目　录

前言

《GO专家编程》

这是一本开源的Go语言进阶图书，帮助初级GO程序员成为资深专家，欢迎加入~

项目地址（欢迎勘误或投稿）

GitHub

联系作者(发现内容有误或参与写作)

邮件

常见数据结构实现原理

本章主要介绍常见的数据结构，比如channel、slice、map等，通过对其底层实现原理的分析，来加深认识，以此避免一些使用过程中的误区。

chan

1. 前言

channel是Golang在语言层面提供的goroutine间的通信方式，比Unix管道更易用也更轻便。channel主要用于进程内各goroutine间通信，如果需要跨进程通信，建议使用分布式系统的方法来解决。

本章从源码角度分析channel的实现机制，实际上这部分源码非常简单易读。

2. chan数据结构

`src/runtime/chan.go:hchan` 定义了channel的数据结构：

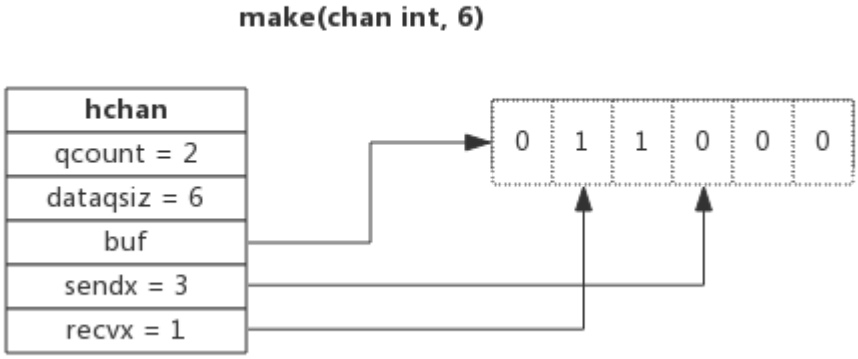
```
type hchan struct {
    qcount   uint           // 当前队列中剩余元素个数
    dataqsiz uint           // 环形队列长度，即可以存放的元素个数
    buf      unsafe.Pointer // 环形队列指针
    elemsize uint16         // 每个元素的大小
    closed   uint32           // 标识关闭状态
    elemtype *_type         // 元素类型
    sendx    uint           // 队列下标，指示元素写入时存放到队列中的位置
    recvx    uint           // 队列下标，指示元素从队列的该位置读出
    recvq    waitq          // 等待读消息的goroutine队列
    sendq    waitq          // 等待写消息的goroutine队列
    lock mutex              // 互斥锁，chan不允许并发读写
}
```

从数据结构可以看出channel由队列、类型信息、goroutine等待队列组成，下面分别说明其原理。

2.1 环形队列

chan内部实现了一个环形队列作为其缓冲区，队列的长度是创建chan时指定的。

下图展示了一个可缓存6个元素的channel示意图：



- dataqsiz指示了队列长度为6，即可缓存6个元素；
- buf指向队列的内存，队列中还剩余两个元素；
- qcount表示队列中还有两个元素；
- sendx指示后续写入的数据存储的位置，取值[0, 6)；
- recvx指示从该位置读取数据, 取值[0, 6)；

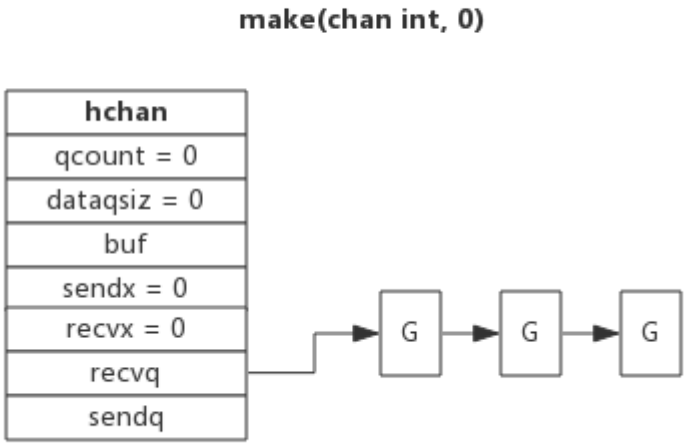
2.2 等待队列

从channel读数据，如果channel缓冲区为空或者没有缓冲区，当前goroutine会被阻塞。
向channel写数据，如果channel缓冲区已满或者没有缓冲区，当前goroutine会被阻塞。

被阻塞的goroutine将会挂在channel的等待队列中：

- 因读阻塞的goroutine会被向channel写入数据的goroutine唤醒；
- 因写阻塞的goroutine会被从channel读数据的goroutine唤醒；

下图展示了一个没有缓冲区的channel，有几个goroutine阻塞等待读数据：



注意，一般情况下recvq和sendq至少有一个为空。只有一个例外，那就是同一个goroutine使用select语句向channel一边写数据，一边读数据。

2.3 类型信息

一个channel只能传递一种类型的值，类型信息存储在hchan数据结构中。

- elemtype代表类型，用于数据传递过程中的赋值；
- elemsize代表类型大小，用于在buf中定位元素位置。

2.4 锁

一个channel同时仅允许被一个goroutine读写，为简单起见，本章后续部分说明读写过程时不再涉及加锁和解锁。

3. channel读写

3.1 创建channel

创建channel的过程实际上是初始化hchan结构。其中类型信息和缓冲区长度由make语句传入，buf的大小则与元素大小和缓冲区长度共同决定。

创建channel的伪代码如下所示：

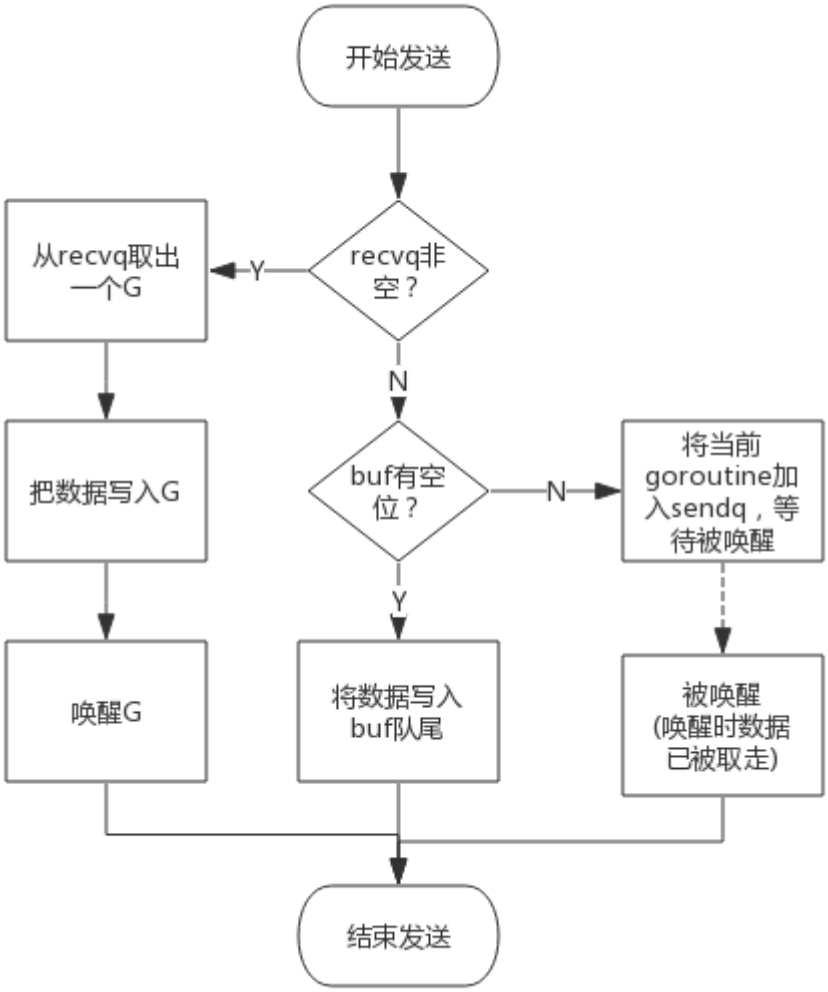
```
func makechan(t *chantype, size int) *hchan {
    var c *hchan
    c = new(hchan)
    c.buf = malloc(元素类型大小*size)
    c.elemsize = 元素类型大小
    c.elemtype = 元素类型
    c.dataqsiz = size

    return c
}
```

3.2 向channel写数据

向一个channel中写数据简单过程如下：

1. 如果等待接收队列recvq不为空，说明缓冲区中没有数据或者没有缓冲区，此时直接从recvq取出G,并把数据写入，最后把该G唤醒，结束发送过程；
2. 如果缓冲区中有空余位置，将数据写入缓冲区，结束发送过程；
3. 如果缓冲区中没有空余位置，将待发送数据写入G，将当前G加入sendq，进入睡眠，等待被读goroutine唤醒；

简单流程图如下：

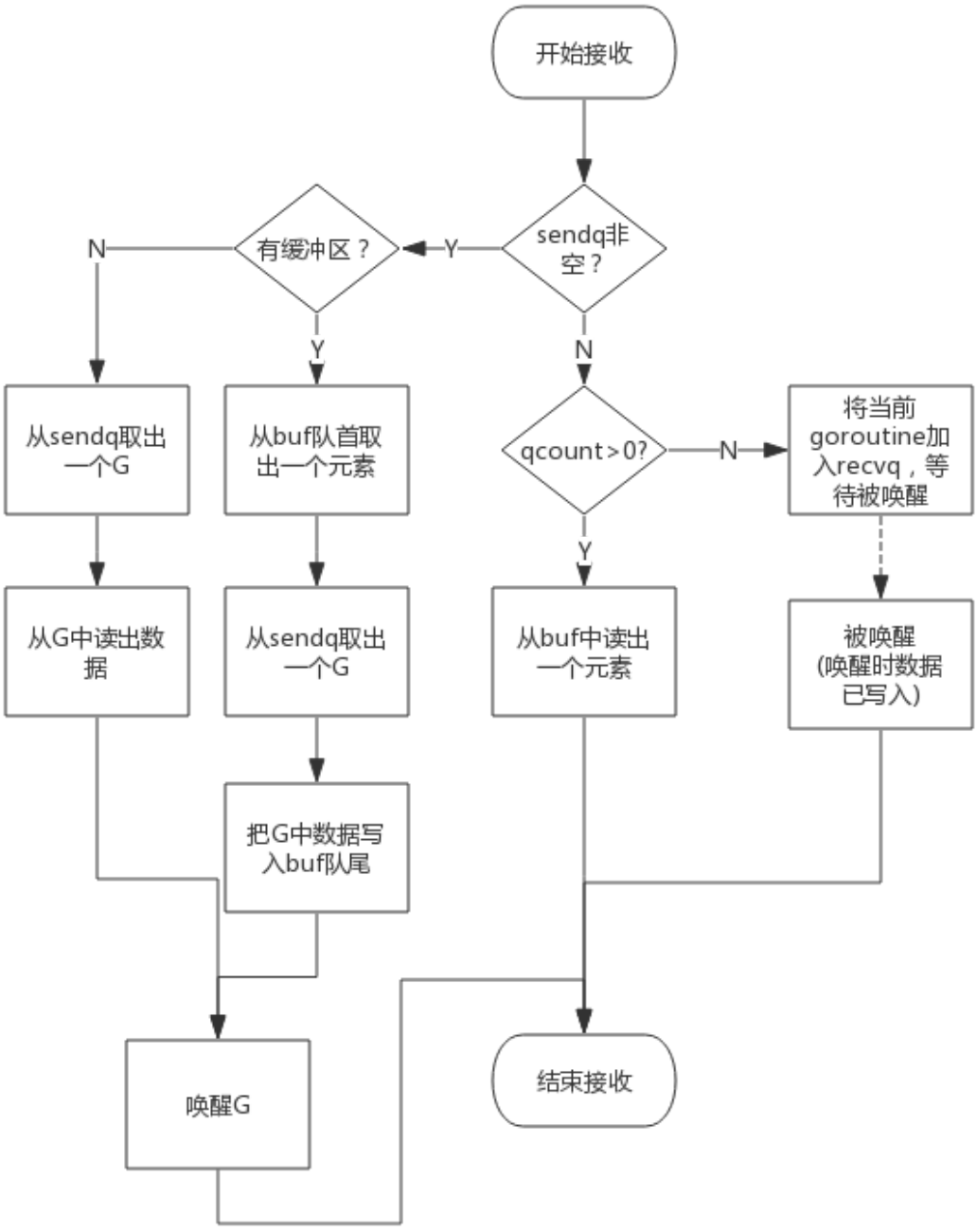


3.3 从channel读数据

从一个channel读数据简单过程如下：

1. 如果等待发送队列sendq不为空，且没有缓冲区，直接从sendq中取出G，把G中数据读出，最后把G唤醒，结束读取过程；
2. 如果等待发送队列sendq不为空，此时说明缓冲区已满，从缓冲区中首部读出数据，把G中数据写入缓冲区尾部，把G唤醒，结束读取过程；
3. 如果缓冲区中有数据，则从缓冲区取出数据，结束读取过程；
4. 将当前goroutine加入recvq，进入睡眠，等待被写goroutine唤醒；

简单流程图如下：



3.4 关闭channel

关闭channel时会把recvq中的G全部唤醒，本该写入G的数据位置为nil。把sendq中的G全部唤醒，但这些G会panic。

除此之外，panic出现的常见场景还有：

1. 关闭值为nil的channel
2. 关闭已经被关闭的channel
3. 向已经关闭的channel写数据

4. 常见用法

4.1 单向channel

顾名思义，单向channel指只能用于发送或接收数据，实际上也没有单向channel。

我们知道channel可以通过参数传递，所谓单向channel只是对channel的一种使用限制，这跟C语言使用const修饰函数参数为只读是一个道理。

- func readChan(chanName <-chan int)： 通过形参限定函数内部只能从channel中读取数据
- func writeChan(chanName chan<- int)： 通过形参限定函数内部只能向channel中写入数据

一个简单的示例程序如下：

```
func readChan(chanName <-chan int) {
    <- chanName
}

func writeChan(chanName chan<- int) {
    chanName <- 1
}

func main() {
    var mychan = make(chan int, 10)

    writeChan(mychan)
    readChan(mychan)
}
```

mychan是个正常的channel，而readChan()参数限制了传入的channel只能用来读，writeChan()参数限制了传入的channel只能用来写。

4.2 select

使用select可以监控多channel，比如监控多个channel，当其中某一个channel有数据时，就从其读出数据。

一个简单的示例程序如下：

```
package main

import (
    "fmt"
    "time"
)

func addNumberToChan(chanName chan int) {
    for {
        chanName <- 1
        time.Sleep(1 * time.Second)
    }
}

func main() {
    var chan1 = make(chan int, 10)
```

```
    var chan2 = make(chan int, 10)

    go addNumberToChan(chan1)
    go addNumberToChan(chan2)

    for {
        select {
        case e := <- chan1 :
            fmt.Printf("Get element from chan1: %d\n", e)
        case e := <- chan2 :
            fmt.Printf("Get element from chan2: %d\n", e)
        default:
            fmt.Printf("No element in chan1 and chan2.\n")
            time.Sleep(1 * time.Second)
        }
    }
}
```

程序中创建两个channel： chan1和chan2。函数addNumberToChan()函数会向两个channel中周期性写入数据。通过select可以监控两个channel，任意一个可读时就从其中读出数据。

程序输出如下：

```
D:\SourceCode\GoExpert\src>go run main.go
Get element from chan1: 1
Get element from chan2: 1
No element in chan1 and chan2.
Get element from chan2: 1
Get element from chan1: 1
No element in chan1 and chan2.
Get element from chan2: 1
Get element from chan1: 1
No element in chan1 and chan2.
```

从输出可见，从channel中读出数据的顺序是随机的，事实上select语句的多个case执行顺序是随机的，关于select的实现原理会有专门章节分析。

通过这个示例想说的是：select的case语句读channel不会阻塞，尽管channel中没有数据。这是由于case语句编译后调用读channel时会明确传入不阻塞的参数，此时读不到数据时不会将当前goroutine加入到等待队列，而是直接返回。

4.3 range

通过range可以持续从channel中读出数据，好像在遍历一个数组一样，当channel中没有数据时会阻塞当前goroutine，与读channel时阻塞处理机制一样。

```
func chanRange(chanName chan int) {
    for e := range chanName {
        fmt.Printf("Get element from chan: %d\n", e)
    }
}
```

注意：如果向此channel写数据的goroutine退出时，系统检测到这种情况后会panic，否则range将会永久阻塞。

slice

1. 前言

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。

正因为灵活，如果不了解其内部实现机制，有可能遭遇莫名的异常现象。Slice的实现原理很简单，本节试图根据真实的使用场景，在源码中总结实现原理。

2. 热身环节

按照惯例，我们开始前先看几段代码用于检测对Slice的理解程度。

2.1 题目一

下面程序输出什么？

```
package main

import (
    "fmt"
)

func main() {
    var array [10]int

    var slice = array[5:6]

    fmt.Println("lenth of slice: ", len(slice))
    fmt.Println("capacity of slice: ", cap(slice))
    fmt.Println(&slice[0] == &array[5])
}
```

程序解释：
main函数中定义了一个10个长度的整型数组array，然后定义了一个切片slice，切取数组的第6个元素，最后打印slice的长度和容量，判断切片的第一个元素和数组的第6个元素地址是否相等。

参考答案：
slice根据数组array创建，与数组共享存储空间，slice起始位置是array[5]，长度为1，容量为5，slice[0]和array[5]地址相同。

2.2 题目二

下面程序输出什么？

```
package main

import (
    "fmt"
)
```

```
func AddElement(slice []int, e int) []int {
    return append(slice, e)
}

func main() {
    var slice []int
    slice = append(slice, 1, 2, 3)

    newSlice := AddElement(slice, 4)
    fmt.Println(&slice[0] == &newSlice[0])
}
```

程序解释：
函数AddElement()接受一个切片和一个元素，把元素append进切片中，并返回切片。main()函数中定义一个切片，并向切片中append 3个元素，接着调用AddElement()继续向切片append进第4个元素同时定义一个新的切片newSlice。最后判断新切片newSlice与旧切片slice是否共用一块存储空间。

参考答案：
append函数执行时会判断切片容量是否能够存放新增元素，如果不能，则会重新申请存储空间，新存储空间将是原来的2倍或1.25倍（取决于扩展原空间大小），本例中实际执行了两次append操作，第一次空间增长到4，所以第二次append不会再扩容，所以新旧两个切片将共用一块存储空间。程序会输出”true”。

2.3 题目三

下面程序由Golang源码改编而来，程序输出什么？

```
package main

import (
    "fmt"
)

func main() {
    orderLen := 5
    order := make([]uint16, 2 * orderLen)

    pollorder := order[:orderLen:orderLen]
    lockorder := order[orderLen:][:orderLen:orderLen]

    fmt.Println("len(pollorder) = ", len(pollorder))
    fmt.Println("cap(pollorder) = ", cap(pollorder))
    fmt.Println("len(lockorder) = ", len(lockorder))
    fmt.Println("cap(lockorder) = ", cap(lockorder))
}
```

程序解释：
该段程序源自select的实现代码，程序中定义一个长度为10的切片order，pollorder和lockorder分别是对order切片做了order[low:high:max]操作生成的切片，最后程序分别打印pollorder和lockorder的容量和长度。

参考答案：
order[low:high:max]操作意思是对order进行切片，新切片范围是[low, high),新切片容量是max。order长度为2倍的orderLen，pollorder切片指的是order的前半部分切片，lockorder指的是order的后半部分切片，即原order分成了两段。所以，pollorder和lockerorder的长度和容量都是orderLen，即5。

3. Slice实现原理

Slice依托数组实现，底层数组对用户屏蔽，在底层数组容量不足时可以实现自动重分配并生成新的Slice。
接下来按照实际使用场景分别介绍其实现机制。

3.1 Slice数据结构

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

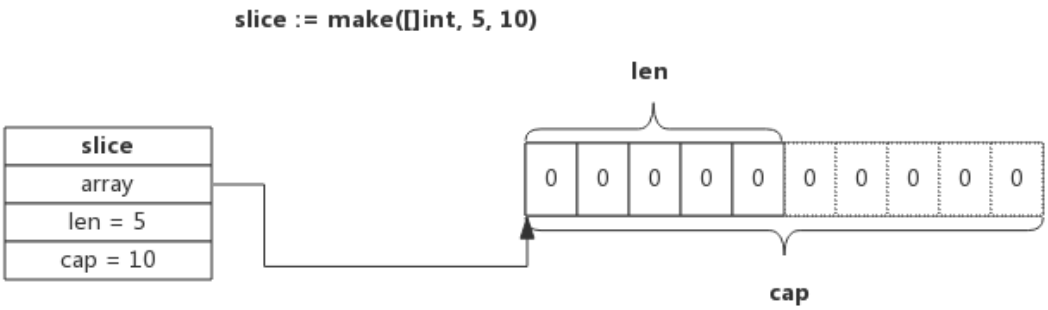
```
type slice struct {
    array unsafe.Pointer
    len   int
    cap   int
}
```

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。

3.2 使用make创建Slice

使用make来创建Slice时，可以同时指定长度和容量，创建时底层会分配一个数组，数组的长度即容量。

例如，语句 `slice := make([]int, 5, 10)` 所创建的Slice，结构如下图所示：

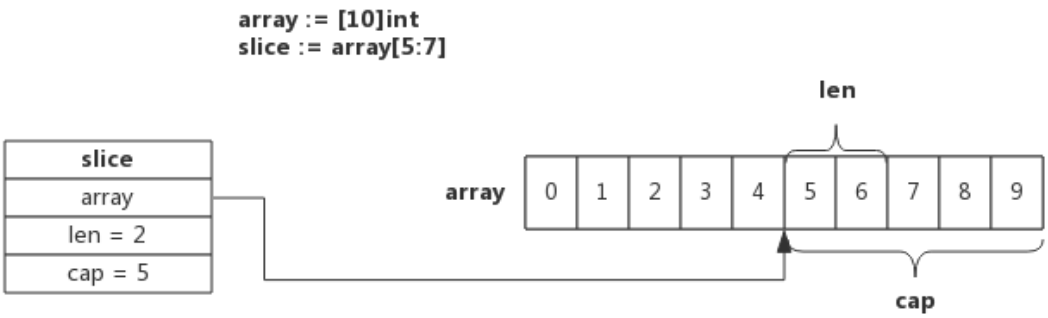


该Slice长度为5，即可以使用下标slice[0] ~ slice[4]来操作里面的元素，capacity为10，表示后续向slice添加新的元素时可以不必重新分配内存，直接使用预留内存即可。

3.3 使用数组创建Slice

使用数组来创建Slice时，Slice将与原数组共用一部分内存。

例如，语句 `slice := array[5:7]` 所创建的Slice，结构如下图所示：



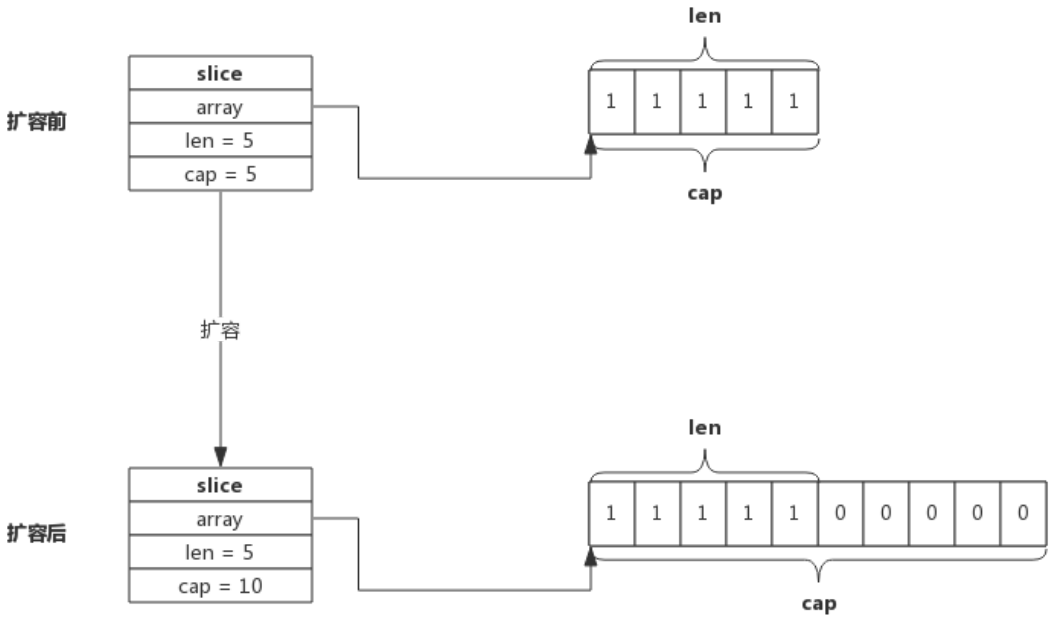
切片从数组array[5]开始，到数组array[7]结束（不含array[7]），即切片长度为2，数组后面的内容都作为切片的预留内存，即capacity为5。

数组和切片操作可能作用于同一块内存，这也是使用过程中需要注意的地方。

3.4 Slice 扩容

使用append向Slice追加元素时，如果Slice空间不足，将会触发Slice扩容，扩容实际上是重新分配一块更大的内存，将原Slice数据拷贝进新Slice，然后返回新Slice，扩容后再将数据追加进去。

例如，当向一个capacity为5，且length也为5的Slice再次追加1个元素时，就会发生扩容，如下图所示：



扩容操作只关心容量，会把原Slice数据拷贝到新Slice，追加数据由append在扩容结束后完成。上图可见，扩容后新的Slice长度仍然是5，但容量由5提升到了10，原Slice的数据也都拷贝到了新Slice指向的数组中。

扩容容量的选择遵循以下规则：

- 如果原Slice容量小于1024，则新Slice容量将扩大为原来的2倍；
- 如果原Slice容量大于等于1024，则新Slice容量将扩大为原来的1.25倍；

使用append()向Slice添加一个元素的实现步骤如下：

- 假如Slice容量够用，则将新元素追加进去，Slice.len++，返回原Slice
- 原Slice容量不够，则将Slice先扩容，扩容后得到新Slice
- 将新元素追加进新Slice，Slice.len++，返回新的Slice。

3.5 Slice Copy

使用copy()内置函数拷贝两个切片时，会将源切片的数据逐个拷贝到目的切片指向的数组中，拷贝数量取两个切片长度的最小值。

例如长度为10的切片拷贝到长度为5的切片时，将会拷贝5个元素。

也就是说，copy过程中不会发生扩容。

3.5 特殊切片

根据数组或切片生成新的切片一般使用 `slice := array[start:end]` 方式，这种新生成的切片并没有指定切片的容量，实际上新切片的容量是从start开始直至array的结束。

比如下面两个切片，长度和容量都是一致的，使用共同的内存地址：

```
sliceA := make([]int, 5, 10)
sliceB := sliceA[0:5]
```

根据数组或切片生成切片还有另一种写法，即切片同时也指定容量，即slice[start:end:cap], 其中cap即为新切片的容量，当然容量不能超过原切片实际值，如下所示：

```
    sliceA := make([]int, 5, 10)  //length = 5; capacity = 10
    sliceB := sliceA[0:5]         //length = 5; capacity = 10
    sliceC := sliceA[0:5:5]       //length = 5; capacity = 5
```

这切片方法不常见，在Golang源码里能够见到，不过非常利于切片的理解。

4. 编程Tips

- 创建切片时可根据实际需要预分配容量，尽量避免追加过程中扩容操作，有利于提升性能；
- 切片拷贝时需要判断实际拷贝的元素个数
- 谨慎使用多个切片操作同一个数组，以防读写冲突

5. Slice总结

- 每个切片都指向一个底层数组
- 每个切片都保存了当前切片的长度、底层数组可用容量

- 使用len()计算切片长度时间复杂度为O(1)，不需要遍历切片
- 使用cap()计算切片容量时间复杂度为O(1)，不需要遍历切片
- 通过函数传递切片时，不会拷贝整个切片，因为切片本身只是个结构体而已
- 使用append()向切片追加元素时有可能触发扩容，扩容后将会生成新的切片

map

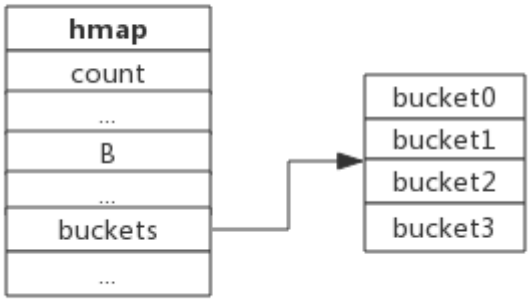
1. map数据结构

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。

map数据结构由 `runtime/map.go:hmap` 定义:

```
type hmap struct {
    count     int // 当前保存的元素个数
    ...
    B         uint8
    ...
    buckets    unsafe.Pointer // bucket数组指针，数组的大小为2^B
    ...
}
```

下图展示一个拥有4个bucket的map：



本例中， `hmap.B=2` ， 而hmap.buckets长度是2^B为4. 元素经过哈希运算后会落到某个bucket中进行存储。查找过程类似。

`bucket` 很多时候被翻译为桶，所谓的 `哈希桶` 实际上就是bucket。

2. bucket数据结构

bucket数据结构由 `runtime/map.go:bmap` 定义：

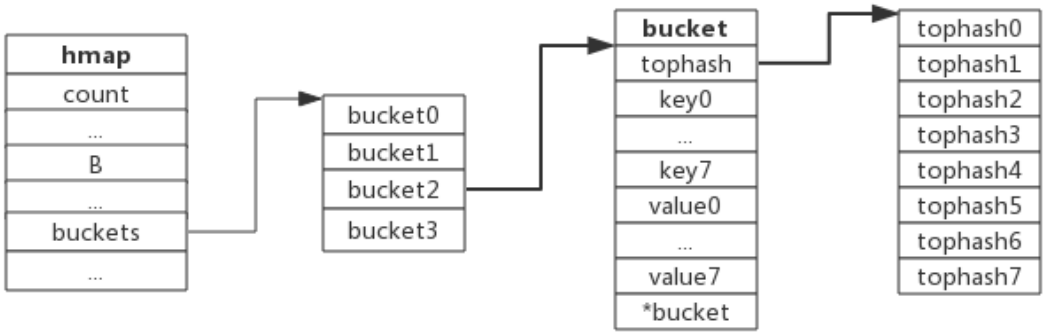
```
type bmap struct {
    tophash [8]uint8 //存储哈希值的高8位
    data    byte[1]  //key value数据:key/key/key/.../value/value/value...
    overflow *bmap   //溢出bucket的地址
}
```

每个bucket可以存储8个键值对。

- tophash是个长度为8的数组，哈希值相同的键（准确的说是哈希值低位相同的键）存入当前bucket时会将哈希值的高位存储在该数组中，以方便后续匹配。
- data区存放的是key-value数据，存放顺序是key/key/key/...value/value/value，如此存放是为了节省字节对齐带来的空间浪费。
- overflow 指针指向的是下一个bucket，据此将所有冲突的键连接起来。

注意：上述中data和overflow并不是在结构体中显示定义的，而是直接通过指针运算进行访问的。

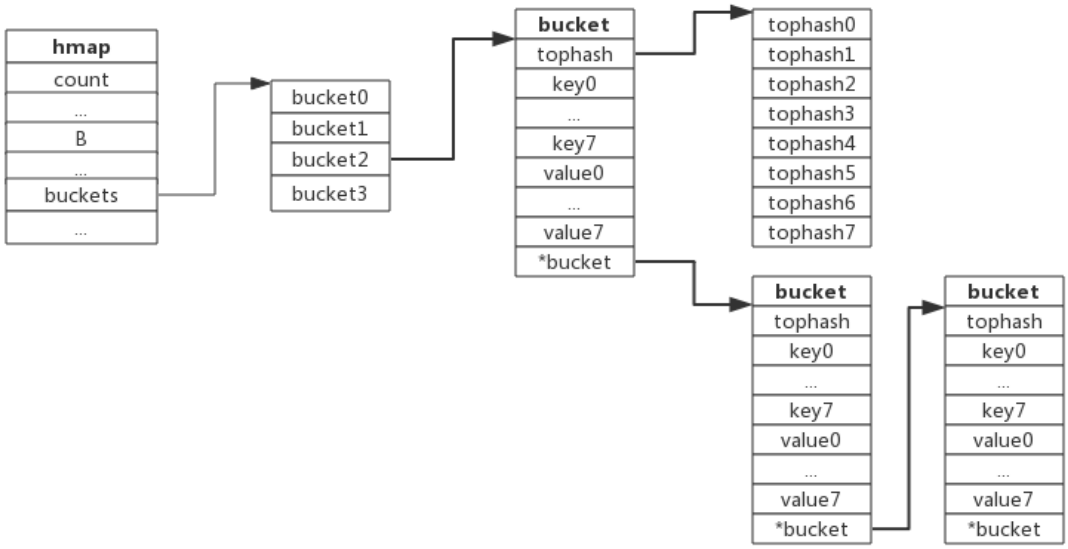
下图展示bucket存放8个key-value对：



3. 哈希冲突

当有两个或以上数量的键被哈希到了同一个bucket时，我们称这些键发生了冲突。Go使用链地址法来解决键冲突。
由于每个bucket可以存放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。

下图展示产生冲突后的map：



bucket数据结构指示下一个bucket的指针称为overflow bucket，意为当前bucket盛不下而溢出的部分。事实上哈希冲突并不是好事情，它降低了存取效率，好的哈希算法可以保证哈希值的随机性，但冲突过多也是要控制的，后面会再详细介绍。

4. 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。

5. 渐进式扩容

5.1 扩容的前提条件

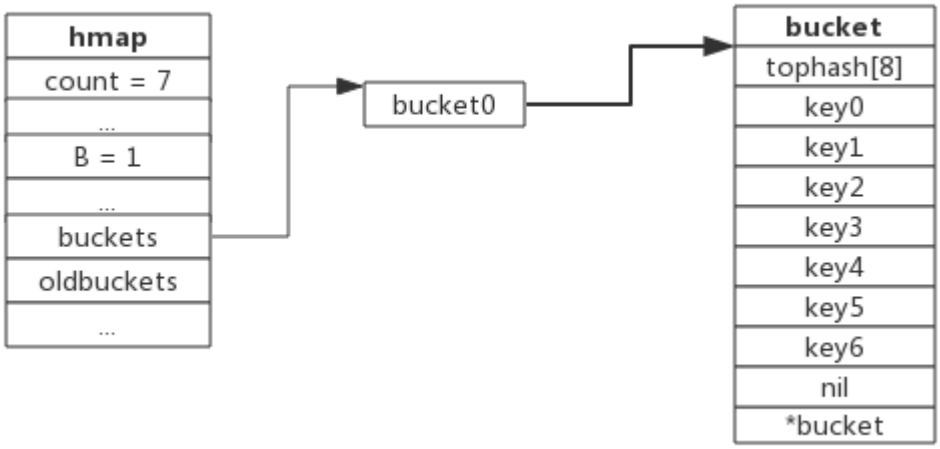
为了保证访问效率，当新元素将要添加进map时，都会检查是否需要扩容，扩容实际上是以空间换时间的手段。
触发扩容的条件有二个：

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。

5.2 增量扩容

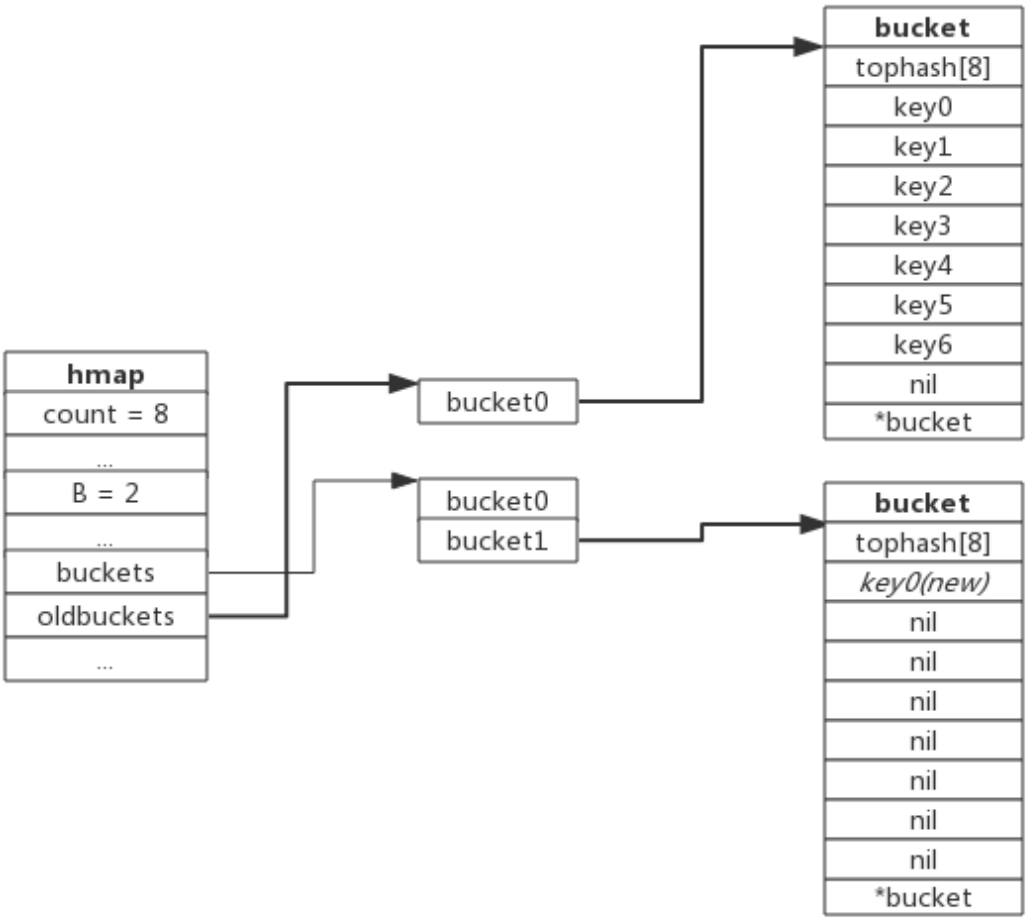
当负载因子过大时，就新建一个bucket，新的bucket长度是原来的2倍，然后旧bucket数据搬迁到新的bucket。
考虑到如果map存储了数以亿计的key-value，一次性搬迁将会造成比较大的延时，Go采用逐步搬迁策略，即每次访问map时都会触发一次搬迁，每次搬迁2个键值对。

下图展示了包含一个bucket满载的map(为了描述方便，图中bucket省略了value区域):



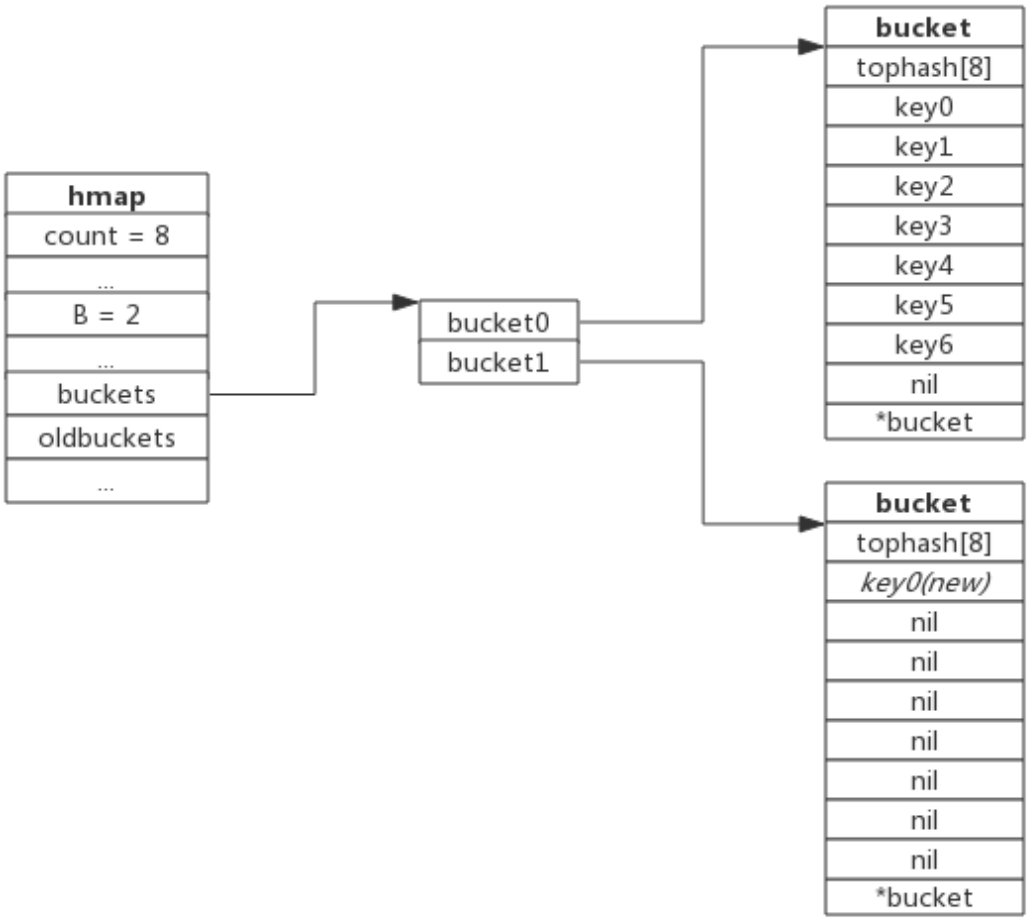
当前map存储了7个键值对，只有1个bucket。此地负载因子为7。再次插入数据时将会触发扩容操作，扩容之后再将新插入键写入新的bucket。

当第8个键值对插入时，将会触发扩容，扩容后示意图如下：



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。

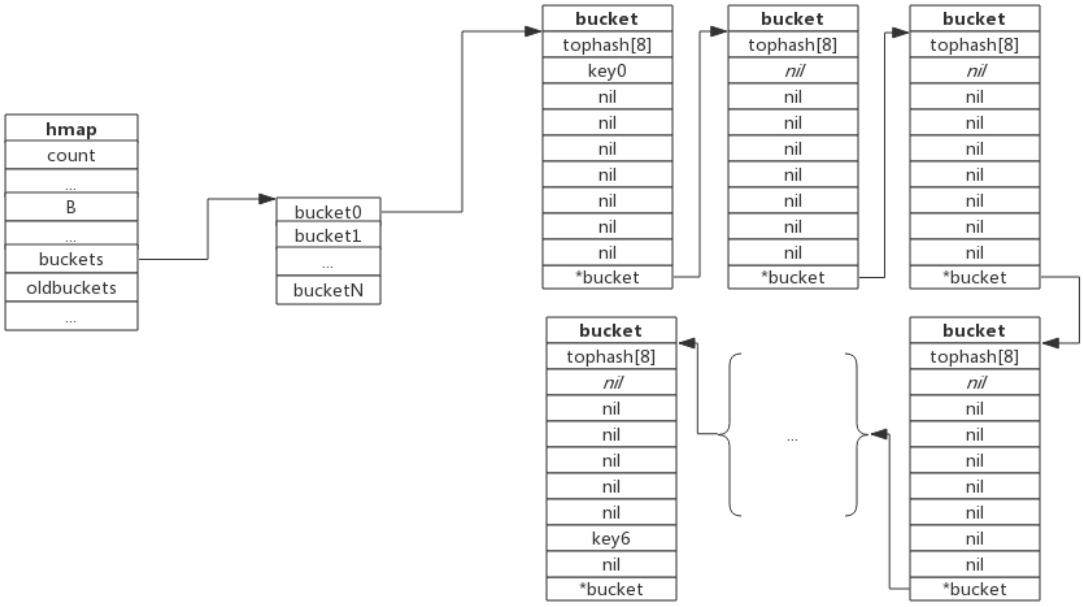
搬迁完成后的示意图如下：



数据搬迁过程中原bucket中的键值对将存在于新bucket的前面，新插入的键值对将存在于新bucket的后面。
实际搬迁过程中比较复杂，将在后续源码分析中详细介绍。

5.3 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。
在极端场景下，比如不断地增删，而键值对正好集中在一小部分的bucket，这样会造成overflow的bucket数量增多，但负载因子又不高，从而无法执行增量搬迁的情况，如下图所示：



上图可见，overflow的bucket中大部分是空的，访问效率会很差。此时进行一次等量扩容，即buckets数量不变，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。

6. 查找过程

查找过程如下：

1. 根据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. 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。

7. 插入过程

新元素插入过程如下：

1. 根据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

struct

1. 前言

Go的struct声明允许字段附带 `Tag` 来对字段做一些标记。

该 `Tag` 不仅仅是一个字符串那么简单，因为其主要用于反射场景， `reflect` 包中提供了操作 `Tag` 的方法，所以 `Tag` 写法也要遵循一定的规则。

2. Tag的本质

2.1 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"`
}
```

上述代码 `ServerName` 字段的 `Tag` 包含两个key-value对。 `ServerIP` 字段的 `Tag` 只包含一个key-value对。

2.2 Tag是Struct的一部分

前面说过， `Tag` 只有在反射场景中才有用，而反射包中提供了操作 `Tag` 的方法。在说方法前，有必要先了解一下Go是如何管理struct字段的。

以下是 `reflect` 包中的类型声明，省略了部分与本文无关的字段。

```
// 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
    ...
}

type StructTag string
```

可见，描述一个结构体成员的结构中包含了 `StructTag` ，而其本身是一个 `string` 。也就是说 `Tag` 其实是结构体字段的一个组成部分。

2.3 获取Tag

`StructTag` 提供了 `Get(key string) string` 方法来获取 `Tag` ，示例如下：

```
package main

import (
    "reflect"
    "fmt"
)

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

func main() {
    s := Server{}
    st := reflect.TypeOf(s)

    field1 := st.Field(0)
    fmt.Printf("key1:%v\n", field1.Tag.Get("key1"))
    fmt.Printf("key11:%v\n", field1.Tag.Get("key11"))

    filed2 := st.Field(1)
    fmt.Printf("key2:%v\n", filed2.Tag.Get("key2"))
}
```

程序输出如下：

```
key1:value1
key11:value11
key2:value2
```

3. Tag存在的意义

本文示例中tag没有任何实际意义，这是为了阐述tag的定义与操作方法，也为了避免与你之前见过的诸如 `json:xxx` 混淆。

使用反射可以动态的给结构体成员赋值，正是因为有tag，在赋值前可以使用tag来决定赋值的动作。
比如，官方的 `encoding/json` 包，可以将一个JSON数据 `Unmarshal` 进一个结构体，此过程中就使用了Tag。该包定义一些规则，只要参考该规则设置tag就可以将不同的JSON数据转换成结构体。

总之：正是基于struct的tag特性，才有了诸如json、orm等等的应用。理解这个关系是至关重要的。或许，你可以定义另一种tag规则，来处理你特有的数据。

4. Tag常见用法

常见的tag用法，主要是JSON数据解析、ORM映射等。

iota

1. 前言

我们知道iota常用于const表达式中，我们还知道其值是从零开始，const声明块中每增加一行iota值自增1。

使用iota可以简化常量定义，但其规则必须要牢牢掌握，否则在我们阅读别人源码时可能会造成误解或障碍。本节我们尝试全面的总结其使用场景，另外花一小部分时间看一下其实现原理，从原理上把握可以更深刻的记忆这些规则。

2. 热身

按照惯例，我们看几个有意思的小例子，用于检测我们对于iota的理解是否准确。

2.1 题目一

下面常量定义源于GO源码，下面每个常量的值是多少？

```
type Priority int
const (
    LOG_EMERG Priority = iota
    LOG_ALERT
    LOG_CRIT
    LOG_ERR
    LOG_WARNING
    LOG_NOTICE
    LOG_INFO
    LOG_DEBUG
)
```

题目解释：

上面代码源于日志模块，定义了一组代表日志级别的常量，常量类型为Priority，实际为int类型。

参考答案：

iota初始值为0，也即LOG_EMERG值为0，下面每个常量递增1。

2.2 题目二

下面代码取自Go源码，请问每个常量值是多少？

```
const (
    mutexLocked = 1 << iota // mutex is locked
    mutexWoken
    mutexStarving
    mutexWaiterShift = iota
    starvationThresholdNs = 1e6
)
```

题目解释：

以上代码取自Go互斥锁Mutex的实现，用于指示各种状态位的地址偏移。

参考答案：

mutexLocked == 1；mutexWoken == 2；mutexStarving == 4；mutexWaiterShift == 3；
starvationThresholdNs == 1000000。

2.3 题目三

请问每个常量值是多少？

```
const (
    bit0, mask0 = 1 << iota, 1<<iota - 1
    bit1, mask1
    _, _
    bit3, mask3
)
```

题目解释：

以上代码取自Go官方文档。

参考答案：

bit0 == 1， mask0 == 0， bit1 == 2， mask1 == 1， bit3 == 8， mask3 == 7

3. 规则

很多书上或博客描述的规则是这样的：

1. iota在const关键字出现时被重置为0
2. const声明块中每新增一行iota值自增1

我曾经也这么理解，看过编译器代码后发现，其实规则只有一条：

- iota代表了const声明块的行索引（下标从0开始）

这样理解更贴近编译器实现逻辑，也更准确。除此之外，const声明还有个特点，即第一个常量必须指定一个表达式，后续的常量如果没有表达式，则继承上面的表达式。

下面再来根据这个规则看下这段代码：

```
const (
    bit0, mask0 = 1 << iota, 1<<iota - 1   //const声明第0行，即iota==0
    bit1, mask1                            //const声明第1行，即iota==1，表达式继承上面的语句
    _, _                                   //const声明第2行，即iota==2
    bit3, mask3                            //const声明第3行，即iota==3
)
```

- 第0行的表达式展开即 `bit0, mask0 = 1 << 0, 1<<0 - 1` ，所以bit0 == 1，mask0 == 0；
- 第1行没有指定表达式继承第一行，即 `bit1, mask1 = 1 << 1, 1<<1 - 1` ，所以bit1 == 2，mask1 == 1；
- 第2行没有定义常量
- 第3行没有指定表达式继承第一行，即 `bit3, mask3 = 1 << 3, 1<<3 - 1` ，所以bit0 == 8，mask0 == 7；

4. 编译原理

const块中每一行在GO中使用spec数据结构描述，spec声明如下：

```
    // A ValueSpec node represents a constant or variable declaration
    // (ConstSpec or VarSpec production).
    //
    ValueSpec struct {
        Doc     *CommentGroup // associated documentation; or nil
        Names   []*Ident      // value names (len(Names) > 0)
        Type    Expr          // value type; or nil
        Values  []Expr        // initial values; or nil
        Comment *CommentGroup // line comments; or nil
    }
```

这里我们只关注ValueSpec.Names， 这个切片中保存了一行中定义的常量，如果一行定义N个常量，那么ValueSpec.Names切片长度即为N。

const块实际上是spec类型的切片，用于表示const中的多行。

所以编译期间构造常量时的伪算法如下：

```
    for iota, spec := range ValueSpecs {
        for i, name := range spec.Names {
            obj := NewConst(name, iota...) //此处将iota传入，用于构造常量
            ...
        }
    }
```

从上面可以更清晰的看出iota实际上是遍历const块的索引，每行中即便多次使用iota，其值也不会递增。

string

string标准概念

Go标准库 `builtin` 给出了所有内置类型的定义。
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，通常是但并不一定非得是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
    len int
}
```

其数据结构很简单：

- stringStruct.str：字符串的首地址；
- stringStruct.len：字符串的长度；

string数据结构跟切片有些类似，只不过切片还有一个表示容量的成员，事实上string和切片，准确的说是byte切片经常发生转换。这个后面再详细介绍。

string操作

声明

如下代码所示，可以声明一个string变量变赋予初值：

```
    var str string
    str = "Hello World"
```

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

```
func gostringnocopy(str *byte) string { // 根据字符串地址构建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
}
```

string在runtime包中就是stringStruct，对外呈现叫做string。

[]byte转string

byte切片可以很方便的转换成string，如下所示：

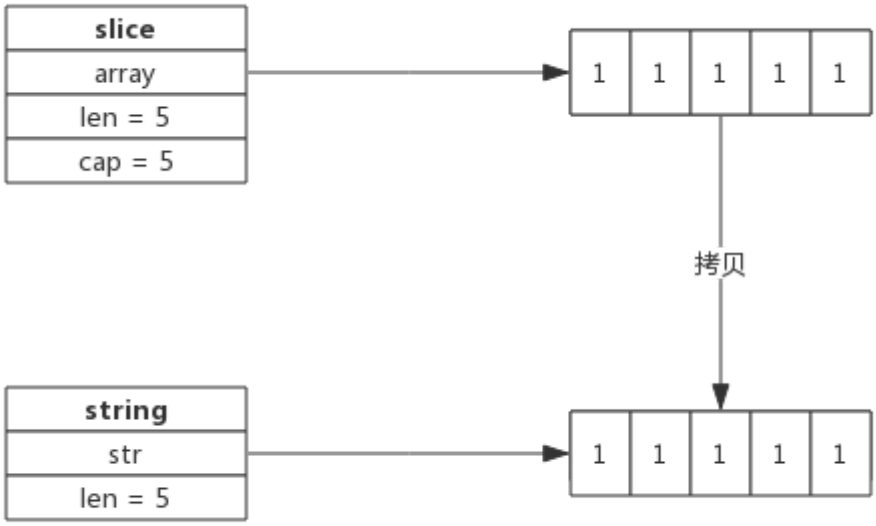
```
func GetStringBySlice(s []byte) string {
    return string(s)
}
```

需要注意的是这种转换需要一次内存拷贝。

转换过程如下：

1. 根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)

转换示意图：



string转[]byte

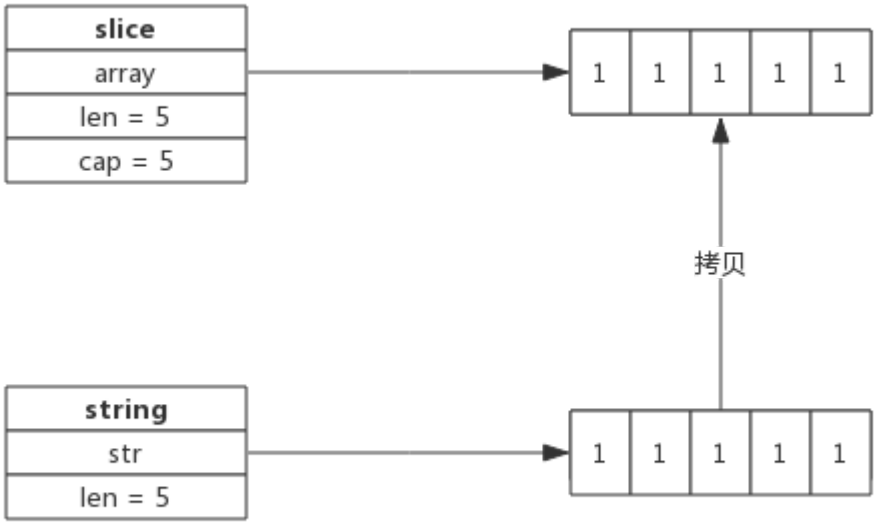
string也可以方便的转成byte切片，如下所示：

```
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

转换示意图：



字符串拼接

字符串可以很方便的拼接，像下面这样：

```
str := "Str1" + "Str2" + "Str3"
```

即便有非常多的字符串需要拼接，性能上也有比较好的保证，因为新字符串的内存空间是一次分配完成的，所以性能消耗主要在拷贝数据上。

一个拼接语句的字符串编译时都会被存放到一个切片中，拼接过程需要遍历两次切片，第一次遍历获取总的字符串长度，据此申请内存，第二次遍历会把字符串逐个拷贝过去。

字符串拼接伪代码如下：

```
func concatstrings(a []string) string { // 字符串拼接
    length := 0        // 拼接后总的字符串长度

    for _, str := range a {
        length += len(str)
    }

    s, b := rawstring(length) // 生成指定大小的字符串，返回一个string和切片，二者共享内存空间

    for _, str := range a {
        copy(b, str)    // string无法修改，只能通过切片修改
        b = b[len(str):]
```

```
    }

    return s
}
```

因为string是无法直接修改的，所以这里使用rawstring()方法初始化一个指定大小的string，同时返回一个切片，二者共享同一块内存空间，后面向切片中拷贝数据，也就间接修改了string。

rawstring()源代码如下：

```
func rawstring(size int) (s string, b []byte) { // 生成一个新的string，返回的string和切片共享相同的空间
    p := mallocgc(uintptr(size), nil, false)

    stringStructOf(&s).str = p
    stringStructOf(&s).len = size

    *(*slice)(unsafe.Pointer(&b)) = slice{p, size, size}

    return
}
```

为什么字符串不允许修改？

像C++语言中的string，其本身拥有内存空间，修改string是支持的。但Go的实现中，string不包含内存空间，只有一个内存的指针，这样做的好处是string变得非常轻量，可以很方便的进行传递而不用担心内存拷贝。

因为string通常指向字符串字面量，而字符串字面量存储位置是只读段，而不是堆或栈上，所以才有了string不可修改的约定。

[]byte转换成string一定会拷贝内存吗？

byte切片转换成string的场景很多，为了性能上的考虑，有时候只是临时需要字符串的场景下，byte切片转换成string时并不会拷贝内存，而是直接返回一个string，这个string的指针(string.str)指向切片的内存。

比如，编译器会识别如下临时场景：

- 使用m[string(b)]来查找map（map是string为key，临时把切片b转成string）；
- 字符串拼接，如”<” + “string(b)” + “>”；
- 字符串比较：string(b) == “foo”

因为是临时把byte切片转换成string，也就避免了因byte切片同容改成而导致string引用失败的情况，所以此时可以不必拷贝内存新建一个string。

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，要根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。

常见控制结构实现原理

本章主要介绍常见的控制结构，比如defer、select、range等，通过对其底层实现原理的分析，来加深认识，以此避免一些使用过程中的误区。

defer

1. 前言

defer语句用于延迟函数的调用，每次defer都会把一个函数压入栈中，函数返回前再把延迟的函数取出并执行。

为了方便描述，我们把创建defer的函数称为主函数，defer语句后面的函数称为延迟函数。

延迟函数可能有输入参数，这些参数可能来源于定义defer的函数，延迟函数也可能引用主函数用于返回的变量，也就是说延迟函数可能会影响主函数的一些行为，这些场景下，如果不了解defer的规则很容易出错。

其实官方说明的defer的三个原则很清楚，本节试图汇总defer的使用场景并做简单说明。

2. 热身

按照惯例，我们看几个有意思的题目，用于检验对defer的了解程度。

2.1 题目一

下面函数输出结果是什么？

```
func deferFuncParameter() {
    var aInt = 1

    defer fmt.Println(aInt)

    aInt = 2
    return
}
```

题目说明：
函数deferFuncParameter()定义一个整型变量并初始化为1，然后使用defer语句打印出变量值，最后修改变量值为2.

参考答案：
输出1。延迟函数fmt.Println(aInt)的参数在defer语句出现时就已经确定了，所以无论后面如何修改aInt变量都不会影响延迟函数。

2.2 题目二

下面程序输出什么？

```
package main

import "fmt"

func printArray(array *[3]int) {
    for i := range array {
        fmt.Println(array[i])
    }
}
```

```
func deferFuncParameter() {
    var aArray = [3]int{1, 2, 3}

    defer printArray(&aArray)

    aArray[0] = 10
    return
}

func main() {
    deferFuncParameter()
}
```

函数说明：
函数deferFuncParameter()定义一个数组，通过defer延迟函数printArray()的调用，最后修改数组第一个元素。printArray()函数接受数组的指针并把数组全部打印出来。

参考答案：
输出10、2、3三个值。延迟函数printArray()的参数在defer语句出现时就已经确定了，即数组的地址，由于延迟函数执行时机是在return语句之前，所以对数组的最终修改值会被打印出来。

2.3 题目三

下面函数输出什么？

```
func deferFuncReturn() (result int) {
    i := 1

    defer func() {
       result++
    }()

    return i
}
```

函数说明：
函数拥有一个具名返回值result，函数内部声明一个变量i，defer指定一个延迟函数，最后返回变量i。延迟函数中递增result。

参考答案：
函数输出2。函数的return语句并不是原子的，实际执行分为设置返回值->ret，defer语句实际执行在返回前，即拥有defer的函数返回过程是：设置返回值->执行defer->ret。所以return语句先把result设置为i的值，即1，defer语句中又把result递增1，所以最终返回2。

3. defer规则

Golang官方博客里总结了defer的行为规则，只有三条，我们围绕这三条进行说明。

3.1 规则一：延迟函数的参数在defer语句出现时就已经确定下来了

官方给出一个例子，如下所示：

```
func a() {
    i := 0
    defer fmt.Println(i)
    i++
```

```
    return
}
```

defer语句中的fmt.Println()参数i值在defer出现时就已经确定下来，实际上是拷贝了一份。后面对变量i的修改不会影响fmt.Println()函数的执行，仍然打印”0”。

注意：对于指针类型参数，规则仍然适用，只不过延迟函数的参数是一个地址值，这种情况下，defer后面的语句对变量的修改可能会影响延迟函数。

3.2 规则二：延迟函数执行按后进先出顺序执行，即先出现的defer最后执行

这个规则很好理解，定义defer类似于入栈操作，执行defer类似于出栈操作。

设计defer的初衷是简化函数返回时资源清理的动作，资源往往有依赖顺序，比如先申请A资源，再根据A资源申请B资源，根据B资源申请C资源，即申请顺序是:A->B->C，释放时往往又要反向进行。这就是把defer设计成LIFO的原因。

每申请到一个用完需要释放的资源时，立即定义一个defer来释放资源是个很好的习惯。

3.3 规则三：延迟函数可能操作主函数的具名返回值

定义defer的函数，即主函数可能有返回值，返回值有没有名字没有关系，defer所作用的函数，即延迟函数可能会影响到返回值。

若要理解延迟函数是如何影响主函数返回值的，只要明白函数是如何返回的就足够了。

3.3.1 函数返回过程

有一个事实必须要了解，关键字*return*不是一个原子操作，实际上*return*只代理汇编指令*ret*，即将跳转程序执行。比如语句 `return i` ，实际上分两步进行，即将i值存入栈中作为返回值，然后执行跳转，而defer的执行时机正是跳转前，所以说defer执行时还是有机会操作返回值的。

举个实际的例子进行说明这个过程：

```
func deferFuncReturn() (result int) {
    i := 1

    defer func() {
        result++
    }()

    return i
}
```

该函数的return语句可以拆分成下面两行：

```
result = i
return
```

而延迟函数的执行正是在return之前，即加入defer后的执行过程如下：

```
result = i
result++
return
```

所以上面函数实际返回i++值。

关于主函数有不同的返回方式，但返回机制就如上机介绍所说，只要把return语句拆开都可以很好的理解，下面分别举例说明

3.3.2 主函数拥有匿名返回值，返回字面值

一个主函数拥有一个匿名的返回值，返回时使用字面值，比如返回”1”、”2”、”Hello”这样的值，这种情况下defer语句是无法操作返回值的。

一个返回字面值的函数，如下所示：

```
func foo() int {
    var i int

    defer func() {
        i++
    }()

    return 1
}
```

上面的return语句，直接把1写入栈中作为返回值，延迟函数无法操作该返回值，所以就无法影响返回值。

3.3.3 主函数拥有匿名返回值，返回变量

一个主函数拥有一个匿名的返回值，返回使用本地或全局变量，这种情况下defer语句可以引用到返回值，但不会改变返回值。

一个返回本地变量的函数，如下所示：

```
func foo() int {
    var i int

    defer func() {
        i++
    }()

    return i
}
```

上面的函数，返回一个局部变量，同时defer函数也会操作这个局部变量。对于匿名返回值来说，可以假定仍然有一个变量存储返回值，假定返回值变量为”anony”，上面的返回语句可以拆分成以下过程：

```
anony = i
i++
return
```

由于i是整型，会将值拷贝给anony，所以defer语句中修改i值，对函数返回值不造成影响。

3.3.4 主函数拥有具名返回值

主函声明语句中带名字的返回值，会被初始化成一个局部变量，函数内部可以像使用局部变量一样使用该返回值。如果defer语句操作该返回值，可能会改变返回结果。

一个影响函返回值的例子：

```
func foo() (ret int) {
    defer func() {
        ret++
    }()

    return 0
}
```

上面的函数拆解出来，如下所示：

```
ret = 0
ret++
return
```

函数真正返回前，在defer中对返回值做了+1操作，所以函数最终返回1。

4. defer实现原理

本节我们尝试了解一些defer的实现机制。

4.1 defer数据结构

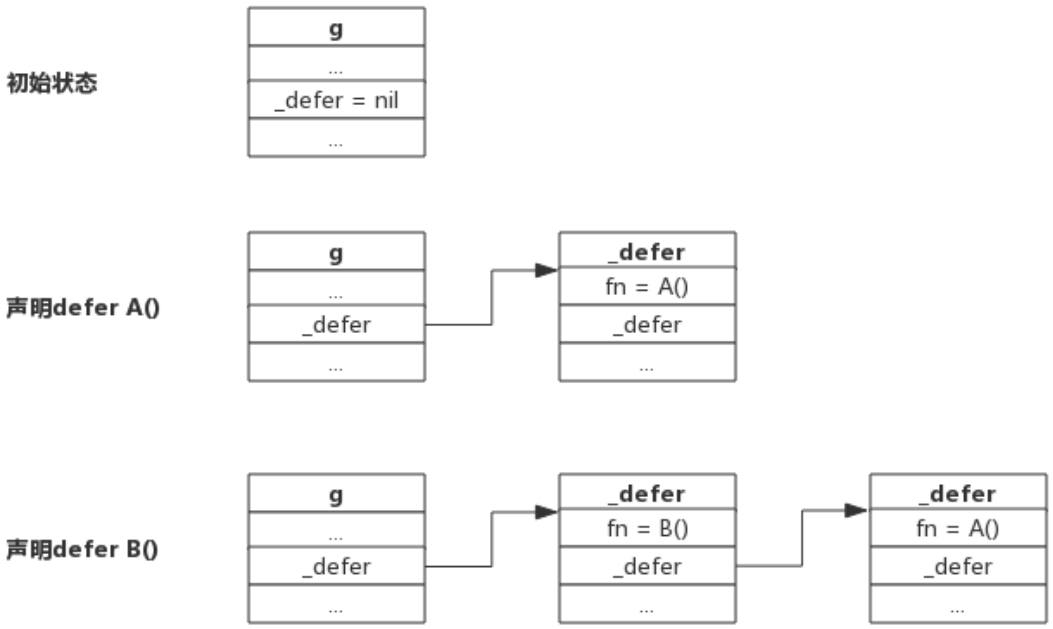
源码包 `src/src/runtime/runtime2.go:_defer` 定义了defer的数据结构：

```
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。

与函数不同的一点是它含有一个指针，可用于指向另一个defer，每个goroutine数据结构中实际上也有一个defer指针，该指针指向一个defer的单链表，每次声明一个defer时就将defer插入到单链表表头，每次执行defer时就从单链表表头取出一个defer执行。

下图展示多个defer被链接的过程：



从上图可以看到，新声明的defer总是添加到链表头部。

函数返回前执行defer则是从链表首部依次取出执行，不再赘述。

一个goroutine可能连续调用多个函数，defer添加过程跟上述流程一致，进入函数时添加defer，离开函数时取出defer，所以即便调用多个函数，也总是能保证defer是按LIFO方式执行的。

4.2 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，在编译阶段，声明defer处插入了函数deferproc()，在函数return前插入了函数deferreturn()。

5. 总结

- defer定义的延迟函数参数在defer语句出现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

defer 陷阱

前言

项目中，有时为了让程序更健壮，也即不 `panic` ，我们或许会使用 `recover()` 来接收异常并处理。

比如以下代码：

```
func NoPanic() {
    if err := recover(); err != nil {
        fmt.Println("Recover success...")
    }
}

func Dived(n int) {
    defer NoPanic()

    fmt.Println(1/n)
}
```

`func NoPanic()` 会自动接收异常，并打印相关日志，算是一个通用的异常处理函数。

业务处理函数中只要使用了 `defer NoPanic()` ，那么就不会再有 `panic` 发生。

关于是否应该使用recover接收异常，以及什么场景下使用等问题不在本节讨论范围内。
本节关注的是这种用法的一个变体，曾经出现在笔者经历的一个真实项目，在该变体下，recover再也无法接收异常。

recover使用误区

在项目中，有众多的数据库更新操作，正常的更新操作需要提交，而失败的就需要回滚，如果异常分支比较多，
就会有很多重复的回滚代码，所以有人尝试了一个做法：即在defer中判断是否出现异常，有异常则回滚，否则提交。

简化代码如下所示：

```
func IsPanic() bool {
    if err := recover(); err != nil {
        fmt.Println("Recover success...")
        return true
    }

    return false
}

func UpdateTable() {
    // defer中决定提交还是回滚
    defer func() {
        if IsPanic() {
            // Rollback transaction
        } else {
            // Commit transaction
        }
    }()
```

```
    // Database update operation...
}
```

`func IsPanic() bool` 用来接收异常，返回值用来说明是否发生了异常。 `func UpdateTable()` 函数中，使用defer来判断最终应该提交还是回滚。

上面代码初步看起来还算合理，但是此处的 `IsPanic()` 再也不会返回 `true` ，不是 `IsPanic()` 函数的问题，而是其调用的位置不对。

recover 失效的条件

上面代码 `IsPanic()` 失效了，其原因是违反了recover的一个限制，导致recover()失效（永远返回 `nil` ）。

以下三个条件会让recover()返回 `nil` ：

1. panic时指定的参数为 `nil` ；（一般panic语句如 `panic("xxx failed...")` ）
2. 当前协程没有发生panic；
3. recover没有被defer方法直接调用；

前两条都比较容易理解，上述例子正是匹配第3个条件。

本例中，recover() 调用栈为“defer （匿名）函数” -> IsPanic() -> recover()。也就是说，recover并没有被defer方法直接调用。符合第3个条件，所以recover() 永远返回nil。

> 赠人玫瑰手留余香，如果觉得不错请给个赞~
>
> 本篇文章已归档到GitHub项目，求星~ 点我即达

select

1. 前言

select是Golang在语言层面提供的多路IO复用的机制，其可以检测多个channel是否ready(即是否可读或可写)，使用起来非常方便。

本章试图根据源码总结其实现原理，从而发现一些使用误区或解释一些不太常见的现象。

2. 热身环节

我们先看几个题目，用于测试对select的了解程度，每个题目代表一个知识点，本章后面的部分会进行略为详细的介绍。

2.1 题目1

下面的程序输出是什么？

```
package main

import (
    "fmt"
    "time"
)

func main() {
    chan1 := make(chan int)
    chan2 := make(chan int)

    go func() {
        chan1 <- 1
        time.Sleep(5 * time.Second)
    }()

    go func() {
        chan2 <- 1
        time.Sleep(5 * time.Second)
    }()

    select {
    case <-chan1:
        fmt.Println("chan1 ready.")
    case <-chan2:
        fmt.Println("chan2 ready.")
    default:
        fmt.Println("default")
    }

    fmt.Println("main exit.")
}
```

程序中声明两个channel，分别为chan1和chan2，依次启动两个协程，分别向两个channel中写入一个数据就进入睡眠。select语句两个case分别检测chan1和chan2是否可读，如果都不可读则执行default语句。

参考答案：
select中各个case执行顺序是随机的，如果某个case中的channel已经ready，则执行相应的语句并退出select流程，如果所有case中的channel都未ready，则执行default中的语句然后退出select流程。另外，由于启动的协程和select语句并不能保证执行顺序，所以也有可能select执行时协程还未向channel中写入数据，所以select直接执行default语句并退出。所以，以下三种输出都有可能：

可能的输出一：

```
chan1 ready.
main exit.
```

可能的输出二：

```
chan2 ready.
main exit.
```

可能的输出三：

```
default
main exit.
```

2.2 题目2

下面的程序执行到select时会发生什么？

```
package main

import (
    "fmt"
    "time"
)

func main() {
    chan1 := make(chan int)
    chan2 := make(chan int)

    writeFlag := false
    go func() {
        for {
            if writeFlag {
                chan1 <- 1
            }
            time.Sleep(time.Second)
        }
    }()

    go func() {
        for {
            if writeFlag {
                chan2 <- 1
            }
            time.Sleep(time.Second)
        }
    }()

    select {
```

```
    case <-chan1:
        fmt.Println("chan1 ready.")
    case <-chan2:
        fmt.Println("chan2 ready.")
    }

    fmt.Println("main exit.")
}
```

程序中声明两个channel，分别为chan1和chan2，依次启动两个协程，协程会判断一个bool类型的变量writeFlag来决定是否要向channel中写入数据，由于writeFlag永远为false，所以实际上协程什么也没做。select语句两个case分别检测chan1和chan2是否可读，这个select语句不包含default语句。

参考答案：select会按照随机的顺序检测各case语句中channel是否ready，如果某个case中的channel已经ready则执行相应的case语句然后退出select流程，如果所有的channel都未ready且没有default的话，则会阻塞等待各个channel。所以上述程序会一直阻塞。

2.3 题目3

下面程序有什么问题？

```
package main

import (
    "fmt"
)

func main() {
    chan1 := make(chan int)
    chan2 := make(chan int)

    go func() {
        close(chan1)
    }()

    go func() {
        close(chan2)
    }()

    select {
    case <-chan1:
        fmt.Println("chan1 ready.")
    case <-chan2:
        fmt.Println("chan2 ready.")
    }

    fmt.Println("main exit.")
}
```

程序中声明两个channel，分别为chan1和chan2，依次启动两个协程，协程分别关闭两个channel。select语句两个case分别检测chan1和chan2是否可读，这个select语句不包含default语句。

参考答案：select会按照随机的顺序检测各case语句中channel是否ready，考虑到已关闭的channel也是可读的，所以上述程序中select不会阻塞，具体执行哪个case语句具是随机的。

2.4 题目4

下面程序会发生什么？

```
package main

func main() {
    select {
    }
}
```

上面程序中只有一个空的select语句。

参考答案：对于空的select语句，程序会被阻塞，准确的说是当前协程被阻塞，同时Golang自带死锁检测机制，当发现当前协程再也没有机会被唤醒时，则会panic。所以上述程序会panic。

3. 实现原理

Golang实现select时，定义了一个数据结构表示每个case语句(含defaut，default实际上是一种特殊的case)，select执行过程可以类比成一个函数，函数输入case数组，输出选中的case，然后程序流程转到选中的case块。

3.1 case数据结构

源码包 `src/runtime/select.go:scase` 定义了表示case语句的数据结构：

```
type scase struct {
    c           *hchan         // chan
    kind        uint16
    elem        unsafe.Pointer // data element
}
```

scase.c为当前case语句所操作的channel指针，这也说明了一个case语句只能操作一个channel。
scase.kind表示该case的类型，分为读channel、写channel和default，三种类型分别由常量定义：

- caseRecv：case语句中尝试读取scase.c中的数据；
- caseSend：case语句中尝试向scase.c中写入数据；
- caseDefault： default语句

scase.elem表示缓冲区地址，根据scase.kind不同，有不同的用途：

- scase.kind == caseRecv ： scase.elem表示读出channel的数据存放地址；
- scase.kind == caseSend ： scase.elem表示将要写入channel的数据存放地址；

3.2 select实现逻辑

源码包 `src/runtime/select.go:selectgo()` 定义了select选择case的函数：

```
func selectgo(cas0 *scase, order0 *uint16, ncases int) (int, bool)
```

函数参数：

- cas0为scase数组的首地址，selectgo()就是从这些scase中找出一个返回。
- order0为一个两倍cas0数组长度的buffer，保存scase随机序列pollorder和scase中channel地址序列lockorder

    - pollorder：每次selectgo执行都会把scase序列打乱，以达到随机检测case的目的。
    - lockorder：所有case语句中channel序列，以达到去重防止对channel加锁时重复加锁的目的。
- ncases表示scase数组的长度

函数返回值：

1. int： 选中case的编号，这个case编号跟代码一致
2. bool: 是否成功从channle中读取了数据，如果选中的case是从channel中读数据，则该返回值表示是否读取成功。

selectgo实现伪代码如下：

```
func selectgo(cas0 *scase, order0 *uint16, ncases int) (int, bool) {
    //1. 锁定scase语句中所有的channel
    //2. 按照随机顺序检测scase中的channel是否ready
    //   2.1 如果case可读，则读取channel中数据，解锁所有的channel，然后返回(case index, true)
    //   2.2 如果case可写，则将数据写入channel，解锁所有的channel，然后返回(case index, false)
    //   2.3 所有case都未ready，则解锁所有的channel，然后返回（default index, false）
    //3. 所有case都未ready，且没有default语句
    //   3.1 将当前协程加入到所有channel的等待队列
    //   3.2 当将协程转入阻塞，等待被唤醒
    //4. 唤醒后返回channel对应的case index
    //   4.1 如果是读操作，解锁所有的channel，然后返回(case index, true)
    //   4.2 如果是写操作，解锁所有的channel，然后返回(case index, false)
}
```

特别说明：对于读channel的case来说，如 `case elem, ok := <-chan1:` ，如果channel有可能被其他协程关闭的情况下，一定要检测读取是否成功，因为close的channel也有可能返回，此时ok == false。

4. 总结

- select语句中除default外，每个case操作一个channel，要么读要么写
- select语句中除default外，各case执行顺序是随机的
- select语句中如果没有default语句，则会阻塞等待任一case
- select语句中读操作要判断是否成功读取，关闭的channel也可以读取

range

1. 前言

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。

探索range的实现机制是很有意思的事情，这可能会改变你使用range的习惯。

2. 热身

按照惯例，我们看几个有意思的题目，用于检测对range的了解程度。

2.1 题目一：切片遍历

下面函数通过遍历切片，打印切片的下标和元素值，请问性能上有没有可优化的空间？

```
func RangeSlice(slice []int) {
    for index, value := range slice {
        _, _ = index, value
    }
}
```

程序解释：
函数中使用for-range对切片进行遍历，获取切片的下标和元素素值，这里忽略函数的实际意义。

参考答案：
遍历过程中每次迭代会对index和value进行赋值，如果数据量大或者value类型为string时，对value的赋值操作可能是多余的，可以在for-range中忽略value值，使用slice[index]引用value值。

2.2 题目二：Map遍历

下面函数通过遍历Map，打印Map的key和value，请问性能上有没有可优化的空间？

```
func RangeMap(myMap map[int]string) {
    for key, _ := range myMap {
        _, _ = key, myMap[key]
    }
}
```

程序解释：
函数中使用for-range对map进行遍历，获取map的key值，并根据key值获取获取value值，这里忽略函数的实际意义。

参考答案：
函数中for-range语句中只获取key值，然后根据key值获取value值，虽然看似减少了一次赋值，但通过key值查找value值的性能消耗可能高于赋值消耗。能否优化取决于map所存储数据结构特征、结合实际情况进行。

2.3 题目三：动态遍历

请问如下程序是否能正常结束？

```
func main() {
    v := []int{1, 2, 3}
    for i:= range v {
        v = append(v, i)
    }
}
```

程序解释：
main()函数中定义一个切片v，通过range遍历v，遍历过程中不断向v中添加新的元素。

参考答案：
能够正常结束。循环内改变切片的长度，不影响循环次数，循环次数在循环开始前就已经确定了。

3. 实现原理

对于for-range语句的实现，可以从编译器源码中找到答案。
编译器源码 `gofrontend/go/statements.cc/For_range_statement::do_lower()` 方法中有如下注释。

```
// Arrange to do a loop appropriate for the type.  We will produce
//   for INIT ; COND ; POST {
//           ITER_INIT
//           INDEX = INDEX_TEMP
//           VALUE = VALUE_TEMP // If there is a value
//           original statements
//   }
```

可见range实际上是一个C风格的循环结构。range支持数组、数组指针、切片、map和channel类型，对于不同类型有些细节上的差异。

3.1 range for slice

下面的注释解释了遍历slice的过程：

```
// 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前会先获取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。

3.2 range for map

下面的注释解释了遍历map的过程：

```
// 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。

3.3 range for channel

遍历channel是最特殊的，这是由channel的实现机制决定的：

```
// 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。

4. 编程Tips

- 遍历过程中可以视情况放弃接收index或value，可以一定程度上提升性能
- 遍历channel时，如果channel中没有数据，可能会阻塞
- 尽量避免遍历过程中修改原数据

5. 总结

- for-range的实现实际上是C风格的for循环
- 使用index,value接收range返回值会发生一次数据拷贝

mutex

1. 前言

互斥锁是并发程序中对共享资源进行访问控制的主要手段，对此Go语言提供了非常简单易用的Mutex，Mutex为一结构体类型，对外暴露两个方法Lock()和Unlock()分别用于加锁和解锁。

Mutex使用起来非常方便，但其内部实现却复杂得多，这包括Mutex的几种状态。另外，我们也想探究一下Mutex重复解锁引起panic的原因。

按照惯例，本节内容从源码入手，提取出实现原理，又不会过分纠结于实现细节。

2. Mutex数据结构

2.1 Mutex结构体

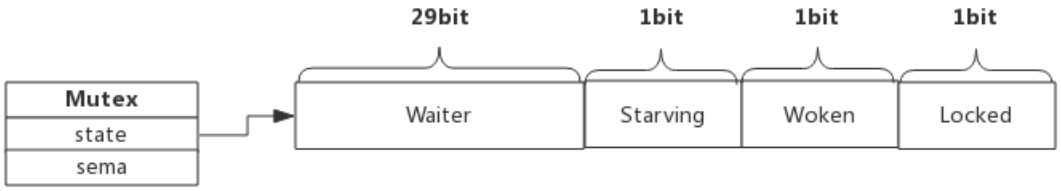
源码包 `src/sync/mutex.go:Mutex` 定义了互斥锁的数据结构：

```
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：



- 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是否处于饥饿状态，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。

2.2 Mutex方法

Mutex对外提供两个方法，实际上也只有这两个方法：

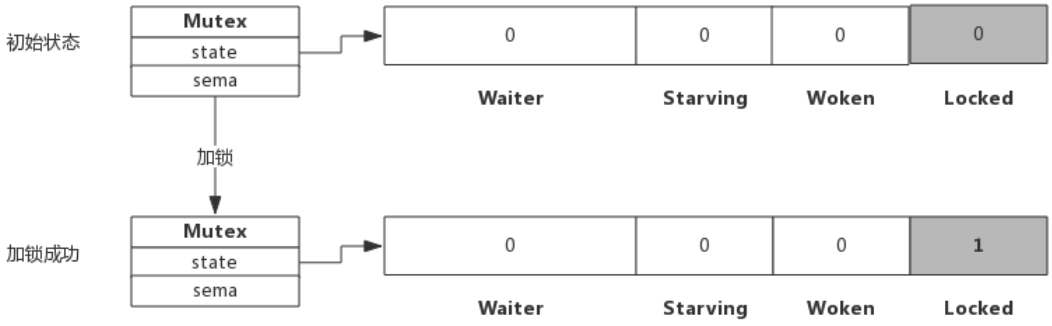
- Lock() : 加锁方法
- Unlock(): 解锁方法

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，解锁时根据是否有阻塞协程也有两种处理。

3. 加解锁过程

3.1 简单加锁

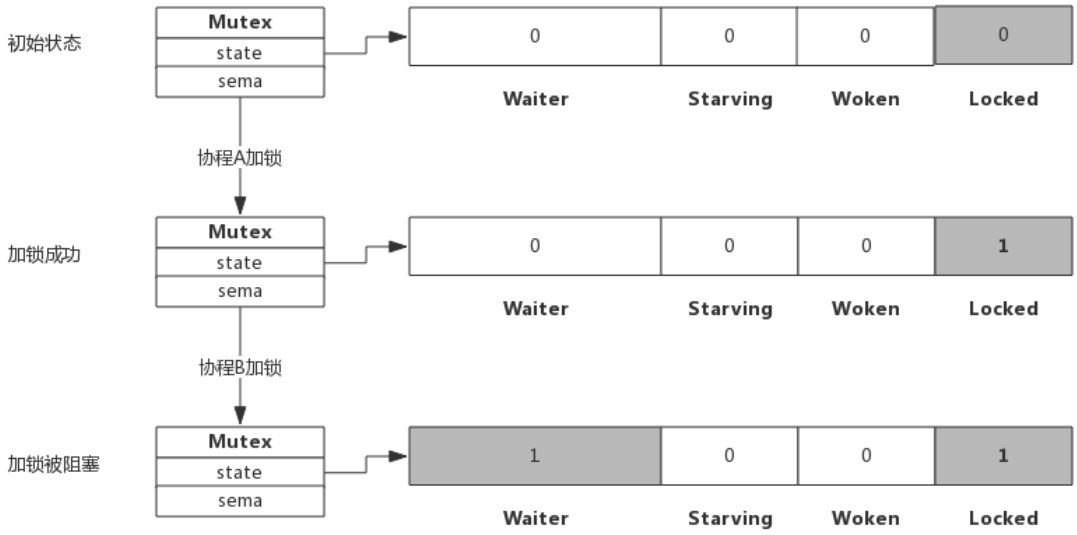
假定当前只有一个协程在加锁，没有其他协程干扰，那么过程如下图所示：



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。

3.2 加锁被阻塞

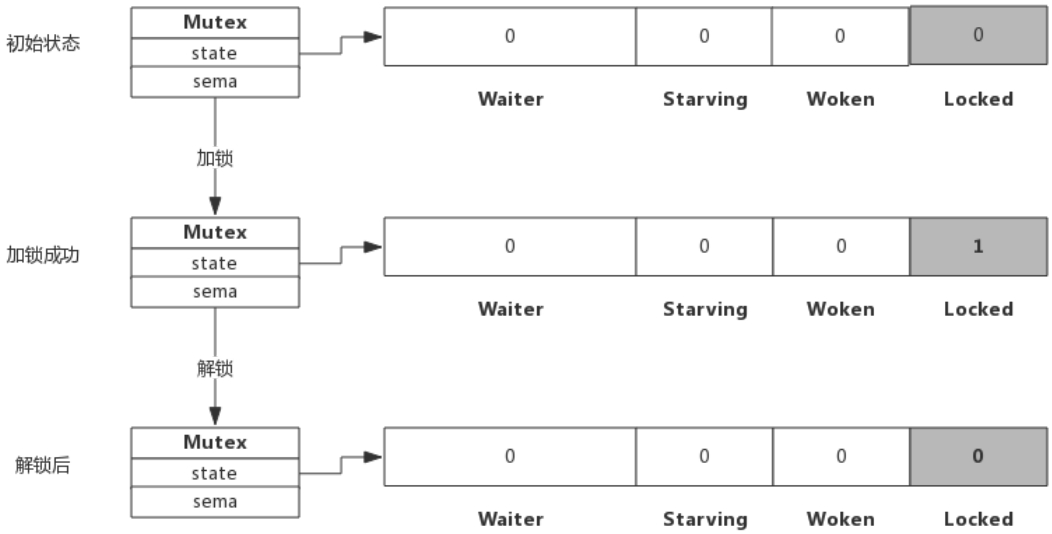
假定加锁时，锁已被其他协程占用了，此时加锁过程如下图所示：



从上图可看到，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。

3.3 简单解锁

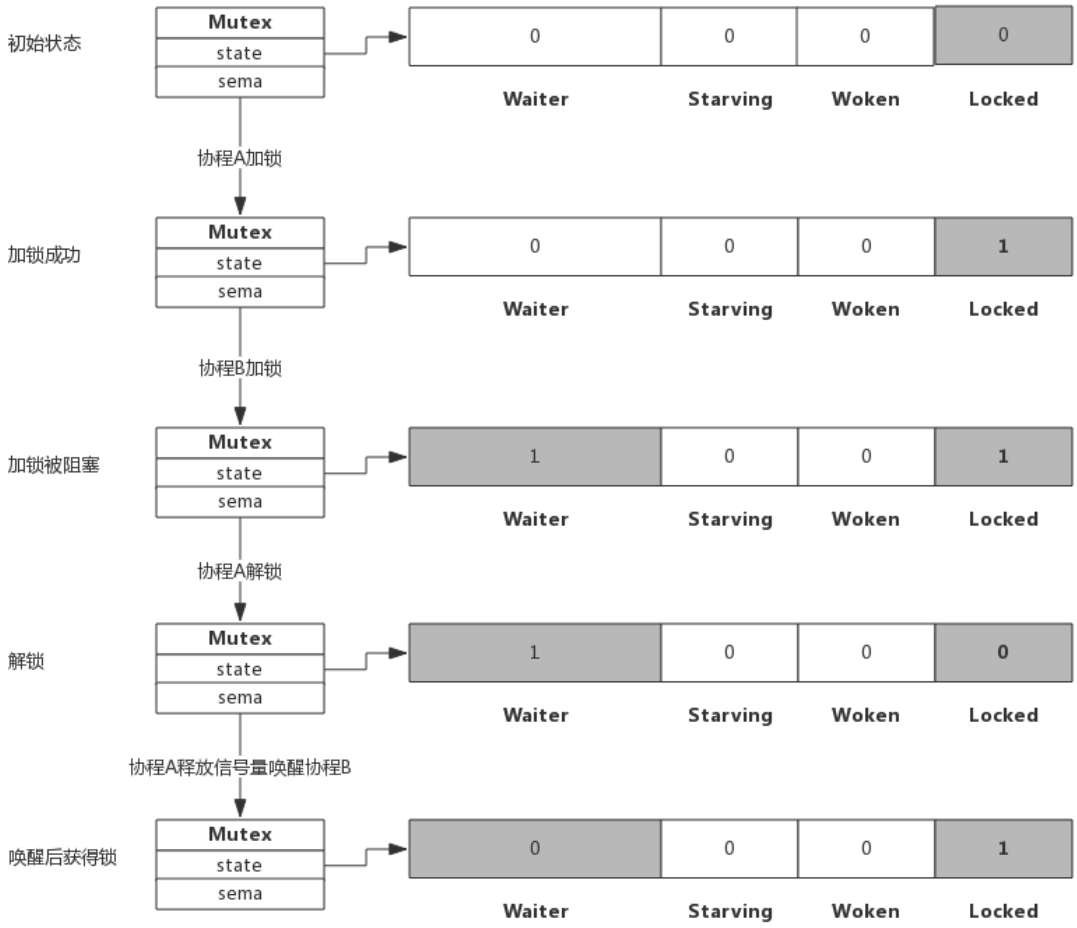
假定解锁时，没有其他协程阻塞，此时解锁过程如下图所示：



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。

3.4 解锁并唤醒协程

假定解锁时，有1个或多个协程阻塞，此时解锁过程如下图所示：



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。

4. 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。

自旋时间很短，但如果在自旋过程中发现锁已被释放，那么协程可以立即获取锁。此时即便有协程被唤醒也无法获取锁，只能再次阻塞。

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。

4.1 什么是自旋

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。

4.1 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。

4.2 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。

4.3 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。

5. 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。

4.1 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。

4.2 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。

5. Woken状态

Woken状态用于加锁和解锁过程的通信，举个例子，同一时刻，两个协程一个在加锁，一个在解锁，在加锁的协程可能在自旋过程中，此时把Woken标记为1，用于通知解锁协程不必释放信号量了，好比在说：你只管解锁好了，不必释放信号量，我马上就拿到锁了。

6. 为什么重复解锁要panic

可能你会想，为什么Go不能实现得更健壮些，多次执行Unlock()也不要panic？

仔细想想Unlock的逻辑就可以理解，这实际上很难做到。Unlock过程分为将Locked置为0，然后判断Waiter值，如果值>0，则释放信号量。

如果多次Unlock()，那么可能每次都释放一个信号量，这样会唤醒多个协程，多个协程唤醒后会继续在Lock()的逻辑里抢锁，势必会增加Lock()实现的复杂度，也会引起不必要的协程切换。

7. 编程Tips

7.1 使用defer避免死锁

加锁后立即使用defer对其解锁，可以有效的避免死锁。

7.2 加锁和解锁应该成对出现

加锁和解锁最好出现在同一个层次的代码块中，比如同一个函数。

重复解锁会引起panic，应避免这种操作的可能性。

rwmutex

1. 前言

前面我们聊了互斥锁Mutex，所谓读写锁RWMutex，完整的表述应该是读写互斥锁，可以说是Mutex的一个改进版，在某些场景下可以发挥更加灵活的控制能力，比如：读取数据频率远远大于写数据频率的场景。

例如，程序中写操作少而读操作多，简单的说，如果执行过程是1次写然后N次读的话，使用Mutex，这个过程将是串行的，因为即便N次读操作互相之间并不影响，但也都需要持有Mutex后才可以操作。如果使用读写锁，多个读操作可以同时持有锁，并发能力将大大提升。

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

下面我们将按照这个思路，即读写锁如何解决这些问题的，来分析读写锁的实现。

读写锁基于Mutex实现，实现源码非常简单和简洁，又有一定的技巧在里面。

2. 读写锁数据结构

2.1 类型定义

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
}
```

由以上数据结构可见，读写锁内部仍有一个互斥锁，用于将两个写操作隔离开来，其他的几个都用于隔离读操作和写操作。

下面我们简单看下RWMutex提供的4个接口，后面再根据使用场景具体分析这几个成员是如何配合工作的。

2.2 接口定义

RWMutex提供4个简单的接口来提供服务：

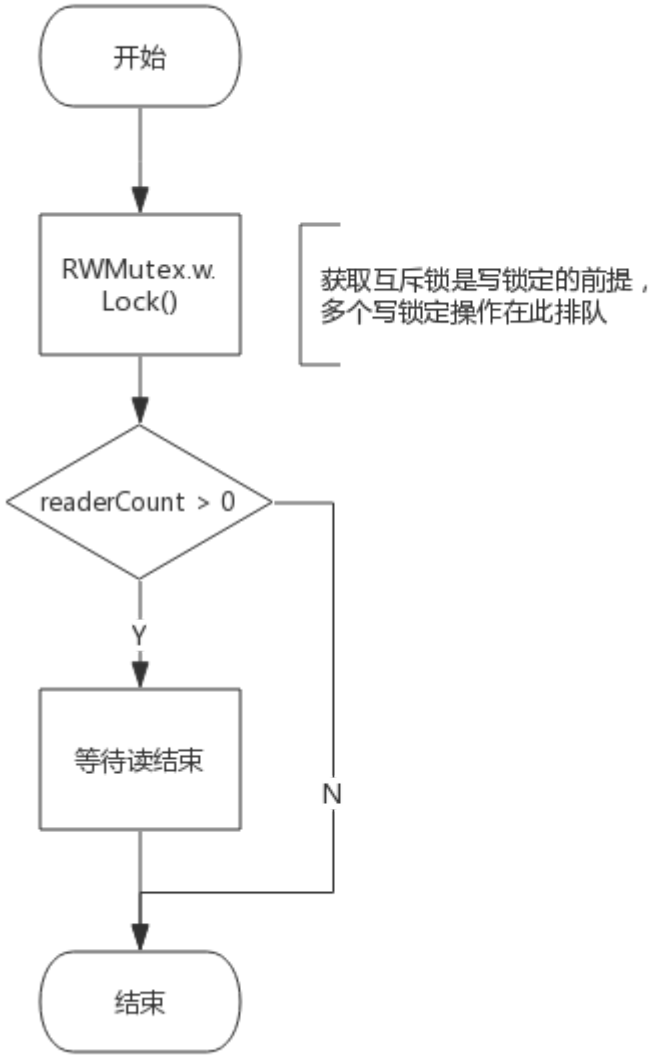
- RLock()：读锁定
- RUnlock()：解除读锁定
- Lock(): 写锁定，与Mutex完全一致
- Unlock()：解除写锁定，与Mutex完全一致

2.2.1 Lock()实现逻辑

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）

所以 `func (rw *RWMutex) Lock()` 接口实现流程如下图所示：

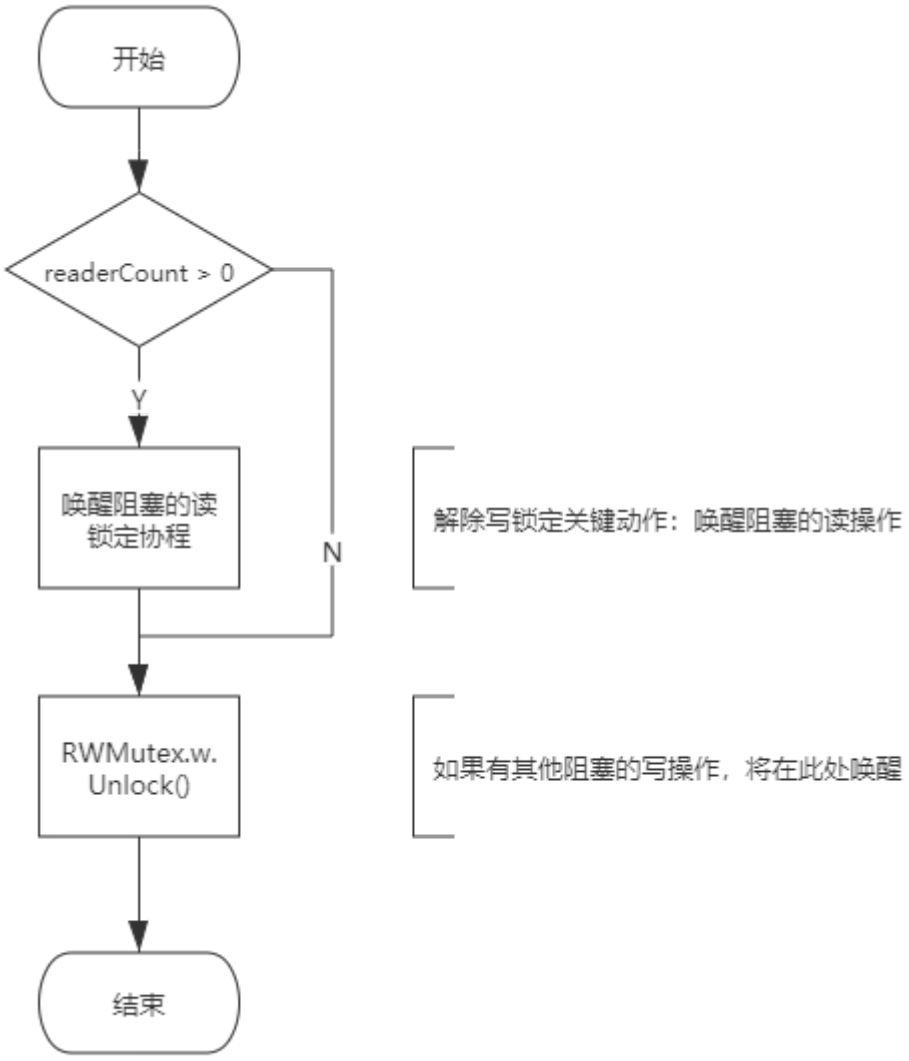


2.2.2 Unlock()实现逻辑

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

所以 `func (rw *RWMutex) Unlock()` 接口实现流程如下图所示：

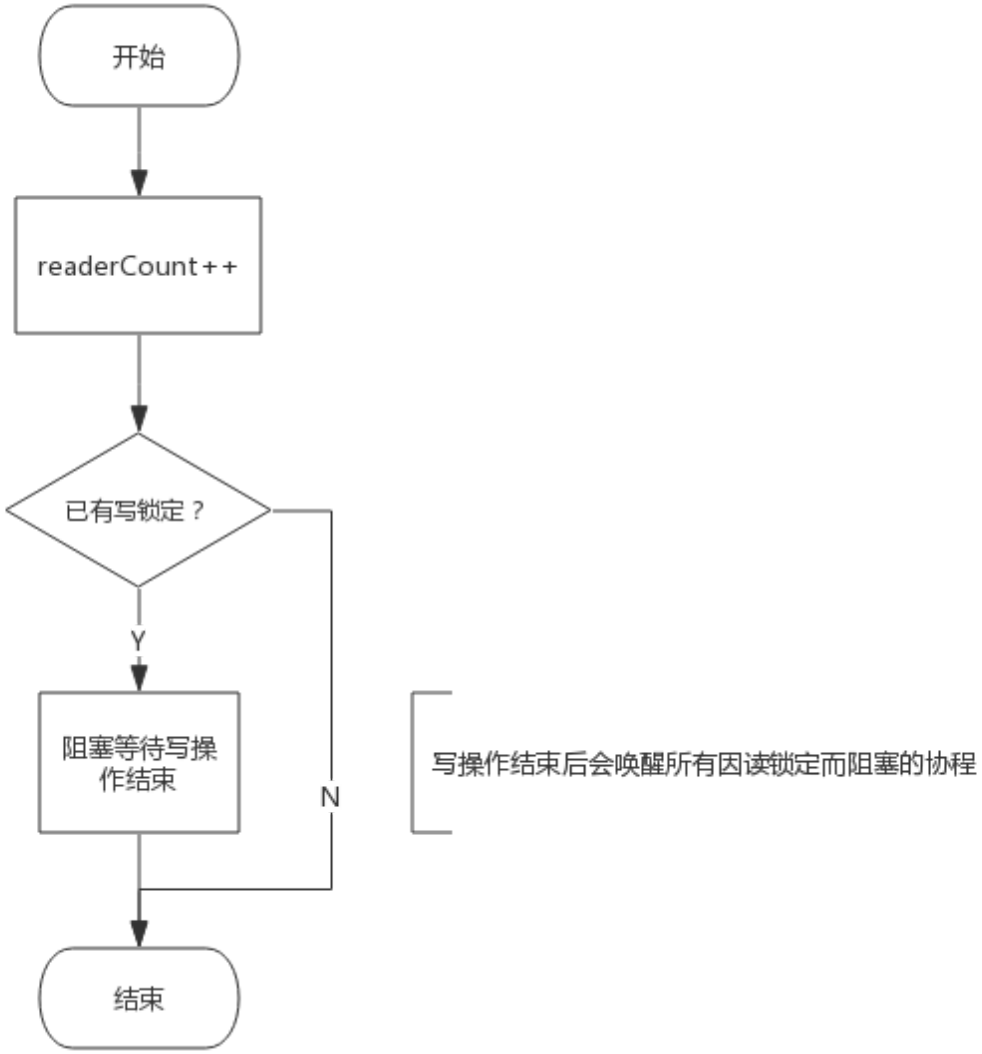


2.2.3 RLock()实现逻辑

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)

所以 `func (rw *RWMutex) RLock()` 接口实现流程如下图所示：

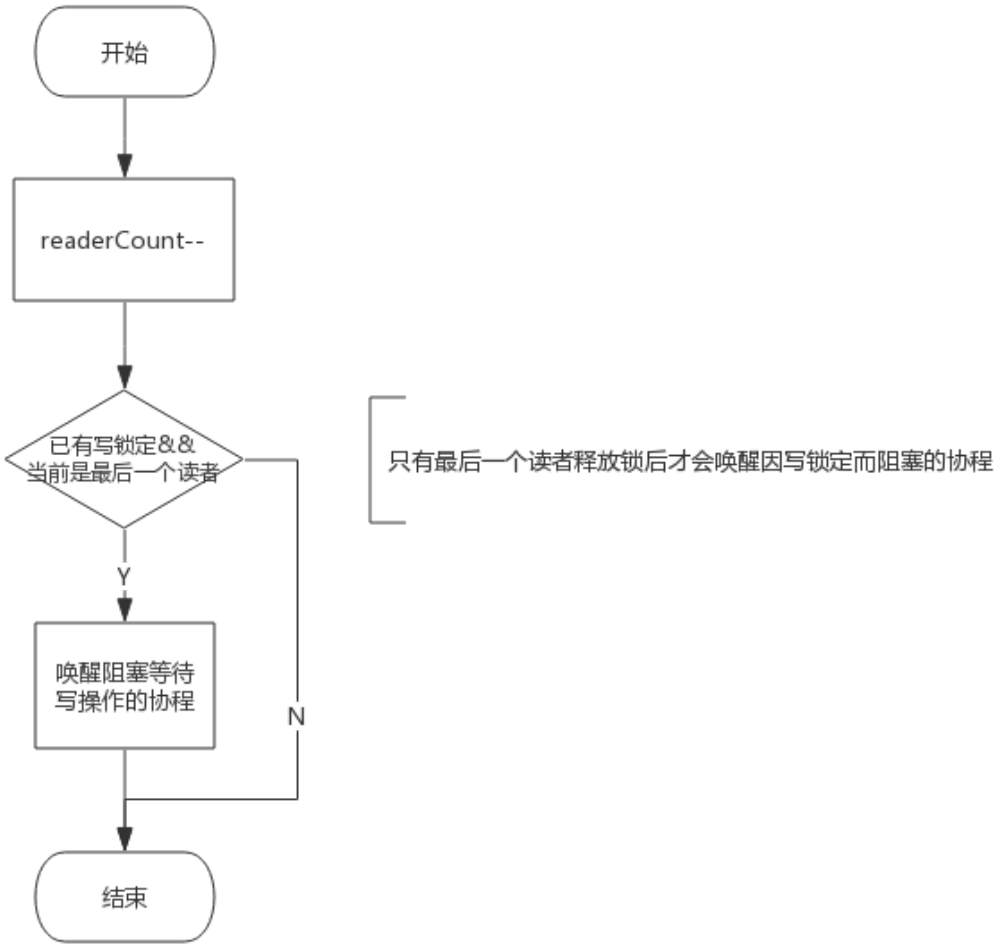


2.2.4 RUnlock()实现逻辑

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）

所以 `func (rw *RWMutex) RUnlock()` 接口实现流程如下图所示：



注意：即便有协程阻塞等待写操作，并不是所有的解除读锁定操作都会唤醒该协程，而是最后一个解除读锁定的协程才会释放信号量将该协程唤醒，因为只有当所有读操作的协程释放锁后才可以唤醒协程。

3. 场景分析

上面我们简单看了下4个接口实现原理，接下来我们看一下是如何解决前面提到的几个问题的。

3.1 写操作是如何阻止写操作的

读写锁包含一个互斥锁(Mutex)，写锁定必须要先获取该互斥锁，如果互斥锁已被协程A获取（或者协程A在阻塞等待读结束），意味着协程A获取了互斥锁，那么协程B只能阻塞等待该互斥锁。

所以，写操作依赖互斥锁阻止其他的写操作。

3.2 写操作是如何阻止读操作的

这个是读写锁实现中最精华的技巧。

我们知道RWMutex.readerCount是个整型值，用于表示读者数量，不考虑写操作的情况下，每次读锁定将该值+1，每次解除读锁定将该值-1，所以readerCount取值为[0, N]，N为读者个数，实际上最大可支持2^30个并发读者。

当写锁定进行时，会先将readerCount减去2^30，从而readerCount变成了负值，此时再有读锁定到来时检测到readerCount为负值，便知道有写操作在进行，只好阻塞等待。而真实的读操作个数并不会丢失，只需要将readerCount加上2^30即可获得。

所以，写操作将readerCount变成负值来阻止读操作的。

3.3 读操作是如何阻止写操作的

读锁定会先将RWMutext.readerCount加1，此时写操作到来时发现读者数量不为0，会阻塞等待所有读操作结束。

所以，读操作通过readerCount来将来阻止写操作的。

3.4 为什么写锁定不会被饿死

我们知道，写操作要等待读操作结束后才可以获得锁，写操作等待期间可能还有新的读操作持续到来，如果写操作等待所有读操作结束，很可能被饿死。然而，通过RWMutex.readerWait可完美解决这个问题。

写操作到来时，会把RWMutex.readerCount值拷贝到RWMutex.readerWait中，用于标记排在写操作前面的读者个数。

前面的读操作结束后，除了会递减RWMutex.readerCount，还会递减RWMutex.readerWait值，当RWMutex.readerWait值变为0时唤醒写操作。

所以，写操作就相当于把一段连续的读操作划分成两部分，前面的读操作结束后唤醒写操作，写操作结束后唤醒后面的读操作。如下图所示：

4. 关于源码

关于读写锁的实现源码，我添加了大量的中文注释，如有兴趣，请自行查看。
源码地址注解：https://github.com/RainbowMango/GoComments

协程

本章主要介绍协程及其调度机制。

协程是GO语言最大的特色之一，本章我们从协程的概念、GO协程的实现、GO协程调度机制等角度来分析。

协程调度

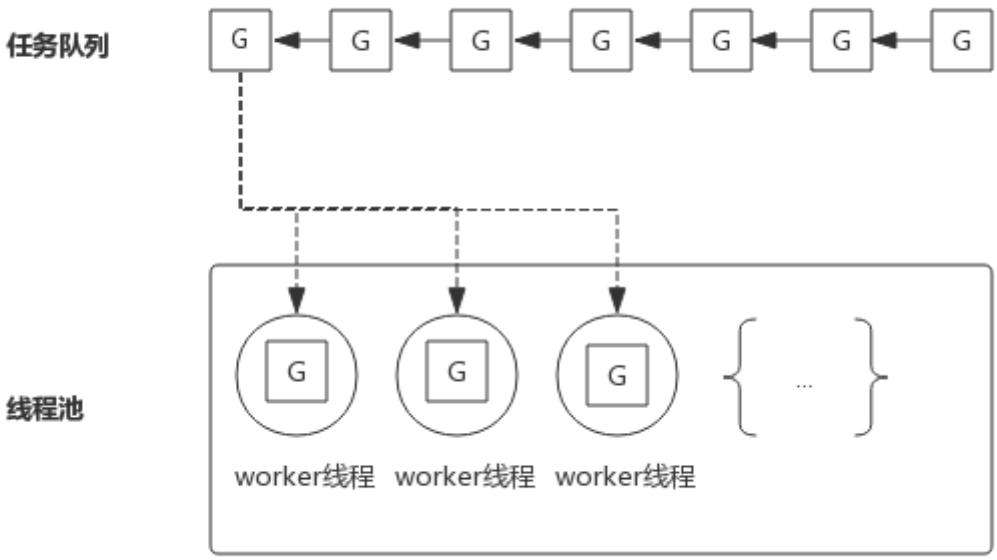
前言

Goroutine调度是一个很复杂的机制，尽管Go源码中提供了大量的注释，但对其原理没有一个好的理解的情况下去读源码收获不会很大。下面尝试用简单的语言描述一下Goroutine调度机制，在此基础上再去研读源码效果可能更好一些。

1. 线程池的缺陷

我们知道，在高并发应用中频繁创建线程会造成不必要的开销，所以有了线程池。线程池中预先保存一定数量的线程，而新任务将不再以创建线程的方式去执行，而是将任务发布到任务队列，线程池中的线程不断地从任务队列中取出任务并执行，可以有效的减少线程创建和销毁所带来的开销。

下图展示一个典型的线程池：

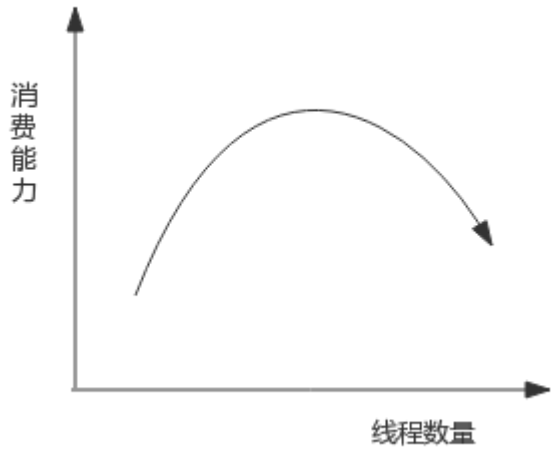


为了方便下面的叙述，我们把任务队列中的每一个任务称作G，而G往往代表一个函数。
线程池中的worker线程不断地从任务队列中取出任务并执行。而worker线程的调度则交给操作系统进行调度。

如果worker线程执行的G任务中发生系统调用，则操作系统会将该线程置为阻塞状态，也意味着该线程在怠工，也意味着消费任务队列的worker线程变少了，也就是说线程池消费任务队列的能力变弱了。

如果任务队列中的大部分任务都会进行系统调用，则会让这种状态恶化，大部分worker线程进入阻塞状态，从而任务队列中的任务产生堆积。

解决这个问题的一个思路就是重新审视线程池中线程的数量，增加线程池中线程数量可以一定程度上提高消费能力，但随着线程数量增多，由于过多线程争抢CPU，消费能力会有上限，甚至出现消费能力下降。 如下图所示：



2. Goroutine调度器

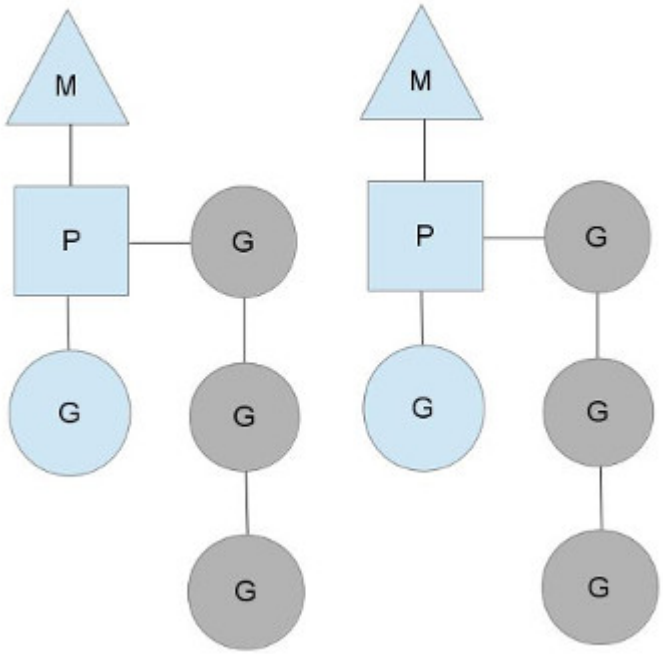
线程数过多，意味着操作系统会不断地切换线程，频繁的上下文切换就成了性能瓶颈。
Go提供一种机制，可以在线程中自己实现调度，上下文切换更轻量，从而达到了线程数少，而并发数并不少的效果。而线程中调度的就是Goroutine.

早期Go版本，比如1.9.2版本的源码注释中有关于调度器的解释。
Goroutine 调度器的工作就是把“ready-to-run”的goroutine分发到线程中。

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。

M必须拥有P才可以执行G中的代码，P含有一个包含多个G的队列，P可以调度G交由M执行。其关系如下图所示：



图中M是交给操作系统调度的线程，M持有一个P，P将G调度进M中执行。P同时还维护着一个包含G的队列（图中灰色部分），可以按照一定的策略将G调度到M中执行。

P的个数在程序启动时决定，默认情况下等同于CPU的核数，由于M必须持有一个P才可以运行Go代码，所以同时运行的M个数，也即线程数一般等同于CPU的个数，以达到尽可能的使用CPU而又不至于产生过多的线程切换开销。

程序中可以使用 `runtime.GOMAXPROCS()` 设置P的个数，在某些IO密集型的场景下可以在一定程度上提高性能。这个后面再详细介绍。

3. Goroutine调度策略

3.1 队列轮转

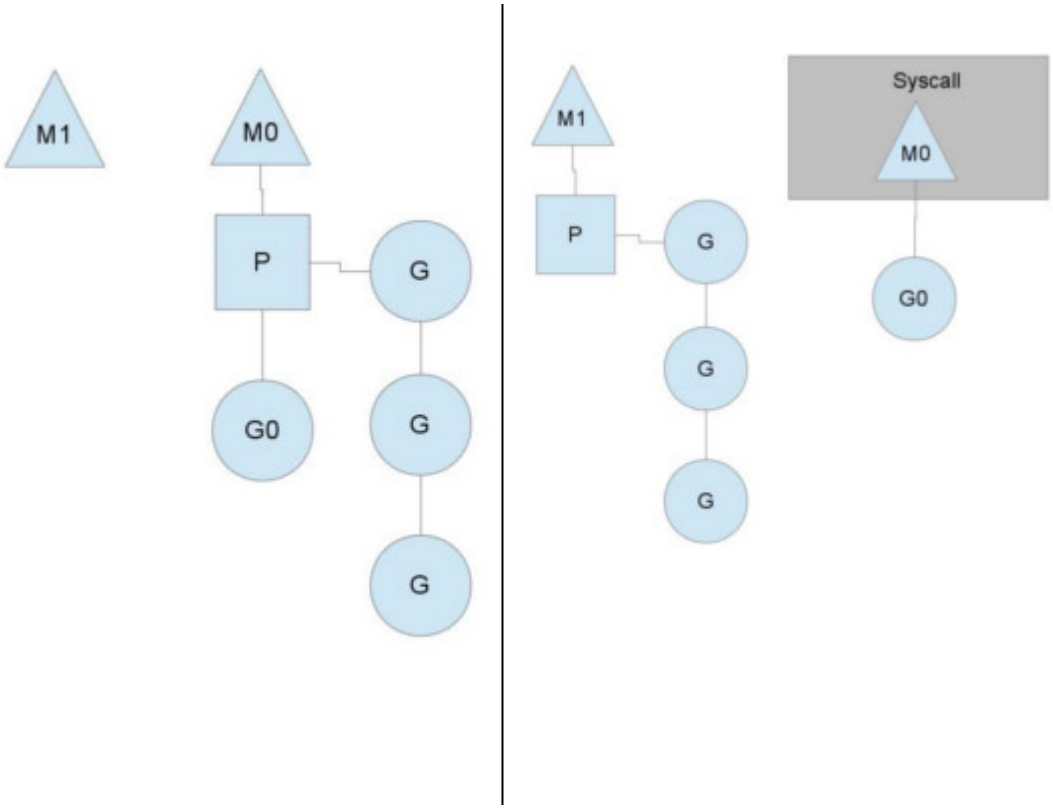
上图中可见每个P维护着一个包含G的队列，不考虑G进入系统调用或IO操作的情况下，P周期性的将G调度到M中执行，执行一小段时间，将上下文保存下来，然后将G放到队列尾部，然后从队列中重新取出一个G进行调度。

除了每个P维护的G队列以外，还有一个全局的队列，每个P会周期性地查看全局队列中是否有G待运行并将其调度到M中执行，全局队列中G的来源，主要有从系统调用中恢复的G。之所以P会周期性地查看全局队列，也是为了防止全局队列中的G被饿死。

3.2 系统调用

上面说到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。类似线程池，Go也提供一个M的池子，需要时从池子中获取，用完放回池子，不够用时就再创建一个。

当M运行的某个G产生系统调用时，如下图所示：



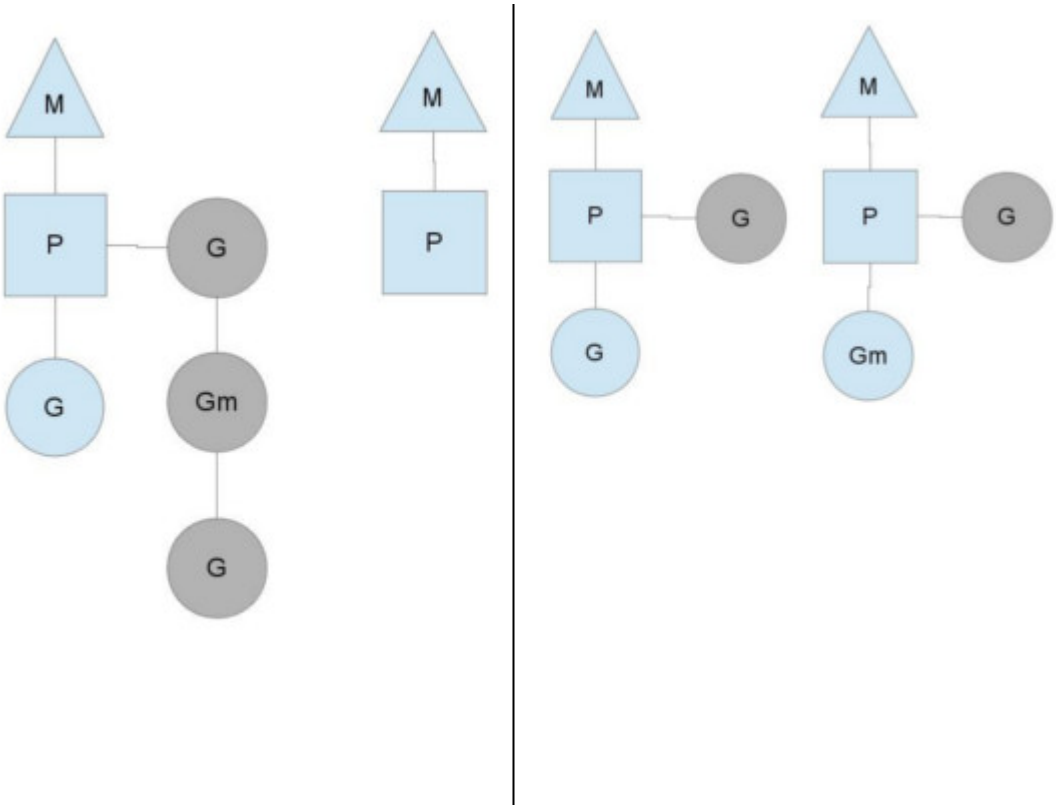
如图所示，当G0即将进入系统调用时，M0将释放P，进而某个空闲的M1获取P，继续执行P队列中剩下的G。而M0由于陷入系统调用而进被阻塞，M1接替M0的工作，只要P不空闲，就可以保证充分利用CPU。

M1的来源有可能是M的缓存池，也可能是新建的。当G0系统调用结束后，根据M0是否能获取到P，将会将G0做不同的处理：

1. 如果有空闲的P，则获取一个P，继续执行G0。
2. 如果没有空闲的P，则将G0放入全局队列，等待被其他的P调度。然后M0将进入缓存池睡眠。

3.3 工作量窃取

多个P中维护的G队列有可能是不均衡的，比如下图：



竖线左侧中右边的P已经将G全部执行完，然后去查询全局队列，全局队列中也没有G，而另一个M中除了正在运行的G外，队列中还有3个G待运行。此时，空闲的P会将其他P中的G偷取一部分过来，一般每次偷取一半。偷取完如右图所示。

4. GOMAXPROCS设置对性能的影响

一般来讲，程序运行时就将GOMAXPROCS大小设置为CPU核数，可让Go程序充分利用CPU。
在某些IO密集型的应用里，这个值可能并不意味着性能最好。
理论上当某个Goroutine进入系统调用时，会有一个新的M被启用或创建，继续占满CPU。
但由于Go调度器检测到M被阻塞是有一定延迟的，也即旧的M被阻塞和新的M得到运行之间是有一定间隔的，所以在IO密集型应用中不妨把GOMAXPROCS设置的大一些，或许会有好的效果。

5.参考文章

5.1 《The Go scheduler》http://morsmachine.dk/go-scheduler

内存管理

本章主要介绍GO语言的自动垃圾回收机制。

自动垃圾回收是GO语言最大的特色之一，也是很有争议的话题。因为自动垃圾回收解放了程序员，使其不用担心内存泄露的问题，争议在于垃圾回收的性能，在某些应用场景下垃圾回收会暂时停止程序运行。

本章从内存分配原理讲起，然后再看垃圾回收原理，最后再聊一些与垃圾回收性能优化相关的话题。

内存分配原理

1. 前言

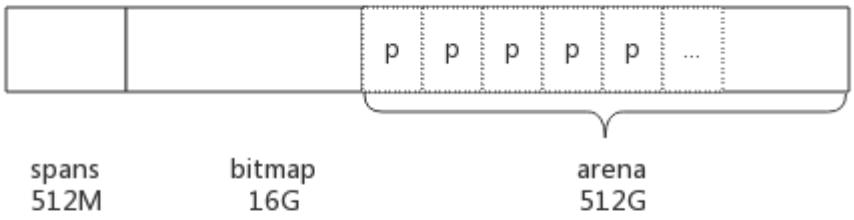
编写过C语言程序的肯定知道通过malloc()方法动态申请内存，其中内存分配器使用的是glibc提供的ptmalloc2。
除了glibc，业界比较出名的内存分配器有Google的tcmalloc和Facebook的jemalloc。二者在避免内存碎片和性能上均比glibc有比较大的优势，在多线程环境中效果更明显。

Golang中也实现了内存分配器，原理与tcmalloc类似，简单的说就是维护一块大的全局内存，每个线程(Golang中为P)维护一块小的私有内存，私有内存不足再从全局申请。

另外，内存分配与GC（垃圾回收）关系密切，所以了解GC前有必要了解内存分配的原理。

2. 基础概念

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。
以64位系统为例，Golang程序启动时会向系统申请的内存如下图所示：



预申请的内存划分为spans、bitmap、arena三部分。其中arena即为所谓的堆区，应用中需要的内存从这里分配。其中spans和bitmap是为了管理arena区而存在的。

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，每个指针对应一个或多个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。

2.1 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。

2.1.1 class

根据对象大小，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
//     6        80        8192      102           32
//     7        96        8192       85           32
//     8       112        8192       73           16
//     9       128        8192       64            0
//    10       144        8192       56          128
//    11       160        8192       51           32
//    12       176        8192       46           96
//    13       192        8192       42          128
//    14       208        8192       39           80
//    15       224        8192       36          128
//    16       240        8192       34           32
//    17       256        8192       32            0
//    18       288        8192       28          128
//    19       320        8192       25          192
//    20       352        8192       23           96
//    21       384        8192       21          128
//    22       416        8192       19          288
//    23       448        8192       18          128
//    24       480        8192       17           32
//    25       512        8192       16            0
//    26       576        8192       14          128
//    27       640        8192       12          512
//    28       704        8192       11          448
//    29       768        8192       10          512
//    30       896        8192        9          128
//    31      1024        8192        8            0
//    32      1152        8192        7          128
//    33      1280        8192        6          512
//    34      1408       16384       11          896
//    35      1536        8192        5          512
//    36      1792       16384        9          256
//    37      2048        8192        4            0
//    38      2304       16384        7          256
//    39      2688        8192        3          128
//    40      3072       24576        8            0
//    41      3200       16384        5          384
//    42      3456       24576        7          384
//    43      4096        8192        2            0
//    44      4864       24576        5          256
//    45      5376       16384        3          256
//    46      6144       24576        4            0
//    47      6528       32768        5          128
//    48      6784       40960        6          256
//    49      6912       49152        7          768
//    50      8192        8192        1            0
//    51      9472       57344        6          512
//    52      9728       49152        5          512
//    53     10240       40960        4            0
//    54     10880       32768        3          128
//    55     12288       24576        2            0
//    56     13568       40960        3          256
//    57     14336       57344        4            0
//    58     16384       16384        1            0
//    59     18432       73728        4            0
//    60     19072       57344        3          128
```

```
//     61      20480       40960       2           0
//     62      21760       65536       3         256
//     63      24576       24576       1           0
//     64      27264       81920       3         128
//     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。

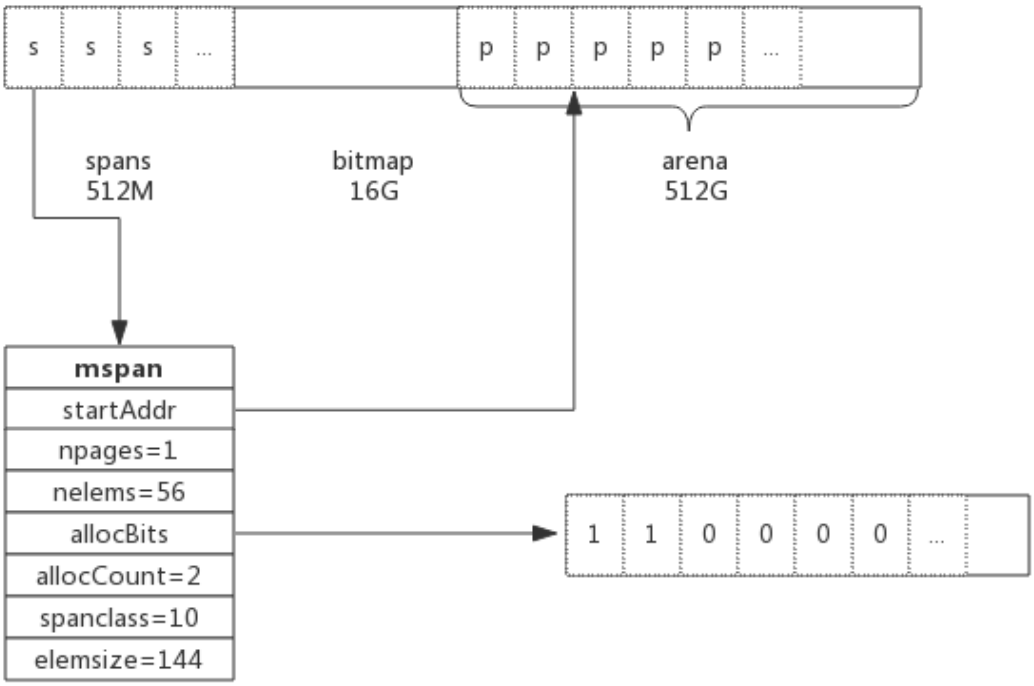
2.1.2 span数据结构

span是内存管理的基本单位，每个span用于管理特定的class对象，根据对象大小，span将一个或多个页拆分成多个块进行管理。

`src/runtime/mheap.go:mspan` 定义了其数据结构：

```
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
}
```

以class 10为例，span和管理的内存如下图所示：



spanclass为10，参照class表可得出npages=1,nelems=56,elemsize为144。其中startAddr是在span初始化时就指定了某个页的地址。allocBits指向一个位图，每位代表一个块是否被分配，本例中有两个块已经被分配，其allocCount也为2。

next和prev用于将多个span链接起来，这有利于管理多个span，接下来会进行说明。

2.2 cache

有了管理内存的基本单位span，还要有个数据结构来管理span，这个数据结构叫mcentral，各线程需要内存时从mcentral管理的span中申请内存，为了避免多线程申请内存时不断地加锁，Golang为每个线程分配了span的缓存，这个缓存即是cache。

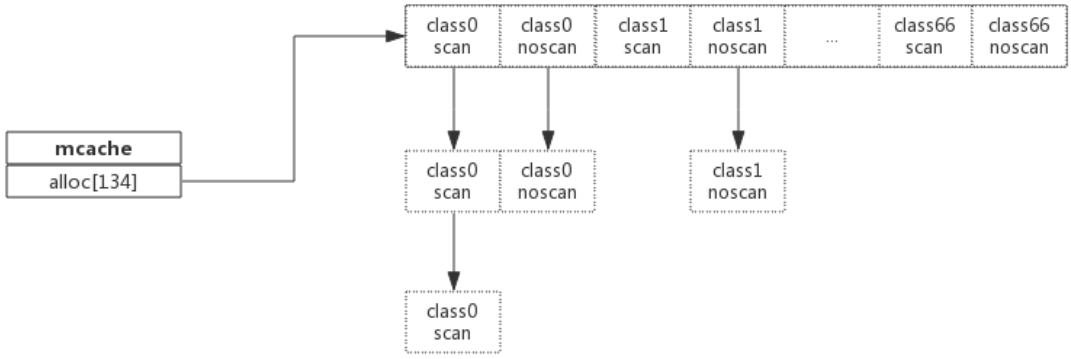
`src/runtime/mcache.go:mcache` 定义了cache的数据结构：

```
type mcache struct {
    alloc [67*2]*mspan // 按class分组的mspan列表
}
```

alloc为mspan的指针数组，数组大小为class总数的2倍。数组中每个元素代表了一种class类型的span列表，每种class类型都有两组span列表，第一组列表中所表示的对象中包含了指针，第二组列表中所表示的对象不含有指针，这么做是为了提高GC扫描性能，对于不包含指针的span列表，没必要去扫描。

根据对象是否包含指针，将对象分为noscan和scan两类，其中noscan代表没有指针，而scan则代表有指针，需要GC进行扫描。

mcache和span的对应关系如下图所示：



mcache在初始化时是没有任何span的，在使用过程中会动态地从central中获取并缓存下来，根据使用情况，每种class的span个数也不相同。上图所示，class 0的span数比class1的要多，说明本线程中分配的小对象要多一些。

2.3 central

cache作为线程的私有资源为单个线程服务，而central则是全局资源，为多个线程服务，当某个线程内存不足时会向central申请，当某个线程释放内存时又会回收进central。

`src/runtime/mcentral.go:mcentral` 定义了central数据结构：

```
type mcentral struct {
    lock      mutex     //互斥锁
    spanclass spanClass // span class ID
    nonempty  mSpanList // non-empty 指还有空闲块的span列表
    empty     mSpanList // 指没有空闲块的span列表

    nmalloc uint64      // 已累计分配的对象个数
}
```

- lock: 线程间互斥锁，防止多线程读写冲突
- spanclass：每个mcentral管理着一组有相同class的span列表
- nonempty: 指还有内存可用的span列表
- empty: 指没有内存可用的span列表
- nmalloc: 指累计分配的对象个数

线程从central获取span步骤如下：

1. 加锁
2. 从nonempty列表获取一个可用span，并将其从链表中删除
3. 将取出的span放入empty链表
4. 将span返回给线程
5. 解锁
6. 线程将该span缓存进cache

线程将span归还步骤如下：

1. 加锁
2. 将span从empty列表删除
3. 将span加入noneempty列表
4. 解锁

上述线程从central中获取span和归还span只是简单流程，为简单起见，并未对具体细节展开。

2.4 heap

从mcentral数据结构可见，每个mcentral对象只管理特定的class规格的span。事实上每种class都会对应一个mcentral,这个mcentral的集合存放于mheap数据结构中。

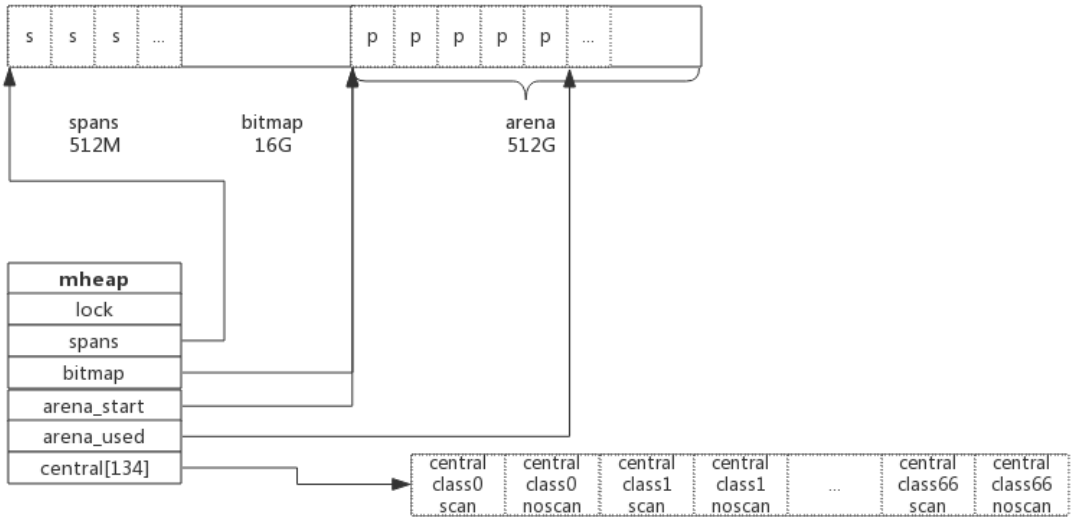
`src/runtime/mheap.go:mheap` 定义了heap的数据结构：

```
type mheap struct {
    lock      mutex

    spans []*mspan

    bitmap        uintptr     //指向bitmap首地址，bitmap是从高地址向低地址增长的

    arena_start uintptr         //指示arena区首地址
    arena_used  uintptr         //指示arena区已使用地址位置

    central [67*2]struct {
        mcentral mcentral
        pad      [sys.CacheLineSize - unsafe.Sizeof(mcentral{})%sys.CacheLineSize]byte
    }
}
```

- lock： 互斥锁
- spans: 指向spans区域，用于映射span和page的关系
- bitmap：bitmap的起始地址
- arena_start: arena区域首地址
- arena_used: 当前arena已使用区域的最大地址
- central: 每种class对应的两个mcentral

从数据结构可见，mheap管理着全部的内存，事实上Golang就是通过一个mheap类型的全局变量进行内存管理的。

mheap内存管理示意图如下：



系统预分配的内存分为spans、bitmap、arean三个区域，通过mheap管理起来。接下来看内存分配过程。

3. 内存分配过程

针对待分配对象的大小不同有不同的分配逻辑：

- (0, 16B) 且不包含指针的对象： Tiny分配
- (0, 16B) 包含指针的对象：正常分配
- [16B, 32KB] : 正常分配
- (32KB, -) : 大对象分配

  其中Tiny分配和大对象分配都属于内存管理的优化范畴，这里暂时仅关注一般的分配方法。

以申请size为n的内存为例，分配步骤如下：

1. 获取当前线程的私有缓存mcache
2. 根据size计算出适合的class的ID
3. 从mcache的alloc[class]链表中查询可用的span
4. 如果mcache没有可用的span则从mcentral申请一个新的span加入mcache中
5. 如果mcentral中也没有可用的span则从mheap中申请一个新的span加入mcentral
6. 从该span中获取到空闲对象地址并返回

4. 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

垃圾回收原理

1. 前言

所谓垃圾就是不再需要的内存块，这些垃圾如果不清理就没办法再次被分配使用，在不支持垃圾回收的编程语言里，这些垃圾内存就是泄露的内存。

Golang的垃圾回收（GC）也是内存管理的一部分，了解垃圾回收最好先了解前面介绍的内存分配原理。

2. 垃圾回收算法

业界常见的垃圾回收算法有以下几种：

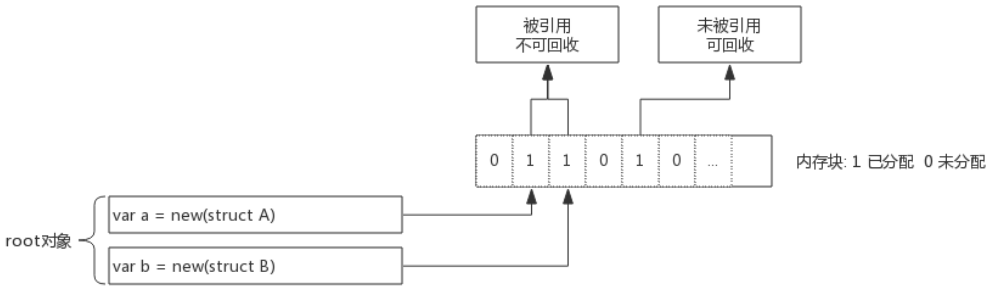
- 引用计数：对每个对象维护一个引用计数，当引用该对象的对象被销毁时，引用计数减1，当引用计数器为0时回收该对象。
  - 优点：对象可以很快地被回收，不会出现内存耗尽或达到某个阀值时才回收。
  - 缺点：不能很好地处理循环引用，而且实时维护引用计数，也有一定的代价。
  - 代表语言：Python、PHP、Swift
- 标记-清除：从根变量开始遍历所有引用的对象，引用的对象标记为”被引用”，没有被标记的进行回收。
  - 优点：解决了引用计数的缺点。
  - 缺点：需要STW，即要暂时停掉程序运行。
  - 代表语言：Golang（其采用三色标记法）
- 分代收集：按照对象生命周期长短划分不同的代空间，生命周期长的放入老年代，而短的放入新生代，不同代有不同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

3. Golang垃圾回收

3.1 垃圾回收原理

简单的说，垃圾回收的核心就是标记出哪些内存还在使用中（即被引用到），哪些内存不再使用了（即未被引用），把未被引用的内存回收掉，以供后续内存分配时使用。

下图展示了一段内存，内存中既有已分配掉的内存，也有未分配的内存，垃圾回收的目标就是把那些已经分配的但没有对象引用的内存找出来并回收掉：

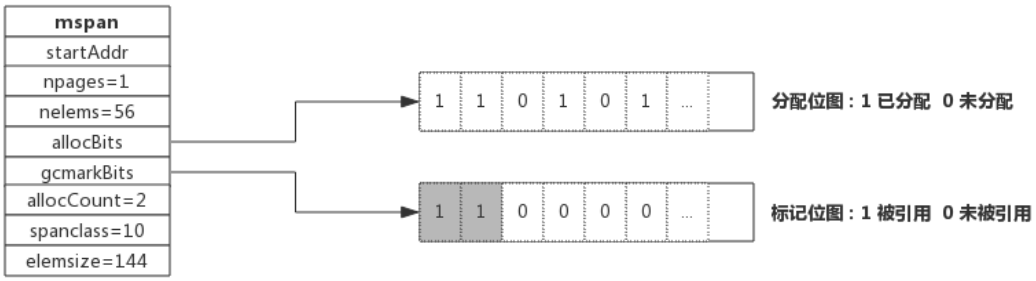


上图中，内存块1、2、4号位上的内存块已被分配（数字1代表已被分配，0 未分配）。变量a, b为一指针，指向内存的1、2号位。内存块的4号位曾经被使用过，但现在没有任何对象引用了，就需要被回收掉。

垃圾回收开始时从root对象开始扫描，把root对象引用的内存标记为”被引用”，考虑到内存块中存放的可能是指针，所以还需要递归的进行标记，全部标记完成后，只保留被标记的内存，未被标记的全部标识为未分配即完成了回收。

3.2 内存标记(Mark)

前面介绍内存分配时，介绍过span数据结构，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。在span数据结构中还有另一个位图gcmarkBits用于标记内存块被引用情况。



如上图所示，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。

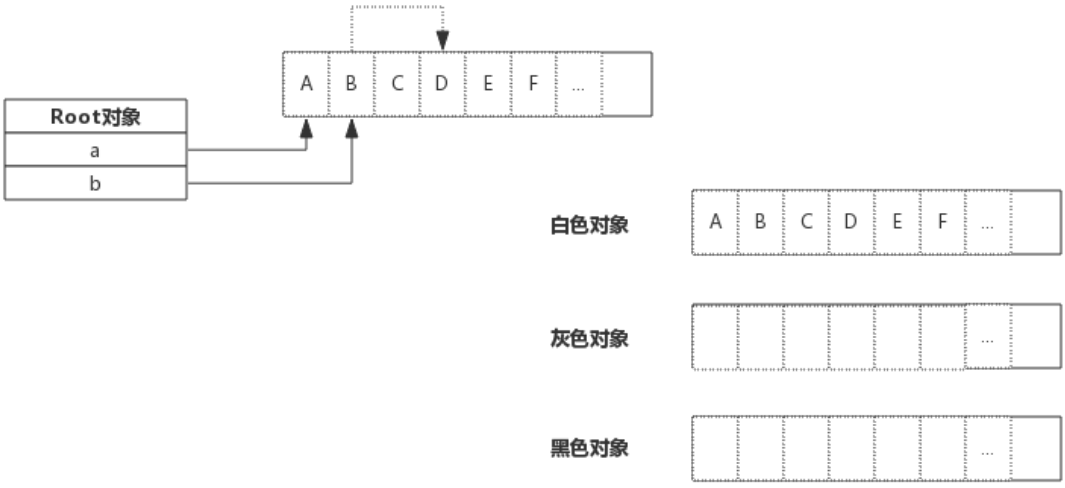
3.3 三色标记法

前面介绍了对象标记状态的存储方式，还需要有一个标记队列来存放待标记的对象，可以简单想象成把对象从标记队列中取出，将对象的引用状态标记在span的gcmarkBits，把对象引用到的其他对象再放入队列中。

三色只是为了叙述上方便抽象出来的一种说法，实际上对象并没有颜色之分。这里的三色，对应了垃圾回收过程中对象的三种状态：

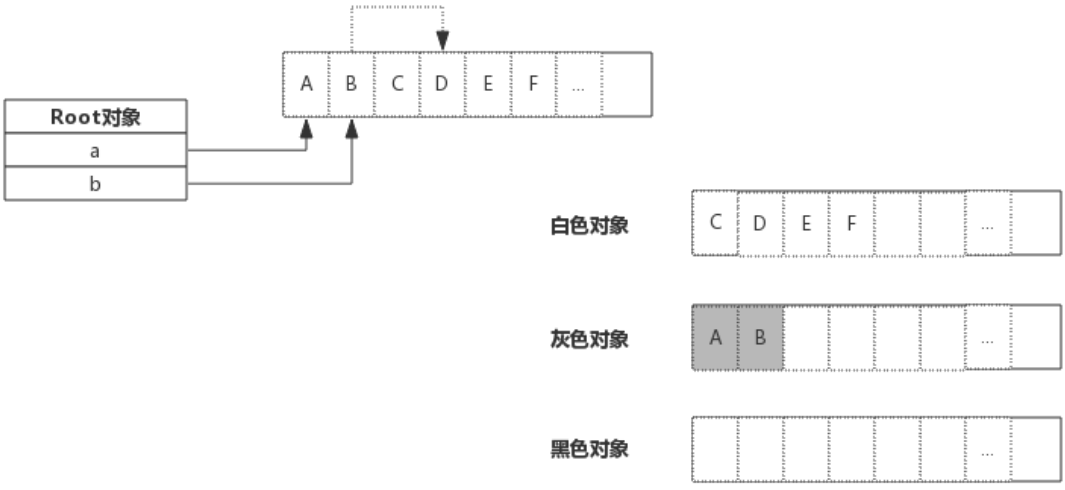
- 灰色：对象还在标记队列中等待
- 黑色：对象已被标记，gcmarkBits对应的位为1（该对象不会在本次GC中被清理）
- 白色：对象未被标记，gcmarkBits对应的位为0（该对象将会在本次GC中被清理）

例如，当前内存中有A~F一共6个对象，根对象a,b本身为栈上分配的局部变量，根对象a、b分别引用了对象A、B, 而B对象又引用了对象D，则GC开始前各对象的状态如下图所示:

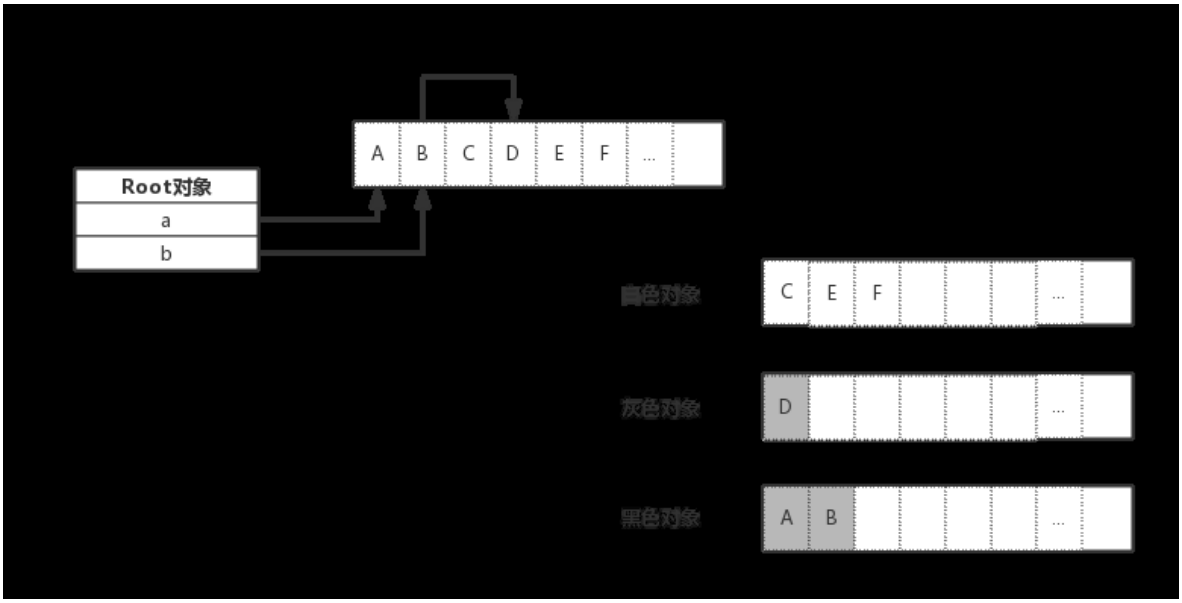


初始状态下所有对象都是白色的。

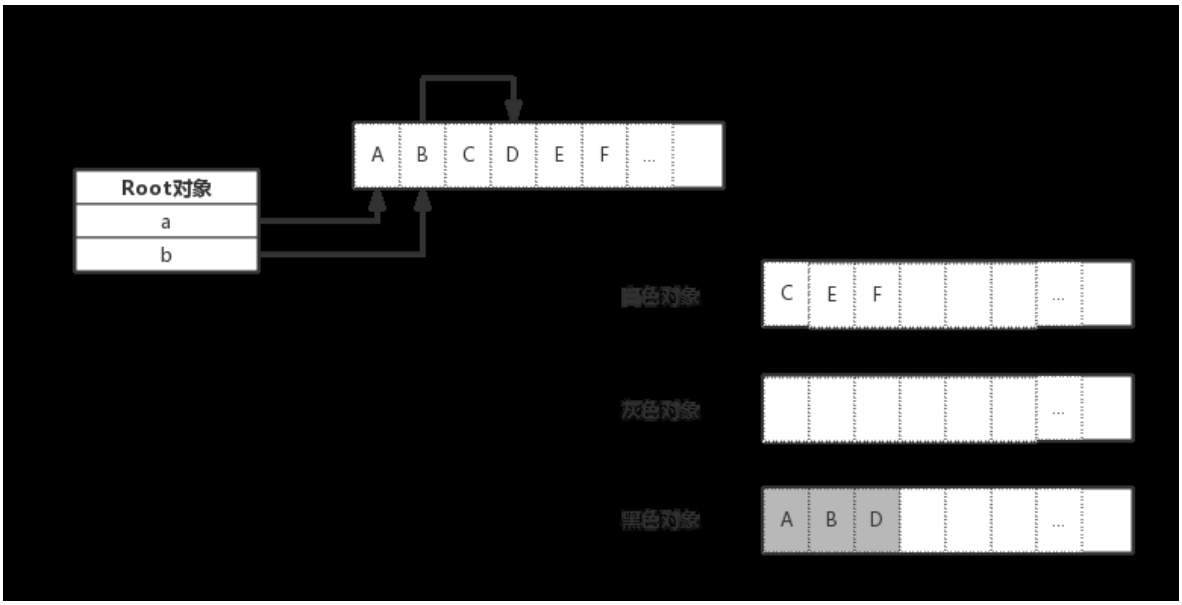
接着开始扫描根对象a、b:



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。如下图所示：



上图中灰色对象只有D，由于D没有引用其他对象，所以D转入黑色。标记过程结束：



最终，黑色的对象会被保留下来，白色对象会被回收掉。

3.4 Stop The World

印度电影《苏丹》中有句描述摔跤的一句台词是：“所谓摔跤，就是把对手控制住，然后摔倒他。”

对于垃圾回收来说，回收过程中也需要控制住内存的变化，否则回收过程中指针传递会引起内存引用关系变化，如果错误的回收了还在使用的内存，结果将是灾难性的。

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。

STW时间的长短直接影响了应用的执行，时间过长对于一些web应用来说是不可接受的，这也是广受诟病的原因之一。

为了缩短STW的时间，Golang不断优化垃圾回收算法，这种情况得到了很大的改善。

4. 垃圾回收优化

4.1 写屏障(Write Barrier)

前面说过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。

4.2 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。

5. 垃圾回收触发时机

5.1 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。

5.2 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
```

5.3 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。

6. 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。

关于GC性能优化的具体方法，后面单独介绍。

逃逸分析

1 前言

所谓逃逸分析（Escape analysis）是指由编译器决定内存分配的位置，不需要程序员指定。
函数中申请一个新的对象

- 如果分配在栈中，则函数执行结束可自动将内存回收；
- 如果分配在堆中，则函数执行结束可交给GC（垃圾回收）处理；

有了逃逸分析，返回函数局部变量将变得可能，除此之外，逃逸分析还跟闭包息息相关，了解哪些场景下对象会逃逸至关重要。

2 逃逸策略

每当函数中申请新的对象，编译器会根据该对象是否被函数外部引用来决定是否逃逸：

1. 如果函数外部没有引用，则优先放到栈中；
2. 如果函数外部存在引用，则必定放到堆中；

注意，对于函数外部没有引用的对象，也有可能放到堆中，比如内存过大超过栈的存储能力。

3 逃逸场景

3.1 指针逃逸

我们知道Go可以返回局部变量指针，这其实是一个典型的变量逃逸案例，示例代码如下：

```
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)
}
```

函数StudentRegister()内部s为局部变量，其值通过函数返回值返回，s本身为一指针，其指向的内存地址不会是栈而是堆，这就是典型的逃逸案例。

通过编译参数-gcflag=-m可以查看编译过程中的逃逸分析：

```
D:\SourceCode\GoExpert\src>go build -gcflags=-m
# _/D_/SourceCode/GoExpert/src
.\main.go:8: can inline StudentRegister
.\main.go:17: can inline main
.\main.go:18: inlining call to StudentRegister
.\main.go:8: leaking param: name
.\main.go:9: new(Student) escapes to heap
.\main.go:18: main new(Student) does not escape
```

可见在StudentRegister()函数中，也即代码第9行显示”escapes to heap”，代表该行内存分配发生了逃逸现象。

3.2 栈空间不足逃逸

看下面的代码，是否会产生逃逸呢？

```
package main

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

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

func main() {
    Slice()
}
```

上面代码Slice()函数中分配了一个1000个长度的切片，是否逃逸取决于栈空间是否足够大。
直接查看编译提示，如下：

```
D:\SourceCode\GoExpert\src>go build -gcflags=-m
# _/D_/SourceCode/GoExpert/src
.\main.go:4: Slice make([]int, 1000, 1000) does not escape
```

我们发现此处并没有发生逃逸。那么把切片长度扩大10倍即10000会如何呢?

```
D:\SourceCode\GoExpert\src>go build -gcflags=-m
# _/D_/SourceCode/GoExpert/src
.\main.go:4: make([]int, 10000, 10000) escapes to heap
```

我们发现当切片长度扩大到10000时就会逃逸。

实际上当栈空间不足以存放当前对象时或无法判断当前切片长度时会将对象分配到堆中。

3.3 动态类型逃逸

很多函数参数为interface类型，比如fmt.Println(a ...interface{})，编译期间很难确定其参数的具体类型，也会产生逃逸。
如下代码所示：

```
package main

import "fmt"

func main() {
    s := "Escape"
    fmt.Println(s)
}
```

上述代码s变量只是一个string类型变量，调用fmt.Println()时会产生逃逸：

```
D:\SourceCode\GoExpert\src>go build -gcflags=-m
# _/D_/SourceCode/GoExpert/src
.\main.go:7: s escapes to heap
.\main.go:7: main ... argument does not escape
```

3.4 闭包引用对象逃逸

某著名的开源框架实现了某个返回Fibonacci数列的函数：

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

该函数返回一个闭包，闭包引用了函数的局部变量a和b，使用时通过该函数获取该闭包，然后每次执行闭包都会依次输出Fibonacci数列。
完整的示例程序如下所示：

```
package main

import "fmt"

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

func main() {
    f := Fibonacci()

    for i := 0; i < 10; i++ {
        fmt.Printf("Fibonacci: %d\n", f())
    }
}
```

上述代码通过Fibonacci()获取一个闭包，每次执行闭包就会打印一个Fibonacci数值。输出如下所示：

```
D:\SourceCode\GoExpert\src>src.exe
Fibonacci: 1
Fibonacci: 1
Fibonacci: 2
Fibonacci: 3
Fibonacci: 5
Fibonacci: 8
Fibonacci: 13
Fibonacci: 21
Fibonacci: 34
Fibonacci: 55
```

Fibonacci()函数中原本属于局部变量的a和b由于闭包的引用，不得不将二者放到堆上，以致产生逃逸：

```
D:\SourceCode\GoExpert\src>go build -gcflags=-m
# _/D_/SourceCode/GoExpert/src
.\main.go:7: can inline Fibonacci.func1
.\main.go:7: func literal escapes to heap
.\main.go:7: func literal escapes to heap
.\main.go:8: &a escapes to heap
.\main.go:6: moved to heap: a
.\main.go:8: &b escapes to heap
.\main.go:6: moved to heap: b
.\main.go:17: f() escapes to heap
.\main.go:17: main ... argument does not escape
```

4 逃逸总结

- 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

5 编程Tips

思考一下这个问题：函数传递指针真的比传值效率高吗？
我们知道传递指针可以减少底层值的拷贝，可以提高效率，但是如果拷贝的数据量小，由于指针传递会产生逃逸，可能会使用堆，也可能会增加GC的负担，所以传递指针不一定是高效的。

并发控制

本章主要介绍GO语言开发过程中经常使用的并发控制手段。

我们考虑这么一种场景，协程A执行过程中需要创建子协程A1、A2、A3...An，协程A创建完子协程后就等待子协程退出。
针对这种场景，GO提供了三种解决方案：

- Channel: 使用channel控制子协程
- WaitGroup：使用信号量机制控制子协程
- Context: 使用上下文控制子协程

三种方案各有优劣，比如Channel优点是实现简单，清晰易懂，WaitGroup优点是子协程个数动态可调整，Context优点是对子协程派生出来的孙子协程的控制。
缺点是相对而言的，要结合实例应用场景进行选择。

Channel

1. 前言

channel一般用于协程之间的通信，channel也可以用于并发控制。比如主协程启动N个子协程，主协程等待所有子协程退出后再继续后续流程，这种场景下channel也可轻易实现。

2. 场景示例

下面程序展示一个使用channel控制子协程的例子：

```
package main

import (
    "time"
    "fmt"
)

func Process(ch chan int) {
    //Do some work...
    time.Sleep(time.Second)

    ch <- 1 //管道中写入一个元素表示当前协程已结束
}

func main() {
    channels := make([]chan int, 10) //创建一个10个元素的切片，元素类型为channel

    for i:= 0; i < 10; i++ {
        channels[i] = make(chan int) //切片中放入一个channel
        go Process(channels[i])      //启动协程，传一个管道用于通信
    }

    for i, ch := range channels {  //遍历切片，等待子协程结束
        <-ch
        fmt.Println("Routine ", i, " quit!")
    }
}
```

上面程序通过创建N个channel来管理N个协程，每个协程都有一个channel用于跟父协程通信，父协程创建完所有协程后等待所有协程结束。

这个例子中，父协程仅仅是等待子协程结束，其实父协程也可以向管道中写入数据通知子协程结束，这时子协程需要定期地探测管道中是否有消息出现。

3. 总结

使用channel来控制子协程的优点是实现简单，缺点是当需要大量创建协程时就需要有相同数量的channel，而且对于子协程继续派生出来的协程不方便控制。

后面继续介绍的WaitGroup、Context看起来比channel优雅一些，在各种开源组件中使用频率比channel高得多。

WaitGroup

1 前言

WaitGroup是Golang应用开发过程中经常使用的并发控制技术。

WaitGroup，可理解为Wait-Goroutine-Group，即等待一组goroutine结束。比如某个goroutine需要等待其他几个goroutine全部完成，那么使用WaitGroup可以轻松实现。

下面程序展示了一个goroutine等待另外两个goroutine结束的例子：

```
package main

import (
    "fmt"
    "time"
    "sync"
)

func main() {
    var wg sync.WaitGroup

    wg.Add(2) //设置计数器，数值即为goroutine的个数
    go func() {
        //Do some work
        time.Sleep(1*time.Second)

        fmt.Println("Goroutine 1 finished!")
        wg.Done() //goroutine执行结束后将计数器减1
    }()

    go func() {
        //Do some work
        time.Sleep(2*time.Second)

        fmt.Println("Goroutine 2 finished!")
        wg.Done() //goroutine执行结束后将计数器减1
    }()

    wg.Wait() //主goroutine阻塞等待计数器变为0
    fmt.Printf("All Goroutine finished!")
}
```

简单的说，上面程序中wg内部维护了一个计数器：

1. 启动goroutine前将计数器通过Add(2)将计数器设置为待启动的goroutine个数。
2. 启动goroutine后，使用Wait()方法阻塞自己，等待计数器变为0。
3. 每个goroutine执行结束通过Done()方法将计数器减1。
4. 计数器变为0后，阻塞的goroutine被唤醒。

其实WaitGroup也可以实现一组goroutine等待另一组goroutine，这有点像玩杂技，很容出错，如果不了解其实现原理更是如此。实际上，WaitGroup的实现源码非常简单。

2 基础知识

2.1 信号量

信号量是Unix系统提供的一种保护共享资源的机制，用于防止多个线程同时访问某个资源。

可简单理解为信号量为一个数值：

- 当信号量>0时，表示资源可用，获取信号量时系统自动将信号量减1；
- 当信号量==0时，表示资源暂不可用，获取信号量时，当前线程会进入睡眠，当信号量为正时被唤醒；

由于WaitGroup实现中也使用了信号量，在此做个简单介绍。

3 WaitGroup

3.1 数据结构

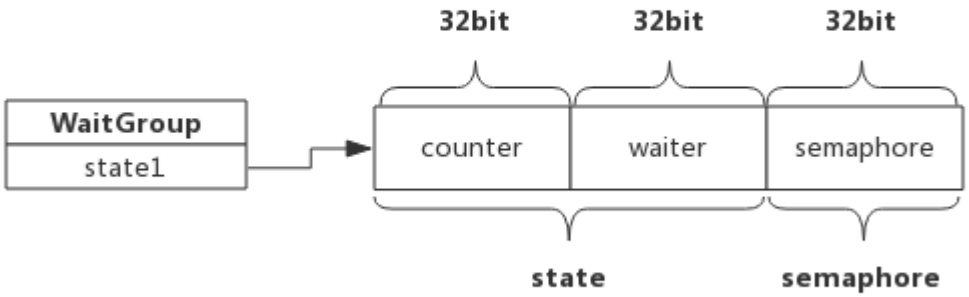
源码包中 `src/sync/waitgroup.go:WaitGroup` 定义了其数据结构：

```
type WaitGroup struct {
    state1 [3]uint32
}
```

state1是个长度为3的数组，其中包含了state和一个信号量，而state实际上是两个计数器：

- counter： 当前还未执行结束的goroutine计数器
- waiter count: 等待goroutine-group结束的goroutine数量，即有多少个等候者
- semaphore: 信号量

考虑到字节是否对齐，三者出现的位置不同，为简单起见，依照字节已对齐情况下，三者在内存中的位置如下所示：



WaitGroup对外提供三个接口：

- Add(delta int): 将delta值加到counter中
- Wait()： waiter递增1，并阻塞等待信号量semaphore

- Done()： counter递减1，按照waiter数值释放相应次数信号量

下面分别介绍这三个函数的实现细节。

3.2 Add(delta int)

Add()做了两件事，一是把delta值累加到counter中，因为delta可以为负值，也就是说counter有可能变成0或负值，所以第二件事就是当counter值变为0时，根据waiter数值释放等量的信号量，把等待的goroutine全部唤醒，如果counter变为负值，则panic.

Add()伪代码如下：

```
func (wg *WaitGroup) Add(delta int) {
    statep, semap := wg.state() //获取state和semaphore地址指针

    state := atomic.AddUint64(statep, uint64(delta)<<32) //把delta左移32位累加到state，即累加到counter中
    v := int32(state >> 32) //获取counter值
    w := uint32(state)      //获取waiter值

    if v < 0 {              //经过累加后counter值变为负值，panic
        panic("sync: negative WaitGroup counter")
    }

    //经过累加后，此时，counter >= 0
    //如果counter为正，说明不需要释放信号量，直接退出
    //如果waiter为零，说明没有等待者，也不需要释放信号量，直接退出
    if v > 0 || w == 0 {
        return
    }

    //此时，counter一定等于0，而waiter一定大于0（内部维护waiter，不会出现小于0的情况），
    //先把counter置为0，再释放waiter个数的信号量
    *statep = 0
    for ; w != 0; w-- {
        runtime_Semrelease(semap, false) //释放信号量，执行一次释放一个，唤醒一个等待者
    }
}
```

3.3 Wait()

Wait()方法也做了两件事，一是累加waiter, 二是阻塞等待信号量

```
func (wg *WaitGroup) Wait() {
    statep, semap := wg.state() //获取state和semaphore地址指针
    for {
        state := atomic.LoadUint64(statep) //获取state值
        v := int32(state >> 32)            //获取counter值
        w := uint32(state)                 //获取waiter值
        if v == 0 {                        //如果counter值为0，说明所有goroutine都退出了，不需要待待，直接返
回
            return
        }

        // 使用CAS（比较交换算法）累加waiter，累加可能会失败，失败后通过for loop下次重试
        if atomic.CompareAndSwapUint64(statep, state, state+1) {
            runtime_Semacquire(semap) //累加成功后，等待信号量唤醒自己
            return
```

```
        }
    }
}
```

这里用到了CAS算法保证有多个goroutine同时执行Wait()时也能正确累加waiter。

3.4 Done()

Done()只做一件事，即把counter减1，我们知道Add()可以接受负值，所以Done实际上只是调用了Add(-1)。

源码如下：

```
func (wg *WaitGroup) Done() {
    wg.Add(-1)
}
```

Done()的执行逻辑就转到了Add()，实际上也正是最后一个完成的goroutine把等待者唤醒的。

4 小结

简单说来， `WaitGroup` 通常用于等待一组“工作协程”结束的场景，其内部维护两个计数器，这里把它们称为“工作协程”计数器和“坐等协程”计数器，
`WaitGroup` 对外提供的三个方法分工非常明确：

- `Add(delta int)` 方法用于增加“工作协程”计数，通常在启动新的“工作协程”之前调用；
- `Done()` 方法用于减少“工作协程”计数，每次调用递减 `1` ，通常在“工作协程”内部且在临近返回之前调用；
- `Wait()` 方法用于增加“坐等协程”计数，通常在所有”工作协程”全部启动之后调用；

`Done()` 方法除了负责递减“工作协程”计数以外，还会在“工作协程”计数变为 `0` 时检查“坐等协程”计数器并把“坐等协程”唤醒。
需要注意的是， `Done()` 方法递减“工作协程”计数后，如果“工作协程”计数变成负数时，将会触发 `panic` ，这就要求 `Add()` 方法调用要早于 `Done()` 方法。

此外，通过 `Add()` 方法累加的“工作协程”计数要与实际需要等待的“工作协程”数量一致，否则也会触发 `panic` 。
当“工作协程”计数多于实际需要等待的“工作协程”数量时，“坐等协程”可能会永远无法被唤醒而产生列锁，此时，Go运行时检测到死锁会触发 `panic` ，
当“工作协程”计数小于实际需要等待的“工作协程”数量时， `Done()` 会在“工作协程”计数变为负数时触发 `panic` 。

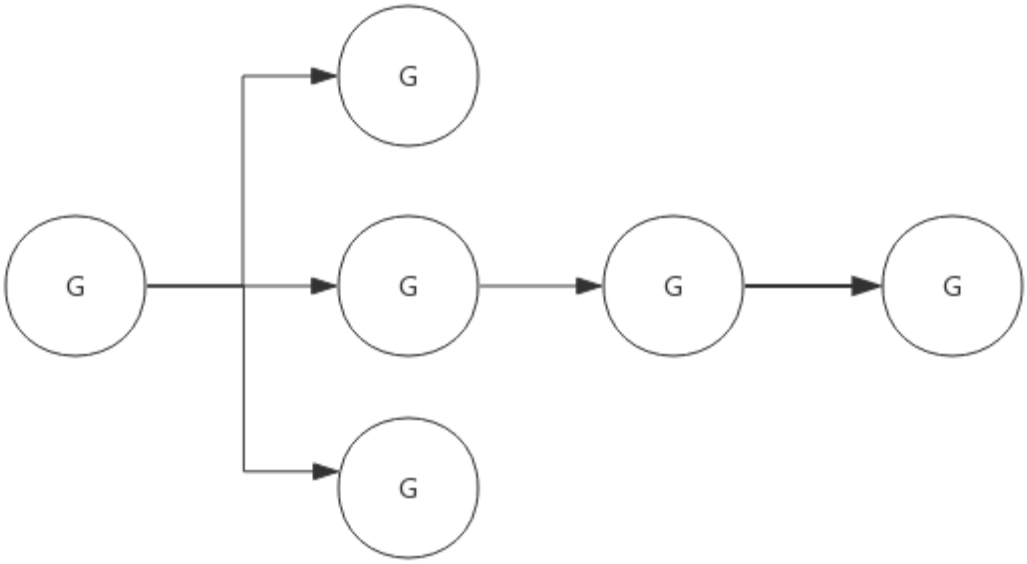
Context

1. 前言

Golang context是Golang应用开发常用的并发控制技术，它与WaitGroup最大的不同点是context对于派生goroutine有更强的控制力，它可以控制多级的goroutine。

context翻译成中文是”上下文”，即它可以控制一组呈树状结构的goroutine，每个goroutine拥有相同的上下文。

典型的使用场景如下图所示：



上图中由于goroutine派生出子goroutine，而子goroutine又继续派生新的goroutine，这种情况下使用WaitGroup就不太容易，因为子goroutine个数不容易确定。而使用context就可以很容易实现。

2. Context实现原理

context实际上只定义了接口，凡是实现该接口的类都可称为是一种context，官方包中实现了几个常用的context，分别可用于不同的场景。

2.1 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{}
}
```

基础的context接口只定义了4个方法，下面分别简要说明一下：

2.1.1 Deadline()

该方法返回一个deadline和标识是否已设置deadline的bool值，如果没有设置deadline，则ok == false，此时deadline为一个初始值的time.Time值

2.1.2 Done()

该方法返回一个channel，需要在select-case语句中使用，如”case <-context.Done():”。

当context关闭后，Done()返回一个被关闭的管道，关闭的管道仍然是可读的，据此goroutine可以收到关闭请求；
当context还未关闭时，Done()返回nil。

2.1.3 Err()

该方法描述context关闭的原因。关闭原因由context实现控制，不需要用户设置。比如Deadline context，关闭原因可能是因为deadline，也可能提前被主动关闭，那么关闭原因就会不同:

- 因deadline关闭：“context deadline exceeded”；
- 因主动关闭：“context canceled”。

当context关闭后，Err()返回context的关闭原因；
当context还未关闭时，Err()返回nil；

2.1.4 Value()

有一种context，它不是用于控制呈树状分布的goroutine，而是用于在树状分布的goroutine间传递信息。

Value()方法就是用于此种类型的context，该方法根据key值查询map中的value。具体使用后面示例说明。

2.2 空context

context包中定义了一个空的context， 名为emptyCtx，用于context的根节点，空的context只是简单的实现了Context，本身不包含任何值，仅用于其他context的父节点。

emptyCtx类型定义如下代码所示：

```
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
}
```

context包中定义了一个公用的emptCtx全局变量，名为background，可以使用context.Background()获取它，实现代码如下所示：

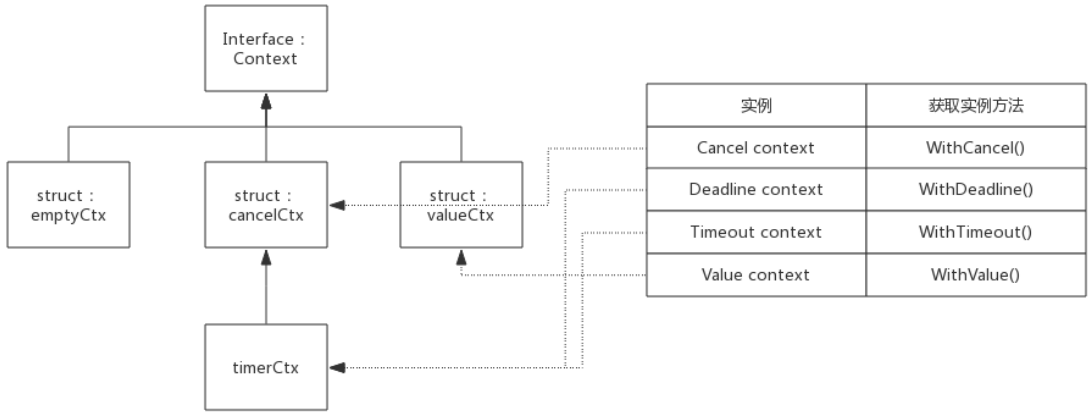
```
var background = new(emptyCtx)
func Background() Context {
    return background
}
```

context包提供了4个方法创建不同类型的context，使用这四个方法时如果没有父context，都需要传入backgroud，即backgroud作为其父节点：

- WithCancel()
- WithDeadline()
- WithTimeout()
- WithValue()

context包中实现Context接口的struct，除了emptyCtx外，还有cancelCtx、timerCtx和valueCtx三种，正是基于这三种context实例，实现了上述4种类型的context。

context包中各context类型之间的关系，如下图所示：



struct cancelCtx、timerCtx、valueCtx都继承于Context，下面分别介绍这三个struct。

2.3 cancelCtx

源码包中 `src/context/context.go:cancelCtx` 定义了该类型context：

```
type cancelCtx struct {
    Context

    mu       sync.Mutex            // protects following fields
    done     chan struct{}         // created lazily, closed by first cancel call
    children map[canceler]struct{} // set to nil by the first cancel call
```

```
    err      error                // set to non-nil by the first cancel call
}
```

children中记录了由此context派生的所有child，此context被cancel时会把其中的所有child都cancel掉。

cancelCtx与deadline和value无关，所以只需要实现Done()和Err()外露接口即可。

2.3.1 Done()接口实现

按照Context定义，Done()接口只需要返回一个channel即可，对于cancelCtx来说只需要返回成员变量done即可。

这里直接看下源码，非常简单：

```
func (c *cancelCtx) Done() <-chan struct{} {
    c.mu.Lock()
    if c.done == nil {
        c.done = make(chan struct{})
    }
    d := c.done
    c.mu.Unlock()
    return d
}
```

由于cancelCtx没有指定初始化函数，所以cancelCtx.done可能还未分配，所以需要考虑初始化。
cancelCtx.done会在context被cancel时关闭，所以cancelCtx.done的值一般经历如下三个阶段：
nil -> chan struct{} -> closed chan。

2.3.2 Err()接口实现

按照Context定义，Err()只需要返回一个error告知context被关闭的原因。对于cancelCtx来说只需要返回成员变量err即可。

还是直接看下源码：

```
func (c *cancelCtx) Err() error {
    c.mu.Lock()
    err := c.err
    c.mu.Unlock()
    return err
}
```

cancelCtx.err默认是nil，在context被cancel时指定一个error变量：`var Canceled = errors.New("context canceled")`。

2.3.3 cancel()接口实现

cancel()内部方法是理解cancelCtx的最关键的方法，其作用是关闭自己和其后代，其后代存储在cancelCtx.children的map中，其中key值即后代对象，value值并没有意义，这里使用map只是为了方便查询而已。

cancel方法实现伪代码如下所示：

```
func (c *cancelCtx) cancel(removeFromParent bool, err error) {
    c.mu.Lock()

    c.err = err                          //设置一个error，说明关闭原因
    close(c.done)                        //将channel关闭，以此通知派生的context
```

```
    for child := range c.children {   //遍历所有children，逐个调用cancel方法
        child.cancel(false, err)
    }
    c.children = nil
    c.mu.Unlock()

    if removeFromParent {            //正常情况下，需要将自己从parent删除
        removeChild(c.Context, c)
    }
}
```

实际上，WithCancel()返回的第二个用于cancel context的方法正是此cancel()。

2.3.4 WithCancel()方法实现

WithCancel()方法作了三件事：

- 初始化一个cancelCtx实例
- 将cancelCtx实例添加到其父节点的children中(如果父节点也可以被cancel的话)
- 返回cancelCtx实例和cancel()方法

其实现源码如下所示：

```
func WithCancel(parent Context) (ctx Context, cancel CancelFunc) {
    c := newCancelCtx(parent)
    propagateCancel(parent, &c)   //将自身添加到父节点
    return &c, func() { c.cancel(true, Canceled) }
}
```

这里将自身添加到父节点的过程有必要简单说明一下：

1. 如果父节点也支持cancel，也就是说其父节点肯定有children成员，那么把新context添加到children里即可；
2. 如果父节点不支持cancel，就继续向上查询，直到找到一个支持cancel的节点，把新context添加到children里；
3. 如果所有的父节点均不支持cancel，则启动一个协程等待父节点结束，然后再把当前context结束。

2.3.5 典型使用案例

一个典型的使用cancel context的例子如下所示：

```
package main

import (
    "fmt"
    "time"
    "context"
)

func HandelRequest(ctx context.Context) {
    go WriteRedis(ctx)
    go WriteDatabase(ctx)
    for {
        select {
        case <-ctx.Done():
            fmt.Println("HandelRequest Done.")
            return
        default:
            fmt.Println("HandelRequest running")
```

```
            time.Sleep(2 * time.Second)
        }
    }
}

func WriteRedis(ctx context.Context) {
    for {
        select {
        case <-ctx.Done():
            fmt.Println("WriteRedis Done.")
            return
        default:
            fmt.Println("WriteRedis running")
            time.Sleep(2 * time.Second)
        }
    }
}

func WriteDatabase(ctx context.Context) {
    for {
        select {
        case <-ctx.Done():
            fmt.Println("WriteDatabase Done.")
            return
        default:
            fmt.Println("WriteDatabase running")
            time.Sleep(2 * time.Second)
        }
    }
}

func main() {
    ctx, cancel := context.WithCancel(context.Background())
    go HandelRequest(ctx)

    time.Sleep(5 * time.Second)
    fmt.Println("It's time to stop all sub goroutines!")
    cancel()

    //Just for test whether sub goroutines exit or not
    time.Sleep(5 * time.Second)
}
```

上面代码中协程HandelRequest()用于处理某个请求，其又会创建两个协程：WriteRedis()、WriteDatabase()，main协程创建context，并把context在各子协程间传递，main协程在适当的时机可以cancel掉所有子协程。

程序输出如下所示：

```
HandelRequest running
WriteDatabase running
WriteRedis running
HandelRequest running
WriteDatabase running
WriteRedis running
HandelRequest running
WriteDatabase running
WriteRedis running
It's time to stop all sub goroutines!
WriteDatabase Done.
```

```
HandelRequest Done.
WriteRedis Done.
```

2.4 timerCtx

源码包中 `src/context/context.go:timerCtx` 定义了该类型context：

```
type timerCtx struct {
    cancelCtx
    timer *time.Timer // Under cancelCtx.mu.

    deadline time.Time
}
```

timerCtx在cancelCtx基础上增加了deadline用于标示自动cancel的最终时间，而timer就是一个触发自动cancel的定时器。

由此，衍生出WithDeadline()和WithTimeout()。实现上这两种类型实现原理一样，只不过使用语境不一样：

- deadline: 指定最后期限，比如context将2018.10.20 00:00:00之时自动结束
- timeout: 指定最长存活时间，比如context将在30s后结束。

对于接口来说，timerCtx在cancelCtx基础上还需要实现Deadline()和cancel()方法，其中cancel()方法是重写的。

2.4.1 Deadline()接口实现

Deadline()方法仅仅是返回timerCtx.deadline而矣。而timerCtx.deadline是WithDeadline()或WithTimeout()方法设置的。

2.4.2 cancel()接口实现

cancel()方法基本继承cancelCtx，只需要额外把timer关闭。

timerCtx被关闭后，timerCtx.cancelCtx.err将会存储关闭原因：

- 如果deadline到来之前手动关闭，则关闭原因与cancelCtx显示一致；
- 如果deadline到来时自动关闭，则原因为：”context deadline exceeded”

2.4.3 WithDeadline()方法实现

WithDeadline()方法实现步骤如下：

- 初始化一个timerCtx实例
- 将timerCtx实例添加到其父节点的children中(如果父节点也可以被cancel的话)
- 启动定时器，定时器到期后会自动cancel本context
- 返回timerCtx实例和cancel()方法

也就是说，timerCtx类型的context不仅支持手动cancel，也会在定时器到来后自动cancel。

2.4.4 WithTimeout()方法实现

WithTimeout()实际调用了WithDeadline，二者实现原理一致。

看代码会非常清晰：

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

2.4.5 典型使用案例

下面例子中使用WithTimeout()获得一个context并在其子协程中传递：

```
package main

import (
    "fmt"
    "time"
    "context"
)

func HandelRequest(ctx context.Context) {
    go WriteRedis(ctx)
    go WriteDatabase(ctx)
    for {
        select {
        case <-ctx.Done():
            fmt.Println("HandelRequest Done.")
            return
        default:
            fmt.Println("HandelRequest running")
            time.Sleep(2 * time.Second)
        }
    }
}

func WriteRedis(ctx context.Context) {
    for {
        select {
        case <-ctx.Done():
            fmt.Println("WriteRedis Done.")
            return
        default:
            fmt.Println("WriteRedis running")
            time.Sleep(2 * time.Second)
        }
    }
}

func WriteDatabase(ctx context.Context) {
    for {
        select {
        case <-ctx.Done():
            fmt.Println("WriteDatabase Done.")
            return
        default:
            fmt.Println("WriteDatabase running")
            time.Sleep(2 * time.Second)
        }
    }
}

func main() {
    ctx, _ := context.WithTimeout(context.Background(), 5 * time.Second)
```

```
    go HandelRequest(ctx)

    time.Sleep(10 * time.Second)
}
```

主协程中创建一个10s超时的context，并将其传递给子协程，10s自动关闭context。程序输出如下：

```
HandelRequest running
WriteRedis running
WriteDatabase running
HandelRequest running
WriteRedis running
WriteDatabase running
HandelRequest running
WriteRedis running
WriteDatabase running
HandelRequest Done.
WriteDatabase Done.
WriteRedis Done.
```

2.5 valueCtx

源码包中 `src/context/context.go:valueCtx` 定义了该类型context：

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

valueCtx只是在Context基础上增加了一个key-value对，用于在各级协程间传递一些数据。

由于valueCtx既不需要cancel，也不需要deadline，那么只需要实现Value()接口即可。

2.5.1 Value（）接口实现

由valueCtx数据结构定义可见，valueCtx.key和valueCtx.val分别代表其key和value值。 实现也很简单：

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

这里有个细节需要关注一下，即当前context查找不到key时，会向父节点查找，如果查询不到则最终返回interface{}。也就是说，可以通过子context查询到父的value值。

2.5.2 WithValue（）方法实现

WithValue()实现也是非常的简单，伪代码如下：

```
func WithValue(parent Context, key, val interface{}) Context {
    if key == nil {
        panic("nil key")
    }
```

```
    return &valueCtx{parent, key, val}
}
```

2.5.3 典型使用案例

下面示例程序展示valueCtx的用法：

```
package main

import (
    "fmt"
    "time"
    "context"
)

func HandelRequest(ctx context.Context) {
    for {
        select {
        case <-ctx.Done():
            fmt.Println("HandelRequest Done.")
            return
        default:
            fmt.Println("HandelRequest running, parameter: ", ctx.Value("parameter"))
            time.Sleep(2 * time.Second)
        }
    }
}

func main() {
    ctx := context.WithValue(context.Background(), "parameter", "1")
    go HandelRequest(ctx)

    time.Sleep(10 * time.Second)
}
```

上例main()中通过WithValue()方法获得一个context，需要指定一个父context、key和value。然后通将该context传递给子协程HandelRequest，子协程可以读取到context的key-value。

注意：本例中子协程无法自动结束，因为context是不支持cancle的，也就是说<-ctx.Done()永远无法返回。如果需要返回，需要在创建context时指定一个可以cancel的context作为父节点，使用父节点的cancel()在适当的时机结束整个context。

总结

- Context仅仅是一个接口定义，根据实现的不同，可以衍生出不同的context类型；
- cancelCtx实现了Context接口，通过WithCancel()创建cancelCtx实例；
- timerCtx实现了Context接口，通过WithDeadline()和WithTimeout()创建timerCtx实例；
- valueCtx实现了Context接口，通过WithValue()创建valueCtx实例；
- 三种context实例可互为父节点，从而可以组合成不同的应用形式；

反射

本章主要介绍GO语言反射机制。

反射机制

1. 前言

个人觉得，反射讲得最透彻的还是官方博客。官方博客略显晦涩，多读几遍就慢慢理解了。
本文既是学习笔记，也是总结。
官方博客地址：https://blog.golang.org/laws-of-reflection

2. 反射概念

官方对此有个非常简明的介绍，两句话耐人寻味：

1. 反射提供一种让程序检查自身结构的能力
2. 反射是困惑的源泉

第1条，再精确点的描述是“反射是一种检查interface变量的底层类型和值的机制”。
第2条，很有喜感的自嘲，不过往后看就笑不出来了，因为你很可能产生困惑.

想深入了解反射，必须深入理解类型和接口概念。下面开始复习一下这些基础概念。

2.1 关于静态类型

你肯定知道Go是静态类型语言，比如”int”、”float32”、”[]byte”等等。每个变量都有一个静态类型，且在编译时就确定了。
那么考虑一下如下一种类型声明:

```
type Myint int

var i int
var j Myint
```

Q: i 和j 类型相同吗？
A：i 和j类型是不同的。 二者拥有不同的静态类型，没有类型转换的话是不可以互相赋值的，尽管二者底层类型是一样的。

2.2 特殊的静态类型interface

interface类型是一种特殊的类型，它代表方法集合。 它可以存放任何实现了其方法的值。

经常被拿来举例的是io包里的这两个接口类型：

```
// Reader is the interface that wraps the basic Read method.
type Reader interface {
    Read(p []byte) (n int, err error)
}

// Writer is the interface that wraps the basic Write method.
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

任何类型，比如某struct，只要实现了其中的Read()方法就被认为是实现了Reader接口，只要实现了Write()方法，就被认为是实现了Writer接口，不过方法参数和返回值要跟接口声明的一致。

接口类型的变量可以存储任何实现该接口的值。

2.3 特殊的interface类型

最特殊的interface类型为空interface类型，即 `interface {}` ，前面说了，interface用来表示一组方法集合，所有实现该方法集合的类型都被认为是实现了该接口。那么空interface类型的方法集合为空，也就是说所有类型都可以认为是实现了该接口。

一个类型实现空interface并不重要，重要的是一个空interface类型变量可以存放所有值，记住是所有值，这才是最最重要的。这也是有些人认为Go是动态类型的原因，这是个错觉。

2.4 interface类型是如何表示的

前面讲了，interface类型的变量可以存放任何实现了该接口的值。还是以上面的 `io.Reader` 为例进行说明， `io.Reader` 是一个接口类型， `os.OpenFile()` 方法返回一个 `File` 结构体类型变量，该结构体类型实现了 `io.Reader` 的方法，那么 `io.Reader` 类型变量就可以用来接收该返回值。如下所示：

```
var r io.Reader
tty, err := os.OpenFile("/dev/tty", os.O_RDWR, 0)
if err != nil {
    return nil, err
}
r = tty
```

那么问题来了。
Q： r的类型是什么？
A: r的类型始终是 `io.Reader` interface类型，无论其存储什么值。

Q：那 `File` 类型体现在哪里？
A：r保存了一个(value, type)对来表示其所存储值的信息。 value即为r所持有元素的值，type即为所持有元素的底层类型

Q：如何将r转换成另一个类型结构体变量？比如转换成 `io.Writer`
A：使用类型断言，如 `w = r.(io.Writer)` . 意思是如果r所持有的元素如果同样实现了io.Writer接口,那么就把值传递给w。

3. 反射三定律

前面之所以讲类型，是为了引出interface，之所以讲interface是想说interface类型有个(value，type)对，而反射就是检查interface的这个(value, type)对的。具体一点说就是Go提供一组方法提取interface的value，提供另一组方法提取interface的type.

官方提供了三条定律来说明反射，比较清晰，下面也按照这三定律来总结。

反射包里有两个接口类型要先了解一下.

- `reflect.Type` 提供一组接口处理interface的类型，即（value, type）中的type
- `reflect.Value` 提供一组接口处理interface的值,即(value, type)中的value

下面会提到反射对象，所谓反射对象即反射包里提供的两种类型的对象。

- `reflect.Type` 类型对象
- `reflect.Value` 类型对象

3.1 反射第一定律：反射可以将interface类型变量转换成反射对象

下面示例，看看是如何通过反射获取一个变量的值和类型的：

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var x float64 = 3.4
    t := reflect.TypeOf(x)  //t is reflect.Type
    fmt.Println("type:", t)

    v := reflect.ValueOf(x) //v is reflect.Value
    fmt.Println("value:", v)
}
```

程序输出如下：

```
type: float64
value: 3.4
```

注意：反射是针对interface类型变量的，其中 `TypeOf()` 和 `ValueOf()` 接受的参数都是 `interface{}` 类型的，也即x值是被转成了interface传入的。

除了 `reflect.TypeOf()` 和 `reflect.ValueOf()` ，还有其他很多方法可以操作，本文先不过多介绍，否则一不小心会会引起困惑。

3.2 反射第二定律：反射可以将反射对象还原成interface对象

之所以叫'反射'，反射对象与interface对象是可以互相转化的。看以下例子：

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var x float64 = 3.4

    v := reflect.ValueOf(x) //v is reflect.Value

    var y float64 = v.Interface().(float64)
    fmt.Println("value:", y)
}
```

对象x转换成反射对象v，v又通过Interface()接口转换成interface对象，interface对象通过.(float64)类型断言获取float64类型的值。

3.3 反射第三定律：反射对象可修改，value值必须是可设置的

通过反射可以将interface类型变量转换成反射对象，可以使用该反射对象设置其持有的值。在介绍何谓反射对象可修改前，先看一下失败的例子：

```
package main

import (
    "reflect"
)

func main() {
    var x float64 = 3.4
    v := reflect.ValueOf(x)
    v.SetFloat(7.1) // Error: will panic.
}
```

如下代码，通过反射对象v设置新值，会出现panic。报错如下：

```
panic: reflect: reflect.Value.SetFloat using unaddressable value
```

错误原因即是v是不可修改的。

反射对象是否可修改取决于其所存储的值，回想一下函数传参时是传值还是传址就不难理解上例中为何失败了。

上例中，传入reflect.ValueOf()函数的其实是x的值，而非x本身。即通过v修改其值是无法影响x的，也即是无效的修改，所以golang会报错。

想到此处，即可明白，如果构建v时使用x的地址就可实现修改了，但此时v代表的是指针地址，我们要设置的是指针所指向的内容，也即我们想要修改的是 `*v` 。 那怎么通过v修改x的值呢？

`reflect.Value` 提供了 `Elem()` 方法，可以获得指针向指向的 `value` 。看如下代码：

```
package main

import (
"reflect"
    "fmt"
)

func main() {
    var x float64 = 3.4
    v := reflect.ValueOf(&x)
    v.Elem().SetFloat(7.1)
    fmt.Println("x :", v.Elem().Interface())
}
```

输出为：

```
x : 7.1
```

4. 总结

结合官方博客及本文，至少可以对反射理解个大概，还有很多方法没有涉及。
对反射的深入理解，个人觉得还需要继续看的内容：

- 参考业界，尤其是开源框架中是如何使用反射的

- 研究反射实现原理，探究其性能优化的手段

测试

Go语言提供了go test 命令行工具，使用该工具可以很方便的进行测试。

不仅Go语言源码中大量使用go test，在各种开源框架中的应用也极为普遍。

目前go test支持的测试类型有：

- 单元测试
- 性能测试
- 示例测试

本章，我们先快速掌握这几种测试的基本用法，然后我们根据源码来学习这些测试的实现机制。

快速开始

使用Go自身的测试系统使用起来非常简单，只需要添加很少的代码就可以快速开始测试。

目前Go测试系统支持单元测试、性能测试和示例测试。

单元测试

单元测试是指对软件中的最小可测试单元进行检查和验证，比如对一个函数的测试。

性能测试

性能测试，也称基准测试，可以测试一段程序的性能，可以得到时间消耗、内存使用情况的报告。

示例测试

示例测试，广泛应用于Go源码和各种开源框架中，用于展示某个包或某个方法的用法。

比如，Go标准库中，mail包展示如何从一个字符串解析出邮件列表的用法，非常直观易懂。

源码位于 `src/net/mail/example_test.go` 中：

```
func ExampleParseAddressList() {
    const list = "Alice <alice@example.com>, Bob <bob@example.com>, Eve <eve@example.com>"
    emails, err := mail.ParseAddressList(list)
    if err != nil {
        log.Fatal(err)
    }

    for _, v := range emails {
        fmt.Println(v.Name, v.Address)
    }

    // Output:
    // Alice alice@example.com
    // Bob bob@example.com
    // Eve eve@example.com
}
```

本节，我们通过简单的例子，快速体验一下如何使用Go的测试系统进行测试。

单元测试

源代码目录结构

我们在gotest包中创建两个文件，目录结构如下所示：

```
[GoExpert]
|--[src]
   |--[gotest]
      |--unit.go
      |--unit_test.go
```

其中 `unit.go` 为源代码文件， `unit_test.go` 为测试文件。要保证测试文件以“_test.go”结尾。

源代码文件

源代码文件 `unit.go` 中包含一个 `Add()` 方法，如下所示：

```
package gotest

// Add 方法用于演示go test使用
func Add(a int, b int) int {
    return a + b
}
```

`Add()` 方法仅提供两数加法，实际项目中不可能出现类似的方法，此处仅供单元测试示例。

测试文件

测试文件 `unit_test.go` 中包含一个测试方法 `TestAdd()` ，如下所示：

```
package gotest_test

import (
    "testing"
    "gotest"
)

func TestAdd(t *testing.T) {
    var a = 1
    var b = 2
    var expected = 3

    actual := gotest.Add(a, b)
    if actual != expected {
        t.Errorf("Add(%d, %d) = %d; expected: %d", a, b, actual, expected)
    }
}
```

通过package语句可以看到，测试文件属于“gotest_test”包，测试文件也可以跟源文件在同一个包，但常见的做法是创建一个包专用于测试，这样可以使测试文件和源文件隔离。GO源代码以及其他知名的开源框架通常会创建测试包，而且规则是在原包名上加上”_test”。

测试函数命名规则为”TestXxx”，其中“Test”为单元测试的固定开头，go test只会执行以此为开头的方法。紧跟“Test”是以首字母大写的单词，用于识别待测试函数。

测试函数参数并不是必须要使用的，但”testing.T”提供了丰富的方法帮助控制测试流程。

t.Errorf()用于标记测试失败，标记失败还有几个方法，在介绍testing.T结构时再详细介绍。

执行测试

命令行下，使用 `go test` 命令即可启动单元测试，如下所示：

```
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src\gotest>go test
PASS
ok      gotest  0.378s

E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src\gotest>
```

通过打印可知，测试通过，花费时间为0.378s。

总结

从上面可以看出，编写一个单元测试并执行是非常方便的，只需要遵循一定的规则：

- 测试文件名必须以”_test.go”结尾；
- 测试函数名必须以“TestXxx”开始；
- 命令行下使用”go test”即可启动测试；

性能测试

源代码目录结构

我们在gotest包中创建两个文件，目录结构如下所示：

```
[GoExpert]
|--[src]
   |--[gotest]
      |--benchmark.go
      |--benchmark_test.go
```

其中 `benchmark.go` 为源代码文件， `benchmark_test.go` 为测试文件。

源代码文件

源代码文件 `benchmark.go` 中包含 `MakeSliceWithoutAlloc()` 和 `MakeSliceWithPreAlloc()` 两个方法，如下所示：

```
package gotest

// MakeSliceWithPreAlloc 不预分配
func MakeSliceWithoutAlloc() []int {
    var newSlice []int

    for i := 0; i < 100000; i++ {
        newSlice = append(newSlice, i)
    }

    return newSlice
}

// MakeSliceWithPreAlloc 通过预分配Slice的存储空间构造
func MakeSliceWithPreAlloc() []int {
    var newSlice []int

    newSlice = make([]int, 0, 100000)
    for i := 0; i < 100000; i++ {
        newSlice = append(newSlice, i)
    }

    return newSlice
}
```

两个方法都会构造一个容量为100000的切片，所不同的是 `MakeSliceWithPreAlloc()` 会预先分配内存，而 `MakeSliceWithoutAlloc()` 不预先分配内存，二者理论上存在性能差异，本次就来测试一下二者的性能差异。

测试文件

测试文件 `benchmark_test.go` 中包含两个测试方法，用于测试源代码中两个方法的性能，测试文件如下所示：

```
package gotest_test

import (
    "testing"
    "gotest"
)

func BenchmarkMakeSliceWithoutAlloc(b *testing.B) {
    for i := 0; i < b.N; i++ {
        gotest.MakeSliceWithoutAlloc()
    }
}

func BenchmarkMakeSliceWithPreAlloc(b *testing.B) {
    for i := 0; i < b.N; i++ {
        gotest.MakeSliceWithPreAlloc()
    }
}
```

性能测试函数命名规则为"BenchmarkXxx"，其中"Xxx"为自定义的标识，需要以大写字母开始，通常为待测函数。

testing.B提供了一系列的用于辅助性能测试的方法或成员，比如本例中的 `b.N` 表示循环执行的次数，而N值不用程序员特别关心，按照官方说法，N值是动态调整的，直到可靠地算出程序执行时间后才会停止，具体执行次数会在执行结束后打印出来。

执行测试

命令行下，使用 `go test -bench=.` 命令即可启动性能测试，如下所示：

```
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src\gotest>go test -bench=.
BenchmarkMakeSliceWithoutAlloc-4            2000           1103822 ns/op
BenchmarkMakeSliceWithPreAlloc-4            5000            328944 ns/op
PASS
ok      gotest  4.445s
```

其中 `-bench` 为go test的flag，该flag指示go test进行性能测试，即执行测试文件中符合"BenchmarkXxx"规则的方法。
紧跟flag的为flag的参数，本例表示执行当前所有的性能测试。

通过输出可以直观的看出，`BenchmarkMakeSliceWithoutAlloc` 执行了2000次，平均每次 `1103822` 纳秒，`BenchmarkMakeSliceWithPreAlloc` 执行了5000次，平均每次 `328944` 纳秒。

从测试结果上看，虽然构造切片很快，但通过给切片预分配内存，性能还可以进一步提升，符合预期。关于原理分析，请参考Slice相关章节。

总结

从上面的例子可以看出，编写并执行性能测试是非常简单的，只需要遵循一些规则：

- 文件名必须以"_test.go"结尾；
- 函数名必须以"BenchmarkXxx"开始；
- 使用命令"go test -bench=."即可开始性能测试；

示例测试

源代码目录结构

我们在gotest包中创建两个文件，目录结构如下所示：

```
[GoExpert]
|--[src]
   |--[gotest]
      |--example.go
      |--example_test.go
```

其中 `example.go` 为源代码文件， `example_test.go` 为测试文件。

源代码文件

源代码文件 `example.go` 中包含 `SayHello()` 、 `SayGoodbye()` 和 `PrintNames()` 三个方法，如下所示：

```
package gotest

import "fmt"

// SayHello 打印一行字符串
func SayHello() {
    fmt.Println("Hello World")
}

// SayGoodbye 打印两行字符串
func SayGoodbye() {
    fmt.Println("Hello,")
    fmt.Println("goodbye")
}

// PrintNames 打印学生姓名
func PrintNames() {
    students := make(map[int]string, 4)
    students[1] = "Jim"
    students[2] = "Bob"
    students[3] = "Tom"
    students[4] = "Sue"
    for _, value := range students {
        fmt.Println(value)
    }
}
```

这几个方法打印内容略有不同，分别代表一种典型的场景：

- SayHello()：只有一行打印输出
- SayGoodbye()：有两行打印输出
- PrintNames()：有多行打印输出，且由于Map数据结构的原因，多行打印次序是随机的。

测试文件

测试文件 `example_test.go` 中包含3个测试方法，于源代码文件中的3个方法一一对应，测试文件如下所示：

```
package gotest_test

import "gotest"

// 检测单行输出
func ExampleSayHello() {
    gotest.SayHello()
    // OutPut: Hello World
}

// 检测多行输出
func ExampleSayGoodbye() {
    gotest.SayGoodbye()
    // OutPut:
    // Hello,
    // goodbye
}

// 检测乱序输出
func ExamplePrintNames() {
    gotest.PrintNames()
    // Unordered output:
    // Jim
    // Bob
    // Tom
    // Sue
}
```

例子测试函数命名规则为"ExampleXxx"，其中"Xxx"为自定义的标识，通常为待测函数名称。

这三个测试函数分别代表三种场景：

- ExampleSayHello()： 待测试函数只有一行输出，使用"// OutPut: "检测。
- ExampleSayGoodbye()：待测试函数有多行输出，使用"// OutPut: "检测，其中期望值也是多行。
- ExamplePrintNames()：待测试函数有多行输出，但输出次序不确定，使用"// Unordered output:"检测。

注：字符串比较时会忽略前后的空白字符。

执行测试

命令行下，使用 `go test` 或 `go test example_test.go` 命令即可启动测试，如下所示：

```
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src\gotest>go test example_test.go
ok      command-line-arguments  0.331s
```

总结

1. 例子测试函数名需要以"Example"开头；
2. 检测单行输出格式为"// Output: <期望字符串>"；
3. 检测多行输出格式为"// Output: \ <期望字符串> \ <期望字符串>"，每个期望字符串占一行；
4. 检测无序输出格式为"// Unordered output: \ <期望字符串> \ <期望字符串>"，每个期望字符串占一行；
5. 测试字符串时会自动忽略字符串前后的空白字符；
6. 如果测试函数中没有"Output"标识，则该测试函数不会被执行；
7. 执行测试可以使用 `go test`，此时该目录下的其他测试文件也会一并执行；

8. 执行测试可以使用`go test <xxx_test.go>`，此时仅执行特定文件中的测试函数；

进阶测试

前面我们通过简单示例快速了解了测试的用法。

本节，我们进一步探索测试的其他用法，以便应对稍微复杂一些的场景。

子测试

简介

简单的说，子测试提供一种在一个测试函数中执行多个测试的能力，比如原来有TestA、TestB和TestC三个测试函数，每个测试函数执行开始都需要做些相同的初始化工作，那么可以利用子测试将这三个测试合并到一个测试中，这样初始化工作只需要做一次。

除此之外，子测试还提供了诸多便利，下面我们逐一说明。

简单例子

我们先看一个简单的例子，以便快速了解子测试的基本用法。

```
package gotest_test

import (
    "testing"
    "gotest"
)

// sub1 为子测试，只做加法测试
func sub1(t *testing.T) {
    var a = 1
    var b = 2
    var expected = 3

    actual := gotest.Add(a, b)
    if actual != expected {
        t.Errorf("Add(%d, %d) = %d; expected: %d", a, b, actual, expected)
    }
}

// sub2 为子测试，只做加法测试
func sub2(t *testing.T) {
    var a = 1
    var b = 2
    var expected = 3

    actual := gotest.Add(a, b)
    if actual != expected {
        t.Errorf("Add(%d, %d) = %d; expected: %d", a, b, actual, expected)
    }
}

// sub3 为子测试，只做加法测试
func sub3(t *testing.T) {
    var a = 1
    var b = 2
    var expected = 3

    actual := gotest.Add(a, b)
    if actual != expected {
        t.Errorf("Add(%d, %d) = %d; expected: %d", a, b, actual, expected)
    }
}
```

```
// TestSub 内部调用sub1、sub2和sub3三个子测试
func TestSub(t *testing.T) {
    // setup code

    t.Run("A=1", sub1)
    t.Run("A=2", sub2)
    t.Run("B=1", sub3)

    // tear-down code
}
```

本例中 `TestSub()` 通过 `t.Run()` 依次执行三个子测试。 `t.Run()` 函数声明如下：

```
func (t *T) Run(name string, f func(t *T)) bool
```

`name` 参数为子测试的名字， `f` 为子测试函数，本例中 `Run()` 一直阻塞到 `f` 执行结束后才返回，返回值为f的执行结果。
`Run()` 会启动新的协程来执行 `f` ，并阻塞等待 `f` 执行结束才返回，除非 `f` 中使用 `t.Parallel()` 设置子测试为并发。

本例中 `TestSub()` 把三个子测试合并起来，可以共享setup和tear-down部分的代码。

我们在命令行下，使用 `-v` 参数执行测试：

```
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src\gotest>go test subunit_test.go -v
=== RUN   TestSub
=== RUN   TestSub/A=1
=== RUN   TestSub/A=2
=== RUN   TestSub/B=1
--- PASS: TestSub (0.00s)
    --- PASS: TestSub/A=1 (0.00s)
    --- PASS: TestSub/A=2 (0.00s)
    --- PASS: TestSub/B=1 (0.00s)
PASS
ok      command-line-arguments  0.354s
```

从输出中可以看出，三个子测试都被执行到了，而且执行次序与调用次序一致。

子测试命名规则

通过上面的例子我们知道 `Run()` 方法第一个参数为子测试的名字，而实际上子测试的内部命名规则为："*<父测试名字>/<传递给Run的名字>*"。比如，传递给 `Run()` 的名字是"A=1"，那么子测试名字为"TestSub/A=1"。这个在上面的命令行输出中也可以看出。

过滤筛选

通过测试的名字，可以在执行中过滤掉一部分测试。

比如，只执行上例中"A=*"的子测试，那么执行时使用 `-run Sub/A=` 参数即可：

```
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src\gotest>go test subunit_test.go -v -run Sub/A=
=== RUN   TestSub
=== RUN   TestSub/A=1
```

```
=== RUN   TestSub/A=2
--- PASS: TestSub (0.00s)
    --- PASS: TestSub/A=1 (0.00s)
    --- PASS: TestSub/A=2 (0.00s)
PASS
ok      command-line-arguments  0.340s
```

上例中，使用参数 `-run Sub/A=` 则只会执行 `TestSub/A=1` 和 `TestSub/A=2` 两个子测试。

对于子性能测试则使用 `-bench` 参数来筛选，此处不再赘述。

注意：此处的筛选不是严格的正则匹配，而是包含匹配。比如， `-run A=` 那么所有测试（含子测试）的名字中如果包含“A=”则会被选中执行。

子测试并发

前面提到的多个子测试共享setup和teardown有一个前提是子测试没有并发，如果子测试使用 `t.Parallel()` 指定并发，那么就没办法共享teardown了，因为执行顺序很可能是setup->子测试1->teardown->子测试2...。

如果子测试可能并发，则可以把子测试通过 `Run()` 再嵌套一层， `Run()` 可以保证其下的所有子测试执行结束后再返回。

为便于说明，我们创建文件 `subparallel_test.go` 用于说明：

```
package gotest_test

import (
    "testing"
    "time"
)

// 并发子测试，无实际测试工作，仅用于演示
func parallelTest1(t *testing.T) {
    t.Parallel()
    time.Sleep(3 * time.Second)
    // do some testing
}

// 并发子测试，无实际测试工作，仅用于演示
func parallelTest2(t *testing.T) {
    t.Parallel()
    time.Sleep(2 * time.Second)
    // do some testing
}

// 并发子测试，无实际测试工作，仅用于演示
func parallelTest3(t *testing.T) {
    t.Parallel()
    time.Sleep(1 * time.Second)
    // do some testing
}

// TestSubParallel 通过把多个子测试放到一个组中并发执行，同时多个子测试可以共享setup和tear-down
func TestSubParallel(t *testing.T) {
    // setup
    t.Logf("Setup")

    t.Run("group", func(t *testing.T) {
        t.Run("Test1", parallelTest1)
```

```
        t.Run("Test2", parallelTest2)
        t.Run("Test3", parallelTest3)
    })

    // tear down
    t.Logf("teardown")
}
```

上面三个子测试中分别sleep了3s、2s、1s用于观察并发执行顺序。通过 `Run()` 将多个子测试“封装”到一个组中，可以保证所有子测试全部执行结束后再执行tear-down。

命令行下的输出如下：

```
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src\gotest>go test subparallel_test.go -v -run SubParallel
=== RUN   TestSubParallel
=== RUN   TestSubParallel/group
=== RUN   TestSubParallel/group/Test1
=== RUN   TestSubParallel/group/Test2
=== RUN   TestSubParallel/group/Test3
--- PASS: TestSubParallel (3.01s)
        subparallel_test.go:25: Setup
    --- PASS: TestSubParallel/group (0.00s)
        --- PASS: TestSubParallel/group/Test3 (1.00s)
        --- PASS: TestSubParallel/group/Test2 (2.01s)
        --- PASS: TestSubParallel/group/Test1 (3.01s)
        subparallel_test.go:34: teardown
PASS
ok      command-line-arguments  3.353s
```

通过该输出可以看出：

1. 子测试是并发执行的（Test1最先被执行却最后结束）
2. tear-down在所有子测试结束后才执行

总结

- 子测试适用于单元测试和性能测试；
- 子测试可以控制并发；
- 子测试提供一种类似table-driven风格的测试；
- 子测试可以共享setup和tear-down；

Main测试

简介

我们知道子测试的一个方便之处在于可以让多个测试共享Setup和Tear-down。但这种程度的共享有时并不满足需求，有时希望在整个测试程序做一些全局的setup和Tear-down，这时就需要Main测试了。

所谓Main测试，即声明一个 `func TestMain(m *testing.M)` ，它是名字比较特殊的测试，参数类型为 `testing.M` 指针。如果声明了这样一个函数，当前测试程序将不是直接执行各项测试，而是将测试交给TestMain调度。

示例

下面通过一个例子来展示Main测试用法：

```
// TestMain 用于主动执行各种测试，可以测试前后做setup和tear-down操作
func TestMain(m *testing.M) {
    println("TestMain setup.")

    retCode := m.Run() // 执行测试，包括单元测试、性能测试和示例测试

    println("TestMain tear-down.")

    os.Exit(retCode)
}
```

上述例子中，日志打印的两行分别对应Setup和Tear-down代码，m.Run()即为执行所有的测试，m.Run()的返回结果通过os.Exit()返回。

如果所有测试均通过测试，m.Run()返回0，否同m.Run()返回1，代表测试失败。

有一点需要注意的是，TestMain执行时，命令行参数还未解析，如果测试程序需要依赖参数，可以使用 `flag.Parse()` 解析参数，m.Run()方法内部还会再次解析参数，此处解析不会影响原测试过程。

实现原理

本节，我们重点探索一下go test的实现原理。

首先，我们会先从数据结构入手，查看测试是如何被组织起来的。其次，我们会关注测试的关键实现方法，尽量呈现源码并配以示例来了解其实现原理。

为了叙述方便，本节部分源码隐去了部分与话题无关的代码，更多的源码解释，可以通过源码注释来了解。

testing.common公共类

简介

我们知道单元测试函数需要传递一个 `testing.T` 类型的参数，而性能测试函数需要传递一个 `testing.B` 类型的参数，该参数可用于控制测试的流程，比如标记测试失败等。

`testing.T` 和 `testing.B` 属于 `testing` 包中的两个数据类型，该类型提供一系列的方法用于控制函数执行流程，考虑到二者有一定的相似性，所以Go实现时抽象出一个 `testing.common` 作为一个基础类型，而 `testing.T` 和 `testing.B` 则属于 `testing.common` 的扩展。

本节，我们重点看 `testing.common` ，通过其成员及方法，来了解其实现原理。

数据结构

```
// common holds the elements common between T and B and
// captures common methods such as Errorf.
type common struct {
    mu       sync.RWMutex         // guards this group of fields
    output   []byte               // Output generated by test or benchmark.
    w        io.Writer            // For flushToParent.
    ran      bool                 // Test or benchmark (or one of its subtests) was executed.
    failed   bool                 // Test or benchmark has failed.
    skipped  bool                 // Test of benchmark has been skipped.
    done     bool                 // Test is finished and all subtests have completed.
    helpers map[string]struct{} // functions to be skipped when writing file/line info

    chatty     bool   // A copy of the chatty flag.
    finished   bool   // Test function has completed.
    hasSub     int32  // written atomically
    raceErrors int    // number of races detected during test
    runner     string // function name of tRunner running the test

    parent   *common
    level    int       // Nesting depth of test or benchmark.
    creator  []uintptr // If level > 0, the stack trace at the point where the parent called t.Run.
    name     string    // Name of test or benchmark.
    start    time.Time // Time test or benchmark started
    duration time.Duration
    barrier  chan bool // To signal parallel subtests they may start.
    signal   chan bool // To signal a test is done.
    sub      []*T      // Queue of subtests to be run in parallel.
}
```

common.mu

读写锁，仅用于控制本数据内的成员访问。

common.output

存储当前测试产生的日志，每产生一条日志则追加到该切片中，待测试结束后再一并输出。

common.w

子测试执行结束需要把产生的日志输送到父测试中的output切片中，传递时需要考虑缩进等格式调整，通过w把日志传递到父测试。

common.ran

仅表示是否已执行过。比如，根据某个规范筛选测试，如果没有测试被匹配到的话，则common.ran为false，表示没有测试运行过。

common.failed

如果当前测试执行失败，则置为true。

common.skipped

标记当前测试是否已跳过。

common.done

表示当前测试及其子测试已结束，此状态下再执行Fail()之类的方法标记测试状态会产生panic。

common.helpers

标记当前为函数为help函数，其中打印的日志，在记录日志时不会显示其文件名及行号。

common.chatty

对应命令行中的-v参数，默认为false，true则打印更多详细日志。

common.finished

如果当前测试结束，则置为true。

common.hasSub

标记当前测试是否包含子测试，当测试使用t.Run()方法启动子测试时，t.hasSub则置为1。

common.raceErrors

竞态检测错误数。

common.runner

执行当前测试的函数名。

common.parent

如果当前测试为子测试，则置为父测试的指针。

common.level

测试嵌套层数，比如创建子测试时，子测试嵌套层数就会加1。

common.creator

测试函数调用栈。

common.name

记录每个测试函数名，比如测试函数 `TestAdd(t *testing.T)` ，其中t.name即“TestAdd”。
测试结束，打印测试结果会用到该成员。

common.start

记录测试开始的时间。

common.duration

记录测试所花费的时间。

common.barrier

用于控制父测试和子测试执行的channel，如果测试为Parallel，则会阻塞等待父测试结束后再继续。

common.signal

通知当前测试结束。

common.sub

子测试列表。

成员方法

common.Name()

```
// Name returns the name of the running test or benchmark.
func (c *common) Name() string {
    return c.name
}
```

该方法直接返回common结构体中存储的名称。

common.Fail()

```

// Fail marks the function as having failed but continues execution.
func (c *common) Fail() {
    if c.parent != nil {
        c.parent.Fail()
    }
    c.mu.Lock()
    defer c.mu.Unlock()
    // c.done needs to be locked to synchronize checks to c.done in parent tests.
    if c.done {
        panic("Fail in goroutine after " + c.name + " has completed")
    }
    c.failed = true
}
```

Fail()方法会标记当前测试为失败，然后继续运行，并不会立即退出当前测试。如果是子测试，则除了标记当前测试结果外还通过 `c.parent.Fail()` 来标记父测试失败。

common.FailNow()

```
func (c *common) FailNow() {
    c.Fail()
    c.finished = true
    runtime.Goexit()
}
```

FailNow()内部会调用Fail()标记测试失败，还会标记测试结束并退出当前测试协程。
可以简单的把一个测试理解为一个协程，FailNow()只会退出当前协程，并不会影响其他测试协程，但要保证在当前测试协程中调用FailNow()才有效，不可以在当前测试创建的协程中调用该方法。

common.log()

```
func (c *common) log(s string) {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.output = append(c.output, c.decorate(s)...)
}
```

common.log()为内部记录日志入口，日志会统一记录到common.output切片中，测试结束时再统一打印出来。
日志记录时会调用common.decorate()进行装饰，即加上文件名和行号，还会做一些其他格式化处理。
调用common.log()的方法，有Log()、Logf()、Error()、Errorf()、Fatal()、Fatalf()、Skip()、Skipf()等。

注意：单元测试中记录的日志只有在执行失败或指定了 `-v` 参数才会打印，否则不会打印。而在性能测试中则总是被打印出来，因为是否打印日志有可能影响性能测试结果。

common.Log(args ...interface{})

```
func (c *common) Log(args ...interface{}) {
    c.log(fmt.Sprintln(args...))
}
```

common.Log()方法用于记录简单日志，通过fmt.Sprintln()方法生成日志字符串后记录。

common.Logf(format string, args ...interface{})

```
func (c *common) Logf(format string, args ...interface{}) {
    c.log(fmt.Sprintf(format, args...))
}
```

common.Logf()方法用于格式化记录日志，通过fmt.Sprintf()生成字符串后记录。

common.Error(args ...interface{})

```
// Error is equivalent to Log followed by Fail.
func (c *common) Error(args ...interface{}) {
    c.log(fmt.Sprintln(args...))
    c.Fail()
}
```

common.Error()方法等同于common.Log()+common.Fail()，即记录日志并标记失败，但测试继续进行。

common.Errorf(format string, args ...interface{})

```
// Errorf is equivalent to Logf followed by Fail.
func (c *common) Errorf(format string, args ...interface{}) {
    c.log(fmt.Sprintf(format, args...))
    c.Fail()
}
```

common.Errorf()方法等同于common.Logf()+common.Fail()，即记录日志并标记失败，但测试继续进行。

common.Fatal(args ...interface{})

```
// Fatal is equivalent to Log followed by FailNow.
func (c *common) Fatal(args ...interface{}) {
    c.log(fmt.Sprintln(args...))
    c.FailNow()
}
```

common.Fatal()方法等同于common.Log()+common.FailNow()，即记录日志、标记失败并退出当前测试。

common.Fatalf(format string, args ...interface{})

```
// Fatalf is equivalent to Logf followed by FailNow.
func (c *common) Fatalf(format string, args ...interface{}) {
    c.log(fmt.Sprintf(format, args...))
    c.FailNow()
}
```

common.Fatalf()方法等同于common.Logf()+common.FailNow()，即记录日志、标记失败并退出当前测试。

common.skip()

```
func (c *common) skip() {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.skipped = true
}
```

common.skip()方法标记当前测试为已跳过状态，比如测试中检测到某种条件，不再继续测试。该函数仅标记测试跳过，与测试结果无关。测试结果仍然取决于common.failed。

common.SkipNow()

```
func (c *common) SkipNow() {
    c.skip()
    c.finished = true
    runtime.Goexit()
}
```

common.SkipNow()方法标记测试跳过，并标记测试结束，最后退出当前测试。

common.Skip(args ...interface{})

```
// Skip is equivalent to Log followed by SkipNow.
func (c *common) Skip(args ...interface{}) {
    c.log(fmt.Sprintln(args...))
    c.SkipNow()
}
```

common.Skip()方法等同于common.Log()+common.SkipNow()。

common.Skipf(format string, args ...interface{})

```
// Skipf is equivalent to Logf followed by SkipNow.
func (c *common) Skipf(format string, args ...interface{}) {
    c.log(fmt.Sprintf(format, args...))
    c.SkipNow()
}
```

common.Skipf()方法等同于common.Logf() + common.SkipNow()。

common.Helper()

```
// Helper marks the calling function as a test helper function.
// When printing file and line information, that function will be skipped.
// Helper may be called simultaneously from multiple goroutines.
func (c *common) Helper() {
    c.mu.Lock()
    defer c.mu.Unlock()
    if c.helpers == nil {
        c.helpers = make(map[string]struct{})
    }
    c.helpers[callerName(1)] = struct{}{}
}
```

common.Helper()方法标记当前函数为 `help` 函数，所谓 `help` 函数，即其中打印的日志，不记录 `help` 函数的函数名及行号，而是记录上一层函数的函数名和行号。

testing.TB接口

简介

TB接口，顾名思义，是testing.T(单元测试)和testing.B(性能测试)共用的接口。

TB接口通过在接口中定义一个名为private()的私有方法，保证了即使用户实现了类似的接口，也不会跟testing.TB接口冲突。

其实，这些接口在testing.T和testing.B公共成员testing.common中已经实现。

接口定义

在 `src/testing/testing.go` 中定义了testing.TB接口：

```
// TB is the interface common to T and B.
type TB interface {
    Error(args ...interface{})
    Errorf(format string, args ...interface{})
    Fail()
    FailNow()
    Failed() bool
    Fatal(args ...interface{})
    Fatalf(format string, args ...interface{})
    Log(args ...interface{})
    Logf(format string, args ...interface{})
    Name() string
    Skip(args ...interface{})
    SkipNow()
    Skipf(format string, args ...interface{})
    Skipped() bool
    Helper()

    // A private method to prevent users implementing the
    // interface and so future additions to it will not
    // violate Go 1 compatibility.
    private()
}
```

其中对外接口需要testing.T和testing.B实现，但由于testing.T和testing.B都继承了testing.common，而testing.common已经实现了这些接口，所以testing.T和testing.B天然实现了TB接口。

其中私有接口 `private()` 用于控制该接口的唯一性，即便用户代码中某个类型实现了这些方法，由于无法实现这个私有接口，也不能被认为是实现了TB接口，所以不会跟用户代码产生冲突。

接口分类

我们在testing.common部分介绍过每个接口的实现，我们接下来就从函数功能上对接口进行分类。

以单元测试为例，每个测试函数都需要接收一个testing.T类型的指针作为函数参数，该参数主要用于控制测试流程（如结束和跳过）和记录日志。

记录日志

- Log(args ...interface{})
- Logf(format string, args ...interface{})

Log()和Logf()负责记录日志，其区别在于是否支持格式化参数；

标记失败+记录日志

- Error(args ...interface{})
- Errorf(format string, args ...interface{})

Error()和Errorf()负责标记当前测试失败并记录日志。
只标记测试状态为失败，并不影响测试函数流程，不会结束当前测试，也不会退出当前测试。

标记失败+记录日志+结束测试

- Fatal(args ...interface{})
- Fatalf(format string, args ...interface{})

Fatal()和Fatalf()负责标记当前测试失败、记录日志，并退出当前测试。

标记失败

- Fail()

Fail()仅标记录前测试状态为失败。

标记失败并退出

- FailNow()

FailNow()标记当前测试状态为失败并退出当前测试。

跳过测试+记录日志并退出

- Skip(args ...interface{})
- Skipf(format string, args ...interface{})

Skip()和Skipf()标记当前测试状态为跳过并记录日志，最后退出当前测试。

跳过测试并退出

- SkipNow()

SkipNow()标记测试状态为跳过，并退出当前测试。

私有接口避免冲突

接口定义中的private()方法是一个值得学习的用法。其目的是限定testing.TB接口的全局唯一性，即便用户的某个类型实现了除private()方法以外的其他方法，也不能说明实现了testing.TB接口，因为无法实现private()方法，private()方法属于testing包内部可见，外部不可见。

单元测试实现原理

简介

在了解过testing.common后，我们进一步了解testing.T数据结构，以便了解更多单元测试执行的更多细节。

数据结构

源码包 `src/testing/testing.go:T` 定义了其数据结构：

```
type T struct {
    common
    isParallel bool
    context    *testContext // For running tests and subtests.
}
```

其成员简单介绍如下：

- common： 即前面绍的testing.common
- isParallel： 表示当前测试是否需要并发，如果测试中执行了t.Parallel()，则此值为true
- context： 控制测试的并发调度

因为context直接决定了单元测试的调度，在介绍testing.T支持的方法前，有必要先了解一下context。

testContext

源码包 `src/testing/testing.go:testContext` 定义了其数据结构：

```
type testContext struct {
    match *matcher

    mu sync.Mutex

    // Channel used to signal tests that are ready to be run in parallel.
    startParallel chan bool

    // running is the number of tests currently running in parallel.
    // This does not include tests that are waiting for subtests to complete.
    running int

    // numWaiting is the number tests waiting to be run in parallel.
    numWaiting int

    // maxParallel is a copy of the parallel flag.
    maxParallel int
}
```

testContext成员简单介绍如下：

- match：匹配器，用于管理测试名称匹配、过滤等。
- mu：互斥锁，用于控制testContext成员的互斥访问；

- startParallel： 用于通知测试可以并发执行的控制管道，测试并发达到最大限制时，需要阻塞等待该管道的通知事件；
- running： 当前并发执行的测试个数；
- numWaiting：等待并发执行的测试个数，所有等待执行的测试都阻塞在startParallel管道处；
- maxParallel：最大并发数，默认为系统CPU数，可以通过参数 `-parallel n` 指定。

testContext实现了两个方法用于控制测试的并发调度。

等待并发执行：testContext.waitParallel()

如果一个测试使用 `t.Parallel()` 启动并发，这个测试并不是立即被并发执行，需要检查当前并发执行的测试数量是否达到最大值，这个检查工作统一放在testContext.waitParallel()实现的。

testContext.waitParallel()函数的源码如下：

```
func (c *testContext) waitParallel() {
    c.mu.Lock()
    if c.running < c.maxParallel {  // 如果当前运行的测试数未达到最大值，直接返回
        c.running++
        c.mu.Unlock()
        return
    }
    c.numWaiting++                  // 如果当前运行的测试数已达最大值，需要阻塞等待
    c.mu.Unlock()
    <-c.startParallel
}
```

函数实现比较简单，如果当前运行的测试数未达最大值，将c.running++后直接返回即可，否则将c.numWaiting++并阻塞等待其他并发测试结束。

这里有个小细节，阻塞等待后面并没有累加c.running，因为其他并发的测试结束后也不会递减c.running，所以这里阻塞返回时也不用累加，一个测试结束，随即另一个测试开始，c.running个数没有变化。

并发测试结束：testContext.release()

当并发测试结束后，会通过release()方法释放一个信号，用于启动其他等待并发测试的函数。

testContext.release()函数的源码如下：

```
func (c *testContext) release() {
    c.mu.Lock()
    if c.numWaiting == 0 {          // 如果没有函数在等待，直接返回
        c.running--
        c.mu.Unlock()
        return
    }
    c.numWaiting--                  // 如果有函数在等待，释放一个信号
    c.mu.Unlock()
    c.startParallel <- true // Pick a waiting test to be run.
}
```

测试执行：tRunner()

函数tRunner用于执行一个测试，在不考虑并发测试、子测试场景下，其处理逻辑如下：

```
func tRunner(t *T, fn func(t *T)) {
    defer func() {
        t.duration += time.Since(t.start)
        signal := true

        t.report() // 测试执行结束后向父测试报告日志

        t.done = true
        t.signal <- signal // 向调度者发送结束信号
    }()

    t.start = time.Now()
    fn(t)

    t.finished = true
}
```

tRunner传一个经调度者设置过的testing.T参数和一个测试函数，执行时记录开始时间，然后将testing.T参数传入测试函数并同步等待其结束。

tRunner在defer语句中记录测试执行耗时，并上报日志，最后发送结束信号。

为了避免困惑，上述代码屏蔽了一些子测试和并发测试的细节，比如，defer语句中，如果当前测试包含子测试，则需要等所有子测试结束，如果当前测试为并发测试，则需要唤醒其他等待并发的测试。更多细节，等我们分析Parallel()和Run()时再讨论。

启动子测试：Run()

Run()函数的完整函数声明为：

```
func (t *T) Run(name string, f func(t *T)) bool
```

Run()函数启动一个单独的协程来运行名字为 `name` 的子测试 `f` ，并且会阻塞等待其执行结束，
除非子测试 `f` 显式地调用 `t.Parallel()` 将自己变成一个可并行的测试，最后返回 `bool` 类型的测试结果。

比如，当在测试 `func TestXxx(t *testing.T)` 中调用 `Run(name, f)` 时，
Run()将启动一个名为 `TestXxx/name` 的子测试。

另外，需要知道的是所有的测试，包括 `func TestXxx(t *testing.T)` 自身，都是由 `TestMain` 使用Run()方法直接或间接启动的。

按照惯例，隐去部分代码后的Run()方法如下所示：

```
func (t *T) Run(name string, f func(t *T)) bool {
    t = &T{ // 创建一个新的testing.T用于执行子测试
        common: common{
            barrier: make(chan bool),
            signal:  make(chan bool),
            name:    testName,  // 测试名字，由name及父测试名字组合而成
            parent:  &t.common,
            level:   t.level + 1, // 子测试层次+1
            chatty:  t.chatty,
        },
        context: t.context, // 子测试的context与父测试相同
    }
    go tRunner(t, f) // 启动协程执行子测试
    if !<-t.signal { // 阻塞等待子测试结束信号，子测试要么执行结束，要么以Parallel()执行。如果信号为'false',
```

```
        说明出现异常退出
        runtime.Goexit()
    }
    return !t.failed // 返回子测试的执行结果
}
```

每启动一个子测试都会创建一个testing.T变量，该变量继承当前测试的部分属性，然后以新协程去执行，
当前测试会在子测试结束后返回子测试的结果。

子测试退出条件要么是子测试执行结束，要么是子测试设置了Paraller()，否则是异常退出。

启动并发测试：Parallel()

Parallel()方法将当前测试加入到并发队列中，其实现方法如下所示：

```
func (t *T) Parallel() {
    t.isParallel = true

    t.duration += time.Since(t.start) // 启动并发测试有可能要等待，等待期间耗时需要剔除，此处相当于先记录当前耗时，并发执行开始后再累加

    t.parent.sub = append(t.parent.sub, t) // 将当前测试加入到父测试的列表中，由父测试调度

    t.signal <- true   // Release calling test. 当前测试即将进入并发模式，标记测试结束，以便父测试不必等待并退出Run()
    <-t.parent.barrier // Wait for the parent test to complete.  等待父测试发送子测试启动信号
    t.context.waitParallel() // 阻塞等待并发调度

    t.start = time.Now() // 开始并发执行，重新标记启动时间，这是第二段耗时
}
```

关于测试耗时统计，看过前面的testContext实现我们知道，启动一个并发测试时，当并发数达到最大时，新的并发测试需要等待，那么等待期间的时间消耗不能统计到测试的耗时中，所以需要先计算当前耗时，在真正被并发调度后才清空t.start以跳过等待时间。

看过前面的Run()方法实现机制后，我们知道一旦子测试以并发模式执行时，需要通知父测试，其通知机制便是向t.signal管道中写入一个信号，父测试便从Run()方法中唤醒，继续执行。

看过前面的tRunner()方法实现机制后，不难理解，父测试唤醒后继续执行，结束后进入defer流程中，在defer中将启动所有子测试并等待子测试执行结束。

完整的测试执行：tRunner()

与简单版的测试执行所不同的是，defer语句中增加了子测试、并发测试的处理逻辑，相对完整的tRunner()代码如下所示：

```
func tRunner(t *T, fn func(t *T)) {
    t.runner = callerName(0)

    defer func() {
        t.duration += time.Since(t.start) // 进入defer后立即记录测试执行时间,后续流程所花费的时间不应该统计到本测试执行用时中

        if len(t.sub) > 0 { // 如果存在子测试，则启动并等待其完成
            t.context.release() // 减少运行计数
            close(t.barrier)   // 启动子测试
            for _, sub := range t.sub { // 等待所有子测试结束
```

```
                <-sub.signal
            }
            if !t.isParallel { // 如果当前测试非并发模式，则等待并发执行，类似于测试函数中执行t.Parallel()
                t.context.waitParallel()
            }
        } else if t.isParallel { // 如果当前测试是并发模式，则释放信号以启动新的测试
            t.context.release()
        }
        t.report() // 测试执行结束后向父测试报告日志

        t.done = true
        t.signal <- signal // 向父测试发送结束信号，以便结束Run()
    }()

    t.start = time.Now() // 记录测试开始时间
    fn(t)

    // code beyond here will not be executed when FailNow is invoked
    t.finished = true
}
```

测试执行结束，进入defer后需要启动子测试，启动方法为关闭t.barrier管道，然后等待所有子测试执行结束。

需要注意的是，关闭t.barrier管道，阻塞在t.barrier管道上的协程同样会被唤醒，也是发送信号的一种方式，关于管道的更多实现细节，请参考管道实现原理相关章节。

defer中，如果检测到当前测试本身也处于并发中，那么结束后需要释放一个信号（t.context.release()）来启动一个等待的测试。

性能测试实现原理

简介

根据前面章节，我们可以快速的写出一个性能测试并执行，最令我感到神奇的是b.N的值，虽然官方资料中说b.N会自动调整以保证可靠的计时，可还是想了解具体的实现机制。

本节，我们先分析testing.B数据结构，再看几个典型的成员函数，以期从源码中寻找以下问题的答案：

- b.N是如何自动调整的？
- 内存统计是如何实现的？
- SetBytes()其使用场景是什么？

数据结构

源码包 `src/testing/benchmark.go:B` 定义了性能测试的数据结构，我们提取其比较重要的一些成员进行分析：

```
type B struct {
    common                      // 与testing.T共享的testing.common，负责记录日志、状态等
    importPath       string // import path of the package containing the benchmark
    context          *benchContext
    N                int           // 目标代码执行次数，不需要用户了解具体值，会自动调整
    previousN        int           // number of iterations in the previous run
    previousDuration time.Duration // total duration of the previous run
    benchFunc        func(b *B)    // 性能测试函数
    benchTime        time.Duration // 性能测试函数最少执行的时间，默认为1s，可以通过参数'-benchtime 10s'指定
    bytes            int64         // 每次迭代处理的字节数
    missingBytes     bool // one of the subbenchmarks does not have bytes set.
    timerOn          bool // 是否已开始计时
    showAllocResult  bool
    result           BenchmarkResult // 测试结果
    parallelism      int // RunParallel creates parallelism*GOMAXPROCS goroutines
    // The initial states of memStats.Mallocs and memStats.TotalAlloc.
    startAllocs uint64  // 计时开始时堆中分配的对象总数
    startBytes  uint64  // 计时开始时时堆中分配的字节总数
    // The net total of this test after being run.
    netAllocs uint64 // 计时结束时，堆中增加的对象总数
    netBytes  uint64 // 计时结束时，堆中增加的字节总数
}
```

其主要成员如下：

- common： 与testing.T共享的testing.common，管理着日志、状态等；
- N：每个测试中用户代码执行次数
- benchFunc：测试函数
- benchTime：性能测试最少执行时间，默认为1s，可以通过能数-benchtime 2s指定
- bytes：每次迭代处理的字节数
- timerOn：计时启动标志，默认为false，启动计时为true
- startAllocs：测试启动时记录堆中分配的对象数
- startBytes：测试启动时记录堆中分配的字节数

- netAllocs：测试结束后记录堆中新增加的对象数，公式：结束时堆中分配的对象数-
- netBytes：测试对事后记录堆中新增加的字节数

关键函数

启动计时：B.StartTimer()

StartTimer()负责启动计时并初始化内存相关计数，测试执行时会自动调用，一般不需要用户启动。

```
func (b *B) StartTimer() {
    if !b.timerOn {
        runtime.ReadMemStats(&memStats)     // 读取当前堆内存分配信息
        b.startAllocs = memStats.Mallocs    // 记录当前堆内存分配的对象数
        b.startBytes = memStats.TotalAlloc  // 记录当前堆内存分配的字节数
        b.start = time.Now()                // 记录测试启动时间
        b.timerOn = true                    // 标记计时标志
    }
}
```

StartTimer()负责启动计时，并记录当前内存分配情况，不管是否有“-benchmem”参数，内存都会被统计，参数只决定是否要在结果中输出。

停止计时：B.StopTimer()

StopTimer()负责停止计时，并累加相应的统计值。

```
func (b *B) StopTimer() {
    if b.timerOn {
        b.duration += time.Since(b.start)                   // 累加测试耗时
        runtime.ReadMemStats(&memStats)                     // 读取当前堆内存分配信息
        b.netAllocs += memStats.Mallocs - b.startAllocs     // 累加堆内存分配的对象数
        b.netBytes += memStats.TotalAlloc - b.startBytes    // 累加堆内存分配的字节数
        b.timerOn = false                                   // 标记计时标志
    }
}
```

需要注意的是，StopTimer()并不一定是测试结束，一个测试中有可能有多个统计阶段，所以其统计值是累加的。

重置计时：B.ResetTimer()

ResetTimer()用于重置计时器，相应的也会把其他统计值也重置。

```
func (b *B) ResetTimer() {
    if b.timerOn {
        runtime.ReadMemStats(&memStats)     // 读取当前堆内存分配信息
        b.startAllocs = memStats.Mallocs    // 记录当前堆内存分配的对象数
        b.startBytes = memStats.TotalAlloc  // 记录当前堆内存分配的字节数
        b.start = time.Now()                // 记录测试启动时间
    }
    b.duration = 0                          // 清空耗时
    b.netAllocs = 0                         // 清空内存分配对象数
    b.netBytes = 0                          // 清空内存分配字节数
}
```

ResetTimer()比较常用，典型使用场景是一个测试中，初始化部分耗时较长，初始化后再开始计时。

设置处理字节数：B.SetBytes(n int64)

```
// SetBytes records the number of bytes processed in a single operation.
// If this is called, the benchmark will report ns/op and MB/s.
func (b *B) SetBytes(n int64) {
    b.bytes = n
}
```

这是一个比较含糊的函数，通过其函数说明很难明白其作用。

其实它是用来设置单次迭代处理的字节数，一旦设置了这个字节数，那么输出报告中将会呈现“xxx MB/s”的信息，用来表示待测函数处理字节的性能。待测函数每次处理多少字节数只有用户清楚，所以需要用户设置。

举个例子，待测函数每次执行处理1M数据，如果我们想看待测函数处理数据的性能，那么我们在测试中设置SetBytes(1024 *1024)，假如待测函数需要执行1s的话，那么结果中将会出现 “1 MB/s”（约等于）的信息。示例代码如下所示：

```
func BenchmarkSetBytes(b *testing.B) {
    b.SetBytes(1024 * 1024)
    for i := 0; i < b.N; i++ {
        time.Sleep(1 * time.Second) // 模拟待测函数
    }
}
```

打印结果：

```
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src\gotest>go test -bench SetBytes benchmark_test.go
BenchmarkSetBytes-4            1        1010392800 ns/op           1.04 MB/s
PASS
ok      command-line-arguments  1.412s
```

可以看到测试执行了一次，花费时间约1S，数据处理能力约为1MB/s。

报告内存信息：

```
func (b *B) ReportAllocs() {
    b.showAllocResult = true
}
```

ReportAllocs() 用于设置是否打印内存统计信息，与命令行参数“-benchmem”一致，但本方法只作用于单个测试函数。

性能测试是如何启动的

性能测试要经过多次迭代，每次迭代可能会有不同的b.N值，每次迭代执行测试函数一次，根据此次迭代的测试结果来分析要不要继续下一次迭代。

我们先看一下每次迭代时所用到的方法，runN():

```
func (b *B) runN(n int) {
    b.N = n                         // 指定B.N
    b.ResetTimer()                  // 清空统计数据
    b.StartTimer()                  // 开始计时
    b.benchFunc(b)                  // 执行测试
```

```
    b.StopTimer()                   // 停止计时
}
```

该方法指定b.N的值，执行一次测试函数。

与T.Run()类似，B.Run()也用于启动一个子测试，实际上用户编写的任何一个测试都是使用Run()方法启动的，我们看下B.Run()的伪代码：

```
func (b *B) Run(name string, f func(b *B)) bool {
    sub := &B{                          // 新建子测试数据结构
        common: common{
            signal:  make(chan bool),
            name:    name,
            parent:  &b.common,
        },
        benchFunc:  f,
    }
    if sub.run1() { // 先执行一次子测试，如果子测试不出错且子测试没有子测试的话继续执行sub.run()
        sub.run()       // run()里决定要执行多少次runN()
    }
    b.add(sub.result) // 累加统计结果到父测试中
    return !sub.failed
}
```

所有的测试都是先使用run1()方法执行一次测试，run1()方法中实际上调用了runN(1)，执行一次后再决定要不要继续迭代。

测试结果实际上以最后一次迭代的数据为准，当然，最后一次迭代往往意味着b.N更大，测试准确性相对更高。

B.N是如何调整的？

B.launch()方法里最终决定B.N的值。我们看下伪代码：

```
func (b *B) launch() { // 此方法自动测算执行次数，但调用前必须调用run1以便自动计算次数
    d := b.benchTime
    for n := 1; !b.failed && b.duration < d && n < 1e9; { // 最少执行b.benchTime（默认为1s）时间，最多执行1e9
次
        last := n
        n = int(d.Nanoseconds()) // 预测接下来要执行多少次，b.benchTime/每个操作耗时
        if nsop := b.nsPerOp(); nsop != 0 {
            n /= int(nsop)
        }
        n = max(min(n+n/5, 100*last), last+1) // 避免增长较快，先增长20%，至少增长1次
        n = roundUp(n) // 下次迭代次数向上取整到10的指数，方便阅读
        b.runN(n)
    }
}
```

不考虑程序出错，而且用户没有主动停止测试的场景下，每个性能测试至少要执行b.benchTime长的秒数，默认为1s。先执行一遍的意义在于看用户代码执行一次要花费多长时间，如果时间较短，那么b.N值要足够大才可以测得更精确，如果时间较长，b.N值相应的会减少，否则会影响测试效率。

最终的b.N会被定格在某个10的指数级，是为了方便阅读测试报告。

内存是如何统计的？

我们知道在测试开始时，会把当前内存值记入到b.startAllocs和b.startBytes中，测试结束时，会用最终内存值与开始时的内存值相减，得到净增加的内存值，并记入到b.netAllocs和b.netBytes中。

每个测试结束，会把结果保存到BenchmarkResult对象里，该对象里保存了输出报告所必需的统计信息：

```
type BenchmarkResult struct {
    N         int           // 用户代码执行的次数
    T         time.Duration // 测试耗时
    Bytes     int64         // 用户代码每次处理的字节数，SetBytes()设置的值
    MemAllocs uint64        // 内存对象净增加值
    MemBytes  uint64        // 内存字节净增加值
}
```

其中MemAllocs和MemBytes分别对应b.netAllocs和b.netBytes。

那么最终统计时只需要把净增加值除以b.N即可得到每次新增多少内存了。

每个操作内存对象新增值：

```
func (r BenchmarkResult) AllocsPerOp() int64 {
    return int64(r.MemAllocs) / int64(r.N)
}
```

每个操作内存字节数新增值：

```
func (r BenchmarkResult) AllocedBytesPerOp() int64 {
    if r.N <= 0 {
        return 0
    }
    return int64(r.MemBytes) / int64(r.N)
}
```

示例测试实现原理

简介

示例测试相对于单元测试和性能测试来说，其实现机制比较简单。它没有复杂的数据结构，也不需要额外的流程控制，其核心工作原理在于收集测试过程中的打印日志，然后与期望字符串做比较，最后得出是否一致的报告。

数据结构

每个测试经过编译后都有一个数据结构来承载，这个数据结构即 `InternalExample` ：

```
type InternalExample struct {
    Name      string    // 测试名称
    F         func()    // 测试函数
    Output    string    // 期望字符串
    Unordered bool      // 输出是否是无序的
}
```

比如，示例测试如下：

```
// 检测乱序输出
func ExamplePrintNames() {
    gotest.PrintNames()
    // Unordered output:
    // Jim
    // Bob
    // Tom
    // Sue
}
```

该示例测试经过编译后，产生的数据结构成员如下：

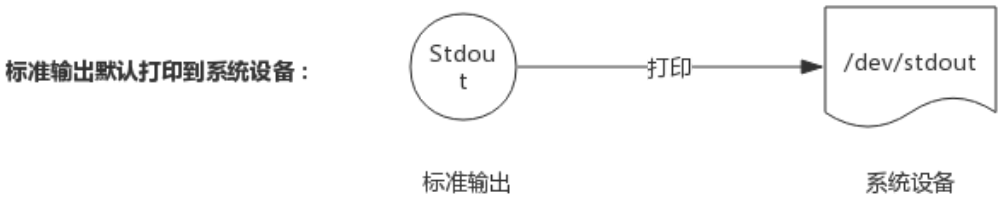
- InternalExample.Name = “ExamplePrintNames”
- InternalExample.F = ExamplePrintNames()
- InternalExample.Output = “Jim\n Bob\n Tom\n Sue\n”
- InternalExample.Unordered = true

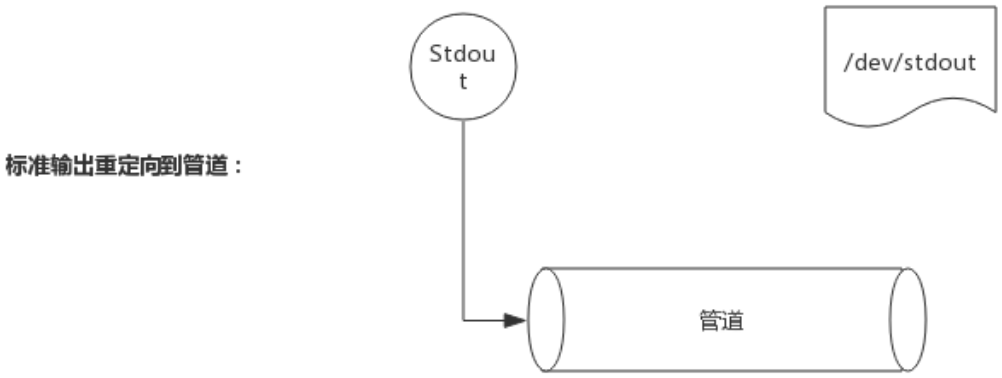
其中Output是包含换行符的字符串。

捕获标准输出

在示例测试开始前，需要先把标准输出捕获，以便获取测试执行过程中的打印日志。

捕获标准输出方法是新建一个管道，将标准输出重定向到管道的入口(写口)，这样所有打印到屏幕的日志都会输入到管道中，如下图所示：





测试开始前捕获，测试结束恢复标准输出，这样测试过程中的日志就可以从管道中读取了。

测试结果比较

测试执行过程的输出内容最终也会保存到一个string类型变量里，该变量会与InternalExample.Output进行比较，二者一致即代表测试通过，否则测试失败。

输出有序的情况下，比较很简单只是比较两个String内容是否一致即可。无序的情况下则需要把两个String变量排序后再进行对比。

比如，期望字符串为：”Jim\n Bob\n Tom\n Sue\n”，排序后则变为：”Bob\n Jim\n Sue\n Tom\n”

测试执行

一个完整的测试，过程将分为如下步骤：

1. 捕获标准输出
2. 执行测试
3. 恢复标准输出
4. 比较结果

下面，由于源码非常简单，下面直接给出源码：

```
func runExample(eg InternalExample) (ok bool) {
    if *chatty {
        fmt.Printf("=== RUN   %s\n", eg.Name)
    }
```

```
    // Capture stdout.
    stdout := os.Stdout      // 备份标输出文件
    r, w, err := os.Pipe()   // 创建一个管道
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    os.Stdout = w            // 标准输出文件暂时修改为管道的入口，即所有的标准输出实际上都会进入管道
    outC := make(chan string)
    go func() {
        var buf strings.Builder
        _, err := io.Copy(&buf, r)  // 从管道中读出数据
        r.Close()
        if err != nil {
            fmt.Fprintf(os.Stderr, "testing: copying pipe: %v\n", err)
            os.Exit(1)
        }
        outC <- buf.String()  // 管道中读出的数据写入channel中
    }()

    start := time.Now()
    ok = true

    // Clean up in a deferred call so we can recover if the example panics.
    defer func() {
        dstr := fmtDuration(time.Since(start))         // 计时结束，记录测试用时

        // Close pipe, restore stdout, get output.
        w.Close()                // 关闭管道
        os.Stdout = stdout       // 恢复原标准输出
        out := <-outC            // 从channel中取出数据

        var fail string
        err := recover()
        got := strings.TrimSpace(out)         // 实际得到的打印字符串
        want := strings.TrimSpace(eg.Output) // 期望的字符串
        if eg.Unordered { // 如果输出是无序的，则把输出字符串和期望字符串排序后比较
            if sortLines(got) != sortLines(want) && err == nil {
                fail = fmt.Sprintf("got:\n%s\nwant (unordered):\n%s\n", out, eg.Output)
            }
        } else { // 如果输出是有序的，则直接比较输出字符串和期望字符串
            if got != want && err == nil {
                fail = fmt.Sprintf("got:\n%s\nwant:\n%s\n", got, want)
            }
        }
        if fail != "" || err != nil {
            fmt.Printf("--- FAIL: %s (%s)\n%s", eg.Name, dstr, fail)
            ok = false
        } else if *chatty {
            fmt.Printf("--- PASS: %s (%s)\n", eg.Name, dstr)
        }
        if err != nil {
            panic(err)
        }
    }()

    // Run example.
    eg.F()
    return
}
```

示例测试执行时，捕获标准输出后，马上启动一个协程阻塞在管道处读取数据，一直阻塞到管道关闭，管道关闭也即读取结束，然后把日志通过channel发送到主协程中。

主协程直接执行示例测试，而在defer中去执行关闭管道、接收日志、判断结果等操作。

Main测试实现原理

简介

每一种测试（单元测试、性能测试或示例测试），都有一个数据类型与其对应。

- 单元测试：InternalTest
- 性能测试：InternalBenchmark
- 示例测试：InternalExample

测试编译阶段，每个测试都会被放到指定类型的切片中，测试执行时，这些测试将会被放到testing.M数据结构中进行调度。

而testing.M即是MainTest对应的数据结构。

数据结构

源码 `src/testing/testing.go:M` 定义了testing.M的数据结构：

```
// M is a type passed to a TestMain function to run the actual tests.
type M struct {
    tests      []InternalTest       // 单元测试
    benchmarks []InternalBenchmark  // 性能测试
    examples   []InternalExample    // 示例测试
    timer     *time.Timer           // 测试超时时间
}
```

单元测试、性能测试和示例测试在经过编译后都会被存放到一个testing.M数据结构中，在测试执行时该数据结构将传递给TestMain()，真正执行测试的是testing.M的Run()方法，这个后面我们会继续分析。

timer用于指定测试的超时时间，可以通过参数 `timeout <n>` 指定，当测试执行超时后将会立即结束并判定为失败。

执行测试

TestMain()函数通常会有一个m.Run()方法，该方法会执行单元测试、性能测试和示例测试，如果用户实现了TestMain()但没有调用m.Run()的话，那么什么测试都不会被执行。

m.Run()不仅会执行测试，还会做一些初始化工作，比如解析参数、启动定时器、根据参数指示创建一系列的文件等。

m.Run()使用三个独立的方法来执行三种测试：

- 单元测试：runTests(m.deps.MatchString, m.tests)
- 性能测试：runExamples(m.deps.MatchString, m.examples)
- 示例测试：runBenchmarks(m.deps.ImportPath(), m.deps.MatchString, m.benchmarks)

  其中m.deps里存放了测试匹配相关的内容，暂时先不用关注。

go test工作机制

前言

前面的章节我们分析了每种测试的数据结构及其实现原理，本节我们看一下go test的执行机制。

Go 有多个命令行工具，go test只是其中一个。go test命令的函数入口
在 `src/cmd/go/internal/test/test.go:runTest()` ，这个函数就是go test的大脑。

runTest()

runTest()函数场景如下：

```
func runTest(cmd *base.Command, args []string)
```

GO 命令行工具的实现中，都遵循这种函数声明，其中args即命令行输入的全部参数。

runTest首先会分析所有需要测试的包，为每个待测包生成一个二进制文件，然后执行。

两种运行模式

go test运行时，根据是否指定package分为两种模式，即本地目录模式和包列表模式。

本地目录模式

当执行测试并没有指定package时，即以本地目录模式运行，例如使用”go test”或者”go test -v”来启动测试。

本地目录模式下，go test编译当前目录的源码文件和测试文件，并生成一个二进制文件，最后执行并打印结果。

包列表模式

当执行测试并显式指定package时，即以包列表模式运行，例如使用”go test math”来启动测试。

包列表模式下，go test为每个包生成一个测试二进制文件，并分别执行它。
包列表模式是在Go 1.10版本才引入的，它会把每个包的测试结果写入到本地临时文件中作为缓存，下次执行时会直接从缓存中读取测试结果，以便节省测试时间。

缓存机制

当满足一定的条件，测试的缓存是自动启用的，也可以显式地关闭缓存。

测试结果缓存

如果一次测试中，其参数全部来自”可缓存参数”集合，那么本次测试结果将被缓存。

可缓存参数集合如下：

- -cpu
- -list
- -parallel

- -run
- -short
- -v

需要注意的是，测试参数必须全部来自这个集合，其结果才会被缓存，没有参数或包含任一此集合之外的参数，结果都不会缓存。

使用缓存结果

如果满足条件，测试不会真正执行，而是从缓存中取出结果并呈现，结果中会有”cached”字样，表示来自缓存。

使用缓存结果也需要满足一定的条件：

- 本次测试的二进制及测试参数与之前的一次完全一致；
- 本次测试的源文件及环境变量与之前的一次完全一致；
- 之前的一次测试结果是成功的；
- 本次测试运行模式是列表模式

下面演示一个使用缓存的例子：

```
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src>go test gotest
ok      gotest  3.434s

E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src>go test gotest
ok      gotest  (cached)
```

前后两次执行测试，参数没变，源文件也没变化，第二次执行时会自动从缓存中获取结果，结果中“cached”即表示结果从缓存中获取。

禁用缓存

测试时使用一个不在“可缓存参数”集合中的参数，就不会使用缓存，比较常用的方法是指定一个参数“-count=1”。

下面演示一个禁用缓存的例子：

```
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src>go test gotest
ok      gotest  3.434s

E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src>go test gotest
ok      gotest  (cached)

E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src>go test gotest -count=1
ok      gotest  3.354s
```

第三次执行使用了参数”-count=1”，所以执行时不会从缓存中获取结果。

扩展阅读

go test非常容易上手，但并不代表其功能单一，它提供了丰富的参数接口以便满足各种测试场景。

本节，我们主要介绍一些常用的参数，通过前面实现原理的学习和本节的示例，希望读者可以准确掌握其用法，以便在工作中提供便利。

测试参数

前言

go test有非常丰富的参数，一些参数用于控制测试的编译，另一些参数控制测试的执行。

有关测试覆盖率、vet和pprof相关的参数先略过，我们在讨论相关内容时再详细介绍。

控制编译的参数

-args

指示go test把-args后面的参数带到测试中去。具体的测试函数会根据此参数来控制测试流程。

-args后面可以附带多个参数，所有参数都将以字符串形式传入，每个参数作为一个string，并存放到字符串切片中。

```
// TestArgs 用于演示如何解析-args参数
func TestArgs(t *testing.T) {
    if !flag.Parsed() {
        flag.Parse()
    }

    argList := flag.Args() // flag.Args() 返回 -args 后面的所有参数，以切片表示，每个元素代表一个参数
    for _, arg := range argList {
        if arg == "cloud" {
            t.Log("Running in cloud.")
        } else {
            t.Log("Running in other mode.")
        }
    }
}
```

执行测试时带入参数：

```
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src\gotest>go test -run TestArgs -v
 -args "cloud"
TestMain setup.
=== RUN   TestArgs
--- PASS: TestArgs (0.00s)
    unit_test.go:28: Running in cloud.
PASS
TestMain tear-down.
ok      gotest  0.353s
```

通过参数-args指定传递给测试的参数。

-json

-json 参数用于指示go test将结果输出转换成json格式，以方便自动化测试解析使用。

示例如下：

```
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src\gotest>go test -run TestAdd -jso
n
{"Time":"2019-02-28T15:46:50.3756322+08:00","Action":"output","Package":"gotest","Output":"TestMain setup.\n"
}
{"Time":"2019-02-28T15:46:50.4228258+08:00","Action":"run","Package":"gotest","Test":"TestAdd"}
{"Time":"2019-02-28T15:46:50.423809+08:00","Action":"output","Package":"gotest","Test":"TestAdd","Output":"==
= RUN   TestAdd\n"}
{"Time":"2019-02-28T15:46:50.423809+08:00","Action":"output","Package":"gotest","Test":"TestAdd","Output":"--
- PASS: TestAdd (0.00s)\n"}
{"Time":"2019-02-28T15:46:50.423809+08:00","Action":"pass","Package":"gotest","Test":"TestAdd","Elapsed":0}
{"Time":"2019-02-28T15:46:50.4247922+08:00","Action":"output","Package":"gotest","Output":"PASS\n"}
{"Time":"2019-02-28T15:46:50.4247922+08:00","Action":"output","Package":"gotest","Output":"TestMain tear-dow
n.\n"}
{"Time":"2019-02-28T15:46:50.4257754+08:00","Action":"output","Package":"gotest","Output":"ok  \tgotest\t0.46
5s\n"}
{"Time":"2019-02-28T15:46:50.4257754+08:00","Action":"pass","Package":"gotest","Elapsed":0.465}
```

-o

-o 参数指定生成的二进制可执行程序，并执行测试，测试结束不会删除该程序。

没有此参数时，go test生成的二进制可执行程序存放到临时目录，执行结束便删除。

示例如下：

```
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src\gotest>go test -run TestAdd -o T
estAdd
TestMain setup.
PASS
TestMain tear-down.
ok      gotest  0.439s
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src\gotest>TestAdd
TestMain setup.
PASS
TestMain tear-down.
```

本例中，使用-o 参数指定生成二进制文件"TestAdd"并存放到当前目录，测试执行结束后，仍然可以直接执行该二进制程序。

控制测试的参数

-bench regexp

go test默认不执行性能测试，使用-bench参数才可以运行，而且只运行性能测试函数。

其中正则表达式用于筛选所要执行的性能测试。如果要执行所有的性能测试，使用参数"-bench ."或"-bench=."。

此处的正则表达式不是严格意义上的正则，而是种包含关系。

比如有如下三个性能测试：

- func BenchmarkMakeSliceWithoutAlloc(b *testing.B)
- func BenchmarkMakeSliceWithPreAlloc(b *testing.B)
- func BenchmarkSetBytes(b *testing.B)

使用参数"-bench=Slice"，那么前两个测试因为都包含"Slice"，所以都会被执行，第三个测试则不会执行。

对于包含子测试的场景下，匹配是按层匹配的。举一个包含子测试的例子：

```
func BenchmarkSub(b *testing.B) {
    b.Run("A=1", benchSub1)
    b.Run("A=2", benchSub2)
    b.Run("B=1", benchSub3)
}
```

测试函数命名规则中，子测试的名字需要以父测试名字作为前缀并以"/"连接，上面的例子实际上是包含4个测试：

- Sub
- Sub/A=1
- Sub/A=2
- Sub/B=1

如果想执行三个子测试，那么使用参数"-bench Sub"。如果只想执行"Sub/A=1"，则使用参数"-bench Sub/A=1"。如果想执行"Sub/A=1"和"Sub/A=2"，则使用参数"-bench Sub/A="。

-benchtime s

-benchtime指定每个性能测试的执行时间，如果不指定，则使用默认时间1s。

例如，执定每个性能测试执行2s，则参数为："go test -bench Sub/A=1 -benchtime 2s"。

-cpu 1,2,4

-cpu 参数提供一个CPU个数的列表，提供此列表后，那么测试将按照这个列表指定的CPU数设置GOMAXPROCS并分别测试。

比如"-cpu 1,2"，那么每个测试将执行两次，一次是用1个CPU执行，一次是用2个CPU执行。
例如，使用命令"go test -bench Sub/A=1 -cpu 1,2,3,4" 执行测试：

```
BenchmarkSub/A=1                    1000           1256835 ns/op
BenchmarkSub/A=1-2                  2000            912109 ns/op
BenchmarkSub/A=1-3                  2000            888671 ns/op
BenchmarkSub/A=1-4                  2000            894531 ns/op
```

测试结果中测试名后面的-2、-3、-4分别代表执行时GOMAXPROCS的数值。 如果GOMAXPROCS为1，则不显示。

-count n

-count指定每个测试执行的次数，默认执行一次。

例如，指定测试执行2次：

```
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src\gotest>go test -bench Sub/A=1 -c
ount 2
TestMain setup.
goos: windows
goarch: amd64
pkg: gotest
BenchmarkSub/A=1-4                  2000            917968 ns/op
BenchmarkSub/A=1-4                  2000            882812 ns/op
PASS
TestMain tear-down.
ok      gotest  10.236s
```

可以看到结果中也将呈现两次的测试结果。

如果使用-count指定执行次数的同时还指定了-cpu列表，那么测试将在每种CPU数量下执行count指定的次数。

注意，示例测试不关心-count和-cpu参数，它总是执行一次。

-failfast

默认情况下，go test将会执行所有匹配到的测试，并最后打印测试结果，无论成功或失败。

-failfast指定如果有测试出现失败，则立即停止测试。这在有大量的测试需要执行时，能够更快的发现问题。

-list regexp

-list 只是列出匹配成功的测试函数，并不真正执行。而且，不会列出子函数。

例如，使用参数”-list Sub”则只会列出包含子测试的三个测试，但不会列出子测试：

```
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src\gotest>go test -list Sub
TestMain setup.
TestSubParallel
TestSub
BenchmarkSub
TestMain tear-down.
ok      gotest  0.396s
```

-parallel n

指定测试的最大并发数。

当测试使用t.Parallel()方法将测试转为并发时，将受到最大并发数的限制，默认情况下最多有GOMAXPROCS个测试并发，其他的测试只能阻塞等待。

-run regexp

根据正则表达式执行单元测试和示例测试。正则匹配规则与-bench 类似。

-timeout d

默认情况下，测试执行超过10分钟就会超时而退出。

例时，我们把超时时间设置为1s，由本来需要3s的测试就会因超时而退出：

```
E:\OpenSource\GitHub\RainbowMango\GoExpertProgrammingSourceCode\GoExpert\src\gotest>go test -timeout=1s
TestMain setup.
panic: test timed out after 1s
```

设置超时可以按秒、按分和按时：

- 按秒设置：-timeout xs或-timeout=xs
- 按分设置：-timeout xm或-timeout=xm
- 按时设置：-timeout xh或-timeout=xh

-v

默认情况下，测试结果只打印简单的测试结果，-v 参数可以打印详细的日志。

性能测试下，总是打印日志，因为日志有时会影响性能结果。

-benchmem

默认情况下，性能测试结果只打印运行次数、每个操作耗时。使用-benchmem则可以打印每个操作分配的字节数、每个操作分配的对象数。

```
// 没有使用-benchmem
BenchmarkMakeSliceWithoutAlloc-4            2000            971191 ns/op

// 使用-benchmem
BenchmarkMakeSliceWithoutAlloc-4            2000            914550 ns/op         4654335 B/op         30 allo
cs/op
```

此处，每个操作的含义是放到循环中的操作，如下示例所示：

```
func BenchmarkMakeSliceWithoutAlloc(b *testing.B) {
    for i := 0; i < b.N; i++ {
        gotest.MakeSliceWithoutAlloc() // 一次操作
    }
}
```

基准测试分析

`benchmark` 测试是实际项目中经常使用的性能测试方法，我们可以针对某个函数或者某个功能点增加 `benchmark` 测试，
以便在CI测试中监测其性能变化，当该函数或功能性能下降时能够及时发现。

此外，在日常开发活动中或者参与开源贡献时也有可能针对某个函数或功能点做一些性能优化，此时，如何把 `benchmark` 测试数据呈现出来便非常重要了，因为你很可能在优化前后执行多次 `benchmark` 测试，手工分析这些测试结果无疑是低效的。

本节结合笔者在 `Golang` 社区参与开源贡献时的经历，介绍一下由官方推荐的性能测试分析工具 `benchstat` ，权当抛砖引玉之用。

认识数据

我们先看一个 `benchmark` 测试样本：

```
BenchmarkReadGoSum-4            2223          521556 ns/op
```

该样本包含一个测试名字 `BenchmarkReadGoSum-4` （其中 `-4` 表示测试环境为4个cpu）、测试迭代次数（2223）和每次迭代的花费的时间（521556ns）。

尽管每个样本中的时间已经是多次迭代后的平均值，但为了更好的分析性能，往往需要多个样本。

使用 `go test` 的 `-count=N` 参数可以指定执行 `benchmark` N次，从而产生N个样本，比如产生15个样本：

```
BenchmarkReadGoSum-4            2223          521556 ns/op
BenchmarkReadGoSum-4            2347          516675 ns/op
BenchmarkReadGoSum-4            2340          538406 ns/op
BenchmarkReadGoSum-4            2130          548440 ns/op
BenchmarkReadGoSum-4            2391          514602 ns/op
BenchmarkReadGoSum-4            2394          527955 ns/op
BenchmarkReadGoSum-4            2313          536693 ns/op
BenchmarkReadGoSum-4            2330          538244 ns/op
BenchmarkReadGoSum-4            2360          516426 ns/op
BenchmarkReadGoSum-4            2407          541435 ns/op
BenchmarkReadGoSum-4            2154          544386 ns/op
BenchmarkReadGoSum-4            2362          540411 ns/op
BenchmarkReadGoSum-4            2305          581713 ns/op
BenchmarkReadGoSum-4            2204          519633 ns/op
BenchmarkReadGoSum-4            1867          602543 ns/op
```

手工分析多个样本将会是一项非常有挑战的工作，因为你可能需要根据统计学规则抛弃一些异常的样本，剩下的样本再取平均值。

benchstat

`benchstat` 为Golang官方推荐的一款命令行工具，可以针对一组或多组样本进行分析，如果同时分析两组样本（比如优化前和优化后），还可以给出性能变化结果。

使用命令 `go get golang.org/x/perf/cmd/benchstat` 即可快捷安装，它将被安装到 `$GOPATH/bin` 目录中。通常我们会将该目录添加到 `PATH` 环境变量中。

使用时我们需要把 `benchmark` 测试样子输出到文件中， `benchstat` 会读取这些文件，命令格式如下：

```
benchstat [options] old.txt [new.txt] [more.txt ...]
```

分析一组样本

我们把上面的15个样本输出到名为 `BenchmarkReadGoSum.before` 的文件，然后使用 `benchstat` 分析：

```
# benchstat BenchmarkReadGoSum.before
name          time/op
ReadGoSum-4   531µs ± 3%
```

输出结果包括一个耗时平均值（531µs）和样本离散值（3%）。

分析两组样本

同上，我们把性能优化后的结果输出到名为 `BenchmarkReadGoSum.after` 的文件，然后使用 `benchstat` 分析优化的效果：

```
# benchstat BenchmarkReadGoSum.before BenchmarkReadGoSum.after
name          old time/op  new time/op  delta
ReadGoSum-4   531µs ± 3%   518µs ± 7%  -2.41%  (p=0.033 n=13+15)
```

当只有两组样本时， `benchstat` 还会额外计算出差值，比如本例中，平均花费时间下降了 `2.41%` 。

另外， `p=0.033` 表示结果的可信程度，p 值越大可信程度越低，统计学中通常把 `p=0.05` 作为临界值，超过此值说明结果不可信，可能是样本过少等原因。

`n=13+15` 表示采用的样本数量，出于某些原因(比如数据值反常，过大或过小)， `benchstat` 会舍弃某些样本，本例中优化前的数据中舍弃了两个样本，优化后的数据没有舍弃，所以 `13+15` ，表示两组样本分别采用了13和15个样本。

小结

在 `Golang` 贡献者指导文档中，特别提到如果提交的代码涉及性能变化，需要将 `benchstat` 结果上传，以便代码审核者查看。

当然，我们也可以在闭源项目中使用，比起手工分析样本， `benchstat` 明显可以大大提升效率。

> 赠人玫瑰手留余香，如果觉得不错请给个赞~
>
> 本篇文章已归档到GitHub项目，求星~ 点我即达

httptest

前面介绍了go test用于单元测试、性能测试和示例测试，但Web应用程序中往往需要与其他系统进行交互，比如通过http访问其他系统，此时就需要有一种方法用于打桩来模拟Web服务器和客户端，httptest包即Go语言针对Web应用提供的解决方案。

httptest可以方便的模拟各种Web服务器和客户端，以达到测试目的。

定时器

定时器在Go语言应用中使用非常广泛，准确掌握其用法和实现原理至关重要。

Go提供了两种定时器，此处分为一次性定时器、周期性定时器。

- 一次性定时器：定时器只计时一次，结束便停止；
- 周期性定时器：定时器周期性进行计时，除非主动停止，否则将永久运行；

本章会快速介绍这两种定时器的基本用法，重点介绍其内部实现原理，最后再给出一个案例揭示使用定时器的风险。

Timer

本小节，我们关注一次性定时器的使用方法及其使用原理。

快速开始

简介

Timer实际上是一种单一事件的定时器，即经过指定的时间后触发一个事件，这个事件通过其本身提供的channel进行通知。之所以叫单一事件，是因为Timer只执行一次就结束，这也是Timer与Ticker的最重要的区别之一。

通过timer.NewTimer(d Duration)可以创建一个timer，参数即等待的时间，时间到来后立即触发一个事件。

源码包 `src/time/sleep.go:Timer` 定义了Timer数据结构：

```
type Timer struct { // Timer代表一次定时，时间到来后仅发生一个事件。
    C <-chan Time
    r runtimeTimer
}
```

Timer对外仅暴露一个channel，指定的时间到来时就往该channel中写入系统时间，也即一个事件。

本节我们介绍Timer的几个使用场景，同时再介绍其对外呈现的方法。

使用场景

设定超时时间

有时我们希望从一个管道中读取数据，在管道中没有数据时，我们不想让程序永远阻塞在管道中，而是设定一个超时时间，在此时间段中如果管道中还是没有数据到来，则判定为超时。

Go源码包中有大量类似的用法，比如从一个连接中等待数据，其简单的用法如下代码所示：

```
func WaitChannel(conn <-chan string) bool {
    timer := time.NewTimer(1 * time.Second)

    select {
    case <- conn:
        timer.Stop()
        return true
    case <- timer.C: // 超时
        println("WaitChannel timeout!")
        return false
    }
}
```

WaitChannel作用就是检测指定的管道中是否有数据到来，通过select语句轮询conn和timer.C两个管道，timer会在1s后向timer.C写入数据，如果1s内conn还没有数据，则会判断为超时。

延迟执行某个方法

有时我们希望某个方法在今后的某个时刻执行，如下代码所示：

```
func DelayFunction() {
    timer := time.NewTimer(5 * time.Second)
```

```
    select {
    case <- timer.C:
        log.Println("Delayed 5s, start to do something.")
    }
}
```

DelayFunction()会一直等待timer的事件到来才会执行后面的方法(打印)。

Timer对外接口

创建定时器

使用方法 `func NewTimer(d Duration) *Timer` 指定一个时间即可创建一个Timer，Timer一经创建便开始计时，不需要额外的启动命令。

实际上，创建Timer意味着把一个计时任务交给系统守护协程，该协程管理着所有的Timer，当Timer的时间到达后向Timer的管道中发送当前的时间作为事件。详细的实现原理我们后面会单独介绍。

停止定时器

Timer创建后可以随时停止，停止计时器的方法是：

```
func (t *Timer) Stop() bool
```

其返回值代表定时器有没有超时：

- true: 定时器超时前停止，后续不会再有事件发送；
- false: 定时器超时后停止；

实际上，停止计时器意味着通知系统守护协程移除该定时器。详细的实现原理我们后面单独介绍。

重置定时器

已过期的定时器或者已停止的定时器，可以通过重置动作重新激活，重置方法如下：

```
func (t *Timer) Reset(d Duration) bool
```

重置的动作实质上是先停掉定时器，再启动。其返回值也即停掉计时器的返回值。

需要注意的是，重置定时器虽然可以用于修改还未超时的定时器，但正确的使用方式还是针对已过期的定时器或已被停止的定时器，同时其返回值也不可靠，返回值存在的价值仅仅是与前面版本兼容。

实际上，重置定时器意味着通知系统守护协程移除该定时器，重新设定时间后，再把定时器交给守护协程。详细的实现原理我们后面单独介绍。

简单接口

前面介绍了Timer的标准接口，time包同时还提供了一些简单的方法，在特定的场景下可以简化代码。

After()

有时我们就是想等指定的时间，没有需求提前停止定时器，也没有需求复用该定时器，那么可以使用匿名的定时器。

`func After(d Duration) <-chan Time` 方法创建一个定时器，并返回定时器的管道，如下代码所示：

```
func AfterDemo() {
    log.Println(time.Now())
    <- time.After(1 * time.Second)
    log.Println(time.Now())
}
```

AfterDemo()两条打印时间间隔为1s，实际还是一个定时器，但代码变得更简洁。

AfterFunc()

前面我们例子中讲到延迟一个方法的调用，实际上通过AfterFunc可以更简洁。AfterFunc的原型为：

```
func AfterFunc(d Duration, f func()) *Timer
```

该方法在指定时间到来后会执行函数f。例如：

```
func AfterFuncDemo() {
    log.Println("AfterFuncDemo start: ", time.Now())
    time.AfterFunc(1 * time.Second, func() {
        log.Println("AfterFuncDemo end: ", time.Now())
    })

    time.Sleep(2 * time.Second) // 等待协程退出
}
```

AfterFuncDemo()中先打印一个时间，然后使用AfterFunc启动一个定器，并指定定时器结束时执行一个方法打印结束时间。

与上面的例子所不同的是，time.AfterFunc()是异步执行的，所以需要在函数最后sleep等待指定的协程退出，否则可能函数结束时协程还未执行。

总结

本节简单介绍了Timer的常见使用场景和接口，后面的章节再介绍Ticker、以及二者的实际细节。

Timer内容总结如下：

- time.NewTimer(d)创建一个Timer;
- timer.Stop()停掉当前Timer;
- timer.Reset(d)重置当前Timer;

实现原理

前言

本节我们从Timer数据结构入手，结合源码分析Timer的实现原理。

很多人想当然的以为，启动一个Timer意味着启动了一个协程，这个协程会等待Timer到期，然后向Timer的管道中发送当前时间。

实际上，每个Go应用程序都有一个协程专门负责管理所有的Timer，这个协程负责监控Timer是否过期，过期后执行一个预定义的动作，这个动作对于Timer而言就是发送当前时间到管道中。

数据结构

Timer

源码包 `src/time/sleep.go:Timer` 定义了其数据结构：

```
type Timer struct {
    C <-chan Time
    r runtimeTimer
}
```

Timer只有两个成员：

- C: 管道，上层应用根据此管道接收事件；
- r: runtime定时器，该定时器即系统管理的定时器，对上层应用不可见；

这里应该按照层次来理解Timer数据结构，Timer.C即面向Timer用户的，Timer.r是面向底层的定时器实现。

runtimeTimer

前面我们说过，创建一个Timer实质上是把一个定时任务交给专门的协程进行监控，这个任务的载体便是 `runtimeTimer` ，简单的讲，每创建一个Timer意味着创建一个runtimeTimer变量，然后把它交给系统进行监控。我们通过设置runtimeTimer过期后的行为来达到定时的目的。

源码包 `src/time/sleep.go:runtimeTimer` 定义了其数据结构：

```
type runtimeTimer struct {
    tb uintptr                          // 存储当前定时器的数组地址
    i  int                              // 存储当前定时器的数组下标

    when   int64                        // 当前定时器触发时间
    period int64                        // 当前定时器周期触发间隔
    f      func(interface{}, uintptr)   // 定时器触发时执行的函数
    arg    interface{}                  // 定时器触发时执行函数传递的参数一
    seq    uintptr                      // 定时器触发时执行函数传递的参数二(该参数只在网络收发场景下使用)
}
```

其成员如下：

- tb: 系统底层存储runtimeTimer的数组地址；
- i: 当前runtimeTimer在tb数组中的下标；
- when: 定时器触发事件的时间；
- period: 定时器周期性触发间隔（对于Timer来说，此值恒为0）；
- f: 定时器触发时执行的回调函数，回调函数接收两个参数；
- arg: 定时器触发时执行回调函数的参数一；
- seq: 定时器触发时执行回调函数的参数二（Timer并不使用该参数）；

实现原理

一个进程中的多个Timer都由底层的一个协程来管理，为了描述方便我们把这个协程称为系统协程。

我们想在后面的章节中单独介绍系统协程工作机制，本节，我们先简单介绍其工作过程。

系统协程把runtimeTimer存放在数组中，并按照 `when` 字段对所有的runtimeTimer进行堆排序，定时器触发时执行runtimeTimer中的预定义函数 `f` ，即完成了一次定时任务。

创建Timer

我们来看创建Timer的实现，非常简单：

```
func NewTimer(d Duration) *Timer {
    c := make(chan Time, 1)  // 创建一个管道
    t := &Timer{ // 构造Timer数据结构
        C: c,                // 新创建的管道
        r: runtimeTimer{
            when: when(d),   // 触发时间
            f:    sendTime, // 触发后执行函数sendTime
            arg:  c,         // 触发后执行函数sendTime时附带的参数
        },
    }
    startTimer(&t.r) // 此处启动定时器，只是把runtimeTimer放到系统协程的堆中，由系统协程维护
    return t
}
```

NewTimer()只是构造了一个Timer，然后把Timer.r通过startTimer()交给系统协程维护。

其中when()方法是计算下一次定时器触发的绝对时间，即当前时间+NewTimer()参数d。

其中sendTime()方法便是定时器触发时的动作：

```
func sendTime(c interface{}, seq uintptr) {
    select {
    case c.(chan Time) <- Now():
    default:
    }
}
```

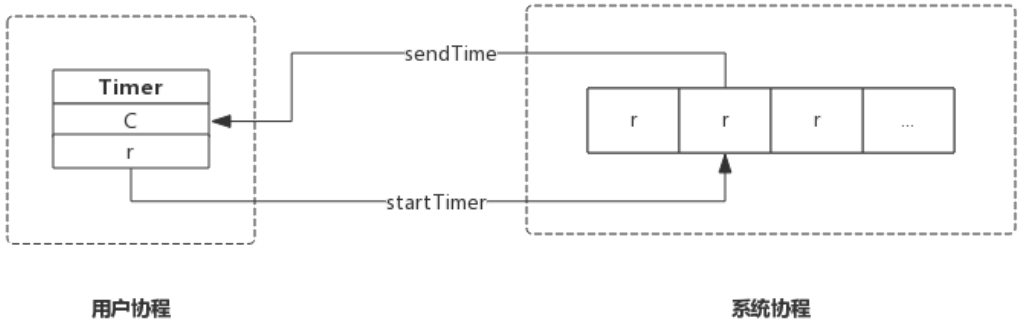
sendTime接收一个管道作为参数，其主要任务是向管道中写入当前时间。

创建Timer时生成的管道含有一个缓冲区（ `make(chan Time, 1)` ），所以Timer触发时向管道写入时间永远不会阻塞，sendTime写完即退出。

之所以sendTime()使用select并搭配一个空的default分支，是因为后面所要讲的Ticker也复用sendTime()，Ticker触发时也会向管道中写入时间，但无法保证之前的数据已被取走，所以使用select并搭配一个空的default分支，确保sendTime()不会阻塞，Ticker触发时，如果管道中还有值，则本次不再向管道中写入时间，本次触发的事件直接丢弃。

`startTimer(&t.r)` 的具体实现在runtime包，其主要作用是把runtimeTimer写入到系统协程的数组中，并启动系统协程（如果系统协程还未开始运行的话）。更详细的内容，待后面讲解系统协程时再介绍。

综上，创建一个Timer示意图如下：



停止Timer

停止Timer，只是简单的把Timer从系统协程中移除。函数主要实现如下：

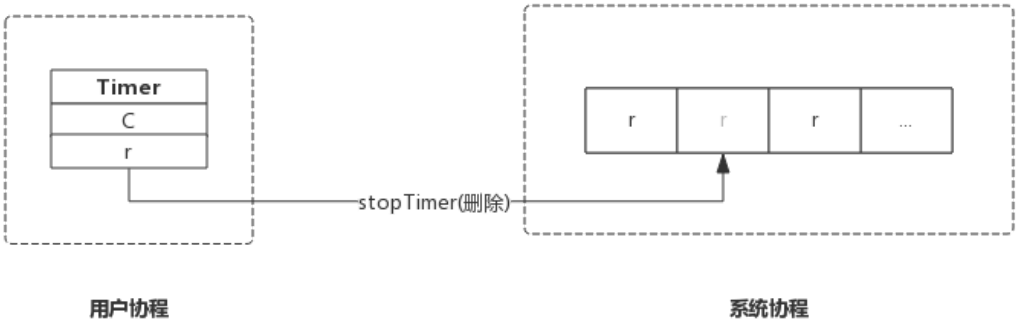
```
func (t *Timer) Stop() bool {
    return stopTimer(&t.r)
}
```

stopTimer()即通知系统协程把该Timer移除，即不再监控。系统协程只是移除Timer并不会关闭管道，以避免用户协程读取错误。

系统协程监控Timer是否需要触发，Timer触发后，系统协程会删除该Timer。所以在Stop()执行时有两种情况：

- Timer还未触发，系统协程已经删除该Timer，Stop()返回false；
- Timer已经触发，系统协程还未删除该Timer，Stop()返回true;

综上，停止一个Timer示意图如下：



重置Timer

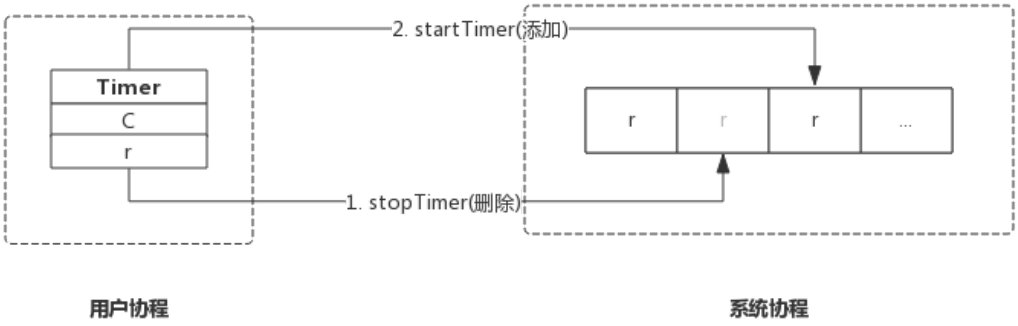
重置Timer时会先把timer从系统协程中删除，修改新的时间后重新添加到系统协程中。

重置函数主要实现如下所示：

```
func (t *Timer) Reset(d Duration) bool {
    w := when(d)
    active := stopTimer(&t.r)
    t.r.when = w
    startTimer(&t.r)
    return active
}
```

其返回值与Stop()保持一致，即如果Timer成功停止，则返回true，如果Timer已经触发，则返回false。

重置一个Timer示意图如下：



由于新加的Timer时间很可能变化，所以其在系统协程的位置也会发生变化。

需要注意的是，按照官方说明，Reset()应该作用于已经停掉的Timer或者已经触发的Timer，按照这个约定其返回值将总是返回false，之所以仍然保留是为了保持向前兼容，使用老版本Go编写的应用不需要因为Go升级而修改代码。

如果不按照此约定使用Reset()，有可能遇到Reset()和Timer触发同时执行的情况，此时有可能会收到两个事件，从而对应用程序造成一些负面影响，使用时一定要注意。

总结

- NewTimer()创建一个新的Timer交给系统协程监控；
- Stop()通知系统协程删除指定的Timer;
- Reset()通知系统协程删除指定的Timer并再添加一个新的Timer；

Ticker

本小节，我们关注周期性定时器Ticker的使用方法及其使用原理。

快速开始

简介

Ticker是周期性定时器，即周期性的触发一个事件，通过Ticker本身提供的管道将事件传递出去。

Ticker的数据结构与Timer完全一致：

```
type Ticker struct {
    C <-chan Time
    r runtimeTimer
}
```

Ticker对外仅暴露一个channel，指定的时间到来时就往该channel中写入系统时间，也即一个事件。

在创建Ticker时会指定一个时间，作为事件触发的周期。这也是Ticker与Timer的最主要的区别。

另外，ticker的英文原意是钟表的”滴哒”声，钟表周期性的产生”滴哒”声，也即周期性的产生事件。

使用场景

简单定时任务

有时，我们希望定时执行一个任务，这时就可以使用ticker来完成。

下面代码演示，每隔1s记录一次日志：

```
// TickerDemo 用于演示ticker基础用法
func TickerDemo() {
    ticker := time.NewTicker(1 * time.Second)
    defer ticker.Stop()

    for range ticker.C {
        log.Println("Ticker tick.")
    }
}
```

上述代码中，`for range ticker.C` 会持续从管道中获取事件，收到事件后打印一行日志，如果管道中没有数据会阻塞等待事件，由于ticker会周期性的向管道中写入事件，所以上述程序会周期性的打印日志。

定时聚合任务

有时，我们希望把一些任务打包进行批量处理。比如，公交车发车场景：

- 公交车每隔5分钟发一班，不管是否已坐满乘客；
- 已坐满乘客情况下，不足5分钟也发车；

下面代码演示公交车发车场景：

```
// TickerLaunch用于演示ticker聚合任务用法
func TickerLaunch() {
```

```
    ticker := time.NewTicker(5 * time.Minute)
    maxPassenger := 30                    // 每车最大装载人数
    passengers := make([]string, 0, maxPassenger)

    for {
        passenger := GetNewPassenger() // 获取一个新乘客
        if passenger != "" {
            passengers = append(passengers, passenger)
        } else {
            time.Sleep(1 * time.Second)
        }

        select {
        case <- ticker.C:                 // 时间到，发车
            Launch(passengers)
            passengers = []string{}
        default:
            if len(passengers) >= maxPassenger {  // 时间没到，车已座满，发车
                Launch(passengers)
                passengers = []string{}
            }
        }
    }
}
```

上面代码中for循环负责接待乘客上车，并决定是否要发车。每当乘客上车，select语句会先判断ticker.C中是否有数据，有数据则代表发车时间已到，如果没有数据，则判断车是否已坐满，坐满后仍然发车。

Ticker对外接口

创建定时器

使用NewTicker方法就可以创建一个周期性定时器，函数原型如下：

```
func NewTicker(d Duration) *Ticker
```

其中参数 `d` 即为定时器事件触发的周期。

停止定时器

使用定时器对外暴露的Stop方法就可以停掉一个周期性定时器，函数原型如下：

```
func (t *Ticker) Stop()
```

需要注意的是，该方法会停止计时，意味着不会向定时器的管道中写入事件，但管道并不会被关闭。管道在使用完成后，生命周期结束后会自动释放。

Ticker在使用完后务必要释放，否则会产生资源泄露，进而会持续消耗CPU资源，最后会把CPU耗尽。更详细的信息，后面我们研究Ticker实现原理时再详细分析。

简单接口

部分场景下，我们启动一个定时器并且永远不会停止，比如定时轮询任务，此时可以使用一个简单的Tick函数来获取定时器的管道，函数原型如下：

```
func Tick(d Duration) <-chan Time
```

这个函数内部实际还是创建一个Ticker，但并不会返回出来，所以没有手段来停止该Ticker。所以，一定要考虑具体的使用场景。

错误示例

Ticker用于for循环时，很容易出现意想不到的资源泄露问题，下面代码演示了一个泄露问题：

```
func WrongTicker() {
    for {
        select {
        case <-time.Tick(1 * time.Second):
            log.Printf("Resource leak!")
        }
    }
}
```

上面代码，select每次检测case语句时都会创建一个定时器，for循环又会不断地执行select语句，所以系统里会有越来越多的定时器不断地消耗CPU资源，最终CPU会被耗尽。

总结

Ticker相关内容总结如下：

- 使用time.NewTicker()来创建一个定时器；
- 使用Stop()来停止一个定时器；
- 定时器使用完毕要释放，否则会产生资源泄露；

> 赠人玫瑰手留余香，如果觉得不错请给个赞~
>
> 本篇文章已归档到GitHub项目，求星~ 点我即达

实现原理

前言

本节我们从Ticker数据结构入手，结合源码分析Ticker的实现原理。

实际上，Ticker与之前讲的Timer几乎完全相同，无论数据结构和内部实现机制都相同，唯一不同的是创建方式。

Timer创建时，不指定事件触发周期，事件触发后Timer自动销毁。而Ticker创建时会指定一个事件触发周期，事件会按照这个周期触发，如果不显式停止，定时器永不停止。

数据结构

Ticker

Ticker数据结构与Timer除名字不同外完全一样。

源码包 `src/time/tick.go:Ticker` 定义了其数据结构：

```
type Ticker struct {
    C <-chan Time // The channel on which the ticks are delivered.
    r runtimeTimer
}
```

Ticker只有两个成员：

- C: 管道，上层应用根据此管道接收事件；
- r: runtime定时器，该定时器即系统管理的定时器，对上层应用不可见；

这里应该按照层次来理解Ticker数据结构，Ticker.C即面向Ticker用户的，Ticker.r是面向底层的定时器实现。

runtimeTimer

runtimeTimer也与Timer一样，这里不再赘述。

实现原理

创建Ticker

我们来看创建Ticker的实现，非常简单：

```
func NewTicker(d Duration) *Ticker {
    if d <= 0 {
        panic(errors.New("non-positive interval for NewTicker"))
    }
    // Give the channel a 1-element time buffer.
    // If the client falls behind while reading, we drop ticks
    // on the floor until the client catches up.
    c := make(chan Time, 1)
    t := &Ticker{
        C: c,
```

```
        r: runtimeTimer{
            when:   when(d),
            period: int64(d), // Ticker跟Timer的重要区就是提供了period这个参数，据此决定timer是一次性的，还是周期性的
            f:      sendTime,
            arg:    c,
        },
    }
    startTimer(&t.r)
    return t
}
```

NewTicker()只是构造了一个Ticker，然后把Ticker.r通过startTimer()交给系统协程维护。

其中period为事件触发的周期。

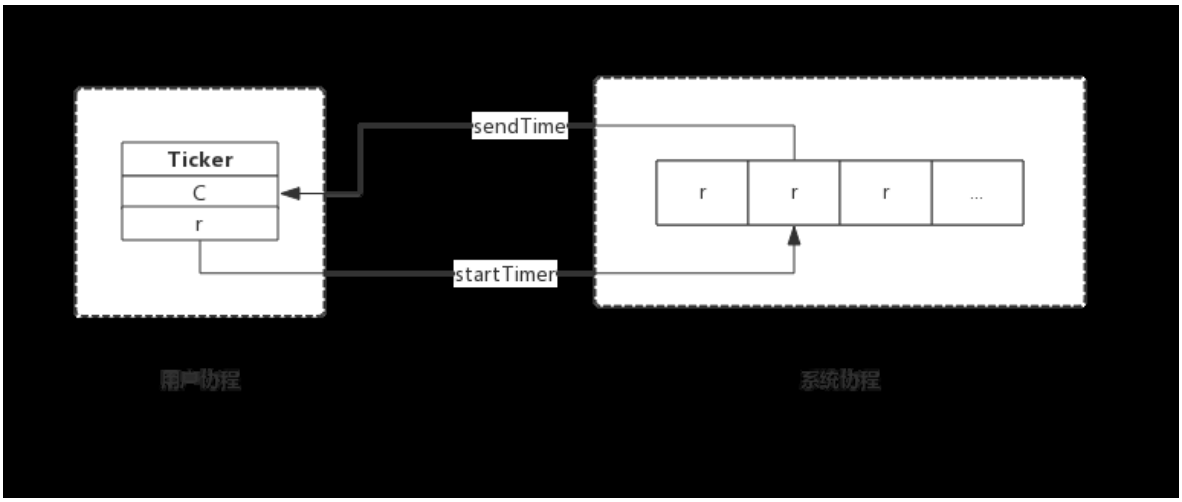
其中sendTime()方法便是定时器触发时的动作：

```
func sendTime(c interface{}, seq uintptr) {
    select {
    case c.(chan Time) <- Now():
    default:
    }
}
```

sendTime接收一个管道作为参数，其主要任务是向管道中写入当前时间。

创建Ticker时生成的管道含有一个缓冲区（ `make(chan Time, 1)` ），但是Ticker触发的事件却是周期性的，如果管道中的数据没有被取走，那么sendTime()也不会阻塞，而是直接退出，带来的后果是本次事件会丢失。

综上，创建一个Ticker示意图如下：



停止Ticker

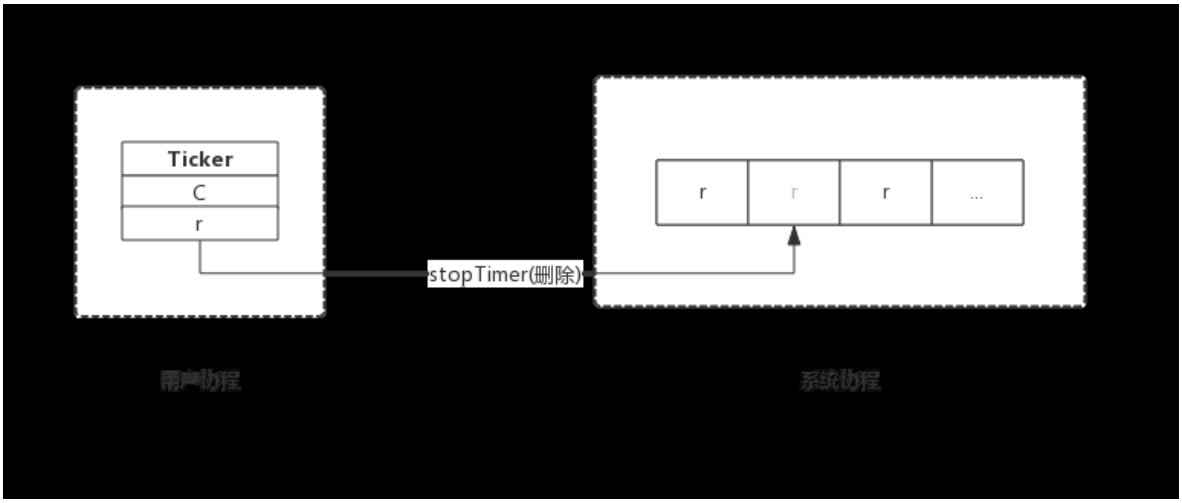
停止Ticker，只是简单的把Ticker从系统协程中移除。函数主要实现如下：

```
func (t *Ticker) Stop() {
    stopTimer(&t.r)
}
```

stopTicker()即通知系统协程把该Ticker移除，即不再监控。系统协程只是移除Ticker并不会关闭管道，以避免用户协程读取错误。

与Timer不同的是，Ticker停止时没有返回值，即不需要关注返回值，实际上返回值也没啥用途。

综上，停止一个Ticker示意图如下：



Ticker没有重置接口，也即Ticker创建后不能通过重置修改周期。

需要格外注意的是Ticker用完后必须主动停止，否则会产生资源泄露，会持续消耗CPU资源。

总结

- NewTicker()创建一个新的Ticker交给系统协程监控；
- Stop()通知系统协程删除指定的Ticker;

> 赠人玫瑰手留余香，如果觉得不错请给个赞~
>
> 本篇文章已归档到GitHub项目，求星~ 点我即达

timer

Timer和Ticker都依赖底层的timer来计时，本小节我们关注timer的实现原理。

实现原理

前言

前面我们介绍了一次性定时器Timer和周期性定时器Ticker，这两种定时器内部实现机制完全相同。创建定时器的协程并不负责计时，而是把任务交给系统协程，系统协程统一处理所有的定时器。

本节，我们重点关注系统协程是如何管理这些定器的，包括以下问题：

- 定时器使用什么数据结构存储？
- 定时器如何触发事件？
- 定时器如何添加进系统协程？
- 定时器如何从系统协程中删除？

定时器存储

timer数据结构

Timer和Ticker数据结构除名字外完全一样，二者都含有一个 `runtimeTimer` 类型的成员，这个就是系统协程所维护的对象。
`runtimeTimer` 类型是 `time` 包的名称，在runtime包中，这个类型叫做 `timer` 。

`timer` 数据结构如下所示：

```
type timer struct {
    tb *timersBucket // the bucket the timer lives in   // 当前定时器寄存于系统timer堆的地址
    i  int           // heap index                      // 当前定时器寄存于系统timer堆的下标

    when   int64                                       // 当前定时器下次触发时间
    period int64                                       // 当前定时器周期触发间隔（如果是Timer，间隔为0，表示不重复触发）
    f      func(interface{}, uintptr)                  // 定时器触发时执行的函数
    arg    interface{}                                 // 定时器触发时执行函数传递的参数一
    seq    uintptr                                     // 定时器触发时执行函数传递的参数二(该参数只在网络收发场景下使用)
}
```

其中 `timersBucket` 便是系统协程存储timer的容器，里面有个切片来存储timer，而 `i` 便是timer所在切片的下标。

timersBucket数据结构

我们来看一下 `timersBucket` 数据结构：

```
type timersBucket struct {
    lock         mutex
    gp           *g          // 处理堆中事件的协程
    created      bool        // 事件处理协程是否已创建，默认为false，添加首个定时器时置为true
    sleeping     bool        // 事件处理协程（gp）是否在睡眠(如果t中有定时器，还未到触发的时间，那么gp会投入睡眠)
    rescheduling bool        // 事件处理协程（gp）是否已暂停（如果t中定时器均已删除，那么gp会暂停）
    sleepUntil   int64       // 事件处理协程睡眠时间
    waitnote     note        // 事件处理协程睡眠事件（据此唤醒协程）
```

```
    t            []*timer    // 定时器切片
}
```

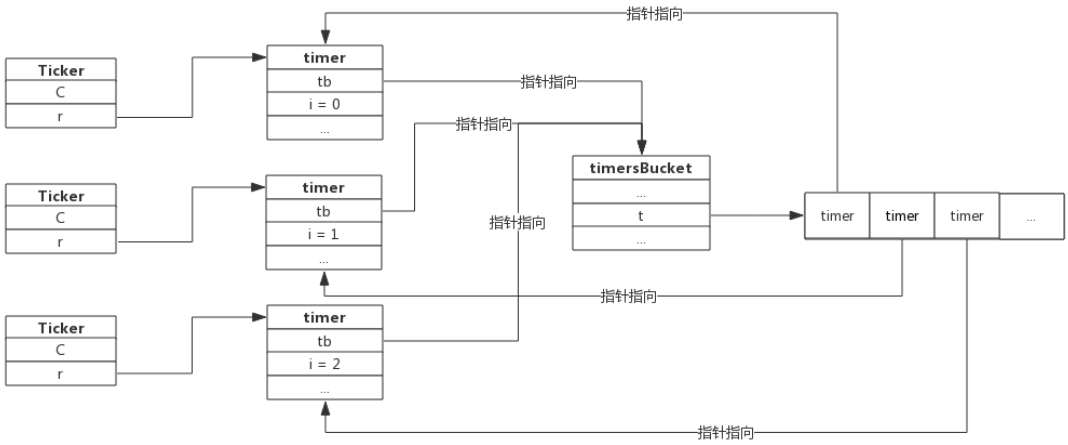
“Bucket”译成中文意为”桶”，顾名思义，`timersBucket`意为存储timer的容器。

- `lock`：互斥锁，在timer增加和删除时需要使用；
- `gp`：事件处理协程，就是我们所说的系统协程，这个协程在首次创建Timer或Ticker时生成；
- `created`：状态值，表示系统协程是否创建；
- `sleeping`：系统协程是否在睡眠；
- `rescheduling`：系统协程是否已暂停；
- `sleepUntil`：系统协程睡眠到指定的时间（如果有新的定时任务可能会提前唤醒）；
- `waitnote`：提前唤醒时使用的通知；
- `t`：保存timer的切片，当调用NewTimer()或NewTicker()时便会有新的timer存到此切片中；

看到这里应该能明白，系统协程在首次创建定时器时创建，定时器存储在切片中，系统协程负责计时并维护这个切片。

存储拓扑

以Ticker为例，我们回顾一下Ticker、timer和timersBucket关系，假设我们已经创建了3个Ticker，那么它们之间的关系如下：



用户创建Ticker时会生成一个timer，这个timer指向timersBucket，timersBucket记录timer的指针。

timersBucket数组

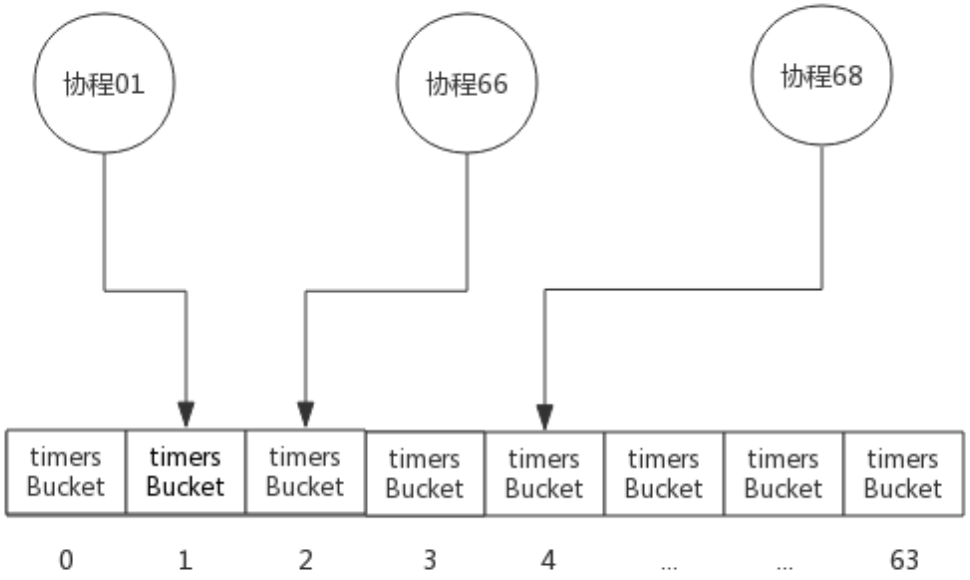
通过timersBucket数据结构可以看到，系统协程负责计时并维护其中的多个timer，一个timersBucket包含一个系统协程。

当系统中定时器非常多时，一个系统协程可能处理能力跟不上，所以Go在实现时实际上提供了多个timersBucket，也就有多个系统协程来处理定时器。

最理想的情况，应该预留`GOMAXPROCS`个timersBucket，以便充分使用CPU资源，但需要根据实际环境动态分配。为了实现简单，Go在实现时预留了64个timersBucket，绝大部分场景下这些足够了。

每当协程创建定时器时，使用协程所属的ProcessID%64来计算定时器存入的timersBucket。

下图三个协程创建定时器时，定时器分布如下图所示：



为描述方便，上图中3个协程均分布于3个Process中。

一般情况下，同一个Process的协程创建的定时器分布于同一个timersBucket中，只有当GOMAXPROCS大于64时才会出现多个Process分布于同一个timersBucket中。

定时器运行机制

看完上面的数据结构，了解了timer是如何存储的。现在开始探究定时器内部运作机制。

创建定时器

回顾一下定时器创建过程，创建Timer或Ticker实际上分为两步：

1. 创建一个管道
2. 创建一个timer并启动（注意此timer不是Timer，而是系统协程所管理的timer。）

创建管道的部分前面已做过介绍，这里我们重点关注timer的启动部分。

首先，每个timer都必须要归属于某个 `timersBucket` 的，所以第一步是先选择一个 `timersBucket` ，选择的算法很简单，将当前协程所属的Processor ID 与 `timersBucket` 数组长度求模，结果就是 `timersBucket` 数组的下标。

```
const timersLen = 64
var timers [timersLen]struct { // timersBucket数组，长度为64
    timersBucket
}
func (t *timer) assignBucket() *timersBucket {
    id := uint8(getg().m.p.ptr().id) % timersLen // Processor ID 与数组长度求模，得到下标
    t.tb = &timers[id].timersBucket
    return t.tb
}
```

至此，第一步，给当前的timer选择一个 `timersBucket` 已经完成。

其次，每个timer都必须要加入到 `timersBucket` 中。前面我们知道， `timersBucket` 中切片中保存着timer的指针，新加入的timer并不是按加入时间顺序存储的，而是把timer按照触发的时间排序的一个小头堆。那么timer加入 `timersBucket` 的过程实际上也是堆排序的过程，只不过这个排序是指的是新加元素后的堆调整过程。

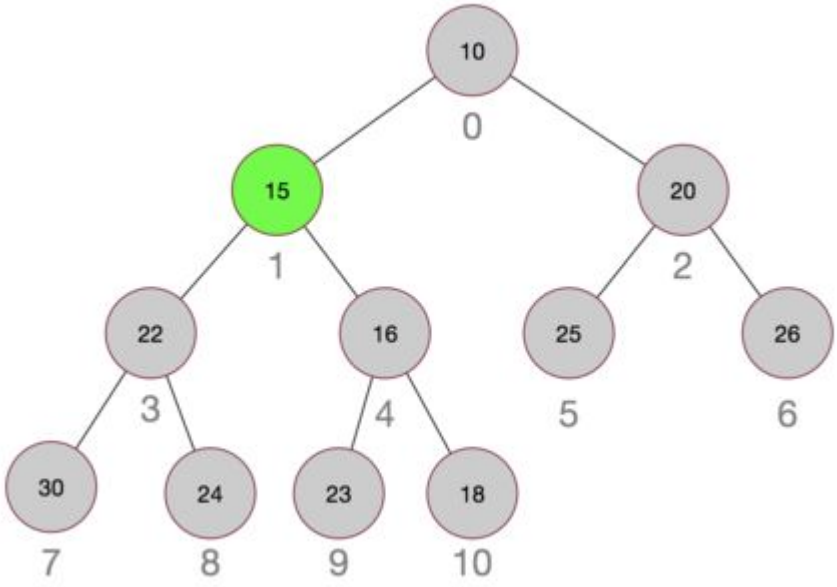
源码 `src/runtime/time.go:addtimerLocked()` 函数负责添加timer：

```
func (tb *timersBucket) addtimerLocked(t *timer) bool {
    if t.when < 0 {
        t.when = 1<<63 - 1
    }
    t.i = len(tb.t)                 // 先把定时器插入到堆尾
    tb.t = append(tb.t, t)          // 保存定时器
    if !siftupTimer(tb.t, t.i) {    // 堆中插入数据，触发堆重新排序
        return false
    }
    if t.i == 0 { // 堆排序后，发现新插入的定时器跑到了栈顶，需要唤醒协程来处理
        // siftup moved to top: new earliest deadline.
        if tb.sleeping {                // 协程在睡眠，唤醒协程来处理新加入的定时器
            tb.sleeping = false
            notewakeup(&tb.waitnote)
        }
        if tb.rescheduling {            // 协程已暂停，唤醒协程来处理新加入的定时器
            tb.rescheduling = false
            goready(tb.gp, 0)
        }
    }
    if !tb.created {        // 如果是系统首个定时器，则启动协程处理堆中的定时器
        tb.created = true
        go timerproc(tb)
    }
    return true
}
```

根据注释来理解上面的代码比较简单，这里附加几点说明：

1. 如果timer的时间是负值，那么会被修改为很大的值，来保证后续定时算法的正确性；
2. 系统协程是在首次添加timer时创建的，并不是一直存在；
3. 新加入timer后，如果新的timer跑到了栈顶，意味着新的timer需要立即处理，那么会唤醒系统协程。

下图展示一个小顶堆结构，图中每个圆圈代表一个timer，圆圈中的数字代表距离触发事件的秒数，圆圈外的数字代表其在切片中的下标。其中timer 15是新加入的，加入后它被最终调整到数组的1号下标。



上图展示的是二叉堆，实际上Go实现时使用的是四叉堆，使用四叉堆的好处是堆的高度降低，堆调整时更快。

删除定时器

当Timer执行结束或Ticker调用Stop()时会触发定时器的删除。从 `timersBucket` 中删除定时器是添加定时器的逆过程，即堆中元素删除后，触发堆调整。在此不再细述。

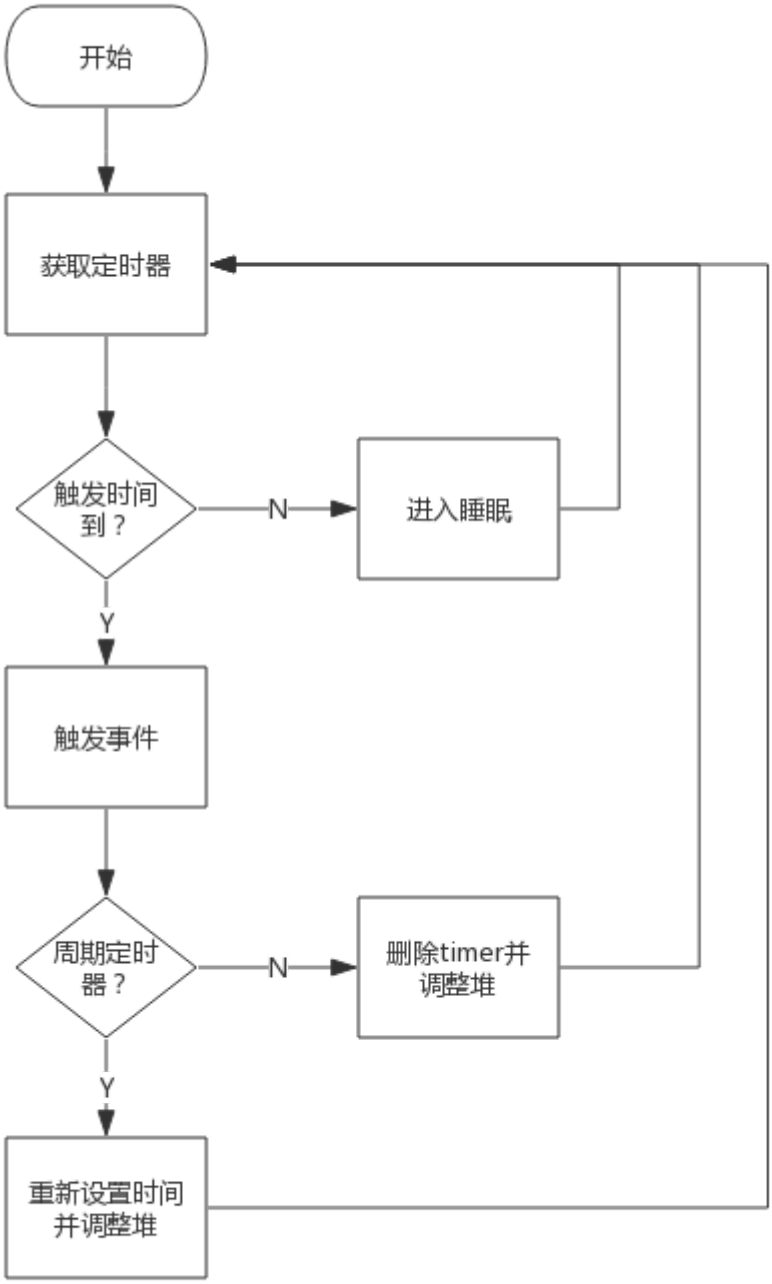
timerproc

timerproc为系统协程的具体实现。它是在首次创建定时器创建并启动的，一旦启动永不销毁。
如果 `timersBucket` 中有定时器，取出堆顶定时器，计算睡眠时间，然后进入睡眠，醒来后触发事件。

某个timer的事件触发后，根据其是否是周期性定时器来决定将其删除还是修改时间后重新加入堆。

如果堆中已没有事件需要触发，则系统协程将进入暂停态，也可认为是无限时睡眠，直到有新的timer加入才会被唤醒。

timerproc处理事件的流程图如下：



资源泄露问题

前面介绍Ticker时格外提醒不使用的Ticker需要显式地Stop()，否则会产生资源泄露。研究过timer实现机制后，可以很好的解释这个问题了。

首先，创建Ticker的协程并不负责计时，只负责从Ticker的管道中获取事件；
其次，系统协程只负责定时器计时，向管道中发送事件，并不关心上层协程如何处理事件；

如果创建了Ticker，则系统协程将持续监控该Ticker的timer，定期触发事件。如果Ticker不再使用且没有Stop()，那么系统协程负担会越来越重，最终将消耗大量的CPU资源。

> 赠人玫瑰手留余香，如果觉得不错请给个赞~
>
> 本篇文章已归档到GitHub项目，求星~ 点我即达

案例

前面我们讨论了如果定时器使用不当会有资源泄露的风险，使用时需要格外注意。

实际项目中发生Ticker资源泄露的场景有如下几种：

1. 创建了Ticker，忘记在使用结束后Stop；
2. 从别处拷贝代码未拷贝Stop语句；
3. 开源或第三方库中发生泄露；

对于前两种，推荐创建Ticker后立即使用defer语句将Ticker停掉，比如类似下面的代码：

```
    ticker := time.NewTicker(1 * time.Second)
    defer ticker.Stop()
```

使用defer是安全的，因为只有当函数退出时才会执行，上面两行代码甚至可以写到一行中。

经常使用开源或第三方库时难免会遇到资源泄露的问题，这时我们就需要掌握一些基本的定位手段，
接下来分享一个自己在做项目遇到的开源库资源泄露的案例。案例中使用的pprof工具，本处只做简单说明，详细的使用方法请参考相关章节。

开源库资源泄露

前言

前面我们研究了Ticker的实现原理，已经知道Ticker如果不主动停止会有资源泄露的问题。

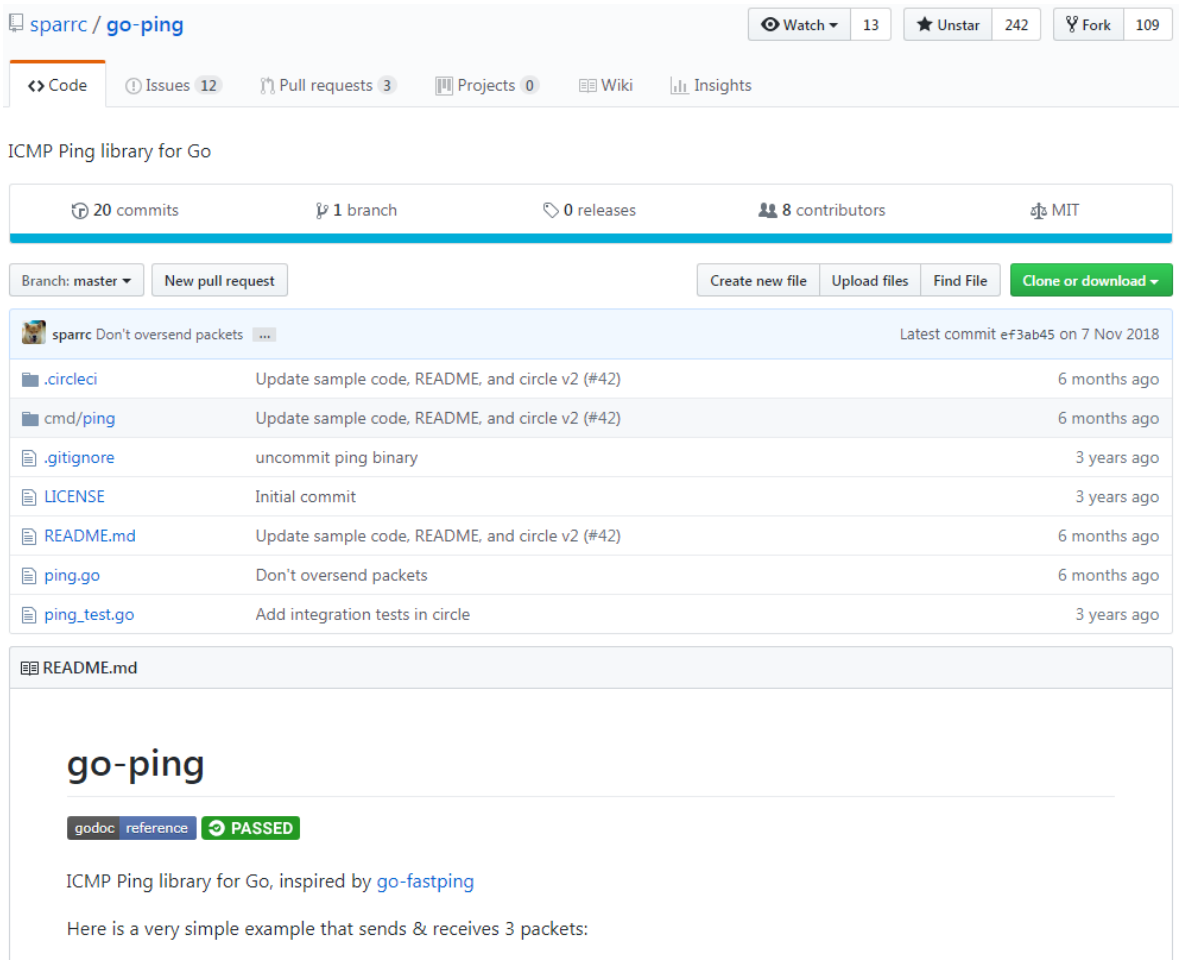
本节介绍一个真实的案例，重点分析产生资源泄露的现象以及排查思路。

应用背景

曾经做过一个产品，不经意间出现了CPU使用率缓慢升高，最后CPU使用率竟然达到了100%，严重影响了业务。经过排查，问题出在Ticker的使用方式上，创建了Ticker，在使用结束后没有释放导致的。

该产品需要监控其他服务器的健康状态，其中很常见的一种做法是心跳检测。简单的说，周期性的ping这些服务器，能在指定时间内收到ack说明与该服务器之间的网络没问题。

当时使用了一个小众的开源组件 `tatsushid/go-fastping` 来做ping。
该组件介绍如下图所示：



问题现象

在做性能测试时，管理了1000台服务器，差不多4天后发现系统越来越慢，查看CPU使用情况，结果发现CPU使用率已经达到100%。

排查性能问题主要使用pprof，关于pprof的使用方法及原理介绍在请参照相关章节。

使用pprof查看CPU使用情况，主要是查看CPU都在忙什么：

```
Showing nodes accounting for 11.33s, 96.26% of 11.77s total
Dropped 94 nodes (cum <= 0.06s)
Showing top 10 nodes out of 29
      flat  flat%   sum%        cum   cum%
     7.80s 66.27% 66.27%      7.80s 66.27%  runtime._ExternalCode
     2.19s 18.61% 84.88%      2.19s 18.61%  runtime.siftdownTimer
     0.51s  4.33% 89.21%      0.51s  4.33%  runtime.futex
     0.39s  3.31% 92.52%      0.39s  3.31%  runtime.chansend
     0.11s  0.93% 93.46%      3.46s 29.40%  runtime.timerproc
     0.09s  0.76% 94.22%      0.09s  0.76%  runtime.usleep
     0.08s  0.68% 94.90%      0.13s  1.10%  runtime.selectgoImpl
     0.07s  0.59% 95.50%      0.09s  0.76%  runtime.lock
     0.05s  0.42% 95.92%      0.45s  3.82%  runtime.selectnbsend
     0.04s  0.34% 96.26%      0.09s  0.76%  time.Now
(pprof)
```

从上图可以看出，CPU主要是被runtime包占用了，其中第二行 `runtime.siftdownTimer` 正是timerproc中的一个动作。

再使用pprof查看函数调用栈，主要看是哪些函数在使用CPU：

```
(pprof) top -cum
Showing nodes accounting for 70ms, 25.93% of 270ms total
Showing top 10 nodes out of 94
      flat  flat%   sum%        cum   cum%
         0     0%     0%      230ms 85.19%  runtime.goexit
         0     0%     0%      140ms 51.85%                /ping.CheckOne
         0     0%     0%       90ms 33.33%                /ping.(*Pinger).Run
         0     0%     0%       90ms 33.33%                /ping.(*Pinger).run
         0     0%     0%       90ms 33.33%                /ping.PingIp
         0     0%     0%       70ms 25.93%                /ping.HeartBeatCheck
         0     0%     0%       70ms 25.93%                /vendor/libs/loglib.(*Syslog).msgSend
         0     0%     0%       70ms 25.93%  net.(*conn).Write
         0     0%     0%       70ms 25.93%  net.(*netFD).Write
      70ms 25.93% 25.93%       70ms 25.93%  syscall.Syscall
(pprof)
```

从上图可以看出，CPU主要是被ping模块占用，其中 `ping.(*Pinger).Run` 正是开源组件的一个接口。

经过pprof分析可以很清晰的指出问题出在go-fastping组件的Run()接口中，而且是与timer相关的。问题定位到这里，解决就很简单了。

此处，可以先总结一下Ticker资源泄露的现象：

- CPU使用率持续升高
- CPU使用率缓慢升高

源码分析

出问题的源码在ping.go的run()方法中。为叙述方便，对代码做了适当简化：

```
func (p *Pinger) run() {
    timeout := time.NewTicker(p.Timeout)    // 创建Ticker timeout
    interval := time.NewTicker(p.Interval)  // 创建Ticker

    for {
        select {
```

```
        case <-p.done:        // 正常退出，未关闭Ticker
            wg.Wait()
            return
        case <-timeout.C:     // 超时退出，未关闭Ticker
            close(p.done)
            wg.Wait()
            return
        case <-interval.C:
            if p.Count > 0 && p.PacketsSent >= p.Count {
                continue
            }
            err = p.sendICMP(conn)
            if err != nil {
                fmt.Println("FATAL: ", err.Error())
            }
        case r := <-recv:
            err := p.processPacket(r)
            if err != nil {
                fmt.Println("FATAL: ", err.Error())
            }
        }
        if p.Count > 0 && p.PacketsRecv >= p.Count {  // 退出，未关闭Ticker
            close(p.done)
            wg.Wait()
            return
        }
    }
}
```

该段代码可以看出，这个函数是有出口的，但在出口处没有关闭Ticker，导致资源泄露。

这个问题已经被修复了，可以看到修复后的局部代码如下：

```
    timeout := time.NewTicker(p.Timeout)
    defer timeout.Stop()  // 使用defer保证Ticker最后被关闭
    interval := time.NewTicker(p.Interval)
    defer interval.Stop() // 使用defer保证Ticker最后被关闭
```

总结

有一种情况使用Ticker不主动关闭也不会造成资源泄露，比如，函数创建Ticker后就不会退出，直到进程结束。这种情况下不会持续的创建Ticker，也就不会造成资源泄露。

但是，不管哪种情况，创建一个Ticker后，紧跟着使用defer语句关闭Ticker总是好的习惯。因为，有可能别人无意间拷贝了你的部分代码，而忽略了关闭Ticker的动作。

> 赠人玫瑰手留余香，如果觉得不错请给个赞~
>
> 本篇文章已归档到GitHub项目，求星~ 点我即达

语法糖

名字由来

语法糖（Syntactic sugar）的概念是由英国计算机科学家 `Peter J. Landin` 提出的，
用于表示编程语言中的某种类型的语法，这些语法不会影响功能，但使用起来却很方便。

语法糖，也称糖语法，这些语法不仅不会影响功能，编译后的结果跟不使用语法糖也一样。

语法糖，有可能让代码编写变得简单，也有可能让代码可读性更高，也有可能让代码出问题。为了避免陷阱才是这个章节的重点。

Go语言语法糖

最常用的语法糖莫过于赋值符 `:=` ，其次，表示函数变参的 `...` 。

接下来，我们会介绍这两种语法糖的用法，更重要的是结合实际的经历跟大家分享其中的陷阱。

简短变量声明

想要声明变量，可以使用关键字 `var` 或者直接使用简短变量声明（ `:=` ）。
后者使用更频繁一些，尤其是在接收函数返回值场景中，你不必使用 `var` 声明一个变量再变量接收函数返回值，使用 `:=` 可以一步到位。

本节我们讨论 `:=` 的一些容易被忽视的规则，以避免一些陷阱。

热身测验

前言

相信你已经大量使用过简短变量声明，比如像下面这样：

```
i := 0
j, k := 1, 2
```

`:=` 用来声明变量并赋值，不管是个人项目、公司项目和开源项目都会大量应用。根据我做过的小范围调查结果看，有多年Go开发经验的工程师也不能很好的回答 `:=` 相关的问题。

在开始讨论 `:=` 前，请试着回答一下这些题目，再根据参考答案对照一下，或许会有令你惊讶的发现。

题目

题目一

问：下面代码输出什么？

```
func fun1() {
    i := 0
    i, j := 1, 2
    fmt.Printf("i = %d, j = %d\n", i, j)
}
```

题目二

问：下面代码为什么不能通过编译？

```
func fun2(i int) {
    i := 0
    fmt.Println(i)
}
```

题目三

问：下面代码输出什么？

```
func fun3() {
    i, j := 0, 0
    if true {
        j, k := 1, 1
        fmt.Printf("j = %d, k = %d\n", j, k)
    }
    fmt.Printf("i = %d, j = %d\n", i, j)
}
```

参考答案

题目一

程序输出如下：

```
i = 1, j = 2
```

再进一步想一下，前一个语句中已经声明了i, 为什么还可以再次声明呢？

题目二

不能通过编译原因是形参已经声明了变量i，使用 `:=` 再次声明是不允许的。

再进一步想一下，编译时会报”no new variable on left side of :=”错误，该怎么理解？

题目三

程序输出如下：

```
j = 1, k = 1
i = 0, j = 0
```

这里要注意的是，block `if` 中声明的j，与上面的j属于不同的作用域。

使用规则

前言

虽然简短变量声明这个语法糖用起来很方便，但有时也会给你一个意外也可能带你掉入陷阱。

我曾因滥用这个 `:=` 语法糖，发生过一次故障，所以才认真研究了一下它的原理和规则，大家可以作为参考。

规则

规则一： 多变量赋值可能会重新声明

我们知道使用 `:=` 一次可以声明多个变量，像下面这样：

```
field1, offset := nextField(str, 0)
```

上面代码定义了两个变量，并用函数返回值进行赋值。

如果这两个变量中的一个再次出现在 `:=` 左侧就会重新声明。像下面这样：

```
field1, offset := nextField(str, 0)
field2, offset := nextField(str, offset)
```

offset被重新声明。

重新声明并没有什么问题，它并没有引入新的变量，只是把变量的值改变了，但要明白，这是Go提供的一个语法糖。

- 当 `:=` 左侧存在新变量时（如field2），那么已声明的变量（如offset）则会被重新声明，不会有其他额外副作用。
- 当 `:=` 左侧没有新变量是不允许的，编译会提示 `no new variable on left side of :=` 。

我们所说的重新声明不会引入问题要满足一个前提，变量声明要在同一个作用域中出现。如果出现在不同的作用域，那很可能就创建了新的同名变量，同一函数不同作用域的同名变量往往不是预期做法，很容易引入缺陷。关于作用域的这个问题，我们在本节后面介绍。

规则二：不能用于函数外部

简短变量场景只能用于函数中，使用 `:=` 来声明和初始化全局变量是行不通的。

比如，像下面这样：

```
package sugar
import fmt

rule := "Short variable declarations" // syntax error: non-declaration statement outside function body
```

这里的编译错误提示 `syntax error: non-declaration statement outside function body` ，表示非声明语句不能出现在函数外部。可以理解成 `:=` 实际上会拆分成两个语句，即声明和赋值。赋值语句不能出现在函数外部的。

变量作用域问题

几乎所有的工程师都了解变量作用域，但是由于 `:=` 使用过于频繁的话，还是有可能掉进陷阱里。

下面代码源自真实项目，但为了描述方便，也为了避免信息安全风险，简化如下：

```
func Redeclare() {
    field, err:= nextField()   // 1号err

    if field == 1{
        field, err:= nextField()     //  2号err
        newField, err := nextField() //  3号err
        ...
    }
    ...
}
```

注意上面声明的三个err变量。
2号err与1号err不属于同一个作用域， `:=` 声明了新的变量，所以2号err与1号err属于两个变量。
2号err与3号err属于同一个作用域， `:=` 重新声明了err但没创建新的变量，所以2号err与3号err是同一个变量。

如果误把2号err与1号err混淆，就很容易产生意想不到的错误。

> 赠人玫瑰手留余香，如果觉得不错请给个赞~
>
> 本篇文章已归档到GitHub项目，求星~ 点我即达

可变参函数

前言

可变参函数是指函数的某个参数可有可无，即这个参数个数可以是0个或多个。
声明可变参数函数的方式是在参数类型前加上 `...` 前缀。

比如 `fmt` 包中的 `Println` ：

```
func Println(a ...interface{})
```

本节我们会总结一下其使用方法，顺便了解一下其原理，以避免在使用过程中进入误区。

函数特征

我们先写一个可变参函数：

```
func Greeting(prefix string, who ...string) {
    if who == nil {
        fmt.Printf("Nobody to say hi.")
        return
    }

    for _, people := range who{
        fmt.Printf("%s %s\n", prefix, people)
    }
}
```

`Greeting` 函数负责给指定的人打招呼，其参数 `who` 为可变参数。

这个函数几乎把可变参函数的特征全部表现出来了：

- 可变参数必须在函数参数列表的尾部，即最后一个（如放前面会引起编译时歧义）；
- 可变参数在函数内部是作为切片来解析的；
- 可变参数可以不填，不填时函数内部当成 `nil` 切片处理；
- 可变参数必须是相同类型的（如果需要是不同类型的可以定义为interface{}类型）；

使用举例

我们使用 `testing` 包中的Example函数来说明上面 `Greeting` 函数（函数位于sugar包中）用法。

不传值

调用可变参函数时，可变参部分是可以不传值的，例如：

```
func ExampleGreetingWithoutParameter() {
    sugar.Greeting("nobody")
    // OutPut:
    // Nobody to say hi.
}
```

这里没有传递第二个参数。可变参数不传递的话，默认为nil。

传递多个参数

调用可变参函数时，可变参数部分可以传递多个值，例如：

```
func ExampleGreetingWithParameter() {
    sugar.Greeting("hello:", "Joe", "Anna", "Eileen")
    // OutPut:
    // hello: Joe
    // hello: Anna
    // hello: Eileen
}
```

可变参数可以有多个。多个参数将会生成一个切片传入，函数内部按照切片来处理。

传递切片

调用可变参函数时，可变参数部分可以直接传递一个切片。参数部分需要使用 `slice...` 来表示切片。例如：

```
func ExampleGreetingWithSlice() {
    guest := []string{"Joe", "Anna", "Eileen"}
    sugar.Greeting("hello:", guest...)
    // OutPut:
    // hello: Joe
    // hello: Anna
    // hello: Eileen
}
```

此时需要注意的一点是，切片传入时不会生成新的切片，也就是说函数内部使用的切片与传入的切片共享相同的存储空间。说得再直白一点就是，如果函数内部修改了切片，可能会影响外部调用的函数。

总结

- 可变参数必须要位于函数列表尾部；
- 可变参数是被当作切片来处理的；
- 函数调用时，可变参数可以不填；
- 函数调用时，可变参数可以填入切片；

> 赠人玫瑰手留余香，如果觉得不错请给个赞~
>
> 本篇文章已归档到GitHub项目，求星~ 点我即达

GO语言版本管理

GO语言的安装和卸载是个基本功，因为GO语言还在不断地演进，为了使用新版本你就需要先把旧版本删除再安装新版本。

另外，看起来非常简单的安装和卸载过程，如果你对 `GOROOT` 和 `GOPATH` 理解不到位，也会在使用过程中遇到麻烦。
再进一步，业界有许多GO语言的版本管理工具，我们借助这些工具可以实现同时安装多个GO语言版本，要理解这些工具的实现原理，
也需要了解GO语言运行机制，比如 `import` 搜索路径，而手动安装GO语言版本可以加深这些认识。

本章我们先介绍手动安装和卸载过程，再介绍一些基础概念，比如 `GOROOT` 、 `GOPATH` 等，最后我们会介绍常见的GO语言版本工具。

GO语言安装

与大多数开源软件一样，Go安装包也分为二进制包、源码包。二进制包为基于源码编译出各个组件，并把这些组件打包在一起供人下载和安装，源码包为 Golang语言源码，供人下载、编译后自行安装。

接下来我们以安装二进制包（go1.12.7.linux-amd64.tar.gz）为例进行说明安装过程。
linux下可以使用 `wget https://dl.google.com/go/go1.12.7.linux-amd64.tar.gz` 命令下载。

Go语言安装比较简单，大体上分为三个步骤：

- 安装可执行命令
- 设置PATH环境变量
- 设置GOPATH环境变量

1. 安装可执行命令

二进制安装包中包含二进制、文档、标准库等内容，我们需要将该二进制完整的解压出来。

一般使用 `/usr/local/go` 来存放解压出来的文件，这个目录也就是 `GOROOT` ，即GO的根目录。
下接使用 `tar` 命令将安装包解压到指定目录即可：

```
tar -C /usr/local -xzf go1.12.7.linux-amd64.tar.gz
```

2. 设置PATH环境变量

Go的二进制可执行文件存在于 `$GOROOT/bin` 目录，需要将该目录加入到 `PATH` 环境变量中。

比如，把下面语句放入 `/etc/profile` 文件中。

```
export PATH=$PATH:/usr/local/go/bin
```

3. 设置GOPATH环境变量

Linux下，自Go 1.8版本起，默认把 `$HOME/go` 作为 `GOPATH` 目录，可以根据需要设置自己的 `GOPATH` 目录。

如需要设置不同的GOPATH目录，可以将其放入 `~/.bash_profile` 中。

```
export GOPATH=$HOME/mygopath
```

即便使用 `GOPATH` 默认目录，推荐也把它加入到环境变量中，这可以让bash识别 `GOPATH` 变量。

这里需要注意的是， `GOPATH` 值不可以与 `GOROOT` 相同，因为如果用户的项目与标准库重名会导致编译时产生歧义。

4. 测试安装

安装完成后，可以写个小程序验证一下，验证前建议重新登录，以便让环境变量生效。

创建 `$GOPATH/src/hello/hello.go` 文件：

```
package main

import "fmt"

func main() {
    fmt.Printf("hello, world\n")
}
```

接着进入 `$GOPATH/src/hello` 编译并执行：

```
# cd $GOPATH/src/hello
# go build
# hello
hello, world
```

程序能正常输出 `hello, world` 就表示安装完成了。

GO语言卸载

当需要升级新的Go语言版本时，你需要先把旧版本删除。Go语言版本升级过程实际上是 `删除旧版本` + `安装新版本` 。

删除Go语言版本是安装新版本的逆过程，即把新版本安装时创建的目录、环境变量删除。

1. 删除Go安装目录

通过 `go env` 命令查询安装目录，安装目录即 `GOROOT` 环境变量所指示的目录，如下所示：

```
# go env
GOPATH="/root/go"
GOROOT="/usr/local/go"
```

`go env` 命令会输出很多Go语言相关的环境变量，上面只保留了最关键的两个 `GOROOT` 和 `GOPATH` 。

接下来使用 `rm` 命令删除 `GOROOT` 指向的目录即可，比如 `# rm -rf /usr/local/go` 。

2. 删除残留的可执行文件

Go程序在运行过程中会在 `GOPATH/bin` 目录下生成可执行文件，为了安全起见，也需要删除。

同样，使用 `rm` 命令删除即可，比如 `# rm -rf /root/go/bin` 。

注：如果GOPATH包含多个目录，需要删除每个目录下的bin目录。

3. 删除环境变量

将环境变量 `GOPATH` 删除，该环境变量一般是前一次安装Go时人为设置的。

环境变量一般存在于以下几个文件中：

- /etc/profile
- /etc/bashrc
- ~/.bash_profile
- ~/.profile
- ~/.bashrc

做完以上步骤，旧版本就被彻底的删除了。

GO依赖管理

Go语言依赖管理经历了三个重要的阶段：

- GOPATH；
- vendor；
- Go Module；

早期Go语言单纯使用GOPATH管理依赖，但GOPATH不方便管理依赖的多个版本，后来增加了vendor，允许把项目依赖连同项目源码一同管理。
自从Go 1.11版本引入了全新的依赖管理工具Go module，直到Go 1.14版本 Go module才走向成熟。

Go官方依赖管理演进过程中还有为数众多的第三方管理工具，比如 `Glide` 、 `Govendor` 等等，但随着Go module的推出，
这些工具终将逐步退出历史舞台，本章节不再涵盖此部分内容。

从GOPATH到vendor，再到Go module是个不断演进的过程，了解每种依赖管理的痛点可以更好的理解下一代依赖管理的设计初衷。
本章先从基础的GOPATH讲起，接着介绍vendor，最后再介绍Go module。

GOPATH

根据我个人的经验，那些对GOPATH感到困惑的同学，往往归属于下面两类：

- 没有亲自安装过Go语言；
- 安装过，但没理解安装细节；

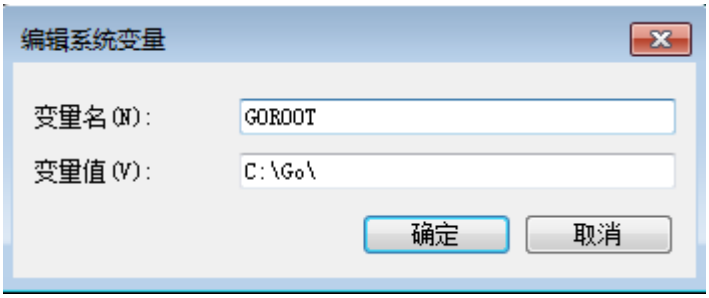
其实，自己亲自动手安装一遍Go语言，然后运行一个Hello World程序，就基本上能理解GOPATH。
本节主要介绍GOPATH以及与其密切相关的GOROOT，关于安装相关的详细内容不再赘述。

GOROOT 是什么

通常我们说安装Go语言，实际上安装的是Go编译器和Go标准库，二者位于同一个安装包中。

假如你在Windows上使用Installer安装的话，它们将会被默认安装到 `c:\Go` 目录下，该目录即GOROOT目录，里面保存了开发GO程序所需要的所有组件，比如编译器、标准库和文档等等。

同时安装程序还会自动帮你设置GOROOT环境变量，如下图所示：



另外，安装程序还会把 `c:\Go\bin` 目录添加到系统的 `PATH` 环境变量中，如下图所示：

该目录主要是GO语言开发包中提供的二进程可执行程序。

所以，GOROOT实际上是指示GO语言安装目录的环境变量，属于GO语言顶级目录。

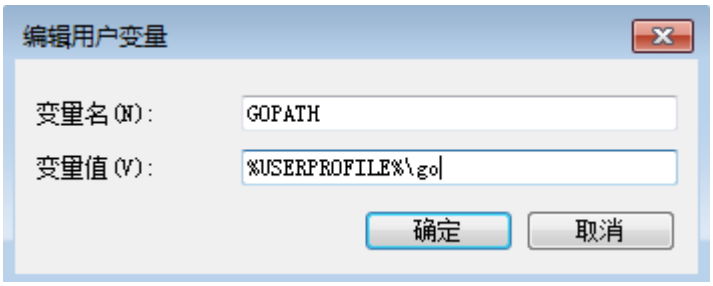
GOPATH 是什么

安装完Go语言，接下来就要写自己的Hello World项目了。实际上Go语言项目是由一个或多个package组成的，这些package按照来源分为以下几种：

- 标准库
- 第三方库
- 项目私有库

其中标准库的package全部位于GOROOT环境变量指示的目录中，而第三方库和项目私有库都位于GOPATH环境变量所指示的目录中。

实际上，安装GO语言时，安装程序会设置一个默认的GOPATH环境变量，如下所示：



与GOROOT不同的是，GOPATH环境变量位于用户域，因为每个用户都可以创建自己的工作空间而互不干扰。
用户的项目需要位于 `GOPATH` 下的 `src/` 目录中。

所以GOPATH指示用户工作空间目录的环境变量，它属于用户域范畴的。

依赖查找

当某个package需要引用其他包时，编译器就会依次从 `GOROOT/src/` 和 `GOPATH/src/` 中去查找，如果某个包从GOROOT下找到的话，就不再到GOPATH目录下查找，所以如果项目中开发的包名与标准库相同的话，将会被自动忽略。

GOPATH的缺点

GOPATH的优点是足够简单，但它不能很好的满足实际项目的工程需求。

比如，你有两个项目A和B，他们都引用某个第三方库T，但这两个项目使用了不同的T版本，即：

- 项目A 使用T v1.0
- 项目B 使用T v2.0

由于编译器依赖查找固定从GOPATH/src下查找 `GOPATH/src/T` ，所以，无法在同一个GOPATH目录下保存第三方库T的两个版本。所以项目A、B无法共享同一个GOPATH，需要各自维护一个，这给广大软件工程师带来了极大的困扰。

针对GOPATH的缺点，GO语言社区提供了Vendor机制，从此依赖管理进入第二个阶段：将项目的依赖包私有化。

> 赠人玫瑰手留余香，如果觉得不错请给个赞~
>
> 本篇文章已归档到GitHub项目，求星~ 点我即达

vendor

前面我们介绍了使用GOPATH时的痛点：多项目无法共享同一个GOPATH。
其实本节介绍的vendor机制也没有彻底解决这个痛点，但是它提供了一个机制让项目的依赖隔离而不互相干扰。

自Go 1.6版本起，vendor机制正式启用，它允许把项目的依赖全部放到一个位于本项目的vendor目录中，这个vendor目录可以简单理解成私有的GOPATH目录。即编译时，优先从vendor中寻找依赖包，如果vendor中找不到再到GOPATH中寻找。

vendor目录位置

一个项目可以有多个vendor目录，分别位于不同的目录级别，但建议每个项目只在根目录放置一个vendor目录。

假如你有一个 `github.com/constabulary/example-gsftp` 项目，项目目录结构如下：

```
$GOPATH
|    src/
|    |    github.com/constabulary/example-gsftp/
|    |    |    cmd/
|    |    |    |    gsftp/
|    |    |    |    |    main.go
```

其中 `main.go` 中依赖如下几个包：

```
import (
    "golang.org/x/crypto/ssh"
    "github.com/pkg/sftp"
)
```

在没有使用vendor目录时，若想编译这个项目，那么GOPATH目录结构应该是如下所示：

```
$GOPATH
|    src/
|    |    github.com/constabulary/example-gsftp/
|    |    golang.org/x/crypto/ssh
|    |    github.com/pkg/sftp
```

即，所有依赖的包，都位于 `$GOPATH/src` 下。

为了把所使用到的 `golang.org/x/crypto/ssh` 和 `github.com/pkg/sftp` 版本固化下来，那么可以使用vendor机制。

在项目 `github.com/constabulary/example-gsftp` 根目录下，创建一个vendor目录，并把 `golang.org/x/crypto/ssh` 和 `github.com/pkg/sftp` 存放到此处，让其成为项目的一部分。如下所示：

```
$GOPATH
|    src/
|    |    github.com/constabulary/example-gsftp/
|    |    |    cmd/
|    |    |    |    gsftp/
|    |    |    |    |    main.go
|    |    |    vendor/
|    |    |    |    github.com/pkg/sftp/
|    |    |    |    golang.org/x/crypto/ssh/
```

使用vendor的好处是在项目 `github.com/constabulary/example-gsftp` 发布时，把其所依赖的软件一并发布，编译时不会受到GOPATH目录的影响，即便GOPATH下也有一个同名但不同版本的依赖包。

搜索顺序

上面的例子中，在编译main.go时，编译器搜索依赖包顺序为：

1. 从 `github.com/constabulary/example-gsftp/cmd/gsftp/` 下寻找vendor目录，没有找到，继续从上层查找；
2. 从 `github.com/constabulary/example-gsftp/cmd/` 下寻找vendor目录，没有找到，继续从上层查找；
3. 从 `github.com/constabulary/example-gsftp/` 下寻找vendor目录，从vendor目录中查找依赖包，结束；

如果 `github.com/constabulary/example-gsftp/` 下的vendor目录中没有依赖包，则返回到GOPATH目录继续查找，这就是前面介绍的GOPATH机制了。

从上面的搜索顺序可以看出，实际上vendor目录可以存在于项目的任意目录的。但非常不推荐这么做，因为如果vendor目录过于分散，很可能会出现同一个依赖包，在项目的多个vendor中出现多次，这样依赖包会多次编译进二进制文件，从而造成二进制大小急剧变大。同时，也很可能出现一个项目中使用同一个依赖包的多个版本的情况，这种情况往往应该避免。

vendor存在的问题

vendor很好的解决了多项目间的隔离问题，但是位于vendor中的依赖包无法指定版本，某个依赖包，在把它放入vendor的那刻起，它就固定在当时版本，项目的使用者很难识别出你所使用的依赖版本。

比起这个，更严重的问题是上面提到的二进制急剧扩大问题，比如你依赖某个开源包A和B，但A中也有一个vendor目录，其中也放了B，那么你的项目中将会出现两个开源包B。再进一步，如果这两个开源包B版本不一致呢？如果二者不兼容，那后果将是灾难性的。

但是，不得不说，vendor能够解决绝大部分项目中的问题，如果你项目在使用vendor，也绝对没有问题。一直到Go 1.11版本，官方社区推出了Modules机制，从此Go的版本管理走进第三个时代。

> 赠人玫瑰手留余香，如果觉得不错请给个赞~
>
> 本篇文章已归档到GitHub项目，求星~ 点我即达

module

在Go v1.11版本中，Module特性被首次引入，这标志着Go的依赖管理开始进入第三个阶段。

Go Module 相比GOPATH和vendor而言功能强大得多，它基本上完全解决了GOPATH和vendor时代遗留的问题。
我们知道，GOPATH时代最大的困扰是无法让多个项目共享同一个pakage的不同版本，在vendor时代，通过把每个项目依赖的package放到vendor
中可以解决这个困扰，但是使用vendor的问题是无法很好的管理依赖的package，比如升级package。

虽然Go Module能够解决GOPATH和vendor时代遗留的问题，但需要注意的是Go Module不是GOPATH 和 vendor的演进，理解这个对于接下来
正确理解Go Module非常重要。

Go Module更像是一种全新的依赖管理方案，它涉及一系列的特性，但究其核心，它主要解决两个重要的问题：

- 准确的记录项目依赖；
- 可重复的构建；

准确的记录项目依赖，是指你的项目依赖哪些package、以及package的版本可以非常精确。比如你的项目依赖 `github.com/prometheus/client_golang` ，
且必须是 `v1.0.0` 版本，那么你可以通过Go Module指定（具体指定方法后面会介绍），任何人在任何环境下编译你的项目，
都必须要使用 `github.com/prometheus/client_golang` 的 `v1.0.0` 版本。

可重复的构建是指，项目无论在谁的环境中（同平台）构建，其产物都是相同的。回想一下GOPATH时代，虽然大家拥有同一个项目的代码，但由于各自
的GOPATH中 `github.com/prometheus/client_golang` 版本不一样，虽然项目可以构建，但构建出的可执行文件很可能是不同的。
可重复构建至关重要，避免出现“我这运行没问题，肯定是你环境问题”等类似问题出现。

一旦项目的依赖被准确记录了，就很容易做到重复构建。

事实上，Go Module是一个非常复杂的特性，一下子全盘托出其特性，往往会让人产生疑惑，所以接下来的章节，我们希望逐个介绍其特性，
并且，尽可以附以实例，希望大家也跟我一样手动实践一下，以加深认识。

基础概念

在开始学习module机制之前，我们有必要初步了解一下其涉及的基本概念，比如到底什么是module等。

Module的定义

首先，module是个新鲜又熟悉的概念。新鲜是指在以往GOPATH和vendor时代都没有提及，它是个新的词汇。
为什么说熟悉呢？因为它不是新的事物，事实上我们经常接触，这次只是官方给了一个统一的称呼而矣。

拿开源项目 `https://github.com/blang/semver` 举例，这个项目是一个语义化版本处理库，当你的项目需要时可以在你的项目中import，比如：

```
"github.com/blang/semver"
```

`https://github.com/blang/semver` 项目中可以包含一个或多个package，不管有多少package，这些package都随项目一起发布，即当我们说 `github.com/blang/semver` 某个版本时，说的是整个项目，而不是具体的package。此时项目 `https://github.com/blang/semver` 就是一个module。

官方给module的定义是：`A module is a collection of related Go packages that are versioned together as a single unit.` ，定义非常晰，一组package的集合，一起被标记版本，即是一个module。

通常而言，一个仓库包含一个module（虽然也可以包含多个，但不推荐），所以仓库、module和package的关系如下：

- 一个仓库包含一个或多个Go module；
- 每个Go module包含一个或多个Go package；
- 每个package包含一个或多个Go源文件；

此外，一个module的版本号规则必须遵循语义化规范（https://semver.org/），版本号必须使用格式`v(major).(minor).(patch)`，比如`v0.1.0`、`v1.2.3` 或 `v1.5.0-rc.1` 。

语义化版本规范

语义化版本（Semantic Versioning）已成为事实上的标准，几乎知名的开源项目都遵循该规范，更详细的信息请前往 https://semver.org/ 查看，在此只提炼一些要点，以便于后续的阅读。

版本格式 `v(major).(minor).(patch)` 中major指的是大版本，minor指小版本，patch指补丁版本。

- major: 当发生不兼容的改动时才可以增加major版本；比如 `v2.x.y` 与 `v1.x.y` 是不兼容的；
- minor: 当有新增特性时才可以增加该版本，比如 `v1.17.0` 是在 `v1.16.0` 基础上加了新的特性，同时兼容 `v1.16.0` ；
- patch: 当有bug修复时才可以 增加该版本，比如 `v1.17.1` 修复了 `v1.17.0` 上的bug，没有新特性增加；

语义化版本规范的好处是，用户通过版本号就能了解版本信息。

除了上面介绍的基础概念以外，还有描述依赖的 `go.mod` 和记录module的checksum的 `go.sum` 等内容，
这部分内容比较多且比较复杂，在后面的章节中我们通过实际的例子逐步展开介绍，否则提前暴露过多的细节容易造成困惑而徒生挫败感。

> 赠人玫瑰手留余香，如果觉得不错请给个赞~
>
> 本篇文章已归档到GitHub项目，求星~ 点我即达

快速实践

本节我们根据实际的例子来演示go module的功能。在开始之前，我希望读者思考如下问题，并带着这些问题阅读下面的内容或者亲自动手实践以便加深认识。

Go module到底是做什么的？

我们在前面的章节已介绍过，但还是想强调一下，Go module实际上只是精准的记录项目的依赖情况，包括每个依赖的精确版本号，仅此而矣。

那么，为什么需要记录这些依赖情况，或者记录这些依赖有什么好处呢？

试想一下，在编译某个项目时，第三方包的版本往往是可以替换的，如果不能精确的控制所使用的第三方包的版本，最终构建出的可执行文件从本质上是不同的，这会给问题诊断带来极大的困扰。

接下来，我们从一个Hello World项目开始，逐步介绍如何初始化module、如何记录依赖的版本信息。
项目托管在GitHub `https://github.com/renhongcai/gomodule` 中，并使用版本号区别使用go module的阶段。

- v1.0.0 未引用任何第三方包，也未使用go module
- v1.1.0 未引用任何第三方包，已开始使用go module，但没有任何外部依赖
- v1.2.0 引用了第三方包，并更新了项目依赖

需要注意的是，下面的例子统一使用go 1.13版本，如果你使用go 1.11 或者go 1.12，运行效果可能略有不同。
本文最后部分我们尽量尝试记录一些版本间的差异，以供参考。

Hello World

在v1.0.0版本时，项目只包含一个main.go文件，只是一个简单的字符串打印：

```
package main

import "fmt"

func main() {
    fmt.Println("Hello World")
}
```

此时，项目还没有引用任何第三方包，也未使用go module。

初始化module

一个项目若要使用Go module，那么其本身需要先成为一个module，也即需要一个module名字。

在Go module机制下，项目的module名字以及其依赖信息记录在一个名为 `go.mod` 的文件中，该文件可以手动创建，也可以使用 `go mod init` 命令自动生成。推荐自动生成的方法，如下所示：

```
[root@ecs-d8b6 gomodule]# go mod init github.com/renhongcai/gomodule
go: creating new go.mod: module github.com/renhongcai/gomodule
```

完整的 `go mod init` 命令格式为 `go mod init [module]` ：其中 `[module]` 为module名字，如果不填， `go mod init` 会尝试从版本控制系统或import的注释中猜测一个。这里推荐指定明确的module名字，因为猜测有时需要一些额外的条件，比如 Go 1.13版本，只有项目位于GOPATH中才可以正确运行，而 Go 1.11版本则没有此要求。

上面的命令会自动创建一个 `go.mod` 文件，其中包括module名字，以及我们所使用的Go 版本：

```
[root@ecs-d8b6 gomodule]# cat go.mod
module github.com/renhongcai/gomodule

go 1.13
```

`go.mod` 文件中的版本号 `go 1.13` 是在Go 1.12引入的，意思是开发此项目的Go语言版本，并不是编译该项目所限制的Go语言版本，但是如果项目中使用了Go 1.13的新特性，而你使用Go 1.11编译的话，编译失败时，编译器会提示你版本不匹配。

由于我们的项目还没有使用任何第三方包，所以 `go.mod` 中并没有记录依赖包的任何信息。我们把自动生成的 `go.mod` 提交，然后我们尝试引用一个第三方包。

管理依赖

现在我们准备引用一个第三方包 `github.com/google/uuid` 来生成一个UUID，这样就会产生一个依赖，代码如下：

```
package main

import (
    "fmt"

    "github.com/google/uuid"
)

func main() {
    id := uuid.New().String()
    fmt.Println("UUID: ", id)
}
```

在开始编译以前，我们先使用 `go get` 来分析依赖情况，并会自动下载依赖：

```
[root@ecs-d8b6 gomodule]# go get
go: finding github.com/google/uuid v1.1.1
go: downloading github.com/google/uuid v1.1.1
go: extracting github.com/google/uuid v1.1.1
```

从输出内容来看， `go get` 帮我们定位到可以使用 `github.com/google/uuid` 的v1.1.1版本，并下载再解压它们。

注意： `go get` 总是获取依赖的最新版本，如果 `github.com/google/uuid` 发布了新的版本，输出的版本信息会相应的变化。关于Go Module机制中版本选择我们将在后续的章节详细介绍。

`go get` 命令会自动修改 `go.mod` 文件：

```
[root@ecs-d8b6 gomodule]# cat go.mod
module github.com/renhongcai/gomodule

go 1.13

require github.com/google/uuid v1.1.1
```

可以看到，现在 `go.mod` 中增加了 `require github.com/google/uuid v1.1.1` 内容，表示当前项目依赖 `github.com/google/uuid` 的 `v1.1.1` 版本，这就是我们所说的 `go.mod` 记录的依赖信息。

由于这是当前项目第一次引用外部依赖， `go get` 命令还会生成一个 `go.sum` 文件，记录依赖包的hash值：

```
[root@ecs-d8b6 gomodule]# cat go.sum
github.com/google/uuid v1.1.1 h1:Gkbcsh/GbpXz7lPftLA3P6TYMwjCLYm83jiFQZF/3gY=
github.com/google/uuid v1.1.1/go.mod h1:TIyPZe4MgqvfeYDBFedMoGGpEw/LqOeaOT+nhxU+yHo=
```

该文件通过记录每个依赖包的hash值，来确保依赖包没有被篡改。关于此部分内容我们在此暂不展开介绍，留待后面的章节详细介绍。

经 `go get` 修改的 `go.mod` 和创建的 `go.sum` 都需要提交到代码库，这样别人获取到项目代码，编译时就会使用项目所要求的依赖版本。

至此，项目已经有一个依赖包，并且可以编译执行了，每次运行都会生成一个独一无二的UUID：

```
[root@ecs-d8b6 gomodule]# go run main.go
UUID:  20047f5a-1a2a-4c00-bfcd-66af6c67bdfb
```

注：如果你没有使用 `go get` 在执行之前下载依赖，而是直接使用 `go build main.go` 运行项目的话，依赖包也会被自动下载。但是在 `v1.13.4` 中有个bug，即此时生成的 `go.mod` 中显示的依赖信息则会是 `require github.com/google/uuid v1.1.1 // indirect` ，注意行末的 `indirect` 表示间接依赖，这明显是错误的，因为我们直接 `import` 的。

版本间差异

由于Go module在Go v1.11初次引入，历经Go v1.12、v1.13的发展，其实现细节上已有了一些变化，按照之前的规划Go module将会在v1.14定型，推荐尽可能使用最新版本，否则可能会产生一些困扰。

比如，在v1.11中使用 `go mod init` 初始化项目时，不填写 `module` 名称是没有问题，但在v1.13中，如果项目不在GOPATH目录中，则必须填写 `module` 名称。

后记

本节，我们通过简单的示例介绍了如何初始化module以及如何添加新的依赖，还有更多的内容没有展开。比如 `go.mod` 文件中除了 `module` 和 `require` 指令外还有 `replace` 和 `exclude` 指令，再比如 `go get` 下载的依赖包如何存储的，以及 `go.sum` 如何保证依赖包未被篡改的，这些内容我们留待后面的章节一一介绍。

> 赠人玫瑰手留余香，如果觉得不错请给个赞~
>
> 本篇文章已归档到GitHub项目，求星~ 点我即达

replace指令

`go.mod` 文件中通过 `指令` 声明module信息，用于控制命令行工具进行版本选择。一共有四个指令可供使用：

- module： 声明module名称；
- require： 声明依赖以及其版本号；
- replace： 替换require中声明的依赖，使用另外的依赖及其版本号；
- exclude： 禁用指定的依赖；

其中 `module` 和 `require` 我们前面已介绍过， `module` 用于指定module的名字，如 `module github.com/renhongcai/gomodule` ，那么其他项目引用该module时其import路径需要指定 `github.com/renhongcai/gomodule` 。 `require` 用于指定依赖，如 `require github.com/google/uuid v1.1.1` ，该指令相当于告诉 `go build` 使用 `github.com/google/uuid` 的 `v1.1.1` 版本进行编译。

本节开始介绍 `replace` 的用法，包括其工作机制和常见的使用场景，下一节再对 `exclude` 展开介绍。

replace 工作机制

顾名思义， `replace` 指替换，它指示编译工具替换 `require` 指定中出现的包，比如，我们在 `require` 中指定的依赖如下：

```
module github.com/renhongcai/gomodule

go 1.13

require github.com/google/uuid v1.1.1
```

此时，我们可以使用 `go list -m all` 命令查看最终选定的版本：

```
[root@ecs-d8b6 gomodule]# go list -m all
github.com/renhongcai/gomodule
github.com/google/uuid v1.1.1
```

毫无意外，最终选定的uuid版本正是我们在require中指定的版本 `v1.1.1` 。

如果我们想使用uuid的v1.1.0版本进行构建，可以修改require指定，还可以使用replace来指定。
需要说明的是，正常情况下不需要使用replace来修改版本，最直接的办法是修改require即可，虽然replace也能够做到，但这不是replace的一般使用场景。
下面我们先通过一个简单的例子来说明replace的功能，随即介绍几种常见的使用场景。

比如，我们修改 `go.mod` ，添加replace指令：

```
[root@ecs-d8b6 gomodule]# cat go.mod
module github.com/renhongcai/gomodule

go 1.13

require github.com/google/uuid v1.1.1

replace github.com/google/uuid v1.1.1 => github.com/google/uuid v1.1.0
```

`replace github.com/google/uuid v1.1.1 => github.com/google/uuid v1.1.0` 指定表示替换uuid v1.1.1版本为v1.1.0，此时再次使用 `go list -m all` 命令查看最终选定的版本：

```
[root@ecs-d8b6 gomodule]# go list -m all
github.com/renhongcai/gomodule
github.com/google/uuid v1.1.1 => github.com/google/uuid v1.1.0
```

可以看到其最终选择的uuid版本为 v1.1.0。如果你本地没有v1.1.0版本，你或许还会看到一条 `go: finding github.com/google/uuid v1.1.0` 信息，它表示在下载uuid v1.1.0包，也从侧面证明最终选择的版本为v1.1.0。

到此，我们可以看出 `replace` 的作用了，它用于替换 `require` 中出现的包，它正常工作还需要满足两个条件：

第一， `replace` 仅在当前module为 `main module` 时有效，比如我们当前在编译 `github.com/renhongcai/gomodule` ，此时就是 `main module` ，如果其他项目引用了 `github.com/renhongcai/gomodule` ，那么其他项目编译时， `replace` 就会被自动忽略。

第二， `replace` 指定中 `=>` 前面的包及其版本号必须出现在 `require` 中才有效，否则指令无效，也会被忽略。比如，上面的例子中，我们指定 `replace github.com/google/uuid => github.com/google/uuid v1.1.0` ，或者指定 `replace github.com/google/uuid v1.0.9 => github.com/google/uuid v1.1.0` ，二者均都无效。

replace 使用场景

前面的例子中，我们使用 `replace` 替换 `require` 中的依赖，在实际项目中 `replace` 在项目中经常被使用，其中不乏一些精彩的用法。
但不管应用在哪种场景，其本质都一样，都是替换 `require` 中的依赖。

替换无法下载的包

由于中国大陆网络问题，有些包无法顺利下载，比如 `golang.org` 组织下的包，值得庆幸的是这些包在GitHub都有镜像，此时
就可以使用GitHub上的包来替换。

比如，项目中使用了 `golang.org/x/text` 包：

```
package main

import (
    "fmt"

    "github.com/google/uuid"
    "golang.org/x/text/language"
    "golang.org/x/text/message"
)

func main() {
    id := uuid.New().String()
    fmt.Println("UUID: ", id)

    p := message.NewPrinter(language.BritishEnglish)
    p.Printf("Number format: %v.\n", 1500)

    p = message.NewPrinter(language.Greek)
    p.Printf("Number format: %v.\n", 1500)
}
```

上面的简单例子，使用两种语言 `language.BritishEnglish` 和 `language.Greek` 分别打印数字 `1500` ，来查看不同语言对数字格式的处理，一个是 `1,500` ，另一个是 `1.500` 。此时就会分别引入 `"golang.org/x/text/language"` 和 `"golang.org/x/text/message"` 。

执行 `go get` 或 `go build` 命令时会就再次分析依赖情况，并更新 `go.mod` 文件。网络正常情况下， `go.mod` 文件将会变成下面的内容：

```
module github.com/renhongcai/gomodule

go 1.13

require (
    github.com/google/uuid v1.1.1
    golang.org/x/text v0.3.2
)

replace github.com/google/uuid v1.1.1 => github.com/google/uuid v1.1.0
```

我们看到，依赖 `golang.org/x/text` 被添加到了require中。（多条require语句会自动使用 `()` 合并）。此外，我们没有刻意指定 `golang.org/x/text` 的版本号，Go命令行工具根据默认的版本计算规则使用了 v0.3.2版本，此处我们暂不关心具体的版本号。

没有合适的网络代理情况下， `golang.org/x/text` 很可能无法下载。那么此时，就可以使用 `replace` 来让我们的项目使用GitHub上相应的镜像包。我们可以添加一条新的 `replace` 条目，如下所示：

```
replace (
    github.com/google/uuid v1.1.1 => github.com/google/uuid v1.1.0
    golang.org/x/text v0.3.2 => github.com/golang/text v0.3.2
)
```

此时，项目编译时就会从GitHub下载包。我们源代码中import路径 `golang.org/x/text/xxx` 不需要改变。

也许有读者会问，是否可以将import路径由 `golang.org/x/text/xxx` 改成 `github.com/golang/text/xxx` ？这样一来，就不需要使用replace来替换包了。

遗憾的是，不可以。因为 `github.com/golang/text` 只是镜像仓库，其 `go.mod` 文件中定义的module还是 `module golang.org/x/text` ，这个module名字直接决定了你的import的路径。

调试依赖包

有时我们需要调试依赖包，此时就可以使用 `replace` 来修改依赖，如下所示：

```
replace (
github.com/google/uuid v1.1.1 => ../uuid
golang.org/x/text v0.3.2 => github.com/golang/text v0.3.2
)
```

语句 `github.com/google/uuid v1.1.1 => ../uuid` 使用本地的uuid来替换依赖包，此时，我们可以任意地修改 `../uuid` 目录的内容来进行调试。

除了使用相对路径，还可以使用绝对路径，甚至还可以使用自己的fork仓库。

使用fork仓库

有时在使用开源的依赖包时发现了bug，在开源版本还未修改或者没有新的版本发布时，你可以使用fork仓库，在fork仓库中进行bug fix。
你可以在fork仓库上发布新的版本，并相应的修改 `go.mod` 来使用fork仓库。

比如，我fork了开源包 `github.com/google/uuid` ，fork仓库地址为 `github.com/RainbowMango/uuid` ，那我们就可以在fork仓库里修改bug并发布新的版本 `v1.1.2` ，此时使用fork仓库的项目中 `go.mod` 中replace部分可以相应的做如下修改：

```
github.com/google/uuid v1.1.1 => github.com/RainbowMango/uuid v1.1.2
```

需要说明的是，使用fork仓库仅仅是临时的做法，一旦开源版本变得可用，需要尽快切换到开源版本。

禁止被依赖

另一种使用 `replace` 的场景是你的module不希望被直接引用，比如开源软件kubernetes，在它的 `go.mod` 中 `require` 部分有大量的 `v0.0.0` 依赖，比如：

```
module k8s.io/kubernetes

require (
    ...
    k8s.io/api v0.0.0
    k8s.io/apiextensions-apiserver v0.0.0
    k8s.io/apimachinery v0.0.0
    k8s.io/apiserver v0.0.0
    k8s.io/cli-runtime v0.0.0
    k8s.io/client-go v0.0.0
    k8s.io/cloud-provider v0.0.0
    ...
)
```

由于上面的依赖都不存在v0.0.0版本，所以其他项目直接依赖 `k8s.io/kubernetes` 时会因无法找到版本而无法使用。因为Kubernetes不希望作为module被直接使用，其他项目可以使用kubernetes其他子组件。

kubernetes 对外隐藏了依赖版本号，其真实的依赖通过 `replace` 指定：

```
replace (
    k8s.io/api => ./staging/src/k8s.io/api
    k8s.io/apiextensions-apiserver => ./staging/src/k8s.io/apiextensions-apiserver
    k8s.io/apimachinery => ./staging/src/k8s.io/apimachinery
    k8s.io/apiserver => ./staging/src/k8s.io/apiserver
    k8s.io/cli-runtime => ./staging/src/k8s.io/cli-runtime
    k8s.io/client-go => ./staging/src/k8s.io/client-go
    k8s.io/cloud-provider => ./staging/src/k8s.io/cloud-provider
)
```

前面我们说过， `replace` 指令在当前模块不是 `main module` 时会被自动忽略的，Kubernetes正是利用了这一特性来实现对外隐藏依赖版本号来实现禁止直接引用的目的。

> 赠人玫瑰手留余香，如果觉得不错请给个赞~
>
> 本篇文章已归档到GitHub项目，求星~ 点我即达

exclude指令

`go.mod` 文件中的 `exclude` 指令用于排除某个包的特定版本，其与 `replace` 类似，也仅在当前module为 `main module` 时有效，其他项目引用当前项目时， `exclude` 指令会被忽略。

`exclude` 指令在实际的项目中很少被使用，因为很少会显式地排除某个包的某个版本，除非我们知道某个版本有严重bug。
比如指令 `exclude github.com/google/uuid v1.1.0` ，表示不使用v1.1.0 版本。

下面我们还是使用 `github.com/renhongcai/gomodule` 来举例说明。

排除指定版本

在 `github.com/renhongcai/gomodule` 的v1.3.0版本中，我们的 `go.mod` 文件如下：

```
module github.com/renhongcai/gomodule

go 1.13

require (
  github.com/google/uuid v1.0.0
  golang.org/x/text v0.3.2
)

replace golang.org/x/text v0.3.2 => github.com/golang/text v0.3.2
```

`github.com/google/uuid v1.0.0` 说明我们期望使用 uuid包的 `v1.0.0` 版本。

假如，当前uuid仅有 `v1.0.0` 、 `v1.1.0` 和 `v1.1.1` 三个版本可用，而且我们假定 `v1.1.0` 版本有严重bug。
此时可以使用 `exclude` 指令将uuid的 `v1.1.0` 版本排除在外，即在 `go.mod` 文件添加如下内容：

```
exclude github.com/google/uuid v1.1.0
```

虽然我们暂时没有使用uuid的 `v1.1.0` 版本，但如果将来引用了其他包，正好其他包引用了uuid的 `v1.1.0` 版本的话，此时添加的 `exclude` 指令就会跳过 `v1.1.0` 版本。

下面我们创建 `github.com/renhongcai/exclude` 包来验证该问题。

创建依赖包

为了进一步说明 `exclude` 用法，我们创建了一个仓库 `github.com/renhongcai/exclude` ，并在其中创建了一个module `github.com/renhongcai/exclude` ，其中 `go.mod` 文件（v1.0.0版本）如下：

```
module github.com/renhongcai/exclude

go 1.13

require github.com/google/uuid v1.1.0
```

可以看出其依赖 `github.com/google/uuid` 的 `v1.1.0` 版本。创建 `github.com/renhongcai/exclude` 的目的是供 `github.com/renhongcai/gomodule` 使用的。

使用依赖包

由于 `github.com/renhongcai/exclude` 也引用了uuid包且引用了更新版本的uuid，那么在 `github.com/renhongcai/gomodule` 引用 `github.com/renhongcai/exclude` 时，会被动的提升uuid的版本。

在没有添加 `exclude` 之前，编译时 `github.com/renhongcai/gomodule` 依赖的uuid版本会提升到 `v1.1.0` ，与 `github.com/renhongcai/exclude` 保持一致，相应的 `go.mod` 也会被自动修改，如下所示：

```
module github.com/renhongcai/gomodule

go 1.13

require (
    github.com/google/uuid v1.1.0
    github.com/renhongcai/exclude v1.0.0
    golang.org/x/text v0.3.2
)

replace golang.org/x/text v0.3.2 => github.com/golang/text v0.3.2
```

但如果添加了 `exclude github.com/google/uuid v1.1.0` 指令后，编译时 `github.com/renhongcai/gomodule` 依赖的uuid版本会自动跳过 `v1.1.0` ，即选择 `v1.1.1` 版本，相应的 `go.mod` 文件如下所示：

```
module github.com/renhongcai/gomodule

go 1.13

require (
    github.com/google/uuid v1.1.1
    github.com/renhongcai/exclude v1.0.0
    golang.org/x/text v0.3.2
)

replace golang.org/x/text v0.3.2 => github.com/golang/text v0.3.2

exclude github.com/google/uuid v1.1.0
```

在本例中，在选择版本时，跳过uuid `v1.1.0` 版本后还有 `v1.1.1` 版本可用，Go 命令行工具可以自动选择 `v1.1.1` 版本，但如果没有更新的版本时将会报错而无法编译。

> 赠人玫瑰手留余香，如果觉得不错请给个赞~
> 本篇文章已归档到GitHub项目，求星~ [点我即达](https://github.com/RainbowMango/GoExpertProgramming)

indirect含义

在使用 Go module 过程中，随着引入的依赖增多，也许你会发现 `go.mod` 文件中部分依赖包后面会出现一个 `// indirect` 的标识。这个标识总是出现在 `require` 指令中，其中 `//` 与代码的行注释一样表示注释的开始，`indirect` 表示间接的依赖。

比如开源软件 Kubernetes（v1.17.0版本）的 go.mod 文件中就有数十个依赖包被标记为 `indirect` ：

```
require (
    github.com/Rican7/retry v0.1.0 // indirect
    github.com/auth0/go-jwt-middleware v0.0.0-20170425171159-5493cabe49f7 // indirect
    github.com/boltdb/bolt v1.3.1 // indirect
    github.com/checkpoint-restore/go-criu v0.0.0-20190109184317-bdb7599cd87b // indirect
    github.com/codegangsta/negroni v1.0.0 // indirect
    ...
)
```

在执行命令 `go mod tidy` 时，Go module 会自动整理 `go.mod 文件` ，如果有必要会在部分依赖包的后面增加 `// indirect` 注释。一般而言，被添加注释的包肯定是间接依赖的包，而没有添加 `// indirect` 注释的包则是直接依赖的包，即明确的出现在某个 `import` 语句中。

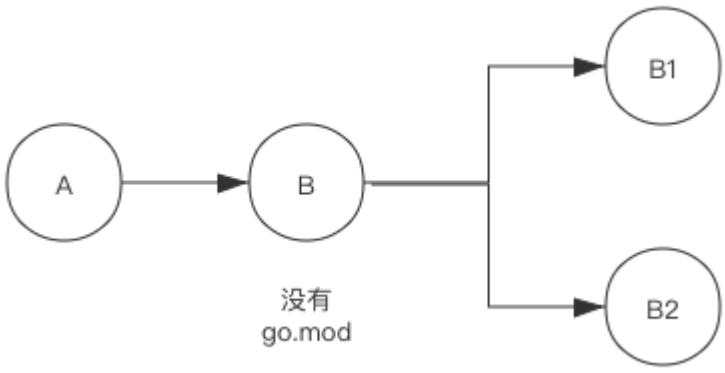
然而，这里需要着重强调的是：并不是所有的间接依赖都会出现在 `go.mod` 文件中。

间接依赖出现在 `go.mod` 文件的情况，可能符合下面所列场景的一种或多种：

- 直接依赖未启用 Go module
- 直接依赖go.mod 文件中缺失部分依赖

直接依赖未启用 Go module

如下图所示，Module A 依赖 B，但是 B 还未切换成 Module，也即没有 `go.mod` 文件，此时，当使用 `go mod tidy` 命令更新A的 `go.mod` 文件时，B的两个依赖B1和B2将会被添加到A的 `go.mod` 文件中（前提是A之前没有依赖B1和B2），并且B1 和B2还会被添加 `// indirect` 的注释。



此时Module A的 `go.mod` 文件中require部分将会变成：

```
require (
    B vx.x.x
    B1 vx.x.x // indirect
```

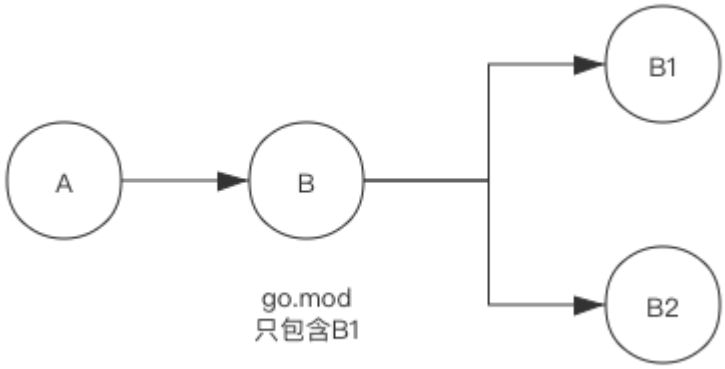
```
    B2 vx.x.x // indirect
)
```

依赖B及B的依赖B1和B2都会出现在 `go.mod` 文件中。

直接依赖 go.mod 文件不完整

如上面所述，如果依赖B没有 `go.mod` 文件，则Module A 将会把B的所有依赖记录到A 的 `go.mod` 文件中。即便B拥有 `go.mod` ，如果 `go.mod` 文件不完整的话，Module A依然会记录部分B的依赖到 `go.mod` 文件中。

如下图所示，Module B虽然提供了 `go.mod` 文件中，但 `go.mod` 文件中只添加了依赖B1，那么此时A在引用B时，则会在A的 `go.mod` 文件中添加B2作为间接依赖，B1则不会出现在A的 `go.mod` 文件中。



此时Module A的 `go.mod` 文件中require部分将会变成：

```
require (
    B vx.x.x
    B2 vx.x.x // indirect
)
```

由于B1已经包含进B的 `go.mod` 文件中，A的 `go.mod` 文件则不必再记录，只会记录缺失的B2。

总结

为什么要记录间接依赖

在上面的例子中，如果某个依赖B 没有 `go.mod` 文件，在A 的 `go.mod` 文件中已经记录了依赖B及其版本号，为什么还要增加间接依赖呢？

我们知道Go module需要精确地记录软件的依赖情况，虽然此处记录了依赖B的版本号，但B的依赖情况没有记录下来，所以如果B的 `go.mod` 文件缺失了（或没有）这个信息，则需要在A的 `go.mod` 文件中记录下来。此时间接依赖的版本号将会根据Go module的版本选择机制确定一个最优版本。

如何处理间接依赖

综上所述间接依赖出现在 `go.mod` 中，可以一定程度上说明依赖有瑕疵，要么是其不支持Go module，要么是其 `go.mod` 文件不完整。

由于Go 语言从v1.11版本才推出module的特性，众多开源软件迁移到go module还需要一段时间，在过渡期必然会出现间接依赖，但随着时间的推进，在 `go.mod` 中出现 `// indirect` 的机率会越来越低。

出现间接依赖可能意味着你在使用过时的软件，如果有精力的话还是推荐尽快消除间接依赖。可以通过使用依赖的新版本或者替换依赖的方式消除间接依赖。

如何查找间接依赖来源

Go module提供了 `go mod why` 命令来解释为什么会依赖某个软件包，若要查看 `go.mod` 中某个间接依赖是被哪个依赖引入的，可以使用命令 `go mod why -m <pkg>` 来查看。

比如，我们有如下的 `go.mod` 文件片断：

```
require (
    github.com/Rican7/retry v0.1.0 // indirect
    github.com/google/uuid v1.0.0
    github.com/renhongcai/indirect v1.0.0
    github.com/spf13/pflag v1.0.5 // indirect
    golang.org/x/text v0.3.2
)
```

我们希望确定间接依赖 `github.com/Rican7/retry v0.1.0 // indirect` 是被哪个依赖引入的，则可以使用命令 `go mod why` 来查看：

```
[root@ecs-d8b6 gomodule]# go mod why -m github.com/Rican7/retry
# github.com/Rican7/retry
github.com/renhongcai/gomodule
github.com/renhongcai/indirect
github.com/Rican7/retry
```

上面的打印信息中 `# github.com/Rican7/retry` 表示当前正在分析的依赖，后面几行则表示依赖链。`github.com/renhongcai/gomodule` 依赖 `github.com/renhongcai/indirect`，而 `github.com/renhongcai/indirect` 依赖 `github.com/Rican7/retry`。由此我们就可以判断出间接依赖 `github.com/Rican7/retry` 是被 `github.com/renhongcai/indirect` 引入的。

另外，命令 `go mod why -m all` 则可以分析所有依赖的依赖链。

> 赠人玫瑰手留余香，如果觉得不错请给个赞~
> 本篇文章已归档到GitHub项目，求星~ [点我即达](https://github.com/RainbowMango/GoExpertProgramming)

版本选择机制

在前面的章节中，我们使用过 `go get <pkg>` 来获取某个依赖，如果没有特别指定依赖的版本号， `go get` 会自动选择一个最优版本，并且如果本地有 `go.mod` 文件的话，还会自动更新 `go.mod` 文件。

事实上除了 `go get` ， `go build` 和 `go mod tidy` 也会自动帮我们选择依赖的版本。这些命令选择依赖版本时都遵循一些规则，本节我们就开始介绍Go module涉及到的版本选择机制。

依赖包版本约定

关于如何管理依赖包的版本，Go语言提供了一个规范，并且Go语言的演进过程中也一直遵循这个规范。

这个非强制性的规范主要围绕包的兼容性展开。对于如何处理依赖包的兼容性，根据是否支持Go module分别有不同的建议。

Go module 之前版本兼容性

在Go v1.11（开始引入Go module的版本）之前，Go 语言建议依赖包需要保持向后兼容，这包括可导出的函数、变量、类型、常量等不可以随便删除。以函数为例，如果需要修改函数的入参，可以增加新的函数而不是直接修改原有的函数。

如果确实需要做一些打破兼容性的修改，建议创建新的包。

比如仓库 `github.com/RainbowMango/xxx` 中包含一个package A，此时该仓库只有一个package：

- `github.com/RainbowMango/xxx/A`

那么其他项目引用该依赖时的import 路径为：

```
import "github.com/RainbowMango/xxx/A"
```

如果该依赖包需要引入一个不兼容的特性，可以在该仓库中增加一个新的package A1，此时该仓库包含两个包：

- `github.com/RainbowMango/xxx/A`
- `github.com/RainbowMango/xxx/A1`

那么其他项目在升级依赖包版本后不需要修改原有的代码可以继续使用package A，如果需要使用新的package A1，只需要将import 路径修改为 `import "github.com/RainbowMango/xxx/A1"` 并做相应的适配即可。

Go module 之后版本兼容性

从Go v1.11版本开始，随着Go module特性的引入，依赖包的兼容性要求有了进一步的延伸，Go module开始关心依赖包版本管理系统（如Git）中的版本号。尽管如此，兼容性要求的核心内容没有改变：

- 如果新package 和旧的package拥有相同的import 路径，那么新package必须兼容旧的package;
- 如果新的package不能兼容旧的package，那么新的package需要更换import路径；

在前面的介绍中，我们知道Go module 的 `go.mod` 中记录的module名字决定了import路径。例如，要引用module `module github.com/renhongcai/indirect` 中的内容时，其import路径需要为 `import github.com/renhongcai/indirect` 。

在Go module时代，module版本号要遵循语义化版本规范，即版本号格式为 `v<major>.<minor>.<patch>` ，如v1.2.3。当有不兼容的改变时，需要增加 `major` 版本号，如v2.1.0。

Go module规定，如果 `major` 版本号大于 `1` ，则 `major` 版本号需要显式地标记在module名字中，如 `module github.com/my/mod/v2` 。这样做的好处是Go module 会把 `module github.com/my/mod/v2` 和

`module github.com/my/mod` 视做两个module，他们甚至可以被同时引用。

另外，如果module的版本为 `v0.x.x` 或 `v1.x.x` 则都不需要在module名字中体现版本号。

版本选择机制

Go 的多个命令行工具都有自动选择依赖版本的能力，如 `go build` 和 `go test` ，当在源代码中增加了新的import，这些命令将会自动选择一个最优的版本，并更新 `go.mod` 文件。

需要特别说明的是，如果 `go.mod` 文件中已标记了某个依赖包的版本号，则这些命令不会主动更新 `go.mod` 中的版本号。所谓自动更新版本号只在 `go.mod` 中缺失某些依赖或者依赖不匹配时才会发生。

最新版本选择

当在源代码中新增加了一个import，比如：

```
import "github.com/RainbowMango/M"
```

如果 `go.mod` 的require指令中并没有包含 `github.com/RainbowMango/M` 这个依赖，那么 `go build` 或 `go test` 命令则会去 `github.com/RainbowMango/M` 仓库寻找最新的符合语义化版本规范的版本，比如v1.2.3，并在 `go.mod` 文件中增加一条require依赖：

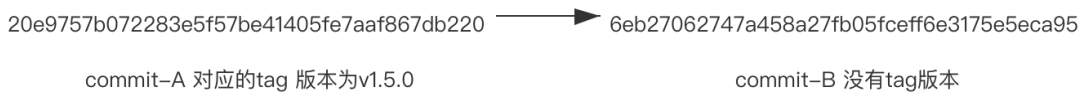
```
require github.com/RainbowMango/M v1.2.3
```

这里，由于import路径里没有类似于 `v2` 或更高的版本号，所以版本选择时只会选择v1.x.x的版本，不会去选择v2.x.x或更高的版本。

最小版本选择

有时记录在 `go.mod` 文件中的依赖包版本会随着引入其他依赖包而发生变化。

如下图所示：



commit-A 对应的tag 版本为v1.5.0　　commit-B 没有tag版本

Module A 依赖 Module M的v1.0.0版本，但之后 Module A 引入了 Module D，而Module D 依赖 Module M的v1.1.1版本，此时，由于依赖的传递，Module A也会选择v1.1.1版本。

需要注意的是，此时会自动选择最小可用的版本，而不是最新的tag版本。

总结

本节我们先是介绍了Go 语言针对依赖包版本管理的约定，这个不能算是强制性的要求，但如果不遵守该约定，后续该依赖包的使用者将会遇到各麻烦，最终有可能弃用这个不规范的依赖包。

接着介绍了Go module机制所采用的自动版本选择算法，除了自动版本选择以外，我们还可以显式地指定依赖包的版本。另外，除了在 `go.mod` 文件中指定依赖的tag 版本号以外，还可以使用假的版本号，这些内容我们将在后续的章节中分别展开介绍。

> 赠人玫瑰手留余香，如果觉得不错请给个赞~

> 本篇文章已归档到GitHub项目，求星~ 点我即达

incompatible

在前面的章节中，我们介绍了Go module的版本选择机制，其中介绍了一个Module的版本号需要遵循 `v<major>.<minor>.<patch>` 的格式，此外，如果 `major` 版本号大于1时，其版本号还需要体现在Module名字中。

比如Module `github.com/RainbowMango/m` ，如果其版本号增长到 `v2.x.x` 时，其Module名字也需要相应的改变为：
`github.com/RainbowMango/m/v2` 。即，如果 `major` 版本号大于1时，需要在Module名字中体现版本。

那么如果Module的 `major` 版本号虽然变成了 `v2.x.x` ，但Module名字仍保持原样会怎么样呢？ 其他项目是否还可以引用呢？其他项目引用时有没有风险呢？这就是今天要讨论的内容。

能否引用不兼容的包

我们还是以Module `github.com/RainbowMango/m` 为例，假如其当前版本为 `v3.6.0` ，因为其Module名字未遵循Golang所推荐的风格，即Module名中附带版本信息，我们称这个Module为不规范的Module。

不规范的Module还是可以引用的，但跟引用规范的Module略有差别。

如果我们在项目A中引用了该module，使用命令 `go mod tidy` ，go 命令会自动查找Module m的最新版本，即 `v3.6.0` 。
由于Module为不规范的Module，为了加以区分，go 命令会在 `go.mod` 中增加 `+incompatible` 标识。

```
require (
    github.com/RainbowMango/m v3.6.0+incompatible
)
```

除了增加 `+incompatible` （不兼容）标识外，在其使用上没有区别。

如何处理incompatible

`go.mod` 文件中出现 `+incompatible` ，说明你引用了一个不规范的Module，正常情况下，只能说明这个Module版本未遵循版本化语义规范。但引用这个规范的Module还是有些困扰，可能还会有一定的风险。

比如，我们拿某开源Module `github.com/blang/semver` 为例，编写本文时，该Module最新版本为 `v3.6.0` ，但其 `go.mod` 中记录的Module却是：

```
module github.com/blang/semver
```

Module `github.com/blang/semver` 在另一个著名的开源软件 `Kubernetes` （github.com/kubernetes/kubernetes）中被引用，那么 `Kubernetes` 的 `go.mod` 文件则会标记这个Module为 `+incompatible` ：

```
require (
    ...
    github.com/blang/semver v3.5.0+incompatible
    ...
)
```

站在 `Kubernetes` 的角度，此处的困扰在于，如果将来 `github.com/blang/semver` 发布了新版本 `v4.0.0` ，但不幸的是Module名字仍然为 `github.com/blang/semver` 。那么，升级这个Module的版本将会变得困难。因为 `v3.6.0` 到 `v4.0.0` 跨越了大版本，按照语义化版本规范来解释说明发生了不兼容的改变，即然不兼容，项目维护者有必须对升级持谨慎态度，甚至放弃升级。

站在 `github.com/blang/semver` 的角度，如果迟迟不能将自身变得”规范”，那么其他项目有可能放弃本Module，转而使用其他更规范的Module来替代，开源项目如果没有使用者，也就走到了尽头。

> 赠人玫瑰手留余香，如果觉得不错请给个赞~
>
> 本篇文章已归档到GitHub项目，求星~ 点我即达

伪版本

在 `go.mod` 中通常使用语义化版本来标记依赖，比如 `v1.2.3` 、 `v0.1.5` 等。因为 `go.mod` 文件通常是 `go` 命令自动生成并修改的，所以实际上是 `go` 命令习惯使用语义化版本。

诸如 `v1.2.3` 和 `v0.1.5` 这样的语义化版本，实际是某个commit ID的标记，真正的版本还是commit ID。比如 `github.com/renhongcai/gomodule` 项目的 `v1.5.0` 对应的真实版本为 `20e9757b072283e5f57be41405fe7aaf867db220` 。

由于语义化版本比 `commit ID` 更直观（方便交流与比较版本大小），所以一般情况下使用语义化版本。

什么是伪版本

在实际项目中，有时不得不直接使用一个 `commit ID` ，比如某项目发布了 `v1.5.0` 版本，但随即又修复了一个bug（引入一个新的commit ID），而且没有发布新的版本。此时，如果我们希望使用最新的版本，就需要直接引用最新的 `commit ID` ，而不是之前的语义化版本 `v1.5.0` 。
使用 `commit ID` 的版本在Go语言中称为 `pseudo-version` ，可译为"伪版本"。

伪版本的版本号通常会使用 `vx.y.z-yyyymmddhhmmss-abcdefabcdef` 格式，其中 `vx.y.z` 看上去像是一个真实的语义化版本，但通常并不存在该版本，所以称为伪版本。另外 `abcdefabcdef` 表示某个commit ID的前12位，而 `yyyymmddhhmmss` 则表示该commit的提交时间，方便做版本比较。

使用伪版本的 `go.mod` 举例如下：

```
...
require (
    go.etcd.io/etcd v0.0.0-20191023171146-3cf2f69b5738
)
...
```

伪版本风格

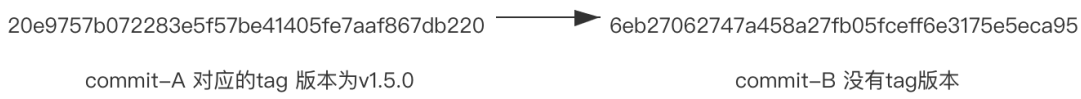
伪版本格式都为 `vx.y.z-yyyymmddhhmmss-abcdefabcdef` ，但 `vx.y.z` 部分在不同情况下略有区别，有时可能是 `vx.y.z-pre.0` 或者 `vx.y.z-0` ，甚至 `vx.y.z-dev.2.0` 等。

`vx.y.z` 的具体格式取决于所引用 `commit ID` 之前的版本号，如果所引用 `commit ID` 之前的最新的tag版本为 `v1.5.0` ，那么伪版本号则在其基础上增加一个标记，即 `v1.5.1-0` ，看上去像是下一个版本一样。

实际使用中 `go` 命令会帮我们自动生成伪版本，不需要手动计算，所以此处我们仅做基本说明。

如何获取伪版本

我们使用具体的例子还演示如何使用伪版本。在仓库 `github.com/renhongcai/gomodule` 中存在 `v1.5.0` tag 版本，在 `v1.5.0` 之后又提交了一个commit，并没有发布新的版本。其版本示意图如下：



为了方便描述，我们把 `v1.5.0` 对应的commit 称为 `commit-A` ，而其随后的commit称为 `commit-B` 。

如果我们要使用commit-A，即 `v1.5.0` ，可使用 `go get github.com/renhongcai/gomodule@v1.5.0` 命令：

```
[root@ecs-d8b6 ~]# go get github.com/renhongcai/gomodule@v1.5.0
go: finding github.com/renhongcai/gomodule v1.5.0
go: downloading github.com/renhongcai/gomodule v1.5.0
go: extracting github.com/renhongcai/gomodule v1.5.0
go: finding github.com/renhongcai/indirect v1.0.1
```

此时，如果存在 `go.mod` 文件， `github.com/renhongcai/gomodule` 体现在 `go.mod` 文件的版本为 `v1.5.0` 。

如果我们要使用commit-B，可使用 `go get github.com/renhongcai/gomodule@6eb27062747a458a27fb05fceff6e3175e5eca95` 命令（可以使用完整的commit id，也可以只使用前12位）：

```
[root@ecs-d8b6 ~]# go get github.com/renhongcai/gomodule@6eb27062747a458a27fb05fceff6e3175e5eca95
go: finding github.com 6eb27062747a458a27fb05fceff6e3175e5eca95
go: finding github.com/renhongcai/gomodule 6eb27062747a458a27fb05fceff6e3175e5eca95
go: finding github.com/renhongcai 6eb27062747a458a27fb05fceff6e3175e5eca95
go: downloading github.com/renhongcai/gomodule v1.5.1-0.20200203082525-6eb27062747a
go: extracting github.com/renhongcai/gomodule v1.5.1-0.20200203082525-6eb27062747a
go: finding github.com/renhongcai/indirect v1.0.2
```

此时，可以看到生成的伪版本号为 `v1.5.1-0.20200203082525-6eb27062747a` ，当前最新版本为 `v1.5.0` ， `go` 命令生成伪版本号时自动增加了版本。此时，如果存在 `go.mod` 文件的话， `github.com/renhongcai/gomodule` 体现在 `go.mod` 文件中的版本则为该伪版本号。

> 赠人玫瑰手留余香，如果觉得不错请给个赞~
>
> 本篇文章已归档到GitHub项目，求星~ 点我即达

依赖包存储

在前面介绍 `GOPATH` 的章节中，我们提到 `GOPATH` 模式下不方便使用同一个依赖包的多个版本。在 `GOMODULE` 模式下这个问题得到了很好的解决。

`GOPATH` 模式下，依赖包存储在 `$GOPATH/src` ，该目录下只保存特定依赖包的一个版本，而在 `GOMODULE` 模式下，依赖包存储在 `$GOPATH/pkg/mod` ，该目录中可以存储特定依赖包的多个版本。

需要注意的是 `$GOPATH/pkg/mod` 目录下有个 `cache` 目录，它用来存储依赖包的缓存，简单说， `go` 命令每次下载新的依赖包都会在该 `cache` 目录中保存一份。关于该目录的工作机制我们留到 `GOPROXY` 章节时再详细介绍。

接下来，我们使用开源项目 `github.com/google/uuid` 为例分别说明 `GOPATH` 模式和 `GOMODULE` 模式下特定依赖包存储机制。在下面的操作中，我们会使用 `GO111MODULE` 环境变量控制具体的模式：

- `export GO111MODULE=off` 切换到 `GOPATH` 模式
- `export GO111MODULE=on` 切换到 `GOMODULE` 模式。

GOPATH 依赖包存储

为了实验 `GOPATH` 模式下依赖包的存储方式，我们可以使用以下命令来获取 `github.com/google/uuid` ：

```
# export GO111MODULE=off
# go get -v github.com/google/uuid
```

在 `GOPATH` 模式下， `go get` 命令会将依赖包下载到 `$GOPATH/src/google` 目录中。

该命令等同于在 `$GOPATH/src/google` 目录下执行 `git clone https://github.com/google/uuid.git` ，也就是 `$GOPATH/src/google/uuid` 目录中存储的是完整的仓库。

GOMODULE 依赖包存储

为了实验 `GOMODULE` 模式下依赖的存储方式，我们使用以下命令来获取 `github.com/google/uuid` ：

```
# export GO111MODULE=on
# go get -v github.com/google/uuid
# go get -v github.com/google/uuid@v1.0.0
# go get -v github.com/google/uuid@v1.1.0
# go get -v github.com/google/uuid@v1.1.1
```

在 `GOMODULE` 模式下， `go get` 命令会将依赖包下载到 `$GOPATH/pkg/mod` 目录下，并且按照依赖包的版本分别存放。（注： `go get` 命令不指定特定版本时，默认会下载最新版本，即v1.1.1，如软件包有新版本发布，实验结果将有所不同。）

此时 `$GOPATH/pkg/mod` 目录结构如下：

```
${GOPATH}/pkg/mod/github.com/google
├── uuid@v1.0.0
├── uuid@v1.1.0
└── uuid@v1.1.1
```

相较于 `GOPATH` 模式， `GOMODULE` 有两处不同点：

- 一是依赖包的目录中包含了版本号，每个版本占用一个目录；
- 二是依赖包的特定版本目录中只包含依赖包文件，不包含`.git`目录；

由于依赖包的每个版本都有一个唯一的目录，所以在多项目场景中需要使用同一个依赖包的多版本时才不会产生冲突。另外，由于依赖包的每个版本都有唯一的目录，也表示该目录内容不会发生改变，也就不必再存储其位于版本管理系统(如git)中的信息。

包名大小写敏感问题

有时我们使用的包名中会包含大写字母，比如`github.com/Azure/azure-sdk-for-go`，`GOMODULE`模式下，在存储时会将包名做大小写编码处理，即每个大写字母将变与`!`+相应的小写字母，比如`github.com/Azure`包在存储时将会被放置在`$GOPATH/pkg/mod/github.com/!azure`目录中。

需要注意的是，`GOMODULE`模式下，我们使用`go get`命令时，如果不小心将某个包名大小写搞错，比如`github.com/google/uuid`写成`github.com/google/UUID`时，在存储依赖包时会严格按照`go get`命令指示的包名进行存储。

如下所示，使用大写的`UUID`：

```
[root@ecs-d8b6 uuid]# go get -v github.com/google/UUID@v1.0.0
go: finding github.com v1.0.0
go: finding github.com/google v1.0.0
go: finding github.com/google/UUID v1.0.0
go: downloading github.com/google/UUID v1.0.0
go: extracting github.com/google/UUID v1.0.0
github.com/google/UUID
```

由于`github.com/google/uuid`域名不区分大小写，所以使用`github.com/google/UUID`下载包时仍然可以下载，但在存储时将会严格区分大小写，此时`$GOPATH/pkg/mod/google/`目录下将会多出一个d@v1.0.0"">`!u!u!i!d@v1.0.0`目录：

```
${GOPATH}/pkg/mod/github.com/google
├─── uuid@v1.0.0
├─── uuid@v1.1.0
├─── uuid@v1.1.1
├─── !u!u!i!d@v1.0.0
```

在`go get`中使用错误的包名，除了会增加额外的不必要存储外，还可能会影响`go`命令解析依赖，还可能将错误的包名使用到`import`指令中，所以在实际使用时应该尽量避免。

> 赠人玫瑰手留余香，如果觉得不错请给个赞~
>
> 本篇文章已归档到GitHub项目，求星~ 点我即达

go.sum文件

为了确保一致性构建，Go引入了 `go.mod` 文件来标记每个依赖包的版本，在构建过程中 `go` 命令会下载 `go.mod` 中的依赖包，下载的依赖包会缓存在本地，以便下次构建。
考虑到下载的依赖包有可能是被黑客恶意篡改的，以及缓存在本地的依赖包也有被篡改的可能，单单一个 `go.mod` 文件并不能保证一致性构建。

为了解决Go module的这一安全隐患，Go开发团队在引入 `go.mod` 的同时也引入了 `go.sum` 文件，用于记录每个依赖包的哈希值，在构建时，如果本地的依赖包hash值与 `go.sum` 文件中记录得不一致，则会拒绝构建。

本节暂不对模块校验细节展开介绍，只从日常应用层面介绍：

- go.sum 文件记录含义
- go.sum文件内容是如何生成的
- go.sum是如何保证一致性构建的

go.sum文件记录

`go.sum` 文件中每行记录由 `module` 名、版本和哈希组成，并由空格分开：

```
<module> <version>[/go.mod] <hash>
```

比如，某个 `go.sum` 文件中记录了 `github.com/google/uuid` 这个依赖包的 `v1.1.1` 版本的哈希值：

```
github.com/google/uuid v1.1.1 h1:Gkbcsh/GbpXz7lPftLA3P6TYMwjCLYm83jiFQZF/3gY=
github.com/google/uuid v1.1.1/go.mod h1:TIyPZe4MgqvfeYDBFedMoGGpEw/LqOeaOT+nhxU+yHo=
```

在Go module机制下，我们需要同时使用依赖包的名称和版本才可以准确的描述一个依赖，为了方便叙述，下面我们使用 `依赖包版本` 来指代依赖包名称和版本。

正常情况下，每个 `依赖包版本` 会包含两条记录，第一条记录为该 `依赖包版本` 整体（所有文件）的哈希值，第二条记录仅表示该 `依赖包版本` 中 `go.mod` 文件的哈希值，如果该 `依赖包版本` 没有 `go.mod` 文件，则只有第一条记录。如上面的例子中，`v1.1.1` 表示该 `依赖包版本` 整体，而 `v1.1.1/go.mod` 表示该 `依赖包版本` 中 `go.mod` 文件。

`依赖包版本` 中任何一个文件（包括 `go.mod` ）改动，都会改变其整体哈希值，此处再额外记录 `依赖包版本` 的 `go.mod` 文件主要用于计算依赖树时不必下载完整的 `依赖包版本` ，只根据 `go.mod` 即可计算依赖树。

每条记录中的哈希值前均有一个表示哈希算法的 `h1:` ，表示后面的哈希值是由算法 `SHA-256` 计算出来的，自Go module从v1.11版本初次实验性引入，直至v1.14，只有这一个算法。

此外，细心的读者或许会发现 `go.sum` 文件中记录的 `依赖包版本` 数量往往比 `go.mod` 文件中要多，这是因为二者记录的粒度不同导致的。`go.mod` 只需要记录直接依赖的 `依赖包版本` ，只在 `依赖包版本` 不包含 `go.mod` 文件时候才会记录间接 `依赖包版本` ，而 `go.sum` 则是要记录构建用到的所有 `依赖包版本` 。

生成

假设我们在开发某个项目，当我们在GOMODULE模式下引入一个新的依赖时，通常会使用 `go get` 命令获取该依赖，比如：

```
go get github.com/google/uuid@v1.0.0
```

`go get` 命令首先会将该依赖包下载到本地缓存目录 `$GOPATH/pkg/mod/cache/download` ，该依赖包为一个后缀为 `.zip` 的压缩包，如 `v1.0.0.zip` 。`go get` 下载完成后会对该 `.zip` 包做哈希运算，并将结果存放在后缀为 `.ziphash` 的文件中，如 `v1.0.0.ziphash` 。如果在项目的根目录中执行 `go get` 命令的话， `go get` 会同步更新 `go.mod` 和 `go.sum` 文件， `go.mod` 中记录的是依赖名及其版本，如：

```
require (
    github.com/google/uuid v1.0.0
)
```

`go.sum` 文件中则会记录依赖包的哈希值（同时还有依赖包中go.mod的哈希值），如：

```
github.com/google/uuid v1.0.0 h1:b4Gk+7WdP/d3HZH8EJsZpvV7EtDOgaZLtnaNGIuladA=
github.com/google/uuid v1.0.0/go.mod h1:TIyPZe4MgqvfeYDBFedMoGGpEw/LqOeaOT+nhxU+yHo=
```

值得一提的是，在更新 `go.sum` 之前，为了确保下载的依赖包是真实可靠的， `go` 命令在下载完依赖包后还会查询 `GOSUMDB` 环境变量所指示的服务器，以得到一个权威的 `依赖包版本` 哈希值。如果 `go` 命令计算出的 `依赖包版本` 哈希值与 `GOSUMDB` 服务器给出的哈希值不一致， `go` 命令将拒绝向下执行，也不会更新 `go.sum` 文件。

`go.sum` 存在的意义在于，我们希望别人或者在别的环境中构建当前项目时所使用依赖包跟 `go.sum` 中记录的是完全一致的，从而达到一致构建的目的。

校验

假设我们拿到某项目的源代码并尝试在本地构建， `go` 命令会从本地缓存中查找所有 `go.mod` 中记录的依赖包，并计算本地依赖包的哈希值，然后与 `go.sum` 中的记录进行对比，即检测本地缓存中使用的 `依赖包版本` 是否满足项目 `go.sum` 文件的期望。

如果校验失败，说明本地缓存目录中 `依赖包版本` 的哈希值和项目中 `go.sum` 中记录的哈希值不一致， `go` 命令将拒绝构建。
这就是 `go.sum` 存在的意义，即如果不使用我期望的版本，就不能构建。

当校验失败时，有必要确认到底是本地缓存错了，还是 `go.sum` 记录错了。
需要说明的是，二者都可能出错，本地缓存目录中的 `依赖包版本` 有可能被有意或无意地修改过，项目中 `go.sum` 中记录的哈希值也可能被篡改过。

当校验失败时， `go` 命令倾向于相信 `go.sum` ，因为一个新的 `依赖包版本` 在被添加到 `go.sum` 前是经过 `GOSUMDB` （校验和数据库）验证过的。此时即便系统中配置了 `GOSUMDB` （校验和数据库）， `go` 命令也不会查询该数据库。

校验和数据库

环境变量 `GOSUMDB` 标识一个 `checksum database` ，即校验和数据库，实际上是一个web服务器，该服务器提供查询 `依赖包版本` 哈希值的服务。

该数据库中记录了很多 `依赖包版本` 的哈希值，比如Google官方的 `sum.golang.org` 则记录了所有的可公开获得的 `依赖包版本` 。除了使用官方的数据库，还可以指定自行搭建的数据库，甚至干脆禁用它（ `export GOSUMDB=off` ）。

如果系统配置了 `GOSUMDB` ，在 `依赖包版本` 被写入 `go.sum` 之前会向该数据库查询该 `依赖包版本` 的哈希值进行二次校验，校验无误后再写入 `go.sum` 。

如果系统禁用了 `GOSUMDB` ，在 `依赖包版本` 被写入 `go.sum` 之前则不会进行二次校验， `go` 命令会相信所有下载到的依赖包，并把其哈希值记录到 `go.sum` 中。

参考文档

- 命令行帮助文档（go 1.14）：`$ go help module-auth`
- Go 1.14 源码

> 赠人玫瑰手留余香，如果觉得不错请给个赞~
>
> 本篇文章已归档到GitHub项目，求星~ 点我即达

编程陷阱

本章介绍一些项目过程中可能会遇到的陷阱，大部分均来源于实际项目，比较具有代表性。

切片追加

Golang 内置方法 `append` 用于向切片中追加一个或多个元素，实际项目中比较常见。

其原型如下：

```
func append(slice []Type, elems ...Type) []Type
```

本节不会对 `append` 的使用方式详细展开，而是重点介绍几个使用中常见的误区或者陷阱。

热身

按照惯例，我们先拿几个小题目来检测一下对 `append` 的理解是否足够深刻。

题目一

函数 `Validation()` 用于一些合法性检查，每遇到一个错误，就生成一个新的 `error` 并追加到切片 `errs` 中，最后返回包含所有错误信息的切片。
为了简单起见，假定函数发现了三个错误，如下所示：

```
func Validation() []error {
    var errs []error

    append(errs, errors.New("error 1"))
    append(errs, errors.New("error 2"))
    append(errs, errors.New("error 3"))

    return errs
}
```

请问函数 `Validation()` 有什么问题？

题目二

函数 `ValidateName()` 用于检查某个名字是否合法，如果不为空则认为合法，否则返回一个 `error` 。
类似的，还可以有很多检查项，比如检查性别、年龄等，我们统称为子检查项。
函数 `Validations()` 用于收集所有子检查项的错误信息，将错误信息汇总到一个切片中返回。

请问函数 `Validations()` 有什么问题？

```
func ValidateName(name string) error {
    if name != "" {
        return nil
    }

    return errors.New("empty name")
}

func Validations(name string) []error {
    var errs []error

    errs = append(errs, ValidateName(name))
```

```
    return errs
}
```

陷阱

前面的热身题目均来源于实际项目（已经做了最大程度的精简），分别代表一个本节将要介绍的陷阱。

陷阱一： append 会改变切片的地址

`append` 的本质是向切片中追加数据，而随着切片中元素逐渐增加，当切片底层的数组将满时，切片会发生扩容，扩容会导致产生一个新的切片（拥有容量更大的底层数组），更多关于切片的信息，请查阅切片相关章节。

`append` 每个追加元素，都有可能触发切片扩容，也即有可能返回一个新的切片，这也是 `append` 函数声明中返回值为切片的原因。实际使用中应该总是接收该返回值。

上述题目一中，由于初始切片长度为0，所以实际上每次 `append` 都会产生一个新的切片并迅速抛弃（被gc回收）。原始切片并没有任何改变。需要特别说明的是，不管初始切片长度为多少，不接收 `append` 返回都是有极大风险的。

另外，目前有很多的工具可以自动检查出类似的问题，比如 `Goland` IDE就会给出很明显的提示。

陷阱二： append 可以追加nil值

向切片中追加一个 `nil` 值是完全不会报错的，如下代码所示：

```
slice := append(slice, nil)
```

经过追加后，slice的长度递增1。

实际上 `nil` 是一个预定义的值，即空值，所以完全有理由向切片中追加。

上述题目二中，就是典型的向切片中追加 `nil` （当名字为空时）的问题。单纯从技术上讲是没有问题，但在题目二场景中就有很大的问题。

题目中函数用于收集所有错误信息，没有错误就不应该追加到切片中。因后，后续极有可能会根据切片的长度来判断是否有错误发生，比如：

```
func foo() {
    errs := Validations("")

    if len(errs) > 0 {
        println(errs)
        os.Exit(1)
    }
}
```

如果向切片中追加一个 `nil` 元素，那么切片长度则不再为0，程序很可能因此而退出，更糟糕的是，这样的切片是没有内容会打印出来的，这无疑又增加了定位难度。

> 赠人玫瑰手留余香，如果觉得不错请给个赞~
> 你的鼓励将成为我继续写作的动力！
>
> 本篇文章已归档到GitHub项目，求星~ 点我即达

循环变量绑定

本节通过几个实例来介绍循环遍历时，尤其是使用循环变量时可能遇到的问题，希望通过本节内容的学习，
读者能够在实际项目中加以避免。

该类问题出现的频率超乎你的想像，不仅笔者本人参与的项目，甚至某些著名的开源项目中也普遍存在类似的问题。
所以，我希望在本文中对该类问题做一次总结性的分析。

热身

按照惯例，我们还是从几个小题目开始，权当热身。

题目一

```
func Process1(tasks []string) {
    for _, task := range tasks {
        // 启动协程并发处理任务
        go func() {
            fmt.Printf("Worker start process task: %s\n", task)
        }()
    }
}
```

函数 `Process1()` 用于处理任务，每个任务均启动一个协程进行处理。
请问函数是否有问题？

题目二

```
func Process2(tasks []string) {
    for _, task := range tasks {
        // 启动协程并发处理任务
        go func(t string) {
            fmt.Printf("Worker start process task: %s\n", t)
        }(task)
    }
}
```

函数 `Process2()` 用于处理任务，每个任务均启动一个协程进行处理。
协程匿名函数接收一个任务作为参数，并进行处理。
请问函数是否有问题？

题目三

项目中经常需要编写单元测试，而单元测试最常见的是 `table-driven` 风格的测试，如下所示：
待测函数很简单，只是计算输入数值的2倍值。

```
func Double(a int) int {
    return a * 2
}
```

测试函数如下：

```
func TestDouble(t *testing.T) {
    var tests = []struct {
        name         string
        input        int
        expectOutput int
    }{
        {
            name:         "double 1 should got 2",
            input:        1,
            expectOutput: 2,
        },
        {
            name:         "double 2 should got 4",
            input:        2,
            expectOutput: 4,
        },
    }

    for _, test := range tests {
        t.Run(test.name, func(t *testing.T) {
            if test.expectOutput != Double(test.input) {
                t.Fatalf("expect: %d, but got: %d", test.input, test.expectOutput)
            }
        })
    }
}
```

上述测试函数也很简单，通过设计多个测试用例，标记输入输出，使用子测试进行验证。
（注：如果不熟悉单元测试，请查阅相关章节）

请问，上述测试有没有问题？

原理剖析

上述三个问题，有个共同点就是都引用了循环变量。即在 `for index, value := range xxx` 语句中，`index` 和 `value` 便是循环变量。不同点是循环变量的使用方式，有的是直接在协程中引用（题目一），有的作为参数传递（题目二），而题目三则是兼而有之。

回答以上问题，记住以下两点即可。

循环变量是易变的

首先，循环变量实际上只是一个普通的变量。

语句 `for index, value := range xxx` 中，每次循环index和value都会被重新赋值（并非生成新的变量）。

如果循环体中会启动协程（并且协程会使用循环变量），就需要格外注意了，因为很可能循环结束后协程才开始执行，此时，所有协程使用的循环变量有可能已被改写。（是否会改写取决于引用循环变量的方式）

循环变量需要绑定

在题目一中，协程函数体中引用了循环变量 `task` ，协程从被创建到被调度执行期间循环变量极有可能被改写，这种情况下，我们称之为变量没有绑定。
所以，题目一打印结果是混乱的。很有可能（随机）所有协程执行的 `task` 都是列表中的最后一个task。

在题目二中，协程函数体中并没有直接引用循环变量 `task` ，而是使用的参数。而在创建协程时，循环变量 `task` 作为函数参数传递给了协程。参数传递的过程实际上也生成了新的变量，也即间接完成了绑定。
所以，题目二实际上是没有问题的。

在题目三中，测试用例名字 `test.name` 通过函数参数完成了绑定，而 `test.input 和 test.expectOutput` 则没有绑定。
然而题目三实际执行却不会有问题，因为 `t.Run(...)` 并不会启动新的协程，也就是循环体并没有并发。
此时，即便循环变量没有绑定也没有问题。
但是风险在于，如果 `t.Run(...)` 执行的测试体有可能并发（比如通过 `t.Parallel()` ），此时就极有可能引入问题。

对于题目三，建议显式地绑定，例如：

```
    for _, test := range tests {
        tc := test // 显式绑定，每次循环都会生成一个新的tc变量
        t.Run(tc.name, func(t *testing.T) {
            if tc.expectOutput != Double(tc.input) {
                t.Fatalf("expect: %d, but got: %d", tc.input, tc.expectOutput)
            }
        })
    }
```

通过 `tc := test` 显式地绑定，每次循环会生成一个新的变量。

总结

简单点来说

- 如果循环体没有并发出现，则引用循环变量一般不会出现问题；
- 如果循环体有并发，则根据引用循环变量的位置不同而有所区别
  - 通过参数完成绑定，则一般没有问题；
  - 函数体中引用，则需要显式地绑定

> 赠人玫瑰手留余香，如果觉得不错请给个赞~
> 你的鼓励将成为我继续写作的动力！
>
> 本篇文章已归档到GitHub项目，求星~ 点我即达