练识课堂 -- 资深Go语言工程师实践课程

第十七章 string 原理

> 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。

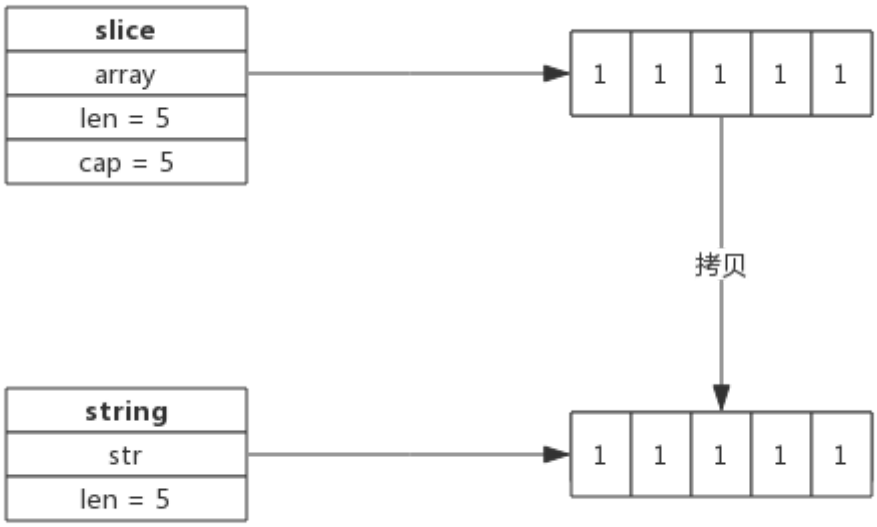
1. 1string操作

> #### 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跟据字符串构建stringStruct，再转换成string。转换的源码如下：
>
> ```
> // 跟据字符串地址构建string
> func gostringnocopy(str *byte) string {
>     // 先构造stringStruct
>     ss := stringStruct{str: unsafe.Pointer(str), len: findnull(str)}
>  // 再将stringStruct转换成string
>     s := *(*string)(unsafe.Pointer(&ss))
>     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. 跟据切片的长度申请内存空间，假设内存地址为p，切片长度为len(b)；
2. 构建string（string.str = p；string.len = len； ）
3. 拷贝数据(切片中数据拷贝到新申请的内存空间)

转换示意图：



string转[]byte

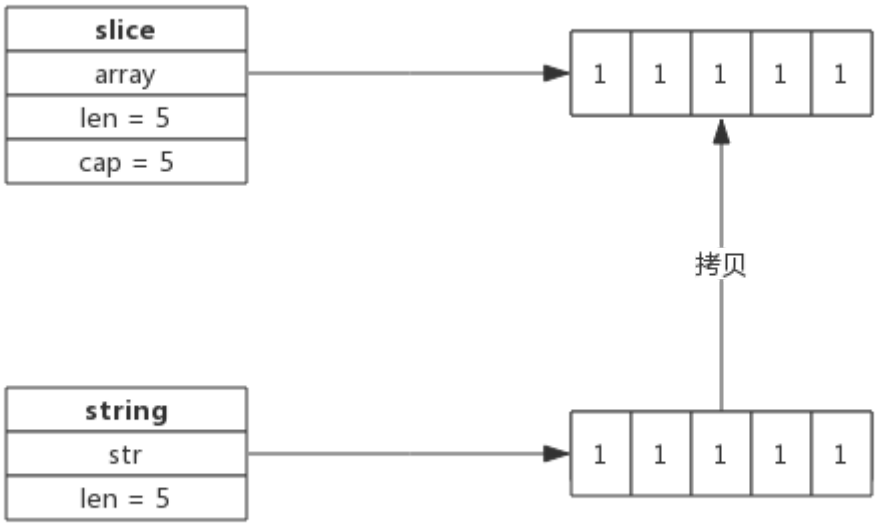
string也可以方便的转成byte切片，如下所示：

```
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：

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

```
// 生成一个新的string，返回的string和切片共享相同的空间
func rawstring(size int) (s string, b []byte) {
    p := mallocgc(uintptr(size), nil, false)
    stringStructOf(&s).str = p
    stringStructOf(&s).len = size
    *(*slice)(unsafe.Pointer(&b)) = slice{p, size, size}
    return
}
```

1.2 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。

1.3 类型转换的拷贝内存

> 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。

1.4 string和[]byte如何取舍

> string和[]byte都可以表示字符串，但因数据结构不同，其衍生出来的方法也不同，要跟据实际应用场景来选择。
>
> string 擅长的场景：
>
> - 需要字符串比较的场景；
> - 不需要nil字符串的场景；
>
> []byte擅长的场景：
>
> - 修改字符串的场景，尤其是修改粒度为1个字节；
> - 函数返回值，需要用nil表示含义的场景；
> - 需要切片操作的场景；
>
> 虽然看起来string适用的场景不如[]byte多，但因为string直观，在实际应用中还是大量存在，在偏底层的实现中[]byte使用更多。

第十八章 切片原理

1.1 切片的原理

> 切片 (slice) 是 Go中一种比较特殊的数据结构，这种数据结构更便于使用和管理数据集合。

切片是围绕动态数组的概念构建的，与数组相比切片的长度是不固定的，可以追加元素，在追加时可能使切片的容量增大。

但是很多同学对 slice 的模糊认识，造成认为Go中的切片作为函数参数是地址传递，结果就是在实际开发中碰到很多坑，以至于出现一些莫名奇妙的问题，数组中的数据丢失了。

```
func Demo(slice []int) {
    slice = append(slice, 6, 6, 6)
    fmt.Println("函数中结果：", slice)
}
func main() {
    //定义一个切片
    slice := []int{1, 2, 3, 4, 5}
    Demo(slice)
    fmt.Println("定义中结果：", slice)
}
```

结果

```
函数中结果：  [1 2 3 4 5 6 6 6]
定义中结果：  [1 2 3 4 5]
```

在上面代码中并没有将切片（slice）的修改结果在主调函数中显示。

因为Go语言所有函数参数都是值传递。

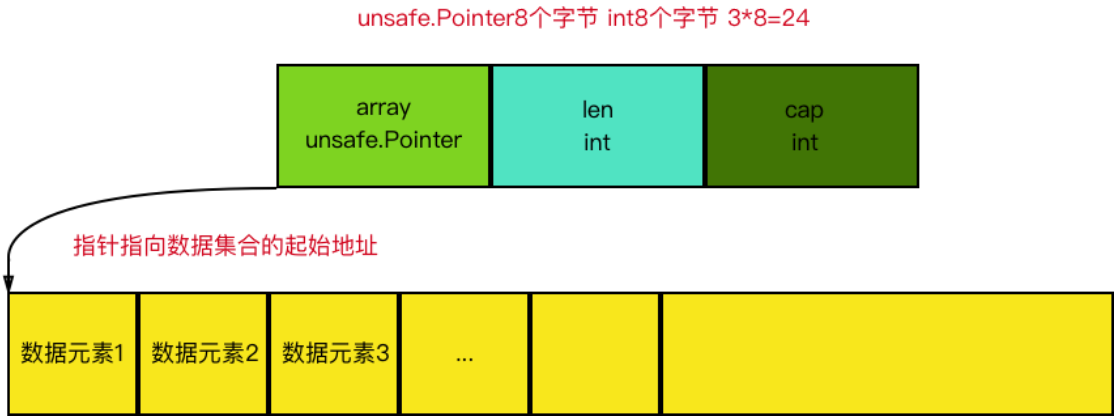
下面就开始详细理解下 slice，理解后会对开发出高效的程序非常有帮助。

```
//定义一个切片
var slice []int
//unsafe.Sizeof功能：计算一个数据在内存中占的字节大小
size := unsafe.Sizeof(slice)
fmt.Println(slice)
//无论切片是否有数据 计算结果都为：24
```

slice 的数据结构很简单，一个指向真实数据集合地址的指针ptr，slice 的长度 len 和容量 cap 。

Go语言中切片数据结构在源码包src的 /runtime/slice.go里面：

```
//path:Go SDK/src/runtime/slice.go
type slice struct {
    array unsafe.Pointer
    len   int
    cap   int
}
```



1.2 切片的源码

在创建切片时，可以根据源码进行分析

```
var slice []int = make([]int, 10, 20)
```

会被编译器翻译为 runtime.makeslice，并执行如下函数：

```
func makeslice(et *_type, len, cap int) slice {
    maxElements := maxSliceCap(et.size)
    //判断len是否满足条件
    if len < 0 || uintptr(len) > maxElements {
        panic(errorString("makeslice: len out of range"))
    }
    //判断cap是否满足条件
    if cap < len || uintptr(cap) > maxElements {
        panic(errorString("makeslice: cap out of range"))
    }
    //根据cap大小，通过mallocgc创建内存空间
    p := mallocgc(et.size*uintptr(cap), et, true)
    //将地址作为其中一个结构体成员返回
    return slice{p, len, cap}
}
```

如果创建切片是基于新建内存空间

```
var slice *[]int = new([]int)
```

会编译为：

```
func newobject(typ *_type) unsafe.Pointer {
//创建内存空间 并返回指针
return mallocgc(typ.size, typ, true)
}
```

第十九章 map 原理

1.1 map原理

go语言中的map是一种内建引用类型

> map存储时key不可重复，无顺序，排序的话可以将key排序，然后取出对应value。只有可以比较的类型才可以作key，value则无限制。
>
> go中的map采用的是哈希map
>
> 给定key后，会通过哈希算法计算一个哈希值，低B位(这里是大写的B，2^B表示当前map中bucket的数量)代表的是存在map中的哪一个bucket，高8位则是了存在bucket中的一个uint[8]数组中，高8位所在的数组indec可用来寻找对应key的index。
>
> map可抽象为bucket结构体组成的结构体，bucket数量一开始是固定的，后期不够用之后会进行扩容，bucket内部含有数组，bucket内部数组存储的即为key和value，为8组数据，存储方式为key0，key1，key2...value0，value1，value2...形式，而不是key和value顺次存储，这样是为了防止key和vaule长度不一致时需要额外padding。key和value存储在同一个数组中。

1.2 map的结构体

> **bucket的结构**
>
> ```
> type hmap struct {
> count     int //map中的元素个数，必须放在 struct的第一个位置，因为 内置的len函数会从这里读取
> flags     uint8
> B         uint8  // 说明包含2^B个bucket
> noverflow uint16 // 溢出的bucket的个数
> hash0     uint32 // hash种子
> buckets    unsafe.Pointer // buckets的数组指针
> oldbuckets unsafe.Pointer // 结构扩容的时候用于复制的buckets数组
> nevacuate  uintptr        // 搬迁进度（已经搬迁的buckets数量）
> extra *mapextra
> }
> ```
>
> **bucket的数据结构**
>
> ```
> type bmap struct {
> // tophash generally contains the top byte of the hash value
> // for each key in this bucket. If tophash[0] < minTopHash,
> // tophash[0] is a bucket evacuation state instead.
> tophash [bucketCnt]uint8    //tophash 是 hash 值的高 8 位
> // Followed by bucketCnt keys and then bucketCnt values.
> // NOTE: packing all the keys together and then all the values together makes the
> // code a bit more complicated than alternating key/value/key/value/... but it allows
> // us to eliminate padding which would be needed for, e.g., map[int64]int8.
> // Followed by an overflow pointer.
> }
> ```

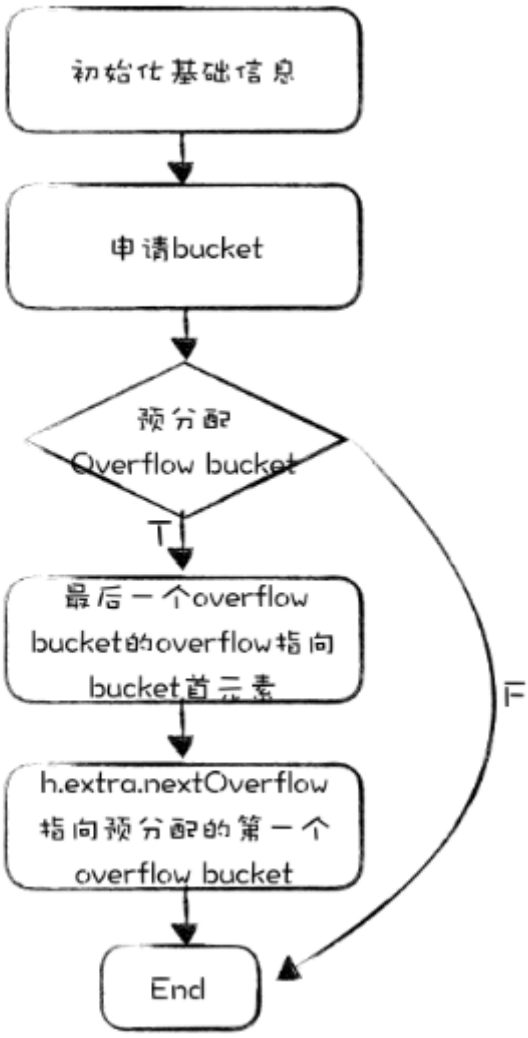
> 注意：bucket中不止有tophash数组，里面存储的是key通过hash函数算出来的哈希值的高8位。数组长度为8，对应了存储key和value的字节数组的index...注意后面几行注释，hmap并非只有一个tophash，而是后面紧跟8组kv对和一个overflow的指针，这样才能使overflow成为一个链表的结构。但是这两个结构体并不是显示定义的，而是直接通过指针运算进行访问的。
>
> kv的存储形式为"key0key1key2key3...key7val1val2val3...val7"，这样做的好处是：在key和value的长度不同的时候，节省padding空间。如上面的例子，在map[int64]int8中，4个相邻的int8可以存储在同一个内存单元中。如果使用kv交错存储的话，每个int8都会被padding占用单独的内存单元（为了提高寻址速度）。

1.3 map中key查找的流程

一个完整的map中key查找的流程:

- 根据传入的key用对应的hash函数算出哈希值
- 取哈希值的低B位定为到是哪一个bucket
- 定位到bucket之后，取哈希值的高8位，和bucket中的uint[8]数组中存储的高8位进行比对，完全匹配根据数组的index在key，value字节数组中查找对应的key，若匹配上则返回key，value，若没有完全匹配则继续比对高8位的值，当前bucket的高8位数组全部比对完，若有溢出bucket则继续比对，若全部比对完未发现则当前map不含有此key

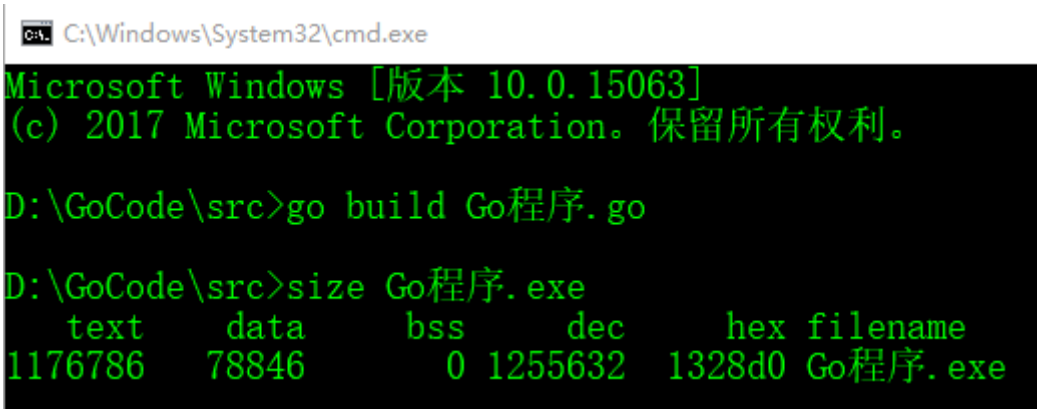
```
┌──────────┬──────────────────────────────────────────────────────────────────────┬──────
──────┐    |              |
|              |
|   10010111    | 0000111101101100100011111001010100010010110010101010 |
01010     |    |              |
|              |
└──────────┴──────────────────────────────────────────────────────────────────────┴──────
──────┘
```



第二十章 Go语言内存详解

1 内存分区

> 代码经过预处理、编译、汇编、链接4步后生成一个可执行程序。
>
> 在 Windows 下，程序是一个普通的可执行文件，以下列出一个二进制可执行文件的基本情况：



> 通过上图可以得知，在没有运行程序前，也就是说程序没有加载到内存前，可执行程序内部已经分好三段信息，分别为**代码区（text）**、**数据区（data）**和**未初始化数据区（bss）**3 个部分。
>
> 有些人直接把data和bss合起来叫做**静态区**或**全局区**。

1.1 代码区（text）

> 存放 CPU 执行的机器指令。通常代码区是可共享的（即另外的执行程序可以调用它），使其可**共享**的目的是对于频繁被执行的程序，只需要在内存中有一份代码即可。代码区通常是**只读**的，使其只读的原因是防止程序意外地修改了它的指令。另外，代码区还规划了局部变量的相关信息。

1.2 全局初始化数据区/静态数据区（data）

> 该区包含了在程序中明确被初始化的全局变量、已经初始化的静态变量（包括全局静态变量和局部静态变量）和常量数据（如字符串常量）。

1.3 未初始化数据区（bss）

> 存入的是全局未初始化变量和未初始化静态变量。未初始化数据区的数据在程序开始执行之前被内核初始化为 0 或者空（nil）。

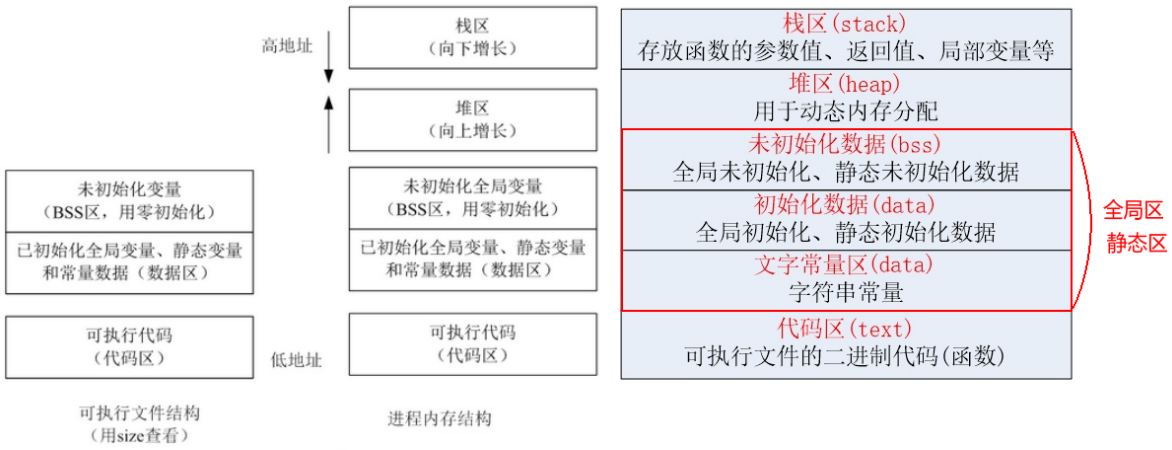
> 程序在加载到内存前，代码区和全局区(data和bss)的大小就是固定的，程序运行期间不能改变。
>
> 然后，运行可执行程序，系统把程序加载到内存，除了根据可执行程序的信息分出代码区（text）、数据区（data）和未初始化数据区（bss）之外，还额外增加了**栈区**、**堆区**。

1.4 栈区（stack）

> 栈是一种先进后出的内存结构，由编译器自动分配释放，存放函数的参数值、返回值、局部变量等。
>
> 在程序运行过程中实时加载和释放，因此，局部变量的生存周期为申请到释放该段栈空间。

1.5 堆区（heap）

> 堆是一个大容器，它的容量要远远大于栈，但没有栈那样先进后出的顺序。用于动态内存分配。堆在内存中位于BSS区和栈区之间。
>
> 根据语言的不同，如C语言、C++语言，一般由程序员分配和释放，若程序员不释放，程序结束时由操作系统回收。
>
> Go语言、Java、python等都有垃圾回收机制（GC），用来自动释放内存。



2 Go Runtime内存分配

> Go语言内置运行时（就是Runtime），抛弃了传统的内存分配方式，改为自主管理。这样可以自主地实现更好的内存使用模式，比如内存池、预分配等等。这样，不会每次内存分配都需要进行系统调用。

> Golang运行时的内存分配算法主要源自 Google 为 C 语言开发的**TCMalloc算法**，全称**Thread-Caching Malloc**。
>
> 核心思想就是把内存分为多级管理，从而降低锁的粒度。它将可用的堆内存采用二级分配的方式进行管理。
>
> 每个线程都会自行维护一个独立的内存池，进行内存分配时优先从该内存池中分配，当内存池不足时才会向全局内存池申请，以避免不同线程对全局内存池的频繁竞争。

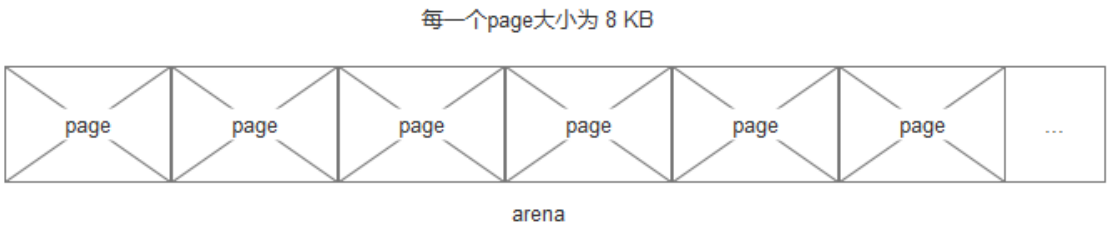
2.1 基本策略

> - 每次从操作系统申请一大块内存，以减少系统调用。
> - 将申请的大块内存按照特定的大小预先的进行切分成小块，构成链表。
> - 为对象分配内存时，只需从大小合适的链表提取一个小块即可。
> - 回收对象内存时，将该小块内存重新归还到原链表，以便复用。
> - 如果闲置内存过多，则尝试归还部分内存给操作系统，降低整体开销。
>
> **注意：**内存分配器只管理内存块，并不关心对象状态，而且不会主动回收，垃圾回收机制在完成清理操作后，触发内存分配器的回收操作

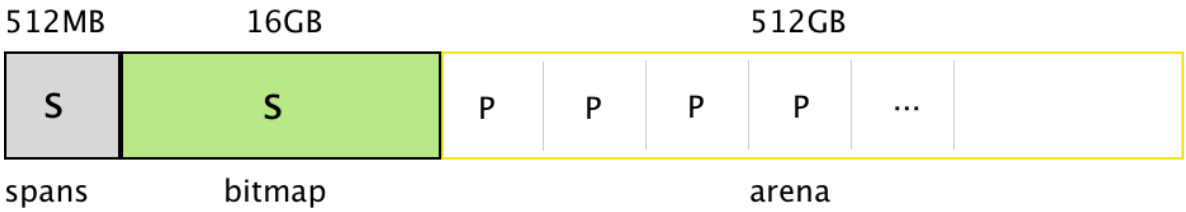
2.2 内存管理单元

> 分配器将其管理的内存块分为两种：
>
> - span：由多个连续的页（page [大小：8KB]）组成的大块内存。
> - object：将span按照特定大小切分成多个小块，每一个小块都可以存储对象。
>
> 用途：
>
> span 面向内部管理
>
> object 面向对象分配

```
//path:Go SDK/src/runtime/malloc.go


_PageShift      = 13
_PageSize = 1 << _PageShift     //8KB
```



> 在基本策略中讲到，Go在程序启动的时候，会先向操作系统申请一块内存，切成小块后自己进行管理。
>
> 申请到的内存块被分配了三个区域，在X64上分别是512MB，16GB，512GB大小。
>
> **注意：**这时还只是一段虚拟的地址空间，并不会真正地分配内存



> - arena区域
>
> 就是所谓的堆区，Go动态分配的内存都是在这个区域，它把内存分割成8KB大小的页，一些页组合起来称为mspan。
>
> ```
> //path:Go SDK/src/runtime/mheap.go
>
> type mspan struct {
>     next           *mspan     // 双向链表中 指向下一个
>     prev           *mspan     // 双向链表中 指向前一个
>     startAddr      uintptr    // 起始序号
>     npages         uintptr    // 管理的页数
>     manualFreeList gclinkptr  // 待分配的 object 链表
>      nelems            uintptr      // 块个数，表示有多少个块可供分配
>      allocCount     uint16     // 已分配块的个数
>     ...
> }
> ```
>
> - bitmap区域
>
> 标识arena区域哪些地址保存了对象，并且用4bit标志位表示对象是否包含指针、GC标记信息。
>
> - spans区域
>
> 存放mspan的指针，每个指针对应一页，所以spans区域的大小就是512GB/8KB*8B=512MB。
>
> 除以8KB是计算arena区域的页数，而最后乘以8是计算spans区域所有指针的大小。

2.3 内存管理组件

> 内存分配由内存分配器完成。分配器由3种组件构成：
>
> - cache
>
> 每个运行期工作线程都会绑定一个cache，用于无锁 object 的分配
>
> - central
>
> 为所有cache提供切分好的后备span资源

- heap

  管理闲置span，需要时向操作系统申请内存

2.3.1 cache

cache：每个工作线程都会绑定一个mcache，本地缓存可用的mspan资源。

这样就可以直接给Go Routine分配，因为不存在多个Go Routine竞争的情况，所以不会消耗锁资源。

mcache 的结构体定义：

```
//path:Go SDK/src/runtime/mcache.go

_NumSizeClasses = 67                    //67

var class_to_size = [_NumSizeClasses]uint16{0, 8, 16, 32, 48, 64, 80, 96,
112, 128, 144, 160, 176, 192, 208, 224, 240, 256, 288, 320, 352, 384, 416,
448, 480, 512, 576, 640, 704, 768, 896, 1024, 1152, 1280, 1408, 1536, 1792,
2048, 2304, 2688, 3072, 3200, 3456, 4096, 4864, 5376, 6144, 6528, 6784,
6912, 8192, 9472, 9728, 10240, 10880, 12288, 13568, 14336, 16384, 18432,
19072, 20480, 21760, 24576, 27264, 28672, 32768}
numSpanClasses = _NumSizeClasses << 1   //134

type mcache struct {
    alloc [numSpanClasses]*mspan        //以numSpanClasses 为索引管理多个用于分
配的 span
}
```

mcache用Span Classes作为索引管理多个用于分配的mspan，它包含所有规格的mspan。

它是 _NumSizeClasses 的2倍，也就是67*2=134，为什么有一个两倍的关系。

为了加速之后内存回收的速度，数组里一半的mspan中分配的对象不包含指针，另一半则包含指针。对于无指针对象的mspan在进行垃圾回收的时候无需进一步扫描它是否引用了其他活跃的对象。

2.3.2 central

central：为所有mcache提供切分好的mspan资源。

每个central保存一种特定大小的全局mspan列表，包括已分配出去的和未分配出去的。

每个mcentral对应一种mspan，而mspan的种类导致它分割的object大小不同。

```
//path:Go SDK/src/runtime/mcentral.go

type mcentral struct {
    lock      mutex          // 互斥锁
    sizeclass int32          // 规格
    nonempty  mSpanList      // 尚有空闲object的mspan链表
    empty     mSpanList      // 没有空闲object的mspan链表，或者是已被mcache取走的
msapn链表
    nmalloc   uint64         // 已累计分配的对象个数
}
```

2.3.3 heap

> heap：代表Go程序持有的所有堆空间，Go程序使用一个mheap的全局对象_mheap来管理堆内存。
>
> 当mcentral没有空闲的mspan时，会向mheap申请。而mheap没有资源时，会向操作系统申请新内存。mheap主要用于大对象的内存分配，以及管理未切割的mspan，用于给mcentral切割成小对象。
>
> 同时我们也看到，mheap中含有所有规格的mcentral，所以，当一个mcache从mcentral申请mspan时，只需要在独立的mcentral中使用锁，并不会影响申请其他规格的mspan。

```
//path:Go SDK/src/runtime/mheap.go
type mheap struct {
    lock        mutex
    spans       []*mspan // spans: 指向mspans区域，用于映射mspan和page的关系
    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
    arena_end   uintptr  // 指示arena区末地址
    central [numSpanClasses]struct {
        mcentral mcentral
        pad      [sys.CacheLineSize-
unsafe.Sizeof(mcentral{})%sys.CacheLineSize]byte
    }                    //每个 central 对应一种 sizeclass
}
```

2.4 分配流程

> - 计算待分配对象的规格（size_class）
> - 从cache.alloc数组中找到规格相同的span
> - 从span.manualFreeList链表提取可用object
> - 如果span.manualFreeList为空，从central获取新的span
> - 如果central.nonempty为空，从heap.free/freelarge获取，并切分成object链表
> - 如果heap没有大小合适的span，向操作系统申请新的内存

2.5 释放流程

> - 将标记为可回收的object交还给所属的span.freelist
> - 该span被放回central，可以提供cache重新获取
> - 如果span以全部回收object，将其交还给heap，以便重新分切复用
> - 定期扫描heap里闲置的span，释放其占用的内存
>
> 注意：以上流程不包含大对象，它直接从heap分配和释放

2.6 总结

> Go语言的内存分配非常复杂，它的一个原则就是能复用的一定要复用。
>
> - Go在程序启动时，会向操作系统申请一大块内存，之后自行管理。
> - Go内存管理的基本单元是mspan，它由若干个页组成，每种mspan可以分配特定大小的object。
> - mcache, mcentral, mheap是Go内存管理的三大组件，层层递进。mcache管理线程在本地缓存的mspan；mcentral管理全局的mspan供所有线程使用；mheap管理Go的所有动态分配内存。
> - 一般小对象通过mspan分配内存；大对象则直接由mheap分配内存。

3 Go GC垃圾回收

Garbage Collection (GC)是一种自动管理内存的方式。支持GC的语言无需手动管理内存，程序后台自动判断对象。是否存活并回收其内存空间，使开发人员从内存管理上解脱出来。

垃圾回收机制

- 引用计数
- 标记清除
- 三色标记
- 分代收集

1959年, GC由 John McCarthy发明, 用于简化Lisp中的手动内存管理，到现在很多语言都提供了GC，不过GC的原理和基本算法都没有太大的改变 。

```
//C语言开辟和释放空间
int* p = (int*)malloc(sizeof(int));
//如果不释放会造成内存泄露
free(p);
```

```
//Go语言开辟内存空间
//采用垃圾回收 不要手动释放内存空间
p := new(int)
```

3.1 Go GC发展

Golang早期版本GC可能问题比较多，但每一个版本的发布都伴随着 GC 的改进

- 1.5版本之后, Go的GC已经能满足大部分大部分生产环境使用要求
- 1.8通过混合写入屏障, 使得STW降到了sub ms。
  下面列出一些GC方面比较重大的改动

版本	发布时间	GC	STW时间
v 1.1	2013/5	STW	百ms-几百ms级别
v 1.3	2014/6	Mark STW, Sweep 并行	百ms级别
v 1.5	2015/8	三色标记法, 并发标记清除	10ms级别
v 1.8	2017/2	hybrid write barrier	sub ms

当前Go GC特征

三色标记， 并发标记和清扫，非分代，非紧缩，混合写屏障。

GC关心什么

程序吞吐量: 回收算法会在多大程度上拖慢程序? 可以通过GC占用的CPU与其他CPU时间的百分比描述

GC吞吐量: 在给定的CPU时间内，回收器可以回收多少垃圾?

堆内存开销: 回收器最少需要多少额外的内存开销?

停顿时间: 回收器会造成多大的停顿?

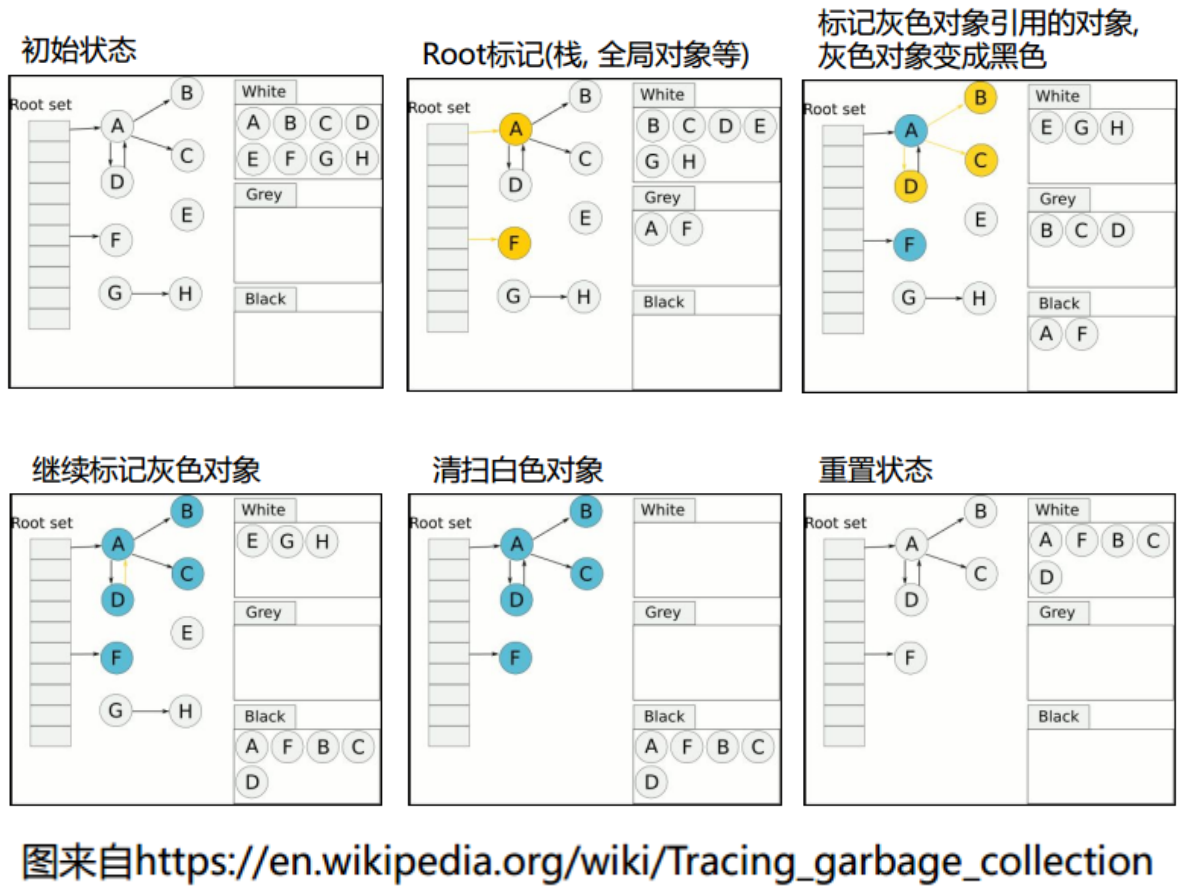
停顿频率: 回收器造成的停顿频率是怎样的?

停顿分布: 停顿有时候很长，有时候很短? 还是选择长一点但保持一致的停顿时间?

> **分配性能:** 新内存的分配是快，慢还是无法预测?
>
> **压缩:** 当堆内存里还有小块碎片化的内存可用时, 回收器是否仍然抛出内存不足（OOM）的错误？如果不是, 那么你是否发现程序越来越慢, 并最终死掉, 尽管仍然还有足够的内存可用？
>
> **并发:**回收器是如何利用多核机器的？
>
> **伸缩:**当堆内存变大时, 回收器该如何工作？
>
> **调优:**回收器的默认使用或在进行调优时, 它的配置有多复杂？
>
> **预热时间:**回收算法是否会根据已发生的行为进行自我调节？如果是, 需要多长时间？
>
> **页释放:**回收算法会把未使用的内存释放回给操作系统吗？如果会, 会在什么时候发生？

3.2 三色标记

> - 有黑白灰三个集合，初始时所有对象都是白色
> - 从Root对象开始标记, 将所有可达对象标记为灰色
> - 从灰色对象集合取出对象, 将其引用的对象标记为灰色, 放入灰色集合，并将自己标记为黑色
> - 重复第三步, 直到灰色集合为空, 即所有可达对象都被标记
> - 标记结束后, 不可达的白色对象即为垃圾. 对内存进行迭代清扫，回收白色对象
> - 重置GC状态



图来自https://en.wikipedia.org/wiki/Tracing_garbage_collection

3.2.1 写屏障

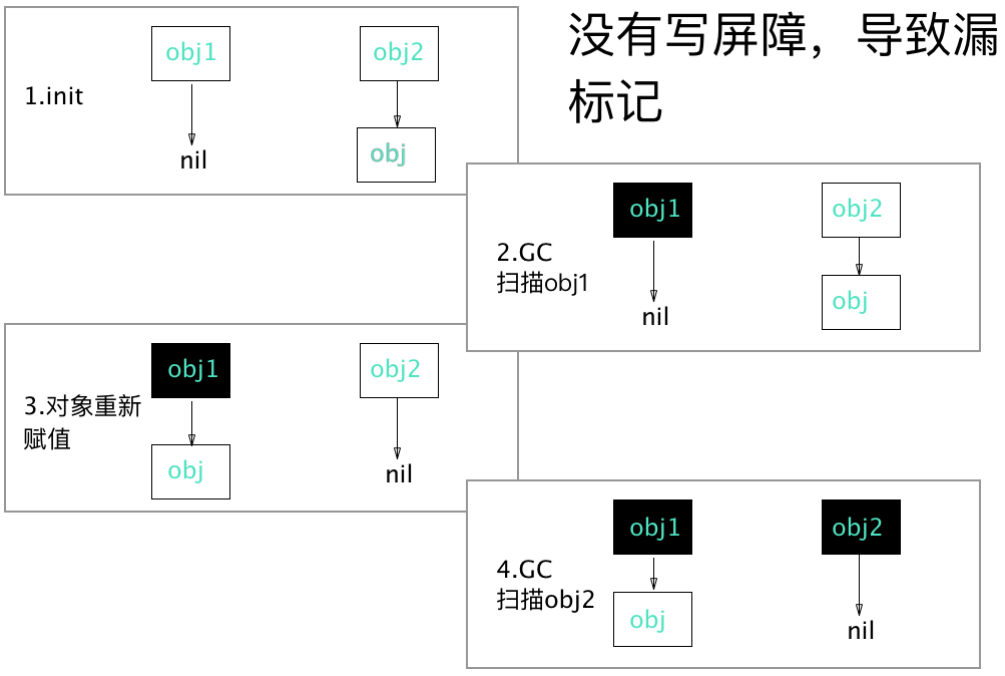
> 三色标记需要维护不变性条件：
>
> 黑色对象不能引用无法被灰色对象可达的白色对象。
>
> 并发标记时, 如果没有做正确性保障措施，可能会导致漏标记对象，导致实际上可达的对象被清扫掉。

为了解决这个问题，go使用了写屏障。

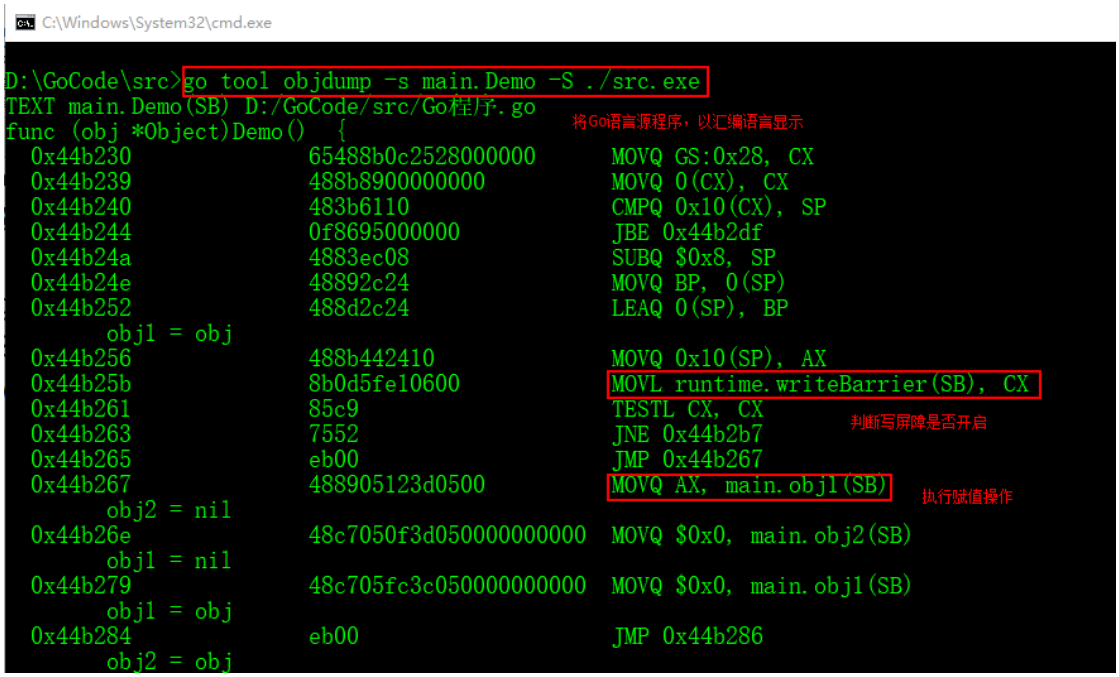
写屏障是在写入指针前执行的一小段代码，用以防止并发标记时指针丢失，这段代码Go是在编译时加入的。

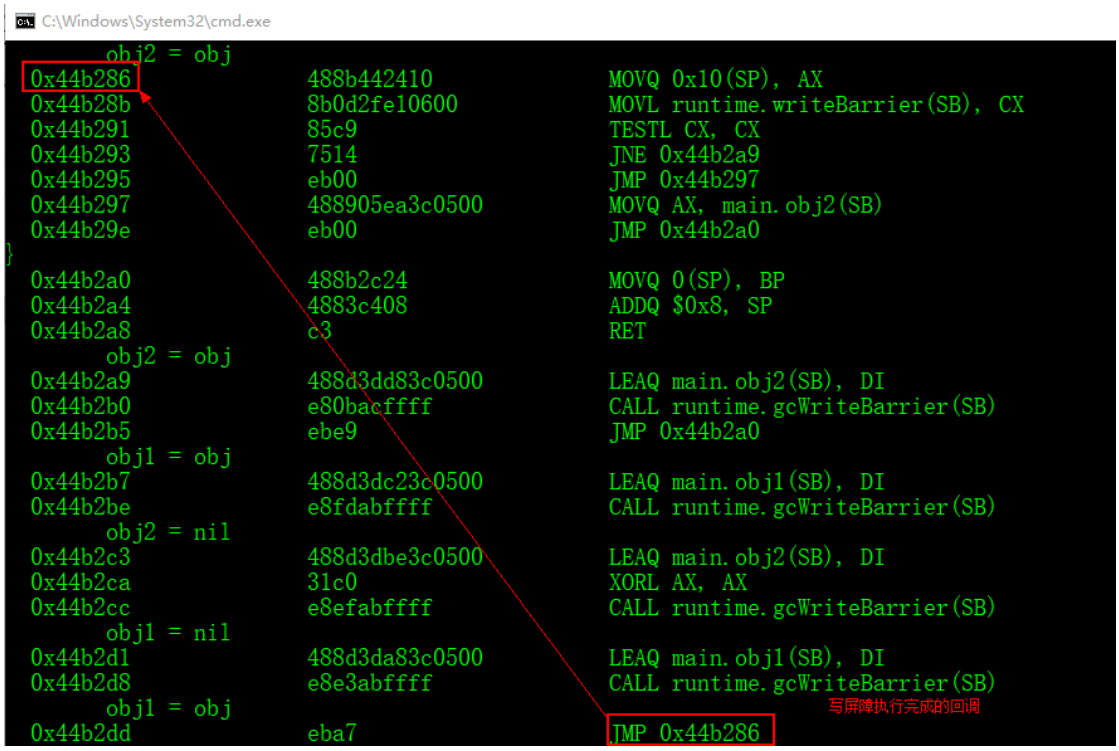
Golang写屏障在mark和mark termination阶段处于开启状态。

```
var obj1 *Object
var obj2 *Object

type Object struct {
    data interface{}
}

func (obj *Object) Demo() {
    //初始化

    obj1 = nil
    obj2 = obj
    //gc 垃圾回收开始工作
    //扫描对象 obj1 完成后

    //代码修改为：对象重新赋值
    obj1 = obj
    obj2 = nil

    //扫描对象 obj2

}
```




```
#将Go语言程序显示为汇编语言
go build -gcflags "-N -l"
go tool objdump -s 'main.Demo' -S ./Go程序.exe
```





3.2.2 三色状态

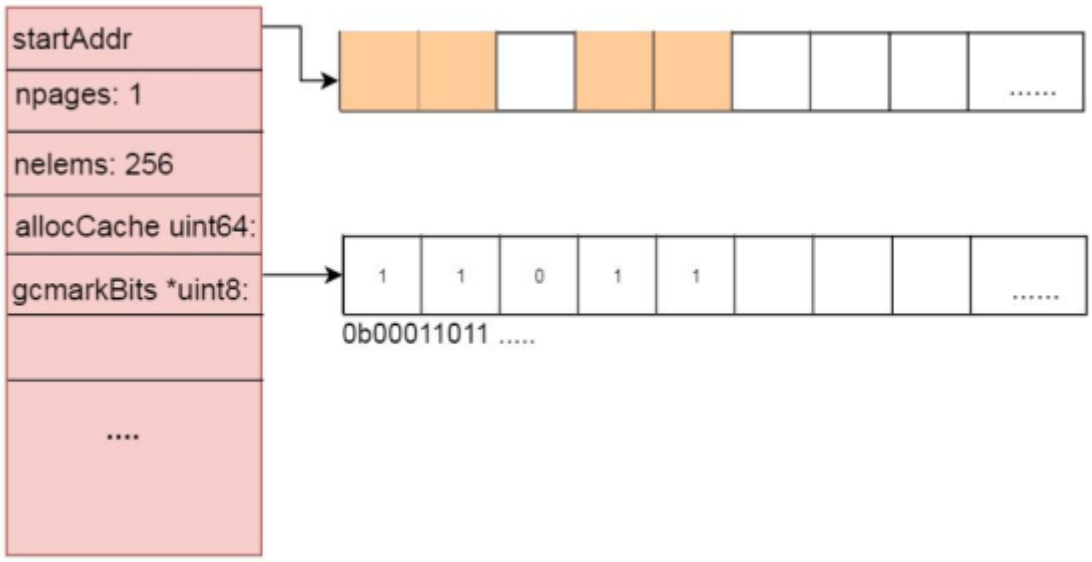
并没有真正的三个集合来分别装三色对象。

前面分析内存的时候, 介绍了go的对象是分配在span中, span里还有一个字段是gcmarkBits, mark阶段里面每个bit代表一个slot已被标记.。

白色对象该bit为0, 灰色或黑色为1. (runtime.markBits)

每个p中都有wbBuf和gcw gcWork，以及全局的workbuf标记队列，实现生产者-消费者模型, 在这些队列中的指针为灰色对象，表示已标记，待扫描。

从队列中出来并把其引用对象入队的为黑色对象, 表示已标记，已扫描(runtime.scanobject)。



3.3 GC执行流程

> **GC 触发**
>
> - gcTriggerHeap：
>
>   分配内存时， 当前已分配内存与上一次GC结束时存活对象的内存达到某个比例时就触发GC。
>
> - gcTriggerTime：
>
>   sysmon检测2min内是否运行过GC，没运行过 则执行GC。
>
> - gcTriggerAlways：
>
>   runtime.GC()强制触发GC。

3.3.1 启动

> 在为对象分配内存后，mallocgc函数会检查垃圾回收触发条件，并按照相关状态启动。
>
> ```
> //path:Go SDK/src/runtime/malloc.go
> func mallocgc(size uintptr, typ *_type, needzero bool) unsafe.Pointer
> ```
>
> 垃圾回收默认是全并发模式运行，GC goroutine 一直循环执行，直到符合触发条件时被唤醒。

3.3.2 标记

> 并发标记分为两个步骤：
>
> - 扫描：遍历相关内存区域，依次按照指针标记找出灰色可达对象，加入队列。
>
> ```
> //path:Go SDK/src/runtime/mgcmark.go
> //扫描和对比bitmap区域信息找出合法指针，将其目标当作灰色可达对象添加到待处理队列
>
> func markroot(gcw *gcWork, i uint32)
> func scanblock(b0, n0 uintptr, ptrmask *uint8, gcw *gcWork)
> ```
>
> - 标记：将灰色对象从队列取出，将其引用对象标记为灰色，自身标记为黑色。

```
//path:Go SDK/src/runtime/mgc.go
func gcBgMarkStartWorkers()
```

3.3.3 清理

> 清理的操作很简单，所有未标记的白色对象不再被引用，可以将其内存回收。

```
//path:Go SDK/src/runtime/mgcsweep.go

//并发清理本质就是一个死循环，被唤醒后开始执行清理任务，完成内存回收操作后，再次休眠，等待下次执行任务
var sweep sweepdata

// 并发清理状态
type sweepdata struct {
    lock    mutex
    g       *g
    parked  bool
    started bool

    nbgsweep    uint32
    npausesweep uint32
}
func bgsweep(c chan int)
func sweepone() uintptr
```

第二十一章 goroutine原理

1 goroutine原理

1.1 概念介绍

> **并发**
>
> 一个CPU上能同时执行多项任务，在很短时间内，CPU来回切换任务执行(在某段很短时间内执行程序a，然后又迅速得切换到程序b去执行)，有时间上的重叠（宏观上是同时的，微观仍是顺序执行），这样看起来多个任务像是同时执行，这就是并发。
>
> **并行**
>
> 当系统有多个CPU时,每个CPU同一时刻都运行任务，互不抢占自己所在的CPU资源，同时进行，称为并行。
>
> **进程**
>
> CPU在切换程序的时候，如果不保存上一个程序的状态（context--上下文），直接切换下一个程序，就会丢失上一个程序的一系列状态，于是引入了进程这个概念，用以划分好程序运行时所需要的资源。因此进程就是一个程序运行时候的所需要的基本资源单位（也可以说是程序运行的一个实体）。
>
> **线程**
>
> CPU切换多个进程的时候，会花费不少的时间，因为切换进程需要切换到内核态，而每次调度需要内核态都需要读取用户态的数据，进程一旦多起来，CPU调度会消耗一大堆资源，因此引入了线程的概念，线程本身几乎不占有资源，他们共享进程里的资源，内核调度起来不会那么像进程切换那么耗费资源。

协程

协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈。因此，协程能保留上一次调用时的状态（即所有局部状态的一个特定组合），每次过程重入时，就相当于进入上一次调用的状态，换种说法：进入上一次离开时所处逻辑流的位置。线程和进程的操作是由程序触发系统接口，最后的执行者是系统；协程的操作执行者则是用户自身程序，goroutine也是协程。

1.2 Go并发模式

生产者消费者模型

- 该模式主要通过平衡生产线程和消费线程的工作能力来提高程序的整体处理数据的速度。
- 生产者生产一些数据，然后放到成果队列中，同时消费者从成果队列中来取这些数据。这样就让生产消费变成了异步的两个过程。
- 当成果队列中没有数据时，消费者就进入饥饿的等待中；而当成果队列中数据已满时，生产者则面临因产品挤压导致CPU被剥夺的下岗问题。

```go
package main

import (
    "fmt"
    "time"
)
// 生产者: 生成 factor 整数倍的序列
func Producer(factor int, out chan<- int) {
    for i := 0; ; i++ {
        out <- i * factor
    }
}

// 消费者
func Consumer(in <-chan int) {
    for v := range in {
        fmt.Println(v)
    }
}
func main() {
    ch := make(chan int, 64) // 成果队列

    go Producer(3, ch) // 生成 3 的倍数的序列
    go Producer(5, ch) // 生成 5 的倍数的序列
    go Consumer(ch)    // 消费 生成的队列

    // 运行一定时间后退出
    time.Sleep(5 * time.Second)
}
```

发布订阅模型

- 发布订阅（publish-and-subscribe）模型通常被简写为pub/sub模型。在这个模型中，消息生产者成为发布者（publisher），而消息消费者则成为订阅者（subscriber），生产者和消费者是M:N的关系。
- 在传统生产者和消费者模型中，是将消息发送到一个队列中，而发布订阅模型则是将消息发布给一个主题。

为此，构建了一个名为pubsub的发布订阅模型支持包：

```
package pubsub

import (
    "sync"
    "time"
)

type (
    subscriber chan interface{}         // 订阅者为一个管道
    topicFunc  func(v interface{}) bool // 主题为一个过滤器
)

// 发布者对象
type Publisher struct {
    m           sync.RWMutex             // 读写锁
    buffer      int                      // 订阅队列的缓存大小
    timeout     time.Duration            // 发布超时时间
    subscribers map[subscriber]topicFunc // 订阅者信息
}

// 构建一个发布者对象，可以设置发布超时时间和缓存队列的长度
func NewPublisher(publishTimeout time.Duration, buffer int) *Publisher {
    return &Publisher{
        buffer:      buffer,
        timeout:     publishTimeout,
        subscribers: make(map[subscriber]topicFunc),
    }
}

// 添加一个新的订阅者，订阅全部主题
func (p *Publisher) Subscribe() chan interface{} {
    return p.SubscribeTopic(nil)
}

// 添加一个新的订阅者，订阅过滤器筛选后的主题
func (p *Publisher) SubscribeTopic(topic topicFunc) chan interface{} {
    ch := make(chan interface{}, p.buffer)
    p.m.Lock()
    p.subscribers[ch] = topic
    p.m.Unlock()
    return ch
}

// 退出订阅
func (p *Publisher) Evict(sub chan interface{}) {
    p.m.Lock()
    defer p.m.Unlock()

    delete(p.subscribers, sub)
    close(sub)
}

// 发布一个主题
func (p *Publisher) Publish(v interface{}) {
    p.m.RLock()
```

```
    defer p.m.RUnlock()

    var wg sync.WaitGroup
    for sub, topic := range p.subscribers {
        wg.Add(1)
        go p.sendTopic(sub, topic, v, &wg)
    }
    wg.Wait()
}

// 关闭发布者对象，同时关闭所有的订阅者管道。
func (p *Publisher) Close() {
    p.m.Lock()
    defer p.m.Unlock()

    for sub := range p.subscribers {
        delete(p.subscribers, sub)
        close(sub)
    }
}

// 发送主题，可以容忍一定的超时
func (p *Publisher) sendTopic(
    sub subscriber, topic topicFunc, v interface{}, wg *sync.WaitGroup,
) {
    defer wg.Done()
    if topic != nil && !topic(v) {
        return
    }

    select {
    case sub <- v:
    case <-time.After(p.timeout):
    }
}
```

两个订阅者分别订阅了全部主题和含有"golang"的主题：

```
  package main

  import "pubsub"

  func main() {
      p := pubsub.NewPublisher(100*time.Millisecond, 10)
      defer p.Close()

      all := p.Subscribe()
      golang := p.SubscribeTopic(func(v interface{}) bool {
          if s, ok := v.(string); ok {
              return strings.Contains(s, "golang")
          }
          return false
      })

      p.Publish("hello,  world!")
      p.Publish("hello, golang!")
```

```
    go func() {
        for  msg := range all {
            fmt.Println("all:", msg)
        }
    } ()

    go func() {
        for  msg := range golang {
            fmt.Println("golang:", msg)
        }
    } ()

    // 运行一定时间后退出
    time.Sleep(3 * time.Second)
}
```

1.3 Go并发模型

Go语言的并发处理参考了CSP（Communicating Sequential Process 通讯顺序进程）模型。

CSP并发模型是Hoare在1978年提出的CSP的概念，不同于传统的多线程通过共享内存来通信，CSP有着精确的数学模型，并实际应用在了Hoare参与设计的T9000通用计算机上。

CSP讲究的是“以通信的方式来共享内存”。

```
Don't communicate by sharing memory; instead, share memory by
communicating.

不要通过共享内存来通信，而应通过通信来共享内存。
```

Go的CSP模型实现与原始的CSP实现有点差别：原始的CSP中channel里的任务都是立即执行的，而go语言为其增加了一个缓存，即任务可以先暂存起来，等待执行线程准备好再顺序执行。

Go的CSP并发模型，是通过goroutine和channel来实现的。

- goroutine 是Go语言中并发的执行单位。有点抽象，其实就是和传统概念上的”线程“类似，可以理解为”线程“。
- channel是Go语言中各个并发结构体(goroutine)之前的通信机制。 通俗的讲，就是各个goroutine之间通信的”管道“，有点类似于Linux中的管道。

生成一个goroutine的方式非常的简单：Go一下，就生成了。

```
go f()
```

通信机制channel也很方便，传数据用channel <- data，取数据用<-channel。

在通信过程中，传数据channel <- data和取数据<-channel必然会成对出现，因为这边传，那边取，两个goroutine之间才会实现通信。

而且不管传还是取，必阻塞，直到另外的goroutine传或者取为止。

1.4 Go调度器GMP

Go语言运行时环境提供了非常强大的管理goroutine和系统内核线程的调度器， 内部提供了三种对象：Goroutine，Machine，Processor。

Goroutine：指应用创建的goroutine

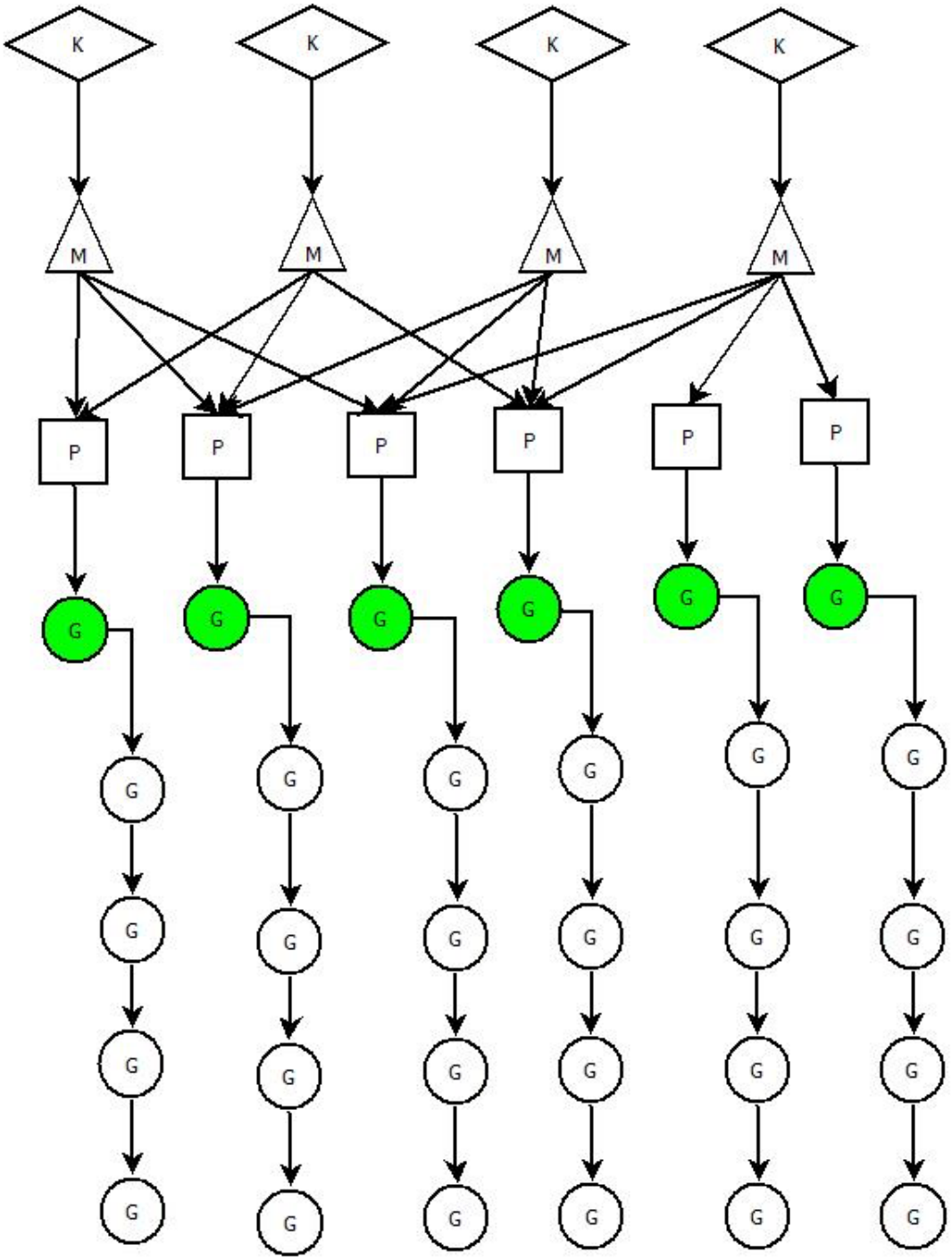
Machine : 指系统内核线程。

Processor : 指承载多个goroutine的运行器

在宏观上说，Goroutine与Machine因为Processor的存在，形成了多对多（M:N）的关系。M个用户线程对应N个系统线程，缺点增加了调度器的实现难度

Goroutine是Go语言中并发的执行单位。 Goroutine底层是使用协程(coroutine)实现，coroutine是一种运行在用户态的用户线程（参考操作系统原理：内核态，用户态）它可以由语言和框架层调度。Go在语言层面实现了调度器，同时对网络，IO库进行了封装处理，屏蔽了操作系统层面的复杂的细节，在语言层面提供统一的关键字支持。

三者与内核级线程的关系如下图所示：



一个Machine会对应一个内核线程（K），同时会有一个Processor与它绑定。一个Processor连接一个或者多个Goroutine。Processor有一个运行时的Goroutine（上图中绿色的G），其它的Goroutine处于等待状态。

> Processor的数量同时可以并发任务的数量，可通过GOMAXPROCS限制同时执行用户级任务的操作系统线程。GOMAXPROCS值默认是CPU的可用核心数，但是其数量是可以指定的。在go语言运行时环境，可以使用
>
> ```
> runtime.GOMAXPROCS(MaxProcs)
> ```
>
> 来指定Processor数量。
>
> - 当一个Goroutine创建被创建时，Goroutine对象被压入Processor的本地队列或者Go运行时全局Goroutine队列。
> - Processor唤醒一个Machine，如果Machine的waiting队列没有等待被 唤醒的Machine，则创建一个（只要不超过Machine的最大值，10000），Processor获取到Machine后，与此Machine绑定，并执行此Goroutine。
>
>   ```
>   func schedinit() {
>       //设置最大的M数量
>       sched.maxmcount = 10000
>   }
>   ```
>
> - Machine执行过程中，随时会发生上下文切换。当发生上下文切换时，需要对执行现场进行保护，以便下次被调度执行时进行现场恢复。Go调度器中Machine的栈保存在Goroutine对象上，只需要将Machine所需要的寄存器(堆栈指针、程序计数器等)保存到Goroutine对象上即可。
> - 如果此时Goroutine任务还没有执行完，Machine可以将Goroutine重新压入Processor的队列，等待下一次被调度执行。
> - 如果执行过程遇到阻塞并阻塞超时，Machine会与Processor分离，并等待阻塞结束。此时Processor可以继续唤醒Machine执行其它的Goroutine，当阻塞结束时，Machine会尝试”偷取”一个Processor，如果失败，这个Goroutine会被加入到全局队列中，然后Machine将自己转入Waiting队列，等待被再次唤醒。

2 channel原理

2.1 channel数据结构

> channel一个类型管道，通过它可以在goroutine之间发送和接收消息。它是Golang在语言层面提供的goroutine间的通信方式。
>
> 众所周知，Go依赖于称为CSP（Communicating Sequential Processes）的并发模型，通过channel实现这种同步模式。
>
> 通过channel来实现通信：
>
> ```
> package main
> import (
>     "fmt"
>     "time"
> )
> func goRoutineA(a <-chan int) {
>     val := <-a
>     fmt.Println("goRoutineA:", val)
> }
> func goRoutineB(b chan int) {
>     val := <-b
>     fmt.Println("goRoutineB:", val)
> ```

```
}
func main() {
    ch := make(chan int, 3)
    go goRoutineA(ch)
    go goRoutineB(ch)
    ch <- 3
    time.Sleep(time.Second)
}
```

channel结构体：

```
//path: src/runtime/chan.go
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。

2.2 channel实现方式

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]。

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。

channel读写

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。

```
func makechan(t *chantype, size int) *hchan {
    elem := t.elem

    // compiler checks this but be safe.
    if elem.size >= 1<<16 {
        throw("makechan: invalid channel element type")
    }
    if hchanSize%maxAlign != 0 || elem.align > maxAlign {
        throw("makechan: bad alignment")
    }

    mem, overflow := math.MulUintptr(elem.size, uintptr(size))
    if overflow || mem > maxAlloc-hchanSize || size < 0 {
        panic(plainError("makechan: size out of range"))
    }

    // Hchan does not contain pointers interesting for GC when elements
stored in buf do not contain pointers.
    // buf points into the same allocation, elemtype is persistent.
    // SudoG's are referenced from their owning thread so they can't be
collected.
    // TODO(dvyukov,rlh): Rethink when collector can move allocated
objects.
    var c *hchan
    switch {
    case mem == 0:
        // Queue or element size is zero.
        c = (*hchan)(mallocgc(hchanSize, nil, true))
        // Race detector uses this location for synchronization.
        c.buf = c.raceaddr()
    case elem.ptrdata == 0:
        // Elements do not contain pointers.
        // Allocate hchan and buf in one call.
        c = (*hchan)(mallocgc(hchanSize+mem, nil, true))
        c.buf = add(unsafe.Pointer(c), hchanSize)
    default:
        // Elements contain pointers.
        c = new(hchan)
        c.buf = mallocgc(mem, elem, true)
    }

    c.elemsize = uint16(elem.size)
    c.elemtype = elem
    c.dataqsiz = uint(size)

    if debugChan {
        print("makechan: chan=", c, "; elemsize=", elem.size, "; elemalg=",
elem.alg, "; dataqsiz=", size, "\n")
    }
    return c
}
```

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：
 img

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：
 img

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。

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

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

    if raceenabled {
        callerpc := getcallerpc()
        racewritepc(c.raceaddr(), callerpc, funcPC(closechan))
        racerelease(c.raceaddr())
    }

    c.closed = 1

    var glist gList

    // 释放所有接收者
    for {
        sg := c.recvq.dequeue()
        if sg == nil {
            break
```

```
            }
            if sg.elem != nil {
                typedmemclr(c.elemtype, sg.elem)
                sg.elem = nil
            }
            if sg.releasetime != 0 {
                sg.releasetime = cputicks()
            }
            gp := sg.g
            gp.param = nil
            if raceenabled {
                raceacquireg(gp, c.raceaddr())
            }
            glist.push(gp)
        }

        // 释放所有发送者
        for {
            sg := c.sendq.dequeue()
            if sg == nil {
                break
            }
            sg.elem = nil
            if sg.releasetime != 0 {
                sg.releasetime = cputicks()
            }
            gp := sg.g
            gp.param = nil
            if raceenabled {
                raceacquireg(gp, c.raceaddr())
            }
            glist.push(gp)
        }
        unlock(&c.lock)

        // 垃圾回收
        for !glist.empty() {
            gp := glist.pop()
            gp.schedlink = 0
            goready(gp, 3)
        }
    }
```

除此之外，panic出现的常见场景还有：

- 关闭值为nil的channel
- 关闭已经被关闭的channel
- 向已经关闭的channel写数据