compilador de Julia es diferente de los intérpretes utilizados para lenguajes como Python o R, podría parecer al principio que el rendimiento de Julia no es intuitivo. Si encuentra que algo es lento, le recomendamos que lea la sección Consejos de Rendimiento antes de intentar otra cosa. Una vez que entienda cómo funciona Julia, será fácil escribir código casi tan rápido como el código C.

Julia ofrece tipado opcional, despacho múltiple, y buen desempeño, logrado usando inferencia de tipos y compilación *just-in-time* (JIT), implementada usando LLVM. Es multi-paradigma, combinando características de programación imperativa, funcional y orientada a objetos. Julia proporciona facilidad y expresividad para la computación numérica de alto nivel, de la misma manera que idiomas como R, MATLAB y Python, pero también soporta la programación general. Para conseguirlo, Julia se basa en el linaje de los lenguajes de programación matemáticos, pero también toma prestado mucho de los lenguajes dinámicos populares, incluyendo Lisp, Perl, Python, Lua, y Ruby.

Las diferencias más significativas de Julia de los lenguajes dinámicos típicos son:

- El lenguaje básico impone muy poco; La biblioteca estándar se ha escrito en el propio Julia, incluyendo operaciones primitivas como la aritmética entera.
- Un lenguaje enriquecido de tipos para construir y describir objetos, que también se puede utilizar opcionalmente para hacer declaraciones de tipo.
- La capacidad de definir el comportamiento de la función a través de muchas combinaciones de tipos de argumentos mediante el despacho múltiple.
- Generación automática de código eficiente y especializado para diferentes tipos de argumentos.
- Buen rendimiento, aproximándose al de los lenguajes compilados estáticamente como C.

Aunque a veces se dice de los lenguajes dinámicos que son lenguajes ”sin tipo”, ésto no es cierto en absoluto: cada objeto, ya sea primitivo o definido por el usuario, tiene un tipo. La falta de declaraciones de tipos en la mayoría de los lenguajes dinámicos, sin embargo, significa que uno no puede instruir al compilador acerca de los tipos de valores

y, a menudo, no puede hablar explícitamente de tipos en absoluto. En lenguajes estáticos, por otro lado, aunque uno puede -y normalmente debe- anotar tipos para el compilador, los tipos sólo existen en tiempo de compilación y no pueden ser manipulados o expresados en tiempo de ejecución. En Julia, los tipos son objetos en tiempo de ejecución y también se pueden utilizar para transmitir información al compilador.

Aunque el programador casual no necesita usar explícitamente los tipos o el despacho múltiple, ellas son las características centrales unificadoras de Julia: las funciones se definen en diferentes combinaciones de tipos de argumentos y se aplican despachando a la definición concordante más específica. Este modelo se ajusta bien a la programación matemática, donde no es natural que el primer argumento ”posea” una operación como en la programación orientada a objetos tradicional. Los operadores son sólo funciones con notación especial - para ampliar la adición a nuevos tipos de datos definidos por el usuario, se definen nuevos métodos para la función +. El código existente se aplica sin problemas a los nuevos tipos de datos.

En parte debido a la inferencia de tipo en tiempo de ejecución (aumentada por anotaciones de tipos opcionales), y en parte debido a enfoque muy basado en el rendimiento desde el inicio del proyecto, la eficiencia computacional de Julia supera la de otros lenguajes dinámicos e incluso rivaliza con la de lenguajes de compilación estática. Para los problemas numéricos a gran escala, la velocidad siempre ha sido, continúa siendo, y probablemente siempre será crucial: la cantidad de datos procesados se ha mantenido fácilmente al ritmo de la Ley de Moore durante las últimas décadas.

Julia tiene como objetivo crear una combinación sin precedentes de facilidad de uso, potencia y eficiencia en un solo lenguaje de programación. Además de lo anterior, algunas ventajas de Julia sobre sistemas comparables son:

- Libre y de código abierto (con licencia MIT)
- Los tipos definidos por el usuario son tan rápidos y compactos como los predefinidos.
- No hay necesidad de vectorizar código para el rendimiento; el código devectorizado es rápido
- Diseñado para el paralelismo y la computación distribuida.
- Hilos ”verdes” de peso ligero (coroutinas).
- Sistema de tipos discreto pero potente.
- Conversiones y promociones elegantes y extensibles para números y otros tipos.
- Soporte eficiente para Unicode, incluyendo pero no limitado a UTF-8
- Llamada a las funciones C directamente (no se necesitan envolturas o API especiales).
- Poderosas capacidades tipo shell para administrar otros procesos.
- Macros similares a Lisp y otras instalaciones de metaprogramación.

Chapter 5

Empezando

La instalación de Julia es sencilla, ya sea utilizando binarios precompilados o compilando desde la fuente. Descargue e instale Julia siguiendo las instrucciones disponibles en https://julialang.org/downloads/.

La forma más fácil de aprender y experimentar con Julia es iniciando una sesión interactiva (también conocida como *read-eval-print loop* o "REPL") haciendo doble clic en el ejecutable de Julia o ejecutando `julia` desde la línea de mandatos:

```
$ julia
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: https://docs.julialang.org
   _ _   _| |_  __ _   |  Type "?help" for help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.5.0-dev+2440 (2016-02-01 02:22 UTC)
 _/ |\__'_|_|_|\__'_|  |  Commit 2bb94d6 (11 days old master)
|__/                   |  x86_64-apple-darwin13.1.0

julia> 1 + 2
3

julia> ans
3
```

Para salir de la sesión interactiva, escriba `^D` (la tecla de control junto con la tecla `D`) o escriba `quit()`. Cuando se ejecuta en modo interactivo, Julia muestra un banner y solicita al usuario la entrada. Una vez que el usuario ha introducido una expresión completa, como `1 + 2`, y pulsa *Enter*, la sesión interactiva evalúa la expresión y muestra su valor. Si se introduce una expresión en una sesión interactiva con un punto y coma al final, no se muestra su valor. La variable `ans` está enlazada al valor de la última expresión evaluada, sea mostrada o no. La variable `ans` sólo está enlazada a las sesiones interactivas, no cuando el código Julia se ejecuta de otras maneras.

Para evaluar expresiones escritas en un archivo de origen `file.jl`, escriba `include ("file.jl")`.

Para ejecutar código en un archivo de forma no interactiva, puede darlo como el primer argumento al mandato Julia:

```
$ julia script.jl arg1 arg2...
```

Como indica el ejemplo, los siguientes argumentos de línea de mandatos de Julia se toman como argumentos de línea de mandatos al programa `script.jl` del programa, pasados a través de la constante global `ARGS`. El nombre del propio *script* se pasa como la variable global `PROGRAM_FILE`. Tenga en cuenta que `ARGS` también se establece cuando se da el código de script usando la opción `-e` en la línea de órdenes (vea la salida de ayuda de `julia` más abajo) pero `PROGRAM_FILE` estará vacío. Por ejemplo, para imprimir los argumentos que se le dan a un script, puede hacer esto:

```
$ julia -e 'println(PROGRAM_FILE); for x in ARGS; println(x); end' foo bar

foo
bar
```

O puede poner ese código en un script y ejecutarlo:

```
$ echo 'println(PROGRAM_FILE); for x in ARGS; println(x); end' > script.jl
$ julia script.jl foo bar
script.jl
foo
bar
```

El delimitador `--` puede usarse para separar argumentos en línea de mandatos al fichero del *script* a los argumentos de Julia:

```
$ julia --color=yes -O -- foo.jl arg1 arg2..
```

Julia se puede iniciar en modo paralelo con las opciones `-p` o `--machinefile`. `-p n` pondrá en marcha un n procesos *worker* adicionales, mientras que `--machinefile archivo` iniciará un *worker* para cada línea en el archivo de archivo. Las máquinas definidas en el archivo deben ser accesibles a través de un login ssh sin contraseña, con Julia instalado en la misma ubicación que el *host* actual. Cada definición de máquina toma la forma `[count *] [user @] host [: port] [bind_addr [: port]]`. El valor por defecto de `user` es el usuario actual, y el de `port` el puerto ssh estándar. Las variables opcionales `bind_to bind_addr [: port]` especifican la dirección IP y el puerto que otros *workers* deberían usar para conectarse a este *worker*.

Si tiene código que desea ejecutar cada vez que Julia se inicia, puede ponerlo en `~/.juliarc.jl`:

```
$ echo 'println("Greetings! ! ?")' > ~/.juliarc.jl
$ julia
Greetings! ! ?

...
```

Hay varias formas de ejecutar el código Julia y proporcionar opciones, similares a las disponibles para los programas `perl` y `ruby`:

```
julia [switches] -- [programfile] [args...]
 -v, --version             Display version information
 -h, --help                Print this message

 -J, --sysimage <file>     Start up with the given system image file
 --precompiled={yes|no}    Use precompiled code from system image if available
 --compilecache={yes|no}   Enable/disable incremental precompilation of modules
 -H, --home <dir>          Set location of `julia` executable
 --startup-file={yes|no}   Load ~/.juliarc.jl
 --handle-signals={yes|no} Enable or disable Julia's default signal handlers

 -e, --eval <expr>         Evaluate <expr>
 -E, --print <expr>        Evaluate and show <expr>
 -L, --load <file>         Load <file> immediately on all processors

 -p, --procs {N|auto}      Integer value N launches N additional local worker processes
                           "auto" launches as many workers as the number of local cores
 --machinefile <file>      Run processes on hosts listed in <file>

 -i                        Interactive mode; REPL runs and isinteractive() is true
```

```
-q, --quiet               Quiet startup (no banner)
--color={yes|no}          Enable or disable color text
--history-file={yes|no}   Load or save history

--compile={yes|no|all|min}Enable or disable JIT compiler, or request exhaustive compilation
-C, --cpu-target <target> Limit usage of cpu features up to <target>
-O, --optimize={0,1,2,3}  Set the optimization level (default is 2 if unspecified or 3 if
    specified as -O)
-g, -g <level>            Enable / Set the level of debug info generation (default is 1 if
    unspecified or 2 if specified as -g)
--inline={yes|no}         Control whether inlining is permitted (overrides functions declared as
     @inline)
--check-bounds={yes|no}   Emit bounds checks always or never (ignoring declarations)
--math-mode={ieee,fast}   Disallow or enable unsafe floating point optimizations (overrides
    @fastmath declaration)

--depwarn={yes|no|error}  Enable or disable syntax and method deprecation warnings ("error"
    turns warnings into errors)

--output-o name           Generate an object file (including system image data)
--output-ji name          Generate a system image data file (.ji)
--output-bc name          Generate LLVM bitcode (.bc)
--output-incremental=no   Generate an incremental output file (rather than complete)

--code-coverage={none|user|all}, --code-coverage
                          Count executions of source lines (omitting setting is equivalent to "
    user")
--track-allocation={none|user|all}, --track-allocation
                          Count bytes allocated by each source line
```

5.1 Resources

Además de este manual, hay otros recursos que pueden ayudar a los usuarios nuevos cuanto empiezan con Julia:

- Julia and IJulia cheatsheet
- Learn Julia in a few minutes
- Learn Julia the Hard Way
- Julia by Example
- Hands-on Julia
- Tutorial for Homer Reid's numerical analysis class
- An introductory presentation
- Videos from the Julia tutorial at MIT
- YouTube videos from the JuliaCons

Chapter 6

Variables

Una variable en Julia es un nombre asociado a un valor. Esto es útil cuando pretendemos almacenar un valor (como el que obtenemos después de un cálculo) para un uso posterior. Por ejemplo:

```
# Assign the value 10 to the variable x
julia> x = 10
10

# Doing math with x's value
julia> x + 1
11

# Reassign x's value
julia> x = 1 + 1
2

# You can assign values of other types, like strings of text
julia> x = "Hello World!"
"Hello World!"
```

Julia proporciona un sistema muy flexible para nombrar las variables. Los nombres de variable son sensibles a las mayúsculas, y no tienen significado semántico (es decir, que el lenguaje no trata de modo distinto a las variables basándose en sus nombres).

```
julia> x = 1.0
1.0

julia> y = -3
-3

julia> Z = "My string"
"My string"

julia> customary_phrase = "Hello world!"
"Hello world!"

julia> UniversalDeclarationOfHumanRightsStart = ""
""
```

Los nombres Unicode (usando codificación UTF-8) están permitidos:

```
julia> δ = 0.00001
1.0e-5

julia>  = "Hello"
"Hello"
```

En el REPL y otros entornos de edición Julia se pueden introducir símbolos matemáticos Unicode usando la notación de *Latex* precedido de backslash y seguido de un tabulador. Por ejemplo, podemos crear el nombre de variable δ tecleando `\delta`-*tab*, o incluso el nombre α tecleando `\alpha`-*tab*-`\hat`- *tab*-`\_2`-*tab*. (Si encuentras un símbolo en algun sitio, como por ejemplo en el código de alguien, y no sabes como escribirlo, el REPL te ayudará: solamente teclea ? y luego pega el símbolo.)

Julia también permite redefinir constantes predefinidas su fuera necesario:

```
julia> pi
π = 3.1415926535897...

julia> pi = 3
WARNING: imported binding for pi overwritten in module Main
3

julia> pi
3

julia> sqrt(100)
10.0

julia> sqrt = 4
WARNING: imported binding for sqrt overwritten in module Main
4
```

Sin embargo, esto no se recomienta para evitar una potencial confusión.

6.1 Nombres de Variables Permitidos

Los nombres de variable deben comenzar con una letra (`A-Z` o `a-z`), símblo de subrayado, o un subconjunto de puntos Unicode mayores que `00A0`. En particular, se permiten las categorías de caracteres Unicode Lu/Ll/Lt/Lm/Lo/Nl (letras), Sc/So (monedas y otros símbolos), y otros pocos caracteres (por ejemplo, un subconjunto de los símbolos matemáticos Sm). Entre los caracteres subsecuentes se pueden también incluir `!` y los dígitos (`0-9` y otros caracteres en las categorías Nd/No), así como otros puntos de código Unicode: diacríticas y otras marcas de modificación (categorías Mn/Mc/Me/Sk), algunos conectores de puntuación (category Pc), primos, y otros cuantos caracteres.

Los operadores como + son también identificadores válidos, pero son analizados sintácticamente de un modo especial. En algunos contextos, los operadores pueden ser usados justo como variables; por ejemplo `(+)` se refiere a la función de suma, y `(+) = f` la reasignará. La mayoría de los operadores infijos Unicode (en la categoría Sm), tal como , son analizados como operadores infijos y están disponibles para métodos definidos por el usuario (por ejemplo, podemos usar `const  = kron` para definir como un operador infijo producto de Kronecker).

Los únicos nombres específicamente prohibidos para nombres de variables son los nombres de las instrucciones predefinidas:

```
julia> else = false
ERROR: syntax: unexpected "else"
```

```
julia> try = "No"
ERROR: syntax: unexpected "="
```

Algunos caracteres Unicode son considerados equivalentes en identificadores. Las distintas formas de introducir caracteres que combinan en Unicode (por ejemplo, acentos) son tratadas como equivalentes (específicamente los identificadores Julia son normalizados NFC). Los caracteres Unicode ɛ (U+025B: Latin small letter open e) y µ (U+00B5: micro sign) son tratados como las letras griegas correspondientes, debido que las primeras son más fácilmente accesibles via alguos métodos de entrada.

6.2 Convenciones de Estilo

Aunque Julia impone pocas restricciones a los nombres válidos, se ha vuelto útil adoptar las siguientes convenciones:

- Los nombres de variable van en minúsculas.
- La separación enre palabras puede indicarse mediante el símbolo de guión bajo, aunque se desaconseja su uso a menos que los símbolos sean difíciles de leer.
- Los nombres de tipos y módulos comienzan con mayúscula y la separación entre palabras se representa con el formato *camel case*.
- Los nombres de funciones y macros van en minúscula, sin símbolos de guión bajo.
- Las funciones que escriben en sus argumentos tienen nombres que finalizan con el símbolo de admiración `!`. Estas suelen ser llamadas funciones "mutadoras" o funciones "*in-place*" debido a que pretenden producir cambios en sus argumentos después de que la función sea invocada, no solo devolver un valor.

Para más información sobre convenciones de estilo, ver la Guía de Estilo.

Chapter 7

Números enteros y en punto flotante

Los valores enteros y punto flotante son los bloques constructivos básicos de la aritmética y la computación. Las representaciones predefinidas para estos valores se denominan *tipos primitivos*, mientras que las reprentaciones de números enteros y en punto flotante como valores inmediatos en código se conocen como *literales numéricos*. Por ejemplo, `1` es un literal entero, mientras que `1.0` es un literal en punto flotante; sus representaciones binarias en memoria como objetos son los tipos primitivos.

Julia proporciona un amplio rango de tipos primitivos numéricos, y un complemento complemento de operadores aritméticos y de bits así como funciones matemáticas estándar definidas sobre ellos. Los operadores establecen una correspondencia enre los tipos numéricos y las operaciones que son soportadas de forma nativa sobre los ordenadores modernos, permitiendo a Julia sacar plena ventaja de los recursos computacionales. Además, Julia proporciona soporte software para *aritmética de precisión arbitraria* que puede manejar operaciones sobre valores numéricos que no puede ser representada de forma efectiva en representaciones hardware nativas, pero al coste de un rendimiento relativamente menor.

Los tipos primitivos de Julia son los siguientes:

- **Tipos enteros:**

Tipo	Signo?	Número de bits	Valor más pequeño	Valor más grande
`Int8`		8	-2^7	2^7 - 1
`UInt8`		8	0	2^8 - 1
`Int16`		16	-2^15	2^15 - 1
`UInt16`		16	0	2^16 - 1
`Int32`		32	-2^31	2^31 - 1
`UInt32`		32	0	2^32 - 1
`Int64`		64	-2^63	2^63 - 1
`UInt64`		64	0	2^64 - 1
`Int128`		128	-2^127	2^127 - 1
`UInt128`		128	0	2^128 - 1
`Bool`	N/A	8	`false` (0)	`true` (1)

- **Tipos en punto flotante:**

Adicionalmente, se ha construído un soporte completo para Números Complejos y Racionales encima de estos tipos primitivos. Todos los tipos primitivos interoperan de forma natural sin tener que realizar conversiones específicas, gracias a un sistema de promoción de tipos flexible y extensible por el usuario.

Tipo	Precisión	Número de bits
`Float16`	media	16
`Float32`	sencilla	32
`Float64`	doble	64

7.1 Enteros

Los literales enteros se representan del modo estándar:

```
julia> 1
1

julia> 1234
1234
```

El tipo por defecto para un literal entero depende de si el sistema de trabajo tiene una aquitectura de 32 o de 64 bits:

```
# 32-bit system:
julia> typeof(1)
Int32

# 64-bit system:
julia> typeof(1)
Int64
```

La variable interna de Julia `Sys.WORD_SIZE` indica si el sistema en el que trabajamos es de 32 bits o de 64 bits:

```
# 32-bit system:
julia> Sys.WORD_SIZE
32

# 64-bit system:
julia> Sys.WORD_SIZE
64
```

Julia también define los tipos `Int` y `UInt`, que son aliases para los tipos enteros nativos del sistema con y sin signo:

```
# 32-bit system:
julia> Int
Int32
julia> UInt
UInt32

# 64-bit system:
julia> Int
Int64
julia> UInt
UInt64
```

Los literales enteros mayores que no pueden ser representados usando sólo 32 bits pero pueden ser representados en 64 bits se crean como enteros de 64 bits, independientemente del tipo que tenga el sistema por defecto:

```
# 32-bit or 64-bit system:
julia> typeof(3000000000)
Int64
```

Los enteros sin signo se introducen y se muestran usando el prefijo `0x` y los dígitos hexadecimales `0-9a-f` (los dígitos capitalizados `A-F` también funcionan para la entrada). El tamaño de un valor sin signo está determinado por el número de dígitos hexadecimales usados:

```
julia> 0x1
0x01

julia> typeof(ans)
UInt8

julia> 0x123
0x0123

julia> typeof(ans)
UInt16

julia> 0x1234567
0x01234567

julia> typeof(ans)
UInt32

julia> 0x123456789abcdef
0x0123456789abcdef

julia> typeof(ans)
UInt64
```

Este comportamiento está basado en la observación de que cuando uno usa literales hexadecimales sin signo para valores enteros, se los suele utilizar para representar una secuencia de bytes numéricos fijos en lugar de un valor entero.

Recuerde que la variable `ans` contiene el valor de la última expresión evaluada en una sesión interactiva. Esto no ocurre cuando el código Julia se ejecuta de otra forma.

Los literales binarios y octales también están soportados:

```
julia> 0b10
0x02

julia> typeof(ans)
UInt8

julia> 0o10
0x08

julia> typeof(ans)
UInt8
```

Los valores máximo y mínimo de tipos primitivos numéricos representables como enteros vienen dados por las funciones `typemin()` y `typemax()`:

```
julia> (typemin(Int32), typemax(Int32))
(-2147483648, 2147483647)

julia> for T in [Int8,Int16,Int32,Int64,Int128,UInt8,UInt16,UInt32,UInt64,UInt128]

           println("$(lpad(T,7)): [$(typemin(T)),$(typemax(T))]")

       end
   Int8: [-128,127]
  Int16: [-32768,32767]
  Int32: [-2147483648,2147483647]
  Int64: [-9223372036854775808,9223372036854775807]
 Int128: [-170141183460469231731687303715884105728,170141183460469231731687303715884105727]
  UInt8: [0,255]
 UInt16: [0,65535]
 UInt32: [0,4294967295]
 UInt64: [0,18446744073709551615]
UInt128: [0,340282366920938463463374607431768211455]
```

Los valores devueltos por `typemin()` y `typemax()` siempre son del tipo de argumento dado. (La expresión anterior utiliza varias características que todavía tenemos que introducir, incluyendo bucles for, Cadenas, e Interpolación, pero debería ser lo suficientemente fácil de entender para los usuarios con cierta experiencia en programación).)

Comportamiento ante el Desbordamiento

En Julia, superar el valor máximo representable de un tipo dado da como resultado un comportamiento envolvente:

```
julia> x = typemax(Int64)
9223372036854775807

julia> x + 1
-9223372036854775808

julia> x + 1 == typemin(Int64)
true
```

Así, la aritmética con enteros de Julia es en realidad una forma de aritmética modular. Esto refleja las características de la aritmética subyacente de números enteros tal como se implementa en las computadoras modernas. En aplicaciones donde es posible el desbordamiento, es esencial comprobar explícitamente el envolvente producido por el desbordamiento. De lo contrario, se recomienda el tipo `BigInt` en Aritmética de Precisión Arbitraria.

Errores de división

La división entera (la función `div`) tiene dos casos excepcionales: división por cero, y dividir el número negativo más bajo (`typemin()`) por -1. Ambos casos lanzan un `DivideError`. El resto y las funciones de módulo (`rem` y `mod`) lanzan un `DivideError` cuando su segundo argumento es cero.

7.2 Números en Punto Flotante

Los literales de números en punto flotante son representados en las formas estándar:

```
julia> 1.0
1.0
```

```
julia> 1.
1.0

julia> 0.5
0.5

julia> .5
0.5

julia> -1.23
-1.23

julia> 1e10
1.0e10

julia> 2.5e-4
0.00025
```

Los resultados anteriores son todos valores `Float64`. Los valores literales `Float32` pueden introducirse escribiendo `f` en lugar de `e`:

```
julia> 0.5f0
0.5f0

julia> typeof(ans)
Float32

julia> 2.5f-4
0.00025f0
```

Los valores pueden ser convertidos a `Float32` fácilmente:

```
julia> Float32(-1.5)
-1.5f0

julia> typeof(ans)
Float32
```

También son válidos los literales de punto flotante en formato hexadecimal, pero sólo como valores `Float64`:

```
julia> 0x1p0
1.0

julia> 0x1.8p3
12.0

julia> 0x.4p-1
0.125

julia> typeof(ans)
Float64
```

También esta soportados los números en punto flotante de media precisión (`Float16`), pero ellos están implementados en software y usan `Float32` para los cálculos.

```
julia> sizeof(Float16(4.))
2

julia> 2*Float16(4.)
Float16(8.0)
```

El guión bajo (*underscore*) puede usarse como separador de dígitos:

```
julia> 10_000, 0.000_000_005, 0xdead_beef, 0b1011_0010
(10000, 5.0e-9, 0xdeadbeef, 0xb2)
```

Cero en punto flotante

Los números en punto flotante tienen dos ceros, positivo y negativo. Ellos son iguales entre sí, pero tienen distintas representaciones, como puede verse si usamos la función `bits`:

```
julia> 0.0 == -0.0
true

julia> bits(0.0)
"0000000000000000000000000000000000000000000000000000000000000000"

julia> bits(-0.0)
"1000000000000000000000000000000000000000000000000000000000000000"
```

Valores especiales en punto flotante

Hay tres valores especificados en el estándar de punto flotante para valores que no se corresponden con ningún punto en la línea de números reales:

`Float16`	`Float32`	`Float64`	Name	Description
`Inf16`	`Inf32`	`Inf`	Infinito positivo	Un valor mayor que todos los valores finitos de punto flotante
`-Inf16`	`-Inf32`	`-Inf`	Infinito negativo	Un valor menos que todos los valores finitos de punto flotante
`NaN16`	`NaN32`	`NaN`	*not a number*	Un valor no == a otro valor en punto flotante (incluido el mismo)

Para más información sobre cómo estos valores de punto flotante no finitos están ordenados entre sí y otros flotantes, vea Comparaciones Numéricas. Mediante el estándar IEEE 754, estos valores de punto flotante son el resultado de ciertas operaciones aritméticas:

```
julia> 1/Inf
0.0

julia> 1/0
Inf

julia> -5/0
```

```
-Inf

julia> 0.000001/0
Inf

julia> 0/0
NaN

julia> 500 + Inf
Inf

julia> 500 - Inf
-Inf

julia> Inf + Inf
Inf

julia> Inf - Inf
NaN

julia> Inf * Inf
Inf

julia> Inf / Inf
NaN

julia> 0 * Inf
NaN
```

Las funciones `typemin()` y `typemax()` también se aplican a los tipos en punto flotante:

```
julia> (typemin(Float16),typemax(Float16))
(-Inf16, Inf16)

julia> (typemin(Float32),typemax(Float32))
(-Inf32, Inf32)

julia> (typemin(Float64),typemax(Float64))
(-Inf, Inf)
```

Epsilon de máquina

La mayoría de los números reales no pueden representarse exactamente con números de coma flotante, por lo que para muchos propósitos es importante conocer la distancia entre dos números de punto flotante representables adyacentes, lo que a menudo se conoce como epsilon de máquina.

Julia proporciona `eps()`, que da la distancia entre 1,0 y el siguiente valor de punto flotante representable más grande:

```
julia> eps(Float32)
1.1920929f-7

julia> eps(Float64)
2.220446049250313e-16

julia> eps() # same as eps(Float64)
2.220446049250313e-16
```

Estos valores son `2.0^-23` y `2.0^-52` como valores `Float32` y `Float64`, respectivamente. La función `eps()` también puede tomar un valor de punto flotante como un argumento y da la diferencia absoluta entre ese valor y el siguiente valor de punto flotante representable. Es decir, `eps(x)` produce un valor del mismo tipo que `x` tal que `x + eps(x)` es el siguiente valor de punto flotante representable mayor que `x`:

```
julia> eps(1.0)
2.220446049250313e-16

julia> eps(1000.)
1.1368683772161603e-13

julia> eps(1e-27)
1.793662034335766e-43

julia> eps(0.0)
5.0e-324
```

La distancia entre dos números de punto flotante representables adyacentes no es constante, pero es menor para valores más pequeños y mayor para valores mayores. En otras palabras, los números de punto flotante representables son más densos en la línea de números reales cerca de cero, y crecen exponencialmente dispersos a medida que uno se aleja de cero. Por definición, `eps(1.0)` es el mismo que `eps(Float64)` ya que `1.0` es un valor de coma flotante de 64 bits.

Julia también proporciona las funciones `nextfloat()` y `prevfloat()` que devuelven el siguiente número de punto flotante representable más grande o más pequeño al argumento, respectivamente:

```
julia> x = 1.25f0
1.25f0

julia> nextfloat(x)
1.2500001f0

julia> prevfloat(x)
1.2499999f0

julia> bits(prevfloat(x))
"00111111100111111111111111111111"

julia> bits(x)
"00111111101000000000000000000000"

julia> bits(nextfloat(x))
"00111111101000000000000000000001"
```

Este ejemplo resalta el principio general de que los números de punto flotante representables adyacentes también tienen representaciones binarias enteras adyacentes.

Modos de Redondeo

Si un número no tiene una representación de punto flotante exacta, debe redondearse a un valor representable apropiado. Sin embargo, si se desea, la forma en que se realiza este redondeo puede cambiarse de acuerdo con los modos de redondeo presentados en el estándar IEEE 754.

```
julia> x = 1.1; y = 0.1;

julia> x + y
1.2000000000000002

julia> setrounding(Float64,RoundDown) do

           x + y

       end
1.2
```

El modo predeterminado utilizado siempre es `RoundNearest`, , que redondea al valor representable más cercano, con arcos redondeados hacia el valor más cercano con un bit menos significativo.

> **Warning**
>
> El redondeo generalmente sólo es correcto para las funciones aritméticas básicas (`+()`, `-()`, `*()`, `/()` and `sqrt()`) y las operaciones de conversión de tipos. Muchas otras funciones asumen que el modo por defecto `RoundNearest` está establecido y pueden dar resultados erróneos al operar bajo otros modos de redondeo.

Antecedentes y referencias

La aritmética de punto flotante supone muchas sutilezas que pueden sorprender a los usuarios que no están familiarizados con los detalles de implementación de bajo nivel. Sin embargo, estas sutilezas se describen en detalle en la mayoría de los libros sobre computación científica, y también en las siguientes referencias:

- La guía definitiva para la aritmética de coma flotante es el estándar [IEEE 754-2008 (http://standards.ieee.org/findstds/standard/754-2008.html); Sin embargo, no está disponible

en línea gratis.

- Para una presentación breve pero lúcida de cómo los números de punto flotante están

representados, vea el artículo de John D. Cook sobre el tema, así como su introducción a algunas de las cuestiones que surgen de cómo esta representación difiere en el comportamiento de la abstracción idealizada de números reales.

- También se recomienda la serie de publicaciones de Bruce Dawson sobre números en punto flotante.
- Para un excelente y profundo análisis de los números de punto flotante y los problemas de precisión numérica encontrados al calcular con ellos, vea el artículo de David Goldberg What Every Computer Scientist Should Know About Floating-Point Arithmetic.
- Para una documentación aún más extensa de la historia de, la razón y las cuestiones con los números de punto flotante, así como la discusión de muchos otros temas en la computación numérica, ver los escritos recolectados de William Kahan, comúnmente conocido como el "padre de punto flotante". De interés particular puede ser An Interview with the Old Man of Floating-Point.

7.3 Aritmética de Precisión Arbitraria

Para permitir cálculos con enteros y números de coma flotante de precisión arbitraria, Julia envuelve la Biblioteca Aritmética de Precisión Múltiple GNU (GMP) y la biblioteca GNU MPFR, respectivamente. Los tipos `BigInt` y `BigFloat` están disponibles en Julia para números enteros de precisión arbitraria y números de coma flotante, respectivamente.

Existen constructores para crear estos tipos de tipos numéricos primitivos, y podemos también utilizar `parse()` para construirlos a partir de `AbstractStrings`. Una vez creados, participan en la aritmética con todos los demás tipos numéricos gracias al mecanismo de promotion y conversion de tipos de Julia:

```
julia> BigInt(typemax(Int64)) + 1
9223372036854775808

julia> parse(BigInt, "123456789012345678901234567890") + 1
123456789012345678901234567891

julia> parse(BigFloat, "1.23456789012345678901")
1.23456789012345678901000000000000000000000000000000000000000000000000000000000000000004

julia> BigFloat(2.0^66) / 3
2.45956587649460688213333333333333333333333333333333333333333333333333333333333333344e+19

julia> factorial(BigInt(40))
815915283247897734345611269596115894272000000000
```

Sin embargo, la promoción de tipos entre los tipos primitivos ya vistos y `BigInt`/`BigFloat` no es automática y debe ser establecida explícitamente:

```
julia> x = typemin(Int64)
-9223372036854775808

julia> x = x - 1
9223372036854775807

julia> typeof(x)
Int64

julia> y = BigInt(typemin(Int64))
-9223372036854775808

julia> y = y - 1
-9223372036854775809

julia> typeof(y)
BigInt
```

La precisión predeterminada (en número de bits del significado) y el modo de redondeo de las operaciones de `BigFloat` pueden cambiarse globalmente llamando `setprecision()` and `setrounding()`, y todos los cálculos adicionales tomarán en cuenta estos cambios. Alternativamente, la precisión o el redondeo se puede cambiar dentro sólo de la ejecución de un bloque particular de código utilizando las mismas funciones dentro de un bloque do:

```
julia> setrounding(BigFloat, RoundUp) do

           BigFloat(1) + parse(BigFloat, "0.1")
```

```
      end
1.100000000000000000000000000000000000000000000000000000000000000000000000000000000000000003

julia> setrounding(BigFloat, RoundDown) do

          BigFloat(1) + parse(BigFloat, "0.1")

      end
1.099999999999999999999999999999999999999999999999999999999999999999999999999999999999999986

julia> setprecision(40) do

          BigFloat(1) + parse(BigFloat, "0.1")

      end
1.1000000000004
```

7.4 Coeficientes Literales Numéricos

Para hacer más claras fórmulas numéricas y expresiones, Julia permite que las variables sean precedidas inmediatamente por un literal numérico, implicando la multiplicación. Esto hace que la escritura de las expresiones polinómicas sea mucho más limpias:

```
julia> x = 3
3

julia> 2x^2 - 3x + 1
10

julia> 1.5x^2 - .5x + 1
13.0
```

También hace que escribir funciones exponenciales sea más elegante:

```
julia> 2^2x
64
```

La precedencia de los coeficientes literales numéricos es la misma que la de los operadores unarios como la negación. Así que `2^3x` se analiza como `2^(3x)`, y `2x^3` se analiza como `2*(x ^ 3)`.

Los literales numéricos también funcionan como coeficientes de las expresiones entre paréntesis:

```
julia> 2(x-1)^2 - 3(x-1) + 1
3
```

Note

La precedencia de coeficientes literales numéricos usada para multiplicación implícita es mayor que otros operadores binarios tales como la multiplicación (`*`), y división (`/`, `\`, and `//`). Esto significa, por ejemplo, que `1 / 2im` es igual a `-0.5im` y `6 // 2(2 + 1)` es igual a `1 // 1`.

Además, las expresiones entre paréntesis se pueden utilizar como coeficientes a las variables, lo que implica la multiplicación de la expresión por la variable:

```
julia> (x-1)x
6
```

Sin embargo, ni la yuxtaposición de dos expresiones entre paréntesis, ni la colocación de una variable antes de una expresión entre paréntesis puede ser usada para implicar multiplicación:

```
julia> (x-1)(x+1)
ERROR: MethodError: objects of type Int64 are not callable

julia> x(x+1)
ERROR: MethodError: objects of type Int64 are not callable
```

Ambas expresiones se interpretan como la aplicación de una función: cualquier expresión que no sea un literal numérico, inmediatamente seguida de una entre paréntesis, se interpreta como una función aplicada a los valores entre paréntesis (ver Funciones para más información sobre las funciones). Por lo tanto, en ambos casos, se produce un error, ya que el valor de la izquierda no es una función.

Las mejoras sintácticas anteriores reducen significativamente el ruido visual producido al escribir fórmulas matemáticas comunes. Obsérvese que ningún espacio en blanco puede encontrarse entre un coeficiente literal numérico y el identificador o la expresión entre paréntesis que multiplica.

Conflictos de Sintaxis

La sintaxis de los coeficientes literales yuxtapuestos puede entrar en conflicto con dos sintaxis numéricas literales: literales enteros hexadecimales y notación ingenieril para literales de punto flotante. Aquí hay algunas situaciones donde surgen conflictos sintácticos:

- La expresión literal de enteros hexadecimales `0xff` podría interpretarse como el literal numérico `0` multiplicado por la variable `xff`.
- La expresión literal de punto flotante `1e10` podría interpretarse como el literal numérico `1` multiplicado por la variable `e10`, e igualmente con la forma `E` equivalente

En ambos casos, resolvemos la ambigüedad a favor de la interpretación como literales numéricos:

- Las expresiones que comienzan con `0x` siempre son literales hexadecimales.
- Las expresiones que empiezan con un literal numérico seguido por `e` o `E` siempre son literales de coma flotante.

7.5 Literales cero and uno

Julia proporciona funciones que devuelven los literales `0` y `1` correspondientes a un tipo especificado o al tipo de una variable dada.

Function	Description
`zero(x)`	Literal cero del tipo x o del tipo de la variable x
`one(x)`	Literal uno del tipo x o del tipo de la variable x

Estas funciones son útiles en comparaciones numéricas para evitar la sobrecarga de una conversión de tipo innecesaria.

Ejemplos:

```
julia> zero(Float32)
0.0f0

julia> zero(1.0)
```

```
0.0

julia> one(Int32)
1

julia> one(BigFloat)
1.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Chapter 8

Mathematical Operations and Elementary Functions

Julia proporciona una colección completa de operadores aritméticos básicos y de operadores de bits para todos sus tipos numéricos primitivos, así como implementaciones portables y eficientes de una colección comprensiva de funciones matemática estándar.

8.1 Operadores Aritméticos

Los siguientes operadores aritméticos están soportados sobre todos los tipos primitivos:

Expression	Name	Description
`+x`	más unario	Operación identidad
`-x`	menos unario	Inverso matemático de un número
`x + y`	suma binaria	suma
`x - y`	menos binario	resta
`x * y`	producto	multiplicación
`x / y`	división	división
`x \ y`	división inversa	Equivalente a `y / x`
`x ^ y`	potencia	eleva `x` a la `y`-ésima potencia
`x % y`	resto	Equivalente a `rem(x,y)`

así como la negación sobre tipos `Bool`:

Expression	Name	Description
`!x`	negación	Cambia `true` a `false` y viceversa

El sistema de promoción de Julia hace que las operaciones aritméticas sobre mezclas de tipos de argumentos funcione de forma natural y automáticamente. Ver Conversión y Promoción para los detalles del sistema de promoción.

He aquí algunos ejemplos simples de usar operadores aritméticos:

```
julia> 1 + 2 + 3
6

julia> 1 - 2
-1

julia> 3*2/12
0.5
```

(Por convención, tendemos a separar con menos distancia los operadores cuando se aplican antes de otros operadores cercanos. Por ejemplo, generalmente escribimos `-x + 2` para reflejar que x primero se niega y, a continuación, 2 se agrega a ese resultado.)

8.2 Operadores bit a bit

Los siguientes operadores bit a bit son soportados sobre todos los tipos enteros primitivos:

Expression	Name
`~x`	Negación bit a bit
`x & y`	Conjunción (*and*) bit a bit
`x | y`	Disyunción (*or*) bit a bit
`x  y`	*Or* exclusivo bit a bit (*xor*)
`x >>> y`	Desplazamiento lógico hacia la derecha
`x >> y`	Desplazamiento aritmético hacia la derecha
`x << y`	Desplazamiento hacia la izquierda lógico/aritmético

He aquí algunos ejemplos de uso de operadores bit a bit:

```
julia> ~123
-124

julia> 123 & 234
106

julia> 123 | 234
251

julia> 123  234
145

julia> xor(123, 234)
145

julia> ~UInt32(123)
0xffffff84

julia> ~UInt8(123)
0x84
```

8.3 Operaciones de actualización

Cada operador binario aritmético y bit a bit también tiene una versión de actualización que asigna el resultado de la operación de nuevo a su operando izquierdo. La versión de actualización del operador binario se forma colocando a `=` inmediatamente después del operador. Por ejemplo, escribir `x += 3` es equivalente a escribir `x = x + 3`:

```
julia> x = 1
1

julia> x += 3
4
```

```
julia> x
4
```

Las versiones de actualización de todos los operadores binarios, aritméticos de bits son:

```
+=  -=  *=  /=  \=  ÷=  %=  ^=  &=  |=  =  >>>=  >>=  <<=
```

Note

Un operador de actualización reasigna la variable sobre la parte izquierda de la ecuación. Como resultado, el tipo de la variable puede cambiar:

```
julia> x = 0x01; typeof(x)
UInt8

julia> x *= 2 # Same as x = x * 2
2

julia> typeof(x)
Int64
```

8.4 Operadores vectorizados con "punto"

Para cada operación binaria como ^ hay su correspondiente operación "con punto" .^ que se define *automáticamente* para realizar la operación ^ elemento a elemento sobre arrays. Por ejemplo, la operación [1, 2, 3]^3 no está definidia, porque no hay un significado matemático estándar para calcular el cubo de un array, pero [1, 2, 3].^3 si lo está como el cálculo de la operación cubo elemento a elemento (o vectorizada) [1^3, 2^3, 3^3]. Lo mismo puede decirse para operadores unarios tales como ! o √, que existe el correspondiente operador vectorizado .√ que aplica el operador elemento a elemento.

```
julia> [1,2,3] .^ 3
3-element Array{Int64,1}:
  1
  8
 27
```

Más específicamente, a .^ b es analizado como la llamada punto (^).(a,b), que realiza una operación de retransmisión (*broadcast*): ella puede combinar arrays y escalares, arrays del mismo tamaño (realizando la operación elemento a elemento), o incluso arrays de diferentes formas (por ejemplo, combinar vectores fila y columna para producir una matriz). Además, como todas las "llamadas punto", estos "operadores punto" están *fusionados*. Por ejemplo, si calculamos 2 .* A.^2 .+ sin.(A) (o, equivalentemente @. 2A^2 + sin(A), usando la macro @.) para un array A, se realiza un *único* bucle sobre A, computando 2a^2 + sin(a) para cada elemento de A. En particular, las llamadas vectorizadas anidadas como f.(g.(x)) están *fusionadas*, y los operadores binarios adyacentes como x .+ 3 .* x.^2 son equivalentes a llamadas vectorizadas anidadas (+).(x, (*).(3, (^).(x, 2))).

Además, los operadores de actualización "vectorizados" como a .+= b (o @. a += b) son transformados en a .= a .+ b, donde .= es un operador de asignación *fusionado in-place* (ver la documentación de la sintaxis vectorizada).

Nótese que la sintaxis de punto es también aplicable a operadores definidos por el usuario. Por ejemplo, si definimos el operador (A,B) = kron(A,B) para dar una sintaxis infija A ⊗ B al producto de Kronecker (kron), entonces [A,B] . [C,D] calculará [AC, BD] sin ninguna codificación adicional.

Operador	Nombre
==	Igualdad
!=, ≠	Desigualdad
<	Menor que
<=, ≤	Menor o igual que
>	Mayor que
>=, ≥	Mayor o igual que

8.5 Comparaciones Numéricas

Los operadores de comparación estándar están definidos para todos los tipos numéricos primitivos:

He aquí algunos ejemplos:

```
julia> 1 == 1
true

julia> 1 == 2
false

julia> 1 != 2
true

julia> 1 == 1.0
true

julia> 1 < 2
true

julia> 1.0 > 3
false

julia> 1 >= 1.0
true

julia> -1 <= 1
true

julia> -1 <= -1
true

julia> -1 <= -2
false

julia> 3 < -0.5
false
```

Los enteros se comparan de un modo estándar, mediante comparación de bits. Los números de punto flotante se comparan de acuerdo al estándar IEEE 754:

- Los números finitos son ordenados del modo habitual.
- El cero positivo es igual pero no mayor que el cero negativo.

- `Inf` es igual a si mismo y mayor que todo excepto `NaN`
- `-Inf` es igual a si mismo y menor que todo excepto `NaN`
- `NaN` no es igual, mayor o menor a nadie, excepto a sí mismo.

Este último punto es potencialmente sorprendente y, por tanto, vale la pena señalar que:

```
julia> NaN == NaN
false

julia> NaN != NaN
true

julia> NaN < NaN
false

julia> NaN > NaN
false
```

y puede causar dolores de cabeza especiales con Arrays:

```
julia> [1 NaN] == [1 NaN]
false
```

Julia proporciona funciones adicionales para comprobar números para valores especiales, lo cuál pueden ser útil en situaciones como las comparaciones de claves hash:

Function	Tests if
`isequal(x, y)`	x e y son idénticos
`isfinite(x)`	x es un número finito
`isinf(x)`	x es infinito
`isnan(x)`	x no es un número

`isequal()` considera los `NaN`s iguales entre sí:

```
julia> isequal(NaN, NaN)
true

julia> isequal([1 NaN], [1 NaN])
true

julia> isequal(NaN, NaN32)
true
```

`isequal()` también puede usarse para distinguir los ceros con signo:

```
julia> -0.0 == 0.0
true

julia> isequal(-0.0, 0.0)
false
```

Las comparaciones de tipos mezclados entre enteros con signo, enteros sin signo y valores en punto flotante pueden ser complicadas. Se ha tomado mucho cuidado para asegurarse de que Julia las realiza correctamente.

Para otros tipos, `isequal()` llama por defecto a `==()`, así que si uno quiere definir la igualdad para sus propios tipos, solo tiene que agregar un método `==()`. Si uno define suu propia función de igualdad, probablemente deba definir un método `hash()` correspondiente para asegurar de que `isequal(x,y)` implica `hash(x) == hash(y)`.

Comparaciones Encadenadas

A diferencia de la mayoría de los idiomas, con la notable excepción de Python, las comparaciones pueden encadenarse arbitrariamente:

```
julia> 1 < 2 <= 2 < 3 == 3 > 2 >= 1 == 1 < 3 != 5
true
```

El encadenamiento de comparaciones suele ser bastante conveniente en el código numérico. Las comparaciones encadenadas utilizan el operador && para comparaciones escalares y el operador & para comparaciones elemento a elemento, lo que les permite trabajar sobre arrays. Por ejemplo, `0 .< A .< 1` da un array booleano cuyas entradas son `true` en posiciones en las que los elementos correspondientes de A están entre 0 y 1.

Nótese el comportamiento de evaluación de las comparaciones encadenadas:

```
julia> v(x) = (println(x); x)
v (generic function with 1 method)

julia> v(1) < v(2) <= v(3)
2
1
3
true

julia> v(1) > v(2) <= v(3)
2
1
false
```

La expresión del medio sólo se evalúa una vez, en lugar de dos veces como lo sería si la expresión se escribiera como `v(1) < v(2) && v(2) <= v(3)`. Sin embargo, el orden de las evaluaciones en una comparación encadenada no está definido. Se recomienda encarecidamente no utilizar expresiones que puedan tener efectos secundarios (como la impresión) en comparaciones encadenadas. Si se requieren efectos secundarios, se debe utilizar explícitamente el operador de cortocircuito && (ver Evaluación en cortocircuito).

Funciones Elementales

Julia proporciona una colección completa de funciones matemáticas y operadores. Estas operaciones matemáticas se definen sobre una clase de valores numéricos suficientemente amplia como para permitir definiciones apropiadas para enteros, números de punto flotante, racionales y complejos, dondequiera que tales definiciones tengan sentido.

Además, estas funciones (como cualquier función de Julia) se pueden aplicar de manera "vectorizada" a matrices y otras colecciones con la sintaxis vectorizada `f.(A)`, por ejemplo, sin.(A) calculará el seno de cada elemento de una matriz `A`.

Categorí	Operadores
Syntax	`.` seguido por `::`
Exponentiation	`^`
Fractions	`//`
Multiplication	`* / % & \`
Bitshifts	`<< >> >>>`
Addition	`+ - |`
Syntax	`: ..` seguido por `|>`
Comparisons	`> < >= <= == === != !== <:`
Control flow	&& seguido por || seguido por ?
Assignments	`= += -= *= /= //= \= ^= ÷= %= |= &= = <<= >>= >>>=`

8.6 Precedencia de Operadores

Julia applies the following order of operations, from highest precedence to lowest:

Para una lista completa de cada una de las precedencias de operadores de Julia, consultar el fichero `src/julia-parser.scm`

También puede encontrarse la precedencia numérica pra cualquier operación dada mediante la función intrínseca `Base.operator_precedence` donde el número mayor corresponde a la operación con mayor precedencia.

```
julia> Base.operator_precedence(:+), Base.operator_precedence(:*), Base.operator_precedence(:.)
(9, 11, 15)

julia> Base.operator_precedence(:+=), Base.operator_precedence(:(=))  # (Note the necessary parens
↪  on `:(=)`)
(1, 1)
```

8.7 Conversiones Numéricas

Julia soporta tres formas de conversión numérica, que difieren en su manejo de las conversiones inexactas.

- La notación `T(x)` o `convert(T,x)` convierte `x` a un valor de tipo `T`.
  - Si `T` es un tipo en punto flotante, el resultado es el valor más cercano representable, que podría ser infinito positivo o negativo.
  - Si `T` es un tipo entero, se lanzará un `InexactError` si `x`no es representable por `T`.
- `x % T`convierte un entero `x` a un valor de un tipo entero `T` congruente a `x` modulo `2^n`, donde `n` es el número de bits en `T`. En otras palabras, la representación binaria es truncada para ajustarse.
- Las Funciones de Redondeo toman un tipo `T` como argumento opcional. Por ejemplo, `round(Int,x)` es una abreviatura de `Int(round(x))`.

Los siguientes ejemplos muestran las siguientes formas:

```
julia> Int8(127)
127

julia> Int8(128)
ERROR: InexactError()
```

```
Stacktrace:
 [1] Int8(::Int64) at ./sysimg.jl:77

julia> Int8(127.0)
127

julia> Int8(3.14)
ERROR: InexactError()
Stacktrace:
 [1] convert(::Type{Int8}, ::Float64) at ./float.jl:658
 [2] Int8(::Float64) at ./sysimg.jl:77

julia> Int8(128.0)
ERROR: InexactError()
Stacktrace:
 [1] convert(::Type{Int8}, ::Float64) at ./float.jl:658
 [2] Int8(::Float64) at ./sysimg.jl:77

julia> 127 % Int8
127

julia> 128 % Int8
-128

julia> round(Int8,127.4)
127

julia> round(Int8,127.6)
ERROR: InexactError()
Stacktrace:
 [1] trunc(::Type{Int8}, ::Float64) at ./float.jl:651
 [2] round(::Type{Int8}, ::Float64) at ./float.jl:337
```

Ver Conversión y Promoción para ver cómo definir tus propias conversiones y promociones.

Funciones de Redondeo

Función	Descripción	Tipo devuelto
`round(x)`	Redondea x al entero más cercano	`typeof(x)`
`round(T, x)`	Redondea x al entero más cercano	`T`
`floor(x)`	Redondea x hacia `-Inf`	`typeof(x)`
`floor(T, x)`	Redondea x hacia `-Inf`	`T`
`ceil(x)`	Redondea x hacia `+Inf`	`typeof(x)`
`ceil(T, x)`	Redondea x hacia `+Inf`	`T`
`trunc(x)`	Redondea x hacia cero	`typeof(x)`
`trunc(T, x)`	Redondea x hacia cero	`T`

Función	Descripción
`div(x,y)`	División truncada; cociente redondeado hacia cero
`fld(x,y)`	División *floored*; cociente redondeado hacia `-Inf`
`cld(x,y)`	División *ceiling*; cociente redondeado hacia `+Inf`
`rem(x,y)`	Resto; satisface `x == div(x,y)*y + rem(x,y)`; el signo se corresponde con el de `x`
`mod(x,y)`	Módulo; satisface `x == fld(x,y)*y + mod(x,y)`; el signo se corresponde con el de `y`
`mod1(x,y)`	Módulo con un desplazamiento de 1; devuelve `r(0,y]` para `y>0` o `r[y,0)` para `y<0`, donde `mod(r, y) == mod(x, y)`
`mod2pi(x)`	Módulo con respecto a 2pi; `0 <= mod2pi(x) < 2pi`
`divrem(x,y)`	Devuelve `(div(x,y),rem(x,y))`
`fldmod(x,y)`	Devuelve `(fld(x,y),mod(x,y))`
`gcd(x,y...)`	Máximo común divisor positivo de x, y,...
`lcm(x,y...)`	Mínimo común múltiplo positivo de x, y,...

Función	Descripción
`abs(x)`	Un valor positivo con la magnitud de x
`abs2(x)`	El cuadrado de la magnitud de x
`sign(x)`	Indica el signo de x, devolviendo -1, 0, o +1
`signbit(x)`	Indica que si el bit de signo está en **on** (`true`) o en **off** (`false`)
`copysign(x,y)`	Indica un valor con la magnitud de x y el signo de y
`flipsign(x,y)`	Indica un valor con la magnitud de x y el signo de `x*y`

Función	Descripción
`sqrt(x)`, √x	Raíz cuadrada de x
`cbrt(x)`, ∛x	Raíz cúbica de x
`hypot(x,y)`	Hipotenusa del triángulo rectángulo cuyos catetos son de longitudes x e y
`exp(x)`	Función exponencial natural sobre x
`expm1(x)`	Valor exacto de `exp(x)-1` para x cercano a zero
`ldexp(x,n)`	`x*2^n` calculado eficientemente para valores enteros de n
`log(x)`	Logaritmo neperiano de x
`log(b,x)`	Logaritmo en base b de x
`log2(x)`	Logaritmo en base 2 de x
`log10(x)`	Logaritmo decimal de x
`log1p(x)`	Valor exacto de `log(1+x)` para x cercano a cero
`exponent(x)`	Exponente binario de x
`significand(x)`	Significando binario (alias *mantisa*) de un número en punto flotante x

Funciones de División

Funciones de signo y valor absoluto

Potencias, logaritmos y raíces

Para una explicación de por qué son necesarias funciones como `hypot()`, `expm1()`, and `log1p()`, véase el excelente par de artículos en el blog de John D. Cook's sobre el tema: expm1, log1p, erfc, e hypot.

Funciones Trigonométricas e Hiperbólicas

Todas las funciones trigonométricas e hiperbólicas estándar están también definidas:

```
sin    cos    tan    cot    sec    csc
sinh   cosh   tanh   coth   sech   csch
asin   acos   atan   acot   asec   acsc
asinh  acosh  atanh  acoth  asech  acsch
sinc   cosc   atan2
```

Son todas funciones de un solo argumento, con la excepción de atan2, que da el ángulo en radians entre el eje *x* y el punto especificado por sus argumentos, interpretado como sus coordenadas *x* e *y*.

Adicionalmente, se proporcionan `sinpi(x)` e `cospi(x)` para cálculos más exactos de `sin(pi*x)` y `cos(pi*x)` respectivamente.

Para computar funciones trigonométricas con grados en lugar de con rdianes, añada al nombre de la función el sufijo d. Por ejemplo, `sind(x)` calcula el seno de x, donde x se especifica en grados. La lista completa de funciones trigonométricas con variantes grados es:

```
sind   cosd   tand   cotd   secd   cscd
asind  acosd  atand  acotd  asecd  acscd
```

Funciones Especiales

Función	Descripción
`gamma(x)`	Función gamma en x
`lgamma(x)`	Valor exacto de `log(gamma(x))` para valores grandes de x
`lfact(x)`	Valor exacto de `log(factorial(x))` para valores grandes de x; igual que `lgamma(x+1)` para x > 1, cero en otros caso
`beta(x,y)`	Función beta en `x,y`
`lbeta(x,y)`	Valor exacto de `log(beta(x,y))` para valores grandes de x o y

Chapter 9

Números Racionales y Complejos

Julia se distribuye con tipos predefinidos que representan números complejos y racionales, y soporta todas las Operaciones Matemáticas y Funciones Elementales estándar sobre ellos. Se han definido conversiones y promociones de modo que las operaciones con cualquier combinación de tipos numéricos predefinidos, primitivos o compuestos, se comporten como se esperaba.

9.1 Números Complejos

La constante global `im` está ligada al número complejo *i*, que representa la raíz cuadrada principal de -1. Se consideró nocivo para co-optar el nombre `i` para una constante global, ya que es un nombre de variable de índice popular. Como Julia permite que los literales numéricos se yuxtapongan con identificadores como coeficientes, esta unión es suficiente para proporcionar sintaxis conveniente para números complejos, similar a la notación matemática tradicional:

```
julia> 1 + 2im
1 + 2im
```

Podemos realizar todas las operaciones aritméticas estándar con los números complejos:

```
julia> (1 + 2im)*(2 - 3im)
8 + 1im

julia> (1 + 2im)/(1 - 2im)
-0.6 + 0.8im

julia> (1 + 2im) + (1 - 2im)
2 + 0im

julia> (-3 + 2im) - (5 - 1im)
-8 + 3im

julia> (-1 + 2im)^2
-3 - 4im

julia> (-1 + 2im)^2.5
2.7296244647840084 - 6.960664459571898im

julia> (-1 + 2im)^(1 + 1im)
-0.27910381075826657 + 0.08708053414102428im
```

```
julia> 3(2 - 5im)
6 - 15im

julia> 3(2 - 5im)^2
-63 - 60im

julia> 3(2 - 5im)^-1.0
0.20689655172413796 + 0.5172413793103449im
```

El mecanismo de promoción asegura qur las combinaciones de operandos de distintos tipos funcionarán:

```
julia> 2(1 - 1im)
2 - 2im

julia> (2 + 3im) - 1
1 + 3im

julia> (1 + 2im) + 0.5
1.5 + 2.0im

julia> (2 + 3im) - 0.5im
2.0 + 2.5im

julia> 0.75(1 + 2im)
0.75 + 1.5im

julia> (2 + 3im) / 2
1.0 + 1.5im

julia> (1 - 3im) / (2 + 2im)
-0.5 - 1.0im

julia> 2im^2
-2 + 0im

julia> 1 + 3/4im
1.0 - 0.75im
```

Nótese que `3/4im == 3/(4*im) == -(3/4*im)`, ya que un coeficiente literal se enlaza más fuerte que la división.

También se proporcionan las funciones estándar para manipular valores complejos:

```
julia> z = 1 + 2im
1 + 2im

julia> real(1 + 2im) # real part of z
1

julia> imag(1 + 2im) # imaginary part of z
2

julia> conj(1 + 2im) # complex conjugate of z
1 - 2im

julia> abs(1 + 2im) # absolute value of z
```

```
2.23606797749979

julia> abs2(1 + 2im) # squared absolute value
5

julia> angle(1 + 2im) # phase angle in radians
1.1071487177940904
```

Como de costumbre, el valor absoluto (`abs()`) de un número complejo es su distancia a cero. `abs2()` da el cuadrado del valor absoluto, y es de uso particular para los números complejos donde se evita tomar una raíz cuadrada. `angle()` devuelve el ángulo de fase en radianes (también conocido como *argumento* o función *arg*). La gama completa de otras funciones elementales está también definida para los números complejos:

```
julia> sqrt(1im)
0.7071067811865476 + 0.7071067811865475im

julia> sqrt(1 + 2im)
1.272019649514069 + 0.7861513777574233im

julia> cos(1 + 2im)
2.0327230070196656 - 3.0518977991518im

julia> exp(1 + 2im)
-1.1312043837568135 + 2.4717266720048188im

julia> sinh(1 + 2im)
-0.4890562590412937 + 1.4031192506220405im
```

Tenga en cuenta que las funciones matemáticas normalmente devuelven valores reales cuando se aplican a números reales y valores complejos cuando se aplican a números complejos. Por ejemplo, `sqrt()` se comporta de forma diferente cuando se aplica a `-1` que cuanso se aplica sobre `-1 + 0im`, aunque `-1 == -1 + 0im`:

```
julia> sqrt(-1)
ERROR: DomainError:
sqrt will only return a complex result if called with a complex argument. Try sqrt(complex(x)).
Stacktrace:
 [1] sqrt(::Int64) at ./math.jl:434

julia> sqrt(-1 + 0im)
0.0 + 1.0im
```

La notación de coeficiente numérico literal no funciona cuando se construye un número complejo a partir de variables. En su lugar, la multiplicación debe expresarse explícitamente:

```
julia> a = 1; b = 2; a + b*im
1 + 2im
```

Sin embargo, esto no es lo recomendable; En su lugar, utilice la función `complex()` para construir un valor complejo directamente de sus partes real e imaginaria:

```
julia> a = 1; b = 2; complex(a, b)
1 + 2im
```

Esta construcción evita las operaciones de multiplicación y adición.

`Inf` y `NaN` se propagan a través de números complejos en las partes real e imaginaria de un número complejo como se describe en la sección valores especiales en punto flotante section:

```
julia> 1 + Inf*im
1.0 + Inf*im

julia> 1 + NaN*im
1.0 + NaN*im
```

9.2 Números Racionales

Julia tiene un tipo numérico racional para representar razones exactas de enteros. Los racionales se construyen usando el operador `//`:

```
julia> 2//3
2//3
```

Si el numerador y el denominador de un racional tienen factores comunes, ellos son reducidos a los términos mínimos tales que el denominador sea no negativo:

```
julia> 6//9
2//3

julia> -4//8
-1//2

julia> 5//-15
-1//3

julia> -4//-12
1//3
```

Esta forma normalizada para una razón de enteros es única, por lo que la igualdad de valores racionales puede ser testada comprobando la igualdad del numerador y el denominador. El numerador estandarizado y el denominador de un valor racional pueden ser extraídos usando las funciones `numerator()` y `denominator()`:

```
julia> numerator(2//3)
2

julia> denominator(2//3)
3
```

La comparación directa de numerador y denominador no suele ser necesaria, ya que la aritmetica estándar y las operaciones de comparación están definidas para los valores racionales:

```
julia> 2//3 == 6//9
true

julia> 2//3 == 9//27
false
```

```
julia> 3//7 < 1//2
true

julia> 3//4 > 2//3
true

julia> 2//4 + 1//6
2//3

julia> 5//12 - 1//4
1//6

julia> 5//8 * 3//12
5//32

julia> 6//5 / 10//7
21//25
```

Los racionales pueden convertirse fácilmente en número en punto flotante:

```
julia> float(3//4)
0.75
```

La conversión de racional a punto flotante respeta la siguiente identidad para dos valores enteros cualesquiera a y b, con las excepciones de los casos `a == 0` and `b == 0`:

```
julia> a = 1; b = 2;

julia> isequal(float(a//b), a/b)
true
```

Construir valores racionales infinitos es aceptable:

```
julia> 5//0
1//0

julia> -3//0
-1//0

julia> typeof(ans)
Rational{Int64}
```

Sin embargo, no lo es tratar de construir un valor NaN `NaN` racional:

```
julia> 0//0
ERROR: ArgumentError: invalid rational: zero(Int64)//zero(Int64)
Stacktrace:
 [1] Rational{Int64}(::Int64, ::Int64) at ./rational.jl:13
 [2] //(::Int64, ::Int64) at ./rational.jl:40
```

Como es natural, el sistema de promoción hace que las interacciones con otros tipos numéricos se hagan sin esfuerzo alguno:

```
julia> 3//5 + 1
8//5

julia> 3//5 - 0.5
0.09999999999999998

julia> 2//7 * (1 + 2im)
2//7 + 4//7*im

julia> 2//7 * (1.5 + 2im)
0.42857142857142855 + 0.5714285714285714im

julia> 3//2 / (1 + 2im)
3//10 - 3//5*im

julia> 1//2 + 2im
1//2 + 2//1*im

julia> 1 + 2//3im
1//1 - 2//3*im

julia> 0.5 == 1//2
true

julia> 0.33 == 1//3
false

julia> 0.33 < 1//3
true

julia> 1//3 - 0.33
0.0033333333333332993
```

Chapter 10

Strings

Las cadenas son secuencias finitas de caracteres. Por supuesto, el verdadero problema viene cuando uno se pregunta qué es un carácter. Los caracteres con los que están familiarizados con los hablantes de inglés son las letras `A`, `B`, `C`, etc., junto con los números y los símbolos de puntuación comunes. Estos caracteres se estandarizan junto con una correspondencia a valores enteros entre 0 y 127 a través del estándar ASCII. Hay, por supuesto, muchos otros caracteres utilizados en lenguas no inglesas, incluyendo variantes de los caracteres ASCII con acentos y otras modificaciones, escrituras relacionadas como cirílico y griego, y escrituras no relacionadas en abosluto con ASCII o inglés, entre los que se incluyen árabe, chino, Hebreo, hindi, japonés y coreano. El estándar Unicode aborda las complejidades de lo que es exactamente un carácter, y es generalmente aceptado como el estándar definitivo que aborda este problema. Dependiendo de tus necesidades, puedes ignorar estas complejidades por completo y fingir que sólo existen caracteres ASCII, o puedes escribir código que pueda manejar cualquiera de los caracteres o codificaciones que se pueden encontrar al manejar texto no ASCII. Julia hace que el manejo de texto ASCII sencillo sea simple y eficiente, y el manejo de Unicode tan simple y eficiente como sea posible. En particular, puedes escribir código de cadenas con estilo C para procesar cadenas ASCII y funcionarán como se esperaba, tanto en términos de rendimiento como de semántica. Si dicho código encuentra texto no ASCII, fallará graciosamente con un mensaje de error claro, en lugar de introducir en silencio resultados corruptos. Cuando esto sucede, modificar el código para manejar datos no ASCII es sencillo.

Hay algunas características destacadas de alto nivel sobre las cadenas de caracteres en Julia:

- El tipo de concreto incorporado utilizado para cadenas (y literales de cadena) en Julia es `String`. Esto soporta el rango completo de caracteres Unicode a través de la codificación UTF-8. (se proporciona una función `transcode()` para convertir a/desde otras codificaciones Unicode).
- Todos los tipos de cadenas son subtipos del tipo abstracto `AbstractString` y los paquetes externos definen subtipos `AbstractString` adicionales (por ejemplo, para otras codificaciones). Si define una función que espera un argumento de cadena, debe declarar el tipo como `AbstractString` para aceptar cualquier tipo de cadena.
- Como C y Java, pero a diferencia de la mayoría de los lenguajes dinámicos, Julia tiene un tipo de primera clase que representa un solo carácter, llamado `Char`. Esto es sólo un tipo especial de bits de 32 bits cuyo valor numérico representa un punto de código Unicode.
- Como en Java, las cadenas son inmutables: el valor de un objeto `AbstractString` no se puede cambiar. Para construir un valor de cadena diferente, se construye una nueva cadena de partes de otras cadenas.
- Conceptualmente, una cadena es una *función parcial* de índices a caracteres: para algunos valores de índice, no se devuelve ningún valor de carácter y, en su lugar, se genera una excepción. Esto permite una indexación eficiente en cadenas por el índice de bytes de una representación codificada en lugar de por un índice de caracteres, que no se puede implementar de manera eficiente y sencilla para encodificaciones de anchura variable de cadenas Unicode.

10.1 Caracteres

Un valor `Char` representa un solo carácter: es sólo un *bitstype* de 32 bits con una representación literal especial y comportamientos aritméticos apropiados, cuyo valor numérico se interpreta como un punto de código Unicode. Aquí se muestra cómo se introducen y se muestran los valores `Char`:

```
julia> 'x'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)

julia> typeof(ans)
Char
```

Podemos convertir un `Char` a su valor entero (su punto de código) fácilmente:

```
julia> Int('x')
120

julia> typeof(ans)
Int64
```

En arquitecturas de 32 bits, `typeof(ans)` será `Int32`. Puede convertir un valor entero de nuevo a un `Char` fácilmente:

```
julia> Char(120)
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)
```

No todos los valores enteros son puntos de código Unicode válidos, pero por una cuestión de rendimiento, la conversión `Char()` no comprueba que cada valor de carácter sea válido. Si desea comprobar que cada valor convertido es un punto de código válido, utilice la función `isvalid()`:

```
julia> Char(0x110000)
'\U110000': Unicode U+110000 (category Cn: Other, not assigned)

julia> isvalid(Char, 0x110000)
false
```

A partir de este momento, los puntos de código Unicode válidos son `U+00` a `U+d7ff` y `U+e000` a `U+10ffff`. A estos no se les han asignado todavía significados inteligibles, ni son necesariamente interpretables por las aplicaciones, pero todos ellos se consideran caracteres Unicode válidos.

Puede introducir cualquier carácter Unicode entre comillas simples utilizando `\u` seguido de hasta cuatro dígitos hexadecimales o `\U` seguido de hasta ocho dígitos hexadecimales (el valor válido más largo sólo requiere seis):

```
julia> '\u0'
'\0': ASCII/Unicode U+0000 (category Cc: Other, control)

julia> '\u78'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)

julia> '\u2200'
'': Unicode U+2200 (category Sm: Symbol, math)

julia> '\U10ffff'
'\U10ffff': Unicode U+10ffff (category Cn: Other, not assigned)
```

Julia utiliza la configuración regional y de idioma de tu sistema para determinar qué caracteres se pueden imprimir tal cual y cuáles se deben imprimir utilizando las formas de entrada genéricas, escapadas con `\u` o `\U`. Además de estas formas de escape de Unicode, también se pueden usar todas las formas de entrada de escape tradicionales de C:

```
julia> Int('\0')
0

julia> Int('\t')
9

julia> Int('\n')
10

julia> Int('\e')
27

julia> Int('\x7f')
127

julia> Int('\177')
127

julia> Int('\xff')
255
```

Puedes hacer comparaciones y una cantidad limitada de aritmética con los valores `Char`:

```
julia> 'A' < 'a'
true

julia> 'A' <= 'a' <= 'Z'
false

julia> 'A' <= 'X' <= 'Z'
true

julia> 'x' - 'a'
23

julia> 'A' + 1
'B': ASCII/Unicode U+0042 (category Lu: Letter, uppercase)
```

10.2 Fundamentos de Cadenas

Los literales de cadenas están delimitados por comillas dobles o comillas dobles triples:

```
julia> str = "Hello, world.\n"
"Hello, world.\n"

julia> """Contains "quote" characters"""
"Contains \"quote\" characters"
```

Si desea extraer un carácter de una cadena, indéxelo:

```
julia> str[1]
'H': ASCII/Unicode U+0048 (category Lu: Letter, uppercase)

julia> str[6]
',': ASCII/Unicode U+002c (category Po: Punctuation, other)

julia> str[end]
'\n': ASCII/Unicode U+000a (category Cc: Other, control)
```

Toda la indexación en Julia está basada en 1: el primer elemento de cualquier objeto indexado medidante enteros se encuentra en el índice 1, y el último elemento se encuentra en el índice n, cuando la cadena tiene una longitud de n.

En cualquier expresión de indexación, puede usarse la palabra clave end como una abreviatura para el último índice (calculado mediante `endof(str)`). Puede realizar operaciones aritméticas y otras con end, como si de un valor normal se tratara:

```
julia> str[end-1]
'.': ASCII/Unicode U+002e (category Po: Punctuation, other)

julia> str[end÷2]
' ': ASCII/Unicode U+0020 (category Zs: Separator, space)
```

Usar un índice menor que 1 o mayor que end lanza un error:

```
julia> str[0]
ERROR: BoundsError: attempt to access "Hello, world.\n"
  at index [0]
[...]

julia> str[end+1]
ERROR: BoundsError: attempt to access "Hello, world.\n"
  at index [15]
[...]
```

También puedes extraer una subcadena usando indexación mediante un rango:

```
julia> str[4:9]
"lo, wo"
```

Nótese que las expresiones `str[k]` y `str[k:k]` no dan el mismo resultado:

```
julia> str[6]
',': ASCII/Unicode U+002c (category Po: Punctuation, other)

julia> str[6:6]
","
```

La primera es un valor carácter de tipo `Char`, mientras que la segunda es un valor cadena que tiene un único carácter. En Julia se trata de cosas muy diferentes.

10.3 Unicode y UTF-8

Julia soporta totalmente caracteres y cadenas Unicode. Como se ha comentado anteriormente, en literales de caracteres, los puntos de código Unicode se pueden representar usando las secuencias de escape Unicode `\u` y `\U`, así como todas las secuencias de escape C estándar. Éstos también se pueden utilizar para escribir literales de cadena:

```
julia> s = "\u2200 x \u2203 y"
" x  y"
```

Si estos caracteres Unicode se muestran como escapes o se muestran como caracteres especiales depende de la configuración regional de tu terminal y su compatibilidad con Unicode. Los literales de cadena se codifican utilizando la codificación UTF-8. UTF-8 es una codificación de ancho variable, lo que significa que no todos los caracteres están codificados en el mismo número de bytes. En UTF-8, los caracteres ASCII -es decir, aquellos con puntos de código inferiores a 0x80 (128) - están codificados como lo están en ASCII, usando un solo byte, mientras que los puntos de código 0x80 y superiores se codifican utilizando múltiples bytes (hasta cuatro por carácter). Esto significa que no todos los índices de bytes en una cadena UTF-8 es necesariamente un índice válido para un carácter. Si indexas una cadena en un índice de bytes no válido, se genera un error:

```
julia> s[1]
'': Unicode U+2200 (category Sm: Symbol, math)

julia> s[2]
ERROR: UnicodeError: invalid character index
[...]

julia> s[3]
ERROR: UnicodeError: invalid character index
[...]

julia> s[4]
' ': ASCII/Unicode U+0020 (category Zs: Separator, space)
```

En este caso, el carácter es un carácter de tres bytes, por lo que los índices 2 y 3 no son válidos y el índice del siguiente carácter es 4; este siguiente índice válido puede ser calculado con `nextind(s,1)`, y el siguiente índice después de éste con `nextind(s,4)` y así sucesivamente.

Debido a las codificaciones de longitud variable, el número de caracteres de una cadena (dada por `length(s)`) no siempre lo mismo que el último índice. Si se itera a través de los índices 1 hasta `endof(s)` y se indexa en s, la secuencia de caracteres devueltos cuando no se lanzan errores es la secuencia de caracteres que contiene la cadena s. Por tanto, tenemos la identidad de que `length(s) <= endof(s)`, ya que cada carácter en una cadena debe tener su propio índice. La siguiente es una forma ineficaz y verbosa de iterar a través de los caracteres de s:

```
julia> for i = 1:endof(s)
           try
               println(s[i])
           catch
               # ignore the index error
           end
       end


x




y
```

Las líneas en blanco en realidad tienen espacios en ellos. Afortunadamente, el idioma anterior incómodo es innecesario para iterar a través de los caracteres de una cadena, ya que se puede utilizar la cadena como un objeto iterable, sin que se requiera el manejo de excepciones:

```
julia> for c in s
           println(c)
       end
```

```
x



y
```

Julia utiliza la codificación UTF-8 de forma predeterminada y el soporte para nuevas codificaciones puede agregarse mediante paquetes. Por ejemplo, el paquete LegacyStrings.jl implementa los tipos `UTF16String` y `UTF32String`. Una mayor discusión sobre otras codificaciones y cómo implementar el soporte para ellas está más allá del alcance de este documento por el momento. Para más información sobre los problemas de codificación UTF-8, consulte la sección siguiente sobre literales byte array. La función `transcode()` se proporciona para convertir datos entre las distintas codificaciones UTF-xx, principalmente para trabajar con datos y bibliotecas externas.

10.4 Concatenación

Una de las operaciones de cadena más comunes y útiles es la concatenación:

```
julia> greet = "Hello"
"Hello"

julia> whom = "world"
"world"

julia> string(greet, ", ", whom, ".\n")
"Hello, world.\n"
```

Julia también proporciona el operador `*` para concatenar cadenas:

```
julia> greet * ", " * whom * ".\n"
"Hello, world.\n"
```

Aunque * puede parecer una elección sorprendente a los usuarios de lenguajes que proporcionan + para concatenación de cadenas, este uso de *tiene precedentes en matemáticas, particularmente en álgebra abstracta.

En matemáticas, + suele denotar una operación *conmutativa*, donde el orden de los operandos no importa. Un ejemplo de esto es la suma de matrices, donde `A + B == B + A` para dos matrices cualesquiera `A` y `B` que tengan la misma forma. En contraste, `*` suele denotar una operación no conmutativa, donde el orden de los operandos *importa*. Un ejemplo de esto es la multiplicación de matrices donde, en general, `A * B != B * A`. Como con la multiplicación de matrices, la concatenación es no conmutativa: `greet * whom != whom * greet`. Por tanto, `*` es una elección más natural para el operador infijo de concatenación, consistente con el uso matemático común.

Más precisamente, el conjunto de todas las cadenas *S* de longitud finita junto con el operador de concatenación `*` forman un monoide libre (S, `*`). El elemento identidad de este conjunto es la cadena vacía "". Siempre que un monoide libre es no conmutativo, la operación suele ser representada por `\cdot`, `*`, o un símbolo similar, en luga de con + que implica conmutatividad.

10.5 Interpolación

Construir cadenas mediante concatenación puede llegar a ser un poco engorroso, sin embargo. Para reducir la necesidad de estas llamadas verbosas a `string()` o multiplicaciones repetidas, Julia permite la interpolación en literales de cadena usando $, como en Perl:

```
julia> "$greet, $whom.\n"
"Hello, world.\n"
```

Esto es más legible y conveniente, y equivalente a la concatenación de cadena anterior – el sistema rescribe este aparente literal de cadena única en una concatenación de literales de cadena con variables.

La expresión completa más corta después de $ se toma como la expresión cuyo valor debe ser interpolado en la cadena. Por lo tanto, puede interpolar cualquier expresión en una cadena usando paréntesis:

```
julia> "1 + 2 = $(1 + 2)"
"1 + 2 = 3"
```

Tanto la concatenación como la interpolación de cadena llaman a `string()` para convertir objetos al formato de cadena. La mayoría de los objetos que no son `AbstractString` se convierten en cadenas que se corresponden estrechamente con la forma en que se introducen como expresiones literales:

```
julia> v = [1,2,3]
3-element Array{Int64,1}:
 1
 2
 3

julia> "v: $v"
"v: [1, 2, 3]"
```

`string()` es la identidad para los valores `AbstractString` y `Char` values, por lo que estos se interpolan en cadenas como ellos mismos, sin entrecomillar y sin escapar:

```
julia> c = 'x'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)

julia> "hi, $c"
"hi, x"
```

Para incluir un literal $ en una cadena, lo escaparemos con un backslash:

```
julia> print("I have \$100 in my account.\n")
I have $100 in my account.
```

10.6 Literales cadena con triples comillas

Cuando las cadenas se crean utilizando comillas triples (`"""..."""`) tienen un comportamiento especial que puede ser útil para crear bloques de texto más largos. En primer lugar, si la apertura """ es seguida por una nueva línea, la nueva línea se quita de la cadena resultante:

```
"""hello"""
```

es equivalente a

```
"""
hello"""
```

pero

```
"""

hello"""
```

contendrá un literal *new line* al principio. Los espacios en blanco no se modifican. Pueden contener símbolos " sin escapar. Las cadenas de triple comilla también se dedican al nivel de la línea menos indentada. Esto es útil para definir cadenas dentro del código que está sangrado Por ejemplo:

```
julia> str = """
           Hello,
           world.
         """
"  Hello,\n  world.\n"
```

En este caso la línea final (vacía) antes del cierre """ establece el nivel de indentación.

Tenga en cuenta que las saltos de línea en cadenas literales, sean de una sola o triple comilla, resultan en un carácter de línea nueva (LF) `\n` en la cadena, incluso si su editor usa una combinación de retorno de carro (CR) o CRLF para finalizar líneas. Para incluir un CR en una cadena, utilice un escape explícito `\r`; Por ejemplo, puede introducir la cadena literal `"una línea CRLF que termina \r \n"`.

10.7 Operaciones Comunes

Podemos comparar cadenas lexicográficamente usando los operadores de comparación estandard:

```
julia> "abracadabra" < "xylophone"
true

julia> "abracadabra" == "xylophone"
false

julia> "Hello, world." != "Goodbye, world."
true

julia> "1 + 2 = 3" == "1 + 2 = $(1 + 2)"
true
```

La función `search()` permite buscar el índice de una carácter en una cadena:

```
julia> search("xylophone", 'x')
1

julia> search("xylophone", 'p')
5

julia> search("xylophone", 'z')
0
```

Y se puede arrancar la búsqueda de un carácter a partir de un desplazamiento proporcionado por un tercer argumento:

```
julia> search("xylophone", 'o')
4

julia> search("xylophone", 'o', 5)
7

julia> search("xylophone", 'o', 8)
0
```

La función `contains()` se usa para comprobar si una subcadena está contenida en una cadena:

```
julia> contains("Hello, world.", "world")
true

julia> contains("Xylophon", "o")
true

julia> contains("Xylophon", "a")
false

julia> contains("Xylophon", 'o')
ERROR: MethodError: no method matching contains(::String, ::Char)
Closest candidates are:
  contains(!Matched::Function, ::Any, !Matched::Any) at reduce.jl:664
  contains(::AbstractString, !Matched::AbstractString) at strings/search.jl:378
```

Este último error es debido a que 'o' es un literal carácter, y `contains()` es una función genérica que busca subsecuencias. Para buscar un elemento en una secuencia, debemos usar la función `in()` en lugar de la anterior.

`repeat()` y `join()` son otras dos funciones de cadena muy útiles:

```
julia> repeat(".:Z:.", 10)
".:Z:..:Z:..:Z:..:Z:..:Z:..:Z:..:Z:..:Z:..:Z:."

julia> join(["apples", "bananas", "pineapples"], ", ", " and ")
"apples, bananas and pineapples"
```

Algunas otras funciones útiles son:

- `endof(str)` el índice máximo (byte) que se puede utilizar para indexar en `str`.
- `length(str)` el número de caracteres en `str`.
- `i = start(str)` da el primer índice válido en el que se puede encontrar un carácter en 'str (típicamente 1).
- `c, j = next(str,i)` devuelve el carácter siguiente en o después del índice `i` y el siguiente índice de carácter válido que sigue a éste. Con `start()` y `endof()`, se puede utilizar para iterar a través de los caracteres en str'.
- `ind2chr(str,i)` da el número de caracteres en `str` hasta e incluyendo cualquiera en el índice `i`.
- `chr2ind(str,j)` da el índice en el cual ocurre el carácter `j`-ésimo en `str`.

10.8 Literales cadena no estándar

Hay situaciones en las que se desea construir una cadena o utilizar semántica de cadenas, pero el comportamiento de la construcción de cadena estándar no es lo que se necesita. Para este tipo de situaciones, Julia proporciona literales cadena no estándar. Un literal de cadena no estándar es como una cadena literal normal de doble comilla, pero va inmediatamente precedido de un identificador y no se comporta como un literal de cadena normal. El convenio es que los literales no estándar con prefijos en mayúsculas producen objetos cadena reales, mientras que aquellos con prefijos en minúsculas producen objetos que no cadena, como arrays de bytes o expresiones regulares compiladas. Las expresiones regulares, literales arrays de bytes y literales de números de versión, como se describe a continuación, son algunos ejemplos de literales de cadena no estándar. Otros ejemplos se dan en la sección Metaprogramación.

10.9 Expresiones Regulares

Julia tiene expresiones regulares compatibles con Perl (expresiones regulares), tal y como las proporciona la biblioteca PCRE. Las expresiones regulares se relacionan con las cadenas de dos maneras: la conexión obvia es que las expresiones regulares se utilizan para encontrar patrones regulares en cadenas; La otra conexión es que las expresiones regulares se introducen ellas mismas como cadenas, que se analizan en una máquina de estado que puede utilizarse para buscar patrones en cadenas de forma eficiente. En Julia, las expresiones regulares se introducen usando literales de cadena no estándar prefijados con varios identificadores comenzando por `r`. El literal de expresión regular más básico sin ninguna opción activada sólo utiliza `r"..."`:

```
julia> r"^\s*(?:#|$)"
r"^\s*(?:#|$)"

julia> typeof(ans)
Regex
```

Para comprobar si una *regex* se corresponde con una cadena, se utiliza `ismatch()`:

```
julia> ismatch(r"^\s*(?:#|$)", "not a comment")
false

julia> ismatch(r"^\s*(?:#|$)", "# a comment")
true
```

Como puede verse aquí, `ismatch()` simplemente devuelve `true` o `false`, indicando si la *regex* dada coincide o no con la cadena. Es común, sin embargo, que uno quiera saber no sólo si una cadena coincide, sino también *cómo* coincide. Para capturar esta información sobre una coincidencia, se utiliza la función `match()`:

```
julia> match(r"^\s*(?:#|$)", "not a comment")

julia> match(r"^\s*(?:#|$)", "# a comment")
RegexMatch("#")
```

Si la expresión regular no coincide con la cadena dada, `match()` devuelve `nothing` – un valor especial que no imprime nada en el indicador interactivo. Aparte de no imprimir, es un valor completamente normal, como podemos comprobar en el siguiente código:

```
m = match(r"^\s*(?:#|$)", line)
if m === nothing
    println("not a comment")
```

```
else
    println("blank or comment")
end
```

Si la expresión regular coincide, el valor devuelto por `match()` es un objeto `RegexMatch`. Estos objetos registran cómo coincide la expresión, incluyendo la subcadena que coincide con el patrón y cualquier subcadena capturada, si la hay. Este ejemplo sólo captura la parte de la subcadena que coincide, pero tal vez quisiéramos capturar cualquier texto no en blanco después del carácter de comentario. Podríamos hacer lo siguiente:

```
julia> m = match(r"^\s*(?:#\s*(.*?)\s*$|$)", "# a comment ")
RegexMatch("# a comment ", 1="a comment")
```

Al invocar a `match()`, tenemos la opción de especificar un índice en el que iniciar la búsqueda. Por ejemplo:

```
julia> m = match(r"[0-9]","aaaa1aaaa2aaaa3",1)
RegexMatch("1")

julia> m = match(r"[0-9]","aaaa1aaaa2aaaa3",6)
RegexMatch("2")

julia> m = match(r"[0-9]","aaaa1aaaa2aaaa3",11)
RegexMatch("3")
```

Puede extraer la siguiente información de un objeto `RegexMatch`:

- La totalidad de la subcadena emparejada: `m.match`
- Las subcadenas capturadas como una matriz de cadenas: `m.captures`
- El desplazamiento en el que comienza la coincidencia del patrón: `m.offset`
- Los desplazamientos de las subcadenas capturadas como un vector: `m.offsets`

Para cuando una captura no coincide, en lugar de una subcadena, `m.captures` no contiene nada en esa posición, y `m.offsets` tiene un desplazamiento de cero (recuerde que los índices en Julia son *1-based*, por lo que un desplazamiento de cero en una cadena es inválido). Aquí hay un par de ejemplos algo artificiales:

```
julia> m = match(r"(a|b)(c)?(d)", "acd")
RegexMatch("acd", 1="a", 2="c", 3="d")

julia> m.match
"acd"

julia> m.captures
3-element Array{Union{SubString{String}, Void},1}:
 "a"
 "c"
 "d"

julia> m.offset
1

julia> m.offsets
```

```
3-element Array{Int64,1}:
 1
 2
 3

julia> m = match(r"(a|b)(c)?(d)", "ad")
RegexMatch("ad", 1="a", 2=nothing, 3="d")

julia> m.match
"ad"

julia> m.captures
3-element Array{Union{SubString{String}, Void},1}:
 "a"
 nothing
 "d"

julia> m.offset
1

julia> m.offsets
3-element Array{Int64,1}:
 1
 0
 2
```

Es conveniente que las capturas sean retornadas como un array para que uno pueda usar la sintaxis de desestructurante para enlazarlas a variables locales:

```
julia> first, second, third = m.captures; first
"a"
```

Las capturas también está accesibles indexando el objeto `RegexMatch` con el número o nombre del grupo captura:

```
julia> m=match(r"(?<hour>\d+):(?<minute>\d+)","12:45")
RegexMatch("12:45", hour="12", minute="45")

julia> m[:minute]
"45"

julia> m[2]
"45"
```

Las capturas pueden referenciarse en una cadena de sustitución cuando se utiliza `replace()` utilizando `\n` para referirse al grupo de captura n-ésimo y prefijando la cadena de subsitución con `s`. El grupo de captura `0` se refiere a todo el objeto de coincidencia. Los grupos de captura nombrados se pueden hacer referencia en la sustitución con g<groupname>. Por ejemplo:

```
julia> replace("first second", r"(\w+) (?<agroup>\w+)", s"\g<agroup> \1")
"second first"
```

Los grupos de captura numerados pueden también ser referenciados como `\g<n>` para evitar ambigüedad, como en:

```
julia> replace("a", r".", s"\g<0>1")
"a1"
```

Puedes modificar el comportamiento de las expresiones regulares mediante una combinación de los flags `i`, `m`, `s` y `x` después de la marca de comillas dobles de cierre. Estas banderas tienen el mismo significado que en Perl, tal y como se describe en este fragmento de la página de manual del referencia de Perl:

```
i   Do case-insensitive pattern matching.

    If locale matching rules are in effect, the case map is taken
    from the current locale for code points less than 255, and
    from Unicode rules for larger code points. However, matches
    that would cross the Unicode rules/non-Unicode rules boundary
    (ords 255/256) will not succeed.

m   Treat string as multiple lines.  That is, change "^" and "$"
    from matching the start or end of the string to matching the
    start or end of any line anywhere within the string.

s   Treat string as single line.  That is, change "." to match any
    character whatsoever, even a newline, which normally it would
    not match.

    Used together, as r""ms, they let the "." match any character
    whatsoever, while still allowing "^" and "$" to match,
    respectively, just after and just before newlines within the
    string.

x   Tells the regular expression parser to ignore most whitespace
    that is neither backslashed nor within a character class. You
    can use this to break up your regular expression into
    (slightly) more readable parts. The '#' character is also
    treated as a metacharacter introducing a comment, just as in
    ordinary code.
```

Por ejemplo, la siguiente regex tiene activados los tres *flags*:

```
julia> r"a+.*b+.*?d$"ism
r"a+.*b+.*?d$"ims

julia> match(r"a+.*b+.*?d$"ism, "Goodbye,\nOh, angry,\nBad world\n")
RegexMatch("angry,\nBad world")
```

Las cadenas *regex* con triples comillas, de la forma `r"""..."""` están también soportadas (y puede ser conveniente para expresiones regulares que contengan comillas o caracteres de salto de línea).

10.10 Byte Array Literals

Otro literal de cadena no estándar útil es el literal de cadena de bytes: `b "..."`. Esta forma nos permite usar la notación de cadena para expresar arrays de bytes literales, es decir, arrays de valores `UInt8`. Las reglas para los literales de arrays de bytes son las siguientes:

- Los caracteres ASCII y los escapes ASCII producen un solo byte.
- `\x` y las secuencias de escape octales producen el *byte* correspondiente al valor de escape.
- Las secuencias de escape Unicode producen una secuencia de bytes que codifican ese punto de código en UTF-8.

Hay una cierta superposición entre estas reglas ya que el comportamiento de `\x` y escapes octales menores de `0x80` (128) están cubiertos por las dos primeras reglas, pero aquí estas reglas están de acuerdo. Juntas, estas reglas permiten usar fácilmente caracteres ASCII, valores arbitrarios de bytes y secuencias UTF-8 para producir matrices de bytes. Aquí hay un ejemplo usando los tres:

```
julia> b"DATA\xff\u2200"
8-element Array{UInt8,1}:
 0x44
 0x41
 0x54
 0x41
 0xff
 0xe2
 0x88
 0x80
```

La secuencia ASCII "DATA" corresponde a los bytes 68, 65, 84, 65. `\xff` produce el byte simple 255. El escape Unicode `\u2200` está codificado en UTF-8 como los tres bytes 226, 136, 128. Nótese que la matriz de bytes resultante no corresponde a una cadena UTF-8 válida - si intenta utilizar esto como una cadena literal normal, obtendrá un error de sintaxis:

```
julia> "DATA\xff\u2200"
ERROR: syntax: invalid UTF-8 sequence
```

Observe también la distinción significativa entre `\xff` y `\uff`: la secuencia de escape anterior codifica el *byte 255*, mientras que la última secuencia de escape representa el *punto de código 255*, que se codifica como dos bytes en UTF-8:

```
julia> b"\xff"
1-element Array{UInt8,1}:
 0xff

julia> b"\uff"
2-element Array{UInt8,1}:
 0xc3
 0xbf
```

En los literales de caracteres, esta distinción se pasa por alto y `\xff` está autorizado a representar el punto de código 255, porque los caracteres *siempre* representan puntos de código. En las cadenas, sin embargo, los escapes `\x` siempre representan bytes, no puntos de código, mientras que los escapes `\u` y `\U` siempre representan puntos de código, que están codificados en uno o más bytes. Para los puntos de código inferiores a `\u80`, ocurre que la codificación UTF-8 de cada punto de código es sólo el byte producido por el escape `\x` correspondiente, por lo que la distinción puede ignorarse con seguridad. Sin embargo, para los escapes `\x80` a `\xff` en comparación con `\u80` a `\uff`, existe una diferencia importante: el primero escapa a todos los bytes sencillos de codificación, los cuales -a menos que sean seguidos por bytes de continuación muy específicos- no forman UTF-8 válido, mientras que los últimos escapes representan puntos de código Unicode con codificaciones de dos bytes.

Si todo esto es muy confuso, intente leer "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets". Es una excelente introducción a Unicode y UTF-8, y puede ayudar a aliviar cierta confusión sobre el asunto.

10.11 Literales Número de Versión

Los números de versión se pueden expresar fácilmente con literales de cadena no estándar del forma `v"..."`. Los literales de número de versión crean objetos `VersionNumber` que siguen las especificaciones del control de versiones semánticas y, por lo tanto, se componen de valores numéricos mayor, menor y de parche, seguidos de anotaciones alfanuméricas de pre-liberación y construcción. Por ejemplo, `v "0.2.1-rc1 + win64"` se divide en versión principal `0`, versión secundaria 2, versión de revisión 1, `rc1` de pre-lanzamiento y construcción `win64`. Al introducir una versión literal, todo excepto el número de versión principal es opcional, por ejemplo, `v"0.2"` es equivalente a `v"0.2.0"` (con anotaciones previas / de compilación vacías), `v"2"` equivale a `v"2.0.0"`, y así sucesivamente.

Los objetos `VersionNumber` son en su mayoría útiles para comparar fácilmente y correctamente dos (o más) versiones. Por ejemplo, la constante `VERSION` contiene el número de versión de Julia como un objeto `VersionNumber` y, por lo tanto, se puede definir algún comportamiento específico de la versión utilizando declaraciones simples como:

```
if v"0.2" <= VERSION < v"0.3-"
    # do something specific to 0.2 release series
end
```

Obsérvese que en el ejemplo anterior se utiliza el número de versión no estándar `v"0.3-"`, con un guión `-` en cola: esta notación es una extensión Julia del estándar, y se usa para indicar una versión que es más baja que cualquier versión `0.3`, Incluyendo todas sus pre-lanzamientos. Por lo tanto, en el ejemplo anterior, el código sólo se ejecuta con versiones estable `0.2` y excluye las versiones `v"0.3.0-rc1"`. Para permitir también versiones `0.2` inestables (es decir, pre-liberación), la verificación del límite inferior debería modificarse de la siguiente manera: `v"0.2-" <= VERSION`.

Otra extensión de especificación de versión no estándar permite usar un `+` de cola para expresar un límite superior en las versiones de compilación, por ej. `VERSIÓN> v "0.2-rc1 +"` se puede utilizar para significar cualquier versión por encima de `0.2-rc1` y cualquiera de sus compilaciones: devolverá `false` para la versión `v"0.2-rc1+win64"` y `true` para `v"0.2-rc2"`.

Es una buena práctica utilizar estas versiones especiales en comparaciones (en particular, el valor `-`de cola siempre debe utilizarse en los límites superiores a menos que haya una buena razón para no hacerlo), pero no deben utilizarse como el número de versión real de nada, ya que son inválidos en el esquema de versiones semánticas.

Además de ser utilizados por la constante `VERSION`, los objetos `VersionNumber` son ampliamente utilizados en el módulo Pkg, para especificar las versiones de paquetes y sus dependencias.

10.12 Raw String Literals

Las cadenas en bruto (*raw*) sin interpolación o *unescaping* pueden ser expresadas con literales cadena no estándar de la forma `raw"..."`. Los literales cadena en bruto crean objetos `String` ordinarios que contienen los contenidos encerrados exactamente como entrados sin interpolación ni separación. Esto es útil para cadenas que contiene código o marcado en otros idiomas que usan `$` o `\` como caracteres especiales. La excepción son las comillas que aún deben ser escapadas, por ejemplo, `raw" \ ""`es equivalente a`"\" "`.

Chapter 11

Funciones

En Julia, una función es un objeto que hace corresponde una tupla de valores argumentos en un valor de retorno. Las funciones de Julia no son funciones matemáticas puras, en el sentido de que pueden alterar y ser afectadas por el estado global del programa. La sintaxis básica a definir funciones en Julia es:

```
julia> function f(x,y)

           x + y

       end
f (generic function with 1 method)
```

Hay una segunda sintaxis, más concisa, para definir una función en Julia. La declaración de función tradicional mostrada anteriormente es equivalente a la denominada "forma de asignación". Por ejemplo:

```
julia> f(x,y) = x + y
f (generic function with 1 method)
```

En esta segunda forma, el cuerpo de la función debe ser una sola expresión, aunque puede tratarse de una expresión compuesta (see Expresiones Compuestas). Estas definiciones de función cortas y simples son comunes en Julia. La sintaxis de funciones cortas es, por tanto, bastante idiomática, reduciendo considerablemente tanto la escritura como el ruido visual.

Para invocar una función se usa la sintaxis tradicional basada en el uso del paréntesis:

```
julia> f(2,3)
5
```

Sin usar paréntesis, la expresión `f` se refiere al objeto función, y puede ser tratada como cualquier otro valor:

```
julia> g = f;

julia> g(2,3)
5
```

Y, como en el caso de las variables, podemos usar Unicode en el caso de los nombres de función:

```
julia> ∑(x,y) = x + y
∑ (generic function with 1 method)

julia> ∑(2, 3)
5
```

11.1 Comportamiento del Paso de Argumentos

Los argumentos de función en Julia siguen un convenio denominado a veces ”paso por compartición”, que significa que los valores no son copiados cuando se pasan a las funciones. Los argumentos de las funciones actúan ellos mismos como nuevos enlaces a variable (nuevas localizaciones que pueden referirse a valores) pero los valores a los que se refieren son idénticos a los valores pasados. Las modificaciones a valores mutables (tales como los Arrays) hechos dentro de la función serán visibles desde fuera de ésta. Este es el mismo comportamiento que presenta Scheme, la mayoría de versiones de Lisp, Python, Ruby y Perl, entre otros lenguajes dinámicos.

11.2 La palabra clave `return`

El valor devuelto por una función es el valor de la última expresión evaluada, el cual, por defecto, es la última expresión en el cuerpo de definición de la función. En la función `f`, mostrada en la sección anterior, el valor devuelto sería la suma `x + y`. Como en C y la mayoría de los demás lenguajes imperativos o funcionales, la palabra clave `return` causa que la función retorne inmediatamente, proporcionando una función cuyo valor es devuelto:

```
function g(x,y)
    return x * y
    x + y
end
```

Como las definiciones a función pueden ser introducidas en una sesión interactiva, es muy sencillo comparar estas definiciones:

```
julia> f(x,y) = x + y
f (generic function with 1 method)

julia> function g(x,y)

           return x * y

           x + y

       end
g (generic function with 1 method)

julia> f(2,3)
5

julia> g(2,3)
6
```

Por supuesto, en una función con cuerpo puramente lineal como g, el uso de `return` es irrelevante ya que la expresión `x+y` nunca va a ser evaluada, por lo que podríamos hacer que `x*y` fuese la última línea de la función y omitir el `return`. Sin embargo, cuando hacemos uso de esta instrucción junto con otras de control de flujo, el resultado puede ser muy interesante. Aquí, por ejemplo, hay una función que calcula la longitud de la hipotenusa de un triángulo equilátero correcto con catetos de longitudes `x` e `y`, evitando un desbordamiento:

```
julia> function hypot(x,y)

           x = abs(x)
```

```
            y = abs(y)

            if x > y

                r = y/x

                return x*sqrt(1+r*r)

            end

            if y == 0

                return zero(x)

            end

            r = x/y

            return y*sqrt(1+r*r)

        end
hypot (generic function with 1 method)

julia> hypot(3, 4)
5.0
```

Hay tres posibles puntos de retorno en esta función, devolviendo los valores de tres expresiones diferentes, dependiendo de los valores de x e y. El `return` de la última línea podría ser omitido ya que es la última expresión.

11.3 Operators Are Functions

En Julia, la mayoría de los operadores son funciones con soporte para una sintaxis especial (la excepción a esta regla son las operaciones con una semática de evaluación especial, tales como && y `||`. Estos operadores no pueden ser funciones porque la Evaluación en Cortocircuito requiere que sus operandos no sean evaluados antes de la evaluación del operador). De acuerdo con ésto, podemos usar listas de argumentos entre paréntesis, tal como en cualquier otra función:

```
julia> 1 + 2 + 3
6

julia> +(1,2,3)
6
```

La forma infija es equivalente a la forma de aplicación función. De hecho, la primera es transformada para producir la llamada a función internamente. Esto también significa que puedes asignar y pasar operadores tales como `+()` y `*()`, tal y como se hace con otros valores función:

```
julia> f = +;

julia> f(1,2,3)
6
```

Sin embargo, cuando se usa el formato de función, como `f`, no se puede usar notación infija.

11.4 Operadores con Nombres Especiales

Hay unas pocas operaciones especiales que corresponden a llamadas a funciones con nombres no obvios. Estas son las siguientes:

Expresión	Llamada
`[A B C ...]`	`hcat()`
`[A; B; C; ...]`	`vcat()`
`[A B; C D; ...]`	`hvcat()`
`A'`	`ctranspose()`
`A.'`	`transpose()`
`1:n`	`colon()`
`A[i]`	`getindex()`
`A[i]=x`	`setindex!()`

Estasa funciones están incluidas en el módulo `Base.0perators` incluso aunque no tengan nombres como operadores.

11.5 Funciones Anónimas

Las funciones en Julia son objetos de primera clase: ellas pueden ser asignadas a variables y ser invocadas usando la sintaxis estándar de llamadas a función desde la variable a la que han sido asignadas. Ellas pueden ser usadas como argumentos y ser devueltas como valores. Ellas pueden también ser usadas de forma anónima sin dárseles un nombre, usando alguna de estas sintaxis:

```
julia> x -> x^2 + 2x - 1
(::#1) (generic function with 1 method)

julia> function (x)

           x^2 + 2x - 1

       end
(::#3) (generic function with 1 method)
```

Esto crea una función que toma un argumento `x` y devuelve el valor del polinomio `x^2 + 2x - 1`. Nótese que el resultado es una función genérica, pero con un nombre generado por el compilador basado en una numeración consecutiva.

El uso primario de las funciones anónimas es pasarlas a funciones que toman otras funciones como argumentos. Un ejemplo clásico es `map()`, , que aplica una función a cada valor de un array y devuelve un nuevo array que contienen los valores resultantes:

```
julia> map(round, [1.2,3.5,1.7])
3-element Array{Float64,1}:
 1.0
 4.0
 2.0
```

Esto está bien si ya exite una función que efectúa la transformación que uno desea para pasarla como primer argumento de `map()`. Sin embargo, no es frecuente que exista este tipo de función. En estas situaciones, el constructor de la función anónima permite una fácil creación de un objeto función de un solo uso sin necesidad de asignarle un nombre:

```
julia> map(x -> x^2 + 2x - 1, [1,3,-1])
3-element Array{Int64,1}:
  2
 14
 -2
```

Para escribir funciones anónimas que aceptan múltiples argumentos puede utilizarse la sintaxis `(x,y,z) -> 2x + y +z`. Una función anónima con cero argumentos se escribe como `() -> 3`. La idea de una función sin argumentos puede parecer extraña, pero es útil para demorar un cálculo. En este uso, un bloque de código es envuelto en una función con cero argumentos, el cual es después invocado mediante una llamada como `f()`.

11.6 Retorno de Múltiples Valores

En Julia, uno devuelve una tupla para simular el retorno de múltiples valores. Sin embargo, como las tuplas puede salteadas y destruidas sin necesitar paréntesis, podemos proporcionar una ilusión de que se están devolviendo múltiples valores. Por ejemplo, la siguiente función devuelve un par de valores:

```
julia> function foo(a,b)
           a+b, a*b
       end
foo (generic function with 1 method)
```

Si invocamos esta función en una sesión interactiva sin asignar los valores en ningún sitio, comprobaremos que la función devuelve una tupla:

```
julia> foo(2,3)
(5, 6)
```

Un uso típico de tal par de valores devueltos es extraer cada valor en una variable. Julia soporta la ”desestructuración” simple de una tupla que facilita esto:

```
julia> x, y = foo(2,3)
(5, 6)

julia> x
5

julia> y
6
```

Y también podemos devolver múltiples valores mediante el uso explícito de la palabra clave `return`:

```
function foo(a,b)
    return a+b, a*b
end
```

Esto tiene exactamente el mismo efecto que la definición anterior de `foo`.

11.7 Funciones con argumentos variables (varargs)

Suele ser muy conveniente ser capaz de escribir funciones que toman un número arbitrario de argumentos. Estas funciones se conocen como *funciones vararg*. Podemos definir funciones de tal tipo poniendo puntos suspensivos ... después del último argumento.

```
julia> bar(a,b,x...) = (a,b,x)
bar (generic function with 1 method)
```

Las variables a y b están asociadas a los dos primeros argumentos como es natural, y la variable x se asocia a una colección, iterable de cero o más valores pasados a la función `bar` después de estos dos argumentos:

```
julia> bar(1,2)
(1, 2, ())

julia> bar(1,2,3)
(1, 2, (3,))

julia> bar(1, 2, 3, 4)
(1, 2, (3, 4))

julia> bar(1,2,3,4,5,6)
(1, 2, (3, 4, 5, 6))
```

En todos los casos, x es asociada a una tupla con el resto de valores pasados a la función.

Es posible restringir el número de argumentos pasados como argumento variable. Esto se discutirá más adelante en la sección métodos *vararg* restringidos paramétricamente.

Como contraposición, es frecuente manejar la división de los valores contenidos en una colección iterable en una llamada a función como argumentos individuales. Para hacer eso, se utilizará la notación de puntos suspensivos, pero esta vez en la llamada a función.

```
julia> x = (3, 4)
(3, 4)

julia> bar(1,2,x...)
(1, 2, (3, 4))
```

En este caso hay una tupla que se divide en una llamada *vararg* precisamente donde está el número de argumentos variable. Esa necesidad no tiene por qué ser el caso:

```
julia> x = (2, 3, 4)
(2, 3, 4)

julia> bar(1,x...)
(1, 2, (3, 4))

julia> x = (1, 2, 3, 4)
(1, 2, 3, 4)

julia> bar(x...)
(1, 2, (3, 4))
```

Además, el objeto iterable dividido durante la llamada a función no tiene que ser una tupla:

```
julia> x = [3,4]
2-element Array{Int64,1}:
 3
 4

julia> bar(1,2,x...)
(1, 2, (3, 4))
```

```
julia> x = [1,2,3,4]
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> bar(x...)
(1, 2, (3, 4))
```

También, la función cuyos argumentos son divididos no tiene por qué ser una función *vararg* (aunque frecuentemente lo sea):

```
julia> baz(a,b) = a + b;

julia> args = [1,2]
2-element Array{Int64,1}:
 1
 2

julia> baz(args...)
3

julia> args = [1,2,3]
3-element Array{Int64,1}:
 1
 2
 3

julia> baz(args...)
ERROR: MethodError: no method matching baz(::Int64, ::Int64, ::Int64)
Closest candidates are:
  baz(::Any, ::Any) at none:1
```

Como puede comprobarse, si el número de elementos que se van a sacar del contenedor es inapropiado para pasar a la función como argumentos, se generará un error, igual que si hubiéramos realizado la llamada a función con un número de argumentos inapropiado.

11.8 Argumentos Opcionales

En muchos casos, los argumentos de función tienen valores por defecto sensibles y, por tanto, puede no ser necesario que se pasen explícitamente en cada llamada. Por ejemplo, la función de librería `parse(T, num, base)` interpreta una cadena como un número en cierta base. El argumento `base` tiene un valor por defecto de `10`. Este comportamiento puede expresarse de forma concisa como:

```
function parse(T, num, base=10)
    ###
end
```

Con esta definición, la función puede ser llamada con dos o tres argumentos y, cuando no se pase el tercer argumento, la función asignara el valor por defecto de `10` al parámetro `base`.

```
julia> parse(Int,"12",10)
12

julia> parse(Int,"12",3)
5

julia> parse(Int,"12")
12
```

Los argumentos opcionales son una sintaxis conveniente para escribir múltiples definiciones de métodos con diferentes números de argumentos (ver Nota sobre Argumentos opcionales y *keyword*).

11.9 Argumentos *keyword*

Algunas funciones necesitan un número de argumentos grande o tienen un gran número de comportamientos. Recordar como llamar a tales funciones puede ser difícil. Los argumentos *keyword* pueden hacer que estas interfaces complejas sean más fáciles de usar y extender permitiendo que los argumentos sean identificados por su nombre en lugar del por su posición.

Por ejemplo, considere una función `plot` que traza una línea. Esta función puede tener muchas opciones para controlar el estilo de línea, su ancho, su color, etc. Si la función aceptara argumentos *keyword*, una posible llamada al método seria `plot(x,y, width=2)`, donde hemos elegido especificar sólo el ancho de línea. Nótese que esto sirve a dos propósitos: La llamada es más sencillo leer, ya que podemos etiquetar un argumento con su significado. También se vuelve posible pasar cualquier subconjunto de un gran número de argumentos, en cualquier orden.

Las funciones con argumentos *keyword* se definen usando un punto y coma en la signatura:

```
function plot(x, y; style="solid", width=1, color="black")
    ###
end
```

Cuando la función es invocada, el punto y coma es opcional: uno puede hacer la llamada como `plot(x, y, width=2)`o como `plot(x, y; width=2)`, aunque el primero es más común. Se requiere un punto y coma explícito sólo en el caso de pasar *vargars* o palabras clave calculadas como se describe abajo.

Los valores por defecto de los argumentos *keyword* son evaluados sólo cuando sea necesario (cuando no se pasa el correspondiente argumento *keyword*) y en orden izquierda a derecha. Por tanto, las expresiones por defecto pueden referirse a argumentos *keyword* previos.

Los tipos de argumentos *keyword* pueden hacerse explícitos de la siguiente forma:

```
function f(;x::Int64=1)
    ###
end
```

Los argumentos *keyword* extra pueden ser recolectados usando `...` como en las funciones *vararg*:

```
function f(x; y=0, kwargs...)
    ###
end
```

Dentro de `f`, `kwargs` será una colección de tuplas `(clave,valor)`, donde cada `clave` es un símbolo. Tales colecciones pueden ser pasadas como argumentos *keyword* usando un punto y coma en la llamada. Por ejemplo: `f(x, z=1; kwargs...)`. Los diccionarios pueden ser también usados para este propósito.

Uno puede también pasar tuplas `(clave, valor)` o cualquier expresión iterable (tal como un par `=>`) que puede ser asignado a una tupla, explícitamente después de un punto y coma. Por ejemplo, `plot(x, y; (:width,2))` y `plot(x, y; :width => 2)` son equivalentes a `plot(x, y, width=2)`. Esto es útil en situaciones donde el nombre de la palabra clave se calcula en tiempo de ejecución.

La naturaleza de los argumentos *keyword* le hace posible especificar el mismo argumento más de una vez. Por ejemplo, en la llamada `plot(x, y; options..., width=2)` es posible que la estructura `options` contenga también un valor para `width`. En tal caso la ocurrencia más a la derecha toma precedencia; en este ejemplo `width` tendrá el valor 2.

11.10 Ámbito de evaluación de Valores por defecto

Los argumentos opcionales y *keyword* difieren ligeramente en cómo sus valores son evaluados. Cuando se evalúan expresiones por defecto con valores opcionales, sólo están en el ámbito lo valores *previos*. En contraste, cuando se evalúan las expresiones por defecto con argumentos *keyword*, *todos* los argumentos están en el ámbito. Por ejemplo, dada esta definición:

```
function f(x, a=b, b=1)
    ###
end
```

la b en a=b se refiere a la b de un ámbito más externo, no el siguiente argumento b. Sin embargo, si a y b fueran argumentos *keyword* en lugar de opcionales, el b en a=b se referiría al argumento posterior b (ocultando a cualquier b de un ámbito ms externo), lo que resultaría en un error de variable indefinida (ya que las expresiones por defecto son evaluadas de izquierda a derecha, y b no ha sido aún asignada).

11.11 Sintaxis Bloque Do para Argumentos Function

Pasar funciones como argumentos a otras funciones es una técnica muy potente, pero su sintaxis no es siempre conveniente. Estas llamadas son especialmente incómodas de escribir cuando la función argumento necesita varias líneas. Por ejemplo, consideremos llamar a `map()` sobre una función con varios casos:

```
map(x->begin
           if x < 0 && iseven(x)
               return 0
           elseif x == 0
               return 1
           else
               return x
           end
       end,
    [A, B, C])
```

Julia proporciona la palabra reservada do para reescribir este código de forma más clara:

```
map([A, B, C]) do x
    if x < 0 && iseven(x)
        return 0
```

```
    elseif x == 0
        return 1
    else
        return x
    end
end
```

La sintaxis `do x` crea una función anónima con argumento `x` y la pasa como primer argumento a `map()`. Similarmente, `do a,b` crearía una función anónima de dos argumentos, y un `do` solo sería una función anónima de la forma `() -> ...`.

Cómo se inicializan estos argumentos depende de la función más externa; aquí `map()` fijará secuencialmente `x` a `A`, `B`, `C` llamando a la función anónima sobre cada uno de ellos, tal y como pasa en la sintaxis `map(func, [A,B,C])`.

Esta sintaxis hace más fácil usar funciones para extender el lenguaje de forma efectiva, ya que las llamadas tiene el aspecto de códigos de bloque normales. Hay muchos usos posibles diferentes al de `map()`, tal como la gestión del estado del sistema. Por ejemplo, hay una versión de `open()` que ejecuta código asegurando que el fichero abierto es cerrado eventualmente:

```
open("outfile", "w") do io
    write(io, data)
end
```

Esto se consigue mediante la siguiente definición:

```
function open(f::Function, args...)
    io = open(args...)
    try
        f(io)
    finally
        close(io)
    end
end
```

Aquí, `open()` primero abre el fichero para escritura y luego pasa el flujo de salida resultante a la función anónima que se define en el bloque `do...end`. Después de que la función exista, `open()` asegurará que el flujo ha sido cerrado apropiadamente, sin preocuparse de si la función salió normalmente o lanzó una excepción (la construcción `try/finally` será descrita en Control de Flujo.)

Con la sintaxis de bloque do se ayuda a chequear la documentación o implementaciones para saber cómo se inicializan los argumentos de la función de usuario.

11.12 Sintaxis Punto para funciones Vectorizadas

En los lenguajes de computación técnicos es común tener versiones "vectorizadas" de funciones, las cuales aplican una función dada `f(x)` a cada elemento de un array `A` para producir un nuevo array vía `f(A)`. Esta clase de sintaxis es conveniente para procesamiento de datos, pero en otros lenguajes la vectorización es también requerida en aras de mejorar el rendimiento: si los bucles son lentos, la versión "vectorizada" de una función podría llamar al código de librería rápido en un lenguaje de bajo nivel. En Julia, las funciones actualizadas *no son requeridas por motivos de vencimiento*; de hecho, suele ser beneficioso que el usuario escriba sus propios bucles (ver Consejos de rendimiento), a veces incluso conveniente. Por tanto *cualquier* función Julia `f` puede ser aplicada elemento a elemento a cualquier array (u otra colección) con la sintaxis `f.(A)`. Por ejemplo `sin` puede ser aplicado a todos los elementos del vector `A` de esta forma:

```
julia> A = [1.0, 2.0, 3.0]
3-element Array{Float64,1}:
 1.0
 2.0
 3.0

julia> sin.(A)
3-element Array{Float64,1}:
 0.841471
 0.909297
 0.14112
```

Por supuesto, uno puede omitir el punto si escribe un método especial para vectores de `f` por ejemplo, vía `f(A::AbstractArray) = map(f, A)` y esto es tan eficiente como `f.(A)`. Pero este enfoque necesitaría que decidas a priori qué funciones quieres vectorizar.

Más generalmente, `f.(args...)` es de hecho equivalente a `broadcast(f, args...)`, que te permite operar sobre múltiples arrays (incluso de formas distintas) o una mezcla de arrays y escalares (ver Broadcasting). Por ejemplo, si tenemos `f(x,y) = 3x + 4y`, entonces `f.(pi,A)` devolverá un nuevo array consistente en `f(pi,a)` para cada a en `A`, y `f.(vector1,vector2)` devolverá un nuevo vector que consiste en `f(vector1[i],vector2[i])` para cada índice `i` (lanzando una excepción si los vectores tienen diferente longitud).

```
julia> f(x,y) = 3x + 4y;

julia> A = [1.0, 2.0, 3.0];

julia> B = [4.0, 5.0, 6.0];

julia> f.(pi, A)
3-element Array{Float64,1}:
 13.4248
 17.4248
 21.4248

julia> f.(A, B)
3-element Array{Float64,1}:
 19.0
 26.0
 33.0
```

Además, las llamadas anidadas `f.(args...)` se funden en un solo `broadcast`. Por ejemplo `sin.(cos.(X))` es equivalent a `broadcast(x->sin(cos(x)), X)`, lo cuál es similar a `[sin(cos(x)) for x in X]`. Hay un solo bucle sobre `X`, y se asigna un solo array para el resultado. En contraste, `sin(cos(X))` en un lenguaje vectorizado típio asignaría primero un array temporal `tmp = cos(X)` y luego calcularía `sin(tmp)` en un bucle separado, asignando un segundo array. Esta fusión de bucles no es una optimización del compilador que puede ocurrir o no, sino que es una *garantía sintáctica* cuando se encuentran llamadas `f.(array...)` anidadas. Técnicamente, la fusión se para en cuanto se encuentr una función sin punto, por ejemplo, en `sin.(srt(cos.(X)))` los bucles de `sin` y `cos` no pueden mezclarse debido a la intervención de la función `sort`.

Finalmente, la eficiencia máxima suele conseguirse cuando el array de salida de una operación vectorizada es *preasignado*, por lo que las llamadas repetidas no asignarán nuevos arrays una y otra vez para los resultados (ver Preasignando salidas). Una sintaxis conveniente para esto es `X .= ...` que es equivalente a `broadcast!(identity, X, ...)` excepto que, como antes, el bucle `broadcast!` es fusionado con cualquier llamada con punto anidada. Por ejemplo, `X .= sin.(Y)` es equivalente a `broadcast!(sin, X, Y)`, sobreescribiendo `X` con `sin.(Y)` en su lugar.

Si el miembro izquierdo de la expresión es una expresión de indexación de un array, como `X[2:end] .= sin.(Y)` entonces ella se traduce a `broadcast!` sobre una vista, por ejemplo `broadcast!(sin, view(X, 2:endof(X)), Y)`.

Como añadir puntos a muchas operaciones y llamadas a función puede resultar tedioso y conducir a código difícil de leer, se proporciona la macro `@.` para convertir cada llamada a función, operación y asignación en una expresion en su versión "con puntos".

```
julia> Y = [1.0, 2.0, 3.0, 4.0];

julia> X = similar(Y); # pre-allocate output array

julia> @. X = sin(cos(Y)) # equivalent to X .= sin.(cos.(Y))
4-element Array{Float64,1}:
  0.514395
 -0.404239
 -0.836022
 -0.608083
```

Los operadores binarios (o unarios) como `.+` se manejan con el mismo mecanismo: son equivalentes a llamadas retransmitidas (broadcast) y son fundidas con otras llamadas que tiene puntos. `X .+= Y` etcetera es equivalente a `X .= X .+ Y` y dan como resultado una asignación fusionada. Ver también dot operators.

11.13 Otras Lecturas

Deberíamos mencionar que esto está lejos de ser una visión completa de las definiciones de función. Julia tiene un sistema de tipos sofisticado y permite despacho múltiple sobre los tipos de argumento. Ninguno de los ejemplos dados aquí proporciona anotaciones de tipo sobre sus argmentos, lo que significa que son aplicables a cualquier tipo de argumento. El sistema de tipos es descrito en Tipos definir una función en términos de métodos elegidos mediante despacho múltiple sobre los tpos de argumento en tiempo de ejecucoión se describe en el capítulo Methods.

Chapter 12

Control Flow

Julia proporciona una variedad de construcciones para control de flujo:

- Expresiones Compuestas: `begin` and `(;)`.
- Evaluación Condicional: `if-elseif-else` and `?:` (ternary operator).
- Evaluación en Cortocircuito: `&&`, `||` and chained comparisons.
- Evaluación Repetida: Bucles: `while` and `for`.
- Manejo de Excepciones: `try-catch`, `error()` and `throw()`.
- Tareas (también denominadas Coroutinas): `yieldto()`.

Los cinco primeros mecanismos de control de flujo son estándar en los lenguajes de programación de alto nivel. Las tareas no son un mecanismo tan estándar: ellas proporcionan control de flujo no local, haciendo posible conmutar entre cálculos suspendidos temporalmente. Esta es una construcción potente: tanto el manejo de excepciones como la multitarea cooperativa se implementan en Julia usando tareas. La programación diaria no suele requerir el uso de tareas, pero ciertos problemas se resuelve de forma mucho más sencilla usando este mecanismo.

12.1 Expresiones Compuestas

Algunas veces es conveniente tener una sola expresión que lleva nueve varias subexpresiones en orden, devolviendo el valor de la última subexpresión como su valor. Hay dos construcciones en Julia que llevan a cabo este trabajo: los bloques `begin` y las cadenas `;` El valor de ambas expresiones compuestas es el de la última subexpresión. He aquí un ejemplo del bloque `begin`:

```
julia> z = begin
           x = 1
           y = 2
           x + y
       end
3
```

Como estas expresiones son bastante pequeñas, podrían ponerse con facilidad en una sola línea, que es donde la sintaxis encadenada (;) es más útil:

```
julia> z = (x = 1; y = 2; x + y)
3
```

Esta sintaxis es particularmente útil con la definición de función de una línea que introdujimos en Funciones. Aunque es típico, no hay obligación de que los bloques `begin` sean multilínea o de que las cadenas de punto y coma (`;`) tengan una única línea.

```
julia> begin x = 1; y = 2; x + y end
3

julia> (x = 1;

        y = 2;

        x + y)
3
```

12.2 Evaluación Condicional

La evaluación condicional permite que porciones de código sean evaluadas o no evaluadas dependiendo del valor de una expresión booleana. Esta es la anatomía de la estructura de `if-elseif-else`:

```
if x < y
    println("x is less than y")
elseif x > y
    println("x is greater than y")
else
    println("x is equal to y")
end
```

En el ejemplo anterior, si la condición x<y es verdadera, entonces se evaluará el bloque correspondiente. En caso contrario se evaluará la expresión x>y, y si esta es verdadera, se ejecutará el bloque correspondiente. Si la expresión también es falsa, se ejecutaría el bloque correspondiente al `else`. Veámoslo en acción:

```
julia> function test(x, y)

          if x < y

              println("x is less than y")

          elseif x > y

              println("x is greater than y")

          else

              println("x is equal to y")

          end
```

```
      end
test (generic function with 1 method)

julia> test(1, 2)
x is less than y

julia> test(2, 1)
x is greater than y

julia> test(1, 1)
x is equal to y
```

Los bloques `elsif` y `else` son opcionales, y además pueden usarse tantos `elsif` como se deseen. Las expresiones condicionales del `if-elsif-else` serán evaluadas hasta que una de ellas se evalúe a `true`, después de lo cuál se evaluará el blqoue asociado, y ya no se evaluarán más expresiones condicionales.

Los bloques `if` son "permeables", es decir, no introducen un ámbito local. Eso significa que las variables que se definen dentro del bloque serán visibles fuera del mismo. Por tanto, podríamos haber definido la relación `test` de antes como...

```
julia> function test(x,y)

           if x < y

               relation = "less than"

           elseif x == y

               relation = "equal to"

           else

               relation = "greater than"

           end

           println("x is ", relation, " y.")

       end
test (generic function with 1 method)

julia> test(2, 1)
x is greater than y.
```

La variable `relation` se ha declarado dentro del bloque `if`, pero se usa fuera. Sin embargo, cuando se hace uso de este tipo de variables, hay que asegurarse de que todos los caminos de código definen un valor para la variable. La siguiente función no lo tiene en cuenta y genera un error en tiempo de ejecución.

```
julia> function test(x,y)

           if x < y

               relation = "less than"
```

```
            elseif x == y

                relation = "equal to"

            end

            println("x is ", relation, " y.")

        end
test (generic function with 1 method)

julia> test(1,2)
x is less than y.

julia> test(2,1)
ERROR: UndefVarError: relation not defined
Stacktrace:
 [1] test(::Int64, ::Int64) at ./none:7
```

Los bloques `if` también devuelven un valor, lo que puede no parecer intuitivo para quienes proceden de otros lenguajes de programación no funcionales. Este valor no es más que el devuelto por la última instrucción en la rama que fue elegida. Por tanto:

```
julia> x = 3
3

julia> if x > 0

            "positive!"

        else

            "negative..."

        end
"positive!"
```

Nótese que las instrucciones condicionales muy cortas (de una línea) se suelen expresar en Julia mediante evaluación en cortocircuito, como se verá en la siguiente sección.

A diferencia de C, MATLAB, Perl, Python y Ruby (pero como en Java y en otros lenguajes tipados, más estrictos) en Julia se produce un error si el valor de una expresión condicional es algo que no sea `true` o `false`.

```
julia> if 1

            println("true")

        end
ERROR: TypeError: non-boolean (Int64) used in boolean context
```

Este error indica que el condicional era de un tipo incorrecto: `Int64` en lugar del requerido `Bool`.

El llamado *operador ternario* (`?`) está muy relacionado con la sintaxis de `if-elsif-else`, pero se usa donde hay que hacer una elección condicional entre expresiones sencillas, a diferencia de la ejecución condicional de grandes bloques

de código. Su nombre se debe a que es el único operador que toma tres operandos en la mayoría de los lenguajes de programación:

```
a ? b : c
```

La expresión a delante del `?` es una expresión condicional, y la operación ternaria evalúa la expresión b (la que está delante del símbolo `:`) si la condición a es `true` o la expresión c si la condición a es `false`. Nótese que los espacios alrededor de `? y :` son obligatorios: una expresión como `a?b:c` no es una expresin ternaria válida (aunque se pueden utilizar saltos de línea entre los símbolos `?` y `:`).

La forma más fácil de comprender este comportamiento es ver un ejemplo. En el ejemplo anterior, la llamada a `println` es compartida por las tres ramas: la única elección real es qué cadena literal imprimir. Esto podría haberse escrito de forma más concisa usando el operador ternario. En aras de la claridad, intentemos primero la versión con dos caminos:

```
julia> x = 1; y = 2;

julia> println(x < y ? "less than" : "not less than")
less than

julia> x = 1; y = 0;

julia> println(x < y ? "less than" : "not less than")
not less than
```

En los ejemplos anteriores, si `x < y` es `true` se devolverá la cadena `"less than"` y, en caso contrario, se devolverá la cadena `"not less than"`. El ejemplo original, que tiene tres opciones, requeriría el uso encadenado del operador `?`:

```
julia> test(x, y) = println(x < y ? "x is less than y"    :

                            x > y ? "x is greater than y" : "x is equal to y")
test (generic function with 1 method)

julia> test(1, 2)
x is less than y

julia> test(2, 1)
x is greater than y

julia> test(1, 1)
x is equal to y
```

Para facilitar el encadenamiento, el operador `?` asocia de derecha a izquierda.

Es también significativo que, como en la construcción `if-elsif-else` las expresiones anterior y posterior al símbolo `:` sólo se evalúan si la expresión condicional es evaluada a `true` o `false`, respectivamente.

```
julia> v(x) = (println(x); x)
v (generic function with 1 method)

julia> 1 < 2 ? v("yes") : v("no")
yes
"yes"
```

```
julia> 1 > 2 ? v("yes") : v("no")
no
"no"
```

12.3 Evaluación en Cortocircuito

La evaluación en cortocircuito es bastante similar a la evaluación condicional. Este comportamiento aparece en la mayoría de los lenguajes de programación imperativos que tiene los operadores booleanos && y ||. En una serie de expresiones booleanas conectadas por estos operadores, sólo se evalúa el número mínimo de expresiones necesarios para determinar el valor booleano final de la cadena completa. Explícitamente, esto significa que:

- En la expresión a && b la subexpresión b sólo se evalúa si la subexpresión a es evaluada a `true`.
- En la expresión a || b la subexpresión b sólo se evalúa si la subexpresión a es evaluada a `false`.

El razonamiento es que a && b debe ser `false` si a is `false`, independientemente del valor de b y, análogamente, el valor de a || b debe ser cierto si a es `true`, independientemente del valor de b. Tanto && como || asocian a la derecha, pero && tiene mayore precedencia que ||. Es fácil experimentar con este comportamiento:

```
julia> t(x) = (println(x); true)
t (generic function with 1 method)

julia> f(x) = (println(x); false)
f (generic function with 1 method)

julia> t(1) && t(2)
1
2
true

julia> t(1) && f(2)
1
2
false

julia> f(1) && t(2)
1
false

julia> f(1) && f(2)
1
false

julia> t(1) || t(2)
1
true

julia> t(1) || f(2)
1
true

julia> f(1) || t(2)
1
2
```

```
true

julia> f(1) || f(2)
1
2
false
```

Se puede experimentar de forma parecida con la asociatividad y la precedencia de varias combinaciones de operadores && y ||.

Este comportamiento se utiliza en Julia con frecuencia para formar una alternativa a instrucciones `if` muy cortas. En lugar de usar la construcción `if <cond> && <instrucción>` uno puede escribir `<cond> && <instrucción>` que puede leerse como `<cond>` y entonces `<instrucción>`. de forma similar, uno puede escribir `<cond> || <instrucción>`, que se leería como `<cond>` o sino `<instrucción>` en lugar de `if !<cond> || <instrucción>`.

Por ejemplo, una rutina recursiva para obtener un factorial podría ser definida como:

```
julia> function fact(n::Int)

           n >= 0 || error("n must be non-negative")

           n == 0 && return 1

           n * fact(n-1)

       end
fact (generic function with 1 method)

julia> fact(5)
120

julia> fact(0)
1

julia> fact(-1)
ERROR: n must be non-negative
Stacktrace:
 [1] fact(::Int64) at ./none:2
```

Las operaciones booleanas sin cortocircuito podrían llevarse a cabo con los operadores a nivel de bit introducidos en la sección Operaciones Matemáticas y Funciones Elementales: & y |. Estas son funciones normales, que suportan la sintaxis infija de los operadores, pero que siempre evalúan sus argumentos:

```
julia> f(1) & t(2)
1
2
false

julia> t(1) | t(2)
1
2
true
```

Como en el caso de las expresiones condicionales usadas en `if`, `elsif` o el operador ternario ?, los operandos de && y de || deben ser valores booleanos. Usar un valor no booleanos en cualquier lugar distinto de la última entrada en una cadena condicional producirá un error.

```
julia> 1 && true
ERROR: TypeError: non-boolean (Int64) used in boolean context
```

Por otra parte, cualquier tipo de expresión puede ser usada al final de una cadena condicional. Ella será evaluada y devuelta dependiendo de los condicionales precedentes:

```
julia> true && (x = (1, 2, 3))
(1, 2, 3)

julia> false && (x = (1, 2, 3))
false
```

12.4 Evaluación Repetida: Bucles

Hay dos construcciones que realizan la evaluación repetida de expresiones: el bucle `while` y el bucle `for`. He aquí un ejemplo del bucle `while`:

```
julia> i = 1;

julia> while i <= 5
           println(i)
           i += 1
       end
1
2
3
4
5
```

El bucle `while` evalúa la expresión condicional (en el ejemplo `i<=5`) y, mientras que esta se evalúe a `true`, sigue evaluando el cuerpo del bucle `while`. Si la expresión se evalúa a `false` la primera vez en que se alcanza el bucle, su cuerpo nunca será evaluado.

El bucle `for` facilita la repetición. Dado que contar arriba y abajo (como en el ejemplo anterior del bucle `while`) es tan común, podemos expresar esto de una forma muy concisa con un bucle `for`:

```
julia> for i = 1:5
           println(i)
       end
1
2
3
4
5
```

En el ejemplo anterior, `1:5` es un objeto `Range` que representa una secuencia de números. El bucle `for` itera sobre estos valores, asignando cada uno de ellos por turno a la variable `i`. Una distinción importante ente esta construcción

(`for`) y la construcción anterior (`while`) es el ámbito durante el cuál la variable es visible. Si la variable `i` no ha sido introducida en otro ámbito, el bucle `for` la verá sólo en su interior y no posteriormente. Para demostrar esto necesitaremos una nueva sesión interactiva o usar un nombre de variable distinto:

```
julia> for j = 1:5

           println(j)

       end
1
2
3
4
5

julia> j
ERROR: UndefVarError: j not defined
```

Ver Ámbito de Variables para una explicación detallada de los ámbitos de las variables y cómo funcionan en Julia.

En general, la construcción `for` puede iterar sobre cualquier contenedor. En estos casos, la palara clave alternativa (pero totalmente equivalente `in` o `∈` es usada en lugar de `=`, dado que hace que la lectura del código sea más clara.

```
julia> for i in [1,4,0]

           println(i)

       end
1
4
0

julia> for s ∈ ["foo","bar","baz"]

           println(s)

       end
foo
bar
baz
```

En otras secciones del manual se introducirán y discutirán varios tipos de contenedores iterables (ver, por ejemplo, Arrays Multi-dimensionales).

Algunas veces es conveniente terminar la repetición de un `while` antes de chequear la condición de test o partar de iterar en un bucle `for` antes de que se alcance el final del objeto iterable. Esto puede conseguirse usando la palabra clave `break`:

```
julia> i = 1;

julia> while true

           println(i)
```

```
            if i >= 5
                break
            end
            i += 1
        end
1
2
3
4
5

julia> for i = 1:1000
            println(i)
            if i >= 5
                break
            end
        end
1
2
3
4
5
```

Si no existiera la palabra clave `break`, el bucle `while` anterior nunca finalizará por si mismo, y el bucle `for` iteraría hasta 10000. Si hacemos uso de la instrucción `break` conseguiremos abandonar el bucle mucho antes.

En otras circunstancias es útil ser capaz de detener una iteración y moverse a la siguiente de forma inmediata. Para ello, se utiliza la palabra clave `continue`:

```
julia> for i = 1:10
            if i % 3 != 0
                continue
            end
            println(i)
        end
3
6
9
```

Este es un ejemplo un tanto artificial, ya que podríamos obtener el mismo comportamiento de forma mucho más clara negando las condiciones y colocando la llamada a `println` dentro del bloque `if`. En usos más reales hay más

código que evaluar después del `continue`, y con frecuencia hay muchos puntos desde los que uno puede llamar a esta instrucción.

Podemos anidar múltiples bucles for en un solo bucle externo, formando el producto cartesiano de sus iterables:

```
julia> for i = 1:2, j = 3:4

           println((i, j))

       end
(1, 3)
(1, 4)
(2, 3)
(2, 4)
```

Una instrucción `break` dentro de tal bucle sale del anidamiento de bucles completo, no sólo del más interior.

12.5 Manejo de Excepciones

Cuando tiene lugar una condición inesperada, una funció puede ser incapaz de devolver un valor razonable al código que la invoca. En tales casos, puede ser mejor para la condición excepcional terminar el programa, imprimiendo un mensaje de error diagnóstico, o si el programador ha proporcionado código para manejar tales circunstancias excepcionales, permitiendo que el código tome la acción apropiada.

Excepciones predefinidas

Las excepciones se lanzan cuando ocurre una condición inesperada. En la siguiente tabla se muestran todas la excepciones predefinidas, que interrumplen todas el flujo de control normal.

Por ejemplo, la función `sqrt()` lanza un `DomainError` si se aplica sobre un valor real negativo:

```
julia> sqrt(-1)
ERROR: DomainError:
sqrt will only return a complex result if called with a complex argument. Try sqrt(complex(x)).
Stacktrace:
 [1] sqrt(::Int64) at ./math.jl:434
```

Uno puede definir sus propias excepciones de la siguiente manera:

```
julia> struct MyCustomException <: Exception end
```

La función `throw()`

Las excepciones pueden crearse explícitamente con `throw()`. Por ejemplo, una función definida sólo para número no negativos podría escribirse para que lanzara un `DomainError` si el argumento es negativo:

```
julia> f(x) = x>=0 ? exp(-x) : throw(DomainError())
f (generic function with 1 method)

julia> f(1)
0.36787944117144233

julia> f(-1)
```

Exception
ArgumentError
BoundsError
CompositeException
DivideError
DomainError
EOFError
ErrorException
InexactError
InitError
InterruptException
InvalidStateException
KeyError
LoadError
OutOfMemoryError
ReadOnlyMemoryError
RemoteException
MethodError
OverflowError
ParseError
SystemError
TypeError
UndefRefError
UndefVarError
UnicodeError

```
ERROR: DomainError:
Stacktrace:
 [1] f(::Int64) at ./none:1
```

Notese que `DomainError` sin paréntesis no es una excepción, sino un tipo de excepción. Ella necesita ser invocada para obtener un objeto `Exception`:

```
julia> typeof(DomainError()) <: Exception
true

julia> typeof(DomainError) <: Exception
false
```

Adicionalmente, algunos tipos de excepción toman uno o más argumentos que se utilizan para reportar errores.

```
julia> throw(UndefVarError(:x))
ERROR: UndefVarError: x not defined
```

Este mecanismo puede ser fácilmente implementado mediante los tipos de excepción personalizados que sigan la forma en que se escribe `UndefVarError`:

```
julia> struct MyUndefVarError <: Exception

```

```
            var::Symbol

        end

julia> Base.showerror(io::IO, e::MyUndefVarError) = print(io, e.var, " not defined")
```

Note

Cuando se escribe un mensaje de error, es preferible que la primera palabra sea minúscula. Por ejemplo,

`size(A) == size(B) || throw(DimensionMismatch("size of A not equal to size of B"))`

es preferible a

`` `size(A) == size(B) || throw(DimensionMismatch("Size of A not equal to size of B"))`. ``

Sin embargo, algunas veces tiene sentido mantener la primera letra en mayúscula, por ejemplo, si un argumento a función es una letra mayúscula:

`` `size(A,1) == size(B,2) || throw(DimensionMismatch("A has first dimension..."))`. ``

Errores

La función `error()` se usa para producir una `ErrorException` que interrumpe el flujo de control normal.

Supóngase que deseamos detener la ejecución inmediatamente si se toma la raíz cuadrad de un número negativo. Para hacer esto, podemos definir una versión ”quisquillosa” de la función `sqrt()` que lanza un error si recibe un número negativo:

```
julia> fussy_sqrt(x) = x >= 0 ? sqrt(x) : error("negative x not allowed")
fussy_sqrt (generic function with 1 method)

julia> fussy_sqrt(2)
1.4142135623730951

julia> fussy_sqrt(-1)
ERROR: negative x not allowed
Stacktrace:
 [1] fussy_sqrt(::Int64) at ./none:1
```

Si `fussy_sqrt()` es invocada con un valor negativo desde otra función, en lugar de intentar continuar la ejecución de la función que la invocó, retorna inmediatamente, mostrando el mensaje de error en la sesión interactiva:

```
julia> function verbose_fussy_sqrt(x)
           println("before fussy_sqrt")
           r = fussy_sqrt(x)
           println("after fussy_sqrt")
           return r
       end
verbose_fussy_sqrt (generic function with 1 method)

julia> verbose_fussy_sqrt(2)
before fussy_sqrt
after fussy_sqrt
1.4142135623730951

julia> verbose_fussy_sqrt(-1)
```

```
before fussy_sqrt
ERROR: negative x not allowed
Stacktrace:
 [1] fussy_sqrt at ./none:1 [inlined]
 [2] verbose_fussy_sqrt(::Int64) at ./none:3
```

Mensajes de aviso y de información

Julia también proporciona otras funciones que escriben mensajes a la salida de error estándar, pero no lanzan ninguna `Exception` y, por tanto, no interrumpen la ejecución:

```
julia> info("Hi"); 1+1
INFO: Hi
2

julia> warn("Hi"); 1+1
WARNING: Hi
2

julia> error("Hi"); 1+1
ERROR: Hi
Stacktrace:
 [1] error(::String) at ./error.jl:21
```

La instrucción `try/catch`

La intrucción `try / catch` permite comprobar a aparición de excepciones. Por ejemplo, puede escribirse una función personalizada para calcular la raíz cuadrada que invoque automáticamente al método de cálculo de la raíz de valores reales y/o complejos en función de la excepción:

```
julia> f(x) = try

           sqrt(x)

       catch

           sqrt(complex(x, 0))

       end
f (generic function with 1 method)

julia> f(1)
1.0

julia> f(-1)
0.0 + 1.0im
```

Es importante notar que en el código real que computa esta función, uno podría comparar `x` con vero en lugar de atrapar la excepción. De hcho, la opción de la excepción ees mucho más lenta de comparar y ramificar.

Las instrucciones `try / catch` también permiten salvar la excepción en una variable. En este ejemplo artificial, se calcula la raíz cuadrada del segundo elemento de `x`. Si `x` es indexable, en caso contrario asume que `x` es un número real y devuelve su raíz cuadrada:

```
julia> sqrt_second(x) = try
           sqrt(x[2])
       catch y
           if isa(y, DomainError)
               sqrt(complex(x[2], 0))
           elseif isa(y, BoundsError)
               sqrt(x)
           end
       end
sqrt_second (generic function with 1 method)

julia> sqrt_second([1 4])
2.0

julia> sqrt_second([1 -4])
0.0 + 2.0im

julia> sqrt_second(9)
3.0

julia> sqrt_second(-9)
ERROR: DomainError:
Stacktrace:
 [1] sqrt_second(::Int64) at ./none:7
```

Note que el símbolo que sigue al `catch` siempre será interpretado como el nombre para la excepción, por lo que hay que tener cuidado cuando se escriben expresiones `try / catch` en una sola línea. El siguiente código no funcionará para devolver el valor de `x` en caso de error:

```
try bad() catch x end
```

En su lugar, es mejor usar un punto u coma o insertar un salto de línea después del `catch`:

```
try bad() catch; x end

try bad()
catch
    x
end
```

La cláusula `catch` no es estrictamente necesaria; cuando se omite el valor de retorno por defecto es `nothing`

```
julia> try error() end # Returns nothing
```

La potencia de la construcción `try / catch` estriba en la capacidad de desplegar inmediatamente un cálculo profundamente anidado de hasta un nivel mucho más elevado en la pila de llamadas a función. Hay situacionces donde no ha ocurrido error, pero la capacidad de desplegar la pila y pasar un valor a un nivel superior es deseable. Julia proporciona las funciones `rethrow()`, `backtrace()` and `catch_backtrace()` para un manejo de errores más avanzado.

Cláusulas `finally`

En código que realiza cambios de estado o usa recursos como ficheros, hay típicamente un trabajo de limpieza (tal como cerrar ficheros) que necesita ser realizado cuando el código finaliza. Las excepciones complican potencialmente esta tarea, ya que pueden causar que un bloque de código salga antes de alcanzar su final normal. La palabra clave `finally` proporciona una forma de ejecutar código cuando existe un blqoue de código dado, sin preocuparse de cómo salga.

Por ejemplo, aquí podemos garantizar que un fichero abierto se cierra:

```
f = open("file")
try
    # operate on file f
finally
    close(f)
end
```

Cuando el contro deja el bloque `try` (por ejemplo, debido a un `return`, o finalizando normalmente) se ejecutará `close()`. Si el bloque `try` saliera debido a una excepción, la excepción continuará propagándose. Un bloque `catch` puede ser combinada con `try` y `finally` también. En este caso el bloque `finally` ejecutará después de que `catch` haya manejado el error.

12.6 Tareas (aka Corutinas)

Las tareas son una característica de control de flujo que permite que los cálculos sean suspendidos y continuados de una forma flexible. Esta característica es llamada algunas veces con otros nombres, tales como corrutinas simétricas, hilos de peso ligero, multitarea cooperativa o continuaciones de un disparo.

Cuando una pieza de trabajo de cómputo (en la práctica, ejecutar una función particular) es designada como tarea (`Task`), se hace posible interrumplirla intercambiándola por otra tarea. La tarea original puede ser continuada después, en el punto en que se encontraba justo cuando fue detenida. A primera vista, esto puede parecer similar a una llamada a función. Sin embargo, hay dos diferencias clave. Primero, conmutar tareas no usa ningún espacio, por lo que puede tener lugar cualquier número de intercambios de tarea sin que se consuma la pila de llamadas. Segundo, conmutar entre tareas puede ocurrir en cualquier orden, a diferencia de lo que pasa en las llamadas a función, donde la función invocada debe terminar la ejecución antes de que el control retorne a la función que la llamó.

Esta clase de flujo de control puede hacer mucho más fácil resolver ciertos problemas. En algunos problemas, las distintas piezas de trabajo requerido no están relacionadas naturalmente mediante llamadas a función: no hay un obvio llamador o llamado entre los trabajos que necesitan ser realizados. Un ejemplo es el problema del productor-consumidor, donde un procedimiento complejo está generando valores u otro procedimiento complejo los está consumiendo. El consumidor no puede simplemente llamar a la función productora para obtener un valor, debido a que el productor puede tener más valores que generar y, por tanto, podría no estar listo todavía para retornar. Con las tareas, el productor y el consumidor pueden ambos ejecutarse mientas que lo necesiten, pasando valores adelante y detrás cuando sea necesario.

Julia proporciona un mecanismo denominado ”canal” (`Channel` para resolver este problema. Un canal es una cola FIFO (primero en entrar, primero en salir) que puede tener múltiples tareas leyendo de y escribiendo en ella.

Definamos una tarea productor, que produce valores a través de una llamada `put!`. Para consumir valores, necesitamos planificar un productor que ejecute una nueva tarea. Para ejecutar una tarea asociada a un canal utilizaremos un

constructor especial `Channel` que recibe como argumento una función de un argumento. Podemos entonce tomar valores repetidamente del objeto canar mediante llamadas a `take!()`:

```
julia> function producer(c::Channel)
           put!(c, "start")
           for n=1:4
               put!(c, 2n)
           end
           put!(c, "stop")
       end;

julia> chnl = Channel(producer);

julia> take!(chnl)
"start"

julia> take!(chnl)
2

julia> take!(chnl)
4

julia> take!(chnl)
6

julia> take!(chnl)
8

julia> take!(chnl)
"stop"
```

Una forma de pensar en este comportamiento es que el `producer` era capaz de retornar múltiples veces. Entre las llamadas a `put!()`, la ejecución del productor se ha suspendido y el consumidor tiene el control.

El objeto `Channel` devuelto puede ser usado como un objeto iterable dentro de un bucle `for` loop, en cuyo caso las variable del bucle tomará todos los objetos producidos. El bucle será terminado cuando el canal se haya cerrado.

```
julia> for x in Channel(producer)
           println(x)
       end
start
2
4
6
8
stop
```

Note que nosotros no tuvimos que cerrar explícitamente el canal en el productor. Esto es debido a que el acto de enlazar un canal (`Channel`) a una tarea (`Task()`) asocia el tiempo de vida abierto de un canal con el de la tarea asociada. El objeto canal se cierra automáticamente cuando la tarea termina. Podemos enlazar múltiplos canales a una tarea, y viceversa.

Aunque el constructor de `Task()` espere una función sin argumentos, el método `Channel()` que crea un enlace entre un canal y una tarea espera una función que acepta un solo argumento de tipo `Channel`. Un patrón común es que el productor esté parametrizado, en cuyo caso se neceesita una aplicación de función parcial para crear una función anónima con 1 ó 0 argumentos..

Para objetos `Task()` esto puede hacerse bien directamente o mediante el uso de una macro conveniente:

```
function mytask(myarg)
    ...
end

taskHdl = Task(() -> mytask(7))
# or, equivalently
taskHdl = @task mytask(7)
```

Para orquestar patrones de distribucióin más avanzados, pueden usarse `bind()` y `schedule()` en conjunción con los constructores de `Task()` y `Channel()` para enlazar explícitamente un conjunto de canales con un conjunto de tareas productor/consumidor.

Note que en la actualidad las tareas Julia no son planificadas para que ejecuten sobre núcleos de CPU separados. Los verdaderos hilos del núcleo se discutirán en la sección Computación Paralela.

Operaciones Básicas de Tareas

Exploremos la construcción de bajo nivel `yieldto()` para comprender cómo funciona la conmutación de tareas. `yieldto(task,value)` suspende la tarea actual, conmuta a la tarea especificada, y causa que la última llamada a `yieldto()` devuelva el valor especificado `value`. Nótese que `yieldto()` para usar control de flujo estilo tarea: en lugar de llamar y retornar nos limitamos a conmutar entre las distintas tareas. Esta es la razón por la que esta característica es también llamada ”corrutinas simétricas”. Cada tarea es conmutada usando el mismo mecanismo.

`yieldto()` es potente, pero la mayoría de los usos de tareas no lo invocan directamente. Consideremos a qué se debe esto. Si tu conmutas desde la tarea actual, probablemente querrás volver a conmutar en otro puento, pero saber cuándo conmutar, y saber qué tarea tiene la responsabilidad de conmutar hacia atrás puede requerir una coordinación considerable. Por ejemplo, `put!()` y `take!()` son operaciones bloqueantes, las cuales, cuando se usan en el contexto de los canales mantienen un estado para recordar quiénes son los consumidores. No necesitar mantener manualmente la traza de la tarea es lo que hace que `put!()` sea más sencilla de usar que la instrucción de bajo nivel `yieldto()`.

Ademas de `yieldto()`, se necesitan otras funciones básicas para usar las tareas de forma efectiva:

- `current_task()` devuelve una referencia a la tarea que se está ejecutando actualmente.
- `istaskdone()` consulta para saber si una tarea ha salido.
- `istaskstarted()` consulta para saber si una tarea se ha iniciado ya.
- `task_local_storage()` manipula un almacenamiento clave-valor específico a la tarea actual.

Tareas y Eventos

Muchos cambios de tarea ocurren como resultado de la espera de eventos tales como peticiones de E/S, y son realizados por un planificador incluido en la librería estándar. El planificador mantiene una cola de tareas ejecutables, y ejecuta un bucle de eventos que reinicia las tareas basándose en eventos externos tales como la llegada de un mensaje.

La función básica para esperar un evento es `wait()`. Hay varios objetos que implementan `wait()`; por ejemplo, dado un objeto `Process`, `wait()` esperará a que este salga. `wait()` suele ser implícito; por ejemplo, una llamada a `wait()` puede tener lugar dentro de una llamada a `read()` para esperar a que haya datos disponibles.

En todos estos casos, `wait()` opera últimamente sobre un objeto `Condition` que es responsable de encolar y reiniciar las tareas. Cuando una tarea llama a `wait()` sobre un objeto `Condition`,la tarea es marcada como no ejecutable, añadida a la cola de esta condición y el control pasa al planificador. El planificador se ocupa entonces de preparar otra tarea para ejecución o se queda bloqueado esperando eventos externos. Si todo va bien, eventualmente un manejador de eventos llamará a `notify()` sobre la condición, lo que causa que las tareas que estaban esperando esa condición se vuelvan ejecutables de nuevo.

Una tarea creada explícitamente llamado a `Task` es inicialmente no conocida por el planificador. Esto nos permite gestionar las tareas manualmente usando `yieldto()` si lo deseamos. Sin embargo, cuando tal tarea espera un evento, sigue siendo reiniciada cuando el evento tiene lugar, como podría esperarse. Es también posible hacer que el planificador ejecute una tarea siempre que pueda, sin esperar ningún evento necesariamente. Esto se hace llamando a `schedule()`, o usando las macros `@schedule` o `@async` macros (ver Parallel Computing para más detalles).

Estados de una Tarea

La tareas tienen un campo `state` que describe su estado de ejecucoión. El estado de una tarea es uno de los siguientes símbolos:

Symbol	Meaning
`:runnable`	Ejecutando actualmente, o disponible para ser intercambiado
`:waiting`	Bloqueado esperando un evento específico
`:queued`	En la cola de ejecución del planiticador a punto de ser reiniciado
`:done`	Finalizada su ejecucción con éxito
`:failed`	Finalizado con alguna excepción no atrapada

Chapter 13

Ámbito de las variables

El *ámbito* de una variable es la región de código donde dicha variable es visible. El ámbito de las variables ayuda a evitar conflictos de nombrado de variables. El concepto es intuitivo: dos funciones pueden tener argumentos denominados x sin que las dos x se refieran a la misma cosa. De forma similar, hay muchos otros casos donde diferentes bloques de código pueden usar el mismo nombre sin referirse a la misma cosa. Las reglas para cuando el mismo nombre de variable se refiere o no a la misma cosa se llaman *reglas de ámbito*. Em este tema se analizan en detalle.

Ciertas construcciones en el lenguaje introducen *bloques de ámbitos*, que son regiones de código que son elegibles para estar en el ámbito de algún conjunto de variables. El ámbito de una variable no puede ser un conjunto arbitrario de líneas de código; en lugar de ello, siempre se alinea con uno de esos bloques. Hay dos tipos principales de ámbitos en Julia: *globales* y *locales*, pudiendo los últimos estar anidados. Las construcciones que introducen estos bloques de ámbito son:

Nombre de ámbito	Bloque/construcción que introduce este tipo de ámbito
Ámbito Global	`module`, `baremodule` y prompt interactivo (REPL)
Ámbito Local	Ámbito local blando: `for`, `while` comprensiones, bloques `try-catch-finally`, `let`
Ámbito Local	Ámbito local duro: funciones (cualquier sintaxis, anónima y bloques do), `struct`, `macro`

Dos notables ausencias en esta tabla son los bloques begin y los bloques if, que no introducen nuevos bloques de ámbito. Los tres tipos de bloques siguen reglas un poco diferentes que serán explicadas más adelante, así como algunas reglas extra para ciertos bloques.

Julia usa un ámbito léxico, lo que significa que el ámbito de una función no hereda del ámbito que lo invocó, pero si del ámbito en que la función fue definida. Por ejemplo, en el siguiente código, x deontro de `foo` se refiere a la x que hay en el ámbito local de su módulo `Bar`:

```
julia> module Bar
           x = 1
           foo() = x
       end;
```

y no a la x en el ámbito en que se ha usado `foo`:

```
julia> import .Bar

julia> x = -1;

julia> Bar.foo()
1
```

Por tanto, *ámbito léxico* significa que el ámbito de las variables puede ser inferido del código fuente sin más.

13.1 Ámbito Global

Cada módulo introduce un nuevo espacio global, separado del ámbito global de todos los otros módulos; no existen ámbitos globales compartidos por todos. Los módulos pueden introducir variables de otros módulos o en su ámbito a través del uso de las instrucciones `using` o `import` o a través de acceso cualificado usando la notación punto. En consecuencia, cada módulo es un espacio de nombres. Notese que los enlaces de nombres pueden sólo ser cambiados dentro de su ámbito global y no desde un módulo exterior.

```
julia> module A

           a = 1 # a global in A's scope

       end;

julia> module B

           module C

               c = 2

           end

           b = C.c    # can access the namespace of a nested global scope

                      # through a qualified access

           import ..A # makes module A available

           d = A.a

       end;

julia> module D

           b = a # errors as D's global scope is separate from A's

       end;
ERROR: UndefVarError: a not defined

julia> module E

           import ..A # make module A available

           A.a = 2    # throws below error

       end;
ERROR: cannot assign variables in other modules
```

Nótese que el prompt interactivo (REPL) está en el ámbito global del módulo `Main`.

13.2 Ámbito Local

La mayoría de los bloques de código introducen un nuevo ámbito local. Los ámbitos locales suelen heredar todas las variables de su ámbito padre, tanto para lectura como para escritura. Hay dos subtipos de ámbitos locales, denominados *duros* y *blandos*, con reglas ligeramente distintas en relación a qué variables son heredadas. A diferencia de los

ámbitos globales, los ámbitos locales no son espacios de nombres, por lo que las variables de un ámbito más interno no pueden ser recuperadas de uno más externo a través de alguna clase de acceso cualificado.

Las siguientes reglas y ejemplos pertenecen tanto a los ámbitos locales como globales. Una variable introducida de nuevo en un ámbito local no se retroprogada a su ámbito padre. Por ejemplo, la z no se introduce en el ámbito de nivel superior:

```
julia> for i = 1:10
           z = i
       end

julia> z
ERROR: UndefVarError: z not defined
```

(Nótese que En este ejemplo y los siguientes se supone que el ámbito de nivel superior es un ámbito global con un espacio de trabajo limpio, por ejemplo un REPL arrancado de nuevo.)

Dentro de un ámbito local una variable puede ser forzada a ser una variables local usando la palabra clave `local`.

```
julia> x = 0;

julia> for i = 1:10
           local x
           x = i + 1
       end

julia> x
0
```

Dentro de un ámbito local puede definirse una nueva variable global usando la palabra clave `global`:

```
julia> for i = 1:10
           global z
           z = i
       end

julia> z
10
```

La localización de las palabras clave `local` y `global` dentro del bloque del ámbito es irrelevante. El siguiente código es totalmente equivalente al ejemplo anterior (aunque estilísticamente es peor):

```
julia> for i = 1:10
           z = i
```

```
            global z

        end

julia> z
10
```

Ámbito local blando

> En un ámbito local blando, todas las variables son heredadas de su ámbito padre a menos que una variable haya sido marcada específicamente con la palabra `local`.

Los ámbitos locales blandos se introducen en los bucles `for`, bucles `while`, comprensiones, bloques `try-catch-finally` y bloques `let`. Hay algunas reglas extra para los bloques `let` y para los bucles `for` y comprensiones.

En el siguiente ejemplo, x e y se refieren siempre a la misma variable dado que el ámbito local blando heredan ambas variables de lectura y escritura:

```
julia> x, y = 0, 1;

julia> for i = 1:10

            x = i + y + 1

        end

julia> x
12
```

Dentro de los ámbitos blandos, la palabra clave *global* no es nunca necesaria, aunque está permitida. El único caso donde podría cambiar la semántica es (actualmente) un error sintáctico:

```
julia> let

            local j = 2

            let

                global j = 3

            end

        end
ERROR: syntax: `global j`: j is local variable in the enclosing scope
```

Ámbito local duro

Los ámbitos locales duros se introducen mediante las definiciones de función (en todas sus formas) bloques de tipos e inmutables y definiciones de macros.

> En el ámbito local duro, todas las variables son heredades de su ámbito padre a menos que:

- Una asignación daría como resultado una variable `global`
- Una variable sea marcada específicamente con la palabra clave `local`.

Por tanto, las variables globales son sólo heredadas para lectura pero no para escritura:

```
julia> x, y = 1, 2;

julia> function foo()
           x = 2        # assignment introduces a new local
           return x + y # y refers to the global
       end;

julia> foo()
4

julia> x
1
```

Se necesita un `global` explícito para asignar a una variable global:

```
julia> x = 1;

julia> function foobar()
           global x = 2
       end;

julia> foobar();

julia> x
2
```

Note que las *funciones anidadas* pueden comportarse diferentemente de las funciones definidas en el ámbito global como si ellas pudieran variar las variables locales del ámbito padre.

```
julia> x, y = 1, 2;

julia> function baz()
           x = 2 # introduces a new local
           function bar()
               x = 10       # modifies the parent's x
               return x + y # y is global
           end
```

```
            return bar() + x # 12 + 10 (x is modified in call of bar())

        end;

julia> baz()
22

julia> x, y
(1, 2)
```

La distinción entre heredar variables locales y globales para asignación puede llevar a ligeras diferencias entre funciones definidas en ámbitos locales y/o globales. Considere la modificación del último ejemplo moviendo `bar` al ámbito global:

```
julia> x, y = 1, 2;

julia> function bar()

            x = 10 # local

            return x + y

        end;

julia> function quz()

            x = 2 # local

            return bar() + x # 12 + 2 (x is not modified)

        end;

julia> quz()
14

julia> x, y
(1, 2)
```

Notemos que lo anterior sutilmente no pertenece a las definiciones de tipo y de macro, por lo que ellas sólo pueden aparecer en el ámbito global. Hay reglas de ámbito especiales relacionadas con la evaluación de argumentos de función por defecto y palabra clave que se describen en la sección de Funciones.

Una asignación que introduce una variable usada dentro de una definicíon de función, tipo o macro no necesita ir antes de su uso interno:

```
julia> f = y -> y + a
(::#1) (generic function with 1 method)

julia> f(3)
ERROR: UndefVarError: a not defined
Stacktrace:
 [1] (::##1#2)(::Int64) at ./none:1

julia> a = 1
1
```

```
julia> f(3)
4
```

Este comportamiento puede parecer un poco raro para una variable normal, pero está permitido para que las funciones nombradas sean usadas antes de ser definidas (las funciones nombradas son exactamente variables normales que almacenan objetos función). Esto permite a las funciones ser definidas en cualquier orden que sea intuitivo y conveniente en lugar de forzar un ordenamiento de abajo a arriba o requerir declaraciones hacia delante, mientras que ellas sean definidas en el momento que sean usadas. Por ejemplo, he aquí una forma ineficiente, mutuamente recursiva de comprobar si un entero positivo es par o impar:

```
julia> even(n) = n == 0 ? true : odd(n-1);

julia> odd(n) = n == 0 ? false : even(n-1);

julia> even(3)
false

julia> odd(3)
true
```

Julia proporciona funciones eficientes y predefinidas para comprobar la paridad o imparidad, llamadas `iseven()` e `isodd()`. Por tanto, las definiciones anteriores deberían ser sólo tomadas como ejemplos.

Ámbitos locales duro vs. blando

Los bloques que introducen un ámbito local blando, como los bucles, se suelen usar para manipular las variables en su ámbito padre. Por tanto, su defecto es acceder completamente a todas las variables de su ámbito padre.

A la inversa, el código dentro de los bloques que introduce un ámbito local duro (definiciones de función, tipo o macro) pueden ser ejecutados en cualquier parte del programa. Cambiar remotamente el estado de variables locales en otros módulos debería ser realizado con cuidado y por tanto esta es una característica de optimización que requiere la palabra clave `global`.

La razón para permitir *modificar local* variables de ámbitos padres en funciones anidadas es permitir la construcción de cierres que tienen un estado privado, por ejemplo la variable `state` del siguiente ejemplo:

```
julia> let
           state = 0
           global counter
           counter() = state += 1
       end;

julia> counter()
1

julia> counter()
2
```

Ver también los cierres en los ejemplos de las dos siguientes secciones.

Bloques Let

A diferencia de las asignaciones a variables locales, las instrucciones `let` asignan nuevas asociaciones de variables cada vez que se ejecutan. Una asignación modifica el valor de una localización existente, y `let` crea nuevas localizaciones. Esta diferencia no suele ser importante, y es sólo detectable en el caso de variables que sobreviven a sus ámbitos vía cierres. La sintaxis de `let` acepta una serie de asignaciones y nombres de variables separados por comas:

```
julia> x, y, z = -1, -1, -1;

julia> let x = 1, z

           println("x: $x, y: $y") # x is local variable, y the global

           println("z: $z") # errors as z has not been assigned yet but is local

       end
x: 1, y: -1
ERROR: UndefVarError: z not defined
```

Las asignaciones se evalúan en orden, con cada término derecho evaluado en el ámbito antes de que se introduzca la nueva variable a través del término izquierdo. Por tanto, tiene sentido escribir algo como `let x = x` ya que las dos variables son distintas y tienen almacenamiento separado. Este es un ejemplo de dónde se necesita el comportamiento de `let`:

```
julia> Fs = Array{Any}(2); i = 1;

julia> while i <= 2

           Fs[i] = ()->i

           i += 1

       end

julia> Fs[1]()
3

julia> Fs[2]()
3
```

Aquí creamos y almacenamos dos cierres que devuelven la variable `i`. Sin embargo, es siempre la misma variable `i`, por lo que los dos cierres se comportan de forma idéntica. Podemos usar `let` para crear una nueva correspondencia para `i`:

```
julia> Fs = Array{Any}(2); i = 1;

julia> while i <= 2

           let i = i

               Fs[i] = ()->i

           end
```

```
            i += 1

        end

julia> Fs[1]()
1

julia> Fs[2]()
2
```

Como la construcción `begin` no construye un nuevo ámbito, puede ser útil usar un `let` con cero argumentos para introducir un nuevo bloque de ámbito sin crear ninguna nueva correspondencia:

```
julia> let

            local x = 1

            let

                local x = 2

            end

            x

        end
1
```

Como `let` introduce un nuevo bloque de ámbito, la variable local interna `x` es diferente de la externa local `x`.

Bucles for y comprensiones

Los bucles `for` y las comprensiones tiene el siguiente comportamiento: cualquier nueva variable introducida en sus ámbitos se reservan de nuevo para cada nueva iteración del bucle. Esto contrasta con los bucles `while` que reservan las variables para todas las iteraciones. Por tanto, estas construcciones son similares a bucles `while` con bloques `let` dentro de ellos:

```
julia> Fs = Array{Any}(2);

julia> for j = 1:2

            Fs[j] = ()->j

        end

julia> Fs[1]()
1

julia> Fs[2]()
2
```

Los bucles `for` reusarán las variables existentes para su variable iteración:

```
julia> i = 0;

julia> for i = 1:3

       end

julia> i
3
```

Sin embargo, las comprensiones no hacen esto, y siempre asignan de nuevo sus variables de iteración:

```
julia> x = 0;

julia> [ x for x = 1:3 ];

julia> x
0
```

13.3 Constantes

Un uso común de las variables es darle nombres de valores específicos, no cambiantes. Tales variables sólo se asignan una vez. Esta intención puede ser transportada al compilador usando la palabra clave `const`:

```
julia> const e  = 2.71828182845904523536;

julia> const pi = 3.14159265358979323846;
```

La declaración `const` es permitida sobre variables globales y locales, pero es especialmente útil para las globales. Es difícil para el compilador optimizar´código en el que están implicadas las variables globales, ya que sus valores (o incluso sus tipos) podrían cambiar en cualquier momento. Si una variable global no va a cambiar, añadir una declaración `const` resolverá este problema de rendimiento.

Las constantes locales son bastante diferentes. El compilador es capaz de determinar cuando una variable local es constante, por lo que las declaraciones de constante local no son necesarias para mejorar el rendimiento.

Las asignaciones especiales de nivel superior, tales como las realizadas por las palabras clave `function` y `type` son constantes por defecto.

Nótese que `const` sólo afecta a la asociación de variables: la variable puede ser asociada a un objeto mutable (tal com un array) y el objeto puede aún ser modificado.

Chapter 14

Tipos

Los sistemas de tipos han caído tradicionalmente en dos categorías muy diferentes: los *sistemas de tipos estáticos*, donde cada expresión del programa debe tener un tipo computable antes de la ejecución del programa, y los *sistemas de tipos dinámicos*, donde nada es sabido sobre los tipos hasta el momento de la ejecución, cuando los valores actuales manipulados por el programa están disponibles. La orientación a objetos permite una flexibilidad en los lenguajes tipados estáticamente dejando que el código sea escrito sin que se conozcan los tipos precisos de los valores en tiempo de compilación. La capacidad de escribir código que pueda operar sobre diferentes tipos se denomina polimorfismo. Todo el código en los lenguajes clásicos tipados dinámicamente es polimórfico: sólo mediante comprobación de equipos explícita o cuando los objetos fallan para soportar las operaciones en tiempo de ejecución, están los tipos de cualquier valor siempre restringidos.

El sistema de tipos de Julia es dinámico, pero tiene algunas de las ventajas de los sistemas de tipos estáticos haciendo posible indicar que ciertos valores son de tipos específicos. Esto puede ser de gran ayuda en la generación de código eficiente, pero incluso más significativamente, permite que el despacho de métodos sobre los tipos de los argumentos a función este profundamente integrado con el lenguaje. El despacho de métodos se explorará en detalle en la sección Methods, pero está enraizado en el sistema de equipos presentado en este capítulo..

El comportamiento por defecto en Julia cuando se omiten los tipos es permitir que los valores sean de cualquier tipo. Por tanto, uno puede escribir muchos programa Julia útiles sin siquiera usar explícitamente los tipos. Cuando se necesita una expresividad adicional, sin embargo, es fácil introducir gradualmente anotaciones de tipo explícitas en código previamente no tipado. Hacer eso suele incrementar el rendimiento y la robustez de estos sistemas, y quizás algo contra intuitivo: simplificarlos frecuentemente.

Describir Julia en el lingo de los sistemas de tipos, es decir que es: *dinámico*, *nominativo* y *paramétrico*. Los tipos genéricos pueden ser parametrizados, y las relaciones jerárquicas entre tipos son declaradas explítamente, en lugar de ser implicadas mediante una estructura compatible. Una característica particularmente distintiva del sistema de tipos de Julia es que los tipos concretos no pueden tener subtipos. Todos los tiempos completos son finales y sólo pueden tener tipos abstractos como supertipos. Aunque esto puede parecer excesivamente restrictivo al principio, tiene muchas consecuencias beneficiosas con sorprendentemente pocos inconveniente. Resulta que ser capaz de heredar comportamientos es mucho más importante que seas capaz de heredar estructura, y heredar ambas cosas causa dificultades significativas en los lenguajes orientados a objetos tradicionales. Otros aspectos de alto nivel del sistema del tipo de Julia que Debería ser mencionados son:

- No hay división entre los valores objetos y no objetos. Todos los valores en Julia son verdaderos objetos que tienen un tipo que pertenece a un solo grafo de tipos totalmente conectado, todos los nodos del cual son de primera clase como tipos.

- No hay un concepto significativo de tiempo en tiempo de compilación. El único tipo que tiene un valor es su tipo actual cuando el programa está corriendo. Esto se denomina tipo en tiempo de ejecución en los lenguajes

orientados a objetos, donde la combinación de complicación estática con polimorfismo hace esta distinción significativa.

- Sólo los valores, no las variables, tienen tipos. Las variables son siempre nombres enlazados a valores.
- Tanto los tipos abstractos como concretos pueden ser paralizados por otros tipos. Ellos pueden ser parametrizados mediante símbolos, mediante valores de cualquier tipo para los cuáles `isbits()` devuelve `true` (esencialmente, cosas como números y booleanos que son almacenados como tipos C o estructuras sin punteros a otros objetos), y también mediante tuplas. Los parámetros de tipo pueden ser omitidos cuando no necesitan ser referenciados o restringidos.

El sistema de tipos de Julia está diseñado para ser potente y expresivo, además de claro, intuitivo y no obstructivo. Muchos programadores Julia nunca sentirán la necesidad de escribir código que use los tipos explícitamente. Algunas clases de programación, sin embargo, se vuelven más claras, rápidas y robustas usando tipos declarados.

14.1 Declaraciones de tipo

El operador `::` puede usarse para adjuntar declaraciones de tipo a expresiones y variables en los programas. Hay dos razones principales para esto:

1. Como una aserción para ayudar a confirmar que nuestro programa funciona como se esperaba.
2. Para proporcionar al compilador información extra sobre tipos, que puede mejorar el rendimiento en algunos casos.

Cuando se añade a una expresión que calcula un valor, el operador `::`se lee como ”es una instancia de”. Puede ser utilizado en cualquier parte para asertar que el valor de la expresión de la izquierda es una instancia del tipo de la derecha. Cuando el tipo de la derecha es concreto, el valor sobre la izquierda debe tener ese tipo como su implementación (recuerde que todos los tipos concretos son finales, por lo que no hay implementaciones que sean subtipos de otros). Cuando el tipo es abstracto, basta que el valor sea implementado por un tipo concreto que sea un subtipo del tipo abstracto. Si la aserción de tipo no es `true` se lanza una excepción. En caso contrario, se devuelve el valor del lado izquierdo.

```
julia> (1+2)::AbstractFloat
ERROR: TypeError: typeassert: expected AbstractFloat, got Int64

julia> (1+2)::Int
3
```

Esto permite que la aserción de tipo sea adjuntada a cualquier expresión *in situ*.

Cuando se añade a una variable sobre el lado izquierdo de una asignación, o como parte de una declaración `local`, el operador `::` significa algo un poco diferente: declara que la variable siempre tendrá el tipo especificado, como una declaración de tipo de los lenguajes tipados estáticamente como C. Cada valor asignado a la variable será convertido al tipo declarado usando `convert()`:

```
julia> function foo()

           x::Int8 = 100

           x
```

```
    end
foo (generic function with 1 method)

julia> foo()
100

julia> typeof(ans)
Int8
```

Esta característica es útil para evitar las ”trampas” de rendimiento que podrían tener lugar si una de las asignaciones a varible cambia su tipo inesperadamente.

Este comportamiento ”declaración” sólo ocurre en contextos específicos:

```
local x::Int8  # in a local declaration
x::Int8 = 10   # as the left-hand side of an assignment
```

y se aplica al ámbito actual completo, incluso antes de la declaración. Actualmente, las declaraciones de tipo no pueden ser usadas en un espacio global, como en el REPL, ya que Julia no tiene aún globales de tipo constante.

Las declaraciones pueden también ser enlazadas a las definiciones de función:

```
function sinc(x)::Float64
    if x == 0
        return 1
    end
    return sin(pi*x)/(pi*x)
end
```

Retornar de esta función se comporta justo como una asignación a una variable con un tipo declarado: el tipo será siempre convertido a `Float64`.

14.2 Tipos Abstractos

Los tipos abstractos no puede ser instanciados, y sólo sirven como nodos en el grafo de tipos, describiendo de este mundo conjuntos de tipos concretos relacionados: aquellos tipos concretos que son sus descendientes. Comenzamos con tipos abstractos incluso aunque no tienen instanciación debido a que ellos son la espina dorsal del sistema de tipos: ellos forman la jerarquí conceptual qu hace al sistema de tipos de Julia más que una colección de implementaciones de objetos.

Recuerde que en Números Enteros y en Punto Flotante, introdujimos una variedad de tipos de valores numéricos concretos: `Int8`, `UInt8`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Int128`, `UInt128`, `Float16`, `Float32`, and `Float64`. Aunque todos ellos tienen diferentes tamaños en representación, `Int8`, `Int16`, `Int32`, `Int64` and `Int128` tienen en común que son tipos enteros con signo. Del mismo modo, `UInt8`, `UInt16`, `UInt32`, `UInt64` and `UInt128` son enteros sin signo, mientras que `Float16`, `Float32` and `Float64` son tipos en punto flotante. Es común para una pieza de código que ésta tenga sentido, por ejemplo, sólo si sus argumentos son algún tipo de entero, pero no que dependa de un tipo de entero particular. Por ejemplo, el algoritmo del máximo común denominador funciona para todas las clases de enteros, pero no funcionará para los números en punto flotante. Los tipos abstractos permiten la construcción de una jerarquía de tipos, proporcionando un contexto en el cuál los tipos concretos pueden ajustarse. Esto te permite, por ejemplo, programar fácilmente a cualquier tipo que sea un entero, sin restringir el algoritmo a un tipo de entero específico.

Los tipos abstractos se declaran usando la palabra clave `abstract`. Las sintaxis generales para declarar un tipo abstracto son:

```
abstract type «name» end
abstract type «name» <: «supertype» end
```

La palabra clave `abstract type` introduce un nuevo tipo abstracto, cuyo nombre viene dado por `«name»`. Este nombre puede ir seguido opcionalmente de `<:` y un nombre de tipo ya existente, lo cuál indica que este tipo abstracto es un subtipo del ya existente..

Cuando no se proporciona supertipo, el supertipo por defecto es `Any` (un tipo `abstract` predefinido del que todos los objetos son instancias y todos los tipos son subtipos). En teoría de tipos, `Any` suele ser denominado *top* porque es la cúspide del grafo de los tipos. Julia tiene también un tipo abstracto *bottom*, en el punto más bajo del grafo de tipos, que se escribe como `Union{}`. Es el opuesto exacto de `Any`: ningún objeto es instancia de `Union{}` y todos los tipos son sus supertipos.

Consideremos algunos de los tipos abstractos que forman parte de la jerarquía numérica de Julia:

```
abstract type Number end
abstract type Real     <: Number end
abstract type AbstractFloat <: Real end
abstract type Integer  <: Real end
abstract type Signed   <: Integer end
abstract type Unsigned <: Integer end
```

El tipo `Number` es un hijo directo de `Any`, y `Real` es su hijo. `Real` tiene dos hijos (tiene más, pero sólo mostraremos dos aquí): `Integer` y `AbstractFloat`, que dividen el mundo entre representaciones de números enteros y reales. Las representaciones de números reales incluyen, por supuesto, los tipos en punto flotante, pero también incluyen otros tipos como los racionales. Por tanto, `AbstractFloat` es un subtipo apropiado de `Real` que incluye sólo representaciones en punto flotante de los números reales. Los enteros están también divididos en las variedades `Signed` y `Unsigned`.

El operador `<:` significa, en general, "es un subtipo de" y, usado en declaraciones como esta, declara que el tipo de la parte derecha es un supertipo inmediato de tipo que acaba de crearse. También puede usarse en expresiones como un obperador de subtipo que devuelve `true` cuando el operando de su izquierda es un subtipo del operando de su derecha:

```
julia> Integer <: Number
true

julia> Integer <: AbstractFloat
false
```

Un uso importante de los tipos abstractos es proporcionar una implementación por defecto para los tipos concretos. Para dar un ejemplo simple, considere:

```
function myplus(x,y)
    x+y
end
```

La primera cosa que hay que notar es que las declaraciones de argumento anteriores son equivalentes a `x::Any` e `y::Any`. Cuando se invoca a estas funciones, digamos con `myplus(2,5)`, el despachador elige el método más específico cuyo nombres sea `myplus` y que se corresponda con los argumentos dados (ver Métodos para más informacíon sobre despacho múltiple).

Asumiendo que no se encuentra método más específico que el anterior, a continuación Julia define y compila un método llamado `myplus` específicamente para dos argumentos `Int` basado en la función genérica dada anteriormente, es decir, implícitamente define y compila:

```
function myplus(x::Int,y::Int)
    x+y
end
```

y, finalmente invoca a este método específico.

Por tanto, los tipos abstractos permiten a los programadores escribir funciones genéricas que puedan ser usadas después como el método por defecto mediante muchas combinaciones de tipos concretos. Gracias al despacho múltipel, el programador tiene control total sobre si se usa el método por defecto o uno más específico.

Un punto importante que notar es que no hay pérdida en el rendimiento si el programador se basa en una función cuyos argumentos sean tipos abstractos, dado que ella es recompilada para cada tupla de argumentos de tipos concretos con la cuál sea invocada (sin embargo, puede haber un problema de rendimiento en el caso de argumentos función que sean contenedores de tipos abstractos; ver Consejos de Rendimiento.)

14.3 Tipos Primitivos

Un tipo primitivo es aquél un tipo concreto cuyos datos consisten en bits normales y corrientes. Ejemplos clásicos de tipos bits son los valores enteros y punto flotante. A diferencia de muchos lenguajes, Julia nos permite declarar nuestros propios tipos bits, en lugar de proporcionar un conjunto fijo de tipos bits predefinidos. De hecho, los tupos bits estándar que están definidos en el propio lenguaje son:

```
primitive type Float16 <: AbstractFloat 16 end
primitive type Float32 <: AbstractFloat 32 end
primitive type Float64 <: AbstractFloat 64 end

primitive type Bool <: Integer 8 end
primitive type Char 32 end

primitive type Int8    <: Signed   8 end
primitive type UInt8   <: Unsigned 8 end
primitive type Int16   <: Signed   16 end
primitive type UInt16  <: Unsigned 16 end
primitive type Int32   <: Signed   32 end
primitive type UInt32  <: Unsigned 32 end
primitive type Int64   <: Signed   64 end
primitive type UInt64  <: Unsigned 64 end
primitive type Int128  <: Signed   128 end
primitive type UInt128 <: Unsigned 128 end
```

Las sintaxis generales para la declaración de un tipo primitivo son:

```
primitive type «name» «bits» end
primitive type «name» <: «supertype» «bits» end
```

«bits» indica cuánta memoria requiere el tipo y «name» indica el nombre del nuevo tipo. Un tipo primitivo puede ser declarado opcionalmente como un subtipo de algún supertipo. Si se omite el supertipo, se asigna como supertipo por defecto el tipo Any. Por tanto, la declaración de Bool significa que un valor booleano consume 8 bits de almacenamiento, y que Integer es su supertipo inmediato. Actualmente sólo se soportan tamaños que sea múltiplo de 8 bits. Por tanto, los valores booleanos, aunque sólo necesitan un bit, no puden ser declarados como menores de 8 bits.

Los tipos Bool, Int8 y UInt8 tienen representaciones idénticas: se trata de bloques de memoria de 8 bits. Como el sistema de tipos de Julia es nominativo, sin embargo, ellos no son intercambiables aunque tengan estructura idéntica. Otra diferencia fundamental entre ellos es que ellos tienen supertipos diferentes: Integer es el supertipo directo de

`Bool`, `Signed` es el de `Int8`, y `Unsigned` es el de `UInt8`. Todas las demás diferencias entre `Bool`, `Int8`, y `UInt8` son cuestiones de comportamiento (la forma en la que las funciones son definidas para actuar cuando se pasan como argumentos objetos de estos tipos). Esta es la razón por la que es necesario un sistema de tipos nominativo: si la estructura determinara el tipo, que a su vez dicta el comportamiento, sería imposible hacer que los valores `Bool` se comportaran de forma diferente a los `Int8` o los `UInt8`.

14.4 Tipos Compuestos

Los tipos compuestos se llaman registros, estructuras (*structs*) u objetos en distintos lenguajes. Un tipo compuesto es una colección de campos nombrados, una instancia de los cuales puede ser tratada como un único valor. En muchos lenguajes, los tiempos compuestos sos la única clase de tipos definidos por el usuario, y ellos son de lejos el tiempo definido por el usuario que se usa mas comúnmente en el lenguaje Julia.

En el mundo de los lenguajes orientados a objetos tales como C++, Java, Python o Ruby, los tipos compuestos también tienen funciones asociadas a ellos, y esa combinación se denomina "objeto". En los lenguajes orientados a objetos más puros, tales como Ruby o SmallTalk, todo los valores son objetos sean compuestos o no. En lenguajes orientados objetos menos puros incuyendo C++ y Java, algunos valores tales como los enteros no se consideran objetos, mientras que las instancias de los tipos compuestos definidos por el usuario son verdaderos objetos con métodos asociados. En Julia, todos los valores son objetos, pero no hay funciones a los objetos sobre los que se opera. Esto es necesario ya que Julia elige qué método de una función usar mediante despacho múltiple, lo que significa que cuando se selecciona un método se consideran los tipos de todos los argumentos de la función y no sólo el primero (ver la sección Métodos para más información). Por tanto, sería inapropiado para las funciones "pertenecer" sólo a su primer argumento (el objeto que la posee). Organizar métodos en objetos función en lugar de tener bolsas de métodos nombrados "dentro" de cada objeto termina por ser un aspecto muy beneficioso de diseño del lenguaje.

Los tipos compuestos son introducidos por la palabra clave `struct` seguida por un bloque de nombres de campos, opcionalmente anotados con tipos usando el operador `::`

```
julia> struct Foo
           bar
           baz::Int
           qux::Float64
       end
```

Los campos sin anotación de tipos tiene asignado Any como tipo por defecto, y pueden según esto almacenar cualquier tipo de valor.

Para crear nuevos objetos del tipo compuesto Foo se crean aplicando el tipo objeto Foo como una función con valores para sus campos:

```
julia> foo = Foo("Hello, world.", 23, 1.5)
Foo("Hello, world.", 23, 1.5)

julia> typeof(foo)
Foo
```

Cuando un tipo es aplicado como una función es llamado *constructor*. Hay dos constructores, denominados *constructores por defecto* que se generan automáticamente al crear el tipo. Uno acepta cualquier argumento y llama a `convert()` para convertirlos a los tipos de los campos, y el otro acepta argumentos que se corresponden exactamente a los tipos de los campos. La razón de que ambos métodos sean generados es que esto hace más sencillo añadir nuevas definiciones sin reemplazar inadvertidamente a un constructor por defecto.

Como el campo `bar` no está restringido en tipo, cualquier valor es válido. Sin embargo, el valor para `baz` debe ser convertible a `Int`:

```
julia> Foo((), 23.5, 1)
```

```
ERROR: InexactError()
Stacktrace:
 [1] convert(::Type{Int64}, ::Float64) at ./float.jl:679
 [2] Foo(::Tuple{}, ::Float64, ::Int64) at ./none:2
```

La función `fieldnames` devuelve una lista de los nombres de campos de un objeto:

```
julia> fieldnames(foo)
3-element Array{Symbol,1}:
 :bar
 :baz
 :qux
```

Para acceder a los valores de los campos de un objeto compuesto puede usarse la notación tradicional `foo.bar`:

```
julia> foo.bar
"Hello, world."

julia> foo.baz
23

julia> foo.qux
1.5
```

Los tipos compuestos declarados con `struct` son *inmutables*, es decir, no pueden ser modificados después de su construcción. Esto puede parecer raro al principio, pero tiene varias ventajas:

- Puede ser más eficiciente. Algunos `struct` pueden ser empquetados eficientemente dentro de los arrays, y en algunos casos el compilador es capaz de evitar asignar objetos inmutables completamente.
- No es posible violar las invariantes proporcionadas por los contructores de tipo.
- El código con objetos inmutables puede ser más sencillo de interpretar.

Un objeto inmutable puede contener objetos mutables, tales como arrays, como campos. Esos objetos contenidos permanecerán inmutables, sólo los cambos del objeto inmutable en sí no podrán ser cambiados para apuntar a objetos distintos.

Cuando se requiera, los objetos compuestos mutables podrán ser declarados con la palabra clave `mutable struct`, lo que será discutido en la siguiente sección.

Los tipos compuestos sin campos son *singletons*, es decir, sólo puede haber una instancia de tales tipos:

```
julia> struct NoFields

       end

julia> NoFields() === NoFields()
true
```

La función `===` confirma que las dos instancias construidas de `NoFields` son de hecho una y la misma. Los tipos *singleton* se describirán en más detalle posteriormente.

Hay mucho más que decir sobre cómo se crean las instancias de los tipos compuestos, pero esta discusión depende de los Tipos Paramétricos y de los Métodos, y es suficientemente importante para ser tratada en su propia sección: Constructores.

14.5 Tipos Compuestos Mutables

Si un tipo compuesto es declarado como `mutable struct` en lugar de como `struct`, sus instancias pueden ser modificadas:

```
julia> mutable struct Bar
           baz
           qux::Float64
       end

julia> bar = Bar("Hello", 1.5);

julia> bar.qux = 2.0
2.0

julia> bar.baz = 1//2
1//2
```

Para soportar la mutación, tales objetos se alojan generalmente en el montón y tienen direcciones de memoria estables. Un objeto mutable es como un pequeño contenedor que podría almacenar distintos valores en el tiempo, y por tanto sólo puede ser identificado de forma confiable con su dirección. En contraste, una instancia de un tipo inmutable está asociada con valores de campos específicos - los valores de campos solos te dicen todo sobre el objeto. En decidir si hacer un tipo mutable, pregúntat si dos instanciasa con los mismos valores de campos tendrían que ser consideradas idénticas, o si ellas podrían necesitar cambiar independientemente con el tiempo. Si ellas fueran consideradas idénticas, el tipo probablemente debería ser inmutable.

Para recapitular, dos propiedades esenciales definen la inmutabilidad en Julia:

- Un objeto con un tipo inmutable es pasado (tanto en instrucciones de asignación como en llamadas a función) mediante copia, mientras que un tipo mutable es pasado mediante referencia.
- No está permitido modificar los campos de un tipo compuesto inmutable.

Es instructivo, particularmente para lectores cuyo background es C/C++, considerar por qué estas dos propiedades van juntas. Si fueran separadas, es decir, si los campos de los objetos pasados mediante copia pudieran ser modificados, entonces sería más difícil razonar sobre ciertas instancias de código genérico. Por ejemplo, supongamos que `x` es un argumento de función de un tipo abstracto, y supongamos que la función cambia un campo: `x.isprocessed = true`. Dependiendo de si xse pasa mediante copia o mediante referencia, esta instrucción puede alterar o no el argumento actual en la rutina que hace la llamada. Julia evita la posibilidad de crear funciones con efectos desconocidos en este escenario prohibiendo modificación de campos de objetos pasados mediante copia.

14.6 Tipos declarados

Las tres clases de tipos discutidos en las tres secciones anteriores están muy relacionados. Ellos comparten las mismas propiedades clave:

- Ellos son declarados explícitamente.
- Ellos tienen nombres.
- Ellos tienen supertipos declarados explícitamente.
- Ellos pueden tener parámeros.

Debido a estas propiedades compartidas, estos tipos son representados internamene como instancias del mismo concepto, `DataType` , que es el tipo de cualquiera de estos tipos:

```
julia> typeof(Real)
DataType

julia> typeof(Int)
DataType
```

Un `DataType` puede ser abstracto o concreto. Si es concreto, tiene un tamaño, disposición de almacenamiento y (opcionalmente) nombres de campos especificados. Por tanto, un tipo bits es un `DataType` con tamaño no nulo, pero sin nombres de campos. Un tipo compuesto es un `DataType` que tiene nombres de campos o es acío (tamaño cero).

Cada valor concreto en el sistema es una instancia de algún `DataType`.

14.7 Uniones de Tipos

Una unión de tipos es un tipo abstracto especial que incluye como objetos todas las instancias de alguno de sus tipos argumentos, construidos usando la función especial `Union`:

```
julia> IntOrString = Union{Int,AbstractString}
Union{AbstractString, Int64}

julia> 1 :: IntOrString
1

julia> "Hello!" :: IntOrString
"Hello!"

julia> 1.0 :: IntOrString
ERROR: TypeError: typeassert: expected Union{AbstractString, Int64}, got Float64
```

Los compiladores de muchos lenguajes tienen una construcción unión interna para razonar sobre los tipos; Julia simplemente la pone a disposición del programador.

14.8 Tipos Paramétricos

Una característica importante y potente del sistema de tipos de Julia es que es paramétrico: los tipos pueden toomar parámetros, por lo que las declaraciones de tipo introducen de hecho un afamilia completa de nuevos tipos (uno por cada posible combinación de valores de parámetros). Hay muchos lenguajes que soportan alguna versión de la programación genérica, donde las estructuras de datos y algoritmos para manipularlos pueden ser especificadas sin especificar los tipos exactos implicados. Por ejemplo, existe alguna forma de programación genérica en ML, Haskell, Ada, Eiffel, C++, Java, C#, F# y Scala, por nombrar unos pocos. Algunos de estos lenguajes soportan un verdadero polimorfismo paramétrico (Por ej., ML, Haskell, Scala) mientras otros soportan estilos de programación genérica *ad-hoc*, basados en plantillas (Por eje., C++ y Java). Con tantas variedades diferentes de programación genérica y de tipos paramétricos en los distintos lenguajes, no queremos ni siquiera intentar comparar los tipos paramétricos de Julia a otros lenguajes, sino que nos centraremos en explicar el propio sistema de Julia. Notaremos, sin embargo, que como Julia es un lenguaje tipado dinámicamente y no necesita hacer todas las decisiones de tipos en tiempo de compilacíon, muchas dificultades tradicionales encontradas en los sistemas de tipos paramétricos estáticos pueden ser manejadas con relativa facilidad.

Todos los tipos declarados (la variedad `DataType`) pueden ser parametrizados, con la misma sintaxis en cada caso. Los discutiremos en el siguiente orden: primero tipos compuestos paramétricos, luego tipos abstractos paramétricos y por último tipos bits paramétricos.

Tipos compuestos paramétricos

Los parámetros de tipo se introducen inmediatamente después del nombre de tipo, rodeado por llaves:

```
julia> struct Point{T}
           x::T
           y::T
       end
```

Esta declaración define un nuevo tipo paramétrico, `Point{T}`, que almacena dos coordenadas de tipo T. Uno podría preguntarse, ¿qué es T? Bien, este es precisamente la clave de los tipos paramétricos: puede ser cualquier tipo (o un valor de cualquier tipo bits, aunque en esta ocasión usado como un tipo, claramente). `Point{Float64}` es un tipo concreto equivalente al tipo definido reemplazando T en la definición de `Point` con `Float64`. Por tanto, esta única declaración declara un número de tipos ilimitado: `Point{Float64}`, `Point{AbstractString}`, `Point{Int64}`, etc. Cada uno de ellos es un tipo concreto usable:

```
julia> Point{Float64}
Point{Float64}

julia> Point{AbstractString}
Point{AbstractString}
```

El tipo `Point{Float64}` es un punto cuyas coordenadas son valores en punto flotante de 64-bits, mientras que el tipo `Point{AbstractString}` es un "punto" cuyas "coordenadas" son objetos `String` (see Strings).

`Point` es en si mismo un tipo objeto válido también, que contiene todas las instancias `Point{Float64}`, `Point{AbstractString}`, etc. como subtipos:

```
julia> Point{Float64} <: Point
true

julia> Point{AbstractString} <: Point
true
```

Otros tipos, por supuesto, no son subtipos de él:

```
julia> Float64 <: Point
false

julia> AbstractString <: Point
false
```

Los tipos `Point` concretos con valores diferentes de T no son nunca subtipos uno de otro:

```
julia> Point{Float64} <: Point{Int64}
false

julia> Point{Float64} <: Point{Real}
false
```

> **Warning**
>
> Este último punto es importante: **Incluso aunque `Float64 <: Real` no es cierto que `Point{Float64} <: Point{Real}`.**

En otras palabras, en términos de teoría de tipos, los parámetros de tipo de Julia son *invariantes*, en lugar de ser covariantes (o incluso contravariantes). Esto es por razones prácticas: aunque alguna instancia de `Point{Float64}` puede ser conceptualmente como una instancia de `Point{Real}`, los dos tipos tienen representaciones diferentes en memoria:

- Una instancia de `Point{Float64}` puede ser representada compactamente y eficientemente como un par de valores de 64 bits inmediatos
- Una instancia de `Point{Real}` debe ser capaz de alojar cualquier par de instancias de `Real`. Como los objetos son instancias de `Real` pueden ser de tamaño y estructura arbitrarios, en la práctica una instancia de `Point{Real}` debe ser representada como un par de punteros a objetos `Real` asignados individualmente.

La eficiencia ganada por ser capaz de almacenar objetos `Point{Float64}` con valores inmediatos es magnificada enormemente en el caso de arrays: un `Array{Float64}` puede almacenarse como un bloque contiguo de memoria de valores en punto flotante de 64 bits, mientras que un `Array{Real}` debe ser un array de punteros a objetos `Real` asignados individualmente (que puede ser valores en punto flotante de 64 bits envueltos (*boxed*), pero que también pueden ser objetos complejos, arbitrariamente grandes, que han sido declarados como implementaciones del tipo abstracto `Real`.

Como `Point{Float64}` no es un subtipo de `Point{Real}`, el siguiente método no puede ser aplicado a argumentos de tipo `Point{Float64}`:

```
function norm(p::Point{Real})
    sqrt(p.x^2 + p.y^2)
end
```

Una forma correcta de definir un método que acepte todos los argumentos de tipo `Point{T}`, donde T es un subtipo de `Real` es:

```
function norm(p::Point{<:Real})
    sqrt(p.x^2 + p.y^2)
end
```

(Equivalentemente, uno podría definir `function norm{T<:Real}(p::Point{T})` o `function norm(p::Point{T} where T<:Real)`; ver tipos UnionAll.)

Más ejemplos se discutirán después en Métodos.

¿Cómo construye uno un objeto `Point`? Es posible definir constructores personalizados para tipos compuestos, que serán discutidos en detalle en el capítulo Constructores, pero en ausencia de ninguna declaración especial de constructor, hay dos formas por defecto de crear nuevos objetos compuestos, uno en el que se dan explícitamente los parámetros de tipo y otro en el que ellos son implicados por los argumentos al objeto constructor.

Como el tipo `Point{Float64}` es un tipo concreto equivalente a `Point` declarado con `Float64` en lugar de T, se puede aplicar como un constructor ede acuerdo a ésto:

```
julia> Point{Float64}(1.0, 2.0)
Point{Float64}(1.0, 2.0)

julia> typeof(ans)
Point{Float64}
```

Para el constructor por defecto, debe proporcionarse exactamente un argumento por cada campo:

```
julia> Point{Float64}(1.0)
ERROR: MethodError: Cannot `convert` an object of type Float64 to an object of type Point{Float64
    }
This may have arisen from a call to the constructor Point{Float64}(...),
since type constructors fall back to convert methods.
Stacktrace:
```

```
 [1] Point{Float64}(::Float64) at ./sysimg.jl:77

julia> Point{Float64}(1.0,2.0,3.0)
ERROR: MethodError: no method matching Point{Float64}(::Float64, ::Float64, ::Float64)
```

Sólo se ha generado un constructor por defecto para tipos paramétricos, ya que sobreescribirlo no es posible. Este constructor acepta cualquier argumento y los convierte a los tipos de los campos.

En muchos casos es redundante proporcionar el tipo del objeto `Point` que uno quiere construir, ya que los tipos de los argumentos en la llamada al constructor ya proporcionan la información de tipos de forma implícita. Por esta razón, también podemos aplicar el propio `Point` como un constructor, dado que el valor implícito del parámetro de tipo `T` no es ambiguo:

```
julia> Point(1.0,2.0)
Point{Float64}(1.0, 2.0)

julia> typeof(ans)
Point{Float64}

julia> Point(1,2)
Point{Int64}(1, 2)

julia> typeof(ans)
Point{Int64}
```

En el caso de `Point` es implicado sin ambigüedad si y sólo si los dos argumentos a `Point` tienen el mismo tipo. Cuando este no es el caso, el constructor fallará con un `MethodError`:

```
julia> Point(1,2.5)
ERROR: MethodError: no method matching Point(::Int64, ::Float64)
Closest candidates are:
  Point(::T, !Matched::T) where T at none:2
```

Los método constructores para manejar apropiadamente estos casos mixtos pueden ser definidos, pero esto no será discutido hasta después en Constructores.

Tipos Abstractos Paramétricos

Las declaraciones de tipos abstractos paramétricos declaran una colección de tipos abstractos, de la misma forma:

```
julia> abstract type Pointy{T} end
```

Con esta declaración, `Pointy{T}` es un tipo abstracto distinto para cada tipo o valor entero de `T`. Como con los tipos compuestos paramétricos, cada una de tales instancias es un subtipo de `Pointy`:

```
julia> Pointy{Int64} <: Pointy
true

julia> Pointy{1} <: Pointy
true
```

Los tipos abstractos paramétricos son invariantes, tal como los tipos compuestos paramétricos:

```
julia> Pointy{Float64} <: Pointy{Real}
false

julia> Pointy{Real} <: Pointy{Float64}
false
```

La notación `Pointy{<:Real}` puede usarse para expresar el análogo Julia de un tipo *covariante*, mientras que `Pointy{>:Int}` es el análogo de un tipo *contravariante*, pero técnicamente estos representan *conjuntos* de tipos (ver tipos UnionAll).

```
julia> Pointy{Float64} <: Pointy{<:Real}
true

julia> Pointy{Real} <: Pointy{>:Int}
true
```

De la misma manera que los tipos abstractos antiguos sirven para crear una jerarquía útil de tipos sobre tipos concretos, los tipos abstractos paramétricos tienen el mismo propósito con respecto a los tipos compuestos paramétricos. Podríamos, por ejemplo, haber declarado `Point {T}` ser un subtipo de `Pointy {T}` de la siguiente manera:

```
julia> struct Point{T} <: Pointy{T}
           x::T
           y::T
       end
```

Dada tal declaración, para cada elección de T tenemos `Point{T}` como un subtipo de `Pointy{T}`:

```
julia> Point{Float64} <: Pointy{Float64}
true

julia> Point{Real} <: Pointy{Real}
true

julia> Point{AbstractString} <: Pointy{AbstractString}
true
```

Esta relación es también invariante:

```
julia> Point{Float64} <: Pointy{Real}
false

julia> Point{Float64} <: Pointy{<:Real}
true
```

¿A qué propósito sirven los tipos abstractos paramétricos como `Pointy`? Considere si creamos una implementación tipo punto que sólo necesita una coordenada debido a que el punto se encuentra en la diagonal del primer cuadrante ($y = x$):

```
julia> struct DiagPoint{T} <: Pointy{T}
           x::T
       end
```

Ahora tanto `Point{Float64}` como `DiagPoint{Float64}` son implementaciones de la abstracción `Pointy{Float64}` y, similarmente, para cada otra posible elección del tipo T. Esto permite programar a un interfaz común compartido por todos los objetos `Pointy`, implementedo tanto por `Pointcomo` por `DiagPoint`. Esto no puede ser totalmente demostrado, sin embargo, hasta que no hayamos introducidos los métodos y el despacho en la siguiente sección, Methods.

Hay situaciones donde puede no tener sentido para los parámetros de tipo varíen libremente sobre todos los tipos posibles. En tales situaciones, uno puede restringir el rango de T como aquí:

```
julia> abstract type Pointy{T<:Real} end
```

Hay situaciones donde puede no tener sentido para los parámetros de tipo varíen libremente sobre todos los tipos posibles. En tales situaciones, uno puede restringir el rango de T como aquí:

```
julia> Pointy{Float64}
Pointy{Float64}

julia> Pointy{Real}
Pointy{Real}

julia> Pointy{AbstractString}
ERROR: TypeError: Pointy: in T, expected T<:Real, got Type{AbstractString}

julia> Pointy{1}
ERROR: TypeError: Pointy: in T, expected T<:Real, got Int64
```

Los parámetros de tipo para tipos compuestos paramétricos pueden ser restringidos de la misma manera:

```
struct Point{T<:Real} <: Pointy{T}
    x::T
    y::T
end
```

Para dar un ejemplo del mundo real de cómo toda esta maquinaria de tipos paramétricos puede ser útil, he aquí la definición actual del tipo inmutable `Rational` de Julia (omitiendo el constructor por simplicidad), que representa una relacíon exacta de enteros:

```
struct Rational{T<:Integer} <: Real
    num::T
    den::T
end
```

Sólo tiene sentido tomar relaciones de valores enteros, por lo que el tipo parametrizado T está restringido a ser un subtipo de `Integer`, y una razón de enteros representa un valor sobre la línea de los números reales, por lo que cualquier `Rational` es una instancia de la abstracción `Real`.

Tipos tupla

Las tuplas son una abstracción de los argumentos de una función (sin la propia función). Los aspectos salientes de los argumentos de una funcíon son su orden y sus tipos. Por tanto, un tipo tupla es muy similar a un tipo inmutable parametrizado donde cada parámetro es el tupo de un campo. Por ejemplo, un tipo tupla de dos elementos se parece al siguiente tipo inmutable:

```
struct Tuple2{A,B}
    a::A
    b::B
end
```

Sin embargo, hay tres diferencias clave:

- Los tipos tupla pueden tener cualquier número de parámetros.
- Los tipos tupla son *covariantes* en sus parámetros: `Tuple{Int}` es un subtipo of `Tuple{Any}`. Por tanto `Tuple{Any}` es considerado un tipo abstracto, y los tipos tupla son solo concretos si sus parámetros lo son.
- Las tuplas no tienen nombres de campo; los campos son sólo accedidos mediante índices.

Los valores tupla son escritos con paréntesis y comas. Cuando se construye una tupla, se genera un tipo tupla apropiado bajo demanda:

```
julia> typeof((1,"foo",2.5))
Tuple{Int64,String,Float64}
```

Note las implicaciones de covarianza:

```
julia> Tuple{Int,AbstractString} <: Tuple{Real,Any}
true

julia> Tuple{Int,AbstractString} <: Tuple{Real,Real}
false

julia> Tuple{Int,AbstractString} <: Tuple{Real,}
false
```

Intuitivamente, esto corresponde al tipo de los argumentos de una función, siendo un subtipo de la signatura de la función (cuando la signatura se corresponde).

Tipos Tupla Vararg

El último parámetro de un tipo tupla puede ser el tipo especial `Vararg`, que denota cualquier número de elementos arrastrados:

```
julia> mytupletype = Tuple{AbstractString,Vararg{Int}}
Tuple{AbstractString,Vararg{Int64,N} where N}

julia> isa(("1",), mytupletype)
true

julia> isa(("1",1), mytupletype)
true

julia> isa(("1",1,2), mytupletype)
true

julia> isa(("1",1,2,3.0), mytupletype)
false
```

Notese que `Varags{T}` corresponde a cero o más elementos del tipo `T`. Los tipos tupla vararg se usan para representar los argumentos aceptados por los métodos vararg (ver Funciones Vararg).

El tipo `Vararg{T,N}` se corresponde a exactamente `N` elementos de tipo `T`. `NTuple{N,T}` es un alias conveniente para `Tuple{Vararg{T,N}}`, es decir, un tipo tupla conteniendo exactamente `N` elementos de tipo `T`.

Tipos Singleton

Hay una clase especial de tipo paramétrico abstracto que hay que mencionar aquí: los tipos singleton. Para cada tipo `T` el tipo singleton `Type{T}` es un tipo abstracto cuya única instancia es el objeto `T`. Como la definición es un poco difícil de analizar, echemos un vistazo a los siguientes ejemplos:

```
julia> isa(Float64, Type{Float64})
true

julia> isa(Real, Type{Float64})
false

julia> isa(Real, Type{Real})
true

julia> isa(Float64, Type{Real})
false
```

En otras palabras, `isa(A,Type{B})` es `true` si y solo si `A` y `B` son el mismo objeto y este objeto es un tipo. Sin el parámetro, `Type` es simplemente un tipo abstracto que tiene como instancias todos los objetos tipo, incluyendo, por supuesto, los tipos singleton:

```
julia> isa(Type{Float64}, Type)
true

julia> isa(Float64, Type)
true

julia> isa(Real, Type)
true
```

Cualquier objeto que no es un tipo no es una instancia de `Type`:

```
julia> isa(1, Type)
false

julia> isa("foo", Type)
false
```

Hasta que discutamos los métodos paramétricos y las conversiones, es difícil explicar la utilidad de la construcción tipo singleton, pero abreviando, permite a uno especializar el comportamiento de una función sobre *valores* de un tipo específico. Esto es útil para escribir métodos (especialmente paramétricos) cuy ocomportamiento dependa de un tipo que es dado como un argumento explícito en lugar de implicado por el tipo de un o de sus argumentos.

Unos pocos lenguajes de programación tienen tipos singleton, incluyendo Haskell, Scala y Ruby. En uso general, el término ”tipo singleton” se refiere a un tipo cuya única instancia es un solo valor. Este significado se aplicaa a los tipos singleton de Julia, pero con la advertencia de que sólo los objetos tipo tienen tipos singleton.

Tipos primitivos paramétricos

Los tipos bits pueden ser declarados paramétricamente. Por ejemplo, los punteros son reprentados como tipos bits encajados que serían declarados en Julia de esta forma:

```
# 32-bit system:
primitive type Ptr{T} 32 end

# 64-bit system:
primitive type Ptr{T} 64 end
```

La característica ligeramente extraña de estas declaraciones comparadas con los tipos compuestos paramétricos típicos es que el parámetro de tipo `T` no se usa en la definición del propio tipo (es justo un *tag* abstracto, esencialmente definiendo una familia entera de tipos con idéntica estructura, diferenciada sólo por su parámetro de tipo. Por tanto `Ptr{Float}` y `Ptr{Int64}` son tipos distintos, incluso auntque ellos tengan representaciones idénticas. Y, por supuesto, todos los tipos puntero específicos son subtipo del tipo sombrilla `Ptr`:

```
julia> Ptr{Float64} <: Ptr
true

julia> Ptr{Int64} <: Ptr
true
```

14.9 Tipos UnionAll

Hemos dicho que un tipo paramétrico como `Ptr` actúa como un supertipo de todas sus instancias (`Ptr{Int64}` etc.). ¿Cómo funciona esto? `Ptr` en si mismo no puede ser un tipo normal, ya que sin saber el tipo de los datos referenciados el tipo claramente no puede ser usado para operaciones en memoria. La respuesta es que `Ptr` (u otros tipos paramétricos como `Array`) es una clase diferente de tipo llamado `UnionAll`. Tal tipo expresa la unión iterada de tipos para todos los valores de algún parámetro.

Los tipos `UnionAll`suelen ser escritos usando la palabra clave `where`. Por ejemplo, `Ptr` podría ser escrito de forma más exacta como `Ptr{T} where T`, lo que significa que todos los valores cuyo tipo es `Ptr{T}` para algún valor de `T`. En este contexto, el parámetro `T` suele llamarse también una "variable tipo", ya que es como una variable que se extiende sobre los tipos. Cada `where` introduce una sola variable tipo, por lo que estas expresiones están anidadas para tipos con múltiples parámetros, por ejemplo `Array{T,N} where N where T`.

La sintaxis de la aplicación de tipo `A{B, C}` requiere que `A` sea un tipo `UnionAll` y primero sustituye `B` por la variable de tipo más externa en `A`. Se espera que el resultado sea otro tipo `UnionAll` en el cuál `C` será sustituido. Por tanto, `A{B,C}` es equivalente a `A{B}{C}`. Esto explica por qué es posible instanciar parcialmente un tipo, como en `Array{Float64}`: El primer valor de parámetro ha sido fijado, pero el segundo aún se extiende sobre todos os posibles valores. Usando explícitamente la sintaxis `where`, cualquier subconjunto de parámetros puede ser fijado. Por ejemplo, el tipo de todos los arrays unidimensionales puede ser escrito como `Array{T,1} where T`.

Las variables de tipo pueden ser restringidas con relaciones de subtipos. `Array{T} where T<:Integer` se refiere a todos los arrays cuyo elemento de tipo es alguna clase de `Integer`. La sintaxis `Array{<:Integer}` es una abreviatura conveniente para `Array{T} where T<:Integer`. Las variables tipo pueden tener tanto límites superiores como inferiores. `Array{T} where Int<:T<:Number` se refiere a todos los arrays de `Number`s que son capaces de contener `Int`s (dado que `T` debe ser al menos tan grande como `Int`). La sintaxis `where T>:Int` también funciona para especificar sólo el límite inferior de una variable de tipo, y `Array{>:Int}` es equivalente a `Array{T} where T>:Int`.

En el caso de expresiones `where` anidadas, los límites de la variable de tipo pueden referirse a las variables de tipo más externas. Por ejemplo, `Tuple{T,Array{S}} where S<:AbstractArray{T} where T<:Real` se refiere a dos tuplas cuyo primer elemento es algo `Real` y cuyo segundo elemento es un `Array` de cualquier clase cuyo tipo de elemento contenga el tipo del primero elmento de la tupla.

La palabra clave `where` en si misma puede ser anidada dentro de una declaración más compleja. Por ejemplo, considere los dos tipos creados por las siguientes declaraciones:

```
julia> const T1 = Array{Array{T,1} where T, 1}
Array{Array{T,1} where T,1}

julia> const T2 = Array{Array{T,1}, 1} where T
Array{Array{T,1},1} where T
```

El tipo T1 define un array unidimensional de arrays unidimensionales; cada uno de los arrays internos consta de objetos del mismo tipo, pero este tipo puede variar de un array interno al siguiente. Por otra parte, el tipo T2 define un array unidimensional de arrays unidimensionales de manera que los arrays internos tienen todos el mismo tipo. Notese que T2 es un tipo abstracto, es decir, `Array{Array{Int,1},1} <: T2`, mientras que T1 es un tipo concreto. Como consecuencia, T1 puede ser construido con un constructor de cero argumentos `a=T1()` pero T2 no puede.

Hay una sintaxis conveniente para nombrar tales tipos, similar a la forma corta de la sintaxis de definición de función:

```
Vector{T} = Array{T,1}
```

Esto es equivalente a `const Vector = Array{T,1} where T`. Escribir `Vector{Float64}` es equivalente a escribir `Array{Float64,1}`, y el tipo paraguas `Vector` tiene como instancias todos los objetos `Array` donde el segundo parámetro (el número de dimensiones del array) es uno, sin importar cuál es el tipo del elemento. En lenguajes donde los tipos paramétricos deben siempre ser especificados por completo, esto no suele ser de ayuda, pero en Julia, esto permite a uno escribir justo `Vector` para el tipo abstract incluyendo arrays densos unidimensionales de cualquier tipo de elementos.

14.10 Aliases de Tipos

Algunas veces es conveniente introducir un nuevo nombre para un tipo ya expresable. Para tales ocasiones, Julia proporciona el mecanismo `typealias`. Por ejemplo, `UInt` es un alias de `UInt32` o `UInt64` dependiendo de los punteros de tamaño del sistema:

```
# 32-bit system:
julia> UInt
UInt32

# 64-bit system:
julia> UInt
UInt64
```

Esto se consigue via el siguiente código en `base/boot.jl`:

```
if Int === Int64
    const UInt = UInt64
else
    const UInt = UInt32
end
```

Por supuesto, esto depende de a qué representa el alias `Int` si a `Int32` o a `Int64`.

(Note que, a diferencia de `Int`, `Float` no existe como un alias de tipo para un tamaño específico de `AbstractFloat`. A diferencia de con los registros enteros, los tamaños de los registros en punto flotante están especificados por el estándar IEEE-754 standard. Mientras que el tamaño de `Int` refleja el tamaño de un puntero nativo de esta máquina.)

14.11 Operaciones sobre tipos

Como los tipos en Julia son objetos en sí mismos, las funcoines ordinarias pueden operar sobre ellos. Algunas funciones que son particularmente útiles para trabajar con o explorar tipos han sido ya introducidas. Por ejemplo, el operador `<:` que indica si el operando a su izquierda es un subtipo del operando a su derecha.

La función `isa` comprueba si un objeto es de un tipo dado y devuelve `true` o `false`:

```
julia> isa(1, Int)
true

julia> isa(1, AbstractFloat)
false
```

La función `typeof()` , ya usada a través del manual en ejemplos, devuelve el tupo de su argumento. Como, con se notó anteriormente, los tipos son objetos, ellos también tienen tipos, y podemos preguntar cuáles son sus tipos:

```
julia> typeof(Rational{Int})
DataType

julia> typeof(Union{Real,Float64,Rational})
DataType

julia> typeof(Union{Real,String})
Union
```

¿Qué pasa si repetimos el proceso? ¿Cuál es el tipo de un tipo de un tipo? Como sucede, los tipos son todos valores compuestos y por tanto todos tendrán un tipo de `DataType`:

```
julia> typeof(DataType)
DataType

julia> typeof(Union)
DataType
```

`DataType` is its own type.

Otra operación que se aplica a algunos tipos es `supertype()`, que revela el supertipo de un tipo. Sólo los tipos declarados (`DataType`) tienen supertipos no ambiguos:

```
julia> supertype(Float64)
AbstractFloat

julia> supertype(Number)
Any

julia> supertype(AbstractString)
Any

julia> supertype(Any)
Any
```

Si aplicamos `supertype()` a otros objetos tipo (u objetos no tipo) se lanzará un `MethodError`:

```
julia> supertype(Union{Float64,Int64})
ERROR: MethodError: no method matching supertype(::Type{Union{Float64, Int64}})
Closest candidates are:
  supertype(!Matched::DataType) at operators.jl:41
  supertype(!Matched::UnionAll) at operators.jl:46
```

14.12 Custom pretty-printing

Frecuentemente, uno quiere personalizar cómo se mostrarán las instancias de un tipo. Esto se consigue sobrecargando la función `show()`. Por ejemplo, supongamos que definimos un tipo para representar número complejos en forma polar:

```
julia> struct Polar{T<:Real} <: Number
           r::T
           θ::T
       end

julia> Polar(r::Real,θ::Real) = Polar(promote(r,θ)...)
Polar
```

Aquí, hemos añadido una función constructor personalizado para que pueda tomar argumentos de distinto tipos `Real` y los promocione a un tipo común (ver Constructores y Conversión y Promoción). (Por supuesto, tendríamos que definir montones de otros métodos también, para hacer que actíe como un `Number`, por ejemplo, `+`, `*`, one, `zero`, reglas de promoción y otras cosas). Por defecto, las instancias de este tipo se muestran de forma bastante simple, con información sobre el nombre del tipo y los valores de los campos, como por ejemplo en `Polar{Float64}(3.0,4.0)`.

Si en lugar de usar este modo de presentación preferimos que su presentación sea `3.0 * exp(4.0im)`, hay que definir el siguiente método para que imprima el objeto a un objeto de salida `io`dado (que representa un fichero, terminal, buffer, etc.; ver Networking and Streams):

```
julia> Base.show(io::IO, z::Polar) = print(io, z.r, " * exp(", z.θ, "im)")
```

Es posible un control de grano más fino sobre la visualizacin de los objetos `Polar`. En particular, algunas veces uno desea un formato de impresión detallado multilínea, utilizado para mostrar un solo objeto en REPL y otros entornos interactivos, y también un formato de línea única más compacto utilizado para [`print()`] @ref) o para mostrar el objeto como parte de otro objeto (por ejemplo, en una matriz). Aunque de forma predeterminada se llama a la función `show(io, z)` en ambos casos, puede definir un formato multilínea *diferente* para mostrar un objeto sobrecargando una forma de tres argumentos de show que toma el tipo MIME `text/plain` como su segundo argumento (consulte E/S multimedia), por ejemplo:

```
julia> Base.show{T}(io::IO, ::MIME"text/plain", z::Polar{T}) =
           print(io, "Polar{$T} complex number:\n   ", z)
```

(Note que `print(..., z)` aquí invocará al método con dos argumentos `show(io, z)`). Esto dará como resultado:

```
julia> Polar(3, 4.0)
Polar{Float64} complex number:
   3.0 * exp(4.0im)

julia> [Polar(3, 4.0), Polar(4.0,5.3)]
2-element Array{Polar{Float64},1}:
 3.0 * exp(4.0im)
 4.0 * exp(5.3im)
```

donde se sigue utilizando la forma de línea `show(io, z)` para un array de valores `Polar`. Técnicamente, el REPL llama a `display(z)` para mostrar el resultado de ejecutar una línea que por defecto es `show (STDOUT, MIME("text/-plain"), z)`, que a su vez por defecto es `show(STDOUT, z)`, pero debe *no* definir nuevos métodos `display()` a menos que esté definiendo un nuevo controlador de pantalla multimedia (consulte E/S multimedia).

Además, también puede definir métodos show para otros tipos MIME para permitir una visualización más rica (HTML, imágenes, etc.) de los objetos en entornos que lo admitan (por ejemplo, IJulia). Por ejemplo, podemos definir la visualización HTML formateada de objetos `Polar`, con superíndices y cursiva, a través de:

```
julia> Base.show{T}(io::IO, ::MIME"text/html", z::Polar{T}) =
           println(io, "<code>Polar{$T}</code> complex number: ",
                   z.r, " <i>e</i><sup>", z0., " <i>i</i></sup>")
```

Un objeto `Polar` se mostrará entonces automáticament usando HtML en un entorno que soporte pantallas HTML, pero podemos llamar a la función show manualmente para obtener una salida HTML si lo deseamos:

```
julia> show(STDOUT, "text/html", Polar(3.0,4.0))
<code>Polar{Float64}</code> complex number: 3.0 <i>e</i><sup>4.0 <i>i</i></sup>
```

14.13 "Valores tipo"

En Julia uno no puede despachar sobre un *valor* tal como `true` o `false`. Sin embargo, se se puede despachar sobre tipos paramétricos, y Julia permite incluir valores "plain bits" (tipos, símbolos, enteros, números en punto flotante, tuplas, etc.) como parámetros de tipo. Un ejemplo común es el parámetro de dimensionalidad en `Array{T,N}`, donde `T` es un tipo (por ejemplo, `Float64`) pero `N` es un `Int`.

Podemos crear nuestros propio tipos personalizados que tomen valores como parámetros, y usarlos para controlar el despacho de los tipos personalizados. A modo de ilustración de esta idea, introduzcamos un tipo paramétrico, `Val{T}` que sirve como una forma tradicional de explotar esta técnica para casos donde tu no necesitas una jerarquía más elaborada.

`Val` es definida como:

```
julia> struct Val{T}
       end
```

No hay más implementaciones de `Val` que esta. Algunas funciones en la librería estándar de Julia aceptan los tipos `Val` como argumentos, y uno también puede usarlos para escribir sus propias funciones. Por ejemplo:

```
julia> firstlast(::Type{Val{true}}) = "First"
firstlast (generic function with 1 method)

julia> firstlast(::Type{Val{false}}) = "Last"
firstlast (generic function with 2 methods)

julia> firstlast(Val{true})
"First"

julia> firstlast(Val{false})
"Last"
```

Por consistencia con Julia, el sitio de llamada debería siempre pasar un tipo `Val` en lugar de crear una instancia, por ejemplo, usar `foo(Val{:bar})` en lugar de `foo(Val{:bar}())`.

Vale la pena señalar que es extremadamente sencillo usar mal los tipos "valor" paramétricos, incluyendo `Val`; en caso desfavorables, tu puedes fácilmente acabar haciendo el rendimiento de tu código mucho *peor*. En particular, tu nunca tendrás que querer escribir código actual como ilustramos anteriormente. Para má información sobre el uso apropiado de `Val` consulte por fabor la discusión más extensa en los consejos de rendimiento.

14.14 Tipos `Nullable`: representando valores perdidos

En muchas situaciones, uno necesita interactuar con un valor de tipo `T` que puede o no existir. Para manejar estas situaciones, Julia proporciona un tipo paramétrico denominado `Nullable{T}` que puede ser pensado como un tipo contenedor especializado que puede contener cero o un valores. `Nullable{T}` proporciona una interfaz mínima

diseñada para asegurar qué interacciones con valores perdidos son seguras. En la actualidad, la interfaz consiste en varias posibles interacciones:

- Construir un objeto `Nullable`.
- Comprobar si un objeto `Nullable` tiene un valor perdido.
- Acceder al valor de un objeto `Nullable` con la garantía de que se lanzará una `NullException` si el valor del objeto está perdido.
- Acceder al valor de un objeto `Nullable` con una garantía de que el valor por defecto del tipo T será devuelto si el valor del objeto se pierde.
- Realizar una operación sobre el valor (si existe) de un objeto `Nullable`, obteniendo un resultado`Nullable`. El resultado faltará si falta el valor original.
- Realizar una prueba sobre el valor (si existe) de un objeto `Nullable`, obteniendo un resultado que será perdido si faltaba el`Nullable` o si la prueba falla.
- Realizar operaciones generales en objetos individuales `Nullable`, propagando los datos faltantes.

Construyendo objetos `Nullable`

Para construir un objeto representando un valor perdido de tipo T, use la siguiente función `Nullable{T}()`:

```
julia> x1 = Nullable{Int64}()
Nullable{Int64}()

julia> x2 = Nullable{Float64}()
Nullable{Float64}()

julia> x3 = Nullable{Vector{Int64}}()
Nullable{Array{Int64,1}}()
```

Para construir un objeto representando un valor no perdido de tipo T, use la función `Nullable(x::T)`:

```
julia> x1 = Nullable(1)
Nullable{Int64}(1)

julia> x2 = Nullable(1.0)
Nullable{Float64}(1.0)

julia> x3 = Nullable([1, 2, 3])
Nullable{Array{Int64,1}}([1, 2, 3])
```

Note que la distinción clave entre estas dos formas de construir un objeto `Nullable`: en un estilo, tu proporcionas un tipo, T como un parámetro a función; en el otro estilo, tu proporcionas un solo valor de tipo T como argumento.

Comprobar si un objeto `Nullable` tiene un valor

Puedes comprobbar si un objeto `Nullable` tiene algún valor usando `isnull()`:

```
julia> isnull(Nullable{Float64}())
true

julia> isnull(Nullable(0.0))
false
```

Acceder de forma segura al valor de un objeto `Nullable`

Puedes acceder al valor de un objeto `Nullable` usando `get()`:

```
julia> get(Nullable{Float64}())
ERROR: NullException()
Stacktrace:
 [1] get(::Nullable{Float64}) at ./nullable.jl:92

julia> get(Nullable(1.0))
1.0
```

Si el valor no está presente, como podría ser para un `Nullable{Float64}` se lanzará un error `NullException`. La naturaleza del error lanzado de la función `get()` asegur aque cualquier intento de acceder al valor perdido falle inmediatamente.

En los casos para los cuales existe un valor por defecto razonable que podría ser usando cuando el valor de los objetos `Nullable` se volviera perdido, uno puede proporcionar este valos por defecto como un segundo argumento a `get()`:

```
julia> get(Nullable{Float64}(), 0.0)
0.0

julia> get(Nullable(1.0), 0.0)
1.0
```

> **Tip**
>
> Asegúrese de que el tipo de valor predeterminado pasado a `get()` y el del objeto `Nullable` coincidan para evitar la inestabilidad de tipo, lo que podría perjudicar el rendimiento. Utilice `convert()` manualmente si es necesario.

Realizando operaciones sobre objetos `Nullable`

Los objetos `Nullable` representan valores que están posiblemente perdidos, y es posible escribir todo el código usando estos objetos primero comprobando para ver si el valor está perdido con `isnull()`, y luego realizar la accin apropiada. Sin embargo, hay algunos casos de uso comunes donde el código podría ser ms conciso o claro usando una función de orden superior.

La función `map` toma como argumentos una función `f` y un valor `x` de tipo `Nullable`. Ella produce un `Nullable`:

- Si `x` es un valor perdido, entonces produce un valor perdido;
- Si `x` tiene un valor, entonces produce un objeto `Nullable` que contiene `f(get(x))` como valor.

Esto es útil para realizar operaciones simples sobre valores que podrían estar perdidos si el comportamiento deseado es simplemente propagar hacia adelante los valores perdidos.

La función `filter` toma como argumentos una función predicado `p` (es decir, una función que devuelve un boolean) y un valor `x` de tipo `Nullable`. Ella produce un `Nullable`:

- Si `x` es un valor perdido, entonces produce un valor perdido;
- Si `p(get(x))` es true, entoces produce el valor original `x`;

- Si `p(get(x))` es false, entonces produce un valor perdido.

De esta forma, `filter` puede ser considerado como seleccionar sólo valores permisibles, y convertir valores no permisibles en valores perdidos.

Mientras que `map` y `filter` son útiles para casos específicos, la función de orden superior más útil es, con diferencia, `broadcast`, que puede manejar una amplia variedad de casos, incluyendo hacer operaciones existentes funcionen y propaguen `Nullables`. El siguiente ejemplo motivará la necesidad de `broadcast`. Supongamos que tenemos una función que calcula la mayor de las dos raíces reales de una ecuación cuadrática, usando la formula cuadrática:

```
julia> root(a::Real, b::Real, c::Real) = (-b + √(b^2 - 4a*c)) / 2a
root (generic function with 1 method)
```

Podemos verificar que el resultado de `root(1, -9, 20)` es `5.0` como esperamos, ya que `5.0` es la mayor de dos raíces reales de la ecuación cuadrática.

Supongamos ahora que queremos encontrar la mayor raíz real de una ecuación cuadrática donde los coeficientes pueden ser valores perdidos. Tener valores perdidos en los conjuntos de datos es una ocurrencia común en los datos del mundo real, por lo que es importante poder tratar con ellos. Pero no podemos encontrar las raíces de una ecuación si no conocemos todos los coeficientes. La mejor solución para esto dependerá del caso de uso particular; quizás deberíamos arrojar un error. Sin embargo, para este ejemplo, asumiremos que la mejor solución es propagar los valores perdidos; es decir, si falta alguna entrada, simplemente producimos una salida faltante.

La función `broadcast()` facilita esta tarea; simplemente podemos pasar la función `root` que escribimos a`broadcast`:

```
julia> broadcast(root, Nullable(1), Nullable(-9), Nullable(20))
Nullable{Float64}(5.0)

julia> broadcast(root, Nullable(1), Nullable{Int}(), Nullable{Int}())
Nullable{Float64}()

julia> broadcast(root, Nullable{Int}(), Nullable(-9), Nullable(20))
Nullable{Float64}()
```

Si faltan una o más de las entradas, faltará la salida de `broadcast()`.

Existe un convenio sintáctico especial para la función `broadcast()` usando la notación punto:

```
julia> root.(Nullable(1), Nullable(-9), Nullable(20))
Nullable{Float64}(5.0)
```

En particular, los operadores aritméticos regulares pueden ser `broadcast ()` convenientemente usando operadores `.`-prefijo:

```
julia> Nullable(2) ./ Nullable(3) .+ Nullable(1.0)
Nullable{Float64}(1.66667)
```

Chapter 15

Métodos

Recordemos de la sección Funciones que una función es un objeto que establece una correspondencia entre una tupla de argumentos y un valor de retorno o lanza una excepción si no puede devolverse el valor apropiado. Para la misma función conceptual es común soy implementada de una forma muy diferente para tipos de argumentos diferentes: sumar dos enteros es distinto de sumar dos valores en punto flotante y ambos son distintos de sumar un entero y 1 punto flotante. A pesar de las diferencias de implementación, éstas operaciones caen todas bajo el concepto general de ”suma”. En consecuencia, en Julia, estos comportamientos pertenecen todos a un solo objeto: la función +.

Para facilitar el uso de muchas implementaciones distintas del mismo concepto suavemente, las funciones necesitan no ser definidas de una vez, sino poder ser definidas a trozos proporcionando comportamientos distintos para ciertas combinaciones de tipos de argumentos y cuentas. Llamamos *método* a la definición de un posible comportamiento para una función. Hasta ahora sólo se han presentado ejemplos de funciones definidas como sólo método, aplicables a todo tipo de argumentos. Sin embargo, las asignaturas de las definiciones de los métodos pueden anotarse para indicar los tipos de los argumentos además de su número, y puede proporcionarse más de una sola definición de método. Cuando una función se aplica a una dupla de argumentos particular, se aplica el método más específico y aplicable a esos argumentos. Por tanto, el comportamiento global de una función es un collage de los comportamientos de sus distintas definiciones de métodos. Si el collage está bien diseñado, incluso aunque las implementaciones de los métodos puedan ser bastante diferentes, el comportamiento exterior de la función pareciera contínuo y consistente.

La elección de qué método ejecutar cuando se aplica una función se llama *despacho*. Julia permite al proceso de despacho elegir qué método de una función llamar basándose en el número de argumentos y en los tipos de todos los argumentos dados a la función. Este mecanismo es diferente al que ocurre en los lenguajes orientados al objeto tradicionales, donde el despacho se basa solo en el primer argumento, que frecuentemente tiene una sintaxis especial, y que es muchas veces implicado el lugar de ser escrito explícitamente como argumento. [1] Usar todos los argumentos de la función para elegir qué método debería ser invocado es conocido como despacho múltiple. El despacho múltiple es particularmente útil para código matemático, donde tiene poco sentido considerar que las operaciones pertenecen a un argumento más que los demás. Más allá de las operaciones matemáticas, sin embargo, el despacho múltiple ha resultado ser un paradigma potente y conveniente para estructurar y organizar los programas.

15.1 Definiendo Métodos

En los ejemplos estudiados hasta ahora, sólo se han definido funciones con un único método que tienen argumentos con los tipos no restringidos. Estas funciones se comporta como las que hay en lenguajes con tipos dinámicos tradicionales. Sin embargo, también se han usado despacho múltiple y métodos sin ser consciente de ello: todas las

[1] En C++ o Java, por ejemplo, en una llamada a un método como `obj.meth(arg1,arg2)`, el objeto obj ”recibe” la llamada al método y es pasado implícitamente vía la palabra clave `this`, en lugar de con un argumento de método explícito. Cuando el objeto `this` actual es el receptor de una llamada a método él puede ser omitido, escribiendo justo `meth(arg1,arg2)`, con `this` implícito como objeto receptor.

funciones estándar y operadores de Julia, tal como función +, tiene muchos métodos que definen su comportamiento sobre varias combinaciones posibles número y tipo de argumentos.

Cuando se define una función, uno puede opcionalmente restringir los tipos de los parámetros sobre los que se aplica usando el operador de la selección de tipos `::`, introducido en la sección Tipos compuestos:

```
julia> f(x::Float64, y::Float64) = 2x + y
f (generic function with 1 method)
```

Esta definición de función se aplica sólo a llamadas en las que x e y sean ambos valores del tipo `Float64`:

```
julia> f(2.0, 3.0)
7.0
```

Aplicar esta definición a otros tipos de argumentos dará como resultado un `MethodError`:

```
julia> f(2.0, 3)
ERROR: MethodError: no method matching f(::Float64, ::Int64)
Closest candidates are:
  f(::Float64, !Matched::Float64) at none:1

julia> f(Float32(2.0), 3.0)
ERROR: MethodError: no method matching f(::Float32, ::Float64)
Closest candidates are:
  f(!Matched::Float64, ::Float64) at none:1

julia> f(2.0, "3.0")
ERROR: MethodError: no method matching f(::Float64, ::String)
Closest candidates are:
  f(::Float64, !Matched::Float64) at none:1

julia> f("2.0", "3.0")
ERROR: MethodError: no method matching f(::String, ::String)
```

Como puede comprobarse, los argumentos tienen que ser exactamente del tipo `Float64`. Otros tipos numéricos tales como `Float32` o `Int` no serán convertidos automáticamente en `Float64`. Algo parecido sucede con los datos `String`. Como el tipo `Float64` es un tipo concreto y los tipos concretos no pueden tener subclases en Julia, esta definición sólo puede aplicarse a argumentos que sean exactamente del tipo `Float64`. Esto puede ser útil en ocasiones, sin embargo, para escribir métodos más generales se hace uso de parámetros cuyos tipos sean abstractos:

```
julia> f(x::Number, y::Number) = 2x - y
f (generic function with 2 methods)

julia> f(2.0, 3)
1.0
```

Esta definición de método se aplica a cualquier par de argumentos que sean instancias de `Number`. Ellas no tienen que ser del mismo tipo, mientras que ambas sean valores numéricos. El problema de manejar tipos numéricos dispares se delega a las operaciones aritméticas en la expresión `2x - y`.

Para definir una función con múltiples métodos, uno simplemente define la función varías veces, con diferentes números de argumentos y tipos. La primera definición de método para la función crea el objeto función y las definiciones de métodos subsecuentes añaden nuevos métodos al objeto función existente. La definición de método más específica que case con el número los tipos de argumentos será la ejecutada cuando se aplique la función. Por tanto, las dos definiciones de métodos anteriores, considerados juntas, define el comportamiento de la función `f` sobre todos los padres de instancias del tipo abstracto `Number` (pero con un comportamiento específico para pares de valores `Float64`. Si uno de los argumentos es un valor en punto flotante de 64 bits, pero el otro no lo es, entonces el método `f(Float64,Float64)` no puede ser invocado y se utilizará el método más general `f(Number,Number)`:

```
julia> f(2.0, 3.0)
7.0

julia> f(2, 3.0)
1.0

julia> f(2.0, 3)
1.0

julia> f(2, 3)
1
```

La definición 2x+y sólo se usa en el primer caso, mientras que la definición 2x-y se usa en los demás. Nunca se realiza conversión automática en los otros: todas las conversiones son no mágicas y completamente exlícitas. En la sección Conversión y promoción, sin embargo, se muestra cómo las aplicaciones inteligentes de tecnología suficientemente avanzada pueden ser indistinguibles de la magia. [2]

Para valores no numéricos, ,y para menores de dos argumentos, la función `f` permanece indefinida, y aplicándola se obtendrá como resultado un `MethodError`:

```
julia> f("foo", 3)
ERROR: MethodError: no method matching f(::String, ::Int64)
Closest candidates are:
  f(!Matched::Number, ::Number) at none:1

julia> f()
ERROR: MethodError: no method matching f()
Closest candidates are:
  f(!Matched::Float64, !Matched::Float64) at none:1
  f(!Matched::Number, !Matched::Number) at none:1
```

Puedes ver fácilmente que métodos existen para una función entrando el propio nombre del objeto en una sesión interactiva:

```
julia> f
f (generic function with 2 methods)
```

La salida nos dice que `f` es un objeto función con dos métodos. Para encontrar cuáles son las signaturas de esos métodos, utilizaremos la función `methods()`:

```
julia> methods(f)
# 2 methods for generic function "f":
f(x::Float64, y::Float64) in Main at none:1
f(x::Number, y::Number) in Main at none:1
```

que muestra que `f` tiene dos métodos: uno que toma dos argumentos `Float64` y una que toma dos argumentos de tipo `Number`. También indica el fichero y el número de línea donde los métodos fueron definidos aunque, si los métodos fueron definidos en el REPL, se obtendrá none:1.

En ausencia de una declaración de tipo con `::` el tipo de un parámetro de un método es `Any` por defecto, lo que significa que está sin restricciones ya que todos los valores en Julia son instancias del tipo abstracto `Any`. Por tanto, podemos definir un método atrapatodo para `f` tal como:

```
julia> f(x,y) = println("Whoa there, Nelly.")
f (generic function with 3 methods)

julia> f("foo", 1)
Whoa there, Nelly.
```

Este atrapatodo es menos específico que cualquier otra posible definición de método para un par de valores de parámetros, por lo que sólo será llamada sobre pares de argumentos a los cuales no pueda aplicarse otra definición de método.

Aunque parece un concepto simple, el despacho múltiple sobre los tipos de valores es quizás la característica más potente y central del lenguaje Julia. Las operaciones del núcleo tienen típicamente docenas de metodos:

```
julia> methods(+)
# 180 methods for generic function "+":
+(x::Bool, z::Complex{Bool}) in Base at complex.jl:224
+(x::Bool, y::Bool) in Base at bool.jl:89
+(x::Bool) in Base at bool.jl:86
+(x::Bool, y::T) where T<:AbstractFloat in Base at bool.jl:96
+(x::Bool, z::Complex) in Base at complex.jl:231
+(a::Float16, b::Float16) in Base at float.jl:372
+(x::Float32, y::Float32) in Base at float.jl:374
+(x::Float64, y::Float64) in Base at float.jl:375
+(z::Complex{Bool}, x::Bool) in Base at complex.jl:225
+(z::Complex{Bool}, x::Real) in Base at complex.jl:239
+(x::Char, y::Integer) in Base at char.jl:40
+(c::BigInt, x::BigFloat) in Base.MPFR at mpfr.jl:303
+(a::BigInt, b::BigInt, c::BigInt, d::BigInt, e::BigInt) in Base.GMP at gmp.jl:303
+(a::BigInt, b::BigInt, c::BigInt, d::BigInt) in Base.GMP at gmp.jl:296
+(a::BigInt, b::BigInt, c::BigInt) in Base.GMP at gmp.jl:290
+(x::BigInt, y::BigInt) in Base.GMP at gmp.jl:258
+(x::BigInt, c::Union{UInt16, UInt32, UInt64, UInt8}) in Base.GMP at gmp.jl:315
...
+(a, b, c, xs...) at operators.jl:119
```

El despacho múltiple junto con el sistema de tipos paramétrico flexible dan a Julia su capacidad para expresar de forma abstract algoritmos de alto nivel desacoplados de los detalles de implementación, generando aún código eficiente y especializado para manejar cada caso en tiempo de ejecución.

15.2 Ambigüedades de Métodos

Es posible definir un conjunto de métodos de función tales que no haya un método único más específico aplicable a alguna combinación de argumentos:

```
julia> g(x::Float64, y) = 2x + y
g (generic function with 1 method)

julia> g(x, y::Float64) = x + 2y
g (generic function with 2 methods)

julia> g(2.0, 3)
7.0

julia> g(2, 3.0)
8.0

julia> g(2.0, 3.0)
ERROR: MethodError: g(::Float64, ::Float64) is ambiguous.
[...]
```

Aquí, la llamada `g(2.0, 3.0)` podría ser manejada por los métodos `g(Float64, Any)` o `g(Any, Float64)` y ninguno es más específico que el otro. En tales casos, Julia lanza un `MethodError` en lugar de elegir uno de los métodos arbitrariamente. Podemos obviar las ambigüedades de los métodos especificando un método apropiado para el caso intersección:

```
julia> g(x::Float64, y::Float64) = 2x + 2y
g (generic function with 3 methods)

julia> g(2.0, 3)
7.0

julia> g(2, 3.0)
8.0

julia> g(2.0, 3.0)
10.0
```

Se recomienda que el método que suprime la ambigüedad sea definido primero, ya que en caso contrario la ambigüedad existe, transitoriamente, hasta que el método más especifico sea definido.

En casos ms complejos, resolver ambigüedades de métodos implica un cierto elemento de diseño; este tema se explorará posteriormente.

15.3 Métodos paramétricos

Las definiciones de métodos pueden tener, opcionalmente, parámetros de tipo cualificando la signatura:

```
julia> same_type(x::T, y::T) where {T} = true
same_type (generic function with 1 method)

julia> same_type(x,y) = false
same_type (generic function with 2 methods)
```

El primer método se aplica cuando ambos argumentos son del mismo tipo concreto, independientemente del tipo que sea, mientras que el segundo actúa como un atrapatodo, cubriendo todos los demás casos. Por tanto, en conjunto, esto define una función booleana que comprueba si dos argumentos son del mismo tipo:

```
julia> same_type(1, 2)
true

julia> same_type(1, 2.0)
false

julia> same_type(1.0, 2.0)
true

julia> same_type("foo", 2.0)
false

julia> same_type("foo", "bar")
true

julia> same_type(Int32(1), Int64(2))
false
```

Tales definiciones corresponden a métodos cuyas signaturas de tipo son tipos `UnionAll` (ver tipos UnionAll).

Esta clase de definición del comportamiento de una función mediante despacho es bastante común (incluso idiomático) en Julia. Los métodos con parámetros de tipo no están restringidos a ser usados como los tipos de los parámetros: ellos pueden ser usados en cualquier parte donde un palo estaría en la signatura de la función o cuerpo de la función. He aquí un eemplo donde el parámetro de tipo del método T se sa como el parámetro de tipo al tipo paramétrico `Vector{T}` en la signatura del método:

```
julia> myappend(v::Vector{T}, x::T) where {T} = [v..., x]
myappend (generic function with 1 method)

julia> myappend([1,2,3],4)
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> myappend([1,2,3],2.5)
ERROR: MethodError: no method matching myappend(::Array{Int64,1}, ::Float64)
Closest candidates are:
  myappend(::Array{T,1}, !Matched::T) where T at none:1

julia> myappend([1.0,2.0,3.0],4.0)
4-element Array{Float64,1}:
 1.0
 2.0
 3.0
 4.0

julia> myappend([1.0,2.0,3.0],4)
ERROR: MethodError: no method matching myappend(::Array{Float64,1}, ::Int64)
Closest candidates are:
  myappend(::Array{T,1}, !Matched::T) where T at none:1
```

Como puedes ver, el tipo del elemento añadido tiene que corresponderse con el tipo de elemento del vector al que se está añadiendo, o se lanzará un `MethodError`. En el siguiente ejemplo, el parámetro de tipo del método T se usa como valor de retorno:

```
julia> mytypeof(x::T) where {T} = T
mytypeof (generic function with 1 method)

julia> mytypeof(1)
Int64

julia> mytypeof(1.0)
Float64
```

Así como puedes poner restricciones de subtipo para los parámetros de tipo en declaraciones de tipo (ver Tipos Paramétricos) también puedes restringir los parámetros de tipo de los métodos:

```
julia> same_type_numeric(x::T, y::T) where {T<:Number} = true
same_type_numeric (generic function with 1 method)

julia> same_type_numeric(x::Number, y::Number) = false
```

```
same_type_numeric (generic function with 2 methods)

julia> same_type_numeric(1, 2)
true

julia> same_type_numeric(1, 2.0)
false

julia> same_type_numeric(1.0, 2.0)
true

julia> same_type_numeric("foo", 2.0)
ERROR: MethodError: no method matching same_type_numeric(::String, ::Float64)
Closest candidates are:
  same_type_numeric(!Matched::T<:Number, ::T<:Number) where T<:Number at none:1
  same_type_numeric(!Matched::Number, ::Number) at none:1

julia> same_type_numeric("foo", "bar")
ERROR: MethodError: no method matching same_type_numeric(::String, ::String)

julia> same_type_numeric(Int32(1), Int64(2))
false
```

La función `same_type_numeric` se comporta como la función `same_type` descrita antes, pero sólo está definida para pares de números.

Los métodos paramétricos permiten la misma sintaxis que las expresiones `where` usadas para escribir tipos (ver tipos UnionAll).

Si hay un único parámetro, las llaves que lo encierran (en `where {T}`) pueden ser omitidas, aunque suele ser preferible mantenerlas por claridad.

Los parámetros múltiples pueden ser separados con comas, por ejemplo `where {T, S<:Real}`, o escritos usando `where` anidados, por ejemplo, `where S<:Real where T`.

15.4 Redefiniendo Métodos

Cuando se redefine un método o se añaden nuevos métodos, es importante comprender que estos cambios no tienen efecto inmediatamente. Esto es clave para la capacidad de Julia para inferir estáticamente y compilar código para ejecutar rápido, sin los trucos de JIT y sobrecargas usuales. De hecho, cualquier nueva definición de método no será visible al entorno de ejecución actual, incluyendo tareas e hilos (y cualquier otra función definida con `@generated`). Comencemos con un ejemplo para ver qué significa esto:

```
julia> function tryeval()

           @eval newfun() = 1

           newfun()

       end
tryeval (generic function with 1 method)

julia> tryeval()
ERROR: MethodError: no method matching newfun()
The applicable method may be too new: running in world age xxxx1, while current world is xxxx2.
```

```
Closest candidates are:
  newfun() at none:1 (method too new to be called from this world context.)
 in tryeval() at none:1
 ...

julia> newfun()
1
```

En este ejemplo, observe que la nueva definición para `newfun` ha sido creada, pero no puede ser llamada inmediatamente. El nuevo global es inmediatamente visible para la función `tryeval`, para que pueda escribir `return newfun` (sin paréntesis). ¡Pero ni usted ni ninguna de las personas que llama, ni las funciones a las que llama, etc. puede llamar a esta nueva definición de método!

Pero hay una excepción: las llamadas futuras a `newfun` *del REPL* funcionan como se esperaba, pudiendo tanto ver como invocar la nueva definición de`newfun`. Sin embargo, las futuras llamadas a `tryeval` continuarán viendo la definición de`newfun` tal como era *en la instrucción anterior en REPL*, y por lo tanto antes de esa llamada a `tryeval`.

Es posible que desee probar esto para ver cómo funciona.

La implementación de este comportamiento es un "contador de edad mundial". Este valor monótonamente creciente rastrea cada operación de definición de método. Esto permite describir "el conjunto de definiciones de métodos visibles para un entorno de tiempo de ejecución dado" como un solo número, o "edad mundial". También permite comparar los métodos disponibles en dos mundos simplemente comparando su valor ordinal. En el ejemplo anterior, vemos que el "mundo actual" (en el que existe el método `newfun()`) es uno mayor que el "mundo de tiempo de ejecución" local de la tarea que se corrigió cuando se inició la ejecución de `tryeval`.

A veces es necesario evitar esto (por ejemplo, si está implementando el REPL anterior). Afortunadamente, hay una solución fácil: llamar a la función usando `Base.invokelatest`:

```
julia> function tryeval2()

           @eval newfun2() = 2

           Base.invokelatest(newfun2)

       end
tryeval2 (generic function with 1 method)

julia> tryeval2()
2
```

Por último, echemos un vistazo a algunos ejemplos más complejos donde esta regla se pone en funcionamiento. Definamos una función `f(x)`, que inicialmente tiene un método:

```
julia> f(x) = "original definition"
f (generic function with 1 method)
```

Iniciamos algunas operaciones que usan `f(x)`:

```
julia> g(x) = f(x)
g (generic function with 1 method)

julia> t = @async f(wait()); yield();
```

Ahora añadimos algunos métodos nuevos a `f(x)`:

```
julia> f(x::Int) = "definition for Int"
f (generic function with 2 methods)

julia> f(x::Type{Int}) = "definition for Type{Int}"
f (generic function with 3 methods)
```

Compare cómo difieren estos resultados:

```
julia> f(1)
"definition for Int"

julia> g(1)
"definition for Int"

julia> wait(schedule(t, 1))
"original definition"

julia> t = @async f(wait()); yield();

julia> wait(schedule(t, 1))
"definition for Int"
```

15.5 Parametrically-constrained Varargs methods

Los parámetros de función pueden también ser usados para restringir el número de argumentos que pueden ser proporcionados a una función "varags" (ver Funciones Vararg). La notación `Vararg{T,N}` se usa para indicar tal restricción. Por ejemplo:

```
julia> bar(a,b,x::Vararg{Any,2}) = (a,b,x)
bar (generic function with 1 method)

julia> bar(1,2,3)
ERROR: MethodError: no method matching bar(::Int64, ::Int64, ::Int64)
Closest candidates are:
  bar(::Any, ::Any, ::Any, !Matched::Any) at none:1

julia> bar(1,2,3,4)
(1, 2, (3, 4))

julia> bar(1,2,3,4,5)
ERROR: MethodError: no method matching bar(::Int64, ::Int64, ::Int64, ::Int64, ::Int64)
Closest candidates are:
  bar(::Any, ::Any, ::Any, ::Any) at none:1
```

Más útilmente, es posible restringir métodos varargs mediante un parámetro. Por ejemplo:

```
function getindex(A::AbstractArray{T,N}, indexes::Vararg{Number,N}) where {T,N}
```

sería llamado sólo cuando el número de `indexes` se correspondiera con la dimensionalidad del array.

15.6 Note on Optional and keyword Arguments

Como se menciona brevemente en Funciones, los argumentos opcionales se implementan como sintaxis para múltiples definiciones de métodos. Por ejemplo, esta definición:

```
f(a=1,b=2) = a+2b
```

se traduce a los siguientes tres métodos:

```
f(a,b) = a+2b
f(a) = f(a,2)
f() = f(1,2)
```

Esto significa que llamar a `f()` es equivalente a llamar a `f(1,2)`. En este caso, el resultado es 5, porque `f(1,2)` invoca el primer método de `f` anterior. Sin embargo, este no siempre es el caso. Si define un cuarto método que es más especializado para enteros:

```
f(a::Int,b::Int) = a-2b
```

entonces el resultado de ambos `f()` y `f(1,2)` es -3. En otras palabras, los argumentos opcionales están vinculados a una función, no a ningún método específico de esa función. Depende de los tipos de argumentos opcionales qué método se invoca. Cuando los argumentos opcionales se definen en términos de una variable global, el tipo de argumento opcional puede incluso cambiar en tiempo de ejecución.

Los argumentos de palabra clave se comportan de manera bastante diferente de los argumentos posicionales ordinarios. En particular, no participan en el envío del método. Los métodos se envían basados *únicamente en argumentos posicionales*, *con argumentos de palabra clave procesados* después de que se identifica el método de coincidencia.

15.7 Funciones como objetos

Los métodos están asociados con tipos, por lo que es posible hacer algún objeto Julia arbitrario ”invocable” añadiendo métoos a este tipo (tales objetos ”invocables” son denominados en ocasiones ”functores”).

Por ejemplo, podemos definir un tipo que almacena los coeficientes de un polinomio, pero que se comporta como una función que evalúa el polinomio:

```
julia> struct Polynomial{R}
           coeffs::Vector{R}
       end

julia> function (p::Polynomial)(x)
           v = p.coeffs[end]
           for i = (length(p.coeffs)-1):-1:1
               v = v*x + p.coeffs[i]
           end
           return v
       end
```

Note que la función es especificada por el tipo en lugar de por el nombre. En el cuerpo de la función, p se referira´al objeto que fue llamado. Un `Polynomial` puede ser usado como sigue:

```
julia> p = Polynomial([1,10,100])
Polynomial{Int64}([1, 10, 100])

julia> p(3)
931
```

Este mecanismo es también la clave de cómo los constructores de tipo y cierres (funciones internas que se refieren al entorno que las rodea) funcionan en Julia, lo cual se discute después en el manual.

15.8 Funciones Genéricas Vacías

Ocasionalmente, es útil introducir una función genérica sin agregar métodos. Esto se puede usar para separar las definiciones de interfaz de las implementaciones. También se puede hacer con el fin de la documentación o la legibilidad del código. La sintaxis para esto es un bloque de `function` vacío sin una tupla de argumentos:

```
function emptyfunc
end
```

15.9 Diseño de métodos y evitación de ambigüedades

El polimorfismo de los métodos de Julia es una de sus características más poderosas, pero explotar este poder puede plantear desafíos de diseño. En particular, en jerarquías de métodos más complejos no es raro que surjan ambigüedades.

Arriba se indicó que uno puede resolver ambigüedades como

```
f(x, y::Int) = 1
f(x::Int, y) = 2
```

definiendo un método

```
f(x::Int, y::Int) = 3
```

Ésta es a menudo la estrategia correcta; sin embargo, hay circunstancias en las que seguir este consejo a ciegas puede ser contraproducente. En particular, cuantos más métodos tenga una función genérica, más posibilidades habrá de ambigüedades. Cuando sus jerarquías de métodos se vuelven más complicadas que este simple ejemplo, puede valer la pena pensar cuidadosamente sobre estrategias alternativas.

A continuación, discutimos los desafíos particulares y algunas formas alternativas de resolver dichos problemas.

Tuple and NTuple arguments

Los argumentos `Tuple` (y `NTuple`) presentan retos especiales. Por ejemplo,

```
f(x::NTuple{N,Int}) where {N} = 1
f(x::NTuple{N,Float64}) where {N} = 2
```

son ambiguos debido a la posibilidad de que `N == 0`: no hay elementos para determinar si se debe invocar a la varianteInt o `Float64`. Para resolver la ambigüedad, un enfoque es definir un método para la tupla vacía:

```
f(x::Tuple{}) = 3
```

Alternativamente, para todos los métodos excepto uno podemos insistir en que hay al menos un elemento en la tupla:

```
f(x::NTuple{N,Int}) where {N} = 1           # this is the fallback
f(x::Tuple{Float64, Vararg{Float64}}) = 2   # this requires at least one Float64
```

Orthogonalice su diseño

Cuando tenga la tentación de despachar en dos o más argumentos, considere si una función de ”envoltura” podría hacer un diseño más simple. Por ejemplo, en lugar de escribir múltiples variantes:

```
f(x::A, y::A) = ...
f(x::A, y::B) = ...
f(x::B, y::A) = ...
f(x::B, y::B) = ...
```

podría considerar definir

```
f(x::A, y::A) = ...
f(x, y) = f(g(x), g(y))
```

donde g convierte el argumento para escribir `A`. Esto es un ejemplo muy específico del principio más general de diseño ortogonal, en el que los conceptos separados se alinean a métodos separados. Aquí, g muy probablemente necesitará una definición de repliegue

```
g(x::A) = x
```

Una estrategia relacionada explota `promote` para llevar `x` y `y` a un tipo común:

```
f(x::T, y::T) where {T} = ...
f(x, y) = f(promote(x, y)...)
```

Un riesgo de este diseño es la posibilidad de que si no hay un método de promoción adecuado para convertir `x` y `y` al mismo tipo, el segundo método se repetirá en sí mismo infinitamente y desencadenará un desbordamiento de la pila. La función no exportada `Base.promote_noncircular` se puede usar como alternativa; cuando la promoción falla, aún arrojará un error, pero uno que falla más rápido con un mensaje de error más específico.

Despacho en un argumento a la vez

Si necesita despachar en múltiples argumentos, y hay muchos retrocesos con demasiadas combinaciones para que sea práctico definir todas las variantes posibles, entonces considere introducir una ”cascada de nombres” donde (por ejemplo) despache en el primer argumento y luego llame un método interno:

```
f(x::A, y) = _fA(x, y)
f(x::B, y) = _fB(x, y)
```

Entonces los métodos internos `_fA` y `_fB` pueden enviarse en `y` sin preocuparse por las ambigüedades entre sí con respecto a `x`.

Tenga en cuenta que esta estrategia tiene al menos una desventaja importante: en muchos casos, no es posible para los usuarios personalizar aún más el comportamiento de `f` definiendo más especializaciones de su función `f` exportada. En su lugar, tienen que definir especializaciones para sus métodos internos `_fA` y `_fB`, y esto borra las líneas entre los métodos exportados e internos.

Contenedores abstractos y tipos de elementos

Donde sea posible, trate de evitar definir los métodos que se despachan en tipos de elementos específicos de contenedores abstractos. Por ejemplo,

```
-(A::AbstractArray{T}, b::Date) where {T<:Date}
```

genera ambigüedades para cualquiera que defina un método

```
-(A::MyArrayType{T}, b::T) where {T}
```

El mejor enfoque es evitar definir *cualquiera* de estos métodos: en su lugar, confíe en un método genérico `-(A::AbstractArray, b)` y haga Asegúrese de que este método se implemente con llamadas genéricas (como `similar` y `-`) que hacen lo correcto para cada tipo de contenedor y tipo de elemento *por separado*. Esta es solo una variante más compleja de los consejos para ortogonalize sus métodos.

Cuando este enfoque no es posible, puede valer la pena comenzar un discusión con otros desarrolladores sobre la resolución de la ambigüedad; sólo porque un método se definió primero no necesariamente significa que no puede ser modificado o eliminado Como último recurso, un desarrollador puede definir el método "tirita"

```
-(A::MyArrayType{T}, b::Date) where {T<:Date} = ...
```

eso resuelve la ambigüedad por la fuerza bruta.

Método complejo "cascadas" con argumentos predeterminados

Si está definiendo un método "cascada" que suministra valores predeterminados, sea cuidado al eliminar cualquier argumento que corresponda al potencial por defecto. Por ejemplo, supongamos que estás escribiendo un filtro digital algoritmo y usted tiene un método que maneja los bordes de la señal mediante la aplicación de relleno:

```
function myfilter(A, kernel, ::Replicate)
    Apadded = replicate_edges(A, size(kernel))
    myfilter(Apadded, kernel)  # now perform the "real" computation
end
```

Esto entrará en conflicto con un método que proporciona relleno predeterminado:

```
myfilter(A, kernel) = myfilter(A, kernel, Replicate()) # replicate the edge by default
```

Juntos, estos dos métodos generan una recursión infinita con 'A' creciendo cada vez más.

El mejor diseño sería definir su jerarquía de llamadas de esta manera:

```
struct NoPad end  # indicate that no padding is desired, or that it's already applied

myfilter(A, kernel) = myfilter(A, kernel, Replicate())  # default boundary conditions

function myfilter(A, kernel, ::Replicate)
    Apadded = replicate_edges(A, size(kernel))
    myfilter(Apadded, kernel, NoPad())  # indicate the new boundary conditions
end
```

```
# other padding methods go here

function myfilter(A, kernel, ::NoPad)
    # Here's the "real" implementation of the core computation
end
```

NoPad se proporciona en la misma posición de argumento que cualquier otro tipo de relleno, por lo que mantiene la jerarquía de despacho bien organizada y con menor probabilidad de ambigüedades. Además, amplía lo ”público”interfaz `myfilter`: un usuario que quiere controlar el relleno explícitamente puede llamar a la variante NoPad directamente.

[2] Arthur C. Clarke, *Profiles of the Future* (1961): Clarke's Third Law.

Chapter 16

Constructores

Los constructores [1] son funciones que crean nuevos objetos (específicamente instancias de tipos compuestos). En Julia, los objetos también sirven como funciones constructor: ellos crean instancias de sí mismos cuando se aplican a una dupla de argumentos como una función. Esto se mencionó brevemente cuando se habló de tipos compuestos. Por ejemplo:

```
julia> struct Foo
           bar
           baz
       end

julia> foo = Foo(1, 2)
Foo(1, 2)

julia> foo.bar
1

julia> foo.baz
2
```

Para muchos tipos, formar nuevos objetos enlazando valores de campo juntos es todo se necesita para crear instancias. Hay, sin embargo, casos donde se requiere más funcionalidad cuando se crean objetos compuestos. Algunas invariantes deben ser forzadas, bien chequeando argumentos o transformándolos. Las estructuras de datos recursivas, especialmente aquellas que pueden ser auto referenciadas frecuentemente, no pueden construirse limpiamente sin que primero sean creadas e n un estado incompleto y después sean alteradas programáticamente para ser completadas, como un paso separado de la creación del objeto. Algunas veces, es conveniente ser capaz de construir objetos con menos o diferentes tipos de parámetros que el número de campos que tiene. El sistema Julia para construcción de objetos cubre estos casos y más.

16.1 Métodos constructores externos

Un constructor es como cualquier otro función en Julia en que es su comportamiento global está definido por el comportamiento combinado de sus métodos. Según esto, se puede añadir funcionalidad a ún constructor simplemente definiendo nuevos métodos. Por ejemplo, supóngase que se desea añadir un método constructor para objetos `Foo` que tomar un argumento y usa el valor dado para los dos campos que presentan `baz` y `bar`. Esto es sencillo:

[1] Nomenclature: while the term ”constructor” generally refers to the entire function which constructs objects of a type, it is common to abuse terminology slightly and refer to specific constructor methods as ”constructors”. In such situations, it is generally clear from context that the term is used to mean ”constructor method” rather than ”constructor function”, especially as it is often used in the sense of singling out a particular method of the constructor from all of the others.

```
julia> Foo(x) = Foo(x,x)
Foo

julia> Foo(1)
Foo(1, 1)
```

Podría también añadirse un constructor `Foo` sin argumentos que proporciona valores por defecto para los campos `bar` y `baz`:

```
julia> Foo() = Foo(0)
Foo

julia> Foo()
Foo(0, 0)
```

Aquí, el método constructor sin argumentos llama al método constructor con un argumento, que a su vez llama al método constructor de dos argumentos proporcionado automáticamente. Por razones que se aclararán pronto, los metodos constructor adicionales declarados como métodos formales como éstos se denominan *métodos constructores externos*. Los métodos constructores externos sólo puede crear una nueva instancia llamando a otro método constructor, tal como los proporcionados automáticamente por defecto.

16.2 Métodos Constructores Internos

Aunque los constructores externos resuelven con éxito el problema de proporcionar métodos adicionales para construir objetos, ellos fallan en los otros dos casos de uso mencionados en la introducción este capítulo: forzar invariantes y permitir la construcción de objetos autorreferenciales. Para estos problemas, se necesitan los *métodos constructores internos*. Un método constructor interno es parecido a uno externo, con dos diferencias:

1. Se declara dentro del bloque de la declaración del tipo, el lugar donde fuera como los métodos normales.
2. Tiene acceso a una función especial, existente totalmente, llamada `new` que crea objetos del tipo del bloque.

Poner ejemplo, supóngase que uno quiere declarar un tipo que almacene un par de elementos reales, sujetos a la restricción de que el primer número no es mayor que el segundo. Uno podría declararlo así:

```
julia> struct OrderedPair
           x::Real
           y::Real
           OrderedPair(x,y) = x > y ? error("out of order") : new(x,y)
       end
```

Ahora sólo pueden construirse objetos `OrderedPair` tales que `x <= y`:

```
julia> OrderedPair(1, 2)
OrderedPair(1, 2)

julia> OrderedPair(2,1)
ERROR: out of order
Stacktrace:
 [1] OrderedPair(::Int64, ::Int64) at ./none:4
```

Si el tipo se declara `mutable`, se puede acceder y cambiar directamente los valores de campo para violar esta invariante, pero se considera deficiente la interacción con las partes internas de un objeto sin invitación. Usted (u otra persona) también puede proporcionar más métodos constructores externos en cualquier momento posterior, pero una vez que se declara un tipo, no hay forma de agregar más métodos internos de construcción. Como los métodos constructores

externos solo pueden crear objetos llamando a otros métodos de construcción, en última instancia, se debe llamar a algún constructor interno para crear un objeto. Esto garantiza que todos los objetos del tipo declarado deben existir mediante una llamada a uno de los métodos de constructor internos proporcionados con el tipo, dando así cierto grado de cumplimiento de las invariantes de un tipo.

Si se define cualquier método de constructor interno, no se proporciona ningún método constructor predeterminado: se supone que se ha provisto de todos los constructores internos que necesita. El constructor predeterminado es equivalente a escribir su propio método constructor interno que toma todos los campos del objeto como parámetros (restringidos para ser del tipo correcto, si el campo correspondiente tiene un tipo), y los pasa a new, devolviendo el objeto resultante:

```
julia> struct Foo
           bar
           baz
           Foo(bar,baz) = new(bar,baz)
       end
```

Esta declaración tiene el mismo efecto que la definición anterior del tipo Foo sin un método constructor interno específico. Los siguientes dos tipos son equivalentes (uno con un constructor por defecto y el otro con un constructor explícito):

```
julia> struct T1
           x::Int64
       end

julia> struct T2
           x::Int64
           T2(x) = new(x)
       end

julia> T1(1)
T1(1)

julia> T2(1)
T2(1)

julia> T1(1.0)
T1(1)

julia> T2(1.0)
T2(1)
```

Se considera una buena práctica proporcionar tan pocos constructores internos como sea posible: sólo aquellos que tomen todos los argumentos explícitamente y fuercen la comprobación de errores y las transformaciones esenciales.

Los demás constructores proporcionados, que proporcionan valores por defecto o transformaciones auxiliares, deberían proporcionarse como constructores externos que llaman a los internos para hacer el trabajo pesado. Esta situación suele ser bastante natural.

16.3 Inicialización incompleta

El problema final que aún no se ha resuelto es la construcción de objetos autorreferenciales o, más generalmente, estructuras de datos recursivas. Como la dificultad fundamental puede no ser obvia de inmediato, se explicará brevemente. Considere la siguiente declararción de tipo recursivo:

```
julia> mutable struct SelfReferential
           obj::SelfReferential
       end
```

Este tipo puede parecer bastante inócuo a menos que uno considere cómo construir una instancia de él. Si a es una instancia de `SelfReferential`, entonces una segunda instancia b podría crearse mediante la llamada:

```
julia> b = SelfReferential(a)
```

¿Pero cómo se construye la primera instancia cuando no existe ninguna otra instancia para propocionar un valor válido para el campo `obj`? La única solución es permitir la creación de una instancia de `SelfReferential` que no esté inicializada por completo, con el campo `obj` no asignado, y usar esta instancia incompleta como un valor válido que se podcría asignar al campo `obj` de otra instancia, o incluso al de ella misma.

Para permitir la creación de objetos inicializados de forma incompleta, Julia permite que la función new sea llamada con menos argumentos del número de campos que el objeto tiene, devolviendo un objeto con los campos no especificados sin inicializar. El método constructor interno pues entonces usar el método incompleto, finalizando su inicialización antes de devolverlo. Aquí por ejemplo, se intenta definir el tipo `SelfReferental` con un constructor interno con cero argumentos que devuelve instancias con sus campos `obj` apuntando a ellos mismos:

```
julia> mutable struct SelfReferential
           obj::SelfReferential
           SelfReferential() = (x = new(); x.obj = x)
       end
```

Podemos verificar que este constructor funciona y construye objetos que son, de hecho, autorreferenciados:

```
julia> x = SelfReferential();

julia> x === x
true

julia> x === x.obj
true

julia> x === x.obj.obj
true
```

Aunque se permite crear objetos con campos no inicializados, cualquier objeto a una referencia no inicializada es un error inmediato:

```
julia> mutable struct Incomplete
           xx
           Incomplete() = new()
       end

julia> z = Incomplete();
```

Aunque se permite crear objetos con campos no inicializados, cualquier objeto a una referencia no inicializada es un error inmediato:

```
julia> z.xx
ERROR: UndefRefError: access to undefined reference
```

Esto evita la necesidad de estar comprobando datos `null` continuamente. Sin embargo, no todos los campos de objetos son referencias. Julia considera algunos tipos Como "datos planos", Lo que significa que todos sus datos son auto contenidos y que no referencian otros objetos. Los tipos de datos planos son los tipos primitivos (es decir `Int`) y las estructuras inmutables de otros tipos de datos planos. Los contenidos iniciales de un tipo de datos planos son indefinidos:

```
julia> struct HasPlain

           n::Int

           HasPlain() = new()

       end

julia> HasPlain()
HasPlain(438103441441)
```

Los arrays de tipos de datos planos exhiben el mismo comportamiento.

Uno puede pasar objetos incompletos a otras funciones desde los constructores internos para delegar su terminación:

```
julia> mutable struct Lazy

           xx

           Lazy(v) = complete_me(new(), v)

       end
```

Como sucede con los objetos incompletos devueltos desde los constructores, si `complete_me` o alguno de los métodos que lo llaman intenta acceder al campo `xx` del objeto `Lazy` antes de que éste sea inicializado, se lanzará un error de inmediato.

16.4 Constructores paramétricos

Los tipos paramétricos añaden algunas complicaciones al tema de los constructores. Recuérdese la sección Tipos Paramétricos que por defecto pueden construirse instancias de estos tipos dando explícitamente los parámetros de tipo o con parámetros de tipo implicados por los tipos de los argumentos dados al constructor. He aquí algunos ejemplos:

```
julia> struct Point{T<:Real}
           x::T
           y::T
       end

julia> Point(1,2) ## implicit T ##
Point{Int64}(1, 2)
```

```
julia> Point(1.0,2.5) ## implicit T ##
Point{Float64}(1.0, 2.5)

julia> Point(1,2.5) ## implicit T ##
ERROR: MethodError: no method matching Point(::Int64, ::Float64)
Closest candidates are:
  Point(::T<:Real, !Matched::T<:Real) where T<:Real at none:2

julia> Point{Int64}(1, 2) ## explicit T ##
Point{Int64}(1, 2)

julia> Point{Int64}(1.0,2.5) ## explicit T ##
ERROR: InexactError()
Stacktrace:
 [1] convert(::Type{Int64}, ::Float64) at ./float.jl:679
 [2] Point{Int64}(::Float64, ::Float64) at ./none:2

julia> Point{Float64}(1.0, 2.5) ## explicit T ##
Point{Float64}(1.0, 2.5)

julia> Point{Float64}(1,2) ## explicit T ##
Point{Float64}(1.0, 2.0)
```

Como podemos ver, para las llamadas a constructor con parámetros de tipo explícito, los argumentos se convierten a los tipos implícitos de los campos: `Point{Int64}(1,2)` funciona, pero `Point{Int64}(1.0,2.5)` lanza un `InexactError` cuando `2.5` se convierte a `Int64`. Cuando el tipo es implicado por los argumentos de la llamada al constructor, como en `Point(1,2)`, entonces los tipos de los argumentos deben concordar para que se pueda determinar T, pero da igual cuáles sean los tipos mientras ambos sean iguales y, además, subclases de `Real`.

Lo que está pasando aquí realmente es que `Point`, `Point{Float64}` y `Point{Int64}` son funcioens constructores diferentes. De hecho, `Point{T}` es una función constructor distinto para cada tipo T. Sin ningún constructor interno propocionado explícitamente, la declaración del tipo compuesto `Point{T<:Real}` proporciona automáticamente un constructor interno `Point{T}` para cada posible ipo `T<:Real` que se comporta justo como lo hacen los constructores internos no paramétricos por defecto. Ella también proporciona un solo constructor general externo que toma pares de argumentos reales, que deben ser del mismo tipo. Esta provisión automática de constructores es equivalente a la siguiente declaración explícita:

```
julia> struct Point{T<:Real}
           x::T
           y::T
           Point{T}(x,y) where {T<:Real} = new(x,y)
       end

julia> Point(x::T, y::T) where {T<:Real} = Point{T}(x,y);
```

Observe que cada definición se parece a la forma de llamada de constructor que maneja. La llamada `Point{Int64}(1,2)` invocará la definición `Point{T}(x, y)` dentro del bloque `type`.

La declaración de constructor externo, por otro lado, define un método para el constructor general de `Point` que sólo se aplica a pares de valores del mismo tipo real. Esta declaración hace que las llamadas al constructor sin parámetros de tipo explícitos, como `Punto(1,2)` y `Punto(1.0,2.5)`, funcionen. Dado que la declaración del método restringe los argumentos para que sean del mismo tipo, las llamadas como `Point(1,2.5)`, con argumentos de diferentes tipos, dan como resultado errores ”no method”.

Supongamos que queremos hacer la llamada a constructor `Point(1, 2.5)` funcione promocionando el valor entero `1` a punto flotante `1.0`. La forma más sencilla de conseguir eso es definir el siguiente método constructor adicional:

```
julia> Point(x::Int64, y::Float64) = Point(convert(Float64,x),y);
```

Este método usa la función `convert()` para convertir explícitamente x a `Float64` y entonces delegar la construcción al constructor general para el caso de que ambos argumentos sean `Float64`. Con esta definición de método, lo que previamente producía un `MethodError` ahora crea con éxito un punto de tipo `Point{Float64}`:

```
julia> Point(1,2.5)
Point{Float64}(1.0, 2.5)

julia> typeof(ans)
Point{Float64}
```

Sin embargo, otras llamadas similares siguen sin funcionar:

```
julia> Point(1.5,2)
ERROR: MethodError: no method matching Point(::Float64, ::Int64)
Closest candidates are:
  Point(::T<:Real, !Matched::T<:Real) where T<:Real at none:1
```

Para una forma mucho más general de hacer que todas estas llamadas funcionen sensiblemente, ver Conversión y promoción. A riesgo de estropear el suspense, podemos revelar aquí que todo lo toma el siguiente método externo para hacer que todas las llamadas al constructor general `Point` trabajen como uno debería esperar:

```
julia> Point(x::Real, y::Real) = Point(promote(x,y)...);
```

La función `promote` convierte todos sus argumentos a un tipo común (en este caso, `Float64`). Con esta definición de método el constructor `Point` promociona sus argumentos de la misma forma que lo hacen los operadores aritméticos como `+` y funciona para todos los tipos de números reales:

```
julia> Point(1.5,2)
Point{Float64}(1.5, 2.0)

julia> Point(1,1//2)
Point{Rational{Int64}}(1//1, 1//2)

julia> Point(1.0,1//2)
Point{Float64}(1.0, 0.5)
```

Por tanto, mientras los constructores con parámetros de tipo implícitos proporcionados por defecto en Julia son muy estrictos, es posible hacer que se comporten de una forma más relajada pero sensible con bastante facilidad. Además, como los constructores pueden sacar ventaja de toda la potencia del sistema de tipos, métodos y despacho múltiple, definir comportamientos sofisticados suele ser bastante simple.

16.5 Case Study: Rational

Quizás la mejor forma de unir todas las piezas es presentar un ejemplo del mundo real de un tipo compuesto paramétrico y sus métodos constructores. Para este fin, he aquí una parte de `rational.jl`, que implementa los Números Racionales en Julia:

```
julia> struct OurRational{T<:Integer} <: Real
           num::T
           den::T
           function OurRational{T}(num::T, den::T) where T<:Integer
               if num == 0 && den == 0
                    error("invalid rational: 0//0")
               end
```

```
            g = gcd(den, num)
            num = div(num, g)
            den = div(den, g)
            new(num, den)
        end
    end

julia> OurRational(n::T, d::T) where {T<:Integer} = OurRational{T}(n,d)
OurRational

julia> OurRational(n::Integer, d::Integer) = OurRational(promote(n,d)...)
OurRational

julia> OurRational(n::Integer) = OurRational(n,one(n))
OurRational

julia> //(n::Integer, d::Integer) = OurRational(n,d)
// (generic function with 1 method)

julia> //(x::OurRational, y::Integer) = x.num // (x.den*y)
// (generic function with 2 methods)

julia> //(x::Integer, y::OurRational) = (x*y.den) // y.num
// (generic function with 3 methods)

julia> //(x::Complex, y::Real) = complex(real(x)//y, imag(x)//y)
// (generic function with 4 methods)

julia> //(x::Real, y::Complex) = x*y'//real(y*y')
// (generic function with 5 methods)

julia> function //(x::Complex, y::Complex)
           xy = x*y'
           yy = real(y*y')
           complex(real(xy)//yy, imag(xy)//yy)
       end
// (generic function with 6 methods)
```

La primera línea – `struct OurRational{T<:Integer} <: Real` – declara que `OurRational` toma un parámetro de un subtipo de `Integer`, aunque él en si mismo es un tipo `Real`. Las declaraciones de campo `num::T` y `den::T` indican que los datos almacenados en un objeto `OurRational{T}` será un par de enteros de tipo T, uno que representará el numerador y otro el denominador.

Ahora las cosas se ponen interesantes. `OurRational` tiene un solo constructor interno que comprueba que tanto `num` como `den` no son cero, y aegura que cada número racional se construye en sus términos mínimos con un denominador no negativo. Esto se consigue dividiendo los valores de numerador y denominador por su máximo común divisor, el cuál se calcula a través de la función `gcd`. Por último, y como `gcd` asigna el signo del primer argumento (en este caso `den`) se garantiza que el denominador ya no sea negativo. Como este es el único constructor interno de `Rational`, podemos estar seguros de que los objetos de este tipo siempre se construyen en forma normalizada.

`Rational` también proporciona varios métodos constructores externos por conveniencia. El primero es el constructor general ”estándar”, que infiere el tupo del parámetro T a partir del tipo del numerado y denominador que tienen que ser del mismo tipo. El segundo se aplica cuando numerador y denominador tiene tipos distintos: los promociona a un tipo común y entonces delega la construcción al otro constructor externo con argumentos del mismo tipo. En tercer constructor externo convierte valores enteros en racionales proporcionando un denominador de valor 1.

Siguiendo las definiciones de constructores externos, tenemos una serie de métodos para el operador `//` que proporcionan una sintaxis para escribir racionales. Antes de estas definiciones, `//`es un operador completamente indefinido con sólo sintaxis y sin significado. Después, se comporta tal y com se describe en Números Racionales – (su comportamiento completo está descrito en estas pocas líneas). La primera y más básica definición hace `a//b` construya un `OurRational` aplicando el constructor de este tipo sobre `a` y `b` cuando ambos son enteros. Cuando uno de los operandos de `//` ya es un número racional se construye un nuevo número racional para la razón resultante con una leve diferencia: este comportamiento es igual a la división de un racional entre un entero. Por último, aplicar `//` a valores complejos enteros crea una instancia de `Complex{Rational}`, que es un complejo cuyas partes real el imaginaria son racionales:

```
julia> ans = (1 + 2im)//(1 - 2im);

julia> typeof(ans)
Complex{OurRational{Int64}}

julia> ans <: Complex{OurRational}
false
```

Por tanto, aunque el operador `//` suela devolver una instancia de `OurRational`, si uno de sus rgumentos es un complejo entero, devolverá una instancia de `Complex{OurRational}`. El lector interesado debería considerar la lectura del resto de `rational.jl`: es corto, autocontenido e implmeenta un tipo básico de Julia al completo.

16.6 Constructores and Conversión

Los constructores `T(args)` se implementan como otros objetos invocables: los métodos se añaden a sus tipos. El tipo de un tipo es `Type` por lo que los métodos constructores se almacenan en la tabla de métodos para el tipo `Type`. Esto significa que se pueden declarar constructores más flexibles, es decir, constructores para tipos abstractos, mediante la definición explícita de métodos para los tipos apropiados.

Sin embargo, en algunos casos, uno debería considerar añadir métodos a `Base.convert` en ougar de definir un constructor, dado que Julia retrocede para llamar a `convert()` si no se encuentra un constructor que coincida. Por ejemplo, si no existe para constructor para `T(args...)` = se llamará a `Base.convert(::Type{T}, args...)=...`

`convert` se usa extensivamente a través de Julia cuando un tipo tenga que ser convertido en otro (por ejemplo, en asignación, `ccall`, etcetera), y sólo debería ser definido (o exitoso) si la conversión se realiza sin pérdidas. Por ejemplo, `convert(Int, 3.0)` produce 3, pero `convert(Int,3.2)` lanza un `InexactError`. Si desea construirse un constructor para una conversión sin pérdidas de un tipo a otro, probablemente sería mejor definir un método `convert`.

Por otra parte, si el constructro no representa una conversión sin pérdida, o no represnta ningua conversion es mejor dejarlo como constructor en lugar de como un método `convert`. Por ejemplo, el constructor `Array{Int}` crea un array cero-dimensional del tipo `Int` pero no es realmente una conversión de `Int` a `Array`.

16.7 Constructores sólo exteriores

Como se ha visto, un tipo paramétrico típico tiene constructroes internos que son invocados cuano se conocen los tupos de los parámetros, por ejemplo, se aplican a `Point{Int}`paro no a `Point`. Opcionalmente, los constructores externos que determinan los parámetros de tipo pueden ser añadidos automáticamente, por ejemplo, construir un `Point{Int}` a partir de la llamada `Point(1,2)`. Los constructores externos llaman a los constructores internos para que hagan el trabajo básico de hacer una instancia. Sin embargo, en algunos caos, uno podría en lugar de eso no proporcionar contructores internos para que los parámetros específicos no puedan ser solicitados manualmente.

Por ejemplo, suponga que se define un tipo que almacena un vector con una representaciócn exacta de su suma:

```
julia> struct SummedArray{T<:Number,S<:Number}

           data::Vector{T}

           sum::S

       end

julia> SummedArray(Int32[1; 2; 3], Int32(6))
SummedArray{Int32,Int32}(Int32[1, 2, 3], 6)
```

El problema es que nosotros queremos que S sea un tipo más grande que T, por lo que podemos sumar muchos elementos con menos pérdida de información. Por ejemplo, cuando T es `Int32`, querríamos que S fiuera `Int64`. Por tano queremos evitar un interfaz que permita al usuario construir instanca del tipo `SummedArray{Int32,Int32}`. Una forma de hacer esto es proporcionar sólo un constructor más exterior para `SummedArray`. Esto puede hacerse usando a definicíon de métdo por tipo pero dentro del bloque de definición `type` para suprimir la generación de bucles por defecto:

```
julia> struct SummedArray{T<:Number,S<:Number}

           data::Vector{T}

           sum::S

           function SummedArray(a::Vector{T}) where T

               S = widen(T)

               new{T,S}(a, sum(S, a))

           end

       end

julia> SummedArray(Int32[1; 2; 3], Int32(6))
ERROR: MethodError: no method matching SummedArray(::Array{Int32,1}, ::Int32)
Closest candidates are:
  SummedArray(::Array{T,1}) where T at none:5
```

El construcror será invocado por la sintaxis `SummedArray(a)`. La sintaxis `new{T,S}` permite especificar parámetrospara el tipo que se va a construir, es decir, esta llamada devolvera un `SummedArray{T, s]`. `new{T,S}` puede usarse en cualquier definicin de constructor, pero por conveniencia los parámtros a `new{}` se derivan automáticamente del tipo que se está construyendo cuando sea posible.

Chapter 17

Conversión y Promoción

Julia tiene un sistema para promocionar argumento de operaciones matemáticas a un tipo común, que ha sido mencionado en varias secciones, incluyendo números enteros y en punto flotante, operaciones matemáticas y funciones elementales, Tipos, and Métodos.En esta sección, explicaremos cómo funciona este sistema de promociones, y también cómo extenderlo a nuevos tipos y aplicarlo a funciones junto a operadores matemáticos predefinidos. Tradicionalmente, los lenguajes de programación caen en dos categorías con respecto a la promoción de los argumentos aritméticos:

- **Promoción automática para operadores y tipos aritméticos predefinidos**. En la mayoría de los lenguajes, los tipos numéricos predefinidos, cuando se usan como operandos de operaciones aritméticas con una sintaxis infija, tal y como `+`, `-`, `*` y `/`, son promocionados automáticamente a un tipo común para producir el resultado esperado. C, Java, Perl y Pytho„ por nombrar unos pocos, calculan todos la suma `1 + 1.5` correctamente como el valor en punto flotante `2.5`, incluso aunque uno de los operandos sea un entero. Estos sistemas son convenientes y diseñados cuidadosamente de forma que generalmente realizan esta labor de forma invisible al programador: a duras penas, alguien consciente piensa que esta promoción está teniendo lugar cuando escribe una expresión, pero los compiladores e intérpretes deben realizar la conversión antes de la adición debido a que los valores enteros y en punto flotante no pueden sumarse tal cual. Por tanto, reglas complejas para tales conversiones automáticas son una parte inevitable de la especificación e implementación de estos lenguajes.
- **No promoción automática** Este campo incluye a Ada y ML (lenguajes tipados estáticamente y muy estrictos). En estos lenguajes, cada conversión debe ser especificada por el programador de foma explícita. Por tanto, la expresión de ejemplo `1 + 1.5` daría un error de compilación en ambos lenguajes. En lugar de esta expresión, uno debería escribir `real(1) + 1.5`, convirtiendo explícitamente el 1 enteror a un valores en punto flotante antes de realizar la adición. La conversión explícita en todos sitios es tan inconveniente, sin embargo, que incluso Ada tiene algún grado de conversión automática.: los literales enteros son promocionaods al tipo entero esperado automáticamente, y los literales en punto flotante son promocionados similarmente a los tipos apropiados en punto flotante.

En cierto sentido, Julia cae en la categoría ”no promoción automática”: los operadores automáticos son funciones con sintaxis especial, y los argumentos de funciones no son nunca convertidos automáticamente. Sin embargo, uno puede observar que aplicar operaciones matemáticas a una amplia variedad de tipos de argumentos mixtos es justo un caso extremo del despacho múltiple polimórfico (algo que el despacho de Julia y los sistemas de tipos manejan bastante bien. La promoción ”automática” de operandos matemáticos simplemente emerge como una aplicación especial: Julia viene un reglas de despacho ”atrapa-todo” predefinidas para los operadores matemáticos, invocadas cuando no existen implementaciones específicas para alguna combinación de tipos de operandos. Estas reglas ”atrapa-todo” primero promocionan todos los operandos a un tipo común usando reglas de promoción definibles por el usuario, y luego invoca a una implementación especializada del operador en cuestión para los valores resultantes, ahora del mismo tipo. Los tipos definidos por el usuario pueden participar fácilmente en este sistema de promoción definiendo métodos para

la conversión hacia o desde otros tipos, y proporcionar un puñado de reglas de promoción que definan a qué tipos deberían ellos promocionarse cuando se mezclan con otros tipos.

17.1 Conversión

La conversión de valores a varios tipos es llevada a cabo mediante la función `convert`. Esta función suele tomar dos argumentos: el primero es un objeto tipo mientras que el segundo es un valor que hay que convertir a ese tipo; el valor devuelto es el valor convertido a una instancia del tipo dado. La forma más simple de comprender esta función es verla en acción:

```
julia> x = 12
12

julia> typeof(x)
Int64

julia> convert(UInt8, x)
0x0c

julia> typeof(ans)
UInt8

julia> convert(AbstractFloat, x)
12.0

julia> typeof(ans)
Float64

julia> a = Any[1 2 3; 4 5 6]
2×3 Array{Any,2}:
 1  2  3
 4  5  6

julia> convert(Array{Float64}, a)
2×3 Array{Float64,2}:
 1.0  2.0  3.0
 4.0  5.0  6.0
```

La conversión no es siempre posible, en cuyo caso se lanza un error no método indicando que `convert` no sabe cómo realizar la conversión solicitada:

```
julia> convert(AbstractFloat, "foo")
ERROR: MethodError: Cannot `convert` an object of type String to an object of type AbstractFloat
This may have arisen from a call to the constructor AbstractFloat(...),
since type constructors fall back to convert methods.
```

Algunos lenguajes consideran que el análisis sintáctico de cadenas como número o el formateo de números a cadenas es una conversión (muchos lenguajes dinámicos realizarán esta conversión por ti automáticamente). Sin embargo, Julia no lo hace. Incluso aunque algunas cadenas puedan ser analizadas como números, la mayoría de las cadenas no son representaciones válidas de números, y sólo un subconjunto muy limitado de ellas lo son. Por tanto, en Julia la función dedicada `parse()` debe ser usada para realizar esta operación, haciéndolo más explícito.

Definiendo nuevas conversiones

Para definir nuevas conversiones, simpemente propocionaremos un nuevo método para `convert()`. Esto es realmente todo lo que hay que hacer. Por ejemplo, el método para convertir un número real a boolean es:

```
convert(::Type{Bool}, x::Real) = x==0 ? false : x==1 ? true : throw(InexactError())
```

El tipo del primer argumento de este método es un tipo singleton, `Type{Bool}`, , la única instancia del cuál es `Bool`. Por tanto, este método es sólo invocado cuando el primer argumento es el valor tipo `Bool`. Nótese la sintaxis usada pora el primer argumento: el nombre del argumento es omitido antes del símbolo `::` y sólo se da el tipo. Esta es la sintaxis de Julia para un argumento a función cuyo tipo está especificado pero su valor nunca se utilizará en el cuerpo de la función. En este ejemplo, como el tipo es un singleton, nunca habría una razón para usar su valor dentro del cuerpo. Cuando se invoca el método determina si un valor numérico es verdadero o falso como un boolean, comparándolo con uno y con cero:

```
julia> convert(Bool, 1)
true

julia> convert(Bool, 0)
false

julia> convert(Bool, 1im)
ERROR: InexactError()
Stacktrace:
 [1] convert(::Type{Bool}, ::Complex{Int64}) at ./complex.jl:31

julia> convert(Bool, 0im)
false
```

Las signaturas de método para métodos de conversión están frecuentemente un poco más implicadas que este ejemplo, especialmente para tipos paramétricos. El ejemplo de antes está pensado para ser pedagógico, y no es el comportamiento actual de Julia. He aquí la implementación actual en Julia:

```
convert(::Type{T}, z::Complex) where {T<:Real} =
    (imag(z) == 0 ? convert(T, real(z)) : throw(InexactError()))
```

Caso de estudio: Conversiones de `Rational`

Para continuar con nuestro caso de estudio sobre el tipo `Rational` de Julia, he aquí las conversiones declaradas en `rational.jl`, después de la declaración del tipo y sus constructores:

```
convert(::Type{Rational{T}}, x::Rational) where {T<:Integer} =
↪   Rational(convert(T,x.num),convert(T,x.den))
convert(::Type{Rational{T}}, x::Integer) where {T<:Integer} = Rational(convert(T,x),
↪   convert(T,1))

function convert(::Type{Rational{T}}, x::AbstractFloat, tol::Real) where T<:Integer
    if isnan(x); return zero(T)//zero(T); end
    if isinf(x); return sign(x)//zero(T); end
    y = x
    a = d = one(T)
    b = c = zero(T)
    while true
```

```
        f = convert(T,round(y)); y -= f
        a, b, c, d = f*a+c, f*b+d, a, b
        if y == 0 || abs(a/b-x) <= tol
            return a//b
        end
        y = 1/y
    end
end
convert(rt::Type{Rational{T}}, x::AbstractFloat) where {T<:Integer} = convert(rt,x,eps(x))

convert(::Type{T}, x::Rational) where {T<:AbstractFloat} = convert(T,x.num)/convert(T,x.den)
convert(::Type{T}, x::Rational) where {T<:Integer} = div(convert(T,x.num),convert(T,x.den))
```

Los cuatro primeros métodos `convert` proporcionan conversión a tipos racionales. El primer método convierte el tipo de reacional a otro tipo de racional convirtiendo el nomerador y el denominador al tipo de entero apropiado. El segundo método hace la misma conversión para enteros tomando el denominado para que sea 1. El tercer método implementa un aalgoritmo estándar para aproximar un número de punto flotante por una razón de enteros dentro de un a tolerancia dada, y el cuarto método lo aplica, usando el epsilo de máquina como el valor dado para elumbral. En general, uno debería tener `a//b == convert(Rational{Int64}, a/b)`.

Los dos últimos métodos conversores proporcionan conversiones de tipos racionales a punto flotante y entero. Para convertir a punto flotante, uno simplemente convierte tanto numerador como denominador a punto flotante y luego divide. Para convertir a entero, uno usa el operador `div` para división entera truncada (redondeo hacia cero).

17.2 Promoción

La promoción se refiere a convertir valores de tipos mezclados a un solo tipo común. Aunque esto no es estrictamente necesario, se supone generalmente que el tipo común al cuál los valores son convertidos puede representar de forma fidedigna todos los valores. En este sentido, el término ”promoción” es apropiado ya que los valores son convertidos a un tipo ”mayor” (es decir, uno que pueda reprsentar todos los valores de entrada en un solo tipo común). Es importante, sin embargo, no confundir esto con los super-tipos orientados a objetos (estructurales), o l nocíon de super-tipos abstractos de Julia: la promoción no tiene nada que ver con la jerarquía de tipos, y todo que ver con convertir entre representaciones alternas. Por ejemplo, aunque cad valor `Int32` puede también ser representado como un valor `Float64`, `Int32` no es un subtipo de `Float64`.

La promoción a un tipo mayor común es realizada por Julia mediante la función `promote`, que toma cualquier número de argumentos, y devuelve un atupla con el mismo número de valores, convertidos a un tipo común, o lanza una excepción si no es posible la promoción. El caso de uso más común para la promoción es convertir argumentos numéricos a un tipo común:

```
julia> promote(1, 2.5)
(1.0, 2.5)

julia> promote(1, 2.5, 3)
(1.0, 2.5, 3.0)

julia> promote(2, 3//4)
(2//1, 3//4)

julia> promote(1, 2.5, 3, 3//4)
(1.0, 2.5, 3.0, 0.75)

julia> promote(1.5, im)
(1.5 + 0.0im, 0.0 + 1.0im)
```

```
julia> promote(1 + 2im, 3//4)
(1//1 + 2//1*im, 3//4 + 0//1*im)
```

Los valores en punto flotante son promocionados al mayor de los tipos de los argumento en punto flotante. Los valores enteros son promocionados al mayor de el tamaño de palabra de máquina nativo o dl mayor tipo dde argumento entero. Las mezclas de valores enteros y en punto flotante son promocionados a tipos en punto flotante bastent grandes como para almacenar todos los valores. Los enteros mezclados con racionales promocionan a racionales. Los racionales mezclados con valores en punto flotante son promocionados a valores en punto flotante. Los valores complejos mezclados con valores relaes se promocionan al tipo apropiado de valor complejo.

Esto es realmente todo lo que es usar promociones. El resto es sólo cuestión de una aplicación inteligente, siendo la definición de métodos "atrapa-todo" para operaciones numéricas tales como las aritméticas `+`, `-`, `*` y `/` las más típicas aplicaciones inteligentes. He aquí algunas de las definiciones de métodos "atrapa-todo" dados en `promotion.jl`:

```
+(x::Number, y::Number) = +(promote(x,y)...)
-(x::Number, y::Number) = -(promote(x,y)...)
*(x::Number, y::Number) = *(promote(x,y)...)
/(x::Number, y::Number) = /(promote(x,y)...)
```

Estas definiciones de métodos dicen que en la ausencia de reglas más específicas para sumar, restar, multiplicar y dividir pares de valores numéricos, promocionemos los valores a un tipo común y entonces intentemos de nuevo. Este es todo aquí: en ninguna otra parte hay que preocuparse por la promoción a un tipo numérico común para las operaciones aritméticas (ello sucede automáticamente). Hay definiciones de métodos de promoción "atrapa-todo" para un número de otras funciones arimtéticas y matemáticas en `promotion.jl`, pero más allá de eso, difícilmente encontremos ninguna llamada a `promote` necesaria en la librería estándar de Julia. Los usos más comunes de `promote` ocurren en los métodos de construcción externos, proporcionados por conveniencia, para permitir llamadas a constructor con tipos mezclados para delegar a un tipo interno con campos promocionados a un tipo común aproximado. Por ejemplo, recordemos que `rational.jl` proporciona el siguiente método constructor externo:

```
Rational(n::Integer, d::Integer) = Rational(promote(n,d)...)
```

This allows calls like the following to work:

```
julia> Rational(Int8(15),Int32(-5))
-3//1

julia> typeof(ans)
Rational{Int32}
```

Para la mayoría de los tipos definidos por usuario, es mejor práctica requerir que los programadores proporcionen los tipos esperados para las funciones constructor explícitamente, pero algunas veces, especialmente para problemas numéricos, puede ser conveniente realizar la conversión de forma automática.

Definiendo reglas de promoción

Aunque uno podría, en principio, definir métodos para la función `promote` directamente, esto requeriría muchas defniciones redundantes para todas las posibles permutaciones de tipos de argumentos. En lugar de ello, el comportamiento de `promote` es definido en términos de una función auxiliar denominada `promote_rule`, para la que uno puede proporcionar métodos. La función `promote_rule` toma un par de objetos tipo y devuelve otro objeto tipo, tal que instancias de los tipos de argumentos sean promocionadas al tipo retornado. De este modo, definiendo la regla:

```
promote_rule(::Type{Float64}, ::Type{Float32}) = Float64
```

uno declara que cuando se promocionan juntos valores en punto flotante de 32 y de 64 bits, ellos deberían ser promocionados a punto flotante de 64 bits. El tipo de promoción no tiene que ser uno de los tipos de los argumentos. He aquí un par de ejemplos de reglas de promoción que aparecen en la librería estándar de Julia:

```
promote_rule(::Type{UInt8}, ::Type{Int8}) = Int
promote_rule(::Type{BigInt}, ::Type{Int8}) = BigInt
```

En el último caso, el tipo de resultado es `BigInt` since `BigInt` ya que éste es el único tipo lo bastante grande como para alojar enteros para aritmética entera de precisión arbitraria. Nótese también que uno no necesita definir dos reglas simétricas `promote_rule(::Type{A}, ::Type{B})` y `promote_rule(::Type{B}, ::Type{A})` (esta simetría es supuesta por la forma en que `promote_rule` es utilizada en el proceso de promoción.

La función `promote_rule` se usa com un bloque constructivo para definir una segunda función llamada `promote_type` la cuál, dado cualquier número de objetos tupo, devuelve el tipo común al cuál esos valores, como argumentos a `promote` deberían ser promocionados. Por tanto, si uno quiere saber, en ausencia de valores actuales, a qué tipo promocionaría una colección de valores de cierto tipo, uno podría usar `promote_type`:

```
julia> promote_type(Int8, UInt16)
Int64
```

Internamente, `promote_type` se usa dentro de `promote` para determinar a qué valores argumento tipo deberían ser convertidos tras una promoción. Él puede, sin embargo, ser útil en sí misma. El lector curioso puede leer el código en `promotion.jl`, que define el mecanismo de promoción completo en aproximadamente 35 líneas.

Caso de estudio: promociones Rational

Finalmente, finalizaremos nuestro caso de estudio el tipo de los números racionales en Julia, que hace un uso relativamente sofistidcado del mecanismo de promoción con las siguientes reglas de promoción:

```
promote_rule(::Type{Rational{T}}, ::Type{S}) where {T<:Integer,S<:Integer} =
↪   Rational{promote_type(T,S)}
promote_rule(::Type{Rational{T}}, ::Type{Rational{S}}) where {T<:Integer,S<:Integer} =
↪   Rational{promote_type(T,S)}
promote_rule(::Type{Rational{T}}, ::Type{S}) where {T<:Integer,S<:AbstractFloat} =
↪   promote_type(T,S)
```

La primera regla dice que promocionar un número racional con algún otro entero promociona a un tipo racional cuyo tipo numerados/denominador es el resultado de la promoción de sus tipos numerador/denoinador con el otro tipo entero. La segunda regla aplica la misma lógic a dos tipos diferentes de números racionales, dando como resultado un racional de la promoción de sus respectivos tipos numerador/denominador. Las reglas tercera y última dictan que promocionar un racional con un punto flotante da como resultado el mismo tipo que promocionar el tipo de numerador/denominador con el float.

Este pequeño puñado de reglas de promoción, junto con los métodos de conversión discutidos antes, son suficiente para hacer que los números racionales interoperen completamente y de forma natural con todos los demás tipos numéricos de Julia (enteros, números en punto flotante y números complejos). Proporcionando métodos de conversión apropiados y reglas de promoción en la misma manera, cualquier tipo numérico definido por el usuario puede interoperar así de forma natural con los numéricos predefinidos en Julia.

Chapter 18

Interfaces

Un montón de la potencia y extensibilidad de Julia viene de una colección de interfaces informales. Extendiendo unos pocos métodos específicos para que trabajen para un tipo personalizado, los objetos de este tipo no sólo reciben estas funcionalidades, sino que son también capaces de ser usados en otros métodos que han sido escritos para ser construidos genéricamente sobre esos comportamientos.

18.1 Iteración

Métodos requeridos		Breve descripción
`start(iter)`		Devuelve el estado inicial de iteración
`next(iter, state)`		Devuelve el ítem actual y pone `state` en el siguiente
`done(iter, state)`		Comprueba si quedan más ítems
Métodos opcionales importantes	**Definiciones por defecto**	**Breve descripción**
`iteratorsize(IterType)`	`HasLength()`	Uno de `HasLength()`,`HasShape()`,`IsInfinite()`, o `sizeUnknown()`, según convenga
`iteratoreltype(IterType)`	`HasEltype()`	Uno de `EltypeUnknown()` o `HasEltype()`, según convenga
`eltype(IterType)`	`Any`	El tipo de los ítems devueltos por `next()`
`length(iter)`	(*indefinido*)	El número de ítems, si es conocido
`size(iter, [dim...])`	(*indefinido*)	El número de ítems en cada dimensión, si es conocido

Valor devuelto por `iteratorsize(IterType)`	Métodos requeridos
`HasLength()`	`length(iter)`
`HasShape()`	`length(iter)` and `size(iter, [dim...])`
`IsInfinite()`	(*ninguno*)
`SizeUnknown()`	(*ninguno*)

Valro devuelto por `iteratoreltype(IterType)`	Métodos requeridos
`HasEltype()`	`eltype(IterType)`
`EltypeUnknown()`	(*ninguno*)

La iteración secuencial es implementada mediante los métodos `start()`, `done()`, y `next()`. En lugar de mutar objetos cuando se itera sobre ellos, Julia proporciona estos tres métodos que llevaqn la traza del estado de la iteración

externamente al objeto. El método `start(iter)` devuvelve el estado inicial para un objeto iterable `iterd`. Este estado se pasa a lo largo de `done(iter, state)` que chequea si quedan más elementos, y `next(iter, state)` que devuelve una tupla que contiene el elemento y el estado actuales. El objeto `state`puede ser cualquier cosa, y suele ser considerado un detalle de implementación privado al objeto iterable.

Cualquier objeto que defina estos tres métodos es iterable y puede ser usado en las muchas funciones que se basan en la iteración. También puede ser usado directamente en un bucle for ya que la sintaxis:

```
for i in iter   # or  "for i = iter"
    # body
end
```

es traducida por:

```
state = start(iter)
while !done(iter, state)
    (i, state) = next(iter, state)
    # body
end
```

Un ejemplo sencillo es una secuencia iterable de cuadrados de número con una longitud definida:

```
julia> struct Squares
           count::Int
       end

julia> Base.start(::Squares) = 1

julia> Base.next(S::Squares, state) = (state*state, state+1)

julia> Base.done(S::Squares, state) = state > S.count

julia> Base.eltype(::Type{Squares}) = Int # Note that this is defined for the type

julia> Base.length(S::Squares) = S.count
```

Con sólo las definiciones de `start`, `next`, y `done`, el tipo `Squares` es ya muy poderoso. Podemos iterar sobre todos los elementos:

```
julia> for i in Squares(7)
           println(i)
       end
1
4
9
16
25
36
49
```

Podemos usar muchos de los métodos predefinidos que trabajan con iterables, como `in()`, `mean()` y `std()`:

```
julia> 25 in Squares(10)
true
```

```
julia> mean(Squares(100))
3383.5

julia> std(Squares(100))
3024.355854282583
```

Hay unos pocos más métodos que se pueden extender para dar a Julia más información sobre esta colección iterable. Se sabe que todos los elementos en una secuencia `Squares` serán `Int`. Extendiendo el método `eltype()`, se puede proporcionar esta información a Julia y ayudarlo a hacer código más especializado en métodos más complicados. También se sabe el número de elementos de esa secuencia, por lo que también se pude extender `length()`.

Ahora, cuando pedimos a Julia que `collect()` todos los elementos en un array ella puede preasignar un `Vector{Int}` en la parte derecha de la expresión, en lugar de ir poniendo a ciegas mediante `push!`ing cada elemento en un `Vector{Any}`.

```
julia> collect(Squares(10))' # transposed to save space
1×10 RowVector{Int64,Array{Int64,1}}:
 1  4  9  16  25  36  49  64  81  100
```

Aunque podemos confiar en las implementaciones genéricas, podemos también extender métodos específicos donde sepamos que hay un algoritmo más simple. Por ejemplo, he aquí una fórmula para sobreescribir la versión iterativa para una solución más eficiente:

```
julia> Base.sum(S::Squares) = (n = S.count; return n*(n+1)*(2n+1)÷6)

julia> sum(Squares(1803))
1955361914
```

Este es un patrón muy común a través de la librería estándar de Julia: un pequeño conjunto de métodos requeridos definen una interfaz informal que permite muchos comportamientos muy atractivos. En algunos casos, los tipos que quieran especializar esos comportamientos extra cuando saben que existe un algoritmo más eficiente que podrán usar en su caso específico.

18.2 Indexación

Métodos a implementar	Breve descripción
`getindex(X, i)`	`X[i]`, acceso indexado a elemento
`setindex!(X, v, i)`	`X[i] = v`, asignación indexada
`endof(X)`	El último índice, usado en `X[end]`

Para el iterable `Squares` anterior, podemos calcular fácilmente el i-ésimo elemento de la secuencia elevándolo al cuadrado. Pordemos exponer esto como una expresión de indexación `S[i]`. Para optar a ese comportamiento, `Squares` sólo tieen que definir `getindex()`:

```
julia> function Base.getindex(S::Squares, i::Int)
           1 <= i <= S.count || throw(BoundsError(S, i))
           return i*i
       end

julia> Squares(100)[23]
529
```

Adicionalmente, para soportar la sintaxis `S[end]`, debemos definir `endof()` para especificar el último índice válido:

```
julia> Base.endof(S::Squares) = length(S)

julia> Squares(23)[end]
529
```

Tenga en cuenta, sin embargo, que lo anterior *sólo* define `getindex()` con un índice entero. Indexar con cualquier cosa que no sea un `Int` lanzará un `MethodError` diciendo que no había ningún método coincidente. Para soportar la indexación con intervalos o vectores de `Int`s, se deben escribir métodos separados:

```
julia> Base.getindex(S::Squares, i::Number) = S[convert(Int, i)]

julia> Base.getindex(S::Squares, I) = [S[i] for i in I]

julia> Squares(10)[[3,4.,5]]
3-element Array{Int64,1}:
  9
 16
 25
```

Aunque que esto está comenzando a soportar más de las operaciones de indexación soportadas por algunos de los tipos incorporados, todavía hay un buen número de comportamientos ausentes. Esta secuencia `Squares` está empezando a parecer más y más como un vector, ya que hemos añadido comportamientos a la misma. En lugar de definir todos estos comportamientos nosotros mismos, podemos definirlos oficialmente como un subtipo de un `AbstractArray`.

18.3 Abstract Arrays

Si un tipo se define como subtipo de `AbstractArray`, hereda un conjunto muy grande de comportamientos ricos, incluyendo la iteración y la indexación multidimensional construida sobre el acceso de un solo elemento. Consulte la página de manual sobre arrays y la sección de la biblioteca estándar para más métodos soportados.

Una parte clave en la definición de un subtipo de `AbstractArray` es `IndexStyle`.Dado que la indexación es una parte tan importante de una matriz y que a menudo se produce en los bucles en caliente, es importante que tanto la indexación y la asignación indexada sean lo más eficientes posible. Las estructuras de datos array se suelen definir de dos maneras: o bien accede de forma más eficaz a sus elementos utilizando sólo un índice (indexación lineal) o accede intrínsecamente a los elementos con índices especificados para cada dimensión. Estas dos modalidades son identificadas por Julia como `IndexLinear()` e `IndexCartesian()`. La conversión de un índice lineal en subíndices de indexación múltiples suele ser muy costosa, por lo que esto proporciona un mecanismo basado en tratos para permitir un código genérico eficiente para todos los tipos de matriz.

Esta distinción determina qué métodos de indexación escalar debe definir cada tipo. Los arrays `IndexLinear()` son sencillos: sólo definen `getindex(A::ArrayType, i::Int)`. Cuando el array se indexa posteriormmente con un conjunto multidimensional de índices, el método de respaldo `getindex(A::AbstractArray, I...)()` convierte eficientemente los indices en un indice lineal y luego llama al metodo anterior. Los arrays `IndexCartesian()`, por otra parte, requieren que se definan métodos para cada dimensionalidad soportada con `ndims(A)` índices Int. Por ejemplo, el tipo `SparseMatrixCSC` incorporado sólo admite dos dimensiones, por lo que sólo define `getindex(A::SparseMatrixCSC, i::Int, j::Int)`. Lo mismo sucede para `setindex!()`.

Volviendo a la secuencia de cuadrados de arriba, podríamos definirla como un subtipo de un `AbstractArray{Int, 1}`:

```
julia> struct SquaresVector <: AbstractArray{Int, 1}
           count::Int
       end

julia> Base.size(S::SquaresVector) = (S.count,)
```

Metodos a implementar		Breve descripción
`size(A)`		Devuelve una tupla que contiene las dimensiones de `A`
`getindex(A, i::Int)`		(if `IndexLinear`) Linear scalar indexing
`getindex(A, I::Vararg{Int, N})`		(if `IndexCartesian`, where `N = ndims(A)`) N-dimensional scalar indexing
`setindex!(A, v, i::Int)`		(if `IndexLinear`) Scalar indexed assignment
`setindex!(A, v, I::Vararg{Int, N})`		(if `IndexCartesian`, where `N = ndims(A)`) N-dimensional scalar indexed assignment
Optional methods	**Default definition**	**Brief description**
`IndexStyle(::Type)`	`IndexCartesian()`	Returns either `IndexLinear()` or `IndexCartesian()`. See the description below.
`getindex(A, I...)`	defined in terms of scalar `getindex()`	Multidimensional and nonscalar indexing
`setindex!(A, I...)`	defined in terms of scalar `setindex!()`	Multidimensional and nonscalar indexed assignment
`start()/next()/done()`	defined in terms of scalar `getindex()`	Iteration
`length(A)`	`prod(size(A))`	Number of elements
`similar(A)`	`similar(A, eltype(A), size(A))`	Return a mutable array with the same shape and element type
`similar(A, ::Type{S})`	`similar(A, S, size(A))`	Return a mutable array with the same shape and the specified element type
`similar(A, dims::NTuple{Int})`	`similar(A, eltype(A), dims)`	Return a mutable array with the same element type and size *dims*
`similar(A, ::Type{S}, dims::NTuple{Int})`	`Array{S}(dims)`	Return a mutable array with the specified element type and size
Non-traditional indices	**Default definition**	**Brief description**
`indices(A)`	`map(OneTo, size(A))`	Return the `AbstractUnitRange` of valid indices
`Base.similar(A, ::Type{S}, inds::NTuple{Ind})`	`similar(A, S, Base.to_shape(inds))`	Return a mutable array with the specified indices `inds` (see below)
`Base.similar(T::Union{Type,Function}, inds)`	`T(Base.to_shape(inds))`	Return an array similar to `T` with the specified indices `inds` (see below)

```
julia> Base.IndexStyle(::Type{<:SquaresVector}) = IndexLinear()

julia> Base.getindex(S::SquaresVector, i::Int) = i*i
```

Note que es muy importante especificar los dos parámetros del `AbstractArray`; la primera define el tipo de elemento `eltype()`, y la segunda el número de dimensiones `ndims()`. Este supertipo y sus tres métodos son todo lo que hace falta para que `SquaresVector` sea un array iterable, indexable y completamente funcional:

```
julia> s = SquaresVector(7)
```

```
7-element SquaresVector:
  1
  4
  9
 16
 25
 36
 49

julia> s[s .> 20]
3-element Array{Int64,1}:
 25
 36
 49

julia> s \ [1 2; 3 4; 5 6; 7 8; 9 10; 11 12; 13 14]
1×2 Array{Float64,2}:
 0.305389  0.335329

julia> s  s # dot(s, s)
4676
```

Un ejemplo un poco más complicado, definamos nuestro propio tipo array *sparse* N-dimensional "de juguete", construído encima de `Dict`:

```
julia> struct SparseArray{T,N} <: AbstractArray{T,N}
           data::Dict{NTuple{N,Int}, T}
           dims::NTuple{N,Int}
       end

julia> SparseArray{T}(::Type{T}, dims::Int...) = SparseArray(T, dims);

julia> SparseArray{T,N}(::Type{T}, dims::NTuple{N,Int}) = SparseArray{T,N}(Dict{NTuple{N,Int}, T
    }(), dims);

julia> Base.size(A::SparseArray) = A.dims

julia> Base.similar(A::SparseArray, ::Type{T}, dims::Dims) where {T} = SparseArray(T, dims)

julia> Base.getindex(A::SparseArray{T,N}, I::Vararg{Int,N}) where {T,N} = get(A.data, I, zero(T))

julia> Base.setindex!(A::SparseArray{T,N}, v, I::Vararg{Int,N}) where {T,N} = (A.data[I] = v)
```

Observe que se trata de un array `IndexCartesian` array, por lo que debemos definir manualmente `getindex()` y `setindex!()` en la dimensionalidad de la matriz. En este caso, a diferencia de en `SquaresVector`, somos capaces de definir `setindex!()` y, en consecuencia, podemos mutar el array:

```
julia> A = SparseArray(Float64, 3, 3)
3×3 SparseArray{Float64,2}:
 0.0  0.0  0.0
 0.0  0.0  0.0
 0.0  0.0  0.0

julia> fill!(A, 2)
3×3 SparseArray{Float64,2}:
 2.0  2.0  2.0
 2.0  2.0  2.0
```

```
 2.0  2.0  2.0

julia> A[:] = 1:length(A); A
3×3 SparseArray{Float64,2}:
 1.0  4.0  7.0
 2.0  5.0  8.0
 3.0  6.0  9.0
```

El resultado de la indexación de un `AbstractArray` puede ser en sí mismo un array (por ejemplo, al indexar por un rango). Los métodos de respaldo de `AbstractArray` utilizan `similar()` para asignar un `Array` del tamaño y tipo de elemento apropiados, que se rellena usando el método de indexación básico descrito anteriormente. Sin embargo, al implementar un *wrapper* de array, a menudo deseamos que el resultado sea también un *wrapper*:

```
julia> A[1:2,:]
2×3 SparseArray{Float64,2}:
 1.0  4.0  7.0
 2.0  5.0  8.0
```

En este ejemplo esto se logra mediante la definición de `Base.similar{T}(A::SparseArray, ::Type{T}, dims::Dims)` para crear la matriz wrapped apropiada. (Tenga en cuenta que aunque `similar` soporta formas de 1 y 2 argumentos, en la mayoría de los casos sólo necesita especializar el formulario de 3 argumentos). Para que esto funcione es importante que `SparseArray` sea mutable (soporte `setindex!`). Definir `similar()`, `getindex()` y `setindex!()` para `SparseArray` también hace posible copiar el array mediante `copy()`:

```
julia> copy(A)
3×3 SparseArray{Float64,2}:
 1.0  4.0  7.0
 2.0  5.0  8.0
 3.0  6.0  9.0
```

Además de todos los métodos iterables e indexables de arriba, estos tipos también pueden interactuar entre sí y utilizar todos los métodos definidos en la biblioteca estándar para `AbstractArrays`:

```
julia> A[SquaresVector(3)]
3-element SparseArray{Float64,1}:
 1.0
 4.0
 9.0

julia> dot(A[:,1],A[:,2])
32.0
```

Si está definiendo un tipo de array que permite la indexación no tradicional (índices que comienzan en algo distinto de 1), debe especializar `indices`. También debe especializarse `similar` para que el argumento `dims` (normalmente una tupla de tamaños `Dims`) pueda aceptar objetos `AbstractUnitRange`, tal vez rango-tipos `Ind` de su propio diseño. Para obtener más información, vea Arrays con índices personalizados.

Chapter 19

Módulos

Los módulos en Julia son espacio de trabajo de variables separados, es decir, ellos introducen un nuevo ámbito global. Ellos están delimitados sintácticamente dentro de `module Nombre ... end`. Los módulos nos permiten crear definiciones de nivel superior (o variables globales) sin preocuparnos sobre conflictos de nombres cuando estamos usando nuestro código con cualquier otro. Dentro de un módulo, podemos controlar qué nombres de otros módulos son visibles (vía importación) y especificar cúales de nuestros nombres queremos que sean públicos (vía exportación).

El siguiente ejemplo muestra las principales características de los módulos. No está destinado para ser ejecutado, sino que se muestra para propósitos ilustrativos:

```
module MyModule
using Lib

using BigLib: thing1, thing2

import Base.show

importall OtherLib

export MyType, foo

struct MyType
    x
end

bar(x) = 2x
foo(a::MyType) = bar(a.x) + 1

show(io::IO, a::MyType) = print(io, "MyType $(a.x)")
end
```

Notese que el estilo es no indentar el cuerpo del módulo, ya que esto llevaría a que el fichero completo estuviera indentado.

Este módulo define un tipo `MyType` y dos funciones. Las funciones `foo` y `MyType` son exportadas y, en consecuencia, estará disponibles para ser importadas en otros módulos. La función `bar` es privada a `MyModule`.

La instrucción `using Lib` significa que un módulo llamado `Lib` estará disponible para resover nombres cuando se necesite. Cuando se encuentra que una variable no tiene definición en el módulo actual, el sistema la buscará entre las variables exportadas por `Lib` y la importará si la encuentra allí. Esto significa que todos los usos de esta global dentro del módulo actual resoverá a la definición de esta variable en `Lib`.

La instrucción `using BigLib: thing1, thing2` es una abreviación sintáctica de `using BigLib.thing1, BigLib.thing2`.

La palabra clave `import` soporta la misma sintaxis que `using`, pero sólo opera sobre un solo nombre cada vez. Ella no añade módulos para ser buscados de la forma que lo hace `using`. También difiere de `using` en que las funciones deben ser importandas mediante `import` para ser extendidas con nuevos métodos.

En el jemplo anterior `MyModule` deseamos añadir un método a la función estándar `show`, por lo que tenemos que escribir `import Base.show`. Las funciones cuyos nombres son sólo visibles via `using` no pueden ser extendidas.

La palabra clave `importall` importa explícitamente todos los nombres exportados por el módulo especificado, con si se hubiera usado `import` individualmente sobre cada uno de ellos.

Una vez que una variable se ha hecho visible vía `using` o `import`, un módulo puede no crear su propia variable con el mismo nombre. Las variables importadas son de solo lectura; asignar a una variable global siempre afecta a una variable propiedad del módulo actual, o si no causar un error.

19.1 Resumen de uso de los módulos

Para cargar un módulo, pueden usarse dos palabras clave principales: `using` e `import`. Para comprender sus diferencias, considérese el siguiente ejemplo:

```
module MyModule

export x, y

x() = "x"
y() = "y"
p() = "p"

end
```

En este módulo exportamos las funciones x e y (con la palabra clave `export`) y también tenemos la función no exportada p. Hay varias diferentes formas de cargar el mósulo y sus funciones internas en el espacio de trabajo actual:

Mandato de importación	Qué se introduce en el ámbito	Disponible para extensión de método
`using MyModule`	All `exported` names (x and y), `MyModule.x`, `MyModule.y` and `MyModule.p`	`MyModule.x`, `MyModule.y` and `MyModule.p`
`using MyModule.x, MyModule.p`	x and p	
`using MyModule: x, p`	x and p	
`import MyModule`	`MyModule.x`, `MyModule.y` and `MyModule.p`	`MyModule.x`, `MyModule.y` and `MyModule.p`
`import MyModule.x, MyModule.p`	x and p	x and p
`import MyModule: x, p`	x and p	x and p
`importall MyModule`	All `exported` names (x and y)	x and y

Módulos y ficheros

Los ficheros y los nombres de ficheros no están relacionados con los módulos. Los módulos están asociados sólo con las expresiones `module`. Uno puede tener múltiples ficheros por módulo, y múltiples módulos por fichero:

```
module Foo

include("file1.jl")
include("file2.jl")

end
```

Incluir el mismo código en módulos diferentes porporciona un comportamiento similar a la mezcla. Uno podría usar esto para ejecutar el mismo código con diferentes definiciones base, por ejemplo, código de test ejecutando una versión ”segura” de algunos operadores:

```
module Normal
include("mycode.jl")
end

module Testing
include("safe_operators.jl")
include("mycode.jl")
end
```

Módulos estándar

Hay tres módulos estándar importantes: Main, Core y Base.

`Main` es el módulo de nivel superior, y Julia arranca con Main fijado como módulo actual. Las variables definidas en el prompt van a Main, y `whos()` lista las variables en Main.

`Core` contiene todos los identificadores considerados predefinidos en el lenguaje, por ejemplo, parte del lenguaje y no librerías. Cada módulo especifica implícitamente `using Core`, ya que uno no puede hacer nada sin sus definiciones.

`Base` es la librería estándar (los contenidos de `base/`). Todos los módulos contiene implícitamente `using Base`, ya que este se necesita en la gran mayoría de los casos.

Definiciones de nivel superior por defecto y módulos esenciales (*bare*)

Además de a `using Base`, los módulos también contiene automáticamente una definició de la función `eval`, que evalúa expresiones dentro del contexto de este módulo.

Si estas definiciones por defecto no son deseadas, los módulos pueden ser definidos usando la palabra clave `baremodule`(nota: `Core` sigue siendo importando, como antes). En términos de `baremodule` un módulo estándar tiene este aspecto.

```
baremodule Mod

using Base

eval(x) = Core.eval(Mod, x)
eval(m,x) = Core.eval(m, x)

...

end
```

Caminos absolutos y relativos de módulos

Dada la instrucción `using Foo`, el sistema buscará `Foo` dentro de `Main`. Si el módulo no existe, el sistema interará un `require("Foo")` que típicamente da como resultado cargar código desde un paquete instalado.

Sin embargo, algunos módulos contienen submódulos, lo que significa que tu algunas veces tienes que acceder a un módulo que no está directamente disponible en `Main`. Hay dos formas de hacer esto. La primera es usar un camino absoluto, por ejemplo `using Base.Sort`. El segundo es usar un camino relativo, que hace más fácil importar submódulos del módulo actual o alguno de sus módulos adjuntos:

```
module Parent

module Utils
...
end

using .Utils

...
end
```

Aquí, el módulo `Parent` contiene un submódulo `Utils` y el código en `Parent` quiere que el contenido de `Utils` esté visible. Este se consigue empezando la instrucción `using` con un punto. Ada punto adicional añadido se muevo hacia arriba niveles adicionales en la jerarquía de módulos. Por ejemplo, `using ..Utils` buscaría en el módulo que contiene a `Parent`, no en el propio `Padre`.

Notese que los cualificadores de importación relativos son sólo válidos para las instrucciones `using` e `import`.

Caminos de ficheros de módulo

La variable global `LOAD_PATH` contiene los directorios donde Julia busca módulos cada vez que se invoca `require`. Esto puede extenderse usandog `push!`:

```
push!(LOAD_PATH, "/Path/To/My/Module/")
```

Aquí, el módulo `Parent` contiene un submódulo `Utils` y el código en `Parent` quiere que el contenido de `Utils` esté visible. Este se consigue empezando la instrucción `using` con un punto. Ada punto adicional añadido se muevo hacia arriba niveles adicionales en la jerarquía de módulos. Por ejemplo, `using ..Utils` buscaría en el módulo que contiene a `Parent`, no en el propio `Padre`.

Notese que los cualificadores de importación relativos son sólo válidos para las instrucciones `uing` e `import`.

Caminos de ficheros de módulo

La variable global `LOAD_PATH` contiene los directorios donde Julia busca módulos cada vez que se invoca `require`. Esto puede extenderse usandog `push!`:

```
push!(LOAD_PATH, "/Path/To/My/Module/")
```

Poniendo esta instrucción en el ficehro `~/.juliarc.jl` extenderemos la variable `LOAD_PATH` on every Julia startup. en cada arranque de Julia. Alternativamente, el camino de carga de módulos puede ser extendido definiendo la variable de entorno `JULIA_LOAD_PATH`.

Miscelánea sobre espacios de nombres

Si un nombre está cualificado (por ejemplo `Base.sin`) entonces puede ser accedido incluso aunque no esté exportado. Esto es bastante útil de cara a depuración.

También puede haber métodos agregados al usar el nombre calificado como el nombre de la función. Sin embargo, debido a las ambigüedades sintácticas que surgen, si uno desea agregar métodos a una función en un módulo diferente cuyo nombre contiene solo símbolos, tal como un operador (por ejemplo, `Base.+`), debe usar la notación `Base.:+` para referirse a él. Si el operador tiene más de un carácter de longitud, debe rodearlo entre paréntesis, por ejemplo: `Base.:(==)`.

Los nombres de macros se escriben con `@` en las instrucciones de importación y exportación. Por ejemplo, `import Mod.@mac`. Las macros en otros módulos pueden ser invocadas como `Mod.@mac` or `@Mod.mac`.

La sintaxis `M.x=y` no funciona para asignar una variable global en otro módulo: la asignación global es siempre local a un módulo.

Una variable puede estar ”reservada” por el módulo actual sin asignar a ella declarándola como `global x` en el nivel superior. Esto puede usarse para prevenir conflictos de nombreas para globales inicializadas después del tiempo de carga.

Inicialización y precompilación de módulos

Los módulos grandes puede necesitar varios segundos para cargar, debido a que ejecutar todas las instrucciones en un módulo implica compilar una gran candidad de código. Julia proporciona la capacidad de crear versiones precompiladas de los módulos para reducir este timepo.

Para crear un fichero de módulo precompilado incremental, añadimos `__precompile--()` al principio de nuestro fichero de módulo (antes de que empiece la palabra `module`). Esto causará que él sea compilado automáticamente la primera vez que se importe. Alternativamete, podemos llamar a `Base.compilecache(modulename)`. Los ficheros caché resultantes se almacenarán en `Base.LOAD_CACHE_PATH[1]`. Posteriormente, el módulo es recompilado automáticamente con `import` cada vez que alguna de sus dependencia cambia; las dependencias son módulos en imports, la construcción de Julia, los ficheros que incluye, o dependencias explícitas declaradas con `include_dependency(path)` en el fichero o ficheros de módulo.

Para dependencias de fichero, un cambio se determina examinando si el momento de modificación (mtime) de cada fichero cargado por `include` o añadido explícitamente mediante `include_dependency` está sin cambios o igual al tiempo de modificaci´no truncado al segundo más cercano (para acomodar sistemas que no pueden copiar mtime con exactitud menor que el segundo). También se tiene en cuenta si el camino al fichero elegido por la lógica de búsqueda en `require` se corresponde con el camino que había creado el fichero precompilado.

También se tiene en cuenta el conjunto de dependencias ya cargadas en el proceso actual y no recompilará esos módulos, incluso si sus ficheros cambian o desaparecen, para evitar crear incompatibilidades entre el sistema en ejecución y la caché precompilada. I queremos tener cambios en la fuente reflejados en el sistema en ejecución, deberíamos llamar a `reload("Module")` sobre el módulo que hemos cambiado, y cualquier módulo que dependa de él en el cuál quieres ver reflejado el cambio.

Precompilar un módulo también recursivamente precompila todo los módulos que son importados allí. Si sabemos que no es seguro precompilar nuestor módulo (por las razones descritas debajo) deberíamos poner `__precompile__(false)` en el fichero del módulo para causar que `Base.compilecache` lance un error (y por tanto evite que el módulo sea importado por cualquier otro módulo precompilado.

`__precompile` **NO** debería ser usado en un módulo a menos que todas sus dependencias estén también usando `__precompile__()`. Un fallo en hacer esto puede dar como resultado un error en tiempo de ejecución cuando se carga el módulo.

Para hacer que nuestro módulo funcione con la precompilación, sin embargo, necesitamos cambiar nuestro módulo para separar explícitamente cualquier paso de inicialización que deba ocurrir en tiempo de ejecución de pasos que

pueden ocurrir en tiempo de compilación. Para este propósito, Julia te permite definir una función `__init__()` en tu módulo que ejecuta cualquier paso de inicialización que deba tener lugar en tiempo de ejecución. Esta función no será llaada durante la compilación (`--ouput-*` o `__precompile__()`). Podemos, por supuesto, llamarla manualmente si es necesario, pero el comportamiento por defecto es asumir que esta función trata con el estado de computación para la máquina local, que no necesita ser (o incluso no debe ser) capturada en la imagen compilada. Ella puede ser llamada después de que el módulo sea cargado en un proceso, incluyendo si el está siendo cargado en una compilación incremental (`--output-incremental=yes`), pero no si está siendo cargado en un proceso de compilación completo.

En particular, si defines un `function __init__()` en un módulo, entonces Julia llamará a `__init__()` inmediatamente después de que el módulo se cargue (es decir, mediante `import`, `using` o `require`) en tiempo de ejecución la *primera* vez (es decir, `__init__` sólo es llamado una vez, y sólo después de que todas las instrucciones en el módulo se hayan ejecutado). Como el es llamado después de que el módulo sea importado por completo, los submódulos u otros módulos importandos tienen sus funciones `__init__`llamadas antes del `__init__`del módulo contenedor.

Dos usos típicos de `__init__` son llamar a funciones de inicialización en tiempo de ejecución de librerías externas en C e inicializar constantes globales que implican punteros devueltos por las librerías externas. Por ejemplo, supongamos que estamos llamado a una librería en C `libfoo` que requiere que llamemos a la función de inicialización `foo_init()` en tiempo de ejecución. Supongamos que también queremos definir una constante global `foo_data_ptr` que almacena el valor de retorno de una función `void *foo_data()` definida por `libfoo` (esta constante tiene que se inicializada en tiempo de ejecución - no de compilación- debido a que el puntero cambiará de una ejecución a otra). Podríamos conseguir esto definiendo la siguiente función `__init__`en nuestro módulo:

```
const foo_data_ptr = Ref{Ptr{Void}}(0)
function __init__()
    ccall((:foo_init, :libfoo), Void, ())
    foo_data_ptr[] = ccall((:foo_data, :libfoo), Ptr{Void}, ())
end
```

Notese que es perfectamente posible definir un global dentro de la función como `__init__`; esta es una de las ventajas de usar un lenguaje dinámico. Pero haciéndolo una constante en un ámbito global, podemos asegurar que el tipo es conocido al compilador y le permite generar código mejor optimizado. Obviamente, otros globales de nuestro módulo que dependan de `foo_data_ptr` también tendrían que ser inicializados en `__init__`.

Las constantes que implican a la mayoría de los objetos Julia que no son producidas por `ccall` no necesitan se colocadas en `__init__`: sus definiciones pueden ser precompiladas y cargadas desde la imagen cacheada del módulo. Esto incluye complicados objetos alojados en el montón (*heap*) como los arrays. Sin embargo, cualquier rutina que devuelva un calor de puntero crudo debe ser llamada en tiempo de ejecución para que la precompilación funcione (los objetos Ptr se convertirán en punteros nulos a menos que sean ocultados dentro de un objeto isbits). Esto incluye los valore de retorno de las funciones Julia `cfunction` y `pointer`.

Los tipos diccionario y conjunto, o en general cualquiera que dependa de la salida de un método `hash(key)`, son un caso más difícil. En el caso común de que las claves sean números, cadenas, símbolos, rangos, `Expr` o composiciones de estos tipos (vía arraysm, tuplas, conjuntos, pares, etc.) son seguros de precompilar. Sin embargo, para otros tipos clave, tales como `Function` o `DataType` y tipos genéricos definidos por el usuario donde no se ha definido un método `hash` el método `hash` de apoyo depende de la dirección en memoria del objeto (vía su `object_id`) y por tanto puede cambiar de ejecución a ejecución. Si tienes uno de esos tipos como clave, o no estás seguro, puedes inicializar este diccionario desde dentro de tu función `__init__` para mayor seguridad. Alternativamente, puedes usar el tipo diccionario `ObjectIdDict`, que es especialmente manejado por precompilación por lo que es seguro de inicializar en tiempo de compilación.

Cuando se usa precompilación, es importante mantener un claro sentido de la distinción entre la fase de compilación y la de ejecución. En este modo, a menudo será mucho más evidente que Julia es un compilador que permite la ejecución de código arbitrario Julia, no un intérprete independiente que también genera código compilado.

Otros escenarios de fallo potencial conocidos incluyen:

1. Contadores globales (por ejemplo, para intentar identificar objetos únicamente). Considere el siguiente código:

```
mutable struct UniquedById
    myid::Int
    let counter = 0
        UniquedById() = new(counter += 1)
    end
end
```

   mientras que la intención de este código era dar a cada instancia un id único, el valor del contador es grabado al final de la compilación. Todos los usos posteriores de este módulo compilado incrementalmente empezarán desde el mismo valor de contador.

Note que `object_id` (que trabaja haciendo *hash* en el puntero a memoria) tieen problemas similares (ver las notas sobre el uso de `Dict` abajo).

One alternative is to store both `current_module()` and the current `counter` value, sin embargo, puede ser mejor rediseñar el código para no depender de este estado global.

2. Las colecciones asociativas (tales como `Dict` y `Set`) necesitan ser re-hasheadas en `__init__` (en el futuro, puede proporcionarse un mecanismo para registrar un inicializador de función).

3. Dependiendo de los efectos secundarios de tiempo de compilación que persisten a través del tiempo de carga. El ejemplo incluye: modificar arrays u otras variables en otros módulos de Julia; mantener manejadores para abrir archivos o dispositivos; almacenar punteros a otros recursos del sistema (incluyendo memoria);

4. Crear "copias" accidentales de estado global desde otro módulo, haciendo referencia directamente a él en vez de a través de su ruta de búsqueda. Por ejemplo, (en el ámbito global).

```
#mystdout = Base.STDOUT #= will not work correctly, since this will copy Base.STDOUT into
↪  this module =#
# instead use accessor functions:
getstdout() = Base.STDOUT #= best option =#
# or move the assignment into the runtime:
__init__() = global mystdout = Base.STDOUT #= also works =#
```

Varias restricciones adicionales se colocan sobre las operaciones que se pueden hacer mientras se precompila el código para ayudar al usuario a evitar otras situaciones de comportamiento incorrecto:

1. Llamar `eval` para provocar un efecto secundario en otro módulo. Esto también provocará que se emita una advertencia cuando se establece el indicador de precompilación incremental.

2. Las sentencias `global consts` del ámbito local después de `__init __()` se han iniciado (vea el problema #12010 para los planes de agregar un error para esto).

3. Reemplazar un módulo (o llamar `workspace`) es un error de tiempo de ejecución al realizar una precompilación incremental.

Algunos otros puntos a tener en cuenta:

1. Ninguna recarga de código / invalidación de caché se realiza después de que se realizan cambios en los propios archivos fuente (incluyendo `Pkg.update`) y no se realiza ninguna limpieza después de `Pkg.rm`.

2. El comportamiento de compartir la memoria de una matriz reestructurada es ignorado por la precompilación (cada vista obtiene su propia copia).

3. Esperar que el sistema de archivos no cambie entre tiempo de compilación y tiempo de ejecución, p.ej `@__FILE__` / `source_path()` para encontrar recursos en tiempo de ejecución, o la macro BinDeps @checked_lib. A veces esto es inevitable. Sin embargo, cuando sea posible, puede ser una buena práctica copiar recursos en el módulo en tiempo de compilación para que no sea necesario encontrarlos en tiempo de ejecución.

4. Los objetos `WeakRef` y los finalizadores no son manejados correctamente por el serializador (esto se arreglará en una próxima versión).

5. Por lo general, es mejor evitar la captura de referencias a instancias de objetos de metadatos internos como `Method`, `MethodInstance`, `MethodTable`, `TypeMapLevel`, `TypeMapEntry` y campos de esos objetos, ya que esto puede confundir al serializador y puede no conducir al resultado deseado. No es necesariamente un error hacer esto, pero sólo tiene que estar preparado para que el sistema intente copiar algunos de ellos y crear una única instancia única de otros.

A veces es útil durante el desarrollo del módulo para desactivar la precompilación incremental. El indicador de línea de comandos `--compilecache = {yes | no}` le permite activar y desactivar la precompilación del módulo. Cuando se inicia Julia con `--compilecache = no` se ignoran los módulos serializados en el caché de compilación al cargar módulos y dependencias de módulo. `Base.compilecache()` todavía se puede llamar manualmente y respetará las directivas `__precompile__()` para el módulo. El estado de este indicador de línea de comandos se pasa a `Pkg.build()` para deshabilitar el desencadenamiento automático de precompilación al instalar, actualizar y crear explícitamente paquetes. updating, and explicitly building packages.

Chapter 20

Documentation

Julia enables package developers and users to document functions, types and other objects easily via a built-in documentation system since Julia 0.4.

The basic syntax is very simple: any string appearing at the top-level right before an object (function, macro, type or instance) will be interpreted as documenting it (these are called *docstrings*). Here is a very simple example:

```
"Tell whether there are too foo items in the array."
foo(xs::Array) = ...
```

Documentation is interpreted as Markdown, so you can use indentation and code fences to delimit code examples from text. Technically, any object can be associated with any other as metadata; Markdown happens to be the default, but one can construct other string macros and pass them to the `@doc` macro just as well.

Here is a more complex example, still using Markdown:

```
"""
    bar(x[, y])

Compute the Bar index between `x` and `y`. If `y` is missing, compute
the Bar index between all pairs of columns of `x`.

# Examples
```julia-repl
julia> bar([1, 2], [1, 2])
1
```
"""
function bar(x, y) ...
```

As in the example above, we recommend following some simple conventions when writing documentation:

1. Always show the signature of a function at the top of the documentation, with a four-space indent so that it is printed as Julia code.

   This can be identical to the signature present in the Julia code (like `mean(x::AbstractArray)`), or a simplified form. Optional arguments should be represented with their default values (i.e. `f(x, y=1)`) when possible, following the actual Julia syntax. Optional arguments which do not have a default value should be put in brackets (i.e. `f(x[, y])` and `f(x[, y[, z]])`). An alternative solution is to use several lines: one without optional

arguments, the other(s) with them. This solution can also be used to document several related methods of a given function. When a function accepts many keyword arguments, only include a `<keyword arguments>` placeholder in the signature (i.e. `f(x; <keyword arguments>)`), and give the complete list under an `# Arguments` section (see point 4 below).

2. Include a single one-line sentence describing what the function does or what the object represents after the simplified signature block. If needed, provide more details in a second paragraph, after a blank line.

   The one-line sentence should use the imperative form ("Do this", "Return that") instead of the third person (do not write "Returns the length...") when documenting functions. It should end with a period. If the meaning of a function cannot be summarized easily, splitting it into separate composable parts could be beneficial (this should not be taken as an absolute requirement for every single case though).

3. Do not repeat yourself.

   Since the function name is given by the signature, there is no need to start the documentation with "The function `bar`...": go straight to the point. Similarly, if the signature specifies the types of the arguments, mentioning them in the description is redundant.

4. Only provide an argument list when really necessary.

   For simple functions, it is often clearer to mention the role of the arguments directly in the description of the function's purpose. An argument list would only repeat information already provided elsewhere. However, providing an argument list can be a good idea for complex functions with many arguments (in particular keyword arguments). In that case, insert it after the general description of the function, under an `# Arguments` header, with one – bullet for each argument. The list should mention the types and default values (if any) of the arguments:

   ```
   """
   ...
   # Arguments
   - `n::Integer`: the number of elements to compute.
   - `dim::Integer=1`: the dimensions along which to perform the computation.
   ...
   """
   ```

5. Include any code examples in an `# Examples` section.

   Examples should, whenever possible, be written as *doctests*. A *doctest* is a fenced code block (see Code blocks) starting with ```` ```jldoctest ```` and contains any number of `julia>` prompts together with inputs and expected outputs that mimic the Julia REPL.

   For example in the following docstring a variable a is defined and the expected result, as printed in a Julia REPL, appears afterwards:

   ````
   """
   Some nice documentation here.

   # Examples

   ```jldoctest
   julia> a = [1 2; 3 4]
   2×2 Array{Int64,2}:
    1  2
    3  4
   ```
   """
   ````

> **Warning**
>
> Calling `rand` and other RNG-related functions should be avoided in doctests since they will not produce consistent outputs during different Julia sessions.
>
> Operating system word size (`Int32` or `Int64`) as well as path separator differences (`/` or `\`) will also affect the reproducibility of some doctests.
>
> Note that whitespace in your doctest is significant! The doctest will fail if you misalign the output of pretty-printing an array, for example.

You can then run `make -C doc doctest` to run all the doctests in the Julia Manual, which will ensure that your example works.

Examples that are untestable should be written within fenced code blocks starting with ```` ```julia ```` so that they are highlighted correctly in the generated documentation.

> **Tip**
>
> Wherever possible examples should be **self-contained** and **runnable** so that readers are able to try them out without having to include any dependencies.

6. Use backticks to identify code and equations.

   Julia identifiers and code excerpts should always appear between backticks `` ` `` to enable highlighting. Equations in the LaTeX syntax can be inserted between double backticks ``` `` ```. Use Unicode characters rather than their LaTeX escape sequence, i.e. ``` ``α = 1`` ``` rather than ``` ``\\alpha = 1`` ```.

7. Place the starting and ending `"""` characters on lines by themselves.

   That is, write:

   ```
   """
   ...

   ...
   """
   f(x, y) = ...
   ```

   rather than:

   ```
   """...

   ..."""
   f(x, y) = ...
   ```

   This makes it more clear where docstrings start and end.

8. Respect the line length limit used in the surrounding code.

   Docstrings are edited using the same tools as code. Therefore, the same conventions should apply. It it advised to add line breaks after 92 characters.

20.1 Accessing Documentation

Documentation can be accessed at the REPL or in IJulia by typing `?` followed by the name of a function or macro, and pressing `Enter`. For example,

```
?fft
?@time
?r""
```

will bring up docs for the relevant function, macro or string macro respectively. In Juno using `Ctrl-J, Ctrl-D` will bring up documentation for the object under the cursor.

20.2 Functions & Methods

Functions in Julia may have multiple implementations, known as methods. While it's good practice for generic functions to have a single purpose, Julia allows methods to be documented individually if necessary. In general, only the most generic method should be documented, or even the function itself (i.e. the object created without any methods by `function bar end`). Specific methods should only be documented if their behaviour differs from the more generic ones. In any case, they should not repeat the information provided elsewhere. For example:

```
"""
    *(x, y, z...)

Multiplication operator. `x * y * z *...` calls this function with multiple
arguments, i.e. `*(x, y, z...)`.
"""
function *(x, y, z...)
    # ... [implementation sold separately] ...
end

"""
    *(x::AbstractString, y::AbstractString, z::AbstractString...)

When applied to strings, concatenates them.
"""
function *(x::AbstractString, y::AbstractString, z::AbstractString...)
    # ... [insert secret sauce here] ...
end

help?> *
search: * .*

  *(x, y, z...)

  Multiplication operator. x * y * z *... calls this function with multiple
  arguments, i.e. *(x,y,z...).

  *(x::AbstractString, y::AbstractString, z::AbstractString...)

  When applied to strings, concatenates them.
```

When retrieving documentation for a generic function, the metadata for each method is concatenated with the `catdoc` function, which can of course be overridden for custom types.

20.3 Advanced Usage

The `@doc` macro associates its first argument with its second in a per-module dictionary called `META`. By default, documentation is expected to be written in Markdown, and the `doc""` string macro simply creates an object representing the Markdown content. In the future it is likely to do more advanced things such as allowing for relative image or link paths.

When used for retrieving documentation, the `@doc` macro (or equally, the `doc` function) will search all `META` dictionaries for metadata relevant to the given object and return it. The returned object (some Markdown content, for example)

will by default display itself intelligently. This design also makes it easy to use the doc system in a programmatic way; for example, to re-use documentation between different versions of a function:

```
@doc "..." foo!
@doc (@doc foo!) foo
```

Or for use with Julia's metaprogramming functionality:

```
for (f, op) in ((:add, :+), (:subtract, :-), (:multiply, :*), (:divide, :/))
    @eval begin
        $f(a,b) = $op(a,b)
    end
end
@doc "`add(a,b)` adds `a` and `b` together" add
@doc "`subtract(a,b)` subtracts `b` from `a`" subtract
```

Documentation written in non-toplevel blocks, such as `begin`, `if`, `for`, and `let`, is added to the documentation system as blocks are evaluated. For example:

```
if VERSION > v"0.5"
    "..."
    f(x) = x
end
```

will add documentation to `f(x)` when the condition is `true`. Note that even if `f(x)` goes out of scope at the end of the block, its documentation will remain.

Dynamic documentation

Sometimes the appropriate documentation for an instance of a type depends on the field values of that instance, rather than just on the type itself. In these cases, you can add a method to `Docs.getdoc` for your custom type that returns the documentation on a per-instance basis. For instance,

```
struct MyType
    value::String
end

Docs.getdoc(t::MyType) = "Documentation for MyType with value $(t.value)"

x = MyType("x")
y = MyType("y")
```

?x will display "Documentation for MyType with value x" while ?y will display "Documentation for MyType with value y".

20.4 Syntax Guide

A comprehensive overview of all documentable Julia syntax.

In the following examples " . . . " is used to illustrate an arbitrary docstring which may be one of the follow four variants and contain arbitrary text:

```
"..."

doc"..."

"""

...
"""

doc"""
...
"""
```

`@doc_str` should only be used when the docstring contains $ or \ characters that should not be parsed by Julia such as LaTeX syntax or Julia source code examples containing interpolation.

Functions and Methods

```
"..."
function f end

"..."
f
```

Adds docstring `"..."` to `Function f`. The first version is the preferred syntax, however both are equivalent.

```
"..."
f(x) = x

"..."
function f(x)
    x
end

"..."
f(x)
```

Adds docstring `"..."` to `Method f(::Any)`.

```
"..."
f(x, y = 1) = x + y
```

Adds docstring `"..."` to two `Methods`, namely `f(::Any)` and `f(::Any, ::Any)`.

Macros

```
"..."
macro m(x) end
```

Adds docstring `"..."` to the `@m(::Any)` macro definition.

```
"..."
:(@m)
```

Adds docstring `"..."` to the macro named `@m`.

Types

```
"..."
abstract type T1 end

"..."
mutable struct T2
    ...
end

"..."
struct T3
    ...
end
```

Adds the docstring `"..."` to types T1, T2, and T3.

```
"..."
struct T
    "x"
    x
    "y"
    y
end
```

Adds docstring `"..."` to type T, `"x"` to field `T.x` and `"y"` to field `T.y`. Also applicable to `mutable struct` types.

Modules

```
"..."
module M end

module M

"..."
M

end
```

Adds docstring `"..."` to the `ModuleM`. Adding the docstring above the `Module` is the preferred syntax, however both are equivalent.

```
"..."
baremodule M
# ...
end

baremodule M

import Base: @doc

"..."
f(x) = x

end
```

Documenting a `baremodule` by placing a docstring above the expression automatically imports `@doc` into the module. These imports must be done manually when the module expression is not documented. Empty `baremodules` cannot be documented.

Global Variables

```
"..."
const a = 1

"..."
b = 2

"..."
global c = 3
```

Adds docstring `"..."` to the `Bindings` a, b, and c.

`Bindings` are used to store a reference to a particular `Symbol` in a `Module` without storing the referenced value itself.

> **Note**
>
> When a `const` definition is only used to define an alias of another definition, such as is the case with the function `div` and its alias ÷ in `Base`, do not document the alias and instead document the actual function.
>
> If the alias is documented and not the real definition then the docsystem (`?` mode) will not return the docstring attached to the alias when the real definition is searched for.
>
> For example you should write
>
> ```
> "..."
> f(x) = x + 1
> const alias = f
> ```
>
> rather than
>
> ```
> f(x) = x + 1
> "..."
> const alias = f
> ```

```
"..."
sym
```

Adds docstring `"..."` to the value associated with `sym`. Users should prefer documenting `sym` at it's definition.

Multiple Objects

```
"..."
a, b
```

Adds docstring `"..."` to a and b each of which should be a documentable expression. This syntax is equivalent to

```
"..."
a

"..."
b
```

Any number of expressions many be documented together in this way. This syntax can be useful when two functions are related, such as non-mutating and mutating versions `f` and `f!`.

Macro-generated code

```
"..."
@m expression
```

Adds docstring `"..."` to expression generated by expanding `@m expression`. This allows for expressions decorated with `@inline`, `@noinline`, `@generated`, or any other macro to be documented in the same way as undecorated expressions.

Macro authors should take note that only macros that generate a single expression will automatically support docstrings. If a macro returns a block containing multiple subexpressions then the subexpression that should be documented must be marked using the `@__doc__` macro.

The `@enum` macro makes use of `@__doc__` to allow for documenting `Enum`s. Examining it's definition should serve as an example of how to use `@__doc__` correctly.

`Core.@__doc__` – Macro.

```
@__doc__(ex)
```

Low-level macro used to mark expressions returned by a macro that should be documented. If more than one expression is marked then the same docstring is applied to each expression.

```
macro example(f)
    quote
        $(f)() = 0
        @__doc__ $(f)(x) = 1
        $(f)(x, y) = 2
    end |> esc
end
```

`@__doc__` has no effect when a macro that uses it is not documented.

source

20.5 Markdown syntax

The following markdown syntax is supported in Julia.

Inline elements

Here "inline" refers to elements that can be found within blocks of text, i.e. paragraphs. These include the following elements.

Bold

Surround words with two asterisks, `**`, to display the enclosed text in boldface.

```
A paragraph containing a **bold** word.
```

Italics

Surround words with one asterisk, `*`, to display the enclosed text in italics.

```
A paragraph containing an *emphasised* word.
```

Literals

Surround text that should be displayed exactly as written with single backticks, `` ` ``.

```
A paragraph containing a `literal` word.
```

Literals should be used when writing text that refers to names of variables, functions, or other parts of a Julia program.

> **Tip**
>
> To include a backtick character within literal text use three backticks rather than one to enclose the text.
>
> ```
> A paragraph containing a ``` `backtick` character ```.
> ```
>
> By extension any odd number of backticks may be used to enclose a lesser number of backticks.

LaTeX

Surround text that should be displayed as mathematics using LaTeXsyntax with double backticks, ``` `` ```.

```
A paragraph containing some ``\LaTeX`` markup.
```

> **Tip**
>
> As with literals in the previous section, if literal backticks need to be written within double backticks use an even number greater than two. Note that if a single literal backtick needs to be included within LaTeXmarkup then two enclosing backticks is sufficient.

Links

Links to either external or internal addresses can be written using the following syntax, where the text enclosed in square brackets, `[ ]`, is the name of the link and the text enclosed in parentheses, `( )`, is the URL.

```
A paragraph containing a link to [Julia](http://www.julialang.org).
```

It's also possible to add cross-references to other documented functions/methods/variables within the Julia documentation itself. For example:

```
"""
    eigvals!(A,[irange,][vl,][vu]) -> values

Same as [`eigvals`](@ref), but saves space by overwriting the input `A`, instead of creating a
↪  copy.
"""
```

This will create a link in the generated docs to the `eigvals` documentation (which has more information about what this function actually does). It's good to include cross references to mutating/non-mutating versions of a function, or to highlight a difference between two similar-seeming functions.

> **Note**
>
> The above cross referencing is *not* a Markdown feature, and relies on Documenter.jl, which is used to build base Julia's documentation.

Footnote references

Named and numbered footnote references can be written using the following syntax. A footnote name must be a single alphanumeric word containing no punctuation.

```
A paragraph containing a numbered footnote [^1] and a named one [^named].
```

> **Note**
>
> The text associated with a footnote can be written anywhere within the same page as the footnote reference. The syntax used to define the footnote text is discussed in the Footnotes section below.

Toplevel elements

The following elements can be written either at the ”toplevel” of a document or within another ”toplevel” element.

Paragraphs

A paragraph is a block of plain text, possibly containing any number of inline elements defined in the Inline elements section above, with one or more blank lines above and below it.

```
This is a paragraph.

And this is *another* one containing some emphasised text.
A new line, but still part of the same paragraph.
```

Headers

A document can be split up into different sections using headers. Headers use the following syntax:

```
# Level One
## Level Two
### Level Three
#### Level Four
##### Level Five
###### Level Six
```

A header line can contain any inline syntax in the same way as a paragraph can.

> **Tip**
>
> Try to avoid using too many levels of header within a single document. A heavily nested document may be indicative of a need to restructure it or split it into several pages covering separate topics.

Code blocks

Source code can be displayed as a literal block using an indent of four spaces as shown in the following example.

```
This is a paragraph.

    function func(x)
        # ...
    end

Another paragraph.
```

Additionally, code blocks can be enclosed using triple backticks with an optional ”language” to specify how a block of code should be highlighted.

````
A code block without a "language":

```
function func(x)
    # ...
end
```

and another one with the "language" specified as `julia`:

```julia
function func(x)
    # ...
end
```
````

> **Note**
>
> ”Fenced” code blocks, as shown in the last example, should be prefered over indented code blocks since there is no way to specify what language an indented code block is written in.

Block quotes

Text from external sources, such as quotations from books or websites, can be quoted using > characters prepended to each line of the quote as follows.

```
Here's a quote:

> Julia is a high-level, high-performance dynamic programming language for
> technical computing, with syntax that is familiar to users of other
> technical computing environments.
```

Note that a single space must appear after the > character on each line. Quoted blocks may themselves contain other toplevel or inline elements.

Images

The syntax for images is similar to the link syntax mentioned above. Prepending a ! character to a link will display an image from the specified URL rather than a link to it.

```
![alternative text](link/to/image.png)
```

Lists

Unordered lists can be written by prepending each item in a list with either *, +, or -.

```
A list of items:

  * item one
  * item two
  * item three
```

Note the two spaces before each `*` and the single space after each one.

Lists can contain other nested toplevel elements such as lists, code blocks, or quoteblocks. A blank line should be left between each list item when including any toplevel elements within a list.

````
Another list:

  * item one

  * item two

    ```
    f(x) = x
    ```

  * And a sublist:

      + sub-item one
      + sub-item two
````

> **Note**
>
> The contents of each item in the list must line up with the first line of the item. In the above example the fenced code block must be indented by four spaces to align with the `i` in `item two`.

Ordered lists are written by replacing the "bullet" character, either `*`, `+`, or `-`, with a positive integer followed by either `.` or `)`.

```
Two ordered lists:

 1. item one
 2. item two
 3. item three

 5) item five
 6) item six
 7) item seven
```

An ordered list may start from a number other than one, as in the second list of the above example, where it is numbered from five. As with unordered lists, ordered lists can contain nested toplevel elements.

Display equations

Large LaTeX equations that do not fit inline within a paragraph may be written as display equations using a fenced code block with the "language" `math` as in the example below.

````
```math
f(a) = \frac{1}{2\pi}\int_{0}^{2\pi} (\alpha+R\cos(\theta))d\theta
```
````

Footnotes

This syntax is paired with the inline syntax for Footnote references. Make sure to read that section as well.

Footnote text is defined using the following syntax, which is similar to footnote reference syntax, aside from the `:` character that is appended to the footnote label.

````
[^1]: Numbered footnote text.

[^note]:

    Named footnote text containing several toplevel elements.

      * item one
      * item two
      * item three

    ```julia
    function func(x)
        # ...
    end
    ```
````

Note

No checks are done during parsing to make sure that all footnote references have matching footnotes.

Horizontal rules

The equivalent of an <hr> HTML tag can be written using the following syntax:

```
Text above the line.

---

And text below the line.
```

Tables

Basic tables can be written using the syntax described below. Note that markdown tables have limited features and cannot contain nested toplevel elements unlike other elements discussed above – only inline elements are allowed. Tables must always contain a header row with column names. Cells cannot span multiple rows or columns of the table.

```
| Column One | Column Two | Column Three |
|:---------- | ---------- |:------------:|
| Row `1`    | Column `2` |              |
| *Row* 2    | **Row** 2  | Column ``3`` |
```

Note

As illustrated in the above example each column of | characters must be aligned vertically.

A : character on either end of a column's header separator (the row containing – characters) specifies whether the row is left-aligned, right-aligned, or (when : appears on both ends) center-aligned. Providing no : characters will default to right-aligning the column.

Admonitions

Specially formatted blocks with titles such as "Notes", "Warning", or "Tips" are known as admonitions and are used when some part of a document needs special attention. They can be defined using the following !!! syntax:

```
!!! note

    This is the content of the note.

!!! warning "Beware!"

    And this is another one.

    This warning admonition has a custom title: `"Beware!"`.
```

Admonitions, like most other toplevel elements, can contain other toplevel elements. When no title text, specified after the admonition type in double quotes, is included then the title used will be the type of the block, i.e. `"Note"` in the case of the `note` admonition.

20.6 Markdown Syntax Extensions

Julia's markdown supports interpolation in a very similar way to basic string literals, with the difference that it will store the object itself in the Markdown tree (as opposed to converting it to a string). When the Markdown content is rendered the usual `show` methods will be called, and these can be overridden as usual. This design allows the Markdown to be extended with arbitrarily complex features (such as references) without cluttering the basic syntax.

In principle, the Markdown parser itself can also be arbitrarily extended by packages, or an entirely custom flavour of Markdown can be used, but this should generally be unnecessary.

Chapter 21

Metaprogramación

El legado más fuerte de Lisp en el lenguaje Julia es su soporte a la metaprogramación. Al igual que Lisp, Julia representa su propio código como una estructura de datos del propio lenguaje. Dado que el código está representado por objetos que pueden ser creados y manipulados desde dentro del lenguaje, es posible que un programa pueda transformar y generar su propio código. Esto permite una sofisticada generación de código sin pasos de construcción adicionales, y también permite las verdaderas macros de estilo Lisp que operan a nivel de los árboles sintácticos abstractos. En contraste, los sistemas de preprocesador ”macro” como el de C y C++, realizan la manipulación y sustitución textual antes de que se realice cualquier análisis o interpretación real. Debido a que todos los tipos de datos y código en Julia están representados por las estructuras de datos Julia, hay disponibles poderosas capacidades de reflexión están disponibles para explorar las características internas de un programa y sus tipos al igual que cualquier otro dato.

21.1 Representación de programas

Cada programa en Julia comienza su vida como una cadena:

```
julia> prog = "1 + 1"
"1 + 1"
```

¿Qué sucede después?

El siguiente paso es analizar sintácticamente cada cadena en un objeto denominado una expresión, representado por el tipo `Expr` de Julia:

```
julia> ex1 = parse(prog)
:(1 + 1)

julia> typeof(ex1)
Expr
```

Los objetos `Expr` contienen trdoses partes:

- Un `Symbol` identificando la clase de expresión. Un símbolo es un identificador de cadena internado (más información a continuación).

```
julia> ex1.head
:call
```

- Los argumentos de expresión, que pueden ser símbolos, otras expresiones o valores literales:

```
julia> ex1.args
3-element Array{Any,1}:
  :+
 1
 1
```

Las expresiones pueden también ser construidas directamente en notación prefija:

```
julia> ex2 = Expr(:call, :+, 1, 1)
:(1 + 1)
```

Las dos expresiones construidas antes (mediante análisis sintáctico y mediante construcción directa) son equivalentes:

```
julia> ex1 == ex2
true
```

El punto clave aquí es que el código Julia se representa internamente como una estructura de datos que es accesible desde el propio lenguaje.

La función `dump()` proporciona una visualización indentada y anotada de objetos `Expr`:

```
julia> dump(ex2)
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Int64 1
    3: Int64 1
  typ: Any
```

Los objetos `Expr` puede ser también anidados:

```
julia> ex3 = parse("(4 + 4) / 2")
:((4 + 4) / 2)
```

Otra forma de ver expresiones es con Meta.show_sexpr, que muestra la expresión-S de una `Expr` dada, que puede resultar muy familiar a los usuarios de Lisp. He aquí un ejemplo que visualiza una expresión (`Expr`) anidada:

```
julia> Meta.show_sexpr(ex3)
(:call, :/, (:call, :+, 4, 4), 2)
```

Símbolos

El carácter `:` tiene dos propósitos sintácticos en Julia. La primera forma crea un símbolo (`Symbol`), una cadena internada usada como un bloque constructivo de expresiones:

```
julia> :foo
:foo

julia> typeof(ans)
Symbol
```

El constructor `Symbol` toma cualquier número de argumentos y crea un símbolo concatenando sus representaciones de cadena juntas:

```
julia> :foo == Symbol("foo")
true

julia> Symbol("func",10)
:func10

julia> Symbol(:var,'_',"sym")
:var_sym
```

En el contexto de una expresión, los símbolos se utilizan para indicar el acceso a variables; Cuando se evalúa una expresión, se sustituye un símbolo por el valor asociado a ese símbolo en el ámbito apropiado.

A veces son necesarios paréntesis adicionales alrededor del argumento a: para evitar la ambigüedad en el análisis:

```
julia> :(:)
:(:)

julia> :(::)
:(::)
```

21.2 Expresiones y evaluación

Citación

El segundo propósito sintáctico del carácter `:` es crear objetos expresión sin utilizar el constructor `Expr` explícito. Esto se conoce como *citación*. El carácter `:`, seguido de pares de paréntesis alrededor de una sola declaración de código Julia, produce un objeto `Expr` basado en el código incluido. He aquí un ejemplo de la forma corta utilizada para citar una expresión aritmética:

```
julia> ex = :(a+b*c+1)
:(a + b * c + 1)

julia> typeof(ex)
Expr
```

(para ver la estructura de esta expresión, podemos usar `ex.head` y `ex.args` o `dump()`como antes)

Nótese que pueden construirse expresiones equivalentes usando `parse()` o la forma directa `Expr`:

```
julia>      :(a + b*c + 1)  ==

       parse("a + b*c + 1") ==

       Expr(:call, :+, :a, Expr(:call, :*, :b, :c), 1)
true
```

Las expresiones proporcionadas por el analizador sólo suelen tener símbolos, otras expresiones y valores literales como sus args, mientras que las expresiones construidas por el código Julia pueden tener valores de ejecución arbitrarios sin formas literales como args. En este ejemplo específico, `+` y `a` son símbolos, `*(b,c)` es una subexpresión, y `1` un entero literal de 64 bits.

Hay una segunda forma sintáctica de citar para expresiones múltiples: bloques de código encerrados en `quote ... end`. Note que esta forma introduce elementos `QuoteNode` al árbol de expresión que deben considerarse cuando se manipule directamente un árbol de expresiones generado a partir de bloques `quote`. Para otros propósitos, los bloques `:(...)` y `quote...end` se tratan de forma idéntica.

```
julia> ex = quote

           x = 1

           y = 2

           x + y

       end
quote
    #= none:2 =#
    x = 1
    #= none:3 =#
    y = 2
    #= none:4 =#
    x + y
end

julia> typeof(ex)
Expr
```

Interpolación

La construcción directa de objetos `Expr` con valores como argumentos es potente, pero los constructores objetos `Expr` pueden ser tediosos comparados con la sintaxis "normal" de Julia. Como alternativa, Julia permite "empalme" o interpolación de literales o expresiones en expresiones citadas. La interpolación se indica con el prefijo $.

En este ejemplo el valor literal de a es interpolado:

```
julia> a = 1;

julia> ex = :($a + b)
:(1 + b)
```

Interpolar en una expresión no citada (*quoted*) no se admite y causará un error en tiempo de compilación:

```
julia> $a + b
ERROR: unsupported or misplaced expression $
 ...
```

En este ejemplo, la tupla `(1,2,3)` es interpolada como una expresión en un test condicional:

```
julia> ex = :(a in $:((1,2,3)) )
:(a in (1, 2, 3))
```

Interpolar símbolos en una expresión anidada requiere encerrar cada símbolo en un bloque de cita que lo encierre:

```
julia> :( :a in $( :(:a + :b) ) )

                  ^^^^^^^^^^

                  quoted inner expression
```

El uso de $ para la interpolación de la expresión recuerda intencionalmente a la interpolación de cadenas y a la interpolación de mandatos. La interpolación de expresiones permite la construcción programática conveniente y legible de expresiones Julia complejas.

`eval()` and efectos

Dado un objeto expresión, uno puede causar que Julia lo evalúe (ejecute) en un ámbito global usando `eval()`:

```
julia> :(1 + 2)
:(1 + 2)

julia> eval(ans)
3

julia> ex = :(a + b)
:(a + b)

julia> eval(ex)
ERROR: UndefVarError: b not defined
[...]

julia> a = 1; b = 2;

julia> eval(ex)
3
```

Cada módulo tiene su propia función `eval()` que evalúa expresiones en su ámbito global. Las expresiones pasadas a `eval()` no están limitadas a valores de retorno (ellas también pueden tener efetos colaterales que alteren el estado del entorno del módulo que las encierra:

```
julia> ex = :(x = 1)
:(x = 1)

julia> x
ERROR: UndefVarError: x not defined

julia> eval(ex)
1

julia> x
1
```

Aquí, la evaluación de un objeto expresión causa que se asigne un valor a la variable global x.

Como las expresiones no son más que objetos `Expr` que pueden ser construidos programáticamente y después evaluados, es posible generar dinámicamente código arbitrario que pueda ser ejecutado luego mediante `eval()`. He aquí Since expressions are just `Expr` objects which can be constructed programmatically and then evaluated, un ejemplo sencillo:

```
julia> a = 1;

julia> ex = Expr(:call, :+, a, :b)
:(1 + b)

julia> a = 0; b = 2;

julia> eval(ex)
3
```

El valor de a se ha usado para construir la expresión ex que aplica la función + al valor 1 y a la variable b. Note la distinción importante entre la forma en que se usan las variables a y b:

- El valor de la *variable* a se utiliza como valor inmediato en la expresión en el tiempo de construcción de la expresión. Por lo tanto, una vez que la expresión es evaluada, el valor de a ya no importa: el valor en la expresión ya es 1, independientemente de lo que pueda ser ahora el valor de a.
- Por otro lado, el símbolo :b se utiliza en la construcción de la expresión, por lo que el valor de la variable b en ese momento es irrelevante - :b es sólo un símbolo y la variable b ni siquiera necesita ser definida. En el momento de la evaluación de la expresión, sin embargo, el valor del símbolo :b se resuelve buscando el valor de la variable b.

Funciones sobre Expresiones

Como se ha sugerido anteriormente, una característica muy útil de Julia es la capacidad de generar y manipular código Julia dentro del propio Julia. Ya hemos visto un ejemplo de una función que devuelve objetos Expr: la función parse(), que toma una cadena de código Julia y devuelve la Expr correspondiente. Una función también puede tomar uno o más objetos Expr como argumentos, y devolver otro Expr. Aquí hay un ejemplo simple y motivador:

```
julia> function math_expr(op, op1, op2)

           expr = Expr(:call, op, op1, op2)

           return expr

       end
math_expr (generic function with 1 method)

julia>  ex = math_expr(:+, 1, Expr(:call, :*, 4, 5))
:(1 + 4 * 5)

julia> eval(ex)
21
```

Otro ejemplo puede ser esta función que dobla cualquier argumento numérico, pero deja las expresiones solas:

```
julia> function make_expr2(op, opr1, opr2)

           opr1f, opr2f = map(x -> isa(x, Number) ? 2*x : x, (opr1, opr2))

           retexpr = Expr(:call, op, opr1f, opr2f)

           return retexpr

       end
make_expr2 (generic function with 1 method)

julia> make_expr2(:+, 1, 2)
:(2 + 4)

julia> ex = make_expr2(:+, 1, Expr(:call, :*, 5, 8))
:(2 + 5 * 8)

julia> eval(ex)
42
```

21.3 Macros

Las macros proporcionan un método para incluir el código generado en el cuerpo final de un programa. Una macro asigna una tupla de argumentos a una expresión devuelta, y la expresión resultante se compila directamente en lugar de requerir una llamada `eval()` de ejecución. Los argumentos de macro pueden incluir expresiones, valores literales y símbolos.

Básico

He aquí una macro extraordinariamente simple:

```
julia> macro sayhello()
           return :( println("Hello, world!") )
       end
@sayhello (macro with 1 method)
```

Las macros tienen un carácter dedicado en la sintaxis de Julia: el `@` (at-sign), seguido por el nombre único declarado en un bloque `macro NAME ... end`. En este ejemplo, el compilador reemplazará todas las instancias de `@sayhello` con:

```
:( println("Hello, world!") )
```

Cuando `@sayhello` se llama en el REPL, la expresión se ejecuta inmediatamente, por lo tanto solo vemos el resultado de la evaluación:

```
julia> @sayhello()
Hello, world!
```

Ahora, considere una macro un poco más compleja:

```
julia> macro sayhello(name)
           return :( println("Hello, ", $name) )
       end
@sayhello (macro with 1 method)
```

Esta macro toma un argumento: `name`. Cuando se encuentra `@sayhello`, la expresión citada se *expande* para interpolar el valor del argumento en la expresión final:

```
julia> @sayhello("human")
Hello, human
```

Podemos ver la expresión de retorno entre comillas usando la función `macroexpand()` (**nota importante:** esta es una herramienta extremadamente útil para depurar macros):

```
julia> ex = macroexpand( :(@sayhello("human")) )
:((println)("Hello, ", "human"))

julia> typeof(ex)
Expr
```

We can see that the `"human"` literal has been interpolated into the expression.

También existe una macro `@macroexpand` que quizás sea un poco más conveniente que la función `macroexpand`:

```
julia> @macroexpand @sayhello "human"
:((println)("Hello, ", "human"))
```

Un momento. ¿Por qué las macros?

Ya hemos visto una función `f(:: Expr ...) -> Expr` en una sección anterior. De hecho, `macroexpand()` es también una función. Entonces, ¿por qué existen macros?

Las macros son necesarias porque se ejecutan cuando se analiza el código, por lo tanto, las macros permiten al programador generar e incluir fragmentos de código personalizado antes de ejecutar el programa completo. Para ilustrar la diferencia, considere el siguiente ejemplo:

```
julia> macro twostep(arg)
           println("I execute at parse time. The argument is: ", arg)
           return :(println("I execute at runtime. The argument is: ", $arg))
       end
@twostep (macro with 1 method)

julia> ex = macroexpand( :(@twostep :(1, 2, 3)) );
I execute at parse time. The argument is: $(Expr(:quote, :((1, 2, 3))))
```

La primera llamada a `println()` se ejecuta cuando se invoca `macroexpand()`. La expresión resultante contiene *sólo* el segundo `println`:

```
julia> typeof(ex)
Expr

julia> ex
:((println)("I execute at runtime. The argument is: ", $(Expr(:copyast, :($(QuoteNode(:((1, 2, 3)
    ))))))))

julia> eval(ex)
I execute at runtime. The argument is: (1, 2, 3)
```

Invocación de macros

Las macros son invocadas con la siguiente sintaxis general:

```
@name expr1 expr2 ...
@name(expr1, expr2, ...)
```

Note la `@` antes del nombre de macro y la falta de domas entre las expresiones de los argumentos en la primera forma, y la falta de espacios en balnco después de `@name` en la segunda forma. Los dos estilos no deberían mezclarse. Por ejemplo, la siguiente sintaxis es diferente que la de los ejemplos anteriores; ella pasa la tupla `(expr1, expr2, ...)` como argumento a la macro:

```
@name (expr1, expr2, ...)
```

Es importante enfatizar que las macros reciben sus argumentos como expresiones, literales o símbolos. Una forma de explorar los argumentos de las macros es llamar a la función `show()` dentro del cuerpo de la macro:

```
julia> macro showarg(x)

           show(x)

           # ... remainder of macro, returning an expression
```

```
        end
@showarg (macro with 1 method)

julia> @showarg(a)
:a

julia> @showarg(1+1)
:(1 + 1)

julia> @showarg(println("Yo!"))
:(println("Yo!"))
```

Construir una macro avanzada

He aquí una versión simplificada de la macro `@assert` de Julia:

```
julia> macro assert(ex)
           return :( $ex ? nothing : throw(AssertionError($(string(ex)))) )
       end
@assert (macro with 1 method)
```

La macro puede ser usada de esta forma:

```
julia> @assert 1 == 1.0

julia> @assert 1 == 0
ERROR: AssertionError: 1 == 0
```

En lugar de la sintaxis escrita, la llamada a macro es expandida en tiempo de análisis para que devuelva un resutlado. Esto es equivalente a escribir:

```
1 == 1.0 ? nothing : throw(AssertionError("1 == 1.0"))
1 == 0 ? nothing : throw(AssertionError("1 == 0"))
```

Es decir, en la primera llamada, la expresión `:(1 == 1.0)` se empalma en la ranura de condición de prueba, mientras que el valor de `string(:( 1 == 1.0))` se empalma en la ranura de mensaje de aserción. Toda la expresión, así construida, se coloca en el árbol de sintaxis donde se produce la llamada de macro `@assert`. Entonces, en el tiempo de ejecución, si la expresión de prueba se evalúa como verdadera, entonces se devuelve `nothing`, mientras que si la prueba es falsa, se genera un error indicando la expresión afirmada que es falsa. Observe que no sería posible escribir esto como una función, ya que sólo está disponible el valor de la condición y sería imposible mostrar la expresión que lo calculó en el mensaje de error.

La definición real de `@assert` en la biblioteca estándar es más complicada. Permite al usuario especificar opcionalmente su propio mensaje de error, en lugar de simplemente imprimir la expresión fallida. Al igual que en las funciones con un número variable de argumentos, esto se especifica con elipses después del último argumento:

```
julia> macro assert(ex, msgs...)
           msg_body = isempty(msgs) ? ex : msgs[1]
           msg = string(msg_body)
           return :($ex ? nothing : throw(AssertionError($msg)))
       end
@assert (macro with 1 method)
```

Ahora `@assert` tiene dos modos de operación, dependiendo del número de argumentos que recibe! Si sólo hay un argumento, la tupla de expresiones capturadas por `msgs` estará vacía y se comportará igual que la definición más simple anterior. Ahora bien, si el usuario especifica un segundo argumento, se imprime en el cuerpo del mensaje en

lugar de la expresión que falla. Puede examinar el resultado de una expansión de macro con la función `macroexpand()` correctamente denominada:

```
julia> macroexpand(:(@assert a == b))
:(if a == b
        nothing
    else
        (throw)((AssertionError)("a == b"))
    end)

julia> macroexpand(:(@assert a==b "a should equal b!"))
:(if a == b
        nothing
    else
        (throw)((AssertionError)("a should equal b!"))
    end)
```

Hay otro caso que la versión real de `@assert` maneja: ¿qué pasa si, además de imprimir "a should be equal b", queremos imprimir sus valores? Uno podría ingenuamente intentar usar interpolación de cadena en el mensaje personalizado, por ejemplo, `@assert a==b "a ($a) should equal b ($b)!"`, pero esto no funcionará como se esperaba con la macro anterior. ¿Puedes ver por qué? Recuerda de string interpolation que una cadena interpolada se reescribe a una llamada a `string()`. Compare:

```
julia> typeof(:("a should equal b"))
String

julia> typeof(:("a ($a) should equal b ($b)!"))
Expr

julia> dump(:("a ($a) should equal b ($b)!"))
Expr
  head: Symbol string
  args: Array{Any}((5,))
    1: String "a ("
    2: Symbol a
    3: String ") should equal b ("
    4: Symbol b
    5: String ")!"
  typ: Any
```

Así que ahora en lugar de obtener una cadena sencilla en `msg_body`, la macro está recibiendo una expresión completa que necesitará ser evaluada para mostrarse como se esperaba. Esto puede ser empalmado directamente en la expresión devuelta como un argumento a la llamada `string()`; Vea `error.jl` para la implementación completa.

La macro `@assert` hace un gran uso del empalme en expresiones entre comillas para simplificar la manipulación de expresiones dentro del cuerpo de la macro.

Higiene

Un problema que surge en las macros más complejas es el de la higiene.En resumen, las macros deben asegurarse de que las variables que introducen en sus expresiones devueltas no chocan accidentalmente con las variables existentes en el código circundante en el que se expanden. A la inversa, a menudo se espera que las expresiones que se pasan a una macro como argumentos evalúen en el contexto del código circundante, interactuando con y modificando las variables existentes. Otra preocupación surge del hecho de que una macro puede ser llamada en un módulo diferente desde donde se definió. En este caso, debemos asegurarnos de que todas las variables globales se resuelvan en el

módulo correcto. Julia ya tiene una gran ventaja sobre los lenguajes con expansión de macro textual (como C) en que sólo necesita considerar la expresión devuelta. Todas las demás variables (como `msg` en `@assert` arriba) siguen el comportamiento normal del bloque de ámbito.

Para demostrar estos problemas, consideremos la posibilidad de escribir una macro `@time` que toma una expresión como su argumento, registra el tiempo, evalúa la expresión, registra el tiempo de nuevo, imprime la diferencia entre los tiempos antes y después y luego tiene el valor de La expresión como su valor final. La macro podría tener este aspecto:

```
macro time(ex)
    return quote
        local t0 = time()
        local val = $ex
        local t1 = time()
        println("elapsed time: ", t1-t0, " seconds")
        val
    end
end
```

Aquí, queremos que `t0`, `t1` y `val` sean variables temporales privadas, y queremos que `time` se refiera a la función `time()` de la biblioteca estándar, no a cualquier variable de tiempo que el usuario pueda tener (lo mismo se aplica a `println`). Imagine los problemas que podrían ocurrir si la expresión de usuario `ex` también contuviera asignaciones a una variable denominada `t0`, o definiese su propia variable `time`. Podríamos obtener errores o comportamiento misteriosamente incorrecto.

El expansor de macro de Julia resuelve estos problemas de la siguiente manera. En primer lugar, las variables dentro de un resultado de macro se clasifican como locales o globales. Una variable se considera local si es asignada (y no se declara global), se declara local o se utiliza como un nombre de argumento de función. De lo contrario, se considera global. Las variables locales son renombradas como únicas (utilizando la función `gensym()`, que genera nuevos símbolos), y las variables globales se resuelven dentro del entorno de definición de macro. Por lo tanto, ambas preocupaciones se manejan; Los locales de la macro no entrarán en conflicto con ninguna variable de usuario, y `time` y `println` se referirán a las definiciones de la biblioteca estándar.

Sin embargo, queda un problema. Considere el siguiente uso de esta macro:

```
module MyModule
import Base.@time

time() = ... # compute something

@time time()
end
```

Aquí la expresión de usuario `ex` es una llamada a `time`, pero no a la misma función `time` que usa la macro, sino que se refiere claramente a `MyModule.time`. Por tanto, debemos arreglar pora que el código en `ex` sea resuelto en el entorno de llamada de la macro. Esto se hace usando `esc()` para ”escapar” la expresión:

```
macro time(ex)
    ...
    local val = $(esc(ex))
    ...
end
```

Una expresión envuelta de esta manera es dejada sola por el expansor de macros y simplemente pegada en la salida. Por tanto, será resuelta en el entorno de llamada de la macro.

El mecanismo de ”escapar” puede ser usado para ”violar” la higiene cuando sea necesario, para introducir o manipular variables de usuario. Por ejemplo, la siguiente macro fija x a cero en el entorno de llamada:

```
julia> macro zerox()

           return esc(:(x = 0))

       end
@zerox (macro with 1 method)

julia> function foo()

           x = 1

           @zerox

           return x # is zero

       end
foo (generic function with 1 method)

julia> foo()
0
```

Esta clase de manipulación de variables debería ser usada juiciosamente, pero es ocasionalmente bastante útil.

Obtener las normas de higiene correctas puede ser un desafío formidable. Antes de usar una macro, es posible que desee considerar si un cierre de función sería suficiente. Otra estrategia útil es diferir tanto trabajo como sea posible para el tiempo de ejecución. Por ejemplo, muchas macros simplemente envuelven sus argumentos en un QuoteNode u otro Expr similar. Algunos ejemplos de esto incluyen `@task body` que simplemente devuelve `schedule (Task(() -> $ body))`, y `@eval expr`, que simplemente devuelve `eval (QuoteNode (expr))`.

Obtener las normas de higiene correctas puede ser un desafío formidable. Antes de usar una macro, es posible que desee considerar si un cierre de función sería suficiente. Otra estrategia útil es diferir tanto trabajo como sea posible para el tiempo de ejecución. Por ejemplo, muchas macros simplemente envuelven sus argumentos en un QuoteNode u otro Expr similar. Algunos ejemplos de esto incluyen `@task body` que simplemente devuelve`schedule (Task (() -> $ body))`, y`@eval expr`, que simplemente devuelve `eval (QuoteNode (expr))`.

Para demostrarlo, podríamos reescribir el ejemplo `@time` anterior como:

```
macro time(expr)
    return :(timeit(() -> $(esc(expr))))
end
function timeit(f)
    t0 = time()
    val = f()
    t1 = time()
    println("elapsed time: ", t1-t0, " seconds")
    return val
end
```

Sin embargo, no hacemos esto por una buena razón: al envolver el `expr` en un nuevo bloque de alcance (la función anónima) también cambia ligeramente el significado de la expresión (el alcance de cualquier variable en él), mientras que queremos `@time` para ser utilizable con un impacto mínimo en el código ajustado.

21.4 Generación de Código

Cuando se requiere una cantidad significativa de código repetitivo, es común generarlo programáticamente para evitar la redundancia. En la mayoría de los lenguajes, esto requiere un paso de construcción adicional y un programa separado para generar el código repetitivo. En Julia, la interpolación de expresión y `eval()` permiten que dicha generación de código tenga lugar en el curso normal de la ejecución del programa. Por ejemplo, el siguiente código define una serie de operadores en tres argumentos en términos de sus formas de 2 argumentos:

```
for op = (:+, :*, :&, :|, :$)
    eval(quote
        ($op)(a,b,c) = ($op)(($op)(a,b),c)
    end)
end
```

De este modo, Julia actúa como su propio preprocesador, y permite la generación de código desde dentro del lenguaje. El código anterior debería ser escrito ligeramente más secamente usando la forma prefija de citación `:`

```
for op = (:+, :*, :&, :|, :$)
    eval(:(($op)(a,b,c) = ($op)(($op)(a,b),c)))
end
```

En este tipo de generación de código dentro del lenguaje utilizando el patrón `eval(quote(...))` es bastante común, sin embargo, que Julia venga con una macro para abreviar este patrón:

```
for op = (:+, :*, :&, :|, :$)
    @eval ($op)(a,b,c) = ($op)(($op)(a,b),c)
end
```

La macro `@eval` reescribe esta llamada para ser precisamente equivalente a las versiones largas anteriores. Para bloques de código generado más grandes, el argumento expresión dado a `@eval` puede ser un bloque:

```
@eval begin
    # multiple lines
end
```

21.5 Literales de cadena no estándar

Recuerde de Strings que los literales de cadena prefijados por un identificador se llaman literales de cadena no estándar y pueden tener semántica distinta que los literales de cadena no prefijados. Por ejemplo:

- `r"^\s*(?:#|$)"` produces un objeto expresión regular en lugar de una cadena.
- `b"DATA\xff\u2200"` es un literal array bytepara `[68,65,84,65,255,226,136,128]`.

Tal vez sorprendentemente, estos comportamientos no están codificados en el analizador de Julia o en el compilador. En su lugar, son comportamientos personalizados proporcionados por un mecanismo general que cualquiera puede utilizar: los literales de cadenas prefijados se analizan como llamadas a macros de nombre especial. Por ejemplo, la macro de expresiones regulares es sólo la siguiente:

```
macro r_str(p)
    Regex(p)
end
```

Eso es todo. Esta macro dice que el contenido literal de la cadena literal `r"^\s*(?:#|$)"` debe ser pasado a la macro `@r_str` y que el resultado de esa expansión debe colocarse en el árbol de sintaxis donde tiene lugar la cadena literal. En otras palabras, la expresión `r"^\s*(?:#|$)"` equivale a colocar el siguiente objeto directamente en el árbol de sintaxis:

```
Regex("^\\s*(?:#|\$)")
```

No sólo la forma literal de la cadena es más corta y mucho más conveniente, sino que también es más eficiente: puesto que la expresión regular se compila y el objeto Regex se crea realmente *cuando el código es compilado*, la compilación se produce sólo una vez, Se ejecuta el código. Considere si la expresión regular se produce en un bucle:

```
for line = lines
    m = match(r"^\s*(?:#|$)", line)
    if m === nothing
        # non-comment
    else
        # comment
    end
end
```

Como la expresión regular `r"^\s*(?:#|$)"` Se compila e inserta en el árbol de sintaxis cuando se analiza este código, la expresión sólo se compila una vez en lugar de cada vez que se ejecuta el bucle. Para lograr esto sin macros, uno tendría que escribir este bucle así:

```
re = Regex("^\\s*(?:#|\$)")
for line = lines
    m = match(re, line)
    if m === nothing
        # non-comment
    else
        # comment
    end
end
```

Por otra parte, si el compilador no pudiera determinar que el objeto regex era constante en todos los bucles, ciertas optimizaciones podrían no ser posibles, haciendo esta versión aún menos eficiente que la forma literal más conveniente de arriba. Por supuesto, todavía hay situaciones en las que la forma no literal es más conveniente: si se necesita interpolar una variable en la expresión regular, se debe tomar este enfoque más detallado; En los casos en que el patrón de expresión regular mismo es dinámico, cambiando potencialmente en cada iteración del bucle, un nuevo objeto expresión regular debe ser construido en cada iteración. Sin embargo, en la gran mayoría de los casos de uso, las expresiones regulares no se construyen sobre la base de datos de tiempo de ejecución. En esta mayoría de casos, la capacidad de escribir expresiones regulares como valores en tiempo de compilación es valiosísima.

Al igual que los literales de cadena no estándar, existen literales de comandos no estándar que usan una variante prefijada de la sintaxis literal del comando. El comando literal `custom \`literal\`` se analiza como `@custom_cmd "literal"`. Julia por sí misma no contiene ningún literal de comando no estándar, pero los paquetes pueden hacer uso de esta sintaxis. Aparte de la sintaxis diferente y el sufijo `_cmd` en lugar del sufijo `_str`, los literales de comandos no estándar se comportan exactamente como los literales de cadena no estándar.

En el caso de que dos módulos proporcionen cadenas o literales de comando con el mismo nombre, es posible calificar la cadena o literal de comando con un nombre de módulo. Por ejemplo, si tanto `Foo` como `Bar` proporcionan literal de cadena no estándar `@x_str`, entonces uno puede escribir `Foo.x "literal"` o `Bar.x "literal"`para desambiguar entre los dos.

El mecanismo para literales de cadena definidos por el usuario es profundo, profundamente poderoso. No sólo son literales no estándar de Julia implementados usándolos, sino que también se implementa la sintaxis literal de comandos (`` `echo "Hello, $person"` ``) se implementa con la siguiente macro de aspecto inofensivo:

```
macro cmd(str)
    :(cmd_gen($(shell_parse(str)[1])))
end
```

Por supuesto, una gran cantidad de complejidad se oculta en las funciones utilizadas en esta definición de macro, pero son sólo funciones, escritas íntegramente en Julia. Usted puede leer su fuente y ver exactamente lo que hacen -y todo lo que hacen es construir objetos de expresión para ser insertados en el árbol de sintaxis de su programa.

21.6 Funciones Generadas

Una macro muy especial es `@generated`, que permite definir las llamadas *funciones generadas*. Éstas tienen la capacidad de generar código especializado dependiendo de los tipos de sus argumentos con más flexibilidad y/o menos código que lo que se puede lograr con el despacho múltiple. Mientras las macros trabajan con expresiones al momento de analizar y no pueden acceder a los tipos de sus entradas, una función generada se amplía en un momento en que se conocen los tipos de los argumentos, pero la función aún no se ha compilado.

En lugar de realizar algún cálculo o acción, una declaración de función generada devuelve una expresión entre comillas que luego forma el cuerpo para el método correspondiente a los tipos de los argumentos. Cuando se llama, la expresión del cuerpo se compila (o se extrae de una caché, en las llamadas posteriores) y sólo se evalúa la expresión devuelta, y no el código que lo generó. Así, las funciones generadas proporcionan un marco flexible para mover el trabajo desde el tiempo de ejecución hasta el tiempo de compilación.

Cuando se definen las funciones generadas, hay tres diferencias principales con las funciones ordinarias:

1. Uno anota la declaración de función con la macro `@generated`. Esto agrega cierta información a la AST que permite al compilador saber que se trata de una función generada.
2. En el cuerpo de la función generada sólo tiene acceso a los *tipos* de los argumentos, no a sus valores – y cualquier función que fuera definida *antes* de la definición de la función generada.
3. En lugar de calcular algo o realizar alguna acción, devuelve una *expresión citada* que, cuando se evalúa, hace lo que uno quiere.
4. Las funciones generadas no deben *mutar* ni *observar* ningún estado global no constante (incluidos, por ejemplo, IO, bloqueos, diccionarios no locales o que usen `method_exists`). Esto significa que solo pueden leer constantes globales y no pueden tener ningún efecto secundario. En otras palabras, deben ser completamente puros. Debido a una limitación de implementación, esto también significa que actualmente no pueden definir un cierre o un generador sin tipo.

Es más fácil ilustrar esto con un ejemplo. Podemos declarar una función generada `foo` como:

```
julia> @generated function foo(x)
           Core.println(x)
           return :(x * x)
       end
foo (generic function with 1 method)
```

Tenga en cuenta que el cuerpo devuelve una expresión entre comillas, a saber `:(x * x)`, en lugar de sólo el valor de `x * x`.

Desde la perspectiva del llamador, son muy similares a las funciones regulares; de hecho, no tienes que saber si estás llamando a una función regular o generada -la sintaxis y el resultado de la llamada son iguales. Veamos cómo se comporta `foo`:

```
julia> x = foo(2); # note: output is from println() statement in the body
Int64

julia> x           # now we print x
4

julia> y = foo("bar");
String

julia> y
"barbar"
```

Así, vemos que en el cuerpo de la función generada, `x` es el tipo del argumento pasado, y el valor devuelto por la función generada es el resultado de la evaluación de la expresión citada que devolvimos de la definición, ahora con el *valor* de `x`.

¿Qué pasa si evaluamos foo de nuevo con un tipo que ya hemos utilizado?

```
julia> foo(4)
16
```

Tenga en cuenta que no hay ninguna impresión de `Int64`. El cuerpo de la función generada sólo se ejecuta una vez (no es enteramente cierto, véase la nota a continuación) cuando se compila el método para ese conjunto específico de tipos de argumentos. Después de eso, la expresión devuelta de la función generada en la primera invocación se vuelve a utilizar como el cuerpo del método.

La cantidad de veces que se genera una función generada *podría* ser solo una vez, pero *podría* también ser más frecuente o no aparecer en absoluto. Como consecuencia, *nunca* debe escribir una función generada con efectos secundarios: cuándo y con qué frecuencia ocurren los efectos secundarios no está definida. (Esto también es válido para las macros, y al igual que para las macros, el uso de `eval()` en una función generada es una señal de que estás haciendo algo incorrecto.) Sin embargo, a diferencia de las macros , el sistema de tiempo de ejecución no puede manejar correctamente una llamada a `eval()`, por lo que no se permite.

También es importante ver cómo las funciones `@generated` interactúan con la redefinición del método. Siguiendo el principio de que una función correcta `@generated` no debe observar ningún estado mutable ni causar ninguna mutación del estado global, vemos el siguiente comportamiento. Observe que la función generada *no puede* llamar a ningún método que no se haya definido antes de la * definición * de la función generada en sí.

Inicialmente `f(x)` tiene una definición:

```
julia> f(x) = "original definition";
```

Define otras operaciones que usan `f(x)`:

```
julia> g(x) = f(x);

julia> @generated gen1(x) = f(x);

julia> @generated gen2(x) = :(f(x));
```

Ahora añadimos algunas definiciones nuevas para `f(x)`:

```
julia> f(x::Int) = "definition for Int";

julia> f(x::Type{Int}) = "definition for Type{Int}";
```

y comparamos cómo difieren estos resultados:

```
julia> f(1)
"definition for Int"

julia> g(1)
"definition for Int"

julia> gen1(1)
"original definition"

julia> gen2(1)
"definition for Int"
```

Cada método de una función generada tiene su propia visión de funciones definidas:

```
julia> @generated gen1(x::Real) = f(x);

julia> gen1(1)
"definition for Type{Int}"
```

La función de ejemplo `foo` anterior no hizo nada de lo que una función normal `foo(x) = x*x` no pudo hacer (excepto imprimir el tipo en la primera invocación y incurrir en una sobrecarga más alta). Sin embargo, el poder de una función generada radica en su capacidad para calcular diferentes expresiones entrecomilladas según los tipos que se le pasen:

```
julia> @generated function bar(x)

           if x <: Integer

               return :(x ^ 2)

           else

               return :(x)

           end

       end
bar (generic function with 1 method)

julia> bar(4)
16

julia> bar("baz")
"baz"
```

(although por supuesto este ejemplo artificial se implementa fácilmente usando despacho múltiple...

Podemos, por supuesto, abusar de esto para producir algún comportamiento interesante:

```
julia> @generated function baz(x)

```

```
            if rand() < .9
                return :(x^2)
            else
                return :("boo!")
            end
        end
baz (generic function with 1 method)
```

dado que el cuerpo de la función generada es no determinista, su comportamiento es indefinido.

¡No copie estos ejemplos!

Estos ejemplos sirven para ilustrar cómo funcionan las funciones generadas, tanto en el extremo de la definición como en el sitio de llamada; Sin embargo, no los copie, por las siguientes razones:

- La función `foo` tiene efectos secundarios (la llamada a `Core.println`) y no está definida exactamente cuándo, con qué frecuencia o cuántas veces se producirán estos efectos secundarios
- La función `bar` resuelve un problema que se resuelve mejor con el despacho múltiple - definiendo `bar(x) = x` y `bar(x :: Integer) = x ^ 2` hará la misma cosa, pero es más simple y más rápido.
- La función `baz` es patológicamente insana

Tenga en cuenta que el conjunto de operaciones que no se deben intentar en una función generada no tiene límites, y el sistema de tiempo de ejecución solo puede detectar actualmente un subconjunto de las operaciones no válidas. Hay muchas otras operaciones que simplemente corromperán el sistema de tiempo de ejecución sin notificación, por lo general de maneras sutiles que obviamente no están conectadas a la mala definición. Debido a que el generador de funciones se ejecuta durante la inferencia, debe respetar todas las limitaciones de ese código.

Algunas operaciones que no deberían intentarse incluyen:

1. Almacenamiento en caché de punteros nativos.
2. Interactuar de cualquier manera con los contenidos o métodos de Core.Inference.
3. Observar cualquier estado mutable.
    - La inferencia sobre la función generada se puede ejecutar en *cualquier* momento, incluso cuando el código intenta observar o modificar este estado.
4. Tomar cualquier bloqueo: código C que llame a puede utilizar bloqueos internos, (por ejemplo, no es problemático para llamar `malloc`, a pesar de que la mayoría de las implementaciones requieren bloqueos internos), pero no pretenden mantener o adquirir cualquier tiempo ejecutando el código de Julia.
5. Llamar a cualquier función que esté definida después del cuerpo de la función generada. Esta condición se relaja para los módulos precompilados de carga incremental para permitir llamar a cualquier función en el módulo.

De acuerdo, ahora que tenemos una mejor comprensión de cómo funcionan las funciones generadas, utilicémoslas para construir algunas funcionalidades más avanzadas (y válidas) ...

An advanced example

La biblioteca base de Julia tiene una función `sub2ind()` para calcular un índice lineal en una matriz n-dimensional, basada en un conjunto de n índices multilineales - en otras palabras, para calcular el índice `i` que se puede usar para indexar en una matriz `A` usando `A[i]`, en lugar de `A[x, y, z, ...]`. Una posible implementación es la siguiente:

```
julia> function sub2ind_loop(dims::NTuple{N}, I::Integer...) where N
           ind = I[N] - 1
           for i = N-1:-1:1
               ind = I[i]-1 + dims[i]*ind
           end
           return ind + 1
       end
sub2ind_loop (generic function with 1 method)

julia> sub2ind_loop((3, 5), 1, 2)
4
```

Lo mismo puede hacerse usando recursión:

```
julia> sub2ind_rec(dims::Tuple{}) = 1;

julia> sub2ind_rec(dims::Tuple{}, i1::Integer, I::Integer...) =

           i1 == 1 ? sub2ind_rec(dims, I...) : throw(BoundsError());

julia> sub2ind_rec(dims::Tuple{Integer, Vararg{Integer}}, i1::Integer) = i1;

julia> sub2ind_rec(dims::Tuple{Integer, Vararg{Integer}}, i1::Integer, I::Integer...) =

           i1 + dims[1] * (sub2ind_rec(Base.tail(dims), I...) - 1);

julia> sub2ind_rec((3, 5), 1, 2)
4
```

Ambas implementaciones, aunque diferentes, hacen esencialmente lo mismo: un bucle de tiempo de ejecución sobre las dimensiones de la matriz, recogiendo el desplazamiento en cada dimensión en el índice final.

Sin embargo, toda la información que necesitamos para el bucle está incrustada en la información de tipo de los argumentos. Por lo tanto, podemos utilizar las funciones generadas para mover la iteración a tiempo de compilación; en la jerga del compilador, usamos las funciones generadas para desenrollar manualmente el bucle. El cuerpo se vuelve casi idéntico, pero en vez de calcular el índice lineal, construimos una *expresión* que calcula el índice:

```
julia> @generated function sub2ind_gen(dims::NTuple{N}, I::Integer...) where N
           ex = :(I[$N] - 1)
           for i = (N - 1):-1:1
               ex = :(I[$i] - 1 + dims[$i] * $ex)
           end
           return :($ex + 1)
       end
sub2ind_gen (generic function with 1 method)

julia> sub2ind_gen((3, 5), 1, 2)
4
```

¿Qué código generará esto?

Una forma sencilla de averiguarlo es extraer el cuerpo en otra función (regular):

```
julia> @generated function sub2ind_gen(dims::NTuple{N}, I::Integer...) where N
           return sub2ind_gen_impl(dims, I...)
       end
sub2ind_gen (generic function with 1 method)

julia> function sub2ind_gen_impl(dims::Type{T}, I...) where T <: NTuple{N,Any} where N
           length(I) == N || return :(error("partial indexing is unsupported"))
           ex = :(I[$N] - 1)
           for i = (N - 1):-1:1
               ex = :(I[$i] - 1 + dims[$i] * $ex)
           end
           return :($ex + 1)
       end
sub2ind_gen_impl (generic function with 1 method)
```

Ahora podemos ejecutar `sub2ind_gen_impl` y examinar la expresión que devuelve

```
julia> sub2ind_gen_impl(Tuple{Int,Int}, Int, Int)
:(((I[1] - 1) + dims[1] * (I[2] - 1)) + 1)
```

Por lo tanto, el cuerpo de método que se utilizará aquí no incluye un bucle en absoluto - sólo indexación en las dos tuplas, multiplicación y suma/resta. Todo el bucle se realiza en tiempo de compilación, y evitamos el bucle durante la ejecución por completo. Por lo tanto, sólo se realiza el bucle una vez por tipo, en este caso una vez por `N` (excepto en casos de borde donde la función se genera más de una vez - véase la exención de responsabilidad anterior).

Chapter 22

Arrays Multi-dimensionales

Julia, como la mayoría de los lenguajes informáticos técnicos, proporciona una implementación de los arrays de primera clase. La mayoría de los lenguajes informáticos técnicos prestan mucha atención a su implementación de arrays a expensas de otros contenedores. Julia no trata los arrays de manera especial. La biblioteca de arrays se ha implementado casi completamente en el propio lenguaje Julia, y deriva su rendimiento del compilador, al igual que cualquier otro código escrito en Julia. Como tal, es también posible definir tipos de arrays personalizados heredando de `AbstractArray`. Consulte la sección de manual en la interfaz AbstractArray para ms detalles sobre implementar un tipo array personalizado.

Un array es una colección de objetos almacenados en una cuadrícula multidimensional. En el caso más general, un array puede contener objetos de tipo `Any`. Para la mayoría de los propósitos computacionales, los arrays deben contener objetos de un tipo más específico, como `Float64` o `Int32`.

En general, a diferencia de muchos otros lenguajes informáticos técnicos, Julia no espera que los programas se escriban en un estilo vectorizado para el rendimiento. El compilador de Julia utiliza la inferencia de tipos y genera código optimizado para la indexación escalar de arrays, permitiendo que los programas se escriban en un estilo que sea conveniente y legible, sin sacrificar el rendimiento y utilizando menos memoria a veces.

En Julia, todos los argumentos a las funciones se pasan por referencia. Algunos lenguajes informáticos técnicos pasan los arrays por valor, y esto es conveniente en muchos casos. En Julia, las modificaciones hechas a los arrays de entrada dentro de una función serán visibles en la función principal. Toda la biblioteca de arrays de Julia garantiza que las entradas no sean modificadas por las funciones de biblioteca. El código de usuario, si necesita mostrar un comportamiento similar, debe tener cuidado de crear una copia de las entradas que puede modificar.

22.1 Arrays

Funciones Básicas

Construcción e Inicialización

Existen muchas funciones para construir e inicializar matrices. En la siguiente lista de tales funciones, las llamadas con un argumento `dims...` pueden tomar una sola tupla de tamaños de dimensión o una serie de tamaños de dimensión pasados como un número variable de argumentos. Muchas de estas funciones también aceptan un primea entrada T, que es el tipo de los elementos del array. Si este tipo es omitido se asumirá como tipo por defecto por defecto `Float64`.

La sintaxis `[A, B, C, ...]` construye un array 1-dimensional (vector) a partir de sus argumentos. Si todos los argumentos tienen un tipo de promocion comun entonces ellos son convertidos a este tipo usando `convert()`.

[1] *iid*, independently and identically distributed.

Function	Description
`eltype(A)`	Tipo de los elementos contenidos en `A`
`length(A)`	Número de elementos en `A`
`ndims(A)`	Número de dimensiones de `A`
`size(A)`	Una tupla que contien las dimensiones de `A`
`size(A,n)`	El tamaño de `A` a lo largo de una dimensión particular `n`
`indices(A)`	Una tupla que contiene los índices válidos de `A`
`indices(A,n)`	Un rango expresando los úndices válidos a lo largo de la dimensión `n`
`eachindex(A)`	Un iterador eficiente para visitar cada posición en `A`
`stride(A,k)`	La zancada (*stride*, distancia de índice lineal entre elementos adyacentes) a lo largo de la dimensión `k`.
`strides(A)`	Una tupla de las zancadas en cada dimensión

Concatenación

Los arrays pueden ser construídos y también concatenados usando las siguientes funciones:

Los valores escalares pasados a estas funciones son tratados como arrays de 1 elemento.

Las funciones de concatenación se usan tan frecuentemente que tiene una sintaxis especial:

`hvcat()` concatena tanto en la dimensión 1 (con puntos y coma) como en la dos (con espacios).

Inicializadores de Array Tipados

Se puede construir una matriz con un tipo de elemento específico utilizando la sintaxis `T[A, B, C, ...]`. Esto construirá un array 1-d con el tipo de elemento `T`, inicializado para contener los elementos `A`, `B`, `C`, etc. Por ejemplo, `Any [x, y, z]` construye un array heterogéneo que puede contener cualquier valor.

La sintaxis de concatenación puede ser prefijada de forma similar con un tipo para especificar el tipo de elemento del resultado.

```
julia> [[1 2] [3 4]]
1×4 Array{Int64,2}:
 1  2  3  4

julia> Int8[[1 2] [3 4]]
1×4 Array{Int8,2}:
 1  2  3  4
```

Comprensiones

Las comprensiones proporcionan una forma general y potente de construir arrays. Su sintaxis es similar a la notación de construcción de conjuntos en matemáticas:

```
A = [ F(x,y,...) for x=rx, y=ry, ... ]
```

El significado de esta forma es que `F(x, y, ...)` es evaluado para las variables `x`, `y`, etc. tomando cada valor de la lista de valores proporcionada. Los valores se pueden especificar mediante cualquier objeto iterable, pero comúnmente serán rangos como `1:n` o `2:(n-1)`, o arrays de valores explícitos como `[1.2, 3.4, 5.7]`. El resultado es una matriz N-d densa con dimensiones que son la concatenación de las dimensiones de los rangos de las variables `rx`, `ry`, etc. y donde cada evaluación `F(x, y, ...)` devuelve un escalar.

Function	Description
`Array{T}(dims...)`	an uninitialized dense `Array`
`zeros(T, dims...)`	an `Array` of all zeros
`zeros(A)`	an array of all zeros with the same type, element type and shape as `A`
`ones(T, dims...)`	an `Array` of all ones
`ones(A)`	an array of all ones with the same type, element type and shape as `A`
`trues(dims...)`	a `BitArray` with all values `true`
`trues(A)`	a `BitArray` with all values `true` and the same shape as `A`
`falses(dims...)`	a `BitArray` with all values `false`
`falses(A)`	a `BitArray` with all values `false` and the same shape as `A`
`reshape(A, dims...)`	an array containing the same data as `A`, but with different dimensions
`copy(A)`	copy `A`
`deepcopy(A)`	copy `A`, recursively copying its elements
`similar(A, T, dims...)`	an uninitialized array of the same type as `A` (dense, sparse, etc.), but with the specified element type and dimensions. The second and third arguments are both optional, defaulting to the element type and dimensions of `A` if omitted.
`reinterpret(T, A)`	an array with the same binary data as `A`, but with element type `T`
`rand(T, dims...)`	an `Array` with random, iid [1] and uniformly distributed values in the half-open interval $[0, 1)$
`randn(T, dims...)`	an `Array` with random, iid and standard normally distributed values
`eye(T, n)`	n-by-n identity matrix
`eye(T, m, n)`	`m`-by-n identity matrix
`linspace(start, stop, n)`	range of n linearly spaced elements from `start` to `stop`
`fill!(A, x)`	fill the array `A` with the value `x`
`fill(x, dims...)`	an `Array` filled with the value `x`

Function	Description
`cat(k, A...)`	concatena n-d arrays a lo largo de la dimensión k
`vcat(A...)`	abreviatura para `cat(1, A...)`
`hcat(A...)`	abreviatura para `cat(2, A...)`

El siguiente ejemplo calcula la media ponderada del elemento actual y su vecino izquierdo y derecho a lo largo de una rejilla unidimensional:

```
julia> x = rand(8)
8-element Array{Float64,1}:
 0.843025
 0.869052
 0.365105
 0.699456
 0.977653
 0.994953
```

Expression	Calls
`[A; B; C; ...]`	`vcat()`
`[A B C ...]`	`hcat()`
`[A B; C D; ...]`	`hvcat()`

```
 0.41084
 0.809411

julia> [ 0.25*x[i-1] + 0.5*x[i] + 0.25*x[i+1] for i=2:length(x)-1 ]
6-element Array{Float64,1}:
 0.736559
 0.57468
 0.685417
 0.912429
 0.8446
 0.656511
```

El tipo del array resultante depende de los tipos de los elementos calculados. Para controlar el tipo explícitamente, un tipo puede ser precedido a la comprensión. Por ejemplo, podríamos haber solicitado el resultado en precisión simple escribiendo:

```
Float32[ 0.25*x[i-1] + 0.5*x[i] + 0.25*x[i+1] for i=2:length(x)-1 ]
```

Expresiones Generador

Las comprensiones también se pueden escribir sin los corchetes que las encierran, produciendo un objeto conocido como **generador**. Este objeto puede ser iterado para producir valores bajo demanda, en lugar de reservar espacio para un array y almacenarlos en él de antemano (véase Iteración). Por ejemplo, la siguiente expresión suma una serie sin asignar memoria:

```
julia> sum(1/n^2 for n=1:1000)
1.6439345666815615
```

Cuando se escribe una expresión generador con múltiples dimensiones dentro de una lista de argumentos, se necesitan paréntesis para separar el generador de argumentos posteriores:

```
julia> map(tuple, 1/(i+j) for i=1:2, j=1:2, [1:4;])
ERROR: syntax: invalid iteration specification
```

Todas las expresiones separadas por comas después del `for` se interpretan como rangos. Añadir paréntesis permite añadir un tercer argumento a `map`:

```
julia> map(tuple, (1/(i+j) for i=1:2, j=1:2), [1 3; 2 4])
2×2 Array{Tuple{Float64,Int64},2}:
 (0.5, 1)       (0.333333, 3)
 (0.333333, 2)  (0.25, 4)
```

Los rangos en generadores y comprensiones pueden depender de rangos anteriores escribiendo varias palabras clave `for`:

```
julia> [(i,j) for i=1:3 for j=1:i]
6-element Array{Tuple{Int64,Int64},1}:
 (1, 1)
 (2, 1)
 (2, 2)
 (3, 1)
 (3, 2)
 (3, 3)
```

En tales casos, el resultado es siempre unidimensional.

Los valores generados se pueden filtrar usando la palabra clave `if`:

```
julia> [(i,j) for i=1:3 for j=1:i if i+j == 4]
2-element Array{Tuple{Int64,Int64},1}:
 (2, 2)
 (3, 1)
```

Indexación

La sintaxis general para indexar en un array n-dimensional `A` es:

```
X = A[I_1, I_2, ..., I_n]
```

donde cada `I_k` puede ser un entero escalar, un array de enteros o cualquier otro índice soportado. Esto incluye `Colon` (`:`) para seleccionar todos los índices dentro de la dimensión completa, rangos de la forma `a:c` o `a:b:c` para seleccionar subsecciones contiguas o con salto, y arrays de booleans para seleccionar elementos en sus índices `true`.

Si todos los índices son escalares, entonces el resultado `X` es un solo elemento del array `A`. De lo contrario, `X` es un array con el mismo número de dimensiones que la suma de las dimensionalidades de todos los índices.

Si todos los índices son vectores, por ejemplo, entonces la forma de `X` sería `(length(I_1), length(I_2), ..., length(I_n))`, donde las ubicaciones `(i_1, i_2, ..., i_n)` de `X` contienen el valor `A[I_1[i_1], I_2[i_2], ..., I_n[i_n]]`. Si `I_1` se cambia por un array bidimensional, entonces `X` se vuelve un n+1-dimensional array de forma `(size(I_1, 1), size(I_1, 2), length(I_2), ..., length(I_n))`. La matriz añade una dimensión. La ubicación `(i_1, i_2, i_3, ..., i_{n+1})` contiene el valor en `A[I_1[i_1, i_2], I_2[i_3], ..., I_n[i_{n+1}]]`. Todas las dimensiones indexadas con escalares se eliminan. Por ejemplo, el resultado de `A[2, I, 3]` es un array de tamaño `size(I)`. Su i-ésimo elemento es poblado por `A[2, I[i], 3]`.

Como parte especial de esta sintaxis, se puede usar la palabra clave end para representar el último índice de cada dimensión dentro de los corchetes de indexación, según lo determinado por el tamaño del array más interno indexado. La sintaxis de indexación sin la palabra end es equivalente a una llamada a `getindex`:

```
X = getindex(A, I_1, I_2, ..., I_n)
```

Ejemplo:

```
julia> x = reshape(1:16, 4, 4)
4×4 Base.ReshapedArray{Int64,2,UnitRange{Int64},Tuple{}}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> x[2:3, 2:end-1]
2×2 Array{Int64,2}:
```

```
 6  10
 7  11

julia> x[1, [2 3; 4 1]]
2×2 Array{Int64,2}:
  5  9
 13  1
```

Los rangos vacío de la forma `n:n-1` se suelen usar para indicar la localización inter-index entre `n-1` y `n`. Por ejemplo, la función `searchsorted()` usa esta convención para indicar el punto de inserción de un valor no encontrados en un array ordenado:

```
julia> a = [1,2,5,6,7];

julia> searchsorted(a, 3)
3:2
```

Asignación

La sintaxis general para asignar valores en un array n-dimensional `A` es:

```
A[I_1, I_2, ..., I_n] = X
```

donde cada `I_k` puede ser un índice escalar, un array de enteros o cualquier otro índice soportado. Esto incluye `Colon` (`:`) para seleccionar todos los índices dentro de la dimensión completa, rangos de la forma `a:c` o `a:b:c` para seleccionar subsecciones contiguas o con salto, y arrays de booleans para seleccionar elementos en sus índices `true`.

Si `X` es un array, debe tener el mismo número de elementos que el producto de las longitudes de los índices `prod(length(I_1), length(I_2), ..., length(I_n))`. El valor en la localización `I_1[i_1], I_2[i_2], ..., I_n[i_n]`de `A` es sobreescrito con el valor `X[i_1, i_2, ..., i_n]`. Si `X` no es un array, su valor es escrito a todas las localizaciones referenciadas de `A`.

Justo como en Indexación, la palabra clave end puede utilizarse para representar el último índice de cada dimensión dentro de los corchetes de los índices, como queda determinado por el tamaño del array en el que se está siendo asignado. La sintaxis de la asignación indexada sin la palabra clave end es equivalente a llamar a la función `setindex!()`:

```
setindex!(A, X, I_1, I_2, ..., I_n)
```

Ejemplo:

```
julia> x = collect(reshape(1:9, 3, 3))
3×3 Array{Int64,2}:
 1  4  7
 2  5  8
 3  6  9

julia> x[1:2, 2:3] = -1
-1

julia> x
3×3 Array{Int64,2}:
 1  -1  -1
 2  -1  -1
 3   6   9
```

Tipos de Índices Soportados

En la expresión `A[I_1, I_2, ..., I_n]`, cada `I_k` puede ser un índice escalar, un array de índices escalares o un objeto que repreenta un array de índices escalares y puede ser convertido a tal mediante `to_indices`:

1. Un índice escalar. Por defecto esto incluye:
   - Enteros no booleanos
   - `CartesianIndex{N}`s, que se comportan como una `N`-tupla de enteros abarcando múltiples dimensiones (ver abajo para ms detalles)
2. Un array de índices escalares. Esto incluye:
   - Vectores y arrays multidimensionales de enteros
   - Arrays vacíos como `[]`, que no selecciona elementos
   - Ranges de la forma `a:c` o `a:b:c`, que seleccionan subsecciones contiguas o con salto desde a hasta c (inclusive)
   - Cualquier array de índices escalares que sea un subtipo de `AbstractArray`
   - Arrays de `CartesianIndex{N}` (ver abajo para ms detalles)
3. Un objeto que representa un array de índice escalares y puede ser convertido a tal mediante `to_indices`. Por defecto esto incluye:
   - `Colon()` (`:`), que representa todos los índices dentro de una dimensión entera o a través del array completo
   - Arrays de booleans, que seleccionan los elementos en los que sus índices son `true` indices (ver abajo para más detalles)

Índices Cartesianos

El objeto especial `CartesianIndex{N}` representa un índice escalar que se comporta como una `N`-tupla de enteros que abarcan multiples dimensioneas. Por ejemplo:

```
julia> A = reshape(1:32, 4, 4, 2);

julia> A[3, 2, 1]
7

julia> A[CartesianIndex(3, 2, 1)] == A[3, 2, 1] == 7
true
```

Considerado solo, esto puede parecer relativamente trivial; `CartesianIndex` simplemente reúne múltiples enteros juntos en un objeto que representa un único índice multidimensional. Sin embargo, cuando se combina con otras formas de indexación e iteradores que producen `CartesianIndex`es, esto puede conducir directamente a un código muy elegante y eficiente. Ver Iteración a continuación, y para algunos ejemplos más avanzados, ver esta publicación en el blog sobre algoritmos multidimensionales e iteración.

Los *arrays* de `CartesianIndex{N}` también sestán soportados. Representan una colección de índices escalares que abarcan `N` dimensiones cada uno, lo que permite una forma de indexación que a veces se denomina *indexación puntual*. Por ejemplo, permite acceder a los elementos diagonales desde la primera "página" de 'A' desde arriba:

```
julia> page = A[:,:,1]
4×4 Array{Int64,2}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> page[[CartesianIndex(1,1),
             CartesianIndex(2,2),
             CartesianIndex(3,3),
             CartesianIndex(4,4)]]
4-element Array{Int64,1}:
  1
  6
 11
 16
```

Esto se puede expresar mucho más simplemente con *dot broadcasting* y combinándolo con un índice entero normal (en lugar de extraer la primera página deA como un paso separado). Incluso se puede combinar con `:` para extraer ambas diagonales de las dos páginas al mismo tiempo:

```
julia> A[CartesianIndex.(indices(A, 1), indices(A, 2)), 1]
4-element Array{Int64,1}:
  1
  6
 11
 16

julia> A[CartesianIndex.(indices(A, 1), indices(A, 2)), :]
4×2 Array{Int64,2}:
  1  17
  6  22
 11  27
 16  32
```

> **Warning**
>
> `CartesianIndex` y los arrays de `CartesianIndex` no son compatibles con la palabra clave end que representa el último índice de una dimensión. No usaremos end cuando se indexen expresiones que puedan contener `CartesianIndex` or arrays de ellos.

Indexación Lógica

A menudo denominada indexación lógica o indexación con una máscara lógica, la indexación mediante una matriz booleana selecciona elementos en los índices cuyos valores son `verdaderos`. La indexación por un vector booleano B es efectivamente igual a la indexación por el vector de enteros que es devuelto por `find(B)`. De forma similar, la indexación por una matriz booleana N-dimensional es efectivamente igual a la indexación por el vector de `CartesianIndex{N}`s donde sus valores son `true`. Un índice lógico debe ser un vector de la misma longitud que la dimensión en la que indexa, o debe ser el único índice proporcionado y debe coincidir con el tamaño y la dimensionalidad de la matriz en la que se indexa. En general, es más eficiente usar matrices booleanas como índices directamente en lugar de llamar primero a `find()`.

```
julia> x = reshape(1:16, 4, 4)
4×4 Base.ReshapedArray{Int64,2,UnitRange{Int64},Tuple{}}:
 1  5   9  13
```

```
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> x[[false, true, true, false], :]
2×4 Array{Int64,2}:
 2  6  10  14
 3  7  11  15

julia> mask = map(ispow2, x)
4×4 Array{Bool,2}:
  true  false  false  false
  true  false  false  false
 false  false  false  false
  true   true  false   true

julia> x[mask]
5-element Array{Int64,1}:
  1
  2
  4
  8
 16
```

Iteración

Las formas recomendadas de iterar sobre un array completo son:

```
for a in A
    # Do something with the element a
end

for i in eachindex(A)
    # Do something with i and/or A[i]
end
```

La primera construcción se usa cuando necesitamos el valor, pero no los índices, de cada elemento. En la segunda construcción, `i` será un `Int` si `A` es un tipo array con indexación lineal rápida; en caso contrario será un `CartesianIndex`:

```
julia> A = rand(4,3);

julia> B = view(A, 1:3, 2:3);

julia> for i in eachindex(B)

           @show i

       end
i = CartesianIndex{2}((1, 1))
i = CartesianIndex{2}((2, 1))
i = CartesianIndex{2}((3, 1))
i = CartesianIndex{2}((1, 2))
i = CartesianIndex{2}((2, 2))
i = CartesianIndex{2}((3, 2))
```

En contraste con `for i = 1:length(A)`, iterar con `eachindex` proporciona una forma eficiente de iterar sobre cualquier tipo de array.

Rasgos de Array

Si uno escribe un tipo `AbstractArray` personalizado, uno puede especificar que el tipo tiene indexación lineal rápida usando:

```
Base.IndexStyle(::Type{<:MyArray}) = IndexLinear()
```

Esta configuración hará que la iteración `eachindex` sobre un objeto `MyArray` use enteros. Si no especifica este rasgo, se usa el valor predeterminado `IndexCartesian()`.

Arrays, Funciones y Operadores Vectorizados

Los siguientes operadores están soportados para arrays:

1. Aritmética unaria – `-`, `+`
2. Aritmética binaria – `-`, `+`, `*`, `/`, `\`, `^`
3. Comparación – `==`, `!=`, ≈ (`isapprox`),

La mayoría de los operadores aritméticos binarios enumerados anteriormente también funcionan elemento a elemento cuando un argumento es escalar: `-`, `+`, y `*`cuando cualquiera de los argumentos es escalar, y `/` y `\` cuando el denominador es escalar. Por ejemplo, `[1, 2] + 3 == [4, 5]` y `[6, 4] / 2 == [3, 2]`.

Además, para permitir una conveniente vectorización de operaciones matemáticas y de otro tipo, Julia proporciona la sintaxis punto `f.(args ...)`, por ejemplo, `sin.(x)` o `min.(x, y)`, para operaciones con elementos sobre arrays o mezclas de matrices y escalares (una Retransmisión (*broadcasting*)); estos tienen la ventaja adicional de ”fusión” en un solo bucle cuando se combina con otras llamadas de puntos, por ejemplo, `sin.(cos.(x))`

También, *cada* operador binario admite una versión de punto que se puede aplicar a matrices (y combinaciones de matrices y escalares) en tales operaciones de retransmisión fusionadas, por ejemplo, `z .== sin.(x. * y)`.

Tenga en cuenta que las comparaciones como == operan en arrays completos, dando un solo booleano como respuesta. Use operadores de punto como `.==` para comparaciones elemento a elemento. (Para operaciones de comparación como <, *solo* la versión de elementos `.<` es aplicable a las matrices).

También note la diferencia entre `max.(a, b)`, que retransmitir `max()` elemento a elemento sobre a yb, y `maximum(a)`, que encuentra el mayor valor dentro de a. La misma relación se cumple para `min.(A, b)` y `minimum(a)`.

Retransmisión

A veces es útil realizar operaciones binarias elemento por elemento en matrices de diferentes tamaños, como agregar un vector a cada columna de una matriz. Una forma ineficiente de hacer esto sería replicar el vector al tamaño de la matriz:

```
julia> a = rand(2,1); A = rand(2,3);

julia> repmat(a,1,3)+A
2×3 Array{Float64,2}:
 1.20813  1.82068  1.25387
 1.56851  1.86401  1.67846
```

Esto es un desperdicio cuando las dimensiones son grandes, por lo que Julia ofrece `broadcast()`, que expande las dimensiones *singleton* en los argumentos array para hacer coincidir la dimensión correspondiente en el otro array sin usar memoria extra, y aplicar la función dada elemento a elemento:

```
julia> broadcast(+, a, A)
2×3 Array{Float64,2}:
 1.20813  1.82068  1.25387
 1.56851  1.86401  1.67846

julia> b = rand(1,2)
1×2 Array{Float64,2}:
 0.867535  0.00457906

julia> broadcast(+, a, b)
2×2 Array{Float64,2}:
 1.71056  0.847604
 1.73659  0.873631
```

Los operadores con punto tales como `.+` y `.*` son equivalentes a llamadas a `broadcast` (excepto que se funden, como se describe a continuación). También hay una función `broadcast!()` para especificar un destino explícito (al que también se puede acceder por fusión mediante asignación `.=`), y funciones `broadcast_getindex()` y `broadcast_setindex!()` que retransmiten los índices antes de indexar. Además, `f.(Args...)` es equivalente a `broadcast(f, args...)`, proporcionando una sintaxis conveniente para retransmitir cualquier función (sintaxis punto). "Llamadas punto" anidadas `f.(...)` (incluidas las llamadas a `.+` Etcétera) fusibles automáticamente en una sola llamada `broadcast`.

Además, `broadcast()` no está limitado a los array (ver la documentación de la función), también maneja tuplas y trata cualquier argumento que no sea un array, tupla o `Ref` (excepto para`Ptr`) como un "escalar".

```
julia> convert.(Float32, [1, 2])
2-element Array{Float32,1}:
 1.0
 2.0

julia> ceil.((UInt8,), [1.2 3.4; 5.6 6.7])
2×2 Array{UInt8,2}:
 0x02  0x04
 0x06  0x07

julia> string.(1:3, ". ", ["First", "Second", "Third"])
3-element Array{String,1}:
 "1. First"
 "2. Second"
 "3. Third"
```

Implementation

El tipo de matriz base en Julia es el tipo abstracto `AbstractArray{T, N}`. Este tipo está parametrizado por el número de dimensiones `N` y el tipo de elementos `T`. `AbstractVector` y `AbstractMatrix` son aliases para los casos 1-d y 2-d. Las operaciones en los objetos `AbstractArray` se definen usando operadores y funciones de alto nivel, de manera que es independiente del almacenamiento subyacente. Estas operaciones generalmente funcionan correctamente como una alternativa para cualquier implementación de matriz específica.

El tipo `AbstractArray` incluye algo vagamente parecido a un array, y las implementaciones de este podrían ser bastante diferentes de los arrays convencionales. Por ejemplo, los elementos se pueden calcular a petición en lugar de ser almacenados. Sin embargo, cualquier tipo concreto de `AbstractArray{T, N}` debería implementar al menos `size(A)` (que devolvería una tupla `Int`), `getindex(A,i)` y `getindex(A, i1, ..., iN)`; Los arrays mutables también deberían implementar `setindex!()`. Se recomienda que estas operaciones tengan complejidad temporal casi constante, o técnicamente complejidad de orden 1 (Õ(1)), ya que de lo contrario algunas funciones podrían ser inesperadamente lentas. Los tipos concretos también deberían proporcionar un método `similar(A,T=eltype(A),dims=size(A))`, que se utiliza para asignar un conjunto similar para `copy()` y otras operaciones de actualización. No importa cómo se represente internamente un `AbstractArray {T, N}`, `T` es el tipo de objeto devuelto por la indización entera (`A [1, ..., 1]`, cuando `A` no está vacío) y`N` debe ser la longitud de la tupla devuelta por `size()`.

`DenseArray` es un subtipo abstracto de`AbstractArray` que pretende incluir todos los arrays que están establecidos en bloques regulares de memoria y que, por tanto, se puede pasar a funciones externas C y Fortran que esperan este diseño de memoria. Los subtipos deberían proporcionar un método `stride(A,k)` que devuelve el ”paso” de la dimensión `k`; incrementar el índice de dimensión `k` en 1 debería incremental el índice `i` de `getindex(A,i)` en `stride(A,k)`. Si se proporciona un método de conversión de puntero `Base.unsafe_convert(Ptr{T}, A)`, el diseño de la memoria debe corresponder de la misma manera a estos pasos.

El tipo `Array` es una instancia específica de `DenseArray` donde los elementos se almacenan en orden de columnas principales (consulte notas adicionales en Sugerencias de rendimiento). `Vector` y `Matrix` son alias para los casos 1-d y 2-d. Las operaciones específicas como la indexación escalar, la asignación y algunas otras operaciones básicas específicas del almacenamiento son todas las que tienen que estar implementadas en `Array`, de modo que el resto de la biblioteca de arrays puede implementarse de forma genérica.

`SubArray` es una especialización de `AbstractArray` que realiza indexación por referencia en lugar de por copia. Un `SubArray` se crea con la función `view()`, que es llamada de la misma manera que `getindex()` (con una matriz y una serie de argumentos de índice) . El resultado de `view()` se ve igual que el resultado de `getindex()`, excepto que los datos se dejan en su lugar. `view()` almacena los vectores de índice de entrada en un objeto `SubArray`, que luego puede usarse para indexar la matriz original de forma indirecta. Al colocar la macro `@views` delante de una expresión o bloque de código, cualquier segmento `array[...]` en esa expresión se convertirá para crear una vista `SubArray` en su lugar.

`StridedVector` y`StridedMatrix` son alias convenientes definidos para que Julia pueda llamar a un rango más amplio de funciones BLAS y LAPACK pasándoles objetos `Array` o `SubArray`, y ahorrando así ineficiencias de asignación de memoria y copia.

El siguiente ejemplo calcula la descomposición QR de una pequeña sección de una matriz más grande, sin crear ningún array temporal, y llamando a la función LAPACK apropiada con los parámetros de tamaño de dimensión y salto correctos.

```
julia> a = rand(10,10)
10×10 Array{Float64,2}:
 0.561255   0.226678   0.203391  0.308912   …  0.750307  0.235023   0.217964
 0.718915   0.537192   0.556946  0.996234      0.666232  0.509423   0.660788
 0.493501   0.0565622  0.118392  0.493498      0.262048  0.940693   0.252965
 0.0470779  0.736979   0.264822  0.228787      0.161441  0.897023   0.567641
 0.343935   0.32327    0.795673  0.452242      0.468819  0.628507   0.511528
 0.935597   0.991511   0.571297  0.74485    …  0.84589   0.178834   0.284413
 0.160706   0.672252   0.133158  0.65554       0.371826  0.770628   0.0531208
 0.306617   0.836126   0.301198  0.0224702     0.39344   0.0370205  0.536062
 0.890947   0.168877   0.32002   0.486136      0.096078  0.172048   0.77672
 0.507762   0.573567   0.220124  0.165816      0.211049  0.433277   0.539476

julia> b = view(a, 2:2:8,2:2:4)
```

```
4×2 SubAr-
↪  ray{Float64,2,Array{Float64,2},Tuple{StepRange{Int64,Int64},StepRange{Int64,Int64}},false}:
 0.537192  0.996234
 0.736979  0.228787
 0.991511  0.74485
 0.836126  0.0224702

julia> (q,r) = qr(b);

julia> q
4×2 Array{Float64,2}:
 -0.338809   0.78934
 -0.464815  -0.230274
 -0.625349   0.194538
 -0.527347  -0.534856

julia> r
2×2 Array{Float64,2}:
 -1.58553  -0.921517
  0.0       0.866567
```

22.2 Vectores y Matrices *Sparse*

Julia tiene soporte integrado para vectores y matrices dispersas (*sparse*). Las matrices *sparse* son matrices que contienen suficientes ceros para almacenarlos en una estructura de datos especial que ahorra espacio y tiempo de ejecución, en comparación con las matrices densas.

Columna Comprimida *Sparse* (CSC) Para Almacenamiento de Matrices *Sparse*

En Julia, las matrices dispersas se almacenan en el formato Compressed Sparse Column (CSC). Las matrices *sparse* de Julia tienen el tipo `SparseMatrixCSC{Tv,Ti}`, donde `Tv` es el tipo de los valores almacenados, `yTi` es el tipo entero para almacenar punteros de columnas e índices de filas. La representación interna de `SparseMatrixCSC` es la siguiente:

```
struct SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrix{Tv,Ti}
    m::Int                  # Number of rows
    n::Int                  # Number of columns
    colptr::Vector{Ti}      # Column i is in colptr[i]:(colptr[i+1]-1)
    rowval::Vector{Ti}      # Row indices of stored values
    nzval::Vector{Tv}       # Stored values, typically nonzeros
end
```

El almacenamiento de columnas dispersas y comprimidas (CSC) facilita y agiliza el acceso a los elementos en la columna de una matriz *sparse*, mientras que el acceso a la matriz *sparse* por filas es considerablemente más lento. Las operaciones como la inserción de entradas previamente no almacenadas de una en una en la estructura de CSC tienden a ser lentas. Esto se debe a que todos los elementos de la matriz *sparse* que están más allá del punto de inserción deben moverse un lugar más.

Todas las operaciones en matrices *sparse* se implementan cuidadosamente para aprovechar la estructura de datos CSC para el rendimiento y para evitar operaciones costosas.

Si tiene datos en formato CSC desde una aplicación o biblioteca diferente, y desea importarlos a Julia, asegúrese de utilizar la indexación basada en 1. Los índices de fila en cada columna deben estar ordenados. Si su objeto `SparseMatrixCSC` contiene índices de filas sin ordenar, una forma rápida de ordenarlos es hacer una doble transposición.

En algunas aplicaciones, es conveniente almacenar valores cero explícitos en una `SparseMatrixCSC`. Estas *son* aceptadas por funciones en `Base` (pero no hay garantía de que se conservarán en las operaciones de mutación). Tales ceros explícitamente almacenados son tratados como no estructurales por muchas rutinas. La función `nnz()` devuelve la cantidad de elementos almacenados explícitamente en la estructura de datos dispersos, incluidos los no-ceros estructurales. Para contar el número exacto de nozeros numéricos, use `countnz()`, que inspecciona todos los elementos almacenados en un a matriz *sparse*. `dropzeros()`, y `dropzeros!()`, se puede usar para eliminar ceros almacenados de la matriz dispersa.

```
julia> A = sparse([1, 2, 3], [1, 2, 3], [0, 2, 0])
3×3 SparseMatrixCSC{Int64,Int64} with 3 stored entries:
  [1, 1]  =  0
  [2, 2]  =  2
  [3, 3]  =  0

julia> dropzeros(A)
3×3 SparseMatrixCSC{Int64,Int64} with 1 stored entry:
  [2, 2]  =  2
```

Almacenamiento de Vectores *Sparse*

Los vectores *sparse* se almacenan en un formato de columna análogo al que se usa en las matrices *sparse*. En Julia, los vectores *sparse* tienen el tipo `SparseVector{Tv,Ti}` donde `Tv` es el tipo de los valores almacenados y`Ti` el tipo entero para los índices. La representación interna es la siguiente:

```
struct SparseVector{Tv,Ti<:Integer} <: AbstractSparseVector{Tv,Ti}
    n::Int              # Length of the sparse vector
    nzind::Vector{Ti}   # Indices of stored values
    nzval::Vector{Tv}   # Stored values, typically nonzeros
end
```

En cuanto a `SparseMatrixCSC`, el tipo `SparseVector` también puede contener ceros almacenados explícitamente. (Consulte Almacenamiento de matriz *sparse*.).

Constructores de Vectores y Matrices *Sparse*

La forma más sencilla de crear matrices *sparse* es usar funciones equivalentes a las funciones `zeros()` y `eye()` que proporciona Julia para trabajar con matrices densas. Para producir matrices *sparse* en su lugar, puede usar los mismos nombres con el prefijo `sp`:

```
julia> spzeros(3)
3-element SparseVector{Float64,Int64} with 0 stored entries

julia> speye(3,5)
3×5 SparseMatrixCSC{Float64,Int64} with 3 stored entries:
  [1, 1]  =  1.0
  [2, 2]  =  1.0
  [3, 3]  =  1.0
```

La función `sparse()` suele ser una forma útil de construir arrays *sparse*. Por ejemplo, para construir una matriz *sparse*, podemos ingresar un vector `I` de índices de fila, un vector`J` de índices de columna, y un vector `V` de valores almacenados (esto también se conoce como formato COO (coordenada)). `sparse (I,J,V)` construye una matriz *sparse* tal que `S[I[k], J[k]] = V[k]`. El constructor de vector *sparse* equivalente es `sparsevec`, que toma el vector de índices (fila) `I` y el vector `V` con los valores almacenados y construye un vector *sparse* `R` tal que `R[I[k]] = V[k]`.

```
julia> I = [1, 4, 3, 5]; J = [4, 7, 18, 9]; V = [1, 2, -5, 3];

julia> S = sparse(I,J,V)
5×18 SparseMatrixCSC{Int64,Int64} with 4 stored entries:
  [1 ,  4]  =  1
  [4 ,  7]  =  2
  [5 ,  9]  =  3
  [3 , 18]  =  -5

julia> R = sparsevec(I,V)
5-element SparseVector{Int64,Int64} with 4 stored entries:
  [1]  =  1
  [3]  =  -5
  [4]  =  2
  [5]  =  3
```

La inversa de las funciones `sparse()` y `sparsevec` es `findnz()`, que recupera las entradas utilizadas para crear el array *sparse*. También hay una función `findn` que solo devuelve los vectores índice.

```
julia> findnz(S)
([1, 4, 5, 3], [4, 7, 9, 18], [1, 2, 3, -5])

julia> findn(S)
([1, 4, 5, 3], [4, 7, 9, 18])

julia> findnz(R)
([1, 3, 4, 5], [1, -5, 2, 3])

julia> findn(R)
4-element Array{Int64,1}:
 1
 3
 4
 5
```

Otra forma de crear un array *sparse* es convertir un array denso en un array *sparse* usando la función `sparse()`:

```
julia> sparse(eye(5))
5×5 SparseMatrixCSC{Float64,Int64} with 5 stored entries:
  [1, 1]  =  1.0
  [2, 2]  =  1.0
  [3, 3]  =  1.0
  [4, 4]  =  1.0
  [5, 5]  =  1.0

julia> sparse([1.0, 0.0, 1.0])
3-element SparseVector{Float64,Int64} with 2 stored entries:
  [1]  =  1.0
  [3]  =  1.0
```

Puede ir en la otra dirección usando el constructor `Array`. La función `issparse()` se puede usar para consultar si una matriz es o no *sparse*.

```
julia> issparse(speye(5))
true
```

Operaciones con matrices *sparse*

Las operaciones aritméticas en matrices *sparse* también funcionan como lo hacen en matrices densas. La indexación de, la asignación en y la concatenación de matrices *sparse* funcionan de la misma manera que las matrices densas. Las operaciones de indexación, especialmente la asignación, son costosas, cuando se llevan a cabo un elemento cada vez. En muchos casos, puede ser mejor convertir la matriz dispersa en formato `(I,J,V)` usando `findnz()`, manipular los valores o la estructura en los vectores densos `(I,J,V)`, y luego reconstruir la matriz *sparse*.

Correspondence of dense and sparse methods

La siguiente tabla proporciona una correspondencia entre los métodos incorporados en matrices *sparse* y sus métodos correspondientes en tipos de matrices densas. En general, los métodos que generan matrices *sparse* difieren de sus contrapartes densas en que la matriz resultante sigue el mismo patrón de dispersión que una matriz *sparse* dada S, o que la matriz *sparse* resultante tiene densidad d, es decir, cada elemento de matriz tiene una probabilidad d de ser diferente de cero.

Los detalles se pueden encontrar en la sección Vectores y Matrices *Sparse* de la referencia de biblioteca estándar.

Sparse	Dense	Description
`spzeros(m,n)`	`zeros(m,n)`	Creates a *m*-by-*n* matrix of zeros. (`spzeros(m,n)` is empty.)
`spones(S)`	`ones(m,n)`	Creates a matrix filled with ones. Unlike the dense version, `spones()` has the same sparsity pattern as *S*.
`speye(n)`	`eye(n)`	Creates a *n*-by-*n* identity matrix.
`full(S)`	`sparse(A)`	Interconverts between dense and sparse formats.
`sprand(m,n,d)`	`rand(m,n)`	Creates a *m*-by-*n* random matrix (of density *d*) with iid non-zero elements distributed uniformly on the half-open interval $[0, 1)$.
`sprandn(m,n,d)`	`randn(m,n)`	Creates a *m*-by-*n* random matrix (of density *d*) with iid non-zero elements distributed according to the standard normal (Gaussian) distribution.
`sprandn(m,n,d,X)`	`randn(m,n)*X`	Creates a *m*-by-*n* random matrix (of density *d*) with iid non-zero elements distributed according to the *X* distribution. (Requires the `Distributions` package.)

Chapter 23

Álgebra Lineal

Además de (y como parte de) su soporte a los arrays multidimensionales, Julia proporciona implementaciones nativas de muchas operaciones de álgebra lineal comunes y útiles. Las operaciones básicas tales como la traza (`trace`), el determinante (`det`) y la inversa (`inv`) están todas soportadas:

```
julia> A = [1 2 3; 4 1 6; 7 8 1]
3×3 Array{Int64,2}:
 1  2  3
 4  1  6
 7  8  1

julia> trace(A)
3

julia> det(A)
104.0

julia> inv(A)
3×3 Array{Float64,2}:
 -0.451923   0.211538    0.0865385
  0.365385  -0.192308    0.0576923
  0.240385   0.0576923  -0.0673077
```

Así como otras operaciones útiles, como buscar autovalores o autovectores:

```
julia> A = [1.5 2 -4; 3 -1 -6; -10 2.3 4]
3×3 Array{Float64,2}:
   1.5   2.0  -4.0
   3.0  -1.0  -6.0
 -10.0   2.3   4.0

julia> eigvals(A)
3-element Array{Complex{Float64},1}:
  9.31908+0.0im
 -2.40954+2.72095im
 -2.40954-2.72095im

julia> eigvecs(A)
3×3 Array{Complex{Float64},2}:
 -0.488645+0.0im  0.182546-0.39813im   0.182546+0.39813im
```

```
 -0.540358+0.0im  0.692926+0.0im       0.692926-0.0im
   0.68501+0.0im  0.254058-0.513301im  0.254058+0.513301im
```

Además, Julia proporciona muchas factorizaciones que pueden usarse para acelerar problemas como la resolución lineal o la exponenciación de matrices mediante la pre-factorización de una matriz en una forma más adecuada (por razones de rendimiento o memoria)) al problema. Consulte la documentación en `factorize` para obtener más información. Como ejemplo:

```
julia> A = [1.5 2 -4; 3 -1 -6; -10 2.3 4]
3×3 Array{Float64,2}:
   1.5   2.0  -4.0
   3.0  -1.0  -6.0
 -10.0   2.3   4.0

julia> factorize(A)
Base.LinAlg.LU{Float64,Array{Float64,2}} with factors L and U:
[1.0 0.0 0.0; -0.15 1.0 0.0; -0.3 -0.132196 1.0]
[-10.0 2.3 4.0; 0.0 2.345 -3.4; 0.0 0.0 -5.24947]
```

Como A no es hermítica, simétrica, triangular o bidiagonal, una factorización LU puede ser lo mejor que podemos hacer. Compara con:

```
julia> B = [1.5 2 -4; 2 -1 -3; -4 -3 5]
3×3 Array{Float64,2}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0

julia> factorize(B)
Base.LinAlg.BunchKaufman{Float64,Array{Float64,2}}([-1.64286 0.142857 -0.8; 2.0 -2.8 -0.6; -4.0
↪  -3.0 5.0], [1, 2, 3], 'U', true, false, 0)
```

Aquí, Julia fue capaz de detectar que B es de hecho simetrica, y usa una factorizacíon más apropiada. Frecuentemente es posible escribir código ms eficiente para una matriz de la que se conocen ciertas propiedades como que sea simétrica o diagonal. Julia proporciona algunos tipos especiales para que uno pueda ”etiquetar” las matrices que tengan estas propiedades. Por ejemplo:

```
julia> B = [1.5 2 -4; 2 -1 -3; -4 -3 5]
3×3 Array{Float64,2}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0

julia> sB = Symmetric(B)
3×3 Symmetric{Float64,Array{Float64,2}}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0
```

sB ha sido etiquetada como una matriz que es simétrica (real) por lo que para algunas operacioneas que podríamos hacer sobre ella, tal como la autofactorización o calbular productos matriz-vector, pueden encontrarse eficiencias sólo referenciando la mitad de ella. Por ejemplo:

```
julia> B = [1.5 2 -4; 2 -1 -3; -4 -3 5]
3×3 Array{Float64,2}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0

julia> sB = Symmetric(B)
3×3 Symmetric{Float64,Array{Float64,2}}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0

julia> x = [1; 2; 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> sB\x
3-element Array{Float64,1}:
 -1.73913
 -1.1087
 -1.45652
```

La operación \ realiza aquí la resolución de la ecuación lineal. El analizador sintáctico de Julia proporciona un despacho conveniente para métodos especializados para la *transpuesta* de una matriz o una matriz dividida por la izquierda por un vector, o para las distintas combinaciones u operacioneas de transposición en soluciones matriz-matriz. Muchas de ellas son incluso más especializadas para ciertos tipos especiales de matrices. Por ejemplo, `A\B` acabará llamando a `Base.LinAlg.A_ldiv_B!` mientras que `A'\B` acabará llamando a `Base.LinAlg.Ac_ldiv_B`, incluso aunque usáramos el mismo operador de división por la izquierda. Esto funcionea también para matrices: `A.'\B.'` invocará a `Base.LinAlg.At_ldiv_Bt`. El operador de división por la izquierda es muy potente y es fácil escribir código compacto y bastante legible para resolver todo tipo de sistemas de ecuaciones lineales.

23.1 Matrices Especiales

Las matrices con simetrías y estructuras especiales surgen a menudo en el álgebra lineal y frecuentemente se asocian con varias factorizaciones matriciales. Julia presenta una rica colección de tipos de matrices especiales, que permiten un cálculo rápido con rutinas especializadas que están especialmente desarrolladas para estos tipos particuales de matrices.

Las siguientes tablas resumen los tipos de matrices especiales que se han implementado en Julia, así como si están disponibles ganchos para varios métodos optimizados para ellos en LAPACK.

Type	Description
`Hermitian`	Matriz hermítica
`UpperTriangular`	Matriz triangular superior
`LowerTriangular`	Matriz triangular inferior
`Tridiagonal`	Matriz tridiagonal
`SymTridiagonal`	Matriz tridiagonal simétrica
`Bidiagonal`	Matriz bidiagonal superior/inferior
`Diagonal`	Matriz diagonal
`UniformScaling`	Operador escalado uniforme

Operaciones elementales

Tipo de matriz	+	-	*	\	Otras funciones con métodos optimizados
Hermitian				MV	inv(), sqrtm(), expm()
UpperTriangular			MV	MV	inv(), det()
LowerTriangular			MV	MV	inv(), det()
SymTridiagonal	M	M	MS	MV	eigmax(), eigmin()
Tridiagonal	M	M	MS	MV	
Bidiagonal	M	M	MS	MV	
Diagonal	M	M	MV	MV	inv(), det(), logdet(), /()
UniformScaling	M	M	MVS	MVS	/()

Legend:

Clave	Descripción
M (matrix)	An optimized method for matrix-matrix operations is available
V (vector)	An optimized method for matrix-vector operations is available
S (scalar)	An optimized method for matrix-scalar operations is available

Factorizaciones de matrices

Matrix type	LAPACK	eig()	eigvals()	eigvecs()	svd()	svdvals()
Hermitian	HE		ARI			
UpperTriangular	TR	A	A	A		
LowerTriangular	TR	A	A	A		
SymTridiagonal	ST	A	ARI	AV		
Tridiagonal	GT					
Bidiagonal	BD				A	A
Diagonal	DI		A			

Legend:

Key	Description	Example
A (all)	An optimized method to find all the characteristic values and/or vectors is available	e.g. eigvals(M)
R (range)	An optimized method to find the ilth through the ihth characteristic values are available	eigvals(M, il, ih)
I (interval)	An optimized method to find the characteristic values in the interval [vl, vh] is available	eigvals(M, vl, vh)
V (vectors)	An optimized method to find the characteristic vectors corresponding to the characteristic values x=[x1, x2,...] is available	eigvecs(M, x)

El operador de escalado uniforme

Un operador `UniformScaling` representa un escalar multiplicado por el operador de identidad, `λ*I`. El operador de identidad `I` se define como una constante y es una instancia de `UniformScaling`. El tamaño de estos operadores es genérico y coincide con la otra matriz en las operaciones binarias `+`, `-`, `*` y `\`. Para `A+I` y`A-I` esto significa que `A` debe ser cuadrado. La multiplicación con el operador de identidad `I` es un *noop* (excepto para comprobar que el factor de escala es uno) y, por lo tanto, casi sin sobrecarga.

23.2 Factorizaciones de matrices

las factorizaciones de matrices (a.k.a. descomposiciones de matrices) calculan la factorizacin de una matriz en un producto de matrices, y son uno de los conceptos centrales del álgebra lineal.

La siguiente tabla resume los tipos de factorizaciones de matrices que han sido implementados en Julia. En la sección Linear Algebra de la documentación de la librería estándar pueden encontrarse más detalles de los métodos asociados.

Type	Description
`Cholesky`	Cholesky factorization
`CholeskyPivoted`	Pivoted Cholesky factorization
`LU`	LU factorization
`LUTridiagonal`	LU factorization for `Tridiagonal` matrices
`UmfpackLU`	LU factorization for sparse matrices (computed by UMFPack)
`QR`	QR factorization
`QRCompactWY`	Compact WY form of the QR factorization
`QRPivoted`	Pivoted QR factorization
`Hessenberg`	Hessenberg decomposition
`Eigen`	Spectral decomposition
`SVD`	Singular value decomposition
`GeneralizedSVD`	Generalized SVD

Chapter 24

Redes y Flujos

Julia proporciona una interfaz rica para tratar objetos que representen un flujo contínuo de E/S como terminales, tuberías y sockets TCP. Esta interfaz, aunque asíncrona a nivel del sistema, se presenta de forma síncrona al programador y normalmente no es necesario pensar en la operación asincrónica subyacente. Esto se logra haciendo un uso intensivo de la funcionalidad de los hilos cooperativos en Julia (o corrutinas).

24.1 Flujos de E/S básico

Todos los flujos en Julia exponen al menos un método `read()` y un `write()`, tomando el flujo como su primer argumento, por ejemplo:

```
julia> write(STDOUT,"Hello World");  # suppress return value 11 with ;
Hello World
julia> read(STDIN,Char)

'\n': ASCII/Unicode U+000a (category Cc: Other, control)
```

Tenga en cuenta que `write()` devuelve 11, el número de bytes (en `"Hello World"`) escrito en `STDOUT`, pero este valor de retorno se suprime con `;`.

Aquí Enter fue presionado nuevamente para que Julia leyera la nueva línea. Ahora, como puede ver en este ejemplo, `write()` toma los datos para escribir como su segundo argumento, mientras que `read()` toma el tipo de datos para ser leído como el segundo argumento.

Por ejemplo, para leer una matriz de bytes simple, podríamos hacer:

```
julia> x = zeros(UInt8, 4)
4-element Array{UInt8,1}:
 0x00
 0x00
 0x00
 0x00

julia> read!(STDIN, x)
abcd
4-element Array{UInt8,1}:
 0x61
 0x62
 0x63
 0x64
```

Sin embargo, dado que esto es un poco engorroso, se proporcionan varios métodos de conveniencia. Por ejemplo, podríamos haber escrito lo anterior como:

```
julia> read(STDIN,4)
abcd
4-element Array{UInt8,1}:
 0x61
 0x62
 0x63
 0x64
```

o si hubiéramos querido leer toda la línea en su lugar:

```
julia> readline(STDIN)
abcd
"abcd"
```

Tenga en cuenta que, dependiendo de la configuración de su terminal, su TTY puede estar almacenado en línea y, por lo tanto, podría requerir una entrada adicional antes de enviar los datos a Julia.

Para leer cada línea desde `STDIN`, puede usar `eachline()`:

```
for line in eachline(STDIN)
    print("Found $line")
end
```

o `read()` si deseamos leer carácter a carácter en lugar de lo anterior:

```
while !eof(STDIN)
    x = read(STDIN, Char)
    println("Found: $x")
end
```

24.2 E/S Texto

Note que el método `write()` mencionado arriba opera sobre flujos binarios. En particular, los valores no son convertidos a ninguna representación de texto canónica sino que son escritas tal cual:

```
julia> write(STDOUT,0x61);  # suppress return value 1 with ;
a
```

Note que a es escrito a `STDOUT` por la función `write()` y que el valor devuelto es 1 (ya que `0x61` es un byte).

Para E/S texto, use los métodos `print()` o `show()` methods, dependiendo de sus necesidades (ver la referencia de la librería estándar para una discusin detallada de la diferencia entre las dos):

```
julia> print(STDOUT, 0x61)
97
```

24.3 Propiedades contextuales de salida IO

En ocasiones, la salida de IO puede beneficiarse de la capacidad de pasar información contextual a los métodos de muestra. El objeto `IOContext` proporciona este marco para asociar metadatos arbitrarios con un objeto IO. Por ejemplo, `showcompact` agrega un parámetro de alusión al objeto IO para que el método del espectáculo invocado imprima un resultado más corto (si corresponde).

24.4 Trabajando con Ficheros

Al igual que muchos otros entornos, Julia tiene una función `open()`, que toma un nombre de archivo y devuelve un objeto `IOStream` que puede usar para leer y escribir cosas del archivo. Por ejemplo, si tenemos un archivo, `hello.txt`, cuyo contenido es `Hello, World!`:

```
julia> f = open("hello.txt")
IOStream(<file hello.txt>)

julia> readlines(f)
1-element Array{String,1}:
 "Hello, World!"
```

Si desea escribir a un fichero, puede abrirlo con el flag de escritura (`"w"`):

```
julia> f = open("hello.txt","w")
IOStream(<file hello.txt>)

julia> write(f,"Hello again.")
12
```

Si examina el contenido de `hello.txt` en este punto, notará que está vacío; no se ha escrito nada en el disco todavía. Esto se debe a que el `IOStream` debe cerrarse antes de que la escritura realmente se vacíe en el disco:

```
julia> close(f)
```

Examinando `hello.txt` nuevamente mostrará que su contenido ha sido cambiado.

Abrir un archivo, hacer algo con su contenido y volver a cerrarlo es un patrón muy común. Para hacerlo más fácil, existe otra invocación de `open()` que toma una función como su primer argumento y nombre de archivo como su segundo, abre el archivo, llama a la función con el archivo como argumento, y luego lo cierra de nuevo. Por ejemplo, dada una función:

```
function read_and_capitalize(f::IOStream)
    return uppercase(readstring(f))
end
```

Uno puede llamar a:

```
julia> open(read_and_capitalize, "hello.txt")
"HELLO AGAIN."
```

para abrir `hello.txt`, llamar `read_and_capitalize on it`, cerrar `hello.txt` y devolver los contenidos capitalizados.

Para incluso evitar tener que definir una función nombrada, puede usarse la sintaxis do, que crea una función anónima sobre la marcha:

```
julia> open("hello.txt") do f

           uppercase(readstring(f))

       end
"HELLO AGAIN."
```

24.5 Un ejemplo TCP simple

Saltemos directamente con un ejemplo simple que involucra sockets TCP. Primero creemos un servidor simple:

```
julia> @async begin

           server = listen(2000)

           while true

               sock = accept(server)

               println("Hello World\n")

           end

       end
Task (runnable) @0x00007fd31dc11ae0
```

Para quienes estén familiarizados con la API de socket de Unix, los nombres de los métodos se sentirán familiares, aunque su uso es algo más simple que la API de socket Raw de Unix. La primera llamada a `listen()` creará un servidor en espera de conexiones entrantes en el puerto especificado (2000) en este caso. La misma función también se puede usar para crear otros tipos de servidores:

```
julia> listen(2000) # Listens on localhost:2000 (IPv4)
TCPServer(active)

julia> listen(ip"127.0.0.1",2000) # Equivalent to the first
TCPServer(active)

julia> listen(ip"::1",2000) # Listens on localhost:2000 (IPv6)
TCPServer(active)

julia> listen(IPv4(0),2001) # Listens on port 2001 on all IPv4 interfaces
TCPServer(active)

julia> listen(IPv6(0),2001) # Listens on port 2001 on all IPv6 interfaces
TCPServer(active)

julia> listen("testsocket") # Listens on a UNIX domain socket/named pipe
PipeServer(active)
```

Tengase en cuenta que el tipo de retorno de la última invocación es diferente. Esto se debe a que este servidor no escucha en TCP, sino sobre una tubería nombrada (Windows) o socket de dominio UNIX. La diferencia es sutil y tiene que ver con los métodos `accept()` y `connect()`. El método `accept()` recupera una conexión con el cliente que se está conectando en el servidor que acabamos de crear, mientras que la función `connect()` se conecta a un servidor usando el método especificado. La función `connect()` toma los mismos argumentos que `listen()`, por lo tanto, suponiendo que el entorno (es decir, host, cwd, etc.) es el mismo uno debería capaz de pasar los mismos argumentos a `connect()` como lo se hizo para escuchar en el establecimiento de la conexión. Así que vamos a intentarlo (después de haber creado el servidor anterior):

```
julia> connect(2000)
TCPSocket(open, 0 bytes waiting)

julia> Hello World
```

Como era de esperar, vimos ”Hello World” impreso. Entonces, analicemos realmente lo que sucedió detrás de escena. Cuando llamamos a `connect()`, nos conectamos al servidor que acabamos de crear. Mientras tanto, la función `accept` devuelve una conexión del lado del servidor al socket recién creado e imprime ”Hello World” para indicar que la conexión fue exitosa.

Una gran fortaleza de Julia es que, dado que la API se expone sincrónicamente a pesar de que la E/S realmente está sucediendo de forma asíncrona, no tuvimos que preocuparnos de las devoluciones de llamadas ni siquiera de asegurarnos de que el servidor se ejecute. Cuando llamamos a `connect()` la tarea actual esperó a que se estableciera la conexión y solo continuó ejecutándose después de que se hizo. En esta pausa, la tarea del servidor reanudó la ejecución (porque una solicitud de conexión ya estaba disponible), aceptó la conexión, imprimió el mensaje y esperó al próximo cliente. Leer y escribir funciona de la misma manera. Para ver esto, considere el siguiente servidor de eco simple:

```
julia> @async begin
           server = listen(2001)
           while true
               sock = accept(server)
               @async while isopen(sock)
                   write(sock,readline(sock))
               end
           end
       end
Task (runnable) @0x00007fd31dc12e60

julia> clientside = connect(2001)
TCPSocket(RawFD(28) open, 0 bytes waiting)

julia> @async while true
           write(STDOUT,readline(clientside))
       end
```

```
Task (runnable) @0x00007fd31dc11870

julia> println(clientside,"Hello World from the Echo Server")
Hello World from the Echo Server
```

Como con otros flujos, use `close()` para desconectar el socket:

```
julia> close(clientside)
```

24.6 Resolviendo Direcciones IP

Uno de los métodos `connect()` que no sigue los métodos `listen()` es `connect(host::String, port)`, que intentará conectarse al host dado por el parámetro `host` en el puerto dado por el parámetro port. Te permite hacer cosas como:

```
julia> connect("google.com",80)
TCPSocket(RawFD(30) open, 0 bytes waiting)
```

En la base de esta funcionalidad está `getaddrinfo()`, que hará la resolución de dirección apropiada:

```
julia> getaddrinfo("google.com")
ip"74.125.226.225"
```

Chapter 25

Computación Paralela

La mayoría de las computadoras modernas poseen más de una CPU, y varias computadoras pueden combinarse en un cluster. Aprovechar la potencia de estas múltiples CPU permite que muchos cálculos se completen más rápidamente. Hay dos factores principales que influyen en el rendimiento: la velocidad de las propias CPUs y la velocidad de su acceso a la memoria. En un clúster, es bastante obvio que una CPU dada tendrá acceso más rápido a la RAM dentro de la misma computadora (nodo). Quizás más sorprendentemente, problemas similares son relevantes en un portátil multicore típico, debido a las diferencias en la velocidad de la memoria principal y la caché. En consecuencia, un buen entorno de multiprocesamiento debe permitir el control sobre la "propiedad" de un trozo de memoria por una CPU particular. Julia proporciona un entorno de multiprocesamiento basado en el paso de mensajes para permitir que los programas se ejecuten en múltiples procesos en distintos dominios de memoria a la vez.

La implementación de Julia del paso del mensajes es diferente de otros entornos tales como MPI [1]. Comunicación en Julia es generalmente "unilateral", lo que significa que el programador necesita explícitamente administrar sólo un proceso en una operación de dos procesos. Además, estas operaciones típicamente no se parecen a "envío de mensajes" y "recepción de mensajes", sino más bien se asemejan a operaciones de nivel superior como llamadas a funciones de usuario.

La programación paralela en Julia se basa en dos primitivas: *referencias remotas* y *llamadas remotas*. Una referencia remota es un objeto que puede utilizarse desde cualquier proceso para referirse a un objeto almacenado en un proceso determinado. Una llamada remota es una petición de un proceso para llamar a una determinada función en ciertos argumentos en otro proceso (posiblemente el mismo).

Las referencias remotas vienen en dos variedades: `Future` y `RemoteChannel`.

Una llamada remota devuelve un `Future` a su resultado. Las llamadas remotas retornan inmediatamente; el proceso que hizo que la llamada procede a su siguiente operación mientras la llamada remota sucede en otro lugar. Podemos esperar a que una llamada remota termine llamando a `wait()` en el `Future` devuelto, y puede obtener el valor completo del resultado usando `fetch()`.

Por otro lado, los objetos `RemoteChannel` son reescribibles. Por ejemplo, varios procesos pueden coordinar su procesamiento haciendo referencia al mismo canal (`Channel`) remoto.

Cada proceso tiene un identificador asociado. El proceso que proporciona el *prompt* interactivo de Julia siempre tiene un `id` igual a 1. Los procesos utilizados por defecto para operaciones paralelas se conocen como "workers". Cuando solo hay un proceso, el proceso 1 se considera trabajador. De lo contrario, se considera que los trabajadores son todos procesos distintos del proceso 1.

Vamos a probar esto. Comenzar con `julia -p n` proporciona n procesos *workers* en la máquina local. Generalmente tiene sentido igualar n al número de núcleos de la CPU en la máquina.

```
$ ./julia -p 2
```

```
julia> r = remotecall(rand, 2, 2, 2)
Future(2, 1, 4, Nullable{Any}())

julia> s = @spawnat 2 1 .+ fetch(r)
Future(2, 1, 5, Nullable{Any}())

julia> fetch(s)
2×2 Array{Float64,2}:
 1.18526  1.50912
 1.16296  1.60607
```

El primer argumento para `remotecall()` es la función para llamar. La mayoría de la programación paralela en Julia no hace referencia a procesos específicos ni a la cantidad de procesos disponibles, pero `remotecall()` se considera una interfaz de bajo nivel que proporciona un control más preciso. El segundo argumento para `remotecall()` es el `id` del proceso que hará el trabajo, y los argumentos restantes se pasarán a la función a la que se llama.

Como puede ver, en la primera línea le pedimos al proceso 2 que construyera una matriz aleatoria de 2 por 2, y en la segunda línea le pedimos que agregara 1 a ella. El resultado de ambos cálculos está disponible en los dos futuros, `r` ys. La macro `@ spawnat` evalúa la expresión en el segundo argumento sobre el proceso especificado por el primer argumento.

Ocasionalmente, es posible que desee un valor calculado de forma remota de inmediato. Esto suele ocurrir cuando lee desde un objeto remoto para obtener los datos que necesita la próxima operación local. La función `remotecall_fetch()` existe para este propósito. Es equivalente a `fetch (remotecall(...))` pero es más eficiente

```
julia> remotecall_fetch(getindex, 2, r, 1, 1)
0.18526337335308085
```

Recuerde que `getindex(r,1,1)` es equivalente a `r[1,1]`, por lo que esta llamada capta el primer elemento del futuro `r`.

La sintaxis de `remotecall()` no es especialmente conveniente. La macro `@spawn` hace las cosas más fáciles. Ella opera sobre una expresión en lugar de sobre una función, y elige dónde hacer la operación por ti:

```
julia> r = @spawn rand(2,2)
Future(2, 1, 4, Nullable{Any}())

julia> s = @spawn 1 .+ fetch(r)
Future(3, 1, 5, Nullable{Any}())

julia> fetch(s)
2×2 Array{Float64,2}:
 1.38854  1.9098
 1.20939  1.57158
```

Note que usamos `1 .+ fetch(r)` en lugar de `1 .+ r`. Esto es debido a que no sabemos dónde se ejecutará el código, por lo que en general puede que se requiera un`fetch()` para mover `r` al proceso que esta realizando la suma. En este caso, `@spawn` es bastante inteligente como para realizar el computo sobre el proceso que posee `r`, por lo que la llamada `fetch()` será una no-op (no se realiza trabajo).

(Hay que notar que `@spawn` no es una función predefinida sino que está definida en Julia como una macro. Es posible definir nuestras propias construcciones de ese tipo).

Una importante cosa a recordar es que, una vez traída, un `Future` cacheará su valor localmente. Las llamadas adicionales a `fetch()` calls no implican un salto de red. Una vez que todos los `Future`s que referencian ha sido recuperados, el valor remoto almacenado es borrado.

25.1 Disponibiliad de Código y Carga de Paquetes

Nuestro código debe estar disponible sobre cualquir proceso que lo ejecuta. Por ejemplo, escriba esto en el *prompt* de Julia:

```
julia> function rand2(dims...)

           return 2*rand(dims...)

       end

julia> rand2(2,2)
2×2 Array{Float64,2}:
 0.153756  0.368514
 1.15119   0.918912

julia> fetch(@spawn rand2(2,2))
ERROR: RemoteException(2, CapturedException(UndefVarError(Symbol("#rand2"))
[...]
```

El proceso 1 sabe dónde se encuentra la función `rand2`, pero el proceso 2 no.

Ms comunmente uno estará cargando código desde ficheros o paquetes, y tendrá una considerable flexibilidad para controlar qué procesos cargan código. Considere un fichero `DummyModule.jl`, que contenga el siguiente código:

```
module DummyModule

export MyType, f

mutable struct MyType
    a::Int
end

f(x) = x^2+1

println("loaded")

end
```

Si se arranca Julia con `julia -p 2`, se puede usar esto para verificar lo siguiente:

- `include("DummyModule.jl")` carga elfichero sólo sobre un proceso (el que ejecuta la instrucción).
- `using DummyModule` causa que el módulo sea cargado sobre todos los procesos; sin embargo, el módulo es llevado al ámbito sólo por el que ejecuta la instrucción.
- En cuanto `DummyModule` sea cargado sobre el proceso 2, mandatos como

  ```
  rr = RemoteChannel(2)
  put!(rr, MyType(7))
  ```

  permiten almacenar un objeto de tipo `MyType` sobre el proceso 2 incluso aunque `DummyModule` no esté en el ámbito del proceso 2.

Uno puede forzar que un mandato se ejecute sobre todos los procesos usando la macro `@everywhere`. Por ejemplo, `@everywhere` puede también ser usado para definir directamente una función sobre todos los procesos:

```
julia> @everywhere id = myid()

julia> remotecall_fetch(()->id, 2)
2
```

Un fichero puede también ser precargado sobre múltiples procesos al inicio, y puede usarse un *driver* para llevar a cabo ese cómputo:

```
julia -p <n> -L file1.jl -L file2.jl driver.jl
```

El proceso Julia que corre el programa *driver* del ejemplo anterior tiene un `id` igual a 1, justo como un proceso que proporciona un prompt interactivo.

La instalación base de Julia tiene soporte intríseco para dos tipos de clusters:

- Un clúster local especificado con la opción `-p`, como se mostró anteriormente.
- Un clúster que abarca máquinas utilizando la opción `--machinefile`. Esto utiliza un inicio de sesión `ssh` sin contraseña para iniciar los procesos de trabajo de Julia (desde la misma ruta que el servidor actual) en las máquinas especificadas.

Las funciones `addprocs()`, `rmprocs()`, `workers()`, y otras están disponibles como una forma programática de añadir, borrar y consultar los procesos en un cluster.

Note que los *workers* no ejecutan un script `.juliarc.jl` de inicio, ni sincronizan su estado global (tal como variables globales, nuevas definiciones de métodos y módulos cargados) con cualquiera de los procesos que están ejecutando.

Pueden soportarse otros tipos de clústers escribiendo nuestro propio `ClusterManager`, como se describe después en la sección ClusterManagers section.

25.2 Movimiento de Datos

Enviar mensaje y mover datos constituye la mayor parte de la sobrecarga de un programa paralelo. Reducir el número de mensajes y la cantidad de datos enviados es crítico para conseguir rendimiento y escalabilidad. Para este fin, es importante comprender el movimiento de datos realizado por varias construcciones de programación paralela de Julia.

`fetch()` puede considerarse una operación explícita de movimiento de datos, ya que directamente pide que un objeto sea movido a la máquina local. `@spawn` (y unas pocas construcciones relacionadas) también mueve datos, pero esto no es tan obvio, por lo tanto, puede denominarse una operación implícita de movimiento de datos. Considere estos dos enfoques para construir y cuadrar una matriz aleatoria:

Method 1:

```
julia> A = rand(1000,1000);

julia> Bref = @spawn A^2;

[...]

julia> fetch(Bref);
```

Method 2:

```
julia> Bref = @spawn rand(1000,1000)^2;

[...]

julia> fetch(Bref);
```

La diferencia parece trivial, pero de hecho es bastante significativa debido al comportamiento de `@spawn`. En el primer método, se construye localmente una matriz aleatoria, y después se manda a otro proceso que la eleva al cuadrado. En el segundo método, una matriz aleatoria es construida y elevada al cuadrado (ambas operaciones) sobre otro proceso. Por tanto, el segundo método envía muchos menos datos que el primero.

En este ejemplo de juquete, los dos métodos son fáciles de distinguir y elegir. Sin embargo, en un programa real dieseñar movimiento de datos puede requerir ms reflexin y, probablemente, alguna medida. Por ejemplo, si el primero proceso necesita la matriz `A`, entonces el primer método sera mejor. O, si calcular `A` es costoso y sólo el proceso actual tiene que hacerlo, entonces mover la matriz al otro proceso puede ser inevitable. o, si el proceso actual tiene muy poco que hacer entre las instrucciones `@spawn` y `fetch(Bref)`, podría ser mejor eliminar el paralelismo por completo. O imagine que `rand(1000,1000)` es reemplazado con un a operación más costosa. Entonces podría tener sentido añadir ora instrucción `@spawn` sólo para este paso.

Chapter 26

Variables Globales

Las expresiones ejecutadas remotamente vía `@spawn`, o los cierres especificados para ejecucin remota usando `remotecall` pueden referirse a variables globales. Las vinculaciones globales en el módulo `Main` son tratadas de un modo un poco diferente comparados a las vinculaciones globales en otros módulos. Considere el siguiente trozo de código:

```
A = rand(10,10)
remotecall_fetch(()->foo(A), 2)
```

Tenga en cuenta que `A` es una variable global definida en el espacio de trabajo local. El *worker* 2 no tiene una variable llamada `A` dentro de `Main`. Por tantom el acto de enviar el cierre `() -> foo(A)` al *worker* 2 da como resultado que `Main.A` se defina en 2. `Main.A` sigue existiendo en el worker 2 incluso después de que la llamada `remotecall_fetch` retorne. Las llamadas remotas con referencias globales integradas (solo bajo el módulo `Main`) administran los datos globales de la siguiente manera:

- Se crean nuevos enlaces globales en los trabajadores de destino si se hace referencia a ellos como parte de una llamada remota.
- Las constantes globales se declaran también como constantes en los nodos remotos.
- Globales se reenvían a un *worker* de destino solo en el contexto de una llamada remota, y solo si su valor ha cambiado. Además, el clúster no sincroniza las asignaciones globales entre nodos. Por ejemplo:

  ```
  A = rand(10,10)
  remotecall_fetch(()->foo(A), 2) # worker 2
  A = rand(10,10)
  remotecall_fetch(()->foo(A), 3) # worker 3
  A = nothing
  ```

  Ejecutar este trozo de código da como resultado que `Main.A` del *worker* 2 tenga un valor diferente del que tiene en `Main.A` del *worker* 3, mientras que el valor de `Main.A` en el nodo 1 se fija a `nothing`.

Como se habrá dado cuenta, aunque la memoria asociada con los globales se puede recopilar cuando se reasignan en el maestro, dicha acción no se toma en los *workers* ya que los enlaces siguen siendo válidos. `clear!` se puede usar para reasignar manualmente globales específicos en nodos remotos a `nothing` una vez que ya no sean necesarios. Esto liberará cualquier memoria asociada con ellos como parte de un ciclo de recolección de basura regular.

Por lo tanto, los programas deben ser cuidadosos al hacer referencia a los globales en las llamadas remotas. De hecho, es preferible evitarlos por completo si es posible. Si hay que hacer referencia a globales, considere usar bloques `let` para localizar variables globales.

Por ejemplo:

```
julia> A = rand(10,10);

julia> remotecall_fetch(()->A, 2);

julia> B = rand(10,10);

julia> let B = B

           remotecall_fetch(()->B, 2)

       end;

julia> @spawnat 2 whos();

julia>  From worker 2:                                A    800 bytes  10×10 Array{Float64,2}

        From worker 2:                          Base              Module

        From worker 2:                          Core              Module

        From worker 2:                          Main              Module
```

Como puede verse, la variable global A es definidia en el *worker* 2, pero B es capturada como una variable local y por tanto no existe una asignacin para B en *worker* 2.

26.1 Map y Bucles Paralelos

Afortunadamente, muchos cálculos paralelos no requieren movimiento de datos. Un ejemplo común es las simulaciones Monte Carlo, donde muchos procesos pueden manejar simultáneamente pruebas de simulación independientes. Podemos usar `@spawn` para lanzar monedas sobre dos procesos. Primero, se escribiría la siguiente función en `count_heads.jl`:

```
function count_heads(n)
    c::Int = 0
    for i = 1:n
        c += rand(Bool)
    end
    c
end
```

La función `count_heads` simplemente añade juntos n bits aleatorios. He aqui cónmo pueden realizarse algunas pruebas sobre dos máquinas, y añadir juntos los resultados:

```
julia> @everywhere include("count_heads.jl")

julia> a = @spawn count_heads(100000000)
Future(2, 1, 6, Nullable{Any}())

julia> b = @spawn count_heads(100000000)
Future(3, 1, 7, Nullable{Any}())
```

```
julia> fetch(a)+fetch(b)
100001564
```

Este ejemplo demuestra un patrón de programación paralela potente y frecuentemente usado. Muchas iteraciones se ejecutan independientemente sobre varios porocesos, y entonces sus resultados se combinan usando alguna función. El proceso de combinación se denomina *reducción* ya que suele ser la reduccin de rango de un tensor: un vector de números es reducido a un solo número o una matriz es reducida a una sola fila o columna, etc. En código esto suele tener el aspecto del patrón `x = f(x,v[i])`, donde `x` es el acumulador, `f` es la funcin de reducción, y los `v[i]` son los elementos que se reducirán. Es deseable que `f` sea asociativa, para que no importe el orden en el que se realizan las operaciones.

Nótese que nuestro uso de este patrón con `count_heads` puede ser generalizado. Se utilizaron dos instrucciones `@spawn` explícitas, que limitan el paralelismo a dos procesos. Para ejecutar sobre cualquier número de procesos, se puede usar el *bucle for paralelo* que puede escribirse en Julia usando la macro `@parallel` como en este ejemplo:

```
nheads = @parallel (+) for i = 1:200000000
    Int(rand(Bool))
end
```

Esta construcción implementa el patrón de asignar iteraciones a múltiples procesos, y combinarlos con una reducción especificada (en este caso `(+)`). El resultado de cada iteración es tomado como el valor de la última expresión dentro del bucle. La expresión total del bucle paralelo en sí misma se evalúa a la respuesta final.

Debe notar que, aunque los bucles for paralelos tienen un aspecto muy parecido al de los bucles for seriales, su comportamiento es dramaticamente diferente. En particular, las iteraciones no tiene lugar en un orden especificado, y l escritura a variables o arrays no será globalmente visible ya que las iteraciones se ejecutan sobre procesos distintos. Cualquier variable usada dentro del bucle paralelo será copiada y retransmitida a cada proceso.

Por ejemplo, el siguiente código no trabajará como se esperaba:

```
a = zeros(100000)
@parallel for i = 1:100000
    a[i] = i
end
```

Este código no inicializará todo a, ya que cada proceso tendrá una copia separada de él. Los bucles for paralelos como éste deben ser evitados. Afortunadamente, podemos usar los arrays compartidos para sortear esta limitación:

```
a = SharedArray{Float64}(10)
@parallel for i = 1:10
    a[i] = i
end
```

Usar variables ”forasteras” en los bucles paralelos es perfectamente razonable si las variables son de sólo lectura:

```
a = randn(1000)
@parallel (+) for i = 1:100000
    f(a[rand(1:end)])
end
```

En este ejemplo, cada iteración aplica `f` a una muestra elegida aleatoriamente de un vector a compartido por todos los procesos.

Como podía ver, el operador de reducción puede ser omitido si no se necesita En este caso, el bucle se ejecuta de forma asíncrona, es decir, engendra tareas independientes sobre todos los *workers* disponibles y devuelve un array de objetos `Future` inmediatamente sin esperar a su terminación. El código invocador puede esperar a la terminación de los `Future` en un punto posterior mediante una llamada a `fetch()` sobre ellos, o esperar la terminancin al final del bucle prefijándolo con `@sync`, como `@sync @parallel for`.

En algunos casos no se necesita un operador de reducción, y simplemente deseamos aplicar una funcin a todos los enteros en algún rango (o, de forma más general, a todos los elementos de una colección). Esta es otra operación útil llamada *parallel map*, implementada en con la función `pmap()`. Por ejemplo, podríamos computar los valores singularees de varias matrices aleatorias en paralelo de la siguiente forma:

```
julia> M = Matrix{Float64}[rand(1000,1000) for i = 1:10];

julia> pmap(svd, M);
```

La función `pmap()` de Julia está diseñada para el caso de que cada llamada a función realice una gran cantidad de trabajo. En contraste `@parallel for` puede manejar situaciones donde cada iteración es pequeña, quizás incluso sumar dos números. Tanto las funciones `pmap()` como `@parallel for` usan exclusivamente procesos *worker* para la computación paralela. En el caso de `@parallel for`, la reducción final se realiza sobre el proceso principal.

26.2 Sincronización con Referencias Remotas

26.3 Planificación

La plataforma de programación paralela de Julia usa Tareas (también conocidas como Coroutinas) para alternar entre múltiples cálculos. Cada vez que el código realiza una operación de comunicación como `fetch()` o `wait()`, la tarea actual se suspende y un planificador elige otra tarea para ejecutar. Una tarea se reinicia cuando finaliza el evento que está esperando.

Para muchos problemas, no es necesario pensar en las tareas directamente. Sin embargo, pueden usarse para esperar múltiples eventos al mismo tiempo, lo que proporciona una *planificación dinámica*. En la planificación dinámica, un programa decide qué calcular o dónde calcularlo en función de cuándo finalizan otros trabajos. Esto es necesario para cargas de trabajo impredecibles o desequilibradas, donde queremos asignar más trabajo a los procesos solo cuando finalizan sus tareas actuales.

Como ejemplo, considere calcular los valores singulares de matrices de diferentes tamaños:

```
julia> M = Matrix{Float64}[rand(800,800), rand(600,600), rand(800,800), rand(600,600)];

julia> pmap(svd, M);
```

Si un proceso maneja la dos matrices de 800 × 800 y otro maneja las dos matrices de 600 × 600, no obtendremos la mayor escalabilidad posible. La solución es hacer una tarea local para ”alimentar” el trabajo a cada proceso cuando completa su tarea actual. Por ejemplo, considere una implementación simple `pmap()`:

```
function pmap(f, lst)
    np = nprocs()  # determine the number of processes available
    n = length(lst)
    results = Vector{Any}(n)
    i = 1
```

```
    # function to produce the next work item from the queue.
    # in this case it's just an index.
    nextidx() = (idx=i; i+=1; idx)
    @sync begin
        for p=1:np
            if p != myid() || np == 1
                @async begin
                    while true
                        idx = nextidx()
                        if idx > n
                            break
                        end
                        results[idx] = remotecall_fetch(f, p, lst[idx])
                    end
                end
            end
        end
    end
    results
end
```

`@async` es similar a `@spawn`, pero solo ejecuta tareas en el proceso local. Lo usamos para crear una tarea "alimentador" para cada proceso. Cada tarea selecciona el siguiente índice que debe calcularse, luego espera a que termine su proceso, y luego se repite hasta que nos quedemos sin índices. Tenga en cuenta que las tareas "alimentadoras no" comienzan a ejecutarse hasta que la tarea principal llega al final del bloque `@sync`, momento en el cual se somete al control y espera a que se completen todas las tareas locales antes de regresar de la función. Las tareas del "alimentador" pueden compartir el estado a través de `nextidx()` porque todas se ejecutan en el mismo proceso. No se requiere bloqueo, ya que los hilos están programados de forma cooperativa y no apropiativa. Esto significa que los cambios de contexto solo ocurren en puntos bien definidos: en este caso, cuando se llama a `remotecall_fetch()`.

26.4 Canales

La sección sobre tareas (`Task`) en Control de flujo discutió la ejecución de múltiples funciones de forma cooperativa. Los canales (`Channel`) pueden ser bastante útiles para pasar datos entre tareas en ejecución, particularmente aquellas que involucran operaciones de E/S.

Ejemplos de operaciones que implican E/S incluyen la lectura/escritura en archivos, acceso a servicios web, ejecución de programas externos, etc. En todos estos casos, el tiempo de ejecución general puede mejorarse si se pueden ejecutar otras tareas mientras se lee un archivo, o mientras se espera a que se complete un servicio o programa externo.

Un canal se puede visualizar como un conducto, es decir, tiene un extremo de escritura y un extremo de lectura.

- Varios escritores en diferentes tareas pueden escribir en el mismo canal concurrentemente a través de llamadas `put!()`.
- Varios lectores en diferentes tareas pueden leer datos simultáneamente a través de llamadas `take!()`.
- Como ejemplo:

```
# Given Channels c1 and c2,
c1 = Channel(32)
c2 = Channel(32)

# and a function `foo()` which reads items from from c1, processes the item read
# and writes a result to c2,
```

```
function foo()
    while true
        data = take!(c1)
        [...]               # process data
        put!(c2, result)    # write out result
    end
end

# we can schedule `n` instances of `foo()` to be active concurrently.
for _ in 1:n
    @schedule foo()
end
```

- Los canales se crean a través del constructor `Channel{T}(sz)`. El canal solo tendrá objetos de tipo `T`. Si no se especifica el tipo, el canal puede contener objetos de cualquier tipo. `sz` se refiere a la cantidad máxima de elementos que pueden mantenerse en el canal en cualquier momento. Por ejemplo, `Channel(32)` crea un canal que puede contener un máximo de 32 objetos de cualquier tipo. Un `Channel{MyType}(64)` puede contener hasta 64 objetos de `MyType` en cualquier momento.
- Si un `Channel` está vacío, los lectores (en una llamada `take!()`) se bloquearán hasta que los datos estén disponibles.
- Si un `Channel` está lleno, los escritores (en una llamada `put!()`) se bloquearán hasta que haya espacio disponible.
- `isready()` comprueba la presencia de cualquier objeto en el canal, mientras que `wait()` espera a que un objeto esté disponible.
- Un `Channel` está inicialmente en un estado abierto. Esto significa que puede leerse y escribirse libremente a través de llamadas `take!()` y `put!()`. `close()` cierra un `Channel`. En un `Channel` cerrado, la función `put!()` fallará. Por ejemplo:

```
julia> c = Channel(2);

julia> put!(c, 1) # `put!` on an open channel succeeds
1

julia> close(c);

julia> put!(c, 2) # `put!` on a closed channel throws an exception.
ERROR: InvalidStateException("Channel is closed.",:closed)
[...]
```

- `take!()` y `fetch()` (que recupera pero no elimina el valor) en un canal cerrado devuelve con éxito cualquier valor existente hasta que se vacíe. Continuando con el ejemplo anterior:

```
julia> fetch(c) # Any number of `fetch` calls succeed.
1

julia> fetch(c)
1

julia> take!(c) # The first `take!` removes the value.
1

julia> take!(c) # No more data available on a closed channel.
```

```
ERROR: InvalidStateException("Channel is closed.",:closed)
[...]
```

Un canal se puede usar como un objeto iterable en un bucle `for`, en cuyo caso el bucle se ejecuta mientras el canal tenga datos o esté abierto. La variable del bucle toma todos los valores agregados al canal. El bucle `for` finaliza una vez que el canal se cierra y se vacía.

Por ejemplo, lo siguiente haría que el bucle `for` esperara más datos:

```
julia> c = Channel{Int}(10);

julia> foreach(i->put!(c, i), 1:3) # add a few entries

julia> data = [i for i in c]
```

mientras que esto retornará después de leer todos los datos:

```
julia> c = Channel{Int}(10);

julia> foreach(i->put!(c, i), 1:3); # add a few entries

julia> close(c);                    # `for` loops can exit

julia> data = [i for i in c]
3-element Array{Int64,1}:
 1
 2
 3
```

Consideremos un ejemplo simple utilizando canales para la comunicación entre tareas. Comenzamos 4 tareas para procesar los datos de un solo canal `jobs`. Los trabajos, identificados por un identificador (`job_id`), se escriben en el canal. Cada tarea en esta simulación lee un `job_id`, espera una cantidad aleatoria de tiempo y escribe una tupla de `job_id` y el tiempo simulado en el canal de resultados. Finalmente todos los `resultados` se imprimen.

```
julia> const jobs = Channel{Int}(32);

julia> const results = Channel{Tuple}(32);

julia> function do_work()

           for job_id in jobs

               exec_time = rand()

               sleep(exec_time)               # simulates elapsed time doing actual work

                                              # typically performed externally.

               put!(results, (job_id, exec_time))

           end

       end;
```

```
julia> function make_jobs(n)
           for i in 1:n
               put!(jobs, i)
           end
       end;
julia> n = 12;
julia> @schedule make_jobs(n); # feed the jobs channel with "n" jobs
julia> for i in 1:4 # start 4 tasks to process requests in parallel
           @schedule do_work()
       end
julia> @elapsed while n > 0 # print out results
           job_id, exec_time = take!(results)
           println("$job_id finished in $(round(exec_time,2)) seconds")
           n = n - 1
       end
4 finished in 0.22 seconds
3 finished in 0.45 seconds
1 finished in 0.5 seconds
7 finished in 0.14 seconds
2 finished in 0.78 seconds
5 finished in 0.9 seconds
9 finished in 0.36 seconds
6 finished in 0.87 seconds
8 finished in 0.79 seconds
10 finished in 0.64 seconds
12 finished in 0.5 seconds
11 finished in 0.97 seconds
0.029772311
```

La versión actual de Julia multiplexa todas las tareas en un solo hilo del sistema operativo. Por lo tanto, aunque las tareas que implican operaciones de E/S se benefician de la ejecución en paralelo, las tareas vinculadas a la computación se ejecutan efectivamente de forma secuencial en un solo hilo del sistema operativo. Las versiones futuras de Julia pueden soportar la planificación de tareas en múltiples subprocesos, en cuyo caso las tareas vínculadas al cálculo también verán los beneficios de la ejecución en paralelo.

26.5 Referencias remotas y AbstractChannels

Las referencias remotas siempre se refieren a una implementación de un `AbstractChannel`.

Se requiere una implementación concreta de un `AbstractChannel` (como `Channel`) para implementar `put!()`, `take!()`, `fetch()`, `isready()` y `wait()`. El objeto remoto al que se hace referencia con un `Future` se almacena en `Channel{Any}(1)`, es decir, un`Channel` de tamaño 1 capaz de contener objetos del tipo `Any`.

`RemoteChannel`, que es reescribible, puede señalar cualquier tipo y tamaño de canales, o cualquier otra implementación de `AbstractChannel`.

El constructor `RemoteChannel(f::Function, pid)()` permite construir referencias a canales que contienen más de un valor de un tipo específico. `f()` es una función ejecutada en un `pid` y debe devolver un `AbstractChannel`.

Por ejemplo, `RemoteChannel(()->Channel{Int}(10), pid)`, devolverá una referencia a un canal de tipo `Int` y tamaño 10. El canal existe sobre el *worker* `pid`.

Los métodos `put!()`, `take!()`, `fetch()`, `isready()` y `wait()` de un `RemoteChannel` son proxys en el backing store en el proceso remoto.

`RemoteChannel` se puede usar para referirse a los objetos `AbstractChannel` implementados por el usuario. Un ejemplo simple de esto se proporciona en `examples/dictchannel.jl` que usa un diccionario como su almacén remoto.

26.6 Channels y RemoteChannels

- Un objeto `Channel` es local a un proceso. El *worker* 2 no puede referirse directamente a un `Channel` sobre el *worker* 3 y viceversa. Un `RemoteChannel`, sin embargo, puedo poner y tomar valores entre *workers*.
- Un `RemoteChannel` se puede considerar como un *manejador* para un `Channel`.
- La identificación del proceso, `pid`, asociada con un `RemoteChannel` identifica el proceso donde el almacén de respaldo, es decir, el `Channel` de respaldo existe.
- Cualquier proceso con una referencia a `RemoteChannel` puede poner y tomar elementos del canal. Los datos se envían automáticamente a (o se recuperan de) el proceso al que está asociado `RemoteChannel`.
- Serializar un `Channel` también serializa cualquier dato presente en el canal. Deserializarlo, por lo tanto, efectivamente hace una copia del objeto original.
- Por otro lado, serializar un `RemoteChannel` solo implica la serialización de un identificador que identifica la ubicación y la instancia del `Channel` al que hace referencia el manejador. Un objeto `RemoteChannel` deserializado (en cualquier *worker*), por lo tanto, también apunta al mismo almacén de respaldo que el original.

El ejemplo de canales anterior puede modificarse para la comunicación entre procesos, como se muestra a continuación.

Comenzamos 4 trabajadores para procesar un solo canal remoto `jobs`. Los trabajos, identificados por una identificación (`job_id`), se escriben en el canal. Cada tarea de ejecución remota en esta simulación lee un `job_id`, espera una cantidad aleatoria de tiempo y escribe una tupla de`job_id`, tiempo tomado y su propio `pid` en el canal de resultados. Finalmente todos los `resultados` se imprimen en el proceso maestro.

```
julia> addprocs(4); # add worker processes

julia> const jobs = RemoteChannel(()->Channel{Int}(32));

julia> const results = RemoteChannel(()->Channel{Tuple}(32));

julia> @everywhere function do_work(jobs, results) # define work function everywhere
```

```
            while true
                job_id = take!(jobs)
                exec_time = rand()
                sleep(exec_time) # simulates elapsed time doing actual work
                put!(results, (job_id, exec_time, myid()))
            end
        end
julia> function make_jobs(n)
            for i in 1:n
                put!(jobs, i)
            end
        end;
julia> n = 12;
julia> @schedule make_jobs(n); # feed the jobs channel with "n" jobs
julia> for p in workers() # start tasks on the workers to process requests in parallel
            @async remote_do(do_work, p, jobs, results)
        end
julia> @elapsed while n > 0 # print out results
            job_id, exec_time, where = take!(results)
            println("$job_id finished in $(round(exec_time,2)) seconds on worker $where")
            n = n - 1
        end
1 finished in 0.18 seconds on worker 4
2 finished in 0.26 seconds on worker 5
6 finished in 0.12 seconds on worker 4
7 finished in 0.18 seconds on worker 4
5 finished in 0.35 seconds on worker 5
4 finished in 0.68 seconds on worker 2
3 finished in 0.73 seconds on worker 3
11 finished in 0.01 seconds on worker 3
12 finished in 0.02 seconds on worker 3
9 finished in 0.26 seconds on worker 5
8 finished in 0.57 seconds on worker 4
10 finished in 0.58 seconds on worker 2
0.055971741
```

26.7 Referencias Remotas y Recolección de Basura Distribuida

Los objetos a los que se refieren las referencias remotas se pueden liberar solo cuando se eliminan *todas* las referencias retenidas en el clúster.

El nodo donde se almacena el valor realiza un seguimiento de cuáles de los trabajadores tienen una referencia. Cada vez que un `RemoteChannel` o un (unfetched) `Future` se serializa a un *worker*, se notifica el nodo al que apunta la referencia. Y cada vez que un `RemoteChannel` o un (unfetched) `Future` es sometido a recolección de basura localmente, el nodo que posee el valor es nuevamente notificado.

Las notificaciones se realizan a través del envío de mensajes de ”seguimiento”: un mensaje de ”agregar referencia” cuando una referencia se serializa a un proceso diferente y un mensaje de ”eliminación de referencia” cuando una referencia se recolecta localmente.

Como los `Future`s son de escritura única y se almacenan en caché localmete, el acto de `fetch()`ing un `Future` también actualiza la información de seguimiento de referencia en el nodo que posee el valor.

El nodo que posee el valor lo libera una vez que se borran todas las referencias a él.

Con `Future`s, la serialización de un `Future` ya obtenido a un nodo diferente también envía el valor ya que el almacén remoto original puede haber recolectado el valor en ese momento.

Es importante tener en cuenta que *cuando* un objeto se recolecta basura localmente depende del tamaño del objeto y la presión de la memoria actual en el sistema.

En el caso de referencias remotas, el tamaño del objeto de referencia local es bastante pequeño, mientras que el valor almacenado en el nodo remoto puede ser bastante grande. Dado que el objeto local puede no receolectarse inmediatamente, es una buena práctica llamar explícitamente a `finalize()` en instancias locales de un `RemoteChannel`, o en unfetched `Future`s. Como llamar a `fetch()` sobre un `Future` también elimina su referencia del almacén remoto, esto no es necesario en fetched `Future`s. Llamar explícitamente a `finalize()` da como resultado un mensaje inmediato enviado al nodo remoto para continuar y eliminar su referencia al valor.

Una vez finalizado, una referencia deja de ser válida y no se puede usar en ninguna otra llamada.

26.8 Arrays Compartidos

Los arrays compartidos usan memoria compartida del sistema para hacer corresponder el mismo array a través de muchos procesos. Aunque hay algunas similaridades a un `DArray`, el comportamiento de un `SharedArray` es bastante diferente. En un `SharedArray`, cada proceso tiene acceso local justo a un trozo de los datos, y do hay dos procesos que compartan el mismo trozo; en contraste, en un `SharedArray` cada proceso ”participante” tiene acceso al array completo. Un `SharedArray` es una buena elección cuando uno quiere tener una gran cantidad de datos conjuntamente accesibles a dos o más procesos sobre la misma máquina.

La indexación de los `SharedArray`s funciona justo como con los arrays regulares, y es eficiente debido a que la memoria subyacente está disponible al proceso local. Por tanto, la mayoría de los algoritmos trabajan de forma natural sobre los `SharedArray`s, aunque en modo uniproceso. En casos donde un algoritmo insiste sobre una entrada `Array`, el array subyacente se puede recuperar desde un `SharedArray` llamando a `sdata()`. Para otros tipos de `AbstractArray`, `sdata()` simplemente devuelve el objeto, por lo que es seguro usar `sdata()` en cualquier objeto de tipo`Array`.

El constructor par aun array compartido es de la forma:

```
SharedArray{T,N}(dims::NTuple; init=false, pids=Int[])
```

que crea un array compartido `N`-dimensional de un tipo de bits `T` y `dims` de tamaño en los procesos especificados por `pids`. A diferencia de los arrays distribuidos, a un array compartido solo se puede acceder desde los *workers* participantes especificados por el argumento denominado `pids` (y el proceso de creación también, si está en el mismo host).

Si se especifica una función `init`, con signatura `initfn(S::SharedArray)`, se llama a todos los trabajadores participantes. Puede especificar que cada trabajador ejecute la función `init` en una parte distinta de la matriz, paralelizando así la inicialización.

He aquí un breve ejemplo:

```
julia> addprocs(3)
3-element Array{Int64,1}:
 2
 3
 4

julia> S = SharedArray{Int,2}((3,4), init = S -> S[Base.localindexes(S)] = myid())
3×4 SharedArray{Int64,2}:
 2  2  3  4
 2  3  3  4
 2  3  4  4

julia> S[3,2] = 7
7

julia> S
3×4 SharedArray{Int64,2}:
 2  2  3  4
 2  3  3  4
 2  7  4  4
```

`Base.localindexes()` proporciona rangos unidimensionales disjuntos de índices, y a veces es conveniente para dividir tareas entre procesos. Uno puede, por supuesto, dividir el trabajo de la manera que desee:

```
julia> S = SharedArray{Int,2}((3,4), init = S -> S[indexpids(S):length(procs(S)):length(S)] =
↪  myid())
3×4 SharedArray{Int64,2}:
 2  2  2  2
 3  3  3  3
 4  4  4  4
```

Como todos los procesos tienen acceso a los datos subyacentes, uno tiene que tener cuidado de no generar conflictos. Por ejemplo:

```
@sync begin
    for p in procs(S)
        @async begin
            remotecall_wait(fill!, p, S, p)
        end
    end
end
```

daría como resultado un comportamiento indefinido. Debido a que cada proceso llena la matriz *entera* con su propio `pid`, el proceso que sea el último en ejecutarse (para cualquier elemento en particular deS) tendrá su `pid` retenido.

Como un ejemplo más extenso y complejo, considere ejecutar el siguiente ”kernel” en paralelo:

```
q[i,j,t+1] = q[i,j,t] + u[i,j,t]
```

En este caso, si tratamos de dividir el trabajo utilizando un índice unidimensional, es probable que tengamos problemas: si `q[i,j,t]` está cerca del final del bloque asignado a un *worker* y `q [i,j,t+1]` está cerca del comienzo del bloque asignado a otro, es muy probable que `q[i,j,t]` no esté listo en el momento en que se necesita para computar `q [i,j,t+1]`. En tales casos, es mejor dividir manualmente la matriz. Vamos a dividirnos a lo largo de la segunda dimensión. Defina una función que devuelve los índices `(irange, jrange)` asignados a este *worker*:

```
julia> @everywhere function myrange(q::SharedArray)

           idx = indexpids(q)

           if idx == 0 # This worker is not assigned a piece

               return 1:0, 1:0

           end

           nchunks = length(procs(q))

           splits = [round(Int, s) for s in linspace(0,size(q,2),nchunks+1)]

           1:size(q,1), splits[idx]+1:splits[idx+1]

       end
```

A continuación, se define el kernel:

```
julia> @everywhere function advection_chunk!(q, u, irange, jrange, trange)

           @show (irange, jrange, trange)  # display so we can see what's happening

           for t in trange, j in jrange, i in irange

               q[i,j,t+1] = q[i,j,t] + u[i,j,t]

           end

           q

       end
```

Podemos también definir un *wrapper* de conveniencia para una implementación de `SharedArray`

```
julia> @everywhere advection_shared_chunk!(q, u) =

           advection_chunk!(q, u, myrange(q)..., 1:size(q,3)-1)
```

Ahora comparemos las tres versiones diferentes, una que ejecuta en un solo proceso:

```
julia> advection_serial!(q, u) = advection_chunk!(q, u, 1:size(q,1), 1:size(q,2), 1:size(q,3)-1);
```

una que usa `@parallel`:

```
julia> function advection_parallel!(q, u)
           for t = 1:size(q,3)-1
               @sync @parallel for j = 1:size(q,2)
                   for i = 1:size(q,1)
                       q[i,j,t+1]= q[i,j,t] + u[i,j,t]
                   end
               end
           end
           q
       end;
```

y una que delega en trozos:

```
julia> function advection_shared!(q, u)
           @sync begin
               for p in procs(q)
                   @async remotecall_wait(advection_shared_chunk!, p, q, u)
               end
           end
           q
       end;
```

Si creamos un `SharedArrays` y controlamos el tiempo de estas funciones, obtendremos sl siguiente resultado (con `julia -p 4`):

```
julia> q = SharedArray{Float64,3}((500,500,500));

julia> u = SharedArray{Float64,3}((500,500,500));
```

Ejecutemos las funciones una vez para tenga lugar la compilación JIT y `@time`, y pasemos después a una segunda ejecución:

```
julia> @time advection_serial!(q, u);
(irange,jrange,trange) = (1:500,1:500,1:499)
 830.220 milliseconds (216 allocations: 13820 bytes)

julia> @time advection_parallel!(q, u);
```

```
  2.495 seconds      (3999 k allocations: 289 MB, 2.09% gc time)

julia> @time advection_shared!(q,u);

       From worker 2:       (irange,jrange,trange) = (1:500,1:125,1:499)

       From worker 4:       (irange,jrange,trange) = (1:500,251:375,1:499)

       From worker 3:       (irange,jrange,trange) = (1:500,126:250,1:499)

       From worker 5:       (irange,jrange,trange) = (1:500,376:500,1:499)
238.119 milliseconds (2264 allocations: 169 KB)
```

La mayor ventaja de `advection_shared!` es que minimiza el tráfico entre los *workers* permitiendo que cada uno compute para un tiempo extendido sobre la pieiza asignada.

26.9 Arrays Compartidos y Recolección de Basura Distribuida

Al igual que las referencias remotas, las matrices compartidas también dependen de la recolección de basura en el nodo de creación para liberar referencias de todos los *workers* participantes. El código que crea muchos arrays compartidas de vida corta se beneficiaría de finalizar explícitamente estos objetos tan pronto como sea posible. Esto da como resultado que tanto la memoria como los manejadores de archivos mapeen el segmento compartido que se libera antes.

26.10 ClusterManagers

El lanzamiento, la administración y la comunicación en red de los procesos de Julia en un clúster lógico se realiza a través de los administradores del clúster. Un `ClusterManager` es responsable de

- Lanzar procesos *worker* en un entorno clúster
- gestión de eventos durante la vida de cada *worker*
- opcionalmente, proporcionar transporte de datos

Un clúster Julia tiene las siguientes características:

- El proceso inicial de Julia, también llamado `master`, es especial y tiene un`id` de 1.
- Solo el proceso `master` puede agregar o eliminar procesos de trabajo.
- Todos los procesos pueden comunicarse directamente entre ellos.

Las conexiones entre los *workers* (utilizando el transporte integrado de TCP/IP) se establecen de la siguiente manera:

- `addprocs()` se invoca en el proceso maestro con un objeto `ClusterManager`.
- `addprocs()` llama al método apropiado `launch()` que engendra el número requerido de procesos de trabajo en las máquinas apropiadas.
- Cada *worker* comienza a escuchar en un puerto libre y escribe su información de host y puerto en `STDOUT`.
- El administrador del clúster captura el `STDOUT` de cada *worker* y lo pone a disposición del proceso maestro.

- El proceso maestro analiza esta información y configura conexiones TCP/IP para cada *worker*.
- Todos los *workers* también reciben notificaciones de otros trabajadores en el clúster.
- Cada *worker* se conecta con todos los *worker* cuyo `id` es menor que su propio`id`.
- De esta forma se establece una red de malla, en la que cada *worker* está directamente conectado con cada otro *worker*.

Aunque la capa de transporte predeterminada usa el `TCPSocket` simple, es posible que un clúster Julia proporcione su propio transporte.

Julia proporciona dos administradores de clúster integrados:

- `LocalManager`, usado cuando se llama a `addprocs()` o a `addprocs(np::Integer)`
- `SSHManager`, utilizado cuando se llama a [`addprocs(hostnames::Array)`](@ref con una lista de nombres de host

`LocalManager` se utiliza para iniciar *workers* adicionales en el mismo host, aprovechando de ese modo los núcleos múltiples y el hardware multiprocesador.

Por lo tanto, un administrador de clúster mínimo necesitaría:

- ser un subtipo del resumen `ClusterManager`
- implementar `launch()`, un método responsable del lanzamiento de nuevos *workers*
- implementar `manage()`, que se invoca en varios eventos durante la vida de un *worker* (por ejemplo, enviando una señal de interrupción)

`addprocs(manager::FooManager)` requiere `FooManager` para implementar:

```
function launch(manager::FooManager, params::Dict, launched::Array, c::Condition)
    [...]
end

function manage(manager::FooManager, id::Integer, config::WorkerConfig, op::Symbol)
    [...]
end
```

Como ejemplo, veamos cómo se implementa el `LocalManager`, el administrador responsable de iniciar los *workers* en el mismo host:

```
struct LocalManager <: ClusterManager
    np::Integer
end

function launch(manager::LocalManager, params::Dict, launched::Array, c::Condition)
    [...]
end

function manage(manager::LocalManager, id::Integer, config::WorkerConfig, op::Symbol)
    [...]
end
```

El método `launch()` toma los siguientes argumentos:

- `manager::ClusterManager`: el administrador de clúster al que se llama con `addprocs()`
- `params::Dict`: todos los argumentos palabra clave pasados a `addprocs()`
- `launched::Array`: el array al que agregar uno o más objetos `WorkerConfig`
- `c::Condition`:la variable de condición que se notificará cuando se inicien los trabajadores

El método `launch()` se llama asincrónicamente en una tarea separada. La finalización de esta tarea indica que se han lanzado todos los *workers* solicitados. Por lo tanto, la función `launch()` DEBE salir tan pronto como se hayan lanzado todos los *workers* solicitados.

Los trabajadores recién lanzados están conectados entre sí y el proceso maestro de una manera integral. Al especificar el argumento de la línea de comando `--worker <cookie>` los procesos iniciados se inicializan a sí mismos como trabajadores y las conexiones se configuran a través de sockets TCP / IP. Optionally, `--bind-to bind_addr[:port]` may also be specified to enable other workers to connect to it at the specified `bind_addr` and `port`. Esto es útil para hosts multi-homed.

Como ejemplo de transporte no TCP / IP, una implementación puede optar por utilizar MPI, en cuyo caso `-worker` NO se debe especificar. En cambio, los trabajadores recién lanzados deberían llamar `init_worker (cookie)` antes de usar cualquiera de las construcciones paralelas.

Para cada trabajador puesto en marcha, el método `launch()` debe agregar un objeto `WorkerConfig` (con los campos apropiados inicializados) al `launched`

```
mutable struct WorkerConfig
    # Common fields relevant to all cluster managers
    io::Nullable{IO}
    host::Nullable{AbstractString}
    port::Nullable{Integer}

    # Used when launching additional workers at a host
    count::Nullable{Union{Int, Symbol}}
    exename::Nullable{AbstractString}
    exeflags::Nullable{Cmd}

    # External cluster managers can use this to store information at a per-worker level
    # Can be a dict if multiple fields need to be stored.
    userdata::Nullable{Any}

    # SSHManager / SSH tunnel connections to workers
    tunnel::Nullable{Bool}
    bind_addr::Nullable{AbstractString}
    sshflags::Nullable{Cmd}
    max_parallel::Nullable{Integer}

    connect_at::Nullable{Any}

    [...]
end
```

La mayoría de los campos en `WorkerConfig` son utilizados por los administradores incorporados. Gestores de cluster personalizados normalmente especificarían solo `io` o`host` / `port`:

- Si se especifica `io`, se usa para leer información de host / puerto. Un *worker* Julia imprime su dirección y puerto de enlace al inicio. Esto permite a los *workers* Julia escuchar en cualquier puerto libre disponible en lugar de requerir que los puertos de los trabajadores se configuren manualmente.
- Si `io` no está especificado,`host` y `port` se utilizan para conectarse.
- `count`,`exename` y `exeflags` son relevantes para el lanzamiento de trabajadores adicionales de un trabajador. Por ejemplo, un administrador de clúster puede iniciar un solo trabajador por nodo y usarlo para iniciar trabajadores adicionales.
  - `count` con un valor enteron lanzará un total de n *workers*.
  - `count` con un valor de `: auto` lanzará tantos trabajadores como la cantidad de núcleos en esa máquina.
  - `exename` es el nombre del ejecutable`julia` que incluye la ruta completa.
  - `exeflags` debe establecerse en los argumentos de línea de comando necesarios para los nuevos *workers*.
- `tunnel`,`bind_addr`, `sshflags` y`max_parallel` se usan cuando se requiere un túnel ssh para conectarse con los *workers* del proceso maestro.
- `userdata` se proporciona para que los administradores de clúster personalizados almacenen su propia información específica del *worker*.

`manage (manager :: FooManager, id :: Integer, config :: WorkerConfig, op :: Symbol)` se llama en diferentes momentos durante la vida del trabajador con valores op apropiados:

- con `: register` / `: deregister` cuando un trabajador se agrega / elimina del grupo de *workers* de Julia.
- con `: interrupt` cuando se invoca `interrupt(workers)`. El `ClusterManager` debe señalar al *worker* apropiado con una señal de interrupción.
- con `: finalize` para fines de limpieza.

26.11 Administradores de Clúster con Transportes Personalizados

Reemplazar las conexiones por defecto de socket TCP/IP con una capa de transporte personalizada es un poco más complicado. Cada proceso de Julia tiene tantas tareas de comunicación como los *workers* a los que está conectado. Por ejemplo, considere un clúster de Julia de 32 procesos en una red de malla todos contra todos:

- Cada proceso de Julia tiene 31 tareas de comunicación.
- Cada tarea maneja todos los mensajes entrantes desde un solo *worker* remoto en un bucle de procesamiento de mensajes.
- El bucle de procesamiento de mensajes espera en un objeto `IO` (por ejemplo, un`TCPSocket` en la implementación predeterminada), lee un mensaje completo, lo procesa y espera el siguiente.
- El envío de mensajes a un proceso se realiza directamente desde cualquier tarea Julia, no solo tareas de comunicación, nuevamente, a través del objeto `IO` apropiado.

Reemplazar el transporte predeterminado requiere que la nueva implementación establezca conexiones con *workers* remotos y que proporcione los objetos `IO` apropiados para que los lazos de procesamiento de mensajes puedan esperar. Las devoluciones de llamada específicas del administrador que se implementarán son:

```
connect(manager::FooManager, pid::Integer, config::WorkerConfig)
kill(manager::FooManager, pid::Int, config::WorkerConfig)
```

La implementación por defecto (que usa sockets TCP/IP) se implementa como `connect (manager::ClusterManager, pid::Integer, config::WorkerConfig)`.

`connect` debería devolver un par de objetos`IO`, uno para leer los datos enviados por el `pid` del *worker*, y el otro para escribir datos que deben ser enviados al `pid` del *worker*. Los administradores de clústeres personalizados pueden usar un `BufferStream` en memoria como la conexión de datos proxy entre el *worker* personalizado, posiblemente transporte no-`IO`y la infraestructura paralela incorporada de Julia.

Un `BufferStream` es un `IOBuffer` en memoria que se comporta como un `IO` - es un flujo que puede manejarse de forma asíncrona.

La carpeta `examples/clustermanager/0MQ` contiene un ejemplo del uso de ZeroMQ para conectar *workers* Julia en una topología en estrella con un intermediario 0MQ en el medio. Nota: Los procesos de Julia todavía están todos *lógicamente* conectados entre sí: cualquier trabajador puede enviar mensajes a cualquier otro trabajador directamente sin que se tenga conocimiento de que se está usando 0MQ como capa de transporte.

Al usar transportes personalizados:

- Los *workers* de Julia NO deben comenzar con `--worker`. Comenzar con `--worker` dará como resultado que los trabajadores recién lanzados adopten de forma predeterminada la implementación de transporte de socket TCP/IP.
- Para cada conexión lógica entrante con un *worker*, se deben llamar `Base.process_messages(rd::IO, wr::IO)()`. Esto inicia una nueva tarea que maneja la lectura y escritura de mensajes desde / hacia el trabajador representado por los objetos `IO`.
- `init_worker(cookie, manager::FooManager)` DEBE invocarse como parte de la inicialización del proceso de trabajo.
- El campo `connect_at::Any` en `WorkerConfig` puede ser configurado por el administrador del clúster cuando se invoca `launch()`. El valor de este campo se transfiere en todas las devoluciones de llamada `connect()`. Por lo general, transmite información sobre *cómo conectarse* a un *worker*. Por ejemplo, el transporte de socket TCP/IP utiliza este campo para especificar la tupla `(host, port)` en la que se conecta a un *worker*.

`kill (manager, pid, config)` se llama para eliminar un *worker* del clúster. En el proceso maestro, los objetos `IO` correspondientes deben ser cerrados por la implementación para garantizar una limpieza adecuada. La implementación predeterminada simplemente ejecuta una llamada `exit()` en el *worker* remoto especificado.

`examples/clustermanager/simple` es un ejemplo que muestra una implementación simple usando el dominio UNIX enchufes para la configuración del clúster.

26.12 Requisitos de Red para LocalManager y SSHManager

Los clústeres de Julia están diseñados para ejecutarse en entornos ya protegidos en infraestructura, como laptops locales, clusters departamentales o incluso en la nube. Esta sección cubre los requisitos de seguridad de red para los `LocalManager` y `SSHManager` incorporados:

- El proceso maestro no escucha en ningún puerto. Solo se conecta con los *workers*.
- Each worker binds to only one of the local interfaces and listens on the first free port starting from `9009`.

- `LocalManager`, usado por `addprocs(N)`, por defecto se une solo a la interfaz *loopback*. Esto significa que los trabajadores que comenzaron más adelante en los hosts remotos (o por cualquier persona con intenciones maliciosas) no pueden conectarse al clúster. Un `addprocs(4)` seguido de un `addprocs(["remote_host"])` fallará. Algunos usuarios pueden necesitar crear un clúster que comprenda su sistema local y algunos sistemas remotos. Esto se puede hacer solicitando explícitamente que `LocalManager` se vincule a una interfaz de red externa mediante el argumento de la palabra clave`restrict`: `addprocs (4; restrict = false)`.

- `SSHManager`, utilizado por`addprocs (list_of_remote_hosts)`, inicia trabajadores en hosts remotos a través de SSH. Por defecto, SSH solo se usa para iniciar los trabajadores de Julia. Las conexiones subsiguientes de maestro-trabajador y trabajador-trabajador usan conectores TCP / IP sin cifrar. Los hosts remotos deben tener habilitado el inicio de sesión sin contraseña. Se pueden especificar indicadores o credenciales SSH adicionales a través del argumento de palabra clave `sshflags`.

- `addprocs (list_of_remote_hosts; tunnel = true, sshflags = <ssh keys y otros flags>)` es útil cuando deseamos usar conexiones SSH para el maestro trabajador también. Un escenario típico para esto es una computadora portátil local que ejecuta el REPL de Julia (es decir, el maestro) con el resto del clúster en la nube, por ejemplo en Amazon EC2. En este caso, solo se debe abrir el puerto 22 en el clúster remoto junto con el cliente SSH autenticado a través de la infraestructura de clave pública (PKI). Las credenciales de autenticación se pueden suministrar a través de `sshflags`, por ejemplo `sshflags =`-e <keyfile>`.

Note that worker-worker connections are still plain TCP and the local security policy on the remote cluster must allow for free connections between worker nodes, at least for ports 9009 and above.

```
Asegurar y encriptar todo el tráfico de trabajador-trabajador (a través de SSH) o encriptar
    mensajes individuales se puede hacer a través de un ClusterManager personalizado.
```

26.13 Cluster Cookie

Todos los procesos en un clúster comparten la misma cookie que, de forma predeterminada, es una cadena generada aleatoriamente en el proceso maestro:

- `Base.cluster_cookie()` devuelve la cookie, mientras `Base.cluster_cookie(cookie)()` lo configura y devuelve la nueva cookie.

- Todas las conexiones están autenticadas en ambos lados para garantizar que solo los *workers* iniciados por el maestro puedan conectarse entre sí.

- The cookie must be passed to the workers at startup via argument `--worker <cookie>`. Custom ClusterManagers can retrieve the cookie on the master by calling `Base.cluster_cookie()`. Cluster managers not using the default TCP/IP transport (and hence not specifying `--worker`) must call `init_worker(cookie, manager)` with the same cookie as on the master.

Tenga en cuenta que los entornos que requieren mayores niveles de seguridad pueden implementar esto a través de un `ClusterManager` personalizado. Por ejemplo, las cookies se pueden compartir previamente y, por lo tanto, no se especifican como un argumento de inicio.

26.14 Specifying Network Topology (Experimental)

El argumento de palabra clave `topología` pasado a `addprocs` se usa para especificar cómo los trabajadores deben estar conectados entre sí:

- `: all_to_all`, el valor predeterminado: todos los trabajadores están conectados entre sí.

- : `master_slave`: solo el proceso del controlador, es decir,`pid` 1, tiene conexiones con los trabajadores.
- : `custom`: el método`launch` del administrador del clúster especifica la topología de conexión a través del campos `ident` y`connect_idents` en `WorkerConfig`. Un trabajador con un cluster-manager-provided identidad `ident` se conectará a todos los trabajadores especificados en `connect_idents`.

Actualmente, enviar un mensaje entre *workers* desconectados genera un error. Este comportamiento, al igual que la funcionalidad y la interfaz, debe considerarse de naturaleza experimental y puede cambiar en versiones futuras.

26.15 Multi-Threading (Experimental)

Además de las tareas, llamadas remotas y referencias remotas, Julia desde la `v0.5` hacia delante admitirá de forma nativa soporte para multi-hilo. Tenga en cuenta que esta sección es experimental y las interfaces pueden cambiar en el futuro.

Setup

Por defecto, Julia se inicia con un único hilo de ejecución. Esto se puede verificar utilizando el mandato `Threads.nthreads()`:

```
julia> Threads.nthreads()
1
```

El número de hilos con los que arranca Julia está controlado por una variable de entorno llamada `JULIA_NUM_THREADS`. Ahora, comencemos Julia con 4 hilos:

```
export JULIA_NUM_THREADS=4
```

(El mandato anterior funciona en shells de Bourne shells de Linux y OSX. Tenga en cuenta que si usa un C shell en estas plataformas, debe usar la palabra clave `set` en lugar de`export`. Si está en Windows, inicie la línea de órdenes en la ubicación de `julia.exe` y use `set` en lugar de `export`.)

Verifiquemos que hay 4 hilos a nuestra disposición.

```
julia> Threads.nthreads()
4
```

Pero actualmente estamos en el hilo maestro. Para verificar, usamos el mandato `Threads.threadid()`

```
julia> Threads.threadid()
1
```

La Macro `@threads`

Vamos a trabajar un ejemplo simple usando nuestros hilos nativos. Creemos un array de ceros:

```
julia> a = zeros(10)
10-element Array{Float64,1}:
 0.0
 0.0
 0.0
 0.0
 0.0
```

```
 0.0
 0.0
 0.0
 0.0
 0.0
```

Operemos sobre este array de forma simultánea utilizando 4 hilos. Haremos que cada hilo escriba su ID de hilo en cada ubicación.

Julia soporta bucles paralelos utilizando la macro `Threads.@threads`. Esta macro está fijada delante de un bucle `for` para indicar a Julia que el bucle es una región con múltiples subprocesos:

```
julia> Threads.@threads for i = 1:10

           a[i] = Threads.threadid()

       end
```

El espacio de iteración se divide entre los hilos, después de lo cual cada hilo escribe su ID de hilo a sus ubicaciones asignadas:

```
julia> a
10-element Array{Float64,1}:
 1.0
 1.0
 1.0
 2.0
 2.0
 2.0
 3.0
 3.0
 4.0
 4.0
```

Tenga en cuenta que `Threads.@threads` no tiene un parámetro de reducción opcional como `@parallel`.

26.16 @threadcall (Experimental)

Todas las tareas de E/S, temporizadores, comandos REPL, etc. se multiplexan en una sola cadena del sistema operativo mediante un bucle de eventos. Una versión parcheada de libuv (http://docs.libuv.org/en/v1.x/] proporciona esta funcionalidad. Los puntos de rendimiento proporcionan la planificación cooperativa de tareas múltiples en el mismo hilo del sistema operativo. Las tareas de E/S y los temporizadores se producen implícitamente mientras se espera que ocurra el evento. Llamar a `yield()` explícitamente permite planificar otras tareas.

Por lo tanto, una tarea que ejecuta un `ccall` evita efectivamente que el planificador Julia ejecute otras tareas hasta que la llamada regrese. Esto es cierto para todas las llamadas a bibliotecas externas. Las excepciones son llamadas al código C personalizado que devuelve la llamada a Julia (que luego puede ceder) o al código C que llama a `jl_yield()` (C equivalente a `yield()`).

Note that while Julia code runs on a single thread (by default), libraries used by Julia may launch their own internal threads. For example, the BLAS library may start as many threads as there are cores on a machine.

La macro `@threadcall` trata los escenarios donde no queremos que un `ccall` bloquee el ciclo principal de eventos de Julia. Planifica una función C para su ejecución en un hilo separado. Para esto, se usa un pool de hilos con un tamaño

predeterminado de 4. El tamaño del pool de hilos se controla mediante la variable de entorno `UV_THREADPOOL_SIZE`. Mientras espera un hilo libre, y durante la ejecución de la función una vez que un hilo está disponible, la tarea solicitante (en el ciclo de eventos principal de Julia) cede a otras tareas. Tenga en cuenta que `@threadcall` no regresa hasta que se completa la ejecución. Desde el punto de vista del usuario, es por lo tanto una llamada de bloqueo como otras API de Julia.

Es muy importante que la función llamada no vuelva a llamar a Julia.

`@threadcall` puede ser eliminada / cambiada en futuras versiones de Julia.

[1] In this context, MPI refers to the MPI-1 standard. Beginning with MPI-2, the MPI standards committee introduced a new set of communication mechanisms, collectively referred to as Remote Memory Access (RMA). The motivation for adding RMA to the MPI standard was to facilitate one-sided communication patterns. For additional information on the latest MPI standard, see http://mpi-forum.org/docs.

Chapter 27

Date and DateTime

The `Dates` module provides two types for working with dates: `Date` and `DateTime`, representing day and millisecond precision, respectively; both are subtypes of the abstract `TimeType`. The motivation for distinct types is simple: some operations are much simpler, both in terms of code and mental reasoning, when the complexities of greater precision don't have to be dealt with. For example, since the `Date` type only resolves to the precision of a single date (i.e. no hours, minutes, or seconds), normal considerations for time zones, daylight savings/summer time, and leap seconds are unnecessary and avoided.

Both `Date` and `DateTime` are basically immutable `Int64` wrappers. The single `instant` field of either type is actually a `UTInstant{P}` type, which represents a continuously increasing machine timeline based on the UT second [1]. The `DateTime` type is not aware of time zones (*naive*, in Python parlance), analogous to a *LocalDateTime* in Java 8. Additional time zone functionality can be added through the TimeZones.jl package, which compiles the IANA time zone database. Both `Date` and `DateTime` are based on the ISO 8601 standard, which follows the proleptic Gregorian calendar. One note is that the ISO 8601 standard is particular about BC/BCE dates. In general, the last day of the BC/BCE era, 1-12-31 BC/BCE, was followed by 1-1-1 AD/CE, thus no year zero exists. The ISO standard, however, states that 1 BC/BCE is year zero, so `0000-12-31` is the day before `0001-01-01`, and year `-0001` (yes, negative one for the year) is 2 BC/BCE, year `-0002` is 3 BC/BCE, etc.

27.1 Constructors

`Date` and `DateTime` types can be constructed by integer or `Period` types, by parsing, or through adjusters (more on those later):

```
julia> DateTime(2013)
2013-01-01T00:00:00

julia> DateTime(2013,7)
2013-07-01T00:00:00

julia> DateTime(2013,7,1)
2013-07-01T00:00:00
```

[1] The notion of the UT second is actually quite fundamental. There are basically two different notions of time generally accepted, one based on the physical rotation of the earth (one full rotation = 1 day), the other based on the SI second (a fixed, constant value). These are radically different! Think about it, a "UT second", as defined relative to the rotation of the earth, may have a different absolute length depending on the day! Anyway, the fact that `Date` and `DateTime` are based on UT seconds is a simplifying, yet honest assumption so that things like leap seconds and all their complexity can be avoided. This basis of time is formally called UT or UT1. Basing types on the UT second basically means that every minute has 60 seconds and every day has 24 hours and leads to more natural calculations when working with calendar dates.

```
julia> DateTime(2013,7,1,12)
2013-07-01T12:00:00

julia> DateTime(2013,7,1,12,30)
2013-07-01T12:30:00

julia> DateTime(2013,7,1,12,30,59)
2013-07-01T12:30:59

julia> DateTime(2013,7,1,12,30,59,1)
2013-07-01T12:30:59.001

julia> Date(2013)
2013-01-01

julia> Date(2013,7)
2013-07-01

julia> Date(2013,7,1)
2013-07-01

julia> Date(Dates.Year(2013),Dates.Month(7),Dates.Day(1))
2013-07-01

julia> Date(Dates.Month(7),Dates.Year(2013))
2013-07-01
```

`Date` or `DateTime` parsing is accomplished by the use of format strings. Format strings work by the notion of defining *delimited* or *fixed-width* "slots" that contain a period to parse and passing the text to parse and format string to a `Date` or `DateTime` constructor, of the form `Date("2015-01-01","y-m-d")` or `DateTime("20150101","yyyymmdd")`.

Delimited slots are marked by specifying the delimiter the parser should expect between two subsequent periods; so `"y-m-d"` lets the parser know that between the first and second slots in a date string like `"2014-07-16"`, it should find the `-` character. The `y`, `m`, and `d` characters let the parser know which periods to parse in each slot.

Fixed-width slots are specified by repeating the period character the number of times corresponding to the width with no delimiter between characters. So `"yyyymmdd"` would correspond to a date string like `"20140716"`. The parser distinguishes a fixed-width slot by the absence of a delimiter, noting the transition `"yyyymm"` from one period character to the next.

Support for text-form month parsing is also supported through the `u` and `U` characters, for abbreviated and full-length month names, respectively. By default, only English month names are supported, so `u` corresponds to "Jan", "Feb", "Mar", etc. And `U` corresponds to "January", "February", "March", etc. Similar to other name=>value mapping functions `dayname()` and `monthname()`, custom locales can be loaded by passing in the `locale=>Dict{String,Int}` mapping to the `MONTHTOVALUEABBR` and `MONTHTOVALUE` dicts for abbreviated and full-name month names, respectively.

One note on parsing performance: using the `Date(date_string,format_string)` function is fine if only called a few times. If there are many similarly formatted date strings to parse however, it is much more efficient to first create a `Dates.DateFormat`, and pass it instead of a raw format string.

```
julia> df = DateFormat("y-m-d");

julia> dt = Date("2015-01-01",df)
2015-01-01
```

```
julia> dt2 = Date("2015-01-02",df)
2015-01-02
```

You can also use the `dateformat""` string macro. This macro creates the `DateFormat` object once when the macro is expanded and uses the same `DateFormat` object even if a code snippet is run multiple times.

```
julia> for i = 1:10^5

           Date("2015-01-01", dateformat"y-m-d")

       end
```

A full suite of parsing and formatting tests and examples is available in `tests/dates/io.jl`.

27.2 Durations/Comparisons

Finding the length of time between two `Date` or `DateTime` is straightforward given their underlying representation as `UTInstant{Day}` and `UTInstant{Millisecond}`, respectively. The difference between `Date` is returned in the number of `Day`, and `DateTime` in the number of `Millisecond`. Similarly, comparing `TimeType` is a simple matter of comparing the underlying machine instants (which in turn compares the internal `Int64` values).

```
julia> dt = Date(2012,2,29)
2012-02-29

julia> dt2 = Date(2000,2,1)
2000-02-01

julia> dump(dt)
Date
  instant: Base.Dates.UTInstant{Base.Dates.Day}
    periods: Base.Dates.Day


      value: Int64 734562
julia> dump(dt2)
Date
  instant: Base.Dates.UTInstant{Base.Dates.Day}
    periods: Base.Dates.Day


      value: Int64 730151
julia> dt > dt2
true

julia> dt != dt2
true

julia> dt + dt2
ERROR: MethodError: no method matching +(::Date, ::Date)
[...]

julia> dt * dt2
ERROR: MethodError: no method matching *(::Date, ::Date)
```

```
[...]

julia> dt / dt2
ERROR: MethodError: no method matching /(::Date, ::Date)
[...]

julia> dt - dt2
4411 days

julia> dt2 - dt
-4411 days

julia> dt = DateTime(2012,2,29)
2012-02-29T00:00:00

julia> dt2 = DateTime(2000,2,1)
2000-02-01T00:00:00

julia> dt - dt2
381110400000 milliseconds
```

27.3 Accessor Functions

Because the `Date` and `DateTime` types are stored as single `Int64` values, date parts or fields can be retrieved through accessor functions. The lowercase accessors return the field as an integer:

```
julia> t = Date(2014, 1, 31)
2014-01-31

julia> Dates.year(t)
2014

julia> Dates.month(t)
1

julia> Dates.week(t)
5

julia> Dates.day(t)
31
```

While propercase return the same value in the corresponding `Period` type:

```
julia> Dates.Year(t)
2014 years

julia> Dates.Day(t)
31 days
```

Compound methods are provided, as they provide a measure of efficiency if multiple fields are needed at the same time:

```
julia> Dates.yearmonth(t)
(2014, 1)

julia> Dates.monthday(t)
(1, 31)
```

```
julia> Dates.yearmonthday(t)
(2014, 1, 31)
```

One may also access the underlying `UTInstant` or integer value:

```
julia> dump(t)
Date
  instant: Base.Dates.UTInstant{Base.Dates.Day}
    periods: Base.Dates.Day
      value: Int64 735264

julia> t.instant
Base.Dates.UTInstant{Base.Dates.Day}(735264 days)

julia> Dates.value(t)
735264
```

27.4 Query Functions

Query functions provide calendrical information about a `TimeType`. They include information about the day of the week:

```
julia> t = Date(2014, 1, 31)
2014-01-31

julia> Dates.dayofweek(t)
5

julia> Dates.dayname(t)
"Friday"

julia> Dates.dayofweekofmonth(t) # 5th Friday of January
5
```

Month of the year:

```
julia> Dates.monthname(t)
"January"

julia> Dates.daysinmonth(t)
31
```

As well as information about the `TimeType`'s year and quarter:

```
julia> Dates.isleapyear(t)
false

julia> Dates.dayofyear(t)
31

julia> Dates.quarterofyear(t)
1

julia> Dates.dayofquarter(t)
31
```

The `dayname()` and `monthname()` methods can also take an optional `locale` keyword that can be used to return the name of the day or month of the year for other languages/locales. There are also versions of these functions returning the abbreviated names, namely `dayabbr()` and `monthabbr()`. First the mapping is loaded into the `LOCALES` variable:

```
julia> french_months = ["janvier", "février", "mars", "avril", "mai", "juin",
                        "juillet", "août", "septembre", "octobre", "novembre", "décembre"];

julia> french_monts_abbrev = ["janv","févr","mars","avril","mai","juin",
                              "juil","août","sept","oct","nov","déc"];

julia> french_days = ["lundi","mardi","mercredi","jeudi","vendredi","samedi","dimanche"];

julia> Dates.LOCALES["french"] = Dates.DateLocale(french_months, french_monts_abbrev, french_days
    , [""]);
```

The above mentioned functions can then be used to perform the queries:

```
julia> Dates.dayname(t;locale="french")
"vendredi"

julia> Dates.monthname(t;locale="french")
"janvier"

julia> Dates.monthabbr(t;locale="french")
"janv"
```

Since the abbreviated versions of the days are not loaded, trying to use the function `dayabbr()` will error.

```
julia> Dates.dayabbr(t;locale="french")
ERROR: BoundsError: attempt to access 1-element Array{String,1} at index [5]
Stacktrace:
 [1] #dayabbr#6(::String, ::Function, ::Int64) at ./dates/query.jl:114
 [2] (::Base.Dates.#kw##dayabbr)(::Array{Any,1}, ::Base.Dates.#dayabbr, ::Int64) at ./<missing>:0
     (repeats 2 times)
```

27.5 TimeType-Period Arithmetic

It's good practice when using any language/date framework to be familiar with how date-period arithmetic is handled as there are some tricky issues to deal with (though much less so for day-precision types).

The `Dates` module approach tries to follow the simple principle of trying to change as little as possible when doing `Period` arithmetic. This approach is also often known as *calendrical* arithmetic or what you would probably guess if someone were to ask you the same calculation in a conversation. Why all the fuss about this? Let's take a classic example: add 1 month to January 31st, 2014. What's the answer? Javascript will say March 3 (assumes 31 days). PHP says March 2 (assumes 30 days). The fact is, there is no right answer. In the `Dates` module, it gives the result of February 28th. How does it figure that out? I like to think of the classic 7-7-7 gambling game in casinos.

Now just imagine that instead of 7-7-7, the slots are Year-Month-Day, or in our example, 2014-01-31. When you ask to add 1 month to this date, the month slot is incremented, so now we have 2014-02-31. Then the day number is checked if it is greater than the last valid day of the new month; if it is (as in the case above), the day number is adjusted down to the last valid day (28). What are the ramifications with this approach? Go ahead and add another month to our date, `2014-02-28 + Month(1) == 2014-03-28`. What? Were you expecting the last day of March? Nope, sorry, remember the 7-7-7 slots. As few slots as possible are going to change, so we first increment the month slot by 1, 2014-03-28, and boom, we're done because that's a valid date. On the other hand, if we were to add 2 months to our original date, 2014-01-31, then we end up with 2014-03-31, as expected. The other ramification of

this approach is a loss in associativity when a specific ordering is forced (i.e. adding things in different orders results in different outcomes). For example:

```
julia> (Date(2014,1,29)+Dates.Day(1)) + Dates.Month(1)
2014-02-28

julia> (Date(2014,1,29)+Dates.Month(1)) + Dates.Day(1)
2014-03-01
```

What's going on there? In the first line, we're adding 1 day to January 29th, which results in 2014-01-30; then we add 1 month, so we get 2014-02-30, which then adjusts down to 2014-02-28. In the second example, we add 1 month *first*, where we get 2014-02-29, which adjusts down to 2014-02-28, and *then* add 1 day, which results in 2014-03-01. One design principle that helps in this case is that, in the presence of multiple Periods, the operations will be ordered by the Periods' *types*, not their value or positional order; this means `Year` will always be added first, then `Month`, then `Week`, etc. Hence the following *does* result in associativity and Just Works:

```
julia> Date(2014,1,29) + Dates.Day(1) + Dates.Month(1)
2014-03-01

julia> Date(2014,1,29) + Dates.Month(1) + Dates.Day(1)
2014-03-01
```

Tricky? Perhaps. What is an innocent `Dates` user to do? The bottom line is to be aware that explicitly forcing a certain associativity, when dealing with months, may lead to some unexpected results, but otherwise, everything should work as expected. Thankfully, that's pretty much the extent of the odd cases in date-period arithmetic when dealing with time in UT (avoiding the "joys" of dealing with daylight savings, leap seconds, etc.).

As a bonus, all period arithmetic objects work directly with ranges:

```
julia> dr = Date(2014,1,29):Date(2014,2,3)
2014-01-29:1 day:2014-02-03

julia> collect(dr)
6-element Array{Date,1}:
 2014-01-29
 2014-01-30
 2014-01-31
 2014-02-01
 2014-02-02
 2014-02-03

julia> dr = Date(2014,1,29):Dates.Month(1):Date(2014,07,29)
2014-01-29:1 month:2014-07-29

julia> collect(dr)
7-element Array{Date,1}:
 2014-01-29
 2014-02-28
 2014-03-29
 2014-04-29
 2014-05-29
 2014-06-29
 2014-07-29
```

27.6 Adjuster Functions

As convenient as date-period arithmetics are, often the kinds of calculations needed on dates take on a *calendrical* or *temporal* nature rather than a fixed number of periods. Holidays are a perfect example; most follow rules such as "Memorial Day = Last Monday of May", or "Thanksgiving = 4th Thursday of November". These kinds of temporal expressions deal with rules relative to the calendar, like first or last of the month, next Tuesday, or the first and third Wednesdays, etc.

The `Dates` module provides the *adjuster* API through several convenient methods that aid in simply and succinctly expressing temporal rules. The first group of adjuster methods deal with the first and last of weeks, months, quarters, and years. They each take a single `TimeType` as input and return or *adjust to* the first or last of the desired period relative to the input.

```
julia> Dates.firstdayofweek(Date(2014,7,16)) # Adjusts the input to the Monday of the input's week
2014-07-14

julia> Dates.lastdayofmonth(Date(2014,7,16)) # Adjusts to the last day of the input's month
2014-07-31

julia> Dates.lastdayofquarter(Date(2014,7,16)) # Adjusts to the last day of the input's quarter
2014-09-30
```

The next two higher-order methods, `tonext()`, and `toprev()`, generalize working with temporal expressions by taking a `DateFunction` as first argument, along with a starting `TimeType`. A `DateFunction` is just a function, usually anonymous, that takes a single `TimeType` as input and returns a `Bool`, `true` indicating a satisfied adjustment criterion. For example:

```
julia> istuesday = x->Dates.dayofweek(x) == Dates.Tuesday # Returns true if the day of the week of
↪  x is Tuesday
(::#1) (generic function with 1 method)

julia> Dates.tonext(istuesday, Date(2014,7,13)) # 2014-07-13 is a Sunday
2014-07-15

julia> Dates.tonext(Date(2014,7,13), Dates.Tuesday) # Convenience method provided for day of the
↪  week adjustments
2014-07-15
```

This is useful with the do-block syntax for more complex temporal expressions:

```
julia> Dates.tonext(Date(2014,7,13)) do x

           # Return true on the 4th Thursday of November (Thanksgiving)

           Dates.dayofweek(x) == Dates.Thursday &&

           Dates.dayofweekofmonth(x) == 4 &&

           Dates.month(x) == Dates.November

       end
2014-11-27
```

The `Base.filter()` method can be used to obtain all valid dates/moments in a specified range:

```
# Pittsburgh street cleaning; Every 2nd Tuesday from April to November
# Date range from January 1st, 2014 to January 1st, 2015
julia> dr = Dates.Date(2014):Dates.Date(2015);

julia> filter(dr) do x

           Dates.dayofweek(x) == Dates.Tue &&

           Dates.April <= Dates.month(x) <= Dates.Nov &&

           Dates.dayofweekofmonth(x) == 2

       end
8-element Array{Date,1}:
 2014-04-08
 2014-05-13
 2014-06-10
 2014-07-08
 2014-08-12
 2014-09-09
 2014-10-14
 2014-11-11
```

Additional examples and tests are available in `test/dates/adjusters.jl`.

27.7 Period Types

Periods are a human view of discrete, sometimes irregular durations of time. Consider 1 month; it could represent, in days, a value of 28, 29, 30, or 31 depending on the year and month context. Or a year could represent 365 or 366 days in the case of a leap year. `Period` types are simple `Int64` wrappers and are constructed by wrapping any `Int64` convertible type, i.e. `Year(1)` or `Month(3.0)`. Arithmetic between `Period` of the same type behave like integers, and limited `Period-Real` arithmetic is available.

```
julia> y1 = Dates.Year(1)
1 year

julia> y2 = Dates.Year(2)
2 years

julia> y3 = Dates.Year(10)
10 years

julia> y1 + y2
3 years

julia> div(y3,y2)
5

julia> y3 - y2
8 years

julia> y3 % y2
0 years
```

```
julia> div(y3,3) # mirrors integer division
3 years
```

27.8 Rounding

`Date` and `DateTime` values can be rounded to a specified resolution (e.g., 1 month or 15 minutes) with `floor()`, `ceil()`, or `round()`:

```
julia> floor(Date(1985, 8, 16), Dates.Month)
1985-08-01

julia> ceil(DateTime(2013, 2, 13, 0, 31, 20), Dates.Minute(15))
2013-02-13T00:45:00

julia> round(DateTime(2016, 8, 6, 20, 15), Dates.Day)
2016-08-07T00:00:00
```

Unlike the numeric `round()` method, which breaks ties toward the even number by default, the `TimeTyperound()` method uses the `RoundNearestTiesUp` rounding mode. (It's difficult to guess what breaking ties to nearest "even" `TimeType` would entail.) Further details on the available `RoundingMode` s can be found in the API reference.

Rounding should generally behave as expected, but there are a few cases in which the expected behaviour is not obvious.

Rounding Epoch

In many cases, the resolution specified for rounding (e.g., `Dates.Second(30)`) divides evenly into the next largest period (in this case, `Dates.Minute(1)`). But rounding behaviour in cases in which this is not true may lead to confusion. What is the expected result of rounding a `DateTime` to the nearest 10 hours?

```
julia> round(DateTime(2016, 7, 17, 11, 55), Dates.Hour(10))
2016-07-17T12:00:00
```

That may seem confusing, given that the hour (12) is not divisible by 10. The reason that `2016-07-17T12:00:00` was chosen is that it is 17,676,660 hours after `0000-01-01T00:00:00`, and 17,676,660 is divisible by 10.

As Julia `Date` and `DateTime` values are represented according to the ISO 8601 standard, `0000-01-01T00:00:00` was chosen as base (or "rounding epoch") from which to begin the count of days (and milliseconds) used in rounding calculations. (Note that this differs slightly from Julia's internal representation of `Date` s using Rata Die notation; but since the ISO 8601 standard is most visible to the end user, `0000-01-01T00:00:00` was chosen as the rounding epoch instead of the `0000-12-31T00:00:00` used internally to minimize confusion.)

The only exception to the use of `0000-01-01T00:00:00` as the rounding epoch is when rounding to weeks. Rounding to the nearest week will always return a Monday (the first day of the week as specified by ISO 8601). For this reason, we use `0000-01-03T00:00:00` (the first day of the first week of year 0000, as defined by ISO 8601) as the base when rounding to a number of weeks.

Here is a related case in which the expected behaviour is not necessarily obvious: What happens when we round to the nearest `P(2)`, where `P` is a `Period` type? In some cases (specifically, when `P <: Dates.TimePeriod`) the answer is clear:

```
julia> round(DateTime(2016, 7, 17, 8, 55, 30), Dates.Hour(2))
2016-07-17T08:00:00
```

```
julia> round(DateTime(2016, 7, 17, 8, 55, 30), Dates.Minute(2))
2016-07-17T08:56:00
```

This seems obvious, because two of each of these periods still divides evenly into the next larger order period. But in the case of two months (which still divides evenly into one year), the answer may be surprising:

```
julia> round(DateTime(2016, 7, 17, 8, 55, 30), Dates.Month(2))
2016-07-01T00:00:00
```

Why round to the first day in July, even though it is month 7 (an odd number)? The key is that months are 1-indexed (the first month is assigned 1), unlike hours, minutes, seconds, and milliseconds (the first of which are assigned 0).

This means that rounding a `DateTime` to an even multiple of seconds, minutes, hours, or years (because the ISO 8601 specification includes a year zero) will result in a `DateTime` with an even value in that field, while rounding a `DateTime` to an even multiple of months will result in the months field having an odd value. Because both months and years may contain an irregular number of days, whether rounding to an even number of days will result in an even value in the days field is uncertain.

See the API reference for additional information on methods exported from the `Dates` module.

Chapter 28

Interactuando con Julia

Julia viene con un REPL (read-eval-print loop) interactivo de línea de comando integrado en el ejecutable `julia`. Además de permitir una evaluación rápida y fácil de las declaraciones de Julia, tiene un historial de búsqueda, finalización de pestañas, muchas combinaciones útiles de teclas, ayuda dedicada y modos de shell. El REPL se puede iniciar simplemente llamando a `julia` sin argumentos o haciendo doble clic en el ejecutable:

```
$ julia
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: https://docs.julialang.org
   _ _   _| |_  __ _   |  Type "?help" for help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.6.0-dev.2493 (2017-01-31 18:53 UTC)
 _/ |\__'_|_|_|\__'_|  |  Commit c99e12c* (0 days old master)
|__/                   |  x86_64-linux-gnu

julia>
```

Para salir de la sesión interactiva, escriba `^D` - la tecla de control junto con la tecla d en una línea en blanco - o escriba `quit()` seguido de la tecla return o enter. El REPL te saluda con un banner y un prompt `julia>`.

28.1 Los distintos modos de prompt

El modo Juliano

El REPL tiene cuatro modos principales de operación. El primero y más común es el prompt Juliano. Es el modo de operación predeterminado; cada nueva línea inicialmente comienza con `julia>`. Es aquí donde puede ingresar las expresiones de Julia. Pulsando *return* o *enter* después de haber ingresado una expresión completa, se evaluará la entrada y se mostrará el resultado de la última expresión.

```
julia> string(1 + 2)
"3"
```

Hay varias funciones útiles únicas para el trabajo interactivo. Además de mostrar el resultado, el REPL también vincula el resultado a la variable `ans`. Un punto y coma final en la línea se puede utilizar como un indicador para suprimir mostrar el resultado.

```
julia> string(3 * 4);
```

```
julia> ans
"12"
```

En el modo Julia, el REPL es compatible con algo llamado *pegado rápido*. Esto se activa al pegar texto que comienza con `julia>` en REPL. En ese caso, solo las expresiones que comienzan con `julia>` se analizan, otras se eliminan. Esto hace que sea posible pegar un trozo de código que ha sido copiado de una sesión REPL sin tener que eliminar los prompts y las salidas. Esta característica está habilitada de forma predeterminada, pero se puede desactivar o habilitar a voluntad con `Base.REPL.enable_promptpaste(:: Bool)`. Si está habilitado, puedes probar pegando el bloque de código arriba de este párrafo directamente en REPL. Esta función no funciona en el símbolo del sistema estándar de Windows debido a su limitación para detectar cuándo se produce un pegado.

Modo Ayuda

Cuando el cursor está al principio de la línea, el aviso se puede cambiar a un modo de ayuda escribiendo `?`. Julia intentará imprimir ayuda o documentación para todo lo ingresado en modo de ayuda:

```
julia> ? # upon typing ?, the prompt changes (in place) to: help?>

help?> string
search: string String stringmime Cstring Cwstring RevString readstring randstring bytestring
↪  SubString

  string(xs...)

  Create a string from any values using the print function.
```

Las macros, los tipos y las variables también se pueden consultar:

```
help?> @time
  @time

  A macro to execute an expression, printing the time it took to execute, the number of
    allocations,
  and the total number of bytes its execution caused to be allocated, before returning the value
    of the
  expression.

  See also @timev, @timed, @elapsed, and @allocated.

help?> AbstractString
search: AbstractString AbstractSparseMatrix AbstractSparseVector AbstractSet

  No documentation found.

  Summary:

  abstract AbstractString <: Any

  Subtypes:

  Base.Test.GenericString
  DirectIndexString
  String
```

El modo de ayuda puede salir presionando la tecla de retroceso al comienzo de la línea.

Modo Shell

Del mismo modo que el modo de ayuda es útil para acceder rápidamente a la documentación, otra tarea común es utilizar el shell del sistema para ejecutar los comandos del sistema. Así como ? Ingresó al modo de ayuda cuando está al principio de la línea, un punto y coma (;) ingresará al modo shell. Y puede salir presionando el retroceso al comienzo de la línea.

```
julia> ; # upon typing ;, the prompt changes (in place) to: shell>

shell> echo hello
hello
```

Modos de búsqueda

En todos los modos anteriores, las líneas ejecutadas se guardan en un archivo de historial, que se puede buscar. Para iniciar una búsqueda incremental a través del historial anterior, escriba `^R` - la tecla de control junto con la tecla `r`. El aviso cambiará a `(reverse-i-search)` ':, y al escribir, la consulta de búsqueda aparecerá en las comillas. El resultado más reciente que coincida con la consulta se actualizará dinámicamente a la derecha de los dos puntos a medida que se escriba más. Para encontrar un resultado anterior usando la misma consulta, simplemente escriba `^R` de nuevo.

Del mismo modo que `^ R` es una búsqueda inversa,`^ S` es una búsqueda directa, con el indicador `(i-search)` ':. Los dos pueden usarse conjuntamente para avanzar entre los resultados de coincidencia anterior o siguiente, respectivamente.

28.2 Asociaciones de teclas

El REPL de Julia hace un gran uso de las asociaciones de teclas. Varias combinaciones de teclas de control ya se han introducido anteriormente (`^D` para salir,`^R` y `^S` para buscar), pero hay muchas más. Además de la tecla de control, también hay enlaces de meta-clave. Estos varían según la plataforma, pero la mayoría de los terminales utilizan de forma predeterminada alt- u opción-, mantenida presionada con una tecla para enviar la meta-clave (o puede configurarse para hacerlo).

Personalizando asociaciones de teclas

Las combinaciones de teclas en el REPL de Julia pueden personalizarse completamente según las preferencias de un usuario pasando un diccionario a `REPL.setup_interface()`. Las claves de este diccionario pueden ser caracteres o cadenas. La tecla `'*'` se refiere a la acción predeterminada. Las asociaciones de control y carácter `x` se indican con`"^x"`. Meta plus `x` se puede escribir`"\\ Mx"`. Los valores del mapa de teclas personalizado deben ser $nothing$ (lo que indica que la entrada debe ignorarse) o las funciones que aceptan la firma `(PromptState, AbstractREPL, Char)`. La función `REPL.setup_interface()` debe invocarse antes de que se inicialice REPL, registrando la operación con `atreplinit()`. Por ejemplo, para enlazar las teclas de flecha hacia arriba y hacia abajo para moverse a través del historial sin búsqueda de prefijos, uno podría poner el siguiente código en `.juliarc.jl`:

```
import Base: LineEdit, REPL

const mykeys = Dict{Any,Any}(
    # Up Arrow
    "\e[A" => (s,o...)->(LineEdit.edit_move_up(s) || LineEdit.history_prev(s,
    ↪  LineEdit.mode(s).hist)),
    # Down Arrow
    "\e[B" => (s,o...)->(LineEdit.edit_move_up(s) || LineEdit.history_next(s,
    ↪  LineEdit.mode(s).hist))
```

Keybinding	Description
Program control	
^D	Exit (when buffer is empty)
^C	Interrupt or cancel
^L	Clear console screen
Return/Enter, ^J	New line, executing if it is complete
meta-Return/Enter	Insert new line without executing it
? or ;	Enter help or shell mode (when at start of a line)
^R, ^S	Incremental history search, described above
Cursor movement	
Right arrow, ^F	Move right one character
Left arrow, ^B	Move left one character
Home, ^A	Move to beginning of line
End, ^E	Move to end of line
^P	Change to the previous or next history entry
^N	Change to the next history entry
Up arrow	Move up one line (or to the previous history entry)
Down arrow	Move down one line (or to the next history entry)
Page-up	Change to the previous history entry that matches the text before the cursor
Page-down	Change to the next history entry that matches the text before the cursor
meta-F	Move right one word
meta-B	Move left one word
Editing	
Backspace, ^H	Delete the previous character
Delete, ^D	Forward delete one character (when buffer has text)
meta-Backspace	Delete the previous word
meta-D	Forward delete the next word
^W	Delete previous text up to the nearest whitespace
^K	"Kill" to end of line, placing the text in a buffer
^Y	"Yank" insert the text from the kill buffer
^T	Transpose the characters about the cursor
^Q	Write a number in REPL and press ^Q to open editor at corresponding stackframe

```
)

function customize_keys(repl)
    repl.interface = REPL.setup_interface(repl; extra_repl_keymap = mykeys)
end

atreplinit(customize_keys)
```

Los usuarios deberían consultar `base/LineEdit.jl` para descubrir las acciones disponibles sobre la entrada clave.

28.3 Uso de Tab para completar expresiones

Tanto en los moddos Juliano como de ayuda del REPL, uno puede entrar los primeros caracteres de una función o tipo y luego pulsar la tecla del tabulador para obtener una lista de posibles coincidencias:

```
julia> stri[TAB]
stride     strides     string      stringmime  strip
```

```
julia> Stri[TAB]
StridedArray    StridedMatrix    StridedVecOrMat  StridedVector    String
```

La tecla tabulador puede también usarse para sustituir los símbolos matemáticos de LaTeX con sus equivalentes Unicode, y obtener también una lista de coincidencias LaTeX:

```
julia> \pi[TAB]
julia> π
π = 3.1415926535897...

julia> e\_1[TAB] = [1,0]
julia> e = [1,0]
2-element Array{Int64,1}:
 1
 0

julia> e\^1[TAB] = [1 0]
julia> e¹ = [1 0]
1×2 Array{Int64,2}:
 1  0

julia> \sqrt[TAB]2     # √ is equivalent to the sqrt() function
julia> √2
1.4142135623730951

julia> \hbar[TAB](h) = h / 2\pi[TAB]
julia> ħ(h) = h / 2π
ħ (generic function with 1 method)

julia> \h[TAB]
\hat              \hermitconjmatrix  \hkswarow          \hrectangle
\hatapprox        \hexagon           \hookleftarrow     \hrectangleblack
\hbar             \hexagonblack      \hookrightarrow    \hslash
\heartsuit        \hksearow          \house             \hspace

julia> α="\alpha[TAB]"   # LaTeX completion also works in strings
julia> α="α"
```

En la sección Entrada Unicode del manual puede encontrarse una lista completa de finalizaciones usando tabulador.

En modo shell también puede usarse el tabulador para completar cadenas relativas a caminos de fichero:

```
julia> path="/[TAB]"
.dockerenv  .juliabox/  boot/       etc/        lib/        media/      opt/        root/
↪  sbin/       sys/        usr/
.dockerinit bin/        dev/        home/       lib64/      mnt/        proc/       run/
↪  srv/        tmp/        var/
shell> /[TAB]
.dockerenv  .juliabox/  boot/       etc/        lib/        media/      opt/        root/
↪  sbin/       sys/        usr/
.dockerinit bin/        dev/        home/       lib64/      mnt/        proc/       run/
↪  srv/        tmp/        var/
```

Esta funcionalidad puede ayudar con la investigación de los métodos disponibles que coinciden con los argumentos de entrada:

```
julia> max([TAB] # All methods are displayed, not shown here due to size of the list

julia> max([1, 2], [TAB] # All methods where `Vector{Int}` matches as first argument
max(x, y) in Base at operators.jl:215
max(a, b, c, xs...) in Base at operators.jl:281

julia> max([1, 2], max(1, 2), [TAB] # All methods matching the arguments.
max(x, y) in Base at operators.jl:215
max(a, b, c, xs...) in Base at operators.jl:281
```

Las palabras clave (*keywords*) son también mostradas en los métodos sugeridos, ver la segunda línea después de ; donde `limit` y `keep` son argumentos palabras clave:

```
julia> split("1 1 1", [TAB]
split(str::AbstractString) in Base at strings/util.jl:278
split{T<:AbstractString}(str::T, splitter; limit, keep) in Base at strings/util.jl:254
```

La acción de completar los métodos usa inferencia de tipos y puede por tanto ver si los argumentos coinciden incluso si los argumentos son la salida de funcionees La función necesita se estable de tipos para que la acción de completar sea capaz de borrar los métodos no coincidentes.

La acción de completar puede también ayudar a completar campos:

```
julia> Pkg.a[TAB]
add       available
```

También pueden completarse los campos para la salida de funciones:

```
julia> split("","")[1].[TAB]
endof  offset  string
```

La acción de completar campos para la salida de funciones usa inferencia de tipos, y sólo puede sugerir campos si la función es estable en los tipos.

28.4 Personalizando Colores

Los colores utilizados por Julia y REPL se pueden personalizar también. Para cambiar el color del *prompt* de Julia, puede agregar algo como lo siguiente a su `.juliarc.jl`, que se debe colocar dentro del directorio de inicio:

```
function customize_colors(repl)
    repl.prompt_color = Base.text_colors[:cyan]
end

atreplinit(customize_colors)
```

Las teclas de color disponibles se pueden ver escribiendo `Base.text_colors` en el modo de ayuda de REPL. Además, los números enteros 0 a 255 se pueden usar como teclas de color para terminales con soporte de 256 colores.

También puede cambiar los colores para la ayuda y las instrucciones del shell e ingresar y contestar el texto configurando el campo apropiado de `repl` en la función `customize_colors` arriba (respectivamente, `help_color`, `shell_color`, `input_color`, y `answer_color`). Para los dos últimos, asegúrese de que el campo `envcolors` también esté configurado a false.

También es posible aplicar el formato en negrita mediante el uso de `Base.text_colors[:bold]` como color. Por ejemplo, para imprimir las respuestas en letra negrita, se puede usar lo siguiente como `.juliarc.jl`:

```
function customize_colors(repl)
    repl.envcolors = false
    repl.answer_color = Base.text_colors[:bold]
end

atreplinit(customize_colors)
```

También puede personalizarse el color utilizado para presentar mensajes de advertencia e información estableciendo las variables de entorno apropiadas. Por ejemplo, para generar mensajes de error, de advertencia y de información respectivamente en magenta, amarillo y cian, puede agregar lo siguiente a su archivo `.juliarc.jl`:

```
ENV["JULIA_ERROR_COLOR"] = :magenta
ENV["JULIA_WARN_COLOR"] = :yellow
ENV["JULIA_INFO_COLOR"] = :cyan
```

Chapter 29

Ejecutando programas externos

Julia toma prestada la notación de tilde inversa para ejecutar mandatos de la shell y programas en Perl y en Ruby. Sin embargo, en Julia, escribir

```
julia> `echo hello`
`echo hello`
```

difiere en algunos aspectos del comportamiento en varios shells, Perl y Ruby:

- En lugar de ejecutar el mandato inmediatamente, las tildes invertidas crean un objeto `Cmd` para representar el mandato. Uno puede usar este objeto para conectar los mandatos a otros vía tuberías (*pipes*), ejecutarlo y leer o escribir en él.
- Cuando el mandato se está ejecutando, Julia no captura su salida a menos que uno lo organice específicamente. En lugar de ello, la salida del mandato va por defecto a `STDOUT` como si se estuviera realizando una llamada al sistema con la `libc`.
- El mandato nunca es ejecutado con un shell. En su lugar, Julia analiza directamente la sintaxis del mandato, interpola variables adecuadametne y dividiendo en palabras como el shell lo haría, respetando su sintaxis. El mandato se ejecuta como un proceso hijo inmediato de Julia, usando las llamadas `fork` y `exec`.

He aquí un ejemplo sencillo de ejecución de un programa externo:

```
julia> mycommand = `echo hello`
`echo hello`

julia> typeof(mycommand)
Cmd

julia> run(mycommand)
hello
```

El mensaje `hello` es la salida de este mandato `echo`, enviado a `STDOUT`. El mensaje `hello` es la salida de este mandato `echo`, enviado a `STDOUT`. El método ejecutado devuelve en si mismo `nothing`, y lanza una `ErrorException` si el mandato externo falla en ejecutarse con éxito.

Si se desea leer la salida de un mandato externo puede usarse `readstring()`:

```
julia> a = readstring(`echo hello`)
"hello\n"

julia> chomp(a) == "hello"
true
```

Más generalmente, puedes usar `open()` para leer desde o escribir hacia un mandato externo.

```
julia> open(`less`, "w", STDOUT) do io

           for i = 1:3

               println(io, i)

           end

       end
1
2
3
```

29.1 Interpolación

Supongamos que uno quiere algo un poco más complicado y usa el nombre de un fichero en la variable `file` como argumento a un mandato. Se puede usar $ para interpolar tal como lo haríamos con un literal cadena (ver la sección Strings):

```
julia> file = "/etc/passwd"
"/etc/passwd"

julia> `sort $file`
`sort /etc/passwd`
```

Un error común es que cuando se ejecutan programas externos a través de un shell es que si el nombre del fichero contiene caracteres que son especiales para el shell, ellos pueden causar un comportamiento indeseaddo. Por ejemplo, en lugar de `/etc/passwd` se desea ordenar los contenidos del fichero `/volumes/External HD/data.csv`. Intentémoslo:

```
julia> file = "/Volumes/External HD/data.csv"
"/Volumes/External HD/data.csv"

julia> `sort $file`
`sort '/Volumes/External HD/data.csv'`
```

¿Cómo se entrecomilla el nombre de fichero? Julia sabe que `file` va a ser interpolado por un solo argumento, por lo que él entrecomilla la cadena. De hecho, esto no es bastante exacto: el valor de `file` no va a ser interpretado por el shell nunca, por lo que no hay necesidad de entrecomillar. Las comillas se insertan sólo para presentar al usuario. Eso funcionará incluso si uno interpola un valor como parte de una palabra del shell:

```
julia> path = "/Volumes/External HD"
"/Volumes/External HD"
```

```
julia> name = "data"
"data"

julia> ext = "csv"
"csv"

julia> `sort $path/$name.$ext`
`sort '/Volumes/External HD/data.csv'`
```

Como puedes ver, el espacio en la variable `path` es apropiadamente "escapado". Pero, ¿qué pasa si lo que uno desea es interpolar múltiples palabras? En este caso, se utilizará un array (u otro contenedor iterable):

```
julia> files = ["/etc/passwd","/Volumes/External HD/data.csv"]
2-element Array{String,1}:
 "/etc/passwd"
 "/Volumes/External HD/data.csv"

julia> `grep foo $files`
`grep foo /etc/passwd '/Volumes/External HD/data.csv'`
```

Si interpolas un array como parte de una palabra de la shell, Julia emula la generación de argumentos de la shell `{a, b, c}`:

```
julia> names = ["foo","bar","baz"]
3-element Array{String,1}:
 "foo"
 "bar"
 "baz"

julia> `grep xylophone $names.txt`
`grep xylophone foo.txt bar.txt baz.txt`
```

Además, si interpolas múltiples arrays en la misma palabra, se emula el comportamiento de generación del shell haciendo el producto cartesiano:

```
julia> names = ["foo","bar","baz"]
3-element Array{String,1}:
 "foo"
 "bar"
 "baz"

julia> exts = ["aux","log"]
2-element Array{String,1}:
 "aux"
 "log"

julia> `rm -f $names.$exts`
`rm -f foo.aux foo.log bar.aux bar.log baz.aux baz.log`
```

Como pudes interpolar arrays de literales, puedes usar esta funcionalidad generativa sin necesidad de crer objetos array temporales primero:

```
julia> `rm -rf $["foo","bar","baz","qux"].$["aux","log","pdf"]`
`rm -rf foo.aux foo.log foo.pdf bar.aux bar.log bar.pdf baz.aux baz.log baz.pdf qux.aux qux.log
↪  qux.pdf`
```

29.2 Entrecomillado

Inevitablemente, uno quiere escribir mandatos que no sean tan simples, y se vueve necesario usar comillas. He aquí un ejemplo simple de un script Perl de una línea en el prompt del shell:

```
sh$ perl -le '$|=1; for (0..3) { print }'
0
1
2
3
```

La expresión Perl necesita estar entre comillas sencillas por dos razones: para que los espacios no rompan la expresión en múltiples palabras en el shell, y para que el uso de variables de Perl, como `$|` no cause interpolación. En otras instancias, puedes querer usar dobles comillas para que la interpolación SI tenga lugar:

```
sh$ first="A"
sh$ second="B"
sh$ perl -le '$|=1; print for @ARGV' "1: $first" "2: $second"
1: A
2: B
```

En general, la sintaxis de comillas invertidas de Julia está diseñada cuidadosamente para que puedas cortar y pegar comendos del shell, los pongas entre comillas y funcionen: los comportamientos del escape, las comillas y las interpolaciones son los mismos que los del shell. La única diferencia es que la interpolación está integrada y consciente de la noción de Julia de que es un valor de cadena simple, y qué es un contenedor para valores múltiples. Intentemos los dos ejemplos anteriores en Julia:

```
julia> A = `perl -le '$|=1; for (0..3) { print }'`
`perl -le '$|=1; for (0..3) { print }'`

julia> run(A)
0
1
2
3

julia> first = "A"; second = "B";

julia> B = `perl -le 'print for @ARGV' "1: $first" "2: $second"`
`perl -le 'print for @ARGV' '1: A' '2: B'`

julia> run(B)
1: A
2: B
```

Los resultados son idénticos, y el comportamiento de interpolación de Julia imita el shell con algunas mejoras debido a que Julia soporta objetos iterables de primera clase mientras la mayoría de los shells usan división de cadenas mediante espacios para ésto, lo cuál introduce ambigüedades. Cuando intentamos portar mandatos del shell a Julia, intentemos cortar y pegar primero. Como Julia te muestra los mandatos antes de que los ejecutes, puedes examinar fácilmente y de forma segura su interpretación sin hacer ningún daño.

29.3 Tuberías

Los metacaracteres del shell tales como `|`, `&`, and `>`, necesitan ser acotados o escapados dentro de las comillas invertidas de Julia:

```
julia> run(`echo hello '|' sort`)
hello | sort

julia> run(`echo hello \| sort`)
hello | sort
```

Esta expresión invoca el mandato echo con tres palabras como argumentos, "hello", "|" y "sort". El resultados es que se imprime una sola línea "hello | sort". Dentro de las comillas traseras, el símbolo "|" no tiene un significado especial. ¿Cómo entonces, podemos construir una tubería? En lugar de usar el símbolo "|" dentro de la tubería, utilizaremos la función `pipeline()`:

```
julia> run(pipeline(`echo hello`, `sort`))
hello
```

Esto entuba la salida del mandato echo al mandato `sort`. Por supuesto, esto no es terriblemente interesante ya que sólo hay una línea que ordenar, pero podemos cosas mucho más interesantes:

```
julia> run(pipeline(`cut -d: -f3 /etc/passwd`, `sort -n`, `tail -n5`))
210
211
212
213
214
```

Esto imprime los cinco identificadores de usuario mayores dentro de un sistema UNIX. Los mandatos `cut`, `sort` y `tail` son "criados" como hijos inmediatos del proceso `julia` actual, sin que intervenga el proceso shell. Julia en sí mismo hace el trabajo (normalmente hecho por el shell) de inicializar las tuberías y conectar los descriptores de fichero. Como Julia hace este trabajo, retiene un mejor control y puede hacer algunas cosas que el shell no puede.

Julia puede ejecutar múltiples órdenes en paralelo:

```
julia> run(`echo hello` & `echo world`)
world
hello
```

El orden de esta salida es no determinista debido a que los dos procesos echo se lanzan casi simultáneamente, y compiten para hacer la primera escritura al descriptor `STDOUT` que ambas comparten con el proceso padre `julia`. Julia te premite entubar la salida desde estos procesos a otro programa:

```
julia> run(pipeline(`echo world` & `echo hello`, `sort`))
hello
world
```

En términos de fontanería UNIX, lo que está pasando aquí es que un único objeto tubería de UNIX se ha creado y es escrito por dos procesos echo y el otro extremo al final de la tubería es leído por la orden `sort`.

La redirección de la E/S puede conseguirse pasando los argumentos clave stdin, stdout y stderr a la función `pipeline`:

```
pipeline(`do_work`, stdout=pipeline(`sort`, "out.txt"), stderr="errs.txt")
```

Evitar interbloqueos en tuberías

Cuando leemos y escribirmos en los dos extremos de una tubería desde un solo proceso, es importante evitar forzar el núcleo a almacenar todos los datos en el buffer.

Por ejemplo, cuando leemos toda la salida de un mandato, llamamos a `readstring(out)`, no `wait(process)`, ya que el primero consumirá activamente todos los datos exritos por el proceso, mientras que el último interntará almacenar los datos en los búferes del kernel mientras espera a que un lector esté conectado.

Otra solución común es separar el lector y el escritor de la tuberían en tareas separadas:

```
writer = @async writeall(process, "data")
reader = @async do_compute(readstring(process))
wait(process)
fetch(reader)
```

Ejemplo complicado

La combinación de un lenguaje de programación de alto nivel, una abstracción de mandatos de primera clase y la inicialización automática de tuberías entre procesos es muy poderos. Para dar algun sentido a las tuberías complejas que pueden ser creadas facilmente, he aquí algunos ejemplos más sofisticado, con nuestras disculpas por el excesito uso de scripts Perl con una sola línea:

```
julia> prefixer(prefix, sleep) = `perl -nle '$|=1; print "'$prefix' ", $_; sleep '$sleep';'`;

julia> run(pipeline(`perl -le '$|=1; for(0..9){ print; sleep 1 }'`, prefixer("A",2) &
↪   prefixer("B",2)))
A 0
B 1
A 2
B 3
A 4
B 5
A 6
B 7
A 8
B 9
```

Este es el ejemplo típico de un solo productor que alimenta dos consumidores concurrentes: un proceso `perl` genera líneas con los número 0 a 9, el otro con la letra ”B”. Qué consumidor llega el primero es no determinista, pero una vez que se ha ganado la carrera, las líneas son consumidas alternativamente primero por un proceso y después por el otro. (Fijas `$|=1` en Perl causa que cada instrucción de impresión vuelque al flujo `STDOUT`, lo cuál es necesario para que este ejemplo funcione. En caso contrario toda la salida va a un buffer y sería impresa en la tubería de una vez, para ser leída por un solo proceso consumidor).

He aquí un ejemplo incluso más complicado de productor consumidor multi-etapa:

```
julia> run(pipeline(`perl -le '$|=1; for(0..9){ print; sleep 1 }'`,

           prefixer("X",3) & prefixer("Y",3) & prefixer("Z",3),

           prefixer("A",2) & prefixer("B",2)))
A X 0
B Y 1
A Z 2
```

```
B X 3
A Y 4
B Z 5
A X 6
B Y 7
A Z 8
B X 9
```

Este ejemplo es similar al anterior, excepto en que hay dos etapas de consumidores y las estapas tiene diferente latencia por lo que usan un número de workers paralelos diferrentes, para mantener saturado el throughput .

Se recomienda intentar todos estos ejemplos y ver cómo funcionan.

Chapter 30

Llamando a código C y Fortran

Aunque la mayoría del código se puede escribir en Julia, hay muchas bibliotecas maduras de alta calidad para computación numérica ya escritas en C y Fortran. Para permitir el uso fácil de este código existente, Julia hace que sea sencillo y eficiente llamar a las funciones C y Fortran. Julia tiene una filosofía de "no repetitivo": las funciones se pueden llamar directamente desde Julia sin ningún código de "pegamento", generación de código o compilación, incluso desde el aviso interactivo. Esto se logra haciendo una llamada apropiada con la sintaxis `ccall`, que se parece a una llamada de función ordinaria.

El código que se debe llamar debe estar disponible como una biblioteca compartida. La mayoría de las bibliotecas C y Fortran ya se han compilado como bibliotecas compartidas, pero si está compilando el código usted mismo usando GCC (o Clang), necesitará usar las opciones `-shared` y `-fPIC`. Las instrucciones de la máquina generadas por el JIT de Julia son las mismas que una llamada C nativa, por lo que la sobrecarga resultante es lo mismo que llamar a una función de biblioteca desde el código C. (Las llamadas a funciones que no son de la biblioteca en C y Julia pueden estar incluidas y, por lo tanto, pueden tener incluso menos gastos generales que las llamadas a funciones de biblioteca compartidas. Cuando LLVM genera bibliotecas y ejecutables, es posible realizar optimizaciones de todo el programa que incluso optimizar a través de este límite, pero Julia aún no lo admite. Sin embargo, en el futuro, puede hacerlo, produciendo ganancias de rendimiento incluso mayores).

Las bibliotecas y funciones compartidas están referenciadas por una tupla de la forma `(:función,"biblioteca")` o `("función", "biblioteca")` donde `función` es el nombre de función exportado por C. `library` se refiere al nombre de la biblioteca compartida: las bibliotecas compartidas disponibles en la ruta de carga (específica de la plataforma) se resolverán por nombre y, si es necesario, se puede especificar una ruta directa.

El nombre de una función se puede usar solo en lugar de la tupla (solo `:función` o`"función"`). En este caso, el nombre se resuelve dentro del proceso actual. Este formulario se puede usar para llamar funciones de biblioteca C, funciones en el tiempo de ejecución de Julia o funciones en una aplicación vinculada a Julia.

Por defecto, los compiladores Fortran generan nombres destrozados (por ejemplo, convirtiendo nombres de funciones a minúsculas o mayúsculas, a menudo añadiendo un guión bajo), y para llamar a una función Fortran a través de `ccall` debe pasar el identificador mutilado correspondiente a la regla seguida por su compilador Fortran . Además, cuando se llama a una función Fortran, todas las entradas se deben pasar por referencia.

Por último, se puede usar `ccall` para generar de hecho una llamada a la función de librería. Los argumentos a `ccall` son los siguientes:

1. Una pareja `(:función, "librería")`, que debe ser escrita como una constante literal,

   O

un puntero a función (por ejemplo, de `dlsym`).

2. Tipo de retorno (ver abajo para la correspondencia entre el tipo declarado en C y Julia)
    - Este argumento será evaluado en tiempo de compilación, cuando se defina el método que lo contiene.
3. Una tupla de tipos de entrada. Los tipos de entrada deben ser escritos como un literal tupla, no como una variable o expresión de valor tupla.
    - Este argumento será evaluado en tiempo de compilación, cuando se defina el método que lo contiene.
4. Los siguientes argumentos, si los hay, son los valores de los argumentos actuales pasados a la función.

Como un ejemplo completo pero simple, el siguiente código llama a la función `clock` de la librería estándar C:

```
julia> t = ccall((:clock, "libc"), Int32, ())
2292761

julia> t
2292761

julia> typeof(ans)
Int32
```

`clock` no toma argumentos y devuelve un `Int32`. Un problema común es que una 1-tupla debe escribirse con una coma al final. Por ejemplo, para llamar a la función `getenv` para obtener un puntero al valor de una variable de entorno, se realiza una llamada como esta:

```
julia> path = ccall((:getenv, "libc"), Cstring, (Cstring,), "SHELL")
Cstring(@0x00007fff5fbffc45)

julia> unsafe_string(path)
"/bin/bash"
```

Note que la tupla del tipo de argumento debe ser escrita como `(Cstring,)`, en lugar de como `(Cstring)`. Esto es debido a que `(Cstring)` es justo la expresión `Cstring` rodeada entre paréntesis, en lugar de una tupla que contiene a `Cstring`:

```
julia> (Cstring)
Cstring

julia> (Cstring,)
(Cstring,)
```

En la práctica, especialmente cuando se proporciona funcionalidad reutilizable, generalmente se envuelve el usp de `ccall` en funciones de Julia que configuran argumentos y luego se comprueban los errores de cualquier forma que la función C o Fortran los indique, propagándose al código que llama desde Julia como excepciones. Esto es especialmente importante ya que las API de C y Fortran son notoriamente inconsistentes sobre cómo indican las condiciones de error. Por ejemplo, la función de biblioteca C `getenv` está envuelta en la siguiente función Julia, que es una versión simplificada de la definición real de `env.jl`:

```
function getenv(var::AbstractString)
    val = ccall((:getenv, "libc"),
                Cstring, (Cstring,), var)
    if val == C_NULL
        error("getenv: undefined variable: ", var)
    end
    unsafe_string(val)
end
```

La función C `getenv` indica un error al devolver`NULL`, pero otras funciones C estándar indican errores de varias maneras diferentes, incluyendo al devolver -1, 0, 1 y otros valores especiales. Este contenedor arroja una excepción que indica claramente el problema si la persona que llama intenta obtener una variable de entorno inexistente:

```
julia> getenv("SHELL")
"/bin/bash"

julia> getenv("FOOBAR")
getenv: undefined variable: FOOBAR
```

Aquí hay un ejemplo ligeramente ms complejo que descubre el nombre de host de la máquina local:

```
function gethostname()
    hostname = Vector{UInt8}(128)
    ccall((:gethostname, "libc"), Int32,
          (Ptr{UInt8}, Csize_t),
          hostname, sizeof(hostname))
    hostname[end] = 0; # ensure null-termination
    return unsafe_string(pointer(hostname))
end
```

Este ejemplo primero asigna un array de bytes, luego llama a la función de biblioteca C `gethostname` para llenar el array con el nombre de host, toma un puntero al buffer de nombre de host, y convierte el puntero a una cadena Julia, asumiendo que es una cadena C terminada en NUL. Es común que las bibliotecas C usen este patrón de requerir al llamador que asigne memoria para que la pase al llamado y la complete. La asignación de la memoria de Julia se logra generalmente creando un array no inicializada y pasando un puntero a sus datos a la función C. Es por eso que no usamos el tipo `Cstring` aquí: como la matriz no está inicializada, podría contener bytes NUL. Convertir a `Cstring` como parte de `ccall` comprueba si hay bytes NUL contenidos y, por lo tanto, puede arrojar un error de conversión.

30.1 Creando Punteros a Función Julia Compatibles con C

Es posible pasar funciones Julia a funciones C nativas que aceptan argumentos punteros a función. Por ejemplo, para emparejar prototipos C de la forma:

```
typedef returntype (*functiontype)(argumenttype,...)
```

La función `cfunction()` genera el puntero a función compatible con C para una llamadan a una función de biblioteca de Julia. Los argumentos a `cfunction()` son los siguientes:

1. Una función Julia
2. Tipo de retorno
3. Una tupla de tipos de entrada

Un ejemplo clásico es la función estándar de biblioteca C `qsort` declarada como:

```
void qsort(void *base, size_t nmemb, size_t size,
           int(*compare)(const void *a, const void *b));
```

The `base` argument is a pointer to an array of length `nmemb`, with elements of `size` bytes each. `compare` is a callback function which takes pointers to two elements a and b and returns an integer less/greater than zero if a should appear before/after b (or zero if any order is permitted). Now, suppose that we have a `1d` array `A` of values in Julia that we want to sort using the `qsort` function (rather than Julia's built-in `sort` function). Before we worry about calling `qsort` and passing arguments, we need to write a comparison function that works for some arbitrary type T:

```
julia> function mycompare(a::T, b::T) where T
           return convert(Cint, a < b ? -1 : a > b ? +1 : 0)::Cint
       end
mycompare (generic function with 1 method)
```

Notice that we have to be careful about the return type: `qsort` expects a function returning a C `int`, so we must be sure to return `Cint` via a call to `convert` and a `typeassert`.

In order to pass this function to C, we obtain its address using the function `cfunction`:

```
julia> const mycompare_c = cfunction(mycompare, Cint, (Ref{Cdouble}, Ref{Cdouble}));
```

`cfunction()` accepts three arguments: the Julia function (`mycompare`), the return type (`Cint`), and a tuple of the argument types, in this case to sort an array of `Cdouble` (`Float64`) elements.

The final call to `qsort` looks like this:

```
julia> A = [1.3, -2.7, 4.4, 3.1]
4-element Array{Float64,1}:
  1.3
 -2.7
  4.4
  3.1

julia> ccall(:qsort, Void, (Ptr{Cdouble}, Csize_t, Csize_t, Ptr{Void}),
             A, length(A), sizeof(eltype(A)), mycompare_c)

julia> A
4-element Array{Float64,1}:
 -2.7
  1.3
  3.1
  4.4
```

As can be seen, `A` is changed to the sorted array `[-2.7, 1.3, 3.1, 4.4]`. Note that Julia knows how to convert an array into a `Ptr{Cdouble}`, how to compute the size of a type in bytes (identical to C's `sizeof` operator), and so on. For fun, try inserting a `println("mycompare($a,$b)")` line into `mycompare`, which will allow you to see the comparisons that `qsort` is performing (and to verify that it is really calling the Julia function that you passed to it).

30.2 Mapping C Types to Julia

It is critical to exactly match the declared C type with its declaration in Julia. Inconsistencies can cause code that works correctly on one system to fail or produce indeterminate results on a different system.

Note that no C header files are used anywhere in the process of calling C functions: you are responsible for making sure that your Julia types and call signatures accurately reflect those in the C header file. (The Clang package can be used to auto-generate Julia code from a C header file.)

Auto-conversion:

Julia automatically inserts calls to the `Base.cconvert()` function to convert each argument to the specified type. For example, the following call:

```
ccall((:foo, "libfoo"), Void, (Int32, Float64), x, y)
```

will behave as if the following were written:

```
ccall((:foo, "libfoo"), Void, (Int32, Float64),
      Base.unsafe_convert(Int32, Base.cconvert(Int32, x)),
      Base.unsafe_convert(Float64, Base.cconvert(Float64, y)))
```

`Base.cconvert()` normally just calls `convert()`, but can be defined to return an arbitrary new object more appropriate for passing to C. For example, this is used to convert an `Array` of objects (e.g. strings) to an array of pointers.

`Base.unsafe_convert()` handles conversion to `Ptr` types. It is considered unsafe because converting an object to a native pointer can hide the object from the garbage collector, causing it to be freed prematurely.

Type Correspondences:

First, a review of some relevant Julia type terminology:

Syntax / Keyword	Example	Description
`mutable struct`	`String`	"Leaf Type" :: A group of related data that includes a type-tag, is managed by the Julia GC, and is defined by object-identity. The type parameters of a leaf type must be fully defined (no `TypeVars` are allowed) in order for the instance to be constructed.
`abstract type`	`Any`, `AbstractArray{T, N}`, `Complex{T}`	"Super Type" :: A super-type (not a leaf-type) that cannot be instantiated, but can be used to describe a group of types.
`T{A}`	`Vector{Int}`	"Type Parameter" :: A specialization of a type (typically used for dispatch or storage optimization).
		"TypeVar" :: The T in the type parameter declaration is referred to as a TypeVar (short for type variable).
`primitive type`	`Int`, `Float64`	"Primitive Type" :: A type with no fields, but a size. It is stored and defined by-value.
`struct`	`Pair{Int, Int}`	"Struct" :: A type with all fields defined to be constant. It is defined by-value, and may be stored with a type-tag.
	`Complex128` (`isbits`)	"Is-Bits" :: A `primitive type`, or a `struct` type where all fields are other `isbits` types. It is defined by-value, and is stored without a type-tag.
`struct ...; end`	`nothing`	"Singleton" :: a Leaf Type or Struct with no fields.
`(...)` or `tuple(...)`	`(1, 2, 3)`	"Tuple" :: an immutable data-structure similar to an anonymous struct type, or a constant array. Represented as either an array or a struct.

Bits Types:

There are several special types to be aware of, as no other type can be defined to behave the same:

- `Float32`

  Exactly corresponds to the `float` type in C (or `REAL*4` in Fortran).

- `Float64`

  Exactly corresponds to the `double` type in C (or `REAL*8` in Fortran).

- `Complex64`

  Exactly corresponds to the `complex float` type in C (or `COMPLEX*8` in Fortran).

- `Complex128`

  Exactly corresponds to the `complex double` type in C (or `COMPLEX*16` in Fortran).

- `Signed`

  Exactly corresponds to the `signed` type annotation in C (or any `INTEGER` type in Fortran). Any Julia type that is not a subtype of `Signed` is assumed to be unsigned.

- `Ref{T}`

  Behaves like a `Ptr{T}` that can manage its memory via the Julia GC.

- `Array{T,N}`

  When an array is passed to C as a `Ptr{T}` argument, it is not reinterpret-cast: Julia requires that the element type of the array matches `T`, and the address of the first element is passed.

  Therefore, if an `Array` contains data in the wrong format, it will have to be explicitly converted using a call such as `trunc(Int32, a)`.

  To pass an array `A` as a pointer of a different type *without* converting the data beforehand (for example, to pass a `Float64` array to a function that operates on uninterpreted bytes), you can declare the argument as `Ptr{Void}`.

  If an array of eltype `Ptr{T}` is passed as a `Ptr{Ptr{T}}` argument, `Base.cconvert()` will attempt to first make a null-terminated copy of the array with each element replaced by its `Base.cconvert()` version. This allows, for example, passing an `argv` pointer array of type `Vector{String}` to an argument of type `Ptr{Ptr{Cchar}}`.

On all systems we currently support, basic C/C++ value types may be translated to Julia types as follows. Every C type also has a corresponding Julia type with the same name, prefixed by C. This can help for writing portable code (and remembering that an `int` in C is not the same as an `Int` in Julia).

System Independent:

The `Cstring` type is essentially a synonym for `Ptr{UInt8}`, except the conversion to `Cstring` throws an error if the Julia string contains any embedded NUL characters (which would cause the string to be silently truncated if the C routine treats NUL as the terminator). If you are passing a `char*` to a C routine that does not assume NUL termination (e.g. because you pass an explicit string length), or if you know for certain that your Julia string does not contain NUL and want to skip the check, you can use `Ptr{UInt8}` as the argument type. `Cstring` can also be used as the `ccall` return type, but in that case it obviously does not introduce any extra checks and is only meant to improve readability of the call.

System-dependent:

Note

When calling a Fortran function, all inputs must be passed by reference, so all type correspondences above should contain an additional `Ptr{..}` or `Ref{..}` wrapper around their type specification.

Warning

For string arguments (`char*`) the Julia type should be `Cstring` (if NUL- terminated data is expected) or either `Ptr{Cchar}` or `Ptr{UInt8}` otherwise (these two pointer types have the same effect), as described above, not `String`. Similarly, for array arguments (`T[]` or `T*`), the Julia type should again be `Ptr{T}`, not `Vector{T}`.

Warning

Julia's `Char` type is 32 bits, which is not the same as the wide character type (`wchar_t` or `wint_t`) on all platforms.

Warning

A return type of `Union{}` means the function will not return i.e. C++11 `[[noreturn]]` or C11 `_Noreturn` (e.g. `jl_throw` or `longjmp`). Do not use this for functions that return no value (`void`) but do return, use `Void` instead.

Note

For `wchar_t*` arguments, the Julia type should be `Cwstring` (if the C routine expects a NUL-terminated string) or `Ptr{Cwchar_t}` otherwise. Note also that UTF-8 string data in Julia is internally NUL-terminated, so it can be passed to C functions expecting NUL-terminated data without making a copy (but using the `Cwstring` type will cause an error to be thrown if the string itself contains NUL characters).

Note

C functions that take an argument of the type `char**` can be called by using a `Ptr{Ptr{UInt8}}` type within Julia. For example, C functions of the form:

```
int main(int argc, char **argv);
```

can be called via the following Julia code:

```
argv = [ "a.out", "arg1", "arg2" ]
ccall(:main, Int32, (Int32, Ptr{Ptr{UInt8}}), length(argv), argv)
```

Note

A C function declared to return `Void` will return the value `nothing` in Julia.

Struct Type correspondences

Composite types, aka `struct` in C or `TYPE` in Fortran90 (or `STRUCTURE` / `RECORD` in some variants of F77), can be mirrored in Julia by creating a `struct` definition with the same field layout.

When used recursively, `isbits` types are stored inline. All other types are stored as a pointer to the data. When mirroring a struct used by-value inside another struct in C, it is imperative that you do not attempt to manually copy the fields over, as this will not preserve the correct field alignment. Instead, declare an `isbits` struct type and use that instead. Unnamed structs are not possible in the translation to Julia.

Packed structs and union declarations are not supported by Julia.

You can get a near approximation of a `union` if you know, a priori, the field that will have the greatest size (potentially including padding). When translating your fields to Julia, declare the Julia field to be only of that type.

Arrays of parameters can be expressed with `NTuple`:

```
in C:
struct B {
    int A[3];
};
b_a_2 = B.A[2];

in Julia:
struct B
    A::NTuple{3, CInt}
end
b_a_2 = B.A[3]  # note the difference in indexing (1-based in Julia, 0-based in C)
```

Arrays of unknown size (C99-compliant variable length structs specified by `[]` or `[0]`) are not directly supported. Often the best way to deal with these is to deal with the byte offsets directly. For example, if a C library declared a proper string type and returned a pointer to it:

```
struct String {
    int strlen;
    char data[];
};
```

In Julia, we can access the parts independently to make a copy of that string:

```
str = from_c::Ptr{Void}
len = unsafe_load(Ptr{Cint}(str))
unsafe_string(str + Core.sizeof(Cint), len)
```

Type Parameters

The type arguments to `ccall` are evaluated statically, when the method containing the ccall is defined. They therefore must take the form of a literal tuple, not a variable, and cannot reference local variables.

This may sound like a strange restriction, but remember that since C is not a dynamic language like Julia, its functions can only accept argument types with a statically-known, fixed signature.

However, while the type layout must be known statically to compute the `ccall` ABI, the static parameters of the function are considered to be part of this static environment. The static parameters of the function may be used as type parameters in the `ccall` signature, as long as they don't affect the layout of the type. For example, `f(x::T) where {T} = ccall(:valid, Ptr{T}, (Ptr{T},), x)` is valid, since `Ptr` is always a word-size primitive type. But, `g(x::T) where {T} = ccall(:notvalid, T, (T,), x)` is not valid, since the type layout of `T` is not known statically.

SIMD Values

Note: This feature is currently implemented on 64-bit x86 and AArch64 platforms only.

If a C/C++ routine has an argument or return value that is a native SIMD type, the corresponding Julia type is a homogeneous tuple of `VecElement` that naturally maps to the SIMD type. Specifically:

- The tuple must be the same size as the SIMD type. For example, a tuple representing an `__m128` on x86 must have a size of 16 bytes.

- The element type of the tuple must be an instance of `VecElement{T}` where `T` is a primitive type that is 1, 2, 4 or 8 bytes.

For instance, consider this C routine that uses AVX intrinsics:

```
#include <immintrin.h>

__m256 dist( __m256 a, __m256 b ) {
    return _mm256_sqrt_ps(_mm256_add_ps(_mm256_mul_ps(a, a),
                                        _mm256_mul_ps(b, b)));
}
```

The following Julia code calls `dist` using `ccall`:

```
const m256 = NTuple{8, VecElement{Float32}}

a = m256(ntuple(i -> VecElement(sin(Float32(i))), 8))
b = m256(ntuple(i -> VecElement(cos(Float32(i))), 8))

function call_dist(a::m256, b::m256)
    ccall((:dist, "libdist"), m256, (m256, m256), a, b)
end

println(call_dist(a,b))
```

The host machine must have the requisite SIMD registers. For example, the code above will not work on hosts without AVX support.

Memory Ownership

malloc/free

Memory allocation and deallocation of such objects must be handled by calls to the appropriate cleanup routines in the libraries being used, just like in any C program. Do not try to free an object received from a C library with `Libc.free` in Julia, as this may result in the `free` function being called via the wrong `libc` library and cause Julia to crash. The reverse (passing an object allocated in Julia to be freed by an external library) is equally invalid.

When to use T, Ptr{T} and Ref{T}

In Julia code wrapping calls to external C routines, ordinary (non-pointer) data should be declared to be of type `T` inside the `ccall`, as they are passed by value. For C code accepting pointers, `Ref{T}` should generally be used for the types of input arguments, allowing the use of pointers to memory managed by either Julia or C through the implicit call to `Base.cconvert()`. In contrast, pointers returned by the C function called should be declared to be of output type `Ptr{T}`, reflecting that the memory pointed to is managed by C only. Pointers contained in C structs should be represented as fields of type `Ptr{T}` within the corresponding Julia struct types designed to mimic the internal structure of corresponding C structs.

In Julia code wrapping calls to external Fortran routines, all input arguments should be declared as of type `Ref{T}`, as Fortran passes all variables by reference. The return type should either be `Void` for Fortran subroutines, or a `T` for Fortran functions returning the type `T`.

30.3 Mapping C Functions to Julia

`ccall`/`cfunction` argument translation guide

For translating a C argument list to Julia:

- `T`, where `T` is one of the primitive types: `char`, `int`, `long`, `short`, `float`, `double`, `complex`, `enum` or any of their `typedef` equivalents
  - `T`, where `T` is an equivalent Julia Bits Type (per the table above)
  - if `T` is an `enum`, the argument type should be equivalent to `Cint` or `Cuint`
  - argument value will be copied (passed by value)
- `struct T` (including typedef to a struct)
  - `T`, where `T` is a Julia leaf type
  - argument value will be copied (passed by value)
- `void*`
  - depends on how this parameter is used, first translate this to the intended pointer type, then determine the Julia equivalent using the remaining rules in this list
  - this argument may be declared as `Ptr{Void}`, if it really is just an unknown pointer
- `jl_value_t*`
  - `Any`
  - argument value must be a valid Julia object
  - currently unsupported by `cfunction()`
- `jl_value_t**`
  - `Ref{Any}`
  - argument value must be a valid Julia object (or `C_NULL`)
  - currently unsupported by `cfunction()`
- `T*`
  - `Ref{T}`, where `T` is the Julia type corresponding to `T`
  - argument value will be copied if it is an `isbits` type otherwise, the value must be a valid Julia object
- `(T*)(...)` (e.g. a pointer to a function)
  - `Ptr{Void}` (you may need to use `cfunction()` explicitly to create this pointer)
- `...` (e.g. a vararg)
  - `T...`, where `T` is the Julia type
- `va_arg`
  - not supported

`ccall`/`cfunction` return type translation guide

For translating a C return type to Julia:

- `void`
  - `Void` (this will return the singleton instance `nothing::Void`)
- T, where T is one of the primitive types: `char`, `int`, `long`, `short`, `float`, `double`, `complex`, `enum` or any of their `typedef` equivalents
  - T, where T is an equivalent Julia Bits Type (per the table above)
  - if T is an `enum`, the argument type should be equivalent to `Cint` or `Cuint`
  - argument value will be copied (returned by-value)
- `struct T` (including typedef to a struct)
  - T, where T is a Julia Leaf Type
  - argument value will be copied (returned by-value)
- `void*`
  - depends on how this parameter is used, first translate this to the intended pointer type, then determine the Julia equivalent using the remaining rules in this list
  - this argument may be declared as `Ptr{Void}`, if it really is just an unknown pointer
- `jl_value_t*`
  - `Any`
  - argument value must be a valid Julia object
- `jl_value_t**`
  - `Ref{Any}`
  - argument value must be a valid Julia object (or `C_NULL`)
- `T*`
  - If the memory is already owned by Julia, or is an `isbits` type, and is known to be non-null:
    - `Ref{T}`, where T is the Julia type corresponding to T
    - a return type of `Ref{Any}` is invalid, it should either be `Any` (corresponding to `jl_value_t*`) or `Ptr{Any}` (corresponding to `Ptr{Any}`)
    - C **MUST NOT** modify the memory returned via `Ref{T}` if T is an `isbits` type
  - If the memory is owned by C:
    - `Ptr{T}`, where T is the Julia type corresponding to T
- `(T*)(...)` (e.g. a pointer to a function)
  - `Ptr{Void}` (you may need to use `cfunction()` explicitly to create this pointer)

Passing Pointers for Modifying Inputs

Because C doesn't support multiple return values, often C functions will take pointers to data that the function will modify. To accomplish this within a `ccall`, you need to first encapsulate the value inside an `Ref{T}` of the appropriate type. When you pass this `Ref` object as an argument, Julia will automatically pass a C pointer to the encapsulated data:

```
width = Ref{Cint}(0)
range = Ref{Cfloat}(0)
ccall(:foo, Void, (Ref{Cint}, Ref{Cfloat}), width, range)
```

Upon return, the contents of `width` and `range` can be retrieved (if they were changed by `foo`) by `width[]` and `range[]`; that is, they act like zero-dimensional arrays.

Special Reference Syntax for ccall (deprecated):

The & syntax is deprecated, use the `Ref{T}` argument type instead.

A prefix & is used on an argument to `ccall` to indicate that a pointer to a scalar argument should be passed instead of the scalar value itself (required for all Fortran function arguments, as noted above). The following example computes a dot product using a BLAS function.

```
function compute_dot(DX::Vector{Float64}, DY::Vector{Float64})
    @assert length(DX) == length(DY)
    n = length(DX)
    incx = incy = 1
    product = ccall((:ddot_, "libLAPACK"),
                    Float64,
                    (Ptr{Int32}, Ptr{Float64}, Ptr{Int32}, Ptr{Float64}, Ptr{Int32}),
                    &n, DX, &incx, DY, &incy)
    return product
end
```

The meaning of prefix & is not quite the same as in C. In particular, any changes to the referenced variables will not be visible in Julia unless the type is mutable (declared via `type`). However, even for immutable structs it will not cause any harm for called functions to attempt such modifications (that is, writing through the passed pointers). Moreover, & may be used with any expression, such as `&0` or `&f(x)`.

When a scalar value is passed with & as an argument of type `Ptr{T}`, the value will first be converted to type `T`.

30.4 Some Examples of C Wrappers

Here is a simple example of a C wrapper that returns a `Ptr` type:

```
mutable struct gsl_permutation
end

# The corresponding C signature is
#     gsl_permutation * gsl_permutation_alloc (size_t n);
function permutation_alloc(n::Integer)
    output_ptr = ccall(
        (:gsl_permutation_alloc, :libgsl), # name of C function and library
        Ptr{gsl_permutation},              # output type
        (Csize_t,),                        # tuple of input types
```

```
        n                                   # name of Julia variable to pass in
    )
    if output_ptr == C_NULL # Could not allocate memory
        throw(OutOfMemoryError())
    end
    return output_ptr
end
```

The GNU Scientific Library (here assumed to be accessible through `:libgsl`) defines an opaque pointer, `gsl_permutation *`, as the return type of the C function `gsl_permutation_alloc()`. As user code never has to look inside the `gsl_permutation` struct, the corresponding Julia wrapper simply needs a new type declaration, `gsl_permutation`, that has no internal fields and whose sole purpose is to be placed in the type parameter of a `Ptr` type. The return type of the `ccall` is declared as `Ptr{gsl_permutation}`, since the memory allocated and pointed to by `output_ptr` is controlled by C (and not Julia).

The input n is passed by value, and so the function's input signature is simply declared as `(Csize_t,)` without any `Ref` or `Ptr` necessary. (If the wrapper was calling a Fortran function instead, the corresponding function input signature should instead be `(Ref{Csize_t},)`, since Fortran variables are passed by reference.) Furthermore, n can be any type that is convertable to a `Csize_t` integer; the `ccall` implicitly calls `Base.cconvert(Csize_t, n)`.

Here is a second example wrapping the corresponding destructor:

```
# The corresponding C signature is
#     void gsl_permutation_free (gsl_permutation * p);
function permutation_free(p::Ref{gsl_permutation})
    ccall(
        (:gsl_permutation_free, :libgsl), # name of C function and library
        Void,                             # output type
        (Ref{gsl_permutation},),          # tuple of input types
        p                                 # name of Julia variable to pass in
    )
end
```

Here, the input p is declared to be of type `Ref{gsl_permutation}`, meaning that the memory that p points to may be managed by Julia or by C. A pointer to memory allocated by C should be of type `Ptr{gsl_permutation}`, but it is convertable using `Base.cconvert()` and therefore can be used in the same (covariant) context of the input argument to a `ccall`. A pointer to memory allocated by Julia must be of type `Ref{gsl_permutation}`, to ensure that the memory address pointed to is valid and that Julia's garbage collector manages the chunk of memory pointed to correctly. Therefore, the `Ref{gsl_permutation}` declaration allows pointers managed by C or Julia to be used.

If the C wrapper never expects the user to pass pointers to memory managed by Julia, then using `p::Ptr{gsl_permutation}` for the method signature of the wrapper and similarly in the `ccall` is also acceptable.

Here is a third example passing Julia arrays:

```
# The corresponding C signature is
#    int gsl_sf_bessel_Jn_array (int nmin, int nmax, double x,
#                                double result_array[])
function sf_bessel_Jn_array(nmin::Integer, nmax::Integer, x::Real)
    if nmax < nmin
        throw(DomainError())
    end
    result_array = Vector{Cdouble}(nmax - nmin + 1)
    errorcode = ccall(
        (:gsl_sf_bessel_Jn_array, :libgsl), # name of C function and library
```

```
        Cint,                               # output type
        (Cint, Cint, Cdouble, Ref{Cdouble}),# tuple of input types
        nmin, nmax, x, result_array         # names of Julia variables to pass in
    )
    if errorcode != 0
        error("GSL error code $errorcode")
    end
    return result_array
end
```

The C function wrapped returns an integer error code; the results of the actual evaluation of the Bessel J function populate the Julia array `result_array`. This variable can only be used with corresponding input type declaration `Ref{Cdouble}`, since its memory is allocated and managed by Julia, not C. The implicit call to `Base.cconvert(Ref{Cdouble}, result_array)` unpacks the Julia pointer to a Julia array data structure into a form understandable by C.

Note that for this code to work correctly, `result_array` must be declared to be of type `Ref{Cdouble}` and not `Ptr{Cdouble}`. The memory is managed by Julia and the `Ref` signature alerts Julia's garbage collector to keep managing the memory for `result_array` while the `ccall` executes. If `Ptr{Cdouble}` were used instead, the `ccall` may still work, but Julia's garbage collector would not be aware that the memory declared for `result_array` is being used by the external C function. As a result, the code may produce a memory leak if `result_array` never gets freed by the garbage collector, or if the garbage collector prematurely frees `result_array`, the C function may end up throwing an invalid memory access exception.

30.5 Garbage Collection Safety

When passing data to a `ccall`, it is best to avoid using the `pointer()` function. Instead define a convert method and pass the variables directly to the `ccall`. `ccall` automatically arranges that all of its arguments will be preserved from garbage collection until the call returns. If a C API will store a reference to memory allocated by Julia, after the `ccall` returns, you must arrange that the object remains visible to the garbage collector. The suggested way to handle this is to make a global variable of type `Array{Ref,1}` to hold these values, until the C library notifies you that it is finished with them.

Whenever you have created a pointer to Julia data, you must ensure the original data exists until you are done with using the pointer. Many methods in Julia such as `unsafe_load()` and `String()` make copies of data instead of taking ownership of the buffer, so that it is safe to free (or alter) the original data without affecting Julia. A notable exception is `unsafe_wrap()` which, for performance reasons, shares (or can be told to take ownership of) the underlying buffer.

The garbage collector does not guarantee any order of finalization. That is, if a contained a reference to b and both a and b are due for garbage collection, there is no guarantee that b would be finalized after a. If proper finalization of a depends on b being valid, it must be handled in other ways.

30.6 Non-constant Function Specifications

A `(name, library)` function specification must be a constant expression. However, it is possible to use computed values as function names by staging through `eval` as follows:

```
@eval ccall(($(string("a", "b")), "lib"), ...
```

This expression constructs a name using `string`, then substitutes this name into a new `ccall` expression, which is then evaluated. Keep in mind that `eval` only operates at the top level, so within this expression local variables will not be available (unless their values are substituted with $). For this reason, `eval` is typically only used to form top-level definitions, for example when wrapping libraries that contain many similar functions.

If your usage is more dynamic, use indirect calls as described in the next section.

30.7 Indirect Calls

The first argument to `ccall` can also be an expression evaluated at run time. In this case, the expression must evaluate to a `Ptr`, which will be used as the address of the native function to call. This behavior occurs when the first `ccall` argument contains references to non-constants, such as local variables, function arguments, or non-constant globals.

For example, you might look up the function via `dlsym`, then cache it in a global variable for that session. For example:

```
macro dlsym(func, lib)
    z, zlocal = gensym(string(func)), gensym()
    eval(current_module(), :(global $z = C_NULL))
    z = esc(z)
    quote
        let $zlocal::Ptr{Void} = $z::Ptr{Void}
            if $zlocal == C_NULL
                $zlocal = dlsym($(esc(lib))::Ptr{Void}, $(esc(func)))
                global $z = $zlocal
            end
            $zlocal
        end
    end
end

mylibvar = Libdl.dlopen("mylib")
ccall(@dlsym("myfunc", mylibvar), Void, ())
```

30.8 Calling Convention

The second argument to `ccall` can optionally be a calling convention specifier (immediately preceding return type). Without any specifier, the platform-default C calling convention is used. Other supported conventions are: `stdcall`, `cdecl`, `fastcall`, and `thiscall`. For example (from `base/libc.jl`) we see the same `gethostname`ccall as above, but with the correct signature for Windows:

```
hn = Vector{UInt8}(256)
err = ccall(:gethostname, stdcall, Int32, (Ptr{UInt8}, UInt32), hn, length(hn))
```

For more information, please see the LLVM Language Reference.

There is one additional special calling convention `llvmcall`, which allows inserting calls to LLVM intrinsics directly. This can be especially useful when targeting unusual platforms such as GPGPUs. For example, for CUDA, we need to be able to read the thread index:

```
ccall("llvm.nvvm.read.ptx.sreg.tid.x", llvmcall, Int32, ())
```

As with any `ccall`, it is essential to get the argument signature exactly correct. Also, note that there is no compatibility layer that ensures the intrinsic makes sense and works on the current target, unlike the equivalent Julia functions exposed by `Core.Intrinsics`.

30.9 Accessing Global Variables

Global variables exported by native libraries can be accessed by name using the `cglobal()` function. The arguments to `cglobal()` are a symbol specification identical to that used by `ccall`, and a type describing the value stored in the variable:

```
julia> cglobal((:errno, :libc), Int32)
Ptr{Int32} @0x00007f418d0816b8
```

The result is a pointer giving the address of the value. The value can be manipulated through this pointer using `unsafe_load()` and `unsafe_store!()`.

30.10 Accessing Data through a Pointer

The following methods are described as "unsafe" because a bad pointer or type declaration can cause Julia to terminate abruptly.

Given a `Ptr{T}`, the contents of type T can generally be copied from the referenced memory into a Julia object using `unsafe_load(ptr, [index])`. The index argument is optional (default is 1), and follows the Julia-convention of 1-based indexing. This function is intentionally similar to the behavior of `getindex()` and `setindex!()` (e.g. `[]` access syntax).

The return value will be a new object initialized to contain a copy of the contents of the referenced memory. The referenced memory can safely be freed or released.

If T is `Any`, then the memory is assumed to contain a reference to a Julia object (a `jl_value_t*`), the result will be a reference to this object, and the object will not be copied. You must be careful in this case to ensure that the object was always visible to the garbage collector (pointers do not count, but the new reference does) to ensure the memory is not prematurely freed. Note that if the object was not originally allocated by Julia, the new object will never be finalized by Julia's garbage collector. If the `Ptr` itself is actually a `jl_value_t*`, it can be converted back to a Julia object reference by `unsafe_pointer_to_objref(ptr)`. (Julia values v can be converted to `jl_value_t*` pointers, as `Ptr{Void}`, by calling `pointer_from_objref(v)`.)

The reverse operation (writing data to a `Ptr{T}`), can be performed using `unsafe_store!(ptr, value, [index])`. Currently, this is only supported for primitive types or other pointer-free (`isbits`) immutable struct types.

Any operation that throws an error is probably currently unimplemented and should be posted as a bug so that it can be resolved.

If the pointer of interest is a plain-data array (primitive type or immutable struct), the function `unsafe_wrap(Array, ptr,dims,[own])` may be more useful. The final parameter should be true if Julia should "take ownership" of the underlying buffer and call `free(ptr)` when the returned `Array` object is finalized. If the own parameter is omitted or false, the caller must ensure the buffer remains in existence until all access is complete.

Arithmetic on the `Ptr` type in Julia (e.g. using +) does not behave the same as C's pointer arithmetic. Adding an integer to a `Ptr` in Julia always moves the pointer by some number of *bytes*, not elements. This way, the address values obtained from pointer arithmetic do not depend on the element types of pointers.

30.11 Thread-safety

Some C libraries execute their callbacks from a different thread, and since Julia isn't thread-safe you'll need to take some extra precautions. In particular, you'll need to set up a two-layered system: the C callback should only *schedule* (via Julia's event loop) the execution of your "real" callback. To do this, create a `AsyncCondition` object and wait on it:

```
cond = Base.AsyncCondition()
wait(cond)
```

The callback you pass to C should only execute a `ccall` to `:uv_async_send`, passing `cond.handle` as the argument, taking care to avoid any allocations or other interactions with the Julia runtime.

Note that events may be coalesced, so multiple calls to `uv_async_send` may result in a single wakeup notification to the condition.

30.12 More About Callbacks

For more details on how to pass callbacks to C libraries, see this blog post.

30.13 C++

For direct C++ interfacing, see the Cxx package. For tools to create C++ bindings, see the CxxWrap package.

C name	Fortran name	Standard Julia Alias	Julia Base Type
`unsigned char`	`CHARACTER`	`Cuchar`	`UInt8`
`bool` (only in C++)		`Cuchar`	`UInt8`
`short`	`INTEGER*2`, `LOGICAL*2`	`Cshort`	`Int16`
`unsigned short`		`Cushort`	`UInt16`
`int`, `BOOL` (C, typical)	`INTEGER*4`, `LOGICAL*4`	`Cint`	`Int32`
`unsigned int`		`Cuint`	`UInt32`
`long long`	`INTEGER*8`, `LOGICAL*8`	`Clong-long`	`Int64`
`unsigned long long`		`Culong-long`	`UInt64`
`intmax_t`		`Cint-max_t`	`Int64`
`uintmax_t`		`Cuint-max_t`	`UInt64`
`float`	`REAL*4i`	`Cfloat`	`Float32`
`double`	`REAL*8`	`Cdouble`	`Float64`
`complex float`	`COMPLEX*8`	`Com-plex64`	`Complex{Float32}`
`complex double`	`COM-PLEX*16`	`Com-plex128`	`Complex{Float64}`
`ptrdiff_t`		`Cp-trdiff_t`	`Int`
`ssize_t`		`Cs-size_t`	`Int`
`size_t`		`Csize_t`	`UInt`
`void`			`Void`
`void` and `[[noreturn]]` or `_Noreturn`			`Union{}`
`void*`			`Ptr{Void}`
`T*` (where T represents an appropriately defined type)			`Ref{T}`
`char*` (or `char[]`, e.g. a string)	`CHARAC-TER*N`		`Cstring` if NUL-terminated, or `Ptr{UInt8}` if not
`char**` (or `*char[]`)			`Ptr{Ptr{UInt8}}`
`jl_value_t*` (any Julia Type)			`Any`
`jl_value_t**` (a reference to a Julia Type)			`Ref{Any}`
`va_arg`			Not supported
`...` (variadic function specification)			`T...` (where T is one of the above types, variadic functions of different argument types are not supported)

C name	Standard Julia Alias	Julia Base Type
`char`	`Cchar`	`Int8` (x86, x86_64), `UInt8` (powerpc, arm)
`long`	`Clong`	`Int` (UNIX), `Int32` (Windows)
`unsigned long`	`Culong`	`UInt` (UNIX), `UInt32` (Windows)
`wchar_t`	`Cwchar_t`	`Int32` (UNIX), `UInt16` (Windows)

Chapter 31

Manejando variaciones en el Sistema Operativo

Cuando se trata con librerías de plataforma, es frecuentemente necesario proporcionar casos especiales para distintas plataformas. La variable `Sys.KERNEL` puede utilizarse para escribir estos casos especiales. Hay varias funciones con intención de hacer esto más sencillo: `is_unix`, `is_linux`, `is_apple`, `is_bsd` e `is_windows`. Ellas pueden usarse de la siguiente forma:

```
if is_windows()
    some_complicated_thing(a)
end
```

Note que `is_linux` e `is_apple` son subconjuntos mutuamente exclusivos de `is_unix`. Adicionalmente, existe una macro `@static` que hace posible usar estas funciones para ocultar código inválido condicionalmente, como demuestran los siguientes ejemplos:

Bloques simples:

```
ccall( (@static is_windows() ? :_fopen : :fopen), ...)
```

Bloques complejos:

```
@static if is_linux()
    some_complicated_thing(a)
else
    some_different_thing(a)
end
```

Cuando se encadenan condicionales (incluyendo if/elsif/end) `@static` debe ser repetido por cada nivel (paréntesis opcional, pero recomendado por legibilidad):

```
@static is_windows() ? :a : (@static is_apple() ? :b : :c)
```

Chapter 32

Variables de Entorno

Julia se puede configurar con varias variables de entorno, ya sea de la manera habitual del sistema operativo o de manera portátil desde Julia. Supongamos que quiere establecer la variable de entorno `JULIA_EDITOR` como `vim`, para ello escribirá `ENV["JULIA_EDITOR"] = "vim"` en el REPL para llevar a cabo este cambio en la sesión actual, o agregará lo mismo en el archivo de configuración `.juliarc.jl` en el directorio de inicio del usuario para tener un efecto permanente. El valor actual de la misma variable de entorno se determina evaluando `ENV ["JULIA_EDITOR"]`.

Las variables de entorno que usa Julia generalmente comienzan con `JULIA`. Si `Base.versioninfo` se llama con `verbose` igual a `true`, la salida mostrará una lista de variables de entorno definidas relevantes para Julia, incluidas aquellas para las cualesJULIA aparece en el nombre.

32.1 Localizaciones de fichero

JULIA_HOME

La ruta absoluta del directorio que contiene el ejecutable de Julia, que establece la variable global `Base.JULIA_HOME`. Si $JULIA_HOME no está configurado, entonces Julia determina el valorBase.JULIA_HOME en el tiempo de ejecución.

El ejecutable en sí es uno de

```
$ JULIA_HOME/julia
$ JULIA_HOME/julia-debug
```

por defecto.

La variable global `Base.DATAROOTDIR` determina una ruta relativa de `Base.JULIA_HOME` al directorio de datos asociado con Julia. Entonces el camino

```
$ JULIA_HOME / $ DATAROOTDIR / julia / base
```

determina el directorio en el que Julia inicialmente busca los archivos fuente (a través de `Base.find_source_file()`).

Del mismo modo, la variable global `Base.SYSCONFDIR` determina una ruta relativa al directorio del archivo de configuración. Entonces Julia busca un archivo `juliarc.jl` en

```
$JULIA_HOME/$SYSCONFDIR/julia/juliarc.jl
$JULIA_HOME/../etc/julia/juliarc.jl
```

por defecto (a través de `Base.load_juliarc()`).

Por ejemplo, una instalación de Linux con un ejecutable de Julia ubicado en `/bin/julia`, unDATAROOTDIR de `../share`, y unSYSCONFDIR de `../ etc` tendrá JULIA_HOME establecido en `/bin`, una ruta de búsqueda de archivo fuente de

```
/share/julia/base
```

y una ruta de búsqueda de configuración global de

```
/etc/julia/juliarc.jl
```

JULIA_LOAD_PATH

Una lista separada de rutas absolutas que se anexarán a la variable `LOAD_PATH`. (En sistemas tipo Unix, el separador de ruta es `:`; en sistemas Windows, el separador de ruta es `;`.) La variable `LOAD_PATH` es donde `Base.require` y `Base.load_in_path()` busca el código; se predetermina a las rutas absolutas

```
$JULIA_HOME/../local/share/julia/site/v$(VERSION.major).$(VERSION.minor)
$JULIA_HOME/../share/julia/site/v$(VERSION.major).$(VERSION.minor)
```

de modo que, por ejemplo, la versión 0.6 de Julia en un sistema Linux con un ejecutable de Julia en `/bin/julia` tendrá un`LOAD_PATH` predeterminado de

```
/local/share/julia/site/v0.6
/share/julia/site/v0.6
```

JULIA_PKGDIR

La ruta del directorio principal `Pkg.Dir._pkgroot()` para los repositorios de paquetes Julia específicos de la versión. Si la ruta es relativa, entonces se toma con respecto al directorio de trabajo. Si `$JULIA_PKGDIR` no está configurado, entonces`Pkg.Dir._pkgroot()`se establece por defecto en

```
$HOME/.julia
```

Entonces la localizacíon del repositorio `Pkg.dir` para una versin dada de Julia es

```
$JULIA_PKGDIR/v$(VERSION.major).$(VERSION.minor)
```

Por ejemplo, para un usuario Linux cuyo directorio home ea `/home/alice`, el directorio que contiene los repositorios de paquetes por defecto sería

```
/home/alice/.julia
```

y el repositorio de paquetes paa la versión 0.6 de Julia sería

```
/home/alice/.julia/v0.6
```

JULIA_HISTORY

El camino absoluto `Base.REPL.find_hist_file()` del fichero de historia del REPL. Si `$JULIA_HISTORY` no está fijado, encontces `Base.REPL.find_hist_file()` tiene como valor por defecto

```
$HOME/.julia_history
```

JULIA_PKGRESOLVE_ACCURACY

Un `Int` positivo que determina cuánto tiempo la subrutina de suma máxima`MaxSum.maxsum ()`del resolvedor de dependencia de paquetes `Base.Pkg.resolve` dedicará a intentar las restricciones satisfactorias antes de abandonar: este valor es por defecto 1, y los valores más grandes corresponden a mayores cantidades de tiempo.

Supongamos que el valor de `$ JULIA_PKGRESOLVE_ACCURACY` es n. Entonces

- el número de iteraciones de pre-decimación es `20*n`,
- el número de iteraciones entre los pasos de aniquilación es `10*n`, y
- en los pasos de aniquilación, como máximo se destruye uno de cada paquetes `20*n`.

32.2 Aplicaciones externas

JULIA_SHELL

La ruta absoluta del shell con el que Julia debe ejecutar comandos externos (a través de `Base.repl_cmd ()`). Se predetermina a la variable de entorno $SHELL, y vuelve a `/bin/sh` si $SHELL está desactivado.

> **Note**
>
> En Windows, esta variable de entorno se ignora y los comandos externos se ejecutan directamente.

JULIA_EDITOR

El editor devuelto por `Base.editor()` y utilizado en, por ejemplo, `Base.edit`, refiriéndose al comando del editor preferido, por ejemplo`vim`.

$JULIA_EDITOR tiene prioridad sobre$VISUAL, que a su vez tiene prioridad sobre $EDITOR. Si no se establece ninguna de estas variables de entorno, entonces el editor se considera `abierto` en Windows y OS X, o`/etc/alternatives/editor` si existe, o `emacs` de lo contrario.

> **Note**
>
> $JULIA_EDITOR *no se usa* en la determinación del editor para `Base.Pkg.edit`: esta función verifica $VISUAL y$EDITOR solo.

32.3 Paralelización

JULIA_CPU_CORES

Sobreescribe la variable global `Base.Sys.CPU_CORES`, el número de núcleos de CPU lógicos disponible.

JULIA_WORKER_TIMEOUT

Un `Float64` que establece el valor de`Base.worker_timeout()` (predeterminado:`60.0`). Esta función proporciona la cantidad de segundos que un proceso de trabajo esperará un proceso maestro para establecer una conexión antes de morir.

JULIA_NUM_THREADS

Un entero sin signo de 64 bits (`uint64_t`) que establece el número máximo de subprocesos disponiblew para Julia. Si $JULIA_NUM_THREADS excede la cantidad disponiblede núcleos de CPU físicos, el número de subprocesos se establece en la cantidad de núcleos. Si $JULIA_NUM_THREADS no es positivo o no está configurado, o si el número de núcleos de CPU no se puede determinar a través de llamadas al sistema, entonces la cantidad de hilos es establecida en 1.

JULIA_THREAD_SLEEP_THRESHOLD

Si se fija a una cadena que comienza con la subcadena insensible a mayúsculas y minúsculas `"infinite"`, entonces los hilos vivos nunva duermen. En caso contrario, $JULIA_THREAD_SLEEP_THRESHOLD es interpretado como un entero sin signo de 64 bits (`uint64_t`) y da, en nanosegundos, la cantidad de tiempo después del cual los hilos deben dormir.

JULIA_EXCLUSIVE

Si se establece en algo además de `0`, entonces la política de hilos de Julia es consistente con la ejecución en una máquina dedicada: el hilo maestro está en proc 0, y los hilos están affinitizados. De lo contrario, Julia deja que el sistema operativo maneje la política de hilos.

32.4 Formateo del REPL

Variables de entorno que determinan cómo debe formatearse la salida REPL en el terminal. En general, estas variables deben establecerse en seciencias de scape de terminal ANSI. Julia proporciona una interfaz de alto nivel con gran parte de la misma funcionalidad: ver la sección sobre Interacción con Julia.

JULIA_ERROR_COLOR

El formato `Base.error_color()` (predeterminado: rojo claro, `"\033[91m "`) que los errores deberían tener en la terminal.

JULIA_WARN_COLOR

El formato `Base.warn_color()` (preeterminado: amarillo, `"\033[93m"`) que deberían tener las advertencias en el terminal.

JULIA_INFO_COLOR

El formato `Base.info_color()` (predeterminado: cyan, `"\033[36m"`) que debería tener la info en el terminal.

JULIA_INPUT_COLOR

El formato `Base.input_color()` (predeterminado: normal, `"\033[0m"`) que debería haber en el terminal.

JULIA_ANSWER_COLOR

El formato `Base.answer_color()` (predeterminado: normal, `"\033[0m"`) que debería haber en el terminal..

JULIA_STACKFRAME_LINEINFO_COLOR

El formato `Base.stackframe_lineinfo_color()` (predeterminado: bold, `"\033[1m"`) que la línea info debería tener durante una traza de la pila en el terminal.

JULIA_STACKFRAME_FUNCTION_COLOR

El formato `Base.stackframe_function_color()` (predeterminado: bold, `"\033[1m"`) que las llamadas a funcin deberían tener durante una traza de la pila en el terminal.

32.5 Depuración y profiling

JULIA_GC_ALLOC_POOL, JULIA_GC_ALLOC_OTHER, JULIA_GC_ALLOC_PRINT

Si se establecen, estas variables de entorno toman cadenas que opcionalmente comienzan con el carácter `'r'`, seguido por una interpolación de cadenas de una lista separada por dos puntos de tres enteros de 64 bits (`int64_t`). Este triplete de enteros `a:b:c` representa la secuencia aritmética `a`, `a + b`,`a + 2*b`, ... `c`.

- Si es la ”n” vez que se ha llamado a `jl_gc_pool_alloc ()`, y n pertenece a la secuencia aritmética representada por`$ JULIA_GC_ALLOC_POOL`, entonces se forzará la recolección de elementos no utilizados.
- Si es la nth vez que`maybe_collect ()`ha sido llamado, y n pertenece a la secuencia aritmética representada por `$JULIA_GC_ALLOC_OTHER`, entonces se forzará la recolección de elementos no utilizados.
- Si es la n th vez que`jl_gc_collect()` ha sido llamado, y n pertenece a la secuencia aritmética representada por `$JULIA_GC_ALLOC_PRINT`, entonces cuenta para el número de llamadas a `jl_gc_pool_alloc()` y `maybe_collect()` están impresos.

Si el valor de la variable de entorno comienza con el carácter `'r'`, entonces el intervalo entre los eventos de recolección de basura es aleatorio.

> **Note**
>
> Estas variables de entorno solo tienen un efecto si Julia se compiló con la depuración de la recolección de basura (es decir, si `WITH_GC_DEBUG_ENV` se fija a 1 en la configuración de compilación).

JULIA_GC_NO_GENERATIONAL

Si se fija a algo por encima de `0`, entonces el recolector de basura de Julia nunca realizará barridos rápidos de memoria.

> **Note**
>
> Esta variable de entorno solo tiene un efecto si Julia se compiló con depuración de recolección de elementos no utilizados (es decir, si `WITH_GC_DEBUG_ENV` está establecido en1 en la configuración de compilación).

JULIA_GC_WAIT_FOR_DEBUGGER

Si se fija a algo por encima de `0`, entonces el recolector de basura esperará a que un depurador la enlace en lugar de abortar cuando haya un error crítico.

> **Note**
>
> Esta variable de entorno solo tiene un efecto si Julia se compiló con depuración de recolección de elementos no utilizados (es decir, si `WITH_GC_DEBUG_ENV` está establecido en1 en la configuración de compilación).

ENABLE_JITPROFILING

Si se fija a un valor por encima de `0`, entonces el compilador creará y registrará un detector de evaentos (*event listener*) para *for just-in-time (JIT) profiling*.

> **Note**
>
> Esta variable de entorno sólo tiene efecto su Julia se compiló con soporte de JIT profiling usando

- VTune™ Amplifier de Intel (`USE_INTEL_JITEVENTS` set to 1 in the build configuration), o bien
- OProfile (`USE_OPROFILE_JITEVENTS` set to 1 in the build configuration).

JULIA_LLVM_ARGS

Arguments para pasar al backend de LLVM.

> **Note**
>
> Esta variable de entorno sólo tiene efecto su Julia se compiló con `JL_DEBUG_BUILD` fijado — en particular, el ejecutable `julia-debug` siempre es compilado con esta variable.

JULIA_DEBUG_LOADING

Si se establece, entonces Julia imprime información detallada sobre la caché en el proceso de carga de `Base.require`.

Chapter 33

Embedding Julia

As we have seen in Calling C and Fortran Code, Julia has a simple and efficient way to call functions written in C. But there are situations where the opposite is needed: calling Julia function from C code. This can be used to integrate Julia code into a larger C/C++ project, without the need to rewrite everything in C/C++. Julia has a C API to make this possible. As almost all programming languages have some way to call C functions, the Julia C API can also be used to build further language bridges (e.g. calling Julia from Python or C#).

33.1 High-Level Embedding

We start with a simple C program that initializes Julia and calls some Julia code:

```
#include <julia.h>

int main(int argc, char *argv[])
{
    /* required: setup the Julia context */
    jl_init();

    /* run Julia commands */
    jl_eval_string("print(sqrt(2.0))");

    /* strongly recommended: notify Julia that the
         program is about to terminate. this allows
         Julia time to cleanup pending write requests
         and run all finalizers
    */
    jl_atexit_hook(0);
    return 0;
}
```

In order to build this program you have to put the path to the Julia header into the include path and link against `libjulia`. For instance, when Julia is installed to `$JULIA_DIR`, one can compile the above test program `test.c` with `gcc` using:

```
gcc -o test -fPIC -I$JULIA_DIR/include/julia -L$JULIA_DIR/lib test.c -ljulia $JULIA_DIR/lib/julia
     /libstdc++.so.6
```

Then if the environment variable `JULIA_HOME` is set to `$JULIA_DIR/bin`, the output `test` program can be executed.

Alternatively, look at the `embedding.c` program in the Julia source tree in the `examples/` folder. The file `ui/repl.c` program is another simple example of how to set `jl_options` options while linking against `libjulia`.

The first thing that has to be done before calling any other Julia C function is to initialize Julia. This is done by calling `jl_init`, which tries to automatically determine Julia's install location. If you need to specify a custom location, or specify which system image to load, use `jl_init_with_image` instead.

The second statement in the test program evaluates a Julia statement using a call to `jl_eval_string`.

Before the program terminates, it is strongly recommended to call `jl_atexit_hook`. The above example program calls this before returning from `main`.

Note

Currently, dynamically linking with the `libjulia` shared library requires passing the `RTLD_GLOBAL` option. In Python, this looks like:

```
>>> julia=CDLL('./libjulia.dylib',RTLD_GLOBAL)
>>> julia.jl_init.argtypes = []
>>> julia.jl_init()
250593296
```

Note

If the julia program needs to access symbols from the main executable, it may be necessary to add `-Wl,--export-dynamic` linker flag at compile time on Linux in addition to the ones generated by `julia-config.jl` described below. This is not necessary when compiling a shared library.

Using julia-config to automatically determine build parameters

The script `julia-config.jl` was created to aid in determining what build parameters are required by a program that uses embedded Julia. This script uses the build parameters and system configuration of the particular Julia distribution it is invoked by to export the necessary compiler flags for an embedding program to interact with that distribution. This script is located in the Julia shared data directory.

Example

```
#include <julia.h>

int main(int argc, char *argv[])
{
    jl_init();
    (void)jl_eval_string("println(sqrt(2.0))");
    jl_atexit_hook(0);
    return 0;
}
```

On the command line

A simple use of this script is from the command line. Assuming that `julia-config.jl` is located in `/usr/local/julia/share/julia`, it can be invoked on the command line directly and takes any combination of 3 flags:

```
/usr/local/julia/share/julia/julia-config.jl
Usage: julia-config [--cflags|--ldflags|--ldlibs]
```

If the above example source is saved in the file `embed_example.c`, then the following command will compile it into a running program on Linux and Windows (MSYS2 environment), or if on OS/X, then substitute `clang` for `gcc`.:

```
/usr/local/julia/share/julia/julia-config.jl --cflags --ldflags --ldlibs | xargs gcc
    embed_example.c
```

Use in Makefiles

But in general, embedding projects will be more complicated than the above, and so the following allows general makefile support as well – assuming GNU make because of the use of the **shell** macro expansions. Additionally, though many times `julia-config.jl` may be found in the directory `/usr/local`, this is not necessarily the case, but Julia can be used to locate `julia-config.jl` too, and the makefile can be used to take advantage of that. The above example is extended to use a Makefile:

```
JL_SHARE = $(shell julia -e 'print(joinpath(JULIA_HOME,Base.DATAROOTDIR,"julia"))')
CFLAGS   += $(shell $(JL_SHARE)/julia-config.jl --cflags)
CXXFLAGS += $(shell $(JL_SHARE)/julia-config.jl --cflags)
LDFLAGS  += $(shell $(JL_SHARE)/julia-config.jl --ldflags)
LDLIBS   += $(shell $(JL_SHARE)/julia-config.jl --ldlibs)

all: embed_example
```

Now the build command is simply `make`.

33.2 Converting Types

Real applications will not just need to execute expressions, but also return their values to the host program. `jl_eval_string` returns a `jl_value_t*`, which is a pointer to a heap-allocated Julia object. Storing simple data types like `Float64` in this way is called `boxing`, and extracting the stored primitive data is called `unboxing`. Our improved sample program that calculates the square root of 2 in Julia and reads back the result in C looks as follows:

```
jl_value_t *ret = jl_eval_string("sqrt(2.0)");

if (jl_typeis(ret, jl_float64_type)) {
    double ret_unboxed = jl_unbox_float64(ret);
    printf("sqrt(2.0) in C: %e \n", ret_unboxed);
}
else {
    printf("ERROR: unexpected return type from sqrt(::Float64)\n");
}
```

In order to check whether `ret` is of a specific Julia type, we can use the `jl_isa`, `jl_typeis`, or `jl_is_...` functions. By typing `typeof(sqrt(2.0))` into the Julia shell we can see that the return type is `Float64` (`double` in C). To convert the boxed Julia value into a C double the `jl_unbox_float64` function is used in the above code snippet.

Corresponding `jl_box_...` functions are used to convert the other way:

```
jl_value_t *a = jl_box_float64(3.0);
jl_value_t *b = jl_box_float32(3.0f);
jl_value_t *c = jl_box_int32(3);
```

As we will see next, boxing is required to call Julia functions with specific arguments.

33.3 Calling Julia Functions

While `jl_eval_string` allows C to obtain the result of a Julia expression, it does not allow passing arguments computed in C to Julia. For this you will need to invoke Julia functions directly, using `jl_call`:

```
jl_function_t *func = jl_get_function(jl_base_module, "sqrt");
jl_value_t *argument = jl_box_float64(2.0);
jl_value_t *ret = jl_call1(func, argument);
```

In the first step, a handle to the Julia function `sqrt` is retrieved by calling `jl_get_function`. The first argument passed to `jl_get_function` is a pointer to the `Base` module in which `sqrt` is defined. Then, the double value is boxed using `jl_box_float64`. Finally, in the last step, the function is called using `jl_call1`. `jl_call0`, `jl_call2`, and `jl_call3` functions also exist, to conveniently handle different numbers of arguments. To pass more arguments, use `jl_call`:

```
jl_value_t *jl_call(jl_function_t *f, jl_value_t **args, int32_t nargs)
```

Its second argument `args` is an array of `jl_value_t*` arguments and `nargs` is the number of arguments.

33.4 Memory Management

As we have seen, Julia objects are represented in C as pointers. This raises the question of who is responsible for freeing these objects.

Typically, Julia objects are freed by a garbage collector (GC), but the GC does not automatically know that we are holding a reference to a Julia value from C. This means the GC can free objects out from under you, rendering pointers invalid.

The GC can only run when Julia objects are allocated. Calls like `jl_box_float64` perform allocation, and allocation might also happen at any point in running Julia code. However, it is generally safe to use pointers in between `jl_...` calls. But in order to make sure that values can survive `jl_...` calls, we have to tell Julia that we hold a reference to a Julia value. This can be done using the JL_GC_PUSH macros:

```
jl_value_t *ret = jl_eval_string("sqrt(2.0)");
JL_GC_PUSH1(&ret);
// Do something with ret
JL_GC_POP();
```

The JL_GC_POP call releases the references established by the previous JL_GC_PUSH. Note that JL_GC_PUSH is working on the stack, so it must be exactly paired with a JL_GC_POP before the stack frame is destroyed.

Several Julia values can be pushed at once using the JL_GC_PUSH2 , JL_GC_PUSH3 , and JL_GC_PUSH4 macros. To push an array of Julia values one can use the JL_GC_PUSHARGS macro, which can be used as follows:

```
jl_value_t **args;
JL_GC_PUSHARGS(args, 2); // args can now hold 2 `jl_value_t*` objects
args[0] = some_value;
args[1] = some_other_value;
// Do something with args (e.g. call jl_... functions)
JL_GC_POP();
```

The garbage collector also operates under the assumption that it is aware of every old-generation object pointing to a young-generation one. Any time a pointer is updated breaking that assumption, it must be signaled to the collector with the `jl_gc_wb` (write barrier) function like so:

```
jl_value_t *parent = some_old_value, *child = some_young_value;
((some_specific_type*)parent)->field = child;
jl_gc_wb(parent, child);
```

It is in general impossible to predict which values will be old at runtime, so the write barrier must be inserted after all explicit stores. One notable exception is if the `parent` object was just allocated and garbage collection was not run since then. Remember that most `jl_...` functions can sometimes invoke garbage collection.

The write barrier is also necessary for arrays of pointers when updating their data directly. For example:

```
jl_array_t *some_array = ...; // e.g. a Vector{Any}
void **data = (void**)jl_array_data(some_array);
jl_value_t *some_value = ...;
data[0] = some_value;
jl_gc_wb(some_array, some_value);
```

Manipulating the Garbage Collector

There are some functions to control the GC. In normal use cases, these should not be necessary.

Function	Description
`jl_gc_collect()`	Force a GC run
`jl_gc_enable(0)`	Disable the GC, return previous state as int
`jl_gc_enable(1)`	Enable the GC, return previous state as int
`jl_gc_is_enabled()`	Return current state as int

33.5 Working with Arrays

Julia and C can share array data without copying. The next example will show how this works.

Julia arrays are represented in C by the datatype `jl_array_t*`. Basically, `jl_array_t` is a struct that contains:

- Information about the datatype
- A pointer to the data block
- Information about the sizes of the array

To keep things simple, we start with a 1D array. Creating an array containing Float64 elements of length 10 is done by:

```
jl_value_t* array_type = jl_apply_array_type(jl_float64_type, 1);
jl_array_t* x          = jl_alloc_array_1d(array_type, 10);
```

Alternatively, if you have already allocated the array you can generate a thin wrapper around its data:

```
double *existingArray = (double*)malloc(sizeof(double)*10);
jl_array_t *x = jl_ptr_to_array_1d(array_type, existingArray, 10, 0);
```

The last argument is a boolean indicating whether Julia should take ownership of the data. If this argument is non-zero, the GC will call `free` on the data pointer when the array is no longer referenced.

In order to access the data of x, we can use `jl_array_data`:

```
double *xData = (double*)jl_array_data(x);
```

Now we can fill the array:

```
for(size_t i=0; i<jl_array_len(x); i++)
    xData[i] = i;
```

Now let us call a Julia function that performs an in-place operation on x:

```
jl_function_t *func = jl_get_function(jl_base_module, "reverse!");
jl_call1(func, (jl_value_t*)x);
```

By printing the array, one can verify that the elements of x are now reversed.

Accessing Returned Arrays

If a Julia function returns an array, the return value of `jl_eval_string` and `jl_call` can be cast to a `jl_array_t*`:

```
jl_function_t *func  = jl_get_function(jl_base_module, "reverse");
jl_array_t *y = (jl_array_t*)jl_call1(func, (jl_value_t*)x);
```

Now the content of y can be accessed as before using `jl_array_data`. As always, be sure to keep a reference to the array while it is in use.

Multidimensional Arrays

Julia's multidimensional arrays are stored in memory in column-major order. Here is some code that creates a 2D array and accesses its properties:

```
// Create 2D array of float64 type
jl_value_t *array_type = jl_apply_array_type(jl_float64_type, 2);
jl_array_t *x  = jl_alloc_array_2d(array_type, 10, 5);

// Get array pointer
double *p = (double*)jl_array_data(x);
// Get number of dimensions
int ndims = jl_array_ndims(x);
// Get the size of the i-th dim
size_t size0 = jl_array_dim(x,0);
size_t size1 = jl_array_dim(x,1);

// Fill array with data
for(size_t i=0; i<size1; i++)
    for(size_t j=0; j<size0; j++)
        p[j + size0*i] = i + j;
```

Notice that while Julia arrays use 1-based indexing, the C API uses 0-based indexing (for example in calling `jl_array_dim`) in order to read as idiomatic C code.

33.6 Exceptions

Julia code can throw exceptions. For example, consider:

```
jl_eval_string("this_function_does_not_exist()");
```

This call will appear to do nothing. However, it is possible to check whether an exception was thrown:

```
if (jl_exception_occurred())
    printf("%s \n", jl_typeof_str(jl_exception_occurred()));
```

If you are using the Julia C API from a language that supports exceptions (e.g. Python, C#, C++), it makes sense to wrap each call into `libjulia` with a function that checks whether an exception was thrown, and then rethrows the exception in the host language.

Throwing Julia Exceptions

When writing Julia callable functions, it might be necessary to validate arguments and throw exceptions to indicate errors. A typical type check looks like:

```
if (!jl_typeis(val, jl_float64_type)) {
    jl_type_error(function_name, (jl_value_t*)jl_float64_type, val);
}
```

General exceptions can be raised using the functions:

```
void jl_error(const char *str);
void jl_errorf(const char *fmt, ...);
```

`jl_error` takes a C string, and `jl_errorf` is called like `printf`:

```
jl_errorf("argument x = %d is too large", x);
```

where in this example x is assumed to be an integer.

Chapter 34

Paquetes

Julia tiene un administrador de paquetes incorporado para instalar la funcionalidad añadida y escrita en Julia. También puede instalar bibliotecas externas utilizando el sistema estándar de su sistema operativo para hacerlo, o compilando desde los fuentes. La lista de paquetes Julia registrados se puede encontrar en http://pkg.julialang.org. Todos los mandatos del gestor de paquetes se encuentran en el módulo Pkg, incluido en la instalación Base de Julia.

Primero revisaremos los mecanismos de la familia de mandatos de Pkg y luego brindaremos algunas pautas sobre cómo registrar sus paquetes. Asegúrese de leer la siguiente sección sobre las convenciones de nombres de paquetes, las versiones de etiquetado y la importancia de un archivo REQUIRE para cuando esté listo para agregar su código al repositorio de METADATA seleccionado.

34.1 Estado de un paquete

La función `Pkg.status()` imprime un resumen del estado de los paquetes que uno ha instalado. Inicialmente uno no tendrá paquetes instalados:

```
julia> Pkg.status()
INFO: Initializing package repository /Users/stefan/.julia/v0.6
INFO: Cloning METADATA from git://github.com/JuliaLang/METADATA.jl
No packages installed.
```

El directorio de paquetes es inicializado automáticamente la primera vez que una ejecute un mandato Pkg que asume su existencia – que incluya `Pkg.status()`. He aquí un conjunto de ejemplo no trivial de paquetes requeridos y adicionales:

```
julia> Pkg.status()
Required packages:
 - Distributions                0.2.8
 - SHA                          0.3.2
Additional packages:
 - NumericExtensions            0.2.17
 - Stats                        0.2.6
```

Estos paquetes están todos en versiones registradas, administradas por Pkg. Los paquetes pueden estar en estados más complicados, indicados por anotaciones a la derecha de la versión del paquete instalado; estos estados y anotaciones serán explicados a medida que los encontremos. Para el uso programático, `Pkg.installed()` devuelve un diccionario, donde se hacen corresponder los nombres de paquetes instalados con la versión instalada de ese paquete:

```
julia> Pkg.installed()
Dict{String,VersionNumber} with 4 entries:
"Distributions"     => v"0.2.8"
"Stats"             => v"0.2.6"
"SHA"               => v"0.3.2"
"NumericExtensions" => v"0.2.17"
```

34.2 Añadir y eliminar paquetes

El administrador de paquetes de Julia es un poco inusual ya que es declarativo en lugar de imperativo. Esto significa que uno le dice lo que quiere y el gestor descubre qué versiones instalar (o eliminar) para satisfacer esos requisitos de manera óptima, - y mínimamente. Por tanto, en lugar de instalar un paquete, simplemente lo agrega a la lista de requisitos y luego ”resuelve” lo que necesita instalar. En particular, esto significa que si algún paquete se ha instalado porque lo necesitaba una versión anterior de algo que usted quería, y una versión más nueva ya no tiene ese requisito, la actualización realmente eliminará ese paquete.

Los requisitos de paquetes están en el archivo `~/.julia/v0.6/REQUIRE`. Este archivo puede ser editado a mano y luego llamarse a `Pkg.resolve()` para instalar, actualizar o eliminar paquetes para satisfacer de manera óptima los requisitos, o puede hacer `Pkg.edit()`, que abrirá `REQUIRE` en su editor (configurado a través de las variables de entorno `EDITOR` o`VISUAL`), y luego llamará automáticamente a `Pkg.resolve()` después si es necesario. Si solo desea agregar o eliminar el requisito para un solo paquete, también puede usar los mandatos no interactivos `Pkg.add()` y `Pkg.rm()`, que agregan o eliminan un solo requisito a `REQUIRE` y luego llaman a `Pkg.resolve()`.

Puede agregarse un paquete a la lista de requisitos con la función `Pkg.add()`, y se instalará el paquete y todos los paquetes de los que depende.

```
julia> Pkg.status()
No packages installed.

julia> Pkg.add("Distributions")
INFO: Cloning cache of Distributions from git://github.com/JuliaStats/Distributions.jl.git
INFO: Cloning cache of NumericExtensions from git://github.com/lindahua/NumericExtensions.jl.git
INFO: Cloning cache of Stats from git://github.com/JuliaStats/Stats.jl.git
INFO: Installing Distributions v0.2.7
INFO: Installing NumericExtensions v0.2.17
INFO: Installing Stats v0.2.6
INFO: REQUIRE updated.

julia> Pkg.status()
Required packages:
 - Distributions                0.2.7
Additional packages:
 - NumericExtensions            0.2.17
 - Stats                        0.2.6
```

Lo que está haciendo es primero agregar `Distribuciones` a su archivo `~/.julia/v0.6/REQUIRE`:

```
$ cat ~/.julia/v0.6/REQUIRE
Distributions
```

A continuación, ejecuta `Pkg.resolve()` utilizando estos nuevos requisitos, lo que lleva a la conclusión de que el paquete `Distributions` debe instalarse ya que es obligatorio pero no está instalado. Como se dijo anteriormente, puede lograr lo mismo editando su archivo `~/.julia/v0.6/REQUIRE` a mano y luego ejecutando `Pkg.resolve()` usted mismo:

```
$ echo SHA >> ~/.julia/v0.6/REQUIRE

julia> Pkg.resolve()
INFO: Cloning cache of SHA from git://github.com/staticfloat/SHA.jl.git
INFO: Installing SHA v0.3.2

julia> Pkg.status()
Required packages:
 - Distributions                0.2.7
 - SHA                          0.3.2
Additional packages:
 - NumericExtensions            0.2.17
 - Stats                        0.2.6
```

Esto es funcionalmente equivalente a llamar a `Pkg.add("SHA")`, excepto que `Pkg.add()` no cambia `REQUIRE` hasta *después* de que la instalación haya finalizado, por lo que si hay problemas, `REQUIRE` quedará como estaba antes de llamar a `Pkg.add()`. El formato del archivo `REQUIRE` se describe en Especificación de requisitos; permite, entre otras cosas, requerir rangos específicos de versiones de paquetes.

Cuando decida que no quiere tener un paquete más, puede usar `Pkg.rm()` para eliminar el requisito del archivo `REQUIRE`:

```
julia> Pkg.rm("Distributions")
INFO: Removing Distributions v0.2.7
INFO: Removing Stats v0.2.6
INFO: Removing NumericExtensions v0.2.17
INFO: REQUIRE updated.

julia> Pkg.status()
Required packages:
 - SHA                          0.3.2

julia> Pkg.rm("SHA")
INFO: Removing SHA v0.3.2
INFO: REQUIRE updated.

julia> Pkg.status()
No packages installed.
```

Una vez más, esto es equivalente a editar el archivo `REQUIRE` para eliminar la línea con cada nombre de paquete y ejecutar `Pkg.resolve()` para actualizar el conjunto de paquetes instalados para que coincidan. Mientras que `Pkg.add()` y `Pkg.rm()` son convenientes para agregar y eliminar requisitos para un solo paquete, cuando desea agregar o eliminar paquetes múltiples, puede llamar a `Pkg.edit()` para cambiar manualmente el contenido de `REQUIRE` y luego actualizar sus paquetes en consecuencia. `Pkg.edit()` no retrotrae el contenido de `REQUIRE` si `Pkg.resolve()` falla, sino que debe ejecutar `Pkg.edit()` otra vez para corregir el contenido de los archivos uno mismo.

Debido a que el administrador de paquetes usa libgit2 internamente para administrar los repositorios git del paquete, los usuarios pueden encontrarse con problemas de protocolo (por ejemplo, si están detrás de un *firewall*) al ejecutar `Pkg.add()`. Por defecto, se accederá a todos los paquetes alojados por GitHub a través de 'https'; este valor predeterminado se puede modificar llamando a `Pkg.setprotocol!()`. El siguiente comando se puede ejecutar desde la línea de comando para decirle a git que use 'https' en lugar del protocolo 'git' cuando clona todos los repositorios, dondequiera que estén alojados:

```
git config --global url."https://".insteadOf git://
```

Sin embargo, este cambio será en todo el sistema y, por lo tanto, es preferible utilizar `Pkg.setprotocol!()`.

> **Note**
>
> Las funciones del administrador de paquetes también aceptan el sufijo `.jl` sobre los nombres de paquetes, aunque el sufijo sea eliminado internamente. Por ejemplo:

```
Pkg.add("Distributions.jl")
Pkg.rm("Distributions.jl")
```

34.3 Instalación de paquetes fuera de línea

Para las máquinas sin conexión a Internet, los paquetes se pueden instalar copiando el directorio raíz del paquete (proporcionado por `Pkg.dir()`) desde una máquina con el mismo sistema operativo y entorno.

`Pkg.add()` hace lo siguiente dentro del directorio raíz del paquete:

1. Agrega el nombre del paquete a `REQUIRE`.
2. Descarga el paquete en `.cache`, luego copia el paquete en el directorio raíz del paquete.
3. Realiza recursivamente el paso 2 contra todos los paquetes enumerados en el archivo `REQUIRE` del paquete.
4. Ejecuta `Pkg.build()`

> **Warning**
>
> Copiar paquetes instalados desde una máquina diferente es frágil para paquetes que requieren dependencias externas binarias. Dichos paquetes pueden romperse debido a diferencias en las versiones del sistema operativo, entornos de compilación y / o dependencias absolutas de rutas.

34.4 Instalar paquetes no registrados

Los paquetes de Julia son simplemente repositorios git, clonables a través de cualquiera de los protocolos que admite git, y que contienen código Julia que sigue ciertas convenciones de diseño. Los paquetes oficiales de Julia están registrados en el repositorio METADATA.jl, disponible en una ubicación conocida [1]. Los mandatos `Pkg.add()` y `Pkg.rm()` de la sección anterior interactúan con los paquetes registrados, pero el administrador de paquetes también puede instalar y trabajar con paquetes no registrados. Para instalar un paquete no registrado, usaremos `Pkg.clone(url)`, donde `url` es una URL de git desde la cual se puede clonar el paquete:

```
julia> Pkg.clone("git://example.com/path/to/Package.jl.git")
INFO: Cloning Package from git://example.com/path/to/Package.jl.git
Cloning into 'Package'...
remote: Counting objects: 22, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 22 (delta 8), reused 22 (delta 8)
Receiving objects: 100% (22/22), 2.64 KiB, done.
Resolving deltas: 100% (8/8), done.
```

Por convención, los nombres de los repositorios de Julia terminan con `.jl` (el `.git` adicional indica un repositorio "vacío" de git), lo que evita que colisionen con los repositorios de otros lenguajes, y también hace que los paquetes de Julia sean fáciles de encontrar en los motores de búsqueda. Sin embargo, cuando los paquetes están instalados en su directorio `.julia/v0.6`, la extensión es redundante, por lo que la dejamos fuera.

Si los paquetes no registrados contienen un archivo REQUIRE en la parte superior de su árbol fuente, ese archivo se usará para determinar de qué paquetes registrados depende el paquete no registrado, y se instalarán automáticamente. Los paquetes no registrados participan en la misma lógica de resolución de versiones que los paquetes registrados, por lo que las versiones de paquetes instalados se ajustarán según sea necesario para satisfacer los requisitos de los paquetes registrados y no registrados.

34.5 Actualizando Paquetes

Cuando los desarrolladores de paquetes publican nuevas versiones registradas de los paquetes que está utilizando, por supuesto, querrá las nuevas versiones brillantes. Para obtener las últimas y mejores versiones de todos sus paquetes, simplemente haga `Pkg.update()`:

When package developers publish new registered versions of packages that you're using, you will, of course, want the new shiny versions. To get the latest and greatest versions of all your packages, just do `Pkg.update()`:

```
julia> Pkg.update()
INFO: Updating METADATA...
INFO: Computing changes...
INFO: Upgrading Distributions: v0.2.8 => v0.2.10
INFO: Upgrading Stats: v0.2.7 => v0.2.8
```

El primer paso para actualizar paquetes es generar nuevos cambios en `~/.julia/v0.6/ METADATA` y ver si se ha publicado alguna nueva versión del paquete registrado. Después de esto, `Pkg.update()` intenta actualizar paquetes que están desprotegidos en una rama y no están sucios (es decir, no se han realizado cambios a los archivos rastreados por git) al extraer los cambios del repositorio en sentido ascendente del paquete. Los cambios en sentido ascendente solo se aplicarán si no es necesaria la fusión o rebase, es decir, si la rama puede ser "fast-forwarded". Si la rama no se puede reenviar rápidamente, se supone que está trabajando en ella y actualizará el repositorio usted mismo.

Finalmente, el proceso de actualización vuelve a calcular un conjunto óptimo de versiones de paquetes a tener instalado para satisfacer sus requisitos de nivel superior y los requisitos de paquetes "fijos". Un paquete es considerado corregido si es uno de los siguientes:

1. **No registrado:** el paquete no está en `METADATA` - uno lo instaló con `Pkg.clone()`.
2. **Retirado:** el repositorio del paquete está en una rama de desarrollo.
3. **Sucio:** se han realizado cambios a los archivos en el repositorio.

Si cualquiera de estos es el caso, el administrador del paquete no puede cambiar libremente la versión instalada de el paquete, por lo que sus requisitos deben ser satisfechos por cualquier otra versión del paquete que elija. La combinación de requisitos de nivel superior en `~/.julia/v0.6/REQUIRE` y el requisito de requisitos fijos los paquetes se usan para determinar qué se debe instalar.

También puede actualizar solo un subconjunto de los paquetes instalados, proporcionando argumentos a la función `Pkg.update`. En ese caso, solo los paquetes proporcionados como argumentos y sus dependencias serán actualizados:

```
julia> Pkg.update("Example")
INFO: Updating METADATA...
INFO: Computing changes...
INFO: Upgrading Example: v0.4.0 => 0.4.1
```

[1] El conjunto oficial de paquetes está en https://github.com/JuliaLang/METADATA.jl, pero los individuos y las organizaciones pueden usar fácilmente un repositorio de metadatos diferente. Esto permite controlar qué paquetes están disponibles para la i nstalación automática. Solo se pueden permitir versiones de paquete auditadas y aprobadas, y hacer paquetes privados u horquillas disponibles. Ver Repositorio METADATA personalizado para más detalles.

Este proceso de actualización parcial todavía calcula el nuevo conjunto de versiones de paquetes de acuerdo con los requisitos de nivel superior y los paquetes "fijos", pero considera además todos los demás paquetes, excepto los explícitamente proporcionados, y sus dependencias, como corregidas.

34.6 Pago, Pin y Gratis

Puede querer usar la versión `maestra` de un paquete en lugar de una de sus versiones registradas. Es posible que haya correcciones o funcionalidades que necesite y que aún no se hayan publicado en ninguna versión registrada, o puede que sea un desarrollador del paquetes y necesite realizar cambios en `master` o en alguna otra rama de desarrollo. En tales casos, puede hacer `Pkg.checkout(pkg)` para verificar la rama `master` depkg o `Pkg.checkout(pkg, branch)` para verificar alguna otra rama:

```
julia> Pkg.add("Distributions")
INFO: Installing Distributions v0.2.9
INFO: Installing NumericExtensions v0.2.17
INFO: Installing Stats v0.2.7
INFO: REQUIRE updated.

julia> Pkg.status()
Required packages:
 - Distributions                0.2.9
Additional packages:
 - NumericExtensions            0.2.17
 - Stats                        0.2.7

julia> Pkg.checkout("Distributions")
INFO: Checking out Distributions master...
INFO: No packages to install, update or remove.

julia> Pkg.status()
Required packages:
 - Distributions                0.2.9+             master
Additional packages:
 - NumericExtensions            0.2.17
 - Stats                        0.2.7
```

Inmediatamente después de instalar `Distributions` con `Pkg.add()` está en la versión registrada más reciente actual - `0.2.9` en el momento de escribir esto. Luego, después de ejecutar `Pkg.checkout("Distributions")`, puede ver en la salida de `Pkg.status()` que `Distributions` está en una versión no registrada mayor que `0.2.9`, indicado por el número de "pseudo-versión" `0.2.9+`.

Cuando compra una versión no registrada de un paquete, la copia del archivo `REQUIRE` en el repositorio del paquete tiene prioridad sobre cualquier requisito registrado en `METADATA`, por lo que es importante que los desarrolladores mantengan este archivo exacto y actualizado, lo que refleja los requisitos reales de la versión actual del paquete. Si el archivo `REQUIRE` en el repositorio del paquete es incorrecto o falta, las dependencias se pueden eliminar cuando se desprotege el paquete. Este archivo también se usa para completar las versiones del paquete publicadas recientemente si utiliza la API que `Pkg` proporciona para esto (que se describe a continuación).

Cuando decide que ya no desea tener un paquete desprotegido en una rama, puede "liberarlo" de nuevo al control del administrador de paquetes con `Pkg.free(pkg)`:

```
julia> Pkg.free("Distributions")
INFO: Freeing Distributions...
INFO: No packages to install, update or remove.
```

```
julia> Pkg.status()
Required packages:
 - Distributions                0.2.9
Additional packages:
 - NumericExtensions            0.2.17
 - Stats                        0.2.7
```

Después de esto, dado que el paquete está en una versión registrada y no en una sucursal, su versión se actualizará a medida que se publiquen nuevas versiones registradas del paquete.

Si desea fijar un paquete en una versión específica para que llamar `Pkg.update()` no cambie la versión en la que está el paquete, puede usar `Pkg.pin()` función:

```
julia> Pkg.pin("Stats")
INFO: Creating Stats branch pinned.47c198b1.tmp

julia> Pkg.status()
Required packages:
 - Distributions                0.2.9
Additional packages:
 - NumericExtensions            0.2.17
 - Stats                        0.2.7              pinned.47c198b1.tmp
```

Después de esto, el paquete `Stats` permanecerá anclado en la versión`0.2.7` - o más específicamente, en el `commit 47c198b1`, pero como las versiones están asociadas permanentemente a un hash git dado, esto es lo mismo. `Pkg.pin()` funciona creando una rama descartable para la confirmación a la que desea fijar el paquete y luego verificando esa ramificación. De forma predeterminada, fija un paquete en la confirmación actual, pero puede elegir una versión diferente pasando un segundo argumento:

```
julia> Pkg.pin("Stats",v"0.2.5")
INFO: Creating Stats branch pinned.1fd0983b.tmp
INFO: No packages to install, update or remove.

julia> Pkg.status()
Required packages:
 - Distributions                0.2.9
Additional packages:
 - NumericExtensions            0.2.17
 - Stats                        0.2.5              pinned.1fd0983b.tmp
```

Ahora el paquete `Stats` está anclado en `commit 1fd0983b`, que corresponde a la versión `0.2.5`. Cuando decides ”desanclar” un paquete y dejar que el administrador de paquetes lo actualice nuevamente, puedes usar `Pkg.free()` como lo harías para deshacerte de cualquier rama:

```
julia> Pkg.free("Stats")
INFO: Freeing Stats...
INFO: No packages to install, update or remove.

julia> Pkg.status()
Required packages:
 - Distributions                0.2.9
Additional packages:
 - NumericExtensions            0.2.17
 - Stats                        0.2.7
```

Después de esto, el paquete `Stats` es gestionado nuevamente por el administrador del paquete, y las futuras llamadas a `Pkg.update()` lo actualizarán a versiones más nuevas cuando se publiquen. La rama descartable `pinned.1fd0983b.tmp` permanece en el repositorio local de`Stats`, pero como las ramas de git son extremadamente livianas, esto realmente no importa; si desea limpiarlos, puede acceder al repositorio y eliminar esas ramas [2].

34.7 Repositorio METADATA Personalizado

Por defecto, Julia supone que utilizará el repositorio oficial METADATA.jl para descargar e instalar paquetes. También podemos proporcionar una ubicación de repositorio de metadatos diferente. Un enfoque común es mantener nuestra rama `metadata-v2` actualizada con la rama oficial de Julia y agregar otra rama con sus paquetes personalizados. Puede inicializar su repositorio de metadatos local utilizando esa ubicación y rama personalizadas y luego volver a establecer la base de nuestra rama personalizada con la rama oficial `metadata-v2`. Para utilizar un repositorio y una sucursal personalizados, utilice el siguiente mandato:

```
julia> Pkg.init("https://me.example.com/METADATA.jl.git", "branch")
```

El argumento de la rama es opcional y se predetermina a `metadata-v2`. Una vez inicializado, un archivo llamado `META_BRANCH` en su ruta `~/.julia/vX.Y/` hará un seguimiento de la rama con la que se inicializó su repositorio METADATA. Si desea cambiar de rama, necesitará modificar el archivo `META_BRANCH` directamente (¡tenga cuidado!) O elimine el directorio`vX.Y` y reinicie su repositorio METADATA utilizando el mandato `Pkg.init`.

[2] Los paquetes que no están en las ramas también se marcarán como sucios si realiza cambios en el repositorio, pero eso es menos común.

Chapter 35

Desarrollo de Paquetes

El administrador de paquetes de Julia está diseñado para que cuando tenga un paquete instalado, ya pueda ver su código fuente y el historial completo de desarrollo. También puede realizar cambios en los paquetes, enviarlos por git y contribuir fácilmente a las correcciones y mejoras en el flujo ascendente. Del mismo modo, el sistema está diseñado para que, si desea crear un nuevo paquete, la forma más sencilla de hacerlo sea dentro de la infraestructura proporcionada por el administrador de paquetes.

35.1 Initial Setup

Dado que los paquetes son repositorios de git, antes de realizar cualquier desarrollo de paquete, tenemos que fijar los siguientes ajustes de configuración de git global estándar:

```
$ git config --global user.name "FULL NAME"
$ git config --global user.email "EMAIL"
```

donde `FULL NAME` es su nombre completo actual (se permiten espacios entre las comillas dobles) y`EMAIL` es su dirección de correo electrónico real. Aunque no es necesario usar GitHub para crear o publicar paquetes de Julia, la mayoría de los paquetes de Julia al momento de escribir esto están alojados en GitHub y el administrador de paquetes sabe cómo formatear correctamente las URL de origen. y de lo contrario trabajar con el servicio sin problemas. Le recomendamos que cree una cuenta gratuita en GitHub y luego haga lo siguiente:

```
$ git config --global github.user "USERNAME"
```

donde `USERNAME` es nuestro nombre real de usuario en GitHub. Una vez que hace esto, el administrador de paquetes reconoce el nombre de usuario GitHub y puede configurar las cosas en consecuencia. También debemos cargar nuestra clave pública SSH a GitHub y configurar un agente SSH en nuestra máquina de desarrollo para que pueda realizar cambios con una molestia mínima. En el futuro, haremos que este sistema sea extensible y admitiremos otras opciones comunes de alojamiento git como BitBucket y permitiremos a los desarrolladores elegir su favorito. Como las funciones de desarrollo del paquete se han movido al paquete PkgDev, debe ejecutar `Pkg.add("PkgDev"); import PkgDev` para acceder a las funciones que comienzan con `PkgDev` en el documento siguiente.

35.2 Hacer cambios a un paquete existente

Cambios en la Documentación

Si desea mejorar la documentación en línea de un paquete, el enfoque más fácil (al menos para pequeños cambios) es utilizar la funcionalidad de edición en línea de GitHub. Primero, vaya a la ”página de inicio” de GitHub del repositorio, busque el archivo (por ejemplo, `README.md`) dentro de la estructura de carpetas del repositorio y haga clic en él. Verá el contenido que se muestra, junto con un pequeño icono de ”lápiz” en la esquina superior derecha. Al hacer clic en ese

icono, se abre el archivo en modo de edición. Realice los cambios, escriba un breve resumen que describa los cambios que desea realizar (este es su *mensaje de confirmación*), y luego presione ”Proponer cambio de archivo”. Sus cambios serán enviados para su consideración por el propietario(s) del paquete y sus colaboradores.

Para cambios de documentación más grandes, y especialmente aquellos que espera tener que actualizar en respuesta a los comentarios, puede que le resulte más fácil utilizar el procedimiento para los cambios de código que se describe a continuación.

Caambios en el Código

Resumen Ejecutivo

Aquí suponemos que ya ha configurado git en su máquina local y tiene una cuenta de GitHub (consulte más arriba). Imaginemos que está solucionando un error en el paquete `Images`:

```
Pkg.checkout("Images")           # check out the master branch
<here, make sure your bug is still a bug and hasn't been fixed already>
cd(Pkg.dir("Images"))
;git checkout -b myfixes         # create a branch for your changes
<edit code>                      # be sure to add a test for your bug
Pkg.test("Images")               # make sure everything works now
;git commit -a -m "Fix foo by calling bar"   # write a descriptive message
using PkgDev
PkgDev.submit("Images")
```

La última línea le presentará un enlace para enviar una solicitud de extracción para incorporar sus cambios.

Descripción Detallada

Si desea corregir un error o agregar una nueva funcionalidad, desea poder probar los cambios antes de enviarlos para su consideración. También debe tener una manera fácil de actualizar su propuesta en respuesta a los comentarios del propietario del paquete. En consecuencia, en este caso, la estrategia es trabajar localmente en su propia máquina; una vez que esté satisfecho con sus cambios, los envía para su consideración. Este proceso se llama *solicitud de extracción* porque usted está solicitando ”extraer” sus cambios en el repositorio principal del proyecto. Debido a que el repositorio en línea no puede ver el código en su máquina privada, primero *envía* sus cambios a una ubicación visible públicamente, su propio *fork* en línea del paquete (alojado en su propia cuenta personal de GitHub).

Supongamos que ya tiene instalado el paquete `Foo`. En la siguiente descripción, todo lo que comience con `Pkg` o`PkgDev` debe escribirse en el prompt de Julia; cualquier cosa que comience con `git` debe escribirse en modo de shell de julia (o usando el shell que viene con su sistema operativo). Dentro de Julia, puedes combinar estos dos modos:

```
julia> cd(Pkg.dir("Foo"))          # go to Foo's folder

shell> git command arguments...    # command will apply to Foo
```

Ahora supongamos que está listo para hacer algunos cambios en `Foo`. Si bien hay varios enfoques posibles, aquí hay uno que se utiliza ampliamente:

- Desde el prompt de Julia, escriba `Pkg.checkout("Foo")`. Esto garantiza que está ejecutando el último código (la rama `master`), en lugar de cualquier copia de la ”versión oficial” que haya instalado. (Si planea corregir un error, en este punto es una buena idea verificar nuevamente si el error ya ha sido corregido por otra persona. Si lo ha hecho, puede solicitar que se etiquete un nuevo lanzamiento oficial para que la corrección se distribuya al resto de la comunidad). Si recibe un error `Foo is dirty, bailing`, consulte Paquetes sucios a continuación.

- Crea una rama para tus cambios: navega a la carpeta del paquete (la que Julia informa desde `Pkg.dir("Foo")`) y (en modo shell) crea una nueva rama usando `git checkout -b <newbranch>`, donde `<newbranch>` podría ser un nombre descriptivo (por ejemplo,`fixbar`). Al crear una rama, se asegura de que pueda moverse fácilmente entre su nuevo trabajo y la rama actual 'principal' (consulte https://git-scm.com/book/es/v2/Git-Branching-Branches-in-a-Nutshell).

  Si olvida hacer este paso hasta que haya realizado algunos cambios, no se preocupe: consulte más detalles sobre la bifurcación a continuación.

- Haz tus cambios. Ya sea para corregir un error o agregar una nueva funcionalidad, en la mayoría de los casos tu cambio debería incluir actualizaciones para las carpetas `src/` y `test/`. Si estás solucionando un error, agrega un ejemplo mínimo que demuestre el error (en el código actual) al conjunto de pruebas; al contribuir con una prueba para el error, te aseguras de que el error no vuelva a aparecer accidentalmente en algún momento posterior debido a otros cambios. Si agregas una nueva funcionalidad, la creación de pruebas le demuestra al propietario del paquete que te has asegurado de que el código funcione según lo previsto.

- Ejecuta las pruebas del paquete y asegúrate de que pasen. Hay varias formas de ejecutar las pruebas:

  - Desde Julia, ejecuta `Pkg.test("Foo")`: esto ejecutará tus pruebas en un proceso separado (nuevo) `julia`.
  - Desde Julia, `include("runtests.jl")` desde la carpeta `test/` del paquete (es posible que el archivo tenga un nombre diferente, busque uno que ejecute todas las pruebas): esto te permite ejecutar las pruebas repetidamente en la misma sesión sin volver a cargar todo el código del paquete; para paquetes que tardan un poco en cargarse, esto puede ser mucho más rápido. Con este enfoque, debes hacer un trabajo adicional para realizar cambios en el código del paquete.
  - Desde el shell, ejecute `julia ../test / runtests.jl` desde dentro de la carpeta`src /`del paquete.

- Confirma tus cambios: consulte https://git-scm.com/book/es/v2/Git-Basics-Recording-Changes-to-the-Repository.

- Envía tus cambios: desde el prompt de Julia, escribe `PkgDev.submit("Foo")`. Esto impulsará los cambios a la bifurcacion de GitHub y la creará si aún no existe. (Si encuentras un error, asegúrate de haber configurado tus claves SSH.) Julia le dará un hipervínculo; abra ese enlace, edite el mensaje y luego haga clic en "enviar". En ese momento, se notificará al propietario del paquete de sus cambios y podrá iniciar el debate. (Si te sientes cómodo con git, también puedes hacer estos pasos manualmente desde el shell).

- El propietario del paquete puede sugerir mejoras adicionales. Para responder a esas sugerencias, puede actualizar fácilmente la solicitud de extracción (esto solo funciona para cambios que aún no se han fusionado, para solicitudes de extracción fusionadas, realice nuevos cambios iniciando una nueva rama):

  - Si ha cambiado ramas mientras tanto, asegúrese de volver a la misma rama con `git checkout fixbar`

  (del modo shell) o `Pkg.checkout("Foo", "fixbar")` (del prompt de Julia).

  - Como arriba, haga sus cambios, ejecute las pruebas y comprometa sus cambios.
  - Desde el shell, escribe `git push`. Esto agregará sus nuevas confirmaciones a la misma solicitud de extracción; debería verlos aparecer automáticamente en la página que contiene la discusión de su solicitud de extracción.

  Un posible tipo de cambio que el propietario puede solicitar es que elimine sus compromisos. Ver Squashing a continuación.

Paquetes Sucios

Si no se pueden cambiar las ramas porque el administrador del paquete se queja de que su paquete está sucio, significa que tiene algunos cambios que no se han confirmado (enviado). Desde el shell, use `git diff` para ver cuáles son estos cambios; puede descartarlos (`git checkout changedfile.jl`) o confirmarlos antes de cambiar de rama. Si no puede resolver los problemas manualmente, como último recurso, puede eliminar toda la carpeta `"Foo"` y reinstalar una nueva copia con `Pkg.add("Foo")`. Naturalmente, esto borra cualquier cambio que haya realizado.

Haciendo una Rama *post hoc*

Especialmente para los recién llegados a git, uno a menudo se olvida de crear una nueva rama hasta después de que ya se hayan hecho algunos cambios. Si todavía no has organizado ni comprometido tus cambios, puedes crear una nueva rama con `git checkout -b <newbranch>` como siempre - git te mostrará amablemente que algunos archivos han sido modificados y creará la nueva rama para ti. *Sus cambios aún no se han comprometido a esta nueva rama*, por lo que las reglas de trabajo normales aún se aplican.

Sin embargo, si ya has hecho un commit a `master` pero deseas volver al `master` oficial (llamado `origin/master`), utiliza el siguiente procedimiento:

- Crea una nueva rama. Esta rama mantendrá tus cambios.
- Asegúrate de que todo esté comprometido con esta rama.
- `git checkout master`. Si esto no funciona, *no* continúes hasta que hayas resuelto los problemas, o puedes perder los cambios.
- *Restablece* `master` (tu rama actual) a un estado anterior con `git reset --hard origin/master` (ver https://git-scm.com/blog/2011/07/11/reset.html).

Esto requiere un poco más de familiaridad con git, por lo que es mucho mejor tener el hábito de crear una rama desde el principio.

Squashing and rebasing

Dependiendo de los gustos del propietario(s) del paquete, él podrá pedirte que ”squash” tus compromisos. Esto es especialmente probable si el cambio es bastante simple, pero su historial de compromiso se ve así:

```
WIP: add new 1-line whizbang function (currently breaks package)
Finish whizbang function
Fix typo in variable name
Oops, don't forget to supply default argument
Split into two 1-line functions
Rats, forgot to export the second function
...
```

Esto entra en el territorio del uso de git más avanzado, y se te anima a leer un poco (https://git-scm.com/book/en/v2/Git-Branching-Rebasing). Sin embargo, un breve resumen del procedimiento es el siguiente:

- Para protegerse del error, comienza desde tu rama `fixbar` y crea una nueva rama con`git checkout -b fixbar_backup`. Como has comenzado desde `fixbar`, esta será una copia. Ahora vuelve a la que intentas modificar con `git checkout fixbar`.
- Desde el shell, escribe `git rebase -i origin/master`.

- Para combinar confirmaciones, cambia `pick` por `squash` (para opciones adicionales, consulta otras fuentes). Guarda el archivo y cierre la ventana del editor.
- Edita el mensaje de confirmación combinado.

Si la operación de rebase funciona mal, puedes volver al principio para intentarlo de nuevo de esta manera:

```
git checkout fixbar
git reset --hard fixbar_backup
```

Ahora supongamos que has realizado la operación de rebase con éxito. Como el repositorio `fixbar` ahora se ha separado del que está en tu fork GitHub, vas a tener que hacer un *force push*:

- Para que sea fácil referirse a tu rama GitHub, cree un ”manejador” para ella con `git remote add myfork https://github.com/myaccount/Foo.jl.git`, donde la URL proviene de la ”URL de clonación” en la página de tu bifurcación de GitHub.
- Para que sea fácil referirse a tu rama GitHub, cree un ”manejador” para ella con `git remote add myfork https://github.com/myaccount/Foo.jl.git`,
- Fuerza el push a tu bifurcación con `git push myfork+fixbar`. El + indica que esto debería reemplazar la rama `fixbar` encontrada en`myfork`.

35.3 Creando un nuevo Paquete

REQUIRE habla por sí mismo

Deberías tener un archivo `REQUIRE` en tu repositorio de paquetes, con una directiva mínima de la versión de Julia que esperas que los usuarios ejecuten para que el paquete funcione. Poniendo un piso en qué versión de Julia apoya tu paquete se hace simplemente agregando `julia 0.x` en este archivo. Si bien esta línea es parcialmente informativa, también tiene la consecuencia de si `Pkg.update()` actualizará el código que se encuentra en los directorios de versiones `.julia`. No actualizará el código encontrado en los directorios de versiones debajo del piso de lo que se especifica en su `REQUIRE`.

A medida que la versión de desarrollo `0.y` madura, es posible que la utilices con más frecuencia y desees que su paquete la admita. Ten cuidado, la rama de desarrollo de Julia es ”tierra de la rotura”, y puede esperar que haya cosas que se rompan. Cuando vayas a arreglar lo que rompió tu paquete en la rama de desarrollo `0.y`, probablemente encontrarás que acaba de romper tu paquete en la versión estable.

Hay un mecanismo que se encuentra en el paquete Compat que le permitirá admitir tanto la versión estable como los cambios de última hora que se encuentran en la versión de desarrollo. Si decide utilizar esta solución, deberá agregar `Compat` a su archivo`REQUIRE`. En este caso, todavía tendrá `julia 0.x` en su`REQUIRE`. La `x` es la versión de piso de lo que su paquete admite.

Es posible que tampoco tengas interés en apoyar la versión de desarrollo de Julia. Del mismo modo que puede agregar un piso a la versión que espera que tengan los usuarios, puede establecer un límite superior. En este caso, pondría `julia 0.x 0.y-` en su archivo `REQUIRE`. El `-` al final del número de versión se refiere a las versiones preliminares de esa versión específica desde el primer compromiso. Al establecerlo como techo, quiere decir que el código es compatible con todo pero no incluye la versión de techo.

Otra situación es que está escribiendo la mayor parte del código para su paquete con Julia `0.y` y no desea admitir la versión estable actual de Julia. Si elige hacer esto, simplemente agregue `julia 0.y-` a su `REQUIRE`. Solo recuerda cambiar `julia 0.y-` a `julia 0.y` en tu archivo`REQUIRE` una vez que `0.y` sea lanzado oficialmente. Si no edita el dash cruft, estás sugiriendo que admita tanto el desarrollo como las versiones estables del mismo número de versión. Eso sería una locura. Consulte la Especificación de requisitos para obtener el formato completo de `REQUIRE`.

Por último, en muchos casos puede necesitar paquetes adicionales para las pruebas. Paquetes adicionales que solo son necesarios para las pruebas deben especificarse en el archivo `test/REQUIRE`. Este archivo `REQUIRE` tiene la misma especificación que el archivo `REQUIRE` estándar.

Líneas Guía para Nombrar un Paquete

Los nombres de los paquetes deben ser sensatos para la mayoría de los usuarios de Julia, *incluso para aquellos que no son expertos en el dominio*. Cuando envíes tu paquete a METADATA, puedes esperar un poco de ida y vuelta sobre el nombre del paquete con tus colaboradores, especialmente si es ambiguo o puede confundirse con algo diferente de lo que es. Durante este intervalo de tiempo, no es raro obtener una variedad de *diferentes* sugerencias de nombres. Sin embargo, estas son solo sugerencias, con la intención de mantener un espacio de nombres ordenado en el repositorio de METADATA seleccionado. Como este repositorio pertenece a toda la comunidad, es probable que haya algunos colaboradores a los que les importe el nombre de su paquete. Aquí hay algunas pautas a seguir para nombrar su paquete:

1. Evita la jerga. En particular, evita los acrónimos a menos que haya una mínima posibilidad de confusión.
   - Está bien decir `USA` si estás hablando de USA.
   - No está bien decir `PMA`, incluso si estás hablando de una actitud mental positiva.
2. Evite usar `Julia` en el nombre de su paquete.
   - Por lo general, es claro, por contexto y para sus usuarios, que el paquete es un paquete Julia.
   - Tener a Julia en el nombre puede implicar que el paquete está conectado a, o avalado por, colaboradores del propio lenguaje Julia.
3. Los paquetes que proporcionan la mayor parte de su funcionalidad en asociación con un nuevo tipo deberían tener nombres pluralizados.
   - `DataFrames` proporciona el tipo `DataFrame`.
   - `BloomFilters` proporciona el tipo `BloomFilter`.
   - Por el contrario, `JuliaParser` no proporciona ningún tipo nuevo, sino una nueva funcionalidad en la función `JuliaParser.parse()`.
4. Err del lado de la claridad, incluso si la claridad te parece larga.
   - `RandomMatrices` es un nombre menos ambiguo que `RndMat` o `RMT`, aunque estos últimos sean más cortos.
5. Un nombre menos sistemático puede adaptarse a un paquete que implemente uno de varios enfoques posibles para su dominio.
   - Julia no tiene un solo paquete completo de graficación. En cambio, `Gadfly`,`PyPlot`, `Winston` y otros paquetes implementan cada uno un enfoque único basado en una filosofía de diseño particular.
   - Por el contrario, `SortingAlgorithms` proporciona una interfaz coherente para usar muchos sistemas bien establecidos de algoritmos de clasificación.
6. Los paquetes que envuelven bibliotecas externas o programas deben tener el nombre de esas bibliotecas o programas.
   - `CPLEX.jl` envuelve la biblioteca `CPLEX`, -que se puede identificar fácilmente en una búsqueda web.
   - `MATLAB.jl` proporciona una interfaz para llamar al motor de MATLAB desde dentro de Julia.

Generando el paquete

Supongamos que quiere crear un nuevo paquete de Julia llamado `FooBar`. Para comenzar, haga `PkgDev.generate(pkg,license)` donde `pkg` es el nuevo nombre del paquete `ylicense` es el nombre de una licencia que el generador de paquetes conoce:

```
julia> PkgDev.generate("FooBar","MIT")
INFO: Initializing FooBar repo: /Users/stefan/.julia/v0.6/FooBar
INFO: Origin: git://github.com/StefanKarpinski/FooBar.jl.git
INFO: Generating LICENSE.md
INFO: Generating README.md
INFO: Generating src/FooBar.jl
INFO: Generating test/runtests.jl
INFO: Generating REQUIRE
INFO: Generating .travis.yml
INFO: Generating appveyor.yml
INFO: Generating .gitignore
INFO: Committing FooBar generated files
```

Esto crea el directorio `~/.julia/v0.6/FooBar`, lo inicializa como un repositorio de git, genera un grupo de archivos que todos los paquetes deben tener y los envía al repositorio:

```
$ cd ~/.julia/v0.6/FooBar && git show --stat

commit 84b8e266dae6de30ab9703150b3bf771ec7b6285
Author: Stefan Karpinski <stefan@karpinski.org>
Date:   Wed Oct 16 17:57:58 2013 -0400

    FooBar.jl generated files.

        license: MIT
        authors: Stefan Karpinski
        years:   2013
        user:    StefanKarpinski

    Julia Version 0.3.0-prerelease+3217 [5fcfb13*]

 .gitignore       |  2 ++
 .travis.yml      | 13 +++++++++++++
 LICENSE.md       | 22 ++++++++++++++++++++++
 README.md        |  3 +++
 REQUIRE          |  1 +
 appveyor.yml     | 34 ++++++++++++++++++++++++++++++++++
 src/FooBar.jl    |  5 +++++
 test/runtests.jl |  5 +++++
 8 files changed, 85 insertions(+)
```

Por el momento, el administrador del paquete conoce la licencia "Expat" del MIT, indicada por `"MIT"`, la licencia simplificada BSD, indicada por `"BSD"`, y la versión 2.0 de la licencia del software Apache, indicada por `"ASL"`. Si desea utilizar una licencia diferente, puede solicitarnos que la agreguemos al generador de paquetes, o simplemente seleccione una de estas tres y luego modifique el archivo `~/.julia/v0.6/PACKAGE/LICENSE.md` después de haberlo generado.

Si creó una cuenta de GitHub y configuró git para saber sobre ello, `PkgDev.generate()` establecerá una URL de origen adecuada para usted. También generará automáticamente un archivo `.travis.yml` para usar el servicio de prueba automatizado Travis, y un archivo `appveyor.yml` para usar AppVeyor. Deberá habilitar las pruebas en los

sitios web de Travis y AppVeyor para su repositorio de paquetes, pero una vez que lo haya hecho, ya tendrá pruebas de funcionamiento. Por supuesto, todas las pruebas predeterminadas hacen es verificar que `usando FooBar` en Julia funciona.

Cargando ficheros estáticos No-Julia

Si su código de paquete necesita cargar archivos estáticos que no son código Julia, p. una biblioteca externa o archivos de datos, y se encuentran dentro del directorio del paquete, use la macro `@__DIR__` para determinar el directorio del archivo fuente actual. Por ejemplo, si `FooBar/src/FooBar.jl` necesita cargar `FooBar/data/foo.csv`, use el siguiente código:

```
datapath = joinpath(@__DIR__, "..", "data")
foo = readcsv(joinpath(datapath, "foo.csv"))
```

Haciendo Disponible tu Paquete

Una vez que haya realizado algunos commits y esté satisfecho con el funcionamiento de `FooBar`, es posible que desee conseguir que otras personas lo prueben. Primero tendrá que crear el repositorio remoto e insertar su código; todavía no hacemos esto automáticamente por usted, pero lo haremos en el futuro y no es demasiado difícil de entender [3]. Una vez que haya hecho esto, dejar que la gente pruebe su código es tan simple como enviarles la URL del repositorio publicado, en este caso:

```
git://github.com/StefanKarpinski/FooBar.jl.git
```

Para su paquete, será su nombre de usuario GitHub y el nombre de su paquete, pero se entiende la idea. Las personas a las que envíes esta URL pueden usar `Pkg.clone()` para instalar el paquete y probarlo:

```
julia> Pkg.clone("git://github.com/StefanKarpinski/FooBar.jl.git")
INFO: Cloning FooBar from git@github.com:StefanKarpinski/FooBar.jl.git
```

Tagging and Publishing Your Package

> **Tip**
>
> Si aloja su paquete en GitHub, puede usar la integración attobot para gestionar el registro, etiquetado y publicación de paquetes.

Una vez que haya decidido que `FooBar` está listo para registrarse como paquete oficial, puede agregarlo a su copia local de `METADATA` usando `PkgDev.register()`:

```
julia> PkgDev.register("FooBar")
INFO: Registering FooBar at git://github.com/StefanKarpinski/FooBar.jl.git
INFO: Committing METADATA for FooBar
```

Esto crea un commit en el repositorio `~/.julia/v0.6/METADATA`:

[3] Se recomienda encarecidamente instalar y usar la herramienta "hub" de GitHub. Esta herramienta permite hacer cosas como ejecutar `hub create` en el repositorio del paquete y hacer que se cree automáticamente a través de la API de GitHub.

```
$ cd ~/.julia/v0.6/METADATA && git show

commit 9f71f4becb05cadacb983c54a72eed744e5c019d
Author: Stefan Karpinski <stefan@karpinski.org>
Date:   Wed Oct 16 18:46:02 2013 -0400

    Register FooBar

diff --git a/FooBar/url b/FooBar/url
new file mode 100644
index 0000000..30e525e
--- /dev/null
+++ b/FooBar/url
@@ -0,0 +1 @@
+git://github.com/StefanKarpinski/FooBar.jl.git
```

Sin embargo, este compromiso sólo es visible localmente. Para que sea visible para la comunidad Julia, debe fusionar su `METADATA` local en el repositorio oficial. El comando `PkgDev.publish()` bifurca el repositorio `METADATA` en GitHub, inserta los cambios en su bifurcación, y abre una solicitud de extracción:

```
julia> PkgDev.publish()
INFO: Validating METADATA
INFO: No new package versions to publish
INFO: Submitting METADATA changes
INFO: Forking JuliaLang/METADATA.jl to StefanKarpinski
INFO: Pushing changes as branch pull-request/ef45f54b
INFO: To create a pull-request open:

  https://github.com/StefanKarpinski/METADATA.jl/compare/pull-request/ef45f54b
```

Tip

Si `PkgDev.publish()` falla con error:

```
ERROR: key not found: "token"
```

entonces puede que hayas encontrado un problema al usar la API de GitHub en múltiples sistemas. La solución es eliminar el token de acceso personal "*Julia Package Manager*" de tu cuenta Github e intentarlo de nuevo.

Otros fallos pueden requerir que evites `PkgDev.publish()` al crear una solicitud de extracción en GitHub. Ver: Publicación de METADATA manualmente a continuación.

Una vez que la URL del paquete para `FooBar` se registra en el repositorio oficial de`METADATA`, las personas saben de dónde clonar el paquete, pero todavía no hay ninguna versión registrada disponible. Puede etiquetarlo y registrarlo con el comando `PkgDev.tag()`:

```
julia> PkgDev.tag("FooBar")
INFO: Tagging FooBar v0.0.1
INFO: Committing METADATA for FooBar
```

Esto etiqueta el reppositorio en `FooBar` con `v0.0.1`:

```
$ cd ~/.julia/v0.6/FooBar && git tag
v0.0.1
```

También crea una nueva entrada de versión en su repositorio METADATA local paraFooBar:

```
$ cd ~/.julia/v0.6/FooBar && git show
commit de77ee4dc0689b12c5e8b574aef7f70e8b311b0e
Author: Stefan Karpinski <stefan@karpinski.org>
Date:   Wed Oct 16 23:06:18 2013 -0400

    Tag FooBar v0.0.1

diff --git a/FooBar/versions/0.0.1/sha1 b/FooBar/versions/0.0.1/sha1
new file mode 100644
index 0000000..c1cb1c1
--- /dev/null
+++ b/FooBar/versions/0.0.1/sha1
@@ -0,0 +1 @@
+84b8e266dae6de30ab9703150b3bf771ec7b6285
```

El mandato `PkgDev.tag()` toma un segundo argumento opcional que es un objeto de número de versión explícito como `v"0.0.1"` o uno de los símbolos `:patch`, `:minor` o `:major`. Estos incrementan el parche, el número de versión menor o mayor de su paquete de forma inteligente.

Agregar una versión etiquetada de su paquete agilizará el registro oficial en METADATA.jl por parte de los colaboradores. Se recomienda enfáticamente que complete este proceso, independientemente de si su paquete está completamente listo para una versión oficial.

Como regla general, los paquetes deben etiquetarse como `0.0.1` primero. Como Julia no ha alcanzado el estado `1.0`, es mejor ser conservador en las versiones etiquetadas de su paquete.

Al igual que con `PkgDev.register()`, estos cambios en `METADATA` no están disponibles para nadie más hasta que se hayan incluido en la versión anterior. De nuevo, use el comando `PkgDev.publish()`, que primero se asegura de que los repos individuales de paquetes hayan sido etiquetados, los empuja si no han sido ya, y luego abre una solicitud de extracción a `METADATA`:

```
julia> PkgDev.publish()
INFO: Validating METADATA
INFO: Pushing FooBar permanent tags: v0.0.1
INFO: Submitting METADATA changes
INFO: Forking JuliaLang/METADATA.jl to StefanKarpinski
INFO: Pushing changes as branch pull-request/3ef4f5c4
INFO: To create a pull-request open:

  https://github.com/StefanKarpinski/METADATA.jl/compare/pull-request/3ef4f5c4
```

Publishing METADATA manually

Si `PkgDev.publish()` falla, puede seguir estas instrucciones para publicar su paquete manualmente.

Al "bifurcar" el repositorio principal de METADATA, puede crear una copia personal (de METADATA.jl) en su cuenta de GitHub. Una vez que existe esa copia, puede enviar sus cambios locales a su copia (al igual que cualquier otro proyecto de GitHub).

1. Vaya a https://github.com/login?return_to=https%3A%2F%2Fgithub.com%2FJuliaLang%2FMETADATA.jl%2Ffork y cree su propia bifurcación.

2. Agregue su bifurcación como un repositorio remoto para el repositorio METADATA en su computadora local (en la terminal donde USERNAME es su nombre de usuario github)::

```
cd ~/.julia/v0.6/METADATA
git remote add USERNAME https://github.com/USERNAME/METADATA.jl.git
```

3. empuja tus cambios a tu bifurcación:

```
git push USERNAME metadata-v2
```

4. Si todo eso funciona, regrese a la página de GitHub para su tenedor y haga clic en el enlace ”solicitar solicitud”.

35.4 Fixing Package Requirements

Si necesita corregir los requisitos registrados de una versión de paquete ya publicada, puede hacerlo simplemente editando los metadatos de esa versión, que seguirá teniendo el mismo hash de confirmación: el hash asociado a una versión es permanente:

```
$ cd ~/.julia/v0.6/METADATA/FooBar/versions/0.0.1 && cat requires
julia 0.3-
$ vi requires
```

Como el hash de confirmación (*commit*) permanece igual, el contenido del archivo `REQUIRE` que se retirará en el repositorio **no** coincidirá con los requisitos en `METADATA` después de dicho cambio; esto es inevitable. Sin embargo, cuando se fijan los requisitos en `METADATA` para una versión anterior de un paquete, también se debe corregir el archivo `REQUIRE` en la versión actual del paquete.

35.5 Especificación de Requisitos

El archivo `~/.julia/v0.6/REQUIRE`, el archivo `REQUIRE` dentro de los paquetes y los archivos `require` del paquete `METADATA` utilizan un formato simple basado en línea para expresar los rangos de las versiones del paquete que necesitan estar instalados. Los archivos `REQUIRE` y `METADATA requires` también deben incluir el rango de versiones de `julia` con las que se espera que funcione el paquete. Además, los paquetes pueden incluir un archivo `test/REQUIRE` para especificar paquetes adicionales que solo son necesarios para la prueba.

Así es cómo se analizan e interpretan estos archivos.

- Todo lo que haya después de que una marca # se elimina de cada línea como un comentario.
- Si solo queda espacio en blanco, la línea se ignorará.
- Si quedan caracteres que no sean espacios en blanco, la línea es un requisito y el espacio en blanco se divide en palabras.

El requisito más simple posible es simplemente el nombre del nombre de un paquete en una línea por sí mismo:

```
Distributions
```

Este requisito se satisface con cualquier versión del paquete `Distributions`. El nombre del paquete puede ir seguido de cero o más números de versión en orden ascendente, lo que indica intervalos aceptables de versiones de ese paquete. Una versión abre un intervalo, la siguiente la cierra y la siguiente abre un nuevo intervalo, y así sucesivamente; si se da un número impar de números de versión, las versiones arbitrariamente grandes satisfarán; si se proporciona un número par de números de versión, el último es un límite superior para los números de versión aceptables. Por ejemplo, la línea:

```
Distributions 0.1
```

se satisface con cualquier versión de `Distribuciones` superior o igual a`0.1.0`. El sufijo de una versión con – también permite versiones preliminares. Por ejemplo:

```
Distributions 0.1-
```

se satisface con las versiones preliminares tales como `0.1-dev` o`0.1-rc1`, o con cualquier versión mayor o igual a `0.1.0`.

Esta entrada de requisito:

```
Distributions 0.1 0.2.5
```

se satisface con versiones desde `0.1.0` hasta, pero sin incluir, `0.2.5`. Si quiere indicar que cualquier versión de `0.1.x` va a funcionar, querrá escribir:

```
Distributions 0.1 0.2-
```

If you want to start accepting versions after `0.2.7`, you can write:

```
Distributions 0.1 0.2- 0.2.7
```

Si una línea de requisitos tiene palabras iniciales que comienzan con `@`, es un requisito dependiente del sistema. Si su sistema coincide con estos condicionales del sistema, se incluye el requisito, de lo contrario, se ignora el requisito. Por ejemplo:

```
@osx Homebrew
```

requerirá el paquete `Homebrew` solo en los sistemas donde el sistema operativo es OS X. Las condiciones del sistema que actualmente son compatibles son (jerárquicamente):

- `@unix`
  - `@linux`
  - `@bsd`
    - `@osx`
- `@windows`

La condición `@unix` se cumple en todos los sistemas UNIX, incluidos Linux y BSD. Los condicionales de sistema negados también son compatibles al agregar un `!` Después del `@` inicial. Ejemplos:

```
@!windows
@unix @!osx
```

La primera condición se aplica a cualquier sistema excepto Windows y la segunda condición se aplica a cualquier sistema UNIX además de OS X.

Los controles de tiempo de ejecución para la versión actual de Julia se pueden realizar utilizando la variable incorporada `VERSION`, que es de tipo `VersionNumber`. Dicho código ocasionalmente es necesario para realizar un seguimiento de la funcionalidad nueva o obsoleta entre varias versiones de Julia. Ejemplos de controles de tiempo de ejecución:

```
VERSION < v"0.3-" #exclude all pre-release versions of 0.3

v"0.2-" <= VERSION < v"0.3-" #get all 0.2 versions, including pre-releases, up to the above

v"0.2" <= VERSION < v"0.3-" #To get only stable 0.2 versions (Note v"0.2" == v"0.2.0")

VERSION >= v"0.2.1" #get at least version 0.2.1
```

See the section on version number literals for a more complete description.

Chapter 36

Elaboración de Perfiles (*Profiling*)

El módulo `Profile` proporciona herramientas para ayudar a los desarrolladores a mejorar el rendimiento de su código. Cuando se usa, toma medidas de código en ejecución y produce resultados que lo ayudan a comprender cuánto tiempo se gasta en línea(s) individual(es). El uso más común es identificar ”cuellos de botella” como objetivos para la optimización.

`Profile` implementa lo que se conoce como ”sampling” o *profiler* estadístico. Funciona periódicamente tomando una traza inversa durante la ejecución de cualquier tarea. Cada traz inversa captura la función que se está ejecutando actualmente y el número de línea, más la cadena completa de llamadas a función que llevó a esta línea, y por lo tanto es una ”instantánea” del estado actual de ejecución.

Si se gasta una gran parte de su tiempo de ejecución al ejecutar una línea particular de código, esta línea aparecerá con frecuencia en el conjunto de todas las trazas inversas. En otras palabras, el ”costo” de una línea determinada -o, en realidad, el costo de la secuencia de llamadas de función hasta e incluyendo esta línea- es proporcional a la frecuencia con que aparece en el conjunto de todas las trazas inversas.

Un generador de perfiles de muestreo (*sampling profiler*) no proporciona una cobertura completa línea por línea, porque las trazas inversas se producen a intervalos (por defecto, 1 ms en sistemas Unix y 10 ms en Windows, aunque la programación real está sujeta a la carga del sistema operativo). Además, como se analiza más adelante, como las muestras se recogen en un subconjunto disperso de todos los puntos de ejecución, los datos recopilados por un generador de perfiles de muestreo están sujetos a ruido estadístico.

A pesar de estas limitaciones, los perfiles de muestreo tienen fortalezas sustanciales:

- No tiene que hacer ninguna modificación en su código para tomar medidas de temporización (en contraste con la alternativa instrumenting profiler).
- Puede perfilarse en el código central de Julia e incluso (opcionalmente) en las bibliotecas C y Fortran.
- Al ejecutarse ”con poca frecuencia”, hay muy poca sobrecarga de rendimiento; durante el perfilado, su código puede ejecutarse a una velocidad casi nativa.

Por estas razones, se recomienda que intente utilizar el generador de perfiles de muestreo incorporado antes de considerar cualquier alternativa.

36.1 Uso básico

Trabajemos con un simple caso de test:

```
julia> function myfunc()

           A = rand(200, 200, 400)

           maximum(A)

       end
```

Es buena idea ejecutar primero el código que se intenta analizar al menos una vez (a menos que uno quiera analizar el compilador JIT de Julia):

```
julia> myfunc() # run once to force compilation
```

Ahora estamos listos para analizar esta función:

```
julia> @profile myfunc()
```

Para ver los resultados del *profiler* hay disponible un navegador gráfico, pero aquí usaremos la pantalla basada en texto que viene con la librería estándar:

```
julia> Profile.print()
80 ./event.jl:73; (::Base.REPL.##1#2{Base.REPL.REPLBackend})()
 80 ./REPL.jl:97; macro expansion
  80 ./REPL.jl:66; eval_user_input(::Any, ::Base.REPL.REPLBackend)
   80 ./boot.jl:235; eval(::Module, ::Any)
    80 ./<missing>:?; anonymous
     80 ./profile.jl:23; macro expansion
```

Cada línea de esta pantalla representa un punto particular (número de línea) en el código. La sangría se usa para indicar la secuencia anidada de llamadas a funciones, con líneas más sangradas que son más profundas en la secuencia de llamadas. En cada línea, el primer "campo" es el número de trazas inversas (muestras) tomadas *en esta línea o en cualquier función ejecutada por esta línea*. El segundo campo es el nombre del archivo y el número de línea, y el tercer campo es el nombre de la función. Tenga en cuenta que los números de línea específicos pueden cambiar como los cambios de código de Julia; si quieres seguir, es mejor que ejecutes este ejemplo tú mismo.

En este ejemplo, podemos ver que la función de nivel superior llamada está en el archivo `event.jl`. Esta es la función que ejecuta REPL cuando se lanza Julia. Si se examina la línea 97 de `REPL.jl`, verá que aquí es donde se llama a la función `eval_user_input()`. Esta es la función que evalúa lo que escribes en el REPL, y dado que estamos trabajando de forma interactiva estas funciones se invocaron cuando ingresamos `@profile myfunc()`. La siguiente línea refleja las acciones tomadas en la macro `@profile`.

La primera línea muestra que se tomaron 80 trazas inversas en la línea 73 de `event.jl`, pero no es que esta línea fuera "costosa" por sí misma: la tercera línea revela que las 80 trazas inversas se desencadenaron dentro de su llamada a `eval_user_input`, y así sucesivamente. Para averiguar qué operaciones se están tomando realmente el tiempo, necesitamos buscar más profundamente en la cadena de llamadas.

La primera línea "importante" en esta salida es esta:

```
52 ./REPL[1]:2; myfunc()
```

REPL se refiere al hecho de que definimos `myfunc` en REPL, en lugar de ponerlo en un archivo; si hubiéramos usado un archivo, esto mostraría el nombre del archivo. El `[1]` muestra que la función `myfunc` fue la primera expresión evaluada en esta sesión REPL. La línea 2 de `myfunc()` contiene la llamada a `rand`, y hubo 52 (de 80) trazas inversas

que ocurrieron en esta línea. Debajo de eso, puede ver una llamada a `dsfmt_fill_array_close_open!` Dentro de `dSFMT.jl`.

Un poco más abajo, ves:

```
28 ./REPL[1]:3; myfunc()
```

La línea 3 de `myfunc` contiene la llamada a `maximum`, y hubo 28 (de 80) trazas inversas tomadas aquí. Debajo de eso, puede ver los lugares específicos en `base/reduce.jl` que llevan a cabo las operaciones que consumen mucho tiempo en la función `maximum` para este tipo de datos de entrada.

En general, podemos concluir tentativamente que generar los números aleatorios es aproximadamente el doble de costoso que encontrar el elemento máximo. Podríamos aumentar nuestra confianza en este resultado recopilando más muestras:

```
julia> @profile (for i = 1:100; myfunc(); end)

julia> Profile.print()
[....]
 3821 ./REPL[1]:2; myfunc()
  3511 ./random.jl:431; rand!(::MersenneTwister, ::Array{Float64,3}, ::Int64, ::Type...
   3511 ./dSFMT.jl:84; dsfmt_fill_array_close_open!(::Base.dSFMT.DSFMT_state, ::Ptr...
  310  ./random.jl:278; rand
   [....]
 2893 ./REPL[1]:3; myfunc()
  2893 ./reduce.jl:270; _mapreduce(::Base.#identity, ::Base.#scalarmax, ::IndexLinea...
   [....]
```

En general, si tiene `N` muestras recopiladas en una línea, puede esperar una incertidumbre en el orden de `sqrt(N)` (excluyendo otras fuentes de ruido, como cuán ocupada está la computadora con otras tareas). La principal excepción a esta regla es la recolección de basura, que se ejecuta con poca frecuencia pero tiende a ser bastante costosa. (Dado que el recolector de basura de Julia está escrito en C, tales eventos pueden ser detectados usando el modo de salida `C = true` descrito a continuación, o usando ProfileView.jl.)

Esto ilustra el volcado de "árbol" predeterminado; una alternativa es el volcado "plano", que acumula conteos independientemente de su anidación:

```
julia> Profile.print(format=:flat)
 Count File          Line Function
  6714 ./<missing>     -1 anonymous
  6714 ./REPL.jl       66 eval_user_input(::Any, ::Base.REPL.REPLBackend)
  6714 ./REPL.jl       97 macro expansion
  3821 ./REPL[1]        2 myfunc()
  2893 ./REPL[1]        3 myfunc()
  6714 ./REPL[7]        1 macro expansion
  6714 ./boot.jl      235 eval(::Module, ::Any)
  3511 ./dSFMT.jl      84 dsfmt_fill_array_close_open!(::Base.dSFMT.DSFMT_s...
  6714 ./event.jl      73 (::Base.REPL.##1#2{Base.REPL.REPLBackend})()
  6714 ./profile.jl    23 macro expansion
  3511 ./random.jl    431 rand!(::MersenneTwister, ::Array{Float64,3}, ::In...
   310 ./random.jl    277 rand
   310 ./random.jl    278 rand
   310 ./random.jl    366 rand
   310 ./random.jl    369 rand
  2893 ./reduce.jl    270 _mapreduce(::Base.#identity, ::Base.#scalarmax, :...
     5 ./reduce.jl    420 mapreduce_impl(::Base.#identity, ::Base.#scalarma...
```

```
  253 ./reduce.jl    426 mapreduce_impl(::Base.#identity, ::Base.#scalarma...
 2592 ./reduce.jl    428 mapreduce_impl(::Base.#identity, ::Base.#scalarma...
   43 ./reduce.jl    429 mapreduce_impl(::Base.#identity, ::Base.#scalarma...
```

Si nuestro código tiene recursión, un punto potencialmente confso es que una línea en una función ”hija” puede acumular ms cuentas que hay de trazas inversas en total.Considere las siguientes definiciones de función:

```
dumbsum(n::Integer) = n == 1 ? 1 : 1 + dumbsum(n-1)
dumbsum3() = dumbsum(3)
```

Si tuviéramos que analizar `dumbsum3`, y una traza inversa fuera tomada mientras estaba ejecutándose `dumbsum(1)`, la traza inversa tendría este aspecto:

```
dumbsum3
    dumbsum(3)
        dumbsum(2)
            dumbsum(1)
```

En consecuencia, esta función hija realiza tres cuentas incluso aunque la padre sólo realiza una. La representación en ”árbol” hace esto mucho ms claro, y por esta razón (entre otras) es probablemente la forma más útil de ver los resultados.

36.2 Acumulación y Limpieza

Los resultados de `@profile` se acumulan en un buffer; si ejecuta varios fragmentos de código en `@profile`, entonces `Profile.print()` le mostrará los resultados combinados. Esto puede ser muy útil, pero a veces desea comenzar de nuevo; puede hacerlo con `Profile.clear()`.

36.3 Opciones para controlar la visión de los resultados del análisis

`Profile.print()` tienes más opciones de las que se han descrito hasta ahora. Veamos la declaración completa:

```
function print(io::IO = STDOUT, data = fetch(); kwargs...)
```

Primero discutamos los dos argumentos posicionales, y luego los argumentos *keyword*:

- `io` – Le permite guardar los resultados en un búfer, por ejemplo, un archivo, pero el predeterminado es imprimir en STDOUT (la consola)..
- `data` – Contiene los datos que desea analizar; de forma predeterminada, se obtiene de `Profile.fetch()`, que extrae trazas inversas de un búfer preasignado. Por ejemplo, si desea perfilar el generador de perfiles, podría decir:

  ```
  data = copy(Profile.fetch())
  Profile.clear()
  @profile Profile.print(STDOUT, data) # Prints the previous results
  Profile.print()                      # Prints results from Profile.print()
  ```

The argumentos *keyword* pueden ser cualquier combinación de:

- `format` – Introducido anteriormente, determina si se imprimen trazas inversas con (por defecto, `: árbol`) o sin (`: plano`) indentación que indica la estructura en árbol.
- `C` – Si`true`, se muestran trazas inversas de C y Fortran (normalmente están excluidas). Intente ejecutar el ejemplo introductorio con `Profile.print(C = true)`. Esto puede ser extremadamente útil para decidir si es el código Julia o el código C lo que está causando un cuello de botella; establecer `C = true` también mejora la interpretabilidad del anidamiento, a coste de unos listados de perfil más largos.
- `combine` – Algunas líneas de código contienen múltiples operaciones; por ejemplo, `s + = A [i]` contiene una referencia de matriz (`A[i]`) y una operación de suma. Estos corresponden a diferentes líneas en el código máquina generado, y por lo tanto puede haber dos o más direcciones diferentes capturadas durante las trazas inversas en esta línea. `combine = true` los agrupa, y es probablemente lo que generalmente desea, pero puede generar un resultado por separado para cada puntero de instrucción con`combine = false`.
- `maxdepth` – Limita los marcos a una profundidad mayor que`maxdepth` en el formato `:tree`.
- `sortedby` – Controla el orden en formato `:flat`. `:filefuncline` (predeterminado) ordena por la línea de origen, mientras que `:count` se ordena según el número de muestras recolectadas.
- `noisefloor` – Límita los marcos que están debajo del umbral de ruido heurístico de la muestra (solo se aplica al formato`: tree`). Un valor sugerido para intentar esto es 2.0 (el valor predeterminado es 0). Este parámetro oculta muestras para las cuales `n <= noisefloor * √N`, donden es el número de muestras en esta línea, y `N` es el número de muestras para el método invocado.
- `mincount` – Limita marcos con menos de `mincount` ocurrencias.

Los nombres de archivo/función a veces se truncan (con `...`), y la sangría se trunca con un `+n` al principio, donde `n` es el número de espacios adicionales que se habrían insertado, si hubiera habido espacio . Si desea un perfil completo de código profundamente anidado, a menudo una buena idea es guardar en un archivo usando un ancho "tamaño de pantalla" en un `IOContext`:

```
open("/tmp/prof.txt", "w") do s
    Profile.print(IOContext(s, :displaysize => (24, 500)))
end
```

36.4 Configuración

`@profile` solo acumula *backtraces*, y el análisis ocurre cuando usted llama a `Profile.print()`. Para un cálculo de larga ejecución, es muy posible que se llene el búfer preasignado para almacenar *backtraces*. Si eso sucede, las trazas inversas se detienen pero su cálculo continúa. Como consecuencia, es posible que omitan algunos datos importantes de generación de perfiles (recibirá una advertencia cuando eso suceda).

Puede obtener y configurar los parámetros relevantes de esta manera:

```
Profile.init() # returns the current settings
Profile.init(n = 10^7, delay = 0.01)
```

`n` es la cantidad total de punteros de instrucción que puede almacenar, con un valor predeterminado de `10 ^ 6`. Si su traza inversa típica es de 20 punteros de instrucción, puede recopilar 50000 trazas inversas, lo que sugiere una incertidumbre estadística de menos del 1%. Esto puede ser lo suficientemente bueno para la mayoría de las aplicaciones.

En consecuencia, es más probable que necesite modificar el `retraso`, expresado en segundos, que establece la cantidad de tiempo que Julia obtiene entre las instantáneas para realizar los cálculos solicitados. Un trabajo de larga

duración puede no necesitar trazas frecuentes. La configuración predeterminada es `delay = 0.001`. Por supuesto, puede disminuir la demora y aumentarla; sin embargo, la sobrecarga de los perfiles crece una vez que la demora se vuelve similar a la cantidad de tiempo necesaria para tomar una traza inversa (~ 30 microsegundos en la computadora portátil del autor).

Chapter 37

Análisis de la asignación de memoria

Una de las técnicas más comunes para mejorar el rendimiento es reducir la asignación de memoria. La cantidad total de asignación se puede medir con `@time` y `@allocated`, y las líneas específicas que desencadenan la asignación a pueden a menudo inferirse a partir del perfil a través del costo de la recolección de basura en la que incurren estas líneas. Sin embargo, a veces es más eficiente medir directamente la cantidad de memoria asignada por cada línea de código.

Para medir la asignación línea por línea, inicie Julia con la opción de línea de comando `--track-allocation = <setting>`, para la cual puede elegir none (el predeterminado, no medir la asignación), `user` (mida la asignación de memoria en todas partes excepto el código central de Julia), o `todo` (mida la asignación de memoria en cada línea del código Julia). La asignación se mide para cada línea de código compilado. Cuando sale de Julia, los resultados acumulativos se escriben en archivos de texto con `.mem` adjunto después del nombre del archivo, que residen en el mismo directorio que el archivo de origen. Cada línea enumera la cantidad total de bytes asignados. El paquete `Coverage` contiene algunas herramientas de análisis elementales, por ejemplo, para ordenar las líneas en orden de cantidad de bytes asignados.

Al interpretar los resultados, hay algunos detalles importantes. Bajo la configuración `user`, la primera línea de cualquier función directamente llamada desde REPL exhibirá asignación debido a eventos que ocurren en el código REPL. Más significativamente, la compilación de JIT también se suma a los recuentos de asignación, porque gran parte del compilador de Julia está escrito en Julia (y la compilación generalmente requiere asignación de memoria). El procedimiento recomendado es forzar la compilación ejecutando todos los comandos que desea analizar, luego llame a `Profile.clear_malloc_data()` para restablecer todos los contadores de asignación. Finalmente, ejecute los comandos deseados y salga de Julia para activar la generación de los archivos `.mem`.

Chapter 38

Trazas de Pila

El módulo `StackTraces` proporciona simples seguimientos de pila que son legibles para los humanos y fáciles de usar mediante programación.

38.1 Viendo un rastro de pila

La función principal utilizada para obtener un seguimiento de pila es `stacktrace()`:

```
julia> stacktrace()
4-element Array{StackFrame,1}:
 eval(::Module, ::Any) at boot.jl:236
 eval_user_input(::Any, ::Base.REPL.REPLBackend) at REPL.jl:66
 macro expansion at REPL.jl:97 [inlined]
 (::Base.REPL.##1#2{Base.REPL.REPLBackend})() at event.jl:73
```

Llamar a `stacktrace()` devuelve un vector de `StackFrame`s. Para facilitar el uso, el alias `StackTrace` se puede usar en lugar de `Vector{StackFrame}`. (Los ejemplos con `[...]` indican que la salida puede variar dependiendo de cómo se ejecuta el código).

```
julia> example() = stacktrace()
example (generic function with 1 method)

julia> example()
5-element Array{StackFrame,1}:
 example() at REPL[1]:1
 eval(::Module, ::Any) at boot.jl:236
[...]

julia> @noinline child() = stacktrace()
child (generic function with 1 method)

julia> @noinline parent() = child()
parent (generic function with 1 method)

julia> grandparent() = parent()
grandparent (generic function with 1 method)

julia> grandparent()
7-element Array{StackFrame,1}:
```

```
 child() at REPL[3]:1
 parent() at REPL[4]:1
 grandparent() at REPL[5]:1
[...]
```

Tenga en cuenta que cuando llama a `stacktrace()` normalmente verá un marco con `eval(...) en boot.jl`. Al invocar `stacktrace()` desde el REPL, también tendrá algunos fotogramas adicionales en la pila de `REPL.jl`, que generalmente se ve así:

```
julia> example() = stacktrace()
example (generic function with 1 method)

julia> example()
5-element Array{StackFrame,1}:
 example() at REPL[1]:1
 eval(::Module, ::Any) at boot.jl:236
 eval_user_input(::Any, ::Base.REPL.REPLBackend) at REPL.jl:66
 macro expansion at REPL.jl:97 [inlined]
 (::Base.REPL.##1#2{Base.REPL.REPLBackend})() at event.jl:73
```

38.2 Extracting useful information

Cada `StackFrame` contiene el nombre de la función, el nombre del archivo, el número de línea, la información lambda, un indicador que indica si el marco ha sido insertado, un indicador que indica si es una función C (por defecto las funciones C no aparecen en el seguimiento de la pila) y una representación entera del puntero devuelto por `backtrace()`:

```
julia> top_frame = stacktrace()[1]
eval(::Module, ::Any) at boot.jl:236

julia> top_frame.func
:eval

julia> top_frame.file
Symbol("./boot.jl")

julia> top_frame.line
236

julia> top_frame.linfo
Nullable{Core.MethodInstance}(MethodInstance for eval(::Module, ::Any))

julia> top_frame.inlined
false

julia> top_frame.from_c
false
```

```
julia> top_frame.pointer
0x00007f390d152a59
```

Esto hace que la información de seguimiento de pila esté disponible programáticamente para el registro, el manejo de errores y más.

38.3 Manejo de Errores

Si bien tener acceso fácil a la información sobre el estado actual de la pila de llamadas puede ser útil en muchos lugares, la aplicación más obvia es la gestión de errores y la depuración.

```
julia> @noinline bad_function() = undeclared_variable
bad_function (generic function with 1 method)

julia> @noinline example() = try

           bad_function()

       catch

           stacktrace()

       end
example (generic function with 1 method)

julia> example()
5-element Array{StackFrame,1}:
 example() at REPL[2]:4
 eval(::Module, ::Any) at boot.jl:236
[...]
```

Puede observar que en el ejemplo anterior, el primer marco apila puntos en la línea 4, donde se llama a `stacktrace()`, en lugar de a la línea 2, donde se llama *bad_function* y el marco de `bad_function` falta por completo. Esto es comprensible, dado que `stacktrace()` se llama desde el contexto de *catch*. Si bien en este ejemplo es bastante fácil encontrar el origen real del error, en casos complejos, rastrear el origen del error no es trivial.

Esto se puede remediar llamando a `catch_stacktrace()` en lugar de `stacktrace()`. En lugar de devolver la información de la pila de llamadas para el contexto actual, `catch_stacktrace()` devuelve la información de la pila para el contexto de la excepción más reciente:

```
julia> @noinline bad_function() = undeclared_variable
bad_function (generic function with 1 method)

julia> @noinline example() = try

           bad_function()

       catch

           catch_stacktrace()

       end
example (generic function with 1 method)

julia> example()
6-element Array{StackFrame,1}:
 bad_function() at REPL[1]:1
 example() at REPL[2]:2
[...]
```

Nótese que la traza de la pila indica ahora el número de línea apropiado y el marco perdido.

```
julia> @noinline child() = error("Whoops!")
child (generic function with 1 method)

julia> @noinline parent() = child()
parent (generic function with 1 method)

julia> @noinline function grandparent()

           try

               parent()

           catch err

               println("ERROR: ", err.msg)

               catch_stacktrace()

           end

       end
grandparent (generic function with 1 method)

julia> grandparent()
ERROR: Whoops!
7-element Array{StackFrame,1}:
 child() at REPL[1]:1
 parent() at REPL[2]:1
 grandparent() at REPL[3]:3
[...]
```

38.4 Comparación con `backtrace()`

Una llamada a `backtrace()` devuelve un vector de `Ptr{Void}`, que puede pasarse luego a `stacktrace()` para la traducción:

```
julia> trace = backtrace()
21-element Array{Ptr{Void},1}:
 Ptr{Void} @0x00007f10049d5b2f
 Ptr{Void} @0x00007f0ffeb4d29c
 Ptr{Void} @0x00007f0ffeb4d2a9
 Ptr{Void} @0x00007f1004993fe7
 Ptr{Void} @0x00007f10049a92be
 Ptr{Void} @0x00007f10049a823a
 Ptr{Void} @0x00007f10049a9fb0
 Ptr{Void} @0x00007f10049aa718
 Ptr{Void} @0x00007f10049c0d5e
 Ptr{Void} @0x00007f10049a3286
 Ptr{Void} @0x00007f0ffe9ba3ba
 Ptr{Void} @0x00007f0ffe9ba3d0
 Ptr{Void} @0x00007f1004993fe7
 Ptr{Void} @0x00007f0ded34583d
 Ptr{Void} @0x00007f0ded345a87
 Ptr{Void} @0x00007f1004993fe7
 Ptr{Void} @0x00007f0ded34308f
```

```
 Ptr{Void} @0x00007f0ded343320
 Ptr{Void} @0x00007f1004993fe7
 Ptr{Void} @0x00007f10049aeb67
 Ptr{Void} @0x0000000000000000

julia> stacktrace(trace)
5-element Array{StackFrame,1}:
 backtrace() at error.jl:46
 eval(::Module, ::Any) at boot.jl:236
 eval_user_input(::Any, ::Base.REPL.REPLBackend) at REPL.jl:66
 macro expansion at REPL.jl:97 [inlined]
 (::Base.REPL.##1#2{Base.REPL.REPLBackend})() at event.jl:73
```

Observe que el vector devuelto por `backtrace()` tenía 21 punteros, mientras que el vector devuelto por `stacktrace()` solo tiene 5. Esto es porque, de forma predeterminada, `stacktrace()` elimina cualquier función C de nivel inferior de la pila. Si desea incluir cuadros de pila de llamadas C, puede hacerlo así:

```
julia> stacktrace(trace, true)
27-element Array{StackFrame,1}:
 jl_backtrace_from_here at stackwalk.c:103
 backtrace() at error.jl:46
 backtrace() at sys.so:?
 jl_call_method_internal at julia_internal.h:248 [inlined]
 jl_apply_generic at gf.c:2215
 do_call at interpreter.c:75
 eval at interpreter.c:215
 eval_body at interpreter.c:519
 jl_interpret_toplevel_thunk at interpreter.c:664
 jl_toplevel_eval_flex at toplevel.c:592
 jl_toplevel_eval_in at builtins.c:614
 eval(::Module, ::Any) at boot.jl:236
 eval(::Module, ::Any) at sys.so:?
 jl_call_method_internal at julia_internal.h:248 [inlined]
 jl_apply_generic at gf.c:2215
 eval_user_input(::Any, ::Base.REPL.REPLBackend) at REPL.jl:66
 ip:0x7f1c707f1846
 jl_call_method_internal at julia_internal.h:248 [inlined]
 jl_apply_generic at gf.c:2215
 macro expansion at REPL.jl:97 [inlined]
 (::Base.REPL.##1#2{Base.REPL.REPLBackend})() at event.jl:73
 ip:0x7f1c707ea1ef
 jl_call_method_internal at julia_internal.h:248 [inlined]
 jl_apply_generic at gf.c:2215
 jl_apply at julia.h:1411 [inlined]
 start_task at task.c:261
 ip:0xffffffffffffffff
```

Los punteros individuales devueltos por `backtrace()` se pueden traducir a `StackFrame`s pasándolos a `StackTraces.lookup()`:

```
julia> pointer = backtrace()[1];

julia> frame = StackTraces.lookup(pointer)
1-element Array{StackFrame,1}:
```

```
 jl_backtrace_from_here at stackwalk.c:103

julia> println("The top frame is from $(frame[1].func)!")
The top frame is from jl_backtrace_from_here!
```

Chapter 39

Performance Tips

In the following sections, we briefly go through a few techniques that can help make your Julia code run as fast as possible.

39.1 Avoid global variables

A global variable might have its value, and therefore its type, change at any point. This makes it difficult for the compiler to optimize code using global variables. Variables should be local, or passed as arguments to functions, whenever possible.

Any code that is performance critical or being benchmarked should be inside a function.

We find that global names are frequently constants, and declaring them as such greatly improves performance:

```
const DEFAULT_VAL = 0
```

Uses of non-constant globals can be optimized by annotating their types at the point of use:

```
global x
y = f(x::Int + 1)
```

Writing functions is better style. It leads to more reusable code and clarifies what steps are being done, and what their inputs and outputs are.

> **Note**
>
> All code in the REPL is evaluated in global scope, so a variable defined and assigned at toplevel will be a **global** variable.

In the following REPL session:

```
julia> x = 1.0
```

is equivalent to:

```
julia> global x = 1.0
```

so all the performance issues discussed previously apply.

39.2 Measure performance with `@time` and pay attention to memory allocation

A useful tool for measuring performance is the `@time` macro. The following example illustrates good working style:

```
julia> function f(n)

           s = 0

           for i = 1:n

               s += i/2

           end

           s

       end
f (generic function with 1 method)

julia> @time f(1)
  0.012686 seconds (2.09 k allocations: 103.421 KiB)
0.5

julia> @time f(10^6)
  0.021061 seconds (3.00 M allocations: 45.777 MiB, 11.69% gc time)
2.5000025e11
```

On the first call (`@time f(1)`), `f` gets compiled. (If you've not yet used `@time` in this session, it will also compile functions needed for timing.) You should not take the results of this run seriously. For the second run, note that in addition to reporting the time, it also indicated that a large amount of memory was allocated. This is the single biggest advantage of `@time` vs. functions like `tic()` and `toc()`, which only report time.

Unexpected memory allocation is almost always a sign of some problem with your code, usually a problem with type-stability. Consequently, in addition to the allocation itself, it's very likely that the code generated for your function is far from optimal. Take such indications seriously and follow the advice below.

For more serious benchmarking, consider the BenchmarkTools.jl package which evaluates the function multiple times in order to reduce noise.

As a teaser, an improved version of this function allocates no memory (the allocation reported below is due to running the `@time` macro in global scope) and has an order of magnitude faster execution after the first call:

```
julia> @time f_improved(1)
  0.007008 seconds (1.32 k allocations: 63.640 KiB)
0.5

julia> @time f_improved(10^6)
  0.002997 seconds (6 allocations: 192 bytes)
2.5000025e11
```

Below you'll learn how to spot the problem with `f` and how to fix it.

In some situations, your function may need to allocate memory as part of its operation, and this can complicate the simple picture above. In such cases, consider using one of the tools below to diagnose problems, or write a version of your function that separates allocation from its algorithmic aspects (see Pre-allocating outputs).

39.3 Tools

Julia and its package ecosystem includes tools that may help you diagnose problems and improve the performance of your code:

- Profiling allows you to measure the performance of your running code and identify lines that serve as bottlenecks. For complex projects, the ProfileView package can help you visualize your profiling results.
- Unexpectedly-large memory allocations–as reported by `@time`, `@allocated`, or the profiler (through calls to the garbage-collection routines)–hint that there might be issues with your code. If you don't see another reason for the allocations, suspect a type problem. You can also start Julia with the `--track-allocation=user` option and examine the resulting `*.mem` files to see information about where those allocations occur. See Memory allocation analysis.
- `@code_warntype` generates a representation of your code that can be helpful in finding expressions that result in type uncertainty. See `@code_warntype` below.
- The Lint package can also warn you of certain types of programming errors.

39.4 Avoid containers with abstract type parameters

When working with parameterized types, including arrays, it is best to avoid parameterizing with abstract types where possible.

Consider the following:

```
a = Real[]    # typeof(a) = Array{Real,1}
if (f = rand()) < .8
    push!(a, f)
end
```

Because `a` is a an array of abstract type `Real`, it must be able to hold any `Real` value. Since `Real` objects can be of arbitrary size and structure, `a` must be represented as an array of pointers to individually allocated `Real` objects. Because `f` will always be a `Float64`, we should instead, use:

```
a = Float64[] # typeof(a) = Array{Float64,1}
```

which will create a contiguous block of 64-bit floating-point values that can be manipulated efficiently.

See also the discussion under Parametric Types.

39.5 Type declarations

In many languages with optional type declarations, adding declarations is the principal way to make code run faster. This is *not* the case in Julia. In Julia, the compiler generally knows the types of all function arguments, local variables, and expressions. However, there are a few specific instances where declarations are helpful.

Avoid fields with abstract type

Types can be declared without specifying the types of their fields:

```
julia> struct MyAmbiguousType
           a
       end
```

This allows a to be of any type. This can often be useful, but it does have a downside: for objects of type `MyAmbiguousType`, the compiler will not be able to generate high-performance code. The reason is that the compiler uses the types of objects, not their values, to determine how to build code. Unfortunately, very little can be inferred about an object of type `MyAmbiguousType`:

```
julia> b = MyAmbiguousType("Hello")
MyAmbiguousType("Hello")

julia> c = MyAmbiguousType(17)
MyAmbiguousType(17)

julia> typeof(b)
MyAmbiguousType

julia> typeof(c)
MyAmbiguousType
```

b and c have the same type, yet their underlying representation of data in memory is very different. Even if you stored just numeric values in field a, the fact that the memory representation of a `UInt8` differs from a `Float64` also means that the CPU needs to handle them using two different kinds of instructions. Since the required information is not available in the type, such decisions have to be made at run-time. This slows performance.

You can do better by declaring the type of a. Here, we are focused on the case where a might be any one of several types, in which case the natural solution is to use parameters. For example:

```
julia> mutable struct MyType{T<:AbstractFloat}
           a::T
       end
```

This is a better choice than

```
julia> mutable struct MyStillAmbiguousType
           a::AbstractFloat
       end
```

because the first version specifies the type of a from the type of the wrapper object. For example:

```
julia> m = MyType(3.2)
MyType{Float64}(3.2)

julia> t = MyStillAmbiguousType(3.2)
MyStillAmbiguousType(3.2)

julia> typeof(m)
MyType{Float64}

julia> typeof(t)
MyStillAmbiguousType
```

The type of field a can be readily determined from the type of `m`, but not from the type of `t`. Indeed, in `t` it's possible to change the type of field a:

```
julia> typeof(t.a)
Float64

julia> t.a = 4.5f0
4.5f0

julia> typeof(t.a)
Float32
```

In contrast, once `m` is constructed, the type of `m.a` cannot change:

```
julia> m.a = 4.5f0
4.5f0

julia> typeof(m.a)
Float64
```

The fact that the type of `m.a` is known from `m`'s type–coupled with the fact that its type cannot change mid-function–allows the compiler to generate highly-optimized code for objects like `m` but not for objects like `t`.

Of course, all of this is true only if we construct `m` with a concrete type. We can break this by explicitly constructing it with an abstract type:

```
julia> m = MyType{AbstractFloat}(3.2)
MyType{AbstractFloat}(3.2)

julia> typeof(m.a)
Float64

julia> m.a = 4.5f0
4.5f0

julia> typeof(m.a)
Float32
```

For all practical purposes, such objects behave identically to those of `MyStillAmbiguousType`.

It's quite instructive to compare the sheer amount code generated for a simple function

```
func(m::MyType) = m.a+1
```

using

```
code_llvm(func,Tuple{MyType{Float64}})
code_llvm(func,Tuple{MyType{AbstractFloat}})
code_llvm(func,Tuple{MyType})
```

For reasons of length the results are not shown here, but you may wish to try this yourself. Because the type is fully-specified in the first case, the compiler doesn't need to generate any code to resolve the type at run-time. This results in shorter and faster code.

Avoid fields with abstract containers

The same best practices also work for container types:

```
julia> mutable struct MySimpleContainer{A<:AbstractVector}
           a::A
       end

julia> mutable struct MyAmbiguousContainer{T}
           a::AbstractVector{T}
       end
```

For example:

```
julia> c = MySimpleContainer(1:3);

julia> typeof(c)
MySimpleContainer{UnitRange{Int64}}

julia> c = MySimpleContainer([1:3;]);

julia> typeof(c)
MySimpleContainer{Array{Int64,1}}

julia> b = MyAmbiguousContainer(1:3);

julia> typeof(b)
MyAmbiguousContainer{Int64}

julia> b = MyAmbiguousContainer([1:3;]);

julia> typeof(b)
MyAmbiguousContainer{Int64}
```

For `MySimpleContainer`, the object is fully-specified by its type and parameters, so the compiler can generate optimized functions. In most instances, this will probably suffice.

While the compiler can now do its job perfectly well, there are cases where *you* might wish that your code could do different things depending on the *element type* of `a`. Usually the best way to achieve this is to wrap your specific operation (here, `foo`) in a separate function:

```
julia> function sumfoo(c::MySimpleContainer)
           s = 0
           for x in c.a
               s += foo(x)
           end
           s
       end
sumfoo (generic function with 1 method)

julia> foo(x::Integer) = x
foo (generic function with 1 method)

julia> foo(x::AbstractFloat) = round(x)
foo (generic function with 2 methods)
```

This keeps things simple, while allowing the compiler to generate optimized code in all cases.

However, there are cases where you may need to declare different versions of the outer function for different element types of `a`. You could do it like this:

```
function myfun(c::MySimpleContainer{Vector{T}}) where T<:AbstractFloat
```

```
    ...
end
function myfun(c::MySimpleContainer{Vector{T}}) where T<:Integer
    ...
end
```

This works fine for `Vector{T}`, but we'd also have to write explicit versions for `UnitRange{T}` or other abstract types. To prevent such tedium, you can use two parameters in the declaration of `MyContainer`:

```
julia> mutable struct MyContainer{T, A<:AbstractVector}
           a::A
       end

julia> MyContainer(v::AbstractVector) = MyContainer{eltype(v), typeof(v)}(v)
MyContainer

julia> b = MyContainer(1:5);

julia> typeof(b)
MyContainer{Int64,UnitRange{Int64}}
```

Note the somewhat surprising fact that `T` doesn't appear in the declaration of field `a`, a point that we'll return to in a moment. With this approach, one can write functions such as:

```
julia> function myfunc(c::MyContainer{<:Integer, <:AbstractArray})
           return c.a[1]+1
       end
myfunc (generic function with 1 method)

julia> function myfunc(c::MyContainer{<:AbstractFloat})
           return c.a[1]+2
       end
myfunc (generic function with 2 methods)

julia> function myfunc(c::MyContainer{T,Vector{T}}) where T<:Integer
           return c.a[1]+3
       end
myfunc (generic function with 3 methods)
```

> **Note**
>
> Because we can only define `MyContainer` for `A<:AbstractArray`, and any unspecified parameters are arbitrary, the first function above could have been written more succinctly as `function myfunc{T<:Integer}(c::MyContainer{T})`

```
julia> myfunc(MyContainer(1:3))
2

julia> myfunc(MyContainer(1.0:3))
3.0

julia> myfunc(MyContainer([1:3;]))
4
```

As you can see, with this approach it's possible to specialize on both the element type `T` and the array type `A`.

However, there's one remaining hole: we haven't enforced that `A` has element type `T`, so it's perfectly possible to construct an object like this:

```
julia> b = MyContainer{Int64, UnitRange{Float64}}(UnitRange(1.3, 5.0));

julia> typeof(b)
MyContainer{Int64,UnitRange{Float64}}
```

To prevent this, we can add an inner constructor:

```
julia> mutable struct MyBetterContainer{T<:Real, A<:AbstractVector}
           a::A
           MyBetterContainer{T,A}(v::AbstractVector{T}) where {T,A} = new(v)
       end

julia> MyBetterContainer(v::AbstractVector) = MyBetterContainer{eltype(v),typeof(v)}(v)
MyBetterContainer

julia> b = MyBetterContainer(UnitRange(1.3, 5.0));

julia> typeof(b)
MyBetterContainer{Float64,UnitRange{Float64}}

julia> b = MyBetterContainer{Int64, UnitRange{Float64}}(UnitRange(1.3, 5.0));
ERROR: MethodError: Cannot `convert` an object of type UnitRange{Float64} to an object of type
    MyBetterContainer{Int64,UnitRange{Float64}}
[...]
```

The inner constructor requires that the element type of `A` be `T`.

Annotate values taken from untyped locations

It is often convenient to work with data structures that may contain values of any type (arrays of type `Array{Any}`). But, if you're using one of these structures and happen to know the type of an element, it helps to share this knowledge with the compiler:

```
function foo(a::Array{Any,1})
    x = a[1]::Int32
    b = x+1
    ...
end
```

Here, we happened to know that the first element of a would be an `Int32`. Making an annotation like this has the added benefit that it will raise a run-time error if the value is not of the expected type, potentially catching certain bugs earlier.

Declare types of keyword arguments

Keyword arguments can have declared types:

```
function with_keyword(x; name::Int = 1)
    ...
end
```

Functions are specialized on the types of keyword arguments, so these declarations will not affect performance of code inside the function. However, they will reduce the overhead of calls to the function that include keyword arguments.

Functions with keyword arguments have near-zero overhead for call sites that pass only positional arguments.

Passing dynamic lists of keyword arguments, as in `f(x; keywords...)`, can be slow and should be avoided in performance-sensitive code.

39.6 Break functions into multiple definitions

Writing a function as many small definitions allows the compiler to directly call the most applicable code, or even inline it.

Here is an example of a "compound function" that should really be written as multiple definitions:

```
function norm(A)
    if isa(A, Vector)
        return sqrt(real(dot(A,A)))
    elseif isa(A, Matrix)
        return maximum(svd(A)[2])
    else
        error("norm: invalid argument")
    end
end
```

This can be written more concisely and efficiently as:

```
norm(x::Vector) = sqrt(real(dot(x,x)))
norm(A::Matrix) = maximum(svd(A)[2])
```

39.7 Write "type-stable" functions

When possible, it helps to ensure that a function always returns a value of the same type. Consider the following definition:

```
pos(x) = x < 0 ? 0 : x
```

Although this seems innocent enough, the problem is that `0` is an integer (of type `Int`) and `x` might be of any type. Thus, depending on the value of `x`, this function might return a value of either of two types. This behavior is allowed, and may be desirable in some cases. But it can easily be fixed as follows:

```
pos(x) = x < 0 ? zero(x) : x
```

There is also a `one()` function, and a more general `oftype(x, y)` function, which returns `y` converted to the type of `x`.

39.8 Avoid changing the type of a variable

An analogous "type-stability" problem exists for variables used repeatedly within a function:

```
function foo()
    x = 1
    for i = 1:10
        x = x/bar()
    end
    return x
end
```

Local variable x starts as an integer, and after one loop iteration becomes a floating-point number (the result of `/` operator). This makes it more difficult for the compiler to optimize the body of the loop. There are several possible fixes:

- Initialize x with `x = 1.0`
- Declare the type of x: `x::Float64 = 1`
- Use an explicit conversion: `x = oneunit(T)`
- Initialize with the first loop iteration, to `x = 1/bar()`, then loop `for i = 2:10`

39.9 Separate kernel functions (aka, function barriers)

Many functions follow a pattern of performing some set-up work, and then running many iterations to perform a core computation. Where possible, it is a good idea to put these core computations in separate functions. For example, the following contrived function returns an array of a randomly-chosen type:

```
julia> function strange_twos(n)
           a = Vector{rand(Bool) ? Int64 : Float64}(n)
           for i = 1:n
               a[i] = 2
           end
           return a
       end
strange_twos (generic function with 1 method)

julia> strange_twos(3)
3-element Array{Float64,1}:
 2.0
 2.0
 2.0
```

This should be written as:

```
julia> function fill_twos!(a)

           for i=1:length(a)

               a[i] = 2

           end

       end
fill_twos! (generic function with 1 method)

julia> function strange_twos(n)

           a = Array{rand(Bool) ? Int64 : Float64}(n)

           fill_twos!(a)

           return a

       end
strange_twos (generic function with 1 method)

julia> strange_twos(3)
3-element Array{Float64,1}:
 2.0
 2.0
 2.0
```

Julia's compiler specializes code for argument types at function boundaries, so in the original implementation it does not know the type of a during the loop (since it is chosen randomly). Therefore the second version is generally faster since the inner loop can be recompiled as part of `fill_twos!` for different types of a.

The second form is also often better style and can lead to more code reuse.

This pattern is used in several places in the standard library. For example, see `hvcat_fill` in `abstractarray.jl`, or the `fill!` function, which we could have used instead of writing our own `fill_twos!`.

Functions like `strange_twos` occur when dealing with data of uncertain type, for example data loaded from an input file that might contain either integers, floats, strings, or something else.

39.10 Types with values-as-parameters

Let's say you want to create an N-dimensional array that has size 3 along each axis. Such arrays can be created like this:

```
julia> A = fill(5.0, (3, 3))
3×3 Array{Float64,2}:
 5.0  5.0  5.0
 5.0  5.0  5.0
 5.0  5.0  5.0
```

This approach works very well: the compiler can figure out that `A` is an `Array{Float64,2}` because it knows the type of the fill value (`5.0::Float64`) and the dimensionality (`(3, 3)::NTuple{2,Int}`). This implies that the compiler can generate very efficient code for any future usage of `A` in the same function.

But now let's say you want to write a function that creates a 3×3×... array in arbitrary dimensions; you might be tempted to write a function

```
julia> function array3(fillval, N)

           fill(fillval, ntuple(d->3, N))

       end
array3 (generic function with 1 method)

julia> array3(5.0, 2)
3×3 Array{Float64,2}:
 5.0  5.0  5.0
 5.0  5.0  5.0
 5.0  5.0  5.0
```

This works, but (as you can verify for yourself using `@code_warntype array3(5.0, 2)`) the problem is that the output type cannot be inferred: the argument `N` is a *value* of type `Int`, and type-inference does not (and cannot) predict its value in advance. This means that code using the output of this function has to be conservative, checking the type on each access of `A`; such code will be very slow.

Now, one very good way to solve such problems is by using the function-barrier technique. However, in some cases you might want to eliminate the type-instability altogether. In such cases, one approach is to pass the dimensionality as a parameter, for example through `Val{T}` (see "Value types"):

```
julia> function array3(fillval, ::Type{Val{N}}) where N

           fill(fillval, ntuple(d->3, Val{N}))

       end
array3 (generic function with 1 method)

julia> array3(5.0, Val{2})
3×3 Array{Float64,2}:
 5.0  5.0  5.0
 5.0  5.0  5.0
 5.0  5.0  5.0
```

Julia has a specialized version of `ntuple` that accepts a `Val{::Int}` as the second parameter; by passing `N` as a type-parameter, you make its "value" known to the compiler. Consequently, this version of `array3` allows the compiler to predict the return type.

However, making use of such techniques can be surprisingly subtle. For example, it would be of no help if you called `array3` from a function like this:

```
function call_array3(fillval, n)
    A = array3(fillval, Val{n})
end
```

Here, you've created the same problem all over again: the compiler can't guess the type of `n`, so it doesn't know the type of `Val{n}`. Attempting to use `Val`, but doing so incorrectly, can easily make performance *worse* in many situations. (Only in situations where you're effectively combining `Val` with the function-barrier trick, to make the kernel function more efficient, should code like the above be used.)

An example of correct usage of `Val` would be:

```
function filter3(A::AbstractArray{T,N}) where {T,N}
    kernel = array3(1, Val{N})
    filter(A, kernel)
end
```

In this example, `N` is passed as a parameter, so its "value" is known to the compiler. Essentially, `Val{T}` works only when `T` is either hard-coded (`Val{3}`) or already specified in the type-domain.

39.11 The dangers of abusing multiple dispatch (aka, more on types with values-as-parameters)

Once one learns to appreciate multiple dispatch, there's an understandable tendency to go crazy and try to use it for everything. For example, you might imagine using it to store information, e.g.

```
struct Car{Make,Model}
    year::Int
    ...more fields...
end
```

and then dispatch on objects like `Car{:Honda,:Accord}(year, args...)`.

This might be worthwhile when the following are true:

- You require CPU-intensive processing on each `Car`, and it becomes vastly more efficient if you know the `Make` and `Model` at compile time.
- You have homogenous lists of the same type of `Car` to process, so that you can store them all in an `Array{Car{:Honda,:Accord},N}`.

When the latter holds, a function processing such a homogenous array can be productively specialized: Julia knows the type of each element in advance (all objects in the container have the same concrete type), so Julia can "look up" the correct method calls when the function is being compiled (obviating the need to check at run-time) and thereby emit efficient code for processing the whole list.

When these do not hold, then it's likely that you'll get no benefit; worse, the resulting "combinatorial explosion of types" will be counterproductive. If `items[i+1]` has a different type than `item[i]`, Julia has to look up the type at run-time, search for the appropriate method in method tables, decide (via type intersection) which one matches, determine whether it has been JIT-compiled yet (and do so if not), and then make the call. In essence, you're asking the full type- system and JIT-compilation machinery to basically execute the equivalent of a switch statement or dictionary lookup in your own code.

Some run-time benchmarks comparing (1) type dispatch, (2) dictionary lookup, and (3) a "switch" statement can be found on the mailing list.

Perhaps even worse than the run-time impact is the compile-time impact: Julia will compile specialized functions for each different `Car{Make, Model}`; if you have hundreds or thousands of such types, then every function that accepts such an object as a parameter (from a custom `get_year` function you might write yourself, to the generic `push!` function in the standard library) will have hundreds or thousands of variants compiled for it. Each of these increases the size of the cache of compiled code, the length of internal lists of methods, etc. Excess enthusiasm for values-as-parameters can easily waste enormous resources.

39.12 Access arrays in memory order, along columns

Multidimensional arrays in Julia are stored in column-major order. This means that arrays are stacked one column at a time. This can be verified using the `vec` function or the syntax `[:]` as shown below (notice that the array is ordered `[1 3 2 4]`, not `[1 2 3 4]`):

```
julia> x = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> x[:]
4-element Array{Int64,1}:
 1
 3
 2
 4
```

This convention for ordering arrays is common in many languages like Fortran, Matlab, and R (to name a few). The alternative to column-major ordering is row-major ordering, which is the convention adopted by C and Python (numpy) among other languages. Remembering the ordering of arrays can have significant performance effects when looping over arrays. A rule of thumb to keep in mind is that with column-major arrays, the first index changes most rapidly. Essentially this means that looping will be faster if the inner-most loop index is the first to appear in a slice expression.

Consider the following contrived example. Imagine we wanted to write a function that accepts a `Vector` and returns a square `Matrix` with either the rows or the columns filled with copies of the input vector. Assume that it is not important whether rows or columns are filled with these copies (perhaps the rest of the code can be easily adapted accordingly). We could conceivably do this in at least four ways (in addition to the recommended call to the built-in `repmat()`):

```
function copy_cols(x::Vector{T}) where T
    n = size(x, 1)
    out = Array{T}(n, n)
    for i = 1:n
        out[:, i] = x
    end
    out
end

function copy_rows(x::Vector{T}) where T
    n = size(x, 1)
    out = Array{T}(n, n)
    for i = 1:n
        out[i, :] = x
    end
    out
end

function copy_col_row(x::Vector{T}) where T
    n = size(x, 1)
    out = Array{T}(n, n)
    for col = 1:n, row = 1:n
        out[row, col] = x[row]
    end
    out
end

function copy_row_col(x::Vector{T}) where T
    n = size(x, 1)
    out = Array{T}(n, n)
    for row = 1:n, col = 1:n
```

```
        out[row, col] = x[col]
    end
    out
end
```

Now we will time each of these functions using the same random `10000` by 1 input vector:

```
julia> x = randn(10000);

julia> fmt(f) = println(rpad(string(f)*": ", 14, ' '), @elapsed f(x))

julia> map(fmt, Any[copy_cols, copy_rows, copy_col_row, copy_row_col]);
copy_cols:    0.331706323
copy_rows:    1.799009911
copy_col_row: 0.415630047
copy_row_col: 1.721531501
```

Notice that `copy_cols` is much faster than `copy_rows`. This is expected because `copy_cols` respects the column-based memory layout of the `Matrix` and fills it one column at a time. Additionally, `copy_col_row` is much faster than `copy_row_col` because it follows our rule of thumb that the first element to appear in a slice expression should be coupled with the inner-most loop.

39.13 Pre-allocating outputs

If your function returns an `Array` or some other complex type, it may have to allocate memory. Unfortunately, oftentimes allocation and its converse, garbage collection, are substantial bottlenecks.

Sometimes you can circumvent the need to allocate memory on each function call by preallocating the output. As a trivial example, compare

```
function xinc(x)
    return [x, x+1, x+2]
end

function loopinc()
    y = 0
    for i = 1:10^7
        ret = xinc(i)
        y += ret[2]
    end
    y
end
```

with

```
function xinc!(ret::AbstractVector{T}, x::T) where T
    ret[1] = x
    ret[2] = x+1
    ret[3] = x+2
    nothing
end

function loopinc_prealloc()
```

```
    ret = Array{Int}(3)
    y = 0
    for i = 1:10^7
        xinc!(ret, i)
        y += ret[2]
    end
    y
end
```

Timing results:

```
julia> @time loopinc()
  0.529894 seconds (40.00 M allocations: 1.490 GiB, 12.14% gc time)
50000015000000

julia> @time loopinc_prealloc()
  0.030850 seconds (6 allocations: 288 bytes)
50000015000000
```

Preallocation has other advantages, for example by allowing the caller to control the "output" type from an algorithm. In the example above, we could have passed a `SubArray` rather than an `Array`, had we so desired.

Taken to its extreme, pre-allocation can make your code uglier, so performance measurements and some judgment may be required. However, for "vectorized" (element-wise) functions, the convenient syntax `x .= f.(y)` can be used for in-place operations with fused loops and no temporary arrays (see the dot syntax for vectorizing functions).

39.14 More dots: Fuse vectorized operations

Julia has a special dot syntax that converts any scalar function into a "vectorized" function call, and any operator into a "vectorized" operator, with the special property that nested "dot calls" are *fusing*: they are combined at the syntax level into a single loop, without allocating temporary arrays. If you use `.=` and similar assignment operators, the result can also be stored in-place in a pre-allocated array (see above).

In a linear-algebra context, this means that even though operations like `vector + vector` and `vector * scalar` are defined, it can be advantageous to instead use `vector .+ vector` and `vector .* scalar` because the resulting loops can be fused with surrounding computations. For example, consider the two functions:

```
f(x) = 3x.^2 + 4x + 7x.^3

fdot(x) = @. 3x^2 + 4x + 7x^3 # equivalent to 3 .* x.^2 .+ 4 .* x .+ 7 .* x.^3
```

Both `f` and `fdot` compute the same thing. However, `fdot` (defined with the help of the `@.` macro) is significantly faster when applied to an array:

```
julia> x = rand(10^6);

julia> @time f(x);
  0.010986 seconds (18 allocations: 53.406 MiB, 11.45% gc time)

julia> @time fdot(x);
  0.003470 seconds (6 allocations: 7.630 MiB)

julia> @time f.(x);
  0.003297 seconds (30 allocations: 7.631 MiB)
```

That is, `fdot(x)` is three times faster and allocates 1/7 the memory of `f(x)`, because each `*` and `+` operation in `f(x)` allocates a new temporary array and executes in a separate loop. (Of course, if you just do `f.(x)` then it is as fast as `fdot(x)` in this example, but in many contexts it is more convenient to just sprinkle some dots in your expressions rather than defining a separate function for each vectorized operation.)

39.15 Consider using views for slices

In Julia, an array "slice" expression like `array[1:5, :]` creates a copy of that data (except on the left-hand side of an assignment, where `array[1:5, :] = ...` assigns in-place to that portion of `array`). If you are doing many operations on the slice, this can be good for performance because it is more efficient to work with a smaller contiguous copy than it would be to index into the original array. On the other hand, if you are just doing a few simple operations on the slice, the cost of the allocation and copy operations can be substantial.

An alternative is to create a "view" of the array, which is an array object (a `SubArray`) that actually references the data of the original array in-place, without making a copy. (If you write to a view, it modifies the original array's data as well.) This can be done for individual slices by calling `view()`, or more simply for a whole expression or block of code by putting `@views` in front of that expression. For example:

```
julia> fcopy(x) = sum(x[2:end-1])

julia> @views fview(x) = sum(x[2:end-1])

julia> x = rand(10^6);

julia> @time fcopy(x);
  0.003051 seconds (7 allocations: 7.630 MB)

julia> @time fview(x);
  0.001020 seconds (6 allocations: 224 bytes)
```

Notice both the 3× speedup and the decreased memory allocation of the `fview` version of the function.

39.16 Avoid string interpolation for I/O

When writing data to a file (or other I/O device), forming extra intermediate strings is a source of overhead. Instead of:

```
println(file, "$a $b")
```

use:

```
println(file, a, " ", b)
```

The first version of the code forms a string, then writes it to the file, while the second version writes values directly to the file. Also notice that in some cases string interpolation can be harder to read. Consider:

```
println(file, "$(f(a))$(f(b))")
```

versus:

```
println(file, f(a), f(b))
```

39.17 Optimize network I/O during parallel execution

When executing a remote function in parallel:

```
responses = Vector{Any}(nworkers())
@sync begin
    for (idx, pid) in enumerate(workers())
        @async responses[idx] = remotecall_fetch(pid, foo, args...)
    end
end
```

is faster than:

```
refs = Vector{Any}(nworkers())
for (idx, pid) in enumerate(workers())
    refs[idx] = @spawnat pid foo(args...)
end
responses = [fetch(r) for r in refs]
```

The former results in a single network round-trip to every worker, while the latter results in two network calls - first by the `@spawnat` and the second due to the `fetch` (or even a `wait`). The `fetch`/`wait` is also being executed serially resulting in an overall poorer performance.

39.18 Fix deprecation warnings

A deprecated function internally performs a lookup in order to print a relevant warning only once. This extra lookup can cause a significant slowdown, so all uses of deprecated functions should be modified as suggested by the warnings.

39.19 Tweaks

These are some minor points that might help in tight inner loops.

- Avoid unnecessary arrays. For example, instead of `sum([x,y,z])` use x+y+z.
- Use `abs2(z)` instead of `abs(z)^2` for complex z. In general, try to rewrite code to use `abs2()` instead of `abs()` for complex arguments.
- Use `div(x,y)` for truncating division of integers instead of `trunc(x/y)`, `fld(x,y)` instead of `floor(x/y)`, and `cld(x,y)` instead of `ceil(x/y)`.

39.20 Performance Annotations

Sometimes you can enable better optimization by promising certain program properties.

- Use `@inbounds` to eliminate array bounds checking within expressions. Be certain before doing this. If the subscripts are ever out of bounds, you may suffer crashes or silent corruption.
- Use `@fastmath` to allow floating point optimizations that are correct for real numbers, but lead to differences for IEEE numbers. Be careful when doing this, as this may change numerical results. This corresponds to the `-ffast-math` option of clang.

- Write `@simd` in front of `for` loops that are amenable to vectorization. **This feature is experimental** and could change or disappear in future versions of Julia.

Note: While `@simd` needs to be placed directly in front of a loop, both `@inbounds` and `@fastmath` can be applied to several statements at once, e.g. using `begin` ... end, or even to a whole function.

Here is an example with both `@inbounds` and `@simd` markup:

```
function inner(x, y)
    s = zero(eltype(x))
    for i=1:length(x)
        @inbounds s += x[i]*y[i]
    end
    s
end

function innersimd(x, y)
    s = zero(eltype(x))
    @simd for i=1:length(x)
        @inbounds s += x[i]*y[i]
    end
    s
end

function timeit(n, reps)
    x = rand(Float32,n)
    y = rand(Float32,n)
    s = zero(Float64)
    time = @elapsed for j in 1:reps
        s+=inner(x,y)
    end
    println("GFlop/sec        = ",2.0*n*reps/time*1E-9)
    time = @elapsed for j in 1:reps
        s+=innersimd(x,y)
    end
    println("GFlop/sec (SIMD) = ",2.0*n*reps/time*1E-9)
end

timeit(1000,1000)
```

On a computer with a 2.4GHz Intel Core i5 processor, this produces:

```
GFlop/sec        = 1.9467069505224963
GFlop/sec (SIMD) = 17.578554163920018
```

(`GFlop/sec` measures the performance, and larger numbers are better.) The range for a `@simd for` loop should be a one-dimensional range. A variable used for accumulating, such as `s` in the example, is called a *reduction variable*. By using `@simd`, you are asserting several properties of the loop:

- It is safe to execute iterations in arbitrary or overlapping order, with special consideration for reduction variables.
- Floating-point operations on reduction variables can be reordered, possibly causing different results than without `@simd`.
- No iteration ever waits on another iteration to make forward progress.

A loop containing `break`, `continue`, or `@goto` will cause a compile-time error.

Using `@simd` merely gives the compiler license to vectorize. Whether it actually does so depends on the compiler. To actually benefit from the current implementation, your loop should have the following additional properties:

- The loop must be an innermost loop.
- The loop body must be straight-line code. This is why `@inbounds` is currently needed for all array accesses. The compiler can sometimes turn short `&&`, `||`, and `?:` expressions into straight-line code, if it is safe to evaluate all operands unconditionally. Consider using `ifelse()` instead of `?:` in the loop if it is safe to do so.
- Accesses must have a stride pattern and cannot be ”gathers” (random-index reads) or ”scatters” (random-index writes).
- The stride should be unit stride.
- In some simple cases, for example with 2-3 arrays accessed in a loop, the LLVM auto-vectorization may kick in automatically, leading to no further speedup with `@simd`.

Here is an example with all three kinds of markup. This program first calculates the finite difference of a one-dimensional array, and then evaluates the L2-norm of the result:

```
function init!(u)
    n = length(u)
    dx = 1.0 / (n-1)
    @fastmath @inbounds @simd for i in 1:n
        u[i] = sin(2pi*dx*i)
    end
end

function deriv!(u, du)
    n = length(u)
    dx = 1.0 / (n-1)
    @fastmath @inbounds du[1] = (u[2] - u[1]) / dx
    @fastmath @inbounds @simd for i in 2:n-1
        du[i] = (u[i+1] - u[i-1]) / (2*dx)
    end
    @fastmath @inbounds du[n] = (u[n] - u[n-1]) / dx
end

function norm(u)
    n = length(u)
    T = eltype(u)
    s = zero(T)
    @fastmath @inbounds @simd for i in 1:n
        s += u[i]^2
    end
    @fastmath @inbounds return sqrt(s/n)
end

function main()
    n = 2000
    u = Array{Float64}(n)
    init!(u)
    du = similar(u)
```

```
    deriv!(u, du)
    nu = norm(du)

    @time for i in 1:10^6
        deriv!(u, du)
        nu = norm(du)
    end

    println(nu)
end

main()
```

On a computer with a 2.7 GHz Intel Core i7 processor, this produces:

```
$ julia wave.jl;
elapsed time: 1.207814709 seconds (0 bytes allocated)

$ julia --math-mode=ieee wave.jl;
elapsed time: 4.487083643 seconds (0 bytes allocated)
```

Here, the option `--math-mode=ieee` disables the `@fastmath` macro, so that we can compare results.

In this case, the speedup due to `@fastmath` is a factor of about 3.7. This is unusually large – in general, the speedup will be smaller. (In this particular example, the working set of the benchmark is small enough to fit into the L1 cache of the processor, so that memory access latency does not play a role, and computing time is dominated by CPU usage. In many real world programs this is not the case.) Also, in this case this optimization does not change the result – in general, the result will be slightly different. In some cases, especially for numerically unstable algorithms, the result can be very different.

The annotation `@fastmath` re-arranges floating point expressions, e.g. changing the order of evaluation, or assuming that certain special cases (inf, nan) cannot occur. In this case (and on this particular computer), the main difference is that the expression `1 / (2*dx)` in the function `deriv` is hoisted out of the loop (i.e. calculated outside the loop), as if one had written `idx = 1 / (2*dx)`. In the loop, the expression `... / (2*dx)` then becomes `... * idx`, which is much faster to evaluate. Of course, both the actual optimization that is applied by the compiler as well as the resulting speedup depend very much on the hardware. You can examine the change in generated code by using Julia's `code_native()` function.

39.21 Treat Subnormal Numbers as Zeros

Subnormal numbers, formerly called denormal numbers, are useful in many contexts, but incur a performance penalty on some hardware. A call `set_zero_subnormals(true)` grants permission for floating-point operations to treat subnormal inputs or outputs as zeros, which may improve performance on some hardware. A call `set_zero_subnormals(false)` enforces strict IEEE behavior for subnormal numbers.

Below is an example where subnormals noticeably impact performance on some hardware:

```
function timestep(b::Vector{T}, a::Vector{T}, Δt::T) where T
    @assert length(a)==length(b)
    n = length(b)
    b[1] = 1                          # Boundary condition
    for i=2:n-1
        b[i] = a[i] + (a[i-1] - T(2)*a[i] + a[i+1]) * Δt
    end
```

```
    b[n] = 0                            # Boundary condition
end

function heatflow(a::Vector{T}, nstep::Integer) where T
    b = similar(a)
    for t=1:div(nstep,2)                # Assume nstep is even
        timestep(b,a,T(0.1))
        timestep(a,b,T(0.1))
    end
end

heatflow(zeros(Float32,10),2)           # Force compilation
for trial=1:6
    a = zeros(Float32,1000)
    set_zero_subnormals(iseven(trial))  # Odd trials use strict IEEE arithmetic
    @time heatflow(a,1000)
end
```

This example generates many subnormal numbers because the values in a become an exponentially decreasing curve, which slowly flattens out over time.

Treating subnormals as zeros should be used with caution, because doing so breaks some identities, such as `x-y == 0` implies `x == y`:

```
julia> x = 3f-38; y = 2f-38;

julia> set_zero_subnormals(true); (x - y, x == y)
(0.0f0, false)

julia> set_zero_subnormals(false); (x - y, x == y)
(1.0000001f-38, false)
```

In some applications, an alternative to zeroing subnormal numbers is to inject a tiny bit of noise. For example, instead of initializing a with zeros, initialize it with:

```
a = rand(Float32,1000) * 1.f-9
```

39.22 `@code_warntype`

The macro `@code_warntype` (or its function variant `code_warntype()`) can sometimes be helpful in diagnosing type-related problems. Here's an example:

```
pos(x) = x < 0 ? 0 : x

function f(x)
    y = pos(x)
    sin(y*x+1)
end

julia> @code_warntype f(3.2)
Variables:
  #self#::#f
  x::Float64
```

```
  y::UNION{FLOAT64,INT64}
  fy::Float64
  #temp#@_5::UNION{FLOAT64,INT64}
  #temp#@_6::Core.MethodInstance
  #temp#@_7::Float64

Body:
  begin
      $(Expr(:inbounds, false))
      # meta: location REPL[1] pos 1
      # meta: location float.jl < 487
      fy::Float64 = (Core.typeassert)((Base.sitofp)(Float64,0)::Float64,Float64)::Float64
      # meta: pop location
      unless
      ↪  (Base.or_int)((Base.lt_float)(x::Float64,fy::Float64)::Bool,(Base.and_int)((Base.and_int)((Base.eq_float
      ↪  goto 9
      #temp#@_5::UNION{FLOAT64,INT64} = 0
      goto 11
      9:
      #temp#@_5::UNION{FLOAT64,INT64} = x::Float64
      11:
      # meta: pop location
      $(Expr(:inbounds, :pop))
      y::UNION{FLOAT64,INT64} = #temp#@_5::UNION{FLOAT64,INT64} # line 3:
      unless (y::UNION{FLOAT64,INT64} isa Int64)::ANY goto 19
      #temp#@_6::Core.MethodInstance = MethodInstance for *(::Int64, ::Float64)
      goto 28
      19:
      unless (y::UNION{FLOAT64,INT64} isa Float64)::ANY goto 23
      #temp#@_6::Core.MethodInstance = MethodInstance for *(::Float64, ::Float64)
      goto 28
      23:
      goto 25
      25:
      #temp#@_7::Float64 = (y::UNION{FLOAT64,INT64} * x::Float64)::Float64
      goto 30
      28:
      #temp#@_7::Float64 = $(Expr(:invoke, :(#temp#@_6), :(Main.*), :(y), :(x)))
      30:
      return $(Expr(:invoke, MethodInstance for sin(::Float64), :(Main.sin),
      ↪  :((Base.add_float)(#temp#@_7,(Base.sitofp)(Float64,1)::Float64)::Float64)))
  end::Float64
```

Interpreting the output of `@code_warntype`, like that of its cousins `@code_lowered`, `@code_typed`, `@code_llvm`, and `@code_native`, takes a little practice. Your code is being presented in form that has been partially digested on its way to generating compiled machine code. Most of the expressions are annotated by a type, indicated by the `::T` (where `T` might be `Float64`, for example). The most important characteristic of `@code_warntype` is that non-concrete types are displayed in red; in the above example, such output is shown in all-caps.

The top part of the output summarizes the type information for the different variables internal to the function. You can see that `y`, one of the variables you created, is a `Union{Int64,Float64}`, due to the type-instability of `pos`. There is another variable, `_var4`, which you can see also has the same type.

The next lines represent the body of `f`. The lines starting with a number followed by a colon (`1:`, `2:`) are labels, and represent targets for jumps (via `goto`) in your code. Looking at the body, you can see that `pos` has been *inlined* into `f`–everything before `2:` comes from code defined in `pos`.

Starting at `2 :`, the variable `y` is defined, and again annotated as a `Union` type. Next, we see that the compiler created the temporary variable `_var1` to hold the result of `y*x`. Because a `Float64` times *either* an `Int64` or `Float64` yields a `Float64`, all type-instability ends here. The net result is that `f(x::Float64)` will not be type-unstable in its output, even if some of the intermediate computations are type-unstable.

How you use this information is up to you. Obviously, it would be far and away best to fix `pos` to be type-stable: if you did so, all of the variables in `f` would be concrete, and its performance would be optimal. However, there are circumstances where this kind of *ephemeral* type instability might not matter too much: for example, if `pos` is never used in isolation, the fact that `f`'s output is type-stable (for `Float64` inputs) will shield later code from the propagating effects of type instability. This is particularly relevant in cases where fixing the type instability is difficult or impossible: for example, currently it's not possible to infer the return type of an anonymous function. In such cases, the tips above (e.g., adding type annotations and/or breaking up functions) are your best tools to contain the "damage" from type instability.

The following examples may help you interpret expressions marked as containing non-leaf types:

- Function body ending in `end::Union{T1,T2})`
  - Interpretation: function with unstable return type
  - Suggestion: make the return value type-stable, even if you have to annotate it
- `f(x::T)::Union{T1,T2}`
  - Interpretation: call to a type-unstable function
  - Suggestion: fix the function, or if necessary annotate the return value
- `(top(arrayref))(A::Array{Any,1},1)::Any`
  - Interpretation: accessing elements of poorly-typed arrays
  - Suggestion: use arrays with better-defined types, or if necessary annotate the type of individual element accesses
- `(top(getfield))(A::ArrayContainer{Float64},:data)::Array{Float64,N}`
  - Interpretation: getting a field that is of non-leaf type. In this case, `ArrayContainer` had a field `data::Array{T}`. But `Array` needs the dimension `N`, too, to be a concrete type.
  - Suggestion: use concrete types like `Array{T,3}` or `Array{T,N}`, where `N` is now a parameter of `ArrayContainer`

Chapter 40

Workflow Tips

Here are some tips for working with Julia efficiently.

40.1 REPL-based workflow

As already elaborated in Interacting With Julia, Julia's REPL provides rich functionality that facilitates an efficient interactive workflow. Here are some tips that might further enhance your experience at the command line.

A basic editor/REPL workflow

The most basic Julia workflows involve using a text editor in conjunction with the `julia` command line. A common pattern includes the following elements:

- **Put code under development in a temporary module.** Create a file, say `Tmp.jl`, and include within it

  ```
  module Tmp

  <your definitions here>

  end
  ```

- **Put your test code in another file.** Create another file, say `tst.jl`, which begins with

  ```
  import Tmp
  ```

  and includes tests for the contents of `Tmp`. The value of using `import` versus `using` is that you can call `reload("Tmp")` instead of having to restart the REPL when your definitions change. Of course, the cost is the need to prepend `Tmp.` to uses of names defined in your module. (You can lower that cost by keeping your module name short.)

  Alternatively, you can wrap the contents of your test file in a module, as

  ```
  module Tst
      using Tmp

      <scratch work>

  end
  ```

  The advantage is that you can now do `using Tmp` in your test code and can therefore avoid prepending `Tmp.` everywhere. The disadvantage is that code can no longer be selectively copied to the REPL without some tweaking.

- **Lather. Rinse. Repeat.** Explore ideas at the `julia` command prompt. Save good ideas in `tst.jl`. Occasionally restart the REPL, issuing

```
reload("Tmp")
include("tst.jl")
```

Simplify initialization

To simplify restarting the REPL, put project-specific initialization code in a file, say `_init.jl`, which you can run on startup by issuing the command:

```
julia -L _init.jl
```

If you further add the following to your `.juliarc.jl` file

```
isfile("_init.jl") && include(joinpath(pwd(), "_init.jl"))
```

then calling `julia` from that directory will run the initialization code without the additional command line argument.

40.2 Browser-based workflow

It is also possible to interact with a Julia REPL in the browser via IJulia. See the package home for details.

Chapter 41

Guía de Estilo

Las siguientes secciones explican unos cuantos aspectos del estilo de codificación idiomático de Julia. Ninguna de estas reglas son absolutas; sólo son sugerencias para ayudar a familiarizarte con el lenguaje y ayudarte a elegir entre diseños alternativos.

41.1 Escribe funciones, no sólo *scripts*

Escribir código como una serie de pasos a nivel superior Es una forma rápida de empezar a resolver un problema, pero uno debería intentar dividir un programa en funciones tan pronto como sea posible. La función son más reusabes y testables, y clarifican qué pasos se están dando y cuáles son sus entradas y sus salidas. Además, el código dentro de las funciones tiende a ejecutar mucho más rápido que el código de nivel superior debido a cómo funciona el compilador de Julia.

También merece la pena señalar que las funciones deberían tomar argumentos, en lugar de operar directamente sobre las variables globales (aparte de constantes como `pi`).

41.2 Evita escribir tipos demasiado específicos

El código debería ser tan genérico como sea posible. En lugar de escribir:

```
convert(Complex{Float64}, x)
```

es mejor usar funciones genéricas disponibles:

```
complex(float(x))
```

La segunda versión convertirá x a un tipo apropiado, en lugrar de siempre al mismo tipo.

Este punto de estilo es especialmente relevante para los argumentos de función. Por ejemplo, no declare que un argumento sea de tipo `Int` o `Int32` si realmente pudiera ser cualquier número entero, expresado con el tipo abstracto `Integer`. De hecho, en muchos casos puede omitir el tipo de argumento por completo, a menos que sea necesario para eliminar la ambigüedad de otras definiciones de método, ya que se lanzará `MethodError` de todos modos si se pasa un tipo que no admite ninguna de las operaciones requeridas. (Esto se conoce como *duck typing*.)

Por ejemplo, considere las siguientes definiciones de una función addone que devuelve uno más su argumento:

```
addone(x::Int) = x + 1                # works only for Int
addone(x::Integer) = x + oneunit(x)   # any integer type
addone(x::Number) = x + oneunit(x)    # any numeric type
addone(x) = x + oneunit(x)            # any type supporting + and oneunit
```

La última definición de addone maneja cualquier tipo que soporte `oneunit` (que devuelve 1 en el mismo tipo que x, lo que evita la promoción de tipos no deseados) y la función `+` con esos argumentos. La clave para darse cuenta es que no hay *ninguna penalización de rendimiento* para definir *solo* el general `addone(x) = x + oneunit(x)`, porque Julia compilará automáticamente versiones especializadas según sea necesario. Por ejemplo, la primera vez que llame a `addone(12)`, Julia compilará automáticamente una función especializada addone para argumentos `x::Int`, reemplazando la llamada a `oneunit` por su valor1. Por lo tanto, las primeras tres definiciones de 'addone' anteriores son completamente redundantes con la cuarta definición.

41.3 Manejar el exceso de diversidad de argumentos en el "código llamador"

En lugar de:

```
function foo(x, y)
    x = Int(x); y = Int(y)
    ...
end
foo(x, y)
```

use:

```
function foo(x::Int, y::Int)
    ...
end
foo(Int(x), Int(y))
```

Este es mucho mejor estilo debido a que `foo` no acepta realmente números de todos los tipos, sino que necesita `Int`s.

Un problema aquí es que si una función requiere números enteros intrínsecamente, podría ser mejor forzar al autor de la llamada a decidir cómo deberían convertirse los no enteros (por ejemplo, redondeando por abajo o por arriba). Otro problema es que la declaración de tipos más específicos deja más "espacio" para las futuras definiciones de métodos.

41.4 Añadir ! para los nombres de funciones que modifican sus argumentos

En lugar de:

```
function double(a::AbstractArray{<:Number})
    for i = 1:endof(a)
        a[i] *= 2
    end
    return a
end
```

use:

```
function double!(a::AbstractArray{<:Number})
    for i = 1:endof(a)
        a[i] *= 2
    end
    return a
end
```

La biblioteca estándar de Julia usa esta convención y contiene ejemplos de funciones con formas tanto de copiado como de modificación (por ejemplo, `sort()` y `sort!()`), y otras que simplemente están modificando (por ejemplo, `push!()`, `pop!()`, `splice!()`). Es típico para tales funciones devolver también la matriz modificada por conveniencia.

41.5 Evitar tipos `Union` extraños

Tipos tales como `Union{Function,AbstractString}` son frecuentemente un signo de que hay que limpiar algo en el diseño.

41.6 Evitar las Uniones de tipos en campos

Cuando se crea un tipo tal como :

```
mutable struct MyType
    ...
    x::Union{Void,T}
end
```

pregunte si la opción de `x` para ser `nada` (de tipo `Void`) es realmente necesaria. Aquí hay algunas alternativas a considerar:

- Encuentre un valor predeterminado seguro con el que inicializar `x`
- Introduce otro tipo del que carece `x`
- Si hay muchos campos como `x`, guárdelos en un diccionario
- Determine si hay una regla simple para cuando `x` es`nada`. Por ejemplo, a menudo el campo comenzará como `nada` pero se inicializará en algún punto bien definido. En ese caso, considere dejarlo indefinido al principio.
- Si `x` realmente no necesita contener ningún valor en algún momento, defínalo como `::Nullable{T}` en su lugar, ya que esto garantiza estabilidad de tipo en el código que accede a este campo (ver TiposNullable).

41.7 Evitar elaborar tipos contenedor

Usualmente no es de mucha ayuda construir arrays como los siguientes:

```
a = Array{Union{Int,AbstractString,Tuple,Array}}(n)
```

En este caso `Array{Any}(n)` es mejor. Es también de más ayuda para el compilador anotar usos específicos (por ejemplo `a[i]::Int`) que intentar empaquetar muchas alternativas en un tipo.

41.8 Usar convenciones de nombrado consistentes con el paquete `base/` de Julia

- Los nombres de los módulos y tipos usan mayúsculas y *came case*: `module SparseArrays`, `struct UnitRange`.
- Las funciones van en minúscula (`maximum()`, `convert()`) y, cuando es legible, con múltiples palabras pegadas juntas (`isequal()`, `haskey()`). Cuando sea necesario, use guiones bajos como separadores de palabra. Los guiones bajos también se usan para indicar una combinacin de conceptos (`remotecall_fetch()` como una implementación más eficiente de `fetch(remotecall(...))`) o como modificadores (`sum_kbn()`).

- Se valora la concisión, pero debe evitarse la abreviatura (`indexin()` en lugar de `indxin()`) ya que se vuelve difícil recordar si se abrevian palabras particulares y cómo se han abreviado.

Si el nombre de una función requiere varias palabras, considere si podría representar más de un concepto y podría dividirse mejor en partes.

41.9 No usar demasiado try-catch

Es mejor evitar errores que basarse en atraparlos.

41.10 No meter entre paréntesis las condiciones

Julia no necesita que se rodeen entre paréntesis las condiciones en `if` and `while`. Escriba:

```
if a == b
```

en lugar de:

```
if (a == b)
```

41.11 No usar demasiado ...

El uso de `...` en los argumentos de función puede ser adictivo. En lugar de `[a..., b...]`, use `[a; b]`, que ya concatena arrays. `collect(a)` es mejor que `[a...]`, pero como a ya es iterable suele ser incluso mejor dejarlo solo, y no convertirlo en array.

41.12 No usar parámetros estáticos innecesarios

Una signatura de función:

```
foo(x::T) where {T<:Real} = ...
```

debería ser escrita como

```
foo(x::Real) = ...
```

especialmente si T no se usa en el cuerpo de la función. Incluso si se usa T, se puede reemplazar con `typeof(x)` si es conveniente. No hay diferencia de rendimiento. Tenga en cuenta que esto no es una precaución general contra los parámetros estáticos, solo contra uso donde no son necesarios.

Tenga en cuenta también que los tipos de contenedores, específicamente pueden necesitar parámetros de tipo en las llamadas a función. Consulte las Preguntas frecuentes Evitar campos con contenedores abstractos para obtener más información.

41.13 Evitar la confusion sobre si algo es una instancia o un tipo

Conjuntos de definiciones como las siguientes son confusas:

```
foo(::Type{MyType}) = ...
foo(::MyType) = foo(MyType)
```

Decida si el concepto en cuestión se escribirá como `MyType` o `MyType()`, y sígalo.

El estilo preferido es usar instancias por defecto, y solo agregue métodos que incluyan `Type{MyType}` más tarde si se vuelven necesarios para resolver algún problema.

Si un tipo es efectivamente una enumeración, debe definirse como un tipo único (idealmente, `immutable struct` o primitivo), con los valores de enumeración como instancias de este. Los constructores y las conversiones pueden verificar si los valores son válidos. Este diseño es preferible a hacer que la enumeración sea un tipo abstracto, con los ”valores” como subtipos.

41.14 No abusar de las macros

Tenga en cuenta cuando una macro realmente podría ser una función en su lugar.

Llamar a `eval()` dentro de una macro es un signo de advertencia particularmente peligroso; significa que la macro solo funcionará cuando se llame al nivel superior. Si tal macro se escribe como una función en su lugar, naturalmente tendrá acceso a los valores en tiempo de ejecución que necesita.

41.15 No exponer operaciones inseguras al nivel de interfaz

Si se tiene un tipo que use un puntero nativo:

```
mutable struct NativeType
    p::Ptr{UInt8}
    ...
end
```

no escriba definiciones como la siguiente:

```
getindex(x::NativeType, i) = unsafe_load(x.p, i)
```

El problema es que los usuarios de este tipo pueden escribir `x[i]` sin darse cuenta de que la operación no es segura y, luego, ser susceptibles a errores de memoria.

Dicha función debería verificar la operación para asegurarse de que sea segura, o incluir `unsafe` en alguna parte de su nombre para alertar a las personas que la invocan.

41.16 No sobrecargar métodos de tipos de contenedores base

Es posible escribir definiciones como la siguiente:

```
show(io::IO, v::Vector{MyType}) = ...
```

Esto proporcionaría una muestra personalizada de vectores con un nuevo tipo de elemento específico. Aunque es tentador, es algo que debe evitarse. El problema es que los usuarios esperarán que un tipo conocido como `Vector()` se comporte de cierta manera, y la personalización excesiva de su comportamiento puede dificultar el trabajo.

41.17 Evitar la piratería de tipos

La "Piratería de Tipos" se refiere a la práctica de extender o redefinir métodos en Base u otros paquetes en tipos que no han definido. En algunos casos, puede la piratería de tipo va a tener un efecto poco negativo. Sin embargo, en casos extremos, incluso puede bloquear Julia (por ejemplo, si la extensión o redefinición de su método hace que se pase una entrada inválida a `ccall`). La piratería de tipos puede complicar el razonamiento sobre el código y puede introducir incompatibilidades que son difíciles de predecir y diagnosticar.

Como ejemplo, suponga que quiere definir la multiplicación en símbolos en un módulo:

```
module A
import Base.*
*(x::Symbol, y::Symbol) = Symbol(x,y)
end
```

El problema es que ahora cualquier otro módulo que use `Base.*` También verá esta definición. Dado que `Symbol` se define en Base y es utilizado por otros módulos, esto puede cambiar el comportamiento del código no relacionado de forma inesperada. Aquí hay varias alternativas, incluido el uso de un nombre de función diferente o el ajuste de `Symbol`s en otro tipo que defina.

Algunas veces, los paquetes acoplados pueden involucrarse en la piratería de tipos para separar las características de las definiciones, especialmente cuando los paquetes fueron diseñados por autores colaboradores, y cuando las definiciones son reutilizables. Por ejemplo, un paquete puede proporcionar algunos tipos útiles para trabajar con colores; otro paquete podría definir métodos para aquellos tipos que permiten conversiones entre espacios de color. Otro ejemplo podría ser un paquete que actúa como un envoltorio delgado para algún código C, que otro paquete podría piratear para implementar una API de nivel superior compatible con Julia.

41.18 Ser cuidadoso con la igualdad de tipos

Por lo general, uno desea utilizar `isa()` y `<:` para los tipos de prueba, no `==`. La comprobación de los tipos para la igualdad exacta normalmente solo tiene sentido cuando se compara con un tipo concreto conocido (por ejemplo, `T == Float64`), o si *realmente* uno sabe lo que está haciendo.

41.19 No escribir `x->f(x)`

Como las funciones de orden superior a menudo se llaman con funciones anónimas, es fácil concluir que esto es deseable o incluso necesario. Pero cualquier función se puede pasar directamente, sin estar "envuelta" en una función anónima. En lugar de escribir `map (x-> f(x), a)`, escriba `map(f, a)`.

41.20 Evitar usar floats para literales numericos en codigo generico cuando sea posible

Si escribe código genérico que maneja números, y que se puede esperar que se ejecute con muchos tipos de argumentos numéricos diferentes, intente utilizar literales de un tipo numérico que afectarán los argumentos lo menos posible mediante la promoción.

Por ejemplo,

```
julia> f(x) = 2.0 * x
f (generic function with 1 method)

julia> f(1//2)
1.0
```

```
julia> f(1/2)
1.0

julia> f(1)
2.0
```

mientras que

```
julia> g(x) = 2 * x
g (generic function with 1 method)

julia> g(1//2)
1//1

julia> g(1/2)
1.0

julia> g(1)
2
```

Como puede ver, la segunda versión, donde usamos un literal `Int`, conserva el tipo de argumento de entrada, mientras que la primera no. Esto se debe a, por ejemplo, `promote_type(Int, Float64) == Float64`, y la promoción ocurre con la multiplicación. De manera similar, los literales `Rational` son menos disruptivos que los literales `Float64`, pero son más perjudiciales que los `Int`s:

```
julia> h(x) = 2//1 * x
h (generic function with 1 method)

julia> h(1//2)
1//1

julia> h(1/2)
1.0

julia> h(1)
2//1
```

Por tanto, use literales `Int` cuando sea posible, con `Rational{Int}` para literales numéricos no enteros, en orden a hacer nuestro código ms fácil de usar.

Chapter 42

Frequently Asked Questions

42.1 Sessions and the REPL

How do I delete an object in memory?

Julia does not have an analog of MATLAB's `clear` function; once a name is defined in a Julia session (technically, in module `Main`), it is always present.

If memory usage is your concern, you can always replace objects with ones that consume less memory. For example, if `A` is a gigabyte-sized array that you no longer need, you can free the memory with `A = 0`. The memory will be released the next time the garbage collector runs; you can force this to happen with `gc()`.

How can I modify the declaration of a type in my session?

Perhaps you've defined a type and then realize you need to add a new field. If you try this at the REPL, you get the error:

```
ERROR: invalid redefinition of constant MyType
```

Types in module `Main` cannot be redefined.

While this can be inconvenient when you are developing new code, there's an excellent workaround. Modules can be replaced by redefining them, and so if you wrap all your new code inside a module you can redefine types and constants. You can't import the type names into `Main` and then expect to be able to redefine them there, but you can use the module name to resolve the scope. In other words, while developing you might use a workflow something like this:

```
include("mynewcode.jl")              # this defines a module MyModule
obj1 = MyModule.ObjConstructor(a, b)
obj2 = MyModule.somefunction(obj1)
# Got an error. Change something in "mynewcode.jl"
include("mynewcode.jl")              # reload the module
obj1 = MyModule.ObjConstructor(a, b) # old objects are no longer valid, must reconstruct
obj2 = MyModule.somefunction(obj1)   # this time it worked!
obj3 = MyModule.someotherfunction(obj2, c)
...
```

42.2 Functions

I passed an argument x to a function, modified it inside that function, but on the outside,

the variable x is still unchanged. Why?

Suppose you call a function like this:

```
julia> x = 10
10

julia> function change_value!(y)

           y = 17

       end
change_value! (generic function with 1 method)

julia> change_value!(x)
17

julia> x # x is unchanged!
10
```

In Julia, the binding of a variable `x` cannot be changed by passing `x` as an argument to a function. When calling `change_value!(x)` in the above example, `y` is a newly created variable, bound initially to the value of `x`, i.e. `10`; then `y` is rebound to the constant 17, while the variable `x` of the outer scope is left untouched.

But here is a thing you should pay attention to: suppose `x` is bound to an object of type `Array` (or any other *mutable* type). From within the function, you cannot "unbind" `x` from this Array, but you can change its content. For example:

```
julia> x = [1,2,3]
3-element Array{Int64,1}:
 1
 2
 3

julia> function change_array!(A)

           A[1] = 5

       end
change_array! (generic function with 1 method)

julia> change_array!(x)
5

julia> x
3-element Array{Int64,1}:
 5
 2
 3
```

Here we created a function `change_array!()`, that assigns 5 to the first element of the passed array (bound to `x` at the call site, and bound to `A` within the function). Notice that, after the function call, `x` is still bound to the same array, but the content of that array changed: the variables `A` and `x` were distinct bindings refering to the same mutable `Array` object.

Can I use `using` or `import` inside a function?

No, you are not allowed to have a `using` or `import` statement inside a function. If you want to import a module but only use its symbols inside a specific function or set of functions, you have two options:

1. Use `import`:

   ```
   import Foo
   function bar(...)
       # ... refer to Foo symbols via Foo.baz ...
   end
   ```

   This loads the module Foo and defines a variable Foo that refers to the module, but does not import any of the other symbols from the module into the current namespace. You refer to the Foo symbols by their qualified names `Foo.bar` etc.

2. Wrap your function in a module:

   ```
   module Bar
   export bar
   using Foo
   function bar(...)
       # ... refer to Foo.baz as simply baz ....
   end
   end
   using Bar
   ```

   This imports all the symbols from Foo, but only inside the module `Bar`.

What does the `...` operator do?

The two uses of the `...` operator: slurping and splatting

Many newcomers to Julia find the use of `...` operator confusing. Part of what makes the `...` operator confusing is that it means two different things depending on context.

`...` combines many arguments into one argument in function definitions

In the context of function definitions, the `...` operator is used to combine many different arguments into a single argument. This use of `...` for combining many different arguments into a single argument is called slurping:

```
julia> function printargs(args...)

           @printf("%s\n", typeof(args))

           for (i, arg) in enumerate(args)

               @printf("Arg %d = %s\n", i, arg)

           end

       end
printargs (generic function with 1 method)

julia> printargs(1, 2, 3)
```

```
Tuple{Int64,Int64,Int64}
Arg 1 = 1
Arg 2 = 2
Arg 3 = 3
```

If Julia were a language that made more liberal use of ASCII characters, the slurping operator might have been written as `<-...` instead of `....`

`...` splits one argument into many different arguments in function calls

In contrast to the use of the `...` operator to denote slurping many different arguments into one argument when defining a function, the `...` operator is also used to cause a single function argument to be split apart into many different arguments when used in the context of a function call. This use of `...` is called splatting:

```
julia> function threeargs(a, b, c)

           @printf("a = %s::%s\n", a, typeof(a))

           @printf("b = %s::%s\n", b, typeof(b))

           @printf("c = %s::%s\n", c, typeof(c))

       end
threeargs (generic function with 1 method)

julia> vec = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> threeargs(vec...)
a = 1::Int64
b = 2::Int64
c = 3::Int64
```

If Julia were a language that made more liberal use of ASCII characters, the splatting operator might have been written as `...->` instead of `....`

42.3 Types, type declarations, and constructors

What does ”type-stable” mean?

It means that the type of the output is predictable from the types of the inputs. In particular, it means that the type of the output cannot vary depending on the *values* of the inputs. The following code is *not* type-stable:

```
julia> function unstable(flag::Bool)

           if flag

               return 1

           else
```

```
            return 1.0

        end

    end
unstable (generic function with 1 method)
```

It returns either an `Int` or a `Float64` depending on the value of its argument. Since Julia can't predict the return type of this function at compile-time, any computation that uses it will have to guard against both types possibly occurring, making generation of fast machine code difficult.

Why does Julia give a `DomainError` for certain seemingly-sensible operations?

Certain operations make mathematical sense but result in errors:

```
julia> sqrt(-2.0)
ERROR: DomainError:
sqrt will only return a complex result if called with a complex argument. Try sqrt(complex(x)).
Stacktrace:
 [1] sqrt(::Float64) at ./math.jl:425

julia> 2^-5
ERROR: DomainError:
Cannot raise an integer x to a negative power -n.
Make x a float by adding a zero decimal (e.g. 2.0^-n instead of 2^-n), or write 1/x^n,
↪  float(x)^-n, or (x//1)^-n.
Stacktrace:
 [1] power_by_squaring(::Int64, ::Int64) at ./intfuncs.jl:173
 [2] literal_pow(::Base.#^, ::Int64, ::Type{Val{-5}}) at ./intfuncs.jl:208
```

This behavior is an inconvenient consequence of the requirement for type-stability. In the case of `sqrt()`, most users want `sqrt(2.0)` to give a real number, and would be unhappy if it produced the complex number `1.4142135623730951 + 0.0im`. One could write the `sqrt()` function to switch to a complex-valued output only when passed a negative number (which is what `sqrt()` does in some other languages), but then the result would not be type-stable and the `sqrt()` function would have poor performance.

In these and other cases, you can get the result you want by choosing an *input type* that conveys your willingness to accept an *output type* in which the result can be represented:

```
julia> sqrt(-2.0+0im)
0.0 + 1.4142135623730951im

julia> 2.0^-5
0.03125
```

Why does Julia use native machine integer arithmetic?

Julia uses machine arithmetic for integer computations. This means that the range of `Int` values is bounded and wraps around at either end so that adding, subtracting and multiplying integers can overflow or underflow, leading to some results that can be unsettling at first:

```
julia> typemax(Int)
9223372036854775807

julia> ans+1
-9223372036854775808

julia> -ans
-9223372036854775808

julia> 2*ans
0
```

Clearly, this is far from the way mathematical integers behave, and you might think it less than ideal for a high-level programming language to expose this to the user. For numerical work where efficiency and transparency are at a premium, however, the alternatives are worse.

One alternative to consider would be to check each integer operation for overflow and promote results to bigger integer types such as `Int128` or `BigInt` in the case of overflow. Unfortunately, this introduces major overhead on every integer operation (think incrementing a loop counter) – it requires emitting code to perform run-time overflow checks after arithmetic instructions and branches to handle potential overflows. Worse still, this would cause every computation involving integers to be type-unstable. As we mentioned above, type-stability is crucial for effective generation of efficient code. If you can't count on the results of integer operations being integers, it's impossible to generate fast, simple code the way C and Fortran compilers do.

A variation on this approach, which avoids the appearance of type instability is to merge the `Int` and `BigInt` types into a single hybrid integer type, that internally changes representation when a result no longer fits into the size of a machine integer. While this superficially avoids type-instability at the level of Julia code, it just sweeps the problem under the rug by foisting all of the same difficulties onto the C code implementing this hybrid integer type. This approach *can* be made to work and can even be made quite fast in many cases, but has several drawbacks. One problem is that the in-memory representation of integers and arrays of integers no longer match the natural representation used by C, Fortran and other languages with native machine integers. Thus, to interoperate with those languages, we would ultimately need to introduce native integer types anyway. Any unbounded representation of integers cannot have a fixed number of bits, and thus cannot be stored inline in an array with fixed-size slots – large integer values will always require separate heap-allocated storage. And of course, no matter how clever a hybrid integer implementation one uses, there are always performance traps – situations where performance degrades unexpectedly. Complex representation, lack of interoperability with C and Fortran, the inability to represent integer arrays without additional heap storage, and unpredictable performance characteristics make even the cleverest hybrid integer implementations a poor choice for high-performance numerical work.

An alternative to using hybrid integers or promoting to BigInts is to use saturating integer arithmetic, where adding to the largest integer value leaves it unchanged and likewise for subtracting from the smallest integer value. This is precisely what Matlab™ does:

```
>> int64(9223372036854775807)

ans =

  9223372036854775807

>> int64(9223372036854775807) + 1

ans =

  9223372036854775807
```

```
>> int64(-9223372036854775808)

ans =

 -9223372036854775808

>> int64(-9223372036854775808) - 1

ans =

 -9223372036854775808
```

At first blush, this seems reasonable enough since 9223372036854775807 is much closer to 9223372036854775808 than -9223372036854775808 is and integers are still represented with a fixed size in a natural way that is compatible with C and Fortran. Saturated integer arithmetic, however, is deeply problematic. The first and most obvious issue is that this is not the way machine integer arithmetic works, so implementing saturated operations requires emitting instructions after each machine integer operation to check for underflow or overflow and replace the result with `typemin(Int)` or `typemax(Int)` as appropriate. This alone expands each integer operation from a single, fast instruction into half a dozen instructions, probably including branches. Ouch. But it gets worse – saturating integer arithmetic isn't associative. Consider this Matlab computation:

```
>> n = int64(2)^62
4611686018427387904

>> n + (n - 1)
9223372036854775807

>> (n + n) - 1
9223372036854775806
```

This makes it hard to write many basic integer algorithms since a lot of common techniques depend on the fact that machine addition with overflow *is* associative. Consider finding the midpoint between integer values `lo` and `hi` in Julia using the expression `(lo + hi) >>> 1`:

```
julia> n = 2^62
4611686018427387904

julia> (n + 2n) >>> 1
6917529027641081856
```

See? No problem. That's the correct midpoint between 2^62 and 2^63, despite the fact that `n + 2n` is -4611686018427387904. Now try it in Matlab:

```
>> (n + 2*n)/2

ans =

  4611686018427387904
```

Oops. Adding a `>>>` operator to Matlab wouldn't help, because saturation that occurs when adding `n` and `2n` has already destroyed the information necessary to compute the correct midpoint.

Not only is lack of associativity unfortunate for programmers who cannot rely it for techniques like this, but it also defeats almost anything compilers might want to do to optimize integer arithmetic. For example, since Julia integers use normal machine integer arithmetic, LLVM is free to aggressively optimize simple little functions like `f(k) = 5k-1`. The machine code for this function is just this:

```
julia> code_native(f, Tuple{Int})
  .text
Filename: none
  pushq %rbp
  movq  %rsp, %rbp
Source line: 1
  leaq  -1(%rdi,%rdi,4), %rax
  popq  %rbp
  retq
  nopl  (%rax,%rax)
```

The actual body of the function is a single `leaq` instruction, which computes the integer multiply and add at once. This is even more beneficial when `f` gets inlined into another function:

```
julia> function g(k, n)

           for i = 1:n

               k = f(k)

           end

           return k

       end
g (generic function with 1 methods)

julia> code_native(g, Tuple{Int,Int})
  .text
Filename: none
  pushq %rbp
  movq  %rsp, %rbp
Source line: 2
  testq %rsi, %rsi
  jle L26
  nopl  (%rax)
Source line: 3
L16:
  leaq  -1(%rdi,%rdi,4), %rdi
Source line: 2
  decq  %rsi
  jne L16
Source line: 5
L26:
  movq  %rdi, %rax
  popq  %rbp
  retq
  nop
```

Since the call to `f` gets inlined, the loop body ends up being just a single `leaq` instruction. Next, consider what happens if we make the number of loop iterations fixed:

```
julia> function g(k)

```

```
            for i = 1:10
                k = f(k)
            end
            return k
        end
g (generic function with 2 methods)

julia> code_native(g,(Int,))
  .text
Filename: none
  pushq %rbp
  movq  %rsp, %rbp
Source line: 3
  imulq $9765625, %rdi, %rax    # imm = 0x9502F9
  addq  $-2441406, %rax         # imm = 0xFFDABF42
Source line: 5
  popq  %rbp
  retq
  nopw  %cs:(%rax,%rax)
```

Because the compiler knows that integer addition and multiplication are associative and that multiplication distributes over addition – neither of which is true of saturating arithmetic – it can optimize the entire loop down to just a multiply and an add. Saturated arithmetic completely defeats this kind of optimization since associativity and distributivity can fail at each loop iteration, causing different outcomes depending on which iteration the failure occurs in. The compiler can unroll the loop, but it cannot algebraically reduce multiple operations into fewer equivalent operations.

The most reasonable alternative to having integer arithmetic silently overflow is to do checked arithmetic everywhere, raising errors when adds, subtracts, and multiplies overflow, producing values that are not value-correct. In this blog post, Dan Luu analyzes this and finds that rather than the trivial cost that this approach should in theory have, it ends up having a substantial cost due to compilers (LLVM and GCC) not gracefully optimizing around the added overflow checks. If this improves in the future, we could consider defaulting to checked integer arithmetic in Julia, but for now, we have to live with the possibility of overflow.

What are the possible causes of an `UndefVarError` during remote execution?

As the error states, an immediate cause of an `UndefVarError` on a remote node is that a binding by that name does not exist. Let us explore some of the possible causes.

```
julia> module Foo
            foo() = remotecall_fetch(x->x, 2, "Hello")
       end

julia> Foo.foo()
ERROR: On worker 2:
UndefVarError: Foo not defined
[...]
```

The closure `x->x` carries a reference to `Foo`, and since `Foo` is unavailable on node 2, an `UndefVarError` is thrown.

Globals under modules other than `Main` are not serialized by value to the remote node. Only a reference is sent. Functions which create global bindings (except under `Main`) may cause an `UndefVarError` to be thrown later.

```
julia> @everywhere module Foo

           function foo()

               global gvar = "Hello"

               remotecall_fetch(()->gvar, 2)

           end

       end

julia> Foo.foo()
ERROR: On worker 2:
UndefVarError: gvar not defined
[...]
```

In the above example, `@everywhere module Foo` defined `Foo` on all nodes. However the call to `Foo.foo()` created a new global binding `gvar` on the local node, but this was not found on node 2 resulting in an `UndefVarError` error.

Note that this does not apply to globals created under module `Main`. Globals under module `Main` are serialized and new bindings created under `Main` on the remote node.

```
julia> gvar_self = "Node1"
"Node1"

julia> remotecall_fetch(()->gvar_self, 2)
"Node1"

julia> remotecall_fetch(whos, 2)
        From worker 2:                                  Base  41762 KB     Module
        From worker 2:                                  Core  27337 KB     Module
        From worker 2:                                   Foo   2477 bytes  Module
        From worker 2:                                  Main  46191 KB     Module
        From worker 2:                             gvar_self     13 bytes  String
```

This does not apply to `function` or `type` declarations. However, anonymous functions bound to global variables are serialized as can be seen below.

```
julia> bar() = 1
bar (generic function with 1 method)

julia> remotecall_fetch(bar, 2)
ERROR: On worker 2:
UndefVarError: #bar not defined
[...]

julia> anon_bar  = ()->1
(::#21) (generic function with 1 method)

julia> remotecall_fetch(anon_bar, 2)
1
```

42.4 Packages and Modules

What is the difference between ”using” and ”importall”?

There is only one difference, and on the surface (syntax-wise) it may seem very minor. The difference between `using` and `importall` is that with `using` you need to say `function Foo.bar(..` to extend module Foo's function bar with a new method, but with `importall` or `import Foo.bar`, you only need to say `function bar(...` and it automatically extends module Foo's function bar.

If you use `importall`, then `function Foo.bar(...` and `function bar(...` become equivalent. If you use `using`, then they are different.

The reason this is important enough to have been given separate syntax is that you don't want to accidentally extend a function that you didn't know existed, because that could easily cause a bug. This is most likely to happen with a method that takes a common type like a string or integer, because both you and the other module could define a method to handle such a common type. If you use `importall`, then you'll replace the other module's implementation of `bar(s::AbstractString)` with your new implementation, which could easily do something completely different (and break all/many future usages of the other functions in module Foo that depend on calling bar).

42.5 Nothingness and missing values

How does ”null” or ”nothingness” work in Julia?

Unlike many languages (for example, C and Java), Julia does not have a ”null” value. When a reference (variable, object field, or array element) is uninitialized, accessing it will immediately throw an error. This situation can be detected using the `isdefined` function.

Some functions are used only for their side effects, and do not need to return a value. In these cases, the convention is to return the value `nothing`, which is just a singleton object of type `Void`. This is an ordinary type with no fields; there is nothing special about it except for this convention, and that the REPL does not print anything for it. Some language constructs that would not otherwise have a value also yield `nothing`, for example `if false; end`.

For situations where a value exists only sometimes (for example, missing statistical data), it is best to use the `Nullable{T}` type, which allows specifying the type of a missing value.

The empty tuple (`()`) is another form of nothingness. But, it should not really be thought of as nothing but rather a tuple of zero values.

In code written for Julia prior to version 0.4 you may occasionally see `None`, which is quite different. It is the empty (or ”bottom”) type, a type with no values and no subtypes (except itself). This is now written as `Union{}` (an empty union type). You will generally not need to use this type.

42.6 Memory

Why does `x += y` allocate memory when x and y are arrays?

In Julia, `x += y` gets replaced during parsing by `x = x + y`. For arrays, this has the consequence that, rather than storing the result in the same location in memory as `x`, it allocates a new array to store the result.

While this behavior might surprise some, the choice is deliberate. The main reason is the presence of immutable objects within Julia, which cannot change their value once created. Indeed, a number is an immutable object; the statements `x = 5; x += 1` do not modify the meaning of 5, they modify the value bound to `x`. For an immutable, the only way to change the value is to reassign it.

To amplify a bit further, consider the following function:

```
function power_by_squaring(x, n::Int)
    ispow2(n) || error("This implementation only works for powers of 2")
    while n >= 2
        x *= x
        n >>= 1
    end
    x
end
```

After a call like `x = 5; y = power_by_squaring(x, 4)`, you would get the expected result: `x == 5 && y == 625`. However, now suppose that `*=`, when used with matrices, instead mutated the left hand side. There would be two problems:

- For general square matrices, `A = A*B` cannot be implemented without temporary storage: `A[1,1]` gets computed and stored on the left hand side before you're done using it on the right hand side.
- Suppose you were willing to allocate a temporary for the computation (which would eliminate most of the point of making `*=` work in-place); if you took advantage of the mutability of `x`, then this function would behave differently for mutable vs. immutable inputs. In particular, for immutable `x`, after the call you'd have (in general) `y != x`, but for mutable `x` you'd have `y == x`.

Because supporting generic programming is deemed more important than potential performance optimizations that can be achieved by other means (e.g., using explicit loops), operators like `+=` and `*=` work by rebinding new values.

42.7 Asynchronous IO and concurrent synchronous writes

Why do concurrent writes to the same stream result in inter-mixed output?

While the streaming I/O API is synchronous, the underlying implementation is fully asynchronous.

Consider the printed output from the following:

```
julia> @sync for i in 1:3

           @async write(STDOUT, string(i), " Foo ", " Bar ")

       end
123 Foo  Foo  Foo  Bar  Bar  Bar
```

This is happening because, while the `write` call is synchronous, the writing of each argument yields to other tasks while waiting for that part of the I/O to complete.

`print` and `println` "lock" the stream during a call. Consequently changing `write` to `println` in the above example results in:

```
julia> @sync for i in 1:3

           @async println(STDOUT, string(i), " Foo ", " Bar ")

       end
1 Foo  Bar
2 Foo  Bar
3 Foo  Bar
```

You can lock your writes with a `ReentrantLock` like this:

```
julia> l = ReentrantLock()
ReentrantLock(Nullable{Task}(), Condition(Any[]), 0)

julia> @sync for i in 1:3

           @async begin

               lock(l)

               try

                   write(STDOUT, string(i), " Foo ", " Bar ")

               finally

                   unlock(l)

               end

           end

       end
1 Foo  Bar 2 Foo  Bar 3 Foo  Bar
```

42.8 Julia Releases

Do I want to use a release, beta, or nightly version of Julia?

You may prefer the release version of Julia if you are looking for a stable code base. Releases generally occur every 6 months, giving you a stable platform for writing code.

You may prefer the beta version of Julia if you don't mind being slightly behind the latest bugfixes and changes, but find the slightly faster rate of changes more appealing. Additionally, these binaries are tested before they are published to ensure they are fully functional.

You may prefer the nightly version of Julia if you want to take advantage of the latest updates to the language, and don't mind if the version available today occasionally doesn't actually work.

Finally, you may also consider building Julia from source for yourself. This option is mainly for those individuals who are comfortable at the command line, or interested in learning. If this describes you, you may also be interested in reading our guidelines for contributing.

Links to each of these download types can be found on the download page at https://julialang.org/downloads/. Note that not all versions of Julia are available for all platforms.

When are deprecated functions removed?

Deprecated functions are removed after the subsequent release. For example, functions marked as deprecated in the 0.1 release will not be available starting with the 0.2 release.

Chapter 43

Diferencias notables con otros idiomas

43.1 Diferencias notables con MATLAB

Aunque los usuarios de MATLAB pueden encontrar la sintaxis de Julia familiar, Julia no es un clon de MATLAB. Hay importantes diferencias sintácticas y funcionales. Las siguientes son algunas diferencias notables que pueden hacer tropezar a los usuarios de Julia acostumbrados a MATLAB:

- Los arrays de Julia están indexados con corchetes, `A[i, j]`.
- Los arrays de Julia se asignan por referencia. Después de `A = B`, el cambio de elementos de `B` también modificará `A`.
- Los valores de Julia se pasan y se asignan por referencia. Si una función modifica una matriz, los cambios serán visibles en el código que la invoca.
- Julia no genera automáticamente matrices en una declaración de asignación. Mientras que en MATLAB `a(4) = 3.2` puede crear la matriz `a = [0 0 0 3.2]` y `a(5) = 7` puede crecer hasta `a = [0 0 0 3.2 7]`, la declaración correspondiente de Julia `a[5] = 7` arroja un error si la longitud de `a` es menor que 5 o si esta afirmación es el primer uso del identificador `a`. Julia tiene `push!()` y `append!()`, que crecen `Vectors` mucho más eficientemente que `a(end + 1) = val` de MATLAB.
- La unidad imaginaria `sqrt(-1)` se representa en Julia como `im`, no como `i` o `j` como en MATLAB.
- En Julia, los números literales sin un punto decimal (como 42) crean números enteros en lugar de números de coma flotante. Se admiten literales enteros arbitrariamente grandes. Como resultado, algunas operaciones como `2^-1` arrojarán un error de dominio ya que el resultado no es un número entero (ver la entrada de preguntas frecuentes sobre errores de dominio para más detalles).
- En Julia, los valores múltiples se devuelven y se asignan como tuplas, p. `(a, b) = (1, 2)` o `a, b = 1, 2`. `"nargout"` de MATLAB, que a menudo se usa en MATLAB para hacer trabajos opcionales basados en el número de valores devueltos, no existe en Julia. En cambio, los usuarios pueden usar argumentos opcionales y de palabras clave para lograr capacidades similares.
  - Julia tiene verdaderos arrays unidimensionales. Los vectores columna son de tamaño `N`, no`Nx1`. Por ejemplo, `rand(N)` forma una matriz de 1 dimensión.
  - En Julia, `[x, y, z]` siempre construirá una matriz de 3 elementos que contiene x,y y z.
    * Para concatenar en la primera dimensión (”vertical”) se usa `vcat(x, y, z)` o se separa con punto y coma (`[x; y; z]`).
    * Para concatenar en la segunda dimensión (”horizontal”), se usa `hcat(x, y, z)` o se separa con espacios (`[x y z]`).

    * Para construir matrices de bloques (concatenando en las dos primeras dimensiones), se usa `hvcat()` o se combinan espacios y puntos y comas (`[a b; c d]`).

- En Julia, `a:b` y `a:b:c` construyen objetos `Range`. Para construir un vector completo como en MATLAB, use `collect(a:b)`. En general, no es necesario llamar a `collect` aunque. `Range` actuará como un array normal en la mayoría de los casos, pero es más eficiente porque calcula perezosamente sus valores. Este patrón de creación de objetos especializados en lugar de matrices completas se usa con frecuencia, y también se ve en funciones como `linspace`, o con iteradores como `enumerate` y `zip`. Los objetos especiales se pueden usar principalmente como si fueran matrices normales.

- Las funciones en Julia devuelven valores de su última expresión o la palabra clave `return` en lugar de enumerar los nombres de las variables a devolver en la definición de la función (ver La palabra clave return para más detalles).

- Un script de Julia puede contener cualquier cantidad de funciones, y todas las definiciones serán visibles externamente cuando se cargue el archivo. Las definiciones de funciones se pueden cargar desde archivos fuera del directorio de trabajo actual.

- En Julia, las reducciones como `sum()`, `prod()`, y `max()` se realizan sobre cada elemento de un array cuando se llama con un solo argumento, como en `sum(A)`, incluso si `A` tiene más de una dimensión.

- En Julia, las funciones como `sort()` que operan en forma de columnas por defecto (`sort(A)` es equivalente a `sort(A,1)`) no tienen un comportamiento especial para Conjuntos `1xN`; el argumento se devuelve sin modificar ya que todavía ejecuta `sort (A,1)`. Para ordenar una matriz `1xN` como un vector, use `sort(A,2)`.

- In Julia, if `A` is a 2-dimensional array, `fft(A)` computes a 2D FFT. In particular, it is not equivalent to `fft(A,1)`, which computes a 1D FFT acting column-wise.

- En Julia, los paréntesis se deben usar para llamar a una función con cero argumentos, como en `tic()` y `toc()`.

- Julia desalienta el uso de punto y coma para finalizar las declaraciones. Los resultados de las declaraciones no se imprimen automáticamente (excepto en el aviso interactivo), y las líneas de código no necesitan terminar con punto y coma. `println()` o `@printf()` se pueden usar para imprimir resultados específicos.

- En Julia, si `A` y`B` son matrices, las operaciones de comparación lógica como `A == B` no devuelven una matriz de booleanos. En cambio, use `A == B`, y de manera similar para los otros operadores booleanos como `<`, `>` y `=`.

- En Julia, los operadores `&`, `|` y `xor` (`xor`) realizan el operaciones a nivel de bit equivalentes a `y`,`o`, y `xor` respectivamente en MATLAB, y tienen precedencia similar a los operadores de bit a bit de Python (a diferencia de C). Pueden operar en escalas o en elementos a través de matrices y se pueden usar para combinar matrices lógicas, pero tenga en cuenta la diferencia en el orden de las operaciones: pueden ser necesarios paréntesis (por ejemplo, para seleccionar elementos de 'A' igual a 1 o 2 use `(A. == 1) | (A. == 2)`).

- En Julia, los elementos de una colección se pueden pasar como argumentos a una función usando el operador splat `...`, como en `xs = [1,2]; f(xs...)`.

- Julia `svd()` devuelve valores singulares como un vector en lugar de una matriz diagonal densa.

- En Julia, `...` no se usa para continuar líneas de código. En cambio, las expresiones incompletas continúan automáticamente en la siguiente línea.

- Tanto en Julia como en MATLAB, la variable `ans` se establece en el valor de la última expresión emitida en una sesión interactiva. En Julia, a diferencia de MATLAB, `ans` no se establece cuando el código de Julia se ejecuta en modo no interactivo.

- Los `types` de Julia no son compatibles con la adición dinámica de campos en el tiempo de ejecución, a diferencia de `classes` es de MATLAB. En su lugar, use un `Dict`.

- En Julia, cada módulo tiene su propio ámbito / espacio de nombres global, mientras que en MATLAB solo hay un ámbito global.
- En MATLAB, una forma idiomática de eliminar valores no deseados es usar la indexación lógica, como en la expresión `x (x> 3)` o en la declaración `x (x> 3) = []` para modificar x en -lugar. Por el contrario, Julia proporciona las funciones de orden superior `filter()` y `filter!()`, permitiendo a los usuarios escribir `filter (z-> z> 3, x)` y `filter! (z-> z> 3, x)` como alternativas a las transliteraciones correspondientes `x [x.> 3]` y `x = x [x.> 3]`. El uso de `filter!()` reduce el uso de matrices temporales.
- El análogo de extracción (o "desreferenciación") de todos los elementos de una matriz de celdas, p. en `vertcat(A {:})` en MATLAB, se escribe utilizando el operador splat en Julia, p. como `vcat(A ...)`.

43.2 Diferencias notables con R

Uno de los objetivos de Julia es proporcionar un lenguaje efectivo para el análisis de datos y la programación estadística. Para los usuarios que vienen a Julia de R, estas son algunas diferencias notables:

- Las comillas simples de Julia encierran caracteres, no cadenas.
- Julia puede crear subcadenas indexando en cadenas. En R, las cadenas deben convertirse en vectores de caracteres antes de crear subcadenas.
- En Julia, como Python pero a diferencia de R, las cadenas se pueden crear con comillas triples `""" ... """`. Esta sintaxis es conveniente para construir cadenas que contienen saltos de línea.
- En Julia, las variables se especifican utilizando el operador splat `...`, que siempre sigue el nombre de una variable específica, a diferencia de R, para el que `...` puede ocurrir de forma aislada.
- En Julia, el módulo es `mod(a, b)`, no a `%% b`. `%` en Julia es el operador restante.
- En Julia, no todas las estructuras de datos admiten la indexación lógica. Además, la indexación lógica en Julia solo se admite con vectores de longitud igual al objeto que se indexa. Por ejemplo:
  - En R, `c (1, 2, 3, 4) [c (TRUE, FALSE)]` es equivalente a `c (1, 3)`.
  - En R, `c (1, 2, 3, 4) [c (TRUE, FALSE, TRUE, FALSE)]` es equivalente a `c (1, 3)`.
  - En Julia, `[1, 2, 3, 4] [[verdadero, falso]]` arroja un `BoundsError`.
  - En Julia, `[1, 2, 3, 4] [[verdadero, falso, verdadero, falso]]` produce `[1, 3]`.
- Como en muchos lenguajes, Julia no siempre permite operaciones en vectores de diferentes longitudes, a diferencia de R, donde los vectores solo necesitan compartir un rango de índice común. Por ejemplo, `c(1, 2, 3, 4) + c (1, 2)` es válido R pero el equivalente `[1, 2, 3, 4] + [1, 2]` arrojará un error en Julia.
- Julia `map()` toma primero la función, luego sus argumentos, a diferencia de `lapply (<structure>, function, ...)` en R. De forma similar, el equivalente de Julia de `apply(X, MARGIN , FUN, ...)` en R es `mapslices()` donde la función es el primer argumento.
- La aplicación multivariada en R, es decir, `mapply(choose, 11:13, 1: 3)`, se puede escribir como `broadcast(binomial, 11:13, 1:3)` en Julia. Equivalentemente, Julia ofrece una sintaxis de punto más corta para vectorizar funciones `binomial.(11:13, 1: 3)`.
- Julia usa `end` para denotar el final de bloques condicionales, como`if`, bloques de bucle, como `while` /`for`, y funciones. En lugar de la sentencia de una línea `if (cond)`, Julia permite las declaraciones de la forma `if cond; declaración; end`, `cond && statement` y `!cond || declaración`. Las declaraciones de asignación en las dos últimas sintaxis deben estar explícitamente entrelazadas entre paréntesis, por ejemplo, `cond && (x = value)`.

- En Julia, `<-`,`<< -`y`->`no son operadores de asignación.
- En Julia, `->` crea una función anónima, como en Python.
- Julia construye vectores usando corchetes. El `[1, 2, 3]` de Julia es el equivalente de R a `c(1, 2, 3)`.
- El operador de Julia `*` puede realizar la multiplicación de la matriz, a diferencia de R. Si `A` y`B` son matrices, entonces `A * B` denota una multiplicación de matriz en Julia, equivalente a R `A %*% B`. En R, esta misma notación realizaría un producto de elemento (Hadamard). Para obtener la operación de multiplicación por elementos, debe escribir `A.*B` en Julia.
- Julia realiza la transposición de la matriz utilizando el operador `.'` y la transposición conjugada con el operador `'`. Por lo tanto, `A.'` de Julia es equivalente a `t(A)`en R.
- Julia no requiere paréntesis cuando escribe bucles `if`o`for` / `while`: use`for i en [1, 2, 3]`en lugar de`for (i en c (1, 2, 3) )` o `if i == 1` en lugar de `if (i == 1)`.
- Julia no trata los números `0` y`1` como booleanos. No puede escribir `if(1)` en Julia, porque las sentencias `if` solo aceptan booleanos. En su lugar, puede escribir `if true` , `if Bool(1)`, o`if 1 == 1`.
- Julia no proporciona `nrow` y`ncol`. En su lugar, use `size(M,1)` para `nrow(M)` y `size(M,2)` para `ncol(M)`.
- Julia tiene cuidado de distinguir escalares, vectores y matrices. En R, `1` y `c(1)` son lo mismo. En Julia, no se pueden usar indistintamente. Un resultado potencialmente confuso de esto es que `x'* y` para los vectores `x` e `y` es un vector de `1` elemento, no escalar. Para obtener un escalar, use `dot(x, y)`.
- Julia's `diag()` y `diagm()` no son como R's.
- Julia no puede asignar los resultados de llamadas a funciones en el lado izquierdo de una operación de asignación: no puede escribir `diag(M) = ones(n)`.
- Julia desaconseja llenar el espacio de nombres principal con funciones. La mayoría de las funcionalidades estadísticas para Julia se encuentran en paquetes bajo la organización JuliaStats. Por ejemplo:
  - Las funciones relacionadas con las distribuciones de probabilidad son proporcionadas por el Paquete Distributions.
  - El paquete DataFrames proporciona el tipo de datos `DataFrame`.
  - El paquete GLM proporciona modelos lineales generalizados.
- Julia proporciona tuplas y tablas hash reales, pero no listas R-style. Al devolver varios elementos, normalmente debe usar una tupla: en lugar de `list(a = 1, b = 2)`, use `(1, 2)`.
- Julia anima a los usuarios a escribir sus propios tipos, que son más fáciles de usar que los objetos S3 o S4 de R. El sistema de despacho múltiple de Julia significa que `table(x::TypeA)` y `table(x::TypeB)` actúan como R `table.TypeA(x)` y `table.TypeB(x)`.
- En Julia, los valores se pasan y se asignan por referencia. Si una función modifica una matriz, los cambios serán visibles en el código que invoca la función. Esto es muy diferente de R y permite que las nuevas funciones operen en estructuras de datos de gran tamaño de manera mucho más eficiente.
- En Julia, los vectores y las matrices se concatenan usando `hcat()`, `vcat()` y `hvcat()`, no `c` , `rbind` y `cbind` como en R.
- En Julia, un rango como `a:b` no es una abreviatura para un vector como en R, sino que es un `Range` especializado que se usa para la iteración sin una gran sobrecarga de memoria. Para convertir un rango en un vector, use `collect(a:b)`.

- Las funciones de Julia `max()` y `min()` son el equivalente de `pmax` y `pmin` respectivamente en R, pero ambos argumentos deben tener las mismas dimensiones. Mientras que `maximum()` y `minimum()` reemplazan `max` y `min` en R, hay diferencias importantes.
- Las funciones de Julia `sum()`, `prod()`, `maximum()`, y `minimum()` son diferentes de sus contrapartes en R. Todos aceptan uno o dos argumentos. El primer argumento es una colección iterable tal como una matriz. Si hay un segundo argumento, este argumento indica las dimensiones sobre las cuales se lleva a cabo la operación. Por ejemplo, deje `A = [[1 2], [3 4]]` en Julia y `B = rbind(c(1,2), c(3,4))` sea la misma matriz en R. Luego `sum(A)` da el mismo resultado que `sum(B)`, pero `sum(A, 1)` es un vector de fila que contiene la suma sobre cada columna y `sum(A, 2)` es un vector de columna que contiene la suma sobre cada fila. Esto contrasta con el comportamiento de R, donde `sum(B, 1) = 11` y`sum(B, 2) = 12`. Si el segundo argumento es un vector, entonces especifica todas las dimensiones sobre las cuales se realiza la suma, por ejemplo, `sum(A, [1,2]) = 10`. Cabe señalar que no hay errores de comprobación con respecto al segundo argumento.
- Julia tiene varias funciones que pueden mutar sus argumentos. Por ejemplo, tiene ambos `sort()` y `sort!()`.
- En R, el rendimiento requiere vectorización. En Julia, casi todo lo contrario es cierto: el código de mejor rendimiento a menudo se logra mediante el uso de bucles devectorized.
- Julia es evaluada con entusiasmo y no es compatible con la evaluación perezosa de estilo R. Para la mayoría de los usuarios, esto significa que hay muy pocas expresiones sin comillas o nombres de columnas.
- Julia no admite el tipo `NULL`.
- A Julia le falta el equivalente de ”asignar” o ”obtener” de R.
- En Julia, `return` no requiere paréntesis.
- En R, una forma idiomática de eliminar valores no deseados es usar indexación lógica, como en la expresión `x[x> 3]` o en la declaración `x = x[x > 3]` para modificar `x` lugar. Por el contrario, Julia proporciona las funciones de orden superior `filter()` y `filter!()`, permitiendo a los usuarios escribir `filter (z -> z>3, x)`y`filter! (z-> z> 3, x)`como alternativas a las transliteraciones correspondientes`x[x .> 3]`y`x = x [x.> 3]`. El uso de `filter!()` reduce el uso de matrices temporales.

43.3 Diferencias notables con Python

- Julia requiere `end` para finalizar un bloque. A diferencia de Python, Julia no tiene la palabra clave `pass`.
- En Julia, la indexación de matrices, cadenas, etc. se basa en 1 y no en 0.
- La indexación de segmentos de Julia incluye el último elemento, a diferencia de Python. `a[2:3]` en Julia es `a[1:3]` en Python.
- Julia no admite índices negativos. En particular, el último elemento de una lista o matriz se indexa con `end` en Julia, no`-1` como en Python.
- Los bloques `for`,`if`, `while`, etc. de Julia terminan con la palabra clave `end`. El nivel de sangría no es significativo ya que está en Python.
- Julia no tiene sintaxis de continuación de línea: si, al final de una línea, la entrada hasta ahora es una expresión completa, se considera hecho; de lo contrario, la entrada continúa. Una forma de forzar que una expresión continúe es envolverla entre paréntesis.
- Los arrays de Julia son *column major* (orden de Fortran) mientras que los arrays de NumPy son *row major* (orden de C) por defecto. Para obtener un rendimiento óptimo al alternar sobre matrices, el orden de los bucles debe invertirse en Julia en relación con NumPy (consulte la sección correspondiente de Sugerencias de rendimiento).

- Los operadores de actualización de Julia (por ejemplo, `+=`, `-=`, ...) *no están en el lugar* mientras que los de NumPy sí lo están. Esto significa `A = ones(4); B = A; B += 3` no cambia los valores en `A`, sino que vuelve a enlazar el nombre `B` con el resultado del lado derecho `B = B + 3`, que es una nueva matriz. Para la operación in-situ, use `B[:] += 3` (vea también operadores de punto), loops explícitos, o `InplaceOps.jl`.
- Julia evalúa los valores predeterminados de los argumentos de la función cada vez que se invoca el método, a diferencia de Python, donde los valores predeterminados se evalúan solo una vez cuando se define la función. Por ejemplo, la función `f(x = rand()) = x` devuelve un nuevo número aleatorio cada vez que se invoca sin argumentos. Por otro lado, la función `g(x = [1,2]) = push!(X, 3)` devuelve `[1,2,3]` cada vez que se llama como `g()`.
- En Julia `%` es el operador restante, mientras que en Python es el módulo.

43.4 Noteworthy differences from C/C++

- Los arrays de Julia están indexados con corchetes y pueden tener más de una dimensión `A[i,j]`. Esta sintaxis no es apropiada para una referencia a un puntero o dirección como en C/C ++. Consulte la documentación de Julia sobre la sintaxis para la construcción de matrices (ha cambiado entre versiones).
- En Julia, la indexación de matrices, cadenas, etc. se basa en 1 y no en 0.
- Los arrays de Julia se asignan por referencia. Después de `A = B`, el cambio de elementos de`B` también modificará `A`. Los operadores de actualización como `+ =` no funcionan in situ, son equivalentes a `A = A + B`, que vuelve a enlazar el lado izquierdo con el resultado de la expresión del lado derecho.
- Los arrays de Julia *column major* (ordenación de Fortran), mientras que los conjuntos de C/C++ son *row major*. Para obtener un rendimiento óptimo al realizar un bucle sobre matrices, el orden de los bucles debe invertirse en Julia en relación con C/C++ (consulte la sección correspondiente de Sugerencias de rendimiento).
- Los valores de Julia se pasan y se asignan por referencia. Si una función modifica una matriz, los cambios serán visibles en la persona que llama.
- En Julia, el espacio en blanco es significativo, a diferencia de C/C ++, por lo que se debe tener cuidado al agregar/eliminar espacios en blanco de un programa Julia.
- En Julia, los números literales sin un punto decimal (como 42) crean enteros con signo, de tipo `Int`, pero los literales demasiado grandes para caber en el tamaño de palabra de la máquina se promocionarán automáticamente a un tipo de tamaño mayor, como `Int64` (si `Int` es `Int32`),`Int128`, o el tipo arbitrariamente grande `BigInt`. No hay sufijos literales numéricos, tales como `L`,`LL`, `U`,`UL`, `ULL` para indicar unsigned y/o signed vs. unsigned. Los literales decimales siempre se firman y los literales hexadecimales (que comienzan con `0x` como C / C ++), no tienen signo. Los literales hexadecimales también, a diferencia de C / C ++ / Java y, a diferencia de los literales decimales en Julia, tienen un tipo basado en la *longitud* del literal, incluidos los primeros 0. Por ejemplo, `0x0` y `0x00` tienen el tipo `UInt8`,`0x000` y `0x0000` tienen el tipo `UInt16`, luego los literales con 5 a 8 dígitos hexadecimales tienen se consideran `UInt32`, de 9 a 16 dígitos hexadecimales `UInt64` y de 17 a 32 dígitos hexadecibles `UInt128`. Esto debe tenerse en cuenta al definir máscaras hexadecimales, por ejemplo `~0xf == 0xf0` es muy diferente de `~0x000f == 0xfff0`. Los literales de 64 bit `Float64` y 32 bit `Float32` los bit se expresan como `1.0` y `1.0f0` respectivamente. Los literales de punto flotante se redondean (y no se promocionan al tipo `BigFloat`) si no se pueden representar exactamente. Los literales de coma flotante tienen un comportamiento más cercano a C/C ++. Los literales Octal (prefijado con `0o`) y binario (prefijado con `0b`) también se tratan como sin signo.
- Los literales de cadenas se pueden delimitar con los literales delimitados por `"` o `"""`, que pueden contener caracteres `"` sin citarlo como `"\" "` Los literales de cadena pueden tener valores de otras variables o expresiones interpoladas en ellos, indicadas por `$nombrevariable` o `$(expresión)`, que evalúa el nombre de la variable o la expresión en el contexto de la función.

- // indica un número Rational, y no un comentario de una sola línea (que es #en Julia)
- #= indica el comienzo de un comentario de líneas múltiples, y =# lo finaliza.
- Las funciones en Julia devuelven valores de sus últimas expresiones o la palabra clave `return`. Pueden devolverse múltiples valores desde funciones y asignarse como tuplas, por ejemplo, `(a, b) = myfunction()` o `a, b = myfunction()`, en lugar de tener que pasar los punteros a los valores como debería hacerse en C / C ++ (es decir, `a = myfunction(&b)`.
- Julia no requiere el uso de punto y coma para finalizar las declaraciones. Los resultados de las expresiones no se imprimen automáticamente (excepto en el prompt interactivo, es decir, el REPL), y las líneas de código no necesitan terminar con punto y coma. `println()` o `@printf()` se pueden usar para imprimir resultados específicos. En REPL, `;` se puede usar para suprimir la salida. `;` también tiene un significado diferente dentro de [ ], algo de lo que hay que tener cuidado. `;` se puede usar para separar expresiones en una sola línea, pero no son estrictamente necesarias en muchos casos, y son más una ayuda para la legibilidad.
- En Julia, el operador (`xor`) realiza la operación XOR bit a bit, es decir `^` en C / C++. Además, los operadores bit a bit no tienen la misma precedencia que C / C++, por lo que puede ser necesario un paréntesis.
- Julia `^` es exponenciación (pow), no bit a bit XOR como en C / C++ (use , o `xor`, en Julia)
- Julia tiene dos operadores de desplazamiento a la derecha, >> y >>>. >>> realiza un desplazamiento aritmético, >> siempre realiza un desplazamiento lógico, a diferencia de C / C ++, donde el significado de >> depende del tipo del valor que se está desplazando.
- Julia's -> crea una función anónima, no accede a un miembro a través de un puntero.
- Julia no requiere paréntesis cuando escribe declaraciones `if` o`for` / `while` loops: use `for i in [1, 2, 3]` en lugar de `for (int i = 1; i <= 3; i++)` y `if i == 1` en lugar de `if (i == 1)`.
- Julia no trata los números `0` y 1 como booleanos. No puede escribir `if (1)` en Julia, porque las sentencias `if` solo aceptan booleanos. En su lugar, puede escribir `if true`, `if Bool (1)`, o `if 1 == 1`.
- Julia usa end para denotar el final de bloques condicionales, como `if`, bloques de bucle, como `while` / `for`, y funciones. En lugar de la sentencia de una línea `if (cond)`, Julia permite las declaraciones de la forma `if cond; declaración; end`, `cond && statement` y `!cond || declaración`. Las declaraciones de asignación en las dos últimas sintaxis deben estar explícitamente entrelazadas entre paréntesis, p. `cond && (x = value)`, debido a la precedencia del operador.
- Julia no tiene sintaxis de continuación de línea: si, al final de una línea, la entrada hasta ahora es una expresión completa, se considera hecho; de lo contrario, la entrada continúa. Una forma de forzar que una expresión continúe es envolverla entre paréntesis.
- Las macros de Julia operan en expresiones analizadas, en lugar del texto del programa, lo que les permite realizar transformaciones sofisticadas del código de Julia. Los nombres de macro comienzan con el carácter `@`, y tienen una sintaxis similar a la función, `@mymacro(arg1, arg2, arg3)`, y una sintaxis similar a una declaración, `@mymacro arg1 arg2 arg3`. Las formas son intercambiables; la forma de función es particularmente útil si la macro aparece dentro de otra expresión, y es a menudo más clara. La forma similar a una declaración se usa a menudo para anotar bloques, como en la construcción paralela `for`: `@parallel para i en 1: n; # = cuerpo = #; fin`. Donde el final de la macroconstrucción puede no ser claro, use la forma similar a la función.
- Julia ahora tiene un tipo de enumeración, expresado con el macro `@enum (name, value1, value2, ...)` Por ejemplo: `@enum (Fruit, banana = 1, apple, pear)`
- Por convención, las funciones que modifican sus argumentos tienen un `!` Al final del nombre, por ejemplo `push!`.

- En C ++, de forma predeterminada, tiene despacho estático, es decir, necesita anotar una función como virtual, para tener un despacho dinámico. Por otro lado, en Julia todos los métodos son ”virtuales” (aunque es más general que eso ya que los métodos se envían en cada tipo de argumento, no solo `this`, usando la regla de declaración más específica).

Chapter 44

Entrada Unicode

La siguiente tabla enumera los caracteres Unicode que pueden ingresarse mediante la terminación de pestañas de las abreviaturas similares a LaTeX en Julia REPL (y en otros entornos de edición). También puede obtener información sobre cómo escribir un símbolo ingresándolo en la ayuda REPL, es decir, escribiendo ? Y luego ingresando el símbolo en REPL (por ejemplo, mediante copiar y pegar desde algún lugar donde vio el símbolo).

> **Warning**
>
> Puede parecer que esta tabla contiene caracteres faltantes en la segunda columna, o incluso muestra caracteres que son inconsistentes con los caracteres tal como se representan en Julia REPL. En estos casos, se recomienda encarecidamente a los usuarios que comprueben la elección de las fuentes en su navegador y entorno REPL, ya que existen problemas conocidos con los glifos en muchas fuentes.

Part IV

Standard Library

Chapter 45

Essentials

45.1 Introducción

La librería estándar de Julia contiene un rango de funciones y marcos apropiados para realizar computación científica y numérica, pero es también tan amplia como la de muchos lenguajes de programación de propósito general. También hay funcionalidad adicional disponible en una creciente colección de paquetes disponibles. Las funciones están agrupadas abajo por temas.

Algunas notas generales:

- Excepto para las funciones en los módulos predefinidos (`Pkg`, `Collections`, `Test` y `Profile`), todas las funciones documentadas aquí están disponibles para ser usadas en programas directamente.
- Para usar funciones de módulos, usar `import Module` para importar el módulo, y `Module.fn(x)` para usar las funciones.
- Alternativamente `using Module` importará todas las funciones exportadas por el módulo en el espacio de nombres actual.
- Por convenio, los nombres de funciones que acaban con un signo de admiración (`!`) modifican sus argumentos. Algunas funciones tienen las dos versiones (con y sin modificación de los argumentos).

45.2 Moviéndose

`Base.exit` – Function.

```
exit([code])
```

Quit (or control-D at the prompt). The default exit code is zero, indicating that the processes completed successfully.

source

`Base.quit` – Function.

```
quit()
```

Quit the program indicating that the processes completed successfully. This function calls `exit(0)` (see `exit`).

source

`Base.atexit` – Function.

```
atexit(f)
```

Register a zero-argument function `f()` to be called at process exit. `atexit()` hooks are called in last in first out (LIFO) order and run before object finalizers.

source

`Base.atreplinit` – Function.

```
atreplinit(f)
```

Register a one-argument function to be called before the REPL interface is initialized in interactive sessions; this is useful to customize the interface. The argument of `f` is the REPL object. This function should be called from within the `.juliarc.jl` initialization file.

source

`Base.isinteractive` – Function.

```
isinteractive() -> Bool
```

Determine whether Julia is running an interactive session.

source

`Base.whos` – Function.

```
whos(io::IO=STDOUT, m::Module=current_module(), pattern::Regex=r"")
```

Print information about exported global variables in a module, optionally restricted to those matching `pattern`.

The memory consumption estimate is an approximate lower bound on the size of the internal structure of the object.

source

`Base.summarysize` – Function.

```
Base.summarysize(obj; exclude=Union{...}, chargeall=Union{...}) -> Int
```

Compute the amount of memory used by all unique objects reachable from the argument.

Keyword Arguments

- `exclude`: specifies the types of objects to exclude from the traversal.
- `chargeall`: specifies the types of objects to always charge the size of all of their fields, even if those fields would normally be excluded.

source

`Base.edit` – Method.

```
edit(path::AbstractString, line::Integer=0)
```

Edit a file or directory optionally providing a line number to edit the file at. Returns to the `julia` prompt when you quit the editor. The editor can be changed by setting `JULIA_EDITOR`, `VISUAL` or `EDITOR` as an environment variable.

source

Base.edit – Method.

```
edit(function, [types])
```

Edit the definition of a function, optionally specifying a tuple of types to indicate which method to edit. The editor can be changed by setting JULIA_EDITOR, VISUAL or EDITOR as an environment variable.

source

Base.@edit – Macro.

```
@edit
```

Evaluates the arguments to the function or macro call, determines their types, and calls the edit function on the resulting expression.

source

Base.less – Method.

```
less(file::AbstractString, [line::Integer])
```

Show a file using the default pager, optionally providing a starting line number. Returns to the julia prompt when you quit the pager.

source

Base.less – Method.

```
less(function, [types])
```

Show the definition of a function using the default pager, optionally specifying a tuple of types to indicate which method to see.

source

Base.@less – Macro.

```
@less
```

Evaluates the arguments to the function or macro call, determines their types, and calls the less function on the resulting expression.

source

Base.clipboard – Method.

```
clipboard(x)
```

Send a printed form of x to the operating system clipboard ("copy").

source

Base.clipboard – Method.

```
clipboard() -> AbstractString
```

Return a string with the contents of the operating system clipboard ("paste").

source

Base.reload – Function.

```
reload(name::AbstractString)
```

Force reloading of a package, even if it has been loaded before. This is intended for use during package development as code is modified.

source

Base.require – Function.

```
require(module::Symbol)
```

This function is part of the implementation of `using` / `import`, if a module is not already defined in `Main`. It can also be called directly to force reloading a module, regardless of whether it has been loaded before (for example, when interactively developing libraries).

Loads a source file, in the context of the `Main` module, on every active node, searching standard locations for files. `require` is considered a top-level operation, so it sets the current `include` path but does not use it to search for files (see help for `include`). This function is typically used to load library code, and is implicitly called by `using` to load packages.

When searching for files, `require` first looks for package code under `Pkg.dir()`, then tries paths in the global array `LOAD_PATH`. `require` is case-sensitive on all platforms, including those with case-insensitive filesystems like macOS and Windows.

source

Base.compilecache – Function.

```
Base.compilecache(module::String)
```

Creates a precompiled cache file for a module and all of its dependencies. This can be used to reduce package load times. Cache files are stored in `LOAD_CACHE_PATH[1]`, which defaults to `~/.julia/lib/VERSION`. See Module initialization and precompilation for important notes.

source

Base.__precompile__ – Function.

```
__precompile__(isprecompilable::Bool=true)
```

Specify whether the file calling this function is precompilable. If `isprecompilable` is `true`, then `__precompile__` throws an exception when the file is loaded by `using`/`import`/`require` *unless* the file is being precompiled, and in a module file it causes the module to be automatically precompiled when it is imported. Typically, `__precompile__()` should occur before the `module` declaration in the file, or better yet `VERSION >= v"0.4" && __precompile__()` in order to be backward-compatible with Julia 0.3.

If a module or file is *not* safely precompilable, it should call `__precompile__(false)` in order to throw an error if Julia attempts to precompile it.

`__precompile__()` should *not* be used in a module unless all of its dependencies are also using `__precompile__()`. Failure to do so can result in a runtime error when loading the module.

source

Base.include – Function.

```
include(path::AbstractString)
```

Evaluate the contents of the input source file in the current context. Returns the result of the last evaluated expression of the input file. During including, a task-local include path is set to the directory containing the file. Nested calls to `include` will search relative to that path. All paths refer to files on node 1 when running in parallel, and files will be fetched from node 1. This function is typically used to load source interactively, or to combine files in packages that are broken into multiple source files.

source

`Base.include_string` – Function.

```
include_string(code::AbstractString, filename::AbstractString="string")
```

Like `include`, except reads code from the given string rather than from a file. Since there is no file path involved, no path processing or fetching from node 1 is done.

source

`Base.include_dependency` – Function.

```
include_dependency(path::AbstractString)
```

In a module, declare that the file specified by `path` (relative or absolute) is a dependency for precompilation; that is, the module will need to be recompiled if this file changes.

This is only needed if your module depends on a file that is not used via `include`. It has no effect outside of compilation.

source

`Base.Docs.apropos` – Function.

```
apropos(string)
```

Search through all documentation for a string, ignoring case.

source

`Base.which` – Method.

```
which(f, types)
```

Returns the method of `f` (a `Method` object) that would be called for arguments of the given `types`.

If `types` is an abstract type, then the method that would be called by `invoke` is returned.

source

`Base.which` – Method.

```
which(symbol)
```

Return the module in which the binding for the variable referenced by `symbol` was created.

source

`Base.@which` – Macro.

```
@which
```

Applied to a function or macro call, it evaluates the arguments to the specified call, and returns the `Method` object for the method that would be called for those arguments. Applied to a variable, it returns the module in which the variable was bound. It calls out to the `which` function.

source

`Base.methods` – Function.

```
methods(f, [types])
```

Returns the method table for `f`.

If `types` is specified, returns an array of methods whose types match.

source

`Base.methodswith` – Function.

```
methodswith(typ[, module or function][, showparents::Bool=false])
```

Return an array of methods with an argument of type `typ`.

The optional second argument restricts the search to a particular module or function (the default is all modules, starting from Main).

If optional `showparents` is `true`, also return arguments with a parent type of `typ`, excluding type `Any`.

source

`Base.@show` – Macro.

```
@show
```

Show an expression and result, returning the result.

source

`Base.versioninfo` – Function.

```
versioninfo(io::IO=STDOUT, verbose::Bool=false)
```

Print information about the version of Julia in use. If the `verbose` argument is `true`, detailed system information is shown as well.

source

`Base.workspace` – Function.

```
workspace()
```

Replace the top-level module (`Main`) with a new one, providing a clean workspace. The previous `Main` module is made available as `LastMain`. A previously-loaded package can be accessed using a statement such as `using LastMain.Package`.

This function should only be used interactively.

source

`ans` – Keyword.

```
ans
```

A variable referring to the last computed value, automatically set at the interactive prompt.

source

45.3 Todos los Objetos

`Core.:===` – Function.

```
===(x,y) -> Bool
(x,y) -> Bool
```

Determine whether x and y are identical, in the sense that no program could distinguish them. Compares mutable objects by address in memory, and compares immutable objects (such as numbers) by contents at the bit level. This function is sometimes called `egal`.

```
julia> a = [1, 2]; b = [1, 2];

julia> a == b
true

julia> a === b
false

julia> a === a
true
```

source

`Core.isa` – Function.

```
isa(x, type) -> Bool
```

Determine whether x is of the given `type`. Can also be used as an infix operator, e.g. `x isa type`.

source

`Base.isequal` – Method.

```
isequal(x, y)
```

Similar to `==`, except treats all floating-point `NaN` values as equal to each other, and treats `-0.0` as unequal to `0.0`. The default implementation of `isequal` calls `==`, so if you have a type that doesn't have these floating-point subtleties then you probably only need to define `==`.

`isequal` is the comparison function used by hash tables (`Dict`). `isequal(x,y)` must imply that `hash(x) == hash(y)`.

This typically means that if you define your own `==` function then you must define a corresponding `hash` (and vice versa). Collections typically implement `isequal` by calling `isequal` recursively on all contents.

Scalar types generally do not need to implement `isequal` separate from `==`, unless they represent floating-point numbers amenable to a more efficient implementation than that provided as a generic fallback (based on `isnan`, `signbit`, and `==`).

```
julia> isequal([1., NaN], [1., NaN])
true

julia> [1., NaN] == [1., NaN]
false

julia> 0.0 == -0.0
```

```
true

julia> isequal(0.0, -0.0)
false
```

source

Base.isequal – Method.

```
isequal(x, y)
```

Similar to ==, except treats all floating-point NaN values as equal to each other, and treats -0.0 as unequal to 0.0. The default implementation of isequal calls ==, so if you have a type that doesn't have these floating-point subtleties then you probably only need to define ==.

isequal is the comparison function used by hash tables (Dict). isequal(x,y) must imply that hash(x) == hash(y).

This typically means that if you define your own == function then you must define a corresponding hash (and vice versa). Collections typically implement isequal by calling isequal recursively on all contents.

Scalar types generally do not need to implement isequal separate from ==, unless they represent floating-point numbers amenable to a more efficient implementation than that provided as a generic fallback (based on isnan, signbit, and ==).

```
julia> isequal([1., NaN], [1., NaN])
true

julia> [1., NaN] == [1., NaN]
false

julia> 0.0 == -0.0
true

julia> isequal(0.0, -0.0)
false
```

source

```
isequal(x::Nullable, y::Nullable)
```

If neither x nor y is null, compare them according to their values (i.e. isequal(get(x), get(y))). Else, return true if both arguments are null, and false if one is null but not the other: nulls are considered equal.

source

Base.isless – Function.

```
isless(x, y)
```

Test whether x is less than y, according to a canonical total order. Values that are normally unordered, such as NaN, are ordered in an arbitrary but consistent fashion. This is the default comparison used by sort. Non-numeric types with a canonical total order should implement this function. Numeric types only need to implement it if they have special values such as NaN.

source

Base.isless – Method.

```
isless(x::Nullable, y::Nullable)
```

If neither x nor y is null, compare them according to their values (i.e. `isless(get(x), get(y))`). Else, return `true` if only y is null, and `false` otherwise: nulls are always considered greater than non-nulls, but not greater than another null.

source

`Base.ifelse` – Function.

```
ifelse(condition::Bool, x, y)
```

Return x if `condition` is `true`, otherwise return y. This differs from `?` or `if` in that it is an ordinary function, so all the arguments are evaluated first. In some cases, using `ifelse` instead of an `if` statement can eliminate the branch in generated code and provide higher performance in tight loops.

```
julia> ifelse(1 > 2, 1, 2)
2
```

source

`Base.lexcmp` – Function.

```
lexcmp(x, y)
```

Compare x and y lexicographically and return -1, 0, or 1 depending on whether x is less than, equal to, or greater than y, respectively. This function should be defined for lexicographically comparable types, and `lexless` will call `lexcmp` by default.

```
julia> lexcmp("abc", "abd")
-1

julia> lexcmp("abc", "abc")
0
```

source

`Base.lexless` – Function.

```
lexless(x, y)
```

Determine whether x is lexicographically less than y.

```
julia> lexless("abc", "abd")
true
```

source

`Core.typeof` – Function.

```
typeof(x)
```

Get the concrete type of x.

source

`Core.tuple` – Function.

```
tuple(xs...)
```

Construct a tuple of the given objects.

Example

```
julia> tuple(1, 'a', pi)
(1, 'a', π = 3.1415926535897...)
```

source

Base.ntuple – Function.

```
ntuple(f::Function, n::Integer)
```

Create a tuple of length n, computing each element as `f(i)`, where `i` is the index of the element.

```
julia> ntuple(i -> 2*i, 4)
(2, 4, 6, 8)
```

source

Base.object_id – Function.

```
object_id(x)
```

Get a hash value for x based on object identity. `object_id(x)==object_id(y)` if `x === y`.

source

Base.hash – Function.

```
hash(x[, h::UInt])
```

Compute an integer hash code such that `isequal(x,y)` implies `hash(x)==hash(y)`. The optional second argument h is a hash code to be mixed with the result.

New types should implement the 2-argument form, typically by calling the 2-argument hash method recursively in order to mix hashes of the contents with each other (and with h). Typically, any type that implements hash should also implement its own == (hence `isequal`) to guarantee the property mentioned above.

source

Base.finalizer – Function.

```
finalizer(x, f)
```

Register a function `f(x)` to be called when there are no program-accessible references to x. The type of x must be a `mutable struct`, otherwise the behavior of this function is unpredictable.

source

Base.finalize – Function.

```
finalize(x)
```

Immediately run finalizers registered for object x.

source

`Base.copy` – Function.

```
copy(x)
```

Create a shallow copy of x: the outer structure is copied, but not all internal values. For example, copying an array produces a new array with identically-same elements as the original.

source

`Base.deepcopy` – Function.

```
deepcopy(x)
```

Create a deep copy of x: everything is copied recursively, resulting in a fully independent object. For example, deep-copying an array produces a new array whose elements are deep copies of the original elements. Calling `deepcopy` on an object should generally have the same effect as serializing and then deserializing it.

As a special case, functions can only be actually deep-copied if they are anonymous, otherwise they are just copied. The difference is only relevant in the case of closures, i.e. functions which may contain hidden internal references.

While it isn't normally necessary, user-defined types can override the default `deepcopy` behavior by defining a specialized version of the function `deepcopy_internal(x::T, dict::ObjectIdDict)` (which shouldn't otherwise be used), where T is the type to be specialized for, and `dict` keeps track of objects copied so far within the recursion. Within the definition, `deepcopy_internal` should be used in place of `deepcopy`, and the `dict` variable should be updated as appropriate before returning.

source

`Core.isdefined` – Function.

```
isdefined([m::Module,] s::Symbol)
isdefined(object, s::Symbol)
isdefined(object, index::Int)
```

Tests whether an assignable location is defined. The arguments can be a module and a symbol or a composite object and field name (as a symbol) or index. With a single symbol argument, tests whether a global variable with that name is defined in `current_module()`.

source

`Base.convert` – Function.

```
convert(T, x)
```

Convert x to a value of type T.

If T is an `Integer` type, an `InexactError` will be raised if x is not representable by T, for example if x is not integer-valued, or is outside the range supported by T.

Examples

```
julia> convert(Int, 3.0)
3

julia> convert(Int, 3.5)
ERROR: InexactError()
Stacktrace:
 [1] convert(::Type{Int64}, ::Float64) at ./float.jl:679
```

If T is a `AbstractFloat` or `Rational` type, then it will return the closest value to x representable by T.

```
julia> x = 1/3
0.3333333333333333

julia> convert(Float32, x)
0.33333334f0

julia> convert(Rational{Int32}, x)
1//3

julia> convert(Rational{Int64}, x)
6004799503160661//18014398509481984
```

If T is a collection type and x a collection, the result of `convert(T, x)` may alias x.

```
julia> x = Int[1,2,3];

julia> y = convert(Vector{Int}, x);

julia> y === x
true
```

Similarly, if T is a composite type and x a related instance, the result of `convert(T, x)` may alias part or all of x.

```
julia> x = speye(5);

julia> typeof(x)
SparseMatrixCSC{Float64,Int64}

julia> y = convert(SparseMatrixCSC{Float64,Int64}, x);

julia> z = convert(SparseMatrixCSC{Float32,Int64}, y);

julia> y === x
true

julia> z === x
false

julia> z.colptr === x.colptr
true
```

source

`Base.promote` – Function.

```
promote(xs...)
```

Convert all arguments to their common promotion type (if any), and return them all (as a tuple).

Example

```
julia> promote(Int8(1), Float16(4.5), Float32(4.1))
(1.0f0, 4.5f0, 4.1f0)
```

source

Base.oftype – Function.

```
oftype(x, y)
```

Convert y to the type of x (`convert(typeof(x), y)`).

source

Base.widen – Function.

```
widen(x)
```

If x is a type, return a "larger" type (for numeric types, this will be a type with at least as much range and precision as the argument, and usually more). Otherwise x is converted to `widen(typeof(x))`.

Examples

```
julia> widen(Int32)
Int64

julia> widen(1.5f0)
1.5
```

source

Base.identity – Function.

```
identity(x)
```

The identity function. Returns its argument.

```
julia> identity("Well, what did you expect?")
"Well, what did you expect?"
```

source

45.4 Tipos

Base.supertype – Function.

```
supertype(T::DataType)
```

Return the supertype of DataType T.

```
julia> supertype(Int32)
Signed
```

source

Core.issubtype – Function.

```
issubtype(type1, type2)
```

Return `true` if and only if all values of `type1` are also of `type2`. Can also be written using the `<:` infix operator as `type1 <: type2`.

Examples

```
julia> issubtype(Int8, Int32)
false

julia> Int8 <: Integer
true
```

source

Base.:<: – Function.

```
<:(T1, T2)
```

Subtype operator, equivalent to `issubtype(T1, T2)`.

```
julia> Float64 <: AbstractFloat
true

julia> Vector{Int} <: AbstractArray
true

julia> Matrix{Float64} <: Matrix{AbstractFloat}
false
```

source

Base.:>: – Function.

```
>:(T1, T2)
```

Supertype operator, equivalent to `issubtype(T2, T1)`.

source

Base.subtypes – Function.

```
subtypes(T::DataType)
```

Return a list of immediate subtypes of DataType T. Note that all currently loaded subtypes are included, including those not visible in the current module.

```
julia> subtypes(Integer)
4-element Array{Union{DataType, UnionAll},1}:
 BigInt
 Bool
 Signed
 Unsigned
```

source

Base.typemin – Function.

```
typemin(T)
```

The lowest value representable by the given (real) numeric DataType T.

Examples

```
julia> typemin(Float16)
-Inf16

julia> typemin(Float32)
-Inf32
```

source

Base.typemax – Function.

```
typemax(T)
```

The highest value representable by the given (real) numeric DataType.

source

Base.realmin – Function.

```
realmin(T)
```

The smallest in absolute value non-subnormal value representable by the given floating-point DataType T.

source

Base.realmax – Function.

```
realmax(T)
```

The highest finite value representable by the given floating-point DataType T.

Examples

```
julia> realmax(Float16)
Float16(6.55e4)

julia> realmax(Float32)
3.4028235f38
```

source

Base.maxintfloat – Function.

```
maxintfloat(T)
```

The largest integer losslessly representable by the given floating-point DataType T.

source

```
maxintfloat(T, S)
```

The largest integer losslessly representable by the given floating-point DataType T that also does not exceed the maximum integer representable by the integer DataType S.

source

Base.sizeof – Method.

```
sizeof(T)
```

Size, in bytes, of the canonical binary representation of the given DataType T, if any.

Examples

```
julia> sizeof(Float32)
4

julia> sizeof(Complex128)
16
```

If T does not have a specific size, an error is thrown.

```
julia> sizeof(Base.LinAlg.LU)
ERROR: argument is an abstract type; size is indeterminate
Stacktrace:
 [1] sizeof(::Type{T} where T) at ./essentials.jl:159
```

source

Base.eps – Method.

```
eps(::Type{T}) where T<:AbstractFloat
eps()
```

Returns the *machine epsilon* of the floating point type T (`T = Float64` by default). This is defined as the gap between 1 and the next largest value representable by T, and is equivalent to `eps(one(T))`.

```
julia> eps()
2.220446049250313e-16

julia> eps(Float32)
1.1920929f-7

julia> 1.0 + eps()
1.0000000000000002

julia> 1.0 + eps()/2
1.0
```

source

Base.eps – Method.

```
eps(x::AbstractFloat)
```

Returns the *unit in last place* (ulp) of x. This is the distance between consecutive representable floating point values at x. In most cases, if the distance on either side of x is different, then the larger of the two is taken, that is

```
eps(x) == max(x-prevfloat(x), nextfloat(x)-x)
```

The exceptions to this rule are the smallest and largest finite values (e.g. `nextfloat(-Inf)` and `prevfloat(Inf)` for Float64), which round to the smaller of the values.

The rationale for this behavior is that `eps` bounds the floating point rounding error. Under the default `RoundNearest` rounding mode, if y is a real number and x is the nearest floating point number to y, then

$$|y - x| \leq \text{eps}(x)/2.$$

```
julia> eps(1.0)
2.220446049250313e-16

julia> eps(prevfloat(2.0))
2.220446049250313e-16

julia> eps(2.0)
4.440892098500626e-16

julia> x = prevfloat(Inf)      # largest finite Float64
1.7976931348623157e308

julia> x + eps(x)/2            # rounds up
Inf

julia> x + prevfloat(eps(x)/2) # rounds down
1.7976931348623157e308
```

source

Base.promote_type – Function.

```
promote_type(type1, type2)
```

Determine a type big enough to hold values of each argument type without loss, whenever possible. In some cases, where no type exists to which both types can be promoted losslessly, some loss is tolerated; for example, `promote_type(Int64, Float64)` returns `Float64` even though strictly, not all `Int64` values can be represented exactly as `Float64` values.

```
julia> promote_type(Int64, Float64)
Float64

julia> promote_type(Int32, Int64)
Int64

julia> promote_type(Float32, BigInt)
BigFloat
```

source

Base.promote_rule – Function.

```
promote_rule(type1, type2)
```

Specifies what type should be used by `promote` when given values of types `type1` and `type2`. This function should not be called directly, but should have definitions added to it for new types as appropriate.

source

Core.getfield – Function.

```
getfield(value, name::Symbol)
```

Extract a named field from a `value` of composite type. The syntax `a.b` calls `getfield(a, :b)`.

Example

```
julia> a = 1//2
1//2

julia> getfield(a, :num)
1
```

source

Core.setfield! – Function.

```
setfield!(value, name::Symbol, x)
```

Assign x to a named field in value of composite type. The syntax a.b = c calls setfield!(a, :b, c).

source

Base.fieldoffset – Function.

```
fieldoffset(type, i)
```

The byte offset of field i of a type relative to the data start. For example, we could use it in the following manner to summarize information about a struct:

```
julia> structinfo(T) = [(fieldoffset(T,i), fieldname(T,i), fieldtype(T,i)) for i =
↪  1:nfields(T)];

julia> structinfo(Base.Filesystem.StatStruct)
12-element Array{Tuple{UInt64,Symbol,DataType},1}:
 (0x0000000000000000, :device, UInt64)
 (0x0000000000000008, :inode, UInt64)
 (0x0000000000000010, :mode, UInt64)
 (0x0000000000000018, :nlink, Int64)
 (0x0000000000000020, :uid, UInt64)
 (0x0000000000000028, :gid, UInt64)
 (0x0000000000000030, :rdev, UInt64)
 (0x0000000000000038, :size, Int64)
 (0x0000000000000040, :blksize, Int64)
 (0x0000000000000048, :blocks, Int64)
 (0x0000000000000050, :mtime, Float64)
 (0x0000000000000058, :ctime, Float64)
```

source

Core.fieldtype – Function.

```
fieldtype(T, name::Symbol | index::Int)
```

Determine the declared type of a field (specified by name or index) in a composite DataType T.

```
julia> struct Foo

           x::Int64

           y::String

       end
```

```
julia> fieldtype(Foo, :x)
Int64

julia> fieldtype(Foo, 2)
String
```

source

Base.isimmutable – Function.

```
isimmutable(v)
```

Return `true` iff value `v` is immutable. See Mutable Composite Types for a discussion of immutability. Note that this function works on values, so if you give it a type, it will tell you that a value of `DataType` is mutable.

```
julia> isimmutable(1)
true

julia> isimmutable([1,2])
false
```

source

Base.isbits – Function.

```
isbits(T)
```

Return `true` if `T` is a "plain data" type, meaning it is immutable and contains no references to other values. Typical examples are numeric types such as `UInt8`, `Float64`, and `Complex{Float64}`.

```
julia> isbits(Complex{Float64})
true

julia> isbits(Complex)
false
```

source

Base.isleaftype – Function.

```
isleaftype(T)
```

Determine whether `T`'s only subtypes are itself and `Union{}`. This means `T` is a concrete type that can have instances.

```
julia> isleaftype(Complex)
false

julia> isleaftype(Complex{Float32})
true

julia> isleaftype(Vector{Complex})
true

julia> isleaftype(Vector{Complex{Float32}})
true
```

source

Base.typejoin – Function.

```
typejoin(T, S)
```

Compute a type that contains both T and S.

source

Base.typeintersect – Function.

```
typeintersect(T, S)
```

Compute a type that contains the intersection of T and S. Usually this will be the smallest such type or one close to it.

source

Base.Val – Type.

```
Val{c}
```

Create a "value type" out of c, which must be an `isbits` value. The intent of this construct is to be able to dispatch on constants, e.g., `f(Val{false})` allows you to dispatch directly (at compile-time) to an implementation `f(::Type{Val{false}})`, without having to test the boolean value at runtime.

source

Base.Enums.@enum – Macro.

```
@enum EnumName[::BaseType] value1[=x] value2[=y]
```

Create an `Enum{BaseType}` subtype with name `EnumName` and enum member values of `value1` and `value2` with optional assigned values of x and y, respectively. `EnumName` can be used just like other types and enum member values as regular values, such as

```
julia> @enum Fruit apple=1 orange=2 kiwi=3

julia> f(x::Fruit) = "I'm a Fruit with value: $(Int(x))"
f (generic function with 1 method)

julia> f(apple)
"I'm a Fruit with value: 1"
```

`BaseType`, which defaults to `Int32`, must be a primitive subtype of `Integer`. Member values can be converted between the enum type and `BaseType`. `read` and `write` perform these conversions automatically.

source

Base.instances – Function.

```
instances(T::Type)
```

Return a collection of all instances of the given type, if applicable. Mostly used for enumerated types (see `@enum`).

```
julia> @enum Color red blue green

julia> instances(Color)
(red::Color = 0, blue::Color = 1, green::Color = 2)
```

source

45.5 Funciones Genéricas

`Core.Function` – Type.

```
Function
```

Abstract type of all functions.

```
julia> isa(+, Function)
true

julia> typeof(sin)
Base.#sin

julia> ans <: Function
true
```

source

`Base.method_exists` – Function.

```
method_exists(f, Tuple type, world=typemax(UInt)) -> Bool
```

Determine whether the given generic function has a method matching the given `Tuple` of argument types with the upper bound of world age given by `world`.

```
julia> method_exists(length, Tuple{Array})
true
```

source

`Core.applicable` – Function.

```
applicable(f, args...) -> Bool
```

Determine whether the given generic function has a method applicable to the given arguments.

Examples

```
julia> function f(x, y)

           x + y

       end;

julia> applicable(f, 1)
false

julia> applicable(f, 1, 2)
true
```

source

`Core.invoke` – Function.

```
invoke(f, types <: Tuple, args...)
```

Invoke a method for the given generic function matching the specified types, on the specified arguments. The arguments must be compatible with the specified types. This allows invoking a method other than the most specific matching method, which is useful when the behavior of a more general definition is explicitly needed (often as part of the implementation of a more specific method of the same function).

source

`Base.invokelatest` – Function.

```
invokelatest(f, args...)
```

Calls `f(args...)`, but guarantees that the most recent method of `f` will be executed. This is useful in specialized circumstances, e.g. long-running event loops or callback functions that may call obsolete versions of a function `f`. (The drawback is that `invokelatest` is somewhat slower than calling `f` directly, and the type of the result cannot be inferred by the compiler.)

source

`Base.:|>` – Function.

```
|>(x, f)
```

Applies a function to the preceding argument. This allows for easy function chaining.

```
julia> [1:5;] |> x->x.^2 |> sum |> inv
0.01818181818181818
```

source

`Base.:∘` – Function.

```
f ∘ g
```

Compose functions: i.e. `(f ∘ g)(args...)` means `f(g(args...))`. The `∘` symbol can be entered in the Julia REPL (and most editors, appropriately configured) by typing `\circ<tab>`. Example:

```
julia> map(uppercasehex, 250:255)
6-element Array{String,1}:
 "FA"
 "FB"
 "FC"
 "FD"
 "FE"
 "FF"
```

source

45.6 Sintaxis

`Core.eval` – Function.

```
eval([m::Module], expr::Expr)
```

Evaluate an expression in the given module and return the result. Every `Module` (except those defined with `baremodule`) has its own 1-argument definition of `eval`, which evaluates expressions in that module.

source

Base.@eval – Macro.

```
@eval [mod,] ex
```

Evaluate an expression with values interpolated into it using `eval`. If two arguments are provided, the first is the module to evaluate in.

source

Base.evalfile – Function.

```
evalfile(path::AbstractString, args::Vector{String}=String[])
```

Load the file using `include`, evaluate all expressions, and return the value of the last one.

source

Base.esc – Function.

```
esc(e::ANY)
```

Only valid in the context of an `Expr` returned from a macro. Prevents the macro hygiene pass from turning embedded variables into gensym variables. See the Macros section of the Metaprogramming chapter of the manual for more details and examples.

source

Base.@inbounds – Macro.

```
@inbounds(blk)
```

Eliminates array bounds checking within expressions.

In the example below the bound check of array A is skipped to improve performance.

```
function sum(A::AbstractArray)
    r = zero(eltype(A))
    for i = 1:length(A)
        @inbounds r += A[i]
    end
    return r
end
```

Warning

Using `@inbounds` may return incorrect results/crashes/corruption for out-of-bounds indices. The user is responsible for checking it manually.

source

Base.@inline – Macro.

```
@inline
```

Give a hint to the compiler that this function is worth inlining.

Small functions typically do not need the `@inline` annotation, as the compiler does it automatically. By using `@inline` on bigger functions, an extra nudge can be given to the compiler to inline it. This is shown in the following example:

```
@inline function bigfunction(x)
    #=
        Function Definition
    =#
end
```

source

Base.@noinline – Macro.

```
@noinline
```

Prevents the compiler from inlining a function.

Small functions are typically inlined automatically. By using `@noinline` on small functions, auto-inlining can be prevented. This is shown in the following example:

```
@noinline function smallfunction(x)
    #=
        Function Definition
    =#
end
```

source

Base.gensym – Function.

```
gensym([tag])
```

Generates a symbol which will not conflict with other variable names.

source

Base.@gensym – Macro.

```
@gensym
```

Generates a gensym symbol for a variable. For example, `@gensym x y` is transformed into `x = gensym("x"); y = gensym("y")`.

source

Base.@polly – Macro.

```
@polly
```

Tells the compiler to apply the polyhedral optimizer Polly to a function.

source

Base.parse – Method.

```
parse(str, start; greedy=true, raise=true)
```

Parse the expression string and return an expression (which could later be passed to eval for execution). `start` is the index of the first character to start parsing. If `greedy` is `true` (default), `parse` will try to consume as much input as it can; otherwise, it will stop as soon as it has parsed a valid expression. Incomplete but otherwise syntactically valid expressions will return `Expr(:incomplete, "(error message)")`. If `raise` is `true` (default), syntax errors other than incomplete expressions will raise an error. If `raise` is `false`, `parse` will return an expression that will raise an error upon evaluation.

```
julia> parse("x = 3, y = 5", 7)
(:(y = 5), 13)

julia> parse("x = 3, y = 5", 5)
(:((3, y) = 5), 13)
```

source

Base.parse – Method.

```
parse(str; raise=true)
```

Parse the expression string greedily, returning a single expression. An error is thrown if there are additional characters after the first expression. If `raise` is `true` (default), syntax errors will raise an error; otherwise, `parse` will return an expression that will raise an error upon evaluation.

```
julia> parse("x = 3")
:(x = 3)

julia> parse("x = ")
:($(Expr(:incomplete, "incomplete: premature end of input")))

julia> parse("1.0.2")
ERROR: ParseError("invalid numeric constant \"1.0.\"")
Stacktrace:
[...]

julia> parse("1.0.2"; raise = false)
:($(Expr(:error, "invalid numeric constant \"1.0.\"")))
```

source

45.7 *Nullables*

Base.Nullable – Type.

```
Nullable(x, hasvalue::Bool=true)
```

Wrap value x in an object of type `Nullable`, which indicates whether a value is present. `Nullable(x)` yields a non-empty wrapper and `Nullable{T}()` yields an empty instance of a wrapper that might contain a value of type T.

`Nullable(x, false)` yields `Nullable{typeof(x)}()` with x stored in the result's `value` field.

Examples

```
julia> Nullable(1)
Nullable{Int64}(1)

julia> Nullable{Int64}()
Nullable{Int64}()

julia> Nullable(1, false)
Nullable{Int64}()

julia> dump(Nullable(1, false))
```

```
Nullable{Int64}
  hasvalue: Bool false
  value: Int64 1
```

source

Base.get – Method.

```
get(x::Nullable[, y])
```

Attempt to access the value of x. Returns the value if it is present; otherwise, returns y if provided, or throws a NullException if not.

source

Base.isnull – Function.

```
isnull(x)
```

Return whether or not x is null for Nullable x; return false for all other x.

Examples

```
julia> x = Nullable(1, false)
Nullable{Int64}()

julia> isnull(x)
true

julia> x = Nullable(1, true)
Nullable{Int64}(1)

julia> isnull(x)
false

julia> x = 1
1

julia> isnull(x)
false
```

source

Base.unsafe_get – Function.

```
unsafe_get(x)
```

Return the value of x for Nullable x; return x for all other x.

This method does not check whether or not x is null before attempting to access the value of x for x::Nullable (hence "unsafe").

```
julia> x = Nullable(1)
Nullable{Int64}(1)

julia> unsafe_get(x)
1
```

```
julia> x = Nullable{String}()
Nullable{String}()

julia> unsafe_get(x)
ERROR: UndefRefError: access to undefined reference
Stacktrace:
 [1] unsafe_get(::Nullable{String}) at ./nullable.jl:125

julia> x = 1
1

julia> unsafe_get(x)
1
```

source

45.8 Sistema

Base.run – Function.

```
run(command, args...)
```

Run a command object, constructed with backticks. Throws an error if anything goes wrong, including the process exiting with a non-zero status.

source

Base.spawn – Function.

```
spawn(command)
```

Run a command object asynchronously, returning the resulting `Process` object.

source

Base.DevNull – Constant.

```
DevNull
```

Used in a stream redirect to discard all data written to it. Essentially equivalent to /dev/null on Unix or NUL on Windows. Usage:

```
run(pipeline(`cat test.txt`, DevNull))
```

source

Base.success – Function.

```
success(command)
```

Run a command object, constructed with backticks, and tell whether it was successful (exited with a code of 0). An exception is raised if the process cannot be started.

source

Base.process_running – Function.

```
process_running(p::Process)
```

Determine whether a process is currently running.

source

Base.process_exited – Function.

```
process_exited(p::Process)
```

Determine whether a process has exited.

source

Base.kill – Method.

```
kill(p::Process, signum=SIGTERM)
```

Send a signal to a process. The default is to terminate the process.

source

Base.Sys.set_process_title – Function.

```
Sys.set_process_title(title::AbstractString)
```

Set the process title. No-op on some operating systems.

source

Base.Sys.get_process_title – Function.

```
Sys.get_process_title()
```

Get the process title. On some systems, will always return an empty string.

source

Base.readandwrite – Function.

```
readandwrite(command)
```

Starts running a command asynchronously, and returns a tuple (stdout,stdin,process) of the output stream and input stream of the process, and the process object itself.

source

Base.ignorestatus – Function.

```
ignorestatus(command)
```

Mark a command object so that running it will not throw an error if the result code is non-zero.

source

Base.detach – Function.

```
detach(command)
```

Mark a command object so that it will be run in a new process group, allowing it to outlive the julia process, and not have Ctrl-C interrupts passed to it.

source

Base.Cmd – Type.

```
Cmd(cmd::Cmd; ignorestatus, detach, windows_verbatim, windows_hide, env, dir)
```

Construct a new Cmd object, representing an external program and arguments, from cmd, while changing the settings of the optional keyword arguments:

- ignorestatus::Bool: If true (defaults to false), then the Cmd will not throw an error if the return code is nonzero.
- detach::Bool: If true (defaults to false), then the Cmd will be run in a new process group, allowing it to outlive the julia process and not have Ctrl-C passed to it.
- windows_verbatim::Bool: If true (defaults to false), then on Windows the Cmd will send a command-line string to the process with no quoting or escaping of arguments, even arguments containing spaces. (On Windows, arguments are sent to a program as a single "command-line" string, and programs are responsible for parsing it into arguments. By default, empty arguments and arguments with spaces or tabs are quoted with double quotes " in the command line, and \ or " are preceded by backslashes. windows_verbatim=true is useful for launching programs that parse their command line in nonstandard ways.) Has no effect on non-Windows systems.
- windows_hide::Bool: If true (defaults to false), then on Windows no new console window is displayed when the Cmd is executed. This has no effect if a console is already open or on non-Windows systems.
- env: Set environment variables to use when running the Cmd. env is either a dictionary mapping strings to strings, an array of strings of the form "var=val", an array or tuple of "var"=>val pairs, or nothing. In order to modify (rather than replace) the existing environment, create env by copy(ENV) and then set env["var"]=val as desired.
- dir::AbstractString: Specify a working directory for the command (instead of the current directory).

For any keywords that are not specified, the current settings from cmd are used. Normally, to create a Cmd object in the first place, one uses backticks, e.g.

```
Cmd(`echo "Hello world"`, ignorestatus=true, detach=false)
```

source

Base.setenv – Function.

```
setenv(command::Cmd, env; dir="")
```

Set environment variables to use when running the given command. env is either a dictionary mapping strings to strings, an array of strings of the form "var=val", or zero or more "var"=>val pair arguments. In order to modify (rather than replace) the existing environment, create env by copy(ENV) and then setting env["var"]=val as desired, or use withenv.

The dir keyword argument can be used to specify a working directory for the command.

source

Base.withenv – Function.

```
withenv(f::Function, kv::Pair...)
```

Execute f() in an environment that is temporarily modified (not replaced as in setenv) by zero or more "var"=>val arguments kv. withenv is generally used via the withenv(kv...) do ... end syntax. A value of nothing can be used to temporarily unset an environment variable (if it is set). When withenv returns, the original environment has been restored.

source

Base.pipeline – Method.

```
pipeline(from, to, ...)
```

Create a pipeline from a data source to a destination. The source and destination can be commands, I/O streams, strings, or results of other `pipeline` calls. At least one argument must be a command. Strings refer to filenames. When called with more than two arguments, they are chained together from left to right. For example `pipeline(a,b,c)` is equivalent to `pipeline(pipeline(a,b),c)`. This provides a more concise way to specify multi-stage pipelines.

Examples:

```
run(pipeline(`ls`, `grep xyz`))
run(pipeline(`ls`, "out.txt"))
run(pipeline("out.txt", `grep xyz`))
```

source

Base.pipeline – Method.

```
pipeline(command; stdin, stdout, stderr, append=false)
```

Redirect I/O to or from the given `command`. Keyword arguments specify which of the command's streams should be redirected. `append` controls whether file output appends to the file. This is a more general version of the 2-argument `pipeline` function. `pipeline(from, to)` is equivalent to `pipeline(from, stdout=to)` when `from` is a command, and to `pipeline(to, stdin=from)` when `from` is another kind of data source.

Examples:

```
run(pipeline(`dothings`, stdout="out.txt", stderr="errs.txt"))
run(pipeline(`update`, stdout="log.txt", append=true))
```

source

Base.Libc.gethostname – Function.

```
gethostname() -> AbstractString
```

Get the local machine's host name.

source

Base.getipaddr – Function.

```
getipaddr() -> IPAddr
```

Get the IP address of the local machine.

source

Base.Libc.getpid – Function.

```
getpid() -> Int32
```

Get Julia's process ID.

source

Base.Libc.time – Method.

```
time()
```

Get the system time in seconds since the epoch, with fairly high (typically, microsecond) resolution.

source

Base.time_ns – Function.

```
time_ns()
```

Get the time in nanoseconds. The time corresponding to 0 is undefined, and wraps every 5.8 years.

source

Base.tic – Function.

```
tic()
```

Set a timer to be read by the next call to toc or toq. The macro call `@time expr` can also be used to time evaluation.

```
julia> tic()
0x0000c45bc7abac95

julia> sleep(0.3)

julia> toc()
elapsed time: 0.302745944 seconds
0.302745944
```

source

Base.toc – Function.

```
toc()
```

Print and return the time elapsed since the last tic. The macro call `@time expr` can also be used to time evaluation.

```
julia> tic()
0x0000c45bc7abac95

julia> sleep(0.3)

julia> toc()
elapsed time: 0.302745944 seconds
0.302745944
```

source

Base.toq – Function.

```
toq()
```

Return, but do not print, the time elapsed since the last tic. The macro calls `@timed expr` and `@elapsed expr` also return evaluation time.

```
julia> tic()
0x0000c46477a9675d

julia> sleep(0.3)

julia> toq()
0.302251004
```

source

Base.@time – Macro.

```
@time
```

A macro to execute an expression, printing the time it took to execute, the number of allocations, and the total number of bytes its execution caused to be allocated, before returning the value of the expression.

See also @timev, @timed, @elapsed, and @allocated.

```
julia> @time rand(10^6);
  0.001525 seconds (7 allocations: 7.630 MiB)

julia> @time begin

           sleep(0.3)

           1+1

       end
  0.301395 seconds (8 allocations: 336 bytes)
```

source

Base.@timev – Macro.

```
@timev
```

This is a verbose version of the @time macro. It first prints the same information as @time, then any non-zero memory allocation counters, and then returns the value of the expression.

See also @time, @timed, @elapsed, and @allocated.

```
julia> @timev rand(10^6);
  0.001006 seconds (7 allocations: 7.630 MiB)
elapsed time (ns): 1005567
bytes allocated:   8000256
pool allocs:       6
malloc() calls:    1
```

source

Base.@timed – Macro.

```
@timed
```

A macro to execute an expression, and return the value of the expression, elapsed time, total bytes allocated, garbage collection time, and an object with various memory allocation counters.

See also @time, @timev, @elapsed, and @allocated.

```
julia> val, t, bytes, gctime, memallocs = @timed rand(10^6);

julia> t
0.006634834

julia> bytes
8000256

julia> gctime
0.0055765

julia> fieldnames(typeof(memallocs))
9-element Array{Symbol,1}:
 :allocd
 :malloc
 :realloc
 :poolalloc
 :bigalloc
 :freecall
 :total_time
 :pause
 :full_sweep

julia> memallocs.total_time
5576500
```

source

Base.@elapsed – Macro.

```
@elapsed
```

A macro to evaluate an expression, discarding the resulting value, instead returning the number of seconds it took to execute as a floating-point number.

See also @time, @timev, @timed, and @allocated.

```
julia> @elapsed sleep(0.3)
0.301391426
```

source

Base.@allocated – Macro.

```
@allocated
```

A macro to evaluate an expression, discarding the resulting value, instead returning the total number of bytes allocated during evaluation of the expression. Note: the expression is evaluated inside a local function, instead of the current context, in order to eliminate the effects of compilation, however, there still may be some allocations due to JIT compilation. This also makes the results inconsistent with the `@time` macros, which do not try to adjust for the effects of compilation.

See also @time, @timev, @timed, and @elapsed.

```
julia> @allocated rand(10^6)
8000080
```

source

Base.EnvHash – Type.

```
EnvHash() -> EnvHash
```

A singleton of this type provides a hash table interface to environment variables.

source

Base.ENV – Constant.

```
ENV
```

Reference to the singleton EnvHash, providing a dictionary interface to system environment variables.

source

Base.is_unix – Function.

```
is_unix([os])
```

Predicate for testing if the OS provides a Unix-like interface. See documentation in Handling Operating System Variation.

source

Base.is_apple – Function.

```
is_apple([os])
```

Predicate for testing if the OS is a derivative of Apple Macintosh OS X or Darwin. See documentation in Handling Operating System Variation.

source

Base.is_linux – Function.

```
is_linux([os])
```

Predicate for testing if the OS is a derivative of Linux. See documentation in Handling Operating System Variation.

source

Base.is_bsd – Function.

```
is_bsd([os])
```

Predicate for testing if the OS is a derivative of BSD. See documentation in Handling Operating System Variation.

source

Base.is_windows – Function.

```
is_windows([os])
```

Predicate for testing if the OS is a derivative of Microsoft Windows NT. See documentation in Handling Operating System Variation.

source

Base.Sys.windows_version – Function.

```
Sys.windows_version()
```

Returns the version number for the Windows NT Kernel as a (major, minor) pair, or `(0, 0)` if this is not running on Windows.

source

Base.@static – Macro.

```
@static
```

Partially evaluates an expression at parse time.

For example, `@static is_windows() ? foo : bar` will evaluate `is_windows()` and insert either `foo` or `bar` into the expression. This is useful in cases where a construct would be invalid on other platforms, such as a `ccall` to a non-existent function. `@static if is_apple() foo end` and `@static foo <&&, ||> bar` are also valid syntax.

source

45.9 Errores

Base.error – Function.

```
error(message::AbstractString)
```

Raise an `ErrorException` with the given message.

source

Core.throw – Function.

```
throw(e)
```

Throw an object as an exception.

source

Base.rethrow – Function.

```
rethrow([e])
```

Throw an object without changing the current exception backtrace. The default argument is the current exception (if called within a `catch` block).

source

Base.backtrace – Function.

```
backtrace()
```

Get a backtrace object for the current program point.

source

Base.catch_backtrace – Function.

```
catch_backtrace()
```

Get the backtrace of the current exception, for use within `catch` blocks.

source

`Base.assert` – Function.

```
assert(cond)
```

Throw an `AssertionError` if `cond` is `false`. Also available as the macro `@assert expr`.

source

`Base.@assert` – Macro.

```
@assert cond [text]
```

Throw an `AssertionError` if `cond` is `false`. Preferred syntax for writing assertions. Message `text` is optionally displayed upon assertion failure.

source

`Base.ArgumentError` – Type.

```
ArgumentError(msg)
```

The parameters to a function call do not match a valid signature. Argument `msg` is a descriptive error string.

source

`Base.AssertionError` – Type.

```
AssertionError([msg])
```

The asserted condition did not evaluate to `true`. Optional argument `msg` is a descriptive error string.

source

`Core.BoundsError` – Type.

```
BoundsError([a],[i])
```

An indexing operation into an array, `a`, tried to access an out-of-bounds element, `i`.

source

`Base.DimensionMismatch` – Type.

```
DimensionMismatch([msg])
```

The objects called do not have matching dimensionality. Optional argument `msg` is a descriptive error string.

source

`Core.DivideError` – Type.

```
DivideError()
```

Integer division was attempted with a denominator value of 0.

source

`Core.DomainError` – Type.

```
DomainError()
```

The arguments to a function or constructor are outside the valid domain.

source

Base.EOFError – Type.

```
EOFError()
```

No more data was available to read from a file or stream.

source

Core.ErrorException – Type.

```
ErrorException(msg)
```

Generic error type. The error message, in the `.msg` field, may provide more specific details.

source

Core.InexactError – Type.

```
InexactError()
```

Type conversion cannot be done exactly.

source

Core.InterruptException – Type.

```
InterruptException()
```

The process was stopped by a terminal interrupt (CTRL+C).

source

Base.KeyError – Type.

```
KeyError(key)
```

An indexing operation into an `Associative` (`Dict`) or `Set` like object tried to access or delete a non-existent element.

source

Base.LoadError – Type.

```
LoadError(file::AbstractString, line::Int, error)
```

An error occurred while `including`, `requireing`, or `using` a file. The error specifics should be available in the `.error` field.

source

Base.MethodError – Type.

```
MethodError(f, args)
```

A method with the required type signature does not exist in the given generic function. Alternatively, there is no unique most-specific method.

source

Base.NullException – Type.

```
NullException()
```

An attempted access to a Nullable with no defined value.

source

Core.OutOfMemoryError – Type.

```
OutOfMemoryError()
```

An operation allocated too much memory for either the system or the garbage collector to handle properly.

source

Core.ReadOnlyMemoryError – Type.

```
ReadOnlyMemoryError()
```

An operation tried to write to memory that is read-only.

source

Core.OverflowError – Type.

```
OverflowError()
```

The result of an expression is too large for the specified type and will cause a wraparound.

source

Base.ParseError – Type.

```
ParseError(msg)
```

The expression passed to the parse function could not be interpreted as a valid Julia expression.

source

Base.Distributed.ProcessExitedException – Type.

```
ProcessExitedException()
```

After a client Julia process has exited, further attempts to reference the dead child will throw this exception.

source

Core.StackOverflowError – Type.

```
StackOverflowError()
```

The function call grew beyond the size of the call stack. This usually happens when a call recurses infinitely.

source

Base.SystemError – Type.

```
SystemError(prefix::AbstractString, [errno::Int32])
```

A system call failed with an error code (in the `errno` global variable).

source

Core.TypeError – Type.

```
TypeError(func::Symbol, context::AbstractString, expected::Type, got)
```

A type assertion failure, or calling an intrinsic function with an incorrect argument type.

source

Core.UndefRefError – Type.

```
UndefRefError()
```

The item or field is not defined for the given object.

source

Core.UndefVarError – Type.

```
UndefVarError(var::Symbol)
```

A symbol in the current scope is not defined.

source

Base.InitError – Type.

```
InitError(mod::Symbol, error)
```

An error occurred when running a module's `__init__` function. The actual error thrown is available in the `.error` field.

source

Base.retry – Function.

```
retry(f::Function;  delays=ExponentialBackOff(), check=nothing) -> Function
```

Returns an anonymous function that calls function `f`. If an exception arises, `f` is repeatedly called again, each time `check` returns `true`, after waiting the number of seconds specified in `delays`. `check` should input `delays`'s current state and the `Exception`.

Examples

```
retry(f, delays=fill(5.0, 3))
retry(f, delays=rand(5:10, 2))
retry(f, delays=Base.ExponentialBackOff(n=3, first_delay=5, max_delay=1000))
retry(http_get, check=(s,e)->e.status == "503")(url)
retry(read, check=(s,e)->isa(e, UVError))(io, 128; all=false)
```

source

Base.ExponentialBackOff – Type.

```
ExponentialBackOff(; n=1, first_delay=0.05, max_delay=10.0, factor=5.0, jitter=0.1)
```

A `Float64` iterator of length `n` whose elements exponentially increase at a rate in the interval `factor` * (1 ± `jitter`). The first element is `first_delay` and all elements are clamped to `max_delay`.

source

45.10 Eventos

`Base.Timer` – Method.

```
Timer(callback::Function, delay, repeat=0)
```

Create a timer to call the given `callback` function. The `callback` is passed one argument, the timer object itself. The callback will be invoked after the specified initial `delay`, and then repeating with the given `repeat` interval. If `repeat` is `0`, the timer is only triggered once. Times are in seconds. A timer is stopped and has its resources freed by calling `close` on it.

source

`Base.Timer` – Type.

```
Timer(delay, repeat=0)
```

Create a timer that wakes up tasks waiting for it (by calling `wait` on the timer object) at a specified interval. Times are in seconds. Waiting tasks are woken with an error when the timer is closed (by `close`. Use `isopen` to check whether a timer is still active.

source

`Base.AsyncCondition` – Type.

```
AsyncCondition()
```

Create a async condition that wakes up tasks waiting for it (by calling `wait` on the object) when notified from C by a call to `uv_async_send`. Waiting tasks are woken with an error when the object is closed (by `close`. Use `isopen` to check whether it is still active.

source

`Base.AsyncCondition` – Method.

```
AsyncCondition(callback::Function)
```

Create a async condition that calls the given `callback` function. The `callback` is passed one argument, the async condition object itself.

source

45.11 Reflexión

`Base.module_name` – Function.

```
module_name(m::Module) -> Symbol
```

Get the name of a `Module` as a `Symbol`.

```
julia> module_name(Base.LinAlg)
:LinAlg
```

source

`Base.module_parent` – Function.

```
module_parent(m::Module) -> Module
```

Get a module's enclosing `Module`. `Main` is its own parent, as is `LastMain` after `workspace()`.

```
julia> module_parent(Main)
Main

julia> module_parent(Base.LinAlg.BLAS)
Base.LinAlg
```

source

`Base.current_module` – Function.

```
current_module() -> Module
```

Get the *dynamically* current `Module`, which is the `Module` code is currently being read from. In general, this is not the same as the module containing the call to this function.

source

`Base.fullname` – Function.

```
fullname(m::Module)
```

Get the fully-qualified name of a module as a tuple of symbols. For example,

```
julia> fullname(Base.Pkg)
(:Base, :Pkg)

julia> fullname(Main)
()
```

source

`Base.names` – Function.

```
names(x::Module, all::Bool=false, imported::Bool=false)
```

Get an array of the names exported by a `Module`, excluding deprecated names. If `all` is true, then the list also includes non-exported names defined in the module, deprecated names, and compiler-generated names. If `imported` is true, then names explicitly imported from other modules are also included.

As a special case, all names defined in `Main` are considered "exported", since it is not idiomatic to explicitly export names from `Main`.

source

`Core.nfields` – Function.

```
nfields(x::DataType) -> Int
```

Get the number of fields of a `DataType`.

source

`Base.fieldnames` – Function.

```
fieldnames(x::DataType)
```

Get an array of the fields of a `DataType`.

```
julia> fieldnames(Hermitian)
2-element Array{Symbol,1}:
 :data
 :uplo
```

source

Base.fieldname – Function.

```
fieldname(x::DataType, i::Integer)
```

Get the name of field `i` of a `DataType`.

```
julia> fieldname(SparseMatrixCSC,1)
:m

julia> fieldname(SparseMatrixCSC,5)
:nzval
```

source

Base.datatype_module – Function.

```
Base.datatype_module(t::DataType) -> Module
```

Determine the module containing the definition of a `DataType`.

source

Base.datatype_name – Function.

```
Base.datatype_name(t) -> Symbol
```

Get the name of a (potentially UnionAll-wrapped) `DataType` (without its parent module) as a symbol.

source

Base.isconst – Function.

```
isconst([m::Module], s::Symbol) -> Bool
```

Determine whether a global is declared `const` in a given `Module`. The default `Module` argument is `current_module()`.

source

Base.function_name – Function.

```
Base.function_name(f::Function) -> Symbol
```

Get the name of a generic `Function` as a symbol, or `:anonymous`.

source

Base.function_module – Method.

```
Base.function_module(f::Function) -> Module
```

Determine the module containing the (first) definition of a generic function.

source

Base.function_module – Method.

```
Base.function_module(f::Function, types) -> Module
```

Determine the module containing a given definition of a generic function.

source

Base.functionloc – Method.

```
functionloc(f::Function, types)
```

Returns a tuple `(filename,line)` giving the location of a generic `Function` definition.

source

Base.functionloc – Method.

```
functionloc(m::Method)
```

Returns a tuple `(filename,line)` giving the location of a `Method` definition.

source

Base.@functionloc – Macro.

```
@functionloc
```

Applied to a function or macro call, it evaluates the arguments to the specified call, and returns a tuple `(filename,line)` giving the location for the method that would be called for those arguments. It calls out to the `functionloc` function.

source

45.12 Interioridades

Base.gc – Function.

```
gc()
```

Perform garbage collection. This should not generally be used.

source

Base.gc_enable – Function.

```
gc_enable(on::Bool)
```

Control whether garbage collection is enabled using a boolean argument (`true` for enabled, `false` for disabled). Returns previous GC state. Disabling garbage collection should be used only with extreme caution, as it can cause memory use to grow without bound.

source

Base.macroexpand – Function.

```
macroexpand(x)
```

Takes the expression `x` and returns an equivalent expression with all macros removed (expanded).

source

Base.@macroexpand – Macro.

```
@macroexpand
```

Return equivalent expression with all macros removed (expanded).

There is a subtle difference between `@macroexpand` and `macroexpand` in that expansion takes place in different contexts. This is best seen in the following example:

```
julia> module M

           macro m()

               1

           end

           function f()

               (@macroexpand(@m), macroexpand(:(@m)))

           end

       end
M

julia> macro m()

           2

       end
@m (macro with 1 method)

julia> M.f()
(1, 2)
```

With `@macroexpand` the expression expands where `@macroexpand` appears in the code (module `M` in the example). With `macroexpand` the expression expands in the current module where the code was finally called (REPL in the example). Note that when calling `macroexpand` or `@macroexpand` directly from the REPL, both of these contexts coincide, hence there is no difference.

source

Base.expand – Function.

```
expand(x)
```

Takes the expression x and returns an equivalent expression in lowered form. See also code_lowered.

source

Base.code_lowered – Function.

```
code_lowered(f, types)
```

Returns an array of lowered ASTs for the methods matching the given generic function and type signature.

source

Base.@code_lowered – Macro.

```
@code_lowered
```

Evaluates the arguments to the function or macro call, determines their types, and calls code_lowered on the resulting expression.

source

Base.code_typed – Function.

```
code_typed(f, types; optimize=true)
```

Returns an array of lowered and type-inferred ASTs for the methods matching the given generic function and type signature. The keyword argument optimize controls whether additional optimizations, such as inlining, are also applied.

source

Base.@code_typed – Macro.

```
@code_typed
```

Evaluates the arguments to the function or macro call, determines their types, and calls code_typed on the resulting expression.

source

Base.code_warntype – Function.

```
code_warntype([io::IO], f, types)
```

Prints lowered and type-inferred ASTs for the methods matching the given generic function and type signature to io which defaults to STDOUT. The ASTs are annotated in such a way as to cause "non-leaf" types to be emphasized (if color is available, displayed in red). This serves as a warning of potential type instability. Not all non-leaf types are particularly problematic for performance, so the results need to be used judiciously. See @code_warntype for more information.

source

Base.@code_warntype – Macro.

```
@code_warntype
```

Evaluates the arguments to the function or macro call, determines their types, and calls code_warntype on the resulting expression.

source

Base.code_llvm – Function.

```
code_llvm([io], f, types)
```

Prints the LLVM bitcodes generated for running the method matching the given generic function and type signature to io which defaults to STDOUT.

All metadata and dbg.* calls are removed from the printed bitcode. Use code_llvm_raw for the full IR.

source

Base.@code_llvm – Macro.

```
@code_llvm
```

Evaluates the arguments to the function or macro call, determines their types, and calls code_llvm on the resulting expression.

source

Base.code_native – Function.

```
code_native([io], f, types, [syntax])
```

Prints the native assembly instructions generated for running the method matching the given generic function and type signature to io which defaults to STDOUT. Switch assembly syntax using syntax symbol parameter set to :att for AT&T syntax or :intel for Intel syntax. Output is AT&T syntax by default.

source

Base.@code_native – Macro.

```
@code_native
```

Evaluates the arguments to the function or macro call, determines their types, and calls code_native on the resulting expression.

source

Base.precompile – Function.

```
precompile(f,args::Tuple{Vararg{Any}})
```

Compile the given function f for the argument tuple (of types) args, but do not execute it.

source

Chapter 46

Colecciones y Estructuras de Datos

46.1 Iteración

La iteración secuencial es implementada por los métodos `start()`, `done()` y `next()`. El bucle `for` general:

```
for i = I   # o  "for i in I"
    # cuerpo
end
```

es traducido a:

```
state = start(I)
while !done(I, state)
    (i, state) = next(I, state)
    # cuerpo
end
```

El objeto `state` puede ser cualquier cosa, y debería ser elegido apropiadamente para cada tipo iterable. Ver la sección del manual sobre la interfaz de iteración para más detalles sobre definir un tipo iterable personalizado.

`Base.start` – Function.

```
start(iter) -> state
```

Get initial iteration state for an iterable object.

Examples

```
julia> start(1:5)
1

julia> start([1;2;3])
1

julia> start([4;2;3])
1
```

source

`Base.done` – Function.

```
done(iter, state) -> Bool
```

Test whether we are done iterating.

Examples

```
julia> done(1:5, 3)
false

julia> done(1:5, 5)
false

julia> done(1:5, 6)
true
```

source

`Base.next` – Function.

```
next(iter, state) -> item, state
```

For a given iterable object and iteration state, return the current item and the next iteration state.

Examples

```
julia> next(1:5, 3)
(3, 4)

julia> next(1:5, 5)
(5, 6)
```

source

`Base.iteratorsize` – Function.

```
iteratorsize(itertype::Type) -> IteratorSize
```

Given the type of an iterator, returns one of the following values:

- `SizeUnknown()` if the length (number of elements) cannot be determined in advance.
- `HasLength()` if there is a fixed, finite length.
- `HasShape()` if there is a known length plus a notion of multidimensional shape (as for an array). In this case the `size` function is valid for the iterator.
- `IsInfinite()` if the iterator yields values forever.

The default value (for iterators that do not define this function) is `HasLength()`. This means that most iterators are assumed to implement `length`.

This trait is generally used to select between algorithms that pre-allocate space for their result, and algorithms that resize their result incrementally.

```
julia> Base.iteratorsize(1:5)
Base.HasShape()

julia> Base.iteratorsize((2,3))
Base.HasLength()
```

source

Base.iteratoreltype – Function.

```
iteratoreltype(itertype::Type) -> IteratorEltype
```

Given the type of an iterator, returns one of the following values:

- `EltypeUnknown()` if the type of elements yielded by the iterator is not known in advance.
- `HasEltype()` if the element type is known, and `eltype` would return a meaningful value.

`HasEltype()` is the default, since iterators are assumed to implement `eltype`.

This trait is generally used to select between algorithms that pre-allocate a specific type of result, and algorithms that pick a result type based on the types of yielded values.

```
julia> Base.iteratoreltype(1:5)
Base.HasEltype()
```

source

Completamente implementada por:

- `Range`
- `UnitRange`
- `Tuple`
- `Number`
- `AbstractArray`
- `IntSet`
- `ObjectIdDict`
- `Dict`
- `WeakKeyDict`
- `EachLine`
- `AbstractString`
- `Set`

46.2 Colecciones Generales

Base.isempty – Function.

```
isempty(collection) -> Bool
```

Determine whether a collection is empty (has no elements).

Examples

```
julia> isempty([])
true

julia> isempty([1 2 3])
false
```

source

`Base.empty!` – Function.

```
empty!(collection) -> collection
```

Remove all elements from a `collection`.

```
julia> A = Dict("a" => 1, "b" => 2)
Dict{String,Int64} with 2 entries:
  "b" => 2
  "a" => 1

julia> empty!(A);

julia> A
Dict{String,Int64} with 0 entries
```

source

`Base.length` – Method.

```
length(collection) -> Integer
```

For ordered, indexable collections, returns the maximum index `i` for which `getindex(collection, i)` is valid. For unordered collections, returns the number of elements.

Examples

```
julia> length(1:5)
5

julia> length([1; 2; 3; 4])
4
```

source

`Base.endof` – Function.

```
endof(collection) -> Integer
```

Returns the last index of the collection.

Example

```
julia> endof([1,2,4])
3
```

source

Completamente implementado por:

- Range
- UnitRange
- Tuple
- Number
- AbstractArray
- IntSet
- ObjectIdDict
- Dict
- WeakKeyDict
- AbstractString
- Set

46.3 Colecciones Iterables

Base.in – Function.

```
in(item, collection) -> Bool
(item,collection) -> Bool
(collection,item) -> Bool
(item,collection) -> Bool
(collection,item) -> Bool
```

Determine whether an item is in the given collection, in the sense that it is == to one of the values generated by iterating over the collection. Some collections need a slightly different definition; for example Sets check whether the item isequal to one of the elements. Dicts look for (key,value) pairs, and the key is compared using isequal. To test for the presence of a key in a dictionary, use haskey or k in keys(dict).

```
julia> a = 1:3:20
1:3:19

julia> 4 in a
true

julia> 5 in a
false
```

source

Base.eltype – Function.

```
eltype(type)
```

Determine the type of the elements generated by iterating a collection of the given type. For associative collection types, this will be a Pair{KeyType,ValType}. The definition eltype(x) = eltype(typeof(x)) is provided for convenience so that instances can be passed instead of types. However the form that accepts a type argument should be defined for new types.

```
julia> eltype(ones(Float32,2,2))
Float32

julia> eltype(ones(Int8,2,2))
Int8
```

source

Base.indexin – Function.

```
indexin(a, b)
```

Returns a vector containing the highest index in b for each value in a that is a member of b . The output vector contains 0 wherever a is not a member of b.

Examples

```
julia> a = ['a', 'b', 'c', 'b', 'd', 'a'];

julia> b = ['a','b','c'];

julia> indexin(a,b)
6-element Array{Int64,1}:
 1
 2
 3
 2
 0
 1

julia> indexin(b,a)
3-element Array{Int64,1}:
 6
 4
 3
```

source

Base.findin – Function.

```
findin(a, b)
```

Returns the indices of elements in collection a that appear in collection b.

Examples

```
julia> a = collect(1:3:15)
5-element Array{Int64,1}:
  1
  4
  7
 10
 13

julia> b = collect(2:4:10)
3-element Array{Int64,1}:
  2
```

```
 6
10

julia> findin(a,b) # 10 is the only common element
1-element Array{Int64,1}:
 4
```

source

`Base.unique` – Function.

```
unique(itr)
```

Returns an array containing one value from `itr` for each unique value, as determined by `isequal`.

```
julia> unique([1; 2; 2; 6])
3-element Array{Int64,1}:
 1
 2
 6
```

source

```
unique(f, itr)
```

Returns an array containing one value from `itr` for each unique value produced by `f` applied to elements of `itr`.

```
julia> unique(isodd, [1; 2; 2; 6])
2-element Array{Int64,1}:
 1
 2
```

source

```
unique(itr[, dim])
```

Returns an array containing only the unique elements of the iterable `itr`, in the order that the first of each set of equivalent elements originally appears. If `dim` is specified, returns unique regions of the array `itr` along `dim`.

```
julia> A = map(isodd, reshape(collect(1:8), (2,2,2)))
2×2×2 Array{Bool,3}:
[:, :, 1] =
  true   true
 false  false

[:, :, 2] =
  true   true
 false  false

julia> unique(A)
2-element Array{Bool,1}:
  true
 false

julia> unique(A, 2)
2×1×2 Array{Bool,3}:
```

```
[:, :, 1] =
  true
 false

[:, :, 2] =
  true
 false

julia> unique(A, 3)
2×2×1 Array{Bool,3}:
[:, :, 1] =
  true   true
 false  false
```

source

Base.allunique – Function.

```
allunique(itr) -> Bool
```

Return `true` if all values from `itr` are distinct when compared with `isequal`.

```
julia> a = [1; 2; 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> allunique([a, a])
false
```

source

Base.reduce – Method.

```
reduce(op, v0, itr)
```

Reduce the given collection `itr` with the given binary operator `op`. `v0` must be a neutral element for `op` that will be returned for empty collections. It is unspecified whether `v0` is used for non-empty collections.

Reductions for certain commonly-used operators have special implementations which should be used instead: `maximum(itr)`, `minimum(itr)`, `sum(itr)`, `prod(itr)`, `any(itr)`, `all(itr)`.

The associativity of the reduction is implementation dependent. This means that you can't use non-associative operations like `-` because it is undefined whether `reduce(-,[1,2,3])` should be evaluated as `(1-2)-3` or `1-(2-3)`. Use `foldl` or `foldr` instead for guaranteed left or right associativity.

Some operations accumulate error, and parallelism will also be easier if the reduction can be executed in groups. Future versions of Julia might change the algorithm. Note that the elements are not reordered if you use an ordered collection.

Examples

```
julia> reduce(*, 1, [2; 3; 4])
24
```

source

Base.reduce – Method.

```
reduce(op, itr)
```

Like `reduce(op, v0, itr)`. This cannot be used with empty collections, except for some special cases (e.g. when op is one of `+`, `*`, `max`, `min`, `&`, `|`) when Julia can determine the neutral element of op.

```
julia> reduce(*, [2; 3; 4])
24
```

source

Base.foldl – Method.

```
foldl(op, v0, itr)
```

Like `reduce`, but with guaranteed left associativity. `v0` will be used exactly once.

```
julia> foldl(-, 1, 2:5)
-13
```

source

Base.foldl – Method.

```
foldl(op, itr)
```

Like `foldl(op, v0, itr)`, but using the first element of `itr` as `v0`. In general, this cannot be used with empty collections (see `reduce(op, itr)`).

```
julia> foldl(-, 2:5)
-10
```

source

Base.foldr – Method.

```
foldr(op, v0, itr)
```

Like `reduce`, but with guaranteed right associativity. `v0` will be used exactly once.

```
julia> foldr(-, 1, 2:5)
-1
```

source

Base.foldr – Method.

```
foldr(op, itr)
```

Like `foldr(op, v0, itr)`, but using the last element of `itr` as `v0`. In general, this cannot be used with empty collections (see `reduce(op, itr)`).

```
julia> foldr(-, 2:5)
-2
```

source

Base.maximum – Method.

```
maximum(itr)
```

Returns the largest element in a collection.

```
julia> maximum(-20.5:10)
9.5

julia> maximum([1,2,3])
3
```

source

Base.maximum – Method.

```
maximum(A, dims)
```

Compute the maximum value of an array over the given dimensions. See also the `max(a,b)` function to take the maximum of two or more arguments, which can be applied elementwise to arrays via `max.(a,b)`.

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> maximum(A, 1)
1×2 Array{Int64,2}:
 3  4

julia> maximum(A, 2)
2×1 Array{Int64,2}:
 2
 4
```

source

Base.maximum! – Function.

```
maximum!(r, A)
```

Compute the maximum value of `A` over the singleton dimensions of `r`, and write results to `r`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> maximum!([1; 1], A)
2-element Array{Int64,1}:
 2
 4
```

```
julia> maximum!([1 1], A)
1×2 Array{Int64,2}:
 3  4
```

source

Base.minimum – Method.

```
minimum(itr)
```

Returns the smallest element in a collection.

```
julia> minimum(-20.5:10)
-20.5

julia> minimum([1,2,3])
1
```

source

Base.minimum – Method.

```
minimum(A, dims)
```

Compute the minimum value of an array over the given dimensions. See also the `min(a,b)` function to take the minimum of two or more arguments, which can be applied elementwise to arrays via `min.(a,b)`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> minimum(A, 1)
1×2 Array{Int64,2}:
 1  2

julia> minimum(A, 2)
2×1 Array{Int64,2}:
 1
 3
```

source

Base.minimum! – Function.

```
minimum!(r, A)
```

Compute the minimum value of `A` over the singleton dimensions of `r`, and write results to `r`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4
```

```
julia> minimum!([1; 1], A)
2-element Array{Int64,1}:
 1
 3

julia> minimum!([1 1], A)
1×2 Array{Int64,2}:
 1  2
```

source

`Base.extrema` – Method.

```
extrema(itr) -> Tuple
```

Compute both the minimum and maximum element in a single pass, and return them as a 2-tuple.

```
julia> extrema(2:10)
(2, 10)

julia> extrema([9,pi,4.5])
(3.141592653589793, 9.0)
```

source

`Base.extrema` – Method.

```
extrema(A, dims) -> Array{Tuple}
```

Compute the minimum and maximum elements of an array over the given dimensions.

Example

```
julia> A = reshape(collect(1:2:16), (2,2,2))
2×2×2 Array{Int64,3}:
[:, :, 1] =
 1  5
 3  7

[:, :, 2] =
  9  13
 11  15

julia> extrema(A, (1,2))
1×1×2 Array{Tuple{Int64,Int64},3}:
[:, :, 1] =
 (1, 7)

[:, :, 2] =
 (9, 15)
```

source

`Base.indmax` – Function.

```
indmax(itr) -> Integer
```

Returns the index of the maximum element in a collection. If there are multiple maximal elements, then the first one will be returned. NaN values are ignored, unless all elements are NaN.

The collection must not be empty.

Examples

```
julia> indmax([8,0.1,-9,pi])
1

julia> indmax([1,7,7,6])
2

julia> indmax([1,7,7,NaN])
2
```

source

`Base.indmin` – Function.

```
indmin(itr) -> Integer
```

Returns the index of the minimum element in a collection. If there are multiple minimal elements, then the first one will be returned. NaN values are ignored, unless all elements are NaN.

The collection must not be empty.

Examples

```
julia> indmin([8,0.1,-9,pi])
3

julia> indmin([7,1,1,6])
2

julia> indmin([7,1,1,NaN])
2
```

source

`Base.findmax` – Method.

```
findmax(itr) -> (x, index)
```

Returns the maximum element of the collection `itr` and its index. If there are multiple maximal elements, then the first one will be returned. NaN values are ignored, unless all elements are NaN.

The collection must not be empty.

Examples

```
julia> findmax([8,0.1,-9,pi])
(8.0, 1)

julia> findmax([1,7,7,6])
(7, 2)

julia> findmax([1,7,7,NaN])
(7.0, 2)
```

source

Base.findmax – Method.

```
findmax(A, region) -> (maxval, index)
```

For an array input, returns the value and index of the maximum over the given region.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> findmax(A,1)
([3 4], [2 4])

julia> findmax(A,2)
([2; 4], [3; 4])
```

source

Base.findmin – Method.

```
findmin(itr) -> (x, index)
```

Returns the minimum element of the collection `itr` and its index. If there are multiple minimal elements, then the first one will be returned. `NaN` values are ignored, unless all elements are `NaN`.

The collection must not be empty.

Examples

```
julia> findmin([8,0.1,-9,pi])
(-9.0, 3)

julia> findmin([7,1,1,6])
(1, 2)

julia> findmin([7,1,1,NaN])
(1.0, 2)
```

source

Base.findmin – Method.

```
findmin(A, region) -> (minval, index)
```

For an array input, returns the value and index of the minimum over the given region.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4
```

```
julia> findmin(A, 1)
([1 2], [1 3])

julia> findmin(A, 2)
([1; 3], [1; 2])
```

source

Base.findmax! – Function.

```
findmax!(rval, rind, A, [init=true]) -> (maxval, index)
```

Find the maximum of A and the corresponding linear index along singleton dimensions of rval and rind, and store the results in rval and rind.

source

Base.findmin! – Function.

```
findmin!(rval, rind, A, [init=true]) -> (minval, index)
```

Find the minimum of A and the corresponding linear index along singleton dimensions of rval and rind, and store the results in rval and rind.

source

Base.sum – Function.

```
sum(f, itr)
```

Sum the results of calling function f on each element of itr.

```
julia> sum(abs2, [2; 3; 4])
29
```

source

```
sum(itr)
```

Returns the sum of all elements in a collection.

```
julia> sum(1:20)
210
```

source

```
sum(A, dims)
```

Sum elements of an array over the given dimensions.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> sum(A, 1)
```

```
1×2 Array{Int64,2}:
 4  6

julia> sum(A, 2)
2×1 Array{Int64,2}:
 3
 7
```

source

`Base.sum!` – Function.

```
sum!(r, A)
```

Sum elements of `A` over the singleton dimensions of `r`, and write results to `r`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> sum!([1; 1], A)
2-element Array{Int64,1}:
 3
 7

julia> sum!([1 1], A)
1×2 Array{Int64,2}:
 4  6
```

source

`Base.prod` – Function.

```
prod(f, itr)
```

Returns the product of `f` applied to each element of `itr`.

```
julia> prod(abs2, [2; 3; 4])
576
```

source

```
prod(itr)
```

Returns the product of all elements of a collection.

```
julia> prod(1:20)
2432902008176640000
```

source

```
prod(A, dims)
```

Multiply elements of an array over the given dimensions.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> prod(A, 1)
1×2 Array{Int64,2}:
 3  8

julia> prod(A, 2)
2×1 Array{Int64,2}:
  2
 12
```

source

Base.prod! – Function.

```
prod!(r, A)
```

Multiply elements of A over the singleton dimensions of r, and write results to r.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> prod!([1; 1], A)
2-element Array{Int64,1}:
  2
 12

julia> prod!([1 1], A)
1×2 Array{Int64,2}:
 3  8
```

source

Base.any – Method.

```
any(itr) -> Bool
```

Test whether any elements of a boolean collection are `true`, returning `true` as soon as the first `true` value in `itr` is encountered (short-circuiting).

```
julia> a = [true,false,false,true]
4-element Array{Bool,1}:
  true
 false
 false
  true

julia> any(a)
true
```

```
julia> any((println(i); v) for (i, v) in enumerate(a))
1
true
```

source

Base.any – Method.

```
any(A, dims)
```

Test whether any values along the given dimensions of an array are `true`.

Examples

```
julia> A = [true false; true false]
2×2 Array{Bool,2}:
 true  false
 true  false

julia> any(A, 1)
1×2 Array{Bool,2}:
 true  false

julia> any(A, 2)
2×1 Array{Bool,2}:
 true
 true
```

source

Base.any! – Function.

```
any!(r, A)
```

Test whether any values in `A` along the singleton dimensions of `r` are `true`, and write results to `r`.

Examples

```
julia> A = [true false; true false]
2×2 Array{Bool,2}:
 true  false
 true  false

julia> any!([1; 1], A)
2-element Array{Int64,1}:
 1
 1

julia> any!([1 1], A)
1×2 Array{Int64,2}:
 1  0
```

source

Base.all – Method.

```
all(itr) -> Bool
```

Test whether all elements of a boolean collection are `true`, returning `false` as soon as the first `false` value in `itr` is encountered (short-circuiting).

```
julia> a = [true,false,false,true]
4-element Array{Bool,1}:
  true
 false
 false
  true

julia> all(a)
false

julia> all((println(i); v) for (i, v) in enumerate(a))
1
2
false
```

source

`Base.all` – Method.

```
all(A, dims)
```

Test whether all values along the given dimensions of an array are `true`.

Examples

```
julia> A = [true false; true true]
2×2 Array{Bool,2}:
 true  false
 true   true

julia> all(A, 1)
1×2 Array{Bool,2}:
 true  false

julia> all(A, 2)
2×1 Array{Bool,2}:
 false
  true
```

source

`Base.all!` – Function.

```
all!(r, A)
```

Test whether all values in `A` along the singleton dimensions of `r` are `true`, and write results to `r`.

Examples

```
julia> A = [true false; true false]
2×2 Array{Bool,2}:
 true  false
 true  false
```

```
julia> all!([1; 1], A)
2-element Array{Int64,1}:
 0
 0

julia> all!([1 1], A)
1×2 Array{Int64,2}:
 1  0
```

source

`Base.count` – Function.

```
count(p, itr) -> Integer
count(itr) -> Integer
```

Count the number of elements in `itr` for which predicate p returns `true`. If p is omitted, counts the number of `true` elements in `itr` (which should be a collection of boolean values).

```
julia> count(i->(4<=i<=6), [2,3,4,5,6])
3

julia> count([true, false, true, true])
3
```

source

`Base.any` – Method.

```
any(p, itr) -> Bool
```

Determine whether predicate p returns `true` for any elements of `itr`, returning `true` as soon as the first item in `itr` for which p returns `true` is encountered (short-circuiting).

```
julia> any(i->(4<=i<=6), [3,5,7])
true

julia> any(i -> (println(i); i > 3), 1:10)
1
2
3
4
true
```

source

`Base.all` – Method.

```
all(p, itr) -> Bool
```

Determine whether predicate p returns `true` for all elements of `itr`, returning `false` as soon as the first item in `itr` for which p returns `false` is encountered (short-circuiting).

```
julia> all(i->(4<=i<=6), [4,5,6])
true
```

```
julia> all(i -> (println(i); i < 3), 1:10)
1
2
3
false
```

source

Base.foreach – Function.

```
foreach(f, c...) -> Void
```

Call function f on each element of iterable c. For multiple iterable arguments, f is called elementwise. foreach should be used instead of map when the results of f are not needed, for example in foreach(println, array).

Example

```
julia> a = 1:3:7;

julia> foreach(x -> println(x^2), a)
1
16
49
```

source

Base.map – Function.

```
map(f, c...) -> collection
```

Transform collection c by applying f to each element. For multiple collection arguments, apply f elementwise.

Examples

```
julia> map(x -> x * 2, [1, 2, 3])
3-element Array{Int64,1}:
 2
 4
 6

julia> map(+, [1, 2, 3], [10, 20, 30])
3-element Array{Int64,1}:
 11
 22
 33
```

source

```
map(f, x::Nullable)
```

Return f applied to the value of x if it has one, as a Nullable. If x is null, then return a null value of type Nullable{S}. S is guaranteed to be either Union{} or a concrete type. Whichever of these is chosen is an implementation detail, but typically the choice that maximizes performance would be used. If x has a value, then the return type is guaranteed to be of type Nullable{typeof(f(x))}.

source

Base.map! – Function.

```
map!(function, destination, collection...)
```

Like map, but stores the result in destination rather than a new collection. destination must be at least as large as the first collection.

Example

```
julia> x = zeros(3);

julia> map!(x -> x * 2, x, [1, 2, 3]);

julia> x
3-element Array{Float64,1}:
 2.0
 4.0
 6.0
```

source

Base.mapreduce – Method.

```
mapreduce(f, op, v0, itr)
```

Apply function f to each element in itr, and then reduce the result using the binary function op. v0 must be a neutral element for op that will be returned for empty collections. It is unspecified whether v0 is used for non-empty collections.

mapreduce is functionally equivalent to calling reduce(op, v0, map(f, itr)), but will in general execute faster since no intermediate collection needs to be created. See documentation for reduce and map.

```
julia> mapreduce(x->x^2, +, [1:3;]) # == 1 + 4 + 9
14
```

The associativity of the reduction is implementation-dependent. Additionally, some implementations may reuse the return value of f for elements that appear multiple times in itr. Use mapfoldl or mapfoldr instead for guaranteed left or right associativity and invocation of f for every value.

source

Base.mapreduce – Method.

```
mapreduce(f, op, itr)
```

Like mapreduce(f, op, v0, itr). In general, this cannot be used with empty collections (see reduce(op, itr)).

source

Base.mapfoldl – Method.

```
mapfoldl(f, op, v0, itr)
```

Like mapreduce, but with guaranteed left associativity, as in foldl. v0 will be used exactly once.

source

Base.mapfoldl – Method.

```
mapfoldl(f, op, itr)
```

Like `mapfoldl(f, op, v0, itr)`, but using the first element of `itr` as `v0`. In general, this cannot be used with empty collections (see `reduce(op, itr)`).

source

`Base.mapfoldr` – Method.

```
mapfoldr(f, op, v0, itr)
```

Like `mapreduce`, but with guaranteed right associativity, as in `foldr`. `v0` will be used exactly once.

source

`Base.mapfoldr` – Method.

```
mapfoldr(f, op, itr)
```

Like `mapfoldr(f, op, v0, itr)`, but using the first element of `itr` as `v0`. In general, this cannot be used with empty collections (see `reduce(op, itr)`).

source

`Base.first` – Function.

```
first(coll)
```

Get the first element of an iterable collection. Returns the start point of a `Range` even if it is empty.

```
julia> first(2:2:10)
2

julia> first([1; 2; 3; 4])
1
```

source

`Base.last` – Function.

```
last(coll)
```

Get the last element of an ordered collection, if it can be computed in O(1) time. This is accomplished by calling `endof` to get the last index. Returns the end point of a `Range` even if it is empty.

```
julia> last(1:2:10)
9

julia> last([1; 2; 3; 4])
4
```

source

`Base.step` – Function.

```
step(r)
```

Get the step size of a `Range` object.

```
julia> step(1:10)
1

julia> step(1:2:10)
2

julia> step(2.5:0.3:10.9)
0.3

julia> step(linspace(2.5,10.9,85))
0.1
```

source

Base.collect – Method.

```
collect(collection)
```

Return an `Array` of all items in a collection or iterator. For associative collections, returns `Pair{KeyType, ValType}`. If the argument is array-like or is an iterator with the `HasShape()` trait, the result will have the same shape and number of dimensions as the argument.

Example

```
julia> collect(1:2:13)
7-element Array{Int64,1}:
  1
  3
  5
  7
  9
 11
 13
```

source

Base.collect – Method.

```
collect(element_type, collection)
```

Return an `Array` with the given element type of all items in a collection or iterable. The result has the same shape and number of dimensions as `collection`.

```
julia> collect(Float64, 1:2:5)
3-element Array{Float64,1}:
 1.0
 3.0
 5.0
```

source

Base.issubset – Method.

```
issubset(a, b)
(a,b) -> Bool
(a,b) -> Bool
(a,b) -> Bool
```

Determine whether every element of a is also in b, using `in`.

Examples

```
julia> issubset([1, 2], [1, 2, 3])
true

julia> issubset([1, 2, 3], [1, 2])
false
```

source

`Base.filter` – Function.

```
filter(function, collection)
```

Return a copy of `collection`, removing elements for which `function` is `false`. For associative collections, the function is passed two arguments (key and value).

Examples

```
julia> a = 1:10
1:10

julia> filter(isodd, a)
5-element Array{Int64,1}:
 1
 3
 5
 7
 9

julia> d = Dict(1=>"a", 2=>"b")
Dict{Int64,String} with 2 entries:
  2 => "b"
  1 => "a"

julia> filter((x,y)->isodd(x), d)
Dict{Int64,String} with 1 entry:
  1 => "a"
```

source

```
filter(p, x::Nullable)
```

Return null if either x is null or `p(get(x))` is false, and x otherwise.

source

`Base.filter!` – Function.

```
filter!(function, collection)
```

Update `collection`, removing elements for which `function` is `false`. For associative collections, the function is passed two arguments (key and value).

Example

```
julia> filter!(isodd, collect(1:10))
5-element Array{Int64,1}:
 1
 3
 5
 7
 9
```

source

46.4 Colecciones Indexables

Base.getindex – Method.

```
getindex(collection, key...)
```

Retrieve the value(s) stored at the given key or index within a collection. The syntax `a[i,j,...]` is converted by the compiler to `getindex(a, i, j, ...)`.

Example

```
julia> A = Dict("a" => 1, "b" => 2)
Dict{String,Int64} with 2 entries:
  "b" => 2
  "a" => 1

julia> getindex(A, "a")
1
```

source

Base.setindex! – Method.

```
setindex!(collection, value, key...)
```

Store the given value at the given key or index within a collection. The syntax `a[i,j,...] = x` is converted by the compiler to `(setindex!(a, x, i, j, ...); x)`.

source

Completamente implementado por:

- Array
- BitArray
- AbstractArray
- SubArray
- ObjectIdDict
- Dict
- WeakKeyDict
- AbstractString

Parcialmente implementado por:

- `Range`
- `UnitRange`
- `Tuple`

46.5 Colecciones Asociativas

`Dict` es la colección asociativa estándar. Su implementación usa `hash()` como función de hashing para la clave, e `isequal()` para determinar la igualdad. Si redefine estas dos funciones en un tipo personalizado, sobreescribirán cómo se almacenan dichos tipos en una tabla hash.

`ObjectIdDict` es una tabla hash especial donde las claves son siempre identidades de objeto.

`WeakKeyDict` es una implementación de tabla hash donde las claves son referencias débiles a los objetos y, por lo tanto, permiten recolección de basura recogida incluso cuando se referencian en una tabla hash.

`Dict`s se pueden crear pasando pares de objetos construidos con `=>()` a un constructor `Dict`: `Dict ("A"=> 1," B "=> 2)`. Esta llamada intentará inferir información sobre el tipo de las claves y los valores (es decir, este ejemplo crea un `Dict{String, Int64}`). Para especificar los tipos explícitamente, use la sintaxis `Dict{KeyType,Value-Type}(...)`. Por ejemplo, `Dict{String,Int32}(" A "=> 1," B "=> 2)`.

Las colecciones asociativas también pueden ser creadas con generadores. Por ejemplo, `Dict(i => f(i) for i = 1:10)`.

Dado un diccionario `D`, la sintaxis `D[x]` devuelve el valor de la clave `x` (si existe) o arroja un error, y `D[x] = y` almacena el par de clave-valor `x => y` en `D` (reemplazando cualquier valor existente para la clave`x`). Múltiples argumentos para `D [...]` se convierten a tuplas; por ejemplo, la sintaxis `D[x,y]` es equivalente a `D[(x,y)]`, es decir, se refiere al valor introducido para la tupla `(x,y)`.

`Base.Dict` – Type.

```
Dict([itr])
```

`Dict{K,V}()` constructs a hash table with keys of type `K` and values of type `V`.

Given a single iterable argument, constructs a `Dict` whose key-value pairs are taken from 2-tuples `(key,value)` generated by the argument.

```
julia> Dict([("A", 1), ("B", 2)])
Dict{String,Int64} with 2 entries:
  "B" => 2
  "A" => 1
```

Alternatively, a sequence of pair arguments may be passed.

```
julia> Dict("A"=>1, "B"=>2)
Dict{String,Int64} with 2 entries:
  "B" => 2
  "A" => 1
```

source

`Base.ObjectIdDict` – Type.

```
ObjectIdDict([itr])
```

`ObjectIdDict()` constructs a hash table where the keys are (always) object identities. Unlike `Dict` it is not parameterized on its key and value type and thus its `eltype` is always `Pair{Any,Any}`.

See `Dict` for further help.

source

`Base.WeakKeyDict` – Type.

```
WeakKeyDict([itr])
```

`WeakKeyDict()` constructs a hash table where the keys are weak references to objects, and thus may be garbage collected even when referenced in a hash table.

See `Dict` for further help.

source

`Base.haskey` – Function.

```
haskey(collection, key) -> Bool
```

Determine whether a collection has a mapping for a given key.

```
julia> a = Dict('a'=>2, 'b'=>3)
Dict{Char,Int64} with 2 entries:
  'b' => 3
  'a' => 2

julia> haskey(a,'a')
true

julia> haskey(a,'c')
false
```

source

`Base.get` – Method.

```
get(collection, key, default)
```

Return the value stored for the given key, or the given default value if no mapping for the key is present.

Examples

```
julia> d = Dict("a"=>1, "b"=>2);

julia> get(d, "a", 3)
1

julia> get(d, "c", 3)
3
```

source

`Base.get` – Function.

```
get(f::Function, collection, key)
```

Return the value stored for the given key, or if no mapping for the key is present, return `f()`. Use `get!` to also store the default value in the dictionary.

This is intended to be called using do block syntax

```
get(dict, key) do
    # default value calculated here
    time()
end
```

source

`Base.get!` – Method.

```
get!(collection, key, default)
```

Return the value stored for the given key, or if no mapping for the key is present, store `key => default`, and return `default`.

Examples

```
julia> d = Dict("a"=>1, "b"=>2, "c"=>3);

julia> get!(d, "a", 5)
1

julia> get!(d, "d", 4)
4

julia> d
Dict{String,Int64} with 4 entries:
  "c" => 3
  "b" => 2
  "a" => 1
  "d" => 4
```

source

`Base.get!` – Method.

```
get!(f::Function, collection, key)
```

Return the value stored for the given key, or if no mapping for the key is present, store `key => f()`, and return `f()`.

This is intended to be called using do block syntax:

```
get!(dict, key) do
    # default value calculated here
    time()
end
```

source

`Base.getkey` – Function.

```
getkey(collection, key, default)
```

Return the key matching argument `key` if one exists in `collection`, otherwise return `default`.

```
julia> a = Dict('a'=>2, 'b'=>3)
Dict{Char,Int64} with 2 entries:
  'b' => 3
  'a' => 2

julia> getkey(a,'a',1)
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> getkey(a,'d','a')
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
```

source

`Base.delete!` – Function.

```
delete!(collection, key)
```

Delete the mapping for the given key in a collection, and return the collection.

Example

```
julia> d = Dict("a"=>1, "b"=>2)
Dict{String,Int64} with 2 entries:
  "b" => 2
  "a" => 1

julia> delete!(d, "b")
Dict{String,Int64} with 1 entry:
  "a" => 1
```

source

`Base.pop!` – Method.

```
pop!(collection, key[, default])
```

Delete and return the mapping for `key` if it exists in `collection`, otherwise return `default`, or throw an error if `default` is not specified.

Examples

```
julia> d = Dict("a"=>1, "b"=>2, "c"=>3);

julia> pop!(d, "a")
1

julia> pop!(d, "d")
ERROR: KeyError: key "d" not found
Stacktrace:
 [1] pop!(::Dict{String,Int64}, ::String) at ./dict.jl:539

julia> pop!(d, "e", 4)
4
```

source

Base.keys – Function.

```
keys(a::Associative)
```

Return an iterator over all keys in a collection. `collect(keys(a))` returns an array of keys. Since the keys are stored internally in a hash table, the order in which they are returned may vary. But `keys(a)` and `values(a)` both iterate a and return the elements in the same order.

```
julia> a = Dict('a'=>2, 'b'=>3)
Dict{Char,Int64} with 2 entries:
  'b' => 3
  'a' => 2

julia> collect(keys(a))
2-element Array{Char,1}:
 'b'
 'a'
```

source

Base.values – Function.

```
values(a::Associative)
```

Return an iterator over all values in a collection. `collect(values(a))` returns an array of values. Since the values are stored internally in a hash table, the order in which they are returned may vary. But `keys(a)` and `values(a)` both iterate a and return the elements in the same order.

```
julia> a = Dict('a'=>2, 'b'=>3)
Dict{Char,Int64} with 2 entries:
  'b' => 3
  'a' => 2

julia> collect(values(a))
2-element Array{Int64,1}:
 3
 2
```

source

Base.merge – Function.

```
merge(d::Associative, others::Associative...)
```

Construct a merged collection from the given collections. If necessary, the types of the resulting collection will be promoted to accommodate the types of the merged collections. If the same key is present in another collection, the value for that key will be the value it has in the last collection listed.

```
julia> a = Dict("foo" => 0.0, "bar" => 42.0)
Dict{String,Float64} with 2 entries:
  "bar" => 42.0
  "foo" => 0.0

julia> b = Dict("baz" => 17, "bar" => 4711)
```

```
Dict{String,Int64} with 2 entries:
  "bar" => 4711
  "baz" => 17

julia> merge(a, b)
Dict{String,Float64} with 3 entries:
  "bar" => 4711.0
  "baz" => 17.0
  "foo" => 0.0

julia> merge(b, a)
Dict{String,Float64} with 3 entries:
  "bar" => 42.0
  "baz" => 17.0
  "foo" => 0.0
```

source

```
merge(combine, d::Associative, others::Associative...)
```

Construct a merged collection from the given collections. If necessary, the types of the resulting collection will be promoted to accommodate the types of the merged collections. Values with the same key will be combined using the combiner function.

```
julia> a = Dict("foo" => 0.0, "bar" => 42.0)
Dict{String,Float64} with 2 entries:
  "bar" => 42.0
  "foo" => 0.0

julia> b = Dict("baz" => 17, "bar" => 4711)
Dict{String,Int64} with 2 entries:
  "bar" => 4711
  "baz" => 17

julia> merge(+, a, b)
Dict{String,Float64} with 3 entries:
  "bar" => 4753.0
  "baz" => 17.0
  "foo" => 0.0
```

source

`Base.merge!` – Function.

```
merge!(d::Associative, others::Associative...)
```

Update collection with pairs from the other collections. See also `merge`.

```
julia> d1 = Dict(1 => 2, 3 => 4);

julia> d2 = Dict(1 => 4, 4 => 5);

julia> merge!(d1, d2);

julia> d1
Dict{Int64,Int64} with 3 entries:
```

```
  4 => 5
  3 => 4
  1 => 4
```

source

```
merge!(combine, d::Associative, others::Associative...)
```

Update collection with pairs from the other collections. Values with the same key will be combined using the combiner function.

```
julia> d1 = Dict(1 => 2, 3 => 4);

julia> d2 = Dict(1 => 4, 4 => 5);

julia> merge!(+, d1, d2);

julia> d1
Dict{Int64,Int64} with 3 entries:
  4 => 5
  3 => 4
  1 => 6

julia> merge!(-, d1, d1);

julia> d1
Dict{Int64,Int64} with 3 entries:
  4 => 0
  3 => 0
  1 => 0
```

source

Merge changes into current head

source

Internal implementation of merge. Returns `true` if merge was successful, otherwise `false`

source

```
merge!(repo::GitRepo; kwargs...) -> Bool
```

Perform a git merge on the repository `repo`, merging commits with diverging history into the current branch. Returns `true` if the merge succeeded, `false` if not.

The keyword arguments are:

- `committish::AbstractString=""`: Merge the named commit(s) in `committish`.
- `branch::AbstractString=""`: Merge the branch `branch` and all its commits since it diverged from the current branch.
- `fastforward::Bool=false`: If `fastforward` is `true`, only merge if the merge is a fast-forward (the current branch head is an ancestor of the commits to be merged), otherwise refuse to merge and return `false`. This is equivalent to the git CLI option `--ff-only`.
- `merge_opts::MergeOptions=MergeOptions()`: `merge_opts` specifies options for the merge, such as merge strategy in case of conflicts.

- `checkout_opts::CheckoutOptions=CheckoutOptions()`: `checkout_opts` specifies options for the checkout step.

Equivalent to `git merge [--ff-only] [<committish> | <branch>]`.

> **Note**
> If you specify a `branch`, this must be done in reference format, since the string will be turned into a `GitReference`. For example, if you wanted to merge branch `branch_a`, you would call `merge!(repo, branch="refs/heads/branch_a")`.

source

`Base.sizehint!` – Function.

```
sizehint!(s, n)
```

Suggest that collection `s` reserve capacity for at least `n` elements. This can improve performance.

source

`Base.keytype` – Function.

```
keytype(type)
```

Get the key type of an associative collection type. Behaves similarly to `eltype`.

```
julia> keytype(Dict(Int32(1) => "foo"))
Int32
```

source

`Base.valtype` – Function.

```
valtype(type)
```

Get the value type of an associative collection type. Behaves similarly to `eltype`.

```
julia> valtype(Dict(Int32(1) => "foo"))
String
```

source

Completamente implementado por:

- `ObjectIdDict`
- `Dict`
- `WeakKeyDict`

Parcialmente implementado por:

- `IntSet`
- `Set`
- `EnvHash`
- `Array`
- `BitArray`

46.6 Colecciones de tipo Conjunto

`Base.Set` – Type.

```
Set([itr])
```

Construct a `Set` of the values generated by the given iterable object, or an empty set. Should be used instead of `IntSet` for sparse integer sets, or for sets of arbitrary objects.

source

`Base.IntSet` – Type.

```
IntSet([itr])
```

Construct a sorted set of positive `Int`s generated by the given iterable object, or an empty set. Implemented as a bit string, and therefore designed for dense integer sets. Only `Int`s greater than 0 can be stored. If the set will be sparse (for example holding a few very large integers), use `Set` instead.

source

`Base.union` – Function.

```
union(s1,s2...)
(s1,s2...)
```

Construct the union of two or more sets. Maintains order with arrays.

Examples

```
julia> union([1, 2], [3, 4])
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> union([1, 2], [2, 4])
3-element Array{Int64,1}:
 1
 2
 4

julia> union([4, 2], [1, 2])
3-element Array{Int64,1}:
 4
 2
 1
```

source

`Base.union!` – Function.

```
union!(s, iterable)
```

Union each element of `iterable` into set `s` in-place.

source

`Base.intersect` – Function.

```
intersect(s1,s2...)
(s1,s2)
```

Construct the intersection of two or more sets. Maintains order and multiplicity of the first argument for arrays and ranges.

source

`Base.setdiff` – Function.

```
setdiff(a, b)
```

Construct the set of elements in a but not b. Maintains order with arrays. Note that both arguments must be collections, and both will be iterated over. In particular, `setdiff(set,element)` where `element` is a potential member of `set`, will not work in general.

Example

```
julia> setdiff([1,2,3],[3,4,5])
2-element Array{Int64,1}:
 1
 2
```

source

`Base.setdiff!` – Function.

```
setdiff!(s, iterable)
```

Remove each element of `iterable` from set s in-place.

source

`Base.symdiff` – Function.

```
symdiff(a, b, rest...)
```

Construct the symmetric difference of elements in the passed in sets or arrays. Maintains order with arrays.

Example

```
julia> symdiff([1,2,3],[3,4,5],[4,5,6])
3-element Array{Int64,1}:
 1
 2
 6
```

source

`Base.symdiff!` – Method.

```
symdiff!(s, n)
```

The set s is destructively modified to toggle the inclusion of integer n.

source

`Base.symdiff!` – Method.

```
symdiff!(s, itr)
```

For each element in `itr`, destructively toggle its inclusion in set `s`.

source

`Base.symdiff!` – Method.

```
symdiff!(s, itr)
```

For each element in `itr`, destructively toggle its inclusion in set `s`.

source

`Base.intersect!` – Function.

```
intersect!(s1::IntSet, s2::IntSet)
```

Intersects sets s1 and s2 and overwrites the set s1 with the result. If needed, s1 will be expanded to the size of s2.

source

`Base.issubset` – Function.

```
issubset(A, S) -> Bool
(A,S) -> Bool
```

Return `true` if `A` is a subset of or equal to `S`.

source

Completamente implementado por:

- `IntSet`
- `Set`

Parcialmente implementado por:

- `Array`

46.7 Acciones relacionadas con Colas

`Base.push!` – Function.

```
push!(collection, items...) -> collection
```

Insert one or more `items` at the end of `collection`.

Example

```
julia> push!([1, 2, 3], 4, 5, 6)
6-element Array{Int64,1}:
 1
 2
 3
 4
 5
 6
```

Use `append!` to add all the elements of another collection to `collection`. The result of the preceding example is equivalent to `append!([1, 2, 3], [4, 5, 6])`.

source

`Base.pop!` – Method.

```
pop!(collection) -> item
```

Remove the last item in `collection` and return it.

Examples

```
julia> A=[1, 2, 3, 4, 5, 6]
6-element Array{Int64,1}:
 1
 2
 3
 4
 5
 6

julia> pop!(A)
6

julia> A
5-element Array{Int64,1}:
 1
 2
 3
 4
 5
```

source

`Base.unshift!` – Function.

```
unshift!(collection, items...) -> collection
```

Insert one or more `items` at the beginning of `collection`.

Example

```
julia> unshift!([1, 2, 3, 4], 5, 6)
6-element Array{Int64,1}:
 5
 6
 1
```

```
2
3
4
```

source

Base.shift! – Function.

```
shift!(collection) -> item
```

Remove the first `item` from `collection`.

Example

```
julia> A = [1, 2, 3, 4, 5, 6]
6-element Array{Int64,1}:
 1
 2
 3
 4
 5
 6

julia> shift!(A)
1

julia> A
5-element Array{Int64,1}:
 2
 3
 4
 5
 6
```

source

Base.insert! – Function.

```
insert!(a::Vector, index::Integer, item)
```

Insert an `item` into `a` at the given `index`. `index` is the index of `item` in the resulting `a`.

Example

```
julia> insert!([6, 5, 4, 2, 1], 4, 3)
6-element Array{Int64,1}:
 6
 5
 4
 3
 2
 1
```

source

Base.deleteat! – Function.

```
deleteat!(a::Vector, i::Integer)
```

Remove the item at the given `i` and return the modified `a`. Subsequent items are shifted to fill the resulting gap.

Example

```
julia> deleteat!([6, 5, 4, 3, 2, 1], 2)
5-element Array{Int64,1}:
 6
 4
 3
 2
 1
```

source

```
deleteat!(a::Vector, inds)
```

Remove the items at the indices given by `inds`, and return the modified `a`. Subsequent items are shifted to fill the resulting gap.

`inds` can be either an iterator or a collection of sorted and unique integer indices, or a boolean vector of the same length as `a` with `true` indicating entries to delete.

Examples

```
julia> deleteat!([6, 5, 4, 3, 2, 1], 1:2:5)
3-element Array{Int64,1}:
 5
 3
 1

julia> deleteat!([6, 5, 4, 3, 2, 1], [true, false, true, false, true, false])
3-element Array{Int64,1}:
 5
 3
 1

julia> deleteat!([6, 5, 4, 3, 2, 1], (2, 2))
ERROR: ArgumentError: indices must be unique and sorted
Stacktrace:
 [1] _deleteat!(::Array{Int64,1}, ::Tuple{Int64,Int64}) at ./array.jl:926
 [2] deleteat!(::Array{Int64,1}, ::Tuple{Int64,Int64}) at ./array.jl:913
```

source

`Base.splice!` – Function.

```
splice!(a::Vector, index::Integer, [replacement]) -> item
```

Remove the item at the given index, and return the removed item. Subsequent items are shifted left to fill the resulting gap. If specified, replacement values from an ordered collection will be spliced in place of the removed item.

Examples

```
julia> A = [6, 5, 4, 3, 2, 1]; splice!(A, 5)
2

julia> A
5-element Array{Int64,1}:
 6
 5
 4
 3
 1

julia> splice!(A, 5, -1)
1

julia> A
5-element Array{Int64,1}:
  6
  5
  4
  3
 -1

julia> splice!(A, 1, [-1, -2, -3])
6

julia> A
7-element Array{Int64,1}:
 -1
 -2
 -3
  5
  4
  3
 -1
```

To insert `replacement` before an index n without removing any items, use `splice!(collection, n:n-1, replacement)`.

source

```
splice!(a::Vector, range, [replacement]) -> items
```

Remove items in the specified index range, and return a collection containing the removed items. Subsequent items are shifted left to fill the resulting gap. If specified, replacement values from an ordered collection will be spliced in place of the removed items.

To insert `replacement` before an index n without removing any items, use `splice!(collection, n:n-1, replacement)`.

Example

```
julia> splice!(A, 4:3, 2)
0-element Array{Int64,1}

julia> A
8-element Array{Int64,1}:
 -1
 -2
```

```
-3
 2
 5
 4
 3
-1
```

source

`Base.resize!` – Function.

```
resize!(a::Vector, n::Integer) -> Vector
```

Resize a to contain n elements. If n is smaller than the current collection length, the first n elements will be retained. If n is larger, the new elements are not guaranteed to be initialized.

Examples

```
julia> resize!([6, 5, 4, 3, 2, 1], 3)
3-element Array{Int64,1}:
 6
 5
 4

julia> a = resize!([6, 5, 4, 3, 2, 1], 8);

julia> length(a)
8

julia> a[1:6]
6-element Array{Int64,1}:
 6
 5
 4
 3
 2
 1
```

source

`Base.append!` – Function.

```
append!(collection, collection2) -> collection.
```

Add the elements of `collection2` to the end of `collection`.

Examples

```
julia> append!([1],[2,3])
3-element Array{Int64,1}:
 1
 2
 3

julia> append!([1, 2, 3], [4, 5, 6])
6-element Array{Int64,1}:
 1
```

```
2
3
4
5
6
```

Use `push!` to add individual items to `collection` which are not already themselves in another collection. The result is of the preceding example is equivalent to `push!([1, 2, 3], 4, 5, 6)`.

source

`Base.prepend!` – Function.

```
prepend!(a::Vector, items) -> collection
```

Insert the elements of `items` to the beginning of a.

Example

```
julia> prepend!([3],[1,2])
3-element Array{Int64,1}:
 1
 2
 3
```

source

Completamente implementado por:

- `Vector` (también conocido como 1-dimensional `Array`)
- `BitVector` (también conocido como 1-dimensional `BitArray`)

Chapter 47

Matemáticas

47.1 Operadores Matemáticos

`Base.:-` – Method.

```
-(x)
```

Unary minus operator.

source

`Base.:+` – Function.

```
+(x, y...)
```

Addition operator. `x+y+z+...` calls this function with all arguments, i.e. `+(x, y, z, ...)`.

source

`Base.:-` – Method.

```
-(x, y)
```

Subtraction operator.

source

`Base.:*` – Method.

```
*(x, y...)
```

Multiplication operator. `x*y*z*...` calls this function with all arguments, i.e. `*(x, y, z, ...)`.

source

`Base.:/` – Function.

```
/(x, y)
```

Right division operator: multiplication of x by the inverse of y on the right. Gives floating-point results for integer arguments.

source

`Base.:\` – Method.

```
\(x, y)
```

Left division operator: multiplication of y by the inverse of x on the left. Gives floating-point results for integer arguments.

```
julia> 3 \ 6
2.0

julia> inv(3) * 6
2.0

julia> A = [1 2; 3 4]; x = [5, 6];

julia> A \ x
2-element Array{Float64,1}:
 -4.0
  4.5

julia> inv(A) * x
2-element Array{Float64,1}:
 -4.0
  4.5
```

source

`Base.:^` – Method.

```
^(x, y)
```

Exponentiation operator. If x is a matrix, computes matrix exponentiation.

If y is an `Int` literal (e.g. 2 in `x^2` or -3 in `x^-3`), the Julia code `x^y` is transformed by the compiler to `Base.literal_pow(^, x, Val{y})`, to enable compile-time specialization on the value of the exponent. (As a default fallback we have `Base.literal_pow(^, x, Val{y}) = ^(x,y)`, where usually `^ == Base.^` unless `^` has been defined in the calling namespace.)

```
julia> 3^5
243

julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> A^3
2×2 Array{Int64,2}:
 37   54
 81  118
```

source

`Base.fma` – Function.

```
fma(x, y, z)
```

Computes x*y+z without rounding the intermediate result x*y. On some systems this is significantly more expensive than x*y+z. fma is used to improve accuracy in certain algorithms. See muladd.

source

Base.muladd – Function.

```
muladd(x, y, z)
```

Combined multiply-add, computes x*y+z in an efficient manner. This may on some systems be equivalent to x*y+z, or to fma(x,y,z). muladd is used to improve performance. See fma.

Example

```
julia> muladd(3, 2, 1)
7

julia> 3 * 2 + 1
7
```

source

Base.div – Function.

```
div(x, y)
÷(x, y)
```

The quotient from Euclidean division. Computes x/y, truncated to an integer.

```
julia> 9 ÷ 4
2

julia> -5 ÷ 3
-1
```

source

Base.fld – Function.

```
fld(x, y)
```

Largest integer less than or equal to x/y.

```
julia> fld(7.3,5.5)
1.0
```

source

Base.cld – Function.

```
cld(x, y)
```

Smallest integer larger than or equal to x/y.

```
julia> cld(5.5,2.2)
3.0
```

source

Base.mod – Function.

```
mod(x, y)
rem(x, y, RoundDown)
```

The reduction of x modulo y, or equivalently, the remainder of x after floored division by y, i.e.

```
x - y*fld(x,y)
```

if computed without intermediate rounding.

The result will have the same sign as y, and magnitude less than `abs(y)` (with some exceptions, see note below).

Note

When used with floating point values, the exact result may not be representable by the type, and so rounding error may occur. In particular, if the exact result is very close to y, then it may be rounded to y.

```
julia> mod(8, 3)
2

julia> mod(9, 3)
0

julia> mod(8.9, 3)
2.9000000000000004

julia> mod(eps(), 3)
2.220446049250313e-16

julia> mod(-eps(), 3)
3.0
```

source

```
rem(x::Integer, T::Type{<:Integer}) -> T
mod(x::Integer, T::Type{<:Integer}) -> T
%(x::Integer, T::Type{<:Integer}) -> T
```

Find `y::T` such that x ≡ y (mod n), where n is the number of integers representable in T, and y is an integer in `[typemin(T),typemax(T)]`. If T can represent any integer (e.g. `T == BigInt`), then this operation corresponds to a conversion to T.

```
julia> 129 % Int8
-127
```

source

Base.rem – Function.

```
rem(x, y)
%(x, y)
```

Remainder from Euclidean division, returning a value of the same sign as x, and smaller in magnitude than y. This value is always exact.

```
julia> x = 15; y = 4;

julia> x % y
3

julia> x == div(x, y) * y + rem(x, y)
true
```

source

Base.Math.rem2pi – Function.

```
rem2pi(x, r::RoundingMode)
```

Compute the remainder of x after integer division by 2π, with the quotient rounded according to the rounding mode r. In other words, the quantity

```
x - π2*round(xπ/(2),r)
```

without any intermediate rounding. This internally uses a high precision approximation of 2π, and so will give a more accurate result than rem(x,2π,r)

- if r == RoundNearest, then the result is in the interval $[-,]$. This will generally be the most accurate result.
- if r == RoundToZero, then the result is in the interval $[0, 2]$ if x is positive,. or $[-2, 0]$ otherwise.
- if r == RoundDown, then the result is in the interval $[0, 2]$.
- if r == RoundUp, then the result is in the interval $[-2, 0]$.

Example

```
julia> rem2pi(7pi/4, RoundNearest)
-0.7853981633974485

julia> rem2pi(7pi/4, RoundDown)
5.497787143782138
```

source

Base.Math.mod2pi – Function.

```
mod2pi(x)
```

Modulus after division by 2π, returning in the range $[0, 2)$.

This function computes a floating point representation of the modulus after division by numerically exact 2π, and is therefore not exactly the same as mod(x,2π), which would compute the modulus of x relative to division by the floating-point number 2π.

Example

```
julia> mod2pi(9*pi/4)
0.7853981633974481
```

source

Base.divrem – Function.

```
divrem(x, y)
```

The quotient and remainder from Euclidean division. Equivalent to `(div(x,y), rem(x,y))` or `(x÷y, x%y)`.

```
julia> divrem(3,7)
(0, 3)

julia> divrem(7,3)
(2, 1)
```

source

Base.fldmod – Function.

```
fldmod(x, y)
```

The floored quotient and modulus after division. Equivalent to `(fld(x,y), mod(x,y))`.

source

Base.fld1 – Function.

```
fld1(x, y)
```

Flooring division, returning a value consistent with `mod1(x,y)`

See also: mod1.

```
julia> x = 15; y = 4;

julia> fld1(x, y)
4

julia> x == fld(x, y) * y + mod(x, y)
true

julia> x == (fld1(x, y) - 1) * y + mod1(x, y)
true
```

source

Base.mod1 – Function.

```
mod1(x, y)
```

Modulus after flooring division, returning a value `r` such that `mod(r, y) == mod(x, y)` in the range $(0, y]$ for positive `y` and in the range $[y, 0)$ for negative `y`.

```
julia> mod1(4, 2)
2

julia> mod1(4, 3)
1
```

source

Base.fldmod1 – Function.

```
fldmod1(x, y)
```

Return `(fld1(x,y), mod1(x,y))`.

See also: fld1, mod1.

source

Base.:// – Function.

```
//(num, den)
```

Divide two integers or rational numbers, giving a Rational result.

```
julia> 3 // 5
3//5

julia> (3 // 5) // (2 // 1)
3//10
```

source

Base.rationalize – Function.

```
rationalize([T<:Integer=Int,] x; tol::Real=eps(x))
```

Approximate floating point number x as a Rational number with components of the given integer type. The result will differ from x by no more than tol. If T is not provided, it defaults to Int.

```
julia> rationalize(5.6)
28//5

julia> a = rationalize(BigInt, 10.3)
103//10

julia> typeof(numerator(a))
BigInt
```

source

Base.numerator – Function.

```
numerator(x)
```

Numerator of the rational representation of x.

```
julia> numerator(2//3)
2

julia> numerator(4)
4
```

source

Base.denominator – Function.

```
denominator(x)
```

Denominator of the rational representation of x.

```
julia> denominator(2//3)
3

julia> denominator(4)
1
```

source

Base.:<< – Function.

```
<<(x, n)
```

Left bit shift operator, `x << n`. For `n >= 0`, the result is x shifted left by n bits, filling with `0`s. This is equivalent to `x * 2^n`. For `n < 0`, this is equivalent to `x >> -n`.

```
julia> Int8(3) << 2
12

julia> bits(Int8(3))
"00000011"

julia> bits(Int8(12))
"00001100"
```

See also >>, >>>.

source

```
<<(B::BitVector, n) -> BitVector
```

Left bit shift operator, `B << n`. For `n >= 0`, the result is B with elements shifted n positions backwards, filling with `false` values. If `n < 0`, elements are shifted forwards. Equivalent to `B >> -n`.

Examples

```
julia> B = BitVector([true, false, true, false, false])
5-element BitArray{1}:
  true
 false
  true
 false
 false

julia> B << 1
5-element BitArray{1}:
 false
  true
 false
 false
 false

julia> B << -1
5-element BitArray{1}:
```

```
 false
  true
 false
  true
 false
```

source

Base.:>> – Function.

```
>>(x, n)
```

Right bit shift operator, `x >> n`. For `n >= 0`, the result is x shifted right by n bits, where `n >= 0`, filling with `0`s if `x >= 0`, 1s if `x < 0`, preserving the sign of x. This is equivalent to `fld(x, 2^n)`. For `n < 0`, this is equivalent to `x << -n`.

```
julia> Int8(13) >> 2
3

julia> bits(Int8(13))
"00001101"

julia> bits(Int8(3))
"00000011"

julia> Int8(-14) >> 2
-4

julia> bits(Int8(-14))
"11110010"

julia> bits(Int8(-4))
"11111100"
```

See also >>>, <<.

source

```
>>(B::BitVector, n) -> BitVector
```

Right bit shift operator, `B >> n`. For `n >= 0`, the result is B with elements shifted n positions forward, filling with `false` values. If `n < 0`, elements are shifted backwards. Equivalent to `B << -n`.

Example

```
julia> B = BitVector([true, false, true, false, false])
5-element BitArray{1}:
  true
 false
  true
 false
 false

julia> B >> 1
5-element BitArray{1}:
 false
  true
```

```
 false
  true
 false

julia> B >> -1
5-element BitArray{1}:
 false
  true
 false
 false
 false
```

source

Base.:>>> – Function.

```
>>>(x, n)
```

Unsigned right bit shift operator, `x >>> n`. For `n >= 0`, the result is `x` shifted right by `n` bits, where `n >= 0`, filling with `0`s. For `n < 0`, this is equivalent to `x << -n`.

For `Unsigned` integer types, this is equivalent to `>>`. For `Signed` integer types, this is equivalent to `signed(unsigned(x) >> n)`.

```
julia> Int8(-14) >>> 2
60

julia> bits(Int8(-14))
"11110010"

julia> bits(Int8(60))
"00111100"
```

`BigInt`s are treated as if having infinite size, so no filling is required and this is equivalent to `>>`.

See also `>>`, `<<`.

source

```
>>>(B::BitVector, n) -> BitVector
```

Unsigned right bitshift operator, `B >>> n`. Equivalent to `B >> n`. See `>>` for details and examples.

source

Base.colon – Function.

```
colon(start, [step], stop)
```

Called by `:` syntax for constructing ranges.

```
julia> colon(1, 2, 5)
1:2:5
```

source

```
:(start, [step], stop)
```

Range operator. `a:b` constructs a range from `a` to `b` with a step size of 1, and `a:s:b` is similar but uses a step size of `s`. These syntaxes call the function `colon`. The colon is also used in indexing to select whole dimensions.

source

`Base.range` – Function.

```
range(start, [step], length)
```

Construct a range by length, given a starting value and optional step (defaults to 1).

source

`Base.OneTo` – Type.

```
Base.OneTo(n)
```

Define an `AbstractUnitRange` that behaves like `1:n`, with the added distinction that the lower limit is guaranteed (by the type system) to be 1.

source

`Base.StepRangeLen` – Type.

```
StepRangeLen{T,R,S}(ref::R, step::S, len, [offset=1])
```

A range `r` where `r[i]` produces values of type `T`, parametrized by a `reference` value, a `step`, and the `length`. By default `ref` is the starting value `r[1]`, but alternatively you can supply it as the value of `r[offset]` for some other index `1 <= offset <= len`. In conjunction with `TwicePrecision` this can be used to implement ranges that are free of roundoff error.

source

`Base.:==` – Function.

```
==(x, y)
```

Generic equality operator, giving a single `Bool` result. Falls back to `===`. Should be implemented for all types with a notion of equality, based on the abstract value that an instance represents. For example, all numeric types are compared by numeric value, ignoring type. Strings are compared as sequences of characters, ignoring encoding.

Follows IEEE semantics for floating-point numbers.

Collections should generally implement `==` by calling `==` recursively on all contents.

New numeric types should implement this function for two arguments of the new type, and handle comparison to other types via promotion rules where possible.

source

`Base.:!=` – Function.

```
!=(x, y)≠
(x,y)
```

Not-equals comparison operator. Always gives the opposite answer as `==`. New types should generally not implement this, and rely on the fallback definition `!=(x,y) = !(x==y)` instead.

```
julia> 3 != 2
true

julia> "foo" ≠ "foo"
false
```

source

Base.:!== – Function.

```
!==(x, y)
≠(x,y)
```

Equivalent to `!(x === y)`.

```
julia> a = [1, 2]; b = [1, 2];

julia> a ≡ b
true

julia> a ≡ a
false
```

source

Base.:< – Function.

```
<(x, y)
```

Less-than comparison operator. New numeric types should implement this function for two arguments of the new type. Because of the behavior of floating-point NaN values, < implements a partial order. Types with a canonical partial order should implement <, and types with a canonical total order should implement `isless`.

```
julia> 'a' < 'b'
true

julia> "abc" < "abd"
true

julia> 5 < 3
false
```

source

Base.:<= – Function.

```
<=(x, y)≤
≤(x,y)
```

Less-than-or-equals comparison operator.

```
julia> 'a' <= 'b'
true

julia> 7 ≤ 7 ≤ 9
true
```

```
julia> "abc" ≤ "abc"
true

julia> 5 <= 3
false
```

source

Base.:> – Function.

```
>(x, y)
```

Greater-than comparison operator. Generally, new types should implement < instead of this function, and rely on the fallback definition `>(x, y) = y < x`.

```
julia> 'a' > 'b'
false

julia> 7 > 3 > 1
true

julia> "abc" > "abd"
false

julia> 5 > 3
true
```

source

Base.:>= – Function.

```
>=(x, y)≥
(x,y)
```

Greater-than-or-equals comparison operator.

```
julia> 'a' >= 'b'
false

julia> 7 ≥ 7 ≥ 3
true

julia> "abc" ≥ "abc"
true

julia> 5 >= 3
true
```

source

Base.cmp – Function.

```
cmp(x,y)
```

Return -1, 0, or 1 depending on whether x is less than, equal to, or greater than y, respectively. Uses the total order implemented by `isless`. For floating-point numbers, uses < but throws an error for unordered arguments.

```
julia> cmp(1, 2)
-1

julia> cmp(2, 1)
1

julia> cmp(2+im, 3-im)
ERROR: MethodError: no method matching isless(::Complex{Int64}, ::Complex{Int64})
[...]
```

source

Base.:~ – Function.

```
~(x)
```

Bitwise not.

Examples

```
julia> ~4
-5

julia> ~10
-11

julia> ~true
false
```

source

Base.:& – Function.

```
&(x, y)
```

Bitwise and.

Examples

```
julia> 4 & 10
0

julia> 4 & 12
4
```

source

Base.:| – Function.

```
|(x, y)
```

Bitwise or.

Examples

```
julia> 4 | 10
14

julia> 4 | 1
5
```

source

Base.xor – Function.

```
xor(x, y)
⊻(x, y)
```

Bitwise exclusive or of x and y. The infix operation a ⊻ b is a synonym for xor(a,b), and ⊻ can be typed by tab-completing \xor or \veebar in the Julia REPL.

```
julia> [true; true; false] .⊻ [true; false; false]
3-element BitArray{1}:
 false
  true
 false
```

source

Base.:! – Function.

```
!(x)
```

Boolean not.

```
julia> !true
false

julia> !false
true

julia> .![true false true]
1×3 BitArray{2}:
 false  true  false
```

source

```
!f::Function
```

Predicate function negation: when the argument of ! is a function, it returns a function which computes the boolean negation of f. Example:

```
julia> str = " ε > 0,  δ > 0: |x-y| < δ  |f(x)-f(y)| < ε"
" ε > 0,  δ > 0: |x-y| < δ  |f(x)-f(y)| < ε"

julia> filter(isalpha, str)
"εδxyδfxfyε"

julia> filter(!isalpha, str)
"  > 0,   > 0: |-| <   |()-()| < "
```

source

&& – Keyword.

```
x && y
```

Short-circuiting boolean AND.

source

|| – Keyword.

```
x || y
```

Short-circuiting boolean OR.

source

47.2 Funciones Matemáticas

Base.isapprox – Function.

```
isapprox(x, y; rtol::Real=sqrt(eps), atol::Real=0, nans::Bool=false, norm::Function)
```

Inexact equality comparison: `true` if `norm(x-y) <= atol + rtol*max(norm(x), norm(y))`. The default `atol` is zero and the default `rtol` depends on the types of `x` and `y`. The keyword argument `nans` determines whether or not NaN values are considered equal (defaults to false).

For real or complex floating-point values, `rtol` defaults to `sqrt(eps(typeof(real(x-y))))`. This corresponds to requiring equality of about half of the significand digits. For other types, `rtol` defaults to zero.

`x` and `y` may also be arrays of numbers, in which case `norm` defaults to `vecnorm` but may be changed by passing a `norm::Function` keyword argument. (For numbers, `norm` is the same thing as `abs`.) When `x` and `y` are arrays, if `norm(x-y)` is not finite (i.e. `±Inf` or `NaN`), the comparison falls back to checking whether all elements of `x` and `y` are approximately equal component-wise.

The binary operator ≈ is equivalent to `isapprox` with the default arguments, and `x  y` is equivalent to `!isapprox(x,y)`.

```
julia> 0.1 ≈ (0.1 - 1e-10)
true

julia> isapprox(10, 11; atol = 2)
true

julia> isapprox([10.0^9, 1.0], [10.0^9, 2.0])
true
```

source

Base.sin – Function.

```
sin(x)
```

Compute sine of `x`, where `x` is in radians.

source

Base.cos – Function.

```
cos(x)
```

Compute cosine of `x`, where `x` is in radians.

source

`Base.tan` – Function.

```
tan(x)
```

Compute tangent of x, where x is in radians.

source

`Base.Math.sind` – Function.

```
sind(x)
```

Compute sine of x, where x is in degrees.

source

`Base.Math.cosd` – Function.

```
cosd(x)
```

Compute cosine of x, where x is in degrees.

source

`Base.Math.tand` – Function.

```
tand(x)
```

Compute tangent of x, where x is in degrees.

source

`Base.Math.sinpi` – Function.

```
sinpi(x)
```

Compute $\sin(\pi x)$ more accurately than `sin(pi*x)`, especially for large x.

source

`Base.Math.cospi` – Function.

```
cospi(x)
```

Compute $\cos(\pi x)$ more accurately than `cos(pi*x)`, especially for large x.

source

`Base.sinh` – Function.

```
sinh(x)
```

Compute hyperbolic sine of x.

source

`Base.cosh` – Function.

```
cosh(x)
```

Compute hyperbolic cosine of x.

source

`Base.tanh` – Function.

```
tanh(x)
```

Compute hyperbolic tangent of `x`.

source

`Base.asin` – Function.

```
asin(x)
```

Compute the inverse sine of `x`, where the output is in radians.

source

`Base.acos` – Function.

```
acos(x)
```

Compute the inverse cosine of `x`, where the output is in radians

source

`Base.atan` – Function.

```
atan(x)
```

Compute the inverse tangent of `x`, where the output is in radians.

source

`Base.Math.atan2` – Function.

```
atan2(y, x)
```

Compute the inverse tangent of `y/x`, using the signs of both `x` and `y` to determine the quadrant of the return value.

source

`Base.Math.asind` – Function.

```
asind(x)
```

Compute the inverse sine of `x`, where the output is in degrees.

source

`Base.Math.acosd` – Function.

```
acosd(x)
```

Compute the inverse cosine of `x`, where the output is in degrees.

source

`Base.Math.atand` – Function.

```
atand(x)
```

Compute the inverse tangent of x, where the output is in degrees.

source

Base.Math.sec – Function.

```
sec(x)
```

Compute the secant of x, where x is in radians.

source

Base.Math.csc – Function.

```
csc(x)
```

Compute the cosecant of x, where x is in radians.

source

Base.Math.cot – Function.

```
cot(x)
```

Compute the cotangent of x, where x is in radians.

source

Base.Math.secd – Function.

```
secd(x)
```

Compute the secant of x, where x is in degrees.

source

Base.Math.cscd – Function.

```
cscd(x)
```

Compute the cosecant of x, where x is in degrees.

source

Base.Math.cotd – Function.

```
cotd(x)
```

Compute the cotangent of x, where x is in degrees.

source

Base.Math.asec – Function.

```
asec(x)
```

Compute the inverse secant of x, where the output is in radians.

source

Base.Math.acsc – Function.

```
acsc(x)
```

Compute the inverse cosecant of x, where the output is in radians.

source

Base.Math.acot – Function.

```
acot(x)
```

Compute the inverse cotangent of x, where the output is in radians.

source

Base.Math.asecd – Function.

```
asecd(x)
```

Compute the inverse secant of x, where the output is in degrees.

source

Base.Math.acscd – Function.

```
acscd(x)
```

Compute the inverse cosecant of x, where the output is in degrees.

source

Base.Math.acotd – Function.

```
acotd(x)
```

Compute the inverse cotangent of x, where the output is in degrees.

source

Base.Math.sech – Function.

```
sech(x)
```

Compute the hyperbolic secant of x

source

Base.Math.csch – Function.

```
csch(x)
```

Compute the hyperbolic cosecant of x.

source

Base.Math.coth – Function.

```
coth(x)
```

Compute the hyperbolic cotangent of x.

source

Base.asinh – Function.

```
asinh(x)
```

Compute the inverse hyperbolic sine of x.

source

Base.acosh – Function.

```
acosh(x)
```

Compute the inverse hyperbolic cosine of x.

source

Base.atanh – Function.

```
atanh(x)
```

Compute the inverse hyperbolic tangent of x.

source

Base.Math.asech – Function.

```
asech(x)
```

Compute the inverse hyperbolic secant of x.

source

Base.Math.acsch – Function.

```
acsch(x)
```

Compute the inverse hyperbolic cosecant of x.

source

Base.Math.acoth – Function.

```
acoth(x)
```

Compute the inverse hyperbolic cotangent of x.

source

Base.Math.sinc – Function.

```
sinc(x)
```

Compute $\sin(\pi x)/(\pi x)$ if $x \neq 0$, and 1 if $x = 0$.

source

Base.Math.cosc – Function.

```
cosc(x)
```

Compute $\cos(\pi x)/x - \sin(\pi x)/(\pi x^2)$ if $x \neq 0$, and 0 if $x = 0$. This is the derivative of `sinc(x)`.

source

`Base.Math.deg2rad` – Function.

```
deg2rad(x)
```

Convert x from degrees to radians.

```
julia> deg2rad(90)
1.5707963267948966
```

source

`Base.Math.rad2deg` – Function.

```
rad2deg(x)
```

Convert x from radians to degrees.

```
julia> rad2deg(pi)
180.0
```

source

`Base.Math.hypot` – Function.

```
hypot(x, y)
```

Compute the hypotenuse $\sqrt{x^2+y^2}$ avoiding overflow and underflow.

Examples

```
julia> a = 10^10;

julia> hypot(a, a)
1.4142135623730951e10

julia> √(a^2 + a^2) # a^2 overflows
ERROR: DomainError:
sqrt will only return a complex result if called with a complex argument. Try
↪  sqrt(complex(x)).
Stacktrace:
 [1] sqrt(::Int64) at ./math.jl:434
```

source

```
hypot(x...)
```

Compute the hypotenuse $\sqrt{\sum x_i^2}$ avoiding overflow and underflow.

source

`Base.log` – Method.

```
log(x)
```

Compute the natural logarithm of x. Throws `DomainError` for negative `Real` arguments. Use complex negative arguments to obtain complex results.

There is an experimental variant in the `Base.Math.JuliaLibm` module, which is typically faster and more accurate.

source

`Base.log` – Method.

```
log(b,x)
```

Compute the base b logarithm of x. Throws `DomainError` for negative `Real` arguments.

```
julia> log(4,8)
1.5

julia> log(4,2)
0.5
```

Note

If b is a power of 2 or 10, `log2` or `log10` should be used, as these will typically be faster and more accurate. For example,

```
julia> log(100,1000000)
2.9999999999999996

julia> log10(1000000)/2
3.0
```

source

`Base.log2` – Function.

```
log2(x)
```

Compute the logarithm of x to base 2. Throws `DomainError` for negative `Real` arguments.

Example

```
julia> log2(4)
2.0

julia> log2(10)
3.321928094887362
```

source

`Base.log10` – Function.

```
log10(x)
```

Compute the logarithm of x to base 10. Throws `DomainError` for negative `Real` arguments.

Example

```
julia> log10(100)
2.0

julia> log10(2)
0.3010299956639812
```

source

`Base.log1p` – Function.

```
log1p(x)
```

Accurate natural logarithm of 1+x. Throws `DomainError` for `Real` arguments less than -1.

There is an experimental variant in the `Base.Math.JuliaLibm` module, which is typically faster and more accurate.

Examples

```
julia> log1p(-0.5)
-0.6931471805599453

julia> log1p(0)
0.0
```

source

`Base.Math.frexp` – Function.

```
frexp(val)
```

Return `(x, exp)` such that x has a magnitude in the interval $[1/2, 1)$ or 0, and `val` is equal to $x \times 2^{exp}$.

source

`Base.exp` – Function.

```
exp(x)
```

Compute the natural base exponential of x, in other words e^x.

source

`Base.exp2` – Function.

```
exp2(x)
```

Compute 2^x.

```
julia> exp2(5)
32.0
```

source

`Base.exp10` – Function.

```
exp10(x)
```

Compute 10^x.

Examples

```
julia> exp10(2)
100.0

julia> exp10(0.2)
1.5848931924611136
```

source

`Base.Math.ldexp` – Function.

```
ldexp(x, n)
```

Compute $x \times 2^n$.

Example

```
julia> ldexp(5., 2)
20.0
```

source

`Base.Math.modf` – Function.

```
modf(x)
```

Return a tuple (fpart,ipart) of the fractional and integral parts of a number. Both parts have the same sign as the argument.

Example

```
julia> modf(3.5)
(0.5, 3.0)
```

source

`Base.expm1` – Function.

```
expm1(x)
```

Accurately compute $e^x - 1$.

source

`Base.round` – Method.

```
round([T,] x, [digits, [base]], [r::RoundingMode])
```

Rounds x to an integer value according to the provided `RoundingMode`, returning a value of the same type as x. When not specifying a rounding mode the global mode will be used (see `rounding`), which by default is round to the nearest integer (`RoundNearest` mode), with ties (fractional values of 0.5) being rounded to the nearest even integer.

```
julia> round(1.7)
2.0

julia> round(1.5)
2.0

julia> round(2.5)
2.0
```

The optional `RoundingMode` argument will change how the number gets rounded.

`round(T, x, [r::RoundingMode])` converts the result to type T, throwing an `InexactError` if the value is not representable.

`round(x, digits)` rounds to the specified number of digits after the decimal place (or before if negative). `round(x, digits, base)` rounds using a base other than 10.

```
julia> round(pi, 2)
3.14

julia> round(pi, 3, 2)
3.125
```

Note

Rounding to specified digits in bases other than 2 can be inexact when operating on binary floating point numbers. For example, the `Float64` value represented by `1.15` is actually *less* than 1.15, yet will be rounded to 1.2.

```
julia> x = 1.15
1.15

julia> @sprintf "%.20f" x
"1.14999999999999991118"

julia> x < 115//100
true

julia> round(x, 1)
1.2
```

source

`Base.Rounding.RoundingMode` – Type.

```
RoundingMode
```

A type used for controlling the rounding mode of floating point operations (via `rounding`/`setrounding` functions), or as optional arguments for rounding to the nearest integer (via the `round` function).

Currently supported rounding modes are:

- `RoundNearest` (default)
- `RoundNearestTiesAway`
- `RoundNearestTiesUp`
- `RoundToZero`
- `RoundFromZero` (`BigFloat` only)
- `RoundUp`
- `RoundDown`

source

`Base.Rounding.RoundNearest` – Constant.

```
RoundNearest
```

The default rounding mode. Rounds to the nearest integer, with ties (fractional values of 0.5) being rounded to the nearest even integer.

source

`Base.Rounding.RoundNearestTiesAway` – Constant.

```
RoundNearestTiesAway
```

Rounds to nearest integer, with ties rounded away from zero (C/C++ `round` behaviour).

source

`Base.Rounding.RoundNearestTiesUp` – Constant.

```
RoundNearestTiesUp
```

Rounds to nearest integer, with ties rounded toward positive infinity (Java/JavaScript `round` behaviour).

source

`Base.Rounding.RoundToZero` – Constant.

```
RoundToZero
```

`round` using this rounding mode is an alias for `trunc`.

source

`Base.Rounding.RoundUp` – Constant.

```
RoundUp
```

`round` using this rounding mode is an alias for `ceil`.

source

`Base.Rounding.RoundDown` – Constant.

```
RoundDown
```

`round` using this rounding mode is an alias for `floor`.

source

`Base.round` – Method.

```
round(z, RoundingModeReal, RoundingModeImaginary)
```

Returns the nearest integral value of the same type as the complex-valued `z` to `z`, breaking ties using the specified `RoundingMode`s. The first `RoundingMode` is used for rounding the real components while the second is used for rounding the imaginary components.

source

`Base.ceil` – Function.

```
ceil([T,] x, [digits, [base]])
```

`ceil(x)` returns the nearest integral value of the same type as `x` that is greater than or equal to `x`.

`ceil(T, x)` converts the result to type `T`, throwing an `InexactError` if the value is not representable.

`digits` and `base` work as for `round`.

source

`Base.floor` – Function.

```
floor([T,] x, [digits, [base]])
```

`floor(x)` returns the nearest integral value of the same type as `x` that is less than or equal to `x`.

`floor(T, x)` converts the result to type `T`, throwing an `InexactError` if the value is not representable.

`digits` and `base` work as for `round`.

source

`Base.trunc` – Function.

```
trunc([T,] x, [digits, [base]])
```

`trunc(x)` returns the nearest integral value of the same type as `x` whose absolute value is less than or equal to `x`.

`trunc(T, x)` converts the result to type `T`, throwing an `InexactError` if the value is not representable.

`digits` and `base` work as for `round`.

source

`Base.unsafe_trunc` – Function.

```
unsafe_trunc(T, x)
```

`unsafe_trunc(T, x)` returns the nearest integral value of type `T` whose absolute value is less than or equal to `x`. If the value is not representable by `T`, an arbitrary value will be returned.

source

`Base.signif` – Function.

```
signif(x, digits, [base])
```

Rounds (in the sense of `round`) `x` so that there are `digits` significant digits, under a base `base` representation, default 10. E.g., `signif(123.456, 2)` is `120.0`, and `signif(357.913, 4, 2)` is `352.0`.

source

`Base.min` – Function.

```
min(x, y, ...)
```

Return the minimum of the arguments. See also the `minimum` function to take the minimum element from a collection.

```
julia> min(2, 5, 1)
1
```

source

`Base.max` – Function.

```
max(x, y, ...)
```

Return the maximum of the arguments. See also the `maximum` function to take the maximum element from a collection.

```
julia> max(2, 5, 1)
5
```

source

Base.minmax – Function.

```
minmax(x, y)
```

Return `(min(x,y), max(x,y))`. See also: `extrema` that returns `(minimum(x), maximum(x))`.

```
julia> minmax('c','b')
('b', 'c')
```

source

Base.Math.clamp – Function.

```
clamp(x, lo, hi)
```

Return x if `lo <= x <= hi`. If `x < lo`, return `lo`. If `x > hi`, return `hi`. Arguments are promoted to a common type.

```
julia> clamp.([pi, 1.0, big(10.)], 2., 9.)
3-element Array{BigFloat,1}:
 3.14159265358979323846264338327950288419716939937510582097494459230781640628619
 2.00000000000000000000000000000000000000000000000000000000000000000000000000000
 9.00000000000000000000000000000000000000000000000000000000000000000000000000000
```

source

Base.Math.clamp! – Function.

```
clamp!(array::AbstractArray, lo, hi)
```

Restrict values in `array` to the specified range, in-place. See also `clamp`.

source

Base.abs – Function.

```
abs(x)
```

The absolute value of x.

When `abs` is applied to signed integers, overflow may occur, resulting in the return of a negative value. This overflow occurs only when `abs` is applied to the minimum representable value of a signed integer. That is, when `x == typemin(typeof(x))`, `abs(x) == x < 0`, not `-x` as might be expected.

```
julia> abs(-3)
3

julia> abs(1 + im)
1.4142135623730951

julia> abs(typemin(Int64))
-9223372036854775808
```

source

Base.Checked.checked_abs – Function.

```
Base.checked_abs(x)
```

Calculates abs(x), checking for overflow errors where applicable. For example, standard two's complement signed integers (e.g. Int) cannot represent abs(typemin(Int)), thus leading to an overflow.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_neg – Function.

```
Base.checked_neg(x)
```

Calculates -x, checking for overflow errors where applicable. For example, standard two's complement signed integers (e.g. Int) cannot represent -typemin(Int), thus leading to an overflow.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_add – Function.

```
Base.checked_add(x, y)
```

Calculates x+y, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_sub – Function.

```
Base.checked_sub(x, y)
```

Calculates x-y, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_mul – Function.

```
Base.checked_mul(x, y)
```

Calculates x*y, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_div – Function.

```
Base.checked_div(x, y)
```

Calculates div(x,y), checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_rem – Function.

```
Base.checked_rem(x, y)
```

Calculates x%y, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_fld – Function.

```
Base.checked_fld(x, y)
```

Calculates `fld(x,y)`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_mod – Function.

```
Base.checked_mod(x, y)
```

Calculates `mod(x,y)`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_cld – Function.

```
Base.checked_cld(x, y)
```

Calculates `cld(x,y)`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.add_with_overflow – Function.

```
Base.add_with_overflow(x, y) -> (r, f)
```

Calculates `r = x+y`, with the flag `f` indicating whether overflow has occurred.

source

Base.Checked.sub_with_overflow – Function.

```
Base.sub_with_overflow(x, y) -> (r, f)
```

Calculates `r = x-y`, with the flag `f` indicating whether overflow has occurred.

source

Base.Checked.mul_with_overflow – Function.

```
Base.mul_with_overflow(x, y) -> (r, f)
```

Calculates `r = x*y`, with the flag `f` indicating whether overflow has occurred.

source

`Base.abs2` – Function.

```
abs2(x)
```

Squared absolute value of `x`.

```
julia> abs2(-3)
9
```

source

`Base.copysign` – Function.

```
copysign(x, y) -> z
```

Return `z` which has the magnitude of `x` and the same sign as `y`.

Examples

```
julia> copysign(1, -2)
-1

julia> copysign(-1, 2)
1
```

source

`Base.sign` – Function.

```
sign(x)
```

Return zero if `x==0` and $x/|x|$ otherwise (i.e., ±1 for real `x`).

source

`Base.signbit` – Function.

```
signbit(x)
```

Returns `true` if the value of the sign of `x` is negative, otherwise `false`.

Examples

```
julia> signbit(-4)
true

julia> signbit(5)
false

julia> signbit(5.5)
false

julia> signbit(-4.1)
true
```

source

`Base.flipsign` – Function.

```
flipsign(x, y)
```

Return x with its sign flipped if y is negative. For example `abs(x) = flipsign(x,x)`.

```
julia> flipsign(5, 3)
5

julia> flipsign(5, -3)
-5
```

source

`Base.sqrt` – Function.

```
sqrt(x)
```

Return $\sqrt{x}$. Throws `DomainError` for negative `Real` arguments. Use complex negative arguments instead. The prefix operator √ is equivalent to `sqrt`.

source

`Base.isqrt` – Function.

```
isqrt(n::Integer)
```

Integer square root: the largest integer `m` such that `m*m <= n`.

```
julia> isqrt(5)
2
```

source

`Base.Math.cbrt` – Function.

```
cbrt(x::Real)
```

Return the cube root of x, i.e. $x^{1/3}$. Negative values are accepted (returning the negative real root when $x < 0$).

The prefix operator ∛ is equivalent to `cbrt`.

```
julia> cbrt(big(27))
3.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

source

`Base.real` – Method.

```
real(z)
```

Return the real part of the complex number z.

```
julia> real(1 + 3im)
1
```

source

`Base.imag` – Function.

```
imag(z)
```

Return the imaginary part of the complex number z.

```
julia> imag(1 + 3im)
3
```

source

Base.reim – Function.

```
reim(z)
```

Return both the real and imaginary parts of the complex number z.

```
julia> reim(1 + 3im)
(1, 3)
```

source

Base.conj – Function.

```
conj(v::RowVector)
```

Returns a ConjArray lazy view of the input, where each element is conjugated.

Example

```
julia> v = [1+im, 1-im].'
1×2 RowVector{Complex{Int64},Array{Complex{Int64},1}}:
 1+1im  1-1im

julia> conj(v)
1×2 RowVector{Complex{Int64},ConjArray{Complex{Int64},1,Array{Complex{Int64},1}}}:
 1-1im  1+1im
```

source

```
conj(z)
```

Compute the complex conjugate of a complex number z.

```
julia> conj(1 + 3im)
1 - 3im
```

source

Base.angle – Function.

```
angle(z)
```

Compute the phase angle in radians of a complex number z.

source

Base.cis – Function.

```
cis(z)
```

Return $\exp(iz)$.

source

`Base.binomial` – Function.

```
binomial(n, k)
```

Number of ways to choose k out of n items.

Example

```
julia> binomial(5, 3)
10

julia> factorial(5) ÷ (factorial(5-3) * factorial(3))
10
```

source

`Base.factorial` – Function.

```
factorial(n)
```

Factorial of n. If n is an `Integer`, the factorial is computed as an integer (promoted to at least 64 bits). Note that this may overflow if n is not small, but you can use `factorial(big(n))` to compute the result exactly in arbitrary precision. If n is not an `Integer`, `factorial(n)` is equivalent to `gamma(n+1)`.

```
julia> factorial(6)
720

julia> factorial(21)
ERROR: OverflowError()
[...]

julia> factorial(21.0)
5.109094217170944e19

julia> factorial(big(21))
51090942171709440000
```

source

`Base.gcd` – Function.

```
gcd(x,y)
```

Greatest common (positive) divisor (or zero if x and y are both zero).

Examples

```
julia> gcd(6,9)
3

julia> gcd(6,-9)
3
```

source

Base.lcm – Function.

```
lcm(x,y)
```

Least common (non-negative) multiple.

Examples

```
julia> lcm(2,3)
6

julia> lcm(-2,3)
6
```

source

Base.gcdx – Function.

```
gcdx(x,y)
```

Computes the greatest common (positive) divisor of x and y and their Bézout coefficients, i.e. the integer coefficients u and v that satisfy $ux + vy = d = gcd(x, y)$. $gcdx(x, y)$ returns (d, u, v).

Examples

```
julia> gcdx(12, 42)
(6, -3, 1)

julia> gcdx(240, 46)
(2, -9, 47)
```

> **Note**
>
> Bézout coefficients are *not* uniquely defined. `gcdx` returns the minimal Bézout coefficients that are computed by the extended Euclidean algorithm. (Ref: D. Knuth, TAoCP, 2/e, p. 325, Algorithm X.) For signed integers, these coefficients u and v are minimal in the sense that $|u| < |y/d|$ and $|v| < |x/d|$. Furthermore, the signs of u and v are chosen so that d is positive. For unsigned integers, the coefficients u and v might be near their `typemax`, and the identity then holds only via the unsigned integers' modulo arithmetic.

source

Base.ispow2 – Function.

```
ispow2(n::Integer) -> Bool
```

Test whether n is a power of two.

Examples

```
julia> ispow2(4)
true

julia> ispow2(5)
false
```

source

Base.nextpow2 – Function.

```
nextpow2(n::Integer)
```

The smallest power of two not less than n. Returns 0 for n==0, and returns -nextpow2(-n) for negative arguments.

Examples

```
julia> nextpow2(16)
16

julia> nextpow2(17)
32
```

source

Base.prevpow2 – Function.

```
prevpow2(n::Integer)
```

The largest power of two not greater than n. Returns 0 for n==0, and returns -prevpow2(-n) for negative arguments.

Examples

```
julia> prevpow2(5)
4

julia> prevpow2(0)
0
```

source

Base.nextpow – Function.

```
nextpow(a, x)
```

The smallest a^n not less than x, where n is a non-negative integer. a must be greater than 1, and x must be greater than 0.

Examples

```
julia> nextpow(2, 7)
8

julia> nextpow(2, 9)
16

julia> nextpow(5, 20)
25

julia> nextpow(4, 16)
16
```

See also prevpow.

source

Base.prevpow – Function.

```
prevpow(a, x)
```

The largest a^n not greater than x, where n is a non-negative integer. a must be greater than 1, and x must not be less than 1.

Examples

```
julia> prevpow(2, 7)
4

julia> prevpow(2, 9)
8

julia> prevpow(5, 20)
5

julia> prevpow(4, 16)
16
```

See also nextpow.

source

Base.nextprod – Function.

```
nextprod([k_1, k_2,...], n)
```

Next integer greater than or equal to n that can be written as $\prod k_i^{p_i}$ for integers p_1, p_2, etc.

Example

```
julia> nextprod([2, 3], 105)
108

julia> 2^2 * 3^3
108
```

source

Base.invmod – Function.

```
invmod(x,m)
```

Take the inverse of x modulo m: y such that $xy = 1 \pmod m$, with $div(x, y) = 0$. This is undefined for $m = 0$, or if $gcd(x, m) \neq 1$.

Examples

```
julia> invmod(2,5)
3

julia> invmod(2,3)
```

```
2

julia> invmod(5,6)
5
```

source

Base.powermod – Function.

```
powermod(x::Integer, p::Integer, m)
```

Compute $x^p \pmod m$.

Examples

```
julia> powermod(2, 6, 5)
4

julia> mod(2^6, 5)
4

julia> powermod(5, 2, 20)
5

julia> powermod(5, 2, 19)
6

julia> powermod(5, 3, 19)
11
```

source

Base.Math.gamma – Function.

```
gamma(x)
```

Compute the gamma function of x.

source

Base.Math.lgamma – Function.

```
lgamma(x)
```

Compute the logarithm of the absolute value of gamma for Real x, while for Complex x compute the principal branch cut of the logarithm of gamma(x) (defined for negative real(x) by analytic continuation from positive real(x)).

source

Base.Math.lfact – Function.

```
lfact(x)
```

Compute the logarithmic factorial of a nonnegative integer x. Equivalent to lgamma of x + 1, but lgamma extends this function to non-integer x.

source

`Base.Math.beta` – Function.

```
beta(x, y)
```

Euler integral of the first kind $\mathrm{B}(x,y) = \Gamma(x)\Gamma(y)/\Gamma(x+y)$.

source

`Base.Math.lbeta` – Function.

```
lbeta(x, y)
```

Natural logarithm of the absolute value of the `beta` function $\log(|\mathrm{B}(x,y)|)$.

source

`Base.ndigits` – Function.

```
ndigits(n::Integer, b::Integer=10)
```

Compute the number of digits in integer n written in base b. The base b must not be in `[-1, 0, 1]`.

Examples

```
julia> ndigits(12345)
5

julia> ndigits(1022, 16)
3

julia> base(16, 1022)
"3fe"
```

source

`Base.widemul` – Function.

```
widemul(x, y)
```

Multiply x and y, giving the result as a larger type.

```
julia> widemul(Float32(3.), 4.)
1.2000000000000000000000000000000000000000000000000000000000000000000000000000000000000000e+01
```

source

`Base.Math.@evalpoly` – Macro.

```
@evalpoly(z, c...)
```

Evaluate the polynomial $\sum_k c[k] z^{k-1}$ for the coefficients `c[1]`, `c[2]`, ...; that is, the coefficients are given in ascending order by power of z. This macro expands to efficient inline code that uses either Horner's method or, for complex z, a more efficient Goertzel-like algorithm.

```
julia> @evalpoly(3, 1, 0, 1)
10

julia> @evalpoly(2, 1, 0, 1)
5

julia> @evalpoly(2, 1, 1, 1)
7
```

source

47.3 Estadística

Base.mean – Function.

```
mean(f::Function, v)
```

Apply the function f to each element of v and take the mean.

```
julia> mean(√, [1, 2, 3])
1.3820881233139908

julia> mean([√1, √2, √3])
1.3820881233139908
```

source

```
mean(v[, region])
```

Compute the mean of whole array v, or optionally along the dimensions in region.

> **Note**
> Julia does not ignore NaN values in the computation. For applications requiring the handling of missing data, the DataArrays.jl package is recommended.

source

Base.mean! – Function.

```
mean!(r, v)
```

Compute the mean of v over the singleton dimensions of r, and write results to r.

source

Base.std – Function.

```
std(v[, region]; corrected::Bool=true, mean=nothing)
```

Compute the sample standard deviation of a vector or array v, optionally along dimensions in region. The algorithm returns an estimator of the generative distribution's standard deviation under the assumption that each entry of v is an IID drawn from that generative distribution. This computation is equivalent to calculating sqrt(sum((v - mean(v)).^2) / (length(v) - 1)). A pre-computed mean may be provided. If corrected is true, then the sum is scaled with n-1, whereas the sum is scaled with n if corrected is false where n = length(x).

> **Note**
> Julia does not ignore NaN values in the computation. For applications requiring the handling of missing data, the DataArrays.jl package is recommended.

source

Base.stdm – Function.

```
stdm(v, m::Number; corrected::Bool=true)
```

Compute the sample standard deviation of a vector `v` with known mean `m`. If `corrected` is `true`, then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = length(x)`.

> **Note**
> Julia does not ignore `NaN` values in the computation. For applications requiring the handling of missing data, the `DataArrays.jl` package is recommended.

source

`Base.var` – Function.

```
var(v[, region]; corrected::Bool=true, mean=nothing)
```

Compute the sample variance of a vector or array `v`, optionally along dimensions in `region`. The algorithm will return an estimator of the generative distribution's variance under the assumption that each entry of `v` is an IID drawn from that generative distribution. This computation is equivalent to calculating `sum(abs2, v - mean(v)) / (length(v) - 1)`. If `corrected` is `true`, then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = length(x)`. The mean `mean` over the region may be provided.

> **Note**
> Julia does not ignore `NaN` values in the computation. For applications requiring the handling of missing data, the `DataArrays.jl` package is recommended.

source

`Base.varm` – Function.

```
varm(v, m[, region]; corrected::Bool=true)
```

Compute the sample variance of a collection `v` with known mean(s) `m`, optionally over `region`. `m` may contain means for each dimension of `v`. If `corrected` is `true`, then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = length(x)`.

> **Note**
> Julia does not ignore `NaN` values in the computation. For applications requiring the handling of missing data, the `DataArrays.jl` package is recommended.

source

`Base.middle` – Function.

```
middle(x)
```

Compute the middle of a scalar value, which is equivalent to `x` itself, but of the type of `middle(x, x)` for consistency.

source

```
middle(x, y)
```

Compute the middle of two reals `x` and `y`, which is equivalent in both value and type to computing their mean (`(x + y) / 2`).

source

```
middle(range)
```

Compute the middle of a range, which consists of computing the mean of its extrema. Since a range is sorted, the mean is performed with the first and last element.

```
julia> middle(1:10)
5.5
```

source

```
middle(a)
```

Compute the middle of an array a, which consists of finding its extrema and then computing their mean.

```
julia> a = [1,2,3.6,10.9]
4-element Array{Float64,1}:
  1.0
  2.0
  3.6
 10.9

julia> middle(a)
5.95
```

source

Base.median – Function.

```
median(v[, region])
```

Compute the median of an entire array v, or, optionally, along the dimensions in region. For an even number of elements no exact median element exists, so the result is equivalent to calculating mean of two median elements.

> **Note**
> Julia does not ignore NaN values in the computation. For applications requiring the handling of missing data, the DataArrays.jl package is recommended.

source

Base.median! – Function.

```
median!(v)
```

Like median, but may overwrite the input vector.

source

Base.quantile – Function.

```
quantile(v, p; sorted=false)
```

Compute the quantile(s) of a vector v at a specified probability or vector p. The keyword argument sorted indicates whether v can be assumed to be sorted.

The p should be on the interval [0,1], and v should not have any NaN values.

Quantiles are computed via linear interpolation between the points ((k-1)/(n-1), v[k]), for k = 1:n where n = length(v). This corresponds to Definition 7 of Hyndman and Fan (1996), and is the same as the R default.

Note

Julia does not ignore NaN values in the computation. For applications requiring the handling of missing data, the `DataArrays.jl` package is recommended. `quantile` will throw an `ArgumentError` in the presence of NaN values in the data array.

- Hyndman, R.J and Fan, Y. (1996) "Sample Quantiles in Statistical Packages", *The American Statistician*, Vol. 50, No. 4, pp. 361-365

source

`Base.quantile!` – Function.

```
quantile!([q, ] v, p; sorted=false)
```

Compute the quantile(s) of a vector `v` at the probabilities `p`, with optional output into array `q` (if not provided, a new output array is created). The keyword argument `sorted` indicates whether `v` can be assumed to be sorted; if `false` (the default), then the elements of `v` may be partially sorted.

The elements of `p` should be on the interval [0,1], and `v` should not have any NaN values.

Quantiles are computed via linear interpolation between the points `((k-1)/(n-1), v[k])`, for `k = 1:n` where `n = length(v)`. This corresponds to Definition 7 of Hyndman and Fan (1996), and is the same as the R default.

Note

Julia does not ignore NaN values in the computation. For applications requiring the handling of missing data, the `DataArrays.jl` package is recommended. `quantile!` will throw an `ArgumentError` in the presence of NaN values in the data array.

- Hyndman, R.J and Fan, Y. (1996) "Sample Quantiles in Statistical Packages", *The American Statistician*, Vol. 50, No. 4, pp. 361-365

source

`Base.cov` – Function.

```
cov(x[, corrected=true])
```

Compute the variance of the vector `x`. If `corrected` is `true` (the default) then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = length(x)`.

source

```
cov(X[, vardim=1, corrected=true])
```

Compute the covariance matrix of the matrix `X` along the dimension `vardim`. If `corrected` is `true` (the default) then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = size(X, vardim)`.

source

```
cov(x, y[, corrected=true])
```

Compute the covariance between the vectors `x` and `y`. If `corrected` is `true` (the default), computes $\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})^*$ where $*$ denotes the complex conjugate and `n = length(x) = length(y)`. If `corrected` is `false`, computes $rac1nsum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})^*$.

source

```
cov(X, Y[, vardim=1, corrected=true])
```

Compute the covariance between the vectors or matrices `X` and `Y` along the dimension `vardim`. If `corrected` is `true` (the default) then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = size(X, vardim) = size(Y, vardim)`.

source

`Base.cor` – Function.

```
cor(x)
```

Return the number one.

source

```
cor(X[, vardim=1])
```

Compute the Pearson correlation matrix of the matrix `X` along the dimension `vardim`.

source

```
cor(x, y)
```

Compute the Pearson correlation between the vectors `x` and `y`.

source

```
cor(X, Y[, vardim=1])
```

Compute the Pearson correlation between the vectors or matrices `X` and `Y` along the dimension `vardim`.

source

47.4 Procesamiento de Señales

Las funciones de transformada rápida de Fourier (*Fast Fourier transform* – FFT) en Julia están implementadas mediante llamadas a funciones de la librería FFTW.

`Base.DFT.fft` – Function.

```
fft(A [, dims])
```

Performs a multidimensional FFT of the array `A`. The optional `dims` argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. Most efficient if the size of `A` along the transformed dimensions is a product of small primes; see `nextprod()`. See also `plan_fft()` for even greater efficiency.

A one-dimensional FFT computes the one-dimensional discrete Fourier transform (DFT) as defined by

$$\mathrm{DFT}(A)[k] = \sum_{n=1}^{\mathrm{length}(A)} \exp\left(-i\frac{2\pi(n-1)(k-1)}{\mathrm{length}(A)}\right) A[n].$$

A multidimensional FFT simply performs this operation along each transformed dimension of `A`.

Note

- Julia starts FFTW up with 1 thread by default. Higher performance is usually possible by increasing number of threads. Use `FFTW.set_num_threads(Sys.CPU_CORES)` to use as many threads as cores on your system.
- This performs a multidimensional FFT by default. FFT libraries in other languages such as Python and Octave perform a one-dimensional FFT along the first non-singleton dimension of the array. This is worth noting while performing comparisons. For more details, refer to the Noteworthy Differences from other Languages section of the manual.

source

`Base.DFT.fft!` – Function.

```
fft!(A [, dims])
```

Same as `fft`, but operates in-place on `A`, which must be an array of complex floating-point numbers.

source

`Base.DFT.ifft` – Function.

```
ifft(A [, dims])
```

Multidimensional inverse FFT.

A one-dimensional inverse FFT computes

$$\mathrm{IDFT}(A)[k] = \frac{1}{\mathrm{length}(A)} \sum_{n=1}^{\mathrm{length}(A)} \exp\left(+i\frac{2\pi(n-1)(k-1)}{\mathrm{length}(A)}\right) A[n].$$

A multidimensional inverse FFT simply performs this operation along each transformed dimension of `A`.

source

`Base.DFT.ifft!` – Function.

```
ifft!(A [, dims])
```

Same as `ifft`, but operates in-place on `A`.

source

`Base.DFT.bfft` – Function.

```
bfft(A [, dims])
```

Similar to `ifft`, but computes an unnormalized inverse (backward) transform, which must be divided by the product of the sizes of the transformed dimensions in order to obtain the inverse. (This is slightly more efficient than `ifft` because it omits a scaling step, which in some applications can be combined with other computational steps elsewhere.)

$$\mathrm{BDFT}(A)[k] = \mathrm{length}(A)\,\mathrm{IDFT}(A)[k]$$

source

Base.DFT.bfft! – Function.

```
bfft!(A [, dims])
```

Same as bfft, but operates in-place on A.

source

Base.DFT.plan_fft – Function.

```
plan_fft(A [, dims]; flags=FFTW.ESTIMATE;  timelimit=Inf)
```

Pre-plan an optimized FFT along given dimensions (dims) of arrays matching the shape and type of A. (The first two arguments have the same meaning as for fft.) Returns an object P which represents the linear operator computed by the FFT, and which contains all of the information needed to compute fft(A, dims) quickly.

To apply P to an array A, use P * A; in general, the syntax for applying plans is much like that of matrices. (A plan can only be applied to arrays of the same size as the A for which the plan was created.) You can also apply a plan with a preallocated output array Â by calling A_mul_B!(Â, plan, A). (For A_mul_B!, however, the input array A must be a complex floating-point array like the output Â.) You can compute the inverse-transform plan by inv(P) and apply the inverse plan with P \ Â (the inverse plan is cached and reused for subsequent calls to inv or \), and apply the inverse plan to a pre-allocated output array A with A_ldiv_B!(A, P, Â).

The flags argument is a bitwise-or of FFTW planner flags, defaulting to FFTW.ESTIMATE. e.g. passing FFTW.MEASURE or FFTW.PATIENT will instead spend several seconds (or more) benchmarking different possible FFT algorithms and picking the fastest one; see the FFTW manual for more information on planner flags. The optional timelimit argument specifies a rough upper bound on the allowed planning time, in seconds. Passing FFTW.MEASURE or FFTW.PATIENT may cause the input array A to be overwritten with zeros during plan creation.

plan_fft! is the same as plan_fft but creates a plan that operates in-place on its argument (which must be an array of complex floating-point numbers). plan_ifft and so on are similar but produce plans that perform the equivalent of the inverse transforms ifft and so on.

source

Base.DFT.plan_ifft – Function.

```
plan_ifft(A [, dims]; flags=FFTW.ESTIMATE;  timelimit=Inf)
```

Same as plan_fft, but produces a plan that performs inverse transforms ifft.

source

Base.DFT.plan_bfft – Function.

```
plan_bfft(A [, dims]; flags=FFTW.ESTIMATE;  timelimit=Inf)
```

Same as plan_fft, but produces a plan that performs an unnormalized backwards transform bfft.

source

Base.DFT.plan_fft! – Function.

```
plan_fft!(A [, dims]; flags=FFTW.ESTIMATE;  timelimit=Inf)
```

Same as plan_fft, but operates in-place on A.

source

Base.DFT.plan_ifft! – Function.

```
plan_ifft!(A [, dims]; flags=FFTW.ESTIMATE;  timelimit=Inf)
```

Same as `plan_ifft`, but operates in-place on `A`.

source

`Base.DFT.plan_bfft!` – Function.

```
plan_bfft!(A [, dims]; flags=FFTW.ESTIMATE;  timelimit=Inf)
```

Same as `plan_bfft`, but operates in-place on `A`.

source

`Base.DFT.rfft` – Function.

```
rfft(A [, dims])
```

Multidimensional FFT of a real array `A`, exploiting the fact that the transform has conjugate symmetry in order to save roughly half the computational time and storage costs compared with `fft`. If `A` has size `(n_1, ..., n_d)`, the result has size `(div(n_1,2)+1, ..., n_d)`.

The optional `dims` argument specifies an iterable subset of one or more dimensions of `A` to transform, similar to `fft`. Instead of (roughly) halving the first dimension of `A` in the result, the `dims[1]` dimension is (roughly) halved in the same way.

source

`Base.DFT.irfft` – Function.

```
irfft(A, d [, dims])
```

Inverse of `rfft`: for a complex array `A`, gives the corresponding real array whose FFT yields `A` in the first half. As for `rfft`, `dims` is an optional subset of dimensions to transform, defaulting to `1:ndims(A)`.

`d` is the length of the transformed real array along the `dims[1]` dimension, which must satisfy `div(d,2)+1 == size(A,dims[1])`. (This parameter cannot be inferred from `size(A)` since both `2*size(A,dims[1])-2` as well as `2*size(A,dims[1])-1` are valid sizes for the transformed real array.)

source

`Base.DFT.brfft` – Function.

```
brfft(A, d [, dims])
```

Similar to `irfft` but computes an unnormalized inverse transform (similar to `bfft`), which must be divided by the product of the sizes of the transformed dimensions (of the real output array) in order to obtain the inverse transform.

source

`Base.DFT.plan_rfft` – Function.

```
plan_rfft(A [, dims]; flags=FFTW.ESTIMATE;  timelimit=Inf)
```

Pre-plan an optimized real-input FFT, similar to `plan_fft` except for `rfft` instead of `fft`. The first two arguments, and the size of the transformed result, are the same as for `rfft`.

source

Base.DFT.plan_brfft – Function.

```
plan_brfft(A, d [, dims]; flags=FFTW.ESTIMATE;  timelimit=Inf)
```

Pre-plan an optimized real-input unnormalized transform, similar to plan_rfft except for brfft instead of rfft. The first two arguments and the size of the transformed result, are the same as for brfft.

source

Base.DFT.plan_irfft – Function.

```
plan_irfft(A, d [, dims]; flags=FFTW.ESTIMATE;  timelimit=Inf)
```

Pre-plan an optimized inverse real-input FFT, similar to plan_rfft except for irfft and brfft, respectively. The first three arguments have the same meaning as for irfft.

source

Base.DFT.FFTW.dct – Function.

```
dct(A [, dims])
```

Performs a multidimensional type-II discrete cosine transform (DCT) of the array A, using the unitary normalization of the DCT. The optional dims argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. Most efficient if the size of A along the transformed dimensions is a product of small primes; see nextprod. See also plan_dct for even greater efficiency.

source

Base.DFT.FFTW.dct! – Function.

```
dct!(A [, dims])
```

Same as dct!, except that it operates in-place on A, which must be an array of real or complex floating-point values.

source

Base.DFT.FFTW.idct – Function.

```
idct(A [, dims])
```

Computes the multidimensional inverse discrete cosine transform (DCT) of the array A (technically, a type-III DCT with the unitary normalization). The optional dims argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. Most efficient if the size of A along the transformed dimensions is a product of small primes; see nextprod. See also plan_idct for even greater efficiency.

source

Base.DFT.FFTW.idct! – Function.

```
idct!(A [, dims])
```

Same as idct!, but operates in-place on A.

source

Base.DFT.FFTW.plan_dct – Function.

```
plan_dct(A [, dims [, flags [, timelimit]]])
```

Pre-plan an optimized discrete cosine transform (DCT), similar to `plan_fft` except producing a function that computes `dct`. The first two arguments have the same meaning as for `dct`.

source

`Base.DFT.FFTW.plan_dct!` – Function.

```
plan_dct!(A [, dims [, flags [, timelimit]]])
```

Same as `plan_dct`, but operates in-place on `A`.

source

`Base.DFT.FFTW.plan_idct` – Function.

```
plan_idct(A [, dims [, flags [, timelimit]]])
```

Pre-plan an optimized inverse discrete cosine transform (DCT), similar to `plan_fft` except producing a function that computes `idct`. The first two arguments have the same meaning as for `idct`.

source

`Base.DFT.FFTW.plan_idct!` – Function.

```
plan_idct!(A [, dims [, flags [, timelimit]]])
```

Same as `plan_idct`, but operates in-place on `A`.

source

`Base.DFT.fftshift` – Method.

```
fftshift(x)
```

Swap the first and second halves of each dimension of `x`.

source

`Base.DFT.fftshift` – Method.

```
fftshift(x,dim)
```

Swap the first and second halves of the given dimension or iterable of dimensions of array `x`.

source

`Base.DFT.ifftshift` – Function.

```
ifftshift(x, [dim])
```

Undoes the effect of `fftshift`.

source

`Base.DSP.filt` – Function.

```
filt(b, a, x, [si])
```

Apply filter described by vectors `a` and `b` to vector `x`, with an optional initial filter state vector `si` (defaults to zeros).

source

Base.DSP.filt! – Function.

```
filt!(out, b, a, x, [si])
```

Same as filt but writes the result into the out argument, which may alias the input x to modify it in-place.

source

Base.DSP.deconv – Function.

```
deconv(b,a) -> c
```

Construct vector c such that b = conv(a,c) + r. Equivalent to polynomial division.

source

Base.DSP.conv – Function.

```
conv(u,v)
```

Convolution of two vectors. Uses FFT algorithm.

source

Base.DSP.conv2 – Function.

```
conv2(u,v,A)
```

2-D convolution of the matrix A with the 2-D separable kernel generated by the vectors u and v. Uses 2-D FFT algorithm.

source

```
conv2(B,A)
```

2-D convolution of the matrix B with the matrix A. Uses 2-D FFT algorithm.

source

Base.DSP.xcorr – Function.

```
xcorr(u,v)
```

Compute the cross-correlation of two vectors.

source

Las siguientes funciones están definidas dentro del módulo Base.FFTW.

Base.DFT.FFTW.r2r – Function.

```
r2r(A, kind [, dims])
```

Performs a multidimensional real-input/real-output (r2r) transform of type kind of the array A, as defined in the FFTW manual. kind specifies either a discrete cosine transform of various types (FFTW.REDFT00, FFTW.REDFT01, FFTW.REDFT10, or FFTW.REDFT11), a discrete sine transform of various types (FFTW.RODFT00, FFTW.RODFT01, FFTW.RODFT10, or FFTW.RODFT11), a real-input DFT with halfcomplex-format output (FFTW.R2HC and its inverse FFTW.HC2R), or a discrete Hartley transform (FFTW.DHT). The kind argument may be an array or tuple in order to specify different transform types along the different dimensions of A; kind[end] is used for any unspecified dimensions. See the FFTW manual for precise definitions of these transform types, at http://www.fftw.org/doc.

The optional `dims` argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. `kind[i]` is then the transform type for `dims[i]`, with `kind[end]` being used for `i > length(kind)`.

See also `plan_r2r` to pre-plan optimized r2r transforms.

source

`Base.DFT.FFTW.r2r!` – Function.

```
r2r!(A, kind [, dims])
```

Same as `r2r`, but operates in-place on `A`, which must be an array of real or complex floating-point numbers.

source

`Base.DFT.FFTW.plan_r2r` – Function.

```
plan_r2r(A, kind [, dims [, flags [, timelimit]]])
```

Pre-plan an optimized r2r transform, similar to `plan_fft` except that the transforms (and the first three arguments) correspond to `r2r` and `r2r!`, respectively.

source

`Base.DFT.FFTW.plan_r2r!` – Function.

```
plan_r2r!(A, kind [, dims [, flags [, timelimit]]])
```

Similar to `plan_fft`, but corresponds to `r2r!`.

source

Chapter 48

Números

48.1 Tipos Numéricos Estándar

Tipos Numéricos Abstractos

`Core.Number` – Type.

```
Number
```

Abstract supertype for all number types.

source

`Core.Real` – Type.

```
Real <: Number
```

Abstract supertype for all real numbers.

source

`Core.AbstractFloat` – Type.

```
AbstractFloat <: Real
```

Abstract supertype for all floating point numbers.

source

`Core.Integer` – Type.

```
Integer <: Real
```

Abstract supertype for all integers.

source

`Core.Signed` – Type.

```
Signed <: Integer
```

Abstract supertype for all signed integers.

source

Core.Unsigned – Type.

```
Unsigned <: Integer
```

Abstract supertype for all unsigned integers.

source

Tipos Numéricos Concretos

Core.Float16 – Type.

```
Float16 <: AbstractFloat
```

16-bit floating point number type.

source

Core.Float32 – Type.

```
Float32 <: AbstractFloat
```

32-bit floating point number type.

source

Core.Float64 – Type.

```
Float64 <: AbstractFloat
```

64-bit floating point number type.

source

Base.MPFR.BigFloat – Type.

```
BigFloat <: AbstractFloat
```

Arbitrary precision floating point number type.

source

Core.Bool – Type.

```
Bool <: Integer
```

Boolean type.

source

Core.Int8 – Type.

```
Int8 <: Signed
```

8-bit signed integer type.

source

Core.UInt8 – Type.

```
UInt8 <: Unsigned
```

8-bit unsigned integer type.

source

Core.Int16 – Type.

```
Int16 <: Signed
```

16-bit signed integer type.

source

Core.UInt16 – Type.

```
UInt16 <: Unsigned
```

16-bit unsigned integer type.

source

Core.Int32 – Type.

```
Int32 <: Signed
```

32-bit signed integer type.

source

Core.UInt32 – Type.

```
UInt32 <: Unsigned
```

32-bit unsigned integer type.

source

Core.Int64 – Type.

```
Int64 <: Signed
```

64-bit signed integer type.

source

Core.UInt64 – Type.

```
UInt64 <: Unsigned
```

64-bit unsigned integer type.

source

Core.Int128 – Type.

```
Int128 <: Signed
```

128-bit signed integer type.

source

Core.UInt128 – Type.

```
UInt128 <: Unsigned
```

128-bit unsigned integer type.

source

Base.GMP.BigInt – Type.

```
BigInt <: Integer
```

Arbitrary precision integer type.

source

Base.Complex – Type.

```
Complex{T<:Real} <: Number
```

Complex number type with real and imaginary part of type T.

Complex32, Complex64 and Complex128 are aliases for Complex{Float16}, Complex{Float32} and Complex{Float64} respectively.

source

Base.Rational – Type.

```
Rational{T<:Integer} <: Real
```

Rational number type, with numerator and denominator of type T.

source

Base.Irrational – Type.

```
Irrational <: Real
```

Irrational number type.

source

48.2 Formatos de Datos

Base.bin – Function.

```
bin(n, pad::Int=1)
```

Convert an integer to a binary string, optionally specifying a number of digits to pad to.

```
julia> bin(10,2)
"1010"

julia> bin(10,8)
"00001010"
```

source

Base.hex – Function.

```
hex(n, pad::Int=1)
```

Convert an integer to a hexadecimal string, optionally specifying a number of digits to pad to.

```
julia> hex(20)
"14"

julia> hex(20, 3)
"014"
```

source

Base.dec – Function.

```
dec(n, pad::Int=1)
```

Convert an integer to a decimal string, optionally specifying a number of digits to pad to.

Examples

```
julia> dec(20)
"20"

julia> dec(20, 3)
"020"
```

source

Base.oct – Function.

```
oct(n, pad::Int=1)
```

Convert an integer to an octal string, optionally specifying a number of digits to pad to.

```
julia> oct(20)
"24"

julia> oct(20, 3)
"024"
```

source

Base.base – Function.

```
base(base::Integer, n::Integer, pad::Integer=1)
```

Convert an integer n to a string in the given base, optionally specifying a number of digits to pad to.

```
julia> base(13,5,4)
"0005"

julia> base(5,13,4)
"0023"
```

source

Base.digits – Function.

```
digits([T<:Integer], n::Integer, base::T=10, pad::Integer=1)
```

Returns an array with element type T (default `Int`) of the digits of n in the given base, optionally padded with zeros to a specified size. More significant digits are at higher indexes, such that `n == sum([digits[k]*base^(k-1) for k=1:length(digits)])`.

Examples

```
julia> digits(10, 10)
2-element Array{Int64,1}:
 0
 1

julia> digits(10, 2)
4-element Array{Int64,1}:
 0
 1
 0
 1

julia> digits(10, 2, 6)
6-element Array{Int64,1}:
 0
 1
 0
 1
 0
 0
```

source

`Base.digits!` – Function.

```
digits!(array, n::Integer, base::Integer=10)
```

Fills an array of the digits of n in the given base. More significant digits are at higher indexes. If the array length is insufficient, the least significant digits are filled up to the array length. If the array length is excessive, the excess portion is filled with zeros.

Examples

```
julia> digits!([2,2,2,2], 10, 2)
4-element Array{Int64,1}:
 0
 1
 0
 1

julia> digits!([2,2,2,2,2,2], 10, 2)
6-element Array{Int64,1}:
 0
 1
 0
 1
 0
 0
```

source

Base.bits – Function.

```
bits(n)
```

A string giving the literal bit representation of a number.

Example

```
julia> bits(4)
"0000000000000000000000000000000000000000000000000000000000000100"

julia> bits(2.2)
"0100000000000001100110011001100110011001100110011001100110011010"
```

source

Base.parse – Method.

```
parse(type, str, [base])
```

Parse a string as a number. If the type is an integer type, then a base can be specified (the default is 10). If the type is a floating point type, the string is parsed as a decimal floating point number. If the string does not contain a valid number, an error is raised.

```
julia> parse(Int, "1234")
1234

julia> parse(Int, "1234", 5)
194

julia> parse(Int, "afc", 16)
2812

julia> parse(Float64, "1.2e-3")
0.0012
```

source

Base.tryparse – Function.

```
tryparse(type, str, [base])
```

Like parse, but returns a Nullable of the requested type. The result will be null if the string does not contain a valid number.

source

Base.big – Function.

```
big(x)
```

Convert a number to a maximum precision representation (typically BigInt or BigFloat). See BigFloat for information about some pitfalls with floating-point numbers.

source

`Base.signed` – Function.

```
signed(x)
```

Convert a number to a signed integer. If the argument is unsigned, it is reinterpreted as signed without checking for overflow.

source

`Base.unsigned` – Function.

```
unsigned(x) -> Unsigned
```

Convert a number to an unsigned integer. If the argument is signed, it is reinterpreted as unsigned without checking for negative values.

Examples

```
julia> unsigned(-2)
0xfffffffffffffffe

julia> unsigned(2)
0x0000000000000002

julia> signed(unsigned(-2))
-2
```

source

`Base.float` – Method.

```
float(x)
```

Convert a number or array to a floating point data type. When passed a string, this function is equivalent to `parse(Float64, x)`.

source

`Base.Math.significand` – Function.

```
significand(x)
```

Extract the `significand(s)` (a.k.a. mantissa), in binary representation, of a floating-point number. If x is a non-zero finite number, then the result will be a number of the same type on the interval $[1, 2)$. Otherwise x is returned.

Examples

```
julia> significand(15.2)/15.2
0.125

julia> significand(15.2)*8
15.2
```

source

`Base.Math.exponent` – Function.

```
exponent(x) -> Int
```

Get the exponent of a normalized floating-point number.

source

Base.complex – Method.

```
complex(r, [i])
```

Convert real numbers or arrays to complex. `i` defaults to zero.

source

Base.bswap – Function.

```
bswap(n)
```

Byte-swap an integer. Flip the bits of its binary representation.

Examples

```
julia> a = bswap(4)
288230376151711744

julia> bswap(a)
4

julia> bin(1)
"1"

julia> bin(bswap(1))
"1000000000000000000000000000000000000000000000000000000000000000"
```

source

Base.num2hex – Function.

```
num2hex(f)
```

Get a hexadecimal string of the binary representation of a floating point number.

Example

```
julia> num2hex(2.2)
"400199999999999a"
```

source

Base.hex2num – Function.

```
hex2num(str)
```

Convert a hexadecimal string to the floating point number it represents.

source

Base.hex2bytes – Function.

```
hex2bytes(s::AbstractString)
```

Convert an arbitrarily long hexadecimal string to its binary representation. Returns an `Array{UInt8,1}`, i.e. an array of bytes.

```
julia> a = hex(12345)
"3039"

julia> hex2bytes(a)
2-element Array{UInt8,1}:
 0x30
 0x39
```

source

`Base.bytes2hex` – Function.

```
bytes2hex(bin_arr::Array{UInt8, 1}) -> String
```

Convert an array of bytes to its hexadecimal representation. All characters are in lower-case.

```
julia> a = hex(12345)
"3039"

julia> b = hex2bytes(a)
2-element Array{UInt8,1}:
 0x30
 0x39

julia> bytes2hex(b)
"3039"
```

source

48.3 Constantes y Funciones de Números Generales

`Base.one` – Function.

```
one(x)
one(T::type)
```

Return a multiplicative identity for `x`: a value such that `one(x)*x == x*one(x) == x`. Alternatively `one(T)` can take a type `T`, in which case `one` returns a multiplicative identity for any `x` of type `T`.

If possible, `one(x)` returns a value of the same type as `x`, and `one(T)` returns a value of type `T`. However, this may not be the case for types representing dimensionful quantities (e.g. time in days), since the multiplicative identity must be dimensionless. In that case, `one(x)` should return an identity value of the same precision (and shape, for matrices) as `x`.

If you want a quantity that is of the same type as `x`, or of type `T`, even if `x` is dimensionful, use `oneunit` instead.

```
julia> one(3.7)
1.0

julia> one(Int)
1
```

```
julia> one(Dates.Day(1))
1
```

source

Base.oneunit – Function.

```
oneunit(x::T)
oneunit(T::Type)
```

Returns T(one(x)), where T is either the type of the argument or (if a type is passed) the argument. This differs from one for dimensionful quantities: one is dimensionless (a multiplicative identity) while oneunit is dimensionful (of the same type as x, or of type T).

```
julia> oneunit(3.7)
1.0

julia> oneunit(Dates.Day)
1 day
```

source

Base.zero – Function.

```
zero(x)
```

Get the additive identity element for the type of x (x can also specify the type itself).

```
julia> zero(1)
0

julia> zero(big"2.0")
0.0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

julia> zero(rand(2,2))
2×2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0
```

source

Base.pi – Constant.

```
pi
π
```

The constant pi.

```
julia> pi
π = 3.1415926535897...
```

source

Base.im – Constant.

```
im
```

The imaginary unit.

source

Base.eu – Constant.

```
e
eu
```

The constant e.

```
julia> e
e = 2.7182818284590...
```

source

Base.catalan – Constant.

```
catalan
```

Catalan's constant.

```
julia> catalan
catalan = 0.9159655941772...
```

source

Base.eulergamma – Constant.

```
γ

eulergamma
```

Euler's constant.

```
julia> eulergamma
γ = 0.5772156649015...
```

source

Base.golden – Constant.

```
φ

golden
```

The golden ratio.

```
julia> golden
φ = 1.6180339887498...
```

source

Base.Inf – Constant.

```
Inf
```

Positive infinity of type Float64.

source

Base.Inf32 – Constant.

```
Inf32
```

Positive infinity of type Float32.

source

Base.Inf16 – Constant.

```
Inf16
```

Positive infinity of type Float16.

source

Base.NaN – Constant.

```
NaN
```

A not-a-number value of type Float64.

source

Base.NaN32 – Constant.

```
NaN32
```

A not-a-number value of type Float32.

source

Base.NaN16 – Constant.

```
NaN16
```

A not-a-number value of type Float16.

source

Base.issubnormal – Function.

```
issubnormal(f) -> Bool
```

Test whether a floating point number is subnormal.

source

Base.isfinite – Function.

```
isfinite(f) -> Bool
```

Test whether a number is finite.

```
julia> isfinite(5)
true

julia> isfinite(NaN32)
false
```

source

Base.isinf – Function.

```
isinf(f) -> Bool
```

Test whether a number is infinite.

source

Base.isnan – Function.

```
isnan(f) -> Bool
```

Test whether a floating point number is not a number (NaN).

source

Base.iszero – Function.

```
iszero(x)
```

Return `true` if `x == zero(x)`; if x is an array, this checks whether all of the elements of x are zero.

source

Base.nextfloat – Function.

```
nextfloat(x::AbstractFloat, n::Integer)
```

The result of n iterative applications of `nextfloat` to x if `n >= 0`, or `-n` applications of `prevfloat` if `n < 0`.

source

```
nextfloat(x::AbstractFloat)
```

Returns the smallest floating point number y of the same type as x such `x < y`. If no such y exists (e.g. if x is `Inf` or `NaN`), then returns x.

source

Base.prevfloat – Function.

```
prevfloat(x::AbstractFloat)
```

Returns the largest floating point number y of the same type as x such `y < x`. If no such y exists (e.g. if x is `-Inf` or `NaN`), then returns x.

source

Base.isinteger – Function.

```
isinteger(x) -> Bool
```

Test whether x is numerically equal to some integer.

```
julia> isinteger(4.0)
true
```

source

Base.isreal – Function.

```
isreal(x) -> Bool
```

Test whether x or all its elements are numerically equal to some real number.

```
julia> isreal(5.)
true

julia> isreal([4.; complex(0,1)])
false
```

source

Core.Float32 – Method.

```
Float32(x [, mode::RoundingMode])
```

Create a Float32 from x. If x is not exactly representable then mode determines how x is rounded.

Examples

```
julia> Float32(1/3, RoundDown)
0.3333333f0

julia> Float32(1/3, RoundUp)
0.33333334f0
```

See RoundingMode for available rounding modes.

source

Core.Float64 – Method.

```
Float64(x [, mode::RoundingMode])
```

Create a Float64 from x. If x is not exactly representable then mode determines how x is rounded.

Examples

```
julia> Float64(pi, RoundDown)
3.141592653589793

julia> Float64(pi, RoundUp)
3.1415926535897936
```

See RoundingMode for available rounding modes.

source

Base.GMP.BigInt – Method.

```
BigInt(x)
```

Create an arbitrary precision integer. x may be an Int (or anything that can be converted to an Int). The usual mathematical operators are defined for this type, and results are promoted to a BigInt.

Instances can be constructed from strings via parse, or using the big string literal.

```
julia> parse(BigInt, "42")
42

julia> big"313"
313
```

source

Base.MPFR.BigFloat – Method.

```
BigFloat(x)
```

Create an arbitrary precision floating point number. x may be an Integer, a Float64 or a BigInt. The usual mathematical operators are defined for this type, and results are promoted to a BigFloat.

Note that because decimal literals are converted to floating point numbers when parsed, BigFloat(2.1) may not yield what you expect. You may instead prefer to initialize constants from strings via parse, or using the big string literal.

```
julia> BigFloat(2.1)
2.100000000000000088817841970012523233890533447265625000000000000000000000000000

julia> big"2.1"
2.099999999999999999999999999999999999999999999999999999999999999999999999999999986
```

source

Base.Rounding.rounding – Function.

```
rounding(T)
```

Get the current floating point rounding mode for type T, controlling the rounding of basic arithmetic functions (+, -, *, / and sqrt) and type conversion.

See RoundingMode for available modes.

source

Base.Rounding.setrounding – Method.

```
setrounding(T, mode)
```

Set the rounding mode of floating point type T, controlling the rounding of basic arithmetic functions (+, -, *, / and sqrt) and type conversion. Other numerical functions may give incorrect or invalid values when using rounding modes other than the default RoundNearest.

Note that this may affect other types, for instance changing the rounding mode of Float64 will change the rounding mode of Float32. See RoundingMode for available modes.

> **Warning**
> This feature is still experimental, and may give unexpected or incorrect values.

source

Base.Rounding.setrounding – Method.

```
setrounding(f::Function, T, mode)
```

Change the rounding mode of floating point type T for the duration of `f`. It is logically equivalent to:

```
old = rounding(T)
setrounding(T, mode)
f()
setrounding(T, old)
```

See `RoundingMode` for available rounding modes.

Warning

This feature is still experimental, and may give unexpected or incorrect values. A known problem is the interaction with compiler optimisations, e.g.

```
julia> setrounding(Float64,RoundDown) do
           1.1 + 0.1
       end
1.2000000000000002
```

Here the compiler is *constant folding*, that is evaluating a known constant expression at compile time, however the rounding mode is only changed at runtime, so this is not reflected in the function result. This can be avoided by moving constants outside the expression, e.g.

```
julia> x = 1.1; y = 0.1;

julia> setrounding(Float64,RoundDown) do
           x + y
       end
1.2
```

source

`Base.Rounding.get_zero_subnormals` – Function.

```
get_zero_subnormals() -> Bool
```

Returns `false` if operations on subnormal floating-point values ("denormals") obey rules for IEEE arithmetic, and `true` if they might be converted to zeros.

source

`Base.Rounding.set_zero_subnormals` – Function.

```
set_zero_subnormals(yes::Bool) -> Bool
```

If `yes` is `false`, subsequent floating-point operations follow rules for IEEE arithmetic on subnormal values ("denormals"). Otherwise, floating-point operations are permitted (but not required) to convert subnormal inputs or outputs to zero. Returns `true` unless `yes==true` but the hardware does not support zeroing of subnormal numbers.

`set_zero_subnormals(true)` can speed up some computations on some hardware. However, it can break identities such as `(x-y==0) == (x==y)`.

source

Enteros

Base.count_ones – Function.

```
count_ones(x::Integer) -> Integer
```

Number of ones in the binary representation of x.

```
julia> count_ones(7)
3
```

source

Base.count_zeros – Function.

```
count_zeros(x::Integer) -> Integer
```

Number of zeros in the binary representation of x.

```
julia> count_zeros(Int32(2 ^ 16 - 1))
16
```

source

Base.leading_zeros – Function.

```
leading_zeros(x::Integer) -> Integer
```

Number of zeros leading the binary representation of x.

```
julia> leading_zeros(Int32(1))
31
```

source

Base.leading_ones – Function.

```
leading_ones(x::Integer) -> Integer
```

Number of ones leading the binary representation of x.

```
julia> leading_ones(UInt32(2 ^ 32 - 2))
31
```

source

Base.trailing_zeros – Function.

```
trailing_zeros(x::Integer) -> Integer
```

Number of zeros trailing the binary representation of x.

```
julia> trailing_zeros(2)
1
```

source

Base.trailing_ones – Function.

```
trailing_ones(x::Integer) -> Integer
```

Number of ones trailing the binary representation of x.

```
julia> trailing_ones(3)
2
```

source

Base.isodd – Function.

```
isodd(x::Integer) -> Bool
```

Returns `true` if x is odd (that is, not divisible by 2), and `false` otherwise.

```
julia> isodd(9)
true

julia> isodd(10)
false
```

source

Base.iseven – Function.

```
iseven(x::Integer) -> Bool
```

Returns `true` is x is even (that is, divisible by 2), and `false` otherwise.

```
julia> iseven(9)
false

julia> iseven(10)
true
```

source

48.4 BigFloats

El tipo `BigFloat` implementa el punto flotante de precisión arbitraria usando la librería GNU MPFR library.

Base.precision – Function.

```
precision(num::AbstractFloat)
```

Get the precision of a floating point number, as defined by the effective number of bits in the mantissa.

source

Base.precision – Method.

```
precision(BigFloat)
```

Get the precision (in bits) currently used for `BigFloat` arithmetic.

source

Base.MPFR.setprecision – Function.

```
setprecision([T=BigFloat,] precision::Int)
```

Set the precision (in bits) to be used for T arithmetic.

source

```
setprecision(f::Function, [T=BigFloat,] precision::Integer)
```

Change the T arithmetic precision (in bits) for the duration of `f`. It is logically equivalent to:

```
old = precision(BigFloat)
setprecision(BigFloat, precision)
f()
setprecision(BigFloat, old)
```

Often used as `setprecision(T, precision) do ... end`

source

Base.MPFR.BigFloat – Method.

```
BigFloat(x, prec::Int)
```

Create a representation of `x` as a `BigFloat` with precision `prec`.

source

Base.MPFR.BigFloat – Method.

```
BigFloat(x, rounding::RoundingMode)
```

Create a representation of `x` as a `BigFloat` with the current global precision and rounding mode `rounding`.

source

Base.MPFR.BigFloat – Method.

```
BigFloat(x, prec::Int, rounding::RoundingMode)
```

Create a representation of `x` as a `BigFloat` with precision `prec` and rounding mode `rounding`.

source

Base.MPFR.BigFloat – Method.

```
BigFloat(x::String)
```

Create a representation of the string `x` as a `BigFloat`.

source

48.5 Números Aleatorios

La generación de números aleatorios en Julia utiliza la librería Mersenne Twister a través de objetos `MersenneTwister`. Julia tiene un RNG global que es usado por defecto. Pueden conectarse otros tipos RNG heredando del tipo `AbstractRNG`; estos pueden ser usados entonces para tener múltiples flujos de números aleatorios. Ademas de `MersenneTwister`, Julia proporciona el tipo RNG `RandomDevice` que es un *wrapper* sobre la entropía proporcionada por el SO.

La mayoría de las funciones relacionadas con la generación aleatoria aceptan un `AbstractRNG` opcional como primer argumento,`rng`, que se predetermina al global si no se proporciona. Además, algunos de ellos aceptan opcionalmente especificaciones de dimensión `dims ...` (que pueden darse como una tupla) para generar matrices de valores aleatorios.

Un RNG de tipo `MersenneTwister` o `RandomDevice` puede generar números aleatorios de los siguientes tipos: `Float16`, `Float32`, `Float64`, `Bool`, `Int8`, `UInt8`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Int128`, `UInt128`, `BigInt` (o números complejos de estos tipos). Los números aleatorios en punto flotante son generados uniformemente en $[0, 1)$. Como `BigInt` representa números sin límite, el intervalo debe ser especificado (por ejemplo, `rand(big(1:6))`).

`Base.Random.srand` – Function.

```
srand([rng=GLOBAL_RNG], [seed]) -> rng
srand([rng=GLOBAL_RNG], filename, n=4) -> rng
```

Reseed the random number generator. If a `seed` is provided, the RNG will give a reproducible sequence of numbers, otherwise Julia will get entropy from the system. For `MersenneTwister`, the `seed` may be a non-negative integer, a vector of `UInt32` integers or a filename, in which case the seed is read from a file (4n bytes are read from the file, where n is an optional argument). `RandomDevice` does not support seeding.

source

`Base.Random.MersenneTwister` – Type.

```
MersenneTwister(seed)
```

Create a `MersenneTwister` RNG object. Different RNG objects can have their own seeds, which may be useful for generating different streams of random numbers.

Example

```
julia> rng = MersenneTwister(1234);
```

source

`Base.Random.RandomDevice` – Type.

```
RandomDevice()
```

Create a `RandomDevice` RNG object. Two such objects will always generate different streams of random numbers.

source

`Base.Random.rand` – Function.

```
rand([rng=GLOBAL_RNG], [S], [dims...])
```

Pick a random element or array of random elements from the set of values specified by S; S can be

- an indexable collection (for example `1:n` or `['x','y','z']`), or
- a type: the set of values to pick from is then equivalent to `typemin(S):typemax(S)` for integers (this is not applicable to `BigInt`), and to $[0, 1)$ for floating point numbers;

S defaults to `Float64`.

source

`Base.Random.rand!` – Function.

```
rand!([rng=GLOBAL_RNG], A, [coll])
```

Populate the array `A` with random values. If the indexable collection `coll` is specified, the values are picked randomly from `coll`. This is equivalent to `copy!(A, rand(rng, coll, size(A)))` or `copy!(A, rand(rng, eltype(A), size(A)))` but without allocating a new array.

Example

```
julia> rng = MersenneTwister(1234);

julia> rand!(rng, zeros(5))
5-element Array{Float64,1}:
 0.590845
 0.766797
 0.566237
 0.460085
 0.794026
```

source

`Base.Random.bitrand` – Function.

```
bitrand([rng=GLOBAL_RNG], [dims...])
```

Generate a `BitArray` of random boolean values.

Example

```
julia> rng = MersenneTwister(1234);

julia> bitrand(rng, 10)
10-element BitArray{1}:
  true
  true
  true
 false
  true
 false
 false
  true
 false
  true
```

source

`Base.Random.randn` – Function.

```
randn([rng=GLOBAL_RNG], [T=Float64], [dims...])
```

Generate a normally-distributed random number of type `T` with mean 0 and standard deviation 1. Optionally generate an array of normally-distributed random numbers. The `Base` module currently provides an implementation for the types `Float16`, `Float32`, and `Float64` (the default).

Examples

```
julia> rng = MersenneTwister(1234);

julia> randn(rng, Float64)
0.8673472019512456

julia> randn(rng, Float32, (2, 4))
2×4 Array{Float32,2}:
 -0.901744  -0.902914  2.21188   -0.271735
 -0.494479   0.864401  0.532813   0.502334
```

source

Base.Random.randn! – Function.

```
randn!([rng=GLOBAL_RNG], A::AbstractArray) -> A
```

Fill the array A with normally-distributed (mean 0, standard deviation 1) random numbers. Also see the rand function.

Example

```
julia> rng = MersenneTwister(1234);

julia> randn!(rng, zeros(5))
5-element Array{Float64,1}:
  0.867347
 -0.901744
 -0.494479
 -0.902914
  0.864401
```

source

Base.Random.randexp – Function.

```
randexp([rng=GLOBAL_RNG], [T=Float64], [dims...])
```

Generate a random number of type T according to the exponential distribution with scale 1. Optionally generate an array of such random numbers. The Base module currently provides an implementation for the types Float16, Float32, and Float64 (the default).

Examples

```
julia> rng = MersenneTwister(1234);

julia> randexp(rng, Float32)
2.4835055f0

julia> randexp(rng, 3, 3)
3×3 Array{Float64,2}:
 1.5167    1.30652   0.344435
 0.604436  2.78029   0.418516
 0.695867  0.693292  0.643644
```

source

Base.Random.randexp! – Function.

```
randexp!([rng=GLOBAL_RNG], A::AbstractArray) -> A
```

Fill the array `A` with random numbers following the exponential distribution (with scale 1).

Example

```
julia> rng = MersenneTwister(1234);

julia> randexp!(rng, zeros(5))
5-element Array{Float64,1}:
 2.48351
 1.5167
 0.604436
 0.695867
 1.30652
```

source

`Base.Random.randjump` – Function.

```
randjump(r::MersenneTwister, jumps::Integer, [jumppoly::AbstractString=dSFMT.JPOLY1e21]) ->
    Vector{MersenneTwister}
```

Create an array of the size `jumps` of initialized `MersenneTwister` RNG objects. The first RNG object given as a parameter and following `MersenneTwister` RNGs in the array are initialized such that a state of the RNG object in the array would be moved forward (without generating numbers) from a previous RNG object array element on a particular number of steps encoded by the jump polynomial `jumppoly`.

Default jump polynomial moves forward `MersenneTwister` RNG state by `10^20` steps.

source

Chapter 49

Cadenas

`Base.length` – Method.

```
length(s::AbstractString)
```

The number of characters in string s.

Example

```
julia> length("jμΛIα")
5
```

source

`Base.sizeof` – Method.

```
sizeof(s::AbstractString)
```

The number of bytes in string s.

Example

```
julia> sizeof("")
3
```

source

`Base.:*` – Method.

```
*(x, y...)
```

Multiplication operator. `x*y*z*...` calls this function with all arguments, i.e. `*(x, y, z, ...)`.

source

`Base.:^` – Method.

```
^(s::AbstractString, n::Integer)
```

Repeat n times the string s. The `repeat` function is an alias to this operator.

```
julia> "Test "^3
"Test Test Test "
```

source

Base.string – Function.

```
string(xs...)
```

Create a string from any values using the print function.

```
julia> string("a", 1, true)
"a1true"
```

source

Base.repr – Function.

```
repr(x)
```

Create a string from any value using the showall function.

source

Core.String – Method.

```
String(s::AbstractString)
```

Convert a string to a contiguous byte array representation encoded as UTF-8 bytes. This representation is often appropriate for passing strings to C.

source

Base.transcode – Function.

```
transcode(T, src)
```

Convert string data between Unicode encodings. src is either a String or a Vector{UIntXX} of UTF-XX code units, where XX is 8, 16, or 32. T indicates the encoding of the return value: String to return a (UTF-8 encoded) String or UIntXX to return a Vector{UIntXX} of UTF-XX data. (The alias Cwchar_t can also be used as the integer type, for converting wchar_t* strings used by external C libraries.)

The transcode function succeeds as long as the input data can be reasonably represented in the target encoding; it always succeeds for conversions between UTF-XX encodings, even for invalid Unicode data.

Only conversion to/from UTF-8 is currently supported.

source

Base.unsafe_string – Function.

```
unsafe_string(p::Ptr{UInt8}, [length::Integer])
```

Copy a string from the address of a C-style (NUL-terminated) string encoded as UTF-8. (The pointer can be safely freed afterwards.) If length is specified (the length of the data in bytes), the string does not have to be NUL-terminated.

This function is labelled ”unsafe” because it will crash if p is not a valid memory address to data of the requested length.

source

Base.codeunit – Method.

```
codeunit(s::AbstractString, i::Integer)
```

Get the `i`th code unit of an encoded string. For example, returns the `i`th byte of the representation of a UTF-8 string.

source

`Base.ascii` – Function.

```
ascii(s::AbstractString)
```

Convert a string to `String` type and check that it contains only ASCII data, otherwise throwing an `ArgumentError` indicating the position of the first non-ASCII byte.

```
julia> ascii("abcdeγfgh")
ERROR: ArgumentError: invalid ASCII at index 6 in "abcdeγfgh"
Stacktrace:
 [1] ascii(::String) at ./strings/util.jl:479

julia> ascii("abcdefgh")
"abcdefgh"
```

source

`Base.@r_str` – Macro.

```
@r_str -> Regex
```

Construct a regex, such as `r"^[a-z]*$"`. The regex also accepts one or more flags, listed after the ending quote, to change its behaviour:

- `i` enables case-insensitive matching
- `m` treats the `^` and `$` tokens as matching the start and end of individual lines, as opposed to the whole string.
- `s` allows the `.` modifier to match newlines.
- `x` enables "comment mode": whitespace is enabled except when escaped with `\`, and `#` is treated as starting a comment.

For example, this regex has all three flags enabled:

```
julia> match(r"a+.*b+.*?d$"ism, "Goodbye,\nOh, angry,\nBad world\n")
RegexMatch("angry,\nBad world")
```

source

`Base.Docs.@html_str` – Macro.

```
@html_str -> Docs.HTML
```

Create an `HTML` object from a literal string.

source

`Base.Docs.@text_str` – Macro.

```
@text_str -> Docs.Text
```

Create a `Text` object from a literal string.

source

`Base.UTF8proc.normalize_string` – Function.

```
normalize_string(s::AbstractString, normalform::Symbol)
```

Normalize the string `s` according to one of the four "normal forms" of the Unicode standard: `normalform` can be `:NFC`, `:NFD`, `:NFKC`, or `:NFKD`. Normal forms C (canonical composition) and D (canonical decomposition) convert different visually identical representations of the same abstract string into a single canonical form, with form C being more compact. Normal forms KC and KD additionally canonicalize "compatibility equivalents": they convert characters that are abstractly similar but visually distinct into a single canonical choice (e.g. they expand ligatures into the individual characters), with form KC being more compact.

Alternatively, finer control and additional transformations may be be obtained by calling `normalize_string(s; keywords...)`, where any number of the following boolean keywords options (which all default to `false` except for `compose`) are specified:

- `compose=false`: do not perform canonical composition
- `decompose=true`: do canonical decomposition instead of canonical composition (`compose=true` is ignored if present)
- `compat=true`: compatibility equivalents are canonicalized
- `casefold=true`: perform Unicode case folding, e.g. for case-insensitive string comparison
- `newline2lf=true`, `newline2ls=true`, or `newline2ps=true`: convert various newline sequences (LF, CRLF, CR, NEL) into a linefeed (LF), line-separation (LS), or paragraph-separation (PS) character, respectively
- `stripmark=true`: strip diacritical marks (e.g. accents)
- `stripignore=true`: strip Unicode's "default ignorable" characters (e.g. the soft hyphen or the left-to-right marker)
- `stripcc=true`: strip control characters; horizontal tabs and form feeds are converted to spaces; newlines are also converted to spaces unless a newline-conversion flag was specified
- `rejectna=true`: throw an error if unassigned code points are found
- `stable=true`: enforce Unicode Versioning Stability

For example, NFKC corresponds to the options `compose=true, compat=true, stable=true`.

source

`Base.UTF8proc.graphemes` – Function.

```
graphemes(s::AbstractString) -> GraphemeIterator
```

Returns an iterator over substrings of `s` that correspond to the extended graphemes in the string, as defined by Unicode UAX #29. (Roughly, these are what users would perceive as single characters, even though they may contain more than one codepoint; for example a letter combined with an accent mark is a single grapheme.)

source

`Base.isvalid` – Method.

```
isvalid(value) -> Bool
```

Returns `true` if the given value is valid for its type, which currently can be either `Char` or `String`.

source

`Base.isvalid` – Method.

```
isvalid(T, value) -> Bool
```

Returns `true` if the given value is valid for that type. Types currently can be either `Char` or `String`. Values for `Char` can be of type `Char` or `UInt32`. Values for `String` can be of that type, or `Vector{UInt8}`.

source

`Base.isvalid` – Method.

```
isvalid(str::AbstractString, i::Integer)
```

Tells whether index `i` is valid for the given string.

Examples

```
julia> str = "αβγdef";

julia> isvalid(str, 1)
true

julia> str[1]
'α': Unicode U+03b1 (category Ll: Letter, lowercase)

julia> isvalid(str, 2)
false

julia> str[2]
ERROR: UnicodeError: invalid character index
[...]
```

source

`Base.UTF8proc.is_assigned_char` – Function.

```
is_assigned_char(c) -> Bool
```

Returns `true` if the given char or integer is an assigned Unicode code point.

source

`Base.ismatch` – Function.

```
ismatch(r::Regex, s::AbstractString) -> Bool
```

Test whether a string contains a match of the given regular expression.

source

`Base.match` – Function.

```
match(r::Regex, s::AbstractString[, idx::Integer[, addopts]])
```

Search for the first match of the regular expression `r` in `s` and return a `RegexMatch` object containing the match, or nothing if the match failed. The matching substring can be retrieved by accessing `m.match` and the captured sequences can be retrieved by accessing `m.captures` The optional `idx` argument specifies an index at which to start the search.

source

Base.eachmatch – Function.

```
eachmatch(r::Regex, s::AbstractString[, overlap::Bool=false])
```

Search for all matches of a the regular expression r in s and return a iterator over the matches. If overlap is true, the matching sequences are allowed to overlap indices in the original string, otherwise they must be from distinct character ranges.

source

Base.matchall – Function.

```
matchall(r::Regex, s::AbstractString[, overlap::Bool=false]) -> Vector{AbstractString}
```

Return a vector of the matching substrings from eachmatch.

source

Base.lpad – Function.

```
lpad(s, n::Integer, p::AbstractString=" ")
```

Make a string at least n columns wide when printed by padding s on the left with copies of p.

```
julia> lpad("March",10)
"     March"
```

source

Base.rpad – Function.

```
rpad(s, n::Integer, p::AbstractString=" ")
```

Make a string at least n columns wide when printed by padding s on the right with copies of p.

```
julia> rpad("March",20)
"March               "
```

source

Base.search – Function.

```
search(string::AbstractString, chars::Chars, [start::Integer])
```

Search for the first occurrence of the given characters within the given string. The second argument may be a single character, a vector or a set of characters, a string, or a regular expression (though regular expressions are only allowed on contiguous strings, such as ASCII or UTF-8 strings). The third argument optionally specifies a starting index. The return value is a range of indexes where the matching sequence is found, such that s[search(s,x)] == x:

search(string, "substring") = start:end such that string[start:end] == "substring", or 0:-1 if unmatched.

search(string, 'c') = index such that string[index] == 'c', or 0 if unmatched.

```
julia> search("Hello to the world", "z")
0:-1

julia> search("JuliaLang","Julia")
1:5
```

source

Base.rsearch – Function.

```
rsearch(s::AbstractString, chars::Chars, [start::Integer])
```

Similar to search, but returning the last occurrence of the given characters within the given string, searching in reverse from start.

```
julia> rsearch("aaabbb","b")
6:6
```

source

Base.searchindex – Function.

```
searchindex(s::AbstractString, substring, [start::Integer])
```

Similar to search, but return only the start index at which the substring is found, or 0 if it is not.

```
julia> searchindex("Hello to the world", "z")
0

julia> searchindex("JuliaLang","Julia")
1

julia> searchindex("JuliaLang","Lang")
6
```

source

Base.rsearchindex – Function.

```
rsearchindex(s::AbstractString, substring, [start::Integer])
```

Similar to rsearch, but return only the start index at which the substring is found, or 0 if it is not.

```
julia> rsearchindex("aaabbb","b")
6

julia> rsearchindex("aaabbb","a")
3
```

source

Base.contains – Method.

```
contains(haystack::AbstractString, needle::AbstractString)
```

Determine whether the second argument is a substring of the first.

```
julia> contains("JuliaLang is pretty cool!", "Julia")
true
```

source

Base.reverse – Method.

```
reverse(s::AbstractString) -> AbstractString
```

Reverses a string.

```
julia> reverse("JuliaLang")
"gnaLailuJ"
```

source

Base.replace – Function.

```
replace(string::AbstractString, pat, r[, n::Integer=0])
```

Search for the given pattern `pat`, and replace each occurrence with `r`. If `n` is provided, replace at most `n` occurrences. As with search, the second argument may be a single character, a vector or a set of characters, a string, or a regular expression. If `r` is a function, each occurrence is replaced with `r(s)` where `s` is the matched substring. If `pat` is a regular expression and `r` is a `SubstitutionString`, then capture group references in `r` are replaced with the corresponding matched text.

source

Base.split – Function.

```
split(s::AbstractString, [chars]; limit::Integer=0, keep::Bool=true)
```

Return an array of substrings by splitting the given string on occurrences of the given character delimiters, which may be specified in any of the formats allowed by `search`'s second argument (i.e. a single character, collection of characters, string, or regular expression). If `chars` is omitted, it defaults to the set of all space characters, and `keep` is taken to be `false`. The two keyword arguments are optional: they are a maximum size for the result and a flag determining whether empty fields should be kept in the result.

```
julia> a = "Ma.rch"
"Ma.rch"

julia> split(a,".")
2-element Array{SubString{String},1}:
 "Ma"
 "rch"
```

source

Base.rsplit – Function.

```
rsplit(s::AbstractString, [chars]; limit::Integer=0, keep::Bool=true)
```

Similar to `split`, but starting from the end of the string.

```
julia> a = "M.a.r.c.h"
"M.a.r.c.h"

julia> rsplit(a,".")
5-element Array{SubString{String},1}:
 "M"
 "a"
 "r"
 "c"
```

```
 "h"

julia> rsplit(a,".";limit=1)
1-element Array{SubString{String},1}:
 "M.a.r.c.h"

julia> rsplit(a,".";limit=2)
2-element Array{SubString{String},1}:
 "M.a.r.c"
 "h"
```

source

Base.strip – Function.

```
strip(s::AbstractString, [chars::Chars])
```

Return s with any leading and trailing whitespace removed. If chars (a character, or vector or set of characters) is provided, instead remove characters contained in it.

```
julia> strip("{3, 5}\n", ['{', '}', '\n'])
"3, 5"
```

source

Base.lstrip – Function.

```
lstrip(s::AbstractString[, chars::Chars])
```

Return s with any leading whitespace and delimiters removed. The default delimiters to remove are ' ', \t, \n, \v, \f, and \r. If chars (a character, or vector or set of characters) is provided, instead remove characters contained in it.

```
julia> a = lpad("March", 20)
"               March"

julia> lstrip(a)
"March"
```

source

Base.rstrip – Function.

```
rstrip(s::AbstractString[, chars::Chars])
```

Return s with any trailing whitespace and delimiters removed. The default delimiters to remove are ' ', \t, \n, \v, \f, and \r. If chars (a character, or vector or set of characters) is provided, instead remove characters contained in it.

```
julia> a = rpad("March", 20)
"March               "

julia> rstrip(a)
"March"
```

source

Base.startswith – Function.

```
startswith(s::AbstractString, prefix::AbstractString)
```

Returns `true` if `s` starts with `prefix`. If `prefix` is a vector or set of characters, tests whether the first character of `s` belongs to that set.

See also `endswith`.

```
julia> startswith("JuliaLang", "Julia")
true
```

source

Base.endswith – Function.

```
endswith(s::AbstractString, suffix::AbstractString)
```

Returns `true` if `s` ends with `suffix`. If `suffix` is a vector or set of characters, tests whether the last character of `s` belongs to that set.

See also `startswith`.

```
julia> endswith("Sunday", "day")
true
```

source

Base.uppercase – Function.

```
uppercase(s::AbstractString)
```

Returns `s` with all characters converted to uppercase.

Example

```
julia> uppercase("Julia")
"JULIA"
```

source

Base.lowercase – Function.

```
lowercase(s::AbstractString)
```

Returns `s` with all characters converted to lowercase.

Example

```
julia> lowercase("STRINGS AND THINGS")
"strings and things"
```

source

Base.titlecase – Function.

```
titlecase(s::AbstractString)
```

Capitalizes the first character of each word in s.

Example

```
julia> titlecase("the julia programming language")
"The Julia Programming Language"
```

source

`Base.ucfirst` – Function.

```
ucfirst(s::AbstractString)
```

Returns `string` with the first character converted to uppercase.

Example

```
julia> ucfirst("python")
"Python"
```

source

`Base.lcfirst` – Function.

```
lcfirst(s::AbstractString)
```

Returns `string` with the first character converted to lowercase.

Example

```
julia> lcfirst("Julia")
"julia"
```

source

`Base.join` – Function.

```
join(io::IO, strings, delim, [last])
```

Join an array of `strings` into a single string, inserting the given delimiter between adjacent strings. If `last` is given, it will be used instead of `delim` between the last two strings. For example,

```
julia> join(["apples", "bananas", "pineapples"], ", ", " and ")
"apples, bananas and pineapples"
```

`strings` can be any iterable over elements `x` which are convertible to strings via `print(io::IOBuffer, x)`. `strings` will be printed to `io`.

source

`Base.chop` – Function.

```
chop(s::AbstractString)
```

Remove the last character from s.

```
julia> a = "March"
"March"

julia> chop(a)
"Marc"
```

source

Base.chomp – Function.

```
chomp(s::AbstractString)
```

Remove a single trailing newline from a string.

```
julia> chomp("Hello\n")
"Hello"
```

source

Base.ind2chr – Function.

```
ind2chr(s::AbstractString, i::Integer)
```

Convert a byte index `i` to a character index with respect to string `s`.

See also chr2ind.

Example

```
julia> str = "αβγdef";

julia> ind2chr(str, 3)
2

julia> chr2ind(str, 2)
3
```

source

Base.chr2ind – Function.

```
chr2ind(s::AbstractString, i::Integer)
```

Convert a character index `i` to a byte index.

See also ind2chr.

Example

```
julia> str = "αβγdef";

julia> chr2ind(str, 2)
3

julia> ind2chr(str, 3)
2
```

source

`Base.nextind` – Function.

```
nextind(str::AbstractString, i::Integer)
```

Get the next valid string index after `i`. Returns a value greater than `endof(str)` at or after the end of the string.

Examples

```
julia> str = "αβγdef";

julia> nextind(str, 1)
3

julia> endof(str)
9

julia> nextind(str, 9)
10
```

source

`Base.prevind` – Function.

```
prevind(str::AbstractString, i::Integer)
```

Get the previous valid string index before `i`. Returns a value less than 1 at the beginning of the string.

Examples

```
julia> prevind("αβγdef", 3)
1

julia> prevind("αβγdef", 1)
0
```

source

`Base.Random.randstring` – Function.

```
randstring([rng,] len=8)
```

Create a random ASCII string of length `len`, consisting of upper- and lower-case letters and the digits 0-9. The optional `rng` argument specifies a random number generator, see Random Numbers.

Example

```
julia> rng = MersenneTwister(1234);

julia> randstring(rng, 4)
"mbDd"
```

source

`Base.UTF8proc.charwidth` – Function.

```
charwidth(c)
```

Gives the number of columns needed to print a character.

source

Base.strwidth – Function.

```
strwidth(s::AbstractString)
```

Gives the number of columns needed to print a string.

Example

```
julia> strwidth("March")
5
```

source

Base.UTF8proc.isalnum – Function.

```
isalnum(c::Char) -> Bool
```

Tests whether a character is alphanumeric. A character is classified as alphabetic if it belongs to the Unicode general category Letter or Number, i.e. a character whose category code begins with 'L' or 'N'.

source

Base.UTF8proc.isalpha – Function.

```
isalpha(c::Char) -> Bool
```

Tests whether a character is alphabetic. A character is classified as alphabetic if it belongs to the Unicode general category Letter, i.e. a character whose category code begins with 'L'.

source

Base.isascii – Function.

```
isascii(c::Union{Char,AbstractString}) -> Bool
```

Tests whether a character belongs to the ASCII character set, or whether this is true for all elements of a string.

source

Base.UTF8proc.iscntrl – Function.

```
iscntrl(c::Char) -> Bool
```

Tests whether a character is a control character. Control characters are the non-printing characters of the Latin-1 subset of Unicode.

source

Base.UTF8proc.isdigit – Function.

```
isdigit(c::Char) -> Bool
```

Tests whether a character is a numeric digit (0-9).

source

Base.UTF8proc.isgraph – Function.

```
isgraph(c::Char) -> Bool
```

Tests whether a character is printable, and not a space. Any character that would cause a printer to use ink should be classified with `isgraph(c)==true`.

source

`Base.UTF8proc.islower` – Function.

```
islower(c::Char) -> Bool
```

Tests whether a character is a lowercase letter. A character is classified as lowercase if it belongs to Unicode category Ll, Letter: Lowercase.

source

`Base.UTF8proc.isnumber` – Function.

```
isnumber(c::Char) -> Bool
```

Tests whether a character is numeric. A character is classified as numeric if it belongs to the Unicode general category Number, i.e. a character whose category code begins with 'N'.

source

`Base.UTF8proc.isprint` – Function.

```
isprint(c::Char) -> Bool
```

Tests whether a character is printable, including spaces, but not a control character.

source

`Base.UTF8proc.ispunct` – Function.

```
ispunct(c::Char) -> Bool
```

Tests whether a character belongs to the Unicode general category Punctuation, i.e. a character whose category code begins with 'P'.

source

`Base.UTF8proc.isspace` – Function.

```
isspace(c::Char) -> Bool
```

Tests whether a character is any whitespace character. Includes ASCII characters '\t', '\n', '\v', '\f', '\r', and ' ', Latin-1 character U+0085, and characters in Unicode category Zs.

source

`Base.UTF8proc.isupper` – Function.

```
isupper(c::Char) -> Bool
```

Tests whether a character is an uppercase letter. A character is classified as uppercase if it belongs to Unicode category Lu, Letter: Uppercase, or Lt, Letter: Titlecase.

source

`Base.isxdigit` – Function.

```
isxdigit(c::Char) -> Bool
```

Tests whether a character is a valid hexadecimal digit. Note that this does not include x (as in the standard `0x` prefix).

Example

```
julia> isxdigit('a')
true

julia> isxdigit('x')
false
```

source

`Core.Symbol` – Type.

```
Symbol(x...) -> Symbol
```

Create a `Symbol` by concatenating the string representations of the arguments together.

source

`Base.escape_string` – Function.

```
escape_string([io,] str::AbstractString[, esc::AbstractString]) -> AbstractString
```

General escaping of traditional C and Unicode escape sequences. Any characters in `esc` are also escaped (with a backslash). See also `unescape_string`.

source

`Base.unescape_string` – Function.

```
unescape_string([io,] s::AbstractString) -> AbstractString
```

General unescaping of traditional C and Unicode escape sequences. Reverse of `escape_string`.

source

Chapter 50

Arrays

50.1 Constructores y Tipos

`Core.AbstractArray` – Type.

```
AbstractArray{T, N}
```

Abstract array supertype which arrays inherit from.

source

`Core.Array` – Type.

```
Array{T}(dims)
Array{T,N}(dims)
```

Construct an uninitialized N-dimensional dense array with element type T, where N is determined from the length or number of `dims`. `dims` may be a tuple or a series of integer arguments corresponding to the lengths in each dimension. If the rank N is supplied explicitly as in `Array{T,N}(dims)`, then it must match the length or number of `dims`.

Example

```
julia> A = Array{Float64, 2}(2, 2);

julia> ndims(A)
2

julia> eltype(A)
Float64
```

source

`Base.getindex` – Method.

```
getindex(type[, elements...])
```

Construct a 1-d array of the specified type. This is usually called with the syntax `Type[]`. Element values can be specified using `Type[a,b,c,...]`.

Example

```
julia> Int8[1, 2, 3]
3-element Array{Int8,1}:
 1
 2
 3

julia> getindex(Int8, 1, 2, 3)
3-element Array{Int8,1}:
 1
 2
 3
```

source

Base.zeros – Function.

```
zeros([A::AbstractArray,] [T=eltype(A)::Type,] [dims=size(A)::Tuple])
```

Create an array of all zeros with the same layout as `A`, element type `T` and size `dims`. The `A` argument can be skipped, which behaves like `Array{Float64,0}()` was passed. For convenience `dims` may also be passed in variadic form.

Examples

```
julia> zeros(1)
1-element Array{Float64,1}:
 0.0

julia> zeros(Int8, 2, 3)
2×3 Array{Int8,2}:
 0  0  0
 0  0  0

julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> zeros(A)
2×2 Array{Int64,2}:
 0  0
 0  0

julia> zeros(A, Float64)
2×2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0

julia> zeros(A, Bool, (3,))
3-element Array{Bool,1}:
 false
 false
 false
```

See also ones, similar.

source

Base.ones – Function.

```
ones([A::AbstractArray,] [T=eltype(A)::Type,] [dims=size(A)::Tuple])
```

Create an array of all ones with the same layout as `A`, element type `T` and size `dims`. The `A` argument can be skipped, which behaves like `Array{Float64,0}()` was passed. For convenience `dims` may also be passed in variadic form.

Examples

```
julia> ones(Complex128, 2, 3)
2×3 Array{Complex{Float64},2}:
 1.0+0.0im  1.0+0.0im  1.0+0.0im
 1.0+0.0im  1.0+0.0im  1.0+0.0im

julia> ones(1,2)
1×2 Array{Float64,2}:
 1.0  1.0

julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> ones(A)
2×2 Array{Int64,2}:
 1  1
 1  1

julia> ones(A, Float64)
2×2 Array{Float64,2}:
 1.0  1.0
 1.0  1.0

julia> ones(A, Bool, (3,))
3-element Array{Bool,1}:
 true
 true
 true
```

See also zeros, similar.

source

Base.BitArray – Type.

```
BitArray(dims::Integer...)
BitArray{N}(dims::NTuple{N,Int})
```

Construct an uninitialized `BitArray` with the given dimensions. Behaves identically to the Array constructor.

```
julia> BitArray(2, 2)
2×2 BitArray{2}:
 false  false
 false  true

julia> BitArray((3, 1))
```

```
3×1 BitArray{2}:
 false
 true
 false
```

source

```
BitArray(itr)
```

Construct a `BitArray` generated by the given iterable object. The shape is inferred from the `itr` object.

```
julia> BitArray([1 0; 0 1])
2×2 BitArray{2}:
  true  false
 false   true

julia> BitArray(x+y == 3 for x = 1:2, y = 1:3)
2×3 BitArray{2}:
 false   true  false
  true  false  false

julia> BitArray(x+y == 3 for x = 1:2 for y = 1:3)
6-element BitArray{1}:
 false
  true
 false
  true
 false
 false
```

source

`Base.trues` – Function.

```
trues(dims)
```

Create a `BitArray` with all values set to `true`.

```
julia> trues(2,3)
2×3 BitArray{2}:
 true  true  true
 true  true  true
```

source

```
trues(A)
```

Create a `BitArray` with all values set to `true` of the same shape as `A`.

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> trues(A)
2×2 BitArray{2}:
 true  true
 true  true
```

source

Base.falses – Function.

```
falses(dims)
```

Create a BitArray with all values set to false.

```
julia> falses(2,3)
2×3 BitArray{2}:
 false  false  false
 false  false  false
```

source

```
falses(A)
```

Create a BitArray with all values set to false of the same shape as A.

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> falses(A)
2×2 BitArray{2}:
 false  false
 false  false
```

source

Base.fill – Function.

```
fill(x, dims)
```

Create an array filled with the value x. For example, fill(1.0, (5,5)) returns a 5×5 array of floats, with each element initialized to 1.0.

```
julia> fill(1.0, (5,5))
5×5 Array{Float64,2}:
 1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0
```

If x is an object reference, all elements will refer to the same object. fill(Foo(), dims) will return an array filled with the result of evaluating Foo() once.

source

Base.fill! – Function.

```
fill!(A, x)
```

Fill array A with the value x. If x is an object reference, all elements will refer to the same object. fill!(A, Foo()) will return A filled with the result of evaluating Foo() once.

Examples

```
julia> A = zeros(2,3)
2×3 Array{Float64,2}:
 0.0  0.0  0.0
 0.0  0.0  0.0

julia> fill!(A, 2.)
2×3 Array{Float64,2}:
 2.0  2.0  2.0
 2.0  2.0  2.0

julia> a = [1, 1, 1]; A = fill!(Vector{Vector{Int}}(3), a); a[1] = 2; A
3-element Array{Array{Int64,1},1}:
 [2, 1, 1]
 [2, 1, 1]
 [2, 1, 1]

julia> x = 0; f() = (global x += 1; x); fill!(Vector{Int}(3), f())
3-element Array{Int64,1}:
 1
 1
 1
```

source

Base.similar – Method.

```
similar(array, [element_type=eltype(array)], [dims=size(array)])
```

Create an uninitialized mutable array with the given element type and size, based upon the given source array. The second and third arguments are both optional, defaulting to the given array's `eltype` and `size`. The dimensions may be specified either as a single tuple argument or as a series of integer arguments.

Custom AbstractArray subtypes may choose which specific array type is best-suited to return for the given element type and dimensionality. If they do not specialize this method, the default is an `Array{element_type}(dims...)`.

For example, `similar(1:10, 1, 4)` returns an uninitialized `Array{Int,2}` since ranges are neither mutable nor support 2 dimensions:

```
julia> similar(1:10, 1, 4)
1×4 Array{Int64,2}:
 4419743872  4374413872  4419743888  0
```

Conversely, `similar(trues(10,10), 2)` returns an uninitialized `BitVector` with two elements since `BitArrays` are both mutable and can support 1-dimensional arrays:

```
julia> similar(trues(10,10), 2)
2-element BitArray{1}:
 false
 false
```

Since `BitArrays` can only store elements of type `Bool`, however, if you request a different element type it will create a regular `Array` instead:

```
julia> similar(falses(10), Float64, 2, 4)
2×4 Array{Float64,2}:
 2.18425e-314  2.18425e-314  2.18425e-314  2.18425e-314
 2.18425e-314  2.18425e-314  2.18425e-314  2.18425e-314
```

source

Base.similar – Method.

```
similar(storagetype, indices)
```

Create an uninitialized mutable array analogous to that specified by `storagetype`, but with `indices` specified by the last argument. `storagetype` might be a type or a function.

Examples:

```
similar(Array{Int}, indices(A))
```

creates an array that "acts like" an `Array{Int}` (and might indeed be backed by one), but which is indexed identically to `A`. If `A` has conventional indexing, this will be identical to `Array{Int}(size(A))`, but if `A` has unconventional indexing then the indices of the result will match `A`.

```
similar(BitArray, (indices(A, 2),))
```

would create a 1-dimensional logical array whose indices match those of the columns of `A`.

```
similar(dims->zeros(Int, dims), indices(A))
```

would create an array of `Int`, initialized to zero, matching the indices of `A`.

source

Base.eye – Function.

```
eye([T::Type=Float64,] m::Integer, n::Integer)
```

`m`-by-`n` identity matrix. The default element type is `Float64`.

Examples

```
julia> eye(3, 4)
3×4 Array{Float64,2}:
 1.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0
 0.0  0.0  1.0  0.0

julia> eye(2, 2)
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0

julia> eye(Int, 2, 2)
2×2 Array{Int64,2}:
 1  0
 0  1
```

source

```
eye(m, n)
```

`m`-by-`n` identity matrix.

source

```
eye([T::Type=Float64,] n::Integer)
```

n-by-n identity matrix. The default element type is `Float64`.

Examples

```
julia> eye(Int, 2)
2×2 Array{Int64,2}:
 1  0
 0  1

julia> eye(2)
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0
```

source

```
eye(A)
```

Constructs an identity matrix of the same dimensions and type as `A`.

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Array{Int64,2}:
 1  2  3
 4  5  6
 7  8  9

julia> eye(A)
3×3 Array{Int64,2}:
 1  0  0
 0  1  0
 0  0  1
```

Note the difference from `ones`.

source

`Base.linspace` – Function.

```
linspace(start, stop, n=50)
```

Construct a range of n linearly spaced elements from `start` to `stop`.

```
julia> linspace(1.3,2.9,9)
1.3:0.2:2.9
```

source

`Base.logspace` – Function.

```
logspace(start::Real, stop::Real, n::Integer=50)
```

Construct a vector of n logarithmically spaced numbers from `10^start` to `10^stop`.

```
julia> logspace(1.,10.,5)
5-element Array{Float64,1}:
   10.0
 1778.28
    3.16228e5
    5.62341e7
    1.0e10
```

source

Base.Random.randsubseq – Function.

```
randsubseq(A, p) -> Vector
```

Return a vector consisting of a random subsequence of the given array A, where each element of A is included (in order) with independent probability p. (Complexity is linear in p*length(A), so this function is efficient even if p is small and A is large.) Technically, this process is known as "Bernoulli sampling" of A.

source

Base.Random.randsubseq! – Function.

```
randsubseq!(S, A, p)
```

Like randsubseq, but the results are stored in S (which is resized as needed).

source

50.2 Funciones básicas

Base.ndims – Function.

```
ndims(A::AbstractArray) -> Integer
```

Returns the number of dimensions of A.

```
julia> A = ones(3,4,5);

julia> ndims(A)
3
```

source

Base.size – Function.

```
size(A::AbstractArray, [dim...])
```

Returns a tuple containing the dimensions of A. Optionally you can specify the dimension(s) you want the length of, and get the length of that dimension, or a tuple of the lengths of dimensions you asked for.

```
julia> A = ones(2,3,4);

julia> size(A, 2)
3

julia> size(A,3,2)
(4, 3)
```

source

Base.indices – Method.

```
indices(A)
```

Returns the tuple of valid indices for array A.

```
julia> A = ones(5,6,7);

julia> indices(A)
(Base.OneTo(5), Base.OneTo(6), Base.OneTo(7))
```

source

Base.indices – Method.

```
indices(A, d)
```

Returns the valid range of indices for array A along dimension d.

```
julia> A = ones(5,6,7);

julia> indices(A,2)
Base.OneTo(6)
```

source

Base.length – Method.

```
length(A::AbstractArray) -> Integer
```

Returns the number of elements in A.

```
julia> A = ones(3,4,5);

julia> length(A)
60
```

source

Base.eachindex – Function.

```
eachindex(A...)
```

Creates an iterable object for visiting each index of an AbstractArray A in an efficient manner. For array types that have opted into fast linear indexing (like Array), this is simply the range 1:length(A). For other array types, this returns a specialized Cartesian range to efficiently index into the array with indices specified for every dimension. For other iterables, including strings and dictionaries, this returns an iterator object supporting arbitrary index types (e.g. unevenly spaced or non-integer indices).

Example for a sparse 2-d array:

```
julia> A = sparse([1, 1, 2], [1, 3, 1], [1, 2, -5])
2×3 SparseMatrixCSC{Int64,Int64} with 3 stored entries:
  [1, 1]  =  1
  [2, 1]  =  -5
  [1, 3]  =  2

julia> for iter in eachindex(A)

           @show iter.I[1], iter.I[2]

           @show A[iter]
```

```
    end
(iter.I[1], iter.I[2]) = (1, 1)
A[iter] = 1
(iter.I[1], iter.I[2]) = (2, 1)
A[iter] = -5
(iter.I[1], iter.I[2]) = (1, 2)
A[iter] = 0
(iter.I[1], iter.I[2]) = (2, 2)
A[iter] = 0
(iter.I[1], iter.I[2]) = (1, 3)
A[iter] = 2
(iter.I[1], iter.I[2]) = (2, 3)
A[iter] = 0
```

If you supply more than one `AbstractArray` argument, `eachindex` will create an iterable object that is fast for all arguments (a `UnitRange` if all inputs have fast linear indexing, a `CartesianRange` otherwise). If the arrays have different sizes and/or dimensionalities, `eachindex` returns an iterable that spans the largest range along each dimension.

source

`Base.linearindices` – Function.

```
linearindices(A)
```

Returns a `UnitRange` specifying the valid range of indices for `A[i]` where `i` is an `Int`. For arrays with conventional indexing (indices start at 1), or any multidimensional array, this is `1:length(A)`; however, for one-dimensional arrays with unconventional indices, this is `indices(A, 1)`.

Calling this function is the "safe" way to write algorithms that exploit linear indexing.

```
julia> A = ones(5,6,7);

julia> b = linearindices(A);

julia> extrema(b)
(1, 210)
```

source

`Base.IndexStyle` – Type.

```
IndexStyle(A)
IndexStyle(typeof(A))
```

`IndexStyle` specifies the "native indexing style" for array `A`. When you define a new `AbstractArray` type, you can choose to implement either linear indexing or cartesian indexing. If you decide to implement linear indexing, then you must set this trait for your array type:

```
Base.IndexStyle(::Type{<:MyArray}) = IndexLinear()
```

The default is `IndexCartesian()`.

Julia's internal indexing machinery will automatically (and invisibly) convert all indexing operations into the preferred style using `sub2ind` or `ind2sub`. This allows users to access elements of your array using any indexing style, even when explicit methods have not been provided.

If you define both styles of indexing for your `AbstractArray`, this trait can be used to select the most performant indexing style. Some methods check this trait on their inputs, and dispatch to different algorithms depending on the most efficient access pattern. In particular, `eachindex` creates an iterator whose type depends on the setting of this trait.

source

`Base.countnz` – Function.

```
countnz(A) -> Integer
```

Counts the number of nonzero values in array `A` (dense or sparse). Note that this is not a constant-time operation. For sparse matrices, one should usually use `nnz`, which returns the number of stored values.

```
julia> A = [1 2 4; 0 0 1; 1 1 0]
3×3 Array{Int64,2}:
 1  2  4
 0  0  1
 1  1  0

julia> countnz(A)
6
```

source

`Base.conj!` – Function.

```
conj!(A)
```

Transform an array to its complex conjugate in-place.

See also `conj`.

Example

```
julia> A = [1+im 2-im; 2+2im 3+im]
2×2 Array{Complex{Int64},2}:
 1+1im  2-1im
 2+2im  3+1im

julia> conj!(A);

julia> A
2×2 Array{Complex{Int64},2}:
 1-1im  2+1im
 2-2im  3-1im
```

source

`Base.stride` – Function.

```
stride(A, k::Integer)
```

Returns the distance in memory (in number of elements) between adjacent elements in dimension k.

```
julia> A = ones(3,4,5);

julia> stride(A,2)
3

julia> stride(A,3)
12
```

source

Base.strides – Function.

```
strides(A)
```

Returns a tuple of the memory strides in each dimension.

```
julia> A = ones(3,4,5);

julia> strides(A)
(1, 3, 12)
```

source

Base.ind2sub – Function.

```
ind2sub(a, index) -> subscripts
```

Returns a tuple of subscripts into array a corresponding to the linear index `index`.

```
julia> A = ones(5,6,7);

julia> ind2sub(A,35)
(5, 1, 2)

julia> ind2sub(A,70)
(5, 2, 3)
```

source

```
ind2sub(dims, index) -> subscripts
```

Returns a tuple of subscripts into an array with dimensions `dims`, corresponding to the linear index `index`.

Example:

```
i, j, ... = ind2sub(size(A), indmax(A))
```

provides the indices of the maximum element.

```
julia> ind2sub((3,4),2)
(2, 1)

julia> ind2sub((3,4),3)
(3, 1)

julia> ind2sub((3,4),4)
(1, 2)
```

source

Base.sub2ind – Function.

```
sub2ind(dims, i, j, k...) -> index
```

The inverse of ind2sub, returns the linear index corresponding to the provided subscripts.

```
julia> sub2ind((5,6,7),1,2,3)
66

julia> sub2ind((5,6,7),1,6,3)
86
```

source

Base.LinAlg.checksquare – Function.

```
LinAlg.checksquare(A)
```

Check that a matrix is square, then return its common dimension. For multiple arguments, return a vector.

Example

```
julia> A = ones(4,4); B = zeros(5,5);

julia> LinAlg.checksquare(A, B)
2-element Array{Int64,1}:
 4
 5
```

source

50.3 Retransmisión y Vectorización

Ver también la sintaxis de puntos para vectorizar funciones; por ejemplo, `f. (args ...)` llama implícitamente a `broadcast(f, args...)`. En lugar de confiar en los métodos "vectorizados" de funciones como `sin` para operar en arrays, debe usar `sin.(A)` para vectorizar a través de `broadcast`.

Base.broadcast – Function.

```
broadcast(f, As...)
```

Broadcasts the arrays, tuples, `Ref`s, nullables, and/or scalars `As` to a container of the appropriate type and dimensions. In this context, anything that is not a subtype of `AbstractArray`, `Ref` (except for `Ptr`s), `Tuple`, or `Nullable` is considered a scalar. The resulting container is established by the following rules:

- If all the arguments are scalars, it returns a scalar.
- If the arguments are tuples and zero or more scalars, it returns a tuple.
- If the arguments contain at least one array or `Ref`, it returns an array (expanding singleton dimensions), and treats `Ref`s as 0-dimensional arrays, and tuples as 1-dimensional arrays.

The following additional rule applies to `Nullable` arguments: If there is at least one `Nullable`, and all the arguments are scalars or `Nullable`, it returns a `Nullable` treating `Nullable`s as "containers".

A special syntax exists for broadcasting: `f.(args...)` is equivalent to `broadcast(f, args...)`, and nested `f.(g.(args...))` calls are fused into a single broadcast loop.

```
julia> A = [1, 2, 3, 4, 5]
5-element Array{Int64,1}:
 1
 2
 3
 4
 5

julia> B = [1 2; 3 4; 5 6; 7 8; 9 10]
5×2 Array{Int64,2}:
 1   2
 3   4
 5   6
 7   8
 9  10

julia> broadcast(+, A, B)
5×2 Array{Int64,2}:
  2   3
  5   6
  8   9
 11  12
 14  15

julia> parse.(Int, ["1", "2"])
2-element Array{Int64,1}:
 1
 2

julia> abs.((1, -2))
(1, 2)

julia> broadcast(+, 1.0, (0, -2.0))
(1.0, -1.0)

julia> broadcast(+, 1.0, (0, -2.0), Ref(1))
2-element Array{Float64,1}:
 2.0
 0.0

julia> (+).([[0,2], [1,3]], Ref{Vector{Int}}([1,-1]))
2-element Array{Array{Int64,1},1}:
 [1, 1]
 [2, 2]

julia> string.(("one","two","three","four"), ": ", 1:4)
4-element Array{String,1}:
 "one: 1"
 "two: 2"
 "three: 3"
 "four: 4"

julia> Nullable("X") .* "Y"
Nullable{String}("XY")

julia> broadcast(/, 1.0, Nullable(2.0))
```

```
Nullable{Float64}(0.5)

julia> (1 + im) ./ Nullable{Int}()
Nullable{Complex{Float64}}()
```

source

Base.broadcast! – Function.

```
broadcast!(f, dest, As...)
```

Like broadcast, but store the result of broadcast(f, As...) in the dest array. Note that dest is only used to store the result, and does not supply arguments to f unless it is also listed in the As, as in broadcast!(f, A, A, B) to perform A[:] = broadcast(f, A, B).

source

Base.Broadcast.@__dot__ – Macro.

```
@. expr
```

Convert every function call or operator in expr into a "dot call" (e.g. convert f(x) to f.(x)), and convert every assignment in expr to a "dot assignment" (e.g. convert += to .+=).

If you want to *avoid* adding dots for selected function calls in expr, splice those function calls in with $. For example, @. sqrt(abs($sort(x))) is equivalent to sqrt.(abs.(sort(x))) (no dot for sort).

(@. is equivalent to a call to @__dot__.)

```
julia> x = 1.0:3.0; y = similar(x);

julia> @. y = x + 3 * sin(x)
3-element Array{Float64,1}:
 3.52441
 4.72789
 3.42336
```

source

Base.Broadcast.broadcast_getindex – Function.

```
broadcast_getindex(A, inds...)
```

Broadcasts the inds arrays to a common size like broadcast and returns an array of the results A[ks...], where ks goes over the positions in the broadcast result A.

```
julia> A = [1, 2, 3, 4, 5]
5-element Array{Int64,1}:
 1
 2
 3
 4
 5

julia> B = [1 2; 3 4; 5 6; 7 8; 9 10]
5×2 Array{Int64,2}:
 1   2
```

```
 3   4
 5   6
 7   8
 9  10

julia> C = broadcast(+,A,B)
5×2 Array{Int64,2}:
  2   3
  5   6
  8   9
 11  12
 14  15

julia> broadcast_getindex(C,[1,2,10])
3-element Array{Int64,1}:
  2
  5
 15
```

source

Base.Broadcast.broadcast_setindex! – Function.

```
broadcast_setindex!(A, X, inds...)
```

Broadcasts the X and inds arrays to a common size and stores the value from each position in X at the indices in A given by the same positions in inds.

source

50.4 Indexación y Asignación

Base.getindex – Method.

```
getindex(A, inds...)
```

Returns a subset of array A as specified by inds, where each ind may be an Int, a Range, or a Vector. See the manual section on array indexing for details.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> getindex(A, 1)
1

julia> getindex(A, [2, 1])
2-element Array{Int64,1}:
 3
 1

julia> getindex(A, 2:4)
3-element Array{Int64,1}:
```

```
3
2
4
```

source

Base.setindex! – Method.

```
setindex!(A, X, inds...)
```

Store values from array X within some subset of A as specified by inds.

source

Base.copy! – Method.

```
copy!(dest, Rdest::CartesianRange, src, Rsrc::CartesianRange) -> dest
```

Copy the block of src in the range of Rsrc to the block of dest in the range of Rdest. The sizes of the two regions must match.

source

Base.isassigned – Function.

```
isassigned(array, i) -> Bool
```

Tests whether the given array has a value associated with index i. Returns false if the index is out of bounds, or has an undefined reference.

```
julia> isassigned(rand(3, 3), 5)
true

julia> isassigned(rand(3, 3), 3 * 3 + 1)
false

julia> mutable struct Foo end

julia> v = similar(rand(3), Foo)
3-element Array{Foo,1}:
 #undef
 #undef
 #undef

julia> isassigned(v, 1)
false
```

source

Base.Colon – Type.

```
Colon()
```

Colons (:) are used to signify indexing entire objects or dimensions at once.

Very few operations are defined on Colons directly; instead they are converted by to_indices to an internal vector type (Base.Slice) to represent the collection of indices they span before being used.

source

Base.IteratorsMD.CartesianIndex – Type.

```
CartesianIndex(i, j, k...)   -> I
CartesianIndex((i, j, k...)) -> I
```

Create a multidimensional index `I`, which can be used for indexing a multidimensional array `A`. In particular, `A[I]` is equivalent to `A[i,j,k...]`. One can freely mix integer and `CartesianIndex` indices; for example, `A[Ipre, i, Ipost]` (where `Ipre` and `Ipost` are `CartesianIndex` indices and `i` is an `Int`) can be a useful expression when writing algorithms that work along a single dimension of an array of arbitrary dimensionality.

A `CartesianIndex` is sometimes produced by `eachindex`, and always when iterating with an explicit `CartesianRange`.

Example

```
julia> A = reshape(collect(1:16), (2, 2, 2, 2))
2×2×2×2 Array{Int64,4}:
[:, :, 1, 1] =
 1  3
 2  4

[:, :, 2, 1] =
 5  7
 6  8

[:, :, 1, 2] =
  9  11
 10  12

[:, :, 2, 2] =
 13  15
 14  16

julia> A[CartesianIndex((1, 1, 1, 1))]
1

julia> A[CartesianIndex((1, 1, 1, 2))]
9

julia> A[CartesianIndex((1, 1, 2, 1))]
5
```

source

Base.IteratorsMD.CartesianRange – Type.

```
CartesianRange(Istart::CartesianIndex, Istop::CartesianIndex) -> R
CartesianRange(sz::Dims) -> R
CartesianRange(istart:istop, jstart:jstop, ...) -> R
```

Define a region `R` spanning a multidimensional rectangular range of integer indices. These are most commonly encountered in the context of iteration, where `for I in R ... end` will return `CartesianIndex` indices `I` equivalent to the nested loops

```
for j = jstart:jstop
    for i = istart:istop
        ...
    end
end
```

Consequently these can be useful for writing algorithms that work in arbitrary dimensions.

```
julia> foreach(println, CartesianRange((2, 2, 2)))
CartesianIndex{3}((1, 1, 1))
CartesianIndex{3}((2, 1, 1))
CartesianIndex{3}((1, 2, 1))
CartesianIndex{3}((2, 2, 1))
CartesianIndex{3}((1, 1, 2))
CartesianIndex{3}((2, 1, 2))
CartesianIndex{3}((1, 2, 2))
CartesianIndex{3}((2, 2, 2))
```

source

`Base.to_indices` – Function.

```
to_indices(A, I::Tuple)
```

Convert the tuple `I` to a tuple of indices for use in indexing into array `A`.

The returned tuple must only contain either `Int`s or `AbstractArray`s of scalar indices that are supported by array `A`. It will error upon encountering a novel index type that it does not know how to process.

For simple index types, it defers to the unexported `Base.to_index(A, i)` to process each index `i`. While this internal function is not intended to be called directly, `Base.to_index` may be extended by custom array or index types to provide custom indexing behaviors.

More complicated index types may require more context about the dimension into which they index. To support those cases, `to_indices(A, I)` calls `to_indices(A, indices(A), I)`, which then recursively walks through both the given tuple of indices and the dimensional indices of `A` in tandem. As such, not all index types are guaranteed to propagate to `Base.to_index`.

source

`Base.checkbounds` – Function.

```
checkbounds(Bool, A, I...)
```

Return `true` if the specified indices `I` are in bounds for the given array `A`. Subtypes of `AbstractArray` should specialize this method if they need to provide custom bounds checking behaviors; however, in many cases one can rely on `A`'s indices and `checkindex`.

See also `checkindex`.

```
julia> A = rand(3, 3);

julia> checkbounds(Bool, A, 2)
true

julia> checkbounds(Bool, A, 3, 4)
false

julia> checkbounds(Bool, A, 1:3)
true

julia> checkbounds(Bool, A, 1:3, 2:4)
false
```

source

```
checkbounds(A, I...)
```

Throw an error if the specified indices `I` are not in bounds for the given array `A`.

source

`Base.checkindex` – Function.

```
checkindex(Bool, inds::AbstractUnitRange, index)
```

Return `true` if the given `index` is within the bounds of `inds`. Custom types that would like to behave as indices for all arrays can extend this method in order to provide a specialized bounds checking implementation.

```
julia> checkindex(Bool,1:20,8)
true

julia> checkindex(Bool,1:20,21)
false
```

source

50.5 Vistas (SubArrays y otros tipos de vistas)

`Base.view` – Function.

```
view(A, inds...)
```

Like `getindex`, but returns a view into the parent array `A` with the given indices instead of making a copy. Calling `getindex` or `setindex!` on the returned `SubArray` computes the indices to the parent array on the fly without checking bounds.

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> b = view(A, :, 1)
2-element SubArray{Int64,1,Array{Int64,2},Tuple{Base.Slice{Base.OneTo{Int64}},Int64},true}:
 1
 3

julia> fill!(b, 0)
2-element SubArray{Int64,1,Array{Int64,2},Tuple{Base.Slice{Base.OneTo{Int64}},Int64},true}:
 0
 0

julia> A # Note A has changed even though we modified b
2×2 Array{Int64,2}:
 0  2
 0  4
```

source

`Base.@view` – Macro.

```
@view A[inds...]
```

Creates a SubArray from an indexing expression. This can only be applied directly to a reference expression (e.g. `@view A[1,2:end]`), and should *not* be used as the target of an assignment (e.g. `@view(A[1,2:end]) = ...`). See also `@views` to switch an entire block of code to use views for slicing.

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> b = @view A[:, 1]
2-element SubArray{Int64,1,Array{Int64,2},Tuple{Base.Slice{Base.OneTo{Int64}},Int64},true}:
 1
 3

julia> fill!(b, 0)
2-element SubArray{Int64,1,Array{Int64,2},Tuple{Base.Slice{Base.OneTo{Int64}},Int64},true}:
 0
 0

julia> A
2×2 Array{Int64,2}:
 0  2
 0  4
```

source

`Base.@views` – Macro.

```
@views expression
```

Convert every array-slicing operation in the given expression (which may be a `begin`/`end` block, loop, function, etc.) to return a view. Scalar indices, non-array types, and explicit `getindex` calls (as opposed to `array[...]`) are unaffected.

Note that the `@views` macro only affects `array[...]` expressions that appear explicitly in the given `expression`, not array slicing that occurs in functions called by that code.

source

`Base.parent` – Function.

```
parent(A)
```

Returns the "parent array" of an array view type (e.g., SubArray), or the array itself if it is not a view.

source

`Base.parentindexes` – Function.

```
parentindexes(A)
```

From an array view `A`, returns the corresponding indexes in the parent.

source

`Base.slicedim` – Function.

```
slicedim(A, d::Integer, i)
```

Return all the data of A where the index for dimension d equals i. Equivalent to A[:,:,...,i,:,:,...] where i is in position d.

Example

```
julia> A = [1 2 3 4; 5 6 7 8]
2×4 Array{Int64,2}:
 1  2  3  4
 5  6  7  8

julia> slicedim(A,2,3)
2-element Array{Int64,1}:
 3
 7
```

source

Base.reinterpret – Function.

```
reinterpret(type, A)
```

Change the type-interpretation of a block of memory. For arrays, this constructs an array with the same binary data as the given array, but with the specified element type. For example, reinterpret(Float32, UInt32(7)) interprets the 4 bytes corresponding to UInt32(7) as a Float32.

> **Warning**
>
> It is not allowed to reinterpret an array to an element type with a larger alignment then the alignment of the array. For a normal Array, this is the alignment of its element type. For a reinterpreted array, this is the alignment of the Array it was reinterpreted from. For example, reinterpret(UInt32, UInt8[0, 0, 0, 0]) is not allowed but reinterpret(UInt32, reinterpret(UInt8, Float32[1.0])) is allowed.

Examples

```
julia> reinterpret(Float32, UInt32(7))
1.0f-44

julia> reinterpret(Float32, UInt32[1 2 3 4 5])
1×5 Array{Float32,2}:
 1.4013f-45  2.8026f-45  4.2039f-45  5.60519f-45  7.00649f-45
```

source

Base.reshape – Function.

```
reshape(A, dims...) -> R
reshape(A, dims) -> R
```

Return an array R with the same data as A, but with different dimension sizes or number of dimensions. The two arrays share the same underlying data, so that setting elements of R alters the values of A and vice versa.

The new dimensions may be specified either as a list of arguments or as a shape tuple. At most one dimension may be specified with a :, in which case its length is computed such that its product with all the specified dimensions is equal to the length of the original array A. The total number of elements must not change.

```
julia> A = collect(1:16)
16-element Array{Int64,1}:
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
 16

julia> reshape(A, (4, 4))
4×4 Array{Int64,2}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> reshape(A, 2, :)
2×8 Array{Int64,2}:
 1  3  5  7   9  11  13  15
 2  4  6  8  10  12  14  16
```

source

`Base.squeeze` – Function.

```
squeeze(A, dims)
```

Remove the dimensions specified by `dims` from array `A`. Elements of `dims` must be unique and within the range `1:ndims(A)`. `size(A,i)` must equal 1 for all `i` in `dims`.

Example

```
julia> a = reshape(collect(1:4),(2,2,1,1))
2×2×1×1 Array{Int64,4}:
[:, :, 1, 1] =
 1  3
 2  4

julia> squeeze(a,3)
2×2×1 Array{Int64,3}:
[:, :, 1] =
 1  3
 2  4
```

source

`Base.vec` – Function.

```
vec(a::AbstractArray) -> Vector
```

Reshape the array a as a one-dimensional column vector. The resulting array shares the same underlying data as a, so modifying one will also modify the other.

Example

```
julia> a = [1 2 3; 4 5 6]
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> vec(a)
6-element Array{Int64,1}:
 1
 4
 2
 5
 3
 6
```

See also `reshape`.

source

50.6 Concatenación y permutación

`Base.cat` – Function.

```
cat(dims, A...)
```

Concatenate the input arrays along the specified dimensions in the iterable `dims`. For dimensions not in `dims`, all input arrays should have the same size, which will also be the size of the output array along that dimension. For dimensions in `dims`, the size of the output array is the sum of the sizes of the input arrays along that dimension. If `dims` is a single number, the different arrays are tightly stacked along that dimension. If `dims` is an iterable containing several dimensions, this allows one to construct block diagonal matrices and their higher-dimensional analogues by simultaneously increasing several dimensions for every new input array and putting zero blocks elsewhere. For example, `cat([1,2], matrices...)` builds a block diagonal matrix, i.e. a block matrix with `matrices[1]`, `matrices[2]`, ... as diagonal blocks and matching zero blocks away from the diagonal.

source

`Base.vcat` – Function.

```
vcat(A...)
```

Concatenate along dimension 1.

```
julia> a = [1 2 3 4 5]
1×5 Array{Int64,2}:
 1  2  3  4  5

julia> b = [6 7 8 9 10; 11 12 13 14 15]
2×5 Array{Int64,2}:
  6   7   8   9  10
 11  12  13  14  15
```

```
julia> vcat(a,b)
3×5 Array{Int64,2}:
  1   2   3   4   5
  6   7   8   9  10
 11  12  13  14  15

julia> c = ([1 2 3], [4 5 6])
([1 2 3], [4 5 6])

julia> vcat(c...)
2×3 Array{Int64,2}:
 1  2  3
 4  5  6
```

source

Base.hcat – Function.

```
hcat(A...)
```

Concatenate along dimension 2.

```
julia> a = [1; 2; 3; 4; 5]
5-element Array{Int64,1}:
 1
 2
 3
 4
 5

julia> b = [6 7; 8 9; 10 11; 12 13; 14 15]
5×2 Array{Int64,2}:
  6   7
  8   9
 10  11
 12  13
 14  15

julia> hcat(a,b)
5×3 Array{Int64,2}:
 1   6   7
 2   8   9
 3  10  11
 4  12  13
 5  14  15

julia> c = ([1; 2; 3], [4; 5; 6])
([1, 2, 3], [4, 5, 6])

julia> hcat(c...)
3×2 Array{Int64,2}:
 1  4
 2  5
 3  6
```

source

Base.hvcat – Function.

```
hvcat(rows::Tuple{Vararg{Int}}, values...)
```

Horizontal and vertical concatenation in one call. This function is called for block matrix syntax. The first argument specifies the number of arguments to concatenate in each block row.

```
julia> a, b, c, d, e, f = 1, 2, 3, 4, 5, 6
(1, 2, 3, 4, 5, 6)

julia> [a b c; d e f]
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> hvcat((3,3), a,b,c,d,e,f)
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> [a b;c d; e f]
3×2 Array{Int64,2}:
 1  2
 3  4
 5  6

julia> hvcat((2,2,2), a,b,c,d,e,f)
3×2 Array{Int64,2}:
 1  2
 3  4
 5  6
```

If the first argument is a single integer n, then all block rows are assumed to have n block columns.

source

Base.flipdim – Function.

```
flipdim(A, d::Integer)
```

Reverse A in dimension d.

Example

```
julia> b = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> flipdim(b,2)
2×2 Array{Int64,2}:
 2  1
 4  3
```

source

Base.circshift – Function.

```
circshift(A, shifts)
```

Circularly shift the data in an array. The second argument is a vector giving the amount to shift in each dimension.

Example

```
julia> b = reshape(collect(1:16), (4,4))
4×4 Array{Int64,2}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> circshift(b, (0,2))
4×4 Array{Int64,2}:
  9  13  1  5
 10  14  2  6
 11  15  3  7
 12  16  4  8

julia> circshift(b, (-1,0))
4×4 Array{Int64,2}:
 2  6  10  14
 3  7  11  15
 4  8  12  16
 1  5   9  13
```

See also `circshift!`.

source

`Base.circshift!` – Function.

```
circshift!(dest, src, shifts)
```

Circularly shift the data in `src`, storing the result in `dest`. `shifts` specifies the amount to shift in each dimension.

The `dest` array must be distinct from the `src` array (they cannot alias each other).

See also `circshift`.

source

`Base.circcopy!` – Function.

```
circcopy!(dest, src)
```

Copy `src` to `dest`, indexing each dimension modulo its length. `src` and `dest` must have the same size, but can be offset in their indices; any offset results in a (circular) wraparound. If the arrays have overlapping indices, then on the domain of the overlap `dest` agrees with `src`.

Example

```
julia> src = reshape(collect(1:16), (4,4))
4×4 Array{Int64,2}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
```

```
 4  8  12  16

julia> dest = OffsetArray{Int}((0:3,2:5))

julia> circcopy!(dest, src)
OffsetArrays.OffsetArray{Int64,2,Array{Int64,2}} with indices 0:3×2:5:
 8  12  16  4
 5   9  13  1
 6  10  14  2
 7  11  15  3

julia> dest[1:3,2:4] == src[1:3,2:4]
true
```

source

`Base.contains` – Method.

```
contains(fun, itr, x) -> Bool
```

Returns `true` if there is at least one element y in `itr` such that `fun(y,x)` is `true`.

```
julia> vec = [10, 100, 200]
3-element Array{Int64,1}:
  10
 100
 200

julia> contains(==, vec, 200)
true

julia> contains(==, vec, 300)
false

julia> contains(>, vec, 100)
true

julia> contains(>, vec, 200)
false
```

source

`Base.find` – Method.

```
find(A)
```

Return a vector of the linear indexes of the non-zeros in `A` (determined by `A[i]!=0`). A common use of this is to convert a boolean array to an array of indexes of the `true` elements. If there are no non-zero elements of `A`, `find` returns an empty array.

Examples

```
julia> A = [true false; false true]
2×2 Array{Bool,2}:
  true  false
 false   true
```

```
julia> find(A)
2-element Array{Int64,1}:
 1
 4

julia> find(zeros(3))
0-element Array{Int64,1}
```

source

Base.find – Method.

```
find(f::Function, A)
```

Return a vector `I` of the linear indexes of `A` where `f(A[I])` returns `true`. If there are no such elements of `A`, find returns an empty array.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> find(isodd,A)
2-element Array{Int64,1}:
 1
 2

julia> find(isodd, [2, 4])
0-element Array{Int64,1}
```

source

Base.findn – Function.

```
findn(A)
```

Return a vector of indexes for each dimension giving the locations of the non-zeros in `A` (determined by `A[i]!=0`). If there are no non-zero elements of `A`, `findn` returns a 2-tuple of empty arrays.

Examples

```
julia> A = [1 2 0; 0 0 3; 0 4 0]
3×3 Array{Int64,2}:
 1  2  0
 0  0  3
 0  4  0

julia> findn(A)
([1, 1, 3, 2], [1, 2, 2, 3])

julia> A = zeros(2,2)
2×2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0

julia> findn(A)
(Int64[], Int64[])
```

source

Base.findnz – Function.

```
findnz(A)
```

Return a tuple `(I, J, V)` where `I` and `J` are the row and column indexes of the non-zero values in matrix `A`, and `V` is a vector of the non-zero values.

Example

```
julia> A = [1 2 0; 0 0 3; 0 4 0]
3×3 Array{Int64,2}:
 1  2  0
 0  0  3
 0  4  0

julia> findnz(A)
([1, 1, 3, 2], [1, 2, 2, 3], [1, 2, 4, 3])
```

source

Base.findfirst – Method.

```
findfirst(A)
```

Return the linear index of the first non-zero value in `A` (determined by `A[i]!=0`). Returns `0` if no such value is found.

Examples

```
julia> A = [0 0; 1 0]
2×2 Array{Int64,2}:
 0  0
 1  0

julia> findfirst(A)
2

julia> findfirst(zeros(3))
0
```

source

Base.findfirst – Method.

```
findfirst(A, v)
```

Return the linear index of the first element equal to `v` in `A`. Returns `0` if `v` is not found.

Examples

```
julia> A = [4 6; 2 2]
2×2 Array{Int64,2}:
 4  6
 2  2

julia> findfirst(A,2)
```

```
2

julia> findfirst(A,3)
0
```

source

Base.findfirst – Method.

```
findfirst(predicate::Function, A)
```

Return the linear index of the first element of `A` for which `predicate` returns `true`. Returns `0` if there is no such element.

Examples

```
julia> A = [1 4; 2 2]
2×2 Array{Int64,2}:
 1  4
 2  2

julia> findfirst(iseven, A)
2

julia> findfirst(x -> x>10, A)
0
```

source

Base.findlast – Method.

```
findlast(A)
```

Return the linear index of the last non-zero value in `A` (determined by `A[i]!=0`). Returns `0` if there is no non-zero value in `A`.

Examples

```
julia> A = [1 0; 1 0]
2×2 Array{Int64,2}:
 1  0
 1  0

julia> findlast(A)
2

julia> A = zeros(2,2)
2×2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0

julia> findlast(A)
0
```

source

Base.findlast – Method.

```
findlast(A, v)
```

Return the linear index of the last element equal to `v` in `A`. Returns `0` if there is no element of `A` equal to `v`.

Examples

```
julia> A = [1 2; 2 1]
2×2 Array{Int64,2}:
 1  2
 2  1

julia> findlast(A,1)
4

julia> findlast(A,2)
3

julia> findlast(A,3)
0
```

source

`Base.findlast` – Method.

```
findlast(predicate::Function, A)
```

Return the linear index of the last element of `A` for which `predicate` returns `true`. Returns `0` if there is no such element.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> findlast(isodd, A)
2

julia> findlast(x -> x > 5, A)
0
```

source

`Base.findnext` – Method.

```
findnext(A, i::Integer)
```

Find the next linear index `>= i` of a non-zero element of `A`, or `0` if not found.

Examples

```
julia> A = [0 0; 1 0]
2×2 Array{Int64,2}:
 0  0
 1  0
```

```
julia> findnext(A,1)
2

julia> findnext(A,3)
0
```

source

Base.findnext – Method.

```
findnext(predicate::Function, A, i::Integer)
```

Find the next linear index >= i of an element of A for which predicate returns true, or 0 if not found.

Examples

```
julia> A = [1 4; 2 2]
2×2 Array{Int64,2}:
 1  4
 2  2

julia> findnext(isodd, A, 1)
1

julia> findnext(isodd, A, 2)
0
```

source

Base.findnext – Method.

```
findnext(A, v, i::Integer)
```

Find the next linear index >= i of an element of A equal to v (using ==), or 0 if not found.

Examples

```
julia> A = [1 4; 2 2]
2×2 Array{Int64,2}:
 1  4
 2  2

julia> findnext(A,4,4)
0

julia> findnext(A,4,3)
3
```

source

Base.findprev – Method.

```
findprev(A, i::Integer)
```

Find the previous linear index <= i of a non-zero element of A, or 0 if not found.

Examples

```
julia> A = [0 0; 1 2]
2×2 Array{Int64,2}:
 0  0
 1  2

julia> findprev(A,2)
2

julia> findprev(A,1)
0
```

source

Base.findprev – Method.

```
findprev(predicate::Function, A, i::Integer)
```

Find the previous linear index <= `i` of an element of `A` for which `predicate` returns `true`, or `0` if not found.

Examples

```
julia> A = [4 6; 1 2]
2×2 Array{Int64,2}:
 4  6
 1  2

julia> findprev(isodd, A, 1)
0

julia> findprev(isodd, A, 3)
2
```

source

Base.findprev – Method.

```
findprev(A, v, i::Integer)
```

Find the previous linear index <= `i` of an element of `A` equal to `v` (using ==), or `0` if not found.

Examples

```
julia> A = [0 0; 1 2]
2×2 Array{Int64,2}:
 0  0
 1  2

julia> findprev(A, 1, 4)
2

julia> findprev(A, 1, 1)
0
```

source

Base.permutedims – Function.

```
permutedims(A, perm)
```

Permute the dimensions of array `A`. `perm` is a vector specifying a permutation of length `ndims(A)`. This is a generalization of transpose for multi-dimensional arrays. Transpose is equivalent to `permutedims(A, [2,1])`.

See also: `PermutedDimsArray`.

Example

```
julia> A = reshape(collect(1:8), (2,2,2))
2×2×2 Array{Int64,3}:
[:, :, 1] =
 1  3
 2  4

[:, :, 2] =
 5  7
 6  8

julia> permutedims(A, [3, 2, 1])
2×2×2 Array{Int64,3}:
[:, :, 1] =
 1  3
 5  7

[:, :, 2] =
 2  4
 6  8
```

source

`Base.permutedims!` – Function.

```
permutedims!(dest, src, perm)
```

Permute the dimensions of array `src` and store the result in the array `dest`. `perm` is a vector specifying a permutation of length `ndims(src)`. The preallocated array `dest` should have `size(dest) == size(src)[perm]` and is completely overwritten. No in-place permutation is supported and unexpected results will happen if `src` and `dest` have overlapping memory regions.

See also `permutedims`.

source

`Base.PermutedDimsArrays.PermutedDimsArray` – Type.

```
PermutedDimsArray(A, perm) -> B
```

Given an AbstractArray `A`, create a view `B` such that the dimensions appear to be permuted. Similar to `permutedims`, except that no copying occurs (`B` shares storage with `A`).

See also: `permutedims`.

Example

```
julia> A = rand(3,5,4);

julia> B = PermutedDimsArray(A, (3,1,2));
```

```
julia> size(B)
(4, 3, 5)

julia> B[3,1,2] == A[1,2,3]
true
```

source

Base.promote_shape – Function.

```
promote_shape(s1, s2)
```

Check two array shapes for compatibility, allowing trailing singleton dimensions, and return whichever shape has more dimensions.

```
julia> a = ones(3,4,1,1,1);

julia> b = ones(3,4);

julia> promote_shape(a,b)
(Base.OneTo(3), Base.OneTo(4), Base.OneTo(1), Base.OneTo(1), Base.OneTo(1))

julia> promote_shape((2,3,1,4), (2, 3, 1, 4, 1))
(2, 3, 1, 4, 1)
```

source

50.7 Funciones de Arrays

Base.accumulate – Method.

```
accumulate(op, A, dim=1)
```

Cumulative operation op along a dimension `dim` (defaults to 1). See also `accumulate!` to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow). For common operations there are specialized variants of `accumulate`, see: `cumsum`, `cumprod`

```
julia> accumulate(+, [1,2,3])
3-element Array{Int64,1}:
 1
 3
 6

julia> accumulate(*, [1,2,3])
3-element Array{Int64,1}:
 1
 2
 6
```

source

```
accumulate(op, v0, A)
```

Like `accumulate`, but using a starting element `v0`. The first entry of the result will be `op(v0, first(A))`. For example:

```
julia> accumulate(+, 100, [1,2,3])
3-element Array{Int64,1}:
 101
 103
 106

julia> accumulate(min, 0, [1,2,-1])
3-element Array{Int64,1}:
  0
  0
 -1
```

source

Base.accumulate! – Function.

```
accumulate!(op, B, A, dim=1)
```

Cumulative operation op on A along a dimension, storing the result in B. The dimension defaults to 1. See also accumulate.

source

Base.cumprod – Function.

```
cumprod(A, dim=1)
```

Cumulative product along a dimension dim (defaults to 1). See also cumprod! to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

```
julia> a = [1 2 3; 4 5 6]
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> cumprod(a,1)
2×3 Array{Int64,2}:
 1   2   3
 4  10  18

julia> cumprod(a,2)
2×3 Array{Int64,2}:
 1   2    6
 4  20  120
```

source

Base.cumprod! – Function.

```
cumprod!(B, A, dim::Integer=1)
```

Cumulative product of A along a dimension, storing the result in B. The dimension defaults to 1. See also cumprod.

source

Base.cumsum – Function.

```
cumsum(A, dim=1)
```

Cumulative sum along a dimension `dim` (defaults to 1). See also `cumsum!` to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

```
julia> a = [1 2 3; 4 5 6]
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> cumsum(a,1)
2×3 Array{Int64,2}:
 1  2  3
 5  7  9

julia> cumsum(a,2)
2×3 Array{Int64,2}:
 1  3   6
 4  9  15
```

source

`Base.cumsum!` – Function.

```
cumsum!(B, A, dim::Integer=1)
```

Cumulative sum of `A` along a dimension, storing the result in `B`. The dimension defaults to 1. See also `cumsum`.

source

`Base.cumsum_kbn` – Function.

```
cumsum_kbn(A, [dim::Integer=1])
```

Cumulative sum along a dimension, using the Kahan-Babuska-Neumaier compensated summation algorithm for additional accuracy. The dimension defaults to 1.

source

`Base.LinAlg.diff` – Function.

```
diff(A, [dim::Integer=1])
```

Finite difference operator of matrix or vector `A`. If `A` is a matrix, compute the finite difference over a dimension `dim` (default 1).

Example

```
julia> a = [2 4; 6 16]
2×2 Array{Int64,2}:
 2   4
 6  16

julia> diff(a,2)
2×1 Array{Int64,2}:
  2
 10
```

source

Base.LinAlg.gradient – Function.

```
gradient(F::AbstractVector, [h::Real])
```

Compute differences along vector F, using h as the spacing between points. The default spacing is one.

Example

```
julia> a = [2,4,6,8];

julia> gradient(a)
4-element Array{Float64,1}:
 2.0
 2.0
 2.0
 2.0
```

source

Base.rot180 – Function.

```
rot180(A)
```

Rotate matrix A 180 degrees.

Example

```
julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> rot180(a)
2×2 Array{Int64,2}:
 4  3
 2  1
```

source

```
rot180(A, k)
```

Rotate matrix A 180 degrees an integer k number of times. If k is even, this is equivalent to a copy.

Examples

```
julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> rot180(a,1)
2×2 Array{Int64,2}:
 4  3
 2  1

julia> rot180(a,2)
2×2 Array{Int64,2}:
 1  2
 3  4
```

source

Base.rotl90 – Function.

```
rotl90(A)
```

Rotate matrix A left 90 degrees.

Example

```
julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> rotl90(a)
2×2 Array{Int64,2}:
 2  4
 1  3
```

source

```
rotl90(A, k)
```

Rotate matrix A left 90 degrees an integer k number of times. If k is zero or a multiple of four, this is equivalent to a copy.

Examples

```
julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> rotl90(a,1)
2×2 Array{Int64,2}:
 2  4
 1  3

julia> rotl90(a,2)
2×2 Array{Int64,2}:
 4  3
 2  1

julia> rotl90(a,3)
2×2 Array{Int64,2}:
 3  1
 4  2

julia> rotl90(a,4)
2×2 Array{Int64,2}:
 1  2
 3  4
```

source

Base.rotr90 – Function.

```
rotr90(A)
```

Rotate matrix A right 90 degrees.

Example

```
julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> rotr90(a)
2×2 Array{Int64,2}:
 3  1
 4  2
```

source

```
rotr90(A, k)
```

Rotate matrix A right 90 degrees an integer k number of times. If k is zero or a multiple of four, this is equivalent to a copy.

Examples

```
julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> rotr90(a,1)
2×2 Array{Int64,2}:
 3  1
 4  2

julia> rotr90(a,2)
2×2 Array{Int64,2}:
 4  3
 2  1

julia> rotr90(a,3)
2×2 Array{Int64,2}:
 2  4
 1  3

julia> rotr90(a,4)
2×2 Array{Int64,2}:
 1  2
 3  4
```

source

`Base.reducedim` – Function.

```
reducedim(f, A, region[, v0])
```

Reduce 2-argument function `f` along dimensions of `A`. `region` is a vector specifying the dimensions to reduce, and `v0` is the initial value to use in the reductions. For `+`, `*`, `max` and `min` the `v0` argument is optional.

The associativity of the reduction is implementation-dependent; if you need a particular associativity, e.g. left-to-right, you should write your own loop. See documentation for `reduce`.

Examples

```
julia> a = reshape(collect(1:16), (4,4))
4×4 Array{Int64,2}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> reducedim(max, a, 2)
4×1 Array{Int64,2}:
 13
 14
 15
 16

julia> reducedim(max, a, 1)
1×4 Array{Int64,2}:
 4  8  12  16
```

source

`Base.mapreducedim` – Function.

```
mapreducedim(f, op, A, region[, v0])
```

Evaluates to the same as `reducedim(op, map(f, A), region, f(v0))`, but is generally faster because the intermediate array is avoided.

Examples

```
julia> a = reshape(collect(1:16), (4,4))
4×4 Array{Int64,2}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> mapreducedim(isodd, *, a, 1)
1×4 Array{Bool,2}:
 false  false  false  false

julia> mapreducedim(isodd, |, a, 1, true)
1×4 Array{Bool,2}:
 true  true  true  true
```

source

`Base.mapslices` – Function.

```
mapslices(f, A, dims)
```

Transform the given dimensions of array A using function f. f is called on each slice of A of the form A[...,:,...,:,...]. dims is an integer vector specifying where the colons go in this expression. The results are concatenated along the remaining dimensions. For example, if dims is [1,2] and A is 4-dimensional, f is called on A[:,:,i,j] for all i and j.

Examples

```
julia> a = reshape(collect(1:16),(2,2,2,2))
2×2×2×2 Array{Int64,4}:
[:, :, 1, 1] =
 1  3
 2  4

[:, :, 2, 1] =
 5  7
 6  8

[:, :, 1, 2] =
  9  11
 10  12

[:, :, 2, 2] =
 13  15
 14  16

julia> mapslices(sum, a, [1,2])
1×1×2×2 Array{Int64,4}:
[:, :, 1, 1] =
 10

[:, :, 2, 1] =
 26

[:, :, 1, 2] =
 42

[:, :, 2, 2] =
 58
```

source

Base.sum_kbn – Function.

```
sum_kbn(A)
```

Returns the sum of all elements of A, using the Kahan-Babuska-Neumaier compensated summation algorithm for additional accuracy.

source

50.8 Combinatoria

Base.Random.randperm – Function.

```
randperm([rng=GLOBAL_RNG,] n::Integer)
```

Construct a random permutation of length n. The optional `rng` argument specifies a random number generator (see Random Numbers). To randomly permute a arbitrary vector, see `shuffle` or `shuffle!`.

Example

```
julia> rng = MersenneTwister(1234);

julia> randperm(rng, 4)
4-element Array{Int64,1}:
 2
 1
 4
 3
```

source

`Base.invperm` – Function.

```
invperm(v)
```

Return the inverse permutation of v. If `B = A[v]`, then `A == B[invperm(v)]`.

Example

```
julia> v = [2; 4; 3; 1];

julia> invperm(v)
4-element Array{Int64,1}:
 4
 1
 3
 2

julia> A = ['a','b','c','d'];

julia> B = A[v]
4-element Array{Char,1}:
 'b'
 'd'
 'c'
 'a'

julia> B[invperm(v)]
4-element Array{Char,1}:
 'a'
 'b'
 'c'
 'd'
```

source

`Base.isperm` – Function.

```
isperm(v) -> Bool
```

Returns `true` if v is a valid permutation.

Examples

```
julia> isperm([1; 2])
true

julia> isperm([1; 3])
false
```

source

Base.permute! – Method.

```
permute!(v, p)
```

Permute vector v in-place, according to permutation p. No checking is done to verify that p is a permutation.

To return a new permutation, use `v[p]`. Note that this is generally faster than `permute!(v,p)` for large vectors.

See also ipermute!.

Example

```
julia> A = [1, 1, 3, 4];

julia> perm = [2, 4, 3, 1];

julia> permute!(A, perm);

julia> A
4-element Array{Int64,1}:
 1
 4
 3
 1
```

source

Base.ipermute! – Function.

```
ipermute!(v, p)
```

Like permute!, but the inverse of the given permutation is applied.

Example

```
julia> A = [1, 1, 3, 4];

julia> perm = [2, 4, 3, 1];

julia> ipermute!(A, perm);

julia> A
4-element Array{Int64,1}:
 4
 1
 3
 1
```

source

Base.Random.randcycle – Function.

```
randcycle([rng=GLOBAL_RNG,] n::Integer)
```

Construct a random cyclic permutation of length n. The optional rng argument specifies a random number generator, see Random Numbers.

Example

```
julia> rng = MersenneTwister(1234);

julia> randcycle(rng, 6)
6-element Array{Int64,1}:
 3
 5
 4
 6
 1
 2
```

source

Base.Random.shuffle – Function.

```
shuffle([rng=GLOBAL_RNG,] v)
```

Return a randomly permuted copy of v. The optional rng argument specifies a random number generator (see Random Numbers). To permute v in-place, see shuffle!. To obtain randomly permuted indices, see randperm.

Example

```
julia> rng = MersenneTwister(1234);

julia> shuffle(rng, collect(1:10))
10-element Array{Int64,1}:
  6
  1
 10
  2
  3
  9
  5
  7
  4
  8
```

source

Base.Random.shuffle! – Function.

```
shuffle!([rng=GLOBAL_RNG,] v)
```

In-place version of shuffle: randomly permute the array v in-place, optionally supplying the random-number generator rng.

Example

```
julia> rng = MersenneTwister(1234);

julia> shuffle!(rng, collect(1:16))
16-element Array{Int64,1}:
  2
 15
  5
 14
  1
  9
 10
  6
 11
  3
 16
  7
  4
 12
  8
 13
```

source

Base.reverse – Function.

```
reverse(v [, start=1 [, stop=length(v) ]] )
```

Return a copy of v reversed from start to stop.

Examples

```
julia> A = collect(1:5)
5-element Array{Int64,1}:
 1
 2
 3
 4
 5

julia> reverse(A)
5-element Array{Int64,1}:
 5
 4
 3
 2
 1

julia> reverse(A, 1, 4)
5-element Array{Int64,1}:
 4
 3
 2
 1
 5

julia> reverse(A, 3, 5)
5-element Array{Int64,1}:
```

```
1
2
5
4
3
```

source

Base.reverseind – Function.

```
reverseind(v, i)
```

Given an index `i` in `reverse(v)`, return the corresponding index in `v` so that `v[reverseind(v,i)] == reverse(v)[i]`. (This can be nontrivial in the case where `v` is a Unicode string.)

source

Base.reverse! – Function.

```
reverse!(v [, start=1 [, stop=length(v) ]]) -> v
```

In-place version of `reverse`.

source

50.9 BitArrays

`BitArrays` son matrices booleanas "compactas" eficientes en el uso del espacio, que almacenan un bit por valor booleano. Se pueden usar de forma similar a los arrays `Array{Bool}` (que almacenan un byte por valor booleano), y se pueden convertir a/desde este último a través de `Array(bitarray)` y `BitArray(array)`, respectivamente.

Base.flipbits! – Function.

```
flipbits!(B::BitArray{N}) -> BitArray{N}
```

Performs a bitwise not operation on B. See ~.

Example

```
julia> A = trues(2,2)
2×2 BitArray{2}:
 true  true
 true  true

julia> flipbits!(A)
2×2 BitArray{2}:
 false  false
 false  false
```

source

Base.rol! – Function.

```
rol!(dest::BitVector, src::BitVector, i::Integer) -> BitVector
```

Performs a left rotation operation on `src` and puts the result into `dest`. `i` controls how far to rotate the bits.

source

```
rol!(B::BitVector, i::Integer) -> BitVector
```

Performs a left rotation operation in-place on `B`. `i` controls how far to rotate the bits.

source

`Base.rol` – Function.

```
rol(B::BitVector, i::Integer) -> BitVector
```

Performs a left rotation operation, returning a new `BitVector`. `i` controls how far to rotate the bits. See also `rol!`.

Examples

```
julia> A = BitArray([true, true, false, false, true])
5-element BitArray{1}:
  true
  true
 false
 false
  true

julia> rol(A,1)
5-element BitArray{1}:
  true
 false
 false
  true
  true

julia> rol(A,2)
5-element BitArray{1}:
 false
 false
  true
  true
  true

julia> rol(A,5)
5-element BitArray{1}:
  true
  true
 false
 false
  true
```

source

`Base.ror!` – Function.

```
ror!(dest::BitVector, src::BitVector, i::Integer) -> BitVector
```

Performs a right rotation operation on `src` and puts the result into `dest`. `i` controls how far to rotate the bits.

source

```
ror!(B::BitVector, i::Integer) -> BitVector
```

Performs a right rotation operation in-place on B. `i` controls how far to rotate the bits.

source

`Base.ror` – Function.

```
ror(B::BitVector, i::Integer) -> BitVector
```

Performs a right rotation operation on B, returning a new `BitVector`. `i` controls how far to rotate the bits. See also `ror!`.

Examples

```
julia> A = BitArray([true, true, false, false, true])
5-element BitArray{1}:
  true
  true
 false
 false
  true

julia> ror(A,1)
5-element BitArray{1}:
  true
  true
  true
 false
 false

julia> ror(A,2)
5-element BitArray{1}:
 false
  true
  true
  true
 false

julia> ror(A,5)
5-element BitArray{1}:
  true
  true
 false
 false
  true
```

source

50.10 Matrices y Vectores *Sparse*

Los vectores y las matrices *sparse* soportan ampliamente el mismo conjunto de operaciones que sus contrapartidas densas. Las siguientes funciones son específicas para arrays *sparse*.

`Base.SparseArrays.SparseVector` – Type.

```
SparseVector{Tv,Ti<:Integer} <: AbstractSparseVector{Tv,Ti}
```

Vector type for storing sparse vectors.

source

Base.SparseArrays.SparseMatrixCSC – Type.

```
SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrix{Tv,Ti}
```

Matrix type for storing sparse matrices in the Compressed Sparse Column format.

source

Base.SparseArrays.sparse – Function.

```
sparse(A)
```

Convert an AbstractMatrix A into a sparse matrix.

Example

```
julia> A = eye(3)
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> sparse(A)
3×3 SparseMatrixCSC{Float64,Int64} with 3 stored entries:
  [1, 1]  =  1.0
  [2, 2]  =  1.0
  [3, 3]  =  1.0
```

source

```
sparse(I, J, V,[ m, n, combine])
```

Create a sparse matrix S of dimensions `m x n` such that `S[I[k], J[k]] = V[k]`. The `combine` function is used to combine duplicates. If `m` and `n` are not specified, they are set to `maximum(I)` and `maximum(J)` respectively. If the `combine` function is not supplied, `combine` defaults to `+` unless the elements of `V` are Booleans in which case `combine` defaults to `|`. All elements of `I` must satisfy `1 <= I[k] <= m`, and all elements of `J` must satisfy `1 <= J[k] <= n`. Numerical zeros in (`I`, `J`, `V`) are retained as structural nonzeros; to drop numerical zeros, use dropzeros!.

For additional documentation and an expert driver, see `Base.SparseArrays.sparse!`.

Example

```
julia> Is = [1; 2; 3];

julia> Js = [1; 2; 3];

julia> Vs = [1; 2; 3];

julia> sparse(Is, Js, Vs)
3×3 SparseMatrixCSC{Int64,Int64} with 3 stored entries:
  [1, 1]  =  1
  [2, 2]  =  2
  [3, 3]  =  3
```

source

Base.SparseArrays.sparsevec – Function.

```
sparsevec(I, V, [m, combine])
```

Create a sparse vector S of length m such that S[I[k]] = V[k]. Duplicates are combined using the combine function, which defaults to + if no combine argument is provided, unless the elements of V are Booleans in which case combine defaults to |.

```
julia> II = [1, 3, 3, 5]; V = [0.1, 0.2, 0.3, 0.2];

julia> sparsevec(II, V)
5-element SparseVector{Float64,Int64} with 3 stored entries:
  [1]  =  0.1
  [3]  =  0.5
  [5]  =  0.2

julia> sparsevec(II, V, 8, -)
8-element SparseVector{Float64,Int64} with 3 stored entries:
  [1]  =  0.1
  [3]  =  -0.1
  [5]  =  0.2

julia> sparsevec([1, 3, 1, 2, 2], [true, true, false, false, false])
3-element SparseVector{Bool,Int64} with 3 stored entries:
  [1]  =  true
  [2]  =  false
  [3]  =  true
```

source

```
sparsevec(d::Dict, [m])
```

Create a sparse vector of length m where the nonzero indices are keys from the dictionary, and the nonzero values are the values from the dictionary.

```
julia> sparsevec(Dict(1 => 3, 2 => 2))
2-element SparseVector{Int64,Int64} with 2 stored entries:
  [1]  =  3
  [2]  =  2
```

source

```
sparsevec(A)
```

Convert a vector A into a sparse vector of length m.

Example

```
julia> sparsevec([1.0, 2.0, 0.0, 0.0, 3.0, 0.0])
6-element SparseVector{Float64,Int64} with 3 stored entries:
  [1]  =  1.0
  [2]  =  2.0
  [5]  =  3.0
```

source

Base.SparseArrays.issparse – Function.

```
issparse(S)
```

Returns `true` if S is sparse, and `false` otherwise.

source

Base.full – Function.

```
full(S)
```

Convert a sparse matrix or vector S into a dense matrix or vector.

Example

```
julia> A = speye(3)
3×3 SparseMatrixCSC{Float64,Int64} with 3 stored entries:
  [1, 1]  =  1.0
  [2, 2]  =  1.0
  [3, 3]  =  1.0

julia> full(A)
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

source

Base.SparseArrays.nnz – Function.

```
nnz(A)
```

Returns the number of stored (filled) elements in a sparse array.

Example

```
julia> A = speye(3)
3×3 SparseMatrixCSC{Float64,Int64} with 3 stored entries:
  [1, 1]  =  1.0
  [2, 2]  =  1.0
  [3, 3]  =  1.0

julia> nnz(A)
3
```

source

Base.SparseArrays.spzeros – Function.

```
spzeros([type,]m[,n])
```

Create a sparse vector of length `m` or sparse matrix of size `m x n`. This sparse array will not contain any nonzero values. No storage will be allocated for nonzero values during construction. The type defaults to `Float64` if not specified.

Examples

```
julia> spzeros(3, 3)
3×3 SparseMatrixCSC{Float64,Int64} with 0 stored entries

julia> spzeros(Float32, 4)
4-element SparseVector{Float32,Int64} with 0 stored entries
```

source

Base.SparseArrays.spones – Function.

```
spones(S)
```

Create a sparse array with the same structure as that of S, but with every nonzero element having the value 1.0.

Example

```
julia> A = sparse([1,2,3,4],[2,4,3,1],[5.,4.,3.,2.])
4×4 SparseMatrixCSC{Float64,Int64} with 4 stored entries:
  [4, 1]  =  2.0
  [1, 2]  =  5.0
  [3, 3]  =  3.0
  [2, 4]  =  4.0

julia> spones(A)
4×4 SparseMatrixCSC{Float64,Int64} with 4 stored entries:
  [4, 1]  =  1.0
  [1, 2]  =  1.0
  [3, 3]  =  1.0
  [2, 4]  =  1.0
```

Note the difference from speye.

source

Base.SparseArrays.speye – Method.

```
speye([type,]m[,n])
```

Create a sparse identity matrix of size m x m. When n is supplied, create a sparse identity matrix of size m x n. The type defaults to Float64 if not specified.

sparse(I, m, n) is equivalent to speye(Int, m, n), and sparse(α*I, m, n) can be used to efficiently create a sparse multiple α of the identity matrix.

source

Base.SparseArrays.speye – Method.

```
speye(S)
```

Create a sparse identity matrix with the same size as S.

Example

```
julia> A = sparse([1,2,3,4],[2,4,3,1],[5.,4.,3.,2.])
4×4 SparseMatrixCSC{Float64,Int64} with 4 stored entries:
  [4, 1]  =  2.0
  [1, 2]  =  5.0
```

```
  [3, 3]  =  3.0
  [2, 4]  =  4.0

julia> speye(A)
4×4 SparseMatrixCSC{Float64,Int64} with 4 stored entries:
  [1, 1]  =  1.0
  [2, 2]  =  1.0
  [3, 3]  =  1.0
  [4, 4]  =  1.0
```

Note the difference from spones.

source

```
speye([type,]m[,n])
```

Create a sparse identity matrix of size `m x m`. When n is supplied, create a sparse identity matrix of size `m x n`. The type defaults to `Float64` if not specified.

`sparse(I, m, n)` is equivalent to `speye(Int, m, n)`, and `sparse(α*I, m, n)` can be used to efficiently create a sparse multiple α of the identity matrix.

source

`Base.SparseArrays.spdiagm` – Function.

```
spdiagm(B, d[, m, n])
```

Construct a sparse diagonal matrix. B is a tuple of vectors containing the diagonals and d is a tuple containing the positions of the diagonals. In the case the input contains only one diagonal, B can be a vector (instead of a tuple) and d can be the diagonal position (instead of a tuple), defaulting to 0 (diagonal). Optionally, `m` and n specify the size of the resulting sparse matrix.

Example

```
julia> spdiagm(([1,2,3,4],[4,3,2,1]),(-1,1))
5×5 SparseMatrixCSC{Int64,Int64} with 8 stored entries:
  [2, 1]  =  1
  [1, 2]  =  4
  [3, 2]  =  2
  [2, 3]  =  3
  [4, 3]  =  3
  [3, 4]  =  2
  [5, 4]  =  4
  [4, 5]  =  1
```

source

`Base.SparseArrays.sprand` – Function.

```
sprand([rng],[type],m,[n],p::AbstractFloat,[rfn])
```

Create a random length `m` sparse vector or `m` by n sparse matrix, in which the probability of any element being nonzero is independently given by p (and hence the mean density of nonzeros is also exactly p). Nonzero values are sampled from the distribution specified by `rfn` and have the type `type`. The uniform distribution is used in case `rfn` is not specified. The optional `rng` argument specifies a random number generator, see Random Numbers.

Example

```
julia> rng = MersenneTwister(1234);

julia> sprand(rng, Bool, 2, 2, 0.5)
2×2 SparseMatrixCSC{Bool,Int64} with 2 stored entries:
  [1, 1]  =  true
  [2, 1]  =  true

julia> sprand(rng, Float64, 3, 0.75)
3-element SparseVector{Float64,Int64} with 1 stored entry:
  [3]  =  0.298614
```

source

Base.SparseArrays.sprandn – Function.

```
sprandn([rng], m[,n],p::AbstractFloat)
```

Create a random sparse vector of length m or sparse matrix of size m by n with the specified (independent) probability p of any entry being nonzero, where nonzero values are sampled from the normal distribution. The optional rng argument specifies a random number generator, see Random Numbers.

Example

```
julia> rng = MersenneTwister(1234);

julia> sprandn(rng, 2, 2, 0.75)
2×2 SparseMatrixCSC{Float64,Int64} with 3 stored entries:
  [1, 1]  =  0.532813
  [2, 1]  =  -0.271735
  [2, 2]  =  0.502334
```

source

Base.SparseArrays.nonzeros – Function.

```
nonzeros(A)
```

Return a vector of the structural nonzero values in sparse array A. This includes zeros that are explicitly stored in the sparse array. The returned vector points directly to the internal nonzero storage of A, and any modifications to the returned vector will mutate A as well. See rowvals and nzrange.

Example

```
julia> A = speye(3)
3×3 SparseMatrixCSC{Float64,Int64} with 3 stored entries:
  [1, 1]  =  1.0
  [2, 2]  =  1.0
  [3, 3]  =  1.0

julia> nonzeros(A)
3-element Array{Float64,1}:
 1.0
 1.0
 1.0
```

source

Base.SparseArrays.rowvals – Function.

```
rowvals(A::SparseMatrixCSC)
```

Return a vector of the row indices of A. Any modifications to the returned vector will mutate A as well. Providing access to how the row indices are stored internally can be useful in conjunction with iterating over structural nonzero values. See also nonzeros and nzrange.

Example

```
julia> A = speye(3)
3×3 SparseMatrixCSC{Float64,Int64} with 3 stored entries:
  [1, 1]  =  1.0
  [2, 2]  =  1.0
  [3, 3]  =  1.0

julia> rowvals(A)
3-element Array{Int64,1}:
 1
 2
 3
```

source

Base.SparseArrays.nzrange – Function.

```
nzrange(A::SparseMatrixCSC, col::Integer)
```

Return the range of indices to the structural nonzero values of a sparse matrix column. In conjunction with nonzeros and rowvals, this allows for convenient iterating over a sparse matrix :

```
A = sparse(I,J,V)
rows = rowvals(A)
vals = nonzeros(A)
m, n = size(A)
for i = 1:n
   for j in nzrange(A, i)
      row = rows[j]
      val = vals[j]
      # perform sparse wizardry...
   end
end
```

source

Base.SparseArrays.dropzeros! – Method.

```
dropzeros!(A::SparseMatrixCSC, trim::Bool = true)
```

Removes stored numerical zeros from A, optionally trimming resulting excess space from A.rowval and A.nzval when trim is true.

For an out-of-place version, see dropzeros. For algorithmic information, see fkeep!.

source

Base.SparseArrays.dropzeros – Method.

```
dropzeros(A::SparseMatrixCSC, trim::Bool = true)
```

Generates a copy of `A` and removes stored numerical zeros from that copy, optionally trimming excess space from the result's `rowval` and `nzval` arrays when `trim` is `true`.

For an in-place version and algorithmic information, see `dropzeros!`.

Example

```
julia> A = sparse([1, 2, 3], [1, 2, 3], [1.0, 0.0, 1.0])
3×3 SparseMatrixCSC{Float64,Int64} with 3 stored entries:
  [1, 1]  =  1.0
  [2, 2]  =  0.0
  [3, 3]  =  1.0

julia> dropzeros(A)
3×3 SparseMatrixCSC{Float64,Int64} with 2 stored entries:
  [1, 1]  =  1.0
  [3, 3]  =  1.0
```

source

`Base.SparseArrays.dropzeros!` – Method.

```
dropzeros!(x::SparseVector, trim::Bool = true)
```

Removes stored numerical zeros from `x`, optionally trimming resulting excess space from `x.nzind` and `x.nzval` when `trim` is `true`.

For an out-of-place version, see `dropzeros`. For algorithmic information, see `fkeep!`.

source

`Base.SparseArrays.dropzeros` – Method.

```
dropzeros(x::SparseVector, trim::Bool = true)
```

Generates a copy of `x` and removes numerical zeros from that copy, optionally trimming excess space from the result's `nzind` and `nzval` arrays when `trim` is `true`.

For an in-place version and algorithmic information, see `dropzeros!`.

Example

```
julia> A = sparsevec([1, 2, 3], [1.0, 0.0, 1.0])
3-element SparseVector{Float64,Int64} with 3 stored entries:
  [1]  =  1.0
  [2]  =  0.0
  [3]  =  1.0

julia> dropzeros(A)
3-element SparseVector{Float64,Int64} with 2 stored entries:
  [1]  =  1.0
  [3]  =  1.0
```

source

`Base.SparseArrays.permute` – Function.

```
permute{Tv,Ti}(A::SparseMatrixCSC{Tv,Ti}, p::AbstractVector{<:Integer},
    q::AbstractVector{<:Integer})
```

Bilaterally permute `A`, returning `PAQ` (`A[p,q]`). Column-permutation `q`'s length must match `A`'s column count (`length(q) == A.n`). Row-permutation `p`'s length must match `A`'s row count (`length(p) == A.m`).

For expert drivers and additional information, see `permute!`.

Example

```
julia> A = spdiagm([1, 2, 3, 4], 0, 4, 4) + spdiagm([5, 6, 7], 1, 4, 4)
4×4 SparseMatrixCSC{Int64,Int64} with 7 stored entries:
  [1, 1]  =  1
  [1, 2]  =  5
  [2, 2]  =  2
  [2, 3]  =  6
  [3, 3]  =  3
  [3, 4]  =  7
  [4, 4]  =  4

julia> permute(A, [4, 3, 2, 1], [1, 2, 3, 4])
4×4 SparseMatrixCSC{Int64,Int64} with 7 stored entries:
  [4, 1]  =  1
  [3, 2]  =  2
  [4, 2]  =  5
  [2, 3]  =  3
  [3, 3]  =  6
  [1, 4]  =  4
  [2, 4]  =  7

julia> permute(A, [1, 2, 3, 4], [4, 3, 2, 1])
4×4 SparseMatrixCSC{Int64,Int64} with 7 stored entries:
  [3, 1]  =  7
  [4, 1]  =  4
  [2, 2]  =  6
  [3, 2]  =  3
  [1, 3]  =  5
  [2, 3]  =  2
  [1, 4]  =  1
```

source

`Base.permute!` – Method.

```
permute!{Tv,Ti}(X::SparseMatrixCSC{Tv,Ti}, A::SparseMatrixCSC{Tv,Ti},
    p::AbstractVector{<:Integer}, q::AbstractVector{<:Integer}[, C::SparseMatrixCSC{Tv,Ti}])
```

Bilaterally permute `A`, storing result `PAQ` (`A[p,q]`) in `X`. Stores intermediate result `(AQ)^T` (`transpose(A[:,q])`) in optional argument `C` if present. Requires that none of `X`, `A`, and, if present, `C` alias each other; to store result `PAQ` back into `A`, use the following method lacking `X`:

```
permute!{Tv,Ti}(A::SparseMatrixCSC{Tv,Ti}, p::AbstractVector{<:Integer},
    q::AbstractVector{<:Integer}[, C::SparseMatrixCSC{Tv,Ti}[, workcolptr::Vector{Ti}]])
```

`X`'s dimensions must match those of `A` (`X.m == A.m` and `X.n == A.n`), and `X` must have enough storage to accommodate all allocated entries in `A` (`length(X.rowval) >= nnz(A)` and `length(X.nzval) >= nnz(A)`).

Column-permutation q's length must match A's column count (`length(q) == A.n`). Row-permutation p's length must match A's row count (`length(p) == A.m`).

C's dimensions must match those of `transpose(A)` (`C.m == A.n` and `C.n == A.m`), and C must have enough storage to accommodate all allocated entries in A (`length(C.rowval) >= nnz(A)` and `length(C.nzval) >= nnz(A)`).

For additional (algorithmic) information, and for versions of these methods that forgo argument checking, see (unexported) parent methods `unchecked_noalias_permute!` and `unchecked_aliasing_permute!`.

See also: `permute`.

source

Chapter 51

Tareas y Computación Paralela

51.1 Tareas

`Core.Task` – Type.

```
Task(func)
```

Create a `Task` (i.e. coroutine) to execute the given function (which must be callable with no arguments). The task exits when this function returns.

Example

```
julia> a() = det(rand(1000, 1000));

julia> b = Task(a);
```

In this example, b is a runnable `Task` that hasn't started yet.

source

`Base.current_task` – Function.

```
current_task()
```

Get the currently running `Task`.

source

`Base.istaskdone` – Function.

```
istaskdone(t::Task) -> Bool
```

Determine whether a task has exited.

```
julia> a2() = det(rand(1000, 1000));

julia> b = Task(a2);

julia> istaskdone(b)
false

julia> schedule(b);
```

```
julia> yield();

julia> istaskdone(b)
true
```

source

Base.istaskstarted – Function.

```
istaskstarted(t::Task) -> Bool
```

Determine whether a task has started executing.

```
julia> a3() = det(rand(1000, 1000));

julia> b = Task(a3);

julia> istaskstarted(b)
false
```

source

Base.yield – Function.

```
yield()
```

Switch to the scheduler to allow another scheduled task to run. A task that calls this function is still runnable, and will be restarted immediately if there are no other runnable tasks.

source

```
yield(t::Task, arg = nothing)
```

A fast, unfair-scheduling version of `schedule(t, arg); yield()` which immediately yields to `t` before calling the scheduler.

source

Base.yieldto – Function.

```
yieldto(t::Task, arg = nothing)
```

Switch to the given task. The first time a task is switched to, the task's function is called with no arguments. On subsequent switches, `arg` is returned from the task's last call to `yieldto`. This is a low-level call that only switches tasks, not considering states or scheduling in any way. Its use is discouraged.

source

Base.task_local_storage – Method.

```
task_local_storage(key)
```

Look up the value of a key in the current task's task-local storage.

source

Base.task_local_storage – Method.

```
task_local_storage(key, value)
```

Assign a value to a key in the current task's task-local storage.

source

Base.task_local_storage – Method.

```
task_local_storage(body, key, value)
```

Call the function body with a modified task-local storage, in which value is assigned to key; the previous value of key, or lack thereof, is restored afterwards. Useful for emulating dynamic scoping.

source

Base.Condition – Type.

```
Condition()
```

Create an edge-triggered event source that tasks can wait for. Tasks that call wait on a Condition are suspended and queued. Tasks are woken up when notify is later called on the Condition. Edge triggering means that only tasks waiting at the time notify is called can be woken up. For level-triggered notifications, you must keep extra state to keep track of whether a notification has happened. The Channel type does this, and so can be used for level-triggered events.

source

Base.notify – Function.

```
notify(condition, val=nothing; all=true, error=false)
```

Wake up tasks waiting for a condition, passing them val. If all is true (the default), all waiting tasks are woken, otherwise only one is. If error is true, the passed value is raised as an exception in the woken tasks.

Returns the count of tasks woken up. Returns 0 if no tasks are waiting on condition.

source

Base.schedule – Function.

```
schedule(t::Task, [val]; error=false)
```

Add a Task to the scheduler's queue. This causes the task to run constantly when the system is otherwise idle, unless the task performs a blocking operation such as wait.

If a second argument val is provided, it will be passed to the task (via the return value of yieldto) when it runs again. If error is true, the value is raised as an exception in the woken task.

```
julia> a5() = det(rand(1000, 1000));

julia> b = Task(a5);

julia> istaskstarted(b)
false

julia> schedule(b);

julia> yield();

julia> istaskstarted(b)
true

julia> istaskdone(b)
true
```

source

`Base.@schedule` – Macro.

```
@schedule
```

Wrap an expression in a `Task` and add it to the local machine's scheduler queue. Similar to `@async` except that an enclosing `@sync` does NOT wait for tasks started with an `@schedule`.

source

`Base.@task` – Macro.

```
@task
```

Wrap an expression in a `Task` without executing it, and return the `Task`. This only creates a task, and does not run it.

```
julia> a1() = det(rand(1000, 1000));

julia> b = @task a1();

julia> istaskstarted(b)
false

julia> schedule(b);

julia> yield();

julia> istaskdone(b)
true
```

source

`Base.sleep` – Function.

```
sleep(seconds)
```

Block the current task for a specified number of seconds. The minimum sleep time is 1 millisecond or input of `0.001`.

source

`Base.Channel` – Type.

```
Channel{T}(sz::Int)
```

Constructs a `Channel` with an internal buffer that can hold a maximum of `sz` objects of type T. `put!` calls on a full channel block until an object is removed with `take!`.

`Channel(0)` constructs an unbuffered channel. `put!` blocks until a matching `take!` is called. And vice-versa.

Other constructors:

- `Channel(Inf)`: equivalent to `Channel{Any}(typemax(Int))`
- `Channel(sz)`: equivalent to `Channel{Any}(sz)`

source

Base.put! – Method.

```
put!(c::Channel, v)
```

Appends an item v to the channel c. Blocks if the channel is full.

For unbuffered channels, blocks until a take! is performed by a different task.

source

Base.take! – Method.

```
take!(c::Channel)
```

Removes and returns a value from a Channel. Blocks until data is available.

For unbuffered channels, blocks until a put! is performed by a different task.

source

Base.isready – Method.

```
isready(c::Channel)
```

Determine whether a Channel has a value stored to it. Returns immediately, does not block.

For unbuffered channels returns `true` if there are tasks waiting on a put!.

source

Base.fetch – Method.

```
fetch(c::Channel)
```

Waits for and gets the first available item from the channel. Does not remove the item. `fetch` is unsupported on an unbuffered (0-size) channel.

source

Base.close – Method.

```
close(c::Channel)
```

Closes a channel. An exception is thrown by:

- put! on a closed channel.
- take! and fetch on an empty, closed channel.

source

Base.bind – Method.

```
bind(chnl::Channel, task::Task)
```

Associates the lifetime of `chnl` with a task. Channel `chnl` is automatically closed when the task terminates. Any uncaught exception in the task is propagated to all waiters on `chnl`.

The `chnl` object can be explicitly closed independent of task termination. Terminating tasks have no effect on already closed Channel objects.

When a channel is bound to multiple tasks, the first task to terminate will close the channel. When multiple channels are bound to the same task, termination of the task will close all of the bound channels.

```
julia> c = Channel(0);

julia> task = @schedule foreach(i->put!(c, i), 1:4);

julia> bind(c,task);

julia> for i in c

           @show i

       end;
i = 1
i = 2
i = 3
i = 4

julia> isopen(c)
false
```

```
julia> c = Channel(0);

julia> task = @schedule (put!(c,1);error("foo"));

julia> bind(c,task);

julia> take!(c)
1

julia> put!(c,1);
ERROR: foo
Stacktrace:
 [1] check_channel_state(::Channel{Any}) at ./channels.jl:131
 [2] put!(::Channel{Any}, ::Int64) at ./channels.jl:261
```

source

`Base.asyncmap` – Function.

```
asyncmap(f, c...; ntasks=0, batch_size=nothing)
```

Uses multiple concurrent tasks to map `f` over a collection (or multiple equal length collections). For multiple collection arguments, `f` is applied elementwise.

`ntasks` specifies the number of tasks to run concurrently. Depending on the length of the collections, if `ntasks` is unspecified, up to 100 tasks will be used for concurrent mapping.

`ntasks` can also be specified as a zero-arg function. In this case, the number of tasks to run in parallel is checked before processing every element and a new task started if the value of `ntasks_func()` is less than the current number of tasks.

If `batch_size` is specified, the collection is processed in batch mode. `f` must then be a function that must accept a `Vector` of argument tuples and must return a vector of results. The input vector will have a length of `batch_size` or less.

The following examples highlight execution in different tasks by returning the `object_id` of the tasks in which the mapping function is executed.

First, with `ntasks` undefined, each element is processed in a different task.

```
julia> tskoid() = object_id(current_task());

julia> asyncmap(x->tskoid(), 1:5)
5-element Array{UInt64,1}:
 0x6e15e66c75c75853
 0x440f8819a1baa682
 0x9fb3eeadd0c83985
 0xebd3e35fe90d4050
 0x29efc93edce2b961

julia> length(unique(asyncmap(x->tskoid(), 1:5)))
5
```

With `ntasks=2` all elements are processed in 2 tasks.

```
julia> asyncmap(x->tskoid(), 1:5; ntasks=2)
5-element Array{UInt64,1}:
 0x027ab1680df7ae94
 0xa23d2f80cd7cf157
 0x027ab1680df7ae94
 0xa23d2f80cd7cf157
 0x027ab1680df7ae94

julia> length(unique(asyncmap(x->tskoid(), 1:5; ntasks=2)))
2
```

With `batch_size` defined, the mapping function needs to be changed to accept an array of argument tuples and return an array of results. `map` is used in the modified mapping function to achieve this.

```
julia> batch_func(input) = map(x->string("args_tuple: ", x, ", element_val: ", x[1], ", task:
      ", tskoid()), input)
batch_func (generic function with 1 method)

julia> asyncmap(batch_func, 1:5; ntasks=2, batch_size=2)
5-element Array{String,1}:
 "args_tuple: (1,), element_val: 1, task: 9118321258196414413"
 "args_tuple: (2,), element_val: 2, task: 4904288162898683522"
 "args_tuple: (3,), element_val: 3, task: 9118321258196414413"
 "args_tuple: (4,), element_val: 4, task: 4904288162898683522"
 "args_tuple: (5,), element_val: 5, task: 9118321258196414413"
```

> **Note**
>
> Currently, all tasks in Julia are executed in a single OS thread co-operatively. Consequently, `ayncmap` is beneficial only when the mapping function involves any I/O - disk, network, remote worker invocation, etc.

source

`Base.asyncmap!` – Function.

```
asyncmap!(f, results, c...; ntasks=0, batch_size=nothing)
```

Like `asyncmap()`, but stores output in `results` rather than returning a collection.

source

51.2 Soporte General a la Computación Paralela

`Base.Distributed.addprocs` – Function.

```
addprocs(manager::ClusterManager; kwargs...) -> List of process identifiers
```

Launches worker processes via the specified cluster manager.

For example, Beowulf clusters are supported via a custom cluster manager implemented in the package `ClusterManagers.jl`.

The number of seconds a newly launched worker waits for connection establishment from the master can be specified via variable `JULIA_WORKER_TIMEOUT` in the worker process's environment. Relevant only when using TCP/IP as transport.

source

```
addprocs(machines; tunnel=false, sshflags=``, max_parallel=10, kwargs...) -> List of process
    identifiers
```

Add processes on remote machines via SSH. Requires `julia` to be installed in the same location on each node, or to be available via a shared file system.

`machines` is a vector of machine specifications. Workers are started for each specification.

A machine specification is either a string `machine_spec` or a tuple - `(machine_spec, count)`.

`machine_spec` is a string of the form `[user@]host[:port] [bind_addr[:port]]`. `user` defaults to current user, `port` to the standard ssh port. If `[bind_addr[:port]]` is specified, other workers will connect to this worker at the specified `bind_addr` and `port`.

`count` is the number of workers to be launched on the specified host. If specified as `:auto` it will launch as many workers as the number of cores on the specific host.

Keyword arguments:

- `tunnel`: if `true` then SSH tunneling will be used to connect to the worker from the master process. Default is `false`.
- `sshflags`: specifies additional ssh options, e.g. `sshflags=`-i /home/foo/bar.pem‘`
- `max_parallel`: specifies the maximum number of workers connected to in parallel at a host. Defaults to 10.
- `dir`: specifies the working directory on the workers. Defaults to the host's current directory (as found by `pwd()`)
- `enable_threaded_blas`: if `true` then BLAS will run on multiple threads in added processes. Default is `false`.
- `exename`: name of the `julia` executable. Defaults to "`$JULIA_HOME/julia`" or "`$JULIA_HOME/julia-debug`" as the case may be.
- `exeflags`: additional flags passed to the worker processes.
- `topology`: Specifies how the workers connect to each other. Sending a message between unconnected workers results in an error.
  - `topology=:all_to_all`: All processes are connected to each other. The default.
  - `topology=:master_slave`: Only the driver process, i.e. `pid` 1 connects to the workers. The workers do not connect to each other.

- `topology=:custom`: The `launch` method of the cluster manager specifies the connection topology via fields `ident` and `connect_idents` in `WorkerConfig`. A worker with a cluster manager identity `ident` will connect to all workers specified in `connect_idents`.

Environment variables :

If the master process fails to establish a connection with a newly launched worker within 60.0 seconds, the worker treats it as a fatal situation and terminates. This timeout can be controlled via environment variable `JULIA_WORKER_TIMEOUT`. The value of `JULIA_WORKER_TIMEOUT` on the master process specifies the number of seconds a newly launched worker waits for connection establishment.

source

```
addprocs(; kwargs...) -> List of process identifiers
```

Equivalent to `addprocs(Sys.CPU_CORES; kwargs...)`

Note that workers do not run a `.juliarc.jl` startup script, nor do they synchronize their global state (such as global variables, new method definitions, and loaded modules) with any of the other running processes.

source

```
addprocs(np::Integer; restrict=true, kwargs...) -> List of process identifiers
```

Launches workers using the in-built `LocalManager` which only launches workers on the local host. This can be used to take advantage of multiple cores. `addprocs(4)` will add 4 processes on the local machine. If `restrict` is `true`, binding is restricted to `127.0.0.1`. Keyword args `dir`, `exename`, `exeflags`, `topology`, and `enable_threaded_blas` have the same effect as documented for `addprocs(machines)`.

source

`Base.Distributed.nprocs` – Function.

```
nprocs()
```

Get the number of available processes.

source

`Base.Distributed.nworkers` – Function.

```
nworkers()
```

Get the number of available worker processes. This is one less than `nprocs()`. Equal to `nprocs()` if `nprocs() == 1`.

source

`Base.Distributed.procs` – Method.

```
procs()
```

Returns a list of all process identifiers.

source

`Base.Distributed.procs` – Method.

```
procs(pid::Integer)
```

Returns a list of all process identifiers on the same physical node. Specifically all workers bound to the same ip-address as `pid` are returned.

source

`Base.Distributed.workers` – Function.

```
workers()
```

Returns a list of all worker process identifiers.

source

`Base.Distributed.rmprocs` – Function.

```
rmprocs(pids...; waitfor=typemax(Int))
```

Removes the specified workers. Note that only process 1 can add or remove workers.

Argument `waitfor` specifies how long to wait for the workers to shut down: - If unspecified, `rmprocs` will wait until all requested `pids` are removed. - An `ErrorException` is raised if all workers cannot be terminated before the requested `waitfor` seconds. - With a `waitfor` value of 0, the call returns immediately with the workers scheduled for removal in a different task. The scheduled `Task` object is returned. The user should call `wait` on the task before invoking any other parallel calls.

source

`Base.Distributed.interrupt` – Function.

```
interrupt(pids::Integer...)
```

Interrupt the current executing task on the specified workers. This is equivalent to pressing Ctrl-C on the local machine. If no arguments are given, all workers are interrupted.

source

```
interrupt(pids::AbstractVector=workers())
```

Interrupt the current executing task on the specified workers. This is equivalent to pressing Ctrl-C on the local machine. If no arguments are given, all workers are interrupted.

source

`Base.Distributed.myid` – Function.

```
myid()
```

Get the id of the current process.

source

`Base.Distributed.pmap` – Function.

```
pmap([::AbstractWorkerPool], f, c...; distributed=true, batch_size=1, on_error=nothing,
     retry_delays=[]), retry_check=nothing) -> collection
```

Transform collection `c` by applying `f` to each element using available workers and tasks.

For multiple collection arguments, apply `f` elementwise.

Note that `f` must be made available to all worker processes; see Code Availability and Loading Packages for details.

If a worker pool is not specified, all available workers, i.e., the default worker pool is used.

By default, pmap distributes the computation over all specified workers. To use only the local process and distribute over tasks, specify `distributed=false`. This is equivalent to using `asyncmap`. For example, `pmap(f, c; distributed=false)` is equivalent to `asyncmap(f,c; ntasks=()->nworkers())`

pmap can also use a mix of processes and tasks via the `batch_size` argument. For batch sizes greater than 1, the collection is processed in multiple batches, each of length `batch_size` or less. A batch is sent as a single request to a free worker, where a local `asyncmap` processes elements from the batch using multiple concurrent tasks.

Any error stops pmap from processing the remainder of the collection. To override this behavior you can specify an error handling function via argument `on_error` which takes in a single argument, i.e., the exception. The function can stop the processing by rethrowing the error, or, to continue, return any value which is then returned inline with the results to the caller.

Consider the following two examples. The first one returns the exception object inline, the second a 0 in place of any exception:

```
julia> pmap(x->iseven(x) ? error("foo") : x, 1:4; on_error=identity)
4-element Array{Any,1}:
 1
  ErrorException("foo")
 3
  ErrorException("foo")

julia> pmap(x->iseven(x) ? error("foo") : x, 1:4; on_error=ex->0)
4-element Array{Int64,1}:
 1
 0
 3
 0
```

Errors can also be handled by retrying failed computations. Keyword arguments `retry_delays` and `retry_check` are passed through to `retry` as keyword arguments `delays` and `check` respectively. If batching is specified, and an entire batch fails, all items in the batch are retried.

Note that if both `on_error` and `retry_delays` are specified, the `on_error` hook is called before retrying. If `on_error` does not throw (or rethrow) an exception, the element will not be retried.

Example: On errors, retry `f` on an element a maximum of 3 times without any delay between retries.

```
pmap(f, c; retry_delays = zeros(3))
```

Example: Retry `f` only if the exception is not of type `InexactError`, with exponentially increasing delays up to 3 times. Return a `NaN` in place for all `InexactError` occurrences.

```
pmap(f, c; on_error = e->(isa(e, InexactError) ? NaN : rethrow(e)), retry_delays =
↪  ExponentialBackOff(n = 3))
```

source

`Base.Distributed.RemoteException` – Type.

```
RemoteException(captured)
```

Exceptions on remote computations are captured and rethrown locally. A `RemoteException` wraps the `pid` of the worker and a captured exception. A `CapturedException` captures the remote exception and a serializable form of the call stack when the exception was raised.

source

`Base.Distributed.Future` – Type.

```
Future(pid::Integer=myid())
```

Create a `Future` on process `pid`. The default `pid` is the current process.

source

`Base.Distributed.RemoteChannel` – Method.

```
RemoteChannel(pid::Integer=myid())
```

Make a reference to a `Channel{Any}(1)` on process `pid`. The default `pid` is the current process.

source

`Base.Distributed.RemoteChannel` – Method.

```
RemoteChannel(f::Function, pid::Integer=myid())
```

Create references to remote channels of a specific size and type. `f()` is a function that when executed on `pid` must return an implementation of an `AbstractChannel`.

For example, `RemoteChannel(()->Channel{Int}(10), pid)`, will return a reference to a channel of type `Int` and size 10 on `pid`.

The default `pid` is the current process.

source

`Base.wait` – Function.

```
wait([x])
```

Block the current task until some event occurs, depending on the type of the argument:

- `RemoteChannel` : Wait for a value to become available on the specified remote channel.
- `Future` : Wait for a value to become available for the specified future.
- `Channel`: Wait for a value to be appended to the channel.
- `Condition`: Wait for `notify` on a condition.
- `Process`: Wait for a process or process chain to exit. The `exitcode` field of a process can be used to determine success or failure.
- `Task`: Wait for a `Task` to finish, returning its result value. If the task fails with an exception, the exception is propagated (re-thrown in the task that called `wait`).
- `RawFD`: Wait for changes on a file descriptor (see `poll_fd` for keyword arguments and return code)

If no argument is passed, the task blocks for an undefined period. A task can only be restarted by an explicit call to `schedule` or `yieldto`.

Often `wait` is called within a `while` loop to ensure a waited-for condition is met before proceeding.

source

`Base.fetch` – Method.

```
fetch(x)
```

Waits and fetches a value from x depending on the type of x:

- Future: Wait for and get the value of a Future. The fetched value is cached locally. Further calls to fetch on the same reference return the cached value. If the remote value is an exception, throws a RemoteException which captures the remote exception and backtrace.
- RemoteChannel: Wait for and get the value of a remote reference. Exceptions raised are same as for a Future .

Does not remove the item fetched.

source

Base.Distributed.remotecall – Method.

```
remotecall(f, id::Integer, args...; kwargs...) -> Future
```

Call a function f asynchronously on the given arguments on the specified process. Returns a Future. Keyword arguments, if any, are passed through to f.

source

Base.Distributed.remotecall_wait – Method.

```
remotecall_wait(f, id::Integer, args...; kwargs...)
```

Perform a faster wait(remotecall(...)) in one message on the Worker specified by worker id id. Keyword arguments, if any, are passed through to f.

See also wait and remotecall.

source

Base.Distributed.remotecall_fetch – Method.

```
remotecall_fetch(f, id::Integer, args...; kwargs...)
```

Perform fetch(remotecall(...)) in one message. Keyword arguments, if any, are passed through to f. Any remote exceptions are captured in a RemoteException and thrown.

See also fetch and remotecall.

source

Base.Distributed.remote_do – Method.

```
remote_do(f, id::Integer, args...; kwargs...) -> nothing
```

Executes f on worker id asynchronously. Unlike remotecall, it does not store the result of computation, nor is there a way to wait for its completion.

A successful invocation indicates that the request has been accepted for execution on the remote node.

While consecutive remotecalls to the same worker are serialized in the order they are invoked, the order of executions on the remote worker is undetermined. For example, remote_do(f1, 2); remotecall(f2, 2); remote_do(f3, 2) will serialize the call to f1, followed by f2 and f3 in that order. However, it is not guaranteed that f1 is executed before f3 on worker 2.

Any exceptions thrown by f are printed to STDERR on the remote worker.

Keyword arguments, if any, are passed through to f.

source

`Base.put!` – Method.

```
put!(rr::RemoteChannel, args...)
```

Store a set of values to the `RemoteChannel`. If the channel is full, blocks until space is available. Returns its first argument.

source

`Base.put!` – Method.

```
put!(rr::Future, v)
```

Store a value to a `Future` `rr`. `Futures` are write-once remote references. A `put!` on an already set `Future` throws an `Exception`. All asynchronous remote calls return `Futures` and set the value to the return value of the call upon completion.

source

`Base.take!` – Method.

```
take!(rr::RemoteChannel, args...)
```

Fetch value(s) from a `RemoteChannel` `rr`, removing the value(s) in the processs.

source

`Base.isready` – Method.

```
isready(rr::RemoteChannel, args...)
```

Determine whether a `RemoteChannel` has a value stored to it. Note that this function can cause race conditions, since by the time you receive its result it may no longer be true. However, it can be safely used on a `Future` since they are assigned only once.

source

`Base.isready` – Method.

```
isready(rr::Future)
```

Determine whether a `Future` has a value stored to it.

If the argument `Future` is owned by a different node, this call will block to wait for the answer. It is recommended to wait for `rr` in a separate task instead or to use a local `Channel` as a proxy:

```
c = Channel(1)
@async put!(c, remotecall_fetch(long_computation, p))
isready(c)  # will not block
```

source

`Base.Distributed.WorkerPool` – Type.

```
WorkerPool(workers::Vector{Int})
```

Create a WorkerPool from a vector of worker ids.

source

Base.Distributed.CachingPool – Type.

```
CachingPool(workers::Vector{Int})
```

An implementation of an AbstractWorkerPool. remote, remotecall_fetch, pmap (and other remote calls which execute functions remotely) benefit from caching the serialized/deserialized functions on the worker nodes, especially closures (which may capture large amounts of data).

The remote cache is maintained for the lifetime of the returned CachingPool object. To clear the cache earlier, use clear!(pool).

For global variables, only the bindings are captured in a closure, not the data. let blocks can be used to capture global data.

For example:

```
const foo=rand(10^8);
wp=CachingPool(workers())
let foo=foo
    pmap(wp, i->sum(foo)+i, 1:100);
end
```

The above would transfer foo only once to each worker.

source

Base.Distributed.default_worker_pool – Function.

```
default_worker_pool()
```

WorkerPool containing idle workers() - used by remote(f) and pmap (by default).

source

Base.Distributed.clear! – Method.

```
clear!(pool::CachingPool) -> pool
```

Removes all cached functions from all participating workers.

source

Base.Distributed.remote – Function.

```
remote([::AbstractWorkerPool], f) -> Function
```

Returns an anonymous function that executes function f on an available worker using remotecall_fetch.

source

Base.Distributed.remotecall – Method.

```
remotecall(f, pool::AbstractWorkerPool, args...; kwargs...) -> Future
```

WorkerPool variant of remotecall(f, pid, ...). Waits for and takes a free worker from pool and performs a remotecall on it.

source

Base.Distributed.remotecall_wait – Method.

```
remotecall_wait(f, pool::AbstractWorkerPool, args...; kwargs...) -> Future
```

`WorkerPool` variant of `remotecall_wait(f, pid, ....)`. Waits for and takes a free worker from `pool` and performs a `remotecall_wait` on it.

source

`Base.Distributed.remotecall_fetch` – Method.

```
remotecall_fetch(f, pool::AbstractWorkerPool, args...; kwargs...) -> result
```

`WorkerPool` variant of `remotecall_fetch(f, pid, ....)`. Waits for and takes a free worker from `pool` and performs a `remotecall_fetch` on it.

source

`Base.Distributed.remote_do` – Method.

```
remote_do(f, pool::AbstractWorkerPool, args...; kwargs...) -> nothing
```

`WorkerPool` variant of `remote_do(f, pid, ....)`. Waits for and takes a free worker from `pool` and performs a `remote_do` on it.

source

`Base.timedwait` – Function.

```
timedwait(testcb::Function, secs::Float64; pollint::Float64=0.1)
```

Waits until `testcb` returns `true` or for `secs` seconds, whichever is earlier. `testcb` is polled every `pollint` seconds.

source

`Base.Distributed.@spawn` – Macro.

```
@spawn
```

Creates a closure around an expression and runs it on an automatically-chosen process, returning a `Future` to the result.

source

`Base.Distributed.@spawnat` – Macro.

```
@spawnat
```

Accepts two arguments, p and an expression. A closure is created around the expression and run asynchronously on process p. Returns a `Future` to the result.

source

`Base.Distributed.@fetch` – Macro.

```
@fetch
```

Equivalent to `fetch(@spawn expr)`. See `fetch` and `@spawn`.

source

`Base.Distributed.@fetchfrom` – Macro.

```
@fetchfrom
```

Equivalent to `fetch(@spawnat p expr)`. See `fetch` and `@spawnat`.

source

`Base.@async` – Macro.

```
@async
```

Like `@schedule`, `@async` wraps an expression in a `Task` and adds it to the local machine's scheduler queue. Additionally it adds the task to the set of items that the nearest enclosing `@sync` waits for.

source

`Base.@sync` – Macro.

```
@sync
```

Wait until all dynamically-enclosed uses of `@async`, `@spawn`, `@spawnat` and `@parallel` are complete. All exceptions thrown by enclosed async operations are collected and thrown as a `CompositeException`.

source

`Base.Distributed.@parallel` – Macro.

```
@parallel
```

A parallel for loop of the form :

```
@parallel [reducer] for var = range
    body
end
```

The specified range is partitioned and locally executed across all workers. In case an optional reducer function is specified, `@parallel` performs local reductions on each worker with a final reduction on the calling process.

Note that without a reducer function, `@parallel` executes asynchronously, i.e. it spawns independent tasks on all available workers and returns immediately without waiting for completion. To wait for completion, prefix the call with `@sync`, like :

```
@sync @parallel for var = range
    body
end
```

source

`Base.Distributed.@everywhere` – Macro.

```
@everywhere expr
```

Execute an expression under `Main` everywhere. Equivalent to calling `eval(Main, expr)` on all processes. Errors on any of the processes are collected into a `CompositeException` and thrown. For example :

```
@everywhere bar=1
```

will define `Main.bar` on all processes.

Unlike `@spawn` and `@spawnat`, `@everywhere` does not capture any local variables. Prefixing `@everywhere` with `@eval` allows us to broadcast local variables using interpolation :

```
foo = 1
@eval @everywhere bar=$foo
```

The expression is evaluated under `Main` irrespective of where `@everywhere` is called from. For example :

```
module FooBar
    foo() = @everywhere bar()=myid()
end
FooBar.foo()
```

will result in `Main.bar` being defined on all processes and not `FooBar.bar`.

source

`Base.Distributed.clear!` – Method.

```
clear!(syms, pids=workers(); mod=Main)
```

Clears global bindings in modules by initializing them to `nothing`. `syms` should be of type `Symbol` or a collection of `Symbols`. `pids` and `mod` identify the processes and the module in which global variables are to be reinitialized. Only those names found to be defined under `mod` are cleared.

An exception is raised if a global constant is requested to be cleared.

source

`Base.Distributed.remoteref_id` – Function.

```
Base.remoteref_id(r::AbstractRemoteRef) -> RRID
```

`Futures` and `RemoteChannels` are identified by fields:

- `where` - refers to the node where the underlying object/storage referred to by the reference actually exists.
- `whence` - refers to the node the remote reference was created from. Note that this is different from the node where the underlying object referred to actually exists. For example calling `RemoteChannel(2)` from the master process would result in a `where` value of 2 and a `whence` value of 1.
- `id` is unique across all references created from the worker specified by `whence`.

Taken together, `whence` and `id` uniquely identify a reference across all workers.

`Base.remoteref_id` is a low-level API which returns a `Base.RRID` object that wraps `whence` and `id` values of a remote reference.

source

`Base.Distributed.channel_from_id` – Function.

```
Base.channel_from_id(id) -> c
```

A low-level API which returns the backing `AbstractChannel` for an `id` returned by `remoteref_id`. The call is valid only on the node where the backing channel exists.

source

`Base.Distributed.worker_id_from_socket` – Function.

```
Base.worker_id_from_socket(s) -> pid
```

A low-level API which given a `IO` connection or a `Worker`, returns the `pid` of the worker it is connected to. This is useful when writing custom `serialize` methods for a type, which optimizes the data written out depending on the receiving process id.

source

`Base.Distributed.cluster_cookie` – Method.

```
Base.cluster_cookie() -> cookie
```

Returns the cluster cookie.

source

`Base.Distributed.cluster_cookie` – Method.

```
Base.cluster_cookie(cookie) -> cookie
```

Sets the passed cookie as the cluster cookie, then returns it.

source

51.3 Arrays Compartidos

`Base.SharedArray` – Type.

```
SharedArray{T}(dims::NTuple; init=false, pids=Int[])
SharedArray{T,N}(...)
```

Construct a `SharedArray` of a bits type `T` and size `dims` across the processes specified by `pids` - all of which have to be on the same host. If `N` is specified by calling `SharedArray{T,N}(dims)`, then `N` must match the length of `dims`.

If `pids` is left unspecified, the shared array will be mapped across all processes on the current host, including the master. But, `localindexes` and `indexpids` will only refer to worker processes. This facilitates work distribution code to use workers for actual computation with the master process acting as a driver.

If an `init` function of the type `initfn(S::SharedArray)` is specified, it is called on all the participating workers.

The shared array is valid as long as a reference to the `SharedArray` object exists on the node which created the mapping.

```
SharedArray{T}(filename::AbstractString, dims::NTuple, [offset=0]; mode=nothing, init=false,
    pids=Int[])
SharedArray{T,N}(...)
```

Construct a `SharedArray` backed by the file `filename`, with element type `T` (must be a bits type) and size `dims`, across the processes specified by `pids` - all of which have to be on the same host. This file is mmapped into the host memory, with the following consequences:

- The array data must be represented in binary format (e.g., an ASCII format like CSV cannot be supported)
- Any changes you make to the array values (e.g., `A[3] = 0`) will also change the values on disk

If `pids` is left unspecified, the shared array will be mapped across all processes on the current host, including the master. But, `localindexes` and `indexpids` will only refer to worker processes. This facilitates work distribution code to use workers for actual computation with the master process acting as a driver.

mode must be one of "r", "r+", "w+", or "a+", and defaults to "r+" if the file specified by filename already exists, or "w+" if not. If an init function of the type initfn(S::SharedArray) is specified, it is called on all the participating workers. You cannot specify an init function if the file is not writable.

offset allows you to skip the specified number of bytes at the beginning of the file.

source

Base.Distributed.procs – Method.

```
procs(S::SharedArray)
```

Get the vector of processes mapping the shared array.

source

Base.sdata – Function.

```
sdata(S::SharedArray)
```

Returns the actual Array object backing S.

source

Base.indexpids – Function.

```
indexpids(S::SharedArray)
```

Returns the current worker's index in the list of workers mapping the SharedArray (i.e. in the same list returned by procs(S)), or 0 if the SharedArray is not mapped locally.

source

Base.localindexes – Function.

```
localindexes(S::SharedArray)
```

Returns a range describing the "default" indexes to be handled by the current process. This range should be interpreted in the sense of linear indexing, i.e., as a sub-range of 1:length(S). In multi-process contexts, returns an empty range in the parent process (or any process for which indexpids returns 0).

It's worth emphasizing that localindexes exists purely as a convenience, and you can partition work on the array among workers any way you wish. For a SharedArray, all indexes should be equally fast for each worker process.

source

51.4 Multi-Hilo

Este interfaz experimental soporta las capacidades multi-hilo de Julia. Los tipos y funciones descritos aquí pueden cambiar en el futuro (y probablemente lo harán).

Base.Threads.threadid – Function.

```
Threads.threadid()
```

Get the ID number of the current thread of execution. The master thread has ID 1.

source

`Base.Threads.nthreads` – Function.

```
Threads.nthreads()
```

Get the number of threads available to the Julia process. This is the inclusive upper bound on `threadid()`.

source

`Base.Threads.@threads` – Macro.

```
Threads.@threads
```

A macro to parallelize a for-loop to run with multiple threads. This spawns `nthreads()` number of threads, splits the iteration space amongst them, and iterates in parallel. A barrier is placed at the end of the loop which waits for all the threads to finish execution, and the loop returns.

source

`Base.Threads.Atomic` – Type.

```
Threads.Atomic{T}
```

Holds a reference to an object of type `T`, ensuring that it is only accessed atomically, i.e. in a thread-safe manner.

Only certain "simple" types can be used atomically, namely the primitive integer and float-point types. These are `Int8`...`Int128`, `UInt8`...`UInt128`, and `Float16`...`Float64`.

New atomic objects can be created from a non-atomic values; if none is specified, the atomic object is initialized with zero.

Atomic objects can be accessed using the `[]` notation:

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> x[] = 1
1

julia> x[]
1
```

Atomic operations use an `atomic_` prefix, such as `atomic_add!`, `atomic_xchg!`, etc.

source

`Base.Threads.atomic_cas!` – Function.

```
Threads.atomic_cas!{T}(x::Atomic{T}, cmp::T, newval::T)
```

Atomically compare-and-set `x`

Atomically compares the value in `x` with `cmp`. If equal, write `newval` to `x`. Otherwise, leaves `x` unmodified. Returns the old value in `x`. By comparing the returned value to `cmp` (via `===`) one knows whether `x` was modified and now holds the new value `newval`.

For further details, see LLVM's `cmpxchg` instruction.

This function can be used to implement transactional semantics. Before the transaction, one records the value in `x`. After the transaction, the new value is stored only if `x` has not been modified in the mean time.

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_cas!(x, 4, 2);

julia> x
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_cas!(x, 3, 2);

julia> x
Base.Threads.Atomic{Int64}(2)
```

source

Base.Threads.atomic_xchg! – Function.

```
Threads.atomic_xchg!{T}(x::Atomic{T}, newval::T)
```

Atomically exchange the value in x

Atomically exchanges the value in x with `newval`. Returns the **old** value.

For further details, see LLVM's `atomicrmw xchg` instruction.

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_xchg!(x, 2)
3

julia> x[]
2
```

source

Base.Threads.atomic_add! – Function.

```
Threads.atomic_add!{T}(x::Atomic{T}, val::T)
```

Atomically add `val` to x

Performs `x[] += val` atomically. Returns the **old** value.

For further details, see LLVM's `atomicrmw add` instruction.

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_add!(x, 2)
3

julia> x[]
5
```

source

Base.Threads.atomic_sub! – Function.

```
Threads.atomic_sub!{T}(x::Atomic{T}, val::T)
```

Atomically subtract `val` from `x`

Performs `x[] -= val` atomically. Returns the **old** value.

For further details, see LLVM's `atomicrmw sub` instruction.

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_sub!(x, 2)
3

julia> x[]
1
```

source

`Base.Threads.atomic_and!` – Function.

```
Threads.atomic_and!{T}(x::Atomic{T}, val::T)
```

Atomically bitwise-and `x` with `val`

Performs `x[] &= val` atomically. Returns the **old** value.

For further details, see LLVM's `atomicrmw and` instruction.

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_and!(x, 2)
3

julia> x[]
2
```

source

`Base.Threads.atomic_nand!` – Function.

```
Threads.atomic_nand!{T}(x::Atomic{T}, val::T)
```

Atomically bitwise-nand (not-and) `x` with `val`

Performs `x[] = ~(x[] & val)` atomically. Returns the **old** value.

For further details, see LLVM's `atomicrmw nand` instruction.

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_nand!(x, 2)
3

julia> x[]
-3
```

source

Base.Threads.atomic_or! – Function.

```
Threads.atomic_or!{T}(x::Atomic{T}, val::T)
```

Atomically bitwise-or x with `val`

Performs `x[] |= val` atomically. Returns the **old** value.

For further details, see LLVM's `atomicrmw or` instruction.

```
julia> x = Threads.Atomic{Int}(5)
Base.Threads.Atomic{Int64}(5)

julia> Threads.atomic_or!(x, 7)
5

julia> x[]
7
```

source

Base.Threads.atomic_xor! – Function.

```
Threads.atomic_xor!{T}(x::Atomic{T}, val::T)
```

Atomically bitwise-xor (exclusive-or) x with `val`

Performs `x[] $= val` atomically. Returns the **old** value.

For further details, see LLVM's `atomicrmw xor` instruction.

```
julia> x = Threads.Atomic{Int}(5)
Base.Threads.Atomic{Int64}(5)

julia> Threads.atomic_xor!(x, 7)
5

julia> x[]
2
```

source

Base.Threads.atomic_max! – Function.

```
Threads.atomic_max!{T}(x::Atomic{T}, val::T)
```

Atomically store the maximum of x and `val` in x

Performs `x[] = max(x[], val)` atomically. Returns the **old** value.

For further details, see LLVM's `atomicrmw max` instruction.

```
julia> x = Threads.Atomic{Int}(5)
Base.Threads.Atomic{Int64}(5)

julia> Threads.atomic_max!(x, 7)
5

julia> x[]
7
```

source

Base.Threads.atomic_min! – Function.

```
Threads.atomic_min!{T}(x::Atomic{T}, val::T)
```

Atomically store the minimum of `x` and `val` in `x`

Performs `x[] = min(x[], val)` atomically. Returns the **old** value.

For further details, see LLVM's `atomicrmw min` instruction.

```
julia> x = Threads.Atomic{Int}(7)
Base.Threads.Atomic{Int64}(7)

julia> Threads.atomic_min!(x, 5)
7

julia> x[]
5
```

source

Base.Threads.atomic_fence – Function.

```
Threads.atomic_fence()
```

Insert a sequential-consistency memory fence

Inserts a memory fence with sequentially-consistent ordering semantics. There are algorithms where this is needed, i.e. where an acquire/release ordering is insufficient.

This is likely a very expensive operation. Given that all other atomic operations in Julia already have acquire/release semantics, explicit fences should not be necessary in most cases.

For further details, see LLVM's `fence` instruction.

source

51.5 ccall utilianzdo una threadpool (Experimental)

Base.@threadcall – Macro.

```
@threadcall((cfunc, clib), rettype, (argtypes...), argvals...)
```

The `@threadcall` macro is called in the same way as `ccall` but does the work in a different thread. This is useful when you want to call a blocking C function without causing the main `julia` thread to become blocked. Concurrency is limited by size of the libuv thread pool, which defaults to 4 threads but can be increased by setting the `UV_THREADPOOL_SIZE` environment variable and restarting the `julia` process.

Note that the called function should never call back into Julia.

source

51.6 Primitivas de Sincronización

`Base.Threads.AbstractLock` – Type.

```
AbstractLock
```

Abstract supertype describing types that implement the thread-safe synchronization primitives: `lock`, `trylock`, `unlock`, and `islocked`

source

`Base.lock` – Function.

```
lock(the_lock)
```

Acquires the lock when it becomes available. If the lock is already locked by a different task/thread, it waits for it to become available.

Each `lock` must be matched by an `unlock`.

source

`Base.unlock` – Function.

```
unlock(the_lock)
```

Releases ownership of the lock.

If this is a recursive lock which has been acquired before, it just decrements an internal counter and returns immediately.

source

`Base.trylock` – Function.

```
trylock(the_lock) -> Success (Boolean)
```

Acquires the lock if it is available, returning `true` if successful. If the lock is already locked by a different task/thread, returns `false`.

Each successful `trylock` must be matched by an `unlock`.

source

`Base.islocked` – Function.

```
islocked(the_lock) -> Status (Boolean)
```

Check whether the lock is held by any task/thread. This should not be used for synchronization (see instead `trylock`).

source

`Base.ReentrantLock` – Type.

```
ReentrantLock()
```

Creates a reentrant lock for synchronizing Tasks. The same task can acquire the lock as many times as required. Each `lock` must be matched with an `unlock`.

This lock is NOT threadsafe. See `Threads.Mutex` for a threadsafe lock.

source

Base.Threads.Mutex – Type.

```
Mutex()
```

These are standard system mutexes for locking critical sections of logic.

On Windows, this is a critical section object, on pthreads, this is a `pthread_mutex_t`.

See also SpinLock for a lighter-weight lock.

source

Base.Threads.SpinLock – Type.

```
SpinLock()
```

Creates a non-reentrant lock. Recursive use will result in a deadlock. Each `lock` must be matched with an `unlock`.

Test-and-test-and-set spin locks are quickest up to about 30ish contending threads. If you have more contention than that, perhaps a lock is the wrong way to synchronize.

See also RecursiveSpinLock for a version that permits recursion.

See also Mutex for a more efficient version on one core or if the lock may be held for a considerable length of time.

source

Base.Threads.RecursiveSpinLock – Type.

```
RecursiveSpinLock()
```

Creates a reentrant lock. The same thread can acquire the lock as many times as required. Each `lock` must be matched with an `unlock`.

See also SpinLock for a slightly faster version.

See also Mutex for a more efficient version on one core or if the lock may be held for a considerable length of time.

source

Base.Semaphore – Type.

```
Semaphore(sem_size)
```

Creates a counting semaphore that allows at most `sem_size` acquires to be in use at any time. Each acquire must be mached with a release.

This construct is NOT threadsafe.

source

Base.acquire – Function.

```
acquire(s::Semaphore)
```

Wait for one of the `sem_size` permits to be available, blocking until one can be acquired.

source

Base.release – Function.

```
release(s::Semaphore)
```

Return one permit to the pool, possibly allowing another task to acquire it and resume execution.

source

51.7 Interfaz de Administración de Cluster

Esta interfaz proporciona un mecanismo para lanzar y gestionar *workers* Julia sobre diferentes entornos cluster. Hay dos tipos de administrafores presentes en Base: `LocalManager`, para lanzar *workers* adicionales sobre el mismo host, y `SSHManager`, para lanzarlos sobre hosts remotos vía `ssh`. Para conectar y transportar mensajes entre procesos se usan los sockets TCP/IP. Es posible que los administradores de clusters proporcionen un transporte diferente.

`Base.Distributed.launch` – Function.

```
launch(manager::ClusterManager, params::Dict, launched::Array, launch_ntfy::Condition)
```

Implemented by cluster managers. For every Julia worker launched by this function, it should append a `WorkerConfig` entry to `launched` and notify `launch_ntfy`. The function MUST exit once all workers, requested by `manager` have been launched. `params` is a dictionary of all keyword arguments `addprocs` was called with.

source

`Base.Distributed.manage` – Function.

```
manage(manager::ClusterManager, id::Integer, config::WorkerConfig. op::Symbol)
```

Implemented by cluster managers. It is called on the master process, during a worker's lifetime, with appropriate op values:

- with `:register`/`:deregister` when a worker is added / removed from the Julia worker pool.
- with `:interrupt` when `interrupt(workers)` is called. The `ClusterManager` should signal the appropriate worker with an interrupt signal.
- with `:finalize` for cleanup purposes.

source

`Base.kill` – Method.

```
kill(manager::ClusterManager, pid::Int, config::WorkerConfig)
```

Implemented by cluster managers. It is called on the master process, by `rmprocs`. It should cause the remote worker specified by `pid` to exit. `kill(manager::ClusterManager.....)` executes a remote `exit()` on `pid`.

source

`Base.Distributed.init_worker` – Function.

```
init_worker(cookie::AbstractString, manager::ClusterManager=DefaultClusterManager())
```

Called by cluster managers implementing custom transports. It initializes a newly launched process as a worker. Command line argument `--worker` has the effect of initializing a process as a worker using TCP/IP sockets for transport. `cookie` is a `cluster_cookie`.

source

`Base.connect` – Method.

```
connect(manager::ClusterManager, pid::Int, config::WorkerConfig) -> (instrm::IO, outstrm::IO)
```

Implemented by cluster managers using custom transports. It should establish a logical connection to worker with id `pid`, specified by `config` and return a pair of `IO` objects. Messages from `pid` to current process will be read off `instrm`, while messages to be sent to `pid` will be written to `outstrm`. The custom transport implementation must ensure that messages are delivered and received completely and in order. `connect(manager::ClusterManager.....)` sets up TCP/IP socket connections in-between workers.

source

`Base.Distributed.process_messages` – Function.

```
Base.process_messages(r_stream::IO, w_stream::IO, incoming::Bool=true)
```

Called by cluster managers using custom transports. It should be called when the custom transport implementation receives the first message from a remote worker. The custom transport must manage a logical connection to the remote worker and provide two `IO` objects, one for incoming messages and the other for messages addressed to the remote worker. If `incoming` is `true`, the remote peer initiated the connection. Whichever of the pair initiates the connection sends the cluster cookie and its Julia version number to perform the authentication handshake.

See also `cluster_cookie`.

source

Chapter 52

Álgebra Lineal

52.1 Funciones Estándar

Las funciones de álgebra lineal en Julia están ampliamente implementadas llamando a funciones de LAPACK. Las factorizaciones *sparse* llaman a funciones de SuiteSparse

`Base.:*` – Method.

```
*(x, y...)
```

Multiplication operator. `x*y*z*...` calls this function with all arguments, i.e. `*(x, y, z, ...)`.

source

`Base.:\` – Method.

```
\(x, y)
```

Left division operator: multiplication of y by the inverse of x on the left. Gives floating-point results for integer arguments.

```
julia> 3 \ 6
2.0

julia> inv(3) * 6
2.0

julia> A = [1 2; 3 4]; x = [5, 6];

julia> A \ x
2-element Array{Float64,1}:
 -4.0
  4.5

julia> inv(A) * x
2-element Array{Float64,1}:
 -4.0
  4.5
```

source

`Base.LinAlg.dot` – Function.

```
dot(n, X, incx, Y, incy)
```

Dot product of two vectors consisting of n elements of array `X` with stride `incx` and n elements of array `Y` with stride `incy`.

Example:

```
julia> dot(10, ones(10), 1, ones(20), 2)
10.0
```

source

`Base.LinAlg.vecdot` – Function.

```
vecdot(x, y)
```

For any iterable containers x and y (including arrays of any dimension) of numbers (or any element type for which `dot` is defined), compute the Euclidean dot product (the sum of `dot(x[i],y[i])`) as if they were vectors.

Examples

```
julia> vecdot(1:5, 2:6)
70

julia> x = fill(2., (5,5));

julia> y = fill(3., (5,5));

julia> vecdot(x, y)
150.0
```

source

`Base.LinAlg.cross` – Function.

```
cross(x, y)
×(x,y)
```

Compute the cross product of two 3-vectors.

Example

```
julia> a = [0;1;0]
3-element Array{Int64,1}:
 0
 1
 0

julia> b = [0;0;1]
3-element Array{Int64,1}:
 0
 0
 1

julia> cross(a,b)
3-element Array{Int64,1}:
 1
 0
 0
```

source

Base.LinAlg.factorize – Function.

```
factorize(A)
```

Compute a convenient factorization of A, based upon the type of the input matrix. factorize checks A to see if it is symmetric/triangular/etc. if A is passed as a generic matrix. factorize checks every element of A to verify/rule out each property. It will short-circuit as soon as it can rule out symmetry/triangular structure. The return value can be reused for efficient solving of multiple systems. For example: A=factorize(A); x=A\b; y=A\C.

Properties of A	type of factorization
Positive-definite	Cholesky (see cholfact)
Dense Symmetric/Hermitian	Bunch-Kaufman (see bkfact)
Sparse Symmetric/Hermitian	LDLt (see ldltfact)
Triangular	Triangular
Diagonal	Diagonal
Bidiagonal	Bidiagonal
Tridiagonal	LU (see lufact)
Symmetric real tridiagonal	LDLt (see ldltfact)
General square	LU (see lufact)
General non-square	QR (see qrfact)

If factorize is called on a Hermitian positive-definite matrix, for instance, then factorize will return a Cholesky factorization.

Example

```
julia> A = Array(Bidiagonal(ones(5, 5), true))
5×5 Array{Float64,2}:
 1.0  1.0  0.0  0.0  0.0
 0.0  1.0  1.0  0.0  0.0
 0.0  0.0  1.0  1.0  0.0
 0.0  0.0  0.0  1.0  1.0
 0.0  0.0  0.0  0.0  1.0

julia> factorize(A) # factorize will check to see that A is already factorized
5×5 Bidiagonal{Float64}:
 1.0  1.0
     1.0  1.0
         1.0  1.0
             1.0  1.0
                 1.0
```

This returns a 5×5 Bidiagonal{Float64}, which can now be passed to other linear algebra functions (e.g. eigensolvers) which will use specialized methods for Bidiagonal types.

source

Base.LinAlg.Diagonal – Type.

```
Diagonal(A::AbstractMatrix)
```

Constructs a matrix from the diagonal of A.

Example

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Array{Int64,2}:
 1  2  3
 4  5  6
 7  8  9

julia> Diagonal(A)
3×3 Diagonal{Int64}:
 1
   5
     9
```

source

```
Diagonal(V::AbstractVector)
```

Constructs a matrix with `V` as its diagonal.

Example

```
julia> V = [1; 2]
2-element Array{Int64,1}:
 1
 2

julia> Diagonal(V)
2×2 Diagonal{Int64}:
 1
   2
```

source

`Base.LinAlg.Bidiagonal` – Type.

```
Bidiagonal(dv, ev, isupper::Bool)
```

Constructs an upper (`isupper=true`) or lower (`isupper=false`) bidiagonal matrix using the given diagonal (`dv`) and off-diagonal (`ev`) vectors. The result is of type `Bidiagonal` and provides efficient specialized linear solvers, but may be converted into a regular matrix with `convert(Array, _)` (or `Array(_)` for short). `ev`'s length must be one less than the length of `dv`.

Example

```
julia> dv = [1; 2; 3; 4]
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> ev = [7; 8; 9]
3-element Array{Int64,1}:
 7
 8
 9

julia> Bu = Bidiagonal(dv, ev, true) # ev is on the first superdiagonal
```

```
4×4 Bidiagonal{Int64}:
 1  7
   2  8
     3  9
       4

julia> Bl = Bidiagonal(dv, ev, false) # ev is on the first subdiagonal
4×4 Bidiagonal{Int64}:
 1
 7  2
   8  3
     9  4
```

source

```
Bidiagonal(dv, ev, uplo::Char)
```

Constructs an upper (`uplo='U'`) or lower (`uplo='L'`) bidiagonal matrix using the given diagonal (`dv`) and off-diagonal (`ev`) vectors. The result is of type `Bidiagonal` and provides efficient specialized linear solvers, but may be converted into a regular matrix with `convert(Array, _)` (or `Array(_)` for short). `ev`'s length must be one less than the length of `dv`.

Example

```
julia> dv = [1; 2; 3; 4]
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> ev = [7; 8; 9]
3-element Array{Int64,1}:
 7
 8
 9

julia> Bu = Bidiagonal(dv, ev, 'U') #e is on the first superdiagonal
4×4 Bidiagonal{Int64}:
 1  7
   2  8
     3  9
       4

julia> Bl = Bidiagonal(dv, ev, 'L') #e is on the first subdiagonal
4×4 Bidiagonal{Int64}:
 1
 7  2
   8  3
     9  4
```

source

```
Bidiagonal(A, isupper::Bool)
```

Construct a `Bidiagonal` matrix from the main diagonal of `A` and its first super- (if `isupper=true`) or sub-diagonal (if `isupper=false`).

Example

```
julia> A = [1 1 1 1; 2 2 2 2; 3 3 3 3; 4 4 4 4]
4×4 Array{Int64,2}:
 1  1  1  1
 2  2  2  2
 3  3  3  3
 4  4  4  4

julia> Bidiagonal(A, true) #contains the main diagonal and first superdiagonal of A
4×4 Bidiagonal{Int64}:
 1  1
   2  2
     3  3
       4

julia> Bidiagonal(A, false) #contains the main diagonal and first subdiagonal of A
4×4 Bidiagonal{Int64}:
 1
 2  2
   3  3
     4  4
```

source

Base.LinAlg.SymTridiagonal – Type.

```
SymTridiagonal(dv, ev)
```

Construct a symmetric tridiagonal matrix from the diagonal and first sub/super-diagonal, respectively. The result is of type `SymTridiagonal` and provides efficient specialized eigensolvers, but may be converted into a regular matrix with `convert(Array, _)` (or `Array(_)` for short).

Example

```
julia> dv = [1; 2; 3; 4]
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> ev = [7; 8; 9]
3-element Array{Int64,1}:
 7
 8
 9

julia> SymTridiagonal(dv, ev)
4×4 SymTridiagonal{Int64}:
 1  7
 7  2  8
   8  3  9
     9  4
```

source

`Base.LinAlg.Tridiagonal` – Type.

```
Tridiagonal(dl, d, du)
```

Construct a tridiagonal matrix from the first subdiagonal, diagonal, and first superdiagonal, respectively. The result is of type `Tridiagonal` and provides efficient specialized linear solvers, but may be converted into a regular matrix with `convert(Array, _)` (or `Array(_)` for short). The lengths of `dl` and `du` must be one less than the length of `d`.

Example

```
julia> dl = [1; 2; 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> du = [4; 5; 6]
3-element Array{Int64,1}:
 4
 5
 6

julia> d = [7; 8; 9; 0]
4-element Array{Int64,1}:
 7
 8
 9
 0

julia> Tridiagonal(dl, d, du)
4×4 Tridiagonal{Int64}:
 7  4
 1  8  5
    2  9  6
       3  0
```

source

```
Tridiagonal(A)
```

returns a `Tridiagonal` array based on (abstract) matrix `A`, using its first lower diagonal, main diagonal, and first upper diagonal.

Example

```
julia> A = [1 2 3 4; 1 2 3 4; 1 2 3 4; 1 2 3 4]
4×4 Array{Int64,2}:
 1  2  3  4
 1  2  3  4
 1  2  3  4
 1  2  3  4

julia> Tridiagonal(A)
4×4 Tridiagonal{Int64}:
 1  2
 1  2  3
```

```
  2  3  4
    3  4
```

source

Base.LinAlg.Symmetric – Type.

```
Symmetric(A, uplo=:U)
```

Construct a Symmetric view of the upper (if uplo = :U) or lower (if uplo = :L) triangle of the matrix A.

Example

```
julia> A = [1 0 2 0 3; 0 4 0 5 0; 6 0 7 0 8; 0 9 0 1 0; 2 0 3 0 4]
5×5 Array{Int64,2}:
 1  0  2  0  3
 0  4  0  5  0
 6  0  7  0  8
 0  9  0  1  0
 2  0  3  0  4

julia> Supper = Symmetric(A)
5×5 Symmetric{Int64,Array{Int64,2}}:
 1  0  2  0  3
 0  4  0  5  0
 2  0  7  0  8
 0  5  0  1  0
 3  0  8  0  4

julia> Slower = Symmetric(A, :L)
5×5 Symmetric{Int64,Array{Int64,2}}:
 1  0  6  0  2
 0  4  0  9  0
 6  0  7  0  3
 0  9  0  1  0
 2  0  3  0  4
```

Note that Supper will not be equal to Slower unless A is itself symmetric (e.g. if A == A.').

source

Base.LinAlg.Hermitian – Type.

```
Hermitian(A, uplo=:U)
```

Construct a Hermitian view of the upper (if uplo = :U) or lower (if uplo = :L) triangle of the matrix A.

Example

```
julia> A = [1 0 2+2im 0 3-3im; 0 4 0 5 0; 6-6im 0 7 0 8+8im; 0 9 0 1 0; 2+2im 0 3-3im 0 4];

julia> Hupper = Hermitian(A)
5×5 Hermitian{Complex{Int64},Array{Complex{Int64},2}}:
 1+0im  0+0im  2+2im  0+0im  3-3im
 0+0im  4+0im  0+0im  5+0im  0+0im
 2-2im  0+0im  7+0im  0+0im  8+8im
 0+0im  5+0im  0+0im  1+0im  0+0im
```

```
 3+3im  0+0im  8-8im  0+0im  4+0im

julia> Hlower = Hermitian(A, :L)
5×5 Hermitian{Complex{Int64},Array{Complex{Int64},2}}:
 1+0im  0+0im  6+6im  0+0im  2-2im
 0+0im  4+0im  0+0im  9+0im  0+0im
 6-6im  0+0im  7+0im  0+0im  3+3im
 0+0im  9+0im  0+0im  1+0im  0+0im
 2+2im  0+0im  3-3im  0+0im  4+0im
```

Note that `Hupper` will not be equal to `Hlower` unless `A` is itself Hermitian (e.g. if `A == A'`).

source

`Base.LinAlg.LowerTriangular` – Type.

```
LowerTriangular(A::AbstractMatrix)
```

Construct a `LowerTriangular` view of the the matrix `A`.

Example

```
julia> A = [1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]
3×3 Array{Float64,2}:
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0

julia> LowerTriangular(A)
3×3 LowerTriangular{Float64,Array{Float64,2}}:
 1.0
 4.0  5.0
 7.0  8.0  9.0
```

source

`Base.LinAlg.UpperTriangular` – Type.

```
UpperTriangular(A::AbstractMatrix)
```

Construct an `UpperTriangular` view of the the matrix `A`.

Example

```
julia> A = [1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]
3×3 Array{Float64,2}:
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0

julia> UpperTriangular(A)
3×3 UpperTriangular{Float64,Array{Float64,2}}:
 1.0  2.0  3.0
     5.0  6.0
          9.0
```

source

Base.LinAlg.lu – Function.

```
lu(A, pivot=Val{true}) -> L, U, p
```

Compute the LU factorization of A, such that A[p,:] = L*U. By default, pivoting is used. This can be overridden by passing Val{false} for the second argument.

See also lufact.

Example

```
julia> A = [4. 3.; 6. 3.]
2×2 Array{Float64,2}:
 4.0  3.0
 6.0  3.0

julia> L, U, p = lu(A)
([1.0 0.0; 0.666667 1.0], [6.0 3.0; 0.0 1.0], [2, 1])

julia> A[p, :] == L * U
true
```

source

Base.LinAlg.lufact – Function.

```
lufact(A [,pivot=Val{true}]) -> F::LU
```

Compute the LU factorization of A.

In most cases, if A is a subtype S of AbstractMatrix{T} with an element type T supporting +, -, * and /, the return type is LU{T,S{T}}. If pivoting is chosen (default) the element type should also support abs and <.

The individual components of the factorization F can be accessed by indexing:

Component	Description
F[:L]	L (lower triangular) part of LU
F[:U]	U (upper triangular) part of LU
F[:p]	(right) permutation Vector
F[:P]	(right) permutation Matrix

The relationship between F and A is

F[:L]*F[:U] == A[F[:p], :]

F further supports the following functions:

Supported function	LU	LU{T,Tridiagonal{T}}
/		
\		
cond		
inv		
det		
logdet		
logabsdet		
size		

Example

```
julia> A = [4 3; 6 3]
2×2 Array{Int64,2}:
 4  3
 6  3

julia> F = lufact(A)
Base.LinAlg.LU{Float64,Array{Float64,2}} with factors L and U:
[1.0 0.0; 1.5 1.0]
[4.0 3.0; 0.0 -1.5]

julia> F[:L] * F[:U] == A[F[:p], :]
true
```

source

```
lufact(A::SparseMatrixCSC) -> F::UmfpackLU
```

Compute the LU factorization of a sparse matrix `A`.

For sparse `A` with real or complex element type, the return type of `F` is `UmfpackLU{Tv, Ti}`, with `Tv` = `Float64` or `Complex128` respectively and `Ti` is an integer type (`Int32` or `Int64`).

The individual components of the factorization `F` can be accessed by indexing:

Component	Description
`F[:L]`	`L` (lower triangular) part of `LU`
`F[:U]`	`U` (upper triangular) part of `LU`
`F[:p]`	right permutation `Vector`
`F[:q]`	left permutation `Vector`
`F[:Rs]`	`Vector` of scaling factors
`F[:(:)]`	`(L,U,p,q,Rs)` components

The relation between `F` and `A` is

```
F[:L]*F[:U] == (F[:Rs] .* A)[F[:p], F[:q]]
```

`F` further supports the following functions:

- `\`
- `cond`
- `det`

Note

`lufact(A::SparseMatrixCSC)` uses the UMFPACK library that is part of SuiteSparse. As this library only supports sparse matrices with `Float64` or `Complex128` elements, `lufact` converts `A` into a copy that is of type `SparseMatrixCSC{Float64}` or `SparseMatrixCSC{Complex128}` as appropriate.

source

`Base.LinAlg.lufact!` – Function.

```
lufact!(A, pivot=Val{true}) -> LU
```

`lufact!` is the same as `lufact`, but saves space by overwriting the input `A`, instead of creating a copy. An `InexactError` exception is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

source

`Base.LinAlg.chol` – Function.

```
chol(A) -> U
```

Compute the Cholesky factorization of a positive definite matrix `A` and return the `UpperTriangular` matrix `U` such that `A = U'U`.

Example

```
julia> A = [1. 2.; 2. 50.]
2×2 Array{Float64,2}:
 1.0   2.0
 2.0  50.0

julia> U = chol(A)
2×2 UpperTriangular{Float64,Array{Float64,2}}:
 1.0  2.0
     6.78233

julia> U'U
2×2 Array{Float64,2}:
 1.0   2.0
 2.0  50.0
```

source

```
chol(x::Number) -> y
```

Compute the square root of a non-negative number `x`.

Example

```
julia> chol(16)
4.0
```

source

`Base.LinAlg.cholfact` – Function.

```
cholfact(A, [uplo::Symbol,] Val{false}) -> Cholesky
```

Compute the Cholesky factorization of a dense symmetric positive definite matrix `A` and return a `Cholesky` factorization. The matrix `A` can either be a `Symmetric` or `Hermitian` `StridedMatrix` or a *perfectly* symmetric or Hermitian `StridedMatrix`. In the latter case, the optional argument `uplo` may be `:L` for using the lower part or `:U` for the upper part of `A`. The default is to use `:U`. The triangular Cholesky factor can be obtained from the factorization `F` with: `F[:L]` and `F[:U]`. The following functions are available for `Cholesky` objects: `size`, `\`, `inv`, and `det`. A `PosDefException` exception is thrown in case the matrix is not positive definite.

Example

```
julia> A = [4. 12. -16.; 12. 37. -43.; -16. -43. 98.]
3×3 Array{Float64,2}:
   4.0   12.0  -16.0
  12.0   37.0  -43.0
 -16.0  -43.0   98.0

julia> C = cholfact(A)
Base.LinAlg.Cholesky{Float64,Array{Float64,2}} with factor:
[2.0 6.0 -8.0; 0.0 1.0 5.0; 0.0 0.0 3.0]

julia> C[:U]
3×3 UpperTriangular{Float64,Array{Float64,2}}:
 2.0  6.0  -8.0
     1.0   5.0
          3.0

julia> C[:L]
3×3 LowerTriangular{Float64,Array{Float64,2}}:
  2.0
  6.0  1.0
 -8.0  5.0  3.0

julia> C[:L] * C[:U] == A
true
```

source

```
cholfact(A, [uplo::Symbol,] Val{true}; tol = 0.0) -> CholeskyPivoted
```

Compute the pivoted Cholesky factorization of a dense symmetric positive semi-definite matrix `A` and return a `CholeskyPivoted` factorization. The matrix `A` can either be a `Symmetric` or `Hermitian` `StridedMatrix` or a *perfectly* symmetric or Hermitian `StridedMatrix`. In the latter case, the optional argument `uplo` may be `:L` for using the lower part or `:U` for the upper part of `A`. The default is to use `:U`. The triangular Cholesky factor can be obtained from the factorization `F` with: `F[:L]` and `F[:U]`. The following functions are available for `PivotedCholesky` objects: `size`, `\`, `inv`, `det`, and `rank`. The argument `tol` determines the tolerance for determining the rank. For negative values, the tolerance is the machine precision.

source

```
cholfact(A; shift = 0.0, perm = Int[]) -> CHOLMOD.Factor
```

Compute the Cholesky factorization of a sparse positive definite matrix `A`. `A` must be a `SparseMatrixCSC` or a `Symmetric`/`Hermitian` view of a `SparseMatrixCSC`. Note that even if `A` doesn't have the type tag, it must still be symmetric or Hermitian. A fill-reducing permutation is used. `F = cholfact(A)` is most frequently used to solve systems of equations with `F\b`, but also the methods `diag`, `det`, and `logdet` are defined for `F`. You can also extract individual factors from `F`, using `F[:L]`. However, since pivoting is on by default, the factorization is internally represented as `A == P'*L*L'*P` with a permutation matrix `P`; using just `L` without accounting for `P` will give incorrect answers. To include the effects of permutation, it's typically preferable to extract "combined" factors like `PtL = F[:PtL]` (the equivalent of `P'*L`) and `LtP = F[:UP]` (the equivalent of `L'*P`).

Setting the optional `shift` keyword argument computes the factorization of `A+shift*I` instead of `A`. If the `perm` argument is nonempty, it should be a permutation of `1:size(A,1)` giving the ordering to use (instead of CHOLMOD's default AMD ordering).

Note

This method uses the CHOLMOD library from SuiteSparse, which only supports doubles or complex doubles. Input matrices not of those element types will be converted to `SparseMatrixCSC{Float64}` or `SparseMatrixCSC{Complex128}` as appropriate.

Many other functions from CHOLMOD are wrapped but not exported from the `Base.SparseArrays.CHOLMOD` module.

source

`Base.LinAlg.cholfact!` – Function.

```
cholfact!(A, [uplo::Symbol,] Val{false}) -> Cholesky
```

The same as `cholfact`, but saves space by overwriting the input `A`, instead of creating a copy. An `InexactError` exception is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

Example

```
julia> A = [1 2; 2 50]
2×2 Array{Int64,2}:
 1   2
 2  50

julia> cholfact!(A)
ERROR: InexactError()
```

source

```
cholfact!(A, [uplo::Symbol,] Val{true}; tol = 0.0) -> CholeskyPivoted
```

The same as `cholfact`, but saves space by overwriting the input `A`, instead of creating a copy. An `InexactError` exception is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

source

```
cholfact!(F::Factor, A; shift = 0.0) -> CHOLMOD.Factor
```

Compute the Cholesky (LL') factorization of `A`, reusing the symbolic factorization `F`. `A` must be a `SparseMatrixCSC` or a `Symmetric`/ `Hermitian` view of a `SparseMatrixCSC`. Note that even if `A` doesn't have the type tag, it must still be symmetric or Hermitian.

See also `cholfact`.

Note

This method uses the CHOLMOD library from SuiteSparse, which only supports doubles or complex doubles. Input matrices not of those element types will be converted to `SparseMatrixCSC{Float64}` or `SparseMatrixCSC{Complex128}` as appropriate.

source

`Base.LinAlg.lowrankupdate` – Function.

```
lowrankupdate(C::Cholesky, v::StridedVector) -> CC::Cholesky
```

Update a Cholesky factorization C with the vector v. If `A = C[:U]'C[:U]` then `CC = cholfact(C[:U]'C[:U] + v*v')` but the computation of CC only uses `O(n^2)` operations.

source

Base.LinAlg.lowrankdowndate – Function.

```
lowrankdowndate(C::Cholesky, v::StridedVector) -> CC::Cholesky
```

Downdate a Cholesky factorization C with the vector v. If `A = C[:U]'C[:U]` then `CC = cholfact(C[:U]'C[:U] - v*v')` but the computation of CC only uses `O(n^2)` operations.

source

Base.LinAlg.lowrankupdate! – Function.

```
lowrankupdate!(C::Cholesky, v::StridedVector) -> CC::Cholesky
```

Update a Cholesky factorization C with the vector v. If `A = C[:U]'C[:U]` then `CC = cholfact(C[:U]'C[:U] + v*v')` but the computation of CC only uses `O(n^2)` operations. The input factorization C is updated in place such that on exit `C == CC`. The vector v is destroyed during the computation.

source

Base.LinAlg.lowrankdowndate! – Function.

```
lowrankdowndate!(C::Cholesky, v::StridedVector) -> CC::Cholesky
```

Downdate a Cholesky factorization C with the vector v. If `A = C[:U]'C[:U]` then `CC = cholfact(C[:U]'C[:U] - v*v')` but the computation of CC only uses `O(n^2)` operations. The input factorization C is updated in place such that on exit `C == CC`. The vector v is destroyed during the computation.

source

Base.LinAlg.ldltfact – Function.

```
ldltfact(S::SymTridiagonal) -> LDLt
```

Compute an `LDLt` factorization of a real symmetric tridiagonal matrix such that `A = L*Diagonal(d)*L'` where L is a unit lower triangular matrix and d is a vector. The main use of an `LDLt` factorization `F = ldltfact(A)` is to solve the linear system of equations `Ax = b` with `F\b`.

source

```
ldltfact(A; shift = 0.0, perm=Int[]) -> CHOLMOD.Factor
```

Compute the LDL' factorization of a sparse matrix A. A must be a SparseMatrixCSC or a Symmetric/Hermitian view of a SparseMatrixCSC. Note that even if A doesn't have the type tag, it must still be symmetric or Hermitian. A fill-reducing permutation is used. `F = ldltfact(A)` is most frequently used to solve systems of equations `A*x = b` with `F\b`. The returned factorization object F also supports the methods diag, det, logdet, and inv. You can extract individual factors from F using `F[:L]`. However, since pivoting is on by default, the factorization is internally represented as `A == P'*L*D*L'*P` with a permutation matrix P; using just L without accounting for P will give incorrect answers. To include the effects of permutation, it is typically preferable to extract "combined" factors like `PtL = F[:PtL]` (the equivalent of `P'*L`) and `LtP = F[:UP]` (the equivalent of `L'*P`). The complete list of supported factors is `:L, :PtL, :D, :UP, :U, :LD, :DU, :PtLD, :DUP`.

Setting the optional `shift` keyword argument computes the factorization of `A+shift*I` instead of `A`. If the `perm` argument is nonempty, it should be a permutation of `1:size(A,1)` giving the ordering to use (instead of CHOLMOD's default AMD ordering).

Note

This method uses the CHOLMOD library from SuiteSparse, which only supports doubles or complex doubles. Input matrices not of those element types will be converted to `SparseMatrixCSC{Float64}` or `SparseMatrixCSC{Complex128}` as appropriate.

Many other functions from CHOLMOD are wrapped but not exported from the `Base.SparseArrays.CHOLMOD` module.

source

`Base.LinAlg.ldltfact!` – Function.

```
ldltfact!(S::SymTridiagonal) -> LDLt
```

Same as `ldltfact`, but saves space by overwriting the input `A`, instead of creating a copy.

source

```
ldltfact!(F::Factor, A; shift = 0.0) -> CHOLMOD.Factor
```

Compute the LDL' factorization of `A`, reusing the symbolic factorization `F`. `A` must be a `SparseMatrixCSC` or a `Symmetric`/`Hermitian` view of a `SparseMatrixCSC`. Note that even if `A` doesn't have the type tag, it must still be symmetric or Hermitian.

See also `ldltfact`.

Note

This method uses the CHOLMOD library from SuiteSparse, which only supports doubles or complex doubles. Input matrices not of those element types will be converted to `SparseMatrixCSC{Float64}` or `SparseMatrixCSC{Complex128}` as appropriate.

source

`Base.LinAlg.qr` – Function.

```
qr(A, pivot=Val{false}; thin::Bool=true) -> Q, R, [p]
```

Compute the (pivoted) QR factorization of `A` such that either `A = Q*R` or `A[:,p] = Q*R`. Also see `qrfact`. The default is to compute a thin factorization. Note that `R` is not extended with zeros when the full `Q` is requested.

source

```
qr(v::AbstractVector) -> w, r
```

Computes the polar decomposition of a vector. Returns `w`, a unit vector in the direction of `v`, and `r`, the norm of `v`.

See also `normalize`, `normalize!`, and `LinAlg.qr!`.

Example

```
julia> v = [1; 2]
2-element Array{Int64,1}:
 1
 2

julia> w, r = qr(v)
([0.447214, 0.894427], 2.23606797749979)

julia> w*r == v
true
```

source

Base.LinAlg.qr! – Function.

```
LinAlg.qr!(v::AbstractVector) -> w, r
```

Computes the polar decomposition of a vector. Instead of returning a new vector as `qr(v::AbstractVector)`, this function mutates the input vector `v` in place. Returns `w`, a unit vector in the direction of `v` (this is a mutation of `v`), and `r`, the norm of `v`.

See also `normalize`, `normalize!`, and `qr`.

source

Base.LinAlg.qrfact – Function.

```
qrfact(A, pivot=Val{false}) -> F
```

Compute the QR factorization of the matrix `A`: an orthogonal (or unitary if `A` is complex-valued) matrix `Q`, and an upper triangular matrix `R` such that

$$A = QR$$

The returned object `F` stores the factorization in a packed format:

- if `pivot == Val{true}` then `F` is a `QRPivoted` object,
- otherwise if the element type of `A` is a BLAS type (`Float32`, `Float64`, `Complex64` or `Complex128`), then `F` is a `QRCompactWY` object,
- otherwise `F` is a `QR` object.

The individual components of the factorization `F` can be accessed by indexing with a symbol:

- `F[:Q]`: the orthogonal/unitary matrix `Q`
- `F[:R]`: the upper triangular matrix `R`
- `F[:p]`: the permutation vector of the pivot (`QRPivoted` only)
- `F[:P]`: the permutation matrix of the pivot (`QRPivoted` only)

The following functions are available for the `QR` objects: `inv`, `size`, and `\`. When `A` is rectangular, `\` will return a least squares solution and if the solution is not unique, the one with smallest norm is returned.

Multiplication with respect to either thin or full `Q` is allowed, i.e. both `F[:Q]*F[:R]` and `F[:Q]*A` are supported. A `Q` matrix can be converted into a regular matrix with `full` which has a named argument `thin`.

Example

```
julia> A = [3.0 -6.0; 4.0 -8.0; 0.0 1.0]
3×2 Array{Float64,2}:
 3.0  -6.0
 4.0  -8.0
 0.0   1.0

julia> F = qrfact(A)
Base.LinAlg.QRCompactWY{Float64,Array{Float64,2}} with factors Q and R:
```

```
[-0.6 0.0 0.8; -0.8 0.0 -0.6; 0.0 -1.0 0.0]
[-5.0 10.0; 0.0 -1.0]

julia> F[:Q] * F[:R] == A
true
```

Note

`qrfact` returns multiple types because LAPACK uses several representations that minimize the memory storage requirements of products of Householder elementary reflectors, so that the `Q` and `R` matrices can be stored compactly rather as two separate dense matrices.

source

```
qrfact(A) -> SPQR.Factorization
```

Compute the `QR` factorization of a sparse matrix `A`. A fill-reducing permutation is used. The main application of this type is to solve least squares problems with `\`. The function calls the C library SPQR and a few additional functions from the library are wrapped but not exported.

source

`Base.LinAlg.qrfact!` – Function.

```
qrfact!(A, pivot=Val{false})
```

`qrfact!` is the same as `qrfact` when `A` is a subtype of `StridedMatrix`, but saves space by overwriting the input `A`, instead of creating a copy. An `InexactError` exception is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

source

`Base.LinAlg.QR` – Type.

```
QR <: Factorization
```

A QR matrix factorization stored in a packed format, typically obtained from `qrfact`. If A is an `m`×`n` matrix, then

$$A = QR$$

where Q is an orthogonal/unitary matrix and R is upper triangular. The matrix Q is stored as a sequence of Householder reflectors v_i and coefficients τ_i where:

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T).$$

The object has two fields:

- `factors` is an `m`×`n` matrix.
  - The upper triangular part contains the elements of R, that is `R = triu(F.factors)` for a `QR` object `F`.
  - The subdiagonal part contains the reflectors v_i stored in a packed format where v_i is the ith column of the matrix `V = eye(m,n) + tril(F.factors,-1)`.

- `τ` is a vector of length `min(m,n)` containing the coefficients au_i.

source

`Base.LinAlg.QRCompactWY` – Type.

```
QRCompactWY <: Factorization
```

A QR matrix factorization stored in a compact blocked format, typically obtained from `qrfact`. If A is an `m×n` matrix, then

$$A = QR$$

where Q is an orthogonal/unitary matrix and R is upper triangular. It is similar to the `QR` format except that the orthogonal/unitary matrix Q is stored in *Compact WY* format [1], as a lower trapezoidal matrix V and an upper triangular matrix T where

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T) = I - VTV^T$$

such that v_i is the ith column of V, and au_i is the ith diagonal element of T.

The object has two fields:

- `factors`, as in the `QR` type, is an `m×n` matrix.
  - The upper triangular part contains the elements of R, that is `R = triu(F.factors)` for a `QR` object `F`.
  - The subdiagonal part contains the reflectors v_i stored in a packed format such that `V = eye(m,n) + tril(F.factors,-1)`.
- `T` is a square matrix with `min(m,n)` columns, whose upper triangular part gives the matrix T above (the subdiagonal elements are ignored).

Note

This format should not to be confused with the older *WY* representation [2].

source

`Base.LinAlg.QRPivoted` – Type.

```
QRPivoted <: Factorization
```

A QR matrix factorization with column pivoting in a packed format, typically obtained from `qrfact`. If A is an `m×n` matrix, then

[2] C Bischof and C Van Loan, "The WY representation for products of Householder matrices", SIAM J Sci Stat Comput 8 (1987), s2-s13. doi:10.1137/0908009

[1] R Schreiber and C Van Loan, "A storage-efficient WY representation for products of Householder transformations", SIAM J Sci Stat Comput 10 (1989), 53-57. doi:10.1137/0910005

$$AP = QR$$

where P is a permutation matrix, Q is an orthogonal/unitary matrix and R is upper triangular. The matrix Q is stored as a sequence of Householder reflectors:

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T).$$

The object has three fields:

- `factors` is an `m`×`n` matrix.
  - The upper triangular part contains the elements of R, that is `R = triu(F.factors)` for a `QR` object `F`.
  - The subdiagonal part contains the reflectors v_i stored in a packed format where v_i is the ith column of the matrix `V = eye(m,n) + tril(F.factors,-1)`.
- `τ` is a vector of length `min(m,n)` containing the coefficients au_i.
- `jpvt` is an integer vector of length `n` corresponding to the permutation P.

source

`Base.LinAlg.lqfact!` – Function.

```
lqfact!(A) -> LQ
```

Compute the LQ factorization of `A`, using the input matrix as a workspace. See also `lq`.

source

`Base.LinAlg.lqfact` – Function.

```
lqfact(A) -> LQ
```

Compute the LQ factorization of `A`. See also `lq`.

source

`Base.LinAlg.lq` – Function.

```
lq(A; [thin=true]) -> L, Q
```

Perform an LQ factorization of `A` such that `A = L*Q`. The default is to compute a thin factorization. The LQ factorization is the QR factorization of `A.'`. `L` is not extended with zeros if the full `Q` is requested.

source

`Base.LinAlg.bkfact` – Function.

```
bkfact(A, uplo::Symbol=:U, symmetric::Bool=issymmetric(A), rook::Bool=false) -> BunchKaufman
```

Compute the Bunch-Kaufman [3] factorization of a symmetric or Hermitian matrix `A` and return a `BunchKaufman` object. `uplo` indicates which triangle of matrix `A` to reference. If `symmetric` is `true`, `A` is assumed to be symmetric. If `symmetric` is `false`, `A` is assumed to be Hermitian. If `rook` is `true`, rook pivoting is used. If `rook` is false, rook pivoting is not used. The following functions are available for `BunchKaufman` objects: `size`, `\`, `inv`, `issymmetric`, `ishermitian`.

source

`Base.LinAlg.bkfact!` – Function.

```
bkfact!(A, uplo::Symbol=:U, symmetric::Bool=issymmetric(A), rook::Bool=false) -> BunchKaufman
```

`bkfact!` is the same as `bkfact`, but saves space by overwriting the input `A`, instead of creating a copy.

source

`Base.LinAlg.eig` – Function.

```
eig(A::Union{SymTridiagonal, Hermitian, Symmetric}, irange::UnitRange) -> D, V
eig(A::Union{SymTridiagonal, Hermitian, Symmetric}, vl::Real, vu::Real) -> D, V
eig(A, permute::Bool=true, scale::Bool=true) -> D, V
```

Computes eigenvalues (`D`) and eigenvectors (`V`) of `A`. See `eigfact` for details on the `irange`, `vl`, and `vu` arguments (for `SymTridiagonal`, `Hermitian`, and `Symmetric` matrices) and the `permute` and `scale` keyword arguments. The eigenvectors are returned columnwise.

Example

```
julia> eig([1.0 0.0 0.0; 0.0 3.0 0.0; 0.0 0.0 18.0])
([1.0, 3.0, 18.0], [1.0 0.0 0.0; 0.0 1.0 0.0; 0.0 0.0 1.0])
```

`eig` is a wrapper around `eigfact`, extracting all parts of the factorization to a tuple; where possible, using `eigfact` is recommended.

source

```
eig(A, B) -> D, V
```

Computes generalized eigenvalues (`D`) and vectors (`V`) of `A` with respect to `B`.

`eig` is a wrapper around `eigfact`, extracting all parts of the factorization to a tuple; where possible, using `eigfact` is recommended.

Example

```
julia> A = [1 0; 0 -1]
2×2 Array{Int64,2}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Array{Int64,2}:
 0  1
 1  0
```

[3] J R Bunch and L Kaufman, Some stable methods for calculating inertia and solving symmetric linear systems, Mathematics of Computation 31:137 (1977), 163-179. url.

```
julia> eig(A, B)
(Complex{Float64}[0.0+1.0im, 0.0-1.0im], Complex{Float64}[0.0-1.0im 0.0+1.0im; -1.0-0.0im
↪  -1.0+0.0im])
```

source

Base.LinAlg.eigvals – Function.

```
eigvals(A; permute::Bool=true, scale::Bool=true) -> values
```

Returns the eigenvalues of A.

For general non-symmetric matrices it is possible to specify how the matrix is balanced before the eigenvalue calculation. The option `permute=true` permutes the matrix to become closer to upper triangular, and `scale=true` scales the matrix by its diagonal elements to make rows and columns more equal in norm. The default is `true` for both options.

source

```
eigvals(A, B) -> values
```

Computes the generalized eigenvalues of A and B.

Example

```
julia> A = [1 0; 0 -1]
2×2 Array{Int64,2}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Array{Int64,2}:
 0  1
 1  0

julia> eigvals(A,B)
2-element Array{Complex{Float64},1}:
 0.0+1.0im
 0.0-1.0im
```

source

```
eigvals(A::Union{SymTridiagonal, Hermitian, Symmetric}, irange::UnitRange) -> values
```

Returns the eigenvalues of A. It is possible to calculate only a subset of the eigenvalues by specifying a `UnitRange` `irange` covering indices of the sorted eigenvalues, e.g. the 2nd to 8th eigenvalues.

```
julia> A = SymTridiagonal([1.; 2.; 1.], [2.; 3.])
3×3 SymTridiagonal{Float64}:
 1.0  2.0
 2.0  2.0  3.0
     3.0  1.0

julia> eigvals(A, 2:2)
1-element Array{Float64,1}:
 1.0

julia> eigvals(A)
```

```
3-element Array{Float64,1}:
 -2.14005
  1.0
  5.14005
```

source

```
eigvals(A::Union{SymTridiagonal, Hermitian, Symmetric}, vl::Real, vu::Real) -> values
```

Returns the eigenvalues of `A`. It is possible to calculate only a subset of the eigenvalues by specifying a pair `vl` and `vu` for the lower and upper boundaries of the eigenvalues.

```
julia> A = SymTridiagonal([1.; 2.; 1.], [2.; 3.])
3×3 SymTridiagonal{Float64}:
 1.0  2.0
 2.0  2.0  3.0
      3.0  1.0

julia> eigvals(A, -1, 2)
1-element Array{Float64,1}:
 1.0

julia> eigvals(A)
3-element Array{Float64,1}:
 -2.14005
  1.0
  5.14005
```

source

`Base.LinAlg.eigvals!` – Function.

```
eigvals!(A; permute::Bool=true, scale::Bool=true) -> values
```

Same as `eigvals`, but saves space by overwriting the input `A`, instead of creating a copy. The option `permute=true` permutes the matrix to become closer to upper triangular, and `scale=true` scales the matrix by its diagonal elements to make rows and columns more equal in norm.

source

```
eigvals!(A, B) -> values
```

Same as `eigvals`, but saves space by overwriting the input `A` (and `B`), instead of creating copies.

source

```
eigvals!(A::Union{SymTridiagonal, Hermitian, Symmetric}, irange::UnitRange) -> values
```

Same as `eigvals`, but saves space by overwriting the input `A`, instead of creating a copy. `irange` is a range of eigenvalue *indices* to search for - for instance, the 2nd to 8th eigenvalues.

source

```
eigvals!(A::Union{SymTridiagonal, Hermitian, Symmetric}, vl::Real, vu::Real) -> values
```

Same as `eigvals`, but saves space by overwriting the input `A`, instead of creating a copy. `vl` is the lower bound of the interval to search for eigenvalues, and `vu` is the upper bound.

source

Base.LinAlg.eigmax – Function.

```
eigmax(A; permute::Bool=true, scale::Bool=true)
```

Returns the largest eigenvalue of A. The option `permute=true` permutes the matrix to become closer to upper triangular, and `scale=true` scales the matrix by its diagonal elements to make rows and columns more equal in norm. Note that if the eigenvalues of A are complex, this method will fail, since complex numbers cannot be sorted.

Example

```
julia> A = [0 im; -im 0]
2×2 Array{Complex{Int64},2}:
 0+0im  0+1im
 0-1im  0+0im

julia> eigmax(A)
1.0

julia> A = [0 im; -1 0]
2×2 Array{Complex{Int64},2}:
  0+0im  0+1im
 -1+0im  0+0im

julia> eigmax(A)
ERROR: DomainError:
Stacktrace:
 [1] #eigmax#46(::Bool, ::Bool, ::Function, ::Array{Complex{Int64},2}) at
↪  ./linalg/eigen.jl:238
 [2] eigmax(::Array{Complex{Int64},2}) at ./linalg/eigen.jl:236
```

source

Base.LinAlg.eigmin – Function.

```
eigmin(A; permute::Bool=true, scale::Bool=true)
```

Returns the smallest eigenvalue of A. The option `permute=true` permutes the matrix to become closer to upper triangular, and `scale=true` scales the matrix by its diagonal elements to make rows and columns more equal in norm. Note that if the eigenvalues of A are complex, this method will fail, since complex numbers cannot be sorted.

Example

```
julia> A = [0 im; -im 0]
2×2 Array{Complex{Int64},2}:
 0+0im  0+1im
 0-1im  0+0im

julia> eigmin(A)
-1.0

julia> A = [0 im; -1 0]
2×2 Array{Complex{Int64},2}:
  0+0im  0+1im
 -1+0im  0+0im
```

```
julia> eigmin(A)
ERROR: DomainError:
Stacktrace:
 [1] #eigmin#47(::Bool, ::Bool, ::Function, ::Array{Complex{Int64},2}) at
↪   ./linalg/eigen.jl:280
 [2] eigmin(::Array{Complex{Int64},2}) at ./linalg/eigen.jl:278
```

source

Base.LinAlg.eigvecs – Function.

```
eigvecs(A::SymTridiagonal[, eigvals]) -> Matrix
```

Returns a matrix `M` whose columns are the eigenvectors of `A`. (The kth eigenvector can be obtained from the slice `M[:, k]`.)

If the optional vector of eigenvalues `eigvals` is specified, `eigvecs` returns the specific corresponding eigenvectors.

Example

```
julia> A = SymTridiagonal([1.; 2.; 1.], [2.; 3.])
3×3 SymTridiagonal{Float64}:
 1.0  2.0
 2.0  2.0  3.0
     3.0  1.0

julia> eigvals(A)
3-element Array{Float64,1}:
 -2.14005
  1.0
  5.14005

julia> eigvecs(A)
3×3 Array{Float64,2}:
  0.418304  -0.83205      0.364299
 -0.656749  -7.39009e-16  0.754109
  0.627457   0.5547       0.546448

julia> eigvecs(A, [1.])
3×1 Array{Float64,2}:
  0.83205
  4.26351e-17
 -0.5547
```

source

```
eigvecs(A; permute::Bool=true, scale::Bool=true) -> Matrix
```

Returns a matrix `M` whose columns are the eigenvectors of `A`. (The kth eigenvector can be obtained from the slice `M[:, k]`.) The `permute` and `scale` keywords are the same as for `eigfact`.

Example

```
julia> eigvecs([1.0 0.0 0.0; 0.0 3.0 0.0; 0.0 0.0 18.0])
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

source

```
eigvecs(A, B) -> Matrix
```

Returns a matrix `M` whose columns are the generalized eigenvectors of `A` and `B`. (The `k`th eigenvector can be obtained from the slice `M[:, k]`.)

Example

```
julia> A = [1 0; 0 -1]
2×2 Array{Int64,2}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Array{Int64,2}:
 0  1
 1  0

julia> eigvecs(A, B)
2×2 Array{Complex{Float64},2}:
  0.0-1.0im   0.0+1.0im
 -1.0-0.0im  -1.0+0.0im
```

source

`Base.LinAlg.eigfact` – Function.

```
eigfact(A; permute::Bool=true, scale::Bool=true) -> Eigen
```

Computes the eigenvalue decomposition of `A`, returning an `Eigen` factorization object `F` which contains the eigenvalues in `F[:values]` and the eigenvectors in the columns of the matrix `F[:vectors]`. (The `k`th eigenvector can be obtained from the slice `F[:vectors][:, k]`.)

The following functions are available for `Eigen` objects: `inv`, `det`, and `isposdef`.

For general nonsymmetric matrices it is possible to specify how the matrix is balanced before the eigenvector calculation. The option `permute=true` permutes the matrix to become closer to upper triangular, and `scale=true` scales the matrix by its diagonal elements to make rows and columns more equal in norm. The default is `true` for both options.

Example

```
julia> F = eigfact([1.0 0.0 0.0; 0.0 3.0 0.0; 0.0 0.0 18.0])
Base.LinAlg.Eigen{Float64,Float64,Array{Float64,2},Array{Float64,1}}([1.0, 3.0, 18.0], [1.0
↪  0.0 0.0; 0.0 1.0 0.0; 0.0 0.0 1.0])

julia> F[:values]
3-element Array{Float64,1}:
  1.0
  3.0
 18.0

julia> F[:vectors]
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

source

```
eigfact(A, B) -> GeneralizedEigen
```

Computes the generalized eigenvalue decomposition of A and B, returning a GeneralizedEigen factorization object F which contains the generalized eigenvalues in F[:values] and the generalized eigenvectors in the columns of the matrix F[:vectors]. (The kth generalized eigenvector can be obtained from the slice F[:vectors][:, k].)

source

```
eigfact(A::Union{SymTridiagonal, Hermitian, Symmetric}, irange::UnitRange) -> Eigen
```

Computes the eigenvalue decomposition of A, returning an Eigen factorization object F which contains the eigenvalues in F[:values] and the eigenvectors in the columns of the matrix F[:vectors]. (The kth eigenvector can be obtained from the slice F[:vectors][:, k].)

The following functions are available for Eigen objects: inv, det, and isposdef.

The UnitRange irange specifies indices of the sorted eigenvalues to search for.

> **Note**
> If irange is not 1:n, where n is the dimension of A, then the returned factorization will be a *truncated* factorization.

source

```
eigfact(A::Union{SymTridiagonal, Hermitian, Symmetric}, vl::Real, vu::Real) -> Eigen
```

Computes the eigenvalue decomposition of A, returning an Eigen factorization object F which contains the eigenvalues in F[:values] and the eigenvectors in the columns of the matrix F[:vectors]. (The kth eigenvector can be obtained from the slice F[:vectors][:, k].)

The following functions are available for Eigen objects: inv, det, and isposdef.

vl is the lower bound of the window of eigenvalues to search for, and vu is the upper bound.

> **Note**
> If [vl, vu] does not contain all eigenvalues of A, then the returned factorization will be a *truncated* factorization.

source

Base.LinAlg.eigfact! – Function.

```
eigfact!(A, [B])
```

Same as eigfact, but saves space by overwriting the input A (and B), instead of creating a copy.

source

Base.LinAlg.hessfact – Function.

```
hessfact(A) -> Hessenberg
```

Compute the Hessenberg decomposition of A and return a Hessenberg object. If F is the factorization object, the unitary matrix can be accessed with F[:Q] and the Hessenberg matrix with F[:H]. When Q is extracted, the resulting type is the HessenbergQ object, and may be converted to a regular matrix with convert(Array, _) (or Array(_) for short).

Example

```
julia> A = [4. 9. 7.; 4. 4. 1.; 4. 3. 2.]
3×3 Array{Float64,2}:
 4.0  9.0  7.0
 4.0  4.0  1.0
 4.0  3.0  2.0

julia> F = hessfact(A);

julia> F[:Q] * F[:H] * F[:Q]'
3×3 Array{Float64,2}:
 4.0  9.0  7.0
 4.0  4.0  1.0
 4.0  3.0  2.0
```

source

Base.LinAlg.hessfact! – Function.

```
hessfact!(A) -> Hessenberg
```

`hessfact!` is the same as `hessfact`, but saves space by overwriting the input `A`, instead of creating a copy.

source

Base.LinAlg.schurfact – Function.

```
schurfact(A::StridedMatrix) -> F::Schur
```

Computes the Schur factorization of the matrix `A`. The (quasi) triangular Schur factor can be obtained from the Schur object `F` with either `F[:Schur]` or `F[:T]` and the orthogonal/unitary Schur vectors can be obtained with `F[:vectors]` or `F[:Z]` such that `A = F[:vectors]*F[:Schur]*F[:vectors]'`. The eigenvalues of `A` can be obtained with `F[:values]`.

Example

```
julia> A = [-2. 1. 3.; 2. 1. -1.; -7. 2. 7.]
3×3 Array{Float64,2}:
 -2.0  1.0   3.0
  2.0  1.0  -1.0
 -7.0  2.0   7.0

julia> F = schurfact(A)
Base.LinAlg.Schur{Float64,Array{Float64,2}} with factors T and Z:
[2.0 0.801792 6.63509; -8.55988e-11 2.0 8.08286; 0.0 0.0 1.99999]
[0.577351 0.154299 -0.801784; 0.577346 -0.77152 0.267262; 0.577354 0.617211 0.534522]
and values:
Complex{Float64}[2.0+8.28447e-6im, 2.0-8.28447e-6im, 1.99999+0.0im]

julia> F[:vectors] * F[:Schur] * F[:vectors]'
3×3 Array{Float64,2}:
 -2.0  1.0   3.0
  2.0  1.0  -1.0
 -7.0  2.0   7.0
```

source

```
schurfact(A::StridedMatrix, B::StridedMatrix) -> F::GeneralizedSchur
```

Computes the Generalized Schur (or QZ) factorization of the matrices A and B. The (quasi) triangular Schur factors can be obtained from the Schur object F with F[:S] and F[:T], the left unitary/orthogonal Schur vectors can be obtained with F[:left] or F[:Q] and the right unitary/orthogonal Schur vectors can be obtained with F[:right] or F[:Z] such that A=F[:left]*F[:S]*F[:right]' and B=F[:left]*F[:T]*F[:right]'. The generalized eigenvalues of A and B can be obtained with F[:alpha]./F[:beta].

source

Base.LinAlg.schurfact! – Function.

```
schurfact!(A::StridedMatrix) -> F::Schur
```

Same as schurfact but uses the input argument as workspace.

source

```
schurfact!(A::StridedMatrix, B::StridedMatrix) -> F::GeneralizedSchur
```

Same as schurfact but uses the input matrices A and B as workspace.

source

Base.LinAlg.schur – Function.

```
schur(A::StridedMatrix) -> T::Matrix, Z::Matrix, λ::Vector
```

Computes the Schur factorization of the matrix A. The methods return the (quasi) triangular Schur factor T and the orthogonal/unitary Schur vectors Z such that A = Z*T*Z'. The eigenvalues of A are returned in the vector λ.

See schurfact.

Example

```
julia> A = [-2. 1. 3.; 2. 1. -1.; -7. 2. 7.]
3×3 Array{Float64,2}:
 -2.0  1.0   3.0
  2.0  1.0  -1.0
 -7.0  2.0   7.0

julia> T, Z, lambda = schur(A)
([2.0 0.801792 6.63509; -8.55988e-11 2.0 8.08286; 0.0 0.0 1.99999], [0.577351 0.154299
↪  -0.801784; 0.577346 -0.77152 0.267262; 0.577354 0.617211 0.534522],
↪  Complex{Float64}[2.0+8.28447e-6im, 2.0-8.28447e-6im, 1.99999+0.0im])

julia> Z * T * Z'
3×3 Array{Float64,2}:
 -2.0  1.0   3.0
  2.0  1.0  -1.0
 -7.0  2.0   7.0
```

source

```
schur(A::StridedMatrix, B::StridedMatrix) -> S::StridedMatrix, T::StridedMatrix, Q::
    StridedMatrix, Z::StridedMatrix, α::Vector, β::Vector
```

See schurfact.

source

Base.LinAlg.ordschur – Function.

```
ordschur(F::Schur, select::Union{Vector{Bool},BitVector}) -> F::Schur
```

Reorders the Schur factorization F of a matrix `A = Z*T*Z'` according to the logical array `select` returning the reordered factorization F object. The selected eigenvalues appear in the leading diagonal of `F[:Schur]` and the corresponding leading columns of `F[:vectors]` form an orthogonal/unitary basis of the corresponding right invariant subspace. In the real case, a complex conjugate pair of eigenvalues must be either both included or both excluded via `select`.

source

```
ordschur(T::StridedMatrix, Z::StridedMatrix, select::Union{Vector{Bool},BitVector}) -> T::
    StridedMatrix, Z::StridedMatrix, λ::Vector
```

Reorders the Schur factorization of a real matrix `A = Z*T*Z'` according to the logical array `select` returning the reordered matrices T and Z as well as the vector of eigenvalues λ. The selected eigenvalues appear in the leading diagonal of T and the corresponding leading columns of Z form an orthogonal/unitary basis of the corresponding right invariant subspace. In the real case, a complex conjugate pair of eigenvalues must be either both included or both excluded via `select`.

source

```
ordschur(F::GeneralizedSchur, select::Union{Vector{Bool},BitVector}) -> F::GeneralizedSchur
```

Reorders the Generalized Schur factorization F of a matrix pair `(A, B) = (Q*S*Z', Q*T*Z')` according to the logical array `select` and returns a GeneralizedSchur object F. The selected eigenvalues appear in the leading diagonal of both `F[:S]` and `F[:T]`, and the left and right orthogonal/unitary Schur vectors are also reordered such that `(A, B) = F[:Q]*(F[:S], F[:T])*F[:Z]'` still holds and the generalized eigenvalues of A and B can still be obtained with `F[:alpha]./F[:beta]`.

source

```
ordschur(S::StridedMatrix, T::StridedMatrix, Q::StridedMatrix, Z::StridedMatrix, select) -> S
    ::StridedMatrix, T::StridedMatrix, Q::StridedMatrix, Z::StridedMatrix, α::Vector, β::
    Vector
```

Reorders the Generalized Schur factorization of a matrix pair `(A, B) = (Q*S*Z', Q*T*Z')` according to the logical array `select` and returns the matrices S, T, Q, Z and vectors α and β. The selected eigenvalues appear in the leading diagonal of both S and T, and the left and right unitary/orthogonal Schur vectors are also reordered such that `(A, B) = Q*(S, T)*Z'` still holds and the generalized eigenvalues of A and B can still be obtained with `α./β`.

source

`Base.LinAlg.ordschur!` – Function.

```
ordschur!(F::Schur, select::Union{Vector{Bool},BitVector}) -> F::Schur
```

Same as `ordschur` but overwrites the factorization F.

source

```
ordschur!(T::StridedMatrix, Z::StridedMatrix, select::Union{Vector{Bool},BitVector}) -> T::
    StridedMatrix, Z::StridedMatrix, λ::Vector
```

Same as `ordschur` but overwrites the input arguments.

source

```
ordschur!(F::GeneralizedSchur, select::Union{Vector{Bool},BitVector}) -> F::GeneralizedSchur
```

Same as `ordschur` but overwrites the factorization `F`.

source

```
ordschur!(S::StridedMatrix, T::StridedMatrix, Q::StridedMatrix, Z::StridedMatrix, select) ->
    S::StridedMatrix, T::StridedMatrix, Q::StridedMatrix, Z::StridedMatrix, α::Vector, β::
    Vector
```

Same as `ordschur` but overwrites the factorization the input arguments.

source

`Base.LinAlg.svdfact` – Function.

```
svdfact(A; thin::Bool=true) -> SVD
```

Compute the singular value decomposition (SVD) of `A` and return an `SVD` object.

`U`, `S`, `V` and `Vt` can be obtained from the factorization `F` with `F[:U]`, `F[:S]`, `F[:V]` and `F[:Vt]`, such that `A = U*diagm(S)*Vt`. The algorithm produces `Vt` and hence `Vt` is more efficient to extract than `V`. The singular values in `S` are sorted in descending order.

If `thin=true` (default), a thin SVD is returned. For a $M \times N$ matrix `A`, `U` is $M \times M$ for a full SVD (`thin=false`) and $M \times \min(M, N)$ for a thin SVD.

Example

```
julia> A = [1. 0. 0. 0. 2.; 0. 0. 3. 0. 0.; 0. 0. 0. 0. 0.; 0. 2. 0. 0. 0.]
4×5 Array{Float64,2}:
 1.0  0.0  0.0  0.0  2.0
 0.0  0.0  3.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  2.0  0.0  0.0  0.0

julia> F = svdfact(A)
Base.LinAlg.SVD{Float64,Float64,Array{Float64,2}}([0.0 1.0 0.0 0.0; 1.0 0.0 0.0 0.0; 0.0 0.0
↪  0.0 -1.0; 0.0 0.0 1.0 0.0], [3.0, 2.23607, 2.0, 0.0], [-0.0 0.0 … -0.0 0.0; 0.447214 0.0
↪  … 0.0 0.894427; -0.0 1.0 … -0.0 0.0; 0.0 0.0 … 1.0 0.0])

julia> F[:U] * diagm(F[:S]) * F[:Vt]
4×5 Array{Float64,2}:
 1.0  0.0  0.0  0.0  2.0
 0.0  0.0  3.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  2.0  0.0  0.0  0.0
```

source

```
svdfact(A, B) -> GeneralizedSVD
```

Compute the generalized SVD of `A` and `B`, returning a `GeneralizedSVD` factorization object `F`, such that `A = F[:U]*F[:D1]*F[:R0]*F[:Q]'` and `B = F[:V]*F[:D2]*F[:R0]*F[:Q]'`.

For an M-by-N matrix `A` and P-by-N matrix `B`,

- `F[:U]` is a M-by-M orthogonal matrix,
- `F[:V]` is a P-by-P orthogonal matrix,
- `F[:Q]` is a N-by-N orthogonal matrix,

- `F[:R0]` is a (K+L)-by-N matrix whose rightmost (K+L)-by-(K+L) block is nonsingular upper block triangular,
- `F[:D1]` is a M-by-(K+L) diagonal matrix with 1s in the first K entries,
- `F[:D2]` is a P-by-(K+L) matrix whose top right L-by-L block is diagonal,

K+L is the effective numerical rank of the matrix `[A; B]`.

The entries of `F[:D1]` and `F[:D2]` are related, as explained in the LAPACK documentation for the generalized SVD and the xGGSVD3 routine which is called underneath (in LAPACK 3.6.0 and newer).

source

`Base.LinAlg.svdfact!` – Function.

```
svdfact!(A, thin::Bool=true) -> SVD
```

`svdfact!` is the same as `svdfact`, but saves space by overwriting the input `A`, instead of creating a copy.

source

```
svdfact!(A, B) -> GeneralizedSVD
```

`svdfact!` is the same as `svdfact`, but modifies the arguments `A` and `B` in-place, instead of making copies.

source

`Base.LinAlg.svd` – Function.

```
svd(A; thin::Bool=true) -> U, S, V
```

Computes the SVD of `A`, returning `U`, vector `S`, and `V` such that `A == U*diagm(S)*V'`. The singular values in `S` are sorted in descending order.

If `thin=true` (default), a thin SVD is returned. For a $M \times N$ matrix `A`, `U` is $M \times M$ for a full SVD (`thin=false`) and $M \times \min(M, N)$ for a thin SVD.

`svd` is a wrapper around `svdfact`, extracting all parts of the SVD factorization to a tuple. Direct use of `svdfact` is therefore more efficient.

Example

```
julia> A = [1. 0. 0. 0. 2.; 0. 0. 3. 0. 0.; 0. 0. 0. 0. 0.; 0. 2. 0. 0. 0.]
4×5 Array{Float64,2}:
 1.0  0.0  0.0  0.0  2.0
 0.0  0.0  3.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  2.0  0.0  0.0  0.0

julia> U, S, V = svd(A)
([0.0 1.0 0.0 0.0; 1.0 0.0 0.0 0.0; 0.0 0.0 0.0 -1.0; 0.0 0.0 1.0 0.0], [3.0, 2.23607, 2.0,
↪  0.0], [-0.0 0.447214 -0.0 0.0; 0.0 0.0 1.0 0.0; … ; -0.0 0.0 -0.0 1.0; 0.0 0.894427 0.0
↪  0.0])

julia> U*diagm(S)*V'
4×5 Array{Float64,2}:
 1.0  0.0  0.0  0.0  2.0
 0.0  0.0  3.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  2.0  0.0  0.0  0.0
```

source

```
svd(A, B) -> U, V, Q, D1, D2, R0
```

Wrapper around svdfact extracting all parts of the factorization to a tuple. Direct use of svdfact is therefore generally more efficient. The function returns the generalized SVD of A and B, returning U, V, Q, D1, D2, and R0 such that A = U*D1*R0*Q' and B = V*D2*R0*Q'.

source

Base.LinAlg.svdvals – Function.

```
svdvals(A)
```

Returns the singular values of A in descending order.

Example

```
julia> A = [1. 0. 0. 0. 2.; 0. 0. 3. 0. 0.; 0. 0. 0. 0. 0.; 0. 2. 0. 0. 0.]
4×5 Array{Float64,2}:
 1.0  0.0  0.0  0.0  2.0
 0.0  0.0  3.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  2.0  0.0  0.0  0.0

julia> svdvals(A)
4-element Array{Float64,1}:
 3.0
 2.23607
 2.0
 0.0
```

source

```
svdvals(A, B)
```

Return the generalized singular values from the generalized singular value decomposition of A and B. See also svdfact.

source

Base.LinAlg.Givens – Type.

```
LinAlg.Givens(i1,i2,c,s) -> G
```

A Givens rotation linear operator. The fields c and s represent the cosine and sine of the rotation angle, respectively. The Givens type supports left multiplication G*A and conjugated transpose right multiplication A*G'. The type doesn't have a size and can therefore be multiplied with matrices of arbitrary size as long as i2<=size(A,2) for G*A or i2<=size(A,1) for A*G'.

See also: givens

source

Base.LinAlg.givens – Function.

```
givens{T}(f::T, g::T, i1::Integer, i2::Integer) -> (G::Givens, r::T)
```

Computes the Givens rotation G and scalar r such that for any vector x where

```
x[i1] = f
x[i2] = g
```

the result of the multiplication

```
y = G*x
```

has the property that

```
y[i1] = r
y[i2] = 0
```

See also: LinAlg.Givens

source

```
givens(A::AbstractArray, i1::Integer, i2::Integer, j::Integer) -> (G::Givens, r)
```

Computes the Givens rotation G and scalar r such that the result of the multiplication

```
B = G*A
```

has the property that

```
B[i1,j] = r
B[i2,j] = 0
```

See also: LinAlg.Givens

source

```
givens(x::AbstractVector, i1::Integer, i2::Integer) -> (G::Givens, r)
```

Computes the Givens rotation G and scalar r such that the result of the multiplication

```
B = G*x
```

has the property that

```
B[i1] = r
B[i2] = 0
```

See also: LinAlg.Givens

source

Base.LinAlg.triu – Function.

```
triu(M)
```

Upper triangle of a matrix.

Example

```
julia> a = ones(4,4)
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
```

```
julia> triu(a)
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 0.0  1.0  1.0  1.0
 0.0  0.0  1.0  1.0
 0.0  0.0  0.0  1.0
```

source

```
triu(M, k::Integer)
```

Returns the upper triangle of `M` starting from the `k`th superdiagonal.

Example

```
julia> a = ones(4,4)
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0

julia> triu(a,3)
4×4 Array{Float64,2}:
 0.0  0.0  0.0  1.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0

julia> triu(a,-3)
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
```

source

`Base.LinAlg.triu!` – Function.

```
triu!(M)
```

Upper triangle of a matrix, overwriting `M` in the process. See also `triu`.

source

```
triu!(M, k::Integer)
```

Returns the upper triangle of `M` starting from the `k`th superdiagonal, overwriting `M` in the process.

Example

```
julia> M = [1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5]
5×5 Array{Int64,2}:
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
```

```
 1  2  3  4  5

julia> triu!(M, 1)
5×5 Array{Int64,2}:
 0  2  3  4  5
 0  0  3  4  5
 0  0  0  4  5
 0  0  0  0  5
 0  0  0  0  0
```

source

Base.LinAlg.tril – Function.

```
tril(M)
```

Lower triangle of a matrix.

Example

```
julia> a = ones(4,4)
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0

julia> tril(a)
4×4 Array{Float64,2}:
 1.0  0.0  0.0  0.0
 1.0  1.0  0.0  0.0
 1.0  1.0  1.0  0.0
 1.0  1.0  1.0  1.0
```

source

```
tril(M, k::Integer)
```

Returns the lower triangle of `M` starting from the kth superdiagonal.

Example

```
julia> a = ones(4,4)
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0

julia> tril(a,3)
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0

julia> tril(a,-3)
```

```
4×4 Array{Float64,2}:
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 1.0  0.0  0.0  0.0
```

source

Base.LinAlg.tril! – Function.

```
tril!(M)
```

Lower triangle of a matrix, overwriting `M` in the process. See also `tril`.

source

```
tril!(M, k::Integer)
```

Returns the lower triangle of `M` starting from the kth superdiagonal, overwriting `M` in the process.

Example

```
julia> M = [1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5]
5×5 Array{Int64,2}:
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5

julia> tril!(M, 2)
5×5 Array{Int64,2}:
 1  2  3  0  0
 1  2  3  4  0
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
```

source

Base.LinAlg.diagind – Function.

```
diagind(M, k::Integer=0)
```

A Range giving the indices of the kth diagonal of the matrix `M`.

Example

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Array{Int64,2}:
 1  2  3
 4  5  6
 7  8  9

julia> diagind(A,-1)
2:4:6
```

source

Base.LinAlg.diag – Function.

```
diag(M, k::Integer=0)
```

The kth diagonal of a matrix, as a vector. Use diagm to construct a diagonal matrix.

Example

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Array{Int64,2}:
 1  2  3
 4  5  6
 7  8  9

julia> diag(A,1)
2-element Array{Int64,1}:
 2
 6
```

source

Base.LinAlg.diagm – Function.

```
diagm(v, k::Integer=0)
```

Construct a matrix by placing v on the kth diagonal.

Example

```
julia> diagm([1,2,3],1)
4×4 Array{Int64,2}:
 0  1  0  0
 0  0  2  0
 0  0  0  3
 0  0  0  0
```

source

Base.LinAlg.scale! – Function.

```
scale!(A, b)
scale!(b, A)
```

Scale an array A by a scalar b overwriting A in-place.

If A is a matrix and b is a vector, then `scale!(A,b)` scales each column `i` of A by `b[i]` (similar to `A*Diagonal(b)`), while `scale!(b,A)` scales each row `i` of A by `b[i]` (similar to `Diagonal(b)*A`), again operating in-place on A. An `InexactError` exception is thrown if the scaling produces a number not representable by the element type of A, e.g. for integer types.

Example

```
julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> b = [1; 2]
```

```
2-element Array{Int64,1}:
 1
 2

julia> scale!(a,b)
2×2 Array{Int64,2}:
 1  4
 3  8

julia> a = [1 2; 3 4];

julia> b = [1; 2];

julia> scale!(b,a)
2×2 Array{Int64,2}:
 1  2
 6  8
```

source

Base.LinAlg.rank – Function.

```
rank(M[, tol::Real])
```

Compute the rank of a matrix by counting how many singular values of `M` have magnitude greater than `tol`. By default, the value of `tol` is the largest dimension of `M` multiplied by the `eps` of the `eltype` of `M`.

Example

```
julia> rank(eye(3))
3

julia> rank(diagm([1, 0, 2]))
2

julia> rank(diagm([1, 0.001, 2]), 0.1)
2

julia> rank(diagm([1, 0.001, 2]), 0.00001)
3
```

source

Base.LinAlg.norm – Function.

```
norm(A::AbstractArray, p::Real=2)
```

Compute the p-norm of a vector or the operator norm of a matrix `A`, defaulting to the 2-norm.

```
norm(A::AbstractVector, p::Real=2)
```

For vectors, this is equivalent to `vecnorm` and equal to:

$$\|A\|_p = \left(\sum_{i=1}^{n} |a_i|^p \right)^{1/p}$$

with a_i the entries of A and n its length.

p can assume any numeric value (even though not all values produce a mathematically valid vector norm). In particular, `norm(A, Inf)` returns the largest value in `abs(A)`, whereas `norm(A, -Inf)` returns the smallest.

Example

```
julia> v = [3, -2, 6]
3-element Array{Int64,1}:
  3
 -2
  6

julia> norm(v)
7.0

julia> norm(v, Inf)
6.0
```

source

```
norm(A::AbstractMatrix, p::Real=2)
```

For matrices, the matrix norm induced by the vector p-norm is used, where valid values of p are 1, 2, or `Inf`. (Note that for sparse matrices, p=2 is currently not implemented.) Use `vecnorm` to compute the Frobenius norm.

When p=1, the matrix norm is the maximum absolute column sum of `A`:

$$\|A\|_1 = \max_{1jn} \sum_{i=1}^{m} |a_{ij}|$$

with a_{ij} the entries of A, and m and n its dimensions.

When p=2, the matrix norm is the spectral norm, equal to the largest singular value of `A`.

When p=`Inf`, the matrix norm is the maximum absolute row sum of `A`:

$$\|A\|_\infty = \max_{1im} \sum_{j=1}^{n} |a_{ij}|$$

Example

```
julia> A = [1 -2 -3; 2 3 -1]
2×3 Array{Int64,2}:
 1  -2  -3
 2   3  -1

julia> norm(A, Inf)
6.0
```

source

```
norm(x::Number, p::Real=2)
```

For numbers, return $\left(|x|^p\right)^{1/p}$. This is equivalent to `vecnorm`.

source

```
norm(A::RowVector, q::Real=2)
```

For row vectors, return the q-norm of `A`, which is equivalent to the p-norm with value `p = q/(q-1)`. They coincide at `p = q = 2`.

The difference in norm between a vector space and its dual arises to preserve the relationship between duality and the inner product, and the result is consistent with the p-norm of `1 × n` matrix.

source

`Base.LinAlg.vecnorm` – Function.

```
vecnorm(A, p::Real=2)
```

For any iterable container `A` (including arrays of any dimension) of numbers (or any element type for which `norm` is defined), compute the p-norm (defaulting to p=2) as if `A` were a vector of the corresponding length.

The p-norm is defined as:

$$\|A\|_p = \left(\sum_{i=1}^{n} |a_i|^p \right)^{1/p}$$

with a_i the entries of A and n its length.

p can assume any numeric value (even though not all values produce a mathematically valid vector norm). In particular, `vecnorm(A, Inf)` returns the largest value in `abs(A)`, whereas `vecnorm(A, -Inf)` returns the smallest. If `A` is a matrix and p=2, then this is equivalent to the Frobenius norm.

Example

```
julia> vecnorm([1 2 3; 4 5 6; 7 8 9])
16.881943016134134

julia> vecnorm([1 2 3 4 5 6 7 8 9])
16.881943016134134
```

source

```
vecnorm(x::Number, p::Real=2)
```

For numbers, return $\left(|x|^p\right)^{1/p}$.

source

`Base.LinAlg.normalize!` – Function.

```
normalize!(v::AbstractVector, p::Real=2)
```

Normalize the vector v in-place so that its p-norm equals unity, i.e. `norm(v, p) == 1`. See also `normalize` and `vecnorm`.

source

`Base.LinAlg.normalize` – Function.

```
normalize(v::AbstractVector, p::Real=2)
```

Normalize the vector v so that its p-norm equals unity, i.e. `norm(v, p) == vecnorm(v, p) == 1`. See also `normalize!` and `vecnorm`.

Examples

```
julia> a = [1,2,4];

julia> b = normalize(a)
3-element Array{Float64,1}:
 0.218218
 0.436436
 0.872872

julia> norm(b)
1.0

julia> c = normalize(a, 1)
3-element Array{Float64,1}:
 0.142857
 0.285714
 0.571429

julia> norm(c, 1)
1.0
```

source

`Base.LinAlg.cond` – Function.

```
cond(M, p::Real=2)
```

Condition number of the matrix `M`, computed using the operator p-norm. Valid values for p are 1, 2 (default), or `Inf`.

source

`Base.LinAlg.condskeel` – Function.

```
condskeel(M, [x, p::Real=Inf])
```

$$\kappa_S(M,p) = \left\| |M| \left|M^{-1}\right| \right\|_p$$
$$\kappa_S(M,x,p) = \left\| |M| \left|M^{-1}\right| |x| \right\|_p$$

Skeel condition number κ_S of the matrix `M`, optionally with respect to the vector x, as computed using the operator p-norm. $|M|$ denotes the matrix of (entry wise) absolute values of M; $|M|_{ij} = |M_{ij}|$. Valid values for p are 1, 2 and `Inf` (default).

This quantity is also known in the literature as the Bauer condition number, relative condition number, or componentwise relative condition number.

source

`Base.LinAlg.trace` – Function.

```
trace(M)
```

Matrix trace. Sums the diagonal elements of `M`.

Example

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> trace(A)
5
```

source

`Base.LinAlg.det` – Function.

```
det(M)
```

Matrix determinant.

Example

```
julia> M = [1 0; 2 2]
2×2 Array{Int64,2}:
 1  0
 2  2

julia> det(M)
2.0
```

source

`Base.LinAlg.logdet` – Function.

```
logdet(M)
```

Log of matrix determinant. Equivalent to `log(det(M))`, but may provide increased accuracy and/or speed.

Examples

```
julia> M = [1 0; 2 2]
2×2 Array{Int64,2}:
 1  0
 2  2

julia> logdet(M)
0.6931471805599453

julia> logdet(eye(3))
0.0
```

source

`Base.LinAlg.logabsdet` – Function.

```
logabsdet(M)
```

Log of absolute value of matrix determinant. Equivalent to `(log(abs(det(M))), sign(det(M)))`, but may provide increased accuracy and/or speed.

source

`Base.inv` – Function.

```
inv(M)
```

Matrix inverse. Computes matrix `N` such that `M * N = I`, where `I` is the identity matrix. Computed by solving the left-division `N = M \ I`.

Example

```
julia> M = [2 5; 1 3]
2×2 Array{Int64,2}:
 2  5
 1  3

julia> N = inv(M)
2×2 Array{Float64,2}:
  3.0  -5.0
 -1.0   2.0

julia> M*N == N*M == eye(2)
true
```

source

`Base.LinAlg.pinv` – Function.

```
pinv(M[, tol::Real])
```

Computes the Moore-Penrose pseudoinverse.

For matrices `M` with floating point elements, it is convenient to compute the pseudoinverse by inverting only singular values above a given threshold, `tol`.

The optimal choice of `tol` varies both with the value of `M` and the intended application of the pseudoinverse. The default value of `tol` is `eps(real(float(one(eltype(M)))))*maximum(size(A))`, which is essentially machine epsilon for the real part of a matrix element multiplied by the larger matrix dimension. For inverting dense ill-conditioned matrices in a least-squares sense, `tol = sqrt(eps(real(float(one(eltype(M))))))` is recommended.

For more information, see [4], [5], [6], [7].

Example

```
julia> M = [1.5 1.3; 1.2 1.9]
2×2 Array{Float64,2}:
 1.5  1.3
 1.2  1.9

julia> N = pinv(M)
2×2 Array{Float64,2}:
  1.47287   -1.00775
 -0.930233   1.16279
```

```
julia> M * N
2×2 Array{Float64,2}:
 1.0          -2.22045e-16
 4.44089e-16   1.0
```

source

Base.LinAlg.nullspace – Function.

```
nullspace(M)
```

Basis for nullspace of M.

Example

```
julia> M = [1 0 0; 0 1 0; 0 0 0]
3×3 Array{Int64,2}:
 1  0  0
 0  1  0
 0  0  0

julia> nullspace(M)
3×1 Array{Float64,2}:
 0.0
 0.0
 1.0
```

source

Base.repmat – Function.

```
repmat(A, m::Integer, n::Integer=1)
```

Construct a matrix by repeating the given matrix (or vector) m times in dimension 1 and n times in dimension 2.

Examples

```
julia> repmat([1, 2, 3], 2)
6-element Array{Int64,1}:
 1
 2
 3
 1
 2
 3

julia> repmat([1, 2, 3], 2, 3)
```

[4] Issue 8859, "Fix least squares", https://github.com/JuliaLang/julia/pull/8859

[5] Åke Björck, "Numerical Methods for Least Squares Problems", SIAM Press, Philadelphia, 1996, "Other Titles in Applied Mathematics", Vol. 51. doi:10.1137/1.9781611971484

[6] G. W. Stewart, "Rank Degeneracy", SIAM Journal on Scientific and Statistical Computing, 5(2), 1984, 403-413. doi:10.1137/0905030

[7] Konstantinos Konstantinides and Kung Yao, "Statistical analysis of effective singular values in matrix rank determination", IEEE Transactions on Acoustics, Speech and Signal Processing, 36(5), 1988, 757-763. doi:10.1109/29.1585

```
6×3 Array{Int64,2}:
 1  1  1
 2  2  2
 3  3  3
 1  1  1
 2  2  2
 3  3  3
```

source

Base.repeat – Function.

```
repeat(A::AbstractArray; inner=ntuple(x->1, ndims(A)), outer=ntuple(x->1, ndims(A)))
```

Construct an array by repeating the entries of `A`. The i-th element of `inner` specifies the number of times that the individual entries of the i-th dimension of `A` should be repeated. The i-th element of `outer` specifies the number of times that a slice along the i-th dimension of `A` should be repeated. If `inner` or `outer` are omitted, no repetition is performed.

Examples

```
julia> repeat(1:2, inner=2)
4-element Array{Int64,1}:
 1
 1
 2
 2

julia> repeat(1:2, outer=2)
4-element Array{Int64,1}:
 1
 2
 1
 2

julia> repeat([1 2; 3 4], inner=(2, 1), outer=(1, 3))
4×6 Array{Int64,2}:
 1  2  1  2  1  2
 1  2  1  2  1  2
 3  4  3  4  3  4
 3  4  3  4  3  4
```

source

Base.kron – Function.

```
kron(A, B)
```

Kronecker tensor product of two vectors or two matrices.

Example

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4
```

```
julia> B = [im 1; 1 -im]
2×2 Array{Complex{Int64},2}:
 0+1im  1+0im
 1+0im  0-1im

julia> kron(A, B)
4×4 Array{Complex{Int64},2}:
 0+1im  1+0im  0+2im  2+0im
 1+0im  0-1im  2+0im  0-2im
 0+3im  3+0im  0+4im  4+0im
 3+0im  0-3im  4+0im  0-4im
```

source

Base.SparseArrays.blkdiag – Function.

```
blkdiag(A...)
```

Concatenate matrices block-diagonally. Currently only implemented for sparse matrices.

Example

```
julia> blkdiag(speye(3), 2*speye(2))
5×5 SparseMatrixCSC{Float64,Int64} with 5 stored entries:
  [1, 1]  =  1.0
  [2, 2]  =  1.0
  [3, 3]  =  1.0
  [4, 4]  =  2.0
  [5, 5]  =  2.0
```

source

Base.LinAlg.linreg – Function.

```
linreg(x, y)
```

Perform simple linear regression using Ordinary Least Squares. Returns a and b such that `a + b*x` is the closest straight line to the given points `(x, y)`, i.e., such that the squared error between y and `a + b*x` is minimized.

Examples:

```
using PyPlot
x = 1.0:12.0
y = [5.5, 6.3, 7.6, 8.8, 10.9, 11.79, 13.48, 15.02, 17.77, 20.81, 22.0, 22.99]
a, b = linreg(x, y)          # Linear regression
plot(x, y, "o")              # Plot (x, y) points
plot(x, a + b*x)             # Plot line determined by linear regression
```

See also:

\, cov, std, mean.

source

Base.LinAlg.expm – Function.

```
expm(A)
```

Compute the matrix exponential of A, defined by

$$e^A = \sum_{n=0}^{\infty} \frac{A^n}{n!}.$$

For symmetric or Hermitian A, an eigendecomposition (eigfact) is used, otherwise the scaling and squaring algorithm (see [8]) is chosen.

Example

```
julia> A = eye(2, 2)
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0

julia> expm(A)
2×2 Array{Float64,2}:
 2.71828  0.0
 0.0      2.71828
```

source

Base.LinAlg.logm – Function.

```
logm(A{T}::StridedMatrix{T})
```

If A has no negative real eigenvalue, compute the principal matrix logarithm of A, i.e. the unique matrix X such that $e^X = A$ and $-\pi < Im(\lambda) < \pi$ for all the eigenvalues λ of X. If A has nonpositive eigenvalues, a nonprincipal matrix function is returned whenever possible.

If A is symmetric or Hermitian, its eigendecomposition (eigfact) is used, if A is triangular an improved version of the inverse scaling and squaring method is employed (see [9] and [10]). For general matrices, the complex Schur form (schur) is computed and the triangular algorithm is used on the triangular factor.

Example

```
julia> A = 2.7182818 * eye(2)
2×2 Array{Float64,2}:
 2.71828  0.0
 0.0      2.71828

julia> logm(A)
2×2 Symmetric{Float64,Array{Float64,2}}:
 1.0  0.0
 0.0  1.0
```

source

[8] Nicholas J. Higham, "The squaring and scaling method for the matrix exponential revisited", SIAM Journal on Matrix Analysis and Applications, 26(4), 2005, 1179-1193. doi:10.1137/090768539

[9] Awad H. Al-Mohy and Nicholas J. Higham, "Improved inverse scaling and squaring algorithms for the matrix logarithm", SIAM Journal on Scientific Computing, 34(4), 2012, C153-C169. doi:10.1137/110852553

[10] Awad H. Al-Mohy, Nicholas J. Higham and Samuel D. Relton, "Computing the Fréchet derivative of the matrix logarithm and estimating the condition number", SIAM Journal on Scientific Computing, 35(4), 2013, C394-C410. doi:10.1137/120885991

Base.LinAlg.sqrtm – Function.

```
sqrtm(A)
```

If A has no negative real eigenvalues, compute the principal matrix square root of A, that is the unique matrix X with eigenvalues having positive real part such that $X^2 = A$. Otherwise, a nonprincipal square root is returned.

If A is symmetric or Hermitian, its eigendecomposition (eigfact) is used to compute the square root. Otherwise, the square root is determined by means of the Björck-Hammarling method [11], which computes the complex Schur form (schur) and then the complex square root of the triangular factor.

Example

```
julia> A = [4 0; 0 4]
2×2 Array{Int64,2}:
 4  0
 0  4

julia> sqrtm(A)
2×2 Array{Float64,2}:
 2.0  0.0
 0.0  2.0
```

source

Base.LinAlg.lyap – Function.

```
lyap(A, C)
```

Computes the solution X to the continuous Lyapunov equation AX + XA' + C = 0, where no eigenvalue of A has a zero real part and no two eigenvalues are negative complex conjugates of each other.

source

Base.LinAlg.sylvester – Function.

```
sylvester(A, B, C)
```

Computes the solution X to the Sylvester equation AX + XB + C = 0, where A, B and C have compatible dimensions and A and -B have no eigenvalues with equal real part.

source

Base.LinAlg.issymmetric – Function.

```
issymmetric(A) -> Bool
```

Test whether a matrix is symmetric.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Array{Int64,2}:
 1   2
 2  -1
```

[11] Åke Björck and Sven Hammarling, "A Schur method for the square root of a matrix", Linear Algebra and its Applications, 52-53, 1983, 127-140. doi:10.1016/0024-3795(83)80010-X

```
julia> issymmetric(a)
true

julia> b = [1 im; -im 1]
2×2 Array{Complex{Int64},2}:
 1+0im  0+1im
 0-1im  1+0im

julia> issymmetric(b)
false
```

source

Base.LinAlg.isposdef – Function.

```
isposdef(A) -> Bool
```

Test whether a matrix is positive definite.

Example

```
julia> A = [1 2; 2 50]
2×2 Array{Int64,2}:
 1   2
 2  50

julia> isposdef(A)
true
```

source

Base.LinAlg.isposdef! – Function.

```
isposdef!(A) -> Bool
```

Test whether a matrix is positive definite, overwriting `A` in the process.

Example

```
julia> A = [1. 2.; 2. 50.];

julia> isposdef!(A)
true

julia> A
2×2 Array{Float64,2}:
 1.0  2.0
 2.0  6.78233
```

source

Base.LinAlg.istril – Function.

```
istril(A) -> Bool
```

Test whether a matrix is lower triangular.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Array{Int64,2}:
 1   2
 2  -1

julia> istril(a)
false

julia> b = [1 0; -im -1]
2×2 Array{Complex{Int64},2}:
 1+0im   0+0im
 0-1im  -1+0im

julia> istril(b)
true
```

source

Base.LinAlg.istriu – Function.

```
istriu(A) -> Bool
```

Test whether a matrix is upper triangular.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Array{Int64,2}:
 1   2
 2  -1

julia> istriu(a)
false

julia> b = [1 im; 0 -1]
2×2 Array{Complex{Int64},2}:
 1+0im   0+1im
 0+0im  -1+0im

julia> istriu(b)
true
```

source

Base.LinAlg.isdiag – Function.

```
isdiag(A) -> Bool
```

Test whether a matrix is diagonal.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Array{Int64,2}:
 1   2
 2  -1
```

```
julia> isdiag(a)
false

julia> b = [im 0; 0 -im]
2×2 Array{Complex{Int64},2}:
 0+1im  0+0im
 0+0im  0-1im

julia> isdiag(b)
true
```

source

Base.LinAlg.ishermitian – Function.

```
ishermitian(A) -> Bool
```

Test whether a matrix is Hermitian.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Array{Int64,2}:
 1   2
 2  -1

julia> ishermitian(a)
true

julia> b = [1 im; -im 1]
2×2 Array{Complex{Int64},2}:
 1+0im  0+1im
 0-1im  1+0im

julia> ishermitian(b)
true
```

source

Base.LinAlg.RowVector – Type.

```
RowVector(vector)
```

A lazy-view wrapper of an `AbstractVector`, which turns a length-n vector into a 1×n shaped row vector and represents the transpose of a vector (the elements are also transposed recursively). This type is usually constructed (and unwrapped) via the `transpose` function or `.'` operator (or related `ctranspose` or `'` operator).

By convention, a vector can be multiplied by a matrix on its left (`A * v`) whereas a row vector can be multiplied by a matrix on its right (such that `v.' * A = (A.' * v).'`). It differs from a 1×n-sized matrix by the facts that its transpose returns a vector and the inner product `v1.' * v2` returns a scalar, but will otherwise behave similarly.

source

Base.LinAlg.ConjArray – Type.

```
ConjArray(array)
```

A lazy-view wrapper of an `AbstractArray`, taking the elementwise complex conjugate. This type is usually constructed (and unwrapped) via the `conj` function (or related `ctranspose`), but currently this is the default behavior for `RowVector` only. For other arrays, the `ConjArray` constructor can be used directly.

Examples

```
julia> [1+im, 1-im]'
1×2 RowVector{Complex{Int64},ConjArray{Complex{Int64},1,Array{Complex{Int64},1}}}:
 1-1im  1+1im

julia> ConjArray([1+im 0; 0 1-im])
2×2 ConjArray{Complex{Int64},2,Array{Complex{Int64},2}}:
 1-1im  0+0im
 0+0im  1+1im
```

source

`Base.transpose` – Function.

```
transpose(A::AbstractMatrix)
```

The transposition operator (`.'`).

Example

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Array{Int64,2}:
 1  2  3
 4  5  6
 7  8  9

julia> transpose(A)
3×3 Array{Int64,2}:
 1  4  7
 2  5  8
 3  6  9
```

source

```
transpose(v::AbstractVector)
```

The transposition operator (`.'`).

Example

```
julia> v = [1,2,3]
3-element Array{Int64,1}:
 1
 2
 3

julia> transpose(v)
1×3 RowVector{Int64,Array{Int64,1}}:
 1  2  3
```

source

`Base.LinAlg.transpose!` – Function.

```
transpose!(dest,src)
```

Transpose array `src` and store the result in the preallocated array `dest`, which should have a size corresponding to `(size(src,2),size(src,1))`. No in-place transposition is supported and unexpected results will happen if `src` and `dest` have overlapping memory regions.

source

`Base.ctranspose` – Function.

```
ctranspose(A)
```

The conjugate transposition operator (`'`).

Example

```
julia> A =  [3+2im 9+2im; 8+7im  4+6im]
2×2 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im

julia> ctranspose(A)
2×2 Array{Complex{Int64},2}:
 3-2im  8-7im
 9-2im  4-6im
```

source

`Base.LinAlg.ctranspose!` – Function.

```
ctranspose!(dest,src)
```

Conjugate transpose array `src` and store the result in the preallocated array `dest`, which should have a size corresponding to `(size(src,2),size(src,1))`. No in-place transposition is supported and unexpected results will happen if `src` and `dest` have overlapping memory regions.

source

`Base.LinAlg.eigs` – Method.

```
eigs(A; nev=6, ncv=max(20,2*nev+1), which=:LM, tol=0.0, maxiter=300, sigma=nothing, ritzvec=
    true, v0=zeros((0,))) -> (d,[v,],nconv,niter,nmult,resid)
```

Computes eigenvalues d of `A` using implicitly restarted Lanczos or Arnoldi iterations for real symmetric or general nonsymmetric matrices respectively.

The following keyword arguments are supported:

- `nev`: Number of eigenvalues
- `ncv`: Number of Krylov vectors used in the computation; should satisfy `nev+1 <= ncv <= n` for real symmetric problems and `nev+2 <= ncv <= n` for other problems, where `n` is the size of the input matrix `A`. The default is `ncv = max(20,2*nev+1)`. Note that these restrictions limit the input matrix `A` to be of dimension at least 2.
- `which`: type of eigenvalues to compute. See the note below.

which	type of eigenvalues
:LM	eigenvalues of largest magnitude (default)
:SM	eigenvalues of smallest magnitude
:LR	eigenvalues of largest real part
:SR	eigenvalues of smallest real part
:LI	eigenvalues of largest imaginary part (nonsymmetric or complex A only)
:SI	eigenvalues of smallest imaginary part (nonsymmetric or complex A only)
:BE	compute half of the eigenvalues from each end of the spectrum, biased in favor of the high end. (real symmetric A only)

- `tol`: parameter defining the relative tolerance for convergence of Ritz values (eigenvalue estimates). A Ritz value is considered converged when its associated residual is less than or equal to the product of `tol` and $max(^{2/3}, ||)$, where `= eps(real(eltype(A)))/2` is LAPACK's machine epsilon. The residual associated with and its corresponding Ritz vector v is defined as the norm $||Av - v||$. The specified value of `tol` should be positive; otherwise, it is ignored and is used instead. Default: .
- `maxiter`: Maximum number of iterations (default = 300)
- `sigma`: Specifies the level shift used in inverse iteration. If `nothing` (default), defaults to ordinary (forward) iterations. Otherwise, find eigenvalues close to `sigma` using shift and invert iterations.
- `ritzvec`: Returns the Ritz vectors v (eigenvectors) if `true`
- `v0`: starting vector from which to start the iterations

`eigs` returns the `nev` requested eigenvalues in `d`, the corresponding Ritz vectors `v` (only if `ritzvec=true`), the number of converged eigenvalues `nconv`, the number of iterations `niter` and the number of matrix vector multiplications `nmult`, as well as the final residual vector `resid`.

Example

```
julia> A = spdiagm(1:4);

julia> λ,  = eigs(A, nev = 2);

julia> λ
2-element Array{Float64,1}:
 4.0
 3.0
```

Note

The `sigma` and `which` keywords interact: the description of eigenvalues searched for by `which` do *not* necessarily refer to the eigenvalues of `A`, but rather the linear operator constructed by the specification of the iteration mode implied by `sigma`.

sigma	iteration mode	which refers to eigenvalues of
nothing	ordinary (forward)	A
real or complex	inverse with level shift sigma	$(A - \sigma I)^{-1}$

Note

Although `tol` has a default value, the best choice depends strongly on the matrix `A`. We recommend that users _always_ specify a value for `tol` which suits their specific needs.

For details of how the errors in the computed eigenvalues are estimated, see:

- B. N. Parlett, "The Symmetric Eigenvalue Problem", SIAM: Philadelphia, 2/e (1998), Ch. 13.2, "Accessing Accuracy in Lanczos Problems", pp. 290-292 ff.
- R. B. Lehoucq and D. C. Sorensen, "Deflation Techniques for an Implicitly Restarted Arnoldi Iteration", SIAM Journal on Matrix Analysis and Applications (1996), 17(4), 789–821. doi:10.1137/S0895479895281484

source

`Base.LinAlg.eigs` – Method.

```
eigs(A, B; nev=6, ncv=max(20,2*nev+1), which=:LM, tol=0.0, maxiter=300, sigma=nothing,
     ritzvec=true, v0=zeros((0,))) -> (d,[v,],nconv,niter,nmult,resid)
```

Computes generalized eigenvalues `d` of `A` and `B` using implicitly restarted Lanczos or Arnoldi iterations for real symmetric or general nonsymmetric matrices respectively.

The following keyword arguments are supported:

- `nev`: Number of eigenvalues
- `ncv`: Number of Krylov vectors used in the computation; should satisfy `nev+1 <= ncv <= n` for real symmetric problems and `nev+2 <= ncv <= n` for other problems, where `n` is the size of the input matrices `A` and `B`. The default is `ncv = max(20,2*nev+1)`. Note that these restrictions limit the input matrix `A` to be of dimension at least 2.
- `which`: type of eigenvalues to compute. See the note below.

`which`	type of eigenvalues
`:LM`	eigenvalues of largest magnitude (default)
`:SM`	eigenvalues of smallest magnitude
`:LR`	eigenvalues of largest real part
`:SR`	eigenvalues of smallest real part
`:LI`	eigenvalues of largest imaginary part (nonsymmetric or complex `A` only)
`:SI`	eigenvalues of smallest imaginary part (nonsymmetric or complex `A` only)
`:BE`	compute half of the eigenvalues from each end of the spectrum, biased in favor of the high end. (real symmetric `A` only)

- `tol`: relative tolerance used in the convergence criterion for eigenvalues, similar to `tol` in the `eigs(A)` method for the ordinary eigenvalue problem, but effectively for the eigenvalues of $B^{-1}A$ instead of A. See the documentation for the ordinary eigenvalue problem in `eigs(A)` and the accompanying note about `tol`.
- `maxiter`: Maximum number of iterations (default = 300)
- `sigma`: Specifies the level shift used in inverse iteration. If `nothing` (default), defaults to ordinary (forward) iterations. Otherwise, find eigenvalues close to `sigma` using shift and invert iterations.
- `ritzvec`: Returns the Ritz vectors `v` (eigenvectors) if `true`
- `v0`: starting vector from which to start the iterations

`eigs` returns the `nev` requested eigenvalues in `d`, the corresponding Ritz vectors `v` (only if `ritzvec=true`), the number of converged eigenvalues `nconv`, the number of iterations `niter` and the number of matrix vector multiplications `nmult`, as well as the final residual vector `resid`.

Example

```
julia> A = speye(4, 4); B = spdiagm(1:4);

julia> λ,  = eigs(A, B, nev = 2);

julia> λ
2-element Array{Float64,1}:
 1.0
 0.5
```

Note

The `sigma` and `which` keywords interact: the description of eigenvalues searched for by `which` do *not* necessarily refer to the eigenvalue problem $Av = Bv\lambda$, but rather the linear operator constructed by the specification of the iteration mode implied by `sigma`.

`sigma`	iteration mode	`which` refers to the problem
`nothing`	ordinary (forward)	$Av = Bv\lambda$
real or complex	inverse with level shift `sigma`	$(A - \sigma B)^{-1}B = v\nu$

source

`Base.LinAlg.svds` – Function.

```
svds(A; nsv=6, ritzvec=true, tol=0.0, maxiter=1000, ncv=2*nsv, u0=zeros((0,)), v0=zeros((0,))
    ) -> (SVD([left_sv,] s, [right_sv,]), nconv, niter, nmult, resid)
```

Computes the largest singular values `s` of `A` using implicitly restarted Lanczos iterations derived from `eigs`.

Inputs

- `A`: Linear operator whose singular values are desired. `A` may be represented as a subtype of `AbstractArray`, e.g., a sparse matrix, or any other type supporting the four methods `size(A)`, `eltype(A)`, `A * vector`, and `A' * vector`.
- `nsv`: Number of singular values. Default: 6.
- `ritzvec`: If `true`, return the left and right singular vectors `left_sv` and `right_sv`. If `false`, omit the singular vectors. Default: `true`.
- `tol`: tolerance, see `eigs`.
- `maxiter`: Maximum number of iterations, see `eigs`. Default: 1000.
- `ncv`: Maximum size of the Krylov subspace, see `eigs` (there called `nev`). Default: `2*nsv`.
- `u0`: Initial guess for the first left Krylov vector. It may have length `m` (the first dimension of `A`), or 0.
- `v0`: Initial guess for the first right Krylov vector. It may have length `n` (the second dimension of `A`), or 0.

Outputs

- `svd`: An `SVD` object containing the left singular vectors, the requested values, and the right singular vectors. If `ritzvec = false`, the left and right singular vectors will be empty.
- `nconv`: Number of converged singular values.
- `niter`: Number of iterations.
- `nmult`: Number of matrix–vector products used.
- `resid`: Final residual vector.

Example

```
julia> A = spdiagm(1:4);

julia> s = svds(A, nsv = 2)[1];

julia> s[:S]
2-element Array{Float64,1}:
 4.0
 3.0
```

Implementation

`svds(A)` is formally equivalent to calling `eigs` to perform implicitly restarted Lanczos tridiagonalization on the Hermitian matrix $\begin{pmatrix} 0 & A' \\ A & 0 \end{pmatrix}$, whose eigenvalues are plus and minus the singular values of A.

source

`Base.LinAlg.peakflops` – Function.

```
peakflops(n::Integer=2000; parallel::Bool=false)
```

`peakflops` computes the peak flop rate of the computer by using double precision `gemm!`. By default, if no arguments are specified, it multiplies a matrix of size `n x n`, where `n = 2000`. If the underlying BLAS is using multiple threads, higher flop rates are realized. The number of BLAS threads can be set with `BLAS.set_num_threads(n)`.

If the keyword argument `parallel` is set to `true`, `peakflops` is run in parallel on all the worker processors. The flop rate of the entire parallel computer is returned. When running in parallel, only 1 BLAS thread is used. The argument n still refers to the size of the problem that is solved on each processor.

source

52.2 Operaciones matriciales de bajo nivel

Las operaciones de matrices que involucran operaciones de transposición como `A' \ B` son convertidas por el analizador de Julia en llamadas a funciones especialmente nombradas como `Ac_ldiv_B`. Si desea sobrecargar estas operaciones para sus propios tipos, le será útil conocer los nombres de estas funciones.

Además, en muchos casos, hay versiones in situ de operaciones matriciales que le permiten suministrar un vector o matriz de salida preasignada. Esto es útil cuando se optimiza código crítico para evitar la sobrecarga de las asignaciones repetidas. Estas operaciones in situ tienen el sufijo `!` a continuación (por ejemplo, `A_mul_B!`) de acuerdo con la convención habitual de Julia.

`Base.LinAlg.A_ldiv_B!` – Function.

```
A_ldiv_B!([Y,] A, B) -> Y
```

Compute `A \ B` in-place and store the result in `Y`, returning the result. If only two arguments are passed, then `A_ldiv_B!(A, B)` overwrites `B` with the result.

The argument `A` should *not* be a matrix. Rather, instead of matrices it should be a factorization object (e.g. produced by `factorize` or `cholfact`). The reason for this is that factorization itself is both expensive and typically allocates memory (although it can also be done in-place via, e.g., `lufact!`), and performance-critical situations requiring `A_ldiv_B!` usually also require fine-grained control over the factorization of `A`.

source

Base.A_ldiv_Bc – Function.

```
A_ldiv_Bc(A, B)
```

For matrices or vectors A and B, calculates $A \setminus B$.

source

Base.A_ldiv_Bt – Function.

```
A_ldiv_Bt(A, B)
```

For matrices or vectors A and B, calculates $A \setminus B$.

source

Base.LinAlg.A_mul_B! – Function.

```
A_mul_B!(Y, A, B) -> Y
```

Calculates the matrix-matrix or matrix-vector product AB and stores the result in Y, overwriting the existing value of Y. Note that Y must not be aliased with either A or B.

Example

```
julia> A=[1.0 2.0; 3.0 4.0]; B=[1.0 1.0; 1.0 1.0]; Y = similar(B); A_mul_B!(Y, A, B);

julia> Y
2×2 Array{Float64,2}:
 3.0  3.0
 7.0  7.0
```

source

Base.A_mul_Bc – Function.

```
A_mul_Bc(A, B)
```

For matrices or vectors A and B, calculates AB.

source

Base.A_mul_Bt – Function.

```
A_mul_Bt(A, B)
```

For matrices or vectors A and B, calculates AB.

source

Base.A_rdiv_Bc – Function.

```
A_rdiv_Bc(A, B)
```

For matrices or vectors A and B, calculates A/B.

source

Base.A_rdiv_Bt – Function.

```
A_rdiv_Bt(A, B)
```

For matrices or vectors A and B, calculates A/B.

source

Base.Ac_ldiv_B – Function.

```
Ac_ldiv_B(A, B)
```

For matrices or vectors A and B, calculates $A \setminus B$.

source

Base.LinAlg.Ac_ldiv_B! – Function.

```
Ac_ldiv_B!([Y,] A, B) -> Y
```

Similar to A_ldiv_B!, but return $A \setminus B$, computing the result in-place in Y (or overwriting B if Y is not supplied).

source

Base.Ac_ldiv_Bc – Function.

```
Ac_ldiv_Bc(A, B)
```

For matrices or vectors A and B, calculates $A \setminus B$.

source

Base.Ac_mul_B – Function.

```
Ac_mul_B(A, B)
```

For matrices or vectors A and B, calculates AB.

source

Base.Ac_mul_Bc – Function.

```
Ac_mul_Bc(A, B)
```

For matrices or vectors A and B, calculates AB.

source

Base.Ac_rdiv_B – Function.

```
Ac_rdiv_B(A, B)
```

For matrices or vectors A and B, calculates A/B.

source

Base.Ac_rdiv_Bc – Function.

```
Ac_rdiv_Bc(A, B)
```

For matrices or vectors A and B, calculates A/B.

source

Base.At_ldiv_B – Function.

```
At_ldiv_B(A, B)
```

For matrices or vectors A and B, calculates $A \setminus B$.

source

Base.LinAlg.At_ldiv_B! – Function.

```
At_ldiv_B!([Y,] A, B) -> Y
```

Similar to A_ldiv_B!, but return $A \setminus B$, computing the result in-place in Y (or overwriting B if Y is not supplied).

source

Base.At_ldiv_Bt – Function.

```
At_ldiv_Bt(A, B)
```

For matrices or vectors A and B, calculates $A \setminus B$.

source

Base.At_mul_B – Function.

```
At_mul_B(A, B)
```

For matrices or vectors A and B, calculates AB.

source

Base.At_mul_Bt – Function.

```
At_mul_Bt(A, B)
```

For matrices or vectors A and B, calculates AB.

source

Base.At_rdiv_B – Function.

```
At_rdiv_B(A, B)
```

For matrices or vectors A and B, calculates A/B.

source

Base.At_rdiv_Bt – Function.

```
At_rdiv_Bt(A, B)
```

For matrices or vectors A and B, calculates A/B.

source

52.3 Funciones BLAS

En Julia (como en gran parte de la computación científica), las operaciones de álgebra lineal densa se basan en la biblioteca LAPACK, que a su vez se construye sobre bloques de construcción básicos de álgebra lineal conocidos como BLAS. Hay implementaciones altamente optimizadas de BLAS disponibles para cada arquitectura de computadora, y algunas veces en rutinas de álgebra lineal de alto rendimiento, es útil llamar directamente a las funciones de BLAS.

`Base.LinAlg.BLAS` proporciona envoltorios para algunas de las funciones de BLAS. Esas funciones de BLAS que sobrescriben una de las matrices de entrada tienen nombres que terminan en `'!'`. Normalmente, una función BLAS tiene cuatro métodos definidos, para los arrays `Float64`, `Float32`, `Complex128` y`Complex64`.

Argumentos de tipo carácter en BLAS

Muchas funciones BLAS aceptan argumentos que determinan si se debe transponer un argumento (`trans`), qué triángulo de una matriz referenciar (`uplo` o `ul`), si se puede suponer que la diagonal de una matriz triangular está formada por unos (`dA`) o a qué lado de una multiplicación de matrices pertenece el argumento de entrada (`side`). Las posibilidades son:

Orden de Multiplicación

side	Significado
`'L'`	El argumento va al lado *izquierdo* de una operación matriz-matriz.
`'R'`	El argumento va al lado *derecho* de una operación matriz-matriz.

Referencia sobre el Triángulo

uplo/ul	Significado
`'U'`	Solo se usará el triángulo *superior* de la matriz.
`'L'`	Solo se usará el triángulo *inferior* de la matriz.

Operación de Transposición

trans/tX	Significado
`'N'`	La matriz de entrada `X` no es transpuesta ni conjugada.
`'T'`	La matriz de entrada `X` será transpuesta.
`'C'`	La matriz de entrada `X` será conjugada y transpuesta.

Unidades en la Diagonal

diag/dX	Significado
`'N'`	Los valores diagonales de la matriz `X` serán leídos.
`'U'`	Se supone que los elementos de la diagonal de la matriz `X` son todos unos.

`Base.LinAlg.BLAS.dotu` – Function.

```
dotu(n, X, incx, Y, incy)
```

Dot function for two complex vectors consisting of `n` elements of array `X` with stride `incx` and `n` elements of array `Y` with stride `incy`.

Example:

```
julia> Base.BLAS.dotu(10, im*ones(10), 1, complex.(ones(20), ones(20)), 2)
-10.0 + 10.0im
```

source

Base.LinAlg.BLAS.dotc – Function.

```
dotc(n, X, incx, U, incy)
```

Dot function for two complex vectors, consisting of n elements of array X with stride incx and n elements of array U with stride incy, conjugating the first vector.

Example:

```
julia> Base.BLAS.dotc(10, im*ones(10), 1, complex.(ones(20), ones(20)), 2)
10.0 - 10.0im
```

source

Base.LinAlg.BLAS.blascopy! – Function.

```
blascopy!(n, X, incx, Y, incy)
```

Copy n elements of array X with stride incx to array Y with stride incy. Returns Y.

source

Base.LinAlg.BLAS.nrm2 – Function.

```
nrm2(n, X, incx)
```

2-norm of a vector consisting of n elements of array X with stride incx.

Example:

```
julia> Base.BLAS.nrm2(4, ones(8), 2)
2.0

julia> Base.BLAS.nrm2(1, ones(8), 2)
1.0
```

source

Base.LinAlg.BLAS.asum – Function.

```
asum(n, X, incx)
```

Sum of the absolute values of the first n elements of array X with stride incx.

Example:

```
julia> Base.BLAS.asum(5, im*ones(10), 2)
5.0

julia> Base.BLAS.asum(2, im*ones(10), 5)
2.0
```

source

Base.LinAlg.axpy! – Function.

```
axpy!(a, X, Y)
```

Overwrite Y with a*X + Y, where a is a scalar. Returns Y.

Example:

```
julia> x = [1; 2; 3];

julia> y = [4; 5; 6];

julia> Base.BLAS.axpy!(2, x, y)
3-element Array{Int64,1}:
  6
  9
 12
```

source

Base.LinAlg.BLAS.scal! – Function.

```
scal!(n, a, X, incx)
```

Overwrite X with a*X for the first n elements of array X with stride `incx`. Returns X.

source

Base.LinAlg.BLAS.scal – Function.

```
scal(n, a, X, incx)
```

Returns X scaled by a for the first n elements of array X with stride `incx`.

source

Base.LinAlg.BLAS.ger! – Function.

```
ger!(alpha, x, y, A)
```

Rank-1 update of the matrix A with vectors x and y as `alpha*x*y' + A`.

source

Base.LinAlg.BLAS.syr! – Function.

```
syr!(uplo, alpha, x, A)
```

Rank-1 update of the symmetric matrix A with vector x as `alpha*x*x.' + A`. `uplo` controls which triangle of A is updated. Returns A.

source

Base.LinAlg.BLAS.syrk! – Function.

```
syrk!(uplo, trans, alpha, A, beta, C)
```

Rank-k update of the symmetric matrix C as `alpha*A*A.' + beta*C` or `alpha*A.'*A + beta*C` according to `trans`. Only the `uplo` triangle of C is used. Returns C.

source

Base.LinAlg.BLAS.syrk – Function.

```
syrk(uplo, trans, alpha, A)
```

Returns either the upper triangle or the lower triangle of A, according to uplo, of alpha*A*A.' or alpha*A.'*A, according to trans.

source

Base.LinAlg.BLAS.her! – Function.

```
her!(uplo, alpha, x, A)
```

Methods for complex arrays only. Rank-1 update of the Hermitian matrix A with vector x as alpha*x*x' + A. uplo controls which triangle of A is updated. Returns A.

source

Base.LinAlg.BLAS.herk! – Function.

```
herk!(uplo, trans, alpha, A, beta, C)
```

Methods for complex arrays only. Rank-k update of the Hermitian matrix C as alpha*A*A' + beta*C or alpha*A'*A + beta*C according to trans. Only the uplo triangle of C is updated. Returns C.

source

Base.LinAlg.BLAS.herk – Function.

```
herk(uplo, trans, alpha, A)
```

Methods for complex arrays only. Returns the uplo triangle of alpha*A*A' or alpha*A'*A, according to trans.

source

Base.LinAlg.BLAS.gbmv! – Function.

```
gbmv!(trans, m, kl, ku, alpha, A, x, beta, y)
```

Update vector y as alpha*A*x + beta*y or alpha*A'*x + beta*y according to trans. The matrix A is a general band matrix of dimension m by size(A,2) with kl sub-diagonals and ku super-diagonals. alpha and beta are scalars. Returns the updated y.

source

Base.LinAlg.BLAS.gbmv – Function.

```
gbmv(trans, m, kl, ku, alpha, A, x)
```

Returns alpha*A*x or alpha*A'*x according to trans. The matrix A is a general band matrix of dimension m by size(A,2) with kl sub-diagonals and ku super-diagonals, and alpha is a scalar.

source

Base.LinAlg.BLAS.sbmv! – Function.

```
sbmv!(uplo, k, alpha, A, x, beta, y)
```

Update vector y as `alpha*A*x + beta*y` where `A` is a a symmetric band matrix of order `size(A,2)` with k super-diagonals stored in the argument `A`. The storage layout for `A` is described the reference BLAS module, level-2 BLAS at http://www.netlib.org/lapack/explore-html/. Only the `uplo` triangle of `A` is used.

Returns the updated y.

source

`Base.LinAlg.BLAS.sbmv` – Method.

```
sbmv(uplo, k, alpha, A, x)
```

Returns `alpha*A*x` where `A` is a symmetric band matrix of order `size(A,2)` with k super-diagonals stored in the argument `A`. Only the `uplo` triangle of `A` is used.

source

`Base.LinAlg.BLAS.sbmv` – Method.

```
sbmv(uplo, k, A, x)
```

Returns `A*x` where `A` is a symmetric band matrix of order `size(A,2)` with k super-diagonals stored in the argument `A`. Only the `uplo` triangle of `A` is used.

source

`Base.LinAlg.BLAS.gemm!` – Function.

```
gemm!(tA, tB, alpha, A, B, beta, C)
```

Update `C` as `alpha*A*B + beta*C` or the other three variants according to `tA` and `tB`. Returns the updated `C`.

source

`Base.LinAlg.BLAS.gemm` – Method.

```
gemm(tA, tB, alpha, A, B)
```

Returns `alpha*A*B` or the other three variants according to `tA` and `tB`.

source

`Base.LinAlg.BLAS.gemm` – Method.

```
gemm(tA, tB, A, B)
```

Returns `A*B` or the other three variants according to `tA` and `tB`.

source

`Base.LinAlg.BLAS.gemv!` – Function.

```
gemv!(tA, alpha, A, x, beta, y)
```

Update the vector y as `alpha*A*x + beta*y` or `alpha*A'x + beta*y` according to `tA`. `alpha` and `beta` are scalars. Returns the updated y.

source

`Base.LinAlg.BLAS.gemv` – Method.

```
gemv(tA, alpha, A, x)
```

Returns `alpha*A*x` or `alpha*A'x` according to `tA`. `alpha` is a scalar.

source

`Base.LinAlg.BLAS.gemv` – Method.

```
gemv(tA, A, x)
```

Returns `A*x` or `A'x` according to `tA`.

source

`Base.LinAlg.BLAS.symm!` – Function.

```
symm!(side, ul, alpha, A, B, beta, C)
```

Update `C` as `alpha*A*B + beta*C` or `alpha*B*A + beta*C` according to `side`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used. Returns the updated `C`.

source

`Base.LinAlg.BLAS.symm` – Method.

```
symm(side, ul, alpha, A, B)
```

Returns `alpha*A*B` or `alpha*B*A` according to `side`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used.

source

`Base.LinAlg.BLAS.symm` – Method.

```
symm(side, ul, A, B)
```

Returns `A*B` or `B*A` according to `side`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used.

source

`Base.LinAlg.BLAS.symv!` – Function.

```
symv!(ul, alpha, A, x, beta, y)
```

Update the vector `y` as `alpha*A*x + beta*y`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used. `alpha` and `beta` are scalars. Returns the updated `y`.

source

`Base.LinAlg.BLAS.symv` – Method.

```
symv(ul, alpha, A, x)
```

Returns `alpha*A*x`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used. `alpha` is a scalar.

source

`Base.LinAlg.BLAS.symv` – Method.

```
symv(ul, A, x)
```

Returns `A*x`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used.

source

`Base.LinAlg.BLAS.trmm!` – Function.

```
trmm!(side, ul, tA, dA, alpha, A, B)
```

Update `B` as `alpha*A*B` or one of the other three variants determined by `side` and `tA`. Only the `ul` triangle of `A` is used. `dA` determines if the diagonal values are read or are assumed to be all ones. Returns the updated `B`.

source

`Base.LinAlg.BLAS.trmm` – Function.

```
trmm(side, ul, tA, dA, alpha, A, B)
```

Returns `alpha*A*B` or one of the other three variants determined by `side` and `tA`. Only the `ul` triangle of `A` is used. `dA` determines if the diagonal values are read or are assumed to be all ones.

source

`Base.LinAlg.BLAS.trsm!` – Function.

```
trsm!(side, ul, tA, dA, alpha, A, B)
```

Overwrite `B` with the solution to `A*X = alpha*B` or one of the other three variants determined by `side` and `tA`. Only the `ul` triangle of `A` is used. `dA` determines if the diagonal values are read or are assumed to be all ones. Returns the updated `B`.

source

`Base.LinAlg.BLAS.trsm` – Function.

```
trsm(side, ul, tA, dA, alpha, A, B)
```

Returns the solution to `A*X = alpha*B` or one of the other three variants determined by determined by `side` and `tA`. Only the `ul` triangle of `A` is used. `dA` determines if the diagonal values are read or are assumed to be all ones.

source

`Base.LinAlg.BLAS.trmv!` – Function.

```
trmv!(ul, tA, dA, A, b)
```

Returns `op(A)*b`, where `op` is determined by `tA`. Only the `ul` triangle of `A` is used. `dA` determines if the diagonal values are read or are assumed to be all ones. The multiplication occurs in-place on `b`.

source

`Base.LinAlg.BLAS.trmv` – Function.

```
trmv(ul, tA, dA, A, b)
```

Returns `op(A)*b`, where `op` is determined by `tA`. Only the `ul` triangle of `A` is used. `dA` determines if the diagonal values are read or are assumed to be all ones.

source

Base.LinAlg.BLAS.trsv! – Function.

```
trsv!(ul, tA, dA, A, b)
```

Overwrite b with the solution to `A*x = b` or one of the other two variants determined by `tA` and `ul`. `dA` determines if the diagonal values are read or are assumed to be all ones. Returns the updated b.

source

Base.LinAlg.BLAS.trsv – Function.

```
trsv(ul, tA, dA, A, b)
```

Returns the solution to `A*x = b` or one of the other two variants determined by `tA` and `ul`. `dA` determines if the diagonal values are read or are assumed to be all ones.

source

Base.LinAlg.BLAS.set_num_threads – Function.

```
set_num_threads(n)
```

Set the number of threads the BLAS library should use.

source

Base.LinAlg.I – Constant.

```
I
```

An object of type `UniformScaling`, representing an identity matrix of any size.

Example

```
julia> ones(5, 6) * I == ones(5, 6)
true

julia> [1 2im 3; 1im 2 3] * I
2×3 Array{Complex{Int64},2}:
 1+0im  0+2im  3+0im
 0+1im  2+0im  3+0im
```

source

52.4 Funciones LAPACK

`Base.LinAlg.LAPACK` proporciona *wrappers* para algunas de las funciones LAPACK para álgebra lineal. Las funciones que sobrescriben una de las matrices de entrada tienen nombres que terminan en `'!'`.

Por lo general, una función tiene 4 métodos definidos, uno para los arrays `Float64`, `Float32`, `Complex128` y `Complex64`.

Tenga en cuenta que la API LAPACK proporcionada por Julia puede y va a cambiar en el futuro. Dado que esta API no está orientada al usuario, no existe el compromiso de admitir/desaprobar este conjunto específico de funciones en futuras versiones.

Base.LinAlg.LAPACK.gbtrf! – Function.

```
gbtrf!(kl, ku, m, AB) -> (AB, ipiv)
```

Compute the LU factorization of a banded matrix AB. kl is the first subdiagonal containing a nonzero band, ku is the last superdiagonal containing one, and m is the first dimension of the matrix AB. Returns the LU factorization in-place and ipiv, the vector of pivots used.

source

Base.LinAlg.LAPACK.gbtrs! – Function.

```
gbtrs!(trans, kl, ku, m, AB, ipiv, B)
```

Solve the equation AB * X = B. trans determines the orientation of AB. It may be N (no transpose), T (transpose), or C (conjugate transpose). kl is the first subdiagonal containing a nonzero band, ku is the last superdiagonal containing one, and m is the first dimension of the matrix AB. ipiv is the vector of pivots returned from gbtrf!. Returns the vector or matrix X, overwriting B in-place.

source

Base.LinAlg.LAPACK.gebal! – Function.

```
gebal!(job, A) -> (ilo, ihi, scale)
```

Balance the matrix A before computing its eigensystem or Schur factorization. job can be one of N (A will not be permuted or scaled), P (A will only be permuted), S (A will only be scaled), or B (A will be both permuted and scaled). Modifies A in-place and returns ilo, ihi, and scale. If permuting was turned on, A[i,j] = 0 if j > i and 1 < j < ilo or j > ihi. scale contains information about the scaling/permutations performed.

source

Base.LinAlg.LAPACK.gebak! – Function.

```
gebak!(job, side, ilo, ihi, scale, V)
```

Transform the eigenvectors V of a matrix balanced using gebal! to the unscaled/unpermuted eigenvectors of the original matrix. Modifies V in-place. side can be L (left eigenvectors are transformed) or R (right eigenvectors are transformed).

source

Base.LinAlg.LAPACK.gebrd! – Function.

```
gebrd!(A) -> (A, d, e, tauq, taup)
```

Reduce A in-place to bidiagonal form A = QBP'. Returns A, containing the bidiagonal matrix B; d, containing the diagonal elements of B; e, containing the off-diagonal elements of B; tauq, containing the elementary reflectors representing Q; and taup, containing the elementary reflectors representing P.

source

Base.LinAlg.LAPACK.gelqf! – Function.

```
gelqf!(A, tau)
```

Compute the LQ factorization of A, A = LQ. tau contains scalars which parameterize the elementary reflectors of the factorization. tau must have length greater than or equal to the smallest dimension of A.

Returns A and tau modified in-place.

source

```
gelqf!(A) -> (A, tau)
```

Compute the LQ factorization of A, A = LQ.

Returns A, modified in-place, and tau, which contains scalars which parameterize the elementary reflectors of the factorization.

source

Base.LinAlg.LAPACK.geqlf! – Function.

```
geqlf!(A, tau)
```

Compute the QL factorization of A, A = QL. tau contains scalars which parameterize the elementary reflectors of the factorization. tau must have length greater than or equal to the smallest dimension of A.

Returns A and tau modified in-place.

source

```
geqlf!(A) -> (A, tau)
```

Compute the QL factorization of A, A = QL.

Returns A, modified in-place, and tau, which contains scalars which parameterize the elementary reflectors of the factorization.

source

Base.LinAlg.LAPACK.geqrf! – Function.

```
geqrf!(A, tau)
```

Compute the QR factorization of A, A = QR. tau contains scalars which parameterize the elementary reflectors of the factorization. tau must have length greater than or equal to the smallest dimension of A.

Returns A and tau modified in-place.

source

```
geqrf!(A) -> (A, tau)
```

Compute the QR factorization of A, A = QR.

Returns A, modified in-place, and tau, which contains scalars which parameterize the elementary reflectors of the factorization.

source

Base.LinAlg.LAPACK.geqp3! – Function.

```
geqp3!(A, jpvt, tau)
```

Compute the pivoted QR factorization of A, AP = QR using BLAS level 3. P is a pivoting matrix, represented by jpvt. tau stores the elementary reflectors. jpvt must have length length greater than or equal to n if A is an (m x n) matrix. tau must have length greater than or equal to the smallest dimension of A.

A, jpvt, and tau are modified in-place.

source

```
geqp3!(A, jpvt) -> (A, jpvt, tau)
```

Compute the pivoted QR factorization of A, AP = QR using BLAS level 3. P is a pivoting matrix, represented by jpvt. jpvt must have length greater than or equal to n if A is an (m x n) matrix.

Returns A and jpvt, modified in-place, and tau, which stores the elementary reflectors.

source

```
geqp3!(A) -> (A, jpvt, tau)
```

Compute the pivoted QR factorization of A, AP = QR using BLAS level 3.

Returns A, modified in-place, jpvt, which represents the pivoting matrix P, and tau, which stores the elementary reflectors.

source

Base.LinAlg.LAPACK.gerqf! – Function.

```
gerqf!(A, tau)
```

Compute the RQ factorization of A, A = RQ. tau contains scalars which parameterize the elementary reflectors of the factorization. tau must have length greater than or equal to the smallest dimension of A.

Returns A and tau modified in-place.

source

```
gerqf!(A) -> (A, tau)
```

Compute the RQ factorization of A, A = RQ.

Returns A, modified in-place, and tau, which contains scalars which parameterize the elementary reflectors of the factorization.

source

Base.LinAlg.LAPACK.geqrt! – Function.

```
geqrt!(A, T)
```

Compute the blocked QR factorization of A, A = QR. T contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization. The first dimension of T sets the block size and it must be between 1 and n. The second dimension of T must equal the smallest dimension of A.

Returns A and T modified in-place.

source

```
geqrt!(A, nb) -> (A, T)
```

Compute the blocked QR factorization of A, A = QR. nb sets the block size and it must be between 1 and n, the second dimension of A.

Returns A, modified in-place, and T, which contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization.

source

Base.LinAlg.LAPACK.geqrt3! – Function.

```
geqrt3!(A, T)
```

Recursively computes the blocked QR factorization of A, A = QR. T contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization. The first dimension of T sets the block size and it must be between 1 and n. The second dimension of T must equal the smallest dimension of A.

Returns A and T modified in-place.

source

```
geqrt3!(A) -> (A, T)
```

Recursively computes the blocked QR factorization of A, A = QR.

Returns A, modified in-place, and T, which contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization.

source

Base.LinAlg.LAPACK.getrf! – Function.

```
getrf!(A) -> (A, ipiv, info)
```

Compute the pivoted LU factorization of A, A = LU.

Returns A, modified in-place, ipiv, the pivoting information, and an info code which indicates success (info = 0), a singular value in U (info = i, in which case U[i,i] is singular), or an error code (info < 0).

source

Base.LinAlg.LAPACK.tzrzf! – Function.

```
tzrzf!(A) -> (A, tau)
```

Transforms the upper trapezoidal matrix A to upper triangular form in-place. Returns A and tau, the scalar parameters for the elementary reflectors of the transformation.

source

Base.LinAlg.LAPACK.ormrz! – Function.

```
ormrz!(side, trans, A, tau, C)
```

Multiplies the matrix C by Q from the transformation supplied by tzrzf!. Depending on side or trans the multiplication can be left-sided (side = L, Q*C) or right-sided (side = R, C*Q) and Q can be unmodified (trans = N), transposed (trans = T), or conjugate transposed (trans = C). Returns matrix C which is modified in-place with the result of the multiplication.

source

Base.LinAlg.LAPACK.gels! – Function.

```
gels!(trans, A, B) -> (F, B, ssr)
```

Solves the linear equation A * X = B, A.' * X =B, or A' * X = B using a QR or LQ factorization. Modifies the matrix/vector B in place with the solution. A is overwritten with its QR or LQ factorization. trans may be one of N (no modification), T (transpose), or C (conjugate transpose). gels! searches for the minimum norm/least squares solution. A may be under or over determined. The solution is returned in B.

source

Base.LinAlg.LAPACK.gesv! – Function.

```
gesv!(A, B) -> (B, A, ipiv)
```

Solves the linear equation `A * X = B` where `A` is a square matrix using the LU factorization of `A`. `A` is overwritten with its LU factorization and `B` is overwritten with the solution `X`. `ipiv` contains the pivoting information for the LU factorization of `A`.

source

`Base.LinAlg.LAPACK.getrs!` – Function.

```
getrs!(trans, A, ipiv, B)
```

Solves the linear equation `A * X = B`, `A.' * X =B`, or `A' * X = B` for square `A`. Modifies the matrix/vector `B` in place with the solution. `A` is the LU factorization from `getrf!`, with `ipiv` the pivoting information. `trans` may be one of `N` (no modification), `T` (transpose), or `C` (conjugate transpose).

source

`Base.LinAlg.LAPACK.getri!` – Function.

```
getri!(A, ipiv)
```

Computes the inverse of `A`, using its LU factorization found by `getrf!`. `ipiv` is the pivot information output and `A` contains the LU factorization of `getrf!`. `A` is overwritten with its inverse.

source

`Base.LinAlg.LAPACK.gesvx!` – Function.

```
gesvx!(fact, trans, A, AF, ipiv, equed, R, C, B) -> (X, equed, R, C, B, rcond, ferr, berr,
    work)
```

Solves the linear equation `A * X = B` (`trans = N`), `A.' * X =B` (`trans = T`), or `A' * X = B` (`trans = C`) using the LU factorization of `A`. `fact` may be `E`, in which case `A` will be equilibrated and copied to `AF`; `F`, in which case `AF` and `ipiv` from a previous LU factorization are inputs; or `N`, in which case `A` will be copied to `AF` and then factored. If `fact = F`, `equed` may be `N`, meaning `A` has not been equilibrated; `R`, meaning `A` was multiplied by `diagm(R)` from the left; `C`, meaning `A` was multiplied by `diagm(C)` from the right; or `B`, meaning `A` was multiplied by `diagm(R)` from the left and `diagm(C)` from the right. If `fact = F` and `equed = R` or `B` the elements of `R` must all be positive. If `fact = F` and `equed = C` or `B` the elements of `C` must all be positive.

Returns the solution `X`; `equed`, which is an output if `fact` is not `N`, and describes the equilibration that was performed; `R`, the row equilibration diagonal; `C`, the column equilibration diagonal; `B`, which may be overwritten with its equilibrated form `diagm(R)*B` (if `trans = N` and `equed = R,B`) or `diagm(C)*B` (if `trans = T,C` and `equed = C,B`); `rcond`, the reciprocal condition number of `A` after equilbrating; `ferr`, the forward error bound for each solution vector in `X`; `berr`, the forward error bound for each solution vector in `X`; and `work`, the reciprocal pivot growth factor.

source

```
gesvx!(A, B)
```

The no-equilibration, no-transpose simplification of `gesvx!`.

source

`Base.LinAlg.LAPACK.gelsd!` – Function.

```
gelsd!(A, B, rcond) -> (B, rnk)
```

Computes the least norm solution of `A * X = B` by finding the SVD factorization of `A`, then dividing-and-conquering the problem. `B` is overwritten with the solution `X`. Singular values below `rcond` will be treated as zero. Returns the solution in `B` and the effective rank of `A` in `rnk`.

source

`Base.LinAlg.LAPACK.gelsy!` – Function.

```
gelsy!(A, B, rcond) -> (B, rnk)
```

Computes the least norm solution of `A * X = B` by finding the full `QR` factorization of `A`, then dividing-and-conquering the problem. `B` is overwritten with the solution `X`. Singular values below `rcond` will be treated as zero. Returns the solution in `B` and the effective rank of `A` in `rnk`.

source

`Base.LinAlg.LAPACK.gglse!` – Function.

```
gglse!(A, c, B, d) -> (X,res)
```

Solves the equation `A * x = c` where `x` is subject to the equality constraint `B * x = d`. Uses the formula `||c - A*x||^2 = 0` to solve. Returns `X` and the residual sum-of-squares.

source

`Base.LinAlg.LAPACK.geev!` – Function.

```
geev!(jobvl, jobvr, A) -> (W, VL, VR)
```

Finds the eigensystem of `A`. If `jobvl = N`, the left eigenvectors of `A` aren't computed. If `jobvr = N`, the right eigenvectors of `A` aren't computed. If `jobvl = V` or `jobvr = V`, the corresponding eigenvectors are computed. Returns the eigenvalues in `W`, the right eigenvectors in `VR`, and the left eigenvectors in `VL`.

source

`Base.LinAlg.LAPACK.gesdd!` – Function.

```
gesdd!(job, A) -> (U, S, VT)
```

Finds the singular value decomposition of `A`, `A = U * S * V'`, using a divide and conquer approach. If `job = A`, all the columns of `U` and the rows of `V'` are computed. If `job = N`, no columns of `U` or rows of `V'` are computed. If `job = O`, `A` is overwritten with the columns of (thin) `U` and the rows of (thin) `V'`. If `job = S`, the columns of (thin) `U` and the rows of (thin) `V'` are computed and returned separately.

source

`Base.LinAlg.LAPACK.gesvd!` – Function.

```
gesvd!(jobu, jobvt, A) -> (U, S, VT)
```

Finds the singular value decomposition of `A`, `A = U * S * V'`. If `jobu = A`, all the columns of `U` are computed. If `jobvt = A` all the rows of `V'` are computed. If `jobu = N`, no columns of `U` are computed. If `jobvt = N` no rows of `V'` are computed. If `jobu = O`, `A` is overwritten with the columns of (thin) `U`. If `jobvt = O`, `A` is overwritten with the rows of (thin) `V'`. If `jobu = S`, the columns of (thin) `U` are computed and returned separately. If `jobvt = S` the rows of (thin) `V'` are computed and returned separately. `jobu` and `jobvt` can't both be `O`.

Returns `U`, `S`, and `Vt`, where `S` are the singular values of `A`.

source

Base.LinAlg.LAPACK.ggsvd! – Function.

```
ggsvd!(jobu, jobv, jobq, A, B) -> (U, V, Q, alpha, beta, k, l, R)
```

Finds the generalized singular value decomposition of A and B, U'*A*Q = D1*R and V'*B*Q = D2*R. D1 has alpha on its diagonal and D2 has beta on its diagonal. If jobu = U, the orthogonal/unitary matrix U is computed. If jobv = V the orthogonal/unitary matrix V is computed. If jobq = Q, the orthogonal/unitary matrix Q is computed. If jobu, jobv or jobq is N, that matrix is not computed. This function is only available in LAPACK versions prior to 3.6.0.

source

Base.LinAlg.LAPACK.ggsvd3! – Function.

```
ggsvd3!(jobu, jobv, jobq, A, B) -> (U, V, Q, alpha, beta, k, l, R)
```

Finds the generalized singular value decomposition of A and B, U'*A*Q = D1*R and V'*B*Q = D2*R. D1 has alpha on its diagonal and D2 has beta on its diagonal. If jobu = U, the orthogonal/unitary matrix U is computed. If jobv = V the orthogonal/unitary matrix V is computed. If jobq = Q, the orthogonal/unitary matrix Q is computed. If jobu, jobv, or jobq is N, that matrix is not computed. This function requires LAPACK 3.6.0.

source

Base.LinAlg.LAPACK.geevx! – Function.

```
geevx!(balanc, jobvl, jobvr, sense, A) -> (A, w, VL, VR, ilo, ihi, scale, abnrm, rconde,
    rcondv)
```

Finds the eigensystem of A with matrix balancing. If jobvl = N, the left eigenvectors of A aren't computed. If jobvr = N, the right eigenvectors of A aren't computed. If jobvl = V or jobvr = V, the corresponding eigenvectors are computed. If balanc = N, no balancing is performed. If balanc = P, A is permuted but not scaled. If balanc = S, A is scaled but not permuted. If balanc = B, A is permuted and scaled. If sense = N, no reciprocal condition numbers are computed. If sense = E, reciprocal condition numbers are computed for the eigenvalues only. If sense = V, reciprocal condition numbers are computed for the right eigenvectors only. If sense = B, reciprocal condition numbers are computed for the right eigenvectors and the eigenvectors. If sense = E,B, the right and left eigenvectors must be computed.

source

Base.LinAlg.LAPACK.ggev! – Function.

```
ggev!(jobvl, jobvr, A, B) -> (alpha, beta, vl, vr)
```

Finds the generalized eigendecomposition of A and B. If jobvl = N, the left eigenvectors aren't computed. If jobvr = N, the right eigenvectors aren't computed. If jobvl = V or jobvr = V, the corresponding eigenvectors are computed.

source

Base.LinAlg.LAPACK.gtsv! – Function.

```
gtsv!(dl, d, du, B)
```

Solves the equation A * X = B where A is a tridiagonal matrix with dl on the subdiagonal, d on the diagonal, and du on the superdiagonal.

Overwrites B with the solution X and returns it.

source

Base.LinAlg.LAPACK.gttrf! – Function.

```
gttrf!(dl, d, du) -> (dl, d, du, du2, ipiv)
```

Finds the LU factorization of a tridiagonal matrix with `dl` on the subdiagonal, `d` on the diagonal, and `du` on the superdiagonal.

Modifies `dl`, `d`, and `du` in-place and returns them and the second superdiagonal `du2` and the pivoting vector `ipiv`.

source

Base.LinAlg.LAPACK.gttrs! – Function.

```
gttrs!(trans, dl, d, du, du2, ipiv, B)
```

Solves the equation `A * X = B` (`trans = N`), `A.' * X = B` (`trans = T`), or `A' * X = B` (`trans = C`) using the LU factorization computed by `gttrf!`. `B` is overwritten with the solution `X`.

source

Base.LinAlg.LAPACK.orglq! – Function.

```
orglq!(A, tau, k = length(tau))
```

Explicitly finds the matrix `Q` of a `LQ` factorization after calling `gelqf!` on `A`. Uses the output of `gelqf!`. `A` is overwritten by `Q`.

source

Base.LinAlg.LAPACK.orgqr! – Function.

```
orgqr!(A, tau, k = length(tau))
```

Explicitly finds the matrix `Q` of a `QR` factorization after calling `geqrf!` on `A`. Uses the output of `geqrf!`. `A` is overwritten by `Q`.

source

Base.LinAlg.LAPACK.orgql! – Function.

```
orgql!(A, tau, k = length(tau))
```

Explicitly finds the matrix `Q` of a `QL` factorization after calling `geqlf!` on `A`. Uses the output of `geqlf!`. `A` is overwritten by `Q`.

source

Base.LinAlg.LAPACK.orgrq! – Function.

```
orgrq!(A, tau, k = length(tau))
```

Explicitly finds the matrix `Q` of a `RQ` factorization after calling `gerqf!` on `A`. Uses the output of `gerqf!`. `A` is overwritten by `Q`.

source

Base.LinAlg.LAPACK.ormlq! – Function.

```
ormlq!(side, trans, A, tau, C)
```

Computes `Q * C` (`trans = N`), `Q.' * C` (`trans = T`), `Q' * C` (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using `Q` from a `LQ` factorization of `A` computed using `gelqf!`. `C` is overwritten.

source

`Base.LinAlg.LAPACK.ormqr!` – Function.

```
ormqr!(side, trans, A, tau, C)
```

Computes `Q * C` (`trans = N`), `Q.' * C` (`trans = T`), `Q' * C` (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using `Q` from a `QR` factorization of `A` computed using `geqrf!`. `C` is overwritten.

source

`Base.LinAlg.LAPACK.ormql!` – Function.

```
ormql!(side, trans, A, tau, C)
```

Computes `Q * C` (`trans = N`), `Q.' * C` (`trans = T`), `Q' * C` (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using `Q` from a `QL` factorization of `A` computed using `geqlf!`. `C` is overwritten.

source

`Base.LinAlg.LAPACK.ormrq!` – Function.

```
ormrq!(side, trans, A, tau, C)
```

Computes `Q * C` (`trans = N`), `Q.' * C` (`trans = T`), `Q' * C` (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using `Q` from a `RQ` factorization of `A` computed using `gerqf!`. `C` is overwritten.

source

`Base.LinAlg.LAPACK.gemqrt!` – Function.

```
gemqrt!(side, trans, V, T, C)
```

Computes `Q * C` (`trans = N`), `Q.' * C` (`trans = T`), `Q' * C` (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using `Q` from a `QR` factorization of `A` computed using `geqrt!`. `C` is overwritten.

source

`Base.LinAlg.LAPACK.posv!` – Function.

```
posv!(uplo, A, B) -> (A, B)
```

Finds the solution to `A * X = B` where `A` is a symmetric or Hermitian positive definite matrix. If `uplo = U` the upper Cholesky decomposition of `A` is computed. If `uplo = L` the lower Cholesky decomposition of `A` is computed. `A` is overwritten by its Cholesky decomposition. `B` is overwritten with the solution `X`.

source

`Base.LinAlg.LAPACK.potrf!` – Function.

```
potrf!(uplo, A)
```

Computes the Cholesky (upper if `uplo = U`, lower if `uplo = L`) decomposition of positive-definite matrix `A`. `A` is overwritten and returned with an info code.

source

`Base.LinAlg.LAPACK.potri!` – Function.

```
potri!(uplo, A)
```

Computes the inverse of positive-definite matrix `A` after calling `potrf!` to find its (upper if `uplo = U`, lower if `uplo = L`) Cholesky decomposition.

`A` is overwritten by its inverse and returned.

source

`Base.LinAlg.LAPACK.potrs!` – Function.

```
potrs!(uplo, A, B)
```

Finds the solution to `A * X = B` where `A` is a symmetric or Hermitian positive definite matrix whose Cholesky decomposition was computed by `potrf!`. If `uplo = U` the upper Cholesky decomposition of `A` was computed. If `uplo = L` the lower Cholesky decomposition of `A` was computed. `B` is overwritten with the solution `X`.

source

`Base.LinAlg.LAPACK.pstrf!` – Function.

```
pstrf!(uplo, A, tol) -> (A, piv, rank, info)
```

Computes the (upper if `uplo = U`, lower if `uplo = L`) pivoted Cholesky decomposition of positive-definite matrix `A` with a user-set tolerance `tol`. `A` is overwritten by its Cholesky decomposition.

Returns `A`, the pivots `piv`, the rank of `A`, and an `info` code. If `info = 0`, the factorization succeeded. If `info = i > 0`, then `A` is indefinite or rank-deficient.

source

`Base.LinAlg.LAPACK.ptsv!` – Function.

```
ptsv!(D, E, B)
```

Solves `A * X = B` for positive-definite tridiagonal `A`. `D` is the diagonal of `A` and `E` is the off-diagonal. `B` is overwritten with the solution `X` and returned.

source

`Base.LinAlg.LAPACK.pttrf!` – Function.

```
pttrf!(D, E)
```

Computes the LDLt factorization of a positive-definite tridiagonal matrix with `D` as diagonal and `E` as off-diagonal. `D` and `E` are overwritten and returned.

source

`Base.LinAlg.LAPACK.pttrs!` – Function.

```
pttrs!(D, E, B)
```

Solves `A * X = B` for positive-definite tridiagonal `A` with diagonal `D` and off-diagonal `E` after computing `A`'s LDLt factorization using `pttrf!`. `B` is overwritten with the solution `X`.

source

Base.LinAlg.LAPACK.trtri! – Function.

```
trtri!(uplo, diag, A)
```

Finds the inverse of (upper if `uplo = U`, lower if `uplo = L`) triangular matrix `A`. If `diag = N`, `A` has non-unit diagonal elements. If `diag = U`, all diagonal elements of `A` are one. `A` is overwritten with its inverse.

source

Base.LinAlg.LAPACK.trtrs! – Function.

```
trtrs!(uplo, trans, diag, A, B)
```

Solves `A * X = B` (`trans = N`), `A.' * X = B` (`trans = T`), or `A' * X = B` (`trans = C`) for (upper if `uplo = U`, lower if `uplo = L`) triangular matrix `A`. If `diag = N`, `A` has non-unit diagonal elements. If `diag = U`, all diagonal elements of `A` are one. `B` is overwritten with the solution `X`.

source

Base.LinAlg.LAPACK.trcon! – Function.

```
trcon!(norm, uplo, diag, A)
```

Finds the reciprocal condition number of (upper if `uplo = U`, lower if `uplo = L`) triangular matrix `A`. If `diag = N`, `A` has non-unit diagonal elements. If `diag = U`, all diagonal elements of `A` are one. If `norm = I`, the condition number is found in the infinity norm. If `norm = O` or `1`, the condition number is found in the one norm.

source

Base.LinAlg.LAPACK.trevc! – Function.

```
trevc!(side, howmny, select, T, VL = similar(T), VR = similar(T))
```

Finds the eigensystem of an upper triangular matrix `T`. If `side = R`, the right eigenvectors are computed. If `side = L`, the left eigenvectors are computed. If `side = B`, both sets are computed. If `howmny = A`, all eigenvectors are found. If `howmny = B`, all eigenvectors are found and backtransformed using `VL` and `VR`. If `howmny = S`, only the eigenvectors corresponding to the values in `select` are computed.

source

Base.LinAlg.LAPACK.trrfs! – Function.

```
trrfs!(uplo, trans, diag, A, B, X, Ferr, Berr) -> (Ferr, Berr)
```

Estimates the error in the solution to `A * X = B` (`trans = N`), `A.' * X = B` (`trans = T`), `A' * X = B` (`trans = C`) for `side = L`, or the equivalent equations a right-handed `side = R` `X * A` after computing `X` using `trtrs!`. If `uplo = U`, `A` is upper triangular. If `uplo = L`, `A` is lower triangular. If `diag = N`, `A` has non-unit diagonal elements. If `diag = U`, all diagonal elements of `A` are one. `Ferr` and `Berr` are optional inputs. `Ferr` is the forward error and `Berr` is the backward error, each component-wise.

source

Base.LinAlg.LAPACK.stev! – Function.

```
stev!(job, dv, ev) -> (dv, Zmat)
```

Computes the eigensystem for a symmetric tridiagonal matrix with `dv` as diagonal and `ev` as off-diagonal. If `job = N` only the eigenvalues are found and returned in `dv`. If `job = V` then the eigenvectors are also found and returned in `Zmat`.

source

`Base.LinAlg.LAPACK.stebz!` – Function.

```
stebz!(range, order, vl, vu, il, iu, abstol, dv, ev) -> (dv, iblock, isplit)
```

Computes the eigenvalues for a symmetric tridiagonal matrix with `dv` as diagonal and `ev` as off-diagonal. If `range = A`, all the eigenvalues are found. If `range = V`, the eigenvalues in the half-open interval `(vl, vu]` are found. If `range = I`, the eigenvalues with indices between `il` and `iu` are found. If `order = B`, eigvalues are ordered within a block. If `order = E`, they are ordered across all the blocks. `abstol` can be set as a tolerance for convergence.

source

`Base.LinAlg.LAPACK.stegr!` – Function.

```
stegr!(jobz, range, dv, ev, vl, vu, il, iu) -> (w, Z)
```

Computes the eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) for a symmetric tridiagonal matrix with `dv` as diagonal and `ev` as off-diagonal. If `range = A`, all the eigenvalues are found. If `range = V`, the eigenvalues in the half-open interval `(vl, vu]` are found. If `range = I`, the eigenvalues with indices between `il` and `iu` are found. The eigenvalues are returned in `w` and the eigenvectors in `Z`.

source

`Base.LinAlg.LAPACK.stein!` – Function.

```
stein!(dv, ev_in, w_in, iblock_in, isplit_in)
```

Computes the eigenvectors for a symmetric tridiagonal matrix with `dv` as diagonal and `ev_in` as off-diagonal. `w_in` specifies the input eigenvalues for which to find corresponding eigenvectors. `iblock_in` specifies the submatrices corresponding to the eigenvalues in `w_in`. `isplit_in` specifies the splitting points between the submatrix blocks.

source

`Base.LinAlg.LAPACK.syconv!` – Function.

```
syconv!(uplo, A, ipiv) -> (A, work)
```

Converts a symmetric matrix `A` (which has been factorized into a triangular matrix) into two matrices `L` and `D`. If `uplo = U`, `A` is upper triangular. If `uplo = L`, it is lower triangular. `ipiv` is the pivot vector from the triangular factorization. `A` is overwritten by `L` and `D`.

source

`Base.LinAlg.LAPACK.sysv!` – Function.

```
sysv!(uplo, A, B) -> (B, A, ipiv)
```

Finds the solution to `A * X = B` for symmetric matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `B` is overwritten by the solution `X`. `A` is overwritten by its Bunch-Kaufman factorization. `ipiv` contains pivoting information about the factorization.

source

Base.LinAlg.LAPACK.sytrf! – Function.

```
sytrf!(uplo, A) -> (A, ipiv, info)
```

Computes the Bunch-Kaufman factorization of a symmetric matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored.

Returns `A`, overwritten by the factorization, a pivot vector `ipiv`, and the error code `info` which is a non-negative integer. If `info` is positive the matrix is singular and the diagonal part of the factorization is exactly zero at position `info`.

source

Base.LinAlg.LAPACK.sytri! – Function.

```
sytri!(uplo, A, ipiv)
```

Computes the inverse of a symmetric matrix `A` using the results of `sytrf!`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `A` is overwritten by its inverse.

source

Base.LinAlg.LAPACK.sytrs! – Function.

```
sytrs!(uplo, A, ipiv, B)
```

Solves the equation `A * X = B` for a symmetric matrix `A` using the results of `sytrf!`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `B` is overwritten by the solution `X`.

source

Base.LinAlg.LAPACK.hesv! – Function.

```
hesv!(uplo, A, B) -> (B, A, ipiv)
```

Finds the solution to `A * X = B` for Hermitian matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `B` is overwritten by the solution `X`. `A` is overwritten by its Bunch-Kaufman factorization. `ipiv` contains pivoting information about the factorization.

source

Base.LinAlg.LAPACK.hetrf! – Function.

```
hetrf!(uplo, A) -> (A, ipiv, info)
```

Computes the Bunch-Kaufman factorization of a Hermitian matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored.

Returns `A`, overwritten by the factorization, a pivot vector `ipiv`, and the error code `info` which is a non-negative integer. If `info` is positive the matrix is singular and the diagonal part of the factorization is exactly zero at position `info`.

source

Base.LinAlg.LAPACK.hetri! – Function.

```
hetri!(uplo, A, ipiv)
```

Computes the inverse of a Hermitian matrix `A` using the results of `sytrf!`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `A` is overwritten by its inverse.

source

Base.LinAlg.LAPACK.hetrs! – Function.

```
hetrs!(uplo, A, ipiv, B)
```

Solves the equation `A * X = B` for a Hermitian matrix `A` using the results of `sytrf!`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `B` is overwritten by the solution `X`.

source

Base.LinAlg.LAPACK.syev! – Function.

```
syev!(jobz, uplo, A)
```

Finds the eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) of a symmetric matrix `A`. If `uplo = U`, the upper triangle of `A` is used. If `uplo = L`, the lower triangle of `A` is used.

source

Base.LinAlg.LAPACK.syevr! – Function.

```
syevr!(jobz, range, uplo, A, vl, vu, il, iu, abstol) -> (W, Z)
```

Finds the eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) of a symmetric matrix `A`. If `uplo = U`, the upper triangle of `A` is used. If `uplo = L`, the lower triangle of `A` is used. If `range = A`, all the eigenvalues are found. If `range = V`, the eigenvalues in the half-open interval `(vl, vu]` are found. If `range = I`, the eigenvalues with indices between `il` and `iu` are found. `abstol` can be set as a tolerance for convergence.

The eigenvalues are returned in `W` and the eigenvectors in `Z`.

source

Base.LinAlg.LAPACK.sygvd! – Function.

```
sygvd!(itype, jobz, uplo, A, B) -> (w, A, B)
```

Finds the generalized eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) of a symmetric matrix `A` and symmetric positive-definite matrix `B`. If `uplo = U`, the upper triangles of `A` and `B` are used. If `uplo = L`, the lower triangles of `A` and `B` are used. If `itype = 1`, the problem to solve is `A * x = lambda * B * x`. If `itype = 2`, the problem to solve is `A * B * x = lambda * x`. If `itype = 3`, the problem to solve is `B * A * x = lambda * x`.

source

Base.LinAlg.LAPACK.bdsqr! – Function.

```
bdsqr!(uplo, d, e_, Vt, U, C) -> (d, Vt, U, C)
```

Computes the singular value decomposition of a bidiagonal matrix with `d` on the diagonal and `e_` on the off-diagonal. If `uplo = U`, `e_` is the superdiagonal. If `uplo = L`, `e_` is the subdiagonal. Can optionally also compute the product `Q' * C`.

Returns the singular values in `d`, and the matrix `C` overwritten with `Q' * C`.

source

Base.LinAlg.LAPACK.bdsdc! – Function.

```
bdsdc!(uplo, compq, d, e_) -> (d, e, u, vt, q, iq)
```

Computes the singular value decomposition of a bidiagonal matrix with `d` on the diagonal and `e_` on the off-diagonal using a divide and conqueq method. If `uplo = U`, `e_` is the superdiagonal. If `uplo = L`, `e_` is the subdiagonal. If `compq = N`, only the singular values are found. If `compq = I`, the singular values and vectors are found. If `compq = P`, the singular values and vectors are found in compact form. Only works for real types.

Returns the singular values in `d`, and if `compq = P`, the compact singular vectors in `iq`.

source

`Base.LinAlg.LAPACK.gecon!` – Function.

```
gecon!(normtype, A, anorm)
```

Finds the reciprocal condition number of matrix `A`. If `normtype = I`, the condition number is found in the infinity norm. If `normtype = 0` or `1`, the condition number is found in the one norm. `A` must be the result of `getrf!` and `anorm` is the norm of `A` in the relevant norm.

source

`Base.LinAlg.LAPACK.gehrd!` – Function.

```
gehrd!(ilo, ihi, A) -> (A, tau)
```

Converts a matrix `A` to Hessenberg form. If `A` is balanced with `gebal!` then `ilo` and `ihi` are the outputs of `gebal!`. Otherwise they should be `ilo = 1` and `ihi = size(A,2)`. `tau` contains the elementary reflectors of the factorization.

source

`Base.LinAlg.LAPACK.orghr!` – Function.

```
orghr!(ilo, ihi, A, tau)
```

Explicitly finds `Q`, the orthogonal/unitary matrix from `gehrd!`. `ilo`, `ihi`, `A`, and `tau` must correspond to the input/output to `gehrd!`.

source

`Base.LinAlg.LAPACK.gees!` – Function.

```
gees!(jobvs, A) -> (A, vs, w)
```

Computes the eigenvalues (`jobvs = N`) or the eigenvalues and Schur vectors (`jobvs = V`) of matrix `A`. `A` is overwritten by its Schur form.

Returns `A`, `vs` containing the Schur vectors, and `w`, containing the eigenvalues.

source

`Base.LinAlg.LAPACK.gges!` – Function.

```
gges!(jobvsl, jobvsr, A, B) -> (A, B, alpha, beta, vsl, vsr)
```

Computes the generalized eigenvalues, generalized Schur form, left Schur vectors (`jobsvl = V`), or right Schur vectors (`jobvsr = V`) of `A` and `B`.

The generalized eigenvalues are returned in `alpha` and `beta`. The left Schur vectors are returned in `vsl` and the right Schur vectors are returned in `vsr`.

source

Base.LinAlg.LAPACK.trexc! – Function.

```
trexc!(compq, ifst, ilst, T, Q) -> (T, Q)
```

Reorder the Schur factorization of a matrix. If `compq = V`, the Schur vectors `Q` are reordered. If `compq = N` they are not modified. `ifst` and `ilst` specify the reordering of the vectors.

source

Base.LinAlg.LAPACK.trsen! – Function.

```
trsen!(compq, job, select, T, Q) -> (T, Q, w)
```

Reorder the Schur factorization of a matrix and optionally finds reciprocal condition numbers. If `job = N`, no condition numbers are found. If `job = E`, only the condition number for this cluster of eigenvalues is found. If `job = V`, only the condition number for the invariant subspace is found. If `job = B` then the condition numbers for the cluster and subspace are found. If `compq = V` the Schur vectors `Q` are updated. If `compq = N` the Schur vectors are not modified. `select` determines which eigenvalues are in the cluster.

Returns `T`, `Q`, and reordered eigenvalues in `w`.

source

Base.LinAlg.LAPACK.tgsen! – Function.

```
tgsen!(select, S, T, Q, Z) -> (S, T, alpha, beta, Q, Z)
```

Reorders the vectors of a generalized Schur decomposition. `select` specifices the eigenvalues in each cluster.

source

Base.LinAlg.LAPACK.trsyl! – Function.

```
trsyl!(transa, transb, A, B, C, isgn=1) -> (C, scale)
```

Solves the Sylvester matrix equation `A * X +/- X * B = scale*C` where `A` and `B` are both quasi-upper triangular. If `transa = N`, `A` is not modified. If `transa = T`, `A` is transposed. If `transa = C`, `A` is conjugate transposed. Similarly for `transb` and `B`. If `isgn = 1`, the equation `A * X + X * B = scale * C` is solved. If `isgn = -1`, the equation `A * X - X * B = scale * C` is solved.

Returns `X` (overwriting `C`) and `scale`.

source

Chapter 53

Constantes

`Core.nothing` – Constant.

```
nothing
```

The singleton instance of type `Void`, used by convention when there is no value to return (as in a C `void` function). Can be converted to an empty `Nullable` value.

source

`Base.PROGRAM_FILE` – Constant.

```
PROGRAM_FILE
```

A string containing the script name passed to Julia from the command line. Note that the script name remains unchanged from within included files. Alternatively see `@__FILE__`.

source

`Base.ARGS` – Constant.

```
ARGS
```

An array of the command line arguments passed to Julia, as strings.

source

`Base.C_NULL` – Constant.

```
C_NULL
```

The C null pointer constant, sometimes used when calling external code.

source

`Base.VERSION` – Constant.

```
VERSION
```

A `VersionNumber` object describing which version of Julia is in use. For details see Version Number Literals.

source

`Base.LOAD_PATH` – Constant.

```
LOAD_PATH
```

An array of paths as strings or custom loader objects for the `require` function and `using` and `import` statements to consider when loading code. To create a custom loader type, define the type and then add appropriate methods to the `Base.load_hook` function with the following signature:

```
Base.load_hook(loader::Loader, name::String, found::Any)
```

The `loader` argument is the current value in `LOAD_PATH`, `name` is the name of the module to load, and `found` is the path of any previously found code to provide `name`. If no provider has been found earlier in `LOAD_PATH` then the value of `found` will be `nothing`. Custom loader functionality is experimental and may break or change in Julia 1.0.

source

`Base.JULIA_HOME` – Constant.

```
JULIA_HOME
```

A string containing the full path to the directory containing the `julia` executable.

source

`Core.ANY` – Constant.

```
ANY
```

Equivalent to `Any` for dispatch purposes, but signals the compiler to skip code generation specialization for that field.

source

`Base.Sys.CPU_CORES` – Constant.

```
Sys.CPU_CORES
```

The number of logical CPU cores available in the system.

See the Hwloc.jl package for extended information, including number of physical cores.

source

`Base.Sys.WORD_SIZE` – Constant.

```
Sys.WORD_SIZE
```

Standard word size on the current machine, in bits.

source

`Base.Sys.KERNEL` – Constant.

```
Sys.KERNEL
```

A symbol representing the name of the operating system, as returned by `uname` of the build configuration.

source

`Base.Sys.ARCH` – Constant.

```
Sys.ARCH
```

A symbol representing the architecture of the build configuration.

source

`Base.Sys.MACHINE` – Constant.

```
Sys.MACHINE
```

A string containing the build triple.

source

Ver también:

- `STDIN`
- `STDOUT`
- `STDERR`
- `ENV`
- `ENDIAN_BOM`
- `Libc.MS_ASYNC`
- `Libc.MS_INVALIDATE`
- `Libc.MS_SYNC`
- `Libdl.DL_LOAD_PATH`
- `Libdl.RTLD_DEEPBIND`
- `Libdl.RTLD_LOCAL`
- `Libdl.RTLD_NOLOAD`
- `Libdl.RTLD_LAZY`
- `Libdl.RTLD_NOW`
- `Libdl.RTLD_GLOBAL`
- `Libdl.RTLD_NODELETE`
- `Libdl.RTLD_FIRST`

Chapter 54

Sistema de Ficheros

`Base.Filesystem.pwd` – Function.

```
pwd() -> AbstractString
```

Get the current working directory.

source

`Base.Filesystem.cd` – Method.

```
cd(dir::AbstractString=homedir())
```

Set the current working directory.

source

`Base.Filesystem.cd` – Method.

```
cd(f::Function, dir::AbstractString=homedir())
```

Temporarily changes the current working directory and applies function `f` before returning.

source

`Base.Filesystem.readdir` – Function.

```
readdir(dir::AbstractString=".") -> Vector{String}
```

Returns the files and directories in the directory `dir` (or the current working directory if not given).

source

`Base.Filesystem.walkdir` – Function.

```
walkdir(dir; topdown=true, follow_symlinks=false, onerror=throw)
```

The `walkdir` method returns an iterator that walks the directory tree of a directory. The iterator returns a tuple containing `(rootpath, dirs, files)`. The directory tree can be traversed top-down or bottom-up. If `walkdir` encounters a `SystemError` it will rethrow the error by default. A custom error handling function can be provided through `onerror` keyword argument. `onerror` is called with a `SystemError` as argument.

```
for (root, dirs, files) in walkdir(".")
    println("Directories in $root")
    for dir in dirs
        println(joinpath(root, dir)) # path to directories
    end
    println("Files in $root")
    for file in files
        println(joinpath(root, file)) # path to files
    end
end
```

source

Base.Filesystem.mkdir – Function.

```
mkdir(path::AbstractString, mode::Unsigned=0o777)
```

Make a new directory with name `path` and permissions `mode`. `mode` defaults to `0o777`, modified by the current file creation mask. This function never creates more than one directory. If the directory already exists, or some intermediate directories do not exist, this function throws an error. See `mkpath` for a function which creates all required intermediate directories.

source

Base.Filesystem.mkpath – Function.

```
mkpath(path::AbstractString, mode::Unsigned=0o777)
```

Create all directories in the given `path`, with permissions `mode`. `mode` defaults to `0o777`, modified by the current file creation mask.

source

Base.Filesystem.symlink – Function.

```
symlink(target::AbstractString, link::AbstractString)
```

Creates a symbolic link to `target` with the name `link`.

> **Note**
>
> This function raises an error under operating systems that do not support soft symbolic links, such as Windows XP.

source

Base.Filesystem.readlink – Function.

```
readlink(path::AbstractString) -> AbstractString
```

Returns the target location a symbolic link `path` points to.

source

Base.Filesystem.chmod – Function.

```
chmod(path::AbstractString, mode::Integer; recursive::Bool=false)
```

Change the permissions mode of `path` to `mode`. Only integer `mode`s (e.g. `0o777`) are currently supported. If `recursive=true` and the path is a directory all permissions in that directory will be recursively changed.

source

`Base.Filesystem.chown` – Function.

```
chown(path::AbstractString, owner::Integer, group::Integer=-1)
```

Change the owner and/or group of `path` to `owner` and/or `group`. If the value entered for `owner` or `group` is `-1` the corresponding ID will not change. Only integer `owner`s and `group`s are currently supported.

source

`Base.stat` – Function.

```
stat(file)
```

Returns a structure whose fields contain information about the file. The fields of the structure are:

Name	Description
size	The size (in bytes) of the file
device	ID of the device that contains the file
inode	The inode number of the file
mode	The protection mode of the file
nlink	The number of hard links to the file
uid	The user id of the owner of the file
gid	The group id of the file owner
rdev	If this file refers to a device, the ID of the device it refers to
blksize	The file-system preferred block size for the file
blocks	The number of such blocks allocated
mtime	Unix timestamp of when the file was last modified
ctime	Unix timestamp of when the file was created

source

`Base.Filesystem.lstat` – Function.

```
lstat(file)
```

Like `stat`, but for symbolic links gets the info for the link itself rather than the file it refers to. This function must be called on a file path rather than a file object or a file descriptor.

source

`Base.Filesystem.ctime` – Function.

```
ctime(file)
```

Equivalent to `stat(file).ctime`

source

`Base.Filesystem.mtime` – Function.

```
mtime(file)
```

Equivalent to `stat(file).mtime`.

source

`Base.Filesystem.filemode` – Function.

```
filemode(file)
```

Equivalent to `stat(file).mode`

source

`Base.Filesystem.filesize` – Function.

```
filesize(path...)
```

Equivalent to `stat(file).size`.

source

`Base.Filesystem.uperm` – Function.

```
uperm(file)
```

Gets the permissions of the owner of the file as a bitfield of

Value	Description
01	Execute Permission
02	Write Permission
04	Read Permission

For allowed arguments, see `stat`.

source

`Base.Filesystem.gperm` – Function.

```
gperm(file)
```

Like `uperm` but gets the permissions of the group owning the file.

source

`Base.Filesystem.operm` – Function.

```
operm(file)
```

Like `uperm` but gets the permissions for people who neither own the file nor are a member of the group owning the file

source

`Base.Filesystem.cp` – Function.

```
cp(src::AbstractString, dst::AbstractString; remove_destination::Bool=false, follow_symlinks
    ::Bool=false)
```

Copy the file, link, or directory from `src` to `dest`. `remove_destination=true` will first remove an existing `dst`.

If `follow_symlinks=false`, and `src` is a symbolic link, `dst` will be created as a symbolic link. If `follow_symlinks=true` and `src` is a symbolic link, `dst` will be a copy of the file or directory `src` refers to.

source

`Base.download` – Function.

```
download(url::AbstractString, [localfile::AbstractString])
```

Download a file from the given url, optionally renaming it to the given local file name. Note that this function relies on the availability of external tools such as `curl`, `wget` or `fetch` to download the file and is provided for convenience. For production use or situations in which more options are needed, please use a package that provides the desired functionality instead.

source

`Base.Filesystem.mv` – Function.

```
mv(src::AbstractString, dst::AbstractString; remove_destination::Bool=false)
```

Move the file, link, or directory from `src` to `dst`. `remove_destination=true` will first remove an existing `dst`.

source

`Base.Filesystem.rm` – Function.

```
rm(path::AbstractString; force::Bool=false, recursive::Bool=false)
```

Delete the file, link, or empty directory at the given path. If `force=true` is passed, a non-existing path is not treated as error. If `recursive=true` is passed and the path is a directory, then all contents are removed recursively.

source

`Base.Filesystem.touch` – Function.

```
touch(path::AbstractString)
```

Update the last-modified timestamp on a file to the current time.

source

`Base.Filesystem.tempname` – Function.

```
tempname()
```

Generate a unique temporary file path.

source

`Base.Filesystem.tempdir` – Function.

```
tempdir()
```

Obtain the path of a temporary directory (possibly shared with other processes).

source

`Base.Filesystem.mktemp` – Method.

```
mktemp(parent=tempdir())
```

Returns `(path, io)`, where `path` is the path of a new temporary file in `parent` and `io` is an open file object for this path.

source

Base.Filesystem.mktemp – Method.

```
mktemp(f::Function, parent=tempdir())
```

Apply the function f to the result of mktemp(parent) and remove the temporary file upon completion.

source

Base.Filesystem.mktempdir – Method.

```
mktempdir(parent=tempdir())
```

Create a temporary directory in the parent directory and return its path. If parent does not exist, throw an error.

source

Base.Filesystem.mktempdir – Method.

```
mktempdir(f::Function, parent=tempdir())
```

Apply the function f to the result of mktempdir(parent) and remove the temporary directory upon completion.

source

Base.Filesystem.isblockdev – Function.

```
isblockdev(path) -> Bool
```

Returns true if path is a block device, false otherwise.

source

Base.Filesystem.ischardev – Function.

```
ischardev(path) -> Bool
```

Returns true if path is a character device, false otherwise.

source

Base.Filesystem.isdir – Function.

```
isdir(path) -> Bool
```

Returns true if path is a directory, false otherwise.

source

Base.Filesystem.isfifo – Function.

```
isfifo(path) -> Bool
```

Returns true if path is a FIFO, false otherwise.

source

Base.Filesystem.isfile – Function.

```
isfile(path) -> Bool
```

Returns `true` if `path` is a regular file, `false` otherwise.

source

`Base.Filesystem.islink` – Function.

```
islink(path) -> Bool
```

Returns `true` if `path` is a symbolic link, `false` otherwise.

source

`Base.Filesystem.ismount` – Function.

```
ismount(path) -> Bool
```

Returns `true` if `path` is a mount point, `false` otherwise.

source

`Base.Filesystem.ispath` – Function.

```
ispath(path) -> Bool
```

Returns `true` if `path` is a valid filesystem path, `false` otherwise.

source

`Base.Filesystem.issetgid` – Function.

```
issetgid(path) -> Bool
```

Returns `true` if `path` has the setgid flag set, `false` otherwise.

source

`Base.Filesystem.issetuid` – Function.

```
issetuid(path) -> Bool
```

Returns `true` if `path` has the setuid flag set, `false` otherwise.

source

`Base.Filesystem.issocket` – Function.

```
issocket(path) -> Bool
```

Returns `true` if `path` is a socket, `false` otherwise.

source

`Base.Filesystem.issticky` – Function.

```
issticky(path) -> Bool
```

Returns `true` if `path` has the sticky bit set, `false` otherwise.

source

`Base.Filesystem.homedir` – Function.

```
homedir() -> AbstractString
```

Return the current user's home directory.

> **Note**
>
> homedir determines the home directory via libuv's uv_os_homedir. For details (for example on how to specify the home directory via environment variables), see the uv_os_homedir documentation.

source

Base.Filesystem.dirname – Function.

```
dirname(path::AbstractString) -> AbstractString
```

Get the directory part of a path.

```
julia> dirname("/home/myuser")
"/home"
```

See also: basename

source

Base.Filesystem.basename – Function.

```
basename(path::AbstractString) -> AbstractString
```

Get the file name part of a path.

```
julia> basename("/home/myuser/example.jl")
"example.jl"
```

See also: dirname

source

Base.@__FILE__ – Macro.

```
@__FILE__ -> AbstractString
```

@__FILE__ expands to a string with the absolute file path of the file containing the macro. Returns nothing if run from a REPL or an empty string if evaluated by julia -e <expr>. Alternatively see PROGRAM_FILE.

source

Base.@__DIR__ – Macro.

```
@__DIR__ -> AbstractString
```

@__DIR__ expands to a string with the directory part of the absolute path of the file containing the macro. Returns nothing if run from a REPL or an empty string if evaluated by julia -e <expr>.

source

@__LINE__ – Macro.

```
@__LINE__ -> Int
```

`@__LINE__` expands to the line number of the call-site.

source

`Base.Filesystem.isabspath` – Function.

```
isabspath(path::AbstractString) -> Bool
```

Determines whether a path is absolute (begins at the root directory).

```
julia> isabspath("/home")
true

julia> isabspath("home")
false
```

source

`Base.Filesystem.isdirpath` – Function.

```
isdirpath(path::AbstractString) -> Bool
```

Determines whether a path refers to a directory (for example, ends with a path separator).

```
julia> isdirpath("/home")
false

julia> isdirpath("/home/")
true
```

source

`Base.Filesystem.joinpath` – Function.

```
joinpath(parts...) -> AbstractString
```

Join path components into a full path. If some argument is an absolute path, then prior components are dropped.

```
julia> joinpath("/home/myuser","example.jl")
"/home/myuser/example.jl"
```

source

`Base.Filesystem.abspath` – Function.

```
abspath(path::AbstractString) -> AbstractString
```

Convert a path to an absolute path by adding the current directory if necessary.

source

```
abspath(path::AbstractString, paths::AbstractString...) -> AbstractString
```

Convert a set of paths to an absolute path by joining them together and adding the current directory if necessary. Equivalent to `abspath(joinpath(path, paths...))`.

source

Base.Filesystem.normpath – Function.

```
normpath(path::AbstractString) -> AbstractString
```

Normalize a path, removing "." and ".." entries.

```
julia> normpath("/home/myuser/../example.jl")
"/home/example.jl"
```

source

Base.Filesystem.realpath – Function.

```
realpath(path::AbstractString) -> AbstractString
```

Canonicalize a path by expanding symbolic links and removing "." and ".." entries.

source

Base.Filesystem.relpath – Function.

```
relpath(path::AbstractString, startpath::AbstractString = ".") -> AbstractString
```

Return a relative filepath to `path` either from the current directory or from an optional start directory. This is a path computation: the filesystem is not accessed to confirm the existence or nature of `path` or `startpath`.

source

Base.Filesystem.expanduser – Function.

```
expanduser(path::AbstractString) -> AbstractString
```

On Unix systems, replace a tilde character at the start of a path with the current user's home directory.

source

Base.Filesystem.splitdir – Function.

```
splitdir(path::AbstractString) -> (AbstractString, AbstractString)
```

Split a path into a tuple of the directory name and file name.

```
julia> splitdir("/home/myuser")
("/home", "myuser")
```

source

Base.Filesystem.splitdrive – Function.

```
splitdrive(path::AbstractString) -> (AbstractString, AbstractString)
```

On Windows, split a path into the drive letter part and the path part. On Unix systems, the first component is always the empty string.

source

Base.Filesystem.splitext – Function.

```
splitext(path::AbstractString) -> (AbstractString, AbstractString)
```

If the last component of a path contains a dot, split the path into everything before the dot and everything including and after the dot. Otherwise, return a tuple of the argument unmodified and the empty string.

```
julia> splitext("/home/myuser/example.jl")
("/home/myuser/example", ".jl")

julia> splitext("/home/myuser/example")
("/home/myuser/example", "")
```

source

Chapter 55

E/S y Redes

55.1 E/S General

`Base.STDOUT` – Constant.

```
STDOUT
```

Global variable referring to the standard out stream.

source

`Base.STDERR` – Constant.

```
STDERR
```

Global variable referring to the standard error stream.

source

`Base.STDIN` – Constant.

```
STDIN
```

Global variable referring to the standard input stream.

source

`Base.open` – Function.

```
open(filename::AbstractString, [read::Bool, write::Bool, create::Bool, truncate::Bool, append
    ::Bool]) -> IOStream
```

Open a file in a mode specified by five boolean arguments. The default is to open files for reading only. Returns a stream for accessing the file.

source

```
open(filename::AbstractString, [mode::AbstractString]) -> IOStream
```

Alternate syntax for open, where a string-based mode specifier is used instead of the five booleans. The values of `mode` correspond to those from `fopen(3)` or Perl `open`, and are equivalent to setting the following boolean groups:

source

Mode	Description
r	read
r+	read, write
w	write, create, truncate
w+	read, write, create, truncate
a	write, create, append
a+	read, write, create, append

```
open(f::Function, args...)
```

Apply the function `f` to the result of `open(args...)` and close the resulting file descriptor upon completion.

Example: `open(readstring, "file.txt")`

source

```
open(command, mode::AbstractString="r", stdio=DevNull)
```

Start running `command` asynchronously, and return a tuple `(stream,process)`. If `mode` is `"r"`, then `stream` reads from the process's standard output and `stdio` optionally specifies the process's standard input stream. If `mode` is `"w"`, then `stream` writes to the process's standard input and `stdio` optionally specifies the process's standard output stream.

source

```
open(f::Function, command, mode::AbstractString="r", stdio=DevNull)
```

Similar to `open(command, mode, stdio)`, but calls `f(stream)` on the resulting read or write stream, then closes the stream and waits for the process to complete. Returns the value returned by `f`.

source

`Base.IOBuffer` – Type.

```
IOBuffer([data,],[readable::Bool=true, writable::Bool=true, [maxsize::Int=typemax(Int)]])
```

Create an `IOBuffer`, which may optionally operate on a pre-existing array. If the readable/writable arguments are given, they restrict whether or not the buffer may be read from or written to respectively. The last argument optionally specifies a size beyond which the buffer may not be grown.

source

```
IOBuffer() -> IOBuffer
```

Create an in-memory I/O stream.

source

```
IOBuffer(size::Int)
```

Create a fixed size IOBuffer. The buffer will not grow dynamically.

source

```
IOBuffer(string::String)
```

Create a read-only `IOBuffer` on the data underlying the given string.

```
julia> io = IOBuffer("Haho");

julia> String(take!(io))
"Haho"

julia> String(take!(io))
"Haho"
```

source

Base.take! – Method.

```
take!(b::IOBuffer)
```

Obtain the contents of an IOBuffer as an array, without copying. Afterwards, the IOBuffer is reset to its initial state.

source

Base.fdio – Function.

```
fdio([name::AbstractString, ]fd::Integer[, own::Bool=false]) -> IOStream
```

Create an IOStream object from an integer file descriptor. If own is true, closing this object will close the underlying descriptor. By default, an IOStream is closed when it is garbage collected. name allows you to associate the descriptor with a named file.

source

Base.flush – Function.

```
flush(stream)
```

Commit all currently buffered writes to the given stream.

source

Base.close – Function.

```
close(stream)
```

Close an I/O stream. Performs a flush first.

source

Base.write – Function.

```
write(stream::IO, x)
write(filename::AbstractString, x)
```

Write the canonical binary representation of a value to the given I/O stream or file. Returns the number of bytes written into the stream.

You can write multiple values with the same write call. i.e. the following are equivalent:

```
write(stream, x, y...)
write(stream, x) + write(stream, y...)
```

source

`Base.read` – Function.

```
read(filename::AbstractString, args...)
```

Open a file and read its contents. `args` is passed to `read`: this is equivalent to `open(io->read(io, args...), filename)`.

source

```
read(stream::IO, T, dims)
```

Read a series of values of type `T` from `stream`, in canonical binary representation. `dims` is either a tuple or a series of integer arguments specifying the size of the `Array{T}` to return.

source

```
read(s::IO, nb=typemax(Int))
```

Read at most nb bytes from s, returning a `Vector{UInt8}` of the bytes read.

source

```
read(s::IOStream, nb::Integer; all=true)
```

Read at most nb bytes from s, returning a `Vector{UInt8}` of the bytes read.

If `all` is `true` (the default), this function will block repeatedly trying to read all requested bytes, until an error or end-of-file occurs. If `all` is `false`, at most one `read` call is performed, and the amount of data returned is device-dependent. Note that not all stream types support the `all` option.

source

```
read(stream::IO, T)
```

Read a single value of type `T` from `stream`, in canonical binary representation.

source

`Base.read!` – Function.

```
read!(stream::IO, array::Union{Array, BitArray})
read!(filename::AbstractString, array::Union{Array, BitArray})
```

Read binary data from an I/O stream or file, filling in `array`.

source

`Base.readbytes!` – Function.

```
readbytes!(stream::IO, b::AbstractVector{UInt8}, nb=length(b))
```

Read at most nb bytes from `stream` into b, returning the number of bytes read. The size of b will be increased if needed (i.e. if nb is greater than `length(b)` and enough bytes could be read), but it will never be decreased.

source

```
readbytes!(stream::IOStream, b::AbstractVector{UInt8}, nb=length(b); all::Bool=true)
```

Read at most nb bytes from `stream` into b, returning the number of bytes read. The size of b will be increased if needed (i.e. if nb is greater than `length(b)` and enough bytes could be read), but it will never be decreased.

See `read` for a description of the `all` option.

source

Base.unsafe_read – Function.

```
unsafe_read(io::IO, ref, nbytes::UInt)
```

Copy nbytes from the IO stream object into ref (converted to a pointer).

It is recommended that subtypes T<:IO override the following method signature to provide more efficient implementations: unsafe_read(s::T, p::Ptr{UInt8}, n::UInt)

source

Base.unsafe_write – Function.

```
unsafe_write(io::IO, ref, nbytes::UInt)
```

Copy nbytes from ref (converted to a pointer) into the IO object.

It is recommended that subtypes T<:IO override the following method signature to provide more efficient implementations: unsafe_write(s::T, p::Ptr{UInt8}, n::UInt)

source

Base.position – Function.

```
position(s)
```

Get the current position of a stream.

source

Base.seek – Function.

```
seek(s, pos)
```

Seek a stream to the given position.

source

Base.seekstart – Function.

```
seekstart(s)
```

Seek a stream to its beginning.

source

Base.seekend – Function.

```
seekend(s)
```

Seek a stream to its end.

source

Base.skip – Function.

```
skip(s, offset)
```

Seek a stream relative to the current position.

source

`Base.mark` – Function.

```
mark(s)
```

Add a mark at the current position of stream `s`. Returns the marked position.

See also `unmark`, `reset`, `ismarked`.

source

`Base.unmark` – Function.

```
unmark(s)
```

Remove a mark from stream `s`. Returns `true` if the stream was marked, `false` otherwise.

See also `mark`, `reset`, `ismarked`.

source

`Base.reset` – Function.

```
reset(s)
```

Reset a stream `s` to a previously marked position, and remove the mark. Returns the previously marked position. Throws an error if the stream is not marked.

See also `mark`, `unmark`, `ismarked`.

source

`Base.ismarked` – Function.

```
ismarked(s)
```

Returns `true` if stream `s` is marked.

See also `mark`, `unmark`, `reset`.

source

`Base.eof` – Function.

```
eof(stream) -> Bool
```

Tests whether an I/O stream is at end-of-file. If the stream is not yet exhausted, this function will block to wait for more data if necessary, and then return `false`. Therefore it is always safe to read one byte after seeing `eof` return `false`. `eof` will return `false` as long as buffered data is still available, even if the remote end of a connection is closed.

source

`Base.isreadonly` – Function.

```
isreadonly(stream) -> Bool
```

Determine whether a stream is read-only.

source

`Base.iswritable` – Function.

```
iswritable(io) -> Bool
```

Returns `true` if the specified IO object is writable (if that can be determined).

source

`Base.isreadable` – Function.

```
isreadable(io) -> Bool
```

Returns `true` if the specified IO object is readable (if that can be determined).

source

`Base.isopen` – Function.

```
isopen(object) -> Bool
```

Determine whether an object - such as a stream, timer, or mmap – is not yet closed. Once an object is closed, it will never produce a new event. However, a closed stream may still have data to read in its buffer, use `eof` to check for the ability to read data. Use `poll_fd` to be notified when a stream might be writable or readable.

source

`Base.Serializer.serialize` – Function.

```
serialize(stream, value)
```

Write an arbitrary value to a stream in an opaque format, such that it can be read back by `deserialize`. The read-back value will be as identical as possible to the original. In general, this process will not work if the reading and writing are done by different versions of Julia, or an instance of Julia with a different system image. `Ptr` values are serialized as all-zero bit patterns (`NULL`).

source

`Base.Serializer.deserialize` – Function.

```
deserialize(stream)
```

Read a value written by `serialize`. `deserialize` assumes the binary data read from `stream` is correct and has been serialized by a compatible implementation of `serialize`. It has been designed with simplicity and performance as a goal and does not validate the data read. Malformed data can result in process termination. The caller has to ensure the integrity and correctness of data read from `stream`.

source

`Base.Grisu.print_shortest` – Function.

```
print_shortest(io, x)
```

Print the shortest possible representation, with the minimum number of consecutive non-zero digits, of number `x`, ensuring that it would parse to the exact same number.

source

`Base.fd` – Function.

```
fd(stream)
```

Returns the file descriptor backing the stream or file. Note that this function only applies to synchronous `File`'s and `IOStream`'s not to any of the asynchronous streams.

source

`Base.redirect_stdout` – Function.

```
redirect_stdout([stream]) -> (rd, wr)
```

Create a pipe to which all C and Julia level `STDOUT` output will be redirected. Returns a tuple `(rd, wr)` representing the pipe ends. Data written to `STDOUT` may now be read from the `rd` end of the pipe. The `wr` end is given for convenience in case the old `STDOUT` object was cached by the user and needs to be replaced elsewhere.

> **Note**
> `stream` must be a `TTY`, a `Pipe`, or a `TCPSocket`.

source

`Base.redirect_stdout` – Method.

```
redirect_stdout(f::Function, stream)
```

Run the function `f` while redirecting `STDOUT` to `stream`. Upon completion, `STDOUT` is restored to its prior setting.

> **Note**
> `stream` must be a `TTY`, a `Pipe`, or a `TCPSocket`.

source

`Base.redirect_stderr` – Function.

```
redirect_stderr([stream]) -> (rd, wr)
```

Like `redirect_stdout`, but for `STDERR`.

> **Note**
> `stream` must be a `TTY`, a `Pipe`, or a `TCPSocket`.

source

`Base.redirect_stderr` – Method.

```
redirect_stderr(f::Function, stream)
```

Run the function `f` while redirecting `STDERR` to `stream`. Upon completion, `STDERR` is restored to its prior setting.

> **Note**
> `stream` must be a `TTY`, a `Pipe`, or a `TCPSocket`.

source

`Base.redirect_stdin` – Function.

```
redirect_stdin([stream]) -> (rd, wr)
```

Like `redirect_stdout`, but for `STDIN`. Note that the order of the return tuple is still `(rd, wr)`, i.e. data to be read from `STDIN` may be written to `wr`.

> **Note**
> `stream` must be a `TTY`, a `Pipe`, or a `TCPSocket`.

source

`Base.redirect_stdin` – Method.

```
redirect_stdin(f::Function, stream)
```

Run the function `f` while redirecting `STDIN` to `stream`. Upon completion, `STDIN` is restored to its prior setting.

> **Note**
> `stream` must be a `TTY`, a `Pipe`, or a `TCPSocket`.

source

`Base.readchomp` – Function.

```
readchomp(x)
```

Read the entirety of `x` as a string and remove a single trailing newline. Equivalent to `chomp!(readstring(x))`.

source

`Base.truncate` – Function.

```
truncate(file,n)
```

Resize the file or buffer given by the first argument to exactly `n` bytes, filling previously unallocated space with '\0' if the file or buffer is grown.

source

`Base.skipchars` – Function.

```
skipchars(stream, predicate; linecomment::Char)
```

Advance the stream until before the first character for which `predicate` returns `false`. For example `skipchars(stream, isspace)` will skip all whitespace. If keyword argument `linecomment` is specified, characters from that character through the end of a line will also be skipped.

source

`Base.DataFmt.countlines` – Function.

```
countlines(io::IO, eol::Char='\n')
```

Read `io` until the end of the stream/file and count the number of lines. To specify a file pass the filename as the first argument. EOL markers other than `'\n'` are supported by passing them as the second argument.

source

`Base.PipeBuffer` – Function.

```
PipeBuffer(data::Vector{UInt8}=UInt8[],[maxsize::Int=typemax(Int)])
```

An IOBuffer that allows reading and performs writes by appending. Seeking and truncating are not supported. See IOBuffer for the available constructors. If data is given, creates a PipeBuffer to operate on a data vector, optionally specifying a size beyond which the underlying Array may not be grown.

source

Base.readavailable – Function.

```
readavailable(stream)
```

Read all available data on the stream, blocking the task only if no data is available. The result is a Vector{UInt8,1}.

source

Base.IOContext – Type.

```
IOContext
```

IOContext provides a mechanism for passing output configuration settings among show methods.

In short, it is an immutable dictionary that is a subclass of IO. It supports standard dictionary operations such as getindex, and can also be used as an I/O stream.

source

Base.IOContext – Method.

```
IOContext(io::IO, KV::Pair)
```

Create an IOContext that wraps a given stream, adding the specified key=>value pair to the properties of that stream (note that io can itself be an IOContext).

- use (key => value) in dict to see if this particular combination is in the properties set
- use get(dict, key, default) to retrieve the most recent value for a particular key

The following properties are in common use:

- :compact: Boolean specifying that small values should be printed more compactly, e.g. that numbers should be printed with fewer digits. This is set when printing array elements.
- :limit: Boolean specifying that containers should be truncated, e.g. showing … in place of most elements.
- :displaysize: A Tuple{Int,Int} giving the size in rows and columns to use for text output. This can be used to override the display size for called functions, but to get the size of the screen use the displaysize function.

```
julia> function f(io::IO)
           if get(io, :short, false)
               print(io, "short")
           else
               print(io, "looooong")
           end
```

```
        end
f (generic function with 1 method)

julia> f(STDOUT)
looooooong
julia> f(IOContext(STDOUT, :short => true))
short
```

source

Base.IOContext – Method.

```
IOContext(io::IO, context::IOContext)
```

Create an IOContext that wraps an alternate IO but inherits the properties of context.

source

55.2 E/S Texto

Base.show – Method.

```
show(x)
```

Write an informative text representation of a value to the current output stream. New types should overload show(io, x) where the first argument is a stream. The representation used by show generally includes Julia-specific formatting and type information.

source

Base.showcompact – Function.

```
showcompact(x)
```

Show a compact representation of a value.

This is used for printing array elements without repeating type information (which would be redundant with that printed once for the whole array), and without line breaks inside the representation of an element.

To offer a compact representation different from its standard one, a custom type should test get(io, :compact, false) in its normal show method.

source

Base.showall – Function.

```
showall(x)
```

Similar to show, except shows all elements of arrays.

source

Base.summary – Function.

```
summary(x)
```

Return a string giving a brief description of a value. By default returns string(typeof(x)), e.g. Int64.

For arrays, returns a string of size and type info, e.g. 10-element Array{Int64,1}.

```
julia> summary(1)
"Int64"

julia> summary(zeros(2))
"2-element Array{Float64,1}"
```

source

Base.print – Function.

```
print(io::IO, x)
```

Write to io (or to the default output stream STDOUT if io is not given) a canonical (un-decorated) text representation of a value if there is one, otherwise call show. The representation used by print includes minimal formatting and tries to avoid Julia-specific details.

```
julia> print("Hello World!")
Hello World!
julia> io = IOBuffer();

julia> print(io, "Hello World!")

julia> String(take!(io))
"Hello World!"
```

source

Base.println – Function.

```
println(io::IO, xs...)
```

Print (using print) xs followed by a newline. If io is not supplied, prints to STDOUT.

source

Base.print_with_color – Function.

```
print_with_color(color::Union{Symbol, Int}, [io], xs...; bold::Bool = false)
```

Print xs in a color specified as a symbol.

color may take any of the values :normal, :default, :bold, :black, :blue, :cyan, :green, :light_black, :light_blue, :light_cyan, :light_green, :light_magenta, :light_red, :light_yellow, :magenta, :nothing, :red, :white, or :yellow or an integer between 0 and 255 inclusive. Note that not all terminals support 256 colors. If the keyword bold is given as true, the result will be printed in bold.

source

Base.info – Function.

```
info([io, ] msg..., [prefix="INFO: "])
```

Display an informational message. Argument msg is a string describing the information to be displayed. The prefix keyword argument can be used to override the default prepending of msg.

```
julia> info("hello world")
INFO: hello world

julia> info("hello world"; prefix="MY INFO: ")
MY INFO: hello world
```

See also logging.

source

Base.warn – Function.

```
warn([io, ] msg..., [prefix="WARNING: ", once=false, key=nothing, bt=nothing, filename=
    nothing, lineno::Int=0])
```

Display a warning. Argument msg is a string describing the warning to be displayed. Set once to true and specify a key to only display msg the first time warn is called. If bt is not nothing a backtrace is displayed. If filename is not nothing both it and lineno are displayed.

See also logging.

source

```
warn(msg)
```

Display a warning. Argument msg is a string describing the warning to be displayed.

```
julia> warn("Beep Beep")
WARNING: Beep Beep
```

source

Base.logging – Function.

```
logging(io [, m [, f]][; kind=:all])
logging([; kind=:all])
```

Stream output of informational, warning, and/or error messages to io, overriding what was otherwise specified. Optionally, divert stream only for module m, or specifically function f within m. kind can be :all (the default), :info, :warn, or :error. See Base.log_{info,warn,error}_to for the current set of redirections. Call logging with no arguments (or just the kind) to reset everything.

source

Base.Printf.@printf – Macro.

```
@printf([io::IOStream], "%Fmt", args...)
```

Print args using C printf() style format specification string, with some caveats: Inf and NaN are printed consistently as Inf and NaN for flags %a, %A, %e, %E, %f, %F, %g, and %G. Furthermore, if a floating point number is equally close to the numeric values of two possible output strings, the output string further away from zero is chosen.

Optionally, an IOStream may be passed as the first argument to redirect output.

Examples

```
julia> @printf("%f %F %f %F\n", Inf, Inf, NaN, NaN)
Inf Inf NaN NaN


julia> @printf "%.0f %.1f %f\n" 0.5 0.025 -0.0078125
1 0.0 -0.007813
```

source

Base.Printf.@sprintf – Macro.

```
@sprintf("%Fmt", args...)
```

Return `@printf` formatted output as string.

Examples

```
julia> s = @sprintf "this is a %s %15.1f" "test" 34.567;

julia> println(s)
this is a test            34.6
```

source

Base.sprint – Function.

```
sprint(f::Function, args...)
```

Call the given function with an I/O stream and the supplied extra arguments. Everything written to this I/O stream is returned as a string.

```
julia> sprint(showcompact, 66.66666)
"66.6667"
```

source

Base.showerror – Function.

```
showerror(io, e)
```

Show a descriptive representation of an exception object.

source

Base.dump – Function.

```
dump(x)
```

Show every part of the representation of a value.

source

Base.readstring – Function.

```
readstring(stream::IO)
readstring(filename::AbstractString)
```

Read the entire contents of an I/O stream or a file as a string. The text is assumed to be encoded in UTF-8.

source

Base.readline – Function.

```
readline(stream::IO=STDIN; chomp::Bool=true)
readline(filename::AbstractString; chomp::Bool=true)
```

Read a single line of text from the given I/O stream or file (defaults to STDIN). When reading from a file, the text is assumed to be encoded in UTF-8. Lines in the input end with '\n' or "\r\n" or the end of an input stream. When chomp is true (as it is by default), these trailing newline characters are removed from the line before it is returned. When chomp is false, they are returned as part of the line.

source

Base.readuntil – Function.

```
readuntil(stream::IO, delim)
readuntil(filename::AbstractString, delim)
```

Read a string from an I/O stream or a file, up to and including the given delimiter byte. The text is assumed to be encoded in UTF-8.

source

Base.readlines – Function.

```
readlines(stream::IO=STDIN; chomp::Bool=true)
readlines(filename::AbstractString; chomp::Bool=true)
```

Read all lines of an I/O stream or a file as a vector of strings. Behavior is equivalent to saving the result of reading readline repeatedly with the same arguments and saving the resulting lines as a vector of strings.

source

Base.eachline – Function.

```
eachline(stream::IO=STDIN; chomp::Bool=true)
eachline(filename::AbstractString; chomp::Bool=true)
```

Create an iterable EachLine object that will yield each line from an I/O stream or a file. Iteration calls readline on the stream argument repeatedly with chomp passed through, determining whether trailing end-of-line characters are removed. When called with a file name, the file is opened once at the beginning of iteration and closed at the end. If iteration is interrupted, the file will be closed when the EachLine object is garbage collected.

source

Base.DataFmt.readdlm – Method.

```
readdlm(source, delim::Char, T::Type, eol::Char; header=false, skipstart=0, skipblanks=true,
    use_mmap, quotes=true, dims, comments=true, comment_char='#')
```

Read a matrix from the source where each line (separated by eol) gives one row, with elements separated by the given delimiter. The source can be a text file, stream or byte array. Memory mapped files can be used by passing the byte array representation of the mapped segment as source.

If T is a numeric type, the result is an array of that type, with any non-numeric elements as NaN for floating-point types, or zero. Other useful values of T include String, AbstractString, and Any.

If header is true, the first row of data will be read as header and the tuple (data_cells, header_cells) is returned instead of only data_cells.

Specifying skipstart will ignore the corresponding number of initial lines from the input.

If skipblanks is true, blank lines in the input will be ignored.

If use_mmap is true, the file specified by source is memory mapped for potential speedups. Default is true except on Windows. On Windows, you may want to specify true if the file is large, and is only read once and not written to.

If `quotes` is `true`, columns enclosed within double-quote (") characters are allowed to contain new lines and column delimiters. Double-quote characters within a quoted field must be escaped with another double-quote. Specifying `dims` as a tuple of the expected rows and columns (including header, if any) may speed up reading of large files. If `comments` is `true`, lines beginning with `comment_char` and text following `comment_char` in any line are ignored.

source

`Base.DataFmt.readdlm` – Method.

```
readdlm(source, delim::Char, eol::Char; options...)
```

If all data is numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned.

source

`Base.DataFmt.readdlm` – Method.

```
readdlm(source, delim::Char, T::Type; options...)
```

The end of line delimiter is taken as `\n`.

source

`Base.DataFmt.readdlm` – Method.

```
readdlm(source, delim::Char; options...)
```

The end of line delimiter is taken as `\n`. If all data is numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned.

source

`Base.DataFmt.readdlm` – Method.

```
readdlm(source, T::Type; options...)
```

The columns are assumed to be separated by one or more whitespaces. The end of line delimiter is taken as `\n`.

source

`Base.DataFmt.readdlm` – Method.

```
readdlm(source; options...)
```

The columns are assumed to be separated by one or more whitespaces. The end of line delimiter is taken as `\n`. If all data is numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned.

source

`Base.DataFmt.writedlm` – Function.

```
writedlm(f, A, delim='\t'; opts)
```

Write `A` (a vector, matrix, or an iterable collection of iterable rows) as text to `f` (either a filename string or an `IO` stream) using the given delimiter `delim` (which defaults to tab, but can be any printable Julia object, typically a `Char` or `AbstractString`).

For example, two vectors `x` and `y` of the same length can be written as two columns of tab-delimited text to `f` by either `writedlm(f, [x y])` or by `writedlm(f, zip(x, y))`.

source

Base.DataFmt.readcsv – Function.

```
readcsv(source, [T::Type]; options...)
```

Equivalent to readdlm with delim set to comma, and type optionally defined by T.

source

Base.DataFmt.writecsv – Function.

```
writecsv(filename, A; opts)
```

Equivalent to writedlm with delim set to comma.

source

Base.Base64.Base64EncodePipe – Type.

```
Base64EncodePipe(ostream)
```

Returns a new write-only I/O stream, which converts any bytes written to it into base64-encoded ASCII bytes written to ostream. Calling close on the Base64EncodePipe stream is necessary to complete the encoding (but does not close ostream).

```
julia> io = IOBuffer();

julia> iob64_encode = Base64EncodePipe(io);

julia> write(iob64_encode, "Hello!")
6

julia> close(iob64_encode);

julia> str = String(take!(io))
"SGVsbG8h"

julia> String(base64decode(str))
"Hello!"
```

source

Base.Base64.Base64DecodePipe – Type.

```
Base64DecodePipe(istream)
```

Returns a new read-only I/O stream, which decodes base64-encoded data read from istream.

```
julia> io = IOBuffer();

julia> iob64_decode = Base64DecodePipe(io);

julia> write(io, "SGVsbG8h")
8

julia> seekstart(io);

julia> String(read(iob64_decode))
"Hello!"
```

source

Base.Base64.base64encode – Function.

```
base64encode(writefunc, args...)
base64encode(args...)
```

Given a write-like function writefunc, which takes an I/O stream as its first argument, base64encode(writefunc, args...) calls writefunc to write args... to a base64-encoded string, and returns the string. base64encode(args...) is equivalent to base64encode(write, args...): it converts its arguments into bytes using the standard write functions and returns the base64-encoded string.

See also base64decode.

source

Base.Base64.base64decode – Function.

```
base64decode(string)
```

Decodes the base64-encoded string and returns a Vector{UInt8} of the decoded bytes.

See also base64encode

```
julia> b = base64decode("SGVsbG8h")
6-element Array{UInt8,1}:
 0x48
 0x65
 0x6c
 0x6c
 0x6f
 0x21

julia> String(b)
"Hello!"
```

source

Base.displaysize – Function.

```
displaysize(io) -> (lines, columns)
```

Return the nominal size of the screen that may be used for rendering output to this io object

source

55.3 E/S Multimedia

Del mismo modo que la salida de texto se realiza mediante print y los tipos definidos por el usuario pueden indicar su representación textual sobrecargando show, Julia proporciona un mecanismo estandarizado para una salida multimedia enriquecida (como imágenes, texto formateado, o incluso audio y video), que consta de tres partes:

- Una función display(x) para solicitar la visualización multimedia más completa disponible de un objeto Julia x (con una reserva de texto sin formato).
- Sobrecargar show permite indicar representaciones multimedia arbitrarias (codificadas mediante tipos MIME estándar) de tipos definidos por el usuario.

- Pueden registrarse backends de visualización con capacidad multimedia subclasificando un tipo genérico de `Display` y poniéndolos en una pila de backends de visualización mediante `pushdisplay`.

El tiempo de ejecución base de Julia solo proporciona visualización de texto sin formato, pero las pantallas más ricas pueden habilitarse cargando módulos externos o utilizando entornos gráficos de Julia (como el *notebook* IJulia, basado en IPython).

`Base.Multimedia.display` – Function.

```
display(x)
display(d::Display, x)
display(mime, x)
display(d::Display, mime, x)
```

Display x using the topmost applicable display in the display stack, typically using the richest supported multimedia output for x, with plain-text `STDOUT` output as a fallback. The `display(d, x)` variant attempts to display x on the given display d only, throwing a `MethodError` if d cannot display objects of this type.

There are also two variants with a `mime` argument (a MIME type string, such as `"image/png"`), which attempt to display x using the requested MIME type *only*, throwing a `MethodError` if this type is not supported by either the display(s) or by x. With these variants, one can also supply the "raw" data in the requested MIME type by passing `x::AbstractString` (for MIME types with text-based storage, such as text/html or application/postscript) or `x::Vector{UInt8}` (for binary MIME types).

source

`Base.Multimedia.redisplay` – Function.

```
redisplay(x)
redisplay(d::Display, x)
redisplay(mime, x)
redisplay(d::Display, mime, x)
```

By default, the `redisplay` functions simply call `display`. However, some display backends may override `redisplay` to modify an existing display of x (if any). Using `redisplay` is also a hint to the backend that x may be redisplayed several times, and the backend may choose to defer the display until (for example) the next interactive prompt.

source

`Base.Multimedia.displayable` – Function.

```
displayable(mime) -> Bool
displayable(d::Display, mime) -> Bool
```

Returns a boolean value indicating whether the given `mime` type (string) is displayable by any of the displays in the current display stack, or specifically by the display d in the second variant.

source

`Base.show` – Method.

```
show(stream, mime, x)
```

The `display` functions ultimately call `show` in order to write an object x as a given `mime` type to a given I/O `stream` (usually a memory buffer), if possible. In order to provide a rich multimedia representation of a user-defined type T, it is only necessary to define a new `show` method for T, via: `show(stream, ::MIME"mime",`

`x::T) = ...`, where `mime` is a MIME-type string and the function body calls `write` (or similar) to write that representation of `x` to `stream`. (Note that the `MIME""` notation only supports literal strings; to construct `MIME` types in a more flexible manner use `MIME{Symbol("")}`.)

For example, if you define a `MyImage` type and know how to write it to a PNG file, you could define a function `show(stream, ::MIME"image/png", x::MyImage) = ...` to allow your images to be displayed on any PNG-capable `Display` (such as IJulia). As usual, be sure to `import Base.show` in order to add new methods to the built-in Julia function show.

The default MIME type is `MIME"text/plain"`. There is a fallback definition for `text/plain` output that calls show with 2 arguments. Therefore, this case should be handled by defining a 2-argument `show(stream::IO, x::MyType)` method.

Technically, the `MIME"mime"` macro defines a singleton type for the given `mime` string, which allows us to exploit Julia's dispatch mechanisms in determining how to display objects of any given type.

The first argument to show can be an `IOContext` specifying output format properties. See `IOContext` for details.

source

`Base.Multimedia.mimewritable` – Function.

```
mimewritable(mime, x)
```

Returns a boolean value indicating whether or not the object `x` can be written as the given `mime` type. (By default, this is determined automatically by the existence of the corresponding `show` method for `typeof(x)`.)

source

`Base.Multimedia.reprmime` – Function.

```
reprmime(mime, x)
```

Returns an `AbstractString` or `Vector{UInt8}` containing the representation of `x` in the requested `mime` type, as written by show (throwing a `MethodError` if no appropriate show is available). An `AbstractString` is returned for MIME types with textual representations (such as `"text/html"` or `"application/postscript"`), whereas binary data is returned as `Vector{UInt8}`. (The function `istextmime(mime)` returns whether or not Julia treats a given `mime` type as text.)

As a special case, if `x` is an `AbstractString` (for textual MIME types) or a `Vector{UInt8}` (for binary MIME types), the `reprmime` function assumes that `x` is already in the requested `mime` format and simply returns `x`. This special case does not apply to the `"text/plain"` MIME type. This is useful so that raw data can be passed to `display(m::MIME, x)`.

source

`Base.Multimedia.stringmime` – Function.

```
stringmime(mime, x)
```

Returns an `AbstractString` containing the representation of `x` in the requested `mime` type. This is similar to `reprmime` except that binary data is base64-encoded as an ASCII string.

source

Como se mencionó anteriormente, también se pueden definir nuevos backends de pantalla. Por ejemplo, un módulo que puede mostrar imágenes PNG en una ventana puede registrar esta capacidad con Julia, de modo que al llamar a `display(x)` en tipos con representaciones PNG, se mostrará automáticamente la imagen usando la ventana del módulo.

Para definir un nuevo backend de pantalla, primero se debe crear un subtipo `D` de la clase abstracta `Display`. Luego, para cada tipo MIME (cadena `mime`) que se puede mostrar en `D`, se debe definir una función `display(d::D, ::MIME"mime", x) = ...` que muestra `x` como ese tipo MIME, generalmente llamando a `reprmime(mime, x)`. Se debe lanzar un `MethodError` si `x` no se puede mostrar como ese tipo MIME; esto es automático si se llama a `reprmime`. Finalmente, se debe definir una función `display(d::D, x)` que consulte `mimewritable(mime,x)` para los tipos `mime` soportados por `D` y muestre el "mejor"; debe lanzarse un `MethodError` si no se encuentran tipos MIME soportados para `x`. Del mismo modo, algunos subtipos pueden sobreescribir `redisplay(d::D, ...)`. (De nuevo, se debe hacer `import Base.display` para agregar nuevos métodos a `display`.) Los valores de retorno de estas funciones dependen de la implementación (ya que en algunos casos puede ser útil devolver un "manejador" *handle* de visualización de algún tipo). Las funciones de visualización para `D` se pueden llamar directamente, pero también se pueden invocar automáticamente desde `display(x)` simplemente presionando una nueva pantalla en la pila display-backend con:

`Base.Multimedia.pushdisplay` – Function.

```
pushdisplay(d::Display)
```

Pushes a new display `d` on top of the global display-backend stack. Calling `display(x)` or `display(mime, x)` will display `x` on the topmost compatible backend in the stack (i.e., the topmost backend that does not throw a `MethodError`).

source

`Base.Multimedia.popdisplay` – Function.

```
popdisplay()
popdisplay(d::Display)
```

Pop the topmost backend off of the display-backend stack, or the topmost copy of `d` in the second variant.

source

`Base.Multimedia.TextDisplay` – Type.

```
TextDisplay(io::IO)
```

Returns a `TextDisplay <: Display`, which displays any object as the text/plain MIME type (by default), writing the text representation to the given I/O stream. (This is how objects are printed in the Julia REPL.)

source

`Base.Multimedia.istextmime` – Function.

```
istextmime(m::MIME)
```

Determine whether a MIME type is text data. MIME types are assumed to be binary data except for a set of types known to be text data (possibly Unicode).

source

55.4 E/S Mapeada en Memoria

`Base.Mmap.Anonymous` – Type.

```
Mmap.Anonymous(name, readonly, create)
```

Create an `IO`-like object for creating zeroed-out mmapped-memory that is not tied to a file for use in `Mmap.mmap`. Used by `SharedArray` for creating shared memory arrays.

source

Base.Mmap.mmap – Method.

```
Mmap.mmap(io::Union{IOStream,AbstractString,Mmap.AnonymousMmap}[, type::Type{Array{T,N}},
    dims, offset]; grow::Bool=true, shared::Bool=true)
      Mmap.mmap(type::Type{Array{T,N}}, dims)
```

Create an `Array` whose values are linked to a file, using memory-mapping. This provides a convenient way of working with data too large to fit in the computer's memory.

The type is an `Array{T,N}` with a bits-type element of `T` and dimension `N` that determines how the bytes of the array are interpreted. Note that the file must be stored in binary format, and no format conversions are possible (this is a limitation of operating systems, not Julia).

`dims` is a tuple or single `Integer` specifying the size or length of the array.

The file is passed via the stream argument, either as an open `IOStream` or filename string. When you initialize the stream, use `"r"` for a "read-only" array, and `"w+"` to create a new array used to write values to disk.

If no `type` argument is specified, the default is `Vector{UInt8}`.

Optionally, you can specify an offset (in bytes) if, for example, you want to skip over a header in the file. The default value for the offset is the current stream position for an `IOStream`.

The `grow` keyword argument specifies whether the disk file should be grown to accommodate the requested size of array (if the total file size is < requested array size). Write privileges are required to grow the file.

The `shared` keyword argument specifies whether the resulting `Array` and changes made to it will be visible to other processes mapping the same file.

For example, the following code

```
# Create a file for mmapping
# (you could alternatively use mmap to do this step, too)
A = rand(1:20, 5, 30)
s = open("/tmp/mmap.bin", "w+")
# We'll write the dimensions of the array as the first two Ints in the file
write(s, size(A,1))
write(s, size(A,2))
# Now write the data
write(s, A)
close(s)

# Test by reading it back in
s = open("/tmp/mmap.bin")   # default is read-only
m = read(s, Int)
n = read(s, Int)
A2 = Mmap.mmap(s, Matrix{Int}, (m,n))
```

creates a `m`-by-`n` `Matrix{Int}`, linked to the file associated with stream `s`.

A more portable file would need to encode the word size – 32 bit or 64 bit – and endianness information in the header. In practice, consider encoding binary data using standard formats like HDF5 (which can be used with memory-mapping).

source

Base.Mmap.mmap – Method.

```
Mmap.mmap(io, BitArray, [dims, offset])
```

Create a `BitArray` whose values are linked to a file, using memory-mapping; it has the same purpose, works in the same way, and has the same arguments, as `mmap`, but the byte representation is different.

Example: `B = Mmap.mmap(s, BitArray, (25,30000))`

This would create a 25-by-30000 `BitArray`, linked to the file associated with stream `s`.

source

`Base.Mmap.sync!` – Function.

```
Mmap.sync!(array)
```

Forces synchronization between the in-memory version of a memory-mapped `Array` or `BitArray` and the on-disk version.

source

55.5 E/S por Red

`Base.connect` – Method.

```
connect([host], port::Integer) -> TCPSocket
```

Connect to the host `host` on port `port`.

source

`Base.connect` – Method.

```
connect(path::AbstractString) -> PipeEndpoint
```

Connect to the named pipe / UNIX domain socket at `path`.

source

`Base.listen` – Method.

```
listen([addr, ]port::Integer; backlog::Integer=BACKLOG_DEFAULT) -> TCPServer
```

Listen on port on the address specified by `addr`. By default this listens on `localhost` only. To listen on all interfaces pass `IPv4(0)` or `IPv6(0)` as appropriate. `backlog` determines how many connections can be pending (not having called `accept`) before the server will begin to reject them. The default value of `backlog` is `511`.

source

`Base.listen` – Method.

```
listen(path::AbstractString) -> PipeServer
```

Create and listen on a named pipe / UNIX domain socket.

source

`Base.getaddrinfo` – Function.

```
getaddrinfo(host::AbstractString) -> IPAddr
```

Gets the IP address of the `host` (may have to do a DNS lookup)

source

`Base.getsockname` – Function.

```
getsockname(sock::Union{TCPServer, TCPSocket}) -> (IPAddr, UInt16)
```

Get the IP address and the port that the given `TCPSocket` is connected to (or bound to, in the case of `TCPServer`).

source

`Base.IPv4` – Type.

```
IPv4(host::Integer) -> IPv4
```

Returns an IPv4 object from ip address `host` formatted as an `Integer`.

```
julia> IPv4(3223256218)
ip"192.30.252.154"
```

source

`Base.IPv6` – Type.

```
IPv6(host::Integer) -> IPv6
```

Returns an IPv6 object from ip address `host` formatted as an `Integer`.

```
julia> IPv6(3223256218)
ip"::c01e:fc9a"
```

source

`Base.nb_available` – Function.

```
nb_available(stream)
```

Returns the number of bytes available for reading before a read from this stream or buffer will block.

source

`Base.accept` – Function.

```
accept(server[,client])
```

Accepts a connection on the given server and returns a connection to the client. An uninitialized client stream may be provided, in which case it will be used instead of creating a new stream.

source

`Base.listenany` – Function.

```
listenany([host::IPAddr,] port_hint) -> (UInt16, TCPServer)
```

Create a `TCPServer` on any port, using hint as a starting point. Returns a tuple of the actual port that the server was created on and the server itself.

source

`Base.Filesystem.poll_fd` – Function.

```
poll_fd(fd, timeout_s::Real=-1; readable=false, writable=false)
```

Monitor a file descriptor `fd` for changes in the read or write availability, and with a timeout given by `timeout_s` seconds.

The keyword arguments determine which of read and/or write status should be monitored; at least one of them must be set to `true`.

The returned value is an object with boolean fields `readable`, `writable`, and `timedout`, giving the result of the polling.

source

`Base.Filesystem.poll_file` – Function.

```
poll_file(path::AbstractString, interval_s::Real=5.007, timeout_s::Real=-1) -> (previous::
    StatStruct, current::StatStruct)
```

Monitor a file for changes by polling every `interval_s` seconds until a change occurs or `timeout_s` seconds have elapsed. The `interval_s` should be a long period; the default is 5.007 seconds.

Returns a pair of `StatStruct` objects `(previous, current)` when a change is detected.

To determine when a file was modified, compare `mtime(prev) != mtime(current)` to detect notification of changes. However, using `watch_file` for this operation is preferred, since it is more reliable and efficient, although in some situations it may not be available.

source

`Base.Filesystem.watch_file` – Function.

```
watch_file(path::AbstractString, timeout_s::Real=-1)
```

Watch file or directory `path` for changes until a change occurs or `timeout_s` seconds have elapsed.

The returned value is an object with boolean fields `changed`, `renamed`, and `timedout`, giving the result of watching the file.

This behavior of this function varies slightly across platforms. See https://nodejs.org/api/fs.html#fs_caveats for more detailed information.

source

`Base.bind` – Function.

```
bind(socket::Union{UDPSocket, TCPSocket}, host::IPAddr, port::Integer; ipv6only=false,
    reuseaddr=false, kws...)
```

Bind `socket` to the given `host:port`. Note that `0.0.0.0` will listen on all devices.

- The `ipv6only` parameter disables dual stack mode. If `ipv6only=true`, only an IPv6 stack is created.
- If `reuseaddr=true`, multiple threads or processes can bind to the same address without error if they all set `reuseaddr=true`, but only the last to bind will receive any traffic.

source

```
bind(chnl::Channel, task::Task)
```

Associates the lifetime of `chnl` with a task. Channel `chnl` is automatically closed when the task terminates. Any uncaught exception in the task is propagated to all waiters on `chnl`.

The `chnl` object can be explicitly closed independent of task termination. Terminating tasks have no effect on already closed Channel objects.

When a channel is bound to multiple tasks, the first task to terminate will close the channel. When multiple channels are bound to the same task, termination of the task will close all of the bound channels.

```
julia> c = Channel(0);

julia> task = @schedule foreach(i->put!(c, i), 1:4);

julia> bind(c,task);

julia> for i in c

           @show i

       end;
i = 1
i = 2
i = 3
i = 4

julia> isopen(c)
false
```

```
julia> c = Channel(0);

julia> task = @schedule (put!(c,1);error("foo"));

julia> bind(c,task);

julia> take!(c)
1

julia> put!(c,1);
ERROR: foo
Stacktrace:
 [1] check_channel_state(::Channel{Any}) at ./channels.jl:131
 [2] put!(::Channel{Any}, ::Int64) at ./channels.jl:261
```

source

`Base.send` – Function.

```
send(socket::UDPSocket, host, port::Integer, msg)
```

Send `msg` over `socket` to `host:port`.

source

`Base.recv` – Function.

```
recv(socket::UDPSocket)
```

Read a UDP packet from the specified socket, and return the bytes received. This call blocks.

source

Base.recvfrom – Function.

```
recvfrom(socket::UDPSocket) -> (address, data)
```

Read a UDP packet from the specified socket, returning a tuple of `(address, data)`, where `address` will be either IPv4 or IPv6 as appropriate.

source

Base.setopt – Function.

```
setopt(sock::UDPSocket; multicast_loop = nothing, multicast_ttl=nothing, enable_broadcast=
    nothing, ttl=nothing)
```

Set UDP socket options.

- `multicast_loop`: loopback for multicast packets (default: `true`).
- `multicast_ttl`: TTL for multicast packets (default: `nothing`).
- `enable_broadcast`: flag must be set to `true` if socket will be used for broadcast messages, or else the UDP system will return an access error (default: `false`).
- `ttl`: Time-to-live of packets sent on the socket (default: `nothing`).

source

Base.ntoh – Function.

```
ntoh(x)
```

Converts the endianness of a value from Network byte order (big-endian) to that used by the Host.

source

Base.hton – Function.

```
hton(x)
```

Converts the endianness of a value from that used by the Host to Network byte order (big-endian).

source

Base.ltoh – Function.

```
ltoh(x)
```

Converts the endianness of a value from Little-endian to that used by the Host.

source

Base.htol – Function.

```
htol(x)
```

Converts the endianness of a value from that used by the Host to Little-endian.

source

Base.ENDIAN_BOM – Constant.

```
ENDIAN_BOM
```

The 32-bit byte-order-mark indicates the native byte order of the host machine. Little-endian machines will contain the value `0x04030201`. Big-endian machines will contain the value `0x01020304`.

source

Chapter 56

Puntuación

Puede encontrar documentación extendida sobre símbolos y funciones matemáticas aquí.

Símbolo	Significado
`@m`	Invoca la macro `m`; seguido de expresiones separadas por espacios
`!`	Operador ”not” prefijo
`a!( )`	Al final de un nombre de función, `!` indica que la función modifica su(s) argumento(s)
#	Inicio de un comentario de una sola línea
#=	Inicio de un comentario multilínea (pueden ser anidados)
=#	Final de un comentario multilínea
$	Interpolación de cadena y expresión
%	Operador resto
^	Operador exponenente
&	Operador and bit-a-bit
&&	Operador and booleano (en corto-circuito)
|	Operador or bit-a-bit
||	Operador or booleano (en corto-circuito)
	Operador xor bit-a-bit
*	Multiplicación o producto matricial
( )	Tupla vacía
~	Operador not bit-a-bit
\	Operador backslash
'	Operador transpuesto complejo A^H
a[ ]	Indexación de array
[,]	Concatenación vertical
[;]	Concatenación vertical (también)
[ ]	Con expresiones separadas por espacios, concatenación horizontal
T{ }	Instanciación de tipo paramétrico
;	Separador de instrucciones
,	Separador de argumentos de función o de componentes de una tupla
?	Operador condicional ternario (conditional ? if_true : if_false)
" "	Delimitador de literales cadena
' '	Delimitador de literales carácter
` `	Delimitador de especificaciones de proceso externo (mandato)
. . .	Une argumentos en una llamada a función o declara una función o tipo varargs
.	Acceso nombrado a campos en objectos/módulos, también llamadas a operadores/funciones vectorizadas
a:b	Rango a, a+1, a+2, ..., b
a:s:b	Rango a, a+s, a+2s, ..., b
:	Indexa una dimensión entera (1:final)
::	Anotación de tipo, dependiendo del contexto
:( )	Expresión citada
:a	Símbolo a
<:	`subtype operator`
>:	`supertype operator` (contrario al anterior)
===	`egal comparison operator`

Chapter 57

Ordenación y Funciones Relacionadas

Julia tiene una API amplia y flexible para ordenar e interactuar con matrices de valores ya ordenados. Por defecto, Julia selecciona algoritmos y ordenaciones razonables en orden ascendente estándar:

```
julia> sort([2,3,1])
3-element Array{Int64,1}:
 1
 2
 3
```

También se puede ordenar en orden inverso:

```
julia> sort([2,3,1], rev=true)
3-element Array{Int64,1}:
 3
 2
 1
```

Para ordenar un array sobre sí mismo, use la versión con admiración de la función de ordenación:

```
julia> a = [2,3,1];

julia> sort!(a);

julia> a
3-element Array{Int64,1}:
 1
 2
 3
```

En lugar de ordenar un array directamente, podemos computar una permutación de los índices del array que ponen el array en un orden determinado:

```
julia> v = randn(5)
5-element Array{Float64,1}:
  0.297288
  0.382396
 -0.597634
```

```
 -0.0104452
 -0.839027

julia> p = sortperm(v)
5-element Array{Int64,1}:
 5
 3
 4
 1
 2

julia> v[p]
5-element Array{Float64,1}:
 -0.839027
 -0.597634
 -0.0104452
  0.297288
  0.382396
```

Los arrays pueden ser ordenados fácilmente de acuerdo a una transformación arbitraria de sus valores:

```
julia> sort(v, by=abs)
5-element Array{Float64,1}:
 -0.0104452
  0.297288
  0.382396
 -0.597634
 -0.839027
```

O en orden reverso mediante una transformación:

```
julia> sort(v, by=abs, rev=true)
5-element Array{Float64,1}:
 -0.839027
 -0.597634
  0.382396
  0.297288
 -0.0104452
```

Si es necesario, puede elegirse el algoritmo de ordenación:

```
julia> sort(v, alg=InsertionSort)
5-element Array{Float64,1}:
 -0.839027
 -0.597634
 -0.0104452
  0.297288
  0.382396
```

Todas las funciones de ordenación y relacionadas con el orden se basan en una relación ”menor que” que define un orden total sobre los valores que van a manipularse. La función `isless` es la invocada por defecto, pero la relación puede ser especificada mediante la palabra clave `lt`.

57.1 Funciones de Ordenación

`Base.sort!` – Function.

```
sort!(v; alg::Algorithm=defalg(v), lt=isless, by=identity, rev::Bool=false, order::Ordering=
    Forward)
```

Sort the vector `v` in place. `QuickSort` is used by default for numeric arrays while `MergeSort` is used for other arrays. You can specify an algorithm to use via the `alg` keyword (see Sorting Algorithms for available algorithms). The `by` keyword lets you provide a function that will be applied to each element before comparison; the `lt` keyword allows providing a custom "less than" function; use `rev=true` to reverse the sorting order. These options are independent and can be used together in all possible combinations: if both `by` and `lt` are specified, the `lt` function is applied to the result of the `by` function; `rev=true` reverses whatever ordering specified via the `by` and `lt` keywords.

Examples

```
julia> v = [3, 1, 2]; sort!(v); v
3-element Array{Int64,1}:
 1
 2
 3

julia> v = [3, 1, 2]; sort!(v, rev = true); v
3-element Array{Int64,1}:
 3
 2
 1

julia> v = [(1, "c"), (3, "a"), (2, "b")]; sort!(v, by = x -> x[1]); v
3-element Array{Tuple{Int64,String},1}:
 (1, "c")
 (2, "b")
 (3, "a")

julia> v = [(1, "c"), (3, "a"), (2, "b")]; sort!(v, by = x -> x[2]); v
3-element Array{Tuple{Int64,String},1}:
 (3, "a")
 (2, "b")
 (1, "c")
```

source

`Base.sort` – Function.

```
sort(v; alg::Algorithm=defalg(v), lt=isless, by=identity, rev::Bool=false, order::Ordering=
    Forward)
```

Variant of `sort!` that returns a sorted copy of `v` leaving `v` itself unmodified.

Examples

```
julia> v = [3, 1, 2];

julia> sort(v)
3-element Array{Int64,1}:
 1
```

```
 2
 3

julia> v
3-element Array{Int64,1}:
 3
 1
 2
```

source

```
sort(A, dim::Integer; alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity, rev::Bool=
    false, order::Ordering=Forward, initialized::Bool=false)
```

Sort a multidimensional array A along the given dimension. See `sort!` for a description of possible keyword arguments.

Examples

```
julia> A = [4 3; 1 2]
2×2 Array{Int64,2}:
 4  3
 1  2

julia> sort(A, 1)
2×2 Array{Int64,2}:
 1  2
 4  3

julia> sort(A, 2)
2×2 Array{Int64,2}:
 3  4
 1  2
```

source

`Base.sortperm` – Function.

```
sortperm(v; alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity, rev::Bool=false, order::
    Ordering=Forward)
```

Return a permutation vector of indices of v that puts it in sorted order. Specify `alg` to choose a particular sorting algorithm (see Sorting Algorithms). `MergeSort` is used by default, and since it is stable, the resulting permutation will be the lexicographically first one that puts the input array into sorted order – i.e. indices of equal elements appear in ascending order. If you choose a non-stable sorting algorithm such as `QuickSort`, a different permutation that puts the array into order may be returned. The order is specified using the same keywords as `sort!`.

See also `sortperm!`.

Examples

```
julia> v = [3, 1, 2];

julia> p = sortperm(v)
3-element Array{Int64,1}:
 2
 3
```

```
 1

julia> v[p]
3-element Array{Int64,1}:
 1
 2
 3
```

source

Base.Sort.sortperm! – Function.

```
sortperm!(ix, v; alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity, rev::Bool=false,
    order::Ordering=Forward, initialized::Bool=false)
```

Like sortperm, but accepts a preallocated index vector ix. If initialized is false (the default), ix is initialized to contain the values 1:length(v).

Examples

```
julia> v = [3, 1, 2]; p = zeros(Int, 3);

julia> sortperm!(p, v); p
3-element Array{Int64,1}:
 2
 3
 1

julia> v[p]
3-element Array{Int64,1}:
 1
 2
 3
```

source

Base.Sort.sortrows – Function.

```
sortrows(A; alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity, rev::Bool=false, order::
    Ordering=Forward)
```

Sort the rows of matrix A lexicographically. See sort! for a description of possible keyword arguments.

Examples

```
julia> sortrows([7 3 5; -1 6 4; 9 -2 8])
3×3 Array{Int64,2}:
 -1   6  4
  7   3  5
  9  -2  8

julia> sortrows([7 3 5; -1 6 4; 9 -2 8], lt=(x,y)->isless(x[2],y[2]))
3×3 Array{Int64,2}:
  9  -2  8
  7   3  5
 -1   6  4
```

```
julia> sortrows([7 3 5; -1 6 4; 9 -2 8], rev=true)
3×3 Array{Int64,2}:
  9  -2  8
  7   3  5
 -1   6  4
```

source

Base.Sort.sortcols – Function.

```
sortcols(A; alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity, rev::Bool=false, order::
    Ordering=Forward)
```

Sort the columns of matrix A lexicographically. See sort! for a description of possible keyword arguments.

Examples

```
julia> sortcols([7 3 5; 6 -1 -4; 9 -2 8])
3×3 Array{Int64,2}:
  3   5  7
 -1  -4  6
 -2   8  9

julia> sortcols([7 3 5; 6 -1 -4; 9 -2 8], alg=InsertionSort, lt=(x,y)->isless(x[2],y[2]))
3×3 Array{Int64,2}:
  5   3  7
 -4  -1  6
  8  -2  9

julia> sortcols([7 3 5; 6 -1 -4; 9 -2 8], rev=true)
3×3 Array{Int64,2}:
 7   5   3
 6  -4  -1
 9   8  -2
```

source

57.2 Funciones Relacionadas con Orden

Base.issorted – Function.

```
issorted(v, lt=isless, by=identity, rev:Bool=false, order::Ordering=Forward)
```

Test whether a vector is in sorted order. The lt, by and rev keywords modify what order is considered to be sorted just as they do for sort.

Examples

```
julia> issorted([1, 2, 3])
true

julia> issorted([(1, "b"), (2, "a")], by = x -> x[1])
true

julia> issorted([(1, "b"), (2, "a")], by = x -> x[2])
false
```

```
julia> issorted([(1, "b"), (2, "a")], by = x -> x[2], rev=true)
true
```

source

Base.Sort.searchsorted – Function.

```
searchsorted(a, x, [by=<transform>,] [lt=<comparison>,] [rev=false])
```

Returns the range of indices of a which compare as equal to x (using binary search) according to the order specified by the by, lt and rev keywords, assuming that a is already sorted in that order. Returns an empty range located at the insertion point if a does not contain values equal to x.

Examples

```
julia> a = [4, 3, 2, 1]
4-element Array{Int64,1}:
 4
 3
 2
 1

julia> searchsorted(a, 4)
5:4

julia> searchsorted(a, 4, rev=true)
1:1
```

source

Base.Sort.searchsortedfirst – Function.

```
searchsortedfirst(a, x, [by=<transform>,] [lt=<comparison>,] [rev=false])
```

Returns the index of the first value in a greater than or equal to x, according to the specified order. Returns length(a)+1 if x is greater than all values in a. a is assumed to be sorted.

Examples

```
julia> searchsortedfirst([1, 2, 4, 5, 14], 4)
3

julia> searchsortedfirst([1, 2, 4, 5, 14], 4, rev=true)
1

julia> searchsortedfirst([1, 2, 4, 5, 14], 15)
6
```

source

Base.Sort.searchsortedlast – Function.

```
searchsortedlast(a, x, [by=<transform>,] [lt=<comparison>,] [rev=false])
```

Returns the index of the last value in a less than or equal to x, according to the specified order. Returns 0 if x is less than all values in a. a is assumed to be sorted.

Examples

```
julia> searchsortedlast([1, 2, 4, 5, 14], 4)
3

julia> searchsortedlast([1, 2, 4, 5, 14], 4, rev=true)
5

julia> searchsortedlast([1, 2, 4, 5, 14], -1)
0
```

source

`Base.Sort.select!` – Function.

```
select!(v, k, [by=<transform>,] [lt=<comparison>,] [rev=false])
```

Partially sort the vector v in place, according to the order specified by by, `lt` and `rev` so that the value at index k (or range of adjacent values if k is a range) occurs at the position where it would appear if the array were fully sorted via a non-stable algorithm. If k is a single index, that value is returned; if k is a range, an array of values at those indices is returned. Note that `select!` does not fully sort the input array.

Examples

```
julia> a = [1, 2, 4, 3, 4]
5-element Array{Int64,1}:
 1
 2
 4
 3
 4

julia> select!(a, 4)
4

julia> a
5-element Array{Int64,1}:
 1
 2
 3
 4
 4

julia> a = [1, 2, 4, 3, 4]
5-element Array{Int64,1}:
 1
 2
 4
 3
 4

julia> select!(a, 4, rev=true)
2

julia> a
5-element Array{Int64,1}:
 4
 4
 3
```

```
2
1
```

source

Base.Sort.select – Function.

```
select(v, k, [by=<transform>,] [lt=<comparison>,] [rev=false])
```

Variant of select! which copies v before partially sorting it, thereby returning the same thing as select! but leaving v unmodified.

source

Base.Sort.selectperm – Function.

```
selectperm(v, k, [alg=<algorithm>,] [by=<transform>,] [lt=<comparison>,] [rev=false])
```

Return a partial permutation of the vector v, according to the order specified by by, lt and rev, so that v[output] returns the first k (or range of adjacent values if k is a range) values of a fully sorted version of v. If k is a single index (Integer), an array of the first k indices is returned; if k is a range, an array of those indices is returned. Note that the handling of integer values for k is different from select in that it returns a vector of k elements instead of just the k th element. Also note that this is equivalent to, but more efficient than, calling sortperm(...)[k].

source

Base.Sort.selectperm! – Function.

```
selectperm!(ix, v, k, [alg=<algorithm>,] [by=<transform>,] [lt=<comparison>,] [rev=false,] [
    initialized=false])
```

Like selectperm, but accepts a preallocated index vector ix. If initialized is false (the default), ix is initialized to contain the values 1:length(ix).

source

57.3 Algoritmos de Ordenación

Actualmente hay cuatro algoritmos de ordenación disponibles en Julia base:

- InsertionSort
- QuickSort
- PartialQuickSort(k)
- MergeSort

InsertionSort es un algoritmo de ordenación estable cuyo coste es O(n^2). Es eficiente para n muy pequeños, y es usado internamente por QuickSort.

QuickSort es un algoritmo de ordenación que es *in-place* muy rápido pero no estable (es decir, los elementos que son considerados iguales no permanecerán en el mismo orden en que se encontraban originalmente en el array antes de ser ordenados). Su coste computacional es O(n log n). QuickSort es el algoritmo por defecto para valores numéricos, incluyendo enteros y punto flotante.

PartialQuickSort(k) es similar a QuickSort, pero el array de salida es sólo ordenado hasta el índice k si k es un entero, o en el rango de k si k es un OrdinalRange. Por ejemplo:

```
x = rand(1:500, 100)
k = 50
k2 = 50:100
s = sort(x; alg=QuickSort)
ps = sort(x; alg=PartialQuickSort(k))
qs = sort(x; alg=PartialQuickSort(k2))
map(issorted, (s, ps, qs))            # => (true, false, false)
map(x->issorted(x[1:k]), (s, ps, qs))  # => (true, true, false)
map(x->issorted(x[k2]), (s, ps, qs))   # => (true, false, true)
s[1:k] == ps[1:k]                     # => true
s[k2] == qs[k2]                       # => true
```

`MergeSort` es un algoritmo de ordenación estable, pero no *in-place* (requiere un array temporal de la mitad del tamaño del array de entrada), de coste O(n log n) y no suele ser tan rapido como `QuickSort`. Es el algoritmo por defecto para datos no numéricos.

Los algoritmos de ordenación por defecto se eligen sobre la base de que son rápidos y estables, o *parezcan* serlo. Para los tipos numéricos, de hecho, se selecciona `QuickSort` ya que es más rápido e indistinguible en este caso de un tipo estable (a menos que la matriz registre sus mutaciones de alguna manera). La propiedad de estabilidad tiene un coste no despreciable, por lo que si no la necesita, puede especificar explícitamente su algoritmo preferido, p. `sort!(v, alg=QuickSort)`.

El mecanismo por el cual Julia selecciona los algoritmos de ordenación predeterminados se implementa a través de la función `Base.Sort.defalg`. Permite que un algoritmo particular se registre como el predeterminado en todas las funciones de ordenación para arrays específicos. Por ejemplo, aquí están los dos métodos predeterminados de `sort.jl`:

```
defalg(v::AbstractArray) = MergeSort
defalg{T<:Number}(v::AbstractArray{T}) = QuickSort
```

En cuanto a los arrays numéricos, la elección de un algoritmo predeterminado no estable para los tipos de array para los cuales la noción de ordenación estable no tiene sentido (es decir, cuando dos valores que comparan iguales no se pueden distinguir) puede tener sentido.

Chapter 58

Funciones del Administrador de Paquetes

Todas las funciones del administrador de paquetes están definidas en el módulo `Pkg`. Ninguna de las funciones del módulo `Pkg` están exportadas. Por tanto, para usarlas, necesitamos prefijar cada llamada a función con un `Pkg.` explícito, por ejemplo `Pkg.status()` o `Pkg.dir()`.

Las funciones para el desarrollo de paquetes (por ejemplo, `tag`, `publish`, etc.) se han movido al paquete PkgDev. Ver PkgDev README para la documentación de estas funciones.

`Base.Pkg.dir` – Function.

```
dir() -> AbstractString
```

Returns the absolute path of the package directory. This defaults to `joinpath(homedir(),".julia","v$(VERSION.major).$(VERSION.minor)")` on all platforms (i.e. `~/.julia/v0.6` in UNIX shell syntax). If the `JULIA_PKGDIR` environment variable is set, then that path is used in the returned value as `joinpath(ENV["JULIA_PKGDIR"],"v$(VERSION.major).$(VERSION.minor)")`. If `JULIA_PKGDIR` is a relative path, it is interpreted relative to whatever the current working directory is.

source

```
dir(names...) -> AbstractString
```

Equivalent to `normpath(Pkg.dir(),names...)` – i.e. it appends path components to the package directory and normalizes the resulting path. In particular, `Pkg.dir(pkg)` returns the path to the package `pkg`.

source

`Base.Pkg.init` – Function.

```
init(meta::AbstractString=DEFAULT_META, branch::AbstractString=META_BRANCH)
```

Initialize `Pkg.dir()` as a package directory. This will be done automatically when the `JULIA_PKGDIR` is not set and `Pkg.dir()` uses its default value. As part of this process, clones a local METADATA git repository from the site and branch specified by its arguments, which are typically not provided. Explicit (non-default) arguments can be used to support a custom METADATA setup.

source

`Base.Pkg.resolve` – Function.

```
resolve()
```

Determines an optimal, consistent set of package versions to install or upgrade to. The optimal set of package versions is based on the contents of `Pkg.dir("REQUIRE")` and the state of installed packages in `Pkg.dir()`, Packages that are no longer required are moved into `Pkg.dir(".trash")`.

source

`Base.Pkg.edit` – Function.

```
edit()
```

Opens `Pkg.dir("REQUIRE")` in the editor specified by the `VISUAL` or `EDITOR` environment variables; when the editor command returns, it runs `Pkg.resolve()` to determine and install a new optimal set of installed package versions.

source

`Base.Pkg.add` – Function.

```
add(pkg, vers...)
```

Add a requirement entry for `pkg` to `Pkg.dir("REQUIRE")` and call `Pkg.resolve()`. If `vers` are given, they must be `VersionNumber` objects and they specify acceptable version intervals for `pkg`.

source

`Base.Pkg.rm` – Function.

```
rm(pkg)
```

Remove all requirement entries for `pkg` from `Pkg.dir("REQUIRE")` and call `Pkg.resolve()`.

source

`Base.Pkg.clone` – Function.

```
clone(pkg)
```

If `pkg` has a URL registered in `Pkg.dir("METADATA")`, clone it from that URL on the default branch. The package does not need to have any registered versions.

source

```
clone(url, [pkg])
```

Clone a package directly from the git URL `url`. The package does not need to be registered in `Pkg.dir("METADATA")`. The package repo is cloned by the name `pkg` if provided; if not provided, `pkg` is determined automatically from `url`.

source

`Base.Pkg.setprotocol!` – Function.

```
setprotocol!(proto)
```

Set the protocol used to access GitHub-hosted packages. Defaults to 'https', with a blank `proto` delegating the choice to the package developer.

source

`Base.Pkg.available` – Function.

```
available() -> Vector{String}
```

Returns the names of available packages.

source

```
available(pkg) -> Vector{VersionNumber}
```

Returns the version numbers available for package pkg.

source

Base.Pkg.installed – Function.

```
installed() -> Dict{String,VersionNumber}
```

Returns a dictionary mapping installed package names to the installed version number of each package.

source

```
installed(pkg) -> Void | VersionNumber
```

If pkg is installed, return the installed version number. If pkg is registered, but not installed, return nothing.

source

Base.Pkg.status – Function.

```
status()
```

Prints out a summary of what packages are installed and what version and state they're in.

source

```
status(pkg)
```

Prints out a summary of what version and state pkg, specifically, is in.

source

Base.Pkg.update – Function.

```
update(pkgs...)
```

Update the metadata repo – kept in Pkg.dir("METADATA") – then update any fixed packages that can safely be pulled from their origin; then call Pkg.resolve() to determine a new optimal set of packages versions.

Without arguments, updates all installed packages. When one or more package names are provided as arguments, only those packages and their dependencies are updated.

source

Base.Pkg.checkout – Function.

```
checkout(pkg, [branch="master"]; merge=true, pull=true)
```

Checkout the Pkg.dir(pkg) repo to the branch branch. Defaults to checking out the "master" branch. To go back to using the newest compatible released version, use Pkg.free(pkg). Changes are merged (fast-forward only) if the keyword argument merge == true, and the latest version is pulled from the upstream repo if pull == true.

source

`Base.Pkg.pin` – Function.

```
pin(pkg)
```

Pin `pkg` at the current version. To go back to using the newest compatible released version, use `Pkg.free(pkg)`

source

```
pin(pkg, version)
```

Pin `pkg` at registered version `version`.

source

`Base.Pkg.free` – Function.

```
free(pkg)
```

Free the package `pkg` to be managed by the package manager again. It calls `Pkg.resolve()` to determine optimal package versions after. This is an inverse for both `Pkg.checkout` and `Pkg.pin`.

You can also supply an iterable collection of package names, e.g., `Pkg.free(("Pkg1", "Pkg2"))` to free multiple packages at once.

source

`Base.Pkg.build` – Function.

```
build()
```

Run the build scripts for all installed packages in depth-first recursive order.

source

```
build(pkgs...)
```

Run the build script in `deps/build.jl` for each package in `pkgs` and all of their dependencies in depth-first recursive order. This is called automatically by `Pkg.resolve()` on all installed or updated packages.

source

`Base.Pkg.test` – Function.

```
test(; coverage=false)
```

Run the tests for all installed packages ensuring that each package's test dependencies are installed for the duration of the test. A package is tested by running its `test/runtests.jl` file and test dependencies are specified in `test/REQUIRE`. Coverage statistics for the packages may be generated by passing `coverage=true`. The default behavior is not to run coverage.

source

```
test(pkgs...; coverage=false)
```

Run the tests for each package in `pkgs` ensuring that each package's test dependencies are installed for the duration of the test. A package is tested by running its `test/runtests.jl` file and test dependencies are specified in `test/REQUIRE`. Coverage statistics for the packages may be generated by passing `coverage=true`. The default behavior is not to run coverage.

source

`Base.Pkg.dependents` – Function.

```
dependents(pkg)
```

List the packages that have pkg as a dependency.

source

Chapter 59

Fechas y Tiempo

59.1 Tipos para Fechas y Tiempo

`Base.Dates.Period` – Type.

```
Period
Year
Month
Week
Day
Hour
Minute
Second
Millisecond
Microsecond
Nanosecond
```

`Period` types represent discrete, human representations of time.

source

`Base.Dates.CompoundPeriod` – Type.

```
CompoundPeriod
```

A `CompoundPeriod` is useful for expressing time periods that are not a fixed multiple of smaller periods. For example, "a year and a day" is not a fixed number of days, but can be expressed using a `CompoundPeriod`. In fact, a `CompoundPeriod` is automatically generated by addition of different period types, e.g. `Year(1) + Day(1)` produces a `CompoundPeriod` result.

source

`Base.Dates.Instant` – Type.

```
Instant
```

`Instant` types represent integer-based, machine representations of time as continuous timelines starting from an epoch.

source

`Base.Dates.UTInstant` – Type.

```
UTInstant{T}
```

The `UTInstant` represents a machine timeline based on UT time (1 day = one revolution of the earth). The `T` is a `Period` parameter that indicates the resolution or precision of the instant.

source

`Base.Dates.TimeType` – Type.

```
TimeType
```

`TimeType` types wrap `Instant` machine instances to provide human representations of the machine instant. `Time`, `DateTime` and `Date` are subtypes of `TimeType`.

source

`Base.Dates.DateTime` – Type.

```
DateTime
```

`DateTime` wraps a `UTInstant{Millisecond}` and interprets it according to the proleptic Gregorian calendar.

source

`Base.Dates.Date` – Type.

```
Date
```

`Date` wraps a `UTInstant{Day}` and interprets it according to the proleptic Gregorian calendar.

source

`Base.Dates.Time` – Type.

```
Time
```

`Time` wraps a `Nanosecond` and represents a specific moment in a 24-hour day.

source

59.2 Funciones para Fechas

Todas las funciones para fechas están definidas en el módulo `Dates`; nótese que solo se han exportado las funciones `Date`, `DateTime` y `now`; para usar el resto de funciones de `Dates`, es necesario prefijar cada llamada a función con `Dates.`, por ejemplo, `Dates.dayofweek(dt)`. Alternativamente, se puede escribir `using Base.Dates` para llevar todas las funciones exportadas a `Main` y que sean usadas sin el prefijo `Dates.`

`Base.Dates.DateTime` – Method.

```
DateTime(y, [m, d, h, mi, s, ms]) -> DateTime
```

Construct a `DateTime` type by parts. Arguments must be convertible to `Int64`.

source

`Base.Dates.DateTime` – Method.

```
DateTime(periods::Period...) -> DateTime
```

Construct a `DateTime` type by `Period` type parts. Arguments may be in any order. DateTime parts not provided will default to the value of `Dates.default(period)`.

source

`Base.Dates.DateTime` – Method.

```
DateTime(f::Function, y[, m, d, h, mi, s]; step=Day(1), limit=10000) -> DateTime
```

Create a `DateTime` through the adjuster API. The starting point will be constructed from the provided `y, m, d...` arguments, and will be adjusted until `f::Function` returns `true`. The step size in adjusting can be provided manually through the `step` keyword. `limit` provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (in the case that `f::Function` is never satisfied).

source

`Base.Dates.DateTime` – Method.

```
DateTime(dt::Date) -> DateTime
```

Converts a `Date` to a `DateTime`. The hour, minute, second, and millisecond parts of the new `DateTime` are assumed to be zero.

source

`Base.Dates.DateTime` – Method.

```
DateTime(dt::AbstractString, format::AbstractString; locale="english") -> DateTime
```

Construct a `DateTime` by parsing the `dt` date string following the pattern given in the `format` string.

This method creates a `DateFormat` object each time it is called. If you are parsing many date strings of the same format, consider creating a `DateFormat` object once and using that as the second argument instead.

source

`Base.Dates.format` – Method.

```
format(dt::TimeType, format::AbstractString; locale="english") -> AbstractString
```

Construct a string by using a `TimeType` object and applying the provided `format`. The following character codes can be used to construct the `format` string:

Code	Examples	Comment
y	6	Numeric year with a fixed width
Y	1996	Numeric year with a minimum width
m	1, 12	Numeric month with a minimum width
u	Jan	Month name shortened to 3-chars according to the `locale`
U	January	Full month name according to the `locale` keyword
d	1, 31	Day of the month with a minimum width
H	0, 23	Hour (24-hour clock) with a minimum width
M	0, 59	Minute with a minimum width
S	0, 59	Second with a minimum width
s	000, 500	Millisecond with a minimum width of 3
e	Mon, Tue	Abbreviated days of the week
E	Monday	Full day of week name

The number of sequential code characters indicate the width of the code. A format of `yyyy-mm` specifies that the code y should have a width of four while `m` a width of two. Codes that yield numeric digits have an associated mode: fixed-width or minimum-width. The fixed-width mode left-pads the value with zeros when it is shorter than the specified width and truncates the value when longer. Minimum-width mode works the same as fixed-width except that it does not truncate values longer than the width.

When creating a `format` you can use any non-code characters as a separator. For example to generate the string "1996-01-15T00:00:00" you could use `format`: "yyyy-mm-ddTHH:MM:SS". Note that if you need to use a code character as a literal you can use the escape character backslash. The string "1996y01m" can be produced with the format "yyyy\ymm\m".

source

`Base.Dates.DateFormat` – Type.

```
DateFormat(format::AbstractString, locale="english") -> DateFormat
```

Construct a date formatting object that can be used for parsing date strings or formatting a date object as a string. The following character codes can be used to construct the `format` string:

Code	Matches	Comment
y	1996, 96	Returns year of 1996, 0096
Y	1996, 96	Returns year of 1996, 0096. Equivalent to y
m	1, 01	Matches 1 or 2-digit months
u	Jan	Matches abbreviated months according to the `locale` keyword
U	January	Matches full month names according to the `locale` keyword
d	1, 01	Matches 1 or 2-digit days
H	00	Matches hours
M	00	Matches minutes
S	00	Matches seconds
s	.500	Matches milliseconds
e	Mon, Tues	Matches abbreviated days of the week
E	Monday	Matches full name days of the week
yyyymmdd	19960101	Matches fixed-width year, month, and day

Characters not listed above are normally treated as delimiters between date and time slots. For example a `dt` string of "1996-01-15T00:00:00.0" would have a `format` string like "y-m-dTH:M:S.s". If you need to use a code character as a delimiter you can escape it using backslash. The date "1995y01m" would have the format "y\ym\m".

Creating a DateFormat object is expensive. Whenever possible, create it once and use it many times or try the `dateformat""` string macro. Using this macro creates the DateFormat object once at macro expansion time and reuses it later. see `@dateformat_str`.

See `DateTime` and `format` for how to use a DateFormat object to parse and write Date strings respectively.

source

`Base.Dates.@dateformat_str` – Macro.

```
dateformat"Y-m-d H:M:S"
```

Create a `DateFormat` object. Similar to `DateFormat("Y-m-d H:M:S")` but creates the DateFormat object once during macro expansion.

See `DateFormat` for details about format specifiers.

source

Base.Dates.DateTime – Method.

```
DateTime(dt::AbstractString, df::DateFormat) -> DateTime
```

Construct a DateTime by parsing the dt date string following the pattern given in the DateFormat object. Similar to DateTime(::AbstractString, ::AbstractString) but more efficient when repeatedly parsing similarly formatted date strings with a pre-created DateFormat object.

source

Base.Dates.Date – Method.

```
Date(y, [m, d]) -> Date
```

Construct a Date type by parts. Arguments must be convertible to Int64.

source

Base.Dates.Date – Method.

```
Date(period::Period...) -> Date
```

Construct a Date type by Period type parts. Arguments may be in any order. Date parts not provided will default to the value of Dates.default(period).

source

Base.Dates.Date – Method.

```
Date(f::Function, y[, m, d]; step=Day(1), limit=10000) -> Date
```

Create a Date through the adjuster API. The starting point will be constructed from the provided y, m, d arguments, and will be adjusted until f::Function returns true. The step size in adjusting can be provided manually through the step keyword. limit provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (given that f::Function is never satisfied).

source

Base.Dates.Date – Method.

```
Date(dt::DateTime) -> Date
```

Converts a DateTime to a Date. The hour, minute, second, and millisecond parts of the DateTime are truncated, so only the year, month and day parts are used in construction.

source

Base.Dates.Date – Method.

```
Date(dt::AbstractString, format::AbstractString; locale="english") -> Date
```

Construct a Date object by parsing a dt date string following the pattern given in the format string. Follows the same conventions as DateTime(::AbstractString, ::AbstractString).

source

Base.Dates.Date – Method.

```
Date(dt::AbstractString, df::DateFormat) -> Date
```

Parse a date from a date string `dt` using a `DateFormat` object `df`.

source

Base.Dates.Time – Method.

```
Time(h, [mi, s, ms, us, ns]) -> Time
```

Construct a `Time` type by parts. Arguments must be convertible to `Int64`.

source

Base.Dates.Time – Method.

```
Time(period::TimePeriod...) -> Time
```

Construct a `Time` type by `Period` type parts. Arguments may be in any order. `Time` parts not provided will default to the value of `Dates.default(period)`.

source

Base.Dates.Time – Method.

```
Time(f::Function, h, mi=0; step::Period=Second(1), limit::Int=10000)
Time(f::Function, h, mi, s; step::Period=Millisecond(1), limit::Int=10000)
Time(f::Function, h, mi, s, ms; step::Period=Microsecond(1), limit::Int=10000)
Time(f::Function, h, mi, s, ms, us; step::Period=Nanosecond(1), limit::Int=10000)
```

Create a `Time` through the adjuster API. The starting point will be constructed from the provided `h, mi, s, ms, us` arguments, and will be adjusted until `f::Function` returns `true`. The step size in adjusting can be provided manually through the `step` keyword. `limit` provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (in the case that `f::Function` is never satisfied). Note that the default step will adjust to allow for greater precision for the given arguments; i.e. if hour, minute, and second arguments are provided, the default step will be `Millisecond(1)` instead of `Second(1)`.

source

Base.Dates.Time – Method.

```
Time(dt::DateTime) -> Time
```

Converts a `DateTime` to a `Time`. The hour, minute, second, and millisecond parts of the `DateTime` are used to create the new `Time`. Microsecond and nanoseconds are zero by default.

source

Base.Dates.now – Method.

```
now() -> DateTime
```

Returns a `DateTime` corresponding to the user's system time including the system timezone locale.

source

Base.Dates.now – Method.

```
now(::Type{UTC}) -> DateTime
```

Returns a `DateTime` corresponding to the user's system time as UTC/GMT.

source

Base.eps – Function.

```
eps(::DateTime) -> Millisecond
eps(::Date) -> Day
eps(::Time) -> Nanosecond
```

Returns Millisecond(1) for DateTime values, Day(1) for Date values, and Nanosecond(1) for Time values.

source

Funciones de Acceso

Base.Dates.year – Function.

```
year(dt::TimeType) -> Int64
```

The year of a Date or DateTime as an Int64.

source

Base.Dates.month – Function.

```
month(dt::TimeType) -> Int64
```

The month of a Date or DateTime as an Int64.

source

Base.Dates.week – Function.

```
week(dt::TimeType) -> Int64
```

Return the ISO week date of a Date or DateTime as an Int64. Note that the first week of a year is the week that contains the first Thursday of the year which can result in dates prior to January 4th being in the last week of the previous year. For example week(Date(2005,1,1)) is the 53rd week of 2004.

source

Base.Dates.day – Function.

```
day(dt::TimeType) -> Int64
```

The day of month of a Date or DateTime as an Int64.

source

Base.Dates.hour – Function.

```
hour(dt::DateTime) -> Int64
```

The hour of day of a DateTime as an Int64.

source

```
hour(t::Time) -> Int64
```

The hour of a Time as an Int64.

source

Base.Dates.minute – Function.

```
minute(dt::DateTime) -> Int64
```

The minute of a `DateTime` as an `Int64`.

source

```
minute(t::Time) -> Int64
```

The minute of a `Time` as an `Int64`.

source

`Base.Dates.second` – Function.

```
second(dt::DateTime) -> Int64
```

The second of a `DateTime` as an `Int64`.

source

```
second(t::Time) -> Int64
```

The second of a `Time` as an `Int64`.

source

`Base.Dates.millisecond` – Function.

```
millisecond(dt::DateTime) -> Int64
```

The millisecond of a `DateTime` as an `Int64`.

source

```
millisecond(t::Time) -> Int64
```

The millisecond of a `Time` as an `Int64`.

source

`Base.Dates.microsecond` – Function.

```
microsecond(t::Time) -> Int64
```

The microsecond of a `Time` as an `Int64`.

source

`Base.Dates.nanosecond` – Function.

```
nanosecond(t::Time) -> Int64
```

The nanosecond of a `Time` as an `Int64`.

source

`Base.Dates.Year` – Method.

```
Year(v)
```

Construct a `Year` object with the given `v` value. Input must be losslessly convertible to an `Int64`.

source

Base.Dates.Month – Method.

```
Month(v)
```

Construct a Month object with the given v value. Input must be losslessly convertible to an Int64.

source

Base.Dates.Week – Method.

```
Week(v)
```

Construct a Week object with the given v value. Input must be losslessly convertible to an Int64.

source

Base.Dates.Day – Method.

```
Day(v)
```

Construct a Day object with the given v value. Input must be losslessly convertible to an Int64.

source

Base.Dates.Hour – Method.

```
Hour(dt::DateTime) -> Hour
```

The hour part of a DateTime as a Hour.

source

Base.Dates.Minute – Method.

```
Minute(dt::DateTime) -> Minute
```

The minute part of a DateTime as a Minute.

source

Base.Dates.Second – Method.

```
Second(dt::DateTime) -> Second
```

The second part of a DateTime as a Second.

source

Base.Dates.Millisecond – Method.

```
Millisecond(dt::DateTime) -> Millisecond
```

The millisecond part of a DateTime as a Millisecond.

source

Base.Dates.Microsecond – Method.

```
Microsecond(dt::Time) -> Microsecond
```

The microsecond part of a Time as a Microsecond.

source

Base.Dates.Nanosecond – Method.

```
Nanosecond(dt::Time) -> Nanosecond
```

The nanosecond part of a Time as a Nanosecond.

source

Base.Dates.yearmonth – Function.

```
yearmonth(dt::TimeType) -> (Int64, Int64)
```

Simultaneously return the year and month parts of a Date or DateTime.

source

Base.Dates.monthday – Function.

```
monthday(dt::TimeType) -> (Int64, Int64)
```

Simultaneously return the month and day parts of a Date or DateTime.

source

Base.Dates.yearmonthday – Function.

```
yearmonthday(dt::TimeType) -> (Int64, Int64, Int64)
```

Simultaneously return the year, month and day parts of a Date or DateTime.

source

Funciones de Consulta

Base.Dates.dayname – Function.

```
dayname(dt::TimeType; locale="english") -> AbstractString
```

Return the full day name corresponding to the day of the week of the Date or DateTime in the given locale.

source

Base.Dates.dayabbr – Function.

```
dayabbr(dt::TimeType; locale="english") -> AbstractString
```

Return the abbreviated name corresponding to the day of the week of the Date or DateTime in the given locale.

source

Base.Dates.dayofweek – Function.

```
dayofweek(dt::TimeType) -> Int64
```

Returns the day of the week as an Int64 with 1 = Monday, 2 = Tuesday, etc..

source

Base.Dates.dayofmonth – Function.

```
dayofmonth(dt::TimeType) -> Int64
```

The day of month of a `Date` or `DateTime` as an `Int64`.

source

`Base.Dates.dayofweekofmonth` – Function.

```
dayofweekofmonth(dt::TimeType) -> Int
```

For the day of week of `dt`, returns which number it is in `dt`'s month. So if the day of the week of `dt` is Monday, then `1 = First Monday of the month, 2 = Second Monday of the month, etc.` In the range 1:5.

source

`Base.Dates.daysofweekinmonth` – Function.

```
daysofweekinmonth(dt::TimeType) -> Int
```

For the day of week of `dt`, returns the total number of that day of the week in `dt`'s month. Returns 4 or 5. Useful in temporal expressions for specifying the last day of a week in a month by including `dayofweekofmonth(dt) == daysofweekinmonth(dt)` in the adjuster function.

source

`Base.Dates.monthname` – Function.

```
monthname(dt::TimeType; locale="english") -> AbstractString
```

Return the full name of the month of the `Date` or `DateTime` in the given `locale`.

source

`Base.Dates.monthabbr` – Function.

```
monthabbr(dt::TimeType; locale="english") -> AbstractString
```

Return the abbreviated month name of the `Date` or `DateTime` in the given `locale`.

source

`Base.Dates.daysinmonth` – Function.

```
daysinmonth(dt::TimeType) -> Int
```

Returns the number of days in the month of `dt`. Value will be 28, 29, 30, or 31.

source

`Base.Dates.isleapyear` – Function.

```
isleapyear(dt::TimeType) -> Bool
```

Returns `true` if the year of `dt` is a leap year.

source

`Base.Dates.dayofyear` – Function.

```
dayofyear(dt::TimeType) -> Int
```

Returns the day of the year for `dt` with January 1st being day 1.

source

Base.Dates.daysinyear – Function.

```
daysinyear(dt::TimeType) -> Int
```

Returns 366 if the year of `dt` is a leap year, otherwise returns 365.

source

Base.Dates.quarterofyear – Function.

```
quarterofyear(dt::TimeType) -> Int
```

Returns the quarter that `dt` resides in. Range of value is 1:4.

source

Base.Dates.dayofquarter – Function.

```
dayofquarter(dt::TimeType) -> Int
```

Returns the day of the current quarter of `dt`. Range of value is 1:92.

source

Funciones de Ajuste

Base.trunc – Method.

```
trunc(dt::TimeType, ::Type{Period}) -> TimeType
```

Truncates the value of `dt` according to the provided `Period` type. E.g. if `dt` is `1996-01-01T12:30:00`, then `trunc(dt,Day) == 1996-01-01T00:00:00`.

source

Base.Dates.firstdayofweek – Function.

```
firstdayofweek(dt::TimeType) -> TimeType
```

Adjusts `dt` to the Monday of its week.

source

Base.Dates.lastdayofweek – Function.

```
lastdayofweek(dt::TimeType) -> TimeType
```

Adjusts `dt` to the Sunday of its week.

source

Base.Dates.firstdayofmonth – Function.

```
firstdayofmonth(dt::TimeType) -> TimeType
```

Adjusts `dt` to the first day of its month.

source

Base.Dates.lastdayofmonth – Function.

```
lastdayofmonth(dt::TimeType) -> TimeType
```

Adjusts `dt` to the last day of its month.

source

`Base.Dates.firstdayofyear` – Function.

```
firstdayofyear(dt::TimeType) -> TimeType
```

Adjusts `dt` to the first day of its year.

source

`Base.Dates.lastdayofyear` – Function.

```
lastdayofyear(dt::TimeType) -> TimeType
```

Adjusts `dt` to the last day of its year.

source

`Base.Dates.firstdayofquarter` – Function.

```
firstdayofquarter(dt::TimeType) -> TimeType
```

Adjusts `dt` to the first day of its quarter.

source

`Base.Dates.lastdayofquarter` – Function.

```
lastdayofquarter(dt::TimeType) -> TimeType
```

Adjusts `dt` to the last day of its quarter.

source

`Base.Dates.tonext` – Method.

```
tonext(dt::TimeType, dow::Int; same::Bool=false) -> TimeType
```

Adjusts `dt` to the next day of week corresponding to `dow` with `1 = Monday, 2 = Tuesday, etc`. Setting `same=true` allows the current `dt` to be considered as the next `dow`, allowing for no adjustment to occur.

source

`Base.Dates.toprev` – Method.

```
toprev(dt::TimeType, dow::Int; same::Bool=false) -> TimeType
```

Adjusts `dt` to the previous day of week corresponding to `dow` with `1 = Monday, 2 = Tuesday, etc`. Setting `same=true` allows the current `dt` to be considered as the previous `dow`, allowing for no adjustment to occur.

source

`Base.Dates.tofirst` – Function.

```
tofirst(dt::TimeType, dow::Int; of=Month) -> TimeType
```

Adjusts `dt` to the first dow of its month. Alternatively, `of=Year` will adjust to the first dow of the year.

source

Base.Dates.tolast – Function.

```
tolast(dt::TimeType, dow::Int; of=Month) -> TimeType
```

Adjusts `dt` to the last dow of its month. Alternatively, `of=Year` will adjust to the last dow of the year.

source

Base.Dates.tonext – Method.

```
tonext(func::Function, dt::TimeType; step=Day(1), limit=10000, same=false) -> TimeType
```

Adjusts `dt` by iterating at most `limit` iterations by `step` increments until `func` returns `true`. `func` must take a single `TimeType` argument and return a `Bool`. `same` allows `dt` to be considered in satisfying `func`.

source

Base.Dates.toprev – Method.

```
toprev(func::Function, dt::TimeType; step=Day(-1), limit=10000, same=false) -> TimeType
```

Adjusts `dt` by iterating at most `limit` iterations by `step` increments until `func` returns `true`. `func` must take a single `TimeType` argument and return a `Bool`. `same` allows `dt` to be considered in satisfying `func`.

source

Períodos

Base.Dates.Period – Method.

```
Year(v)
Month(v)
Week(v)
Day(v)
Hour(v)
Minute(v)
Second(v)
Millisecond(v)
Microsecond(v)
Nanosecond(v)
```

Construct a `Period` type with the given `v` value. Input must be losslessly convertible to an `Int64`.

source

Base.Dates.CompoundPeriod – Method.

```
CompoundPeriod(periods) -> CompoundPeriod
```

Construct a `CompoundPeriod` from a `Vector` of `Period`s. All `Period`s of the same type will be added together.

Examples

```
julia> Dates.CompoundPeriod(Dates.Hour(12), Dates.Hour(13))
25 hours

julia> Dates.CompoundPeriod(Dates.Hour(-1), Dates.Minute(1))
-1 hour, 1 minute

julia> Dates.CompoundPeriod(Dates.Month(1), Dates.Week(-2))
1 month, -2 weeks

julia> Dates.CompoundPeriod(Dates.Minute(50000))
50000 minutes
```

source

Base.Dates.default – Function.

```
default(p::Period) -> Period
```

Returns a sensible "default" value for the input Period by returning `T(1)` for Year, Month, and Day, and `T(0)` for Hour, Minute, Second, and Millisecond.

source

Funciones de Redondeo

Los valores `Date` y `DateTime` pueden ser redondeados a una precisión especificada (por ejemplo, 1 mes o 15 minutos) con `floor`, `ceil`, o `round`.

Base.floor – Method.

```
floor(dt::TimeType, p::Period) -> TimeType
```

Returns the nearest `Date` or `DateTime` less than or equal to `dt` at resolution `p`.

For convenience, `p` may be a type instead of a value: `floor(dt, Dates.Hour)` is a shortcut for `floor(dt, Dates.Hour(1))`.

```
julia> floor(Date(1985, 8, 16), Dates.Month)
1985-08-01

julia> floor(DateTime(2013, 2, 13, 0, 31, 20), Dates.Minute(15))
2013-02-13T00:30:00

julia> floor(DateTime(2016, 8, 6, 12, 0, 0), Dates.Day)
2016-08-06T00:00:00
```

source

Base.ceil – Method.

```
ceil(dt::TimeType, p::Period) -> TimeType
```

Returns the nearest `Date` or `DateTime` greater than or equal to `dt` at resolution `p`.

For convenience, `p` may be a type instead of a value: `ceil(dt, Dates.Hour)` is a shortcut for `ceil(dt, Dates.Hour(1))`.

```
julia> ceil(Date(1985, 8, 16), Dates.Month)
1985-09-01

julia> ceil(DateTime(2013, 2, 13, 0, 31, 20), Dates.Minute(15))
2013-02-13T00:45:00

julia> ceil(DateTime(2016, 8, 6, 12, 0, 0), Dates.Day)
2016-08-07T00:00:00
```

source

Base.round – Method.

```
round(dt::TimeType, p::Period, [r::RoundingMode]) -> TimeType
```

Returns the Date or DateTime nearest to dt at resolution p. By default (RoundNearestTiesUp), ties (e.g., rounding 9:30 to the nearest hour) will be rounded up.

For convenience, p may be a type instead of a value: round(dt, Dates.Hour) is a shortcut for round(dt, Dates.Hour(1)).

```
julia> round(Date(1985, 8, 16), Dates.Month)
1985-08-01

julia> round(DateTime(2013, 2, 13, 0, 31, 20), Dates.Minute(15))
2013-02-13T00:30:00

julia> round(DateTime(2016, 8, 6, 12, 0, 0), Dates.Day)
2016-08-07T00:00:00
```

Valid rounding modes for round(::TimeType, ::Period, ::RoundingMode) are RoundNearestTiesUp (default), RoundDown (floor), and RoundUp (ceil).

source

Las siguientes funciones no están exportadas:

Base.Dates.floorceil – Function.

```
floorceil(dt::TimeType, p::Period) -> (TimeType, TimeType)
```

Simultaneously return the floor and ceil of a Date or DateTime at resolution p. More efficient than calling both floor and ceil individually.

source

Base.Dates.epochdays2date – Function.

```
epochdays2date(days) -> Date
```

Takes the number of days since the rounding epoch (0000-01-01T00:00:00) and returns the corresponding Date.

source

Base.Dates.epochms2datetime – Function.

```
epochms2datetime(milliseconds) -> DateTime
```

Takes the number of milliseconds since the rounding epoch (`0000-01-01T00:00:00`) and returns the corresponding `DateTime`.

source

`Base.Dates.date2epochdays` – Function.

```
date2epochdays(dt::Date) -> Int64
```

Takes the given `Date` and returns the number of days since the rounding epoch (`0000-01-01T00:00:00`) as an `Int64`.

source

`Base.Dates.datetime2epochms` – Function.

```
datetime2epochms(dt::DateTime) -> Int64
```

Takes the given `DateTime` and returns the number of milliseconds since the rounding epoch (`0000-01-01T00:00:00`) as an `Int64`.

source

Funciones de Conversión

`Base.Dates.today` – Function.

```
today() -> Date
```

Returns the date portion of `now()`.

source

`Base.Dates.unix2datetime` – Function.

```
unix2datetime(x) -> DateTime
```

Takes the number of seconds since unix epoch `1970-01-01T00:00:00` and converts to the corresponding `DateTime`.

source

`Base.Dates.datetime2unix` – Function.

```
datetime2unix(dt::DateTime) -> Float64
```

Takes the given `DateTime` and returns the number of seconds since the unix epoch `1970-01-01T00:00:00` as a `Float64`.

source

`Base.Dates.julian2datetime` – Function.

```
julian2datetime(julian_days) -> DateTime
```

Takes the number of Julian calendar days since epoch `-4713-11-24T12:00:00` and returns the corresponding `DateTime`.

source

`Base.Dates.datetime2julian` – Function.

```
datetime2julian(dt::DateTime) -> Float64
```

Takes the given `DateTime` and returns the number of Julian calendar days since the julian epoch `-4713-11-24T12:00:00` as a `Float64`.

source

`Base.Dates.rata2datetime` – Function.

```
rata2datetime(days) -> DateTime
```

Takes the number of Rata Die days since epoch `0000-12-31T00:00:00` and returns the corresponding `DateTime`.

source

`Base.Dates.datetime2rata` – Function.

```
datetime2rata(dt::TimeType) -> Int64
```

Returns the number of Rata Die days since epoch from the given `Date` or `DateTime`.

source

Constantes

Días de la Semana:

Variable	Abbr.	Value (Int)
`Monday`	`Mon`	1
`Tuesday`	`Tue`	2
`Wednesday`	`Wed`	3
`Thursday`	`Thu`	4
`Friday`	`Fri`	5
`Saturday`	`Sat`	6
`Sunday`	`Sun`	7

Meses del Año:

Variable	Abbr.	Value (Int)
`January`	`Jan`	1
`February`	`Feb`	2
`March`	`Mar`	3
`April`	`Apr`	4
`May`	`May`	5
`June`	`Jun`	6
`July`	`Jul`	7
`August`	`Aug`	8
`September`	`Sep`	9
`October`	`Oct`	10
`November`	`Nov`	11
`December`	`Dec`	12

Chapter 60

Utilidades para Iteración

`Base.Iterators.zip` – Function.

```
zip(iters...)
```

For a set of iterable objects, returns an iterable of tuples, where the `i`th tuple contains the `i`th component of each input iterable.

Note that `zip` is its own inverse: `collect(zip(zip(a...)...)) == collect(a)`.

Example

```
julia> a = 1:5
1:5

julia> b = ["e","d","b","c","a"]
5-element Array{String,1}:
 "e"
 "d"
 "b"
 "c"
 "a"

julia> c = zip(a,b)
Base.Iterators.Zip2{UnitRange{Int64},Array{String,1}}(1:5, String["e", "d", "b", "c", "a"])

julia> length(c)
5

julia> first(c)
(1, "e")
```

source

`Base.Iterators.enumerate` – Function.

```
enumerate(iter)
```

An iterator that yields `(i, x)` where `i` is a counter starting at 1, and `x` is the `i`th value from the given iterator. It's useful when you need not only the values `x` over which you are iterating, but also the number of iterations so far. Note that `i` may not be valid for indexing `iter`; it's also possible that `x != iter[i]`, if `iter` has indices

that do not start at 1. See the `enumerate(IndexLinear(), iter)` method if you want to ensure that `i` is an index.

Example

```
julia> a = ["a", "b", "c"];

julia> for (index, value) in enumerate(a)

           println("$index $value")

       end
1 a
2 b
3 c
```

source

```
enumerate(IndexLinear(), A)
enumerate(IndexCartesian(), A)
enumerate(IndexStyle(A), A)
```

An iterator that accesses each element of the array `A`, returning `(i, x)`, where `i` is the index for the element and `x = A[i]`. This is similar to `enumerate(A)`, except `i` will always be a valid index for `A`.

Specifying `IndexLinear()` ensures that `i` will be an integer; specifying `IndexCartesian()` ensures that `i` will be a `CartesianIndex`; specifying `IndexStyle(A)` chooses whichever has been defined as the native indexing style for array `A`.

Examples

```
julia> A = ["a" "d"; "b" "e"; "c" "f"];

julia> for (index, value) in enumerate(IndexStyle(A), A)

           println("$index $value")

       end
1 a
2 b
3 c
4 d
5 e
6 f

julia> S = view(A, 1:2, :);

julia> for (index, value) in enumerate(IndexStyle(S), S)

           println("$index $value")

       end
CartesianIndex{2}((1, 1)) a
CartesianIndex{2}((2, 1)) b
CartesianIndex{2}((1, 2)) d
CartesianIndex{2}((2, 2)) e
```

Note that `enumerate(A)` returns `i` as a *counter* (always starting at 1), whereas `enumerate(IndexLinear(), A)` returns `i` as an *index* (starting at the first linear index of `A`, which may or may not be 1).

See also: `IndexStyle`, `indices`.

source

`Base.Iterators.rest` – Function.

```
rest(iter, state)
```

An iterator that yields the same elements as `iter`, but starting at the given `state`.

source

`Base.Iterators.countfrom` – Function.

```
countfrom(start=1, step=1)
```

An iterator that counts forever, starting at `start` and incrementing by `step`.

source

`Base.Iterators.take` – Function.

```
take(iter, n)
```

An iterator that generates at most the first n elements of `iter`.

Example

```
julia> a = 1:2:11
1:2:11

julia> collect(a)
6-element Array{Int64,1}:
  1
  3
  5
  7
  9
 11

julia> collect(Iterators.take(a,3))
3-element Array{Int64,1}:
 1
 3
 5
```

source

`Base.Iterators.drop` – Function.

```
drop(iter, n)
```

An iterator that generates all but the first n elements of `iter`.

Example

```
julia> a = 1:2:11
1:2:11

julia> collect(a)
6-element Array{Int64,1}:
  1
  3
  5
  7
  9
 11

julia> collect(Iterators.drop(a,4))
2-element Array{Int64,1}:
  9
 11
```

source

Base.Iterators.cycle – Function.

```
cycle(iter)
```

An iterator that cycles through `iter` forever.

source

Base.Iterators.repeated – Function.

```
repeated(x[, n::Int])
```

An iterator that generates the value `x` forever. If `n` is specified, generates `x` that many times (equivalent to `take(repeated(x), n)`).

Example

```
julia> a = Iterators.repeated([1 2], 4);

julia> collect(a)
4-element Array{Array{Int64,2},1}:
 [1 2]
 [1 2]
 [1 2]
 [1 2]
```

source

Base.Iterators.product – Function.

```
product(iters...)
```

Returns an iterator over the product of several iterators. Each generated element is a tuple whose `i`th element comes from the `i`th argument iterator. The first iterator changes the fastest. Example:

Example

```
julia> collect(Iterators.product(1:2,3:5))
2×3 Array{Tuple{Int64,Int64},2}:
 (1, 3)  (1, 4)  (1, 5)
 (2, 3)  (2, 4)  (2, 5)
```

source

Base.Iterators.flatten – Function.

```
flatten(iter)
```

Given an iterator that yields iterators, return an iterator that yields the elements of those iterators. Put differently, the elements of the argument iterator are concatenated.

Example

```
julia> collect(Iterators.flatten((1:2, 8:9)))
4-element Array{Int64,1}:
 1
 2
 8
 9
```

source

Base.Iterators.partition – Function.

```
partition(collection, n)
```

Iterate over a collection n elements at a time.

Example

```
julia> collect(Iterators.partition([1,2,3,4,5], 2))
3-element Array{Array{Int64,1},1}:
 [1, 2]
 [3, 4]
 [5]
```

source

Chapter 61

Haciendo Pruebas Unitarias

61.1 Probando Julia Base

Julia está en rápido desarrollo y cuenta con un amplio conjunto de pruebas para verificar su funcionalidad en múltiples plataformas. Si compila Julia desde el origen, puede ejecutar este conjunto de pruebas con `make test`. En una instalación binaria, puede ejecutar el conjunto de pruebas utilizando `Base.runtests()`.

`Base.runtests` – Function.

```
runtests([tests=["all"] [, numcores=ceil(Int, Sys.CPU_CORES / 2) ]])
```

Run the Julia unit tests listed in `tests`, which can be either a string or an array of strings, using `numcores` processors. (not exported)

source

61.2 Pruebas Unitarias Básicas

El módulo `Base.Test` proporciona una funcionalidad simple de *realización de pruebas unitarias*. Las pruebas unitarias son una forma de ver si su código es correcto al verificar que los resultados sean los esperados. Puede ser útil asegurarse de que su código aún funcione después de realizar los cambios, y se puede usar al desarrollar como una forma de especificar los comportamientos que su código debería tener cuando se complete.

Se pueden realizar pruebas unitarias simples con las macros `@test()` y `@test_throws()`:

`Base.Test.@test` – Macro.

```
@test ex
@test f(args...) key=val ...
```

Tests that the expression `ex` evaluates to `true`. Returns a `Pass` `Result` if it does, a `Fail` `Result` if it is `false`, and an `Error` `Result` if it could not be evaluated.

The `@test f(args...) key=val...` form is equivalent to writing `@test f(args..., key=val...)` which can be useful when the expression is a call using infix syntax such as approximate comparisons:

```
@test a ≈ b atol=ε
```

This is equivalent to the uglier test `@test ≈(a, b, atol=ε)`. It is an error to supply more than one expression unless the first is a call expression and the rest are assignments (`k=v`).

source

`Base.Test.@test_throws` – Macro.

```
@test_throws extype ex
```

Tests that the expression `ex` throws an exception of type `extype`. Note that `@test_throws` does not support a trailing keyword form.

source

Por ejemplo, supongamos que queremos comprobar que nuestra nueva función `foo(x)` funciona como se esperaba:

```
julia> using Base.Test

julia> foo(x) = length(x)^2
foo (función genérica con 1 método)
```

Si la condición es cierta, se devuelve un `Pass`:

```
julia> @test foo("bar") == 9
Test Passed

julia> @test foo("fizz") >= 10
Test Passed
```

Si la condición es falsa, se devuelve un `Fail` y se lanza una excepción:

```
julia> @test foo("f") == 20
Test Failed
  Expression: foo("f") == 20
   Evaluated: 1 == 20
ERROR: There was an error during testing
```

Si la condición no pudo ser evaluada porque se lanzó una excepción, lo que ocurre en este caso porque `length()` no está definido para símbolos, se devuelve un objeto `Error` y se lanza una excepción:

```
julia> @test foo(:cat) == 1
Error During Test
  Test threw an exception of type MethodError
  Expression: foo(:cat) == 1
  MethodError: no method matching length(::Symbol)
  Closest candidates are:
    length(::SimpleVector) at essentials.jl:256
    length(::Base.MethodList) at reflection.jl:521
    length(::MethodTable) at reflection.jl:597
    ...
  Stacktrace:
   [...]
ERROR: There was an error during testing
```

Si esperamos que al evaluar una expresión *debería* lanzarse una excepción, entonces podemos usar `@test_throws()` para comprobar que esto es lo que ocurre:

```
julia> @test_throws MethodError foo(:cat)
Test Passed
      Thrown: MethodError
```

61.3 Trabajando con Conjuntos de Test

Normalmente, se utiliza una gran cantidad de pruebas para garantizar que las funciones trabajan correctamente sobre distintas entradas. En el caso de que una prueba falle, el comportamiento predeterminado es lanzar una excepción de inmediato. Sin embargo, normalmente es preferible ejecutar el resto de las pruebas primero para obtener una mejor idea de cuántos errores hay en el código que se prueba.

La macro `@testset()` se puede usar para agrupar las pruebas en *conjuntos*. En un conjunto de pruebas, se ejecutarán varias y al final de su realización se imprimirá un resumen. Si alguna de las pruebas falla o no se puede evaluar debido a un error, el conjunto de prueba arrojará una `TestSetException`.

`Base.Test.@testset` – Macro.

```
@testset [CustomTestSet] [option=val  ...] ["description"] begin ... end
@testset [CustomTestSet] [option=val  ...] ["description $v"] for v in (...) ... end
@testset [CustomTestSet] [option=val  ...] ["description $v, $w"] for v in (...), w in (...)
     ... end
```

Starts a new test set, or multiple test sets if a `for` loop is provided.

If no custom testset type is given it defaults to creating a `DefaultTestSet`. `DefaultTestSet` records all the results and, if there are any `Fails` or `Errors`, throws an exception at the end of the top-level (non-nested) test set, along with a summary of the test results.

Any custom testset type (subtype of `AbstractTestSet`) can be given and it will also be used for any nested `@testset` invocations. The given options are only applied to the test set where they are given. The default test set type does not take any options.

The description string accepts interpolation from the loop indices. If no description is provided, one is constructed based on the variables.

By default the `@testset` macro will return the testset object itself, though this behavior can be customized in other testset types. If a `for` loop is used then the macro collects and returns a list of the return values of the `finish` method, which by default will return a list of the testset objects used in each iteration.

`source`

Podemos poner nuestras pruebas para la función `foo(x)` en un conjuntos de pruebas:

```
julia> @testset "Foo Tests" begin
           @test foo("a")   == 1
           @test foo("ab")  == 4
           @test foo("abc") == 9
       end;
Test Summary: | Pass  Total
Foo Tests     |    3      3
```

Los conjuntos de pruebas también pueden anidarse:

```
julia> @testset "Foo Tests" begin
           @testset "Animals" begin
               @test foo("cat") == 9
               @test foo("dog") == foo("cat")
           end
           @testset "Arrays $i" for i in 1:3
               @test foo(zeros(i)) == i^2
               @test foo(ones(i)) == i^2
           end
       end;
```

```
Test Summary: | Pass  Total
Foo Tests     |    8      8
```

En el caso de que un conjunto de pruebas anidado no tenga fallos, como pasa aquí, se ocultará en el resumen. Si tenemos una prueba que falle, sólo se mostrarán los detalles para este conjunto de pruebas que ha fallado:

```
julia> @testset "Foo Tests" begin

           @testset "Animals" begin

               @testset "Felines" begin

                   @test foo("cat") == 9

               end

               @testset "Canines" begin

                   @test foo("dog") == 9

               end

           end

           @testset "Arrays" begin

               @test foo(zeros(2)) == 4

               @test foo(ones(4)) == 15

           end

       end

Arrays: Test Failed
  Expression: foo(ones(4)) == 15
   Evaluated: 16 == 15
Stacktrace:
    [...]
Test Summary: | Pass  Fail  Total
Foo Tests     |    3     1      4
  Animals     |    2            2
  Arrays      |    1     1      2
ERROR: Some tests did not pass: 3 passed, 1 failed, 0 errored, 0 broken.
```

61.4 Otras Macros para Pruebas

Como los cálculos sobre valores en punto flotane pueden ser imprecisos, podemos realizar comprobaciones de igualdad aproximada usando `@test a ≈ b` (donde ≈, se obtiene mediante terminación con tabulador de `\approx`, es la función `isapprox()`) o usar directamente `isapprox()`.

```
julia> @test 1 ≈ 0.999999999
Test Passed
```

```
julia> @test 1 ≈ 0.999999
Test Failed
  Expression: 1 ≈ 0.999999
   Evaluated: 1 ≈ 0.999999
ERROR: There was an error during testing
```

Base.Test.@inferred – Macro.

```
@inferred f(x)
```

Tests that the call expression `f(x)` returns a value of the same type inferred by the compiler. It is useful to check for type stability.

`f(x)` can be any call expression. Returns the result of `f(x)` if the types match, and an `Error Result` if it finds different types.

```
julia> using Base.Test

julia> f(a,b,c) = b > 1 ? 1 : 1.0
f (generic function with 1 method)

julia> typeof(f(1,2,3))
Int64

julia> @code_warntype f(1,2,3)
Variables:
  #self# <optimized out>
  a <optimized out>
  b::Int64
  c <optimized out>

Body:
  begin
  end::UNION{FLOAT64, INT64}


      unless (Base.slt_int)(1, b::Int64)::Bool goto 3

      return 1

      3:

      return 1.0
julia> @inferred f(1,2,3)
ERROR: return type Int64 does not match inferred return type Union{Float64, Int64}
Stacktrace:
 [1] error(::String) at ./error.jl:21

julia> @inferred max(1,2)
2
```

source

Base.Test.@test_warn – Macro.

```
@test_warn msg expr
```

Test whether evaluating `expr` results in STDERR output that contains the `msg` string or matches the `msg` regular expression. If `msg` is a boolean function, tests whether `msg(output)` returns `true`. If `msg` is a tuple or array, checks that the error output contains/matches each item in `msg`. Returns the result of evaluating `expr`.

See also `@test_nowarn` to check for the absence of error output.

source

`Base.Test.@test_nowarn` – Macro.

```
@test_nowarn expr
```

Test whether evaluating `expr` results in empty STDERR output (no warnings or other messages). Returns the result of evaluating `expr`.

source

61.5 Pruebas Rotas

Si una prueba falla consistentemente, puede ser cambiada para utilizar la macro `@test_broken()`. Esto denotará la prueba como Rota (`Broken`) si la prueba continúa fallando y alerta al usuario a través de un `Error` si la prueba tiene éxito.

`Base.Test.@test_broken` – Macro.

```
@test_broken ex
@test_broken f(args...) key=val ...
```

Indicates a test that should pass but currently consistently fails. Tests that the expression `ex` evaluates to `false` or causes an exception. Returns a `Broken Result` if it does, or an `Error Result` if the expression evaluates to `true`.

The `@test_broken f(args...) key=val...` form works as for the `@test` macro.

source

`@test_skip()` está también disponible para obviar una prueba sin evaluación, pero contabilizarla en el informe del conjunto de pruebas. La prueba no se ejecutará pero dará un `Broken Result`.

`Base.Test.@test_skip` – Macro.

```
@test_skip ex
@test_skip f(args...) key=val ...
```

Marks a test that should not be executed but should be included in test summary reporting as `Broken`. This can be useful for tests that intermittently fail, or tests of not-yet-implemented functionality.

The `@test_skip f(args...) key=val...` form works as for the `@test` macro.

source

61.6 Creando Tipos `AbstractTestSet` Personalizados

Los paquetes pueden crear sus propios subtipos `AbstractTestSet` implementando los métodos `record` y `finish`. El subtipo debe tener un constructor de un argumento que tome una cadena de descripción, con todas las opciones pasadas como argumentos de tipo keyword.

`Base.Test.record` – Function.

```
record(ts::AbstractTestSet, res::Result)
```

Record a result to a testset. This function is called by the `@testset` infrastructure each time a contained `@test` macro completes, and is given the test result (which could be an `Error`). This will also be called with an `Error` if an exception is thrown inside the test block but outside of a `@test` context.

source

`Base.Test.finish` – Function.

```
finish(ts::AbstractTestSet)
```

Do any final processing necessary for the given testset. This is called by the `@testset` infrastructure after a test block executes. One common use for this function is to record the testset to the parent's results list, using `get_testset`.

source

`Base.Test` asume la responsabilidad de mantener una pila de conjuntos de pruebas anidados a medida que se ejecutan, pero cualquier acumulación de resultados es responsabilidad del subtipo`AbstractTestSet`. Puede acceder a esta pila con los métodos `get_testset` y`get_testset_depth`. Tenga en cuenta que estas funciones no se exportan.

`Base.Test.get_testset` – Function.

```
get_testset()
```

Retrieve the active test set from the task's local storage. If no test set is active, use the fallback default test set.

source

`Base.Test.get_testset_depth` – Function.

```
get_testset_depth()
```

Returns the number of active test sets, not including the defaut test set

source

`Base.Test` también se asegura de que las invocaciones `@testset` anidadas utilicen el mismo subtipo `AbstractTestSet` que sus padres, a menos que se establezca explícitamente. No propaga ninguna propiedad del conjunto de pruebas. El comportamiento de herencia de opciones se puede implementar mediante paquetes que usan la infraestructura de pila que proporciona `Base.Test`.

La definición de un subtipo básico de 'AbstractTestSet' podría verse así:

```
import Base.Test: record, finish
using Base.Test: AbstractTestSet, Result, Pass, Fail, Error
using Base.Test: get_testset_depth, get_testset
struct CustomTestSet <: Base.Test.AbstractTestSet
    description::AbstractString
    foo::Int
    results::Vector
    # constructor takes a description string and options keyword arguments
    CustomTestSet(desc; foo=1) = new(desc, foo, [])
end

record(ts::CustomTestSet, child::AbstractTestSet) = push!(ts.results, child)
record(ts::CustomTestSet, res::Result) = push!(ts.results, res)
```

```
function finish(ts::CustomTestSet)
    # just record if we're not the top-level parent
    if get_testset_depth() > 0
        record(get_testset(), ts)
    end
    ts
end
```

Y usar este conjunto de prueba tiene el siguiente aspecto:

```
@testset CustomTestSet foo=4 "custom testset inner 2" begin
    # this testset should inherit the type, but not the argument.
    @testset "custom testset inner" begin
        @test true
    end
end
```

Chapter 62

Interfaz C

`ccall` – Keyword.

```
ccall((symbol, library) or function_pointer, ReturnType, (ArgumentType1, ...), ArgumentValue1
    , ...)
```

Call function in C-exported shared library, specified by `(function name, library)` tuple, where each component is a string or symbol.

Note that the argument type tuple must be a literal tuple, and not a tuple-valued variable or expression. Alternatively, `ccall` may also be used to call a function pointer, such as one returned by `dlsym`.

Each `ArgumentValue` to the `ccall` will be converted to the corresponding `ArgumentType`, by automatic insertion of calls to `unsafe_convert(ArgumentType, cconvert(ArgumentType, ArgumentValue))`. (See also the documentation for each of these functions for further details.) In most cases, this simply results in a call to `convert(ArgumentType, ArgumentValue)`.

source

`Core.Intrinsics.cglobal` – Function.

```
cglobal((symbol, library) [, type=Void])
```

Obtain a pointer to a global variable in a C-exported shared library, specified exactly as in `ccall`. Returns a `Ptr{Type}`, defaulting to `Ptr{Void}` if no `Type` argument is supplied. The values can be read or written by `unsafe_load` or `unsafe_store!`, respectively.

source

`Base.cfunction` – Function.

```
cfunction(function::Function, ReturnType::Type, ArgumentTypes::Type)
```

Generate C-callable function pointer from Julia function. Type annotation of the return value in the callback function is a must for situations where Julia cannot infer the return type automatically.

Examples

```
julia> function foo(x::Int, y::Int)

           return x + y

       end
```

```
julia> cfunction(foo, Int, Tuple{Int,Int})
Ptr{Void} @0x000000001b82fcd0
```

source

Base.unsafe_convert – Function.

```
unsafe_convert(T,x)
```

Convert x to a value of type T

In cases where convert would need to take a Julia object and turn it into a Ptr, this function should be used to define and perform that conversion.

Be careful to ensure that a Julia reference to x exists as long as the result of this function will be used. Accordingly, the argument x to this function should never be an expression, only a variable name or field reference. For example, x=a.b.c is acceptable, but x=[a,b,c] is not.

The unsafe prefix on this function indicates that using the result of this function after the x argument to this function is no longer accessible to the program may cause undefined behavior, including program corruption or segfaults, at any later time.

source

Base.cconvert – Function.

```
cconvert(T,x)
```

Convert x to a value of type T, typically by calling convert(T,x)

In cases where x cannot be safely converted to T, unlike convert, cconvert may return an object of a type different from T, which however is suitable for unsafe_convert to handle.

Neither convert nor cconvert should take a Julia object and turn it into a Ptr.

source

Base.unsafe_load – Function.

```
unsafe_load(p::Ptr{T}, i::Integer=1)
```

Load a value of type T from the address of the ith element (1-indexed) starting at p. This is equivalent to the C expression p[i-1].

The unsafe prefix on this function indicates that no validation is performed on the pointer p to ensure that it is valid. Incorrect usage may segfault your program or return garbage answers, in the same manner as C.

source

Base.unsafe_store! – Function.

```
unsafe_store!(p::Ptr{T}, x, i::Integer=1)
```

Store a value of type T to the address of the ith element (1-indexed) starting at p. This is equivalent to the C expression p[i-1] = x.

The unsafe prefix on this function indicates that no validation is performed on the pointer p to ensure that it is valid. Incorrect usage may corrupt or segfault your program, in the same manner as C.

source

Base.unsafe_copy! – Method.

```
unsafe_copy!(dest::Ptr{T}, src::Ptr{T}, N)
```

Copy N elements from a source pointer to a destination, with no checking. The size of an element is determined by the type of the pointers.

The unsafe prefix on this function indicates that no validation is performed on the pointers dest and src to ensure that they are valid. Incorrect usage may corrupt or segfault your program, in the same manner as C.

source

Base.unsafe_copy! – Method.

```
unsafe_copy!(dest::Array, do, src::Array, so, N)
```

Copy N elements from a source array to a destination, starting at offset so in the source and do in the destination (1-indexed).

The unsafe prefix on this function indicates that no validation is performed to ensure that N is inbounds on either array. Incorrect usage may corrupt or segfault your program, in the same manner as C.

source

Base.copy! – Method.

```
copy!(dest, src) -> dest
```

Copy all elements from collection src to array dest.

source

Base.copy! – Method.

```
copy!(dest, do, src, so, N)
```

Copy N elements from collection src starting at offset so, to array dest starting at offset do. Returns dest.

source

Base.pointer – Function.

```
pointer(array [, index])
```

Get the native address of an array or string element. Be careful to ensure that a Julia reference to a exists as long as this pointer will be used. This function is "unsafe" like unsafe_convert.

Calling Ref(array[, index]) is generally preferable to this function.

source

Base.unsafe_wrap – Method.

```
unsafe_wrap(Array, pointer::Ptr{T}, dims, own=false)
```

Wrap a Julia Array object around the data at the address given by pointer, without making a copy. The pointer element type T determines the array element type. dims is either an integer (for a 1d array) or a tuple of the array dimensions. own optionally specifies whether Julia should take ownership of the memory, calling free on the pointer when the array is no longer referenced.

This function is labelled "unsafe" because it will crash if pointer is not a valid memory address to data of the requested length.

source

Base.pointer_from_objref – Function.

```
pointer_from_objref(x)
```

Get the memory address of a Julia object as a Ptr. The existence of the resulting Ptr will not protect the object from garbage collection, so you must ensure that the object remains referenced for the whole time that the Ptr will be used.

source

Base.unsafe_pointer_to_objref – Function.

```
unsafe_pointer_to_objref(p::Ptr)
```

Convert a Ptr to an object reference. Assumes the pointer refers to a valid heap-allocated Julia object. If this is not the case, undefined behavior results, hence this function is considered ”unsafe” and should be used with care.

source

Base.disable_sigint – Function.

```
disable_sigint(f::Function)
```

Disable Ctrl-C handler during execution of a function on the current task, for calling external code that may call julia code that is not interrupt safe. Intended to be called using do block syntax as follows:

```
disable_sigint() do
    # interrupt-unsafe code
    ...
end
```

This is not needed on worker threads (Threads.threadid() != 1) since the InterruptException will only be delivered to the master thread. External functions that do not call julia code or julia runtime automatically disable sigint during their execution.

source

Base.reenable_sigint – Function.

```
reenable_sigint(f::Function)
```

Re-enable Ctrl-C handler during execution of a function. Temporarily reverses the effect of disable_sigint.

source

Base.systemerror – Function.

```
systemerror(sysfunc, iftrue)
```

Raises a SystemError for errno with the descriptive string sysfunc if iftrue is true

source

Core.Ptr – Type.

```
Ptr{T}
```

A memory address referring to data of type T. However, there is no guarantee that the memory is actually valid, or that it actually represents data of the specified type.

source

Core.Ref – Type.

```
Ref{T}
```

An object that safely references data of type T. This type is guaranteed to point to valid, Julia-allocated memory of the correct type. The underlying data is protected from freeing by the garbage collector as long as the `Ref` itself is referenced.

When passed as a `ccall` argument (either as a `Ptr` or `Ref` type), a `Ref` object will be converted to a native pointer to the data it references.

There is no invalid (NULL) `Ref`.

source

Base.Cchar – Type.

```
Cchar
```

Equivalent to the native `char` c-type.

source

Base.Cuchar – Type.

```
Cuchar
```

Equivalent to the native `unsigned char` c-type (UInt8).

source

Base.Cshort – Type.

```
Cshort
```

Equivalent to the native `signed short` c-type (Int16).

source

Base.Cushort – Type.

```
Cushort
```

Equivalent to the native `unsigned short` c-type (UInt16).

source

Base.Cint – Type.

```
Cint
```

Equivalent to the native `signed int` c-type (Int32).

source

Base.Cuint – Type.

```
Cuint
```

Equivalent to the native `unsigned int` c-type (UInt32).

source

`Base.Clong` – Type.

```
Clong
```

Equivalent to the native `signed long` c-type.

source

`Base.Culong` – Type.

```
Culong
```

Equivalent to the native `unsigned long` c-type.

source

`Base.Clonglong` – Type.

```
Clonglong
```

Equivalent to the native `signed long long` c-type (`Int64`).

source

`Base.Culonglong` – Type.

```
Culonglong
```

Equivalent to the native `unsigned long long` c-type (`UInt64`).

source

`Base.Cintmax_t` – Type.

```
Cintmax_t
```

Equivalent to the native `intmax_t` c-type (`Int64`).

source

`Base.Cuintmax_t` – Type.

```
Cuintmax_t
```

Equivalent to the native `uintmax_t` c-type (`UInt64`).

source

`Base.Csize_t` – Type.

```
Csize_t
```

Equivalent to the native `size_t` c-type (`UInt`).

source

`Base.Cssize_t` – Type.

```
Cssize_t
```

Equivalent to the native `ssize_t` c-type.

source

`Base.Cptrdiff_t` – Type.

```
Cptrdiff_t
```

Equivalent to the native `ptrdiff_t` c-type (`Int`).

source

`Base.Cwchar_t` – Type.

```
Cwchar_t
```

Equivalent to the native `wchar_t` c-type (`Int32`).

source

`Base.Cfloat` – Type.

```
Cfloat
```

Equivalent to the native `float` c-type (`Float32`).

source

`Base.Cdouble` – Type.

```
Cdouble
```

Equivalent to the native `double` c-type (`Float64`).

source

Chapter 63

Interfaz LLVM

`Core.Intrinsics.llvmcall` – Function.

```
llvmcall(IR::String, ReturnType, (ArgumentType1, ...), ArgumentValue1, ...)
llvmcall((declarations::String, IR::String), ReturnType, (ArgumentType1, ...), ArgumentValue1
    , ...)
```

Call LLVM IR string in the first argument. Similar to an LLVM function `define` block, arguments are available as consecutive unnamed SSA variables (%0, %1, etc.).

The optional declarations string contains external functions declarations that are necessary for llvm to compile the IR string. Multiple declarations can be passed in by separating them with line breaks.

Note that the argument type tuple must be a literal tuple, and not a tuple-valued variable or expression.

Each `ArgumentValue` to `llvmcall` will be converted to the corresponding `ArgumentType`, by automatic insertion of calls to `unsafe_convert(ArgumentType, cconvert(ArgumentType, ArgumentValue))`. (see also the documentation for each of these functions for further details). In most cases, this simply results in a call to `convert(ArgumentType, ArgumentValue)`.

See `test/llvmcall.jl` for usage examples.

source

Chapter 64

Librería Estándar C

`Base.Libc.malloc` – Function.

```
malloc(size::Integer) -> Ptr{Void}
```

Call `malloc` from the C standard library.

source

`Base.Libc.calloc` – Function.

```
calloc(num::Integer, size::Integer) -> Ptr{Void}
```

Call `calloc` from the C standard library.

source

`Base.Libc.realloc` – Function.

```
realloc(addr::Ptr, size::Integer) -> Ptr{Void}
```

Call `realloc` from the C standard library.

See warning in the documentation for `free` regarding only using this on memory originally obtained from `malloc`.

source

`Base.Libc.free` – Function.

```
free(addr::Ptr)
```

Call `free` from the C standard library. Only use this on memory obtained from `malloc`, not on pointers retrieved from other C libraries. `Ptr` objects obtained from C libraries should be freed by the free functions defined in that library, to avoid assertion failures if multiple `libc` libraries exist on the system.

source

`Base.Libc.errno` – Function.

```
errno([code])
```

Get the value of the C library's `errno`. If an argument is specified, it is used to set the value of `errno`.

The value of `errno` is only valid immediately after a `ccall` to a C library routine that sets it. Specifically, you cannot call `errno` at the next prompt in a REPL, because lots of code is executed between prompts.

source

Base.Libc.strerror – Function.

```
strerror(n=errno())
```

Convert a system call error code to a descriptive string

source

Base.Libc.GetLastError – Function.

```
GetLastError()
```

Call the Win32 `GetLastError` function [only available on Windows].

source

Base.Libc.FormatMessage – Function.

```
FormatMessage(n=GetLastError())
```

Convert a Win32 system call error code to a descriptive string [only available on Windows].

source

Base.Libc.time – Method.

```
time(t::TmStruct)
```

Converts a `TmStruct` struct to a number of seconds since the epoch.

source

Base.Libc.strftime – Function.

```
strftime([format], time)
```

Convert time, given as a number of seconds since the epoch or a `TmStruct`, to a formatted string using the given format. Supported formats are the same as those in the standard C library.

source

Base.Libc.strptime – Function.

```
strptime([format], timestr)
```

Parse a formatted time string into a `TmStruct` giving the seconds, minute, hour, date, etc. Supported formats are the same as those in the standard C library. On some platforms, timezones will not be parsed correctly. If the result of this function will be passed to `time` to convert it to seconds since the epoch, the `isdst` field should be filled in manually. Setting it to `-1` will tell the C library to use the current system settings to determine the timezone.

source

Base.Libc.TmStruct – Type.

```
TmStruct([seconds])
```

Convert a number of seconds since the epoch to broken-down format, with fields `sec`, `min`, `hour`, `mday`, `month`, `year`, `wday`, `yday`, and `isdst`.

source

`Base.Libc.flush_cstdio` – Function.

```
flush_cstdio()
```

Flushes the C `stdout` and `stderr` streams (which may have been written to by external C code).

source

Chapter 65

Enlazador Dinámico

Los nombres en `Base.Libdl` no son exportados y necesitan ser llamados como, por ejemplo, `Libdl.dlopen()`.

`Base.Libdl.dlopen` – Function.

```
dlopen(libfile::AbstractString [, flags::Integer])
```

Load a shared library, returning an opaque handle.

The extension given by the constant `dlext` (`.so`, `.dll`, or `.dylib`) can be omitted from the `libfile` string, as it is automatically appended if needed. If `libfile` is not an absolute path name, then the paths in the array `DL_LOAD_PATH` are searched for `libfile`, followed by the system load path.

The optional flags argument is a bitwise-or of zero or more of `RTLD_LOCAL`, `RTLD_GLOBAL`, `RTLD_LAZY`, `RTLD_NOW`, `RTLD_NODELETE`, `RTLD_NOLOAD`, `RTLD_DEEPBIND`, and `RTLD_FIRST`. These are converted to the corresponding flags of the POSIX (and/or GNU libc and/or MacOS) dlopen command, if possible, or are ignored if the specified functionality is not available on the current platform. The default flags are platform specific. On MacOS the default `dlopen` flags are `RTLD_LAZY|RTLD_DEEPBIND|RTLD_GLOBAL` while on other platforms the defaults are `RTLD_LAZY|RTLD_DEEPBIND|RTLD_LOCAL`. An important usage of these flags is to specify non default behavior for when the dynamic library loader binds library references to exported symbols and if the bound references are put into process local or global scope. For instance `RTLD_LAZY|RTLD_DEEPBIND|RTLD_GLOBAL` allows the library's symbols to be available for usage in other shared libraries, addressing situations where there are dependencies between shared libraries.

source

`Base.Libdl.dlopen_e` – Function.

```
dlopen_e(libfile::AbstractString [, flags::Integer])
```

Similar to `dlopen`, except returns a `NULL` pointer instead of raising errors.

source

`Base.Libdl.RTLD_NOW` – Constant.

```
RTLD_DEEPBIND
RTLD_FIRST
RTLD_GLOBAL
RTLD_LAZY
RTLD_LOCAL
RTLD_NODELETE
RTLD_NOLOAD
RTLD_NOW
```

Enum constant for `dlopen`. See your platform man page for details, if applicable.

source

`Base.Libdl.dlsym` – Function.

```
dlsym(handle, sym)
```

Look up a symbol from a shared library handle, return callable function pointer on success.

source

`Base.Libdl.dlsym_e` – Function.

```
dlsym_e(handle, sym)
```

Look up a symbol from a shared library handle, silently return `NULL` pointer on lookup failure.

source

`Base.Libdl.dlclose` – Function.

```
dlclose(handle)
```

Close shared library referenced by handle.

source

`Base.Libdl.dlext` – Constant.

```
dlext
```

File extension for dynamic libraries (e.g. dll, dylib, so) on the current platform.

source

`Base.Libdl.find_library` – Function.

```
find_library(names, locations)
```

Searches for the first library in `names` in the paths in the `locations` list, `DL_LOAD_PATH`, or system library paths (in that order) which can successfully be dlopen'd. On success, the return value will be one of the names (potentially prefixed by one of the paths in locations). This string can be assigned to a `global const` and used as the library name in future `ccall`'s. On failure, it returns the empty string.

source

`Base.Libdl.DL_LOAD_PATH` – Constant.

```
DL_LOAD_PATH
```

When calling `dlopen`, the paths in this list will be searched first, in order, before searching the system locations for a valid library handle.

source

Chapter 66

Profiling

`Base.Profile.@profile` – Macro.

```
@profile
```

`@profile <expression>` runs your expression while taking periodic backtraces. These are appended to an internal buffer of backtraces.

source

Los métodos en `Base.Profile` no son exportados y necesitan ser llamados como, por ejemplo, `Profile.print()`.

`Base.Profile.clear` – Function.

```
clear()
```

Clear any existing backtraces from the internal buffer.

source

`Base.Profile.print` – Function.

```
print([io::IO = STDOUT,] [data::Vector]; kwargs...)
```

Prints profiling results to `io` (by default, `STDOUT`). If you do not supply a `data` vector, the internal buffer of accumulated backtraces will be used.

The keyword arguments can be any combination of:

- `format` – Determines whether backtraces are printed with (default, `:tree`) or without (`:flat`) indentation indicating tree structure.
- `C` – If `true`, backtraces from C and Fortran code are shown (normally they are excluded).
- `combine` – If `true` (default), instruction pointers are merged that correspond to the same line of code.
- `maxdepth` – Limits the depth higher than `maxdepth` in the `:tree` format.
- `sortedby` – Controls the order in `:flat` format. `:filefuncline` (default) sorts by the source line, whereas `:count` sorts in order of number of collected samples.
- `noisefloor` – Limits frames that exceed the heuristic noise floor of the sample (only applies to format `:tree`). A suggested value to try for this is 2.0 (the default is 0). This parameter hides samples for which `n <= noisefloor * √N`, where `n` is the number of samples on this line, and `N` is the number of samples for the callee.

- `mincount` – Limits the printout to only those lines with at least `mincount` occurrences.

source

```
print([io::IO = STDOUT,] data::Vector, lidict::LineInfoDict; kwargs...)
```

Prints profiling results to `io`. This variant is used to examine results exported by a previous call to `retrieve`. Supply the vector `data` of backtraces and a dictionary `lidict` of line information.

See `Profile.print([io], data)` for an explanation of the valid keyword arguments.

source

`Base.Profile.init` – Function.

```
init(; n::Integer, delay::Float64)
```

Configure the `delay` between backtraces (measured in seconds), and the number `n` of instruction pointers that may be stored. Each instruction pointer corresponds to a single line of code; backtraces generally consist of a long list of instruction pointers. Default settings can be obtained by calling this function with no arguments, and each can be set independently using keywords or in the order `(n, delay)`.

source

`Base.Profile.fetch` – Function.

```
fetch() -> data
```

Returns a reference to the internal buffer of backtraces. Note that subsequent operations, like `clear`, can affect `data` unless you first make a copy. Note that the values in `data` have meaning only on this machine in the current session, because it depends on the exact memory addresses used in JIT-compiling. This function is primarily for internal use; `retrieve` may be a better choice for most users.

source

`Base.Profile.retrieve` – Function.

```
retrieve() -> data, lidict
```

"Exports" profiling results in a portable format, returning the set of all backtraces (`data`) and a dictionary that maps the (session-specific) instruction pointers in `data` to `LineInfo` values that store the file name, function name, and line number. This function allows you to save profiling results for future analysis.

source

`Base.Profile.callers` – Function.

```
callers(funcname, [data, lidict], [filename=<filename>], [linerange=<start:stop>]) -> Vector{
    Tuple{count, lineinfo}}
```

Given a previous profiling run, determine who called a particular function. Supplying the filename (and optionally, range of line numbers over which the function is defined) allows you to disambiguate an overloaded method. The returned value is a vector containing a count of the number of calls and line information about the caller. One can optionally supply backtrace `data` obtained from `retrieve`; otherwise, the current internal profile buffer is used.

source

`Base.Profile.clear_malloc_data` – Function.

```
clear_malloc_data()
```

Clears any stored memory allocation data when running julia with `--track-allocation`. Execute the command(s) you want to test (to force JIT-compilation), then call `clear_malloc_data`. Then execute your command(s) again, quit Julia, and examine the resulting `*.mem` files.

source

Chapter 67

StackTraces

`Base.StackTraces.StackFrame` – Type.

```
StackFrame
```

Stack information representing execution context, with the following fields:

- `func::Symbol`
  The name of the function containing the execution context.
- `linfo::Nullable{Core.MethodInstance}`
  The MethodInstance containing the execution context (if it could be found).
- `file::Symbol`
  The path to the file containing the execution context.
- `line::Int`
  The line number in the file containing the execution context.
- `from_c::Bool`
  True if the code is from C.
- `inlined::Bool`
  True if the code is from an inlined frame.
- `pointer::UInt64`
  Representation of the pointer to the execution context as returned by `backtrace`.

source

`Base.StackTraces.StackTrace` – Type.

```
StackTrace
```

An alias for `Vector{StackFrame}` provided for convenience; returned by calls to `stacktrace` and `catch_stacktrace`.

source

`Base.StackTraces.stacktrace` – Function.

```
stacktrace([trace::Vector{Ptr{Void}},] [c_funcs::Bool=false]) -> StackTrace
```

Returns a stack trace in the form of a vector of `StackFrames`. (By default stacktrace doesn't return C functions, but this can be enabled.) When called without specifying a trace, `stacktrace` first calls `backtrace`.

source

`Base.StackTraces.catch_stacktrace` – Function.

```
catch_stacktrace([c_funcs::Bool=false]) -> StackTrace
```

Returns the stack trace for the most recent error thrown, rather than the current execution context.

source

Los siguientes métodos y tipos de `Base.StackTraces` no son exportados y, por tanto, deben ser prefijados en sus invocaciones. Por ejemplo, `StackTraces.lookup(ptr)`.

`Base.StackTraces.lookup` – Function.

```
lookup(pointer::Union{Ptr{Void}, UInt}) -> Vector{StackFrame}
```

Given a pointer to an execution context (usually generated by a call to `backtrace`), looks up stack frame context information. Returns an array of frame information for all functions inlined at that point, innermost function first.

source

`Base.StackTraces.remove_frames!` – Function.

```
remove_frames!(stack::StackTrace, name::Symbol)
```

Takes a `StackTrace` (a vector of `StackFrames`) and a function name (a `Symbol`) and removes the `StackFrame` specified by the function name from the `StackTrace` (also removing all frames above the specified function). Primarily used to remove `StackTraces` functions from the `StackTrace` prior to returning it.

source

```
remove_frames!(stack::StackTrace, m::Module)
```

Returns the `StackTrace` with all `StackFrames` from the provided `Module` removed.

source

Chapter 68

Soporte SIMD

El tipo `VecElement{T}` está pensado para construir librerías de operaciones SIMD. Su uso práctico requiere usar `llvmcall`. El tipo está definido como:

```
struct VecElement{T}
    value::T
end
```

Tiene una regla de compilación especial: una tupla homogénea de `VecElement{T}` se corresponde con un tipo `vector` LLVM cuando T es un tipo de bits primitivo y la longitud de la tupla está en el conjunto {2-6,8-10,16}.

En `-O3`, el compilador *podría* vectorizar operaciones sobre tales tuplas automáticamente. Por ejemplo, el siguiente programa, cuando se compila con `julia -O3` genera dos instrucciones de adición SIMD (`addps`) sobre los sistemas x86:

```
const m128 = NTuple{4,VecElement{Float32}}

function add(a::m128, b::m128)
    (VecElement(a[1].value+b[1].value),
     VecElement(a[2].value+b[2].value),
     VecElement(a[3].value+b[3].value),
     VecElement(a[4].value+b[4].value))
end

triple(c::m128) = add(add(c,c),c)

code_native(triple,(m128,))
```

Sin embargo, dado que no se puede confiar en la vectorización automática, el uso futuro se realizará principalmente a través de bibliotecas que usen `llvmcall`.

Part V

Developer Documentation

Chapter 69

Reflection and introspection

Julia provides a variety of runtime reflection capabilities.

69.1 Module bindings

The exported names for a `Module` are available using `names(m::Module)`, which will return an array of `Symbol` elements representing the exported bindings. `names(m::Module, true)` returns symbols for all bindings in `m`, regardless of export status.

69.2 DataType fields

The names of `DataType` fields may be interrogated using `fieldnames()`. For example, given the following type, `fieldnames(Point)` returns an arrays of `Symbol` elements representing the field names:

```
julia> struct Point
           x::Int
           y
       end

julia> fieldnames(Point)
2-element Array{Symbol,1}:
 :x
 :y
```

The type of each field in a `Point` object is stored in the `types` field of the `Point` variable itself:

```
julia> Point.types
svec(Int64, Any)
```

While `x` is annotated as an `Int`, `y` was unannotated in the type definition, therefore `y` defaults to the `Any` type.

Types are themselves represented as a structure called `DataType`:

```
julia> typeof(Point)
DataType
```

Note that `fieldnames(DataType)` gives the names for each field of `DataType` itself, and one of these fields is the `types` field observed in the example above.

69.3 Subtypes

The *direct* subtypes of any `DataType` may be listed using `subtypes()`. For example, the abstract `DataType` `AbstractFloat` has four (concrete) subtypes:

```
julia> subtypes(AbstractFloat)
4-element Array{Union{DataType, UnionAll},1}:
 BigFloat
 Float16
 Float32
 Float64
```

Any abstract subtype will also be included in this list, but further subtypes thereof will not; recursive application of `subtypes()` may be used to inspect the full type tree.

69.4 DataType layout

The internal representation of a `DataType` is critically important when interfacing with C code and several functions are available to inspect these details. `isbits(T::DataType)` returns true if T is stored with C-compatible alignment. `fieldoffset(T::DataType, i::Integer)` returns the (byte) offset for field *i* relative to the start of the type.

69.5 Function methods

The methods of any generic function may be listed using `methods()`. The method dispatch table may be searched for methods accepting a given type using `methodswith()`.

69.6 Expansion and lowering

As discussed in the Metaprogramming section, the `macroexpand()` function gives the unquoted and interpolated expression (`Expr`) form for a given macro. To use `macroexpand`, `quote` the expression block itself (otherwise, the macro will be evaluated and the result will be passed instead!). For example:

```
julia> macroexpand( :(@edit println("")) )
:((Base.edit)(println, (Base.typesof)("")))
```

The functions `Base.Meta.show_sexpr()` and `dump()` are used to display S-expr style views and depth-nested detail views for any expression.

Finally, the `expand()` function gives the `lowered` form of any expression and is of particular interest for understanding both macros and top-level statements such as function declarations and variable assignments:

```
julia> expand( :(f() = 1) )
:(begin
    end)), false))
    end)
```

69.7 Intermediate and compiled representations

Inspecting the lowered form for functions requires selection of the specific method to display, because generic functions may have many methods with different type signatures. For this purpose, method-specific code-lowering is available using `code_lowered(f::Function, (Argtypes...))`, and the type-inferred form is available using `code_typed(f::Function, (Argtypes...))`. `code_warntype(f::Function, (Argtypes...))` adds highlighting to the output of `code_typed()` (see `@code_warntype`).

Closer to the machine, the LLVM intermediate representation of a function may be printed using by `code_llvm(f::Function, (Argtypes...))`, and finally the compiled machine code is available using `code_native(f::Function, (Argtypes...))` (this will trigger JIT compilation/code generation for any function which has not previously been called).

For convenience, there are macro versions of the above functions which take standard function calls and expand argument types automatically:

```
julia> @code_llvm +(1,1)

; Function Attrs: sspreq
define i64 @"julia_+_130862"(i64, i64) #0 {
top:
    %2 = add i64 %1, %0, !dbg !8
    ret i64 %2, !dbg !8
}
```

(likewise `@code_typed`, `@code_warntype`, `@code_lowered`, and `@code_native`)

Chapter 70

Documentation of Julia's Internals

70.1 Initialization of the Julia runtime

How does the Julia runtime execute `julia -e 'println("Hello World!")'`?

main()

Execution starts at `main()` in `ui/repl.c`.

`main()` calls `libsupport_init()` to set the C library locale and to initialize the "ios" library (see `ios_init_stdstreams()` and Legacy `ios.c` library).

Next `parse_opts()` is called to process command line options. Note that `parse_opts()` only deals with options that affect code generation or early initialization. Other options are handled later by `process_options()` in `base/client.jl`.

`parse_opts()` stores command line options in the global `jl_options` struct.

julia_init()

`julia_init()` in `task.c` is called by `main()` and calls `_julia_init()` in `init.c`.

`_julia_init()` begins by calling `libsupport_init()` again (it does nothing the second time).

`restore_signals()` is called to zero the signal handler mask.

`jl_resolve_sysimg_location()` searches configured paths for the base system image. See Building the Julia system image.

`jl_gc_init()` sets up allocation pools and lists for weak refs, preserved values and finalization.

`jl_init_frontend()` loads and initializes a pre-compiled femtolisp image containing the scanner/parser.

`jl_init_types()` creates `jl_datatype_t` type description objects for the built-in types defined in `julia.h`. e.g.

```
jl_any_type = jl_new_abstracttype(jl_symbol("Any"), NULL, jl_null);
jl_any_type->super = jl_any_type;

jl_type_type = jl_new_abstracttype(jl_symbol("Type"), jl_any_type, jl_null);

jl_int32_type = jl_new_bitstype(jl_symbol("Int32"),
                                jl_any_type, jl_null, 32);
```

`jl_init_tasks()` creates the `jl_datatype_t* jl_task_type` object; initializes the global `jl_root_task` struct; and sets `jl_current_task` to the root task.

`jl_init_codegen()` initializes the LLVM library.

`jl_init_serializer()` initializes 8-bit serialization tags for 256 frequently used `jl_value_t` values. The serialization mechanism uses these tags as shorthand (in lieu of storing whole objects) to save storage space.

If there is no sysimg file (`!jl_options.image_file`) then the `Core` and `Main` modules are created and `boot.jl` is evaluated:

`jl_core_module = jl_new_module(jl_symbol("Core"))` creates the Julia `Core` module.

`jl_init_intrinsic_functions()` creates a new Julia module `Intrinsics` containing constant `jl_intrinsic_type` symbols. These define an integer code for each intrinsic function. `emit_intrinsic()` translates these symbols into LLVM instructions during code generation.

`jl_init_primitives()` hooks C functions up to Julia function symbols. e.g. the symbol `Base.is()` is bound to C function pointer `jl_f_is()` by calling `add_builtin_func("eval", jl_f_top_eval)`.

`jl_new_main_module()` creates the global "Main" module and sets `jl_current_task->current_module = jl_main_module`.

Note: `_julia_init()` then sets `jl_root_task->current_module = jl_core_module`. `jl_root_task` is an alias of `jl_current_task` at this point, so the `current_module` set by `jl_new_main_module()` above is overwritten.

`jl_load("boot.jl", sizeof("boot.jl"))` calls `jl_parse_eval_all` which repeatedly calls `jl_toplevel_eval_flex()` to execute `boot.jl`. <!– TODO – drill down into eval? –>

`jl_get_builtin_hooks()` initializes global C pointers to Julia globals defined in `boot.jl`.

`jl_init_box_caches()` pre-allocates global boxed integer value objects for values up to 1024. This speeds up allocation of boxed ints later on. e.g.:

```
jl_value_t *jl_box_uint8(uint32_t x)
{
    return boxed_uint8_cache[(uint8_t)x];
}
```

`_julia_init()` iterates over the `jl_core_module->bindings.table` looking for `jl_datatype_t` values and sets the type name's module prefix to `jl_core_module`.

`jl_add_standard_imports(jl_main_module)` does "using Base" in the "Main" module.

Note: `_julia_init()` now reverts to `jl_root_task->current_module = jl_main_module` as it was before being set to `jl_core_module` above.

Platform specific signal handlers are initialized for `SIGSEGV` (OSX, Linux), and `SIGFPE` (Windows).

Other signals (`SIGINFO`, `SIGBUS`, `SIGILL`, `SIGTERM`, `SIGABRT`, `SIGQUIT`, `SIGSYS` and `SIGPIPE`) are hooked up to `sigdie_handler()` which prints a backtrace.

`jl_init_restored_modules()` calls `jl_module_run_initializer()` for each deserialized module to run the `__init__()` function.

Finally `sigint_handler()` is hooked up to `SIGINT` and calls `jl_throw(jl_interrupt_exception)`.

`_julia_init()` then returns back to `main()` in `ui/repl.c` and `main()` calls `true_main(argc, (char**)argv)`.

sysimg

If there is a sysimg file, it contains a pre-cooked image of the `Core` and `Main` modules (and whatever else is created by `boot.jl`). See Building the Julia system image.

`jl_restore_system_image()` deserializes the saved sysimg into the current Julia runtime environment and initialization continues after `jl_init_box_caches()` below...

Note: `jl_restore_system_image()` (and `dump.c` in general) uses the Legacy `ios.c` library.

true_main()

`true_main()` loads the contents of `argv[]` into `Base.ARGS`.

If a `.jl` "program" file was supplied on the command line, then `exec_program()` calls `jl_load(program,len)` which calls `jl_parse_eval_all` which repeatedly calls `jl_toplevel_eval_flex()` to execute the program.

However, in our example (`julia -e 'println("Hello World!")'`), `jl_get_global(jl_base_module, jl_symbol("_start"))` looks up `Base._start` and `jl_apply()` executes it.

Base._start

`Base._start` calls `Base.process_options` which calls `jl_parse_input_line("println("Hello World!")")` to create an expression object and `Base.eval()` to execute it.

Base.eval

`Base.eval()` was mapped to `jl_f_top_eval` by `jl_init_primitives()`.

`jl_f_top_eval()` calls `jl_toplevel_eval_in(jl_main_module, ex)`, where ex is the parsed expression `println("Hello World!")`.

`jl_toplevel_eval_in()` calls `jl_toplevel_eval_flex()` which calls `eval()` in `interpreter.c`.

The stack dump below shows how the interpreter works its way through various methods of `Base.println()` and `Base.print()` before arriving at `write(s::IO, a::Array{T}) where T` which does `ccall(jl_uv_write())`.

`jl_uv_write()` calls `uv_write()` to write "Hello World!" to JL_STDOUT. See Libuv wrappers for stdio.:

```
Hello World!
```

Since our example has just one function call, which has done its job of printing "Hello World!", the stack now rapidly unwinds back to `main()`.

jl_atexit_hook()

`main()` calls `jl_atexit_hook()`. This calls `_atexit` for each module, then calls `jl_gc_run_all_finalizers()` and cleans up libuv handles.

julia_save()

Finally, `main()` calls `julia_save()`, which if requested on the command line, saves the runtime state to a new system image. See `jl_compile_all()` and `jl_save_system_image()`.

70.2 Julia ASTs

Julia has two representations of code. First there is a surface syntax AST returned by the parser (e.g. the `parse()` function), and manipulated by macros. It is a structured representation of code as it is written, constructed by `julia-parser.scm` from a character stream. Next there is a lowered form, or IR (intermediate representation), which is used by type inference and code generation. In the lowered form there are fewer types of nodes, all macros are expanded, and all control flow is converted to explicit branches and sequences of statements. The lowered form is constructed by `julia-syntax.scm`.

Stack frame	Source code	Notes
jl_uv_write()	jl_uv.c	called though ccall
julia_write_282942	stream.jl	function write!(s::IO, a::Array{T}) where T
julia_print_284639	ascii.jl	print(io::IO, s::String) = (write(io, s); nothing)
jlcall_print_284639		
jl_apply()	julia.h	
jl_trampoline()	builtins.c	
jl_apply()	julia.h	
jl_apply_generic()	gf.c	Base.print(Base.TTY, String)
jl_apply()	julia.h	
jl_trampoline()	builtins.c	
jl_apply()	julia.h	
jl_apply_generic()	gf.c	Base.print(Base.TTY, String, Char, Char...)
jl_apply()	julia.h	
jl_f_apply()	builtins.c	
jl_apply()	julia.h	
jl_trampoline()	builtins.c	
jl_apply()	julia.h	
jl_apply_generic()	gf.c	Base.println(Base.TTY, String, String...)
jl_apply()	julia.h	
jl_trampoline()	builtins.c	
jl_apply()	julia.h	
jl_apply_generic()	gf.c	Base.println(String,)
jl_apply()	julia.h	
do_call()	interpreter.c	
eval()	interpreter.c	
jl_interpret_toplevel_expr()	interpreter.c	
jl_toplevel_eval_flex()	toplevel.c	
jl_toplevel_eval()	toplevel.c	
jl_toplevel_eval_in()	builtins.c	
jl_f_top_eval()	builtins.c	

First we will focus on the lowered form, since it is more important to the compiler. It is also less obvious to the human, since it results from a significant rearrangement of the input syntax.

Lowered form

The following data types exist in lowered form:

- `Expr`

  Has a node type indicated by the `head` field, and an `args` field which is a `Vector{Any}` of subexpressions.

- `Slot`

  Identifies arguments and local variables by consecutive numbering. `Slot` is an abstract type with subtypes `SlotNumber` and `TypedSlot`. Both types have an integer-valued `id` field giving the slot index. Most slots have the same type at all uses, and so are represented with `SlotNumber`. The types of these slots are found in the

`slottypes` field of their `MethodInstance` object. Slots that require per-use type annotations are represented with `TypedSlot`, which has a `typ` field.

- `CodeInfo`

  Wraps the IR of a method.

- `LineNumberNode`

  Contains a single number, specifying the line number the next statement came from.

- `LabelNode`

  Branch target, a consecutively-numbered integer starting at 0.

- `GotoNode`

  Unconditional branch.

- `QuoteNode`

  Wraps an arbitrary value to reference as data. For example, the function `f() = :a` contains a `QuoteNode` whose `value` field is the symbol `a`, in order to return the symbol itself instead of evaluating it.

- `GlobalRef`

  Refers to global variable `name` in module `mod`.

- `SSAValue`

  Refers to a consecutively-numbered (starting at 0) static single assignment (SSA) variable inserted by the compiler.

- `NewvarNode`

  Marks a point where a variable is created. This has the effect of resetting a variable to undefined.

Expr types

These symbols appear in the `head` field of `Exprs` in lowered form.

- `call`

  Function call (dynamic dispatch). `args[1]` is the function to call, `args[2:end]` are the arguments.

- `invoke`

  Function call (static dispatch). `args[1]` is the MethodInstance to call, `args[2:end]` are the arguments (including the function that is being called, at `args[2]`).

- `static_parameter`

  Reference a static parameter by index.

- `line`

  Line number and file name metadata. Unlike a `LineNumberNode`, can also contain a file name.

- `gotoifnot`

  Conditional branch. If `args[1]` is false, goes to label identified in `args[2]`.

- `=`

  Assignment.

- `method`

  Adds a method to a generic function and assigns the result if necessary.

  Has a 1-argument form and a 4-argument form. The 1-argument form arises from the syntax `function foo end`. In the 1-argument form, the argument is a symbol. If this symbol already names a function in the current scope, nothing happens. If the symbol is undefined, a new function is created and assigned to the identifier specified by the symbol. If the symbol is defined but names a non-function, an error is raised. The definition of "names a function" is that the binding is constant, and refers to an object of singleton type. The rationale for this is that an instance of a singleton type uniquely identifies the type to add the method to. When the type has fields, it wouldn't be clear whether the method was being added to the instance or its type.

  The 4-argument form has the following arguments:

  - `args[1]`

    A function name, or `false` if unknown. If a symbol, then the expression first behaves like the 1-argument form above. This argument is ignored from then on. When this is `false`, it means a method is being added strictly by type, `(::T)(x) = x`.

  - `args[2]`

    A `SimpleVector` of argument type data. `args[2][1]` is a `SimpleVector` of the argument types, and `args[2][2]` is a `SimpleVector` of type variables corresponding to the method's static parameters.

  - `args[3]`

    A `CodeInfo` of the method itself. For "out of scope" method definitions (adding a method to a function that also has methods defined in different scopes) this is an expression that evaluates to a `:lambda` expression.

  - `args[4]`

    `true` or `false`, identifying whether the method is staged (`@generated function`).

- `const`

  Declares a (global) variable as constant.

- `null`

  Has no arguments; simply yields the value `nothing`.

- `new`

  Allocates a new struct-like object. First argument is the type. The `new` pseudo-function is lowered to this, and the type is always inserted by the compiler. This is very much an internal-only feature, and does no checking. Evaluating arbitrary `new` expressions can easily segfault.

- `return`

  Returns its argument as the value of the enclosing function.

- `the_exception`

  Yields the caught exception inside a `catch` block. This is the value of the run time system variable `jl_exception_in_transit`.

- `enter`

  Enters an exception handler (`setjmp`). `args[1]` is the label of the catch block to jump to on error.

- `leave`

  Pop exception handlers. `args[1]` is the number of handlers to pop.

- `inbounds`

  Controls turning bounds checks on or off. A stack is maintained; if the first argument of this expression is true or false (`true` means bounds checks are disabled), it is pushed onto the stack. If the first argument is `:pop`, the stack is popped.

- `boundscheck`

  Indicates the beginning or end of a section of code that performs a bounds check. Like `inbounds`, a stack is maintained, and the second argument can be one of: `true`, `false`, or `:pop`.

- `copyast`

  Part of the implementation of quasi-quote. The argument is a surface syntax AST that is simply copied recursively and returned at run time.

- `meta`

  Metadata. `args[1]` is typically a symbol specifying the kind of metadata, and the rest of the arguments are free-form. The following kinds of metadata are commonly used:

  - `:inline` and `:noinline`: Inlining hints.
  - `:push_loc`: enters a sequence of statements from a specified source location.
    * `args[2]` specifies a filename, as a symbol.
    * `args[3]` optionally specifies the name of an (inlined) function that originally contained the code.
  - `:pop_loc`: returns to the source location before the matching `:push_loc`.

Method

A unique'd container describing the shared metadata for a single method.

- `name`, `module`, `file`, `line`, `sig`

  Metadata to uniquely identify the method for the computer and the human.

- `ambig`

  Cache of other methods that may be ambiguous with this one.

- `specializations`

  Cache of all MethodInstance ever created for this Method, used to ensure uniqueness. Uniqueness is required for efficiency, especially for incremental precompile and tracking of method invalidation.

- `source`

  The original source code (usually compressed).

- `roots`

  Pointers to non-AST things that have been interpolated into the AST, required by compression of the AST, type-inference, or the generation of native code.

- `nargs`, `isva`, `called`, `isstaged`, `pure`

  Descriptive bit-fields for the source code of this Method.

- `min_world` / `max_world`

  The range of world ages for which this method is visible to dispatch.

MethodInstance

A unique'd container describing a single callable signature for a Method. See especially Proper maintenance and care of multi-threading locks for important details on how to modify these fields safely.

- `specTypes`

  The primary key for this MethodInstance. Uniqueness is guaranteed through a `def.specializations` lookup.

- `def`

  The `Method` that this function describes a specialization of. Or `#undef`, if this is a top-level Lambda that is not part of a Method.

- `sparam_vals`

  The values of the static parameters in `specTypes` indexed by `def.sparam_syms`. For the `MethodInstance` at `Method.unspecialized`, this is the empty `SimpleVector`. But for a runtime `MethodInstance` from the `MethodTable` cache, this will always be defined and indexable.

- `rettype`

  The inferred return type for the `specFunctionObject` field, which (in most cases) is also the computed return type for the function in general.

- `inferred`

  May contain a cache of the inferred source for this function, or other information about the inference result such as a constant return value may be put here (if `jlcall_api == 2`), or it could be set to `nothing` to just indicate `rettype` is inferred.

- `ftpr`

  The generic jlcall entry point.

- `jlcall_api`

  The ABI to use when calling `fptr`. Some significant ones include:

  - 0 - Not compiled yet
  - 1 - JL_CALLABLE `jl_value_t *(*)(jl_function_t *f, jl_value_t *args[nargs], uint32_t nargs)`
  - 2 - Constant (value stored in `inferred`)
  - 3 - With Static-parameters forwarded `jl_value_t *(*)(jl_svec_t *sparams, jl_function_t *f, jl_value_t *args[nargs], uint32_t nargs)`
  - 4 - Run in interpreter `jl_value_t *(*)(jl_method_instance_t *meth, jl_function_t *f, jl_value_t *args[nargs], uint32_t nargs)`

- `min_world` / `max_world`

  The range of world ages for which this method instance is valid to be called.

CodeInfo

A temporary container for holding lowered source code.

- `code`

  An `Any` array of statements

- `slotnames`

  An array of symbols giving the name of each slot (argument or local variable).

- `slottypes`

  An array of types for the slots.

- `slotflags`

  A `UInt8` array of slot properties, represented as bit flags:

  - 2 - assigned (only false if there are *no* assignment statements with this var on the left)
  - 8 - const (currently unused for local variables)
  - 16 - statically assigned once
  - 32 - might be used before assigned. This flag is only valid after type inference.

- `ssavaluetypes`

  Either an array or an `Int`.

  If an `Int`, it gives the number of compiler-inserted temporary locations in the function. If an array, specifies a type for each location.

Boolean properties:

- `inferred`

  Whether this has been produced by type inference.

- `inlineable`

  Whether this should be inlined.

- `propagate_inbounds`

  Whether this should should propagate `@inbounds` when inlined for the purpose of eliding `@boundscheck` blocks.

- `pure`

  Whether this is known to be a pure function of its arguments, without respect to the state of the method caches or other mutable global state.

Surface syntax AST

Front end ASTs consist entirely of `Exprs` and atoms (e.g. symbols, numbers). There is generally a different expression head for each visually distinct syntactic form. Examples will be given in s-expression syntax. Each parenthesized list corresponds to an Expr, where the first element is the head. For example `(call f x)` corresponds to `Expr(:call, :f, :x)` in Julia.

Input	AST
`f(x)`	`(call f x)`
`f(x, y=1, z=2)`	`(call f x (kw y 1) (kw z 2))`
`f(x; y=1)`	`(call f (parameters (kw y 1)) x)`
`f(x...)`	`(call f (... x))`

Calls

do syntax:

```
f(x) do a,b
    body
end
```

parses as `(call f (-> (tuple a b) (block body)) x)`.

Operators

Most uses of operators are just function calls, so they are parsed with the head `call`. However some operators are special forms (not necessarily function calls), and in those cases the operator itself is the expression head. In julia-parser.scm these are referred to as "syntactic operators". Some operators (+ and *) use N-ary parsing; chained calls are parsed as a single N-argument call. Finally, chains of comparisons have their own special expression structure.

Input	AST
`x+y`	`(call + x y)`
`a+b+c+d`	`(call + a b c d)`
`2x`	`(call * 2 x)`
`a&&b`	`(&& a b)`
`x += 1`	`(+= x 1)`
`a ? 1 : 2`	`(if a 1 2)`
`a:b`	`(: a b)`
`a:b:c`	`(: a b c)`
`a,b`	`(tuple a b)`
`a==b`	`(call == a b)`
`1<i<=n`	`(comparison 1 < i <= n)`
`a.b`	`(. a (quote b))`
`a.(b)`	`(. a b)`

Bracketed forms

Macros

Strings

Doc string syntax:

```
"some docs"
f(x) = x
```

parses as `(macrocall (|.| Core '@doc) "some docs" (= (call f x) (block x)))`.

Input	AST
`a[i]`	`(ref a i)`
`t[i;j]`	`(typed_vcat t i j)`
`t[i j]`	`(typed_hcat t i j)`
`t[a b; c d]`	`(typed_vcat t (row a b) (row c d))`
`a{b}`	`(curly a b)`
`a{b;c}`	`(curly a (parameters c) b)`
`[x]`	`(vect x)`
`[x,y]`	`(vect x y)`
`[x;y]`	`(vcat x y)`
`[x y]`	`(hcat x y)`
`[x y; z t]`	`(vcat (row x y) (row z t))`
`[x for y in z, a in b]`	`(comprehension x (= y z) (= a b))`
`T[x for y in z]`	`(typed_comprehension T x (= y z))`
`(a, b, c)`	`(tuple a b c)`
`(a; b; c)`	`(block a (block b c))`

Input	AST
`@m x y`	`(macrocall @m x y)`
`Base.@m x y`	`(macrocall (. Base (quote @m)) x y)`
`@Base.m x y`	`(macrocall (. Base (quote @m)) x y)`

Input	AST
`"a"`	`"a"`
`x"y"`	`(macrocall @x_str "y")`
`x"y"z`	`(macrocall @x_str "y" "z")`
`"x = $x"`	`(string "x = " x)`
`` `a b c` ``	`(macrocall @cmd "a b c")`

Imports and such

Numbers

Julia supports more number types than many scheme implementations, so not all numbers are represented directly as scheme numbers in the AST.

Block forms

A block of statements is parsed as `(block stmt1 stmt2 ...)`.

If statement:

```
if a
    b
elseif c
    d
else e
    f
end
```

parses as:

Input	AST
`import a`	`(import a)`
`import a.b.c`	`(import a b c)`
`import ...a`	`(import . . . a)`
`import a.b, c.d`	`(toplevel (import a b) (import c d))`
`import Base: x`	`(import Base x)`
`import Base: x, y`	`(toplevel (import Base x) (import Base y))`
`export a, b`	`(export a b)`

Input	AST
`11111111111111111111`	`(macrocall @int128_str "11111111111111111111")`
`0xffffffffffffffff`	`(macrocall @uint128_str "0xffffffffffffffff")`
`1111...many digits...`	`(macrocall @big_str "1111....")`

```
(if a (block (line 2) b)
    (block (line 3) (if c (block (line 4) d)
                          (block (line 5) e (line 6) f))))
```

A `while` loop parses as `(while condition body)`.

A `for` loop parses as `(for (= var iter) body)`. If there is more than one iteration specification, they are parsed as a block: `(for (block (= v1 iter1) (= v2 iter2)) body)`.

`break` and `continue` are parsed as 0-argument expressions `(break)` and `(continue)`.

`let` is parsed as `(let body (= var1 val1) (= var2 val2) ...)`.

A basic function definition is parsed as `(function (call f x) body)`. A more complex example:

```
function f{T}(x::T; k = 1)
    return x+1
end
```

parses as:

```
(function (call (curly f T) (parameters (kw k 1))
                (:: x T))
          (block (line 2 file.jl) (return (call + x 1))))
```

Type definition:

```
mutable struct Foo{T<:S}
    x::T
end
```

parses as:

```
(type #t (curly Foo (<: T S))
      (block (line 2 none) (:: x T)))
```

The first argument is a boolean telling whether the type is mutable.

`try` blocks parse as `(try try_block var catch_block finally_block)`. If no variable is present after `catch`, `var` is `#f`. If there is no `finally` clause, then the last argument is not present.

70.3 Más sobre tipos

Si ha usado Julia durante un tiempo, comprenderá el papel fundamental que juegan los tipos. Aquí intentamos meternos debajo del capó, centrándonos fundamentalmente en los Tipos Paramétricos.

Tipos y conjuntos (y `Any` y `Union{}`/`Bottom`)

Tal vez sea ms sencillo concebir el sistema de tipos de Julia en términos de conjuntos. Aunque los programas pueden manipular los valores individuales, un tipo se refiere a un conjunto de valores. Esto no es la misma cosa que una colección; por ejemplo, un `Set` de valores es en sí mismo un solo valor de tipo `Set`. En lugar de ello, un tipo describe un conjunto de *posible* valores, expresando incertidumbre sobre qué valor tenemos.

Un tipo *concreto* T describe el conjunto de valores cuya etiqueta directa, tal y como es obtenida por la función `typeof` es T. Un tipo *abstracto* describe un conjunto de valores posiblemente más grande.

`Any` describe el universo completo de valores posibles. `Integer` es un subconjunto de `Any` que incluye `Int`, `Int8`, y otros tipos concretos. Internamente, Julia también hace un uso intensivo de otro tipo conocido como `Bottom`, que puede también ser escrito como `Union{}`. Esto corresponde al conjunto vacío.

Los tipos de Julia soportan las operaciones estándar de la teora de conjuntos: uno puede preguntar si `T1` es un "subconjunto" de `T2` con `T1 <: T2`. Análogamente, uno intersecta dos tipos usando `typeintersect`, realiza su unión con `Union`, y calcula un tipo que contiene su unión con `typejoin`:

```
julia> typeintersect(Int, Float64)
Union{}

julia> Union{Int, Float64}
Union{Float64, Int64}

julia> typejoin(Int, Float64)
Real

julia> typeintersect(Signed, Union{UInt8, Int8})
Int8

julia> Union{Signed, Union{UInt8, Int8}}
Union{Signed, UInt8}

julia> typejoin(Signed, Union{UInt8, Int8})
Integer

julia> typeintersect(Tuple{Integer,Float64}, Tuple{Int,Real})
Tuple{Int64,Float64}

julia> Union{Tuple{Integer,Float64}, Tuple{Int,Real}}
Union{Tuple{Int64,Real}, Tuple{Integer,Float64}}

julia> typejoin(Tuple{Integer,Float64}, Tuple{Int,Real})
Tuple{Integer,Real}
```

While these operations may seem abstract, they lie at the heart of Julia. For example, method dispatch is implemented by stepping through the items in a method list until reaching one for which the type of the argument tuple is a subtype of the method signature. For this algorithm to work, it's important that methods be sorted by their specificity, and that the search begins with the most specific methods. Consequently, Julia also implements a partial order on types; this is achieved by functionality that is similar to `<:`, but with differences that will be discussed below.

UnionAll types

Julia's type system can also express an *iterated union* of types: a union of types over all values of some variable. This is needed to describe parametric types where the values of some parameters are not known.

For example, :obj:`Array` has two parameters as in `Array{Int,2}`. If we did not know the element type, we could write `Array{T,2} where T`, which is the union of `Array{T,2}` for all values of `T`: `Union{Array{Int8,2}, Array{Int16,2}, ...}`.

Such a type is represented by a `UnionAll` object, which contains a variable (`T` in this example, of type `TypeVar`), and a wrapped type (`Array{T,2}` in this example).

Consider the following methods:

```
f1(A::Array) = 1
f2(A::Array{Int}) = 2
f3(A::Array{T}) where {T<:Any} = 3
f4(A::Array{Any}) = 4
```

The signature of `f3` is a `UnionAll` type wrapping a tuple type. All but `f4` can be called with `a = [1,2]`; all but `f2` can be called with `b = Any[1,2]`.

Let's look at these types a little more closely:

```
julia> dump(Array)
UnionAll
  var: TypeVar
    name: Symbol T
    lb: Core.TypeofBottom Union{}
    ub: Any
  body: UnionAll
    var: TypeVar
    body: Array{T,N} <: DenseArray{T,N}
```

This indicates that `Array` actually names a `UnionAll` type. There is one `UnionAll` type for each parameter, nested. The syntax `Array{Int,2}` is equivalent to `Array{Int}{2}`; internally each `UnionAll` is instantiated with a particular variable value, one at a time, outermost-first. This gives a natural meaning to the omission of trailing type parameters; `Array{Int}` gives a type equivalent to `Array{Int,N} where N`.

A `TypeVar` is not itself a type, but rather should be considered part of the structure of a `UnionAll` type. Type variables have lower and upper bounds on their values (in the fields `lb` and `ub`). The symbol `name` is purely cosmetic. Internally, `TypeVars` are compared by address, so they are defined as mutable types to ensure that "different" type variables can be distinguished. However, by convention they should not be mutated.

One can construct `TypeVars` manually:

```
julia> TypeVar(:V, Signed, Real)
Signed<:V<:Real
```

There are convenience versions that allow you to omit any of these arguments except the `name` symbol.

The syntax `Array{T} where T<:Integer` is lowered to

```
let T = TypeVar(:T,Integer)
    UnionAll(T, Array{T})
end
```

so it is seldom necessary to construct a `TypeVar` manually (indeed, this is to be avoided).

Free variables

The concept of a *free* type variable is extremely important in the type system. We say that a variable `V` is free in type `T` if `T` does not contain the `UnionAll` that introduces variable `V`. For example, the type `Array{Array{V} where V<:Integer}` has no free variables, but the `Array{V}` part inside of it does have a free variable, `V`.

A type with free variables is, in some sense, not really a type at all. Consider the type `Array{Array{T}} where T`, which refers to all homogeneous arrays of arrays. The inner type `Array{T}`, seen by itself, might seem to refer to any kind of array. However, every element of the outer array must have the *same* array type, so `Array{T}` cannot refer to just any old array. One could say that `Array{T}` effectively "occurs" multiple times, and `T` must have the same value each "time".

For this reason, the function `jl_has_free_typevars` in the C API is very important. Types for which it returns true will not give meaningful answers in subtyping and other type functions.

TypeNames

The following two `Array` types are functionally equivalent, yet print differently:

```
julia> TV, NV = TypeVar(:T), TypeVar(:N)
(T, N)

julia> Array
Array

julia> Array{TV,NV}
Array{T,N}
```

These can be distinguished by examining the `name` field of the type, which is an object of type `TypeName`:

```
julia> dump(Array{Int,1}.name)
TypeName
  name: Symbol Array
  module: Module Core
  names: empty SimpleVector
  wrapper: UnionAll
    var: TypeVar
    body: UnionAll
  cache: SimpleVector
    ...

  linearcache: SimpleVector
    ...

  hash: Int64 -7900426068641098781
  mt: MethodTable
    name: Symbol Array
    defs: Void nothing
    cache: Void nothing
    max_args: Int64 0
    kwsorter: #undef
    module: Module Core
    : Int64 0
    : Int64 0
```

In this case, the relevant field is `wrapper`, which holds a reference to the top-level type used to make new `Array` types.

```
julia> pointer_from_objref(Array)
Ptr{Void} @0x00007fcc7de64850

julia> pointer_from_objref(Array.body.body.name.wrapper)
Ptr{Void} @0x00007fcc7de64850

julia> pointer_from_objref(Array{TV,NV})
Ptr{Void} @0x00007fcc80c4d930

julia> pointer_from_objref(Array{TV,NV}.name.wrapper)
Ptr{Void} @0x00007fcc7de64850
```

The `wrapper` field of `Array` points to itself, but for `Array{TV,NV}` it points back to the original definition of the type.

What about the other fields? `hash` assigns an integer to each type. To examine the `cache` field, it's helpful to pick a type that is less heavily used than Array. Let's first create our own type:

```
julia> struct MyType{T,N} end

julia> MyType{Int,2}
MyType{Int64,2}

julia> MyType{Float32, 5}
MyType{Float32,5}

julia> MyType.body.body.name.cache
svec(MyType{Float32,5}, MyType{Int64,2}, #undef, #undef, #undef, #undef, #undef, #undef)
```

(The cache is pre-allocated to have length 8, but only the first two entries are populated.) Consequently, when you instantiate a parametric type, each concrete type gets saved in a type cache. However, instances containing free type variables are not cached.

Tuple types

Tuple types constitute an interesting special case. For dispatch to work on declarations like `x::Tuple`, the type has to be able to accommodate any tuple. Let's check the parameters:

```
julia> Tuple
Tuple

julia> Tuple.parameters
svec(Vararg{Any,N} where N)
```

Unlike other types, tuple types are covariant in their parameters, so this definition permits `Tuple` to match any type of tuple:

```
julia> typeintersect(Tuple, Tuple{Int,Float64})
Tuple{Int64,Float64}

julia> typeintersect(Tuple{Vararg{Any}}, Tuple{Int,Float64})
Tuple{Int64,Float64}
```

However, if a variadic (`Vararg`) tuple type has free variables it can describe different kinds of tuples:

```
julia> typeintersect(Tuple{Vararg{T} where T}, Tuple{Int,Float64})
Tuple{Int64,Float64}

julia> typeintersect(Tuple{Vararg{T}} where T, Tuple{Int,Float64})
Union{}
```

Notice that when T is free with respect to the `Tuple` type (i.e. its binding `UnionAll` type is outside the `Tuple` type), only one T value must work over the whole type. Therefore a heterogeneous tuple does not match.

Finally, it's worth noting that `Tuple{}` is distinct:

```
julia> Tuple{}
Tuple{}

julia> Tuple{}.parameters
svec()

julia> typeintersect(Tuple{}, Tuple{Int})
Union{}
```

What is the "primary" tuple-type?

```
julia> pointer_from_objref(Tuple)
Ptr{Void} @0x00007f5998a04370

julia> pointer_from_objref(Tuple{})
Ptr{Void} @0x00007f5998a570d0

julia> pointer_from_objref(Tuple.name.wrapper)
Ptr{Void} @0x00007f5998a04370

julia> pointer_from_objref(Tuple{}.name.wrapper)
Ptr{Void} @0x00007f5998a04370
```

so `Tuple == Tuple{Vararg{Any}}` is indeed the primary type.

Diagonal types

Consider the type `Tuple{T,T} where T`. A method with this signature would look like:

```
f(x::T, y::T) where {T} = ...
```

According to the usual interpretation of a `UnionAll` type, this T ranges over all types, including `Any`, so this type should be equivalent to `Tuple{Any,Any}`. However, this interpretation causes some practical problems.

First, a value of T needs to be available inside the method definition. For a call like `f(1, 1.0)`, it's not clear what T should be. It could be `Union{Int,Float64}`, or perhaps `Real`. Intuitively, we expect the declaration `x::T` to mean `T === typeof(x)`. To make sure that invariant holds, we need `typeof(x) === typeof(y) === T` in this method. That implies the method should only be called for arguments of the exact same type.

It turns out that being able to dispatch on whether two values have the same type is very useful (this is used by the promotion system for example), so we have multiple reasons to want a different interpretation of `Tuple{T,T} where T`. To make this work we add the following rule to subtyping: if a variable occurs more than once in covariant position, it is restricted to ranging over only concrete types. ("Covariant position" means that only `Tuple` and `Union` types occur between an occurrence of a variable and the `UnionAll` type that introduces it.) Such variables are called "diagonal variables" or "concrete variables".

So for example, `Tuple{T,T} where T` can be seen as `Union{Tuple{Int8,Int8}, Tuple{Int16,Int16}, ...}`, where `T` ranges over all concrete types. This gives rise to some interesting subtyping results. For example `Tuple{Real,Real}` is not a subtype of `Tuple{T,T} where T`, because it includes some types like `Tuple{Int8,Int16}` where the two elements have different types. `Tuple{Real,Real}` and `Tuple{T,T} where T` have the non-trivial intersection `Tuple{T,T} where T<:Real`. However, `Tuple{Real}` *is* a subtype of `Tuple{T} where T`, because in that case `T` occurs only once and so is not diagonal.

Next consider a signature like the following:

```
f(a::Array{T}, x::T, y::T) where {T} = ...
```

In this case, `T` occurs in invariant position inside `Array{T}`. That means whatever type of array is passed unambiguously determines the value of `T` -- we say `T` has an *equality constraint* on it. Therefore in this case the diagonal rule is not really necessary, since the array determines `T` and we can then allow `x` and `y` to be of any subtypes of `T`. So variables that occur in invariant position are never considered diagonal. This choice of behavior is slightly controversial -- some feel this definition should be written as

```
f(a::Array{T}, x::S, y::S) where {T, S<:T} = ...
```

to clarify whether `x` and `y` need to have the same type. In this version of the signature they would, or we could introduce a third variable for the type of `y` if `x` and `y` can have different types.

The next complication is the interaction of unions and diagonal variables, e.g.

```
f(x::Union{Void,T}, y::T) where {T} = ...
```

Consider what this declaration means. `y` has type `T`. `x` then can have either the same type `T`, or else be of type `Void`. So all of the following calls should match:

```
f(1, 1)
f("", "")
f(2.0, 2.0)
f(nothing, 1)
f(nothing, "")
f(nothing, 2.0)
```

These examples are telling us something: when `x` is `nothing::Void`, there are no extra constraints on `y`. It is as if the method signature had `y::Any`. This means that whether a variable is diagonal is not a static property based on where it appears in a type. Rather, it depends on where a variable appears when the subtyping algorithm *uses* it. When `x` has type `Void`, we don't need to use the `T` in `Union{Void,T}`, so `T` does not "occur". Indeed, we have the following type equivalence:

```
(Tuple{Union{Void,T},T} where T) == Union{Tuple{Void,Any}, Tuple{T,T} where T}
```

Subtyping diagonal variables

The subtyping algorithm for diagonal variables has two components: (1) identifying variable occurrences, and (2) ensuring that diagonal variables range over concrete types only.

The first task is accomplished by keeping counters `occurs_inv` and `occurs_cov` (in `src/subtype.c`) for each variable in the environment, tracking the number of invariant and covariant occurrences, respectively. A variable is diagonal when `occurs_inv == 0 && occurs_cov > 1`.

The second task is accomplished by imposing a condition on a variable's lower bound. As the subtyping algorithm runs, it narrows the bounds of each variable (raising lower bounds and lowering upper bounds) to keep track of the range of variable values for which the subtype relation would hold. When we are done evaluating the body of a `UnionAll` type whose variable is diagonal, we look at the final values of the bounds. Since the variable must be concrete, a contradiction occurs if its lower bound could not be a subtype of a concrete type. For example, an abstract type like `AbstractArray` cannot be a subtype of a concrete type, but a concrete type like `Int` can be, and the empty type `Bottom` can be as well. If a lower bound fails this test the algorithm stops with the answer `false`.

For example, in the problem `Tuple{Int,String} <: Tuple{T,T} where T`, we derive that this would be true if `T` were a supertype of `Union{Int,String}`. However, `Union{Int,String}` is an abstract type, so the relation does not hold.

This concreteness test is done by the function `is_leaf_bound`. Note that this test is slightly different from `jl_is_leaf_type`, since it also returns `true` for `Bottom`. Currently this function is heuristic, and does not catch all possible concrete types. The difficulty is that whether a lower bound is concrete might depend on the values of other type variable bounds. For example, `Vector{T}` is equivalent to the concrete type `Vector{Int}` only if both the upper and lower bounds of `T` equal `Int`. We have not yet worked out a complete algorithm for this.

Introduction to the internal machinery

Most operations for dealing with types are found in the files `jltypes.c` and `subtype.c`. A good way to start is to watch subtyping in action. Build Julia with `make debug` and fire up Julia within a debugger. gdb debugging tips has some tips which may be useful.

Because the subtyping code is used heavily in the REPL itself–and hence breakpoints in this code get triggered often–it will be easiest if you make the following definition:

```
julia> function mysubtype(a,b)

           ccall(:jl_breakpoint, Void, (Any,), nothing)

           issubtype(a, b)

       end
```

and then set a breakpoint in `jl_breakpoint`. Once this breakpoint gets triggered, you can set breakpoints in other functions.

As a warm-up, try the following:

```
mysubtype(Tuple{Int,Float64}, Tuple{Integer,Real})
```

We can make it more interesting by trying a more complex case:

```
mysubtype(Tuple{Array{Int,2}, Int8}, Tuple{Array{T}, T} where T)
```

Subtyping and method sorting

The `type_morespecific` functions are used for imposing a partial order on functions in method tables (from most-to-least specific). Specificity is strict; if a is more specific than b, then a does not equal b and b is not more specific than a.

If a is a strict subtype of b, then it is automatically considered more specific. From there, `type_morespecific` employs some less formal rules. For example, `subtype` is sensitive to the number of arguments, but `type_morespecific` may not be. In particular, `Tuple{Int,AbstractFloat}` is more specific than `Tuple{Integer}`, even though it is not a subtype. (Of `Tuple{Int,AbstractFloat}` and `Tuple{Integer,Float64}`, neither is more specific than the other.) Likewise, `Tuple{Int,Vararg{Int}}` is not a subtype of `Tuple{Integer}`, but it is considered more specific. However, `morespecific` does get a bonus for length: in particular, `Tuple{Int,Int}` is more specific than `Tuple{Int,Vararg{Int}}`.

If you're debugging how methods get sorted, it can be convenient to define the function:

```
type_morespecific(a, b) = ccall(:jl_type_morespecific, Cint, (Any,Any), a, b)
```

which allows you to test whether tuple type a is more specific than tuple type b.

70.4 Memory layout of Julia Objects

Object layout (jl_value_t)

The `jl_value_t` struct is the name for a block of memory owned by the Julia Garbage Collector, representing the data associated with a Julia object in memory. Absent any type information, it is simply an opaque pointer:

```
typedef struct jl_value_t* jl_pvalue_t;
```

Each `jl_value_t` struct is contained in a `jl_typetag_t` struct that contains metadata information about the Julia object, such as its type and garbage collector (gc) reachability:

```
typedef struct {
    opaque metadata;
    jl_value_t value;
} jl_typetag_t;
```

The type of any Julia object is an instance of a leaf `jl_datatype_t` object. The `jl_typeof()` function can be used to query for it:

```
jl_value_t *jl_typeof(jl_value_t *v);
```

The layout of the object depends on its type. Reflection methods can be used to inspect that layout. A field can be accessed by calling one of the get-field methods:

```
jl_value_t *jl_get_nth_field_checked(jl_value_t *v, size_t i);
jl_value_t *jl_get_field(jl_value_t *o, char *fld);
```

If the field types are known, a priori, to be all pointers, the values can also be extracted directly as an array access:

```
jl_value_t *v = value->fieldptr[n];
```

As an example, a "boxed" `uint16_t` is stored as follows:

```
struct {
    opaque metadata;
    struct {
        uint16_t data;        // -- 2 bytes
    } jl_value_t;
};
```

This object is created by `jl_box_uint16()`. Note that the `jl_value_t` pointer references the data portion, not the metadata at the top of the struct.

A value may be stored ”unboxed” in many circumstances (just the data, without the metadata, and possibly not even stored but just kept in registers), so it is unsafe to assume that the address of a box is a unique identifier. The ”egal” test (corresponding to the === function in Julia), should instead be used to compare two unknown objects for equivalence:

```
int jl_egal(jl_value_t *a, jl_value_t *b);
```

This optimization should be relatively transparent to the API, since the object will be ”boxed” on-demand, whenever a `jl_value_t` pointer is needed.

Note that modification of a `jl_value_t` pointer in memory is permitted only if the object is mutable. Otherwise, modification of the value may corrupt the program and the result will be undefined. The mutability property of a value can be queried for with:

```
int jl_is_mutable(jl_value_t *v);
```

If the object being stored is a `jl_value_t`, the Julia garbage collector must be notified also:

```
void jl_gc_wb(jl_value_t *parent, jl_value_t *ptr);
```

However, the Embedding Julia section of the manual is also required reading at this point, for covering other details of boxing and unboxing various types, and understanding the gc interactions.

Mirror structs for some of the built-in types are defined in `julia.h`. The corresponding global `jl_datatype_t` objects are created by `jl_init_types` in `jltypes.c`.

Garbage collector mark bits

The garbage collector uses several bits from the metadata portion of the `jl_typetag_t` to track each object in the system. Further details about this algorithm can be found in the comments of the garbage collector implementation in `gc.c`.

Object allocation

Most new objects are allocated by `jl_new_structv()`:

```
jl_value_t *jl_new_struct(jl_datatype_t *type, ...);
jl_value_t *jl_new_structv(jl_datatype_t *type, jl_value_t **args, uint32_t na);
```

Although, `isbits` objects can be also constructed directly from memory:

```
jl_value_t *jl_new_bits(jl_value_t *bt, void *data)
```

And some objects have special constructors that must be used instead of the above functions:

Types:

```
jl_datatype_t *jl_apply_type(jl_datatype_t *tc, jl_tuple_t *params);
jl_datatype_t *jl_apply_array_type(jl_datatype_t *type, size_t dim);
jl_uniontype_t *jl_new_uniontype(jl_tuple_t *types);
```

While these are the most commonly used options, there are more low-level constructors too, which you can find declared in `julia.h`. These are used in `jl_init_types()` to create the initial types needed to bootstrap the creation of the Julia system image.

Tuples:

```
jl_tuple_t *jl_tuple(size_t n, ...);
jl_tuple_t *jl_tuplev(size_t n, jl_value_t **v);
jl_tuple_t *jl_alloc_tuple(size_t n);
```

The representation of tuples is highly unique in the Julia object representation ecosystem. In some cases, a `Base.tuple()` object may be an array of pointers to the objects contained by the tuple equivalent to:

```
typedef struct {
    size_t length;
    jl_value_t *data[length];
} jl_tuple_t;
```

However, in other cases, the tuple may be converted to an anonymous `isbits` type and stored unboxed, or it may not stored at all (if it is not being used in a generic context as a `jl_value_t*`).

Symbols:

```
jl_sym_t *jl_symbol(const char *str);
```

Functions and MethodInstance:

```
jl_function_t *jl_new_generic_function(jl_sym_t *name);
jl_method_instance_t *jl_new_method_instance(jl_value_t *ast, jl_tuple_t *sparams);
```

Arrays:

```
jl_array_t *jl_new_array(jl_value_t *atype, jl_tuple_t *dims);
jl_array_t *jl_new_arrayv(jl_value_t *atype, ...);
jl_array_t *jl_alloc_array_1d(jl_value_t *atype, size_t nr);
jl_array_t *jl_alloc_array_2d(jl_value_t *atype, size_t nr, size_t nc);
jl_array_t *jl_alloc_array_3d(jl_value_t *atype, size_t nr, size_t nc, size_t z);
jl_array_t *jl_alloc_vec_any(size_t n);
```

Note that many of these have alternative allocation functions for various special-purposes. The list here reflects the more common usages, but a more complete list can be found by reading the `julia.h` header file.

Internal to Julia, storage is typically allocated by `newstruct()` (or `newobj()` for the special types):

```
jl_value_t *newstruct(jl_value_t *type);
jl_value_t *newobj(jl_value_t *type, size_t nfields);
```

And at the lowest level, memory is getting allocated by a call to the garbage collector (in `gc.c`), then tagged with its type:

```
jl_value_t *jl_gc_allocobj(size_t nbytes);
void jl_set_typeof(jl_value_t *v, jl_datatype_t *type);
```

Note that all objects are allocated in multiples of 4 bytes and aligned to the platform pointer size. Memory is allocated from a pool for smaller objects, or directly with `malloc()` for large objects.

Singleton Types

Singleton types have only one instance and no data fields. Singleton instances have a size of 0 bytes, and consist only of their metadata. e.g. `nothing::Void`.

See Singleton Types and Nothingness and missing values

70.5 Eval of Julia code

One of the hardest parts about learning how the Julia Language runs code is learning how all of the pieces work together to execute a block of code.

Each chunk of code typically makes a trip through many steps with potentially unfamiliar names, such as (in no particular order): flisp, AST, C++, LLVM, `eval`, `typeinf`, `macroexpand`, sysimg (or system image), bootstrapping, compile, parse, execute, JIT, interpret, box, unbox, intrinsic function, and primitive function, before turning into the desired result (hopefully).

Definitions

- REPL
  REPL stands for Read-Eval-Print Loop. It's just what we call the command line environment for short.
- AST
  Abstract Syntax Tree The AST is the digital representation of the code structure. In this form the code has been tokenized for meaning so that it is more suitable for manipulation and execution.

Julia Execution

The 10,000 foot view of the whole process is as follows:

1. The user starts `julia`.
2. The C function `main()` from `ui/repl.c` gets called. This function processes the command line arguments, filling in the `jl_options` struct and setting the variable `ARGS`. It then initializes Julia (by calling `julia_init` in `task.c`, which may load a previously compiled sysimg). Finally, it passes off control to Julia by calling `Base._start()`.
3. When `_start()` takes over control, the subsequent sequence of commands depends on the command line arguments given. For example, if a filename was supplied, it will proceed to execute that file. Otherwise, it will start an interactive REPL.
4. Skipping the details about how the REPL interacts with the user, let's just say the program ends up with a block of code that it wants to run.
5. If the block of code to run is in a file, `jl_load(char *filename)` gets invoked to load the file and parse it. Each fragment of code is then passed to `eval` to execute.
6. Each fragment of code (or AST), is handed off to `eval()` to turn into results.
7. `eval()` takes each code fragment and tries to run it in `jl_toplevel_eval_flex()`.
8. `jl_toplevel_eval_flex()` decides whether the code is a "toplevel" action (such as `using` or `module`), which would be invalid inside a function. If so, it passes off the code to the toplevel interpreter.
9. `jl_toplevel_eval_flex()` then expands the code to eliminate any macros and to "lower" the AST to make it simpler to execute.
10. `jl_toplevel_eval_flex()` then uses some simple heuristics to decide whether to JIT compiler the AST or to interpret it directly.
11. The bulk of the work to interpret code is handled by `eval` in `interpreter.c`.

12. If instead, the code is compiled, the bulk of the work is handled by `codegen.cpp`. Whenever a Julia function is called for the first time with a given set of argument types, type inference will be run on that function. This information is used by the codegen step to generate faster code.

13. Eventually, the user quits the REPL, or the end of the program is reached, and the `_start()` method returns.

14. Just before exiting, `main()` calls `jl_atexit_hook(exit_code)`. This calls `Base._atexit()` (which calls any functions registered to `atexit()` inside Julia). Then it calls `jl_gc_run_all_finalizers()`. Finally, it gracefully cleans up all `libuv` handles and waits for them to flush and close.

Parsing

The Julia parser is a small lisp program written in femtolisp, the source-code for which is distributed inside Julia in src/flisp.

The interface functions for this are primarily defined in `jlfrontend.scm`. The code in `ast.c` handles this handoff on the Julia side.

The other relevant files at this stage are `julia-parser.scm`, which handles tokenizing Julia code and turning it into an AST, and `julia-syntax.scm`, which handles transforming complex AST representations into simpler, "lowered" AST representations which are more suitable for analysis and execution.

Macro Expansion

When `eval()` encounters a macro, it expands that AST node before attempting to evaluate the expression. Macro expansion involves a handoff from `eval()` (in Julia), to the parser function `jl_macroexpand()` (written in `flisp`) to the Julia macro itself (written in - what else - Julia) via `fl_invoke_julia_macro()`, and back.

Typically, macro expansion is invoked as a first step during a call to `expand()`/`jl_expand()`, although it can also be invoked directly by a call to `macroexpand()`/`jl_macroexpand()`.

Type Inference

Type inference is implemented in Julia by `typeinf()` in `inference.jl`. Type inference is the process of examining a Julia function and determining bounds for the types of each of its variables, as well as bounds on the type of the return value from the function. This enables many future optimizations, such as unboxing of known immutable values, and compile-time hoisting of various run-time operations such as computing field offsets and function pointers. Type inference may also include other steps such as constant propagation and inlining.

More Definitions

- JIT

  Just-In-Time Compilation The process of generating native-machine code into memory right when it is needed.

- LLVM

  Low-Level Virtual Machine (a compiler) The Julia JIT compiler is a program/library called libLLVM. Codegen in Julia refers both to the process of taking a Julia AST and turning it into LLVM instructions, and the process of LLVM optimizing that and turning it into native assembly instructions.

- C++

  The programming language that LLVM is implemented in, which means that codegen is also implemented in this language. The rest of Julia's library is implemented in C, in part because its smaller feature set makes it more usable as a cross-language interface layer.

- box

  This term is used to describe the process of taking a value and allocating a wrapper around the data that is tracked by the garbage collector (gc) and is tagged with the object's type.

- unbox

  The reverse of boxing a value. This operation enables more efficient manipulation of data when the type of that data is fully known at compile-time (through type inference).

- generic function

  A Julia function composed of multiple "methods" that are selected for dynamic dispatch based on the argument type-signature

- anonymous function or "method"

  A Julia function without a name and without type-dispatch capabilities

- primitive function

  A function implemented in C but exposed in Julia as a named function "method" (albeit without generic function dispatch capabilities, similar to a anonymous function)

- intrinsic function

  A low-level operation exposed as a function in Julia. These pseudo-functions implement operations on raw bits such as add and sign extend that cannot be expressed directly in any other way. Since they operate on bits directly, they must be compiled into a function and surrounded by a call to `Core.Intrinsics.box(T, ...)` to reassign type information to the value.

JIT Code Generation

Codegen is the process of turning a Julia AST into native machine code.

The JIT environment is initialized by an early call to `jl_init_codegen` in `codegen.cpp`.

On demand, a Julia method is converted into a native function by the function `emit_function(jl_method_instance_t*)`. (note, when using the MCJIT (in LLVM v3.4+), each function must be JIT into a new module.) This function recursively calls `emit_expr()` until the entire function has been emitted.

Much of the remaining bulk of this file is devoted to various manual optimizations of specific code patterns. For example, `emit_known_call()` knows how to inline many of the primitive functions (defined in `builtins.c`) for various combinations of argument types.

Other parts of codegen are handled by various helper files:

- `debuginfo.cpp`

  Handles backtraces for JIT functions

- `ccall.cpp`

  Handles the ccall and llvmcall FFI, along with various `abi_*.cpp` files

- `intrinsics.cpp`

  Handles the emission of various low-level intrinsic functions

Bootstrapping

The process of creating a new system image is called "bootstrapping".

The etymology of this word comes from the phrase "pulling oneself up by the bootstraps", and refers to the idea of starting from a very limited set of available functions and definitions and ending with the creation of a full-featured environment.

System Image

The system image is a precompiled archive of a set of Julia files. The `sys.ji` file distributed with Julia is one such system image, generated by executing the file `sysimg.jl`, and serializing the resulting environment (including Types, Functions, Modules, and all other defined values) into a file. Therefore, it contains a frozen version of the `Main`, `Core`, and `Base` modules (and whatever else was in the environment at the end of bootstrapping). This serializer/deserializer is implemented by `jl_save_system_image/jl_restore_system_image` in `dump.c`.

If there is no sysimg file (`jl_options.image_file == NULL`), this also implies that `--build` was given on the command line, so the final result should be a new sysimg file. During Julia initialization, minimal `Core` and `Main` modules are created. Then a file named `boot.jl` is evaluated from the current directory. Julia then evaluates any file given as a command line argument until it reaches the end. Finally, it saves the resulting environment to a "sysimg" file for use as a starting point for a future Julia run.

70.6 Calling Conventions

Julia uses three calling conventions for four distinct purposes:

Name	Prefix	Purpose
Native	`julia_`	Speed via specialized signatures
JL Call	`jlcall_`	Wrapper for generic calls
JL Call	`jl_`	Builtins
C ABI	`jlcapi_`	Wrapper callable from C

Julia Native Calling Convention

The native calling convention is designed for fast non-generic calls. It usually uses a specialized signature.

- LLVM ghosts (zero-length types) are omitted.
- LLVM scalars and vectors are passed by value.
- LLVM aggregates (arrays and structs) are passed by reference.

A small return values is returned as LLVM return values. A large return values is returned via the "structure return" (`sret`) convention, where the caller provides a pointer to a return slot.

An argument or return values thta is a homogeneous tuple is sometimes represented as an LLVM vector instead of an LLVM array.

JL Call Convention

The JL Call convention is for builtins and generic dispatch. Hand-written functions using this convention are declared via the macro `JL_CALLABLE`. The convention uses exactly 3 parameters:

- F - Julia representation of function that is being applied
- `args` - pointer to array of pointers to boxes
- `nargs` - length of the array

The return value is a pointer to a box.

C ABI

C ABI wrappers enable calling Julia from C. The wrapper calls a function using the native calling convention.

Tuples are always represented as C arrays.

70.7 High-level Overview of the Native-Code Generation Process

Representation of Pointers

When emitting code to an object file, pointers will be emitted as relocations. The deserialization code will ensure any object that pointed to one of these constants gets recreated and contains the right runtime pointer.

Otherwise, they will be emitted as literal constants.

To emit one of these objects, call `literal_pointer_val`. It'll handle tracking the Julia value and the LLVM global, ensuring they are valid both for the current runtime and after deserialization.

When emitted into the object file, these globals are stored as references in a large `gvals` table. This allows the deserializer to reference them by index, and implement a custom manual mechanism similar to a Global Offset Table (GOT) to restore them.

Function pointers are handled similarly. They are stored as values in a large `fvals` table. Like globals, this allows the deserializer to reference them by index.

Note that `extern` functions are handled separately, with names, via the usual symbol resolution mechanism in the linker.

Note too that `ccall` functions are also handled separately, via a manual GOT and Procedure Linkage Table (PLT).

Representation of Intermediate Values

Values are passed around in a `jl_cgval_t` struct. This represents an R-value, and includes enough information to determine how to assign or pass it somewhere.

They are created via one of the helper constructors, usually: `mark_julia_type` (for immediate values) and `mark_julia_slot` (for pointers to values).

The function `convert_julia_type` can transform between any two types. It returns an R-value with `cgval.typ` set to `typ`. It'll cast the object to the requested representation, making heap boxes, allocating stack copies, and computing tagged unions as needed to change the representation.

By contrast `update_julia_type` will change `cgval.typ` to `typ`, only if it can be done at zero-cost (i.e. without emitting any code).

Union representation

Inferred union types may be stack allocated via a tagged type representation.

The primitive routines that need to be able to handle tagged unions are:

- mark-type
- load-local
- store-local
- isa

- is
- emit_typeof
- emit_sizeof
- boxed
- unbox
- specialized cc-ret

Everything else should be possible to handle in inference by using these primitives to implement union-splitting.

The representation of the tagged-union is as a pair of `< void* union, byte selector >`. The selector is fixed-size as `byte & 0x7f`, and will union-tag the first 126 isbits. It records the one-based depth-first count into the type-union of the isbits objects inside. An index of zero indicates that the `union*` is actually a tagged heap-allocated `jl_value_t*`, and needs to be treated as normal for a boxed object rather than as a tagged union.

The high bit of the selector (`byte & 0x80`) can be tested to determine if the `void*` is actually a heap-allocated (`jl_value_t*`) box, thus avoiding the cost of re-allocating a box, while maintaining the ability to efficiently handle union-splitting based on the low bits.

It is guaranteed that `byte & 0x7f` is an exact test for the type, if the value can be represented by a tag – it will never be marked `byte = 0x80`. It is not necessary to also test the type-tag when testing `isa`.

The `union*` memory region may be allocated at *any* size. The only constraint is that it is big enough to contain the data currently specified by `selector`. It might not be big enough to contain the union of all types that could be stored there according to the associated Union type field. Use appropriate care when copying.

Specialized Calling Convention Signature Representation

A `jl_returninfo_t` object describes the calling convention details of any callable.

If any of the arguments or return type of a method can be represented unboxed, and the method is not varargs, it'll be given an optimized calling convention signature based on its `specTypes` and `rettype` fields.

The general principles are that:

- Primitive types get passed in int/float registers.
- Tuples of VecElement types get passed in vector registers.
- Structs get passed on the stack.
- Return values are handle similarly to arguments, with a size-cutoff at which they will instead be returned via a hidden sret argument.

The total logic for this is implemented by `get_specsig_function` and `deserves_sret`.

Additionally, if the return type is a union, it may be returned as a pair of values (a pointer and a tag). If the union values can be stack-allocated, then sufficient space to store them will also be passed as a hidden first argument. It is up to the callee whether the returned pointer will point to this space, a boxed object, or even other constant memory.

70.8 Julia Functions

This document will explain how functions, method definitions, and method tables work.

Method Tables

Every function in Julia is a generic function. A generic function is conceptually a single function, but consists of many definitions, or methods. The methods of a generic function are stored in a method table. Method tables (type `MethodTable`) are associated with `TypeNames`. A `TypeName` describes a family of parameterized types. For example `Complex{Float32}` and `Complex{Float64}` share the same `Complex` type name object.

All objects in Julia are potentially callable, because every object has a type, which in turn has a `TypeName`.

Function calls

Given the call `f(x,y)`, the following steps are performed: first, the method table to use is accessed as `typeof(f).name.mt`. Second, an argument tuple type is formed, `Tuple{typeof(f), typeof(x), typeof(y)}`. Note that the type of the function itself is the first element. This is because the type might have parameters, and so needs to take part in dispatch. This tuple type is looked up in the method table.

This dispatch process is performed by `jl_apply_generic`, which takes two arguments: a pointer to an array of the values f, x, and y, and the number of values (in this case 3).

Throughout the system, there are two kinds of APIs that handle functions and argument lists: those that accept the function and arguments separately, and those that accept a single argument structure. In the first kind of API, the "arguments" part does *not* contain information about the function, since that is passed separately. In the second kind of API, the function is the first element of the argument structure.

For example, the following function for performing a call accepts just an `args` pointer, so the first element of the args array will be the function to call:

```
jl_value_t *jl_apply(jl_value_t **args, uint32_t nargs)
```

This entry point for the same functionality accepts the function separately, so the `args` array does not contain the function:

```
jl_value_t *jl_call(jl_function_t *f, jl_value_t **args, int32_t nargs);
```

Adding methods

Given the above dispatch process, conceptually all that is needed to add a new method is (1) a tuple type, and (2) code for the body of the method. `jl_method_def` implements this operation. `jl_first_argument_datatype` is called to extract the relevant method table from what would be the type of the first argument. This is much more complicated than the corresponding procedure during dispatch, since the argument tuple type might be abstract. For example, we can define:

```
(::Union{Foo{Int},Foo{Int8}})(x) = 0
```

which works since all possible matching methods would belong to the same method table.

Creating generic functions

Since every object is callable, nothing special is needed to create a generic function. Therefore `jl_new_generic_function` simply creates a new singleton (0 size) subtype of `Function` and returns its instance. A function can have a mnemonic "display name" which is used in debug info and when printing objects. For example the name of `Base.sin` is `sin`. By convention, the name of the created *type* is the same as the function name, with a # prepended. So `typeof(sin)` is `Base.#sin`.

Closures

A closure is simply a callable object with field names corresponding to captured variables. For example, the following code:

```
function adder(x)
    return y->x+y
end
```

is lowered to (roughly):

```
struct ##1{T}
    x::T
end

(_::##1)(y) = _.x + y

function adder(x)
    return ##1(x)
end
```

Constructors

A constructor call is just a call to a type. The type of most types is `DataType`, so the method table for `DataType` contains most constructor definitions. One wrinkle is the fallback definition that makes all types callable via `convert`:

```
(::Type{T}){T}(args...) = convert(T, args...)::T
```

In this definition the function type is abstract, which is not normally supported. To make this work, all subtypes of `Type` (`Type`, `UnionAll`, `Union`, and `DataType`) currently share a method table via special arrangement.

Builtins

The "builtin" functions, defined in the `Core` module, are:

```
=== typeof sizeof issubtype isa typeassert throw tuple getfield setfield! fieldtype
nfields isdefined arrayref arrayset arraysize applicable invoke apply_type _apply
_expr svec
```

These are all singleton objects whose types are subtypes of `Builtin`, which is a subtype of `Function`. Their purpose is to expose entry points in the run time that use the "jlcall" calling convention:

```
jl_value_t *(jl_value_t*, jl_value_t**, uint32_t)
```

The method tables of builtins are empty. Instead, they have a single catch-all method cache entry (`Tuple{Vararg{Any}}`) whose jlcall fptr points to the correct function. This is kind of a hack but works reasonably well.

Keyword arguments

Keyword arguments work by associating a special, hidden function object with each method table that has definitions with keyword arguments. This function is called the "keyword argument sorter" or "keyword sorter", or "kwsorter", and is stored in the `kwsorter` field of `MethodTable` objects. Every definition in the kwsorter function has the same arguments as some definition in the normal method table, except with a single `Array` argument prepended. This array contains alternating symbols and values that represent the passed keyword arguments. The kwsorter's job is to move

keyword arguments into their canonical positions based on name, plus evaluate and substite any needed default value expressions. The result is a normal positional argument list, which is then passed to yet another function.

The easiest way to understand the process is to look at how a keyword argument method definition is lowered. The code:

```
function circle(center, radius; color = black, fill::Bool = true, options...)
    # draw
end
```

actually produces *three* method definitions. The first is a function that accepts all arguments (including keywords) as positional arguments, and includes the code for the method body. It has an auto-generated name:

```
function #circle#1(color, fill::Bool, options, circle, center, radius)
    # draw
end
```

The second method is an ordinary definition for the original `circle` function, which handles the case where no keyword arguments are passed:

```
function circle(center, radius)
    #circle#1(black, true, Any[], circle, center, radius)
end
```

This simply dispatches to the first method, passing along default values. Finally there is the kwsorter definition:

```
function (::Core.kwftype(typeof(circle)))(kw::Array, circle, center, radius)
    options = Any[]
    color = arg associated with :color, or black if not found
    fill = arg associated with :fill, or true if not found
    # push remaining elements of kw into options array
    #circle#1(color, fill, options, circle, center, radius)
end
```

The front end generates code to loop over the `kw` array and pick out arguments in the right order, evaluating default expressions when an argument is not found.

The function `Core.kwftype(t)` fetches (and creates, if necessary) the field `t.name.mt.kwsorter`.

This design has the feature that call sites that don't use keyword arguments require no special handling; everything works as if they were not part of the language at all. Call sites that do use keyword arguments are dispatched directly to the called function's kwsorter. For example the call:

```
circle((0,0), 1.0, color = red; other...)
```

is lowered to:

```
kwfunc(circle)(Any[:color,red,other...], circle, (0,0), 1.0)
```

The unpacking procedure represented here as `other...` actually further unpacks each *element* of `other`, expecting each one to contain two values (a symbol and a value). `kwfunc` (also in `Core`) fetches the kwsorter for the called function. Notice that the original `circle` function is passed through, to handle closures.

Compiler efficiency issues

Generating a new type for every function has potentially serious consequences for compiler resource use when combined with Julia's "specialize on all arguments by default" design. Indeed, the initial implementation of this design suffered from much longer build and test times, higher memory use, and a system image nearly 2x larger than the baseline. In a naive implementation, the problem is bad enough to make the system nearly unusable. Several significant optimizations were needed to make the design practical.

The first issue is excessive specialization of functions for different values of function-valued arguments. Many functions simply "pass through" an argument to somewhere else, e.g. to another function or to a storage location. Such functions do not need to be specialized for every closure that might be passed in. Fortunately this case is easy to distinguish by simply considering whether a function *calls* one of its arguments (i.e. the argument appears in "head position" somewhere). Performance-critical higher-order functions like `map` certainly call their argument function and so will still be specialized as expected. This optimization is implemented by recording which arguments are called during the `analyze-variables` pass in the front end. When `cache_method` sees an argument in the `Function` type hierarchy passed to a slot declared as `Any` or `Function`, it pretends the slot was declared as `ANY` (the "don't specialize" hint). This heuristic seems to be extremely effective in practice.

The next issue concerns the structure of method cache hash tables. Empirical studies show that the vast majority of dynamically-dispatched calls involve one or two arguments. In turn, many of these cases can be resolved by considering only the first argument. (Aside: proponents of single dispatch would not be surprised by this at all. However, this argument means "multiple dispatch is easy to optimize in practice", and that we should therefore use it, *not* "we should use single dispatch"!) So the method cache uses the type of the first argument as its primary key. Note, however, that this corresponds to the *second* element of the tuple type for a function call (the first element being the type of the function itself). Typically, type variation in head position is extremely low – indeed, the majority of functions belong to singleton types with no parameters. However, this is not the case for constructors, where a single method table holds constructors for every type. Therefore the `Type` method table is special-cased to use the *first* tuple type element instead of the second.

The front end generates type declarations for all closures. Initially, this was implemented by generating normal type declarations. However, this produced an extremely large number of constructors, all of which were trivial (simply passing all arguments through to `new`). Since methods are partially ordered, inserting all of these methods is O(n^2), plus there are just too many of them to keep around. This was optimized by generating `composite_type` expressions directly (bypassing default constructor generation), and using `new` directly to create closure instances. Not the prettiest thing ever, but you do what you gotta do.

The next problem was the `@test` macro, which generated a 0-argument closure for each test case. This is not really necessary, since each test case is simply run once in place. Therefore I modified `@test` to expand to a try-catch block that records the test result (true, false, or exception raised) and calls the test suite handler on it.

However this caused a new problem. When many tests are grouped together in a single function, e.g. a single top level expression, or some other test grouping function, that function could have a very large number of exception handlers. This triggered a kind of dataflow analysis worst case, where type inference spun around for minutes enumerating possible paths through the forest of handlers. This was fixed by simply bailing out of type inference when it encounters more than some number of handlers (currently 25). Presumably no performance-critical function will have more than 25 exception handlers. If one ever does, I'm willing to raise the limit to 26.

A minor issue occurs during the bootstrap process due to storing all constructors in a single method table. In the second bootstrap step, where `inference.ji` is compiled using `inference0.ji`, constructors for `inference0`'s types remain in the table, so there are still references to the old inference module and `inference.ji` is 2x the size it should be. This was fixed in `dump.c` by filtering definitions from "replaced modules" out of method tables and caches before saving a system image. A "replaced module" is one that satisfies the condition `m != jl_get_global(m->parent, m->name)` – in other words, some newer module has taken its name and place.

Another type inference worst case was triggered by the following code from the QuadGK.jl package, formerly part of Base:

```
function do_quadgk(f, s, n, ::Type{Tw}, abstol, reltol, maxevals, nrm) where Tw
    if eltype(s) <: Real # check for infinite or semi-infinite intervals
        s1 = s[1]; s2 = s[end]; inf1 = isinf(s1); inf2 = isinf(s2)
        if inf1 || inf2
            if inf1 && inf2 # x = t/(1-t^2) coordinate transformation
                return do_quadgk(t -> begin t2 = t*t; den = 1 / (1 - t2);
                                            f(t*den) * (1+t2)*den*den; end,
                                 map(x -> isinf(x) ? copysign(one(x), x) : 2x / (1+hypot(1,2x)),
                                 ↪  s),
                                 n, Tw, abstol, reltol, maxevals, nrm)
            end
            s0,si = inf1 ? (s2,s1) : (s1,s2)
            if si < 0 # x = s0 - t/(1-t)
                return do_quadgk(t -> begin den = 1 / (1 - t);
                                            f(s0 - t*den) * den*den; end,
                                 reverse!(map(x -> 1 / (1 + 1 / (s0 - x)), s)),
                                 n, Tw, abstol, reltol, maxevals, nrm)
            else # x = s0 + t/(1-t)
                return do_quadgk(t -> begin den = 1 / (1 - t);
                                            f(s0 + t*den) * den*den; end,
                                 map(x -> 1 / (1 + 1 / (x - s0)), s),
                                 n, Tw, abstol, reltol, maxevals, nrm)
            end
        end
    end
```

This code has a 3-way tail recursion, where each call wraps the current function argument `f` in a different new closure. Inference must consider 3^n (where n is the call depth) possible signatures. This blows up way too quickly, so logic was added to `typeinf_uncached` to immediately widen any argument that is a subtype of `Function` and that grows in depth down the stack.

70.9 Base.Cartesian

El módulo `Cartesian` (no exportado) proporciona macros que facilitan escribir algoritmos multidimensionales. Es deseable que, a largo plazo, este módulo `Cartesian` no sea necesario; sin embargo, en la actualidad es una de las pocas formas de escribir código multidimensional compacto y con rendimiento.

Principios de uso

Un ejemplo de uso simple podría ser:

```
@nloops 3 i A begin
    s += @nref 3 A i
end
```

que genera el siguiente código:

```
for i_3 = 1:size(A,3)
    for i_2 = 1:size(A,2)
        for i_1 = 1:size(A,1)
            s += A[i_1,i_2,i_3]
        end
    end
end
```

En general, `Cartesian` permitirá escribir código que contiene elementos repetitivos, como los bucles anidados de este ejemplo. Otras aplicaciones incluyen expresiones repetidas (por ejemplo, desenrollado de bucles) o crear llamadas a función con números variables de argumentos sin usar la construcción "*splat*" (`i...`).

Sintaxis Básica

La sintaxis básica de `@nloops` es la siguiente:

* El primer argumento debe ser un entero (*no* una variable) que especifica el número de bucles. * El segundo argumento es el prefijo simbólico que se utilizará para la variable iteradora. De este modo, en el ejemplo anterior usamos `i`, y se generaron las variables `i_1, i_2, i_3`. * El tercer argumento especifica el rango para cada variable iteradora. Si se usa una variable (símbolo) aquí, es considerado como `1:size(A,dim)`. De forma más flexible, se puede usar la sintaxis de expresiones basadas en funciones anónimas que se decribe más adelante. * El último argumento es el cuerpo del bucle. En el ejemplo anterior, lo que aparece entre `begin...end`.

Hay otras características adicionales de `@nloops` descritas en la sección de referencia.

`@nref` sigue un patrn similar, generando `A[i_1,i_2,i_3]` a partir de `@nref 3 A i`. La práctica general es leer de izquierda a derecha, por lo que `@nloops` es `@nloops 3 i A expr` (como en el bucle `for i_2 = 1:size(A,2)`, donde `i_2` está a la izquierda y el rango a la derecha) mientras que `@nref` es `@nref 3 A i` (como en `A[i_1,i_2,i_3]`, donde el array va primero).

Si ests desarrollando código con Cartesian, puedes encontrar que depurar es más sencillo cuando examinas el código generado, usando `macroexpand`:

```
julia> macroexpand(:(@nref 2 A i))
:(A[i_1, i_2])
```

Proporcionando el número de expresiones

El primer argumentos de estas dos macros es el número de expresiones, que debe ser un entero. Cuando estás escribiendo una funcin que pretendes que trabaje en múltiples dimensiones, esto puede no ser algo que desees codificar. Si estás escribiendo código que necesitas que trabaje con versiones antiguas de Julia, deberías usar la macro `@ngenerate` descrita en una versión más antigua de esta documentación.

Empezando en Julia 0.4-pre, el enfoque recomendado es usar una `@generated function`. He aquí un ejemplo:

```
@generated function mysum(A::Array{T,N}) where {T,N}
    quote
        s = zero(T)
        @nloops $N i A begin
            s += @nref $N A i
        end
        s
    end
end
```

Naturalmente, también podemos preparar expresiones o realizar cálculos antes del bloque `quote`.

Expresiones función anónima como argumentos de macros

Quizás la característica más potente de `Cartesian` es la capacidad de proporcionar expresiones función-anónima que son evaluadas en tiempo de análisis sintáctico. Consideremos un ejemplo sencillo:

```
@nexprs 2 j->(i_j = 1)
```

`@nexprs` genera n expresiones que siguen un patrón. Este código generaría las siguientes instrucciones:

```
i_1 = 1
i_2 = 1
```

En cada instrucción generada un `j` aislado (la variable de la función anónima) es reemplazada por valores en el rango `1:2`. Hablando de forma general, Cartesian emplea una sintaxis parecida a LaTeX. Esto te permite hacer operaciones sobre el índice `j`. He aquí un ejemplo que calcula los pasos de un array:

```
s_1 = 1
@nexprs 3 j->(s_{j+1} = s_j * size(A, j))
```

generará las expresiones

```
s_1 = 1
s_2 = s_1 * size(A, 1)
s_3 = s_2 * size(A, 2)
s_4 = s_3 * size(A, 3)
```

Las expresiones función anónima tienen muchos usos en la práctica.

Referencia de las Macros `Base.Cartesian.@nloops` – Macro.

```
@nloops N itersym rangeexpr bodyexpr
@nloops N itersym rangeexpr preexpr bodyexpr
@nloops N itersym rangeexpr preexpr postexpr bodyexpr
```

Generate `N` nested loops, using `itersym` as the prefix for the iteration variables. `rangeexpr` may be an anonymous-function expression, or a simple symbol `var` in which case the range is `indices(var, d)` for dimension `d`.

Optionally, you can provide "pre" and "post" expressions. These get executed first and last, respectively, in the body of each loop. For example:

```
@nloops 2 i A d -> j_d = min(i_d, 5) begin
    s += @nref 2 A j
end
```

would generate:

```
for i_2 = indices(A, 2)
    j_2 = min(i_2, 5)
    for i_1 = indices(A, 1)
        j_1 = min(i_1, 5)
        s += A[j_1, j_2]
    end
end
```

If you want just a post-expression, supply `nothing` for the pre-expression. Using parentheses and semicolons, you can supply multi-statement expressions.

`source`

`Base.Cartesian.@nref` – Macro.

```
@nref N A indexexpr
```

Generate expressions like `A[i_1, i_2, ...]`. `indexexpr` can either be an iteration-symbol prefix, or an anonymous-function expression.

```
julia> @macroexpand Base.Cartesian.@nref 3 A i
:(A[i_1, i_2, i_3])
```

source

`Base.Cartesian.@nextract` – Macro.

```
@nextract N esym isym
```

Generate `N` variables `esym_1`, `esym_2`, ..., `esym_N` to extract values from `isym`. `isym` can be either a `Symbol` or anonymous-function expression.

`@nextract 2 x y` would generate

```
x_1 = y[1]
x_2 = y[2]
```

while `@nextract 3 x d->y[2d-1]` yields

```
x_1 = y[1]
x_2 = y[3]
x_3 = y[5]
```

source

`Base.Cartesian.@nexprs` – Macro.

```
@nexprs N expr
```

Generate `N` expressions. `expr` should be an anonymous-function expression.

```
julia> @macroexpand Base.Cartesian.@nexprs 4 i -> y[i] = A[i+j]
quote
    y[1] = A[1 + j]
    y[2] = A[2 + j]
    y[3] = A[3 + j]
    y[4] = A[4 + j]
end
```

source

`Base.Cartesian.@ncall` – Macro.

```
@ncall N f sym...
```

Generate a function call expression. `sym` represents any number of function arguments, the last of which may be an anonymous-function expression and is expanded into `N` arguments.

For example `@ncall 3 func a` generates

```
func(a_1, a_2, a_3)
```

while `@ncall 2 func a b i->c[i]` yields

```
func(a, b, c[1], c[2])
```

source

Base.Cartesian.@ntuple – Macro.

```
@ntuple N expr
```

Generates an N-tuple. `@ntuple 2 i` would generate `(i_1, i_2)`, and `@ntuple 2 k->k+1` would generate `(2,3)`.

source

Base.Cartesian.@nall – Macro.

```
@nall N expr
```

Check whether all of the expressions generated by the anonymous-function expression `expr` evaluate to `true`.

`@nall 3 d->(i_d > 1)` would generate the expression `(i_1 > 1 && i_2 > 1 && i_3 > 1)`. This can be convenient for bounds-checking.

source

Base.Cartesian.@nany – Macro.

```
@nany N expr
```

Check whether any of the expressions generated by the anonymous-function expression `expr` evaluate to `true`.

`@nany 3 d->(i_d > 1)` would generate the expression `(i_1 > 1 || i_2 > 1 || i_3 > 1)`.

source

Base.Cartesian.@nif – Macro.

```
@nif N conditionexpr expr
@nif N conditionexpr expr elseexpr
```

Generates a sequence of `if ... elseif ... else ... end` statements. For example:

```
@nif 3 d->(i_d >= size(A,d)) d->(error("Dimension ", d, " too big")) d->println("All OK")
```

would generate:

```
if i_1 > size(A, 1)
    error("Dimension ", 1, " too big")
elseif i_2 > size(A, 2)
    error("Dimension ", 2, " too big")
else
    println("All OK")
end
```

source

70.10 Talking to the compiler (the `:meta` mechanism)

In some circumstances, one might wish to provide hints or instructions that a given block of code has special properties: you might always want to inline it, or you might want to turn on special compiler optimization passes. Starting with version 0.4, Julia has a convention that these instructions can be placed inside a `:meta` expression, which is typically (but not necessarily) the first expression in the body of a function.

`:meta` expressions are created with macros. As an example, consider the implementation of the `@inline` macro:

```
macro inline(ex)
    esc(isa(ex, Expr) ? pushmeta!(ex, :inline) : ex)
end
```

Here, ex is expected to be an expression defining a function. A statement like this:

```
@inline function myfunction(x)
    x*(x+3)
end
```

gets turned into an expression like this:

```
quote
    function myfunction(x)
        Expr(:meta, :inline)
        x*(x+3)
    end
end
```

`Base.pushmeta!(ex, :symbol, args...)` appends `:symbol` to the end of the `:meta` expression, creating a new `:meta` expression if necessary. If `args` is specified, a nested expression containing `:symbol` and these arguments is appended instead, which can be used to specify additional information.

To use the metadata, you have to parse these `:meta` expressions. If your implementation can be performed within Julia, `Base.popmeta!` is very handy: `Base.popmeta!(body, :symbol)` will scan a function *body* expression (one without the function signature) for the first `:meta` expression containing `:symbol`, extract any arguments, and return a tuple `(found::Bool, args::Array{Any})`. If the metadata did not have any arguments, or `:symbol` was not found, the `args` array will be empty.

Not yet provided is a convenient infrastructure for parsing `:meta` expressions from C++.

70.11 SubArrays

El tipo `SubArray` de Julia es un contenedor que codifica una "vista" de un `AbstractArray` padre. Esta pagina documenta algunos de los principios de diseño e implementación de `SubArray`.

Indexación: indexación cartesiana vs. lineal

Ampliamente hablando, hay dos formas principales de acceder a los datos en un array. La primera, frecuentemente llamada indexación cartesiana, usa `N` índices para un `AbstractArray` `N` -dimensional. Por ejemlo, una matriz `A` (bidimensional) puede ser indexada en estilo cartesiano como `A[i,j]`. El segundo método de indexación, denominado indexación lineal, usa un solo índice incluso para objetos de mayor dimensión. Por ejemplo si `A = reshape(1:12, 3, 4)`, la expresión `A[5]` devuelve el valor 5. Julia nos permite combinar estos estilos de indexación: por ejemplo, un array 3d `A3` puede ser indexado como `A3[i,j]`, en cuyo caso `i` es interpretado como un índice cartesiano para la primera dimensión, y `j` es un índice lineal sobre las dimensiones 2 y 3.

Para los `Arrays`, la indexación lineal apela al formato subyacente de almacenamiento: un array se presenta como un bloque contiguo de memoria, y por tanto el índice lineal es justo el desplazamiento (+1) de la correspondiente entrada relativa al principio del array. Sin embargo, esto no es cierto para muchos otros tipos `AbstractArray`: ejemplos de ello incluyen `SparseMatrixCSC`, unos arrays que requieren alguna clase de cálculo (tal como interpolación), y el tipo bajo discusión aquí, `SubArray`. Para estos tipos, la información subyacente es descrita más naturalmente en términos de índices cartesianos.

Uno puede convertir manualmente un índice cartesiano a uno lineal con `sub2ind`, y viceversa usando ìnd2sub`. Las funcionesgetindexandsetindex!de los tiposAbstractArray‘ puden incluir operaciones similares.

Aunque convertir de un índice cartesiano a uno lineal es rápido (es justo una multiplicación y una suma), convertir de un índice lineal a uno cartesiano es muy lento: se basa en la operación `div`, que es una de las operacions de bajo nivel más lentas que uno puede realizar con una CPU. Por esta razón, cualquier código que trate con tipos `AbstractArray` está mejor diseñado en términos de indexación cartesiana en lugar de lineal.

Reemplazo de Índices

Considere hacer rebanadas bidimensionales de un array tridimensional:

```
S1 = view(A, :, 5, 2:6)
S2 = view(A, 5, :, 2:6)
```

`view` elimina las dimensiones "singleton" (las que están especificadas por un `Int`), por lo que tanto `S1` como `S2` son `SubArrays` bidimensionales. En consecuencia, el camino natural para indexar esto es con `S1[i,j]`. Para extraer el valor del array padre `A`, el enfoque natural es reemplazar `S1[i,j]` con `A[i,5,(2:6)[j]]` y `S2[i,j]` con `A[5,i,(2:6)[j]]`.

La característica clave del diseño de SubArrays es que este reemplazo de índices puede realizarse sin ninguna sobrecarga en tiempo de ejecución.

Diseño de SubArray's

Parámetros de Tipo y Campos

La estrategia adoptada está expresada en la definición del tipo:

```
struct SubArray{T,N,P,I,L} <: AbstractArray{T,N}
    parent::P
    indexes::I
    offset1::Int       # for linear indexing and pointer, only valid when L==true
    stride1::Int       # used only for linear indexing
    ...
end
```

`SubArray` tiene cinco parámetros de tipo. Los dos primeros son el tipo de elemento estándar y la dimensionalidad. La siguiente es el tipo del `AbstractArray` padre. El usado más intensamente es el cuarto parámetro, una `Tuple` de los tipos de los índices para cada dimensión. El final, `L`, es sólo proporcionado como una conveniencia para el despacho; es un valor booleano que representa si los tipos del índice soportan indexacion lineal rápida. Más sobre este tema después.

Si en nuestro ejemplo de arriba `A` es un `Array{Float64, 3}`, nuestro caso `S1` sería un `SubArray{Int64,2,Array{Int64,3},Tuple{Colon,Int64,UnitRange{Int64}},false}`. Note en particular el parámetro tupla, que almacena los tipos de los índices usados para crear `S1`. Igualmente,

```
julia> S1.indexes
(Colon(),5,2:6)
```

Almacenar estos valores permite el reemplzao de índices, y tener los tipos codificados como parámetros permite a uno despachar a eficientes algoritmos.

Traducción de Índices

Realizar la traducción de índices requiere que uno haga diferentes cosas para diferentes tipos concretos de `SubArray`. Por ejemplo, para S1 uno necesita aplicar los índices `i, j` a las dimensiones primera y tercera del array padre, mientras que para S2 uno necesita aplicarlas a la segunda y la tercera. El enfoque más sencillo a indexar sería hacer el análisis de tipos en tiempo de ejecución:

```
parentindexes = Array{Any}(0)
for thisindex in S.indexes
    ...
    if isa(thisindex, Int)
        # Don't consume one of the input indexes
        push!(parentindexes, thisindex)
    elseif isa(thisindex, AbstractVector)
        # Consume an input index
        push!(parentindexes, thisindex[inputindex[j]])
        j += 1
    elseif isa(thisindex, AbstractMatrix)
        # Consume two input indices
        push!(parentindexes, thisindex[inputindex[j], inputindex[j+1]])
        j += 2
    elseif ...
end
S.parent[parentindexes...]
```

Desgraciadamente, esto sería desastroso en términos de rendimiento: cada acceso a elemento asignaría memoria, e implicaría la ejecución de un montón de código pobremente tipado.

El mejor enfoque es despachar a métodos específicos para manejar cada tipo de índice almacenado. Esto es lo que hace `reindex`: él despacha sobre el tipo del primer índice almacenado y consume el número apropiado de índices de entrada, y entonces recurre sobre los índices restantes. En el caso de S1, esto expande a

```
Base.reindex(S1, S1.indexes, (i, j)) == (i, S1.indexes[2], S1.indexes[3][j])
```

para cualquier par de índices `(i, j)` (excepto `CartesianIndex`s and arrays de este tipo, ver abajo).

Este es el núcleo de un `SubArray`; los métodos de indexación se basan en `reindex` para hacer esta traducción de índices. Sin embargo, algunas veces, podemos evitar la indirección y hacerlo incluso más rápido.

Indexación Lineal

La indexación lineal puede implementarse de forma eficiente cuando el array completo tiene un solo paso que separe elementos sucesivos, empezando desde cierto desplazamiento. Esto significa que nosotros pre-computamos estos valores y representamos la indexación lineal simplemente como una adición y multiplicación, evitando la indirección de `reindex` y (lo que es más importante) la computación lenta de las coordenadas cartesianas por completo.

Para los tipos `SubArray`, la disponibilidad de una indexación lineal eficiente está basada puramente en los tipos de los índices, y no depende de valores como el tamaño de array padre. Uno puede preguntar si un cnjunto de índices dado soporta indexación lineal rápida con la función interna `Base.viewindexing`:

```
julia> Base.viewindexing(S1.indexes)
IndexCartesian()

julia> Base.viewindexing(S2.indexes)
IndexLinear()
```

Esto se calcula durante la construcción del `SubArray` y se almacena en el parámetro de tipo `L` como un boolean que codifica soporte de indexación lineal rápido. Aunque n oes estrictamente necesario, esto significa qeu podemos definir despacho directamente sobre `SubArray{T,N,A,I,true}` sin intermediarios.

Como esta computación no depende de valores en tiempo de ejecución, puede perder algunos casos en los que el paso sea uniforme:

```
julia> A = reshape(1:4*2, 4, 2)
4×2 Base.ReshapedArray{Int64,2,UnitRange{Int64},Tuple{}}:
 1  5
 2  6
 3  7
 4  8

julia> diff(A[2:2:4,:][:])
3-element Array{Int64,1}:
 2
 2
 2
```

Una vista construída como `view(A, 2:2:4, :)` tiene un paso uniforme y, por tanto la indexación lineal podría llevarse a cabo eficientemente. Sin embargo, el éxito en este caso depende del tamaño del array: Si, a diferencia del caso anterior, la primera dimensión fuera impar,

```
julia> A = reshape(1:5*2, 5, 2)
5×2 Base.ReshapedArray{Int64,2,UnitRange{Int64},Tuple{}}:
 1   6
 2   7
 3   8
 4   9
 5  10

julia> diff(A[2:2:4,:][:])
3-element Array{Int64,1}:
 2
 3
 2
```

entonces `A[2:2:4,:]` no tiene un paso uniforme, por lo que no podemos garantizar indexación lineal eficiente. Como tenemos que basar esta decisión puramente en los tipos codificados en los parámetros del `SubArray`, `S = view(A, 2:2:4, :)` no puede implementar una indexación lienal eficiente.

Unos pocos detalles

- Note que la función `Base.reindex` es agnóstica a los tipos de los índices de entrada; ella simplemente determina como y donde deberían reindexarse los indices almacenados. Ella no solo soporta indices enteros,

sino que también soporta indexación no escalar. Esto significa que las vistas de vistas no necesitan dos niveles de indirección; ellas pueden simplemente recomputar los índices en el array padre original.

- Es de esperar que a estas alturas esté bastante claro que soportar rebanadas en arrays significa que la dimensionalidad, dada por el parámetro `N`, no es necesariamente igual a la dimensionalidad del array padre o la longitud

de la tupla `indexes`. Tampoco los índices proporcionados por el usuario se alinean necesariamente con las entradas en la tupla `indexes` (por ejemplo, el segundo índice proporcionado por el usuario puede corresponder a la tercera dimensión de la matriz padre, y el tercer elemento en la tupla `indexes`).

Lo que podría ser menos obvio es que la dimensionalidad del array padre almacenado sea igual al número de índices efectivos en la tupla `indexes`. Algunos ejemplos:

```
A = reshape(1:35, 5, 7) # A 2d parent Array
S = view(A, 2:7)         # A 1d view created by linear indexing
S = view(A, :, :, 1:1)   # Appending extra indices is supported
```

Ingenuamente, uno pensaría que podría simplemente establecer `S.parent = A` y`S.indexes = (:,:, 1:1)`, pero el hecho de soportar esto complica dramáticamente el proceso de reindexación, especialmente para vistas de vistas. No solo se necesita despachar los tipos de los índices almacenados, sino que se debe examinar si un índice dado es el último y "fusionar" los índices almacenados restantes. Esto no es una tarea fácil, y aún peor: es lenta ya que depende implícitamente de la indexación lineal.

Afortunadamente, este es precisamente el cálculo que 'ReshapedArray' realiza, y lo hace linealmente si es posible. En consecuencia, `view` asegura que el array padre es la dimensionalidad adecuada para los índices dados mediante reformateo (*reshaping*) si es necesario. El constructor interno `SubArray` asegura que este invariante esté satisfecha.

- `CartesianIndex` y sus matrices retuercen de una forma desagradable el esquema `reindex`. Recuerde que `reindex` simplemente despacha sobre el tipo de índices almacenados para determinar cuántos índices pasados deberían usarse y a dónde deberían ir. Pero con `CartesianIndex`, ya no hay una correspondencia uno a uno entre la cantidad de argumentos pasados y la cantidad de dimensiones en las que indexan. Si volvemos al ejemplo anterior de `Base.reindex(S1, S1.indexes, (i, j))`, puede ver que la expansión es incorrecta para `i, j = CartesianIndex (), CartesianIndex (2,1 )`. Él debería *salta* el `CartesianIndex()` por completo y devolver:

  ```
  (CartesianIndex(2,1)[1], S1.indexes[2], S1.indexes[3][CartesianIndex(2,1)[2]])
  ```

  Y si embargo, lo que devuelve es:

  ```
  (CartesianIndex(), S1.indexes[2], S1.indexes[3][CartesianIndex(2,1)])
  ```

  Hacer esto correctamente requeriría el envío *combinado* en los índices almacenados y pasados en todas las combinaciones de dimensionalidades de una manera intratable. Como tal, `reindex` nunca debe invocarse con índices `CartesianIndex`. Afortunadamente, el caso escalar se maneja fácilmente aplanando primero los argumentos `CartesianIndex` a enteros simples. Sin embargo, las matrices de `CartesianIndex` no se pueden dividir en piezas ortogonales tan fácilmente. Antes de intentar usar `reindex`,`view` debe garantizar que no haya matrices de `CartesianIndex` en la lista de argumentos. Si los hay, simplemente puede "puntualizar" evitando por completo el cálculo de 'reindex', construyendo un `SubArray` anidado con dos niveles de indirección en su lugar.

70.12 System Image Building

Building the Julia system image

Julia ships with a preparsed system image containing the contents of the `Base` module, named `sys.ji`. This file is also precompiled into a shared library called `sys.{so,dll,dylib}` on as many platforms as possible, so as to give vastly improved startup times. On systems that do not ship with a precompiled system image file, one can be generated from the source files shipped in Julia's `DATAROOTDIR/julia/base` folder.

This operation is useful for multiple reasons. A user may:

- Build a precompiled shared library system image on a platform that did not ship with one, thereby improving startup times.
- Modify `Base`, rebuild the system image and use the new `Base` next time Julia is started.
- Include a `userimg.jl` file that includes packages into the system image, thereby creating a system image that has packages embedded into the startup environment.

Julia now ships with a script that automates the tasks of building the system image, wittingly named `build_sysimg.jl` that lives in `DATAROOTDIR/julia/`. That is, to include it into a current Julia session, type:

```
include(joinpath(JULIA_HOME, Base.DATAROOTDIR, "julia", "build_sysimg.jl"))
```

This will include a `build_sysimg()` function:

`BuildSysImg.build_sysimg` – Function.

```
build_sysimg(sysimg_path=default_sysimg_path(), cpu_target="native", userimg_path=nothing;
    force=false)
```

Rebuild the system image. Store it in `sysimg_path`, which defaults to a file named `sys.ji` that sits in the same folder as `libjulia.{so,dylib}`, except on Windows where it defaults to `JULIA_HOME/../lib/julia/sys.ji`. Use the cpu instruction set given by `cpu_target`. Valid CPU targets are the same as for the `-C` option to `julia`, or the `-march` option to `gcc`. Defaults to `native`, which means to use all CPU instructions available on the current processor. Include the user image file given by `userimg_path`, which should contain directives such as `using MyPackage` to include that package in the new system image. New system image will not replace an older image unless `force` is set to true.

`source`

Note that this file can also be run as a script itself, with command line arguments taking the place of arguments passed to the `build_sysimg` function. For example, to build a system image in `/tmp/sys.{so,dll,dylib}`, with the `core2` CPU instruction set, a user image of `~/userimg.jl` and `force` set to `true`, one would execute:

```
julia build_sysimg.jl /tmp/sys core2 ~/userimg.jl --force
```

70.13 Working with LLVM

This is not a replacement for the LLVM documentation, but a collection of tips for working on LLVM for Julia.

Overview of Julia to LLVM Interface

Julia statically links in LLVM by default. Build with `USE_LLVM_SHLIB=1` to link dynamically.

The code for lowering Julia AST to LLVM IR or interpreting it directly is in directory `src/`.

Some of the `.cpp` files form a group that compile to a single object.

The difference between an intrinsic and a builtin is that a builtin is a first class function that can be used like any other Julia function. An intrinsic can operate only on unboxed data, and therefore its arguments must be statically typed.

Alias Analysis

Julia currently uses LLVM's Type Based Alias Analysis. To find the comments that document the inclusion relationships, look for `static MDNode*` in `src/codegen.cpp`.

The `-O` option enables LLVM's Basic Alias Analysis.

File	Description
`builtins.c`	Builtin functions
`ccall.cpp`	Lowering `ccall`
`cgutils.cpp`	Lowering utilities, notably for array and tuple accesses
`codegen.cpp`	Top-level of code generation, pass list, lowering builtins
`debuginfo.cpp`	Tracks debug information for JIT code
`disasm.cpp`	Handles native object file and JIT code diassembly
`gf.c`	Generic functions
`intrinsics.cpp`	Lowering intrinsics
`llvm-simdloop.cpp`	Custom LLVM pass for `@simd`
`sys.c`	I/O and operating system utility functions

Building Julia with a different version of LLVM

The default version of LLVM is specified in `deps/Versions.make`. You can override it by creating a file called `Make.user` in the top-level directory and adding a line to it such as:

```
LLVM_VER = 3.5.0
```

Besides the LLVM release numerals, you can also use `LLVM_VER = svn` to bulid against the latest development version of LLVM.

Passing options to LLVM

You can pass options to LLVM using *debug* builds of Julia. To create a debug build, run `make debug`. The resulting executable is `usr/bin/julia-debug`. You can pass LLVM options to this executable via the environment variable `JULIA_LLVM_ARGS`. Here are example settings using `bash` syntax:

- `export JULIA_LLVM_ARGS = -print-after-all` dumps IR after each pass.
- `export JULIA_LLVM_ARGS = -debug-only=loop-vectorize` dumps LLVM `DEBUG(...)` diagnostics for loop vectorizer *if* you built Julia with `LLVM_ASSERTIONS=1`. Otherwise you will get warnings about "Unknown command line argument". Counter-intuitively, building Julia with `LLVM_DEBUG=1` is *not* enough to dump `DEBUG` diagnostics from a pass.

Improving LLVM optimizations for Julia

Improving LLVM code generation usually involves either changing Julia lowering to be more friendly to LLVM's passes, or improving a pass.

If you are planning to improve a pass, be sure to read the LLVM developer policy. The best strategy is to create a code example in a form where you can use LLVM's `opt` tool to study it and the pass of interest in isolation.

1. Create an example Julia code of interest.
2. Use `JULIA_LLVM_ARGS = -print-after-all` to dump the IR.
3. Pick out the IR at the point just before the pass of interest runs.
4. Strip the debug metadata and fix up the TBAA metadata by hand.

The last step is labor intensive. Suggestions on a better way would be appreciated.

70.14 printf() and stdio in the Julia runtime

Libuv wrappers for stdio

`julia.h` defines libuv wrappers for the `stdio.h` streams:

```
uv_stream_t *JL_STDIN;
uv_stream_t *JL_STDOUT;
uv_stream_t *JL_STDERR;
```

... and corresponding output functions:

```
int jl_printf(uv_stream_t *s, const char *format, ...);
int jl_vprintf(uv_stream_t *s, const char *format, va_list args);
```

These `printf` functions are used by the `.c` files in the `src/` and `ui/` directories wherever stdio is needed to ensure that output buffering is handled in a unified way.

In special cases, like signal handlers, where the full libuv infrastructure is too heavy, `jl_safe_printf()` can be used to `write(2)` directly to STDERR_FILENO:

```
void jl_safe_printf(const char *str, ...);
```

Interface between JL_STD* and Julia code

`Base.STDIN`, `Base.STDOUT` and `Base.STDERR` are bound to the `JL_STD*` libuv streams defined in the runtime.

Julia's `__init__()` function (in `base/sysimg.jl`) calls `reinit_stdio()` (in `base/stream.jl`) to create Julia objects for `Base.STDIN`, `Base.STDOUT` and `Base.STDERR`.

`reinit_stdio()` uses `ccall` to retrieve pointers to `JL_STD*` and calls `jl_uv_handle_type()` to inspect the type of each stream. It then creates a Julia `Base.IOStream`, `Base.TTY` or `Base.PipeEndpoint` object to represent each stream, e.g.:

```
$ julia -e 'println(typeof((STDIN, STDOUT, STDERR)))'
Tuple{Base.TTY,Base.TTY,Base.TTY}

$ julia -e 'println(typeof((STDIN, STDOUT, STDERR)))' < /dev/null 2>/dev/null
Tuple{IOStream,Base.TTY,IOStream}

$ echo hello | julia -e 'println(typeof((STDIN, STDOUT, STDERR)))' | cat
Tuple{Base.PipeEndpoint,Base.PipeEndpoint,Base.TTY}
```

The `Base.read()` and `Base.write()` methods for these streams use `ccall` to call libuv wrappers in `src/jl_uv.c`, e.g.:

```
stream.jl: function write(s::IO, p::Ptr, nb::Integer)
               -> ccall(:jl_uv_write, ...)
  jl_uv.c:          -> int jl_uv_write(uv_stream_t *stream, ...)
                        -> uv_write(uvw, stream, buf, ...)
```

printf() during initialization

The libuv streams relied upon by `jl_printf()` etc., are not available until midway through initialization of the runtime (see `init.c`, `init_stdio()`). Error messages or warnings that need to be printed before this are routed to the standard C library `fwrite()` function by the following mechanism:

In `sys.c`, the `JL_STD*` stream pointers are statically initialized to integer constants: `STD*_FILENO (0, 1 and 2)`. In `jl_uv.c` the `jl_uv_puts()` function checks its `uv_stream_t* stream` argument and calls `fwrite()` if stream is set to `STDOUT_FILENO` or `STDERR_FILENO`.

This allows for uniform use of `jl_printf()` throughout the runtime regardless of whether or not any particular piece of code is reachable before initialization is complete.

Legacy `ios.c` library

The `src/support/ios.c` library is inherited from femtolisp. It provides cross-platform buffered file IO and in-memory temporary buffers.

`ios.c` is still used by:

- `src/flisp/*.c`
- `src/dump.c` – for serialization file IO and for memory buffers.
- `base/iostream.jl` – for file IO (see `base/fs.jl` for libuv equivalent).

Use of `ios.c` in these modules is mostly self-contained and separated from the libuv I/O system. However, there is one place where femtolisp calls through to `jl_printf()` with a legacy `ios_t` stream.

There is a hack in `ios.h` that makes the `ios_t.bm` field line up with the `uv_stream_t.type` and ensures that the values used for `ios_t.bm` to not overlap with valid `UV_HANDLE_TYPE` values. This allows `uv_stream_t` pointers to point to `ios_t` streams.

This is needed because `jl_printf()` caller `jl_static_show()` is passed an `ios_t` stream by femtolisp's `fl_print()` function. Julia's `jl_uv_puts()` function has special handling for this:

```
if (stream->type > UV_HANDLE_TYPE_MAX) {
    return ios_write((ios_t*)stream, str, n);
}
```

70.15 Comprobación de Límites

Como muchos lenguajes de programación modernos, Julia usa comprobación de límites para asegurar la seguridad del programa cuando se accede a arrays. En bucles interiores apretados u otras situaciones de rendimiento críticas, uno puede desear saltar estas comprobaciones de límites para mejorar el rendimiento en tiempo de ejecución. Por ejemplo, para emitir instrucciones vectorizadas (SIMD), el cuerpo de nuestro bucle no puede contener ramificaciones y, por tanto no puede contener comprobaciones de límites. En consecuencia, Julia incluye una macro `@inbounds(...)` para decir al compilador que se salte estas comprobaciones de límites dentro del bloque dado. Para el tipo predefinido `Array`, la magia sucede dentro de los intrísecos `arrayref` y `arrayset`. Los tipos array definidos por el usuario usan la macro `@boundscheck(...)` para conseguir una selección de código sensible al contexto.

Omitiendo comprobaciones de límites

La macro `@boundscheck(...)` marca bloques de código que realizan comprobación de límites. Cuando tales bloques aparecen dentro de un bloque `@inbounds(...)`, el compilador elimina esos bloques. Cuando `@boundscheck(...)` es anidado dentro de una función llamadora que contiene un `@inbounds(...)`, el compilador borrará el bloque `@boundscheck` *sólo si este está en línea (`inlined`)* en la función llamadora. Por ejemplo, uno podría escribir el método `sum` como:

```
function sum(A::AbstractArray)
    r = zero(eltype(A))
    for i = 1:length(A)
        @inbounds r += A[i]
    end
    return r
end
```

With a custom array-like type `MyArray` having:

```
@inline getindex(A::MyArray, i::Real) = (@boundscheck checkbounds(A,i); A.data[to_index(i)])
```

Then when `getindex` is inlined into `sum`, the call to `checkbounds(A,i)` will be elided. If your function contains multiple layers of inlining, only `@boundscheck` blocks at most one level of inlining deeper are eliminated. The rule prevents unintended changes in program behavior from code further up the stack.

Propagating inbounds

There may be certain scenarios where for code-organization reasons you want more than one layer between the `@inbounds` and `@boundscheck` declarations. For instance, the default `getindex` methods have the chain `getindex(A::AbstractArray, i::Real)` calls `getindex(IndexStyle(A), A, i)` calls `_getindex(::IndexLinear, A, i)`.

To override the "one layer of inlining" rule, a function may be marked with `@propagate_inbounds` to propagate an inbounds context (or out of bounds context) through one additional layer of inlining.

The bounds checking call hierarchy

The overall hierarchy is:

- `checkbounds(A, I...)` which calls
  - `checkbounds(Bool, A, I...)` which calls
    - `checkbounds_indices(Bool, indices(A), I)` which recursively calls
      - `checkindex` for each dimension

Here `A` is the array, and `I` contains the "requested" indices. `indices(A)` returns a tuple of "permitted" indices of `A`.

`checkbounds(A, I...)` throws an error if the indices are invalid, whereas `checkbounds(Bool, A, I...)` returns `false` in that circumstance. `checkbounds_indices` discards any information about the array other than its `indices` tuple, and performs a pure indices-vs-indices comparison: this allows relatively few compiled methods to serve a huge variety of array types. Indices are specified as tuples, and are usually compared in a 1-1 fashion with individual dimensions handled by calling another important function, `checkindex`: typically,

```
checkbounds_indices(Bool, (IA1, IA...), (I1, I...)) = checkindex(Bool, IA1, I1) &
                                                      checkbounds_indices(Bool, IA, I)
```

so `checkindex` checks a single dimension. All of these functions, including the unexported `checkbounds_indices` have docstrings accessible with `?` .

If you have to customize bounds checking for a specific array type, you should specialize `checkbounds(Bool, A, I...)`. However, in most cases you should be able to rely on `checkbounds_indices` as long as you supply useful `indices` for your array type.

If you have novel index types, first consider specializing `checkindex`, which handles a single index for a particular dimension of an array. If you have a custom multidimensional index type (similar to `CartesianIndex`), then you may have to consider specializing `checkbounds_indices`.

Note this hierarchy has been designed to reduce the likelihood of method ambiguities. We try to make `checkbounds` the place to specialize on array type, and try to avoid specializations on index types; conversely, `checkindex` is intended to be specialized only on index type (especially, the last argument).

70.16 Proper maintenance and care of multi-threading locks

The following strategies are used to ensure that the code is dead-lock free (generally by addressing the 4th Coffman condition: circular wait).

1. structure code such that only one lock will need to be acquired at a time
2. always acquire shared locks in the same order, as given by the table below
3. avoid constructs that expect to need unrestricted recursion

Locks

Below are all of the locks that exist in the system and the mechanisms for using them that avoid the potential for deadlocks (no Ostrich algorithm allowed here):

The following are definitely leaf locks (level 1), and must not try to acquire any other lock:

- safepoint

  Note that this lock is acquired implicitly by `JL_LOCK` and `JL_UNLOCK`. use the `_NOGC` variants to avoid that for level 1 locks.
  While holding this lock, the code must not do any allocation or hit any safepoints. Note that there are safepoints when doing allocation, enabling / disabling GC, entering / restoring exception frames, and taking / releasing locks.
- shared_map
- finalizers
- pagealloc
- gc_perm_lock
- flisp

  flisp itself is already threadsafe, this lock only protects the `jl_ast_context_list_t` pool

The following is a leaf lock (level 2), and only acquires level 1 locks (safepoint) internally:

- typecache

The following is a level 3 lock, which can only acquire level 1 or level 2 locks internally:

- Method->writelock

The following is a level 4 lock, which can only recurse to acquire level 1, 2, or 3 locks:

- MethodTable->writelock

No Julia code may be called while holding a lock above this point.

The following is a level 6 lock, which can only recurse to acquire locks at lower levels:

- codegen

The following is an almost root lock (level end-1), meaning only the root look may be held when trying to acquire it:

- typeinf

  this one is perhaps one of the most tricky ones, since type-inference can be invoked from many points
  currently the lock is merged with the codegen lock, since they call each other recursively

The following is the root lock, meaning no other lock shall be held when trying to acquire it:

- toplevel

  this should be held while attempting a top-level action (such as making a new type or defining a new method): trying to obtain this lock inside a staged function will cause a deadlock condition!
  additionally, it's unclear if *any* code can safely run in parallel with an arbitrary toplevel expression, so it may require all threads to get to a safepoint first

Broken Locks

The following locks are broken:

- toplevel

  doesn't exist right now
  fix: create it

Shared Global Data Structures

These data structures each need locks due to being shared mutable global state. It is the inverse list for the above lock priority list. This list does not include level 1 leaf resources due to their simplicity.

MethodTable modifications (def, cache, kwsorter type) : MethodTable->writelock

Type declarations : toplevel lock

Type application : typecache lock

Module serializer : toplevel lock

JIT & type-inference : codegen lock

MethodInstance updates : codegen lock

- These fields are generally lazy initialized, using the test-and-test-and-set pattern.
- These are set at construction and immutable:

  - specTypes
  - sparam_vals
  - def
- These are set by `jl_type_infer` (while holding codegen lock):
  - rettype
  - inferred
  - these can also be reset, see `jl_set_lambda_rettype` for that logic as it needs to keep `functionObjectsDecls` in sync
- `inInference` flag:
  - optimization to quickly avoid recurring into `jl_type_infer` while it is already running
  - actual state (of setting `inferred`, then `fptr`) is protected by codegen lock
- Function pointers (`jlcall_api` and `fptr`, `unspecialized_ducttape`):
  - these transition once, from `NULL` to a value, while the codegen lock is held
- Code-generator cache (the contents of `functionObjectsDecls`):
  - these can transition multiple times, but only while the codegen lock is held
  - it is valid to use old version of this, or block for new versions of this, so races are benign, as long as the code is careful not to reference other data in the method instance (such as `rettype`) and assume it is coordinated, unless also holding the codegen lock
- `compile_traced` flag:
  - unknown

LLVMContext : codegen lock

Method : Method->writelock

- roots array (serializer and codegen)
- invoke / specializations / tfunc modifications

70.17 Arrays with custom indices

Julia 0.5 adds experimental support for arrays with arbitrary indices. Conventionally, Julia's arrays are indexed starting at 1, whereas some other languages start numbering at 0, and yet others (e.g., Fortran) allow you to specify arbitrary starting indices. While there is much merit in picking a standard (i.e., 1 for Julia), there are some algorithms which simplify considerably if you can index outside the range `1:size(A,d)` (and not just `0:size(A,d)-1`, either). Such array types are expected to be supplied through packages.

The purpose of this page is to address the question, "what do I have to do to support such arrays in my own code?" First, let's address the simplest case: if you know that your code will never need to handle arrays with unconventional indexing, hopefully the answer is "nothing." Old code, on conventional arrays, should function essentially without alteration as long as it was using the exported interfaces of Julia.

Generalizing existing code

As an overview, the steps are:

- replace many uses of `size` with `indices`

- replace `1:length(A)` with `linearindices(A)`, and `length(A)` with `length(linearindices(A))`
- replace explicit allocations like `Array{Int}(size(B))` with `similar(Array{Int}, indices(B))`

These are described in more detail below.

Background

Because unconventional indexing breaks deeply-held assumptions throughout the Julia ecosystem, early adopters running code that has not been updated are likely to experience errors. The most frustrating bugs would be incorrect results or segfaults (total crashes of Julia). For example, consider the following function:

```
function mycopy!(dest::AbstractVector, src::AbstractVector)
    length(dest) == length(src) || throw(DimensionMismatch("vectors must match"))
    # OK, now we're safe to use @inbounds, right? (not anymore!)
    for i = 1:length(src)
        @inbounds dest[i] = src[i]
    end
    dest
end
```

This code implicitly assumes that vectors are indexed from 1. Previously that was a safe assumption, so this code was fine, but (depending on what types the user passes to this function) it may no longer be safe. If this code continued to work when passed a vector with non-1 indices, it would either produce an incorrect answer or it would segfault. (If you do get segfaults, to help locate the cause try running julia with the option `--check-bounds=yes`.)

To ensure that such errors are caught, in Julia 0.5 both `length` and `size`**should** throw an error when passed an array with non-1 indexing. This is designed to force users of such arrays to check the code, and inspect it for whether it needs to be generalized.

Using `indices` for bounds checks and loop iteration

`indices(A)` (reminiscent of `size(A)`) returns a tuple of `AbstractUnitRange` objects, specifying the range of valid indices along each dimension of `A`. When `A` has unconventional indexing, the ranges may not start at 1. If you just want the range for a particular dimension `d`, there is `indices(A, d)`.

Base implements a custom range type, `OneTo`, where `OneTo(n)` means the same thing as `1:n` but in a form that guarantees (via the type system) that the lower index is 1. For any new `AbstractArray` type, this is the default returned by `indices`, and it indicates that this array type uses "conventional" 1-based indexing. Note that if you don't want to be bothered supporting arrays with non-1 indexing, you can add the following line:

```
@assert all(x->isa(x, Base.OneTo), indices(A))
```

at the top of any function.

For bounds checking, note that there are dedicated functions `checkbounds` and `checkindex` which can sometimes simplify such tests.

Linear indexing (`linearindices`)

Some algorithms are most conveniently (or efficiently) written in terms of a single linear index, `A[i]` even if `A` is multi-dimensional. In "true" linear indexing, the indices always range from `1:length(A)`. However, this raises an ambiguity for one-dimensional arrays (a.k.a., `AbstractVector`): does `v[i]` mean linear indexing, or Cartesian indexing with the array's native indices?

For this reason, if you want to use linear indexing in an algorithm, your best option is to get the index range by calling `linearindices(A)`. This will return `indices(A, 1)` if `A` is an `AbstractVector`, and the equivalent of `1:length(A)` otherwise.

In a sense, one can say that 1-dimensional arrays always use Cartesian indexing. To help enforce this, it's worth noting that `sub2ind(shape, i...)` and `ind2sub(shape, ind)` will throw an error if `shape` indicates a 1-dimensional array with unconventional indexing (i.e., is a `Tuple{UnitRange}` rather than a tuple of `OneTo`). For arrays with conventional indexing, these functions continue to work the same as always.

Using `indices` and `linearindices`, here is one way you could rewrite `mycopy!`:

```
function mycopy!(dest::AbstractVector, src::AbstractVector)
    indices(dest) == indices(src) || throw(DimensionMismatch("vectors must match"))
    for i in linearindices(src)
        @inbounds dest[i] = src[i]
    end
    dest
end
```

Allocating storage using generalizations of `similar`

Storage is often allocated with `Array{Int}(dims)` or `similar(A, args...)`. When the result needs to match the indices of some other array, this may not always suffice. The generic replacement for such patterns is to use `similar(storagetype, shape)`. `storagetype` indicates the kind of underlying "conventional" behavior you'd like, e.g., `Array{Int}` or `BitArray` or even `dims->zeros(Float32, dims)` (which would allocate an all-zeros array). `shape` is a tuple of `Integer` or `AbstractUnitRange` values, specifying the indices that you want the result to use.

Let's walk through a couple of explicit examples. First, if `A` has conventional indices, then `similar(Array{Int}, indices(A))` would end up calling `Array{Int}(size(A))`, and thus return an array. If `A` is an `AbstractArray` type with unconventional indexing, then `similar(Array{Int}, indices(A))` should return something that "behaves like" an `Array{Int}` but with a shape (including indices) that matches `A`. (The most obvious implementation is to allocate an `Array{Int}(size(A))` and then "wrap" it in a type that shifts the indices.)

Note also that `similar(Array{Int}, (indices(A, 2),))` would allocate an `AbstractVector{Int}` (i.e., 1-dimensional array) that matches the indices of the columns of `A`.

Deprecations

In generalizing Julia's code base, at least one deprecation was unavoidable: earlier versions of Julia defined `first(::Colon) = 1`, meaning that the first index along a dimension indexed by `:` is 1. This definition can no longer be justified, so it was deprecated. There is no provided replacement, because the proper replacement depends on what you are doing and might need to know more about the array. However, it appears that many uses of `first(::Colon)` are really about computing an index offset; when that is the case, a candidate replacement is:

```
indexoffset(r::AbstractVector) = first(r) - 1
indexoffset(::Colon) = 0
```

In other words, while `first(:)` does not itself make sense, in general you can say that the offset associated with a colon-index is zero.

Writing custom array types with non-1 indexing

Most of the methods you'll need to define are standard for any `AbstractArray` type, see Abstract Arrays. This page focuses on the steps needed to define unconventional indexing.

Do not implement `size` or `length`

Perhaps the majority of pre-existing code that uses `size` will not work properly for arrays with non-1 indices. For that reason, it is much better to avoid implementing these methods, and use the resulting `MethodError` to identify code that needs to be audited and perhaps generalized.

Do not annotate bounds checks

Julia 0.5 includes `@boundscheck` to annotate code that can be removed for callers that exploit `@inbounds`. Initially, it seems far preferable to run with bounds checking always enabled (i.e., omit the `@boundscheck` annotation so the check always runs).

Custom `AbstractUnitRange` types

If you're writing a non-1 indexed array type, you will want to specialize `indices` so it returns a `UnitRange`, or (perhaps better) a custom `AbstractUnitRange`. The advantage of a custom type is that it "signals" the allocation type for functions like `similar`. If we're writing an array type for which indexing will start at 0, we likely want to begin by creating a new `AbstractUnitRange`, `ZeroRange`, where `ZeroRange(n)` is equivalent to `0:n-1`.

In general, you should probably *not* export `ZeroRange` from your package: there may be other packages that implement their own `ZeroRange`, and having multiple distinct `ZeroRange` types is (perhaps counterintuitively) an advantage: `ModuleA.ZeroRange` indicates that `similar` should create a `ModuleA.ZeroArray`, whereas `ModuleB.ZeroRange` indicates a `ModuleB.ZeroArray` type. This design allows peaceful coexistence among many different custom array types.

Note that the Julia package CustomUnitRanges.jl can sometimes be used to avoid the need to write your own `ZeroRange` type.

Specializing `indices`

Once you have your `AbstractUnitRange` type, then specialize `indices`:

```
Base.indices(A::ZeroArray) = map(n->ZeroRange(n), A.size)
```

where here we imagine that `ZeroArray` has a field called `size` (there would be other ways to implement this).

In some cases, the fallback definition for `indices(A, d)`:

```
indices(A::AbstractArray{T,N}, d) where {T,N} = d <= N ? indices(A)[d] : OneTo(1)
```

may not be what you want: you may need to specialize it to return something other than `OneTo(1)` when `d > ndims(A)`. Likewise, in `Base` there is a dedicated function `indices1` which is equivalent to `indices(A, 1)` but which avoids checking (at runtime) whether `ndims(A) > 0`. (This is purely a performance optimization.) It is defined as:

```
indices1(A::AbstractArray{T,0}) where {T} = OneTo(1)
indices1(A::AbstractArray) = indices(A)[1]
```

If the first of these (the zero-dimensional case) is problematic for your custom array type, be sure to specialize it appropriately.

Specializing `similar`

Given your custom `ZeroRange` type, then you should also add the following two specializations for `similar`:

```
function Base.similar(A::AbstractArray, T::Type, shape::Tuple{ZeroRange,Vararg{ZeroRange}})
    # body
end

function Base.similar(f::Union{Function,DataType}, shape::Tuple{ZeroRange,Vararg{ZeroRange}})
    # body
end
```

Both of these should allocate your custom array type.

Specializing `reshape`

Optionally, define a method

```
Base.reshape(A::AbstractArray, shape::Tuple{ZeroRange,Vararg{ZeroRange}}) = ...
```

and you can `reshape` an array so that the result has custom indices.

Summary

Writing code that doesn't make assumptions about indexing requires a few extra abstractions, but hopefully the necessary changes are relatively straightforward.

As a reminder, this support is still experimental. While much of Julia's base code has been updated to support unconventional indexing, without a doubt there are many omissions that will be discovered only through usage. Moreover, at the time of this writing, most packages do not support unconventional indexing. As a consequence, early adopters should be prepared to identify and/or fix bugs. On the other hand, only through practical usage will it become clear whether this experimental feature should be retained in future versions of Julia; consequently, interested parties are encouraged to accept some ownership for putting it through its paces.

70.18 Base.LibGit2

The LibGit2 module provides bindings to libgit2, a portable C library that implements core functionality for the Git version control system. These bindings are currently used to power Julia's package manager. It is expected that this module will eventually be moved into a separate package.

Functionality

Some of this documentation assumes some prior knowledge of the libgit2 API. For more information on some of the objects and methods referenced here, consult the upstream libgit2 API reference.

`Base.LibGit2.AbstractCredentials` – Type.

> Abstract credentials payload
>
> source

`Base.LibGit2.Buffer` – Type.

```
LibGit2.Buffer
```

A data buffer for exporting data from libgit2. Matches the `git_buf` struct.

When fetching data from LibGit2, a typical usage would look like:

```
buf_ref = Ref(Buffer())
@check ccall(..., (Ptr{Buffer},), buf_ref)
# operation on buf_ref
free(buf_ref)
```

In particular, note that `LibGit2.free` should be called afterward on the `Ref` object.

source

`Base.LibGit2.CachedCredentials` – Type.

Credentials that support caching

source

`Base.LibGit2.CheckoutOptions` – Type.

```
LibGit2.CheckoutOptions
```

Matches the `git_checkout_options` struct.

source

`Base.LibGit2.CloneOptions` – Type.

```
LibGit2.CloneOptions
```

Matches the `git_clone_options` struct.

source

`Base.LibGit2.DiffDelta` – Type.

```
LibGit2.DiffDelta
```

Description of changes to one entry. Matches the `git_diff_delta` struct.

The fields represent:

- `status`: One of `Consts.DELTA_STATUS`, indicating whether the file has been added/modified/deleted.
- `flags`: Flags for the delta and the objects on each side. Determines whether to treat the file(s) as binary/-text, whether they exist on each side of the diff, and whether the object ids are known to be correct.
- `similarity`: Used to indicate if a file has been renamed or copied.
- `nfiles`: The number of files in the delta (for instance, if the delta was run on a submodule commit id, it may contain more than one file).
- `old_file`: A `DiffFile` containing information about the file(s) before the changes.
- `new_file`: A `DiffFile` containing information about the file(s) after the changes.

source

`Base.LibGit2.DiffFile` – Type.

```
LibGit2.DiffFile
```

Description of one side of a delta. Matches the `git_diff_file` struct.

source

`Base.LibGit2.DiffOptionsStruct` – Type.

```
LibGit2.DiffOptionsStruct
```

Matches the `git_diff_options` struct.

source

`Base.LibGit2.FetchHead` – Type.

```
LibGit2.FetchHead
```

Contains the information about HEAD during a fetch, including the name and URL of the branch fetched from, the oid of the HEAD, and whether the fetched HEAD has been merged locally.

source

`Base.LibGit2.FetchOptions` – Type.

```
LibGit2.FetchOptions
```

Matches the `git_fetch_options` struct.

source

`Base.LibGit2.GitBlob` – Type.

```
GitBlob(repo::GitRepo, hash::AbstractGitHash)
GitBlob(repo::GitRepo, spec::AbstractString)
```

Return a `GitBlob` object from `repo` specified by `hash/spec`.

- `hash` is a full (`GitHash`) or partial (`GitShortHash`) hash.
- `spec` is a textual specification: see the git docs for a full list.

source

`Base.LibGit2.GitCommit` – Type.

```
GitCommit(repo::GitRepo, hash::AbstractGitHash)
GitCommit(repo::GitRepo, spec::AbstractString)
```

Return a `GitCommit` object from `repo` specified by `hash/spec`.

- `hash` is a full (`GitHash`) or partial (`GitShortHash`) hash.
- `spec` is a textual specification: see the git docs for a full list.

source

`Base.LibGit2.GitHash` – Type.

```
GitHash
```

A git object identifier, based on the sha-1 hash. It is a 20 byte string (40 hex digits) used to identify a `GitObject` in a repository.

source

`Base.LibGit2.GitObject` – Type.

```
GitObject(repo::GitRepo, hash::AbstractGitHash)
GitObject(repo::GitRepo, spec::AbstractString)
```

Return the specified object (`GitCommit`, `GitBlob`, `GitTree` or `GitTag`) from `repo` specified by `hash`/`spec`.

- `hash` is a full (`GitHash`) or partial (`GitShortHash`) hash.
- `spec` is a textual specification: see the git docs for a full list.

source

`Base.LibGit2.GitRemote` – Type.

```
GitRemote(repo::GitRepo, rmt_name::AbstractString, rmt_url::AbstractString) -> GitRemote
```

Look up a remote git repository using its name and URL. Uses the default fetch refspec.

Example

```
repo = LibGit2.init(repo_path)
remote = LibGit2.GitRemote(repo, "upstream", repo_url)
```

source

```
GitRemote(repo::GitRepo, rmt_name::AbstractString, rmt_url::AbstractString, fetch_spec::
    AbstractString) -> GitRemote
```

Look up a remote git repository using the repository's name and URL, as well as specifications for how to fetch from the remote (e.g. which remote branch to fetch from).

Example

```
repo = LibGit2.init(repo_path)
refspec = "+refs/heads/mybranch:refs/remotes/origin/mybranch"
remote = LibGit2.GitRemote(repo, "upstream", repo_url, refspec)
```

source

`Base.LibGit2.GitRemoteAnon` – Function.

```
GitRemoteAnon(repo::GitRepo, url::AbstractString) -> GitRemote
```

Look up a remote git repository using only its URL, not its name.

Example

```
repo = LibGit2.init(repo_path)
remote = LibGit2.GitRemoteAnon(repo, repo_url)
```

source

`Base.LibGit2.GitRepo` – Type.

```
LibGit2.GitRepo(path::AbstractString)
```

Opens a git repository at `path`.

source

`Base.LibGit2.GitRepoExt` – Function.

```
LibGit2.GitRepoExt(path::AbstractString, flags::Cuint = Cuint(Consts.REPOSITORY_OPEN_DEFAULT)
    )
```

Opens a git repository at `path` with extended controls (for instance, if the current user must be a member of a special access group to read `path`).

source

`Base.LibGit2.GitShortHash` – Type.

```
GitShortHash
```

This is a shortened form of `GitHash`, which can be used to identify a git object when it is unique.

Internally it is stored as two fields: a full-size `GitHash` (`hash`) and a length (`len`). Only the initial `len` hex digits of `hash` are used.

source

`Base.LibGit2.GitSignature` – Type.

```
LibGit2.GitSignature
```

This is a Julia wrapper around a pointer to a `git_signature` object.

source

`Base.LibGit2.GitStatus` – Type.

```
LibGit2.GitStatus(repo::GitRepo; status_opts=StatusOptions())
```

Collect information about the status of each file in the git repository `repo` (e.g. is the file modified, staged, etc.). `status_opts` can be used to set various options, for instance whether or not to look at untracked files or whether to include submodules or not.

source

`Base.LibGit2.GitTag` – Type.

```
GitTag(repo::GitRepo, hash::AbstractGitHash)
GitTag(repo::GitRepo, spec::AbstractString)
```

Return a `GitTag` object from `repo` specified by `hash`/`spec`.

- `hash` is a full (`GitHash`) or partial (`GitShortHash`) hash.
- `spec` is a textual specification: see the git docs for a full list.

source

`Base.LibGit2.GitTree` – Type.

```
GitTree(repo::GitRepo, hash::AbstractGitHash)
GitTree(repo::GitRepo, spec::AbstractString)
```

Return a `GitTree` object from `repo` specified by `hash`/`spec`.

- `hash` is a full (`GitHash`) or partial (`GitShortHash`) hash.
- `spec` is a textual specification: see the git docs for a full list.

source

`Base.LibGit2.IndexEntry` – Type.

```
LibGit2.IndexEntry
```

In-memory representation of a file entry in the index. Matches the `git_index_entry` struct.

source

`Base.LibGit2.IndexTime` – Type.

```
LibGit2.IndexTime
```

Matches the `git_index_time` struct.

source

`Base.LibGit2.MergeOptions` – Type.

```
LibGit2.MergeOptions
```

Matches the `git_merge_options` struct.

source

`Base.LibGit2.ProxyOptions` – Type.

```
LibGit2.ProxyOptions
```

Options for connecting through a proxy.

Matches the `git_proxy_options` struct.

source

`Base.LibGit2.PushOptions` – Type.

```
LibGit2.PushOptions
```

Matches the `git_push_options` struct.

source

`Base.LibGit2.RebaseOperation` – Type.

```
LibGit2.RebaseOperation
```

Describes a single instruction/operation to be performed during the rebase. Matches the `git_rebase_operation` struct.

source

Base.LibGit2.RebaseOptions – Type.

```
LibGit2.RebaseOptions
```

Matches the `git_rebase_options` struct.

source

Base.LibGit2.RemoteCallbacks – Type.

```
LibGit2.RemoteCallbacks
```

Callback settings. Matches the `git_remote_callbacks` struct.

source

Base.LibGit2.SSHCredentials – Type.

SSH credentials type

source

Base.LibGit2.SignatureStruct – Type.

```
LibGit2.SignatureStruct
```

An action signature (e.g. for committers, taggers, etc). Matches the `git_signature` struct.

source

Base.LibGit2.StatusEntry – Type.

```
LibGit2.StatusEntry
```

Providing the differences between the file as it exists in HEAD and the index, and providing the differences between the index and the working directory. Matches the `git_status_entry` struct.

source

Base.LibGit2.StatusOptions – Type.

```
LibGit2.StatusOptions
```

Options to control how `git_status_foreach_ext()` will issue callbacks. Matches the `git_status_opt_t` struct.

source

Base.LibGit2.StrArrayStruct – Type.

```
LibGit2.StrArrayStruct
```

A LibGit2 representation of an array of strings. Matches the `git_strarray` struct.

When fetching data from LibGit2, a typical usage would look like:

```
sa_ref = Ref(StrArrayStruct())
@check ccall(..., (Ptr{StrArrayStruct},), sa_ref)
res = convert(Vector{String}, sa_ref[])
free(sa_ref)
```

In particular, note that `LibGit2.free` should be called afterward on the `Ref` object.

Conversely, when passing a vector of strings to LibGit2, it is generally simplest to rely on implicit conversion:

```
strs = String[...]
@check ccall(..., (Ptr{StrArrayStruct},), strs)
```

Note that no call to `free` is required as the data is allocated by Julia.

source

`Base.LibGit2.TimeStruct` – Type.

```
LibGit2.TimeStruct
```

Time in a signature. Matches the `git_time` struct.

source

`Base.LibGit2.UserPasswordCredentials` – Type.

Credentials that support only `user` and `password` parameters

source

`Base.LibGit2.add_fetch!` – Function.

```
add_fetch!(repo::GitRepo, rmt::GitRemote, fetch_spec::String)
```

Add a *fetch* refspec for the specified `rmt`. This refspec will contain information about which branch(es) to fetch from.

Example

```
julia> LibGit2.add_fetch!(repo, remote, "upstream");

julia> LibGit2.fetch_refspecs(remote)
String["+refs/heads/*:refs/remotes/upstream/*"]
```

source

`Base.LibGit2.add_push!` – Function.

```
add_push!(repo::GitRepo, rmt::GitRemote, push_spec::String)
```

Add a *push* refspec for the specified `rmt`. This refspec will contain information about which branch(es) to push to.

Example

```
julia> LibGit2.add_push!(repo, remote, "refs/heads/master");

julia> remote = LibGit2.get(LibGit2.GitRemote, repo, branch);

julia> LibGit2.push_refspecs(remote)
String["refs/heads/master"]
```

Note

You may need to `close` and reopen the `GitRemote` in question after updating its push refspecs in order for the change to take effect and for calls to `push` to work.

source

Base.LibGit2.addblob! – Function.

```
LibGit2.addblob!(repo::GitRepo, path::AbstractString)
```

Reads the file at path and adds it to the object database of repo as a loose blob. Returns the GitHash of the resulting blob.

Example

```
hash_str = hex(commit_oid)
blob_file = joinpath(repo_path, ".git", "objects", hash_str[1:2], hash_str[3:end])
id = LibGit2.addblob!(repo, blob_file)
```

source

Base.LibGit2.authors – Function.

```
authors(repo::GitRepo) -> Vector{Signature}
```

Returns all authors of commits to the repo repository.

Example

```
repo = LibGit2.GitRepo(repo_path)
repo_file = open(joinpath(repo_path, test_file), "a")

println(repo_file, commit_msg)
flush(repo_file)
LibGit2.add!(repo, test_file)
sig = LibGit2.Signature("TEST", "TEST@TEST.COM", round(time(), 0), 0)
commit_oid1 = LibGit2.commit(repo, "commit1"; author=sig, committer=sig)
println(repo_file, randstring(10))
flush(repo_file)
LibGit2.add!(repo, test_file)
commit_oid2 = LibGit2.commit(repo, "commit2"; author=sig, committer=sig)

# will be a Vector of [sig, sig]
auths = LibGit2.authors(repo)
```

source

Base.LibGit2.branch – Function.

```
branch(repo::GitRepo)
```

Equivalent to git branch. Create a new branch from the current HEAD.

source

Base.LibGit2.branch! – Function.

```
branch!(repo::GitRepo, branch_name::AbstractString, commit::AbstractString=""; kwargs...)
```

Checkout a new git branch in the repo repository. commit is the GitHash, in string form, which will be the start of the new branch. If commit is an empty string, the current HEAD will be used.

The keyword arguments are:

- `track::AbstractString=""`: the name of the remote branch this new branch should track, if any. If empty (the default), no remote branch will be tracked.
- `force::Bool=false`: if `true`, branch creation will be forced.
- `set_head::Bool=true`: if `true`, after the branch creation finishes the branch head will be set as the HEAD of `repo`.

Equivalent to `git checkout [-b|-B] <branch_name> [<commit>] [--track <track>]`.

Example

```
repo = LibGit2.GitRepo(repo_path)
LibGit2.branch!(repo, "new_branch", set_head=false)
```

source

`Base.LibGit2.checkout!` – Function.

```
checkout!(repo::GitRepo, commit::AbstractString=""; force::Bool=true)
```

Equivalent to `git checkout [-f] --detach <commit>`. Checkout the git commit `commit` (a `GitHash` in string form) in `repo`. If `force` is `true`, force the checkout and discard any current changes. Note that this detaches the current HEAD.

Example

```
repo = LibGit2.init(repo_path)
open(joinpath(LibGit2.path(repo), "file1"), "w") do f
    write(f, "111
")
end
LibGit2.add!(repo, "file1")
commit_oid = LibGit2.commit(repo, "add file1")
open(joinpath(LibGit2.path(repo), "file1"), "w") do f
    write(f, "112
")
end
# would fail without the force=true
# since there are modifications to the file
LibGit2.checkout!(repo, string(commit_oid), force=true)
```

source

`Base.LibGit2.checkused!` – Function.

Checks if credentials were used

source

Checks if credentials were used or failed authentication, see `LibGit2.credentials_callback`

source

`Base.LibGit2.clone` – Function.

```
clone(repo_url::AbstractString, repo_path::AbstractString; kwargs...)
```

Clone a remote repository located at `repo_url` to the local filesystem location `repo_path`.

The keyword arguments are:

- `branch::AbstractString=""`: which branch of the remote to clone, if not the default repository branch (usually `master`).
- `isbare::Bool=false`: if `true`, clone the remote as a bare repository, which will make `repo_path` itself the git directory instead of `repo_path/.git`. This means that a working tree cannot be checked out. Plays the role of the git CLI argument `--bare`.
- `remote_cb::Ptr{Void}=C_NULL`: a callback which will be used to create the remote before it is cloned. If `C_NULL` (the default), no attempt will be made to create the remote - it will be assumed to already exist.
- `payload::Nullable{P<:AbstractCredentials}=Nullable{AbstractCredentials}()`: provides credentials if necessary, for instance if the remote is a private repository.

Equivalent to `git clone [-b <branch>] [--bare] <repo_url> <repo_path>`.

Examples

```
repo_url = "https://github.com/JuliaLang/Example.jl"
repo1 = LibGit2.clone(repo_url, "test_path")
repo2 = LibGit2.clone(repo_url, "test_path", isbare=true)
julia_url = "https://github.com/JuliaLang/julia"
julia_repo = LibGit2.clone(julia_url, "julia_path", branch="release-0.6")
```

source

`Base.LibGit2.commit` – Function.

Wrapper around `git_commit_create`

source

Commit changes to repository

source

```
LibGit2.commit(rb::GitRebase, sig::GitSignature)
```

Commits the current patch to the rebase `rb`, using `sig` as the committer. Is silent if the commit has already been applied.

source

`Base.LibGit2.create_branch` – Function.

```
LibGit2.create_branch(repo::GitRepo, bname::AbstractString, commit_obj::GitCommit; force::
    Bool=false)
```

Create a new branch in the repository `repo` with name `bname`, which points to commit `commit_obj` (which has to be part of `repo`). If `force` is `true`, overwrite an existing branch named `bname` if it exists. If `force` is `false` and a branch already exists named `bname`, this function will throw an error.

source

`Base.LibGit2.credentials_callback` – Function.

Credentials callback function

Function provides different credential acquisition functionality w.r.t. a connection protocol. If a payload is provided then `payload_ptr` should contain a `LibGit2.AbstractCredentials` object.

For `LibGit2.Consts.CREDTYPE_USERPASS_PLAINTEXT` type, if the payload contains fields: `user` & `pass`, they are used to create authentication credentials. Empty `user` name and `password` trigger an authentication error.

For `LibGit2.Consts.CREDTYPE_SSH_KEY` type, if the payload contains fields: `user`, `prvkey`, `pubkey` & `pass`, they are used to create authentication credentials. Empty `user` name triggers an authentication error.

Credentials are checked in the following order (if supported):

- ssh key pair (`ssh-agent` if specified in payload's `usesshagent` field)
- plain text

Note: Due to the specifics of the `libgit2` authentication procedure, when authentication fails, this function is called again without any indication whether authentication was successful or not. To avoid an infinite loop from repeatedly using the same faulty credentials, the `checkused!` function can be called. This function returns `true` if the credentials were used. Using credentials triggers a user prompt for (re)entering required information. `UserPasswordCredentials` and `CachedCredentials` are implemented using a call counting strategy that prevents repeated usage of faulty credentials.

source

`Base.LibGit2.credentials_cb` – Function.

C function pointer for `credentials_callback`

source

`Base.LibGit2.default_signature` – Function.

Return signature object. Free it after use.

source

`Base.LibGit2.delete_branch` – Function.

```
LibGit2.delete_branch(branch::GitReference)
```

Delete the branch pointed to by `branch`.

source

`Base.LibGit2.diff_files` – Function.

```
diff_files(repo::GitRepo, branch1::AbstractString, branch2::AbstractString; kwarg...) ->
    Vector{AbstractString}
```

Show which files have changed in the git repository `repo` between branches `branch1` and `branch2`.

The keyword argument is:

- `filter::Set{Consts.DELTA_STATUS}=Set([Consts.DELTA_ADDED, Consts.DELTA_MODIFIED, Consts.DELTA_DELETED]))`, and it sets options for the diff. The default is to show files added, modified, or deleted.

Returns only the *names* of the files which have changed, *not* their contents.

Example

```
LibGit2.branch!(repo, "branch/a")
LibGit2.branch!(repo, "branch/b")
# add a file to repo
open(joinpath(LibGit2.path(repo),"file"),"w") do f
    write(f, "hello repo
")
end
LibGit2.add!(repo, "file")
LibGit2.commit(repo, "add file")
# returns ["file"]
filt = Set([LibGit2.Consts.DELTA_ADDED])
files = LibGit2.diff_files(repo, "branch/a", "branch/b", filter=filt)
# returns [] because existing files weren't modified
filt = Set([LibGit2.Consts.DELTA_MODIFIED])
files = LibGit2.diff_files(repo, "branch/a", "branch/b", filter=filt)
```

Equivalent to `git diff --name-only --diff-filter=<filter> <branch1> <branch2>`.

source

`Base.LibGit2.fetch` – Function.

```
fetch(rmt::GitRemote, refspecs; options::FetchOptions=FetchOptions(), msg="")
```

Fetch from the specified `rmt` remote git repository, using `refspecs` to determine which remote branch(es) to fetch. The keyword arguments are:

- `options`: determines the options for the fetch, e.g. whether to prune afterwards.
- `msg`: a message to insert into the reflogs.

source

```
fetch(repo::GitRepo; kwargs...)
```

Fetches updates from an upstream of the repository `repo`.

The keyword arguments are:

- `remote::AbstractString="origin"`: which remote, specified by name, of `repo` to fetch from. If this is empty, the URL will be used to construct an anonymous remote.
- `remoteurl::AbstractString=""`: the URL of `remote`. If not specified, will be assumed based on the given name of `remote`.
- `refspecs=AbstractString[]`: determines properties of the fetch.
- `payload=Nullable{AbstractCredentials}()`: provides credentials, if necessary, for instance if `remote` is a private repository.

Equivalent to `git fetch [<remoteurl>|<repo>] [<refspecs>]`.

source

`Base.LibGit2.fetch_refspecs` – Function.

```
fetch_refspecs(rmt::GitRemote) -> Vector{String}
```

Get the *fetch* refspecs for the specified `rmt`. These refspecs contain information about which branch(es) to fetch from.

source

`Base.LibGit2.fetchhead_foreach_cb` – Function.

C function pointer for `fetchhead_foreach_callback`

source

`Base.LibGit2.ffmerge!` – Function.

Fastforward merge changes into current head

source

`Base.LibGit2.fullname` – Function.

```
LibGit2.fullname(ref::GitReference)
```

Return the name of the reference pointed to by the symbolic reference `ref`. If `ref` is not a symbolic reference, returns an empty string.

source

`Base.LibGit2.get_creds!` – Function.

Obtain the cached credentials for the given host+protocol (credid), or return and store the default if not found

source

`Base.LibGit2.gitdir` – Function.

```
LibGit2.gitdir(repo::GitRepo)
```

Returns the location of the "git" files of `repo`:

- for normal repositories, this is the location of the `.git` folder.
- for bare repositories, this is the location of the repository itself.

See also `workdir`, `path`.

source

`Base.LibGit2.head` – Function.

```
LibGit2.head(repo::GitRepo) -> GitReference
```

Returns a `GitReference` to the current HEAD of `repo`.

source

```
head(pkg::AbstractString) -> String
```

Return current HEAD `GitHash` of the `pkg` repo as a string.

source

`Base.LibGit2.head!` – Function.

```
LibGit2.head!(repo::GitRepo, ref::GitReference) -> GitReference
```

Set the HEAD of `repo` to the object pointed to by `ref`.

source

`Base.LibGit2.head_oid` – Function.

```
LibGit2.head_oid(repo::GitRepo) -> GitHash
```

Lookup the object id of the current HEAD of git repository `repo`.

source

`Base.LibGit2.headname` – Function.

```
LibGit2.headname(repo::GitRepo)
```

Lookup the name of the current HEAD of git repository `repo`. If `repo` is currently detached, returns the name of the HEAD it's detached from.

source

`Base.LibGit2.init` – Function.

```
LibGit2.init(path::AbstractString, bare::Bool=false) -> GitRepo
```

Opens a new git repository at `path`. If `bare` is `false`, the working tree will be created in `path/.git`. If `bare` is `true`, no working directory will be created.

source

`Base.LibGit2.is_ancestor_of` – Function.

```
is_ancestor_of(a::AbstractString, b::AbstractString, repo::GitRepo) -> Bool
```

Returns `true` if a, a `GitHash` in string form, is an ancestor of b, a `GitHash` in string form.

Example

```
julia> repo = LibGit2.GitRepo(repo_path);

julia> LibGit2.add!(repo, test_file1);

julia> commit_oid1 = LibGit2.commit(repo, "commit1");

julia> LibGit2.add!(repo, test_file2);

julia> commit_oid2 = LibGit2.commit(repo, "commit2");

julia> LibGit2.is_ancestor_of(string(commit_oid1), string(commit_oid2), repo)
true
```

source

`Base.LibGit2.isbinary` – Function.

Use a heuristic to guess if a file is binary: searching for NULL bytes and looking for a reasonable ratio of printable to non-printable characters among the first 8000 bytes.

source

Base.LibGit2.iscommit – Function.

```
iscommit(id::AbstractString, repo::GitRepo) -> Bool
```

Checks if commit `id` (which is a `GitHash` in string form) is in the repository.

Example

```
julia> repo = LibGit2.GitRepo(repo_path);

julia> LibGit2.add!(repo, test_file);

julia> commit_oid = LibGit2.commit(repo, "add test_file");

julia> LibGit2.iscommit(string(commit_oid), repo)
true
```

source

Base.LibGit2.isdiff – Function.

```
LibGit2.isdiff(repo::GitRepo, treeish::AbstractString, pathspecs::AbstractString=""; cached::
    Bool=false)
```

Checks if there are any differences between the tree specified by `treeish` and the tracked files in the working tree (if `cached=false`) or the index (if `cached=true`). `pathspecs` are the specifications for options for the diff.

Example

```
repo = LibGit2.GitRepo(repo_path)
LibGit2.isdiff(repo, "HEAD") # should be false
open(joinpath(repo_path, new_file), "a") do f
    println(f, "here's my cool new file")
end
LibGit2.isdiff(repo, "HEAD") # now true
```

Equivalent to `git diff-index <treeish> [-- <pathspecs>]`.

source

Base.LibGit2.isdirty – Function.

```
LibGit2.isdirty(repo::GitRepo, pathspecs::AbstractString=""; cached::Bool=false) -> Bool
```

Checks if there have been any changes to tracked files in the working tree (if `cached=false`) or the index (if `cached=true`). `pathspecs` are the specifications for options for the diff.

Example

```
repo = LibGit2.GitRepo(repo_path)
LibGit2.isdirty(repo) # should be false
open(joinpath(repo_path, new_file), "a") do f
    println(f, "here's my cool new file")
end
LibGit2.isdirty(repo) # now true
LibGit2.isdirty(repo, new_file) # now true
```

Equivalent to `git diff-index HEAD [-- <pathspecs>]`.

source

Base.LibGit2.isorphan – Function.

```
LibGit2.isorphan(repo::GitRepo)
```

Checks if the current branch is an "orphan" branch, i.e. has no commits. The first commit to this branch will have no parents.

source

Base.LibGit2.lookup_branch – Function.

```
lookup_branch(repo::GitRepo, branch_name::AbstractString, remote::Bool=false) -> Nullable{
    GitReference}
```

Determine if the branch specified by `branch_name` exists in the repository `repo`. If `remote` is `true`, `repo` is assumed to be a remote git repository. Otherwise, it is part of the local filesystem.

`lookup_branch` returns a `Nullable`, which will be null if the requested branch does not exist yet. If the branch does exist, the `Nullable` contains a `GitReference` to the branch.

source

Base.LibGit2.mirror_callback – Function.

Mirror callback function

Function sets `+refs/*:refs/*` refspecs and `mirror` flag for remote reference.

source

Base.LibGit2.mirror_cb – Function.

C function pointer for `mirror_callback`

source

Base.LibGit2.name – Function.

```
LibGit2.name(ref::GitReference)
```

Return the full name of `ref`.

source

```
name(rmt::GitRemote)
```

Get the name of a remote repository, for instance `"origin"`. If the remote is anonymous (see `GitRemoteAnon`) the name will be an empty string `""`.

Example

```
julia> repo_url = "https://github.com/JuliaLang/Example.jl";

julia> repo = LibGit2.clone(cache_repo, "test_directory");

julia> remote = LibGit2.GitRemote(repo, "origin", repo_url);

julia> name(remote)
"origin"
```

source

```
LibGit2.name(tag::GitTag)
```

The name of `tag` (e.g. `"v0.5"`).

source

`Base.LibGit2.need_update` – Function.

```
need_update(repo::GitRepo)
```

Equivalent to `git update-index`. Returns `true` if `repo` needs updating.

source

`Base.LibGit2.objtype` – Function.

```
objtype(obj_type::Consts.OBJECT)
```

Returns the type corresponding to the enum value.

source

`Base.LibGit2.path` – Function.

```
LibGit2.path(repo::GitRepo)
```

The base file path of the repository `repo`.

- for normal repositories, this will typically be the parent directory of the ".git" directory (note: this may be different than the working directory, see `workdir` for more details).
- for bare repositories, this is the location of the "git" files.

See also `gitdir`, `workdir`.

source

`Base.LibGit2.peel` – Function.

```
peel([T,] ref::GitReference)
```

Recursively peel `ref` until an object of type `T` is obtained. If no `T` is provided, then `ref` will be peeled until an object other than a `GitTag` is obtained.

- A `GitTag` will be peeled to the object it references.
- A `GitCommit` will be peeled to a `GitTree`.

Note

Only annotated tags can be peeled to `GitTag` objects. Lightweight tags (the default) are references under `refs/tags/` which point directly to `GitCommit` objects.

source

```
peel([T,] obj::GitObject)
```

Recursively peel `obj` until an object of type `T` is obtained. If no `T` is provided, then `obj` will be peeled until the type changes.

- A `GitTag` will be peeled to the object it references.
- A `GitCommit` will be peeled to a `GitTree`.

source

`Base.LibGit2.posixpath` – Function.

```
LibGit2.posixpath(path)
```

Standardise the path string `path` to use POSIX separators.

source

`Base.LibGit2.push` – Function.

```
push(rmt::GitRemote, refspecs; force::Bool=false, options::PushOptions=PushOptions())
```

Push to the specified `rmt` remote git repository, using `refspecs` to determine which remote branch(es) to push to. The keyword arguments are:

- `force`: if `true`, a force-push will occur, disregarding conflicts.
- `options`: determines the options for the push, e.g. which proxy headers to use.

Note

You can add information about the push refspecs in two other ways: by setting an option in the repository's `GitConfig` (with `push.default` as the key) or by calling `add_push!`. Otherwise you will need to explicitly specify a push refspec in the call to `push` for it to have any effect, like so: `LibGit2.push(repo, refspecs=["refs/heads/master"])`.

source

```
push(repo::GitRepo; kwargs...)
```

Pushes updates to an upstream of `repo`.

The keyword arguments are:

- `remote::AbstractString="origin"`: the name of the upstream remote to push to.
- `remoteurl::AbstractString=""`: the URL of `remote`.
- `refspecs=AbstractString[]`: determines properties of the push.
- `force::Bool=false`: determines if the push will be a force push, overwriting the remote branch.
- `payload=Nullable{AbstractCredentials}()`: provides credentials, if necessary, for instance if `remote` is a private repository.

Equivalent to `git push [<remoteurl>|<repo>] [<refspecs>]`.

source

`Base.LibGit2.push_refspecs` – Function.

```
push_refspecs(rmt::GitRemote) -> Vector{String}
```

Get the *push* refspecs for the specified `rmt`. These refspecs contain information about which branch(es) to push to.

source

Base.LibGit2.read_tree! – Function.

```
LibGit2.read_tree!(idx::GitIndex, tree::GitTree)
LibGit2.read_tree!(idx::GitIndex, treehash::AbstractGitHash)
```

Read the tree `tree` (or the tree pointed to by `treehash` in the repository owned by `idx`) into the index `idx`. The current index contents will be replaced.

source

Base.LibGit2.rebase! – Function.

```
LibGit2.rebase!(repo::GitRepo, upstream::AbstractString="", newbase::AbstractString="")
```

Attempt an automatic merge rebase of the current branch, from `upstream` if provided, or otherwise from the upstream tracking branch. `newbase` is the branch to rebase onto. By default this is `upstream`.

If any conflicts arise which cannot be automatically resolved, the rebase will abort, leaving the repository and working tree in its original state, and the function will throw a `GitError`. This is roughly equivalent to the following command line statement:

```
git rebase --merge [<upstream>]
if [ -d ".git/rebase-merge" ]; then
    git rebase --abort
fi
```

source

Base.LibGit2.ref_list – Function.

```
LibGit2.ref_list(repo::GitRepo) -> Vector{String}
```

Get a list of all reference names in the `repo` repository.

source

Base.LibGit2.reftype – Function.

```
LibGit2.reftype(ref::GitReference) -> Cint
```

Returns a `Cint` corresponding to the type of `ref`:

- `0` if the reference is invalid
- 1 if the reference is an object id
- 2 if the reference is symbolic

source

Base.LibGit2.remotes – Function.

```
LibGit2.remotes(repo::GitRepo)
```

Returns a vector of the names of the remotes of `repo`.

source

Base.LibGit2.reset! – Function.

Resets credentials for another use

source

Updates some entries, determined by the `pathspecs`, in the index from the target commit tree.

source

Sets the current head to the specified commit oid and optionally resets the index and working tree to match.

source

git reset [<committish>] [–] <pathspecs>...

source

```
reset!(repo::GitRepo, id::GitHash, mode::Cint = Consts.RESET_MIXED)
```

Reset the repository `repo` to its state at `id`, using one of three modes set by `mode`:

1. `Consts.RESET_SOFT` - move HEAD to `id`.
2. `Consts.RESET_MIXED` - default, move HEAD to `id` and reset the index to `id`.
3. `Consts.RESET_HARD` - move HEAD to `id`, reset the index to `id`, and discard all working changes.

Equivalent to `git reset [--soft | --mixed | --hard] <id>`.

Example

```
repo = LibGit2.GitRepo(repo_path)
head_oid = LibGit2.head_oid(repo)
open(joinpath(repo_path, "file1"), "w") do f
    write(f, "111
")
end
LibGit2.add!(repo, "file1")
mode = LibGit2.Consts.RESET_HARD
# will discard the changes to file1
# and unstage it
new_head = LibGit2.reset!(repo, head_oid, mode)
```

source

`Base.LibGit2.restore` – Function.

```
restore(s::State, repo::GitRepo)
```

Return a repository `repo` to a previous `State` `s`, for example the HEAD of a branch before a merge attempt. `s` can be generated using the `snapshot` function.

source

`Base.LibGit2.revcount` – Function.

```
LibGit2.revcount(repo::GitRepo, commit1::AbstractString, commit2::AbstractString)
```

List the number of revisions between `commit1` and `commit2` (committish OIDs in string form). Since `commit1` and `commit2` may be on different branches, `revcount` performs a "left-right" revision list (and count), returning a tuple of `Int`s - the number of left and right commits, respectively. A left (or right) commit refers to which side of a symmetric difference in a tree the commit is reachable from.

Equivalent to `git rev-list --left-right --count <commit1> <commit2>`.

source

Base.LibGit2.set_remote_url – Function.

```
set_remote_url(repo::GitRepo, url::AbstractString; remote::AbstractString="origin")
```

Set the `url` for `remote` for the git repository `repo`. The default name of the remote is `"origin"`.

Examples

```
repo_path = joinpath("test_directory", "Example")
repo = LibGit2.init(repo_path)
url1 = "https://github.com/JuliaLang/Example.jl"
LibGit2.set_remote_url(repo, url1, remote="upstream")
url2 = "https://github.com/JuliaLang/Example2.jl"
LibGit2.set_remote_url(repo_path, url2, remote="upstream2")
```

source

```
set_remote_url(path::AbstractString, url::AbstractString; remote::AbstractString="origin")
```

Set the `url` for `remote` for the git repository located at `path`. The default name of the remote is `"origin"`.

source

Base.LibGit2.shortname – Function.

```
LibGit2.shortname(ref::GitReference)
```

Returns a shortened version of the name of `ref` that's "human-readable".

```
julia> repo = LibGit2.GitRepo(path_to_repo);

julia> branch_ref = LibGit2.head(repo);

julia> LibGit2.name(branch_ref)
"refs/heads/master"

julia> LibGit2.shortname(branch_ref)
"master"
```

source

Base.LibGit2.snapshot – Function.

```
snapshot(repo::GitRepo) -> State
```

Take a snapshot of the current state of the repository `repo`, storing the current HEAD, index, and any uncommitted work. The output `State` can be used later during a call to `restore` to return the repository to the snapshotted state.

source

Base.LibGit2.status – Function.

```
LibGit2.status(repo::GitRepo, path::String)
```

Lookup the status of the file at `path` in the git repository `repo`. For instance, this can be used to check if the file at `path` has been modified and needs to be staged and committed.

source

Base.LibGit2.tag_create – Function.

```
LibGit2.tag_create(repo::GitRepo, tag::AbstractString, commit; kwargs...)
```

Create a new git tag `tag` (e.g. `"v0.5"`) in the repository `repo`, at the commit `commit`.

The keyword arguments are:

- `msg::AbstractString=""`: the message for the tag.
- `force::Bool=false`: if `true`, existing references will be overwritten.
- `sig::Signature=Signature(repo)`: the tagger's signature.

source

Base.LibGit2.tag_delete – Function.

```
LibGit2.tag_delete(repo::GitRepo, tag::AbstractString)
```

Remove the git tag `tag` from the repository `repo`.

source

Base.LibGit2.tag_list – Function.

```
LibGit2.tag_list(repo::GitRepo) -> Vector{String}
```

Get a list of all tags in the git repository `repo`.

source

Base.LibGit2.target – Function.

```
LibGit2.target(tag::GitTag)
```

The `GitHash` of the target object of `tag`.

source

Base.LibGit2.treewalk – Function.

Traverse the entries in a tree and its subtrees in post or pre order.

Function parameter should have following signature:

```
(Cstring, Ptr{Void}, Ptr{Void}) -> Cint
```

source

Base.LibGit2.upstream – Function.

```
upstream(ref::GitReference) -> Nullable{GitReference}
```

Determine if the branch containing `ref` has a specified upstream branch.

`upstream` returns a `Nullable`, which will be null if the requested branch does not have an upstream counterpart. If the upstream branch does exist, the `Nullable` contains a `GitReference` to the upstream branch.

source

Base.LibGit2.url – Function.

```
url(rmt::GitRemote)
```

Get the fetch URL of a remote git repository.

Example

```
julia> repo_url = "https://github.com/JuliaLang/Example.jl";

julia> repo = LibGit2.init(mktempdir());

julia> remote = LibGit2.GitRemote(repo, "origin", repo_url);

julia> LibGit2.url(remote)
"https://github.com/JuliaLang/Example.jl"
```

source

`Base.LibGit2.with` – Function.

Resource management helper function

source

`Base.LibGit2.workdir` – Function.

```
LibGit2.workdir(repo::GitRepo)
```

The location of the working directory of `repo`. This will throw an error for bare repositories.

> **Note**
> This will typically be the parent directory of `gitdir(repo)`, but can be different in some cases: e.g. if either the `core.worktree` configuration variable or the `GIT_WORK_TREE` environment variable is set.

See also `gitdir`, `path`.

source

70.19 Module loading

`Base.require`[@ref] is responsible for loading modules and it also manages the precompilation cache. It is the implementation of the `import` statement.

Experimental features

The features below are experimental and not part of the stable Julia API. Before building upon them inform yourself about the current thinking and whether they might change soon.

Module loading callbacks

It is possible to listen to the modules loaded by `Base.require`, by registering a callback.

```
loaded_packages = Channel{Symbol}()
callback = (mod::Symbol) -> put!(loaded_packages, mod)
push!(Base.package_callbacks, callback)
```

Please note that the symbol given to the callback is a non-unique identifier and it is the responsibility of the callback provider to walk the module chain to determine the fully qualified name of the loaded binding.

The callback below is an example of how to do that:

```
# Get the fully-qualified name of a module.
function module_fqn(name::Symbol)
    fqn = Symbol[name]
    mod = getfield(Main, name)
    parent = Base.module_parent(mod)
    while parent !== Main
        push!(fqn, Base.module_name(parent))
        parent = Base.module_parent(parent)
    end
    fqn = reverse!(fqn)
    return join(fqn, '.')
end
```

Chapter 71

Developing/debugging Julia's C code

71.1 Reporting and analyzing crashes (segfaults)

So you managed to break Julia. Congratulations! Collected here are some general procedures you can undergo for common symptoms encountered when something goes awry. Including the information from these debugging steps can greatly help the maintainers when tracking down a segfault or trying to figure out why your script is running slower than expected.

If you've been directed to this page, find the symptom that best matches what you're experiencing and follow the instructions to generate the debugging information requested. Table of symptoms:

- Segfaults during bootstrap (`sysimg.jl`)
- Segfaults when running a script
- Errors during Julia startup

Version/Environment info

No matter the error, we will always need to know what version of Julia you are running. When Julia first starts up, a header is printed out with a version number and date. If your version is `0.2.0` or higher, please include the output of `versioninfo()` in any report you create:

```
julia> versioninfo()
Julia Version 0.6.2
Commit d386e40* (2017-12-13 18:08 UTC)
Platform Info:
  OS: Linux (x86_64-linux-gnu)
  CPU: Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz
  WORD_SIZE: 64
  BLAS: libopenblas (NO_LAPACKE DYNAMIC_ARCH NO_AFFINITY Prescott)
  LAPACK: liblapack
  LIBM: libopenlibm
  LLVM: libLLVM-3.9.1 (ORCJIT, broadwell)
```

Segfaults during bootstrap (`sysimg.jl`)

Segfaults toward the end of the `make` process of building Julia are a common symptom of something going wrong while Julia is preparsing the corpus of code in the `base/` folder. Many factors can contribute toward this process

dying unexpectedly, however it is as often as not due to an error in the C-code portion of Julia, and as such must typically be debugged with a debug build inside of `gdb`. Explicitly:

Create a debug build of Julia:

```
$ cd <julia_root>
$ make debug
```

Note that this process will likely fail with the same error as a normal `make` incantation, however this will create a debug executable that will offer `gdb` the debugging symbols needed to get accurate backtraces. Next, manually run the bootstrap process inside of `gdb`:

```
$ cd base/
$ gdb -x ../contrib/debug_bootstrap.gdb
```

This will start `gdb`, attempt to run the bootstrap process using the debug build of Julia, and print out a backtrace if (when) it segfaults. You may need to hit `<enter>` a few times to get the full backtrace. Create a gist with the backtrace, the version info, and any other pertinent information you can think of and open a new issue on Github with a link to the gist.

Segfaults when running a script

The procedure is very similar to Segfaults during bootstrap (`sysimg.jl`). Create a debug build of Julia, and run your script inside of a debugged Julia process:

```
$ cd <julia_root>
$ make debug
$ gdb --args usr/bin/julia-debug <path_to_your_script>
```

Note that `gdb` will sit there, waiting for instructions. Type `r` to run the process, and `bt` to generate a backtrace once it segfaults:

```
(gdb) r
Starting program: /home/sabae/src/julia/usr/bin/julia-debug ./test.jl
...
(gdb) bt
```

Create a gist with the backtrace, the version info, and any other pertinent information you can think of and open a new issue on Github with a link to the gist.

Errors during Julia startup

Occasionally errors occur during Julia's startup process (especially when using binary distributions, as opposed to compiling from source) such as the following:

```
$ julia
exec: error -5
```

These errors typically indicate something is not getting loaded properly very early on in the bootup phase, and our best bet in determining what's going wrong is to use external tools to audit the disk activity of the `julia` process:

- On Linux, use `strace`:

  ```
  $ strace julia
  ```

- On OSX, use `dtruss`:

```
$ dtruss -f julia
```

Create a gist with the `strace`/ `dtruss` ouput, the version info, and any other pertinent information and open a new issue on Github with a link to the gist.

Glossary

A few terms have been used as shorthand in this guide:

- `<julia_root>` refers to the root directory of the Julia source tree; e.g. it should contain folders such as `base`, `deps`, `src`, `test`, etc.....

71.2 gdb debugging tips

Displaying Julia variables

Within `gdb`, any `jl_value_t*` object `obj` can be displayed using

```
(gdb) call jl_(obj)
```

The object will be displayed in the `julia` session, not in the gdb session. This is a useful way to discover the types and values of objects being manipulated by Julia's C code.

Similarly, if you're debugging some of Julia's internals (e.g., `inference.jl`), you can print `obj` using

```
ccall(:jl_, Void, (Any,), obj)
```

This is a good way to circumvent problems that arise from the order in which julia's output streams are initialized.

Julia's flisp interpreter uses `value_t` objects; these can be displayed with `call fl_print(fl_ctx, ios_stdout, obj)`.

Useful Julia variables for Inspecting

While the addresses of many variables, like singletons, can be be useful to print for many failures, there are a number of additional variables (see `julia.h` for a complete list) that are even more useful.

- (when in `jl_apply_generic`) `mfunc` and `jl_uncompress_ast(mfunc->def, mfunc->code)` :: for figuring out a bit about the call-stack
- `jl_lineno` and `jl_filename` :: for figuring out what line in a test to go start debugging from (or figure out how far into a file has been parsed)
- $1 :: not really a variable, but still a useful shorthand for referring to the result of the last gdb command (such as `print`)
- `jl_options` :: sometimes useful, since it lists all of the command line options that were successfully parsed
- `jl_uv_stderr` :: because who doesn't like to be able to interact with stdio

Useful Julia functions for Inspecting those variables

- `jl_gdblookup($rip)` :: For looking up the current function and line. (use `$eip` on i686 platforms)
- `jlbacktrace()` :: For dumping the current Julia backtrace stack to stderr. Only usable after `record_backtrace()` has been called.
- `jl_dump_llvm_value(Value*)` :: For invoking `Value->dump()` in gdb, where it doesn't work natively. For example, `f->linfo->functionObject`, `f->linfo->specFunctionObject`, and `to_function(f->linfo)`.
- `Type->dump()` :: only works in lldb. Note: add something like `;1` to prevent lldb from printing its prompt over the output
- `jl_eval_string("expr")` :: for invoking side-effects to modify the current state or to lookup symbols
- `jl_typeof(jl_value_t*)` :: for extracting the type tag of a Julia value (in gdb, call `macro define jl_typeof jl_typeof` first, or pick something short like `ty` for the first arg to define a shorthand)

Inserting breakpoints for inspection from gdb

In your gdb session, set a breakpoint in `jl_breakpoint` like so:

```
(gdb) break jl_breakpoint
```

Then within your Julia code, insert a call to `jl_breakpoint` by adding

```
ccall(:jl_breakpoint, Void, (Any,), obj)
```

where `obj` can be any variable or tuple you want to be accessible in the breakpoint.

It's particularly helpful to back up to the `jl_apply` frame, from which you can display the arguments to a function using, e.g.,

```
(gdb) call jl_(args[0])
```

Another useful frame is `to_function(jl_method_instance_t *li, bool cstyle)`. The `jl_method_instance_t*` argument is a struct with a reference to the final AST sent into the compiler. However, the AST at this point will usually be compressed; to view the AST, call `jl_uncompress_ast` and then pass the result to `jl_`:

```
#2  0x00007ffff7928bf7 in to_function (li=0x2812060, cstyle=false) at codegen.cpp:584
584          abort();
(gdb) p jl_(jl_uncompress_ast(li, li->ast))
```

Inserting breakpoints upon certain conditions

Loading a particular file

Let's say the file is `sysimg.jl`:

```
(gdb) break jl_load if strcmp(fname, "sysimg.jl")==0
```

Calling a particular method

```
(gdb) break jl_apply_generic if strcmp((char*)(jl_symbol_name)(jl_gf_mtable(F)->name), "
    method_to_break")==0
```

Since this function is used for every call, you will make everything 1000x slower if you do this.

Dealing with signals

Julia requires a few signal to function property. The profiler uses `SIGUSR2` for sampling and the garbage collector uses `SIGSEGV` for threads synchronization. If you are debugging some code that uses the profiler or multiple threads, you may want to let the debugger ignore these signals since they can be triggered very often during normal operations. The command to do this in GDB is (replace `SIGSEGV` with `SIGUSRS` or other signals you want to ignore):

```
(gdb) handle SIGSEGV noprint nostop pass
```

The corresponding LLDB command is (after the process is started):

```
(lldb) pro hand -p true -s false -n false SIGSEGV
```

If you are debugging a segfault with threaded code, you can set a breakpoint on `jl_critical_error` (`sigdie_handler` should also work on Linux and BSD) in order to only catch the actual segfault rather than the GC synchronization points.

Debugging during Julia's build process (bootstrap)

Errors that occur during `make` need special handling. Julia is built in two stages, constructing `sys0` and `sys.ji`. To see what commands are running at the time of failure, use `make VERBOSE=1`.

At the time of this writing, you can debug build errors during the `sys0` phase from the `base` directory using:

```
julia/base$ gdb --args ../usr/bin/julia-debug -C native --build ../usr/lib/julia/sys0 sysimg.jl
```

You might need to delete all the files in `usr/lib/julia/` to get this to work.

You can debug the `sys.ji` phase using:

```
julia/base$ gdb --args ../usr/bin/julia-debug -C native --build ../usr/lib/julia/sys -J ../usr/
    lib/julia/sys0.ji sysimg.jl
```

By default, any errors will cause Julia to exit, even under gdb. To catch an error "in the act", set a breakpoint in `jl_error` (there are several other useful spots, for specific kinds of failures, including: `jl_too_few_args`, `jl_too_many_args`, and `jl_throw`).

Once an error is caught, a useful technique is to walk up the stack and examine the function by inspecting the related call to `jl_apply`. To take a real-world example:

```
Breakpoint 1, jl_throw (e=0x7ffdf42de400) at task.c:802
802 {
(gdb) p jl_(e)
ErrorException("auto_unbox: unable to determine argument type")
$2 = void
(gdb) bt 10
#0  jl_throw (e=0x7ffdf42de400) at task.c:802
#1  0x00007ffff65412fe in jl_error (str=0x7ffde56be000 <_j_str267> "auto_unbox:
   unable to determine argument type")
   at builtins.c:39
#2  0x00007ffde56bd01a in julia_convert_16886 ()
#3  0x00007ffff6541154 in jl_apply (f=0x7ffdf367f630, args=0x7fffffffc2b0, nargs=2) at julia.h
    :1281
...
```

The most recent `jl_apply` is at frame #3, so we can go back there and look at the AST for the function `julia_convert_16886`. This is the uniqued name for some method of `convert`. `f` in this frame is a `jl_function_t*`, so we can look at the type signature, if any, from the `specTypes` field:

```
(gdb) f 3
#3  0x00007ffff6541154 in jl_apply (f=0x7ffdf367f630, args=0x7fffffffc2b0, nargs=2) at julia.h
    :1281
1281            return f->fptr((jl_value_t*)f, args, nargs);
(gdb) p f->linfo->specTypes
$4 = (jl_tupletype_t *) 0x7ffdf39b1030
(gdb) p jl_( f->linfo->specTypes )
Tuple{Type{Float32}, Float64}           # <-- type signature for julia_convert_16886
```

Then, we can look at the AST for this function:

```
(gdb) p jl_( jl_uncompress_ast(f->linfo, f->linfo->ast) )
Expr(:lambda, Array{Any, 1}[:#s29, :x], Array{Any, 1}[Array{Any, 1}[], Array{Any, 1}[Array{Any,
    1}[:#s29, :Any, 0], Array{Any, 1}[:x, :Any, 0]], Array{Any, 1}[], 0], Expr(:body,
Expr(:line, 90, :float.jl)::Any,
Expr(:return, Expr(:call, :box, :Float32, Expr(:call, :fptrunc, :Float32, :x)::Any)::Any)::Any)::
    Any)::Any
```

Finally, and perhaps most usefully, we can force the function to be recompiled in order to step through the codegen process. To do this, clear the cached `functionObject` from the `jl_lamdbda_info_t*`:

```
(gdb) p f->linfo->functionObject
$8 = (void *) 0x1289d070
(gdb) set f->linfo->functionObject = NULL
```

Then, set a breakpoint somewhere useful (e.g. `emit_function`, `emit_expr`, `emit_call`, etc.), and run codegen:

```
(gdb) p jl_compile(f)
... # your breakpoint here
```

Debugging precompilation errors

Module precompilation spawns a separate Julia process to precompile each module. Setting a breakpoint or catching failures in a precompile worker requires attaching a debugger to the worker. The easiest approach is to set the debugger watch for new process launches matching a given name. For example:

```
(gdb) attach -w -n julia-debug
```

or:

```
(lldb) process attach -w -n julia-debug
```

Then run a script/command to start precompilation. As described earlier, use conditional breakpoints in the parent process to catch specific file-loading events and narrow the debugging window. (some operating systems may require alternative approaches, such as following each `fork` from the parent process)

Mozilla's Record and Replay Framework (rr)

Julia now works out of the box with rr, the lightweight recording and deterministic debugging framework from Mozilla. This allows you to replay the trace of an execution deterministically. The replayed execution's address spaces, register contents, syscall data etc are exactly the same in every run.

A recent version of rr (3.1.0 or higher) is required.

71.3 Usando Valgrind con Julia

Valgrind es una herramienta para deputación de memoria, deteccin de fallos de página y creación de perfiles. Esta sección describe cosas a tner en cuenta cuando se usa Valgrind para depurar cuestiones de memoria con Julia.

Consideraciones Generales

Por defecto, Valgrind asume que no hay código automodificador en el programa que está ejecutando. Esta suposición trabaja bien en la mayoría de las instancias, pero falla miserablemente para un compilador JIT como `julia`. Por esta razón es crucial pasar `--smc-check=all-non-file` a `valgrind`, si no el código puede bloquearse o comportarse de forma inesperada (frecuentemente de una forma sutil).

En algunos casos, para detectar mejor errores de memoria usando Valgrind puede ayudar compilar `julia` con los *pools* de memoria dehabilitados. El flag de tiempo de compilación `MEMDEBUG` desabilita los *pools* de memoria en Julia y el flag `MEMDEBUG2` deshabilita los pools de memoria en FemtoLisp. Para construir `julia` con ambos flags, añada la siguiente línea a `Make.user`:

```
CFLAGS = -DMEMDEBUG -DMEMDEBUG2
```

Otra cosa a notar: si nuestros programa usa múltiples procesos workers, es probable que queramos que todos esos procesos worker se ejecuten bajo Valgrind, no sólo bajo el proceso padre. Para hacer esto, pasaremos `--trace-children=yes` a `valgrind`.

Supresiones

Valgrind tipicamente mostrará avisos falsos mientras se ejecuta. Para reducir el número de tales avisos, ayuda proporcionar un fichero supresiones a Valgrind. Un fichero supresiones de ejemplo se incluye en la distribución fuente de Julia en `contrib/valgrind-julia.supp`.

El fichero de supresiones puede ser usado desde el directorio fuente `julia/` de la siguiente manera:

```
$ valgrind --smc-check=all-non-file --suppressions=contrib/valgrind-julia.supp ./julia progname.
    jl
```

Any memory errors that are displayed should either be reported as bugs or contributed as additional suppressions. Note that some versions of Valgrind are shipped with insufficient default suppressions, so that may be one thing to consider before submitting any bugs.

Running the Julia test suite under Valgrind

It is possible to run the entire Julia test suite under Valgrind, but it does take quite some time (typically several hours). To do so, run the following command from the `julia/test/` directory:

```
valgrind --smc-check=all-non-file --trace-children=yes --suppressions=$PWD/../contrib/valgrind-
    julia.supp ../julia runtests.jl all
```

If you would like to see a report of "definite" memory leaks, pass the flags `--leak-check=full --show-leak-kinds=definite` to `valgrind` as well.

Caveats

Valgrind currently does not support multiple rounding modes, so code that adjusts the rounding mode will behave differently when run under Valgrind.

In general, if after setting `--smc-check=all-non-file` you find that your program behaves differently when run under Valgrind, it may help to pass `--tool=none` to `valgrind` as you investigate further. This will enable the minimal Valgrind machinery but will also run much faster than when the full memory checker is enabled.

71.4 Sanitizer support

General considerations

Using Clang's sanitizers obviously require you to use Clang (`USECLANG=1`), but there's another catch: most sanitizers require a run-time library, provided by the host compiler, while the instrumented code generated by Julia's JIT relies on functionality from that library. This implies that the LLVM version of your host compiler matches that of the LLVM library used within Julia.

An easy solution is to have an dedicated build folder for providing a matching toolchain, by building with `BUILD_LLVM_CLANG=1` and overriding `LLVM_USE_CMAKE=1` (Autotool-based builds are incompatible with ASAN). You can then refer to this toolchain from another build folder by specifying `USECLANG=1` while overriding the `CC` and `CXX` variables.

Address Sanitizer (ASAN)

For detecting or debugging memory bugs, you can use Clang's address sanitizer (ASAN). By compiling with `SANITIZE=1` you enable ASAN for the Julia compiler and its generated code. In addition, you can specify `LLVM_SANITIZE=1` to sanitize the LLVM library as well. Note that these options incur a high performance and memory cost. For example, using ASAN for Julia and LLVM makes `testall1` takes 8-10 times as long while using 20 times as much memory (this can be reduced to respectively a factor of 3 and 4 by using the options described below).

By default, Julia sets the `allow_user_segv_handler=1` ASAN flag, which is required for signal delivery to work properly. You can define other options using the `ASAN_OPTIONS` environment flag, in which case you'll need to repeat the default option mentioned before. For example, memory usage can be reduced by specifying `fast_unwind_on_malloc=0` and `malloc_context_size=2`, at the cost of backtrace accuracy. For now, Julia also sets `detect_leaks=0`, but this should be removed in the future.

Memory Sanitizer (MSAN)

For detecting use of uninitialized memory, you can use Clang's memory sanitizer (MSAN) by compiling with `SANITIZE_MEMORY=1`.