22 Julia编译器（一）：如何让动态语言性能很高?

你好，我是宫文学。

Julia这门语言，其实是最近几年才逐渐获得人们越来越多的关注的。有些人会拿它跟Python进行比较，认为Julia已经成为了Python的劲敌，甚至还有人觉得它在未来可能会取代Python的地位。虽然这样的说法可能是有点夸张了，不过Julia确实是有它的可取之处的。

为什么这么说呢？前面我们已经研究了Java、Python和JavaScript这几门主流语言的编译器，这几门语言都是很有代表性的：Java语言是静态类型的、编译型的语言；Python语言是动态类型的、解释型的语言；JavaScript是动态类型的语言，但可以即时编译成本地代码来执行。

而Julia语言却声称同时兼具了静态编译型和动态解释型语言的优点：**一方面它的性能很高，可以跟Java和C语言媲美；而另一方面，它又是动态类型的，编写程序时不需要指定类型。**一般来说，我们很难能期望一门语言同时具有动态类型和静态类型语言的优点的，那么Julia又是如何实现这一切的呢？

原来它是充分利用了LLVM来实现即时编译的功能。因为LLVM是Clang、Rust、Swift等很多语言所采用的后端工具，所以我们可以借助Julia语言的编译器，来研究如何恰当地利用LLVM。不过，Julia使用LLVM的方法很有创造性，使得它可以同时具备这两类语言的优点。我将在这一讲中给你揭秘。

此外，Julia编译器的类型系统的设计也很独特，它体现了函数式编程的一些设计哲学，能够帮助你启迪思维。

还有一点，Julia来自MIT，这里也曾经是Lisp的摇篮，所以Julia有一种学术风和极客风相结合的品味，也值得你去仔细体会一下。

所以，接下来的两讲，我会带你来好好探究一下Julia的编译器。你从中能够学习到Julia编译器的处理过程，如何创造性地使用LLVM的即时编译功能、如何使用LLVM的优化功能，以及它的独特的类型系统和方法分派。

那今天这一讲，我会先带你来了解Julia的编译过程，以及它高性能背后的原因。

初步认识Julia

Julia的性能有多高呢? 你可以去它的网站上看看与其他编程语言的性能对比：

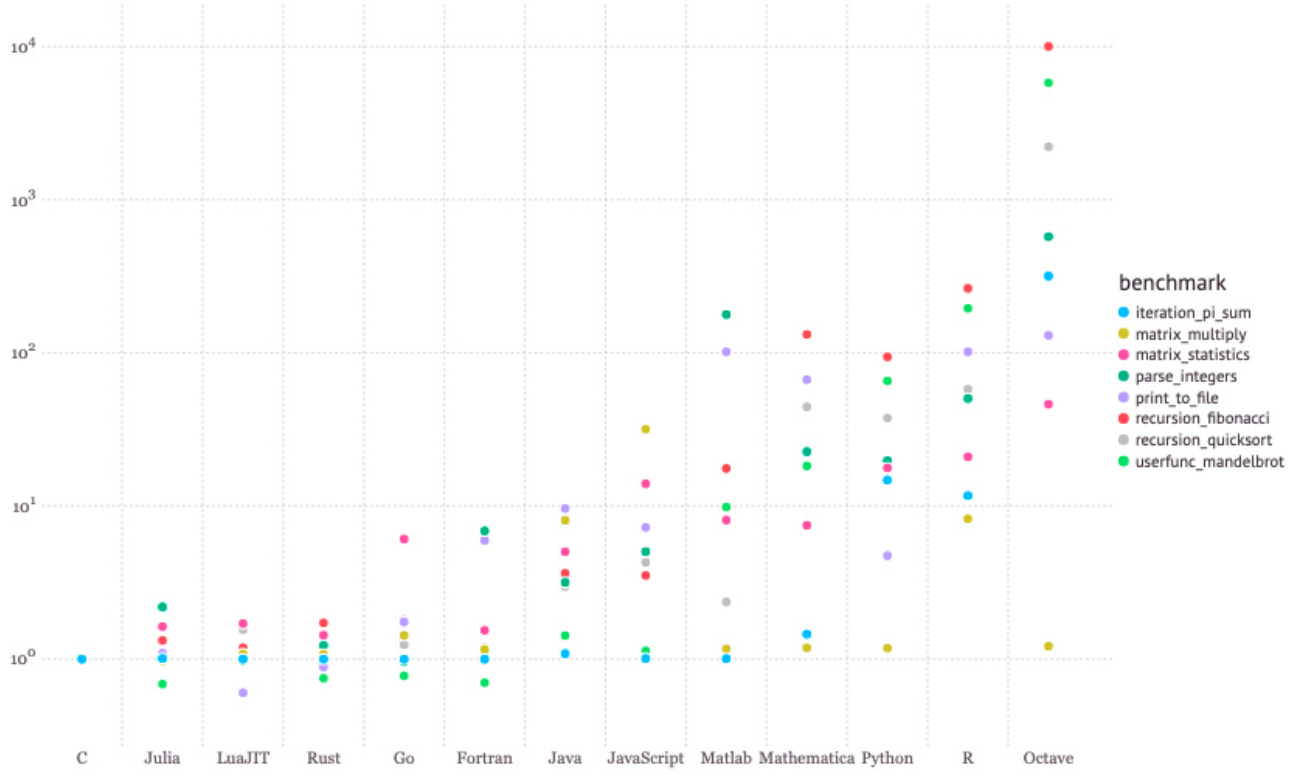


图1：Julia和各种语言的性能对比

可以看出，它的性能是在C、Rust这一个级别的，很多指标甚至比Java还要好，比起那些动态语言（如Python、R和Octave），那更是高了一到两个数量级。

所以，Julia的编译器声称它具备了静态类型语言的性能，确实是不虚此言的。

你可以从Julia的官网下载Julia的二进制版本和源代码。如果你下载的是源代码，那你可以用make debug编译成debug版本，这样比较方便用GDB或LLDB调试。

Julia的设计目的主要是用于**科学计算**。过去，这一领域的用户主要是使用R语言和Python，但麻省理工（MIT）的研究者们对它们的性能不够满意，同时又想要保留R和Python的友好性，于是就设计出了这门新的语言。目前这门语言受到了很多用户的欢迎，使用者也在持续地上升中。

我个人对它感兴趣的点，正是因为它打破了静态编译和动态编译语言的边界，我认为这体现了未来语言的趋势：**编译和优化的过程是全生命周期的，而不局限在某个特定阶段。**

好了，让我们先通过一个例子来认识Juia，直观了解一下这门语言的特点：

```
julia> function countdown(n)
           if n <= 0
```

```
            println("end")
        else
            print(n, " ")
            countdown(n-1)
        end
    end
countdown (generic function with 1 method)

julia> countdown(10)
10 9 8 7 6 5 4 3 2 1 end
```

所以从这段示例代码中，可以看出，Julia主要有这样几个特点：

- 用function关键字来声明一个函数；
- 用end关键字作为块（函数声明、if语句、for语句等）的结尾；
- 函数的参数可以不用指定类型（变量声明也不需要），因为它是动态类型的；
- Julia支持递归函数。

那么Julia的编译器是用什么语言实现的呢？又是如何支持它的这些独特的特性的呢？带着这些好奇，让我们来看一看Julia编译器的源代码。

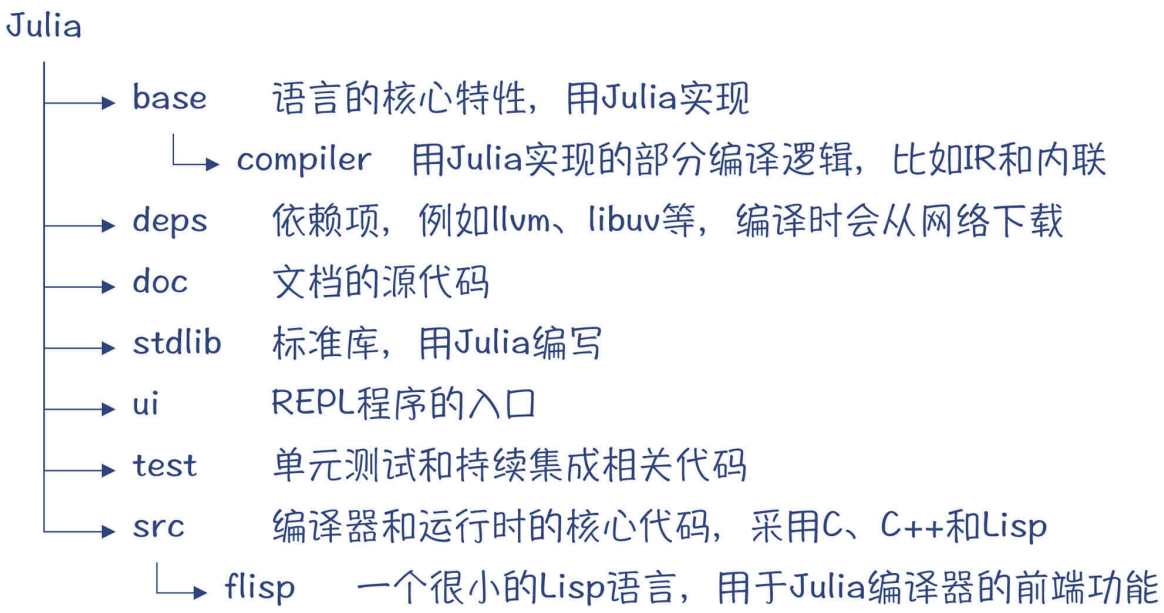


图2：Julia的源代码目录

其实Julia的实现会让人有点困扰，因为它使用了4种语言：C、C++、Lisp和Julia自身。相比而言，CPython的实现只用了两种语言：C语言和Python。这种情况，就对社区的其他技术人员理解这个编译器和参与开发，带来了不小的困难。

Julia的作者用C语言实现了一些运行时的核心功能，包括垃圾收集器。他们是比较偏爱C语言的。C++主要是用来实现跟LLVM衔接的功能，因为LLVM是用C++实现的。

但是，为什么又冒出了一个Lisp语言呢？而且前端部分的主要功能都是用Lisp实现的。

原来，Julia中用到Lisp叫做femtolisp（简称flisp），这是杰夫·贝赞松（Jeff Bezanson）做的一个开源Lisp实现，当时的目标是做一个最小的、编译速度又最快的Lisp版本。后来Jeff Bezanson作为Julia的核心开发人员，又把flisp带进了Julia。

实际上，Julia语言本身也宣称是继承了Lisp语言的精髓。在其核心的设计思想里，在函数式编程风格和元编程功能方面，也确实是如此。Lisp在研究界一直有很多的追随者，Julia这个项目诞生于MIT，同时又主要服务于各种科研工作者，所以它也就带上了这种科学家的味道。它还有其他特性，也能看出这种科研工作者的倾向，比如说：

- 对于类型系统，Julia的开发者们进行了很好的形式化，是我在所有语言中看到的最像数学家做的类型系统。
- 在它的语法和语义设计上，带有Metalab和Mathematics这些数学软件的痕迹，科研工作者们应该很熟悉这种感觉。
- 在很多特性的实现上，都带有很强的前沿探索的特征，锋芒突出，不像我们平常使用的那些商业公司设计的计算机语言一样，追求四平八稳。

以上就是我对Julia的感觉，一种**结合了数据家风格的自由不羁的极客风**。实际上，Lisp最早的设计者约翰·麦卡锡（John McCarthy）就是一位数学博士，所以数学上的美感是Lisp给人的感受之一。而且，Lisp语言本身也是在MIT发源的，所以Julia可以说是继承了这个传统、这种风格。

Julia的编译过程

刚刚说了，Julia的前端主要是用Lisp来实现的。你在启动Julia的时候，通过“–lisp”参数就可以进入flisp的REPL：

```
./julia --lisp
```

在这个REPL界面中调用一个julia-parse函数，就可以把一个Julia语句编译成AST。

```
> (julia-parse "a = 2+3*5")
(= a (call + 2
           (call * 3 5)))
> (julia-parse "function countdown(n)
                        if n <= 0
                                println(\"end\")
```

```
                        else
                                print(n, \" \")
                                countdown(n-1)
                        end
                end")
(function (call countdown n) (block (line 2 none)
                                        (if (call <= n 0)
                                              (block (line 3 none)
                                                     (call println "end"))
                                              (block (line 5 none)
                                                     (call print n " ")
                                                     (line 6 none)
                                                     (call countdown (call - n 1)))
```

编译后的AST，采用的也是Lisp那种括号嵌套括号的方式。

Julia的编译器中，主要用到了几个“.scm”结尾的代码，来完成词法和语法分析的功能：julia-parser.scm、julia-syntax.scm和ast.scm。（.scm文件是Scheme的缩写，而Scheme是Lisp的一种实现，特点是设计精简、语法简单。著名的计算机教科书SICP就是用Scheme作为教学语言，而SICP和Scheme也都是源自MIT。）它的词法分析和语法分析的过程，主要是在parser.scm文件里实现的，我们刚才调用的“julia-parse”函数就是在这个文件中声明的。

Julia的语法分析过程仍然是你非常熟悉的递归下降算法。因为Lisp语言处理符号的能力很强，又有很好的元编程功能（宏），所以Lisp在实现词法分析和语法分析的任务的时候，代码会比其他语言更短。但是不熟悉Lisp语言的人，可能会看得一头雾水，因为这种括号嵌套括号的语言对于人类的阅读不那么友好，不像Java、JavaScript这样的语言一样，更像自然语言。

julia-parser.scm输出的成果是比较经典的AST，Julia的文档里叫做“表面语法AST”（surface syntax AST）。所谓表面语法AST，它是跟另一种IR对应的，叫做Lowered Form。

“Lowered”这个词你应该已经很熟悉了，它的意思是**更靠近计算机的物理实现机制**。比如，LLVM的IR跟AST相比，就更靠近底层实现，也更加不适合人类阅读。

julia-syntax.scm输出的结果就是Lowered Form，这是一种内部IR。它比AST的节点类型更少，所有的宏都被展开了，控制流也简化成了无条件和有条件跳转的节点（“goto”格式）。这种IR后面被用来做类型推断和代码生成。

你查看julia-syntax.scm的代码，会发现Julia编译器的处理过程是由多个Pass构成的，包括了去除语法糖、识别和重命名本地变量、分析变量的作用域和闭包、把闭包函数做转换、转换成线性IR、记录Slot和标签（label）等。

这里，我根据Jeff Bezanson在JuliaCon上讲座的内容，把Julia编译器的工作过程、每个阶段涉及的源代码和主要的函数给你概要地梳理了一下，你可以只看这张图，就能大致把握Julia的编译过程，并且可以把它跟你学过的其他几个编译器做一下对比：

阶段	主要源代码	主要函数和宏
1.Parse：文本-->Expr	julia-parser.scm	parse
2.扩展宏	julia-syntax.scm ast.c	macroexpand
3.去除语法糖	julia-syntax.scm	
4.把控制流变成语句	julia-syntax.scm	
5.作用域消解	julia-syntax.scm	
6.生成IR（“goto”格式）	julia-syntax.scm	expand，code_lowered
7.顶层求值，方法排序	toplevel.c interpreter.c gf.c	methods
8.类型推断	inference.jl	
9.内联(Inlining)、高层优化	optimize.jl	code_typed
10.生成LLVM IR	codegen.cpp cgutils.cpp ccall.cpp intrinsics.cpp	code_llvm
11.LLVM优化，生成本地代码		code_native

图3：Julia的编译过程

Julia有很好的反射(Reflection)和自省(Introspection)的能力，你可以调用一些函数或者宏来观察各个阶段的成果。

比如，采用@code_lowered宏，来看countdown(10)产生的代码，你会看到if...else...的结构被转换成了“goto”语句，这个IR看上去已经有点像LLVM的IR格式了。

```
julia> @code_lowered countdown(10)
CodeInfo(
1 ─ %1 = n <= 0
└──      goto #3 if not %1
2 ─ %3 = Main.println("end")
└──      return %3
3 ─      Main.print(n, " ")
│    %6 = n - 1
│    %7 = Main.countdown(%6)
└──      return %7
)
```

进一步，你还可以用@code_typed宏，来查看它做完类型推断以后的结果，每条语句都标注了类型：

```
julia> @code_typed countdown(10)
CodeInfo(
1 ─ %1  = Base.sle_int(n, 0)::Bool
└──       goto #3 if not %1
2 ─ %3  = invoke Main.println("end"::String)::Core.Compiler.Const(nothing, false)
└──       return %3
3 ─       invoke Main.print(_2::Int64, " "::String)::Any
│   %6  = Base.sub_int(n, 1)::Int64
│   %7  = Base.sle_int(%6, 0)::Bool
└──       goto #5 if not %7
4 ─ %9  = invoke Main.println("end"::String)::Nothing
└──       goto #6
5 ─       invoke Main.print(%6::Int64, " "::String)::Any
│   %12 = Base.sub_int(%6, 1)::Int64
│   %13 = invoke Main.countdown(%12::Int64)::Nothing
└──       goto #6
6 ┄ %15 = φ (#4 => %9, #5 => %13)::Core.Compiler.Const(nothing, false)
└──       return %15
) => Nothing
```

接下来，你可以用@code_llvm和@code_native宏，来查看生成的LLVM IR和汇编代码。这次，我们用一个更简单的函数foo()，让生成的代码更容易看懂：

```
julia> function foo(x,y)   #一个简单的函数，把两个参数相加
           x+y             #最后一句的结果就是返回值，这里可以省略return
       end
```

通过@code_llvm宏生成的LLVM IR，如下图所示：

```
julia> @code_llvm foo(2,3)

;  @ REPL[5]:2 within `foo'
define i64 @julia_foo_17088(i64, i64) {
top:
; ┌ @ int.jl:53 within `+'
   %2 = add i64 %1, %0
; └
  ret i64 %2
}
```

通过@code_native宏输出的汇编代码是这样的：

```
julia> @code_native foo(2,3)
        .section        __TEXT,__text,regular,pure_instructions
; ┌ @ REPL[5]:2 within `foo'
; │┌ @ REPL[5]:2 within `+'
        leaq    (%rdi,%rsi), %rax
; │└
        retq
        nopw    %cs:(%rax,%rax)
; └
```

最后生成的汇编代码，可以通过汇编器迅速生成机器码并运行。

通过上面的梳理，你应该已经了解了Julia的编译过程脉络：**通过Lisp的程序，把程序变成AST，然后再变成更低级一点的IR，在这个过程中编译器要进行类型推断等语义层面的处理；最后，翻译成LLVM的IR，并生成可执行的本地代码。**

对于静态类型的语言来说，我们根据准确的类型信息，就可以生成高效的本地代码，这也是C语言性能高的原因。比如，我们用C语言来写一下foo函数：

```c
long foo(long x, long y){
    return x+y;
}
```

Clang的LLVM IR跟Julia生成的基本一样：

```
richard@Richard-MBP MyJulia % clang -S -O2 -emit-llvm foo.c
richard@Richard-MBP MyJulia % cat foo.ll
; ModuleID = 'foo.c'
source_filename = "foo.c"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.15.0"

; Function Attrs: norecurse nounwind readnone ssp uwtable
define i64 @foo(i64, i64) local_unnamed_addr #0 {
  %3 = add nsw i64 %1, %0
  ret i64 %3
}
```

生成的汇编代码也差不多：

```
richard@Richard-MBP MyJulia % clang -S -O2 foo.c
richard@Richard-MBP MyJulia % cat foo.s
        .section        __TEXT,__text,regular,pure_instructions
        .build_version macos, 10, 15    sdk_version 10, 15
        .globl  _foo                    ## -- Begin function foo
        .p2align        4, 0x90
_foo:                                   ## @foo
        .cfi_startproc
## %bb.0:
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register %rbp
        leaq    (%rdi,%rsi), %rax
        popq    %rbp
        retq
        .cfi_endproc
                                        ## -- End function
```

所以，对于这样的程序，Julia的编译后的本地代码，跟C语言的比起来可以说是完全一样。那性能高也就不足为奇了。

你可能由此就会下结论：因为Julia能够借助LLVM生成本地代码，这就是它性能高的原因。

且慢！事情没有这么简单。为什么这么说？因为在基于前面生成的机器码的这个例子中，当参数是整型的时候，运行效率自然是会比较快。但是，你别忘了Julia是动态类型的语言。我们在Julia中声明foo函数的时候，并没有指定参数的数据类型。如果参数类型变了，会怎样呢？

Julia的最大突破：生成多个版本的目标代码

实际上，我们可以给它传递不同的参数，比如可以传递两个浮点数给它，甚至传递两个向量或者矩阵给它，都能得到正确的结果：

```
julia> foo(2.1, 3.2)
5.300000000000001

julia> foo([1,2,3], [3,4,5])
3-element Array{Int64,1}:
 4
 6
 8
```

显然，如果上面两次对foo()函数的调用，我们也是用之前生成的汇编代码，那是行不通的。因为之前的汇编代码只能用于处理64位的整数。

实际上，如果我们观察调用foo(2.1, 3.2)时，Julia生成的LLVM IR和汇编代码，就会发现，它智能地适应了新的数据类型，生成了用于处理浮点数的代码，使用了不同的指令和寄存器。

```
julia> @code_llvm foo(2.1, 3.2)

;  @ REPL[5]:2 within `foo'
define double @julia_foo_17094(double, double) {
top:
; ┌ @ float.jl:401 within `+'
   %2 = fadd double %0, %1
; └
  ret double %2
}

julia> @code_native foo(2.1, 3.2)
        .section        __TEXT,__text,regular,pure_instructions
; ┌ @ REPL[5]:2 within `foo'
; │┌ @ REPL[5]:2 within `+'
        vaddsd  %xmm1, %xmm0, %xmm0
; │└
        retq
        nopw    %cs:(%rax,%rax)
; └
```

你可以用同样的方法，来试一下 `foo([1,2,3], [3,4,5])` 对应的LLVM IR和汇编代码。这个就要复杂一点了，因为它要处理数组的存储。但不管怎样，Julia生成的代码确实是适应了它的参数类型的。

数学中的很多算法，其实是概念层面的，它不关心涉及的数字是32位整数、64位整数，还是一个浮点数。但同样是实现一个加法操作，对于计算机内部实现来说，不同的数据类型对应的指令则是完全不同的，那么编译器就要弥合抽象的算法和计算机的具体实现之间的差别。

对于C语言这样的静态语言来说，它需要针对x、y的各种不同的数据类型，分别编写不同的函数。这些函数的逻辑是一样的，但就是因为数据类型不同，我们就要写很多遍。这是不太合理的，太啰嗦了。

对于Python这样的动态类型语言来说呢，倒是简洁地写一遍就可以了。但在运行时，对于每一次运算，我们都要根据数据类型来选择合适的操作。这样就大大拉低了整体的运行效率。

所以，这才是Julia真正的突破：**它能针对同一个算法，根据运行时获得的数据，进行类型推断，并编译生成最优化的本地代码。**在每种参数类型组合的情况下，只要编译一次，就可以被缓存下来，可以使用很多次，从而使得程序运行的总体性能很高。

你对比一下JavaScript编译器基于类型推断的编译优化，就会发现它们之间的明显的不同。JavaScript编译器一般只会针对类型推断生成一个版本的目标代码，而Julia则会针对每种参数类型组合，都生成一个版本。

不过，既然Julia编译器存在多个版本的目标代码，那么在运行期，就要有一个程序来确定到底采用哪个版本的目标代码，这就是Julia的一个非常重要的功能：**函数分派算法。**

函数分派，就是指让编译器在编译时或运行时来确定采用函数的哪个实现版本。针对函数分派，我再给你讲一下Julia的一个特色功能，多重分派。这个知识点有助于你加深对于函数分派的理解，也有助于你理解函数式编程的特点。

Julia的多重分派功能

我们在编程的时候，经常会涉及同一个函数名称的多个实现。比如在做面向对象编程的时候，同一个类里可以有多个相同名称的方法，但参数不同，这种现象有时被叫做**重载（Overload）**；同时，在这个类的子类里，也可以定义跟父类完全相同的方法，这种现象叫做**覆盖（Override）**。

而程序在调用一个方法的时候，到底调用的是哪个实现，有时候我们在编译期就能确定下来，有时候却必须到运行期才能确定（就像多态的情形），这两种情形就分别叫做**静态分派（Static Dispatch）和动态分派（Dynamic Dispatch）**。

方法的分派还有另一个分类：**单一分派（Single Dispatch）和多重分派（Multiple Dispatch）**。传统的面向对象的语言使用的都是单一分派。比如，在面向对象语言里面，实现加法的运算：

```
a.add(b)
```

这里我们假设a和b都有多个add方法的版本，但实际上，无论怎样分派，程序的调用都是分派到a对象的方法上。这是因为，对于add方法，实质上它的第一个参数是对象a（编译成目标代码时，a会成为第一个参数，以便访问封装在a里面的数据），也就是相当于这样一个函数：

```
add(a, b)
```

所以，**面向对象的方法分派相当于是由第一个参数决定的。这种就是单一分派。**

实际上，采用面向对象的编程方式，在方法分派时经常会让人觉得很别扭。你回顾一下，我在讲Python编译器的时候，讲到加法操作采用的实现是第一个操作数对象的类型里，定义的与加法有关的函数。**但为什么它是用第一个对象的方法，而不是第二个对象的呢？如果第一个对象和第二个对象的类型不同怎么办呢？**（这就是我在那讲中留给你的问题）

还有一个很不方便的地方。如果你增加了一种新的数据类型，比如矩阵（Matrix），它要能够跟整数、浮点数等进行加减乘除运算，但你没有办法给Integer和Float这些已有的类增加方法。

所以，针对这些很别扭的情况，Julia和Lisp等函数式语言，就支持**多重分派**的方式。

你只需要定义几个相同名称的函数（在Julia里，这被叫做同一个函数的多个方法），编译器在运行时会根据参数，决定分派给哪个方法。

我们来看下面这个例子，foo函数有两个方法，根据调用参数的不同，分别分派给不同的方法。

```
julia> foo(x::Int64, y::Int64) = x + y  #第一个方法
foo (generic function with 1 method)

julia> foo(x, y) = x - y                #第二个方法
foo (generic function with 2 methods)

julia> methods(foo)                     #显示foo函数的所有方法
# 2 methods for generic function "foo":
[1] foo(x::Int64, y::Int64) in Main at REPL[38]:1
[2] foo(x, y) in Main at REPL[39]:1

julia> foo(2, 3)                        #分派到第一个方法
5

julia> foo(2.0, 3)                      #分派到第二个方法
-1.0
```

你可以发现，这种分派方法会**公平对待函数的所有参数，而不是由一个特殊的参数来决定。这种分派方法就叫做多重分派。**

在Julia中，其实“+”操作符（以及其他操作符）也是函数，它有上百个不同的方法，分别处理不同数据类型的加法操作。

```
julia> methods(+)
# 166 methods for generic function "+":
[1] +(x::Bool, z::Complex{Bool}) in Base at complex.jl:282
[2] +(x::Bool, y::Bool) in Base at bool.jl:96
[3] +(x::Bool) in Base at bool.jl:93
[4] +(x::Bool, y::T) where T<:AbstractFloat in Base at bool.jl:104
[5] +(x::Bool, z::Complex) in Base at complex.jl:289
[6] +(a::Float16, b::Float16) in Base at float.jl:398
[7] +(x::Float32, y::Float32) in Base at float.jl:400
[8] +(x::Float64, y::Float64) in Base at float.jl:401
[9] +(z::Complex{Bool}, x::Bool) in Base at complex.jl:283
...
[165] +(J::LinearAlgebra.UniformScaling, F::LinearAlgebra.Hessenberg) in LinearAlgebra
[166] +(a, b, c, xs...) in Base at operators.jl:529
```

最重要的是，当你引入新的数据类型，想要支持加法运算的时候，你只需要为加法函数定义一系列新的方法，那么编译器就可以正确地分派了。这种实现方式就方便多了。这也是某些函数式编程语言的一个优势，你可以体会一下。

而且在Julia中，因为方法分派是动态实现的，所以分派算法的性能就很重要。你看，不同的语言特性的设计，它的运行时就要完成不同的任务。这就是真实世界中，各种编译器的魅力所在。

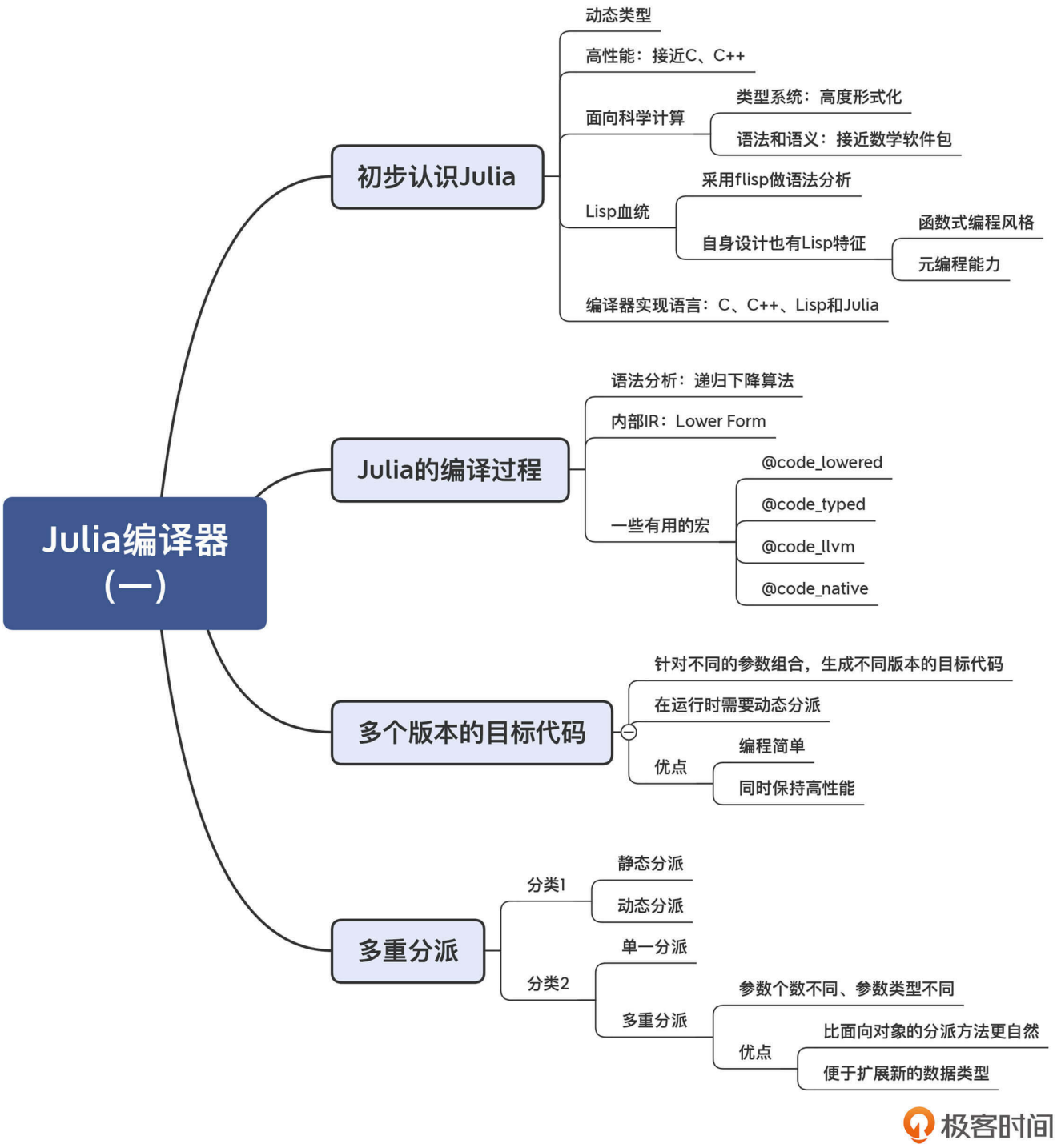
课程小结

这一讲我给你介绍了一门并不是那么大众的语言，Julia。介绍它的原因，就是因为这门语言有自己非常独特的特点，非常值得我们学习。我希望你能记住以下几点核心的知识：

- **编译器的实现语言**：编译器在选择采用什么实现的语言上，拥有很大的自由度。Julia很别具一格地采用了Lisp作为主要的前端语言。不过，我个人猜测，既然Julia本身也算是一种Lisp实现，未来可能就可以用Julia取代flisp，来实现前端的功能，实现更大程度的自举（Bootstraping）了。当然，这仅仅是我自己的猜测。
- **又是递归下降算法**：一如既往地，递归下降算法仍然是最常被用来实现语法分析的方法，Julia也不例外。
- **多种IR**：Julia在AST之外，采用了“goto”格式的IR，还用到了LLVM的IR（实际上，LLVM内部在转换成本地代码之前，还有一个中间格式的IR）。
- **多版本的目标代码**：Julia创造性地利用了LLVM的即时编译功能。它可以在运行期通过类型推断确定变量的类型，进行即时编译，根据不同的参数类型生成多个版本的目标代码，让程序员写一个程序就能适应多种数据类型，既降低了程序员的工作量，同时又保证了程序的高性能。这使得Julia同时拥有了动态类型语言的灵活性和静态类型语言的高性能。
- **多重分派功能**：多重分派能够根据方法参数的类型，确定其分派到哪个实现。它的优点是容易让同一个操作，扩展到支持不同的数据类型。

你学了这讲有什么体会呢？深入探究Julia这样的语言的实现过程，真的有助于我们大开脑洞，突破思维的限制，更好地融合编译原理的各方面的知识，从而为你的实际工作带来更加创新的思路。

这一讲的思维导图我也给你整理出来了，供你参考和复习回顾：



一课一思

一个很有意思的问题：为什么Julia会为一个函数，根据不同的参数类型组合，生成多个版本的目标代码，而JavaScript的引擎一般只会保存一个版本的目标代码？这个问题你可以从多个角度进行思考，欢迎在留言区分享你的观点。

感谢你的阅读，如果你觉得有收获，欢迎把今天的内容分享给更多的朋友。

上一页 下一页