Go-ladder

Simple, Poetic, Pithy

石鸟路遇

```
package main

import "fmt"

func main() {
	fmt.Println("hello, world")
	fmt.Println("你好, 世界")
}
```

前言

阅读本文需要有一定的golang基础，如果你是初学者建议先学习golang基础，这里有一个良好的教程

> 本文仅提供参考，资料均为个人学习和工作收集，仅供学习交流，不涉及商业用途。
>
> 如果你对相关内容感兴趣，请尊重创作者，遵守相关协议，非商业行为下传播。

String

源代码位于`src/builtin/builtin.go`，其中关于string的描述如下:

```
// string is the set of all strings of 8-bit bytes, conventionally but not
// necessarily representing UTF-8-encoded text. A string may be empty, but
// not nil. Values of string type are immutable.
type string string
```

所以string是8比特字节的集合，通常但并不一定是UTF-8编码的文本。

另外，还提到了两点，非常重要:

- string可以为空（长度为0），但不会是nil;
- string对象不可以修改。

string数据结构

源码包`src/runtime/string.go:stringStruct`定义了string的数据结构:

```
type stringStruct struct {
    str unsafe.Pointer  // str 地址
    len int  // 长度
}
```

string 声明

字符串构建过程是先跟据字符串构建stringStruct，再转换成string。转换的源码如下:

```
var str = "hello, world"

func gostringnocopy(str *byte) string { // 跟据字符串地址构建string
    ss := stringStruct{str: unsafe.Pointer(str), len: findnull(str)} // 先构造
stringStruct
    s := *(*string)(unsafe.Pointer(&ss))                             // 再将
stringStruct转换成string
    return s
}
```

[]byte转string

```
func GetStringBySlice(s []byte) string {
    return string(s)
}
```

string() 这种转换需要一次内存拷贝。

转换过程如下:

1. 跟据切片的长度申请内存空间，假设内存地址为p，切片长度为len(b);
2. 构建string（string.str = p；string.len = len；）
3. 拷贝数据(切片中数据拷贝到新申请的内存空间)

> []rune 同理

string转[]byte

```
func GetSliceByString(str string) []byte {
    return []byte(str)
}
```

string转换成byte切片，也需要一次内存拷贝，其过程如下：

- 申请切片内存空间
- 将string拷贝到切片

string和[]byte如何取舍

string和[]byte都可以表示字符串，但因数据结构不同，其衍生出来的方法也不同，要跟据实际应用场景来选择。

string 擅长的场景：

- 需要字符串比较的场景；
- 不需要nil字符串的场景；

[]byte擅长的场景：

- 修改字符串的场景，尤其是修改粒度为1个字节；
- 函数返回值，需要用nil表示含义的场景；
- 需要切片操作的场景；

虽然看起来string适用的场景不如[]byte多，但因为string直观，在实际应用中还是大量存在，在偏底层的实现中[]byte使用更多

字符串拼接性能

常见的字符串拼接方式有如下 5 种：

- 使用 `+`
- 使用 `fmt.Sprintf`
- 使用 `strings.Builder`

```
func builderConcat(n int, str string) string {
    var builder strings.Builder
    for i := 0; i < n; i++ {
        builder.WriteString(str)
    }
    return builder.String()
}
```

- 使用 `bytes.Buffer`

```
func bufferConcat(n int, s string) string {
    buf := new(bytes.Buffer)
    for i := 0; i < n; i++ {
        buf.WriteString(s)
    }
    return buf.String()
}
```

- 使用 `[]byte`

```
func byteConcat(n int, str string) string {
    buf := make([]byte, 0)
    for i := 0; i < n; i++ {
        buf = append(buf, str...)
    }
    return string(buf)
}
```

- prebyte

如果长度是可预知的，那么创建 `[]byte` 时，我们还可以预分配切片的容量(cap)。

```
func preByteConcat(n int, str string) string {
    buf := make([]byte, 0, n*len(str))
    for i := 0; i < n; i++ {
        buf = append(buf, str...)
    }
    return string(buf)
}
```

基准测试

```
$ go test -bench="Concat$" -benchmem .
goos: darwin
goarch: amd64
pkg: example
BenchmarkPlusConcat-8          19      56 ms/op   530 MB/op   10026 allocs/op
BenchmarkSprintfConcat-8       10     112 ms/op   835 MB/op   37435 allocs/op
BenchmarkBuilderConcat-8     8901    0.13 ms/op   0.5 MB/op      23 allocs/op
BenchmarkBufferConcat-8      8130    0.14 ms/op   0.4 MB/op      13 allocs/op
BenchmarkByteConcat-8        8984    0.12 ms/op   0.6 MB/op      24 allocs/op
BenchmarkPreByteConcat-8    17379    0.07 ms/op   0.2 MB/op       2 allocs/op
PASS
ok      example 8.627s
```

使用 `+` 和 `fmt.Sprintf` 的效率是最低的，和其余的方式相比，性能相差约 1000 倍，而且消耗了超过 1000 倍的内存。当然 `fmt.Sprintf` 通常是用来格式化字符串的，一般不会用来拼接字符串。

`strings.Builder`、`bytes.Buffer` 和 `[]byte` 的性能差距不大，而且消耗的内存也十分接近，性能最好且消耗内存最小的是 `preByteConcat`，这种方式预分配了内存，在字符串拼接的过程中，不需要进行字符串的拷贝，也不需要分配新的内存，因此性能最好，且内存消耗最小。

> 综合易用性和性能，一般推荐使用 `strings.Builder` 来拼接字符串。

比较 strings.Builder 和 bytes.Buffer

而 `strings.Builder`，`bytes.Buffer`，包括切片 `[]byte` 的内存是以倍数申请的。**参考slice扩容原理**

`strings.Builder` 和 `bytes.Buffer` 底层都是 `[]byte` 数组，但 `strings.Builder` 性能比 `bytes.Buffer` 略快约 10% 。一个比较重要的区别在于，`bytes.Buffer` 转化为字符串时重新申请了一块空间，存放生成的字符串变量，而 `strings.Builder` 直接将底层的 `[]byte` 转换成了字符串类型返回了回来

- bytes.Buffer

```
// To build strings more efficiently, see the strings.Builder type.
func (b *Buffer) String() string {
    if b == nil {
        // Special case, useful in debugging.
        return "<nil>"
    }
    return string(b.buf[b.off:])
}
```

- strings.Builder

```
// String returns the accumulated string.
func (b *Builder) String() string {
    return *(*string)(unsafe.Pointer(&b.buf))
}
```

数组

数组是由相同类型元素的集合组成的数据结构，计算机会为数组分配一块连续的内存来保存其中的元素，我们可以利用数组中元素的索引快速访问特定元素，

> 数组是单一内存块，并无附加结构

初始化

Go 语言的数组有两种不同的创建方式，一种是显式的指定数组大小，另一种是使用 `[...]T` 声明数组

```
arr1 := [3]int{1, 2, 3}
arr2 := [...]int{1, 2, 3}
```

> 语法糖的方式只能用于外层数组

使用第一种方式 `[n]T`，那么变量的类型在编译进行到类型检查阶段（可以查看编译原理）就会被提取出来，该类型包含两个字段，分别是元素类型 `Elem` 和数组的大小 `Bound`，这两个字段共同构成了数组类型，而当前数组是否应该在堆栈中初始化也在编译期就确定了。

随后使用 `cmd/compile/internal/types.NewArray` 创建包含数组大小的 `cmd/compile/internal/types.Array` 结构体。

```
func NewArray(elem *Type, bound int64) *Type {
    if bound < 0 {
        Fatalf("NewArray: invalid bound %v", bound)
    }
    t := New(TARRAY)
    t.Extra = &Array{Elem: elem, Bound: bound}
    t.SetNotInHeap(elem.NotInHeap())
    return t
}
```

当我们使用 `[...]T` 的方式声明数组时，编译器会在的 `cmd/compile/internal/gc.typecheckcomplit` 函数中对该数组的大小进行推导

```
func typecheckcomplit(n *Node) (res *Node) {
    ...
    if n.Right.Op == OTARRAY && n.Right.Left != nil && n.Right.Left.Op == ODDD {
        n.Right.Right = typecheck(n.Right.Right, ctxType)
        if n.Right.Right.Type == nil {
            n.Type = nil
            return n
        }
        elemType := n.Right.Right.Type

        length := typecheckarraylit(elemType, -1, n.List.Slice(), "array
literal")

        n.Op = OARRAYLIT
        n.Type = types.NewArray(elemType, length)
        n.Right = nil
        return n
    }
```

```
    ...

    switch t.Etype {
    case TARRAY:
        typecheckarraylit(t.Elem(), t.NumElem(), n.List.Slice(), "array
literal")
        n.Op = OARRAYLIT
        n.Right = nil
    }
}
```

存放位置

对于一个由字面量组成的数组，根据数组元素数量的不同，编译器会在负责初始化字面量的 `cmd/compile/internal/gc.anylit` 函数中做两种不同的优化：

- 当元素数量小于或者等于 4 个时，会直接将数组中的元素放置在栈上；
- 当元素数量大于 4 个时，会将数组中的元素放置到静态区并在运行时取出；

```
func anylit(n *Node, var_ *Node, init *Nodes) {
    t := n.Type
    switch n.Op {
    case OSTRUCTLIT, OARRAYLIT:
        if n.List.Len() > 4 {
            ...
        }

        fixedlit(inInitFunction, initKindLocalCode, n, var_, init)
    ...
    }
}
```

总结起来，在不考虑逃逸分析的情况下，如果数组中元素的个数小于或者等于 4 个，那么所有的变量会直接在栈上初始化，如果数组元素大于 4 个，变量就会在静态存储区初始化然后拷贝到栈上，这些转换后的代码才会继续进入 `中间代码生成` 和 `机器码生成` 两个阶段，最后生成可以执行的二进制文件。

访问和赋值

无论是在栈上还是静态存储区，数组在内存中都是一连串的内存空间，我们通过指向数组开头的指针、元素的数量以及元素类型占的空间大小表示数组。

数组访问越界是非常严重的错误，Go 语言中可以在编译期间的静态类型检查判断数组越界，`cmd/compile/internal/gc.typecheck1` 会验证访问数组的索引：

```
func typecheck1(n *Node, top int) (res *Node) {
    switch n.Op {
    case OINDEX:
        ok |= ctxExpr
        l := n.Left  // array
        r := n.Right // index
        switch n.Left.Type.Etype {
        case TSTRING, TARRAY, TSLICE:
            ...
            if n.Right.Type != nil && !n.Right.Type.IsInteger() {
                yyerror("non-integer array index %v", n.Right)
                break
```

```
            }
            if !n.Bounded() && Isconst(n.Right, CTINT) {
                x := n.Right.Int64()
                if x < 0 {
                    yyerror("invalid array index %v (index must be non-
negative)", n.Right)
                } else if n.Left.Type.IsArray() && x >= n.Left.Type.NumElem() {
                    yyerror("invalid array index %v (out of bounds for %d-
element array)", n.Right, n.Left.Type.NumElem())
                }
            }
        }
    ...
    }
}
```

数组的赋值和更新操作 `a[i] = 2` 也会计算出数组当前元素的内存地址，然后修改当前内存地址的内容，这些赋值语句会被转换成如下所示的 SSA 代码：

```
b1:
    ...
    v21 (5) = LocalAddr <*[3]int> {arr} v2 v19
    v22 (5) = PtrIndex <*int> v21 v13
    v23 (5) = Store <mem> {int} v22 v20 v19
    ...
```

Go赋值的过程中会先确定目标数组的地址，再通过 `PtrIndex` 获取目标元素的地址，最后使用 `Store` 指令将数据存入地址中，上述**数组寻址和赋值都是在编译阶段完成的，没有运行时的参与。**

> 可以导出ssa代码查看
>
> dumped SSA to ./ssa.html

切片

Slice又称动态数组，依托数组实现，可以方便的进行扩容、传递等，实际使用中比数组更灵活。

slice数据结构

源码包中`src/runtime/slice.go:slice`定义了Slice的数据结构：

```
type slice struct {
    array unsafe.Pointer
    len   int
    cap   int
}
```

从数据结构看Slice很清晰, array指针指向底层数组，len表示切片长度，cap表示底层数组容量。

编译期间的切片是`cmd/compile/internal/types.Slice`类型的，但是在运行时切片可以由如下的`reflect.SliceHeader`结构体表示，其中:

- `Data`是指向数组的指针;
- `Len`是当前切片的长度;
- `Cap`是当前切片的容量，即`Data`数组的大小:

```
type SliceHeader struct {
    Data uintptr
    Len  int
    Cap  int
}
```

`Data`是一片连续的内存空间，这片内存空间可以用于存储切片中的全部元素，数组中的元素只是逻辑上的概念，底层存储其实都是连续的，所以我们可以将切片理解成一片连续的内存空间加上长度与容量的标识。

初始化

Go 语言中包含三种初始化切片的方式：

1. 通过下标的方式获得数组或者切片的一部分；
2. 使用字面量初始化新的切片；
3. 使用关键字`make`创建切片：

```
arr[0:3] or slice[0:3]
slice := []int{1, 2, 3}
slice := make([]int, 10)
```

使用下标

使用下标创建切片是最原始也最接近汇编语言的方式，它是所有方法中最为底层的一种，编译器会将`arr[0:3]`或者`slice[0:3]`等语句转换成`OpSliceMake`操作

```
func newSlice() []int {
    arr := [3]int{1, 2, 3}
    slice := arr[0:1]
    return slice
}
```

通过 `GOSSAFUNC` 变量编译上述代码可以得到一系列 SSA 中间代码，其中 `slice := arr[0:1]` 语句在 “decompose builtin” 阶段对应的代码如下所示：

```
v27 (+5) = SliceMake <[]int> v11 v14 v17

name &arr[*[3]int]: v11
name slice.ptr[*int]: v11
name slice.len[int]: v14
name slice.cap[int]: v17
```

`SliceMake` 操作会接受四个参数创建新的切片，元素类型、数组指针、切片大小和容量，这也是我们在数据结构一节中提到的切片的几个字段，需要注意的是使用下标初始化切片不会拷贝原数组或者原切片中的数据，它只会创建一个指向原数组的切片结构体，所以修改新切片的数据也会修改原切片。

字面量（块儿）

当我们使用字面量 `[]int{1, 2, 3}` 创建新的切片时，`cmd/compile/internal/gc.slicelit` 函数会在编译期间将它展开成如下所示的代码片段：

```
var vstat [3]int
vstat[0] = 1
vstat[1] = 2
vstat[2] = 3
var vauto *[3]int = new([3]int)
*vauto = vstat
slice := vauto[:]
```

1. 根据切片中的元素数量对底层数组的大小进行推断并创建一个数组；
2. 将这些字面量元素存储到初始化的数组中；
3. 创建一个同样指向 `[3]int` 类型的数组指针；
4. 将静态存储区的数组 `vstat` 赋值给 `vauto` 指针所在的地址；
5. 通过 `[:]` 操作获取一个底层使用 `vauto` 的切片；

第 5 步中的 `[:]` 就是使用下标创建切片的方法，从这一点我们也能看出 `[:]` 操作是创建切片最底层的一种方法。

make

如果使用字面量的方式创建切片，大部分的工作都会在编译期间完成。但是当我们使用 `make` 关键字创建切片时，很多工作都需要运行时的参与；调用方必须向 `make` 函数传入切片的大小以及可选的容量，类型检查期间的 `cmd/compile/internal/gc.typecheck1` 函数会校验入参：

```
func typecheck1(n *Node, top int) (res *Node) {
    switch n.Op {
    ...
    case OMAKE:
        args := n.List.Slice()
```

```
        i := 1
        switch t.Etype {
        case TSLICE:
            if i >= len(args) {
                yyerror("missing len argument to make(%v)", t)
                return n
            }

            l = args[i]
            i++
            var r *Node
            if i < len(args) {
                r = args[i]
            }
            ...
            if Isconst(l, CTINT) && r != nil && Isconst(r, CTINT) && l.Val().U.
(*Mpint).Cmp(r.Val().U.(*Mpint)) > 0 {
                yyerror("len larger than cap in make(%v)", t)
                return n
            }

            n.Left = l
            n.Right = r
            n.Op = OMAKESLICE
        }
    ...
    }
}
```

如果当前的切片不会发生逃逸并且切片非常小的时候，`make([]int, 3, 4)` 会被直接转换成如下所示的代码：

```
var arr [4]int
n := arr[:3]
```

当切片发生逃逸或者非常大时，运行时需要 `runtime.makeslice` 在堆上初始化切片

```
func makeslice(et *_type, len, cap int) unsafe.Pointer {
    mem, overflow := math.MulUintptr(et.size, uintptr(cap))
    if overflow || mem > maxAlloc || len < 0 || len > cap {
        mem, overflow := math.MulUintptr(et.size, uintptr(len))
        if overflow || mem > maxAlloc || len < 0 {
            panicmakeslicelen()
        }
        panicmakeslicecap()
    }

    return mallocgc(mem, et, true)
}
```

访问元素

使用 `len` 和 `cap` 获取长度或者容量是切片最常见的操作，编译器将这它们看成两种特殊操作，即 `OLEN` 和 `OCAP`，`cmd/compile/internal/gc.state.expr` 阶段将它们分别转换成 `OpSliceLen` 和 `OpSliceCap`：

```
func (s *state) expr(n *Node) *ssa.Value {
    switch n.Op {
    case OLEN, OCAP:
        switch {
        case n.Left.Type.IsSlice():
            op := ssa.OpSliceLen
            if n.Op == OCAP {
                op = ssa.OpSliceCap
            }
            return s.newValue1(op, types.Types[TINT], s.expr(n.Left))
        ...
        }
    ...
    }
}
```

访问切片中的字段可能会触发 “decompose builtin” 阶段的优化，`len(slice)` 或者 `cap(slice)` 在一些情况下会直接替换成切片的长度或者容量，不需要在运行时获取：

```
(SlicePtr (SliceMake ptr _ _ )) -> ptr
(SliceLen (SliceMake _ len _)) -> len
(SliceCap (SliceMake _ _ cap)) -> cap
```

除了获取切片的长度和容量之外，访问切片中元素使用的 `OINDEX` 操作也会在中间代码生成期间转换成对地址的直接访问：

```
func (s *state) expr(n *Node) *ssa.Value {
    switch n.Op {
    case OINDEX:
        switch {
        case n.Left.Type.IsSlice():
            p := s.addr(n, false)
            return s.load(n.Left.Type.Elem(), p)
        ...
        }
    ...
    }
}
```

切片的操作基本都是在编译期间完成的，除了访问切片的长度、容量或者其中的元素之外，编译期间也会将包含 `range` 关键字的遍历转换成形式更简单的循环，我们会在后面的章节中介绍使用 `range` 遍历切片的过程。

追加和扩容

使用 `append` 关键字向切片中追加元素也是常见的切片操作，中间代码生成阶段的 `cmd/compile/internal/gc.state.append` 方法会根据返回值是否会覆盖原变量，选择进入两种流程:

如果 `append` 返回的新切片不需要赋值回原有的变量，就会进入如下的处理流程：

```
// append(slice, 1, 2, 3)
ptr, len, cap := slice
newlen := len + 3
if newlen > cap {
    ptr, len, cap = growslice(slice, newlen)
    newlen = len + 3
}
*(ptr+len) = 1
*(ptr+len+1) = 2
*(ptr+len+2) = 3
return makeslice(ptr, newlen, cap)
```

如果使用 `slice = append(slice, 1, 2, 3)` 语句，那么 `append` 后的切片会覆盖原切片，这时 `cmd/compile/internal/gc.state.append` 方法会使用另一种方式展开关键字：

```
// slice = append(slice, 1, 2, 3)
a := &slice
ptr, len, cap := slice
newlen := len + 3
if uint(newlen) > uint(cap) {
   newptr, len, newcap = growslice(slice, newlen)
   vardef(a)
   *a.cap = newcap
   *a.ptr = newptr
}
newlen = len + 3
*a.len = newlen
*(ptr+len) = 1
*(ptr+len+1) = 2
*(ptr+len+2) = 3
```

当切片的容量不足时，我们会调用 `runtime.growslice` 函数为切片扩容，扩容是为切片分配新的内存空间并拷贝原切片中元素的过程，我们先来看新切片的容量是如何确定的：

```
func growslice(et *_type, old slice, cap int) slice {
    newcap := old.cap
    doublecap := newcap + newcap
    if cap > doublecap {
        newcap = cap
    } else {
        if old.len < 1024 {
            newcap = doublecap
        } else {
            for 0 < newcap && newcap < cap {
                newcap += newcap / 4
            }
            if newcap <= 0 {
                newcap = cap
            }
        }
    }
```

在分配内存空间之前需要先确定新的切片容量，运行时根据切片的当前容量选择不同的策略进行扩容：

1. 如果期望容量大于当前容量的两倍就会使用期望容量；

2. 如果当前切片的长度小于 1024 就会将容量翻倍；
3. 如果当前切片的长度大于 1024 就会每次增加 25% 的容量，直到新容量大于期望容量；

上述代码片段仅会确定切片的大致容量，下面还需要根据切片中的元素大小对齐内存，当数组中元素所占的字节大小为 1、8 或者 2 的倍数时，

简单总结一下扩容的过程，当我们执行上述代码时，会触发 `runtime.growslice` 函数扩容 `arr` 切片并传入期望的新容量 5，这时期望分配的内存大小为 40 字节；不过因为切片中的元素大小等于 `sys.PtrSize`，所以运行时会调用 `runtime.roundupsize` 向上取整内存的大小到 48 字节，所以新切片的容量为 48 / 8 = 6。

拷贝

切片的拷贝虽然不是常见的操作，但是却是我们学习切片实现原理必须要涉及的。当我们使用 `copy(a, b)` 的形式对切片进行拷贝时，编译期间的 `cmd/compile/internal/gc.copyany` 也会分两种情况进行处理拷贝操作，如果当前 `copy` 不是在运行时调用的，`copy(a, b)` 会被直接转换成下面的代码：

```
n := len(a)
if n > len(b) {
    n = len(b)
}
if a.ptr != b.ptr {
    memmove(a.ptr, b.ptr, n*sizeof(elem(a)))
}
```

上述代码中的 `runtime.memmove` 会负责拷贝内存。而如果拷贝是在运行时发生的，例如：`go copy(a, b)`，编译器会使用 `runtime.slicecopy` 替换运行期间调用的 `copy`，该函数的实现很简单：

```
func slicecopy(to, fm slice, width uintptr) int {
    if fm.len == 0 || to.len == 0 {
        return 0
    }
    n := fm.len
    if to.len < n {
        n = to.len
    }
    if width == 0 {
        return n
    }
    ...

    size := uintptr(n) * width
    if size == 1 {
        *(*byte)(to.array) = *(*byte)(fm.array)
    } else {
        memmove(to.array, fm.array, size)
    }
    return n
}
```

> copy 的长度为最短切片长度，不是容量，如dst的长度为0，容量大于src的长度，copy长度依然为0

map

Golang的map使用哈希表作为底层实现，一个哈希表里可以有多个哈希表节点，也即bucket，而每个bucket就保存了map中的一个或一组键值对。

设计原理

哈希表是计算机科学中的最重要数据结构之一，这不仅因为它 O(1)O(1) 的读写性能非常优秀，还因为它提供了键值之间的映射。想要实现一个性能优异的哈希表，需要注意两个关键点 —— 哈希函数和冲突解决方法。

hash 函数

在理想情况下，哈希函数应该能够将不同键映射到不同的索引上，这要求**哈希函数的输出范围大于输入范围**（这往往不现实），但是由于键的数量会远远大于映射的范围，所以在实际使用时，这个理想的效果是不可能实现的。

如果使用结果分布较为均匀的哈希函数，那么哈希的增删改查的时间复杂度为 O(1)；但是如果哈希函数的结果分布不均匀，那么所有操作的时间复杂度可能会达到 O(n)，由此看来，使用好的哈希函数是至关重要的。

解决冲突

在通常情况下，哈希函数输入的范围一定会远远大于输出的范围，所以在使用哈希表时一定会遇到冲突，哪怕我们使用了完美的哈希函数，当输入的键足够多也会产生冲突。然而多数的哈希函数都是不够完美的，所以仍然存在发生哈希碰撞的可能，这时就需要一些方法来解决哈希碰撞的问题，常见方法的就是开放寻址法和拉链法。

开放寻址法:

这种方法的核心思想是**依次探测和比较数组中的元素以判断目标键值对是否存在于哈希表中**，如果我们使用开放寻址法来实现哈希表，那么实现哈希表底层的数据结构就是数组，不过因为数组的长度有限，向哈希表写入 (author, draven) 这个键值对时会从如下的索引开始遍历：

```
index := hash("author") % array.len
```

当我们向当前哈希表写入新的数据时，如果发生了冲突，就会将键值对写入到下一个索引不为空的位置

开放寻址法中对性能影响最大的是**装载因子**，它是数组中元素的数量与数组大小的比值。随着装载因子的增加，线性探测的平均用时就会逐渐增加，这会影响哈希表的读写性能。当装载率超过 70% 之后，哈希表的性能就会急剧下降，而一旦装载率达到 100%，整个哈希表就会完全失效，这时查找和插入任意元素的时间复杂度都是 O(n)

拉链法:

拉链法是哈希表最常见的实现方法，大多数的编程语言都用拉链法实现哈希表，它的实现比较开放地址法稍微复杂一些，但是平均查找的长度也比较短，**各个用于存储节点的内存都是动态申请的，可以节省比较多的存储空间。**

实现拉链法一般会使用数组加上链表，不过一些编程语言会在拉链法的哈希中引入红黑树以优化性能，拉链法会使用链表数组作为哈希底层的数据结构，我们可以将它看成可以扩展的二维数组：

当我们需要将一个键值对 (Key6, Value6) 写入哈希表时，键值对中的键 Key6 都会先经过一个哈希函数，哈希函数返回的哈希会帮助我们选择一个桶，和开放地址法一样，选择桶的方式是直接对哈希返回的结果取模：

```
index := hash("Key6") % array.len
```

选择了桶后就可以遍历当前桶中的链表了，在遍历链表的过程中会遇到以下两种情况：

1. 找到键相同的键值对 — 更新键对应的值；
2. 没有找到键相同的键值对 — 在链表的末尾追加新的键值对；

数据结构

map数据结构由`runtime/map.go/hmap`定义：

```
type hmap struct {
    count     int
    flags     uint8
    B         uint8
    noverflow uint16
    hash0     uint32

    buckets    unsafe.Pointer
    oldbuckets unsafe.Pointer
    nevacuate  uintptr

    extra *mapextra
}

type mapextra struct {
    overflow    *[]*bmap
    oldoverflow *[]*bmap
    nextOverflow *bmap
}
```

1. `count` 表示当前哈希表中的元素数量；
2. `B` 表示当前哈希表持有的 `buckets` 数量，但是因为哈希表中桶的数量都 2 的倍数，所以该字段会存储对数，也就是 `len(buckets) == 2^B`；
3. `hash0` 是哈希的种子，它能为哈希函数的结果引入随机性，这个值在创建哈希表时确定，并在调用哈希函数时作为参数传入；
4. `oldbuckets` 是哈希在扩容时用于保存之前 `buckets` 的字段，它的大小是当前 `buckets` 的一半；

每一个 `runtime.bmap` 都能存储 8 个键值对，当哈希表中存储的数据过多，单个桶已经装满时就会使用 `extra.nextOverflow` 中桶存储溢出的数据。

bucket数据结构

bucket数据结构由`runtime/map.go/bmap`定义：

```
type bmap struct {
    tophash [8]uint8 //存储哈希值的高8位
    data    byte[1]  //key value数据:key/key/key/.../value/value/value...
    overflow *bmap   //溢出bucket的地址
}
```

每个bucket可以存储8个键值对。所以同一个bucket存放超过8个键值对时就会再创建一个键值对，用类似链表的方式将bucket连接起来。

初始化

字面量

```
hash := map[string]int{
    "1": 2,
    "3": 4,
    "5": 6,
}
```

使用字面量初始化的方式最终都会通过 `cmd/compile/internal/gc.maplit` 初始化，我们来分析一下该函数初始化哈希的过程：

```
func maplit(n *Node, m *Node, init *Nodes) {
    a := nod(OMAKE, nil, nil)
    a.Esc = n.Esc
    a.List.Set2(typenod(n.Type), nodintconst(int64(n.List.Len())))
    litas(m, a, init)

    entries := n.List.Slice()
    if len(entries) > 25 {
        ...
        return
    }

    // Build list of var[c] = expr.
    // Use temporaries so that mapassign1 can have addressable key, elem.
    ...
}
```

当哈希表中的元素数量少于或者等于 **25 个**时，编译器会将字面量初始化的结构体转换成以下的代码，将所有的键值对一次加入到哈希表中

一旦哈希表中元素的数量超过了 **25 个**，编译器会创建两个数组分别存储键和值，这些键值对会通过如下所示的 for 循环加入哈希：

make

```
hash := make(map[string]int, 26)
```

只要我们使用 `make` 创建哈希，使用字面量初始化哈希也只是语言提供的辅助工具，最后调用的都是 `runtime.makemap`：

```
func makemap(t *maptype, hint int, h *hmap) *hmap {
    mem, overflow := math.MulUintptr(uintptr(hint), t.bucket.size)
    if overflow || mem > maxAlloc {
        hint = 0
    }

    if h == nil {
        h = new(hmap)
    }
    h.hash0 = fastrand()
```

```
    B := uint8(0)
    for overLoadFactor(hint, B) {
        B++
    }
    h.B = B

    if h.B != 0 {
        var nextOverflow *bmap
        h.buckets, nextOverflow = makeBucketArray(t, h.B, nil)
        if nextOverflow != nil {
            h.extra = new(mapextra)
            h.extra.nextOverflow = nextOverflow
        }
    }
    return h
}
```

这个函数会按照下面的步骤执行：

1. 计算哈希占用的内存是否溢出或者超出能分配的最大值；
2. 调用 `runtime.fastrand` 获取一个随机的哈希种子；
3. 根据传入的 `hint` 计算出需要的最小需要的桶的数量；
4. 使用 `runtime.makeBucketArray` 创建用于保存桶的数组；

负载因子

负载因子用于衡量一个哈希表冲突情况，公式为：

```
负载因子 = 键数量/bucket数量
```

例如，对于一个bucket数量为4，包含4个键值对的哈希表来说，这个哈希表的负载因子为1.

哈希表需要将负载因子控制在合适的大小，超过其阀值需要进行rehash，也即键值对重新组织：

- 哈希因子过小，说明空间利用率低
- 哈希因子过大，说明冲突严重，存取效率低

每个哈希表的实现对负载因子容忍程度不同，比如Redis实现中负载因子大于1时就会触发rehash，而Go则在在负载因子达到6.5时才会触发rehash，因为Redis的每个bucket只能存1个键值对，而Go的bucket可能存8个键值对，所以Go可以容忍更高的负载因子。

扩容

为了保证访问效率，当新元素将要添加进map时，都会检查是否需要扩容，扩容实际上是以空间换时间的手段。 触发扩容的条件有二个：

1. 负载因子 > 6.5时，也即平均每个bucket存储的键值对达到6.5个。
2. overflow数量 > 2^15时，也即overflow数量超过32768时。

增量扩容

当负载因子过大时，就新建一个bucket，新的bucket长度是原来的2倍，然后旧bucket数据搬迁到新的bucket。 考虑到如果map存储了数以亿计的key-value，一次性搬迁将会造成比较大的延时，**Go采用逐步搬迁策略**，即每次访问map时都会触发一次搬迁，每次搬迁2个键值对。

hmap数据结构中oldbuckets成员指身原bucket，而buckets指向了新申请的bucket。新的键值对被插入新的bucket中。 后续对map的访问操作会触发迁移，将oldbuckets中的键值对逐步的搬迁过来。当oldbuckets中的键值对全部搬迁完毕后，删除oldbuckets。

等量扩容

所谓等量扩容，实际上并不是扩大容量，buckets数量不变，重新做一遍类似增量扩容的搬迁动作，把松散的键值对重新排列一次，以使bucket的使用率更高，进而保证更快的存取。

经过重新组织后overflow的bucket数量会减少，即节省了空间又会提高访问效率。

查找过程

查找过程如下：

1. 跟据key值算出哈希值
2. 取哈希值低位与hmpa.B取模确定bucket位置
3. 取哈希值高位在tophash数组中查询
4. 如果tophash[i]中存储值也哈希值相等，则去找到该bucket中的key值进行比较
5. 当前bucket没有找到，则继续从下个overflow的bucket中查找。
6. 如果当前处于搬迁过程，则优先从oldbuckets查找

注：如果查找不到，也不会返回空值，而是返回相应类型的0值。

插入过程

新员素插入过程如下：

1. 跟据key值算出哈希值
2. 取哈希值低位与hmap.B取模确定bucket位置
3. 查找该key是否已经存在，如果存在则直接更新值
4. 如果没找到将key，将key插入

接口

两种接口

Go 语言根据接口类型是否包含一组方法将接口类型分成了两类：

- 使用 `runtime.iface` 结构体表示包含方法的接口
- 使用 `runtime.eface` 结构体表示不包含任何方法的 `interface{}` 类型；

`runtime.eface` 结构体在 Go 语言中的定义是这样的：

```
type eface struct { // 16 字节
    _type *_type
    data  unsafe.Pointer
}
```

由于 `interface{}` 类型不包含任何方法，所以它的结构也相对来说比较简单，只包含指向底层数据和类型的两个指针。从上述结构我们也能推断出 — Go 语言的任意类型都可以转换成 `interface{}`。

另一个用于表示接口的结构体是 `runtime.iface`，这个结构体中有指向原始数据的指针 `data`，不过更重要的是 `runtime.itab` 类型的 `tab` 字段。

```
type iface struct { // 16 字节
    tab  *itab
    data unsafe.Pointer
}
```

接下来我们将详细分析 Go 语言接口中的这两个类型，即 `runtime._type` 和 `runtime.itab`。

下面是运行时包中的结构体，其中包含了很多类型的元信息，例如：类型的大小、哈希、对齐以及种类等。

_type 结构体

```
type _type struct {
    size       uintptr
    ptrdata    uintptr
    hash       uint32
    tflag      tflag
    align      uint8
    fieldAlign uint8
    kind       uint8
    equal      func(unsafe.Pointer, unsafe.Pointer) bool
    gcdata     *byte
    str        nameOff
    ptrToThis  typeOff
}
```

- `size` 字段存储了类型占用的内存空间，为内存空间的分配提供信息；
- `hash` 字段能够帮助我们快速确定类型是否相等；
- `equal` 字段用于判断当前类型的多个对象是否相等，该字段是为了减少 Go 语言二进制包大小从 `typeAlg` 结构体中迁移过来的；

itab 结构体

```
type itab struct { // 32 字节
    inter *interfacetype
    _type *_type
    hash  uint32
    _     [4]byte
    fun   [1]uintptr
}
```

类型转换

指针类型

```
package main

type Duck interface {
    Quack()
}

type Cat struct {
    Name string
}

//go:noinline
func (c *Cat) Quack() {
    println(c.Name + " meow")
}

func main() {
    var c Duck = &Cat{Name: "draven"}
    c.Quack()
}
```

我们先来分析结构体 `Cat` 的初始化过程：

```
LEAQ    type.""."".Cat(SB), AX              ;; AX = &type.""."".Cat
MOVQ    AX, (SP)                            ;; SP = &type.""."".Cat
CALL    runtime.newobject(SB)               ;; SP + 8 = &Cat{}
MOVQ    8(SP), DI                           ;; DI = &Cat{}
MOVQ    $6, 8(DI)                           ;; StringHeader(DI.Name).Len = 6
LEAQ    go.string."draven"(SB), AX          ;; AX = &"draven"
MOVQ    AX, (DI)                            ;; StringHeader(DI.Name).Data =
&"draven"
```

1. 获取 `Cat` 结构体类型指针并将其作为参数放到栈上；
2. 通过 `CALL` 指定调用 `runtime.newobject` 函数，这个函数会以 `Cat` 结构体类型指针作为入参，分配一片新的内存空间并将指向这片内存空间的指针返回到 SP+8 上；
3. SP+8 现在存储了一个指向 `Cat` 结构体的指针，我们将栈上的指针拷贝到寄存器 `DI` 上方便操作；
4. 由于 `Cat` 中只包含一个字符串类型的 `Name` 变量，所以在这里会分别将字符串地址 `&"draven"` 和字符串长度 6 设置到结构体上，最后三行汇编指令等价于 `cat.Name = "draven"`；

结构体类型

```
package main

type Duck interface {
    Quack()
}

type Cat struct {
    Name string
}

//go:noinline
func (c Cat) Quack() {
    println(c.Name + " meow")
}

func main() {
    var c Duck = Cat{Name: "draven"}
    c.Quack()
}
```

类型断言

非空接口

```
func main() {
    var c Duck = &Cat{Name: "draven"}
    switch c.(type) {
    case *Cat:
        cat := c.(*Cat)
        cat.Quack()
    }
}
```

我们将编译得到的汇编指令分成两部分分析，第一部分是变量的初始化，第二部分是类型断言，第一部分的代码如下：

```
00000 TEXT  "".main(SB), ABIInternal, $32-0
...
00029 XORPS X0, X0
00032 MOVUPS    X0, ""..autotmp_4+8(SP)
00037 LEAQ  go.string."draven"(SB), AX
00044 MOVQ  AX, ""..autotmp_4+8(SP)
00049 MOVQ  $6, ""..autotmp_4+16(SP)
```

0037 ~ 0049 三个指令初始化了 `Duck` 变量，`Cat` 结构体初始化在 SP+8 ~ SP+24 上。因为 Go 语言的编译器做了一些优化，所以代码中没有 `runtime.iface` 的构建过程，不过对于这一节要介绍的类型断言和转换没有太多的影响。下面进入类型转换的部分：

```
00058 CMPL  go.itab.*"".Cat,"".Duck+16(SB), $593696792
                                        ;; if (c.tab.hash != 593696792) {
00068 JEQ   80                          ;;
00070 MOVQ  24(SP), BP                  ;;      BP = SP+24
```

```
00075 ADDQ  $32, SP                          ;;      SP += 32
00079 RET                                    ;;      return
                                             ;; } else {
00080 LEAQ  ""..autotmp_4+8(SP), AX          ;;      AX = &Cat{Name: "draven"}
00085 MOVQ  AX, (SP)                         ;;      SP = AX
00089 CALL  "".(*Cat).Quack(SB)              ;;      SP.Quack()
00094 JMP   70                               ;;      ...
                                             ;;      BP = SP+24
                                             ;;      SP += 32
                                             ;;      return
                                             ;; }
```

switch语句生成的汇编指令会将目标类型的 `hash` 与接口变量中的 `itab.hash` 进行比较：

1. 获取 SP+8 存储的 `Cat` 结构体指针；
2. 将结构体指针拷贝到栈顶；
3. 调用 `Quack` 方法；
4. 恢复函数的栈并返回；

空接口

当我们使用空接口类型 `interface{}` 进行类型断言时，如果不关闭 Go 语言编译器的优化选项，生成的汇编指令是差不多的。编译器会省略将 `Cat` 结构体转换成 `runtime.eface` 的过程：

```
func main() {
    var c interface{} = &Cat{Name: "draven"}
    switch c.(type) {
    case *Cat:
        cat := c.(*Cat)
        cat.Quack()
    }
}
```

结构体

Go的struct声明允许字段附带`Tag`来对字段做一些标记。

tag 规则

`Tag`本身是一个字符串，但字符串中却是：`以空格分隔的 key:value 对`。

- `key`: 必须是非空字符串，字符串不能包含控制字符、空格、引号、冒号。
- `value`: 以双引号标记的字符串
- 注意：冒号前后不能有空格

如下代码所示，如此写没有实际意义，仅用于说明`Tag`规则

```
type Server struct {
    ServerName string `key1: "value1" key11:"value11"`
    ServerIP   string `key2: "value2"`
}
```

tag是Struct的一部分

```
// A StructField describes a single field in a struct.
type StructField struct {
    // Name is the field name.
    Name string
    ...
    Type      Type      // field type
    Tag       StructTag // field tag string
    ...
}
```

tag常见用法

常见的tag用法，主要是JSON、Bson数据解析、ORM映射、protobuf、http参数解析（header、query）等。

for 和 Range

range是Golang提供的一种迭代遍历手段，可操作的类型有数组、切片、Map、channel等，实际使用频率非常高。

range for slice

```
// The loop we generate:
//   for_temp := range
//   len_temp := len(for_temp)
//   for index_temp = 0; index_temp < len_temp; index_temp++ {
//           value_temp = for_temp[index_temp]
//           index = index_temp
//           value = value_temp
//           original body
//   }
```

遍历slice前会先获以slice的长度len_temp作为循环次数，循环体中，每次循环会先获取元素值，如果for-range中接收index和value的话，则会对index和value进行一次赋值。

由于循环开始前循环次数就已经确定了，所以循环过程中新添加的元素是没办法遍历到的。

另外，数组与数组指针的遍历过程与slice基本一致，不再赘述。

range for map

```
// The loop we generate:
//   var hiter map_iteration_struct
//   for mapiterinit(type, range, &hiter); hiter.key != nil; mapiternext(&hiter)
{
//           index_temp = *hiter.key
//           value_temp = *hiter.val
//           index = index_temp
//           value = value_temp
//           original body
//   }
```

遍历map时没有指定循环次数，循环体与遍历slice类似。由于map底层实现与slice不同，map底层使用hash表实现，插入数据位置是随机的，所以遍历过程中新插入的数据不能保证遍历到。

range for channel

```
// The loop we generate:
//   for {
//           index_temp, ok_temp = <-range
//           if !ok_temp {
//                   break
//           }
//           index = index_temp
//           original body
//   }
```

channel遍历是依次从channel中读取数据,读取前是不知道里面有多少个元素的。如果channel中没有元素，则会阻塞等待，如果channel已被关闭，则会解除阻塞并退出循环。

注：

- 上述注释中index_temp实际上描述是有误的，应该为value_temp，因为index对于channel是没有意义的。
- 使用for-range遍历channel时只能获取一个返回值。

range for string

遍历字符串的过程与数组、切片和哈希表非常相似，只是在遍历时会获取字符串中索引对应的字节并将字节转换成 `rune`。我们在遍历字符串时拿到的值都是 `rune` 类型的变量，`for i, r := range s {}`

defer

defer数据结构

源码包`src/src/runtime/runtime2.go:_defer`定义了defer的数据结构：

```
type _defer struct {
    sp      uintptr   //函数栈指针
    pc      uintptr   //程序计数器
    fn      *funcval  //函数地址
    link    *_defer   //指向自身结构的指针，用于链接多个defer
}
```

我们知道defer后面一定要接一个函数的，所以defer的数据结构跟一般函数类似，也有栈地址、程序计数器、函数地址等等。

defer的创建和执行

源码包`src/runtime/panic.go`定义了两个方法分别用于创建defer和执行defer。

- deferproc()： 在声明defer处调用，其将defer函数存入goroutine的链表中；
- deferreturn()：在return指令，准确的讲是在ret指令前调用，其将defer从goroutine链表中取出并执行。

可以简单这么理解，在编译在阶段，声明defer处插入了函数deferproc()，在函数return前插入了函数deferreturn()。

defer说明

- defer定义的延迟函数参数在defer语句出时就已经确定下来了
- defer定义顺序与实际执行顺序相反
- return不是原子操作，执行过程是: 保存返回值(若有)-->执行defer（若有）-->执行ret跳转
- 申请资源后立即使用defer关闭资源是好习惯

panic 和 recover

- `panic` 能够改变程序的控制流，调用 `panic` 后会立刻停止执行当前函数的剩余代码，并在当前 Goroutine 中递归执行调用方的 `defer`；
- `recover` 可以中止 `panic` 造成的程序崩溃。它是一个只能在 `defer` 中发挥作用的函数，在其他作用域中调用不会发挥作用；

程序多次调用 `panic` 也不会影响 `defer` 函数的正常执行，所以使用 `defer` 进行收尾工作一般来说都是安全的。

数据结构

`panic` 关键字在 Go 语言的源代码是由数据结构 `runtime._panic` 表示的。每当我们调用 `panic` 都会创建一个如下所示的数据结构存储相关信息：

```
type _panic struct {
    argp      unsafe.Pointer
    arg       interface{}
    link      *_panic
    recovered bool
    aborted   bool
    pc        uintptr
    sp        unsafe.Pointer
    goexit    bool
}
```

1. `argp` 是指向 `defer` 调用时参数的指针；
2. `arg` 是调用 `panic` 时传入的参数；
3. `link` 指向了更早调用的 `runtime._panic` 结构；
4. `recovered` 表示当前 `runtime._panic` 是否被 `recover` 恢复；
5. `aborted` 表示当前的 `panic` 是否被强行终止；

从数据结构中的 `link` 字段我们就可以推测出以下的结论：`panic` 函数可以被连续多次调用，它们之间通过 `link` 可以组成链表。

程序崩溃

这里先介绍分析 `panic` 函数是终止程序的实现原理。编译器会将关键字 `panic` 转换成 `runtime.gopanic`

该函数的执行过程包含以下几个步骤：

1. 创建新的 `runtime._panic` 并添加到所在 Goroutine 的 `_panic` 链表的最前面；
2. 在循环中不断从当前 Goroutine 的 `_defer` 中链表获取 `runtime._defer` 并调用 `runtime.reflectcall` 运行延迟调用函数；
3. 调用 `runtime.fatalpanic` 中止整个程序；

```
func gopanic(e interface{}) {
    gp := getg()
    ...
    var p _panic
    p.arg = e
    p.link = gp._panic
    gp._panic = (*_panic)(noescape(unsafe.Pointer(&p)))
```

```
    for {
        d := gp._defer
        if d == nil {
            break
        }

        d._panic = (*_panic)(noescape(unsafe.Pointer(&p)))

        reflectcall(nil, unsafe.Pointer(d.fn), deferArgs(d), uint32(d.siz),
uint32(d.siz))

        d._panic = nil
        d.fn = nil
        gp._defer = d.link

        freedefer(d)
        if p.recovered {
            ...
        }
    }

    fatalpanic(gp._panic)
    *(*int)(nil) = 0
}
```

`runtime.fatalpanic` 实现了无法被恢复的程序崩溃，它在中止程序之前会通过 `runtime.printpanics` 打印出全部的 `panic` 消息以及调用时传入的参数：

```
func fatalpanic(msgs *_panic) {
    pc := getcallerpc()
    sp := getcallersp()
    gp := getg()

    if startpanic_m() && msgs != nil {
        atomic.Xadd(&runningPanicDefers, -1)
        printpanics(msgs)
    }
    if dopanic_m(gp, pc, sp) {
        crash()
    }

    exit(2)
}
```

崩溃恢复

编译器会将关键字 `recover` 转换成 `runtime.gorecover`：

```go
func gorecover(argp uintptr) interface{} {
    gp := getg()
    p := gp._panic
    if p != nil && !p.recovered && argp == uintptr(p.argp) {
        p.recovered = true
        return p.arg
    }
    return nil
}
```

该函数的实现很简单，如果当前 Goroutine 没有调用 `panic`，那么该函数会直接返回 `nil`，这也是崩溃恢复在非 `defer` 中调用会失效的原因。

`runtime.gorecover` 函数中并不包含恢复程序的逻辑，程序的恢复是由 `runtime.gopanic` 函数负责的：

```go
func gopanic(e interface{}) {
    ...

    for {
        // 执行延迟调用函数，可能会设置 p.recovered = true
        ...

        pc := d.pc
        sp := unsafe.Pointer(d.sp)

        ...
        if p.recovered {
            gp._panic = p.link
            for gp._panic != nil && gp._panic.aborted {
                gp._panic = gp._panic.link
            }
            if gp._panic == nil {
                gp.sig = 0
            }
            gp.sigcode0 = uintptr(sp)
            gp.sigcode1 = pc
            mcall(recovery)
            throw("recovery failed")
        }
    }
    ...
}
```

上述这段代码也省略了 `defer` 的内联优化，它从 `runtime._defer` 中取出了程序计数器 `pc` 和栈指针 `sp` 并调用 `runtime.recovery` 函数触发 Goroutine 的调度，调度之前会准备好 `sp`、`pc` 以及函数的返回值：

```
func recovery(gp *g) {
    sp := gp.sigcode0
    pc := gp.sigcode1

    gp.sched.sp = sp
    gp.sched.pc = pc
    gp.sched.lr = 0
    gp.sched.ret = 1
    gogo(&gp.sched)
}
```

当我们在调用 `defer` 关键字时，调用时的栈指针 `sp` 和程序计数器 `pc` 就已经存储到了 `runtime._defer` 结构体中，这里的 `runtime.gogo` 函数会跳回 `defer` 关键字调用的位置。

`runtime.recovery` 在调度过程中会将函数的返回值设置成 1。从 `runtime.deferproc` 的注释中我们会发现，当 `runtime.deferproc` 函数的返回值是 1 时，编译器生成的代码会直接跳转到调用方函数返回之前并执行 `runtime.deferreturn`：

```
func deferproc(siz int32, fn *funcval) {
    ...
    return0()
}
```

跳转到 `runtime.deferreturn` 函数之后，程序就已经从 `panic` 中恢复了并执行正常的逻辑，而 `runtime.gorecover` 函数也能从 `runtime._panic` 结构中取出了调用 `panic` 时传入的 `arg` 参数并返回给调用方。

小结

1. 将 `panic` 和 `recover` 分别转换成 `runtime.gopanic` 和 `runtime.gorecover`；
2. 将 `defer` 转换成 `runtime.deferproc` 函数；
3. 在调用 `defer` 的函数末尾调用 `runtime.deferreturn` 函数；

make 和 new

- `make` 的作用是初始化内置的数据结构，也就是我们在前面提到的切片、哈希表和 Channel；
- `new` 的作用是根据传入的类型分配一片内存空间并返回指向这片内存空间的指针；

Go 语言会将代表 `make` 关键字的 `OMAKE` 节点根据参数类型的不同转换成了 `OMAKESLICE`、`OMAKEMAP` 和 `OMAKECHAN` 三种不同类型的节点，这些节点会调用不同的运行时函数来初始化相应的数据结构。

如果通过 `var` 或者 `new` 创建的变量不需要在当前作用域外生存，例如不用作为返回值返回给调用方，那么就不需要初始化在堆上。

```
func callnew(t *types.Type) *Node {
    ...
    n := nod(ONEWOBJ, typename(t), nil)
    ...
    return n
}

func (s *state) expr(n *Node) *ssa.Value {
    switch n.Op {
    case ONEWOBJ:
        if n.Type.Elem().Size() == 0 {
            return s.newValue1A(ssa.OpAddr, n.Type, zerobaseSym, s.sb)
        }
        typ := s.expr(n.Left)
        vv := s.rtcall(newobject, true, []*types.Type{n.Type}, typ)
        return vv[0]
    }
}
```

需要注意的是，无论是直接使用 `new`，还是使用 `var` 初始化变量，它们在编译器看来都是 `ONEW` 和 `ODCL` 节点。如果变量会逃逸到堆上，这些节点在这一阶段都会被 `cmd/compile/internal/gc.walkstmt` 转换成通过 `runtime.newobject` 函数并在堆上申请内存：

```
func walkstmt(n *Node) *Node {
    switch n.Op {
    case ODCL:
        v := n.Left
        if v.Class() == PAUTOHEAP {
            if prealloc[v] == nil {
                prealloc[v] = callnew(v.Type)
            }
            nn := nod(OAS, v.Name.Param.Heapaddr, prealloc[v])
            nn.SetColas(true)
            nn = typecheck(nn, ctxStmt)
            return walkstmt(nn)
        }
    case ONEW:
        if n.Esc == EscNone {
            r := temp(n.Type.Elem())
            r = nod(OAS, r, nil)
            r = typecheck(r, ctxStmt)
            init.Append(r)
            r = nod(OADDR, r.Left, nil)
            r = typecheck(r, ctxExpr)
```

```
            n = r
        } else {
            n = callnew(n.Type.Elem())
        }
    }
}
```

sync.Mutex

Mutex结构体

源码包`src/sync/mutex.go:Mutex`定义了互斥锁的数据结构：

```
type Mutex struct {
    state int32
    sema  uint32
}
```

- Mutex.state表示互斥锁的状态，比如是否被锁定等。
- Mutex.sema表示信号量，协程阻塞等待该信号量，解锁的协程释放信号量从而唤醒等待信号量的协程。

我们看到Mutex.state是32位的整型变量，内部实现时把该变量分成四份，用于记录Mutex的四种状态。

下图展示Mutex的内存布局：

state: 29bit（Waiter）、1bit（Starving）、1bit（Woken）、1bit（Locked）

- Locked: 表示该Mutex是否已被锁定，0：没有锁定 1：已被锁定。
- Woken: 表示是否有协程已被唤醒，0：没有协程唤醒 1：已有协程唤醒，正在加锁过程中。
- Starving：表示该Mutex是否处理饥饿状态， 0：没有饥饿 1：饥饿状态，说明有协程阻塞了超过1ms。
- Waiter: 表示阻塞等待锁的协程个数，协程解锁时根据此值来判断是否需要释放信号量。

协程之间抢锁实际上是抢给Locked赋值的权利，能给Locked域置1，就说明抢锁成功。抢不到的话就阻塞等待Mutex.sema信号量，一旦持有锁的协程解锁，等待的协程会依次被唤醒。

Woken和Starving主要用于控制协程间的抢锁过程，后面再进行了解。

Mutex方法

Mutext对外提供两个方法，实际上也只有这两个方法：

- Lock() : 加锁方法
- Unlock(): 解锁方法

下面我们分析一下加锁和解锁的过程，加锁分成功和失败两种情况，成功的话直接获取锁，失败后当前协程被阻塞，同样，解锁时跟据是否有阻塞协程也有两种处理。

加锁

假定当前只有一个协程在加锁，没有其他协程干扰，加锁过程会去判断Locked标志位是否为0，如果是0则把Locked位置1，代表加锁成功。从上图可见，加锁成功后，只是Locked位置1，其他状态位没发生变化。

假定加锁时，锁已被其他协程占用了，当协程B对一个已被占用的锁再次加锁时，Waiter计数器增加了1，此时协程B将被阻塞，直到Locked值变为0后才会被唤醒。

解锁

假定解锁时，没有其他协程阻塞，由于没有其他协程阻塞等待加锁，所以此时解锁时只需要把Locked位置为0即可，不需要释放信号量。

假定解锁时，有1个或多个协程阻塞，协程A解锁过程分为两个步骤，一是把Locked位置0，二是查看到Waiter>0，所以释放一个信号量，唤醒一个阻塞的协程，被唤醒的协程B把Locked位置1，于是协程B获得锁。

自旋过程

加锁时，如果当前Locked位为1，说明该锁当前由其他协程持有，尝试加锁的协程并不是马上转入阻塞，而是会持续的探测Locked位是否变为0，这个过程即为自旋过程。

自旋的好处是，当加锁失败时不必立即转入阻塞，有一定机会获取到锁，这样可以避免协程的切换。

自旋对应于CPU的"PAUSE"指令，CPU对该指令什么都不做，相当于CPU空转，对程序而言相当于sleep了一小段时间，时间非常短，当前实现是30个时钟周期。

自旋过程中会持续探测Locked是否变为0，连续两次探测间隔就是执行这些PAUSE指令，它不同于sleep，不需要将协程转为睡眠状态。

自旋条件

加锁时程序会自动判断是否可以自旋，无限制的自旋将会给CPU带来巨大压力，所以判断是否可以自旋就很重要了。

自旋必须满足以下所有条件：

- 自旋次数要足够小，通常为4，即自旋最多4次
- CPU核数要大于1，否则自旋没有意义，因为此时不可能有其他协程释放锁
- 协程调度机制中的Process数量要大于1，比如使用GOMAXPROCS()将处理器设置为1就不能启用自旋
- 协程调度机制中的可运行队列必须为空，否则会延迟协程调度

可见，自旋的条件是很苛刻的，总而言之就是不忙的时候才会启用自旋。

自旋的优势

自旋的优势是更充分的利用CPU，尽量避免协程切换。因为当前申请加锁的协程拥有CPU，如果经过短时间的自旋可以获得锁，当前协程可以继续运行，不必进入阻塞状态。

自旋的问题

如果自旋过程中获得锁，那么之前被阻塞的协程将无法获得锁，如果加锁的协程特别多，每次都通过自旋获得锁，那么之前被阻塞的进程将很难获得锁，从而进入饥饿状态。

为了避免协程长时间无法获取锁，自1.8版本以来增加了一个状态，即Mutex的Starving状态。这个状态下不会自旋，一旦有协程释放锁，那么一定会唤醒一个协程并成功加锁。

Mutex模式

前面分析加锁和解锁过程中只关注了Waiter和Locked位的变化，现在我们看一下Starving位的作用。

每个Mutex都有两个模式，称为Normal和Starving。下面分别说明这两个模式。

normal模式

默认情况下，Mutex的模式为normal。

该模式下，协程如果加锁不成功不会立即转入阻塞排队，而是判断是否满足自旋的条件，如果满足则会启动自旋过程，尝试抢锁。

starvation模式

自旋过程中能抢到锁，一定意味着同一时刻有协程释放了锁，我们知道释放锁时如果发现有阻塞等待的协程，还会释放一个信号量来唤醒一个等待协程，被唤醒的协程得到CPU后开始运行，此时发现锁已被抢占了，自己只好再次阻塞，不过阻塞前会判断自上次阻塞到本次阻塞经过了多长时间，如果超过1ms的话，会将Mutex标记为"饥饿"模式，然后再阻塞。

处于饥饿模式下，不会启动自旋过程，也即一旦有协程释放了锁，那么一定会唤醒协程，被唤醒的协程将会成功获取锁，同时也会把等待计数减1。

sync.RWMutex

实现读写锁需要解决如下几个问题：

1. 写锁需要阻塞写锁：一个协程拥有写锁时，其他协程写锁定需要阻塞
2. 写锁需要阻塞读锁：一个协程拥有写锁时，其他协程读锁定需要阻塞
3. 读锁需要阻塞写锁：一个协程拥有读锁时，其他协程写锁定需要阻塞
4. 读锁不能阻塞读锁：一个协程拥有读锁时，其他协程也可以拥有读锁

RWMutex数据结构

源码包`src/sync/rwmutex.go:RWMutex`定义了读写锁数据结构：

```
type RWMutex struct {
    w           Mutex  //用于控制多个写锁，获得写锁首先要获取该锁，如果有一个写锁在进行，那么再到来的写锁将会阻塞于此
    writerSem   uint32 //写阻塞等待的信号量，最后一个读者释放锁时会释放信号量
    readerSem   uint32 //读阻塞的协程等待的信号量，持有写锁的协程释放锁后会释放信号量
    readerCount int32  //记录读者个数
    readerWait  int32  //记录写阻塞时读者个数
}
```

RWMutex方法

RWMutex提供4个简单的接口来提供服务：

- RLock()：读锁定
- RUnlock()：解除读锁定
- Lock(): 写锁定，与Mutex完全一致
- Unlock()：解除写锁定，与Mutex完全一致

Lock() 逻辑

写锁定操作需要做两件事：

- 获取互斥锁
- 阻塞等待所有读操作结束（如果有的话）

Unlock()实现逻辑

解除写锁定要做两件事：

- 唤醒因读锁定而被阻塞的协程（如果有的话）
- 解除互斥锁

RLock()实现逻辑

读锁定需要做两件事：

- 增加读操作计数，即readerCount++
- 阻塞等待写操作结束(如果有的话)

RUnlock()实现逻辑

解除读锁定需要做两件事：

- 减少读操作计数，即readerCount--
- 唤醒等待写操作的协程（如果有的话）

sync.Cond

cond 使用场景：单生产者多消费者

Cond数据结构

`sync.Cond` 的 struct 定义如下：

```
type Cond struct {
    noCopy noCopy

    // L is held while observing or changing the condition
    L Locker

    notify  notifyList
    checker copyChecker
}
```

其中最核心的就是 `notifyList` 这个数据结构, 其源码在 runtime/sema.go

```
type notifyList struct {
    wait uint32
    notify uint32

    // List of parked waiters.
    lock mutex
    head *sudog
    tail *sudog
}
```

以上代码中，notifyList 包含两类属性：

1. `wait` 和 `notify`。这两个都是ticket值，每次调 Wait 时，ticket 都会递增，作为 goroutine 本次 Wait 的唯一标识，便于下次恢复。 wait 表示下次 `sync.Cond` Wait 的 ticket 值，notify 表示下次要唤醒的 goroutine 的 ticket 的值。这两个值都只增不减的。利用 wait 和 notify 可以实现 goroutine FIFO式的唤醒，具体见下文。
2. `head` 和 `tail`。等待在这个 `sync.Cond` 上的 goroutine 链表，head -> *sudog -> *sudog -> *sudog ... -> tail

Cond 方法

我们看下 `sync.Cond` 接口的用法:

1. `sync.NewCond(l Locker)`: 新建一个 sync.Cond 变量。注意该函数需要一个 Locker 作为必填参数，这是因为在 `cond.Wait()` 中底层会涉及到 Locker 的锁操作。
2. `cond.Wait()`: 等待被唤醒。唤醒期间会解锁并切走 goroutine。
3. `cond.Signal()`: 只唤醒一个最先 Wait 的 goroutine。对应的另外一个唤醒函数是 `Broadcast`，区别是 `Signal` 一次只会唤醒一个 goroutine，而 `Broadcast` 会将全部 Wait 的 goroutine 都唤醒。

Wait 操作

```
func (c *Cond) Wait() {
    c.checker.check()
    // 获取ticket
    t := runtime_notifyListAdd(&c.notify)
    // 注意这里，必须先解锁，因为 runtime_notifyListWait 要切走 goroutine
    // 所以这里要解锁，要不然其他 goroutine 没法获取到锁了
    c.L.Unlock()
    // 将当前 goroutine 加入到 notifyList 里面，然后切走 goroutine
    runtime_notifyListWait(&c.notify, t)
    // 这里已经唤醒了，因此需要再度锁上
    c.L.Lock()
}
```

Wait 函数虽然短短几行代码，但里面蕴含了很多重要的逻辑。整个逻辑可以拆分为 4 步：

第一步：调用 `runtime_notifyListAdd` 获取 ticket。ticket 是一次 Wait 操作的唯一标识，可以用来防止重复唤醒以及保证 FIFO 式的唤醒。

第二步：`c.L.Unlock()` 先把用户传进来的 locker 解锁。因为在 `runtime_notifyListWait` 中会调用 `gopark` 切走 goroutine。因此在切走之前，必须先把 Locker 解锁了。要不然其他 goroutine 获取不到这个锁，将会造成死锁问题。

第三步：`runtime_notifyListWait` 将当前 goroutine 加入到 notifyList 里面，然后切走goroutine。

Signal：唤醒最早 Wait 的 goroutine

```
func (c *Cond) Signal() {
    runtime_notifyListNotifyOne(&c.notify)
}

func notifyListNotifyOne(l *notifyList) {
    // 如果二者相等，说明没有需要唤醒的 goroutine
    if atomic.Load(&l.wait) == atomic.Load(&l.notify) {
        return
    }

    lock(&l.lock)

    t := l.notify
    if t == atomic.Load(&l.wait) {
        unlock(&l.lock)
        return
    }

    // Update the next notify ticket number.
    atomic.Store(&l.notify, t+1)

    for p, s := (*sudog)(nil), l.head; s != nil; p, s = s, s.next {
        if s.ticket == t {
            n := s.next
            if p != nil {
                p.next = n
            } else {
                l.head = n
            }
            if n == nil {
                l.tail = p
```

```
            }
            unlock(&l.lock)
            s.next = nil

            // 唤醒 goroutine
            readyWithTime(s, 4)
            return
        }
    }
    unlock(&l.lock)
}
```

那么，每次唤醒的时候，也会对应一个 `notify` 属性。例如当前 `notify` 属性等于 1，则去逐个检查 `notifyList` 链表中 元素，找到 `ticket` 等于 1 的 goroutine 并唤醒，同时将 `notify` 属性进行原子递增。

Cond 的惯用法及使用注意事项

sync.Cond 在使用时还是有一些需要注意的地方，否则使用不当将造成代码错误。

1. sync.Cond不能拷贝，否则将会造成`panic("sync.Cond is copied")`错误
2. Wait 的调用一定要放在 Lock 和 UnLock 中间，否则将会造成 `panic("sync: unlock of unlocked mutex")` 错误。代码如下:

```
c.L.Lock()
for !condition() {
        c.Wait()
}
... make use of condition ...
c.L.Unlock()
```

1. Wait 调用的条件检查一定要放在 for 循环中，代码如上。这是因为当 Boardcast 唤醒时，有可能其他 goroutine 先于当前 goroutine 唤醒并抢到锁，导致轮到当前 goroutine 抢到锁的时候，条件又不再满足了。因此，需要将条件检查放在 for 循环中。
2. Signal 和 Boardcast 两个唤醒操作不需要加锁。

sync.Once

`sync.Once` 是 Go 标准库提供的使函数只执行一次的实现，常应用于单例模式，例如初始化配置、保持数据库连接等

Once 数据结构

```
// 一个 Once 实例在使用之后不能被拷贝继续使用
type Once struct {
   done uint32 // done 表明了动作是否已经执行
   m    Mutex
}
```

once 方法

Once 对外仅暴露了唯一的方法 `Do(f func())`，f 为需要执行的函数。

```
func (o *Once) Do(f func()) {
    if atomic.LoadUint32(&o.done) == 0 {
      o.doSlow(f)
   }
}
```

- 不要拷贝一个 sync.Once 使用或作为参数传递，然后去执行 `Do`，值传递时 `done` 会归0，无法起到限制一次的效果。
- 不要在 `Do` 的 `f` 中嵌套调用 `Do`

sync.Pool

sync.Pool 是 Golang 内置的对象池技术，可用于缓存临时对象，避免因频繁建立临时对象所带来的消耗以及对 GC 造成的压力。

在许多知名的开源库中，都可以看到 sync.Pool 的大量使用。例如，HTTP 框架 Gin 用 sync.Pool 来复用每个请求都会创建的 `gin.Context` 对象。

> sync.Pool 不仅是并发安全的，而且实现了 lock free

Pool 的数据结构

sync.Pool 的 代码定义如下sync/pool.go

```
type Pool struct {
    noCopy noCopy

    local     unsafe.Pointer // local fixed-size per-P pool, actual type is
[P]poolLocal
    localSize uintptr        // size of the local array

    victim     unsafe.Pointer // local from previous cycle
    victimSize uintptr        // size of victims array

    New func() interface{}
}
```

在 GMP 调度模型中，M 代表了系统线程，而同一时间一个 M 上只能同时运行一个 P。那么也就意味着，从线程维度来看，在 P 上的逻辑都是单线程执行的。

sync.Pool 就是充分利用了 GMP 这一特点。对于同一个 sync.Pool ，每个 P 都有一个自己的本地对象池 `poolLocal`。

其中，我们需要着重关注以下三个字段：

- `local` 是个数组，长度为 P 的个数。其元素类型是 `poolLocal`。这里面存储着各个 P 对应的本地对象池。可以近似的看做 `[P]poolLocal`。
- `localSize`。代表 local 数组的长度。因为 P 可以在运行时通过调用 runtime.GOMAXPROCS 进行修改, 因此我们还是得通过 `localSize` 来对应 `local` 数组的长度。
- `New` 就是用户提供的创建对象的函数。这个选项也不是必需。当不填的时候，Get 有可能返回 nil。

我们再看下本地对象池 `poolLocal` 的定义，如下：

```
// 每个 P 都会有一个 poolLocal 的本地
type poolLocal struct {
    poolLocalInternal

    pad [128 - unsafe.Sizeof(poolLocalInternal{})%128]byte
}

type poolLocalInternal struct {
    private interface{}
    shared  poolChain
}
```

`pad` 变量的作用在下文会讲到，这里暂时不展开讨论。我们可以直接看 poolLocalInternal 的定义，其中每个本地对象池，都会包含两项：

- `private` 私有变量。Get 和 Put 操作都会优先存取 private 变量，如果 private 变量可以满足情况，则不再深入进行其他的复杂操作。
- `shared`。其类型为 poolChain，从名字不难看出这个是链表结构，这个就是 P 的本地对象池了。

poolChain 的实现

poolChain 是个链表结构，其链表头 HEAD 指向最新分配的元素项。链表中的每一项是一个 poolDequeue 对象。poolDequeue 本质上是一个 ring buffer 结构。其对应的代码定义如下：

```
type poolChain struct {
    head *poolChainElt
    tail *poolChainElt
}

type poolChainElt struct {
    poolDequeue
    next, prev *poolChainElt
}

type poolDequeue struct {
    headTail uint64
    vals []eface
}
```

为什么 poolChain 是这么一个链表 + ring buffer 的复杂结构呢？简单的每个链表项为单一元素不行吗？

使用 ring buffer 是因为它有以下优点：

1. 预先分配好内存，且分配的内存项可不断复用。
2. 由于ring buffer 本质上是个数组，是连续内存结构，非常利于 CPU Cache。在访问poolDequeue 某一项时，其附近的数据项都有可能加载到统一 Cache Line 中，访问速度更快。

Put 的实现

```
func (p *Pool) Put(x interface{}) {
    if x == nil {
        return
    }

    l, _ := p.pin()

    if l.private == nil {
        l.private = x
        x = nil
    }

    if x != nil {
        l.shared.pushHead(x)
    }

    runtime_procUnpin()
}
```

在 Put 函数中首先调用了 `pin()`。`pin` 函数非常重要，它有三个作用：

1. **初始化或者重新创建local数组。** 当 local 数组为空，或者和当前的 `runtime.GOMAXPROCS` 不一致时，将触发重新创建 local 数组，以和 P 的个数保持一致。
2. 取当前 P 对应的本地缓存池 poolLocal。其实代码逻辑很简单，就是从 local 数组中根据索引取元素。
3. **防止当前 P 被抢占。** 这点非常重要。

Get 的实现

```
func (p *Pool) Get() interface{} {
    l, pid := p.pin()

    x := l.private
    l.private = nil

    if x == nil {
        x, _ = l.shared.popHead()

        if x == nil {
            x = p.getSlow(pid)
        }
    }
    runtime_procUnpin()

    if x == nil && p.New != nil {
        x = p.New()
    }
    return x
}
```

其中 `pin()` 的作用和 `private` 对象的作用，和 PUT 操作中的一致

从当前 P 的 poolChain 中取数据时，是从链表头部开始取数据。 具体来说，先取位于链表头的 poolDequeue，然后从 poolDequeue 的头部开始取数据。

对象的清理

sync.Pool 没有对外开放对象清理策略和清理接口,当窃取其他 P 的对象时，会逐步淘汰已经为空的 poolDequeue。但除此之外，sync.Pool 一定也还有其他的对象清理机制.

Golang 对 sync.Pool 的清理逻辑非常简单粗暴。首先每个被使用的 sync.Pool，都会在初始化阶段被添加到全局变量 `allPools []*Pool` 对象中。Golang 的 runtime 将会在 **每轮 GC 前**，触发调用 poolCleanup 函数，清理 allPools。

```
func poolCleanup() {
    for _, p := range oldPools {
        p.victim = nil
        p.victimSize = 0
    }

    for _, p := range allPools {
        p.victim = p.local
        p.victimSize = p.localSize
        p.local = nil
        p.localSize = 0
```

```
    }

    oldPools, allPools = allPools, nil
}
```

性能之道

回顾下 sync.Pool 的实现细节，总结来说，sync.Pool 利用以下手段将程序性能做到了极致：

1. 利用 GMP 的特性，为每个 P 创建了一个本地对象池 poolLocal，尽量减少并发冲突。
2. 每个 poolLocal 都有一个 private 对象，优先存取 private 对象，可以避免进入复杂逻辑。
3. 在 Get 和 Put 期间，利用 `pin` 锁定当前 P，防止 goroutine 被抢占，造成程序混乱。
4. 在获取对象期间，利用对象窃取的机制，从其他 P 的本地对象池以及 victim 中获取对象。
5. 充分利用 CPU Cache 特性，提升程序性能。

sync.Map

Go 的内建 `map` 是不支持并发写操作的，原因是 `map` 写操作不是并发安全的，当你尝试多个 Goroutine 操作同一个 `map`，会产生报错：`fatal error: concurrent map writes`。

`sync.Map` 的实现原理可概括为：

- 通过 read 和 dirty 两个字段将读写分离，读的数据存在只读字段 read 上，将最新写入的数据则存在 dirty 字段上
- 读取时会先查询 read，不存在再查询 dirty，写入时则只写入 dirty
- 读取 read 并不需要加锁，而读或写 dirty 都需要加锁
- 另外有 misses 字段来统计 read 被穿透的次数（被穿透指需要读 dirty 的情况），超过一定次数则将 dirty 数据同步到 read 上
- 对于删除数据则直接通过标记来延迟删除

sync.Map数据结构

```
type Map struct {
    // 加锁作用，保护 dirty 字段
    mu Mutex
    // 只读的数据，实际数据类型为 readOnly
    read atomic.Value
    // 最新写入的数据
    dirty map[interface{}]*entry
    // 计数器，每次需要读 dirty 则 +1
    misses int
}
```

readOnly 的数据结构

```
type readOnly struct {
    // 内建 map
    m  map[interface{}]*entry
    // 表示 dirty 里存在 read 里没有的 key，通过该字段决定是否加锁读 dirty
    amended bool
}
```

entry 数据结构则用于存储值的指针

```
type entry struct {
    p unsafe.Pointer  // 等同于 *interface{}
}
```

sync.Map 常用的有以下方法：

- `Load`：读取指定 key 返回 value
- `Store`： 存储（增或改）key-value
- `Delete`： 删除指定 key
- `Range`：遍历所有键值对，参数是回调函数
- `LoadOrStore`：读取数据，若不存在则保存再读取

Load

```
func (m *Map) Load(key interface{}) (value interface{}, ok bool) {
    // 首先尝试从 read 中读取 readOnly 对象
    read, _ := m.read.Load().(readOnly)
    e, ok := read.m[key]

    // 如果不存在则尝试从 dirty 中获取
    if !ok && read.amended {
        m.mu.Lock()
        // 由于上面 read 获取没有加锁，为了安全再检查一次
        read, _ = m.read.Load().(readOnly)
        e, ok = read.m[key]

        // 确实不存在则从 dirty 获取
        if !ok && read.amended {
            e, ok = m.dirty[key]
            // 调用 miss 的逻辑
            m.missLocked()
        }
        m.mu.Unlock()
    }

    if !ok {
        return nil, false
    }
    // 从 entry.p 读取值
    return e.load()
}

func (m *Map) missLocked() {
    m.misses++
    if m.misses < len(m.dirty) {
        return
    }
    // 当 miss 积累过多，会将 dirty 存入 read，然后 将 amended = false，且 m.dirty =
nil
    m.read.Store(readOnly{m: m.dirty})
    m.dirty = nil
    m.misses = 0
}
```

Store

```
func (m *Map) Store(key, value interface{}) {
    read, _ := m.read.Load().(readOnly)
    // 如果 read 里存在，则尝试存到 entry 里
    if e, ok := read.m[key]; ok && e.tryStore(&value) {
        return
    }

    // 如果上一步没执行成功，则要分情况处理
    m.mu.Lock()
    read, _ = m.read.Load().(readOnly)
    // 和 Load 一样，重新从 read 获取一次
    if e, ok := read.m[key]; ok {
```

```
        // 情况 1：read 里存在
        if e.unexpungeLocked() {
            // 如果 p == expunged，则需要先将 entry 赋值给 dirty（因为 expunged 数据不会留在 dirty）
            m.dirty[key] = e
        }
        // 用值更新 entry
        e.storeLocked(&value)
    } else if e, ok := m.dirty[key]; ok {
        // 情况 2：read 里不存在，但 dirty 里存在，则用值更新 entry
        e.storeLocked(&value)
    } else {
        // 情况 3：read 和 dirty 里都不存在
        if !read.amended {
            // 如果 amended == false，则调用 dirtyLocked 将 read 拷贝到 dirty（除了被标记删除的数据）
            m.dirtyLocked()
            // 然后将 amended 改为 true
            m.read.Store(readOnly{m: read.m, amended: true})
        }
        // 将新的键值存入 dirty
        m.dirty[key] = newEntry(value)
    }
    m.mu.Unlock()
}

func (e *entry) tryStore(i *interface{}) bool {
    for {
        p := atomic.LoadPointer(&e.p)
        if p == expunged {
            return false
        }
        if atomic.CompareAndSwapPointer(&e.p, p, unsafe.Pointer(i)) {
            return true
        }
    }
}

func (e *entry) unexpungeLocked() (wasExpunged bool) {
    return atomic.CompareAndSwapPointer(&e.p, expunged, nil)
}

func (e *entry) storeLocked(i *interface{}) {
    atomic.StorePointer(&e.p, unsafe.Pointer(i))
}

func (m *Map) dirtyLocked() {
    if m.dirty != nil {
        return
    }

    read, _ := m.read.Load().(readOnly)
    m.dirty = make(map[interface{}]*entry, len(read.m))
    for k, e := range read.m {
        // 判断 entry 是否被删除，否则就存到 dirty 中
        if !e.tryExpungeLocked() {
            m.dirty[k] = e
        }
```

```
    }
}

func (e *entry) tryExpungeLocked() (isExpunged bool) {
    p := atomic.LoadPointer(&e.p)
    for p == nil {
        // 如果有 p == nil（即键值对被 delete），则会在这个时机被置为 expunged
        if atomic.CompareAndSwapPointer(&e.p, nil, expunged) {
            return true
        }
        p = atomic.LoadPointer(&e.p)
    }
    return p == expunged
}
```

Delete

```
func (m *Map) Delete(key interface{}) {
    m.LoadAndDelete(key)
}

// LoadAndDelete 作用等同于 Delete，并且会返回值与是否存在
func (m *Map) LoadAndDelete(key interface{}) (value interface{}, loaded bool) {
    // 获取逻辑和 Load 类似，read 不存在则查询 dirty
    read, _ := m.read.Load().(readOnly)
    e, ok := read.m[key]
    if !ok && read.amended {
        m.mu.Lock()
        read, _ = m.read.Load().(readOnly)
        e, ok = read.m[key]
        if !ok && read.amended {
            e, ok = m.dirty[key]
            m.missLocked()
        }
        m.mu.Unlock()
    }
    // 查询到 entry 后执行删除
    if ok {
        // 将 entry.p 标记为 nil，数据并没有实际删除
        // 真正删除数据并被被置为 expunged，是在 Store 的 tryExpungeLocked 中
        return e.delete()
    }
    return nil, false
}
```

总结

可见，通过这种读写分离的设计，解决了并发情况的写入安全，又使读取速度在大部分情况可以接近内建 `map`，非常适合读多写少的情况。

sync.WaitGroup

WaitGroup 数据结构

```
type WaitGroup struct {
    // 避免复制使用的一个技巧，可以告诉vet工具违反了复制使用的规则
    noCopy noCopy
    // 64bit(8bytes)的值分成两段，高32bit是计数值，低32bit是waiter的计数
    // 另外32bit是用作信号量的
    // 因为64bit值的原子操作需要64bit对齐，但是32bit编译器不支持，所以数组中的元素在不同的架构中不一样，具体处理看下面的方法
    // 总之，会找到对齐的那64bit作为state，其余的32bit做信号量
    state1 [3]uint32
}
```

`state1` 长度为 3 的 uint32 数组，Golang 内存对齐的概念

当 `state1` 是 *32* 位对齐：`state1` 数组的第一位是 *sema*，第二位是 *counter*，第三位是 *waiter*。

当 `state1` 是 64 位对齐：`state1` 数组的第一位是 counter，第二位是 waiter，第三位是 sema。

- `counter` 代表目前尚未完成的个数。`WaitGroup.Add(n)` 将会导致 `counter += n`, 而 `WaitGroup.Done()` 将导致 `counter--`。
- `waiter` 代表目前已调用 `WaitGroup.Wait` 的 goroutine 的个数。
- `sema` 对应于 golang 中 runtime 内部的信号量的实现。

WaitGroup方法

- Add(int)
- Done()
- Wait()

Add(delta int)

```
func (wg *WaitGroup) Add(delta int) {
  // statep表示wait数和计数值
  // 低32位表示wait数，高32位表示计数值
   statep, semap := wg.state()
   // uint64(delta)<<32 将delta左移32位
    // 因为高32位表示计数值，所以将delta左移32，增加到技术上
   state := atomic.AddUint64(statep, uint64(delta)<<32)
   // 当前计数值
   v := int32(state >> 32)
   // 阻塞在检查点的wait数
   w := uint32(state)
   if v > 0 || w == 0 {
      return
   }

   // 如果计数值v为0并且waiter的数量w不为0，那么state的值就是waiter的数量
    // 将waiter的数量设置为0，因为计数值v也是0,所以它们俩的组合*statep直接设置为0即可。此时需要并唤醒所有的waiter
   *statep = 0
   for ; w != 0; w-- {
```

```
        runtime_Semrelease(semap, false, 0)
    }
}
```

WaitGroup里还对使用逻辑进行了严格的检查，比如Wait()一旦开始不能Add().

Done

```
// Done decrements the WaitGroup counter by one.
func (wg *WaitGroup) Done() {
    wg.Add(-1)
}
```

Wait

```
// Wait blocks until the WaitGroup counter is zero.
func (wg *WaitGroup) Wait() {
    statep := wg.state()
        for {
        state := atomic.LoadUint64(statep)
        v := int32(state >> 32)
        w := uint32(state)
        if v == 0 {
            // Counter is 0, no need to wait.
            if race.Enabled {
                race.Enable()
                race.Acquire(unsafe.Pointer(wg))
            }
            return
        }
        // Increment waiters count.
        // 如果statep和state相等，则增加等待计数，同时进入if等待信号量
        // 此处做CAS，主要是防止多个goroutine里进行Wait()操作，每有一个goroutine进行了
wait，等待计数就加1
        // 如果这里不相等，说明statep，在 从读出来 到 CAS比较 的这个时间区间内，被别的
goroutine改写了，那么不进入if，回去再读一次，这样写避免用锁，更高效些
        if atomic.CompareAndSwapUint64(statep, state, state+1) {
            if race.Enabled && w == 0 {
                // Wait must be synchronized with the first Add.
                // Need to model this is as a write to race with the read in
Add.
                // As a consequence, can do the write only for the first waiter,
                // otherwise concurrent Waits will race with each other.
                race.Write(unsafe.Pointer(&wg.sema))
            }
            // 等待信号量
            runtime_Semacquire(&wg.sema)
            // 信号量来了，代表所有Add都已经Done
            if *statep != 0 {
                // 走到这里，说明在所有Add都已经Done后，触发信号量后，又被执行了Add
                panic("sync: WaitGroup is reused before previous Wait has
returned")
            }
            return
        }
    }
```

```
}
```

channel

Channel 是支撑 Go 语言高性能并发编程模型的重要结构

先入先出 FIFO队列思想

目前的 Channel 收发操作均遵循了先进先出的设计，具体规则如下：

- 先从 Channel 读取数据的 Goroutine 会先接收到数据；
- 先向 Channel 发送数据的 Goroutine 会得到先发送数据的权利；

无锁队列

无锁（lock-free）队列更准确的描述是使用乐观并发控制的队列。

channel数据结构

Go 语言的 Channel 在运行时使用 `runtime.hchan` 结构体表示。

```
type hchan struct {
    qcount   uint
    dataqsiz uint
    buf      unsafe.Pointer
    elemsize uint16
    closed   uint32
    elemtype *_type
    sendx    uint
    recvx    uint
    recvq    waitq
    sendq    waitq

    lock mutex
}
```

- `qcount` — Channel 中的元素个数；
- `dataqsiz` — Channel 中的循环队列的长度；
- `buf` — Channel 的缓冲区数据指针；
- `sendx` — Channel 的发送操作处理到的位置；
- `recvx` — Channel 的接收操作处理到的位置；

`elemsize` 和 `elemtype` 分别表示当前 Channel 能够收发的元素类型和大小； `sendq` 和 `recvq` 存储了当前 Channel 由于缓冲区空间不足而阻塞的 Goroutine 列表，这些等待队列使用双向链表 `runtime.waitq` 表示，链表中所有的元素都是 `runtime.sudog` 结构

```
type waitq struct {
    first *sudog
    last  *sudog
}
```

创建管道

使用 `make` 关键字创建channel。编译器会将 `make(chan int, 10)` 表达式转换成 `OMAKE` 类型的节点，并在类型检查阶段将 `OMAKE` 类型的节点转换成 `OMAKECHAN` 类型：

```
func typecheck1(n *Node, top int) (res *Node) {
    switch n.Op {
    case OMAKE:
        ...
        switch t.Etype {
        case TCHAN:
            l = nil
            if i < len(args) { // 带缓冲区的异步 Channel
                ...
                n.Left = l
            } else { // 不带缓冲区的同步 Channel
                n.Left = nodintconst(0)
            }
            n.Op = OMAKECHAN
        }
    }
}
```

`OMAKECHAN` 类型的节点最终都会在 SSA 中间代码生成阶段之前被转换成调用 `runtime.makechan` 或者 `runtime.makechan64` 的函数：

```
func makechan(t *chantype, size int) *hchan {
    elem := t.elem
    mem, _ := math.MulUintptr(elem.size, uintptr(size))

    var c *hchan
    switch {
    case mem == 0:
        c = (*hchan)(mallocgc(hchanSize, nil, true))
        c.buf = c.raceaddr()
    case elem.kind&kindNoPointers != 0:
        c = (*hchan)(mallocgc(hchanSize+mem, nil, true))
        c.buf = add(unsafe.Pointer(c), hchanSize)
    default:
        c = new(hchan)
        c.buf = mallocgc(mem, elem, true)
    }
    c.elemsize = uint16(elem.size)
    c.elemtype = elem
    c.dataqsiz = uint(size)
    return c
}
```

发送数据

当我们想要向 Channel 发送数据时，就需要使用 `ch <- i` 语句，编译器会将它解析成 `OSEND` 节点并在 `cmd/compile/internal/gc.walkexpr` 中转换成 `runtime.chansend1`：

```
func walkexpr(n *Node, init *Nodes) *Node {
    switch n.Op {
    case OSEND:
        n1 := n.Right
        n1 = assignconv(n1, n.Left.Type.Elem(), "chan send")
        n1 = walkexpr(n1, init)
        n1 = nod(OADDR, n1, nil)
        n = mkcall1(chanfn("chansend1", 2, n.Left.Type), nil, init, n.Left, n1)
    }
}
```

`runtime.chansend1` 只是调用了 `runtime.chansend` 并传入 Channel 和需要发送的数据。`runtime.chansend` 是向 Channel 中发送数据时一定会调用的函数，该函数包含了发送数据的全部逻辑，如果我们在调用时将 `block` 参数设置成 `true`，那么表示当前发送操作是阻塞的：

```
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
    lock(&c.lock)

    if c.closed != 0 {
        unlock(&c.lock)
        panic(plainError("send on closed channel"))
    }
```

在发送数据的逻辑执行之前会先为当前 Channel 加锁，防止多个线程并发修改数据。如果 Channel 已经关闭，那么向该 Channel 发送数据时会报 “send on closed channel” 错误并中止程序。

因为 `runtime.chansend` 函数的实现比较复杂，所以我们这里将该函数的执行过程分成以下的三个部分：

- 当存在等待的接收者时，通过 `runtime.send` 直接将数据发送给阻塞的接收者；
- 当缓冲区存在空余空间时，将发送的数据写入 Channel 的缓冲区；
- 当不存在缓冲区或者缓冲区已满时，等待其他 Goroutine 从 Channel 接收数据；

直接发送

如果目标 Channel 没有被关闭并且已经有处于读等待的 Goroutine，那么 `runtime.chansend` 会从接收队列 `recvq` 中取出最先陷入等待的 Goroutine 并直接向它发送数据：

```
    if sg := c.recvq.dequeue(); sg != nil {
        send(c, sg, ep, func() { unlock(&c.lock) }, 3)
        return true
    }
```

直接发送数据时会调用 `runtime.send`，该函数的执行可以分成两个部分：

1. 调用 `runtime.sendDirect` 将发送的数据直接拷贝到 `x = <-c` 表达式中变量 `x` 所在的内存地址上；
2. 调用 `runtime.goready` 将等待接收数据的 Goroutine 标记成可运行状态 `Grunnable` 并把该 Goroutine 放到发送方所在的处理器的 `runnext` 上等待执行，该处理器在下一次调度时会立刻唤醒数据的接收方；

```
func send(c *hchan, sg *sudog, ep unsafe.Pointer, unlockf func(), skip int) {
    if sg.elem != nil {
        sendDirect(c.elemtype, sg, ep)
        sg.elem = nil
    }
    gp := sg.g
    unlockf()
    gp.param = unsafe.Pointer(sg)
    goready(gp, skip+1)
}
```

需要注意的是，发送数据的过程只是将接收方的 Goroutine 放到了处理器的 `runnext` 中，程序没有立刻执行该 Goroutine。

缓冲区

如果创建的 Channel 包含缓冲区并且 Channel 中的数据没有装满，会执行下面这段代码：

```
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
    ...
    if c.qcount < c.dataqsiz {
        qp := chanbuf(c, c.sendx)
        typedmemmove(c.elemtype, qp, ep)
        c.sendx++
        if c.sendx == c.dataqsiz {
            c.sendx = 0
        }
        c.qcount++
        unlock(&c.lock)
        return true
    }
    ...
}
```

在这里我们首先会使用 `runtime.chanbuf` 计算出下一个可以存储数据的位置，然后通过 `runtime.typedmemmove` 将发送的数据拷贝到缓冲区中并增加 `sendx` 索引和 `qcount` 计数器。

如果当前 Channel 的缓冲区未满，向 Channel 发送的数据会存储在 Channel 的 `sendx` 索引所在的位置并将 `sendx` 索引加一。因为这里的 `buf` 是一个循环数组，所以当 `sendx` 等于 `dataqsiz` 时会重新回到数组开始的位置。

阻塞发送

当 Channel 没有接收者能够处理数据时，向 Channel 发送数据会被下游阻塞，当然使用 `select` 关键字可以向 Channel 非阻塞地发送消息。向 Channel 阻塞地发送数据会执行下面的代码，我们可以简单梳理一下这段代码的逻辑：

```
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
    ...
    if !block {
        unlock(&c.lock)
        return false
    }

    gp := getg()
```

```
    mysg := acquireSudog()
    mysg.elem = ep
    mysg.g = gp
    mysg.c = c
    gp.waiting = mysg
    c.sendq.enqueue(mysg)
    goparkunlock(&c.lock, waitReasonChanSend, traceEvGoBlockSend, 3)

    gp.waiting = nil
    gp.param = nil
    mysg.c = nil
    releaseSudog(mysg)
    return true
}
```

1. 调用 `runtime.getg` 获取发送数据使用的 Goroutine；
2. 执行 `runtime.acquireSudog` 获取 `runtime.sudog` 结构并设置这一次阻塞发送的相关信息，例如发送的 Channel、是否在 select 中和待发送数据的内存地址等；
3. 将刚刚创建并初始化的 `runtime.sudog` 加入发送等待队列，并设置到当前 Goroutine 的 `waiting` 上，表示 Goroutine 正在等待该 `sudog` 准备就绪；
4. 调用 `runtime.goparkunlock` 将当前的 Goroutine 陷入沉睡等待唤醒；
5. 被调度器唤醒后会执行一些收尾工作，将一些属性置零并且释放 `runtime.sudog` 结构体；

总结

我们在这里可以简单梳理和总结一下使用 `ch <- i` 表达式向 Channel 发送数据时遇到的几种情况：

1. 如果当前 Channel 的 `recvq` 上存在已经被阻塞的 Goroutine，那么会直接将数据发送给当前 Goroutine 并将其设置成下一个运行的 Goroutine；
2. 如果 Channel 存在缓冲区并且其中还有空闲的容量，我们会直接将数据存储到缓冲区 `sendx` 所在的位置上；
3. 如果不满足上面的两种情况，会创建一个 `runtime.sudog` 结构并将其加入 Channel 的 `sendq` 队列中，当前 Goroutine 也会陷入阻塞等待其他的协程从 Channel 接收数据；

发送数据的过程中包含几个会触发 Goroutine 调度的时机：

1. 发送数据时发现 Channel 上存在等待接收数据的 Goroutine，立刻设置处理器的 `runnext` 属性，但是并不会立刻触发调度；
2. 发送数据时并没有找到接收方并且缓冲区已经满了，这时会将自己加入 Channel 的 `sendq` 队列并调用 `runtime.goparkunlock` 触发 Goroutine 的调度让出处理器的使用权；

接收数据

我们接下来继续介绍 Channel 操作的另一方：接收数据。Go 语言中可以使用两种不同的方式去接收 Channel 中的数据：

```
i <- ch
i, ok <- ch
```

直接接收

当 Channel 的 `sendq` 队列中包含处于等待状态的 Goroutine 时，该函数会取出队列头等待的 Goroutine，处理的逻辑和发送时相差无几，只是发送数据时调用的是 `runtime.send` 函数，而接收数据时使用 `runtime.recv`：

```
    if sg := c.sendq.dequeue(); sg != nil {
        recv(c, sg, ep, func() { unlock(&c.lock) }, 3)
        return true, true
    }
```

`runtime.recv` 的实现比较复杂：

```
func recv(c *hchan, sg *sudog, ep unsafe.Pointer, unlockf func(), skip int) {
    if c.dataqsiz == 0 {
        if ep != nil {
            recvDirect(c.elemtype, sg, ep)
        }
    } else {
        qp := chanbuf(c, c.recvx)
        if ep != nil {
            typedmemmove(c.elemtype, ep, qp)
        }
        typedmemmove(c.elemtype, qp, sg.elem)
        c.recvx++
        c.sendx = c.recvx // c.sendx = (c.sendx+1) % c.dataqsiz
    }
    gp := sg.g
    gp.param = unsafe.Pointer(sg)
    goready(gp, skip+1)
}
```

该函数会根据缓冲区的大小分别处理不同的情况：

- 如果 Channel 不存在缓冲区；
    1. 调用 `runtime.recvDirect` 将 Channel 发送队列中 Goroutine 存储的 `elem` 数据拷贝到目标内存地址中；
- 如果 Channel 存在缓冲区；
    1. 将队列中的数据拷贝到接收方的内存地址；
    2. 将发送队列头的数据拷贝到缓冲区中，释放一个阻塞的发送方；

缓冲区

当 Channel 的缓冲区中已经包含数据时，从 Channel 中接收数据会直接从缓冲区中 `recvx` 的索引位置中取出数据进行处理：

```
func chanrecv(c *hchan, ep unsafe.Pointer, block bool) (selected, received bool)
{
    ...
    if c.qcount > 0 {
        qp := chanbuf(c, c.recvx)
        if ep != nil {
            typedmemmove(c.elemtype, ep, qp)
        }
```

```
            typedmemclr(c.elemtype, qp)
            c.recvx++
            if c.recvx == c.dataqsiz {
                c.recvx = 0
            }
            c.qcount--
            return true, true
        }
        ...
    }
```

如果接收数据的内存地址不为空，那么会使用 `runtime.typedmemmove` 将缓冲区中的数据拷贝到内存中、清除队列中的数据并完成收尾工作。

收尾工作包括递增 `recvx`，一旦发现索引超过了 Channel 的容量时，会将它归零重置循环队列的索引；除此之外，该函数还会减少 `qcount` 计数器并释放持有 Channel 的锁。

阻塞接收

当 Channel 的发送队列中不存在等待的 Goroutine 并且缓冲区中也不存在任何数据时，从管道中接收数据的操作会变成阻塞的，然而不是所有的接收操作都是阻塞的，与 `select` 语句结合使用时就可能会使用到非阻塞的接收操作：

```
func chanrecv(c *hchan, ep unsafe.Pointer, block bool) (selected, received bool)
{
    ...
    if !block {
        unlock(&c.lock)
        return false, false
    }

    gp := getg()
    mysg := acquireSudog()
    mysg.elem = ep
    gp.waiting = mysg
    mysg.g = gp
    mysg.c = c
    c.recvq.enqueue(mysg)
    goparkunlock(&c.lock, waitReasonChanReceive, traceEvGoBlockRecv, 3)

    gp.waiting = nil
    closed := gp.param == nil
    gp.param = nil
    releaseSudog(mysg)
    return true, !closed
}
```

Go

在正常的接收场景中，我们会使用 `runtime.sudog` 将当前 Goroutine 包装成一个处于等待状态的 Goroutine 并将其加入到接收队列中。

完成入队之后，上述代码还会调用 `runtime.goparkunlock` 立刻触发 Goroutine 的调度，让出处理器的使用权并等待调度器的调度。

小结

我们梳理一下从 Channel 中接收数据时可能会发生的五种情况：

1. 如果 Channel 为空，那么会直接调用 `runtime.gopark` 挂起当前 Goroutine；
2. 如果 Channel 已经关闭并且缓冲区没有任何数据，`runtime.chanrecv` 会直接返回；
3. 如果 Channel 的 `sendq` 队列中存在挂起的 Goroutine，会将 `recvx` 索引所在的数据拷贝到接收变量所在的内存空间上并将 `sendq` 队列中 Goroutine 的数据拷贝到缓冲区；
4. 如果 Channel 的缓冲区中包含数据，那么直接读取 `recvx` 索引对应的数据；
5. 在默认情况下会挂起当前的 Goroutine，将 `runtime.sudog` 结构加入 `recvq` 队列并陷入休眠等待调度器的唤醒；

我们总结一下从 Channel 接收数据时，会触发 Goroutine 调度的两个时机：

1. 当 Channel 为空时；
2. 当缓冲区中不存在数据并且也不存在数据的发送者时；

关闭管道

编译器会将用于关闭管道的 `close` 关键字转换成 `OCLOSE` 节点以及 `runtime.closechan` 函数。

当 Channel 是一个空指针或者已经被关闭时，Go 语言运行时都会直接崩溃并抛出异常：

```
func closechan(c *hchan) {
    if c == nil {
        panic(plainError("close of nil channel"))
    }

    lock(&c.lock)
    if c.closed != 0 {
        unlock(&c.lock)
        panic(plainError("close of closed channel"))
    }
```

Go

处理完了这些异常的情况之后就可以开始执行关闭 Channel 的逻辑了，下面这段代码的主要工作就是将 `recvq` 和 `sendq` 两个队列中的数据加入到 Goroutine 列表 `gList` 中，与此同时该函数会清除所有 `runtime.sudog` 上未被处理的元素：

```
    c.closed = 1

    var glist gList
    for {
        sg := c.recvq.dequeue()
        if sg == nil {
            break
        }
        if sg.elem != nil {
            typedmemclr(c.elemtype, sg.elem)
            sg.elem = nil
        }
        gp := sg.g
        gp.param = nil
        glist.push(gp)
    }
```

```
    for {
        sg := c.sendq.dequeue()
        ...
    }
    for !glist.empty() {
        gp := glist.pop()
        gp.schedlink = 0
        goready(gp, 3)
    }
}
```

该函数在最后会为所有被阻塞的 Goroutine 调用 `runtime.goready` 触发调度。

Select

I/O 多路复用被用来处理同一个事件循环中的多个 I/O 事件。I/O 多路复用需要使用特定的系统调用，最常见的系统调用是 `select`，该函数可以同时监听最多 1024 个文件描述符的可读或者可写状态：

几大特点

- 可以实现两种收发操作，阻塞收发和非阻塞收发
- 当多个case ready的情况下会随机选择一个执行，不是顺序执行
- 没有ready的case时，有default语句，执行default语句;没有default语句，阻塞直到某个case ready
- select中的case必须是channel操作
- default语句总是可运行的

几大用处

- 超时处理，一旦超时就返回，不再等待
- 生产，消费者通信、
- 非阻塞读写

数据结构

select 中的case用runtime.scase结构体来表示，具体如下

```
type scase struct {
    c           *hchan         // 除了default其他都是channel操作，所以需要一个channel变量存储通道信息
    elem        unsafe.Pointer // data element
    kind        uint16//case类型 default语句是caseDefault类型，接收通道是caseRecv，发送通道是caseSend
    pc          uintptr // race pc (for race detector / msan)
    releasetime int64
}
```

通道类型具体代码定义如下

```
const (
    caseNil = iota
    caseRecv
    caseSend
    caseDefault
)
```

编译器在中间代码生成期间会根据 select 中 case 的不同对控制语句进行优化，这一过程都发在 cmd/compile/internal/gc.walkselectcases 函数中，常规编译之后，调用方法runtime.selectgo，具体操作都在这个方法里面，传入的参数是scase数组，传出的参数是随机选择的ready的scase下标，如下

```
func selectgo(cas0 *scase, order0 *uint16, ncases int) (int, bool) {
    if debugSelect {
        print("select: cas0=", cas0, "\n")
    }
...
```

初始化

因为文件 I/O、网络 I/O 以及计时器都依赖网络轮询器，所以 Go 语言会通过以下两条不同路径初始化网络轮询器：

1. `internal/poll.pollDesc.init` — 通过 `net.netFD.init` 和 `os.newFile` 初始化网络 I/O 和文件 I/O 的轮询信息时；
2. `runtime.doaddtimer` — 向处理器中增加新的计时器时；

网络轮询器的初始化会使用 `runtime.poll_runtime_pollServerInit` 和 `runtime.netpollGenericInit` 两个函数：

```
func poll_runtime_pollServerInit() {
    netpollGenericInit()
}

func netpollGenericInit() {
    if atomic.Load(&netpollInited) == 0 {
        lock(&netpollInitLock)
        if netpollInited == 0 {
            netpollinit()
            atomic.Store(&netpollInited, 1)
        }
        unlock(&netpollInitLock)
    }
}
```

`runtime.netpollGenericInit` 会调用平台上特定实现的 `runtime.netpollinit`，即 Linux 上的 `epoll`，它主要做了以下几件事情：

1. 是调用 `epollcreate1` 创建一个新的 `epoll` 文件描述符，这个文件描述符会在整个程序的生命周期中使用；
2. 通过 `runtime.nonblockingPipe` 创建一个用于通信的管道；
3. 使用 `epollctl` 将用于读取数据的文件描述符打包成 `epollevent` 事件加入监听；

```
var (
    epfd int32 = -1
    netpollBreakRd, netpollBreakWr uintptr
)

func netpollinit() {
    epfd = epollcreate1(_EPOLL_CLOEXEC)
    r, w, _ := nonblockingPipe()
    ev := epollevent{
        events: _EPOLLIN,
    }
    *(**uintptr)(unsafe.Pointer(&ev.data)) = &netpollBreakRd
    epollctl(epfd, _EPOLL_CTL_ADD, r, &ev)
    netpollBreakRd = uintptr(r)
    netpollBreakWr = uintptr(w)
}
```

初始化的管道为我们提供了中断多路复用等待文件描述符中事件的方法，`runtime.netpollBreak` 会向管道中写入数据唤醒 `epoll`：

```
func netpollBreak() {
    for {
```

```
            var b byte
            n := write(netpollBreakWr, unsafe.Pointer(&b), 1)
            if n == 1 {
                break
            }
            if n == -_EINTR {
                continue
            }
            if n == -_EAGAIN {
                return
            }
        }
    }
```

因为目前的计时器由网络轮询器管理和触发，它能够让网络轮询器立刻返回并让运行时检查是否有需要触发的计时器。

轮询事件

调用 `internal/poll.pollDesc.init` 初始化文件描述符时不止会初始化网络轮询器，还会通过 `runtime.poll_runtime_pollOpen` 重置轮询信息 `runtime.pollDesc` 并调用 `runtime.netpollopen` 初始化轮询事件：

```
func poll_runtime_pollOpen(fd uintptr) (*pollDesc, int) {
    pd := pollcache.alloc()
    lock(&pd.lock)
    if pd.wg != 0 && pd.wg != pdReady {
        throw("runtime: blocked write on free polldesc")
    }
    ...
    pd.fd = fd
    pd.closing = false
    pd.everr = false
    ...
    pd.wseq++
    pd.wg = 0
    pd.wd = 0
    unlock(&pd.lock)

    var errno int32
    errno = netpollopen(fd, pd)
    return pd, int(errno)
}
```

`runtime.netpollopen` 的实现非常简单，它会调用 `epollctl` 向全局的轮询文件描述符 `epfd` 中加入新的轮询事件监听文件描述符的可读和可写状态：

```
func netpollopen(fd uintptr, pd *pollDesc) int32 {
    var ev epollevent
    ev.events = _EPOLLIN | _EPOLLOUT | _EPOLLRDHUP | _EPOLLET
    *(**pollDesc)(unsafe.Pointer(&ev.data)) = pd
    return -epollctl(epfd, _EPOLL_CTL_ADD, int32(fd), &ev)
}
```

从全局的 `epfd` 中删除待监听的文件描述符可以使用 `runtime.netpollclose`，因为该函数的实现与 `runtime.netpollopen` 比较相似，所以这里不展开分析了。

context

Go 语言中用来设置截止日期、同步信号，传递请求相关值的结构体。上下文与 Goroutine 有比较密切的关系，是 Go 语言中独特的设计，在其他编程语言中我们很少见到类似的概念。

接口定义

源码包中src/context/context.go:Context 定义了该接口：

```
type Context interface {
    Deadline() (deadline time.Time, ok bool)
    Done() <-chan struct{}
    Err() error
    Value(key interface{}) interface{}
}
```

`context.Context` 是 Go 语言在 1.7 版本中引入标准库的接口1，该接口定义了四个需要实现的方法，其中包括：

1. `Deadline` — 返回 `context.Context` 被取消的时间，也就是完成工作的截止日期；
2. `Done` — 返回一个 Channel，这个 Channel 会在当前工作完成或者上下文被取消后关闭，多次调用 `Done` 方法会返回同一个 Channel；
3. Err — 返回`context.Context`结束的原因，它只会在Done方法对应的 Channel 关闭时返回非空的值；
    1. 如果 `context.Context` 被取消，会返回 `Canceled` 错误；
    2. 如果 `context.Context` 超时，会返回 `DeadlineExceeded` 错误；
4. `Value` — 从 `context.Context` 中获取键对应的值，对于同一个上下文来说，多次调用 `Value` 并传入相同的 `Key` 会返回相同的结果，该方法可以用来传递请求特定的数据；

设计原理

在 Goroutine 构成的树形结构中对信号进行同步以减少计算资源的浪费是 `context.Context` 的最大作用。每一个 `context.Context` 都会从最顶层的 Goroutine 一层一层传递到最下层。

我们可以通过一个代码片段了解 `context.Context` 是如何对信号进行同步的。在这段代码中，我们创建了一个过期时间为 1s 的上下文，并向上下文传入 `handle` 函数，该方法会使用 500ms 的时间处理传入的请求：

```
func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 1*time.Second)
    defer cancel()

    go handle(ctx, 500*time.Millisecond)
    select {
    case <-ctx.Done():
        fmt.Println("main", ctx.Err())
    }
```

```
}

func handle(ctx context.Context, duration time.Duration) {
    select {
    case <-ctx.Done():
        fmt.Println("handle", ctx.Err())
    case <-time.After(duration):
        fmt.Println("process request with", duration)
    }
}
```

默认上下文

`context` 包中最常用的方法还是 `context.Background`、`context.TODO`，这两个方法都会返回预先初始化好的私有变量 `background` 和 `todo`，它们会在同一个 Go 程序中被复用：

```
func Background() Context {
    return background
}

func TODO() Context {
    return todo
}
```

这两个私有变量都是通过 `new(emptyCtx)` 语句初始化的，它们是指向私有结构体 `context.emptyCtx` 的指针，这是最简单、最常用的上下文类型：

```
type emptyCtx int

func (*emptyCtx) Deadline() (deadline time.Time, ok bool) {
    return
}

func (*emptyCtx) Done() <-chan struct{} {
    return nil
}

func (*emptyCtx) Err() error {
    return nil
}

func (*emptyCtx) Value(key interface{}) interface{} {
    return nil
}
```

从源代码来看，`context.Background` 和 `context.TODO` 也只是互为别名，没有太大的差别，只是在使用和语义上稍有不同：

- `context.Background` 是上下文的默认值，所有其他的上下文都应该从它衍生出来；
- `context.TODO` 应该仅在不确定应该使用哪种上下文时使用；

取消信号

`context.WithCancel` 函数能够从 `context.Context` 中衍生出一个新的子上下文并返回用于取消该上下文的函数。一旦我们执行返回的取消函数，当前上下文以及它的子上下文都会被取消，所有的 Goroutine 都会同步收到这一取消信号。

```
func WithCancel(parent Context) (ctx Context, cancel CancelFunc) {
    c := newCancelCtx(parent)
    propagateCancel(parent, &c)
    return &c, func() { c.cancel(true, Canceled) }
}
```

- `context.newCancelCtx` 将传入的上下文包装成私有结构体 `context.cancelCtx`；
- `context.propagateCancel` 会构建父子上下文之间的关联，当父上下文被取消时，子上下文也会被取消：

```
func propagateCancel(parent Context, child canceler) {
    done := parent.Done()
    if done == nil {
        return // 父上下文不会触发取消信号
    }
    select {
    case <-done:
        child.cancel(false, parent.Err()) // 父上下文已经被取消
        return
    default:
    }

    if p, ok := parentCancelCtx(parent); ok {
        p.mu.Lock()
        if p.err != nil {
            child.cancel(false, p.err)
        } else {
            p.children[child] = struct{}{}
        }
        p.mu.Unlock()
    } else {
        go func() {
            select {
            case <-parent.Done():
                child.cancel(false, parent.Err())
            case <-child.Done():
            }
        }()
    }
}
```

`context.cancelCtx` 实现的几个接口方法也没有太多值得分析的地方，该结构体最重要的方法是 `context.cancelCtx.cancel`，该方法会关闭上下文中的 Channel 并向所有的子上下文同步取消信号：

```
func (c *cancelCtx) cancel(removeFromParent bool, err error) {
    c.mu.Lock()
    if c.err != nil {
        c.mu.Unlock()
        return
    }
```

```
    c.err = err
    if c.done == nil {
        c.done = closedchan
    } else {
        close(c.done)
    }
    for child := range c.children {
        child.cancel(false, err)
    }
    c.children = nil
    c.mu.Unlock()

    if removeFromParent {
        removeChild(c.Context, c)
    }
}
```

除了 `context.WithCancel` 之外，`context` 包中的另外两个函数 `context.WithDeadline` 和 `context.WithTimeout` 也都能创建可以被取消的计时器上下文 `context.timerCtx`：

```
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc) {
    return WithDeadline(parent, time.Now().Add(timeout))
}

func WithDeadline(parent Context, d time.Time) (Context, CancelFunc) {
    if cur, ok := parent.Deadline(); ok && cur.Before(d) {
        return WithCancel(parent)
    }
    c := &timerCtx{
        cancelCtx: newCancelCtx(parent),
        deadline:  d,
    }
    propagateCancel(parent, c)
    dur := time.Until(d)
    if dur <= 0 {
        c.cancel(true, DeadlineExceeded) // 已经过了截止日期
        return c, func() { c.cancel(false, Canceled) }
    }
    c.mu.Lock()
    defer c.mu.Unlock()
    if c.err == nil {
        c.timer = time.AfterFunc(dur, func() {
            c.cancel(true, DeadlineExceeded)
        })
    }
    return c, func() { c.cancel(true, Canceled) }
}
```

`context.WithDeadline` 在创建 `context.timerCtx` 的过程中判断了父上下文的截止日期与当前日期，并通过 `time.AfterFunc` 创建定时器，当时间超过了截止日期后会调用 `context.timerCtx.cancel` 同步取消信号。

`context.timerCtx` 内部不仅通过嵌入 `context.cancelCtx` 结构体继承了相关的变量和方法，还通过持有的定时器 `timer` 和截止时间 `deadline` 实现了定时取消的功能：

```
type timerCtx struct {
```

```
    cancelCtx
    timer *time.Timer // Under cancelCtx.mu.

    deadline time.Time
}

func (c *timerCtx) Deadline() (deadline time.Time, ok bool) {
    return c.deadline, true
}

func (c *timerCtx) cancel(removeFromParent bool, err error) {
    c.cancelCtx.cancel(false, err)
    if removeFromParent {
        removeChild(c.cancelCtx.Context, c)
    }
    c.mu.Lock()
    if c.timer != nil {
        c.timer.Stop()
        c.timer = nil
    }
    c.mu.Unlock()
}
```

`context.timerCtx.cancel` 方法不仅调用了 `context.cancelCtx.cancel`，还会停止持有的定时器减少不必要的资源浪费。

传值方法

，`context` 包中的 `context.WithValue` 能从父上下文中创建一个子上下文，传值的子上下文使用 `context.valueCtx` 类型：

```
func WithValue(parent Context, key, val interface{}) Context {
    if key == nil {
        panic("nil key")
    }
    if !reflectlite.TypeOf(key).Comparable() {
        panic("key is not comparable")
    }
    return &valueCtx{parent, key, val}
}
```

Go

`context.valueCtx` 结构体会将除了 `Value` 之外的 `Err`、`Deadline` 等方法代理到父上下文中，它只会响应 `context.valueCtx.Value` 方法，该方法的实现也很简单：

```go
type valueCtx struct {
    Context
    key, val interface{}
}

func (c *valueCtx) Value(key interface{}) interface{} {
    if c.key == key {
        return c.val
    }
    return c.Context.Value(key)
}
```

如果 `context.valueCtx` 中存储的键值对与 `context.valueCtx.Value` 方法中传入的参数不匹配，就会从父上下文中查找该键对应的值直到某个父上下文中返回 `nil` 或者查找到对应的值。

goroutine调度

Goroutine主要概念如下：

- G（Goroutine）：即Go协程，每个go关键字都会创建一个协程。
- M（Machine）： 工作线程，在Go中称为Machine。
- P(Processor): 处理器（Go中定义的一个概念，不是指CPU），包含运行Go代码的必要资源，也有调度goroutine的能力。

> P的个数默认等于CPU核数，每个M必须持有一个P才可以执行G，一般情况下M的个数会略大于P的个数，这多出来的M将会在G产生系统调用时发挥作用。

调度器启动

```
func schedinit() {
    _g_ := getg()
    ...

    sched.maxmcount = 10000

    ...
    sched.lastpoll = uint64(nanotime())
    procs := ncpu
    if n, ok := atoi32(gogetenv("GOMAXPROCS")); ok && n > 0 {
        procs = n
    }
    if procresize(procs) != nil {
        throw("unknown runnable goroutine during bootstrap")
    }
}
```

创建 Goroutine

想要启动一个新的 Goroutine 来执行任务时，我们需要使用 Go 语言的 `go` 关键字，编译器会通过 `cmd/compile/internal/gc.state.stmt` 和 `cmd/compile/internal/gc.state.call` 两个方法将该关键字转换成 `runtime.newproc` 函数调用：

```
func (s *state) call(n *Node, k callKind) *ssa.Value {
    if k == callDeferStack {
        ...
    } else {
        switch {
        case k == callGo:
            call = s.newValue1A(ssa.OpStaticCall, types.TypeMem, newproc,
s.mem())
        default:
        }
    }
    ...
}
```

线程管理

Go 语言的运行时会通过调度器改变线程的所有权，它也提供了 `runtime.LockOSThread` 和 `runtime.UnlockOSThread` 让我们有能力绑定 Goroutine 和线程完成一些比较特殊的操作。Goroutine 应该在调用操作系统服务或者依赖线程状态的非 Go 语言库时调用 `runtime.LockOSThread` 函数，例如：C 语言图形库等。

`runtime.LockOSThread` 会通过如下所示的代码绑定 Goroutine 和当前线程：

该方法的作用是可以让当前协程绑定并独立一个线程 M。

```
func LockOSThread() {
    if atomic.Load(&newmHandoff.haveTemplateThread) == 0 && GOOS != "plan9" {
        startTemplateThread()
    }
    _g_ := getg()
    _g_.m.lockedExt++
    dolockOSThread()
}

func dolockOSThread() {
    _g_ := getg()
    _g_.m.lockedg.set(_g_)
    _g_.lockedm.set(_g_.m)
}
```

`runtime.dolockOSThread` 会分别设置线程的 `lockedg` 字段和 Goroutine 的 `lockedm` 字段，这两行代码会绑定线程和 Goroutine。

当 Goroutine 完成了特定的操作之后，会调用以下函数 `runtime.UnlockOSThread` 分离 Goroutine 和线程：

```
func UnlockOSThread() {
    _g_ := getg()
    if _g_.m.lockedExt == 0 {
        return
    }
    _g_.m.lockedExt--
    dounlockOSThread()
}

func dounlockOSThread() {
    _g_ := getg()
    if _g_.m.lockedInt != 0 || _g_.m.lockedExt != 0 {
        return
    }
    _g_.m.lockedg = 0
    _g_.lockedm = 0
}
```

函数执行的过程与 `runtime.LockOSThread` 正好相反。在多数的服务中，我们都用不到这一对函数，不过使用 CGO 或者经常与操作系统打交道的读者可能会见到它们的身影。

线程生命周期

Go 语言的运行时会通过 `runtime.startm` 启动线程来执行处理器 P，如果我们在该函数中没能从闲置列表中获取到线程 M 就会调用 `runtime.newm` 创建新的线程：

```
func newm(fn func(), _p_ *p, id int64) {
    mp := allocm(_p_, fn, id)
    mp.nextp.set(_p_)
    mp.sigmask = initSigmask
    ...
    newm1(mp)
}

func newm1(mp *m) {
    if iscgo {
        ...
    }
    newosproc(mp)
}
```

创建新的线程需要使用如下所示的 `runtime.newosproc`，该函数在 Linux 平台上会通过系统调用 `clone` 创建新的操作系统线程，它也是创建线程链路上距离操作系统最近的 Go 语言函数：

```
func newosproc(mp *m) {
    stk := unsafe.Pointer(mp.g0.stack.hi)
    ...
    ret := clone(cloneFlags, stk, unsafe.Pointer(mp), unsafe.Pointer(mp.g0),
unsafe.Pointer(funcPC(mstart)))
    ...
}
```

使用系统调用 `clone` 创建的线程会在线程主动调用 `exit`、或者传入的函数 `runtime.mstart` 返回会主动退出，`runtime.mstart` 会执行调用 `runtime.newm` 时传入的匿名函数 `fn`，到这里也就完成了从线程创建到销毁的整个闭环。

系统监控

很多系统中都有守护进程，它们能够在后台监控系统的运行状态，在出现意外情况时及时响应。系统监控是 Go 语言运行时的重要组成部分，它会每隔一段时间检查 Go 语言运行时，确保程序没有进入异常状态。

Go 语言的系统监控也起到了很重要的作用，它在内部启动了一个不会中止的循环，在循环的内部会轮询网络、抢占长期运行或者处于系统调用的 Goroutine 以及触发垃圾回收，通过这些行为，它能够让系统的运行状态变得更健康。

监控循环

当 Go 语言程序启动时，运行时会在第一个 Goroutine 中调用 `runtime.main` 启动主程序，该函数会在系统栈中创建新的线程：

```
func main() {
    ...
    if GOARCH != "wasm" {
        systemstack(func() {
            newm(sysmon, nil)
        })
    }
    ...
}
```

`runtime.newm` 会创建一个存储待执行函数和处理器的新结构体 `runtime.m`。运行时执行系统监控不需要处理器，系统监控的 Goroutine 会直接在创建的线程上运行：

```
func newm(fn func(), _p_ *p) {
    mp := allocm(_p_, fn)
    mp.nextp.set(_p_)
    mp.sigmask = initSigmask
    ...
    newm1(mp)
}
```

`runtime.newm1` 会调用特定平台的 `runtime.newosproc` 通过系统调用 `clone` 创建一个新的线程并在新的线程中执行 `runtime.mstart`：

```
func newosproc(mp *m) {
    stk := unsafe.Pointer(mp.g0.stack.hi)
    var oset sigset
    sigprocmask(_SIG_SETMASK, &sigset_all, &oset)
    ret := clone(cloneFlags, stk, unsafe.Pointer(mp), unsafe.Pointer(mp.g0),
unsafe.Pointer(funcPC(mstart)))
    sigprocmask(_SIG_SETMASK, &oset, nil)
    ...
}
```

在新创建的线程中，我们会执行存储在 `runtime.m` 中的 `runtime.sysmon` 启动系统监控：

```
func sysmon() {
    sched.nmsys++
```

```
    checkdead()

    lasttrace := int64(0)
    idle := 0
    delay := uint32(0)
    for {
        if idle == 0 {
            delay = 20
        } else if idle > 50 {
            delay *= 2
        }
        if delay > 10*1000 {
            delay = 10 * 1000
        }
        usleep(delay)
        ...
    }
}
```

当运行时刚刚调用上述函数时，会先通过 `runtime.checkdead` 检查是否存在死锁，然后进入核心的监控循环；系统监控在每次循环开始时都会通过 `usleep` 挂起当前线程，该函数的参数是微秒，运行时会遵循以下的规则决定休眠时间：

- 初始的休眠时间是 20μs；
- 最长的休眠时间是 10ms；
- 当系统监控在 50 个循环中都没有唤醒 Goroutine 时，休眠时间在每个循环都会倍增；

当程序趋于稳定之后，系统监控的触发时间就会稳定在 10ms。它除了会检查死锁之外，还会在循环中完成以下的工作：

- 运行计时器 — 获取下一个需要被触发的计时器；
- 轮询网络 — 获取需要处理的到期文件描述符；
- 抢占处理器 — 抢占运行时间较长的或者处于系统调用的 Goroutine；
- 垃圾回收 — 在满足条件时触发垃圾收集回收内存；

我们在这一节中会依次介绍系统监控是如何完成上述几种不同工作的。

检查死锁

系统监控通过 `runtime.checkdead` 检查运行时是否发生了死锁，我们可以将检查死锁的过程分成以下三个步骤：

1. 检查是否存在正在运行的线程；
2. 检查是否存在正在运行的 Goroutine；
3. 检查处理器上是否存在计时器；

该函数首先会检查 Go 语言运行时中正在运行的线程数量，我们通过调度器中的多个字段计算该值的结果：

```
func checkdead() {
    var run0 int32
    run := mcount() - sched.nmidle - sched.nmidlelocked - sched.nmsys
    if run > run0 {
        return
    }
    if run < 0 {
        print("runtime: checkdead: nmidle=", sched.nmidle, " nmidlelocked=",
sched.nmidlelocked, " mcount=", mcount(), " nmsys=", sched.nmsys, "\n")
        throw("checkdead: inconsistent counts")
    }
    ...
}
```

1. `runtime.mcount` 根据下一个待创建的线程 id 和释放的线程数得到系统中存在的线程数；
2. `nmidle` 是处于空闲状态的线程数量；
3. `nmidlelocked` 是处于锁定状态的线程数量；
4. `nmsys` 是处于系统调用的线程数量；

利用上述几个线程相关数据，我们可以得到正在运行的线程数，如果线程数量大于 0，说明当前程序不存在死锁；如果线程数小于 0，说明当前程序的状态不一致；如果线程数等于 0，我们需要进一步检查程序的运行状态：

```
func checkdead() {
    ...
    grunning := 0
    for i := 0; i < len(allgs); i++ {
        gp := allgs[i]
        if isSystemGoroutine(gp, false) {
            continue
        }
        s := readgstatus(gp)
        switch s &^ _Gscan {
        case _Gwaiting, _Gpreempted:
            grunning++
        case _Grunnable, _Grunning, _Gsyscall:
            print("runtime: checkdead: find g ", gp.goid, " in status ", s,
"\n")
            throw("checkdead: runnable g")
        }
    }
    unlock(&allglock)
    if grunning == 0 {
        throw("no goroutines (main called runtime.Goexit) - deadlock!")
    }
    ...
}
```

1. 当存在 Goroutine 处于 `_Grunnable`、`_Grunning` 和 `_Gsyscall` 状态时，意味着程序发生了死锁；
2. 当所有的 Goroutine 都处于 `_Gidle`、`_Gdead` 和 `_Gcopystack` 状态时，意味着主程序调用了 `runtime.goexit`；

当运行时存在等待的 Goroutine 并且不存在正在运行的 Goroutine 时，我们会检查处理器中存在的计时器[:

```
func checkdead() {
    ...
    for _, _p_ := range allp {
        if len(_p_.timers) > 0 {
            return
        }
    }

    throw("all goroutines are asleep - deadlock!")
}
```

如果处理器中存在等待的计时器，那么所有的 Goroutine 陷入休眠状态是合理的，不过如果不存在等待的计时器，运行时会直接报错并退出程序。

运行计时器

系统监控的循环中，我们通过 `runtime.nanotime` 和 `runtime.timeSleepUntil` 获取当前时间和计时器下一次需要唤醒的时间；当前调度器需要执行垃圾回收或者所有处理器都处于闲置状态时，如果没有需要触发的计时器，那么系统监控可以暂时陷入休眠：

```
func sysmon() {
    ...
    for {
        ...
        now := nanotime()
        next, _ := timeSleepUntil()
        if debug.schedtrace <= 0 && (sched.gcwaiting != 0 ||
atomic.Load(&sched.npidle) == uint32(gomaxprocs)) {
            lock(&sched.lock)
            if atomic.Load(&sched.gcwaiting) != 0 || atomic.Load(&sched.npidle)
== uint32(gomaxprocs) {
                if next > now {
                    atomic.Store(&sched.sysmonwait, 1)
                    unlock(&sched.lock)
                    sleep := forcegcperiod / 2
                    if next-now < sleep {
                        sleep = next - now
                    }
                    ...
                    notetsleep(&sched.sysmonnote, sleep)
                    ...
                    now = nanotime()
                    next, _ = timeSleepUntil()
                    lock(&sched.lock)
                    atomic.Store(&sched.sysmonwait, 0)
                    noteclear(&sched.sysmonnote)
                }
                idle = 0
                delay = 20
            }
            unlock(&sched.lock)
        }
        ...
        if next < now {
            startm(nil, false)
        }
```

```
    }
}
```

休眠的时间会依据强制 GC 的周期 `forcegcperiod` 和计时器下次触发的时间确定，`runtime.notesleep` 会使用信号量同步系统监控即将进入休眠的状态。当系统监控被唤醒之后，我们会重新计算当前时间和下一个计时器需要触发的时间、调用 `runtime.noteclear` 通知系统监控被唤醒并重置休眠的间隔。

如果在这之后，我们发现下一个计时器需要触发的时间小于当前时间，这也说明所有的线程可能正在忙于运行 Goroutine，系统监控会启动新的线程来触发计时器，避免计时器的到期时间有较大的偏差。

轮询网络

如果上一次轮询网络已经过去了 10ms，那么系统监控还会在循环中轮询网络，检查是否有待执行的文件描述符：

```
func sysmon() {
    ...
    for {
        ...
        lastpoll := int64(atomic.Load64(&sched.lastpoll))
        if netpollinited() && lastpoll != 0 && lastpoll+10*1000*1000 < now {
            atomic.Cas64(&sched.lastpoll, uint64(lastpoll), uint64(now))
            list := netpoll(0)
            if !list.empty() {
                incidlelocked(-1)
                injectglist(&list)
                incidlelocked(1)
            }
        }
        ...
    }
}
```

上述函数会非阻塞地调用 `runtime.netpoll` 检查待执行的文件描述符并通过 `runtime.injectglist` 将所有处于就绪状态的 Goroutine 加入全局运行队列中：

```
func injectglist(glist *gList) {
    if glist.empty() {
        return
    }
    lock(&sched.lock)
    var n int
    for n = 0; !glist.empty(); n++ {
        gp := glist.pop()
        casgstatus(gp, _Gwaiting, _Grunnable)
        globrunqput(gp)
    }
    unlock(&sched.lock)
    for ; n != 0 && sched.npidle != 0; n-- {
        startm(nil, false)
    }
    *glist = gList{}
}
```

该函数会将所有 Goroutine 的状态从 `_Gwaiting` 切换至 `_Grunnable` 并加入全局运行队列等待运行，如果当前程序中存在空闲的处理器，会通过 `runtime.startm` 启动线程来执行这些任务。

抢占处理器

系统监控会在循环中调用 `runtime.retake` 抢占处于运行或者系统调用中的处理器，该函数会遍历运行时的全局处理器，每个处理器都存储了一个 `runtime.sysmontick`：

```
type sysmontick struct {
    schedtick   uint32
    schedwhen   int64
    syscalltick uint32
    syscallwhen int64
}
```

该结构体中的四个字段分别存储了处理器的调度次数、处理器上次调度时间、系统调用的次数以及系统调用的时间。`runtime.retake` 的循环包含了两种不同的抢占逻辑：

```
func retake(now int64) uint32 {
    n := 0
    for i := 0; i < len(allp); i++ {
        _p_ := allp[i]
        pd := &_p_.sysmontick
        s := _p_.status
        if s == _Prunning || s == _Psyscall {
            t := int64(_p_.schedtick)
            if pd.schedwhen+forcePreemptNS <= now {
                preemptone(_p_)
            }
        }

        if s == _Psyscall {
            if runqempty(_p_) &&
atomic.Load(&sched.nmspinning)+atomic.Load(&sched.npidle) > 0 &&
pd.syscallwhen+10*1000*1000 > now {
                continue
            }
            if atomic.Cas(&_p_.status, s, _Pidle) {
                n++
                _p_.syscalltick++
                handoffp(_p_)
            }
        }
    }
    return uint32(n)
}
```

1. 当处理器处于 `_Prunning` 或者 `_Psyscall` 状态时，如果上一次触发调度的时间已经过去了 10ms，我们会通过 `runtime.preemptone` 抢占当前处理器；
2. 当处理器处于_Psyscall状态时，在满足以下两种情况下会调用`runtime.handoffp`,让出处理器的使用权：
    1. 当处理器的运行队列不为空或者不存在空闲处理器时；
    2. 当系统调用时间超过了 10ms 时；

系统监控通过在循环中抢占处理器来避免同一个 Goroutine 占用线程太长时间造成饥饿问题。

垃圾回收

在最后，系统监控还会决定是否需要触发强制垃圾回收，`runtime.sysmon` 会构建 `runtime.gcTrigger` 并调用 `runtime.gcTrigger.test` 方法判断是否需要触发垃圾回收：

```
func sysmon() {
    ...
    for {
        ...
        if t := (gcTrigger{kind: gcTriggerTime, now: now}); t.test() &&
atomic.Load(&forcegc.idle) != 0 {
            lock(&forcegc.lock)
            forcegc.idle = 0
            var list gList
            list.push(forcegc.g)
            injectglist(&list)
            unlock(&forcegc.lock)
        }
        ...
    }
}
```

如果需要触发垃圾回收，我们会将用于垃圾回收的 Goroutine 加入全局队列，让调度器选择合适的处理器去执行。

内存管理

内存管理一般包含三个不同的组件，分别是用户程序（Mutator）、分配器（Allocator）和收集器（Collector

分配方法

一种是线性分配器（Sequential Allocator，Bump Allocator），另一种是空闲链表分配器（Free-List Allocator）

线性分配（Bump Allocator）是一种高效的内存分配方法，但是有较大的局限性。当我们使用线性分配器时，只需要在内存中维护一个指向内存特定位置的指针，如果用户程序向分配器申请内存，分配器只需要检查剩余的空闲内存、返回分配的内存区域并修改指针在内存中的位置，虽然线性分配器实现为它带来了较快的执行速度以及较低的实现复杂度，但是线性分配器无法在内存被释放时重用内存。

空闲链表分配器（Free-List Allocator）可以重用已经被释放的内存，它在内部会维护一个类似链表的数据结构。当用户程序申请内存时，空闲链表分配器会依次遍历空闲的内存块，找到足够大的内存，然后申请新的资源并修改链表。

空闲链表分配器可以选择不同的策略在链表中的内存块中进行选择，最常见的是以下四种：

- 首次适应（First-Fit）— 从链表头开始遍历，选择第一个大小大于申请内存的内存块；
- 循环首次适应（Next-Fit）— 从上次遍历的结束位置开始遍历，选择第一个大小大于申请内存的内存块；
- 最优适应（Best-Fit）— 从链表头遍历整个链表，选择最合适的内存块；
- 隔离适应（Segregated-Fit）— 将内存分割成多个链表，每个链表中的内存块大小相同，申请内存时先找到满足条件的链表，再从链表中选择合适的内存块；

内存分配

堆上所有的对象都会通过调用 `runtime.newobject` 函数分配内存，该函数会调用 `runtime.mallocgc` 分配指定大小的内存空间，这也是用户程序向堆上申请内存空间的必经函数：

```
func mallocgc(size uintptr, typ *_type, needzero bool) unsafe.Pointer {
    mp := acquirem()
    mp.mallocing = 1

    c := gomcache()
    var x unsafe.Pointer
    noscan := typ == nil || typ.ptrdata == 0
    if size <= maxSmallSize {
        if noscan && size < maxTinySize {
            // 微对象分配
        } else {
            // 小对象分配
        }
    } else {
        // 大对象分配
    }

    publicationBarrier()
    mp.mallocing = 0
    releasem(mp)

    return x
```

```
}
```

三种对象

- 微对象 `(0, 16B)` — 先使用微型分配器，再依次尝试线程缓存、中心缓存和堆分配内存；
- 小对象 `[16B, 32KB]` — 依次尝试使用线程缓存、中心缓存和堆分配内存；
- 大对象 `(32KB, +∞)` — 直接在堆上分配内存；

为了方便自主管理内存，做法便是先向系统申请一块内存，然后将内存切割成小块，通过一定的内存分配算法管理内存。

- arena的大小为512G，为了方便管理把arena区域划分成一个个的page，每个page为8KB,一共有512GB/8KB个页；
- spans区域存放span的指针，每个指针对应一个page，所以span区域的大小为(512GB/8KB)*指针大小8byte = 512M
- bitmap区域大小也是通过arena计算出来，不过主要用于GC。

span

span是用于管理arena页的关键数据结构，每个span中包含1个或多个连续页，为了满足小对象分配，span中的一页会划分更小的粒度，而对于大对象比如超过页大小，则通过多页实现。

```
type mspan struct {
    next *mspan            //链表前向指针，用于将span链接起来
    prev *mspan            //链表前向指针，用于将span链接起来
    startAddr uintptr // 起始地址，也即所管理页的地址
    npages    uintptr // 管理的页数

    nelems uintptr // 块个数，也即有多少个块可供分配

    allocBits  *gcBits //分配位图，每一位代表一个块是否已分配

    allocCount  uint16     // 已分配块的个数
    spanclass   spanClass  // class表中的class ID

    elemsize    uintptr    // class表中的对象大小，也即块大小
}
```

class

跟据对象大小，划分了一系列class，每个class都代表一个固定大小的对象，以及每个span的大小。如下表所示：

```
// class  bytes/obj  bytes/span  objects  waste bytes
//     1          8        8192     1024            0
//     2         16        8192      512            0
//     3         32        8192      256            0
//     4         48        8192      170           32
//     5         64        8192      128            0
...
//    65      28672       57344        2            0
//    66      32768       32768        1            0
```

上表中每列含义如下：

- class： class ID，每个span结构中都有一个class ID, 表示该span可处理的对象类型

- bytes/obj：该class代表对象的字节数
- bytes/span：每个span占用堆的字节数，也即页数*页大小
- objects: 每个span可分配的对象个数，也即（bytes/spans）/（bytes/obj）
- waste bytes: 每个span产生的内存碎片，也即（bytes/spans）%（bytes/obj）

上表可见最大的对象是32K大小，超过32K大小的由特殊的class表示，该class ID为0，每个class只包含一个对象。

总结

Golang内存分配是个相当复杂的过程，其中还掺杂了GC的处理，这里仅仅对其关键数据结构进行了说明，了解其原理而又不至于深陷实现细节。

1. Golang程序启动时申请一大块内存，并划分成spans、bitmap、arena区域
2. arena区域按页划分成一个个小块
3. span管理一个或多个页
4. mcentral管理多个span供线程申请使用
5. mcache作为线程私有资源，资源来源于mcentral

垃圾回收

业界常见的垃圾回收算法有以下几种：

- 引用计数：对每个对象维护一个引用计数，当引用该对象的对象被销毁时，引用计数减1，当引用计数器为0是回收该对象。
    - 优点：对象可以很快的被回收，不会出现内存耗尽或达到某个阀值时才回收。
    - 缺点：不能很好的处理循环引用，而且实时维护引用计数，有也一定的代价。
    - 代表语言：Python、PHP、Swift
- 标记-清除：从根变量开始遍历所有引用的对象，引用的对象标记为"被引用"，没有被标记的进行回收。
    - 优点：解决了引用计数的缺点。
    - 缺点：需要STW，即要暂时停掉程序运行。
    - 代表语言：Golang(其采用三色标记法)
- 分代收集：按照对象生命周期长短划分不同的代空间，生命周期长的放入老年代，而短的放入新生代，不同代有不能的回收算法和回收频率。
    - 优点：回收性能好
    - 缺点：算法复杂
    - 代表语言： JAVA

内存标记(Mark)

span中维护了一个个内存块，并由一个位图allocBits表示每个内存块的分配情况。allocBits记录了每块内存分配情况，而gcmarkBits记录了每块内存标记情况。标记阶段对每块内存进行标记，有对象引用的的内存标记为1(如图中灰色所示)，没有引用到的保持默认为0.

allocBits和gcmarkBits数据结构是完全一样的，标记结束就是内存回收，回收时将allocBits指向gcmarkBits，则代表标记过的才是存活的，gcmarkBits则会在下次标记时重新分配内存，非常的巧妙。

三色标记法

这里的三色，对应了垃圾回收过程中对象的三种状态：

- 灰色：对象还在标记队列中等待
- 黑色：对象已被标记，gcmarkBits对应的位为1（该对象不会在本次GC中被清理）
- 白色：对象未被标记，gcmarkBits对应的位为0（该对象将会在本次GC中被清理）

初始状态下所有对象都是白色的。

接着开始扫描根对象，由于根对象引用了对象A、B,那么A、B变为灰色对象，接下来就开始分析灰色对象，分析A时，A没有引用其他对象很快就转入黑色，B引用了D，则B转入黑色的同时还需要将D转为灰色，进行接下来的分析。

由于D没有引用其他对象，所以D转入黑色。

最终，黑色的对象会被保留下来，白色对象会被回收掉

Stop The World

Golang中的STW（Stop The World）就是停掉所有的goroutine，专心做垃圾回收，待垃圾回收结束后再恢复goroutine。

垃圾回收优化

写屏障(Write Barrier)

STW目的是防止GC扫描时内存变化而停掉goroutine，而写屏障就是让goroutine与GC同时运行的手段。虽然写屏障不能完全消除STW，但是可以大大减少STW的时间。

写屏障类似一种开关，在GC的特定时机开启，开启后指针传递时会把指针标记，即本轮不回收，下次GC时再确定。

GC过程中新分配的内存会被立即标记，用的并不是写屏障技术，也即GC过程中分配的内存不会在本轮GC中回收。

辅助GC(Mutator Assist)

为了防止内存分配过快，在GC执行过程中，如果goroutine需要分配内存，那么这个goroutine会参与一部分GC的工作，即帮助GC做一部分工作，这个机制叫作Mutator Assist。

垃圾回收触发时机

内存分配量达到阀值触发GC

每次内存分配时都会检查当前内存分配量是否已达到阀值，如果达到阀值则立即启动GC。

```
阀值 = 上次GC内存分配量 * 内存增长率
```

内存增长率由环境变量`GOGC`控制，默认为100，即每当内存扩大一倍时启动GC。

定期触发GC

默认情况下，最长2分钟触发一次GC，这个间隔在`src/runtime/proc.go:forcegcperiod`变量中被声明：

```
// forcegcperiod is the maximum time in nanoseconds between garbage
// collections. If we go this long without a garbage collection, one
// is forced to run.
//
// This is a variable for testing purposes. It normally doesn't change.
var forcegcperiod int64 = 2 * 60 * 1e9
```

手动触发

程序代码中也可以使用`runtime.GC()`来手动触发GC。这主要用于GC性能测试和统计。

GC性能优化

GC性能与对象数量负相关，对象越多GC性能越差，对程序影响越大。

所以GC性能优化的思路之一就是减少对象分配个数，比如对象复用或使用大对象组合多个小对象等等。

另外，由于内存逃逸现象，有些隐式的内存分配也会产生，也有可能成为GC的负担。

内存逃逸

通过编译参数-gcflags=-m可以查年编译过程中的逃逸分析：

`go build -gcflags=-m`

指针逃逸

```
package main

type Student struct {
    Name string
    Age  int
}

func StudentRegister(name string, age int) *Student {
    s := new(Student) //局部变量s逃逸到堆

    s.Name = name
    s.Age = age

    return s
}

func main() {
    StudentRegister("Jim", 18)
}

//.\main.go:9: new(Student) escapes to heap
//.\main.go:18: main new(Student) does not escape
```

> 不管是不是指针，在被引用的情况下都会逃逸到堆上

栈空间不足逃逸

一般的栈空间是2M

```
package main

func Slice() {
    s := make([]int, 1000, 1000)

    for index, _ := range s {
        s[index] = index
    }
```

```
}

func main() {
    Slice()
}

//.\main.go:4: Slice make([]int, 1000, 1000) does not escape
```

动态类型逃逸

interface传参

```
package main

import "fmt"

func main() {
    s := "Escape"
    fmt.Println(s)
}
//.\main.go:7: s escapes to heap
//.\main.go:7: main ... argument does not escape
```

闭包引用对象逃逸

> 匿名函数也会逃逸

```
func Fibonacci() func() int {
    a, b := 0, 1
    return func() int {
        a, b = b, a+b
        return a
    }
}

// .\main.go:7: func literal escapes to heap
// .\main.go:7: func literal escapes to heap
```

总结

- 栈上分配内存比在堆中分配内存有更高的效率
- 栈上分配的内存不需要GC处理
- 堆上分配的内存使用完毕会交给GC处理
- 逃逸分析目的是决定内分配地址是栈还是堆
- 逃逸分析在编译阶段完成

> 在有引用的情况下都会逃逸到堆上

代码生成

> 可以把实现的的步骤放在别处
>
> 或者执行一些generate 代码

Go 语言的代码生成机制会读取包含预编译指令的注释，然后执行注释中的命令读取包中的文件，它们将文件解析成抽象语法树并根据语法树生成新的 Go 语言代码和文件，生成的代码会在项目的编译期间与其他代码一起编译和运行。

```
//go:generate command argument...
```

`go generate` 不会被 `go build` 等命令自动执行，该命令需要显式的触发，手动执行该命令时会在文件中扫描上述形式的注释并执行后面的执行命令

```
package main

import "fmt"

//go:generate echo hello
//go:generate go run main.go
//go:generate  echo file=$GOFILE pkg=$GOPACKAGE
func main() {
    fmt.Println("main func")
}

// go generate
// hello
// main func
// file=main.go pkg=main
```

我们可以用go generate来实现String()

```
//go:generate stringer -type=Pill
package painkiller

type Pill int

const (
    Placebo Pill = iota
    Aspirin
    Ibuprofen
    Paracetamol
    Acetaminophen = Paracetamol
)
```

在运行"go generate"命令前，我们需要安装stringer工具，命令如下：

```
$ go get golang.org/x/tools/cmd/stringer
```

- 然后，在painkiller.go所在的目录下面运行"go generate"命令：

```
$ go generate
```

我们会发现当前目录下面生成一个pill_string.go文件，里面实现了我们需要的String()方法，文件内容如下：

```
// Code generated by "stringer -type=Pill"; DO NOT EDIT.

package painkiller

import "fmt"

const _Pill_name = "PlaceboAspirinIbuprofenParacetamol"

var _Pill_index = [...]uint8{0, 7, 14, 23, 34}

func (i Pill) String() string {
    if i < 0 || i >= Pill(len(_Pill_index)-1) {
        return fmt.Sprintf("Pill(%d)", i)
    }
    return _Pill_name[_Pill_index[i]:_Pill_index[i+1]]
}
```

泛型原理

泛型的实现方式，通常有：

- 模版（stenciling）：为每次调用生成代码实例，即便类型参数相同。
- 字典（dictionaries）：单份代码实例，以字典传递类型参数信息。

模版方式性能最佳，但编译时间较长，且生成文件较大。

字典方式代码最少，但复杂度较高，且性能最差。

> Go 泛型实现介于两者之间，一种称作 “GCShape stenciling with Dictionaries” 概念。
>
> 任意指针类型，或具有相同底层类型（underlying type）的类型，属于同一 GCShape 组。

👉每一个唯一的shape会产生一份代码，每份代码携带一个字典，包含了实例化类型的信息（带了shape、dict的标志）。

```
type Tester interface {
    *A | *B
    test()
}

func test[T Tester](a T) {
    a.test()
}
// ------------------------------
func main() {
    var a A = 1
    var b B = 2
    test(&a)
    test(&b)
}
```

编译后可以使用 `go tool objdump -S -s "main\.main" ./file` 反编译查看汇编情况

```
go build -gcflags "-l"
go tool objdump -S -s "main\.main" ./test
TEXT main.main(SB)
func main() {
test(1)
 0x455214 LEAQ main..dict.test[int](SB), AX
 0x45521b MOVL $0x1, BX
 0x455220 CALL main.test[go.shape.int_0](SB)

test(X(2))
 0x455225 LEAQ main..dict.test[main.X.1](SB), AX ; 字典不同。
 0x45522c MOVL $0x2, BX
 0x455231 CALL main.test[go.shape.int_0](SB) ; 函数相同。

test("abc")
 0x455236 LEAQ main..dict.test[string](SB), AX
 0x45523d LEAQ 0xc84c(IP), BX
 0x455244 MOVL $0x3, CX
 0x455249 CALL main.test[go.shape.string_0](SB) ; 新实例。
}
```

> 指针版本存在内存逃逸

性能问题：

1.18 的泛型实现中，大多数只会让代码运行速度变得更慢,可能引发 动态调用、无法内联、内存逃逸 等性能问题

注意使用场景

- 要在数据结构中使用泛型，这也是泛型目前最理想的用例。能实现实现未装箱类型，就能让这些数据结构更易用、运行更快。（以往使用 interface{}实现的泛型）
- **在任何情况下，都不要将接口传递给泛型函数。**如果对性能比较敏感，请保证只在泛型中使用指针、不用接
- 不要重写基于接口的 API 来使用泛型。受制于当前实现，只要继续使用接口，所有使用非空接口的代码都将更简单、并带来更可预测的性能。在方法调用方面，泛型会将指针转化为两次间接接口，再把接口转换成……总之，特别麻烦、也毫无必要。

SSA代码

后端采用了更加形态更加贴近目标程序的IR来表示源代码，这种结构叫着SSA(Static Single Assignment)。

SSA是三地址代码（Three-address Code）的一种变体，SSA要求代码中每个变量只能被赋值一次，并且任何变量在使用之前必须先申明。这是一种极简的代码形式，基于对变量的以上两点约束，编译器可以很安全地对代码进行各种操作及优化，例如如果一个变量被赋值后没有被使用，那么其赋值语句就是可以删掉的。

Go 的编译器后端也使用 SSA 来作为代码的中间表示，函数编译的过程便是先将函数对应的 IR Tree 节点欢转换为 SSA, 再将该 SSA 翻译成目标机器代码。

我们可以通过两个环境变量来 dump 编译器生成的 SSA 结果：

- `GOSSAFUNC`：指定需要 dump 的函数
- `GOSSADIR`：指定结果文件的目录，默认为当前目录

```
package main

type T struct {
    Name string
}

func (this *T) getName() string {
    if this == nil {
        return "<nil>"
    }
    return this.Name
}

func sum(i, j int) int {
    return i + j + 1
}
```

运行命令：`GOSSAFUNC="sum" go build main.go`：

> dumped SSA to ./ssa.html
>
> runtime.mainmain·f: function main is undeclared in the main package

go编译器指令

编译指令的语法是一行特殊的注释，关键字//go开始(之间没有空格), 在cmd/compilex,很多还未列出如//go:build、Cgo、//line等需要详细看看

//go:linkname

该指令指示编译器使用 `importpath.name` 作为源代码中声明为 `localname` 的变量或函数的目标文件符号名称。但是由于这个伪指令，可以破坏类型系统和包模块化。因此只有引用了 unsafe 包才可以使用

time/time.go

```
...
func now() (sec int64, nsec int32, mono int64)
```

runtime/timestub.go

```
import _ "unsafe" // for go:linkname

//go:linkname time_now time.now
func time_now() (sec int64, nsec int32, mono int64) {
    sec, nsec = walltime()
    return sec, nsec, nanotime() - startNano
}
```

可以看到 `time.now`，它并没有具体的实现。如果你初看可能会懵逼。这时候建议你全局搜索一下源码，你就会发现其实现在 `runtime.time_now` 中

简单来讲，就是 `importpath.name` 是 `localname` 的符号别名，编译器实际上会调用 `localname` 。

//go:noescape

该指令指定下一个有声明但没有主体（意味着实现有可能不是 Go）的函数，不允许编译器对其做逃逸分析。

一般情况下，该指令用于内存分配优化。因为编译器默认会进行逃逸分析，会通过规则判定一个变量是分配到堆上还是栈上。

但凡事有意外，一些函数虽然逃逸分析其是存放到堆上。但是对于我们来说，它是特别的。**我们就可以使用 `go:noescape` 指令强制要求编译器将其分配到函数栈上。**

这个指令告诉编译器 下面的func没有任何参数escape。编译器将会跳过对func参数的检查。

```
// memmove copies n bytes from "from" to "to".
// in memmove_*.s
//go:noescape
func memmove(to, from unsafe.Pointer, n uintptr)
```

//go:nosplit

该指令指定文件中声明的下一个函数不得包含堆栈溢出检查。简单来讲，就是这个**函数跳过堆栈溢出的检查**

```
//go:nosplit
func key32(p *uintptr) *uint32 {
    return (*uint32)(unsafe.Pointer(p))
}
```

//go:norace

该指令表示禁止进行竞态检测。而另外一种常见的形式就是在启动时执行 `go run -race`，能够检测应用程序中是否存在双向的数据竞争。非常有用

```
var sum int

func main() {
    go add()
    go add()
}

//go:norace
func add() {
    sum++
}
```

//go:noinline

该指令表示该函数禁止进行内联

```
//go:noinline
func unexportedPanicForTesting(b []byte, i int) byte {
    return b[i]
}
```

我们观察一下这个案例，是直接通过索引取值，逻辑比较简单。如果不加上 `go:noinline` 的话，就会出现编译器对其进行内联优化

显然，内联有好有坏。该指令就是提供这一特殊处理

//go:notinheap

该指令常用于类型声明，它表示这个类型不允许从 GC 堆上进行申请内存。在运行时中常用其来做较低层次的内部结构，避免调度器和内存分配中的写屏障。能够提高性能

//go:nowritebarrierrec

该指令表示编译器遇到写屏障时就会产生一个错误，并且允许递归。也就是这个函数调用的其他函数如果有写屏障也会报错。简单来讲，就是针对写屏障的处理，防止其死循环

```
//go:nowritebarrierrec
func gcFlushBgCredit(scanWork int64) {
    ...
}
```

//go:yeswritebarrierrec

该指令与 `go:nowritebarrierrec` 相对，在标注 `go:nowritebarrierrec` 指令的函数上，遇到写屏障会产生错误。而当编译器遇到 `go:yeswritebarrierrec` 指令时将会停止

```
//go:yeswritebarrierrec
func gchelper() {
    ...
}
```

//go:uintptrkeepalive

//go:uintptrescapes

//go:uintptrescapes 指令后面必须跟一个函数声明。 它指定函数的 uintptr 参数可能是已转换为 uintptr 的指针值，并且必须在堆上并在调用期间保持活动状态，即使仅从类型来看，在调用期间似乎不再需要该对象 通话。 从指针到 uintptr 的转换必须出现在对该函数的任何调用的参数列表中。 该指令对于某些低级系统调用实现是必需的，否则应避免使用。

```
//go:uintptrescapes
func F1(a uintptr) {} // ERROR "escaping uintptr"
```

//go:embed

指令在编译时使用从包目录或子目录中读取的文件内容来初始化字符串、[]byte 或 FS 类型的变量。

```
import "embed"

//go:embed hello.txt
var f embed.FS
data, _ := f.ReadFile("hello.txt")
print(string(data))
```

Delve、GDB、LLDB调试（了解）

目前 Go 语言支持 GDB、LLDB 和 Delve 几种调试器。其中 GDB 是最早支持的调试工具，LLDB 是 macOS 系统推荐的标准调试工具。但是 GDB 和 LLDB 对 Go 语言的专有特性都缺乏很大支持，而只有 Delve 是专门为 Go 语言设计开发的调试工具。

需要关闭内联调试

```
# 关闭内联优化，方便调试
go build -gcflags "-N -l" demo.go

gdb demo
lldb demo
delve demo
```

delve参数

```
(dlv) help
The following commands are available:
    args ------------------------ Print function arguments.
    break (alias: b) ------------ Sets a breakpoint.
    breakpoints (alias: bp) ----- Print out info for active breakpoints.
    clear ----------------------- Deletes breakpoint.
    clearall -------------------- Deletes multiple breakpoints.
    condition (alias: cond) ----- Set breakpoint condition.
    config ---------------------- Changes configuration parameters.
    continue (alias: c) --------- Run until breakpoint or program termination.
    disassemble (alias: disass) - Disassembler.
    down ------------------------ Move the current frame down.
    exit (alias: quit | q) ------ Exit the debugger.
    frame ----------------------- Set the current frame, or execute command...
    funcs ----------------------- Print list of functions.
    goroutine ------------------- Shows or changes current goroutine
    goroutines ------------------ List program goroutines.
    help (alias: h) ------------- Prints the help message.
    list (alias: ls | l) -------- Show source code.
    locals ---------------------- Print local variables.
    next (alias: n) ------------- Step over to next source line.
    on -------------------------- Executes a command when a breakpoint is hit.
    print (alias: p) ------------ Evaluate an expression.
    regs ------------------------ Print contents of CPU registers.
    restart (alias: r) ---------- Restart process.
    set ------------------------- Changes the value of a variable.
    source ---------------------- Executes a file containing a list of delve...
    sources --------------------- Print list of source files.
    stack (alias: bt) ----------- Print stack trace.
    step (alias: s) ------------- Single step through program.
    step-instruction (alias: si)  Single step a single cpu instruction.
    stepout --------------------- Step out of the current function.
    thread (alias: tr) ---------- Switch to the specified thread.
    threads --------------------- Print out info for every traced thread.
    trace (alias: t) ------------ Set tracepoint.
    types ----------------------- Print list of types
    up -------------------------- Move the current frame up.
    vars ------------------------ Print package variables.
```

```
    whatis ---------------------- Prints type of an expression.
Type help followed by a command for full documentation.
(dlv)
```

要熟练使用 GDB，你得熟悉的掌握它的指令，这里列举一下

- `r`：run，执行程序
- `n`：next，下一步，不进入函数
- `s`：step，下一步，会进入函数
- `b`：breakponit，设置断点
- `l`：list，查看源码
- `c`：continue，继续执行到下一断点
- `bt`：backtrace，查看当前调用栈
- `p`：print，打印查看变量
- `q`：quit，退出 GDB
- `whatis`：查看对象类型
- `info breakpoints`：查看所有的断点
- `info locals`：查看局部变量
- `info args`：查看函数的参数值及要返回的变量值
- `info frame`：堆栈帧信息
- `info goroutines`：查看 goroutines 信息。在使用前，需要注意先执行 source /usr/local/go/src/runtime/runtime-gdb.py
- `goroutine 1 bt`：查看指定序号的 goroutine 调用堆栈
- 回车：重复执行上一次操作

编译原理（了解）

go build 过程

go build 参数说明

```
-a   将命令源码文件与库源码文件全部重新构建，即使是最新的
-n   把编译期间涉及的命令全部打印出来，但不会真的执行，非常方便我们学习
-race    开启竞态条件的检测，支持的平台有限制
-x   打印编译期间用到的命名，它与 -n 的区别是，它不仅打印还会执行
```

`go build -n main.go`

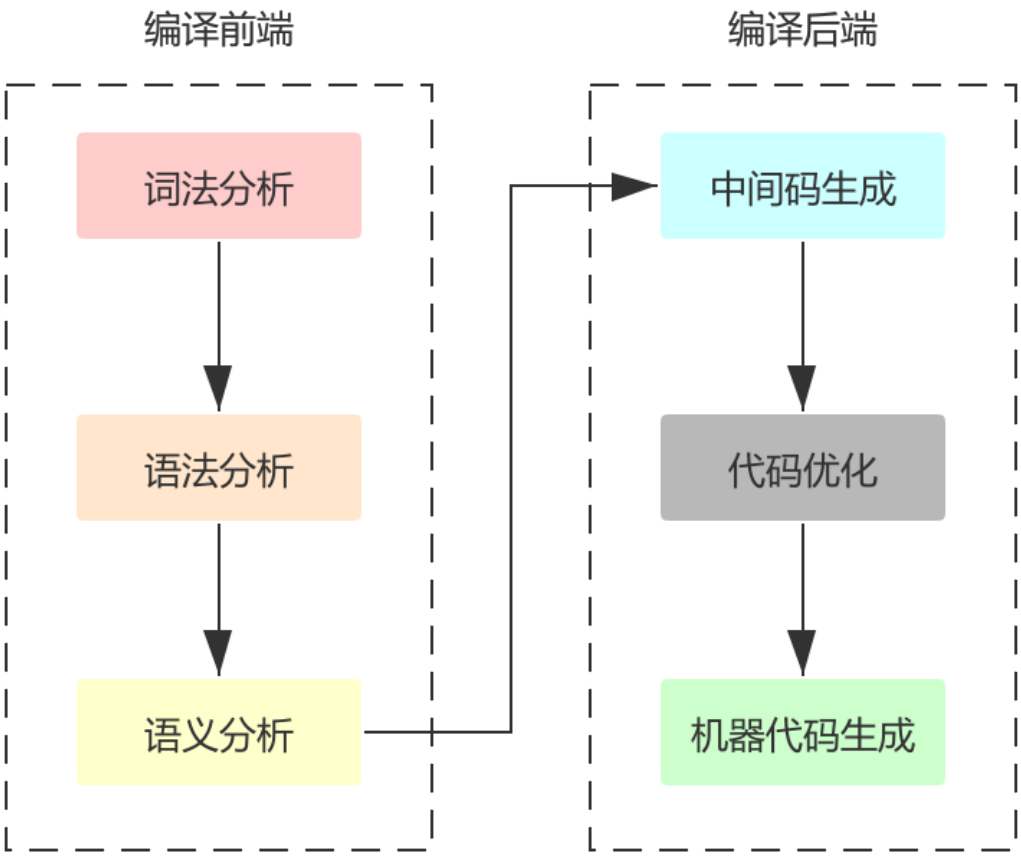
```
E:\SomeFile\gospace\helloworld>go build -n main.go
...
"C:\\Program Files\\Go\\pkg\\tool\\windows_amd64\\compile.exe" -o
"$WORK\\b001\\_pkg_.a" -trimpath "$WORK\\b001=>" -p main -complete -buildid
v3BL3MHm16Q3kjspXXCg/v3BL3MHm16Q3kjspXXCg
 -goversion go1.18.2 -c=4 -nolocalimports -importcfg "$WORK\\b001\\importcfg" -
pack
...
"C:\\Program Files\\Go\\pkg\\tool\\windows_amd64\\buildid.exe" -w
"$WORK\\b001\\_pkg_.a" # internal
...
"C:\\Program Files\\Go\\pkg\\tool\\windows_amd64\\link.exe" -o
"$WORK\\b001\\exe\\a.out.exe" -importcfg "$WORK\\b001\\importcfg.link" -
buildmode=pie -buildid=qEvHouU39HqKxAChmahv/v3BL
3MHm16Q3kjspXXCg/v3BL3MHm16Q3kjspXXCg/qEvHouU39HqKxAChmahv -extld=gcc
"$WORK\\b001\\_pkg_.a"
```

这一部分是编译的核心，通过 `compile`、 `buildid`、 `link` 三个命令会编译出可执行文件 `a.out`。

然后通过 `mv` 更换成最终名字

流程图

Go 语言编译器的源代码在 `src/cmd/compile` 目录中，目录下的文件共同组成了 Go 语言的编译器。



编译器分前后端

- 前端一般承担着词法分析、语法分析、语义分析（类型检查）。
- 后端主要负责 中间码生成、目标代码优化，机器码生成。

语法分析

源代码在计算机眼里其实是一团乱麻，一个由字符组成的、无法被理解的字符串，所有的字符在计算器看来并没有什么区别。词法分析简单来说就是将我们写的源代码翻译成 `Token`

```
package main

import (
    "fmt"
    "go/scanner"
    "go/token"
)

func main() {
    src := []byte(`
package main

import "fmt"

func main() {
    fmt.Println("Hello, bytedance!")
}
`)
```

```
    var s scanner.Scanner
    fset := token.NewFileSet()
    file := fset.AddFile("", fset.Base(), len(src))
    s.Init(file, src, nil, 0)

    for {
        pos, tok, lit := s.Scan()
        fmt.Printf("%-6s%-8s%q\n", fset.Position(pos), tok, lit)

        if tok == token.EOF {
            break
        }
    }
}
```

输出

```
2:1   package "package"
2:9   IDENT   "main"
2:13  ;       "\n"
4:1   import  "import"
4:8   STRING  "\"fmt\""
4:13  ;       "\n"
6:1   func    "func"
6:6   IDENT   "main"
6:10  (       ""
6:11  )       ""
6:13  {       ""
7:5   IDENT   "fmt"
7:8   .       ""
7:9   IDENT   "Println"
7:16  (       ""
7:17  STRING  "\"Hello, bytedance!\""
7:36  )       ""
7:37  ;       "\n"
8:1   }       ""
8:2   ;       "\n"
8:3   EOF     ""
```

语法分析（抽象语法树）

语法分析是拿到Token，它将作为语法分析器的输入。然后经过处理后生成 `AST` 结构作为输出。

所谓的语法分析就是将 `Token` 转化为可识别的程序语法结构，而 `AST` 就是这个语法的抽象表示。构造这颗树有两种方法。

1、自上而下 这种方式会首先构造根节点，然后就开始扫描 `Token`，遇到 `STRING` 或者其它类型就知道这是在进行类型申明，`func` 就表示是函数申明。就这样一直扫描直到程序结束。

2、自下而上 这种是与上一种方式相反的，它先构造子树，然后再组装成一颗完整的树。

```
package main

import (
    "go/ast"
```

```
    "go/parser"
    "go/token"
    "log"
)

func main() {
    src := []byte(`
package main

import "fmt"

func main() {
    fmt.Println("Hello, bytedance!")
}
`)

    fset := token.NewFileSet()

    file, err := parser.ParseFile(fset, "", src, 0)
    if err != nil {
        log.Fatal(err)
    }

    ast.Print(fset, file)
}
```

ATS

```
 0  *ast.File {
     1  .  Package: 2:1
     2  .  Name: *ast.Ident {
     3  .  .  NamePos: 2:9
     4  .  .  Name: "main"
     5  .  }
     6  .  Decls: []ast.Decl (len = 2) {
     7  .  .  0: *ast.GenDecl {
     8  .  .  .  TokPos: 4:1
     9  .  .  .  Tok: import
    10  .  .  .  Lparen: -
    11  .  .  .  Specs: []ast.Spec (len = 1) {
    12  .  .  .  .  0: *ast.ImportSpec {
    13  .  .  .  .  .  Path: *ast.BasicLit {
    14  .  .  .  .  .  .  ValuePos: 4:8
    15  .  .  .  .  .  .  Kind: STRING
    16  .  .  .  .  .  .  Value: "\"fmt\""
    17  .  .  .  .  .  }
    18  .  .  .  .  .  EndPos: -
    19  .  .  .  .  }
    20  .  .  .  }
    21  .  .  .  Rparen: -
    22  .  .  }
    23  .  .  1: *ast.FuncDecl {
    24  .  .  .  Name: *ast.Ident {
    25  .  .  .  .  NamePos: 6:6
    26  .  .  .  .  Name: "main"
    27  .  .  .  .  Obj: *ast.Object {
    28  .  .  .  .  .  Kind: func
```

```
29  .  .  .  .  .  Name: "main"
30  .  .  .  .  .  Decl: *(obj @ 23)
31  .  .  .  .  }
32  .  .  .  }
33  .  .  .  Type: *ast.FuncType {
34  .  .  .  .  Func: 6:1
35  .  .  .  .  Params: *ast.FieldList {
36  .  .  .  .  .  Opening: 6:10
37  .  .  .  .  .  Closing: 6:11
38  .  .  .  .  }
39  .  .  .  }
40  .  .  .  Body: *ast.BlockStmt {
41  .  .  .  .  Lbrace: 6:13
42  .  .  .  .  List: []ast.Stmt (len = 1) {
43  .  .  .  .  .  0: *ast.ExprStmt {
44  .  .  .  .  .  .  X: *ast.CallExpr {
45  .  .  .  .  .  .  .  Fun: *ast.SelectorExpr {
46  .  .  .  .  .  .  .  .  X: *ast.Ident {
47  .  .  .  .  .  .  .  .  .  NamePos: 7:5
48  .  .  .  .  .  .  .  .  .  Name: "fmt"
49  .  .  .  .  .  .  .  .  }
50  .  .  .  .  .  .  .  .  Sel: *ast.Ident {
51  .  .  .  .  .  .  .  .  .  NamePos: 7:9
52  .  .  .  .  .  .  .  .  .  Name: "Println"
53  .  .  .  .  .  .  .  .  }
54  .  .  .  .  .  .  .  }
55  .  .  .  .  .  .  .  Lparen: 7:16
56  .  .  .  .  .  .  .  Args: []ast.Expr (len = 1) {
57  .  .  .  .  .  .  .  .  0: *ast.BasicLit {
58  .  .  .  .  .  .  .  .  .  ValuePos: 7:17
59  .  .  .  .  .  .  .  .  .  Kind: STRING
60  .  .  .  .  .  .  .  .  .  Value: "\"Hello, bytedance!\""
61  .  .  .  .  .  .  .  .  }
62  .  .  .  .  .  .  .  }
63  .  .  .  .  .  .  .  Ellipsis: -
64  .  .  .  .  .  .  .  Rparen: 7:36
65  .  .  .  .  .  .  }
66  .  .  .  .  .  }
67  .  .  .  .  }
68  .  .  .  .  Rbrace: 8:1
69  .  .  .  }
70  .  .  }
71  .  }
72  .  Scope: *ast.Scope {
73  .  .  Objects: map[string]*ast.Object (len = 1) {
74  .  .  .  "main": *(obj @ 27)
75  .  .  }
76  .  }
77  .  Imports: []*ast.ImportSpec (len = 1) {
78  .  .  0: *(obj @ 12)
79  .  }
80  .  Unresolved: []*ast.Ident (len = 1) {
81  .  .  0: *(obj @ 46)
82  .  }
83  }
```

> gofmt的原理就是先转抽象语法树，再转回代码

语义分析（类型检查）

`AST` 生成后，语义分析将使用它作为输入，并且的有一些相关的操作也会直接在这颗树上进行改写。

首先就是 `Golang` 文档中提到的会进行类型检查，还有类型推断，查看类型是否匹配，是否进行隐式转化（go 没有隐式转化）。

第一步是进行名称检查和类型推断，签定每个对象所属的标识符，以及每个表达式具有什么类型。类型检查也还有一些其它的检查要做，像"声明未使用"以及确定函数是否中止。

我们常常在 *debug* 代码的时候，需要禁止内联，其实就是操作的这个阶段。

```
# 编译的时候禁止内联
go build -gcflags '-N -l'

-N 禁止编译优化
-l 禁止内联,禁止内联也可以一定程度上减小可执行程序大小
```

中间码生成（适应平台，重用代码）

然已经拿到 AST，机器运行需要的又是二进制。为什么不直接翻译成二进制呢？其实到目前为止从技术上来说已经完全没有问题了。

但是，我们有各种各样的操作系统，有不同的 CPU 类型，每一种的位数可能不同；寄存器能够使用的指令也不同，像是复杂指令集与精简指令集等；在进行各个平台的兼容之前，我们还需要替换一些底层函数，比如我们使用 make 来初始化 slice，此时会根据传入的类型替换为：`makeslice64` 或者 `makeslice`。当然还有像 painc、channel 等等函数的替换也会在中间码生成过程中进行替换。

中间码存在的另外一个价值是提升后端编译的重用，比如我们定义好了一套中间码应该是长什么样子，那么后端机器码生成就是相对固定的。每一种语言只需要完成自己的编译器前端工作即可。这也是大家可以看到现在开发一门新语言速度比较快的原因。编译是绝大部分都可以重复使用的。

而且为了接下来的优化工作，中间代码存在具有非凡的意义。因为有那么多的平台，如果有中间码我们可以把一些共性的优化都放到这里。

中间码也是有多种格式的，像 `Golang` 使用的就是 SSA 特性的中间码(IR)，这种形式的中间码，最重要的一个特性就是最在使用变量之前总是定义变量，并且每个变量只分配一次。

代码优化

在 go 的编译文档中，我并没找到独立的一步进行代码的优化。不过根据我们上面的分析，可以看到其实代码优化过程遍布编译器的每一个阶段。大家都会力所能及的做些事情。

通常我们除了用高效代码替换低效的之外，还有如下的一些处理：

- 并行性，充分利用现在多核计算机的特性
- 流水线，cpu 有时候在处理 a 指令的时候，还能同时处理 b 指令
- 指令的选择，为了让 cpu 完成某些操作，需要使用指令，但是不同的指令效率有非常大的差别，这里会进行指令优化
- 利用寄存器与高速缓存，我们都知道 cpu 从寄存器取是最快的，从高速缓存取次之。这里会进行充分的利用

机器码生成

经过优化后的中间代码，首先会在这个阶段被转化为汇编代码（Plan9），而汇编语言仅仅是机器码的文本表示，机器还不能真的去执行它。所以这个阶段会调用汇编器，汇编器会根据我们在执行编译时设置的架构，调用对应代码来生成目标机器码。

这里比有意思的是，`Golang` 总说自己的汇编器是跨平台的。其实他也是写了多分代码来翻译最终的机器码。因为在入口的时候他会根据我们所设置的 `GOARCH=xxx` 参数来进行初始化处理，然后最终调用对应架构编写的特定方法来生成机器码。这种上层逻辑一致，底层逻辑不一致的处理方式非常通用，非常值得我们学习。我们简单来一下这个处理。

首先看入口函数 `cmd/compile/main.go:main()`

```
var archInits = map[string]func(*gc.Arch){
    "386":      x86.Init,
    "amd64":    amd64.Init,
    "amd64p32": amd64.Init,
    "arm":      arm.Init,
    "arm64":    arm64.Init,
    "mips":     mips.Init,
    "mipsle":   mips.Init,
    "mips64":   mips64.Init,
    "mips64le": mips64.Init,
    "ppc64":    ppc64.Init,
    "ppc64le":  ppc64.Init,
    "s390x":    s390x.Init,
    "wasm":     wasm.Init,
}

func main() {
    // 从上面的map根据参数选择对应架构的处理
    archInit, ok := archInits[objabi.GOARCH]
    if !ok {
        ......
    }
    // 把对应cpu架构的对应传到内部去
    gc.Main(archInit)
}
```

然后在 `cmd/internal/obj/plist.go` 中调用对应架构的方法进行处理

```
func Flushplist(ctxt *Link, plist *Plist, newprog ProgAlloc, myimportpath string)
{
    ... ...
    for _, s := range text {
        mkfwd(s)
        linkpatch(ctxt, s, newprog)
        // 对应架构的方法进行自己的机器码翻译
        ctxt.Arch.Preprocess(ctxt, s, newprog)
        ctxt.Arch.Assemble(ctxt, s, newprog)

        linkpcln(ctxt, s)
        ctxt.populateDWARF(plist.Curfn, s, myimportpath)
    }
}
```

整个过程下来，可以看到编译器后端有很多工作需要做的，你需要对某一个指令集、cpu 的架构了解，才能正确的进行翻译机器码。同时不能仅仅是正确，一个语言的效率是高还是低，也在很大程度上取决于编译器后端的优化。特别是即将进入 AI 时代，越来越多的芯片厂商诞生，我估计以后对这方面人才的需求会变得越来越旺盛。

Go汇编（了解）

Go 语言使用的 Plan 9 汇编。

以输出`hello，world`为例，Plan 9 汇编代码的整体逻辑同样是分了 Data 段、Text 段，同样是用 mov 等指令给寄存器赋值。

```
#include "textflag.h"

DATA  msg<>+0x00(SB)/8, $"Hello, W"
DATA  msg<>+0x08(SB)/8, $"orld!\n"
GLOBL msg<>(SB),NOPTR,$16

TEXT ·PrintMe(SB), NOSPLIT, $0
    MOVL    $(0x2000000+4), AX   // write 系统调用数字编号 4
    MOVQ    $1, DI                 // 第 1 个参数 fd
    LEAQ    msg<>(SB), SI       // 第 2 个参数 buffer 指针地址
    MOVL    $16, DX                // 第 3 个参数 count
    SYSCALL
    RET
```

go获取汇编代码

```
go tool compile -S main.go
# or
go build -gcflags -S main.go
```

CPU、内存与寄存器

汇编主要是跟 CPU 和内存打交道，CPU 本身只负责运算，不负责存储，数据存储一般都是放在内存中。

我们知道 CPU 的运算速度远高于内存的读写速度，为了 CPU 不被内存读写拖后腿，CPU 内部引入**一级缓存、二级缓存和寄存器**的概念，这些资源都非常宝贵，**寄存器可以认为是在 CPU 内可以存储非常少量数据的超高速的存储单元**。因为寄存器个数有限且非常重要，每个寄存器都有自己的名字，最常用的有以下。

```
%EAX %EBX %ECX %EDX %EDI %ESI %EBP %ESP
```

通用寄存器

plan 9 汇编的通用寄存器包括以下寄存器。

`AX BX CX DX DI SI BP SP R8 R9 R10 R11 R12 R13 R14 PC`

plan9 中使用寄存器**不需要带 r 或 e 的前缀**，例如 rax，只要写 AX 就可以了。

```
eax->AX
ebx->BX
ecx->CX
```

plan9 汇编语言提供了如下映射，在汇编语言中直接引用就可使用物理寄存器了。

amd64	rax	rbx	rcx	rdx	rdi	rsi	rbp	rsp	r8	r9	r10	r11	r12	r13	r14	rip
Plan9	AX	BX	CX	DX	DI	SI	BP	SP	R8	R9	R10	R11	R12	R13	R14	PC

伪寄存器

伪寄存器并不是真正的寄存器，而是虚拟寄存器（由工具链维护），例如帧指针。

Go 汇编引入了四个伪寄存器，这四个伪寄存器非常重要：

- FP(Frame pointer)-帧指针：用来访问函数的参数
- PC(Program counter)-程序计数器：用于分支和跳转
- SB(Static base pointer)-静态基础指针：一般用于声明函数或者全局变量
- SP(Stack pointer)-栈指针：指向当前栈帧的局部变量的开始位置，一般用来引用函数的局部变量

所有用户定义的符号都作为偏移量写入伪寄存器 FP 和 SB。

汇编代码中需要表示用户定义的符号(变量)时，可以通过 SP 与偏移还有变量名的组合，比如`x-8(SP)`，因为 SP 指向的是栈顶，所以偏移值都是负的，`x`则表示变量名

汇编程序指令

全局数据符号由一系列以 DATA 指令起始和一个 GLOBL 指令定义。

宏

在汇编文件中可以定义、引用宏。通过`#define get_tls(r)  MOVQ TLS, r`类似语句来定义一个宏，语法结构与C语言类似；通过`#include "textflag.h"`类似语句来引用一个外部宏定义文件。

```
type vdsoVersionKey struct {
    version string
    verHash uint32
}

#define vdsoVersionKey__size 24
#define vdsoVersionKey_version 0
#define vdsoVersionKey_verHash 16

MOVQ vdsoVersionKey_version(DX) AX
MOVQ (vdsoVersionKey_version+vdsoVersionKey_verHash)(DX) AX
```

汇编常量

```
$1           # 十进制
$0xf4f8fcff  # 十六进制
$1.5         # 浮点数
$'a'         # 字符
$"abcd"      # 字符串

$2+2       # 常量表达式
$3&1<<2    # == $4
$(3&1)<<2 # == $4
```

DATA - 初始化变量

DATA 命令用于初始化变量，格式`DATA symbol+offset(SB)/width, value`,在给定的 offset 和 width 处初始化该符号的内存为 value。 DATA 必须使用增加的偏移量来写入给定符号的指令。

```
DATA  msg<>+0x00(SB)/8, $"Hello, W"
DATA  msg<>+0x08(SB)/8, $"orld!\n"
```

GLOBL - 全局变量

GLOBL 指令声明符号是全局的。参数是可选标志，并且数据的大小被声明为全局， **除非 DATA 指令已初始化，否则初始值将全为零。**该 GLOBL 指令必须遵循任何相应的 DATA 指令。

```
GLOBL msg<>(SB),NOPTR,$16
```

注意到 msg 后面有一个`<>`，这表示这个全局变量只在当前文件中可以被访问

```
GLOBL divtab<>(SB), RODATA, $64
```

这条给变量`divtab<>`加了一个 flag `RODATA` ，表示里边存的是只读变量，最后的`$64`表示的是这个变量占用了 64 字节的空间。

TEXT - 函数

TEXT用于函数定义，表示是汇编中的 .text 分段

```
TEXT pkgname·funname(SB),flag,$framesize-argsize
```

`pkgname`:可以省略，最好省略。不然修改包名还要级联修改;

`funname`: 声明的函数名

`flag`: 标志位，如 `NOSPLIT`，我们知道 Go Runtime 会追踪每个 stack 的使用情况，然后动态自增。而`NOSPLIT`标志位禁止检查，节省开销，但是写程序的人要保证这个函数是安全的。

- NOPROF = 1
  （对于TEXT项目。）不要分析标记的功能。此标志已弃用。
- DUPOK = 2
  在单个二进制文件中具有此符号的多个实例是合法的。链接器将选择要使用的重复项之一。
- NOSPLIT = 4
  （对于TEXT项。）请勿插入序言以检查是否必须拆分堆栈。例程的框架及其所调用的任何内容都必须适合堆栈段顶部的备用空间。用于保护例程，例如堆栈拆分代码本身。
- RODATA = 8
  （对于DATA和GLOBL项目。）将此数据放在只读部分中。
- NOPTR = 16
  （对于DATA和GLOBL项目。）此数据不包含任何指针，因此不需要由垃圾收集器进行扫描。
  包裹= 32
  （对于TEXT项目。）这是包装函数，不应视为禁用恢复。
- NEEDCTXT = 64
  （对于TEXT项目。）此函数是一个闭包，因此它使用其传入的上下文寄存器。

`framesize`: 函数栈帧大小 = 局部变量 + 调用其它函数参数空间的总大小

`argsize`：一些参考资料说这里是 **参数+返回值**大小

```
TEXT runtime·profileloop(SB),NOSPLIT,$8  # $8 指运行时占8字节空间(栈帧大小)
    MOVQ    $runtime·profileloop1(SB), CX
    MOVQ    CX, 0(SP)
    CALL    runtime·externalthreadhandler(SB)
    RET
```

上边整段汇编代码称为一个`TEXT block`，`runtime.profileloop(SB)`后边有一个`NOSPLIT` flag，紧随其后的`$8`表示`frame size`

> 通常`frame size`的构成都是形如`$24-8`(中间的`-`只起到分隔的作用)，表示的是这个`TEXT block`运行的时候需要占用 24 字节空间，参数和返回值要额外占用 8 字节空间（这 8 字节占用的是调用方栈帧里的空间）

常见操作指令

指令有几大类，**一类是用于数据移动的**，**一类是用于跳转的**（无条件跳转，有条件跳转）。**一类是用于逻辑运算和算术运算。**

指令后缀Q说明是64位上的汇编指令

[指令查询]

MOVQ - 搬运长度

`MOV`指令：其后缀表示搬运长度，`$NUM`表示具体的数字，如下面例子

```
MOVB $1, DI          // 1 byte,  DI=1
MOVW $0x10, BX       // 2 bytes, BX=10
MOVD $1, DX          // 4 bytes, DX=1
MOVQ $-10, AX      // 8 bytes, AX=-10
```

MOV 指令有有好几种后缀 MOVB MOVW MOVL MOVQ 分别对应的是 1 字节 、2 字节 、4 字节、8 字节

LEAQ - 将有效地址加载到指定的地址寄存器中

```
// ret+24(FP) 这代表了第三个函数参数，是个地址
LEAQ    ret+24(FP), AX  // 把 ret+24(FP) 地址移到 AX 寄存器中
```

ADDQ - 减法

`ADD`加法计算指令

```
ADDQ  AX, BX   // BX += AX
# 栈空间：高地址向低地址
ADDQ $0x18, SP // 对 SP 做加法，清除函数栈帧
```

SUBQ - 减法

`SUB` 减法计算指令

```
SUBQ  AX, BX   # BX -= AX
# 栈空间：高地址向低地址
SUBQ $0x18, SP # 对 SP 做减法，为函数分配函数栈帧
```

IMULQ - 无符号乘法

```
IMULQ AX, BX   # BX *= AX
```

IDIVQ - 无符号除法

CMP - 比较

AND，OR，XOR - 位运算

JMP - 无条件跳转

`JMP` 指令**无条件**跳转到目标地址，该地址用代码标号来标识，并被汇编器转换为偏移量

```
JMP addr   // 跳转到地址，地址可为代码中的地址，不过实际上手写不会出现这种东西
JMP label  // 跳转到标签，可以跳转到同一函数内的标签位置
JMP 2(PC)  // 以当前指令为基础，向前/后跳转 x 行
JMP -2(PC) // 同上
```

JZ，JLS - 有条件跳转

```
# 有条件跳转
JZ target // 如果 zero flag 被 set 过，则跳转
JLS num     // 如果上一行的比较结果，左边小于右边则执行跳到 num 地址处
```

CALL - 调用函数

```
CALL runtime.printnl(SB)
```

RET - 返回

return

leave - 恢复调用者者栈指针

寻址模式

汇编语言的一个很重要的概念就是它的寻址模式，Plan 9 汇编也不例外，它支持如下寻址模式：

```
R0              数据寄存器
A0              地址寄存器
F0              浮点寄存器
CAAR, CACR, 等   特殊名字
```

```
$con            常量
$fcon           浮点数常量
name+o(SB)      外部符号
name<>+o(SB)    局部符号
name+o(SP)      自动符号
name+o(FP)      实际参数
$name+o(SB)     外部地址
$name<>+o(SB)   局部地址
(A0)+           间接后增量
-(A0)           间接前增量
o(A0)
o()(R0.s)

symbol+offset(SP) 引用函数的局部变量，offset 的合法取值是 [-framesize, 0)
    局部变量都是 8 字节，那么第一个局部变量就可以用 localvar0-8(SP) 来表示

如果是 symbol+offset(SP) 形式，则表示伪寄存器 SP
如果是 offset(SP) 则表示硬件寄存器 SP
```

标志位

助记符	名字	用途
OF	溢出	0为无溢出 1为溢出
CF	进位	0为最高位无进位或错位 1为有
PF	奇偶	0表示数据最低8位中1的个数为奇数，1则表示1的个数为偶数
AF	辅助进位	
ZF	零	0表示结果不为0 1表示结果为0
SF	符号	0表示最高位为0 1表示最高位为1

Go常用命令

go build

```
-a  将命令源码文件与库源码文件全部重新构建，即使是最新的
-n  把编译期间涉及的命令全部打印出来，但不会真的执行，非常方便我们学习
-race    开启竞态条件的检测，支持的平台有限制
-x  打印编译期间用到的命名，它与 -n 的区别是，它不仅打印还会执行
-gcflags '[pattern=]arg list'
    传递go 工具编译调用的参数
-ldflags '[pattern=]arg list'
    传递go 工具链接调用的参数
```

-gcflags 、-ldflags 可以加`-help`参数查看详情

- `go build -gcflags -help`

事件说明

- 【2022-10-22】封面土拨鼠图片来源于谷歌搜索

参考

- 挖坑的张师傅、小贺coding
- 欧长坤
- 小米信息部技术团队
- 煎鱼