Go

零、go 与其他语言

0. 什么是面向对象

在了解 Go 语言是不是面向对象（简称：OOP）之前，我们必须先知道 OOP 是啥，得先给他“下定义”
根据 Wikipedia 的定义，我们梳理出 OOP 的几个基本认知：

- 面向对象编程（OOP）是一种基于“对象”概念的编程范式，它可以包含数据和代码：数据以字段的形式存在（通常称为属性或属性），代码以程序的形式存在（通常称为方法）。
- 对象自己的程序可以访问并经常修改自己的数据字段。
- 对象经常被定义为类的一个实例。
- 对象利用属性和方法的私有/受保护/公共可见性，对象的内部状态受到保护，不受外界影响（被封装）。

基于这几个基本认知进行一步延伸出，面向对象的三大基本特性：

- 封装
- 继承
- 多态

1. Go 语言和 Java 有什么区别?

1. Go 上不允许函数重载，必须具有方法和函数的唯一名称，而 Java 允许函数重载。
2. 在速度方面，Go 的速度要比 Java 快。
3. Java 默认允许多态，而 Go 没有。
4. Go 语言使用 HTTP 协议进行路由配置，而 Java 使用 Akka.routing.ConsistentHashingRouter 和 Akka.routing.ScatterGatherFirstCompletedRouter 进行路由配置。
5. Go 代码可以自动扩展到多个核心，而 Java 并不总是具有足够的可扩展性。
6. Go 语言的继承通过匿名组合完成，基类以 Struct 的方式定义，子类只需要把基类作为成员放在子类的定义中，支持多继承;而 Java 的继承通过 extends 关键字完成，不支持多继承。

2. Go 是面向对象的语言吗?

是的，也不是。原因是：

1. Go 有类型和方法，并且允许面向对象的编程风格，但没有类型层次。

2. Go 中的 "接口" 概念提供了一种不同的方法，我们认为这种方法易于使用，而且在某些方面更加通用。还有一些方法可以将类型嵌入到其他类型中，以提供类似的东西，但不等同于子类。
3. Go 中的方法比 C++ 或 Java 中的方法更通用：它们可以为任何类型的数据定义，甚至是内置类型，如普通的、"未装箱的 "整数。它们并不局限于结构（类）。
4. Go 由于缺乏类型层次，Go 中的 "对象 "比 C++ 或 Java 等语言更轻巧。

3. Go 实现面向对象编程

封装

面向对象中的“封装”指的是可以隐藏对象的内部属性和实现细节，仅对外提供公开接口调用，这样用户就不需要关注你内部是怎么实现的。
在 Go 语言中的属性访问权限，通过首字母大小写来控制：

- 首字母大写，代表是公共的、可被外部访问的。
- 首字母小写，代表是私有的，不可以被外部访问。

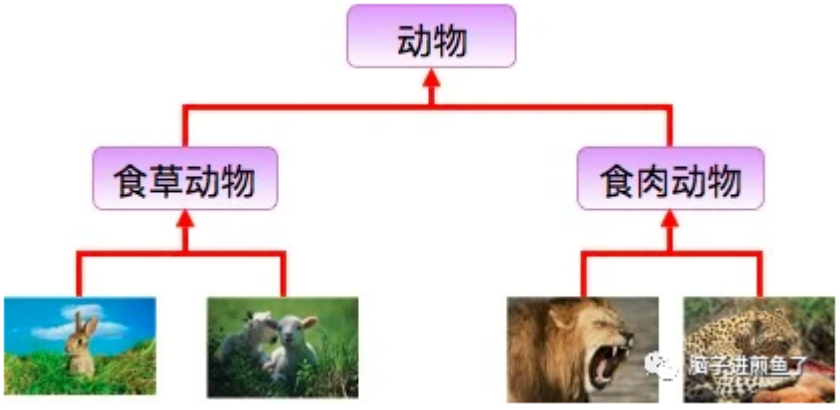
Go 语言的例子如下：

```
type Animal struct {
    name string
}

func NewAnimal() *Animal {
     return &Animal{}
}

func (p *Animal) SetName(name string) {
     p.name = name
}

func (p *Animal) GetName() string {
     return p.name
}
```

在上述例子中，我们声明了一个结构体 Animal，其属性 name 为小写。没法通过外部方法，在配套上存在 Setter 和 Getter 的方法，用于统一的访问和设置控制。
以此实现在 Go 语言中的基本封装。

继承

面向对象中的“继承”指的是子类继承父类的特征和行为，使得子类对象（实例）具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为。



从实际的例子来看，就是动物是一个大父类，下面又能细分为“食草动物”、“食肉动物”，这两者会包含“动物”这个父类的基本定义。

从实际的例子来看，就是动物是一个大父类，下面又能细分为“食草动物”、“食肉动物”，这两者会包含“动物”这个父类的基本定义。

在 Go 语言中，是没有类似 extends 关键字的这种继承的方式，在语言设计上采取的是组合的方式：

```
type Animal struct {
     Name string
}

type Cat struct {
     Animal              // 只有父类类型名，没有变量名
     FeatureA string
}

type Dog struct {
     Animal
     FeatureB string
}

```

在上述例子中，我们声明了 Cat 和 Dog 结构体，其在内部匿名组合了 Animal 结构体。因此 Cat 和 Dog 的实例都可以调用 Animal 结构体的方法：

```
func main() {
     p := NewAnimal()
     p.SetName("我是搬运工，去给煎鱼点赞~")

     dog := Dog{Animal: *p}
     fmt.Println(dog.GetName())
}
```

同时 Cat 和 Dog 的实例可以拥有自己的方法：

```
func (dog *Dog) HelloWorld() {
     fmt.Println("脑子进煎鱼了")
}

func (cat *Cat) HelloWorld() {
     fmt.Println("煎鱼进脑子了")
}
```

上述例子能够正常包含调用 Animal 的相关属性和方法，也能够拥有自己的独立属性和方法，在 Go 语言中达到了类似继承的效果。

多态

多态
面向对象中的“多态”指的同一个行为具有多种不同表现形式或形态的能力，具体是指一个类实例（对象）的相同方法在不同情形有不同表现形式。
多态也使得不同内部结构的对象可以共享相同的外部接口，也就是都是一套外部模板，内部实际是什么，只要符合规格就可以。
在 Go 语言中，多态是通过接口来实现的：

```
type AnimalSounder interface {
     MakeDNA()
}

func MakeSomeDNA(animalSounder AnimalSounder) {     // 参数是 AnimalSounder 接口类型
     animalSounder.MakeDNA()
}
```

在上述例子中，我们声明了一个接口类型 AnimalSounder，配套一个 MakeSomeDNA 方法，其接受 AnimalSounder 接口类型作为入参。
因此在 Go 语言中。只要配套的 Cat 和 Dog 的实例也实现了 MakeSomeDNA 方法，那么我们就可以认为他是 AnimalSounder 接口类型：

```
type AnimalSounder interface {
     MakeDNA()
}

func MakeSomeDNA(animalSounder AnimalSounder) {
     animalSounder.MakeDNA()
}

func (c *Cat) MakeDNA() {                // 注意它的调用者在函数名的前面，这样才能在不同类型的同一个方法上实现多态
     fmt.Println("煎鱼是煎鱼")
}
```

```
12
13  func (c *Dog) MakeDNA() {                    // 它仍然是不能重载的
14      fmt.Println("煎鱼其实不是煎鱼")
15  }
16
17  func main() {
18      MakeSomeDNA(&Cat{})
19      MakeSomeDNA(&Dog{})
20  }
21
```

当 Cat 和 Dog 的实例实现了 AnimalSounder 接口类型的约束后，就意味着满足了条件，他们在 Go 语言中就是一个东西。能够作为入参传入 MakeSomeDNA 方法中，再根据不同的实例实现多态行为。

在日常工作中，基本了解这些概念就可以了。若是面试，可以针对三大特性：“封装、继承、多态”和 五大原则“单一职责原则（SRP）、开放封闭原则（OCP）、里氏替换原则（LSP）、依赖倒置原则（DIP）、接口隔离原则（ISP）”进行深入理解和说明。

4. go 语言和 python 的区别：

1. 范例
   Python 是一种基于面向对象编程的多范式，命令式和函数式编程语言。它坚持这样一种观点，即如果一种语言在某些情境中表现出某种特定的方式，理想情况下它应该在所有情境中都有相似的作用。但是，它又不是纯粹的 OOP 语言，它不支持强封装，这是 OOP 的主要原则之一。
   Go 是一种基于并发编程范式的过程编程语言，它与 C 具有表面相似性。实际上，Go 更像是 C 的更新版本。
2. 类型化
   Python 是动态类型语言，变量类型随赋值动态改变，而 Go 是一种静态类型语言，它实际上有助于在编译时捕获错误，这可以进一步减少生产后期的严重错误。
3. 并发
   Python 没有提供内置的并发机制，而 Go 有内置的并发机制。
4. 安全性
   Python 是一种强类型语言，它是经过编译的，因此增加了一层安全性。Go 具有分配给每个变量的类型，因此，它提供了安全性。但是，如果发生任何错误，用户需要自己运行整个代码。
5. 管理内存
   Go 允许程序员在很大程度上管理内存。而，Python 中的内存管理完全自动化并由 Python VM 管理；它不允许程序员对内存管理负责。
6. 库
   与 Go 相比，Python 提供的库数量要大得多。然而，Go 仍然是新的，并且还没有取得很大进展。

7. 语法
   Python 的语法使用缩进来指示代码块。Go 的语法基于打开和关闭括号。

8. 详细程度
   为了获得相同的功能，Golang 代码通常需要编写比 Python 代码更多的字符。

5. go 与 node.js

深入对比 Node.js 和 Golang 到底谁才是 NO.1：
https://zhuanlan.zhihu.com/p/421352168
从 Node 到 Go：一个粗略的比较 : https://zhuanlan.zhihu.com/p/29847628

一、基础部分

0. 为什么选择 golang

0. 高性能 - 协程
   golang 源码级别支持协程，实现简单；对比进程和线程，协程占用资源少，能够简洁高效地处理高并发问题。

1. 学习曲线容易 - 代码极简
   Go 语言语法简单，包含了类 C 语法。因为 Go 语言容易学习，所以一个普通的大学生花几个星期就能写出来可以上手的、高性能的应用。在国内大家都追求快，这也是为什么国内 Go 流行的原因之一。
   Go 语言的语法特性简直是太简单了，简单到你几乎玩不出什么花招，直来直去的，学习曲线很低，上手非常快。

2. 效率：快速的编译时间，开发效率和运行效率高
   开发过程中相较于 Java 和 C++ 呆滞的编译速度，Go 的快速编译时间是一个主要的效率优势。Go 拥有接近 C 的运行效率和接近 PHP 的开发效率。
   C 语言的理念是信任程序员，保持语言的小巧，不屏蔽底层且底层友好，关注语言的执行效率和性能。而 Python 的姿态是用尽量少的代码完成尽量多的事。于是我能够感觉到，Go 语言想要把 C 和 Python 统一起来，这是多棒的一件事啊。

3. 出身名门、血统纯正
   之所以说 Go 出身名门，从 Go 语言的创造者就可见端倪，Go 语言绝对血统纯正。
   其次 Go 语言出自 Google 公司，Google 在业界的知名度和实力自然不用多说。
   Google 公司聚集了一批牛人，在各种编程语言称雄争霸的局面下推出新的编程语言，自然有它的战略考虑。而且从 Go 语言的发展态势来看，Google 对它这个新的宠儿还是很看重的，Go 自然有一个良好的发展前途。

4. 自由高效：组合的思想、无侵入式的接口
   Go 语言可以说是开发效率和运行效率二者的完美融合，天生的并发编程支持。Go 语言支持当前所有的编程范式，包括过程式编程、面向对象编程、面向接口编程、函数式编程。程序员们可以各取所需、自由组合、想怎么玩就怎么玩。

5. 强大的标准库 - 生态
   背靠谷歌，生态丰富，轻松 go get 获取各种高质量轮子。用户可以专注于业务逻辑，避免重复造轮子。
   这包括互联网应用、系统编程和网络编程。Go 里面的标准库基本上已经是非常稳定

了，特别是我这里提到的三个，网络层、系统层的库非常实用。Go 语言的 lib 库麻雀虽小五脏俱全。Go 语言的 lib 库中基本上有绝大多数常用的库，虽然有些库还不是很好，但我觉得不是问题，因为我相信在未来的发展中会把这些问题解决掉。

6. 部署方便：二进制文件，Copy 部署
   部署简单，源码编译成执行文件后，可以直接运行，减少了对其它插件依赖。不像其它语言，执行文件依赖各种插件，各种库，研发机器运行正常，部署到生产环境，死活跑不起来。

7. 简单的并发
   并行和异步编程几乎无痛点。Go 语言的 Goroutine 和 Channel 这两个神器简直就是并发和异步编程的巨大福音。像 C、C++、Java、Python 和 JavaScript 这些语言的并发和异步方式太控制就比较复杂了，而且容易出错，而 Go 解决这个问题非常地优雅和流畅。这对于编程多年受尽并发和异步折磨的编程者来说，完全就是让人眼前一亮的感觉。Go 是一种非常高效的语言，高度支持并发性。Go 是为大数据、微服务、并发而生的一种编程语言。
   Go 作为一门语言致力于使事情简单化。它并未引入很多新概念，而是聚焦于打造一门简单的语言，它使用起来异常快速并且简单。其唯一的创新之处是 goroutines 和通道。Goroutines 是 Go 面向线程的轻量级方法，而通道是 goroutines 之间通信的优先方式。
   创建 Goroutines 的成本很低，只需几千个字节的额外内存，正由于此，才使得同时运行数百个甚至数千个 goroutines 成为可能。可以借助通道实现 goroutines 之间的通信。Goroutines 以及基于通道的并发性方法使其非常容易使用所有可用的 CPU 内核，并处理并发的 IO。相较于 Python/Java，在一个 goroutine 上运行一个函数需要最小的代码。

8. 稳定性
   Go 拥有强大的编译检查、严格的编码规范和完整的软件生命周期工具，具有很强的稳定性，稳定压倒一切。那么为什么 Go 相比于其他程序会更稳定呢？这是因为 Go 提供了软件生命周期（开发、测试、部署、维护等等）的各个环节的工具，如 go tool、gofmt、go test。

9. 跨平台
   很多语言都支持跨平台，把这个优点单独拿出来，貌似没有什么值得称道的，但是结合上述优点，它的综合能力就非常强了。

Go 调试

1. 输出调试：基本打印调试，日志调试
2. 使用 Delve 作为调试器
   - 打断点（在行号左侧点一下）
   - F5 启动调试
   - 使用 Watch、Call Stack、变量面板
3. 单元测试调试
   - 用测试代码包裹你要调试的函数

golang 缺点

1. 左大括号不允许换行，否则编译报错。这是因为 go 的自动分号机制。每个语句不需要写分号，是因为编译器会自动加分号。如果左大括号单占一行，就会有 `if xxx;`（左括号后不会自动分号，但前面那行没有左括号的话就会自动分号），这是会报错的。右大括号和 else 要在同一行，形成 `} else {`，因为右大括号如果单独占一行，后面会加上分号 `};`，而 else 会另起一行作为新的语句，这会报错。
2. 不允许有未使用的包或变量
3. 错误处理原始，虽然引入了 defer、panic、recover 处理出错后的逻辑，函数可以返回多个值，但基本依靠返回错误是否为空来判断函数是否执行成功，if err != nil 语句较多，比较繁琐，程序没有 java 美观。(官方解释：提供了多个返回值，处理错误方便，如加入异常机制会要求记住一些常见异常，例如 IOException，go 的错误 Error 类型较统一方便)
4. []interface{}不支持下标操作
5. struct 没有构造和析构，一些资源申请和释放动作不太方便
6. 仍然保留 C/C++ 的指针操作，取地址&，取值*
7. 没有隐式转换，必须显式转换。`y typeT := typeT(x)`

除此之外，还有一些需要记忆的奇怪的地方。

1. 不支持函数重载。现有的解决方案，最好的是使用 interface{}。然后在函数里判断类型做相应的处理。其次可以使用泛型，但不能多次使用不同类型调用，因为在泛型单态化时，两个类型会违背不能重载的特性。
2. 数组是值拷贝，而切片是引用。将某个数组赋给另一个变量，会执行拷贝而不是引用传递。生成切片是引用而不是拷贝。对比 rust：这种赋值在 rust 中会所有权转移，原变量会失效。而生成切片在 rust 中是引用，这与 go 一样。
3. 可以定义无类型常量，在使用时确定类型。`const x = 123`
4. string 是浅拷贝，因为数据不可变，只会拷贝胖指针。

1. golang 中 make 和 new 的区别？（基本必问）

共同点：给变量分配内存

不同点：

1. 作用变量类型不同，new 给 string,int 和数组分配内存，make 给切片 slice，map，channel 分配内存；因为这三种类型：不只是分配内存，还需要初始化底层结构

   slice：

   ```
   s := make([]int, 5)
   fmt.Println(s) // [0 0 0 0 0]
   ```

   map：

```go
m := make(map[string]int)
m["hello"] = 1
```

channel：

```go
ch := make(chan int, 10)
```

new

```go
p := new(int)
fmt.Println(*p) // 0
*p = 10
fmt.Println(*p) // 10
```

make (channel)

```go
ch := make(chan int, 2)
ch <- 42
fmt.Println(<-ch) // 42
```

2. 返回类型不一样，

   new 返回指针→ 你必须用 * 解引用。

   make 返回值本身→ 不需要指针。

3. new 分配的空间被清零。make 分配空间后，会进行初始化；

4. 字节的面试官还说了另外一个区别，就是分配的位置，在堆上还是在栈上？golang 会弱化分配的位置的概念，因为编译的时候会自动内存逃逸分析。这个变量是不是只在函数里用？如果是 → 分配在栈上；如果要返回给外面 → 分配在堆上。

Go 不写 malloc/free，全靠垃圾回收。

2. IO 多路复用

单线程或单进程同时监测若干个文件描述符是否可以执行 IO 操作的能力。就是让一个线程同时监听多个 IO（文件描述符），一旦哪个 IO 就绪，就去处理谁，避免阻塞在单个 IO 上。
目的：

- 提高并发
- 降低线程数量

常用于：

- 网络服务器
- 高并发场景

阻塞 IO

这是最常用的简单的 IO 模型。阻塞 IO 意味着当我们发起一次 IO 操作后一直等待成功或失败之后才返回，在这期间程序不能做其它的事情。阻塞 IO 操作只能对单个文件描述符进行操作，详见 read 或 write。

非阻塞 IO

我们在发起 IO 时，通过对文件描述符设置 O_NONBLOCK flag 来指定该文件描述符的 IO 操作为非阻塞。非阻塞 IO 通常发生在一个 for 循环当中，因为每次进行 IO 操作时要么 IO 操作成功，要么当 IO 操作会阻塞时返回错误 EWOULDBLOCK/EAGAIN，然后再根据需要进行下一次的 for 循环操作，这种类似轮询的方式会浪费很多不必要的 CPU 资源，是一种糟糕的设计。和阻塞 IO 一样，非阻塞 IO 也是通过调用 read 或 write 来进行操作的，也只能对单个描述符进行操作。

IO 多路复用

IO 多路复用在 Linux 下包括了三种，select、poll、epoll，抽象来看，他们功能是类似的，但具体细节各有不同：首先都会对一组文件描述符进行相关事件的注册，然后阻塞等待某些事件的发生或等待超时。更多细节详见下面的 "具体怎么用"。IO 多路复用都可以关注多个文件描述符，但对于这三种机制而言，不同数量级文件描述符对性能的影响是不同的。

信号驱动 IO

信号驱动 IO 是利用信号机制，让内核告知应用程序文件描述符的相关事件。但信号驱动 IO 在网络编程的时候通常很少用到，因为在网络环境中，和 socket 相关的读写事件太多了。下面这些事件都会产生 sigio 信号。

- TCP 连接建立
- 一方断开 TCP 连接请求
- 断开 TCP 连接请求完成
- TCP 连接半关闭
- 数据到达 TCP socket
- 数据已经发送出去 (如：写 buffer 有空余空间)

上面所有的这些都会产生 SIGIO 信号，但我们**无法**在 SIGIO 对应的信号处理函数中**区分**上述**不同的事件**，SIGIO 只应该在 IO 事件单一情况下使用，比如说用来监听端口的 socket，因为只有客户端发起新连接的时候才会产生 SIGIO 信号。

异步 IO

异步 IO 和信号驱动 IO 差不多，但它比信号驱动 IO 可以多做一步：相比信号驱动 IO 需要在程序中完成数据从用户态到内核态 (或反方向) 的拷贝，异步 IO 可以把拷贝这一步也帮我们完成之后才通知应用程序。我们使用 aio_read 来读，aio_write 写。

信号驱动 IO 会在内核检测到 I/O 就绪时向进程发送信号（如 SIGIO），这涉及一次内核态到用户态的切换。但它只是通知，数据还要应用程序手动调用 read/write 来完成，因此仍然需要额外的用户态到内核态切换。而异步 IO（AIO）则是内核完成所有操作并通知用户，真正做到不阻塞、不干预。

> 同步 IO vs 异步 IO 1. 同步 IO 指的是程序会一直阻塞到 IO 操作如 read、write 完成 2. 异步 IO 指的是 IO 操作不会阻塞当前程序的继续执行
> 所以根据这个定义，上面阻塞 IO 当然算是同步的 IO，非阻塞 IO 也是同步 IO，因为当文件操作符可用时我们还是需要阻塞的读或写，同理 IO 多路复用和信号驱动 IO 也是同步 IO，只有异步 IO 是完全完成了数据的拷贝之后才通知程序进行处理，没有阻塞的数据读写过程。

解决方案总览

- Linux: select, poll, epoll
- MacOS/FreeBSD: kqueue
- Windows/Solaris: IOCP

常见软件的 IO 多路复用方案

- redis: Linux 下 epoll(level-triggered)，没有 epoll 用 select
- nginx: Linux 下 epoll(edge-triggered)，没有 epoll 用 select

1. select

原理

- 把所有 fd 放到一个数组里 `file descriptor`
- 调用 select 系统调用
- 等待 IO 就绪

优点

- 最早期方案
- 跨平台兼容好（Linux、Windows 都支持）

缺点

- fd 数量有限制（如 1024）
- 每次都要把 fd 集合拷进内核
- 返回后要遍历所有 fd → O(n)
- 不适合大并发

2. poll

原理

- 与 select 类似
- 用 pollfd 数组描述 fd
- 不再限制最大 fd 值

优点

- 没有 fd 上限

- 接口更灵活

缺点

- 本质还是遍历全部 fd → O(n)
- 每次都拷贝 fd 列表
- 不适合大并发

3. epoll（Linux 特有）

原理

- 提供事件驱动机制
- 内核保存 fd 集合
- 只返回活跃的 fd

优点

- 支持超大并发
- 内核保存 fd，不用重复传
- 返回就绪 fd 列表，不用遍历所有 fd → O(活跃连接数)
- 支持边缘触发（Edge Trigger），更高效

缺点

- 仅支持 Linux
- 实现比 select/poll 复杂

特性	**select**	**poll**	**epoll**
fd 上限	有（一般 1024）	无	无
每次传 fd 列表	是	是	否
返回全部或活跃 fd	全部	全部	仅活跃
时间复杂度	O(n)	O(n)	O(活跃数)
支持边缘触发	否	否	是
平台	跨平台	跨平台	Linux 专用

poll vs select

poll 和 select 基本上是一样的，poll 相比 select 好在如下几点：

- poll 传参对用户更友好。比如不需要和 select 一样计算很多奇怪的参数比如 nfds(值最大的文件描述符 +1)，再比如不需要分开三组传入参数。

- poll 会比 select 性能稍好些，因为 select 是每个 bit 位都检测，假设有个值为 1000 的文件描述符，select 会从第一位开始检测一直到第 1000 个 bit 位。但 poll 检测的是一个数组。
- select 的时间参数在返回的时候各个系统的处理方式不统一，如果希望程序可移植性更好，需要每次调用 select 都初始化时间参数。

而 select 比 poll 好在下面几点

- 支持 select 的系统更多，兼容更强大，有一些 unix 系统不支持 poll
- select 提供精度更高 (到 microsecond) 的超时时间，而 poll 只提供到毫秒的精度。

但总体而言 select 和 poll 基本一致。

epoll vs poll&select

epoll 优于 select&poll 在下面几点：

- 在需要同时监听的文件描述符数量增加时，select&poll 是 O(N) 的复杂度，epoll 是 O(1)，在 N 很小的情况下，差距不会特别大，但如果 N 很大的前提下，一次 O(N) 的循环可要比 O(1) 慢很多，所以高性能的网络服务器都会选择 epoll 进行 IO 多路复用。
- epoll 内部用一个文件描述符挂载需要监听的文件描述符，这个 epoll 的文件描述符可以在多个线程/进程共享，所以 epoll 的使用场景要比 select&poll 要多。

总结：IO 多路复用，就是用一个线程同时监听多个 IO。常见方式有 select、poll、epoll。select 和 poll 每次都要遍历所有 fd，不适合高并发；epoll 则由内核保存 fd 集合，只返回活跃 fd，性能更好。

3. for range 的时候它的地址会发生变化么?

答：在 for key, value := range c 遍历中，key 和 value 在内存中只会存在一份，即之后每次循环时遍历到的数据都是以值覆盖的方式赋给 key 和 value，key value 的内存地址始终不变。由于有这个特性，for 循环里面如果开协程，不要直接把 key 或者 value 的地址传给协程。解决办法：在每次循环时，创建一个临时变量。

4. go defer，多个 defer 的顺序，defer 在什么时机会修改返回值?

Golang 中的 Defer 必掌握的 7 知识点 - 地鼠文档

作用：defer 延迟函数，释放资源，收尾工作；如释放锁，关闭文件，关闭链接；捕获 panic;

避坑指南：defer 函数紧跟在资源打开后面，否则 defer 可能得不到执行，导致内存泄露。

多个 defer 调用顺序是 LIFO（后入先出），defer 后的操作可以理解为压入栈中

defer、return、返回值三者的执行逻辑应该是：

1. return最先执行，return负责将结果写入返回值中；

2. 接着defer开始执行一些收尾工作；

3. 最后函数携带**当前返回值**（可能和最初的返回值不相同）退出。

参考：
【Golang】Go 语言 defer 用法大总结 (含 return 返回机制)_奶酪的博客-CSDN 博客_
blog.csdn.net/Cassie_zkq/article/details/108567205

有名返回值

```
func b() (i int) {
    defer func() {
        i++
        fmt.Println("defer2:", i)
    }()
    defer func() {
        i++
        fmt.Println("defer1:", i)
    }()
    return i
}
func main() {
    fmt.Println("return:", b())
}
```

输出

```
defer1: 1
defer2: 2
return: 0
```

返回值由变量i赋值，相当于返回值=i=0。第二个defer中i++ = 1， 第一个defer中i++ = 2，所以最终i的值是2。但是返回值已经被赋值了，即使后续修改i也不会影响返回值。

函数返回指针

```
func c() *int {
    var i int
    defer func() {
        i++
        fmt.Println("defer2:", i)
    }()
    defer func() {
        i++
        fmt.Println("defer1:", i)
    }()
    return &i
}
func main() {
    fmt.Println("return:", *(c()))
```

```
15 }
```

这里defer就能修改返回值。

5. uint 类型溢出问题

超过最大存储值如 uint8 最大是 255

var a uint8 =255

var b uint8 =1

a+b = 0 总之类型溢出会出现难以意料的事

整型的类型

类 型	有无符号	占用存储空间	表数范围	备注
int	有	32位系统4个字节 64位系统8个字节	$-2^{31} \sim 2^{31}-1$ $-2^{63} \sim 2^{63}-1$	
uint	无	32位系统4个字节 64位系统8个字节	$0 \sim 2^{32}-1$ $0 \sim 2^{64}-1$	
rune	有	与int32一样	$-2^{31} \sim 2^{31}-1$	等价 int32，表示一个 Unicode 码
byte	无	与 uint8 等价	$0 \sim 255$	当要存储字符时使用b

6. 能介绍下 rune 类型吗?

在 Go 中，`rune` 是 `int32` 的别名，表示一个 **Unicode 码点（code point）**。`rune` 可以表示一个完整的 Unicode 字符（无论占几个字节）。

1. byte 等同于 int8，常用来处理 ascii 字符
2. rune 等同于 int32，常用来处理 unicode 或 utf-8 字符
3. 单引号里包单个字符 `'A'` 是 rune。双引号里包字符串 `"string"` 是 string 类型。使用 rune 数组来存储多个 Unicode 字符 `[]rune("你好")`。

```go
package main
import (
    "fmt"
    "unicode/utf8"
)
func main() {
    var str = "hello 你好"
    //golang中string底层是通过byte数组实现的，直接求len 实际是在按字节长度计算  所以一个汉字占3个字节算了3个长度
    fmt.Println( a...: "len(str):", len(str))
    //以下两种都可以得到str的字符串长度
    //golang中的unicode/utf8包提供了用utf-8获取长度的方法
    fmt.Println( a...: "RuneCountInString:", utf8.RuneCountInString(str))
    //通过rune类型处理unicode字符
    fmt.Println( a...: "rune:", len([]rune(str)))
}
```

```
<4 go setup calls>
len(str): 12
RuneCountInString: 8
rune: 8
```


7. golang 中解析 tag 是怎么实现的？反射原理是什么？(中高级肯定会问，比较难，需要自己多去总结)

参考如下连接

golang 中 struct 关于反射 tag_paladinosment 的博客-CSDN 博客_golang 反射 tag
blog.csdn.net/paladinosment/article/details/42570937

```go
type User struct {
    name string `json:name-field`
    age  int
}
func main() {
    user := &User{"John Doe The Fourth", 20}
    field, ok := reflect.TypeOf(user).Elem().FieldByName("name")
    if !ok {
        panic("Field not found")
    }
    fmt.Println(getStructTag(field))
}
func getStructTag(f reflect.StructField) string {
    return string(f.Tag)
}
```

Tag 是一种**给字段附加的"注释"信息**，用于告诉其他系统（如数据库、JSON 序列化器、校验器）："这个字段在你那里叫什么/怎么处理"。

Go 中解析的 tag 是通过反射实现的，反射是指计算机程序在运行时（Run time）可以访问、检测和修改它本身状态或行为的一种能力或动态知道给定数据对象的类型和结构，并有机会修改它。反射将接口变量转换成反射对象 Type 和 Value；反射可以通过反射对象 Value 还原成原先的接口变量；反射可以用来修改一个变量的值，前提是这个值可以被修改；tag 是啥：结构

体支持标记，name string `json:name-field` 就是 `json:name-field` 这部分。
gorm json yaml gRPC protobuf gin.Bind() 都是通过反射来实现的

8. 调用函数传入结构体时，应该传值还是指针？（Golang 都是传值）

Go 的函数参数传递都是值传递。所谓值传递：指在调用函数时将实际参数复制一份传递到函数中，这样在函数中如果对参数进行修改，将不会影响到实际参数。参数传递还有引用传递，所谓引用传递是指在调用函数时将实际参数的地址传递到函数中，那么在函数中对参数所进行的修改，将影响到实际参数。

因为 Go 里面的 map，slice，chan 是引用类型。变量区分值类型和引用类型。所谓值类型：变量和变量的值存在同一个位置。所谓引用类型：变量和变量的值是不同的位置，变量的值存储的是对值的引用（指针）。

9. goroutine 什么情况下会阻塞

在 Go 里面阻塞主要分为以下 4 种场景：

1. **锁。**由于 **原子、互斥量或通道操作调用** 导致 Goroutine 阻塞，调度器将把当前阻塞的 Goroutine 切换出去，重新调度 LRQ 上的其他 Goroutine。这也包括 channel 阻塞。
2. **外。**由于 **网络请求和 IO 操作** 导致 Goroutine 阻塞。Go 程序提供了网络轮询器（NetPoller）来处理网络请求和 IO 操作的问题，其后台通过 kqueue（MacOS），epoll（Linux）或 iocp（Windows）来实现 IO 多路复用。通过**使用 NetPoller 进行网络系统调用**，调度器可以防止 Goroutine 在进行这些系统调用时阻塞 M。这可以让 M 执行 P 的 LRQ 中其他的 Goroutines，而不需要创建新的 M。执行网络系统调用不需要额外的 M，**网络轮询器使用系统线程**，它时刻处理一个有效的事件循环，有助于减少操作系统上的调度负载。用户层眼中看到的 Goroutine 中的“block socket”，实现了 goroutine-per-connection 简单的网络编程模式。实际上是通过 Go runtime 中的 netpoller 通过 Non-block socket + I/O 多路复用机制“模拟”出来的。
3. **内。**当调用一些系统方法的时候（如文件 I/O），如果**系统方法调用的时候发生阻塞**，这种情况下，网络轮询器（NetPoller）无法使用，而进行系统调用的 G1 将阻塞当前 M1。调度器引入 其它 M 来服务 M1 的 P。
4. **睡。**如果在 Goroutine 去执行一个**sleep 操作**，导致 M 被阻塞了。Go 程序后台有一个监控线程 sysmon，它监控那些长时间运行的 G 任务然后设置可以强占的标识符，别的 Goroutine 就可以抢先进来执行。

10. 讲讲 Go 的 select 底层数据结构和一些特性？（难点，没有项目经常可能说不清，面试一般会问你项目中怎么使用 select)

答：go 的 select 为 golang 提供了多路 IO 复用机制，和其他 IO 复用一样，用于检测是否有读写事件是否 ready。linux 的系统 IO 模型有 select，poll，epoll，go 的 select 和 linux 系统 select 非常相似。

Linux 的 select

- 是系统调用
- 用 fd 集合检测哪些 IO 就绪

Go 的 select

- 是语言语法
- 用于 channel 的多路选择
- 底层不等于 Linux 的 select

Go 的 `select` 语法由多个 `case` 分支组成，每个分支可以是对不同 channel 的发送或接收操作，以及对应要执行的代码块。

`select` 实现多路复用的原理是：当 goroutine 执行 `select` 时，会把自己注册到各个 channel 的等待队列里，并阻塞等待。只要其中任何一个 channel 准备好进行通信，Go 就会唤醒该 goroutine，执行对应的 `case` 分支。若多个 channel 同时就绪，Go 会随机选择一个分支执行，以保证公平性。

channel 的等待队列：每个 channel 有读等待和写等待队列。如果有 goroutine 在读，但 channel 里没有数据，这个 goroutine 就会被阻塞。反过来如果有 goroutine 在写入却没有接收者，则这个写入者会阻塞。

select 的特性

1. select 操作至少要有一个 case 语句，出现读写 nil 的 channel 该分支会忽略，在 nil 的 channel 上操作则会报错。
2. select 仅支持管道，而且是单协程操作。
3. 每个 case 语句仅能处理一个管道，要么读要么写。
4. 多个 case 语句的执行顺序是随机的。公平性。
5. 存在 default 语句，select 将不会阻塞，但是存在 default 会影响性能。

11. 讲讲 Go 的 defer 底层数据结构和一些特性?

答：每个 defer 语句都对应一个 `_defer` 实例，多个实例使用指针连接起来形成一个单链表，保存在 gotoutine 数据结构中，每次插入_defer 实例，均插入到链表的头部，函数结束再一次从头部取出，从而形成后进先出的效果。
defer 的规则总结：
延迟函数的参数是 defer 语句出现的时候就已经确定了的。

延迟函数执行按照后进先出的顺序执行，即先出现的 defer 最后执行。
延迟函数可能操作主函数的返回值。
申请资源后立即使用 defer 关闭资源是个好习惯。

12. 单引号，双引号，反引号的区别?

- 单引号
    - 单引号表示 **rune 类型**（Go 中 rune 就是 int32）。
    - **不会自动是 byte 类型**，除非用类型转换。
    - Go 的单引号里只能写**单个字符（Unicode code point）**。
    - byte 就是 uint8，常用于 raw data，比如二进制流。rune 表示 Unicode 码点。

```
var ch rune = '中'          // 类型 rune (int32)
var b byte = 'A'            // 类型 byte (uint8)
fmt.Println(ch)             // 20013
fmt.Println(b)              // 65
```

- 双引号
    - 双引号表示 string 类型。
    - Go 中 string 底层确实是字节数组，但并不是 Go 数组类型，而是更类似切片，存的是只读字节序列。
    - 索引访问的单位是 字节，不是字符。

```
s := "Hello 中"
fmt.Println(s[0])           // 72 (ASCII 'H')
fmt.Println(len(s))         // 8
```

- 反引号
    - 反引号，表示字符串字面量，但不支持任何转义序列。字面量 raw literal string 的意思是，你定义时写的啥样，它就啥样，你有换行，它就换行。你写转义字符，它也就展示转义字符。

13. go 出现 panic 的场景

Go 出现 panic 的场景

- 数组/切片越界
- 空指针调用。比如访问一个 nil 结构体指针的成员
- 过早关闭 HTTP 响应体
- 除以 0
- 向已经关闭的 channel 发送消息

- 重复关闭 channel
- 关闭未初始化的 channel
- 未初始化 map。注意访问 map 不存在的 key 不会 panic，而是返回 map 类型对应的零值，但是不能直接赋值
- 跨协程的 panic 处理
- sync 计数为负数。
- 类型断言不匹配。`var a interface{} = 1; fmt.Println(a.(string))` 会 panic，建议用 `s,ok := a.(string)`

14. go 是否支持 while 循环，如何实现这种机制

https://blog.csdn.net/chengqiuming/article/details/115573947

用 `for {}` 实现。

15. go 里面如何实现 set?

Go 中是不提供 Set 类型的，Set 是一个集合，其本质就是一个 List，只是 List 里的元素不能重复。

Go 提供了 map 类型，但是我们知道，map 类型的 key 是不能重复的，因此，我们可以利用这一点，来实现一个 set。那 value 呢？value 我们可以用一个常量来代替，比如一个空结构体，实际上空结构体不占任何内存，使用空结构体，能够帮我们节省内存空间，提高性能。

代码实现：https://blog.csdn.net/haodawang/article/details/80006059

16. go 如何实现类似于 java 当中的继承机制?

两分钟让你明白 Go 中如何继承

说到继承我们都知道，在 Go 中没有 extends 关键字，也就意味着 Go 并没有原生级别的继承支持。这也是为什么我在文章开头用了**伪继承**这个词。本质上，Go 使用 interface 实现的功能叫组合，Go 是使用组合来实现的继承，说的更精确一点，是使用组合来代替的继承，举个很简单的例子：

通过组合实现了继承：

```go
type Animal struct {
    Name string
}

func (a *Animal) Eat() {
    fmt.Printf("%v is eating", a.Name)
    fmt.Println()
}

type Cat struct {
    *Animal
}
```

```
13
14 cat := &Cat{
15     Animal: &Animal{
16         Name: "cat",
17     },
18 }
19 cat.Eat() // cat is eating
```

- 首先，我们实现了一个 Animal 的结构体，代表动物类。并声明了 Name 字段，用于描述动物的名字。
- 然后，实现了一个以 Animal 为 receiver 的 Eat 方法，来描述动物进食的行为。
- 最后，声明了一个 Cat 结构体，组合了 Cat 字段。再实例化一个猫，调用 Eat 方法，可以看到会正常的输出。
- 可以看到，Cat 结构体本身没有 Name 字段，也没有去实现 Eat 方法。唯一有的就是组合了 Animal 父类，至此，我们就证明了已经通过组合实现了继承。

总结:

- 如果一个 struct 嵌套了另一个匿名结构体，那么这个结构可以直接访问匿名结构体的属性和方法，从而用组合实现继承。
- 如果一个 struct 嵌套了另一个有名的结构体，有名就是一个字段了，不是继承。
- 如果一个 struct 嵌套了多个匿名结构体，那么这个结构可以直接访问多个匿名结构体的属性和方法，从而实现多重继承。

17. 怎么去复用一个接口的方法?

[怎么在 golang 中通过接口嵌套实现复用 - 开发技术 - 亿速云]

18. go 里面的 _

1. **忽略返回值**
    - 比如某个函数返回三个参数，但是我们只需要其中的两个，另外一个参数可以忽略，这样的话代码可以这样写:

```
1 v1, v2, _ := function(...)
2 v1, _, _ := function(...)
```

1. **用在变量 (特别是接口断言)**

```
1 type T struct{}
2 var _ X = T{}
3 //其中 I 为 interface
```

上面用来判断 type T 是否实现了 X，用作类型断言，如果 T 没有实现接口 X，则编译错误。这里的意思是：定义一个 `X` 类型的变量 `_`，再将 `T` 赋给 `_`。这样，T 一定是 X 类的，要实现所有的 X 特性。

1. **用在 import package**

```
import _ "test/food"
```

引入包时，会先调用包中的初始化函数，这种使用方式仅让导入的包做初始化，而不使用包中其他功能

19. goroutine 创建的时候如果要传一个参数进去有什么要注意的点?

https://www.cnblogs.com/waken-captain/p/10496454.html
注：Golang1.22 版本对于 for loop 进行了修改，详见 Fixing For Loops in Go 1.22

goroutine 会捕获上下文变量，也会得到参数表中的参数。捕获的变量会因上下文而变化，与 goroutine 创建时不一样，则会出bug。解决方案是把需要用的捕获的变量以参数的形式传入。

20. 写 go 单元测试的规范?

1. **单元测试文件命名规则：**

   单元测试需要创建单独的测试文件，不能在原有文件中书写，名字规则为 xxx_test.go。这个规则很好理解。

2. **单元测试包命令规则**

   单元测试文件的包名为原文件的包名添加下划线接 test，举例如下：

   ```
   // 原文件包名：
   package xxx
   // 单元测试文件包名：
   package xxx_test
   ```

3. **单元测试方法命名规则**

   单元测试文件中的测试方法和原文件中的待测试的方法名相对应，以 Test 开头，举例如下：

   ```
   // 原文件方法：
   func Xxx(name string) error
   // 单元测试文件方法：
   func TestXxx()
   ```

4. **单元测试方法参数**

   单元测试方法的参数必须是 t *testing.T，举例如下：

```
func TestZipFiles(t *testing.T) { ...}
```

21. 单步调试?

golang 单步调试神器 delve

22. 导入一个 go 的工程，有些依赖找不到，改怎么办?

Go 是怎么解决包依赖管理问题的？ - 牛奔 - 博客园

23. 值拷贝 与 引用拷贝，深拷贝 与 浅拷贝

map，slice，chan 是引用拷贝；引用拷贝 是 浅拷贝。

其余的，都是 值拷贝；值拷贝 是 深拷贝。

深浅拷贝的本质区别：

是否真正获取对象实体，而不是引用。

深拷贝：

拷贝的是数据本身，创造一个新的对象，并在内存中开辟一个新的内存地址，与原对象是完全独立的，不共享内存，修改新对象时不会影响原对象的值。释放内存时，也没有任何关联。

值拷贝：

接收的是 整个 array 的值拷贝，所以方法对 array 中元素的重新赋值不起作用。

```
package main

import "fmt"

func modify(a [3]int) {
    a[0] = 4
    fmt.Println("modify",a)             // modify [4 2 3]
}

func main() {
    a := [3]int{1, 2, 3}
    modify(a)
    fmt.Println("main",a)                  // main [1 2 3]
}
```

浅拷贝：

拷贝的是数据地址，只复制指向的对象的指针，新旧对象的内存地址是一样的，修改一个另一个也会变。释放内存时，同时释放。

引用拷贝：

函数的引用拷贝与原始的引用指向同一个数组，所以对数组中元素的修改，是有效的

```
package main

```

```go
import "fmt"

func modify(s []int) {
    s[0] = 4
    fmt.Println("modify",s)          // modify [4 2 3]
}

func main() {
    s := []int{1, 2, 3}
    modify(s)
    fmt.Println("main",s)              // main [4 2 3]
}
```

24. 精通 Golang 项目依赖 Go modules

Go Path 的弊端

- 无版本控制。既不能控制本程序的版本，也不能控制第三方包的版本。

Go Modules

- 一个 Go Module 是一个 Go 包的集合。module 是有版本的，所以 module 下的包也就有了版本属性。这个 module 与这些包会组成一个独立的版本单元，它们一起打版本、发布和分发。一个 Go Module 的顶层目录下会放置一个 go.mod 文件，每个 go.mod 文件会定义唯一一个 module。
- 语义导入版本机制：Go module 使用语义导入版本机制来保证向后兼容。模块在发布 v2 或以上版本时，必须在 module 路径和导入路径中显式包含 `/vN` 后缀。这种机制允许不同版本的模块同时存在于同一项目中，从而避免版本冲突。
- 最小版本选择机制：Go 会在该项目依赖项的所有版本中，选出符合项目整体要求的“最小版本”。

25. Go 多返回值怎么实现的?

答：Go 传参和返回值是通过 FP+offset 实现，并且存储在调用函数的栈帧中。FP 栈底寄存器，指向一个函数栈的底部；PC 程序计数器，指向下一条执行指令；SB 指向静态数据的基指针，全局符号；SP 栈顶寄存器。

26. Go 语言中不同的类型如何比较是否相等?

答：像 string，int，float interface 等可以通过 reflect.DeepEqual **或** 等于号进行比较，像 slice，struct，map 则一般使用 reflect.DeepEqual 来检测是否相等。

27. Go 中 init 函数的特征?

答：一个包下可以有多个 init 函数，每个文件也可以有多个 init 函数。多个 init 函数按照它们的文件名顺序逐个初始化。应用初始化时初始化工作的顺序是，从被导入的最深层包开始进行初始化，层层递出最后到 main 包。不管包被导入多少次，包内的 init 函数只会执行一次。应用初始化时初始化工作的顺序是，从被导入的最深层包开始进行初始化，层层递出最后到 main

包。但包级变量（大概就是 Cpp 全局变量那种，在函数外面定义的变量）的初始化先于包内 init 函数的执行。

28. Go 中 uintptr 和 unsafe.Pointer 的区别?

答：uintptr 足以容纳指针值的整数类型，它不是指针，不指向任何内存，仅能用于计算。unsafe.Pointer 是通用指针类型，它不能参与计算，任何类型的指针都可以转化成 unsafe.Pointer，unsafe.Pointer 可以转化成任何类型的指针，uintptr 可以转换为 unsafe.Pointer，unsafe.Pointer 可以转换为 uintptr。使用 uintptr 计算时要小心 uintptr 存储的数字指向的内存是否有效，如果已经被回收，它转为 unsafe.Pointer 就会出现错误。

29. 什么是 goroutine

1. **定义**：
    - goroutine 是 Go 语言中的一种轻量级线程，由 Go 运行时管理。
2. **使用方法**：
    - 使用 `go` 关键字启动一个新的 goroutine。例如：`go 函数名(参数列表)`。
3. **优势**：
    - goroutine 的创建和销毁开销非常小，可以高效地创建成千上万个 goroutine。
    - goroutine 是并发执行的，可以提高程序的执行效率。
4. **调度**：
    - 由 Go Runtime 调度和管理，无需手动管理线程。
5. **通信**：
    - goroutine 之间通过 channel 进行通信，确保数据传递的安全性和同步性。
6. **典型应用**：
    - 适用于并发任务处理，如网络请求处理、并发计算等。
7. **示例**：
    - 在 Web 服务器中，每个请求可以由一个单独的 goroutine 处理，从而提高并发处理能力。

这样回答简洁明了，可以帮助面试官快速了解你对 goroutine 的理解。

30. GPM 调度并发模型

- **Goroutine**：协程。就是我们常用的用 `go` 关键字创建的执行体，它对应一个结构体 `g`，结构体中保存了 Goroutine 的栈信息、状态、上下文等调度信息。从 go func () 开始创建一个 Goroutine，新建的 Goroutine 优先保存在 P 的本地队列中，如果 P 的本地队列已经满了（最多 256 个），则会保存到全局队列中。
- **Processor**：表示逻辑处理器，有了它才能建立 G 和 M 之间的联系

    - `P` 是调度的核心组件，结构体为 `runtime.p`。它维护一个本地 Goroutine 队列（run queue）和执行所需的上下文。
    - `P` 负责将 `G` 分配给绑定的 `M` 执行。只有绑定了 `P` 的 `M` 才能运行 `G`，`P` 的数量由 `GOMAXPROCS` 决定。
- **Machine**：表示操作系统线程
    - `M` 是操作系统线程的抽象，结构体为 `runtime.m`。调度器会根据需要动态创建或复用 M。并不是固定只有 `GOMAXPROCS` 个线程，但**同时运行的 M 不会超过 `P` 的数量**。每个活跃的 `M` 需要绑定一个 `P` 才能执行 `G`。
    - `GOMAXPROCS` 表示运行时最多允许多少个 P 并发执行 Goroutine，默认等于逻辑 CPU 数。当每个 P 绑定一个 M 时，可以充分利用多核，避免频繁上下文切换，提高执行效率。

M 线程 < P 逻辑处理器 = GOMAXPROCS CPU 内核数

Go 的调度器会智能调度 Goroutine，具体调度策略如下：

- 首先从当前 `P` 的本地队列中获取 `G` 来运行；
- 每执行 **61** 次本地队列拉取后，会尝试从全局队列中获取 `G`；
- 如果本地队列和全局队列都为空，当前 `P` 会尝试从**其他 `P` 的本地队列中“偷取”一半的 G**（work stealing）；
- 如果当前没有可用的 M 来执行 G，调度器会尝试创建新的 M 或唤醒空闲的 M；

二、array、slice、string

先从底层定义入手：

- 数组是值类型，申请变量时，直接分配内存。有一个需要注意：数组的长度是类型的一部分。是可变但不可变长的。长度单位是元素。
- slice 切片是引用类型，它底层是一个 `struct`，包含 `指针`、`长度`、`容量`。是可变且可变长的。长度单位是元素。
- string 是 `struct`，底层是 `指针`、`长度`。指针指向`堆或者数据段（常量池）`。string 是不可变的。长度单位是字节。

关于拼接的字符串：

```
s1 := "hello"
s2 := s1 + " world"
```

s2 会新开辟一块内存，一般在堆上（也看逃逸分析），如果频繁拼接字符串则会频繁触发堆分配。所以拼接尽量使用 Builder。

关于互相转换：

- slice 切片 与 string 字符串 互相转换，切片可变，字符串不可变，所以内存位置不同，一定会产生开销。

- array 数组转 slice 切片，直接生成引用。不深拷贝。

  slice 切片转 array 数组，必须手动 copy `copy(a[:], s)`，这行代码是把 数组 a 作为切片，再把 s copy 给 a 切片。这是因为数组是定长的，而且数组是值类型，它拥有值，而切片是指向内存的。
- array 数组与 slice 切片不能直接互转，但可以借助切片。为了保证string的不可变，一定会深拷贝。`string(a[:])` 会把数组 a 作为切片参数，返回一个 string。

还有需要注意的点：

- 空字符串与直接定义是一样的。也就是说

```
var s1 string
var s2 = ""      // 这俩是一样的
```

- array 作为参数传递，深拷贝。因为是值类型。
- slice 作为参数传递，浅拷贝。
- string 作为参数传递，并不会将内容复制一份（深拷贝），因为它是指针，还指向不可变内容，有一份内容就可以。但是会浅拷贝，把指针复制一份。

1. 数组和切片的区别（基本必问）

相同点：

1. 只能存储一组相同类型的数据结构
2. 都是通过下标来访问

区别：

1. 数组是定长，访问和复制不能超过数组定义的长度，否则就会下标越界。

   切片长度和容量可以自动扩容。
2. 数组是值类型，切片是引用类型，每个切片都引用了一个底层数组，切片本身不能存储任何数据，都是这底层数组存储数据，所以修改切片的时候修改的是底层数组中的数据。切片一旦扩容，指向一个新的底层数组，内存地址也就随之改变

简洁的回答：

1. 定义方式不一样
2. 初始化方式不一样，数组需要指定大小，大小不改变
3. **数组的定义**
   var a1 [3]int
   var a2 [...]int{1,2,3}
   切片的定义
   var a1 []int
   var a2 :=make([]int,3,5)
   数组的初始化
   a1 := [...]int{1,2,3}

a2 := [5]int{1,2,3}

切片的初始化

b:= make([]int,3,5)

数组和切片有什么异同 - 码农桃花源

【引申 1】 [3]int 和 [4]int 是同一个类型吗?

不是。因为数组的长度是类型的一部分，这是与 slice 不同的一点。

2.讲讲 Go 的 slice 底层数据结构和一些特性?

答：Go 的 slice 底层数据结构是由一个 array 指针指向底层数组，len 表示切片长度，cap 表示切片容量。slice 的主要实现是扩容。对于 append 向 slice 添加元素时，假如 slice 容量够用，则追加新元素进去，slice.len++，返回原来的 slice。当原容量不够，则 slice 先扩容，扩容之后 slice 得到新的 slice，将元素追加进新的 slice，slice.len++，返回新的 slice。对于切片的扩容规则：当切片比较小时（容量小于 1024.，则采用较大的扩容倍速进行扩容（新的扩容会是原来的 2 倍），避免频繁扩容，从而减少内存分配的次数和数据拷贝的代价。当切片较大的时（原来的 slice 的容量大于或者等于 1024.，采用较小的扩容倍速（新的扩容将扩大大于或者等于原来 1.25 倍），主要避免空间浪费，网上其实很多总结的是 1.25 倍，那是在不考虑内存对齐的情况下，实际上还要考虑内存对齐，扩容是大于或者等于 1.25 倍。

注：Go 的切片扩容源代码在 runtime 下的 growslice 函数。

（关于刚才问的 slice 为什么传到函数内可能被修改，如果 slice 在函数内没有出现扩容，函数外和函数内 slice 变量指向是同一个数组，则函数内复制的 slice 变量值出现更改，函数外这个 slice 变量值也会被修改。如果 slice 在函数内出现扩容，则函数内变量的值会新生成一个数组（也就是新的 slice，而函数外的 slice 指向的还是原来的 slice，则函数内的修改不会影响函数外的 slice。）

3. golang 中数组和 slice 作为参数的区别？slice 作为参数传递有什么问题?

golang 数组和切片作为参数和返回值*weixin_44387482 的博客-CSDN 博客*golang 返回数组

1. 当使用数组作为参数和返回值的时候，值传递，会深拷贝，在函数内部对数组进行修改并不会影响原数据。如果希望数据被修改，就要指针传递。
2. 当切片作为参数的时候传进去的是指针，会浅拷贝，指向的内存不变，在函数里面修改切片的时候，源数据也会被修改。
3. 切片在传递的过程中，有着共享底层数组的风险，所以如果在函数内部进行了更改的时候，会修改到源数据，所以我们需要根据不同的需求来处理。如果不希望源数据被修改，可以使用 copy 函数深拷贝切片后再传入。

4. 从数组中取一个相同大小的 slice 有成本吗?

或者这么问：从切片中取一个相同大小的数组有成本吗?

从数组中截取切片：https://blog.csdn.net/weixin_42117918/article/details/81913036

成本问题前面已经说过。数组取切片是生成引用，切片转数组要深拷贝。

5. slice 扩容策略

扩容倍数怎么定?

大原则：

- 小容量时 → 容量翻倍
- 大容量时 → 增幅变小

Go 1.16 及之前，主要规则：

- 如果旧容量 < 1024 → 每次翻倍
- 如果旧容量 ≥ 1024 → 每次只增加 25%

Go 1.16 的改进

Go 1.16 做了**大容量 slice 扩容优化。**

官方说：

> **改成更精准的指数增长。**

原先：

- 大于 1024，每次只 +25%
- 会导致多次内存分配

改进后：

- 不是简单加 25%，而是指数增长
- 保证内存分配更少
- 更接近 2x，但没有一次翻那么大

增长比例趋于 1.25x ~ 2x

但：

> Go 不再死板用 +25%，而是采用指数增长，避免频繁分配。

三、map 相关

什么类型可以作为 map 的 key

在 Go 语言中，map 的 key 可以是任何可以**比较**的类型。这包括所有的基本类型，如**整数、浮点数、字符串和布尔值，以及结构体和数组**，只要它们没有被定义为包含不可比较的类型（如切片、映射或函数）。
以下是一些可以作为 map key 的类型的例子：

1. 整数类型：

```go
m := make(map[int]string)
m[1] = "one"
```

2. 字符串类型：

```go
m := make(map[string]int)
m["one"] = 1
```

3. 布尔类型：

```go
m := make(map[bool]string)
m[true] = "yes"
m[false] = "no"
```

4. 结构体类型（只要结构体的所有字段都是可比较的）：

```go
type Point struct {
    X, Y int
}
m := make(map[Point]string)
m[Point{1, 2}] = "1,2"
```

5. 数组类型（数组的元素类型必须是可比较的）：

```go
m := make(map[[2]int]string)
m[[2]int{1, 2}] = "1,2"
```

注意，切片、映射和函数类型是不可比较的，因此不能作为 map 的 key。如果你需要一个包含这些类型的 key，你可以考虑使用一个指向这些类型的指针，或者将它们封装在一个可比较的结构体中，并确保结构体不包含任何不可比较的类型。

map 使用注意的点，是否并发安全?

map 使用的注意点

1. **key 的唯一性**：map 中的每个 key 必须是唯一的。如果尝试使用已存在的 key 插入新值，则会覆盖旧值。
2. **key 的不可变性**：作为 key 的类型必须是**可比较的**，这**通常意味着它们应该是不可变的。**例如，在 Go 语言中，切片、映射和函数类型因为包含可变状态，所以不能直接作为 map 的 key。
3. **初始化和 nil map**：在 Go 语言中，声明一个 map 变量不会自动初始化它。未初始化的 map 变量的零值是 nil，对 nil map 进行读写操作会引发 panic。因此，在使用 map 之前，应该使用 `make` 函数进行初始化。
4. **遍历顺序**：map 的遍历顺序是不确定的，每次遍历的结果可能不同。如果需要按照特定顺序处理 map 中的元素，应该先对 key 进行排序。

5. **并发安全性**：默认情况下，**map 并不是并发安全的。**在并发环境下对同一个 map 进行读写操作可能会导致竞态条件和数据不一致性。

并发安全性

Go 语言中的 map 并发安全性：

- Go 语言中的 map 类型并不是并发安全的。这意味着，如果有多个 goroutine 尝试同时读写同一个 map，可能会导致竞态条件和数据损坏。
- 为了在并发环境下安全地使用 map，可以采取以下几种策略：
    1. **使用互斥锁（sync.Mutex）**：在读写 map 的操作前后加锁，确保同一时间只有一个 goroutine 可以访问 map。
    2. **使用读写互斥锁（sync.RWMutex）**：如果读操作远多于写操作，可以使用读写锁来提高性能。读写锁允许多个 goroutine 同时读取 map，但在写入时需要独占访问。
    3. **使用并发安全的 map（sync.Map）**：从 Go 1.9 版本开始，标准库中的 `sync` 包提供了 `sync.Map` 类型，这是一个专为并发环境设计的 map。它提供了一系列方法来安全地在多个 goroutine 之间共享数据。

结论：
在使用 map 时，需要注意其 key 的唯一性和不可变性，以及初始化和并发安全性的问题。特别是在并发环境下，应该采取适当的措施来确保 map 的安全访问，以避免竞态条件和数据不一致性。在 Go 语言中，可以通过使用互斥锁、读写互斥锁或并发安全的 map（`sync.Map`）来实现这一点。

map 循环是有序的还是无序的?

在 Go 语言中，map 的循环（遍历）是无序的。这意味着当你遍历 map 时，每次遍历的顺序**可能**都不同（要看哈希结构是否改变）。Go 语言的 map 是基于哈希表的，因此元素的存储顺序是不确定的，并且可能会随着元素的添加、删除等操作而改变。

如果你需要按照特定的顺序处理 map 中的元素，你应该先将 key 提取到一个切片中，对切片进行排序，然后按照排序后的顺序遍历切片，并从 map 中取出对应的值。这样，你就可以按照特定的顺序处理 map 中的元素了。

map 中删除一个 key，它的内存会释放么?

在 Go 语言中，从 map 中删除一个 key 时，其内存释放的行为并非直观且立即的，这涉及到 Go 语言的内存管理机制。具体来说，删除 map 中的 key 后，其内存释放情况如下：

内存标记与垃圾回收

1. **删除操作**：使用 `delete` 函数从 map 中删除一个 key 时，该 key 及其关联的值会被从 map 的内部数据结构中移除。此时，这些值在逻辑上不再属于 map 的一部分。
2. **内存标记**：删除操作后，如果没有任何其他变量或数据结构引用被删除的值，那么这些值就变成了垃圾回收器的目标。Go 语言的垃圾回收器（Garbage Collector, GC）会定期扫描内存，标记那些不再被使用的内存区域。

3. **内存释放**：在垃圾回收过程中，被标记为垃圾的内存区域会被释放回堆内存，供后续的内存分配使用。然而，这个过程并不是立即发生的，而是由垃圾回收器的触发条件和回收策略决定的。

注意事项

1. **内存释放时机**：由于垃圾回收器的非确定性，删除 map 中的 key 后，其内存释放的时机是不确定的。因此，不能依赖删除操作来立即释放内存。
2. **map 底层存储不变**：删除操作只是逻辑上移除了 key-value 对，但 map 底层分配的内存（如哈希表的桶和溢出桶）并不会立即减小。这是因为 map 的设计优先考虑的是访问速度，而不是空间效率。如果需要释放大量内存，一种方法是创建一个新的 map，并将旧 map 中需要保留的元素复制过去。
3. **并发安全**：如果 map 在多个 goroutine 之间共享，那么删除操作需要考虑并发安全问题。可以使用互斥锁（如 `sync.Mutex`）来保护对 map 的访问，或者使用 Go 1.9 引入的 `sync.Map`，它提供了内置的并发安全机制。

结论

从 map 中删除一个 key 后，其内存并不会立即释放。内存释放取决于 Go 语言的垃圾回收器何时触发回收过程。在大多数情况下，开发者不需要过于担心内存释放的问题，因为 Go 的内存管理机制相当智能。然而，在处理大量数据时，了解这些内存管理的细节对于优化程序性能是非常有帮助的。

怎么处理对 map 进行并发访问？有没有其他方案？区别是什么？

处理对 map 进行并发访问的问题，主要需要确保在多个 goroutine 同时访问 map 时不会出现竞态条件和数据不一致的情况。以下是几种处理并发访问 map 的方案及其区别：

使用互斥锁（sync.Mutex）

方案描述：
使用 `sync.Mutex` 或 `sync.RWMutex`（读写互斥锁）来控制对 map 的访问。在访问 map 之前加锁，访问完成后释放锁。这样可以保证在同一时间内只有一个 goroutine 可以访问 map。
优点：

- 实现简单，容易理解。
- 对于写操作频繁的场景，能够较好地保证数据一致性。

缺点：

- 在读多写少的场景下，性能可能不是最优的，因为读操作也需要获取锁。
- 锁的粒度较大，可能会影响并发性能。

使用读写互斥锁（sync.RWMutex）

方案描述：
与 `sync.Mutex` 类似，但 `sync.RWMutex` 允许多个 goroutine 同时读取 map，只有在写入时才需要独占访问。
优点：

- 在读多写少的场景下，性能优于 `sync.Mutex`，因为读操作不需要获取写锁。

缺点：

- 写入操作仍然需要独占访问，可能会影响并发写入的性能。
- 实现略复杂于 `sync.Mutex`，需要区分读写操作。

使用并发安全的 map（sync.Map）

方案描述：
从 Go 1.9 版本开始，标准库中的 `sync` 包提供了 `sync.Map` 类型，它是一个专为并发环境设计的 map。`sync.Map` 内部使用了读写锁和其他同步机制来保证并发访问的安全性。
优点：

- 无需显式加锁，简化了并发编程。
- 针对读多写少的场景进行了优化，如读写分离等，提高了并发性能。
- 提供了特定的方法（如 `Load`、`Store`、`Delete` 等）来安全地访问 map。

缺点：

- 在某些情况下，性能可能不如使用 `sync.RWMutex` 的自定义 map（尤其是在写入操作频繁时）。
- `sync.Map` 的 API 与内置 map 不同，可能需要适应新的使用方式。

区别总结

方案	实现复杂度	性能（读多写少）	性能（写多）	使用场景
sync.Mutex	低	中等	中等	写操作频繁，对并发性能要求不高
sync.RWMutex	中等	高	中等	读多写少，需要较高并发读性能
sync.Map	低（API 不同）	高	中等偏下	读多写少，追求简洁的并发编程模型

注意事项

- 在选择方案时，需要根据实际的应用场景（如读写比例、并发级别等）来决定使用哪种方案。
- 如果并发级别不高，且对性能要求不高，也可以考虑使用简单的锁机制（如 `sync.Mutex`）来简化实现。
- 对于性能要求极高的场景，可能需要考虑更复杂的并发数据结构或算法来优化性能。

综上所述，处理对 map 的并发访问需要根据具体情况选择合适的方案，并在实际使用中不断优化和调整以达到最佳性能。

nil map 和空 map 有何不同?

在 Go 语言中，nil map 和空 map 之间存在一些关键的不同点，主要体现在它们的初始状态、对增删查操作的影响以及内存占用等方面。

初始状态与内存占用

- **nil map**：未初始化的 map 的零值是 nil。这意味着 map 变量被声明后，如果没有通过 `make` 函数或其他方式显式初始化，它将保持 nil 状态。nil map 不占用实际的内存空间来存储键值对，因为它没有底层的哈希表结构。
- **空 map**：空 map 是通过 `make` 函数或其他方式初始化但没有添加任何键值对的 map。空 map 已经分配了底层的哈希表结构，但表中没有存储任何键值对。因此，空 map 占用了一定的内存空间，尽管这个空间相对较小。

对增删查操作的影响

- **nil map**：
    - **添加操作**：向 nil map 中添加键值对将导致运行时 panic，因为 nil map 没有底层的哈希表来存储数据。
    - **删除操作**：在早期的 Go 版本中，尝试从 nil map 中删除键值对也可能导致 panic，但在最新的 Go 版本中，这一行为可能已经被改变（具体取决于 Go 的版本），但通常不建议对 nil map 执行删除操作。
    - **查找操作**：从 nil map 中查找键值对不会引发 panic，但会返回对应类型的零值，表示未找到键值对。
- **空 map**：
    - **添加操作**：向空 map 中添加键值对是安全的，键值对会被添加到 map 中。
    - **删除操作**：从空 map 中删除键值对是一个空操作，不会引发 panic，因为 map 中原本就没有该键值对。
    - **查找操作**：从空 map 中查找不存在的键值对也会返回对应类型的零值，表示未找到键值对。

总结

nil map 和空 map 的主要区别在于它们的初始状态和对增删查操作的影响。nil map 未初始化且不能用于存储键值对，而空 map 已初始化且可以安全地用于增删查操作。在编写 Go 程序时，应根据需要选择使用 nil map 还是空 map，并注意处理 nil map 可能引发的 panic。

map 的数据结构是什么?

map - 地鼠文档

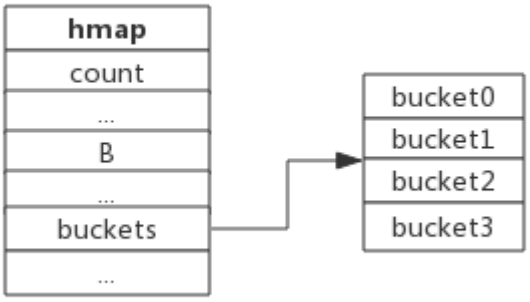
答：golang 中 map 是一个 kv 对集合。底层使用 hash table，用链表来解决冲突，出现冲突时，不是每一个 key 都申请一个结构通过链表串起来，而是以 bmap 为最小粒度挂载，一个 bmap 可以放 8 个 kv。在哈希函数的选择上，会在程序启动时，检测 cpu 是否支持 aes，如果支持，则使用 aes hash，否则使用 memhash。每个 map 的底层结构是 hmap，是有若干个结构为 bmap 的 bucket 组成的数组。每个 bucket 底层都采用链表结构。

map 底层就是 hmap，hmap 中的 bucket 就是 bmap。

hmap 的结构如下：

```
type hmap struct {
    count     int                  // 元素个数
    flags     uint8
    B         uint8                // 扩容常量相关字段 B 是 buckets 数组的长度的对数 2^B
    noverflow uint16               // 溢出的 bucket 个数
    hash0     uint32               // hash seed
    buckets    unsafe.Pointer      // buckets 数组指针
    oldbuckets unsafe.Pointer      // 结构扩容的时候用于赋值的 buckets 数组
    nevacuate  uintptr             // 搬迁进度
    extra *mapextra                // 用于扩容的指针
}
```

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。

bucket 数据结构

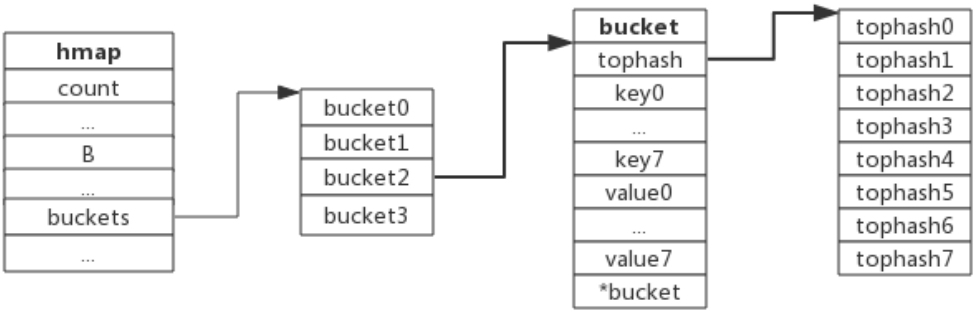
bucket 数据结构由 runtime/map.go:bmap 定义：

```
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。

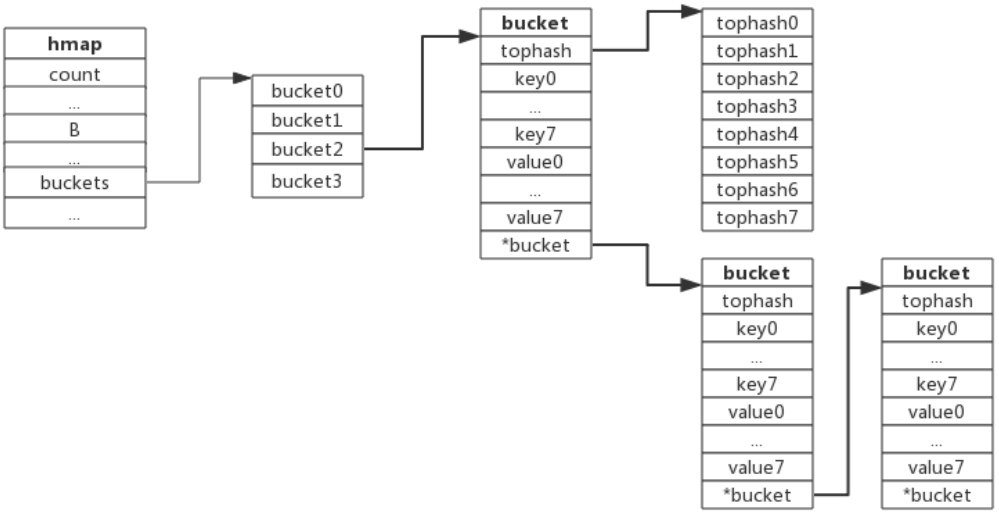
注意：上述中 data 和 overflow 并不是在结构体中显示定义的，而是直接通过指针运算进行访问的。
下图展示 bucket 存放 8 个 key-value 对：



解决哈希冲突（四种方法）

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。

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。

是怎么实现扩容?

map 的容量大小

底层调用 makemap 函数，计算得到合适的 B，map 容量最多可容纳 6.52B 个元素，6.5 为装载因子阈值常量。装载因子的计算公式是：装载因子=填入表中的元素个数/散列表的长度，装载因子越大，说明空闲位置越少，冲突越多，散列表的性能会下降。

触发 map 扩容的条件

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：

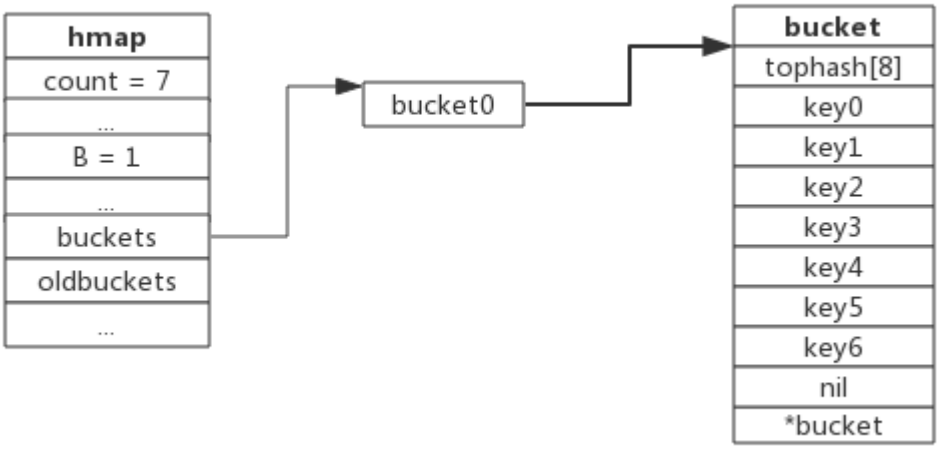
1. 负载因子 > 6.5 时，也即平均每个 bucket 存储的键值对达到 6.5 个。
2. overflow 数量 > 2^15 时，也即 overflow 数量超过 32768 时。

增量扩容

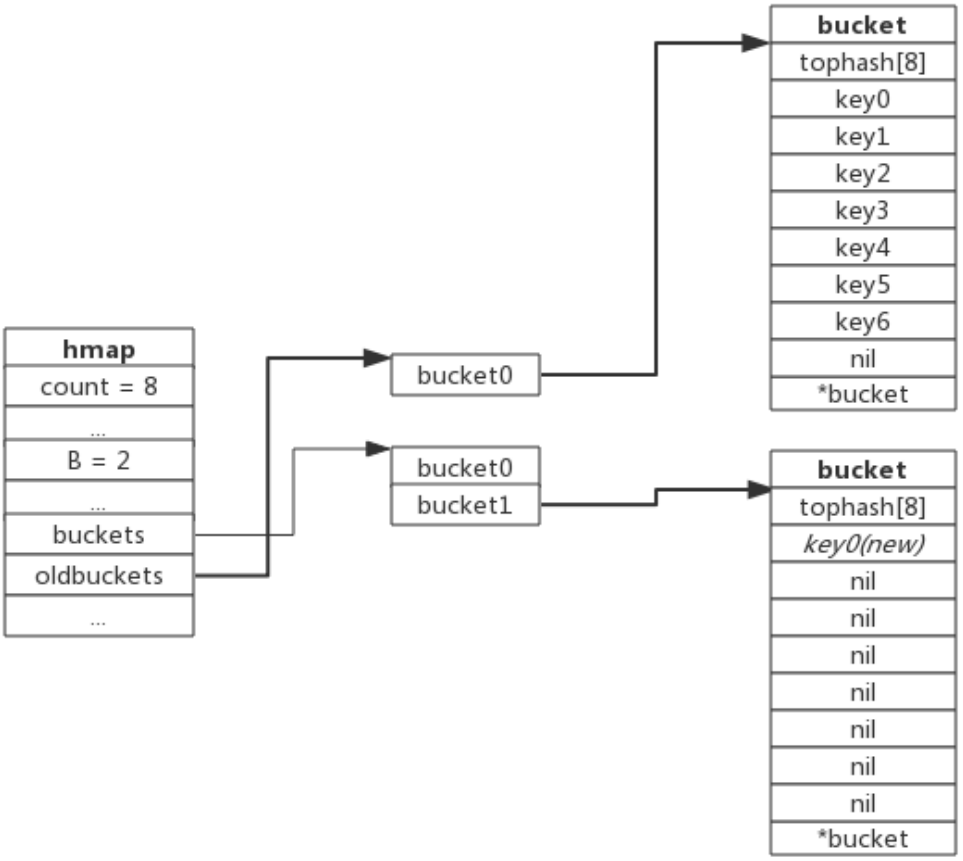
当负载因子过大时，就新建一个 bucket，新的 bucket 长度是原来的 **2 倍**，然后旧 bucket 数据搬迁到新的 bucket。

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)：



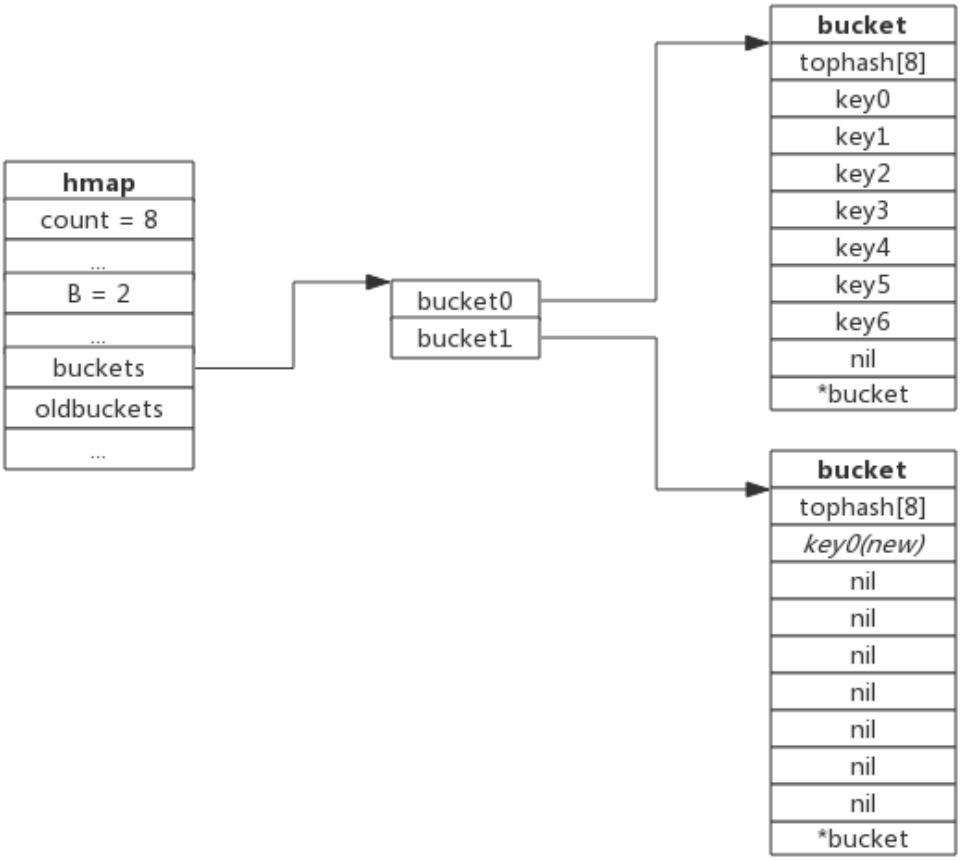
当前 map 存储了 7 个键值对，只有 1 个 bucket。此地负载因子为 7。再次插入数据时将会触发扩容操作，扩容之后再将新插入键写入新的 bucket。
当第 8 个键值对插入时，将会触发扩容，扩容后示意图如下：



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：

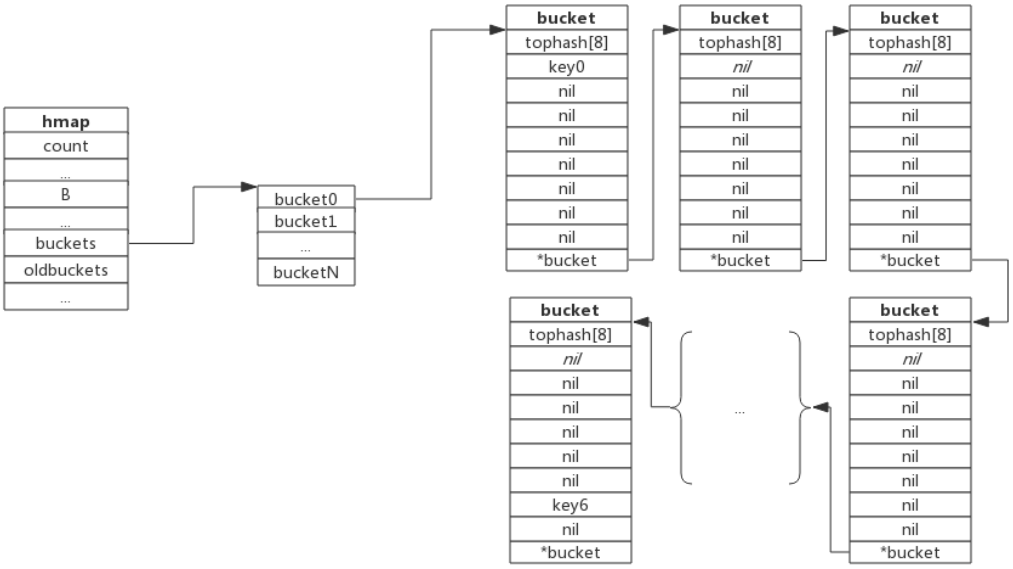


数据搬迁过程中原 bucket 中的键值对将存在于新 bucket 的前面，新插入的键值对将存在于新 bucket 的后面。
实际搬迁过程中比较复杂，将在后续源码分析中详细介绍。

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。

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。

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

可以对 map 里面的一个元素取地址吗

在 Go 语言中，你不能直接对 map 中的元素取地址，因为 map 的元素并不是固定的内存位置。**当你从 map 中获取一个元素的值时，你实际上得到的是该值的一个副本，而不是它的实际存储位置的引用。**这意味着，即使你尝试获取这个值的地址，你也只是得到了这个副本的地址，而不是 map 中原始元素的地址。
例如，考虑以下代码：

```
m := make (map [string] int)
m["key"] = 42
value := m["key"]
fmt.Println (&value) // 打印的是 value 变量的地址，而不是 map 中元素的地址
```

在这个例子中，`&value` 是变量 `value` 的地址，它包含了从 map 中检索出来的值的副本。如果你修改了 `value`，map 中的原始值是不会改变的。

如果你需要修改 map 中的值，你应该直接通过 map 的键来设置新的值：

```
m["key"] = newValue
```

这样，你就会直接修改 map 中存储的值，而不是修改一个副本。
如果你确实需要引用 map 中的值，并且希望这个引用能够反映 map 中值的改变，你可以使用指针类型的值作为 map 的元素。这样，你就可以存储和修改指向实际数据的指针了。例如：

```
m := make (map [string]*int)
m["key"] = new (int)
*m["key"] = 42
fmt.Println (*m["key"]) // 输出 42
```

在这个例子中，map 的值是指向 int 的指针，所以你可以通过指针来修改 map 中的实际值。

sync.map

`sync.Map` 是 Go 语言标准库中提供的并发安全的 Map 类型，它适用于读多写少的场景。以下是 `sync.Map` 的一些关键原理：

1. **读写分离**：`sync.Map` 通过读写分离来提升性能。它内部维护了两种数据结构：一个只读的只读字典（`read`），一个读写字典（`dirty`）。读操作优先访问只读字典，只有在只读字典中找不到数据时才会访问读写字典。
2. **延迟写入**：写操作并不立即更新只读字典（`read`），而是更新读写字典（`dirty`）。只有在读操作发现只读字典的数据过时（即 `misses` 计数器超过阈值）时，才会将读写字典中的数据同步到只读字典。这种策略减少了写操作对读操作的影响。

3. **原子操作**：读操作大部分是无锁的，因为它们主要访问只读的 `read` map，并通过原子操作（`atomic.Value`）来保护读操作；写操作会加锁（使用 `sync.Mutex`）保护写操作，以确保对 `dirty` map 的并发安全，确保高并发环境下的安全性。
4. **条目淘汰**：当一个条目被删除时，它只从读写字典中删除。只有在下一次数据同步时，该条目才会从只读字典中删除。

通过这种设计，`sync.Map` 在读多写少的场景下能够提供较高的性能，同时保证并发安全。

sync.map 的锁机制跟你自己用锁加上 map 有区别么

`sync.Map` 的锁机制和自己使用锁（如 `sync.Mutex` 或 `sync.RWMutex`）加上 map 的方式有一些关键区别：

自己使用锁和 map

1. **全局锁**：
    - 你需要自己管理锁，通常是一个全局的 `sync.Mutex` 或 `sync.RWMutex`。
    - 对于读多写少的场景，使用 `sync.RWMutex` 可以允许多个读操作同时进行，但写操作依然会阻塞所有读操作。
2. **手动处理**：
    - 你需要自己编写代码来处理加锁、解锁、读写操作。
    - 错误使用锁可能导致死锁、竞态条件等问题。
3. **简单直观**：
    - 实现简单，容易理解和调试。

`sync.Map`

1. **读写分离**：
    - `sync.Map` 内部使用读写分离的策略，通过只读和读写两个 map 提高读操作的性能。
    - 读操作大部分情况下是无锁的，只有在只读 map 中找不到数据时，才会加锁访问读写 map。
2. **延迟写入**：
    - 写操作更新读写 map（`dirty`），但不会立即更新只读 map（`read`）。只有当读操作发现只读 map 中的数据过时时，才会将读写 map 的数据同步到只读 map 中。
3. **内置优化**：
    - `sync.Map` 内部有各种优化措施，如原子操作、延迟写入等，使得它在读多写少的场景下性能更高。

区别总结

- **并发性能**：`sync.Map` 通过读写分离和延迟写入在读多写少的场景下提供更高的并发性能，而使用全局锁的 map 在读写频繁时性能较低。
- **复杂性和易用性**：`sync.Map` 封装了复杂的并发控制逻辑，使用起来更简单，而自己管理锁和 map 需要处理更多的并发控制细节。
- **适用场景**：`**sync.Map**` **适用于读多写少的场景**，而使用**全局锁的 map 适用于读写操作较均衡或者对性能要求不高**的场景。

如果你的应用场景是读多写少且对性能要求较高，`sync.Map` 会是一个更好的选择。而对于简单的并发访问控制，使用 `sync.Mutex` 或 `sync.RWMutex` 加上 map 也可以满足需求。

四、接口

0. 接口

接口（interface）是一种抽象类型，定义了一组方法的集合。**只要一个类型实现了接口要求的所有方法，它就自动实现了该接口。**

1. Go 语言与鸭子类型的关系

如果某个东西长得像鸭子，像鸭子一样游泳，像鸭子一样嘎嘎叫，那它就可以被看成是一只鸭子。Go 不要求类型显示地声明实现了某个接口，只要实现了相关的方法即可，编译器就能检测到。

总结一下，鸭子类型是一种动态语言的风格，在这种风格中，一个对象有效的语义，不是由继承自特定的类或实现特定的接口，而是由它 "当前方法和属性的集合" 决定。Go 作为一种静态语言，通过接口实现了 鸭子类型，实际上是 Go 的编译器在其中作了隐匿的转换工作。

2. 值接收者和指针接收者的区别

方法

方法能给用户自定义的类型添加新的行为。它和函数的区别在于方法有一个接收者，给一个函数添加一个接收者，那么它就变成了方法。接收者可以是值接收者，也可以是指针接收者。

在调用方法的时候，值类型既可以调用值接收者的方法，也可以调用指针接收者的方法；指针类型既可以调用指针接收者的方法，也可以调用值接收者的方法。

也就是说，不管方法的接收者是什么类型，该类型的值和指针都可以调用，不必严格符合接收者的类型。

实际上，当类型和方法的接收者类型不同时，其实是编译器在背后做了一些工作，用一个表格来呈现：

-	值接收者	指针接收者
值类型调用者	方法会使用调用者的一个副本，类似于“传值”	使用值的引用来调用方法，上例中，qcrao.growUp () 实际上是 (&qcrao).growUp ()

-	值接收者	指针接收者
指针类型调用者	指针被解引用为值，上例中，stefno.howOld () 实际上是 (*stefno).howOld ()	实际上也是“传值”，方法里的操作会影响到调用者，类似于指针传参，拷贝了一份指针

值接收者和指针接收者

前面说过，不管接收者类型是值类型还是指针类型，都可以通过值类型或指针类型调用，这里面实际上通过语法糖起作用的。

先说结论：实现了接收者是值类型的方法，相当于自动实现了接收者是指针类型的方法；而实现了接收者是指针类型的方法，不会自动生成对应接收者是值类型的方法。

所以，当实现了一个接收者是值类型的方法，就可以自动生成一个接收者是对应指针类型的方法，因为两者都不会影响接收者。但是，当实现了一个接收者是指针类型的方法，如果此时自动生成一个接收者是值类型的方法，原本期望对接收者的改变（通过指针实现），现在无法实现，因为值类型会产生一个拷贝，不会真正影响调用者。

最后，只要记住下面这点就可以了：如果实现了接收者是值类型的方法，会隐含地也实现了接收者是指针类型的方法。

两者分别在何时使用

如果方法的接收者是值类型，无论调用者是对象还是对象指针，修改的都是对象的副本，不影响调用者；如果方法的接收者是指针类型，则调用者修改的是指针指向的对象本身。
使用指针作为方法的接收者的理由：

- 方法能够修改接收者指向的值。
- 避免在每次调用方法时复制该值，在值的类型为大型结构体时，这样做会更加高效。

是使用值接收者还是指针接收者，不是由该方法是否修改了调用者（也就是接收者）来决定，而是应该基于该类型的本质。
如果类型具备“原始的本质”，也就是说它的成员都是由 Go 语言里内置的原始类型，如字符串，整型值等，那就定义值接收者类型的方法。像内置的引用类型，如 slice，map，interface，channel，这些类型比较特殊，声明他们的时候，实际上是创建了一个 header，对于他们也是直接定义值接收者类型的方法。这样，调用函数时，是直接 copy 了这些类型的 header，而 header 本身就是为复制设计的。
如果类型具备非原始的本质，不能被安全地复制，这种类型总是应该被共享，那就定义指针接收者的方法。比如 go 源码里的文件结构体（struct File）就不应该被复制，应该只有一份实体。

3. iface 和 eface 的区别是什么

`eface` 是空接口 `interface{}` 的底层结构，只保存类型和数据；`iface` 是有方法接口的结构，包含类型、数据和方法表。Go 用这两种结构来实现接口的动态特性。

从源码层面看一下：

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

type itab struct {
    inter  *interfacetype
    _type  *_type
    link   *itab
    hash   uint32 // copy of _type.hash. Used for type switches.
    bad    bool   // type does not implement interface
    inhash bool   // has this itab been added to hash?
    unused [2]byte
    fun    [1]uintptr // variable sized
}
```

iface 内部维护两个指针，tab 指向一个 itab 实体，它表示接口的类型以及赋给这个接口的实体类型。data 则指向接口具体的值，一般而言是一个指向堆内存的指针。

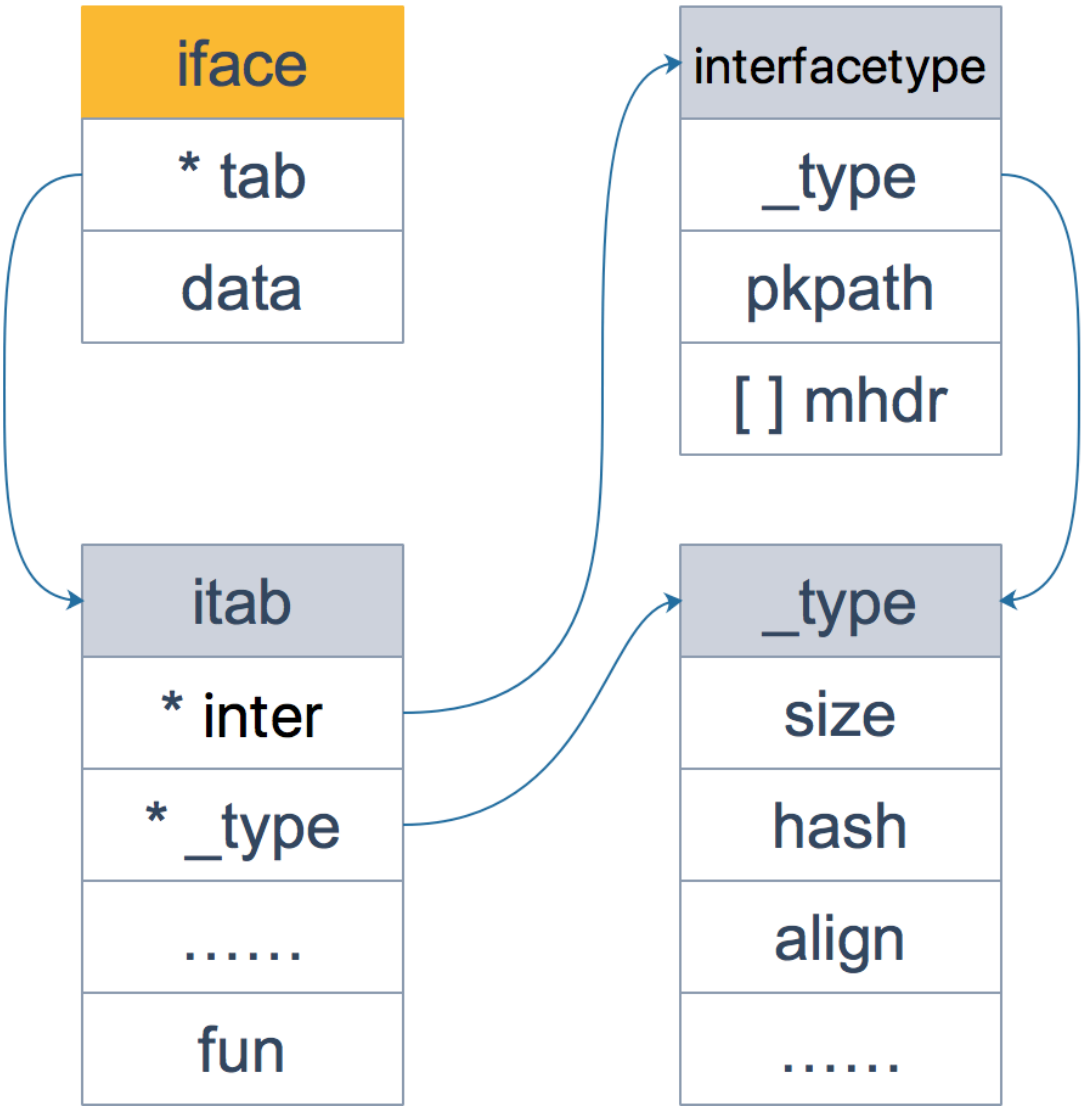
再来仔细看一下 itab 结构体：_type 字段描述了实体的类型，包括内存对齐方式，大小等；inter 字段则描述了接口的类型。fun 字段放置和接口方法对应的具体数据类型的方法地址，实现接口调用方法的动态分派，一般在每次给接口赋值发生转换时会更新此表，或者直接拿缓存的 itab。
这里只会列出实体类型和接口相关的方法，实体类型的其他方法并不会出现在这里。

另外，你可能会觉得奇怪，为什么 fun 数组的大小为 1，要是接口定义了多个方法可怎么办？实际上，这里存储的是第一个方法的函数指针，如果有更多的方法，在它之后的内存空间里继续存储。从汇编角度来看，通过增加地址就能获取到这些函数指针，没什么影响。顺便提一句，这些方法是按照函数名称的字典序进行排列的。

再看一下 interfacetype 类型，它描述的是接口的类型：

```
type interfacetype struct {
    typ     _type
    pkgpath name
    mhdr    []imethod
}
```

可以看到，它包装了 *type 类型*，type 实际上是描述 Go 语言中各种数据类型的结构体。我们注意到，这里还包含一个 mhdr 字段，表示接口所定义的函数列表，pkgpath 记录定义了接口的包名。
这里通过一张图来看下 iface 结构体的全貌：



接着来看一下 eface 的源码：

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

相比 iface，eface 就比较简单了。只维护了一个 _type 字段，表示空接口所承载的具体的实体类型。data 描述了具体的值。

4. 接口的动态类型和动态值

从源码里可以看到：iface 包含两个字段：tab 是接口表指针，指向类型信息；data 是数据指针，则指向具体的数据。它们分别被称为动态类型和动态值。而接口值包括动态类型和动态值。

【引申 1】接口类型和 nil 作比较
接口值的零值是指动态类型和动态值都为 nil。当仅且当这两部分的值都为 nil 的情况下，这个接口值就才会被认为 接口值 == nil。

5. 编译器自动检测类型是否实现接口

6. 接口的构造过程是怎样的

7. 类型转换和断言的区别

我们知道，Go 语言中不允许隐式类型转换，也就是说 = 两边，不允许出现类型不相同的变量。
类型转换、类型断言本质都是把一个类型转换成另外一个类型。不同之处在于，类型断言是对接口变量进行的操作。

类型转换

对于类型转换而言，转换前后的两个类型要相互兼容才行。类型转换的语法为：
<结果类型> := <目标类型> (<表达式>)

```
func main() {
    var i int = 9

    var f float64
    f = float64(i)                    // 类型转换
    fmt.Printf("%T, %v\n", f, f)

    f = 10.8
    a := int(f)
    fmt.Printf("%T, %v\n", a, a)
}
```

断言

前面说过，因为空接口 interface{} 没有定义任何函数，因此 Go 中所有类型都实现了空接口。当一个函数的形参是 interface{}，那么在函数中，需要对形参进行断言，从而得到它的真实类型。

断言的语法为：
<目标类型的值>，<布尔参数> := <表达式>.(目标类型) // 安全类型断言
<目标类型的值> := <表达式>.(目标类型) // 非安全类型断言

类型转换和类型断言有些相似，不同之处，在于类型断言是对接口进行的操作。

```
type Student struct {
    Name string
    Age int
}

func main() {
    var i interface{} = new(Student)
    s, ok := i.(Student)        // 类型断言
    if ok {
        fmt.Println(s)
    }
}
```

断言其实还有另一种形式，就是用在利用 switch 语句判断接口的类型。每一个 case 会被顺序地考虑。当命中一个 case 时，就会执行 case 中的语句，因此 case 语句的顺序是很重要的，因为很有可能会有多个 case 匹配的情况。

8. 接口转换的原理

通过前面提到的 iface 的源码可以看到，实际上它包含接口的类型 interfacetype 和 实体类型的类型 _type，这两者都是 iface 的字段 itab 的成员。也就是说生成一个 itab 同时需要接口的类型和实体的类型。

<interface 类型，实体类型> ->itable

当判定一种类型是否满足某个接口时，Go 使用类型的方法集和接口所需要的方法集进行匹配，如果类型的方法集完全包含接口的方法集，则可认为该类型实现了该接口。

例如某类型有 m 个方法，某接口有 n 个方法，则很容易知道这种判定的时间复杂度为 O(mn)，Go 会对方法集的函数按照函数名的字典序进行排序，所以实际的时间复杂度为 O(m+n)。

这里我们来探索将一个接口转换给另外一个接口背后的原理，当然，能转换的原因必然是类型兼容。

1. 具体类型转空接口时，_type 字段直接复制源类型的 _type；调用 mallocgc 获得一块新内存，把值复制进去，data 再指向这块新内存。
2. 具体类型转非空接口时，入参 tab 是编译器在编译阶段预先生成好的，新接口 tab 字段直接指向入参 tab 指向的 itab；调用 mallocgc 获得一块新内存，把值复制进去，data 再指向这块新内存。
3. 而对于接口转接口，itab 调用 getitab 函数获取。只用生成一次，之后直接从 hash 表中获取。

9. 如何用 interface 实现多态

Go 语言并没有设计诸如虚函数、纯虚函数、继承、多重继承等概念，但它通过接口却非常优雅地支持了面向对象的特性。
多态是一种运行期的行为，它有以下几个特点：

1. 一种类型具有多种类型的能力
2. 允许不同的对象对同一消息做出灵活的反应
3. 以一种通用的方式对待个使用的对象
4. 非动态语言必须通过继承和接口的方式来实现

main 函数里先生成 Student 和 Programmer 的对象，再将它们分别传入到函数 whatJob 和 growUp。函数中，直接调用接口函数，实际执行的时候是看最终传入的实体类型是什么，调用的是实体类型实现的函数。于是，不同对象针对同一消息就有多种表现，多态就实现了。

10. Go 接口与 C++ 接口有何异同

建议背诵：

- Go 的接口采用结构性类型系统（鸭子类型），不需要显式声明实现，适合快速解耦和动态调度。
- 而 C++ 接口采用名称性（名义）类型系统（显式声明继承），通常是通过抽象类和纯虚函数实现的，需要显式继承，适用于静态类型检查和更强的编译期约束。

接口定义了一种规范，描述了类的行为和功能，而不做具体实现。

C++ 的接口是使用抽象类来实现的，如果类中至少有一个函数被声明为纯虚函数，则这个类就是抽象类。纯虚函数是通过在声明中使用“= 0”来指定的。例如：

```
class Shape
{
   public:
      // 纯虚函数
      virtual double getArea() = 0;
   private:
      string name;      // 名称
};
```

设计抽象类的目的，是为了给其他类提供一个可以继承的适当的基类。抽象类不能被用于实例化对象，它只能作为接口使用。
派生类需要明确地声明它继承自基类，并且需要实现基类中所有的纯虚函数。

C++ 定义接口的方式称为“侵入式”，而 Go 采用的是“非侵入式”，不需要显式声明，只需要实现接口定义的函数，编译器自动会识别。

C++ 和 Go 在定义接口方式上的不同，也导致了底层实现上的不同。C++ 通过虚函数表来实现基类调用派生类的函数；而 Go 通过 itab 中的 fun 字段来实现接口变量调用实体类型的函数。C++ 中的虚函数表是在编译期生成的；而 Go 的 itab 中的 fun 字段是在运行期间动态生成的。原因在于，Go 中实体类型可能会无意中实现很多个接口，很多接口并不是本来需要的，所以不能为类型实现的所有接口都生成一个 itab，这也是“非侵入式”带来的影响；这在 C++ 中是不存在的，因为派生需要显示声明它继承自哪个基类。

五、context 相关

Context-地鼠文档

context 不会强制关闭 goroutine，而是通过 channel 通知取消或超时，由协程自行判断退出。虽然类似功能可以用 channel + struct 实现，但 context 提供了统一的接口、取消传播链、超时、错误状态和键值对传递，是更强大且标准化的并发控制工具。

1. context 结构是什么样的？context 使用场景和用途？

（难，也常常问你项目中怎么用，光靠记答案很难让面试官满意，反正有各种结合实际的问题）
参考链接：
go context 详解 - 卷毛狒狒 - 博客园 www.cnblogs.com/juanmaofeifei/p/14439957.html
答：Go 的 Context 的数据结构包含 Deadline，Done，Err，Value。Deadline 方法返回一个 time.Time，表示当前 Context 应该结束的时间，ok 则表示有结束时间，Done 方法当 Context 被取消或者超时时候返回的一个 close 的 channel，告诉给 context 相关的函数要停止当前工作然后返回了，Err 表示 context 被取消的原因，Value 方法表示 context 实现共享数据存储的地方，是协程安全的。context 在业务中是经常被使用的，

2. context 在 go 中一般可以用来做什么?

在 Go 语言中，`context` 包提供了一种管理多个 goroutine 之间的**截止时间**、**取消信号**和**请求范围数据**的方法。以下是 `context` 常见的用途：

1. **取消信号**：
    - `context` 可以用来向多个 goroutine 传递取消信号。当一个 goroutine 需要取消其他 goroutine 时，可以调用 `context` 的 `CancelFunc`。
    - 例如，在处理 HTTP 请求时，如果客户端关闭了连接，可以使用 `context` 取消所有相关的后台操作。
2. **截止时间/超时控制**：
    - `context` 可以设置一个截止时间或超时。当超过这个时间或超时发生时，`context` 会自动取消操作。
    - 例如，在数据库查询或网络请求时，可以使用 `context` 设置一个超时时间，以防止长时间的等待。
3. **传递请求范围的数据**：
    - `context` 可以在多个 goroutine 之间传递请求范围的数据，例如请求的唯一 ID、用户认证信息等。
    - 例如，在处理 HTTP 请求时，可以将请求的元数据存储在 `context` 中，并在各个处理函数之间传递这些数据。

具体示例

1. **创建带取消功能的 context**：

```go
ctx, cancel := context.WithCancel(context.Background())
defer cancel()

go func() {
    // 执行一些操作
    // 在需要取消操作时调用 cancel
    cancel()
}()

select {
case <-ctx.Done():
    fmt.Println("操作取消")
case result := <-someOperation():
    fmt.Println("操作结果：", result)
}
```

2. **创建带超时的 context**：

```go
ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
defer cancel()

select {
case <-ctx.Done():
    if ctx.Err() == context.DeadlineExceeded {
        fmt.Println("操作超时")
    }
case result := <-someOperation():
    fmt.Println("操作结果：", result)
}
```

3. **传递请求范围的数据**：

```go
ctx := context.WithValue(context.Background(), "requestID", "12345")

go func(ctx context.Context) {
    requestID := ctx.Value("requestID").(string)
    fmt.Println("处理请求 ID:", requestID)
}(ctx)
```

常用函数

- `context.Background()`：返回一个空的 `Context`，通常用于根 `Context`。
- `context.TODO()`：返回一个空的 `Context`，用于暂时不知道该使用什么 `Context` 的情况。
- `context.WithCancel(parent Context) (Context, CancelFunc)`：创建一个可以取消的 `Context`。
- `context.WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)`：创建一个带超时的 `Context`。
- `context.WithDeadline(parent Context, d time.Time) (Context, CancelFunc)`：创建一个带截止时间的 `Context`。
- `context.WithValue(parent Context, key, val interface{}) Context`：创建一个携带值的 `Context`。

通过这些功能，`context` 在 Go 中为管理 goroutine 的生命周期和跨 goroutine 传递数据提供了便利和强大的支持。

六、channel 相关

1. channel 是否线程安全？锁用在什么地方？

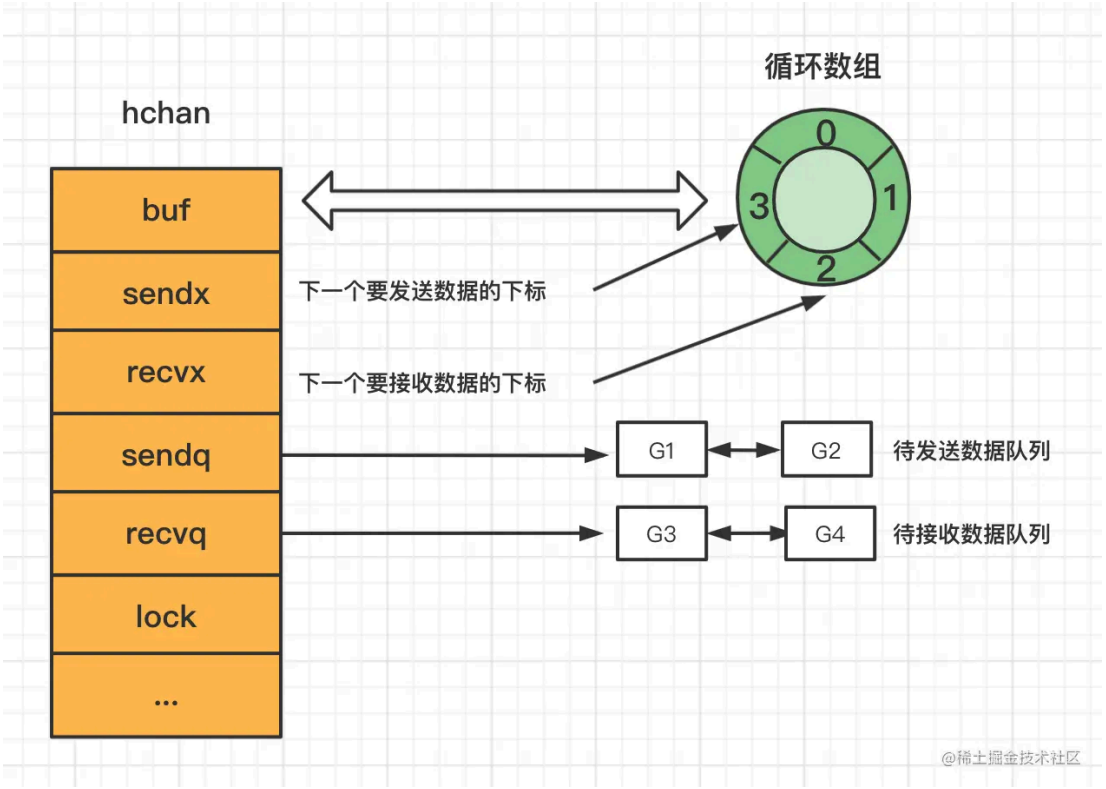
1. Golang 的 Channel，发送一个数据到 Channel 和 从 Channel 接收一个数据 都是 原子性的。
2. 而且 Go 的设计思想就是：不要通过共享内存来通信，而是通过通信来共享内存，前者就是传统的加锁访问文件，后者就是 Channel。
3. 也就是说，设计 Channel 的主要目的就是在多任务间传递数据的，这当然是安全的

2. go channel 的底层实现原理（数据结构）

- Go 面试题 (五)：图解 Golang Channel 的底层原理 - 掘金
- chan-地鼠文档

数据结构

```
type hchan struct {
    //channel 分为无缓冲和有缓冲两种。
    //对于有缓冲的 channel 存储数据，借助的是如下循环数组的结构
    qcount   uint           // 循环数组中的元素数量
    dataqsiz uint           // 循环数组的长度
    buf      unsafe.Pointer // 指向底层循环数组的指针
    elemsize uint16 //能够收发元素的大小


    closed   uint32   //channel 是否关闭的标志
    elemtype *_type //channel 中的元素类型

    //有缓冲 channel 内的缓冲数组会被作为一个“环型”来使用。
    //当下标超过数组容量后会回到第一个位置，所以需要有两个字段记录当前读和写的下标位置
    sendx    uint   // 下一次发送数据的下标位置
    recvx    uint   // 下一次读取数据的下标位置

    //当循环数组中没有数据时，收到了接收请求，那么接收数据的变量地址将会写入读等待队列
    //当循环数组中数据已满时，收到了发送请求，那么发送数据的变量地址将写入写等待队列
    recvq    waitq  // 读等待队列
    sendq    waitq  // 写等待队列


    lock mutex //互斥锁，保证读写 channel 时不存在并发竞争问题
}
```



总结 hchan 结构体的主要组成部分有四个：

- 用来保存 goroutine 之间传递数据的循环链表。--> buf。
- 用来记录此循环链表当前发送或接收数据的下标值。--> sendx 和 recvx。
- 用于保存 向该 chan 发送 和 从该 chan 接收 的 goroutine 的队列。--> sendq 和 recvq
- 保证 channel 写入和读取数据时线程安全的锁。 --> lock

3. nil、关闭的 channel、有数据的 channel，再进行读、写、关闭会怎么样？（各类变种题型，重要）

Channel 读写特性 (15 字口诀)

首先，我们先复习一下 Channel 都有哪些特性？

- 给一个 nil channel 发送数据，造成永远阻塞
- 从一个 nil channel 接收数据，造成永远阻塞
- 给一个已经关闭的 channel 发送数据，引起 panic
- 从一个已经关闭的 channel 接收数据，如果缓冲区中为空，则返回一个零值
- 无缓冲的 channel 是同步的，而有缓冲的 channel 是非同步的

以上 5 个特性是死东西，也可以通过口诀来记忆：“空读写阻塞，写关闭异常，读关闭空零”。

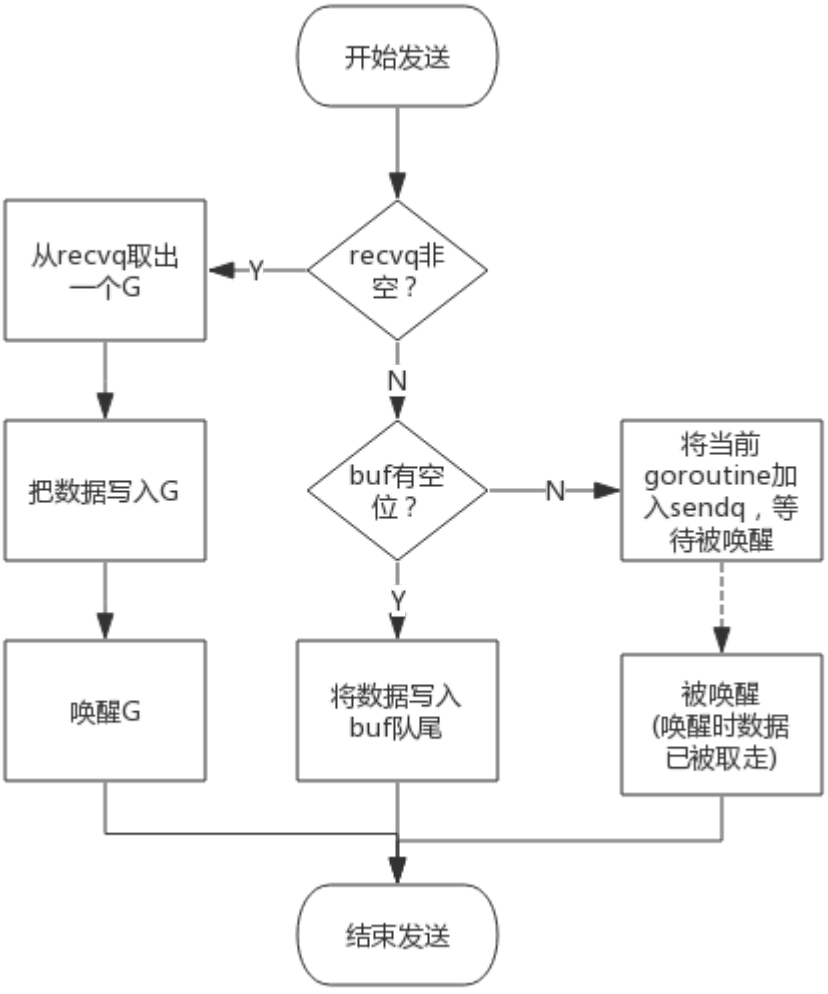
4. 向 channel 发送数据和从 channel 读数据的流程是什么样的?

发送流程:

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:

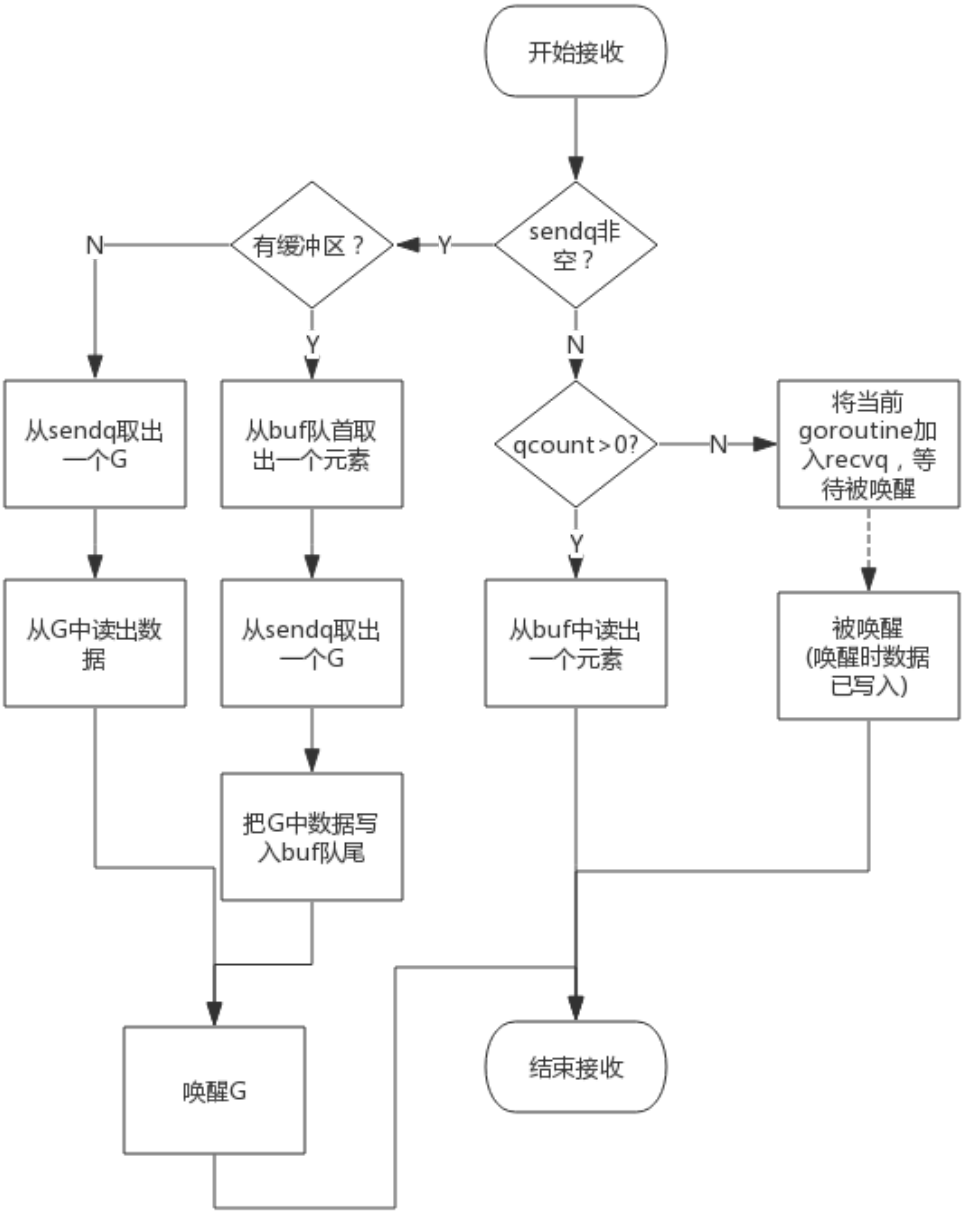


接收流程：

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：



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

5. 讲讲 Go 的 chan 底层数据结构和主要使用场景

答：channel 的数据结构包含 qccount 当前队列中剩余元素个数，dataqsiz 环形队列长度，即可以存放的元素个数，buf 环形队列指针，elemsize 每个元素的大小，closed 标识关闭状态，elemtype 元素类型，sendx 队列下表，指示元素写入时存放到队列中的位置，recv 队列下表，指示元素从队列的该位置读出。recvq 等待读消息的 goroutine 队列，sendq 等待写消息的 goroutine 队列，lock 互斥锁，chan 不允许并发读写。
无缓冲和有缓冲区别：管道没有缓冲区，从管道读数据会阻塞，直到有协程向管道中写入数据。同样，向管道写入数据也会阻塞，直到有协程从管道读取数据。管道有缓冲区但缓冲区没有数据，从管道读取数据也会阻塞，直到协程写入数据，如果管道满了，写数据也会阻塞，直到协程从缓冲区读取数据。
channel 的一些特点

1. 读写值 nil 管道会永久阻塞
2. 关闭的管道读数据仍然可以读数据
3. 往关闭的管道写数据会 panic
4. 关闭为 nil 的管道 panic
5. 关闭已经关闭的管道 panic

向 channel 写数据的流程：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；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；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；

向 channel 读数据的流程：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；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；如果缓冲区中有数据，则从缓冲区取出数据，结束读取过程；将当前 goroutine 加入 recvq，进入睡眠，等待被写 goroutine 唤醒；
使用场景：消息传递、消息过滤，信号广播，事件订阅与广播，请求、响应转发，任务分发，结果汇总，并发控制，限流，同步与异步

6. 有缓存 channel 和无缓存 channel

Go 语言进阶 -- 有缓存 channel 和无缓存 channel

无缓存 channel 适用于数据要求同步的场景，而有缓存 channel 适用于无数据同步的场景。可以根据实现项目需求选择。

七、GPM 相关

- [x] Golang 的协程调度器原理及 GPM 设计思想 - 地鼠文档
- [x] GPM 问题集合 - 幕布- 刘超

0. 进程、线程、协程有什么区别？（必问）

- 进程：是应用程序的启动实例，每个进程都有独立的内存空间，不同的进程通过进程间的通信方式来通信。
- 线程：从属于进程，每个进程至少包含一个线程，线程是 CPU 调度的基本单位，多个线程之间可以共享进程的资源并通过共享内存等线程间的通信方式来通信。
- 协程：为轻量级线程，与线程相比，协程不受操作系统的调度，协程的调度器由用户应用程序提供，协程调度器按照调度策略把协程调度到线程中运行

1. 什么是 GPM？（必问）

- **Goroutine**：协程。就是我们常用的用 `go` 关键字创建的执行体，它对应一个结构体 `g`，结构体中保存了 Goroutine 的栈信息、状态、上下文等调度信息。从 go func () 开始创建一个 Goroutine，新建的 Goroutine 优先保存在 P 的本地队列中，如果 P 的本地队列已经满了（最多 256 个），则会保存到全局队列中。
- **Processor**：表示逻辑处理器，有了它才能建立 G 和 M 之间的联系
    - `P` 是调度的核心组件，结构体为 `runtime.p`。它维护一个本地 Goroutine 队列（run queue）和执行所需的上下文。
    - `P` 负责将 `G` 分配给绑定的 `M` 执行。只有绑定了 `P` 的 `M` 才能运行 `G`，`P` 的数量由 `GOMAXPROCS` 决定。
- **Machine**：表示操作系统线程
    - `M` 是操作系统线程的抽象，结构体为 `runtime.m`。调度器会根据需要动态创建或复用 M。并不是固定只有 `GOMAXPROCS` 个线程，但**同时运行的 M 不会超过 `P` 的数量**。每个活跃的 `M` 需要绑定一个 `P` 才能执行 `G`。
    - `GOMAXPROCS` 表示运行时最多允许多少个 P 并发执行 Goroutine，默认等于逻辑 CPU 数。当每个 P 绑定一个 M 时，可以充分利用多核，避免频繁上下文切换，提高执行效率。

同时运行的 M 线程（有的线程可能挂起） ≤ P 逻辑处理器 = GOMAXPROCS CPU 内核数

M 会从 P 的队列中取一个可执行状态的 G 来执行，如果 P 的本地队列为空，就会从其他的 MP 组合偷取一个可执行的 G 来执行，当 M 执行某一个 G 时候发生系统调用或者阻塞，M 阻塞，如果这个时候 G 在执行，runtime 会把这个线程 M 从 P 中摘除，然后创建一个新的操作系统线程来服务于这个 P，当 M 系统调用结束时，这个 G 会尝试获取一个空闲的 P 来执行，并放入到这个 P 的本地队列，如果这个线程 M 变成休眠状态，加入到空闲线程中，然后整个 G 就会被放入到全局队列中。

G,P,M 的个数问题：

1. G 的个数理论上是无限制的，但是受内存限制，
2. P 的数量一般建议是逻辑 CPU 数量
3. M 的数量
    1. go 语言本身的限制：go 程序启动时，会设置 M 的最大数量，默认 10000. 但是内核很难支持这么多的线程数，所以这个限制可以忽略。
    2. runtime/debug 中的 SetMaxThreads 函数，设置 M 的最大数量
    3. 一个 M 阻塞了，会创建新的 M。
4. M 与 P 的数量没有绝对关系，一个 M 阻塞，P 就会去创建或者切换另一个 M，所以，即使 P 的默认数量是 1，也有可能会创建很多个 M 出来。

带来什么改变

加了 P 之后会带来什么改变呢？我们再更显式的讲一下。

- 每个 P 有自己的本地队列，大幅度的减轻了对全局队列的直接依赖，所带来的效果就是锁竞争的减少。而 GM 模型的性能开销大头就是锁竞争。
- 每个 P 相对的平衡上，在 GPM 模型中也实现了 Work Stealing（工作量窃取机制）算法，如果 P 的本地队列为空，则会从全局队列或其他 P 的本地队列中窃取可运行的 G 来运行，减少空转，提高了资源利用率。

为什么要有 P

这时候就有小伙伴会疑惑了，如果是想实现本地队列、Work Stealing 算法，那为什么不直接在 M 上加呢，M 也照样可以实现类似的组件。为什么又再加多一个 P 组件？

结合 M（系统线程）的定位来看，若这么做，有以下问题：

- 一般来讲，M 的数量都会多于 P。像在 Go 中，M 的数量默认是 10000，P 的默认数量是 CPU 核数。另外由于 M 的属性，也就是如果存在系统阻塞调用，阻塞了 M，又不够用的情况下，M 会不断增加。
- M 不断增加的话，如果本地队列挂载在 M 上，那就意味着本地队列也会随之增加。这显然是不合理的，因为本地队列的管理会变得复杂，且 Work Stealing 性能会大幅度下降。
- M 被系统调用阻塞后，我们是期望把他既有未执行的任务分配给其他继续运行的，而不是一阻塞就导致全部停止。

因此使用 M 是不合理的，那么引入新的组件 P，把本地队列关联到 P 上，就能很好的解决这个问题。

3. 调度器的设计策略

复用线程：避免频繁的创建、销毁线程，而是对线程的复用。

1. work stealing（工作量窃取）机制
   当本线程无可运行的 G 时，尝试从其他线程绑定的 P 偷取 G，而不是销毁线程。
2. hand off（移交）机制
   当本线程因为 G 进行系统调用阻塞时，线程释放绑定的 P，把 P 转移给其他空闲的线程执行。
   利用并行：GOMAXPROCS 设置 P 的数量，最多有 GOMAXPROCS 个线程分布在多个 CPU 上同时运行。GOMAXPROCS 也限制了并发的程度，比如 GOMAXPROCS = 核数/2，则最多利用了一半的 CPU 核进行并行。
   抢占：在 coroutine 中要等待一个协程主动让出 CPU 才执行下一个协程。在 Go 中，**一个 goroutine 最多占用 CPU 10ms**，防止其他 goroutine 被饿死。这就是 goroutine 不同于 coroutine 的一个地方。
   全局 G 队列：在新的调度器中依然有全局 G 队列，但功能已经被弱化了，当 M 执行 work stealing 从其他 P 偷不到 G 时，它可以从全局 G 队列获取 G。

3. 抢占式调度是如何抢占的?

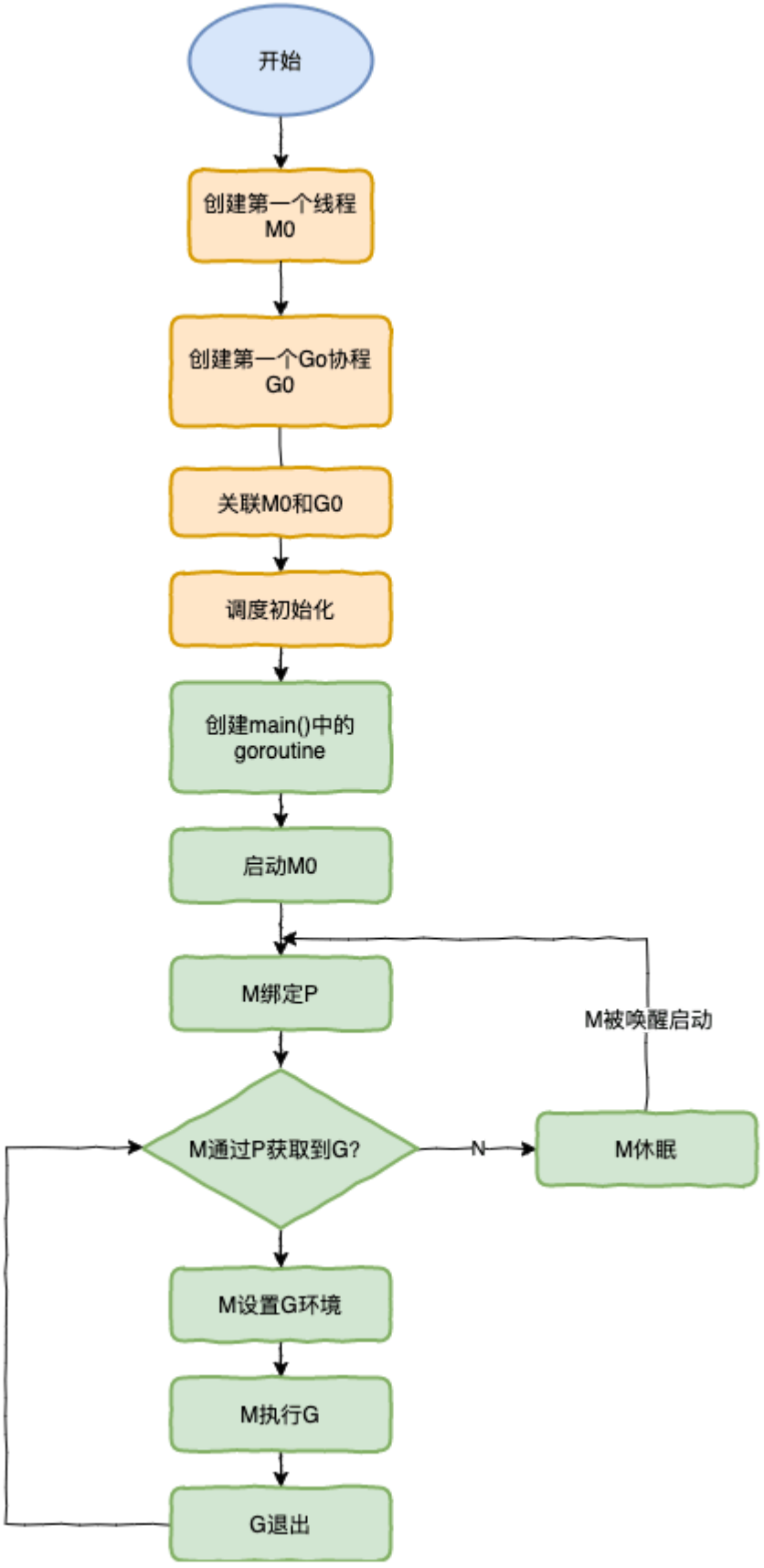
基于协作式抢占/基于信号量抢占

就像操作系统要负责线程的调度一样，Go 的 runtime 要负责 goroutine 的调度。现代操作系统调度线程都是抢占式的，我们不能依赖用户代码主动让出 CPU，或者因为 IO、锁等待而让出，这样会造成调度的不公平。基于经典的时间片算法，当线程的时间片用完之后，会被时钟中断给打断，调度器会将当前线程的执行上下文进行保存，然后恢复下一个线程的上下文，分配新的时间片令其开始执行。这种抢占对于线程本身是无感知的，系统底层支持，不需要开发人员特殊处理。

基于时间片的抢占式调度有个明显的优点，能够避免 CPU 资源持续被少数线程占用，从而使其他线程长时间处于饥饿状态。goroutine 的调度器也用到了时间片算法，但是和操作系统的线程调度还是有些区别的，因为整个 Go 程序都是运行在用户态的，所以不能像操作系统那样利用时钟中断来打断运行中的 goroutine。也得益于完全在用户态实现，goroutine 的调度切换更加轻量。

上面这两段文字只是对调度的一个概括，具体的协作式调度、信号量调度大家还需要去详细了解，这偏底层了，大厂或者中高级开发会问。（字节就问了）

4. 调度器的生命周期


开始
创建第一个线程M0
创建第一个Go协程G0
关联M0和G0
调度初始化
创建main()中的goroutine
启动M0
M绑定P
M通过P获取到G?
N
M休眠
M被唤醒启动
M设置G环境
M执行G
G退出

M0 和 G0

名称	作用	位置
M0	程序启动时创建的首个 OS 线程；启动、初始化阶段由它完成	结构体 `runtime.m0` 静态分配
G0	每启动一个 M 都会创建和持有的调度专用 Goroutine，仅用作调度 Goroutine，在调度或系统调用时会使用 G0 的栈空间，全局变量的 G0 是 M0 的 G0	`m.g0` 字段指向自己的 G0

我们来跟踪一段代码

```
package main
import "fmt"
func main() {
    fmt.Println("Hello world")
}
```

接下来我们来针对上面的代码对调度器里面的结构做一个分析。

如上图所示的过程：

1. runtime 创建最初的线程 m0 和 goroutine g0，并把二者关联。
2. 调度器初始化：初始化 m0、栈、垃圾回收，以及创建和初始化由 GOMAXPROCS 个 P 构成的 P 列表。
3. 代码中的 main 函数是 main.main，runtime 中也有一个 main 函数——runtime.main，代码经过编译后，runtime.main 会调用 main.main，程序启动时会为 runtime.main 创建 goroutine，称它为 main goroutine 吧，然后把 main goroutine 加入到 P 的本地队列。
4. 启动 m0，m0 已经绑定了 P，会从 P 的本地队列获取 G，获取到 main goroutine。
5. G 拥有栈，M 根据 G 中的栈信息和调度信息设置运行环境
6. M 运行 G
7. G 退出，再次回到 M 获取可运行的 G，这样重复下去，直到 main.main 退出，runtime.main 执行 Defer 和 Panic 处理，或调用 runtime.exit 退出程序。

调度器的生命周期几乎占满了一个 Go 程序的一生，runtime.main 的 goroutine 执行之前都是为调度器做准备工作，runtime.main 的 goroutine 运行，才是调度器的真正开始，直到 runtime.main 结束而结束。

八、锁相关

mutex-地鼠文档

rwmutex-地鼠文档

除了 mutex 以外还有那些方式安全读写共享变量?

- 将共享变量的读写放到一个 goroutine 中，其它 goroutine 通过 channel 进行读写操作。
- 可以用个数为 1 的信号量（semaphore）实现互斥

Go 如何实现原子操作?

答：原子操作就是不可中断的操作，外界是看不到原子操作的中间状态，要么看到原子操作已经完成，要么看到原子操作已经结束。在某个值的原子操作执行的过程中，CPU 绝对不会再去执行其他针对该值的操作，那么其他操作也是原子操作。

Go 语言的标准库代码包 sync/atomic 提供了原子的读取（Load 为前缀的函数）或写入（Store 为前缀的函数）某个值（这里细节还要多去查查资料）。

原子操作与互斥锁的区别

1. 互斥锁是一种数据结构，用来使互斥的多个操作不同时进行。它未必是对单个量的读写，它的重点在于互斥的多个操作。
2. 原子操作是针对某个值的单个互斥操作。

Mutex 是悲观锁还是乐观锁？悲观锁、乐观锁是什么?

悲观锁：当要对数据库中的一条数据进行修改的时候，为了避免同时被其他人修改，最好的办法就是直接对该数据进行加锁以防止并发。这种借助数据库锁机制，在修改数据之前先锁定，再修改的方式被称之为悲观并发控制。【Pessimistic Concurrency Control，缩写“PCC”，又名“悲观锁”】。

- 锁机制带来**上下文切换开销**
- 容易导致**死锁**、**性能下降**

乐观锁：是相对悲观锁而言的，乐观锁假设数据**不会被并发修改**，直接执行操作，提交前通过 **版本号或时间戳** 等方式检查是否有其他修改。乐观锁适用于读多写少的场景，这样可以提高程序的吞吐量。

Mutex 有几种模式?

正常模式

1. 当前的 mutex 只有一个 goruntine 来获取，那么没有竞争，直接返回。
2. 新的 goruntine 进来，如果当前 mutex 已经被获取了，则该 goruntine 进入一个先入先出的 waiter 队列，在 mutex 被释放后，waiter 按照先进先出的方式获取锁。该 goruntine 会处于自旋状态 (不挂起，继续占有 cpu)。

3. 新的 goruntine 进来，mutex 处于空闲状态，将参与竞争。新来的 goroutine 有先天的优势，它们正在 CPU 中运行，可能它们的数量还不少，所以，在高并发情况下，被唤醒的 waiter 可能比较悲剧地获取不到锁，这时，它会被插入到队列的前面。如果 waiter 获取不到锁的时间超过阈值 1 毫秒，那么，这个 Mutex 就进入到了饥饿模式。

饥饿模式

在饥饿模式下，Mutex 的拥有者将直接把锁交给队列最前面的 waiter。新来的 goroutine 不会尝试获取锁，即使看起来锁没有被持有，它也不会去抢，也不会 spin（自旋），它会乖乖地加入到等待队列的尾部。如果拥有 Mutex 的 waiter 发现下面两种情况的其中之一，它就会把这个 Mutex 转换成正常模式：

1. 此 waiter 已经是队列中的最后一个 waiter 了，没有其它的等待锁的 goroutine 了；
2. 此 waiter 的等待时间小于 1 毫秒。

sync.Mutex

`sync.Mutex` 是 Go 语言标准库 `sync` 包中的一个互斥锁类型，用于在多个 goroutine 之间同步对共享资源的访问。当多个 goroutine 需要访问同一个资源时，使用 `sync.Mutex` 可以确保在任何时刻只有一个 goroutine 能够访问该资源，从而避免数据竞争和不一致性的问题。

主要特点

- **互斥性**：在任何时刻，只有一个 goroutine 可以持有 `sync.Mutex` 的锁。如果多个 goroutine 尝试同时获取锁，那么除了第一个成功获取锁的 goroutine 之外，其他 goroutine 将被阻塞，直到锁被释放。
- **非重入性**：如果一个 goroutine 已经持有了 `sync.Mutex` 的锁，那么它不能再次请求这个锁，这会导致死锁。

方法

`sync.Mutex` 提供了两个主要方法：

- `Lock()`：尝试获取锁。如果锁已经被其他 goroutine 持有，则调用者将阻塞，直到锁被释放。
- `Unlock()`：释放锁。调用此方法之前必须先成功调用 `Lock()`。如果在一个没有锁的 `sync.Mutex` 上调用 `Unlock()`，将会导致 panic。

使用场景

`sync.Mutex` 适用于需要严格互斥访问共享资源的场景。例如，在并发编程中，如果有多个 goroutine 需要修改同一个数据结构或访问同一个文件，就应该使用 `sync.Mutex` 来确保操作的原子性和数据的一致性。

sync.RWMutex

`sync.RWMutex` 是 Go 语言标准库 `sync` 包中的一个类型，它实现了读写互斥锁（Reader-Writer Mutex）。与普通的互斥锁（如 `sync.Mutex`）相比，`sync.RWMutex` 允许多个读操作同时进行，但写操作会完全互斥。这意味着在任何时刻，可以有多个 goroutine 同时读取某个资源，但写入资源时，必须保证没有其他 goroutine 在读取或写入该资源。

主要特点

- **多个读者，单一写者**：允许多个读操作并发执行，但写操作会阻塞所有其他读写操作。
- **优化读性能**：通过允许多个读操作同时进行，提高了读操作的并发性能。
- **写操作独占性**：写操作在执行时会阻止所有其他读写操作，确保数据的一致性和完整性。

方法

`sync.RWMutex` 提供了以下主要方法：

- `Lock()`：加写锁。如果锁已被其他 goroutine 获取（无论是读锁还是写锁），则调用者将阻塞，直到锁被释放。
- `Unlock()`：释放写锁。调用此方法之前必须先成功调用 `Lock()`。
- `RLock()`：加读锁。如果锁已被其他 goroutine 获取为写锁，则调用者将阻塞，但如果有其他 goroutine 持有读锁，则调用者可以立即获取读锁。
- `RUnlock()`：释放读锁。调用此方法之前必须先成功调用 `RLock()`。

使用场景

`sync.RWMutex` 适用于读多写少的场景，可以显著提高程序的并发性能。例如，在缓存系统、配置管理系统等场景中，读操作远多于写操作，使用 `sync.RWMutex` 可以在保证数据一致性的同时，提高读操作的并发性。

什么是自旋锁

自旋锁是指当一个线程（在 Go 中是 Goroutine）在获取锁的时候，如果锁已经被其他线程获取，那么该线程将循环等待（自旋），不断判断锁是否已经被释放，而不是进入睡眠状态。这种行为在某些情况下可能会导致资源的过度占用，特别是当锁持有时间较长或者自旋的 Goroutine 数量较多时。

自旋锁的**核心思想**是，如果预期锁很快就会被释放（即锁持有时间很短），那么让线程持续运行并检查锁的状态，而不是进入睡眠和唤醒的昂贵操作，可能会更加高效。然而，如果锁被长时间持有，或者多个线程同时竞争锁，自旋锁可能会导致大量的 CPU 时间被浪费在无效的循环等待上，这种情况称为“自旋”。

在 Go 语言中，虽然标准库中没有直接提供自旋锁的实现，但开发者可以**通过原子操作**和其他同步原语来实现自定义的自旋锁。然而，由于自旋锁可能导致 CPU 资源的过度占用，因此在决定使用自旋锁之前，应该仔细考虑其适用性和潜在的性能影响。在许多情况下，使用互斥锁或其他更高级的同步机制可能是更好的选择。

go 里面怎么实现一个自旋锁

其实就是使用 for 循环不断尝试一个操作，直到成功。要求这个操作是线程安全的。

在 Go 语言中，实现一个自旋锁通常涉及使用原子操作来确保对锁状态的并发访问是安全的。下面是一个简单的自旋锁实现的例子：

```go
package main

import (
    "sync/atomic"
    "time"
)

type Spinlock struct {
    locked int32
}

func (s *Spinlock) Lock() {
    for !atomic.CompareAndSwapInt32(&s.locked, 0, 1) {
        // 这里可以添加一些退避策略，比如随机等待一段时间，以避免过多的 CPU 占用
        // time.Sleep(time.Nanosecond) // 注意：实际使用中可能不需要或想要这样的退避
    }
}

func (s *Spinlock) Unlock() {
    atomic.StoreInt32(&s.locked, 0)
}

func main() {
    var lock Spinlock

    // 示例：使用自旋锁
    go func() {
        lock.Lock()
        // 执行一些操作...
        lock.Unlock()
    }()

    // 在另一个 goroutine 中尝试获取锁
    go func() {
        lock.Lock()
        // 执行一些操作...
        lock.Unlock()
    }()

    // 等待足够的时间以确保 goroutines 完成
    time.Sleep(time.Second)
```

```
42 }
```

在这个例子中，

1. `Spinlock` 结构体有一个 `int32` 类型的字段 `locked`，用于表示锁的状态。
2. `Lock` 方法使用 `atomic.CompareAndSwapInt32` 原子操作来尝试将 `locked` 从 0（未锁定）更改为 1（已锁定）。如果锁已经被另一个 goroutine 持有（即 `locked` 为 1.，则 `CompareAndSwapInt32` 会返回 `false`，并且循环会继续。
3. `Unlock` 方法使用 `atomic.StoreInt32` 原子操作将 `locked` 设置回 0，表示锁已被释放。

需要注意的是，在实际应用中，自旋锁可能会导致 CPU 资源的过度占用，特别是在锁被长时间持有或存在大量竞争的情况下。因此，在使用自旋锁之前，应该仔细考虑其适用性和潜在的性能影响。在许多情况下，使用互斥锁（`sync.Mutex`）或其他更高级的同步机制可能是更好的选择。

什么情况下会更改失败

在自旋锁的实现中，更改失败通常指的是尝试获取锁时未能成功将锁的状态从“未锁定”更改为“已锁定”。这种情况通常发生在以下几种情境中：

锁已被其他线程持有

- 当一个线程尝试通过自旋锁获取对共享资源的访问权时，如果该锁当前已被另一个线程持有，那么尝试更改锁状态的原子操作（如`atomic.CompareAndSwap`）将失败，因为条件（锁为未锁定状态）不满足。

竞争条件

- 在多线程环境中，多个线程可能几乎同时尝试获取同一个自旋锁。由于 CPU 调度和线程执行的并发性，这些尝试可能几乎同时发生，导致多个线程在锁被释放后立即尝试获取它。尽管自旋锁设计用于快速响应锁状态的更改，但在高竞争条件下，仍然可能存在多个线程同时看到锁为未锁定状态的情况，从而导致多个更改尝试中只有一个成功。

解决方案

- **设置自旋次数限制**：为了避免在锁被长时间持有时浪费 CPU 资源，可以为自旋锁设置自旋次数的限制。一旦达到该限制，尝试获取锁的线程将放弃自旋并进入睡眠状态，等待锁被释放。
- **使用退避策略**：在自旋期间，可以尝试使用退避策略（如指数退避），以减少 CPU 的占用率并提高系统的整体性能。
- **考虑使用其他同步机制**：如果自旋锁不适用于特定场景（如锁持有时间较长、竞争激烈等），则可以考虑使用其他同步机制（如互斥锁、读写锁等）。

总之，自旋锁更改失败通常是由于锁已被其他线程持有、竞争条件、系统或硬件限制以及编程错误等原因导致的。为了解决这个问题，可以采取设置自旋次数限制、使用退避策略以及考虑使用其他同步机制等措施。

goroutine 的自旋占用资源如何解决

Goroutine 的自旋占用资源问题主要涉及到 Goroutine 在等待锁或其他资源时的一种行为模式，即自旋锁（spinlock）。自旋锁是指当一个线程（在 Go 中是 Goroutine）在获取锁的时候，如果锁已经被其他线程获取，那么该线程将循环等待（自旋），不断判断锁是否已经被释放，而不是进入睡眠状态。这种行为在某些情况下可能会导致资源的过度占用，特别是当锁持有时间较长或者自旋的 Goroutine 数量较多时。
针对 Goroutine 的自旋占用资源问题，可以从以下几个方面进行解决或优化：

1. **减少自旋锁的使用**
   评估必要性：首先评估是否真的需要使用自旋锁。在许多情况下，互斥锁（mutex）已经足够满足需求，因为互斥锁在资源被占用时会让调用者进入睡眠状态，从而减少对 CPU 的占用。
   优化锁的设计：考虑使用更高级的同步机制，如读写锁（rwmutex），它允许多个读操作同时进行，而写操作则是互斥的。这可以显著减少锁的竞争，从而降低自旋的需求。

2. **优化自旋锁的实现**
   设置自旋次数限制：在自旋锁的实现中加入自旋次数的限制，当自旋达到一定次数后，如果仍未获取到锁，则让 Goroutine 进入睡眠状态。这样可以避免长时间的无效自旋，浪费 CPU 资源。
   利用 Go 的调度器特性：Go 的调度器在检测到 Goroutine 长时间占用 CPU 而没有进展时，会主动进行抢占式调度，将 Goroutine 暂停并让出 CPU。这可以在一定程度上缓解自旋锁带来的资源占用问题。

3. 监控和调整系统资源
   监控系统性能：通过工具（如 pprof、statsviz 等）监控 Go 程序的运行时性能，包括 CPU 使用率、内存占用等指标。这有助于及时发现和解决资源占用过高的问题。
   调整 Goroutine 数量：根据系统的负载情况动态调整 Goroutine 的数量。例如，在高并发场景下适当增加 Goroutine 的数量以提高处理能力，但在负载降低时及时减少 Goroutine 的数量以避免资源浪费。

4. 利用 Go 的并发特性
   充分利用多核 CPU：通过设置 runtime.GOMAXPROCS 来指定 Go 运行时使用的逻辑处理器数量，使其尽可能接近或等于物理 CPU 核心数，从而充分利用多核 CPU 的并行处理能力。
   使用 Channel 进行通信：Go 鼓励使用 Channel 进行 Goroutine 之间的通信和同步，而不是直接使用锁。Channel 可以有效地避免死锁和竞态条件，并且减少了锁的使用，从而降低了资源占用的风险。
   综上所述，解决 Goroutine 的自旋占用资源问题需要从多个方面入手，包括减少自旋锁的使用、优化自旋锁的实现、监控和调整系统资源以及充分利用 Go 的并发特性等。通过这些措施的综合应用，可以有效地降低 Goroutine 在自旋过程中对系统资源的占用。

九、并发相关

Go 中主协程如何等待其余协程退出?

Go 的 sync.WaitGroup 是等待一组协程结束，sync.WaitGroup 只有 3 个方法，Add() 是添加计数，Done() 减去一个计数，Wait() 阻塞直到计数器归零。

有缓冲 channel 可以让多个 goroutine 并发执行，**在任务完成后将结果塞入 channel 中**，主线程阻塞地读取 channel 中的消息即可判断 goroutine 是否结束。但 channel 并不保证接收顺序 == 启动顺序。

无缓冲 channel 的发送方必须等接收方接收才能继续执行，多个 goroutine 如果串行地从主线程一个一个 receive，就变成 **串行协程。**

啰嗦一句：循环智能二面，手写代码部分时，三个协程按交替顺序打印数字，最后题目做出来了，问我代码中 Add() 是什么意思，我回答的不是很清晰，这家公司就没有然后了。Add() 表示协程计数，可以一次 Add 多个，如 Add(3),可以多次 Add(1);然后每个子协程必须调用 done(),这样才能保证所有子协程结束，主协程才能结束。

怎么控制并发数?

第一，有缓冲通道

根据通道中没有数据时读取操作陷入阻塞和通道已满时继续写入操作陷入阻塞的特性，正好实现控制并发数量。

```
func main() {
    count := 10                        // 最大支持并发
    sum := 100                         // 任务总数
    wg := sync.WaitGroup{}             // 控制主协程等待所有子协程执行完之后再退出。
    c := make(chan struct{}, count) // 控制任务并发的 chan
    defer close(c)
    for i := 0; i < sum; i++ {
        wg.Add(1)
        c <- struct{}{}                // 作用类似于 waitgroup.Add(1)
        go func(j int) {
            defer wg.Done()
            fmt.Println(j)
            <-c // 执行完毕，释放资源
        }(i)
    }
    wg.Wait()
}
```

第二，三方库实现的协程池

```
import (
    "github.com/Jeffail/tunny"
```

```
    "log"
    "time"
)
func main() {
    pool := tunny.NewFunc(10, func(i interface{}) interface{} {
        log.Println(i)
        time.Sleep(time.Second)
        return nil
    })
    defer pool.Close()
    for i := 0; i < 500; i++ {
        go pool.Process(i)
    }
    time.Sleep(time.Second * 4)
}
```

多个 goroutine 对同一个 map 写会 panic，异常是否可以用 defer 捕获?

可以使用 `defer` + `recover()` 捕获 panic，但这并不是解决并发写 map 的根本方式，只能避免程序崩溃，让你“吞掉”这个错误继续运行，但 map 本身的状态可能已被破坏。

Go 中，对异常处理的原则是：多用 error 包，少用 panic

```
defer func() {
    if err := recover(); err != nil {
        // 打印异常，关闭资源，退出此函数
        fmt.Println(err)
    }
}()
```

如何优雅的实现一个 goroutine 池

（百度、手写代码，本人面传音控股被问道：请求数大于消费能力怎么设计协程池）

这一块能啃下来，offer 满天飞，这应该是保证高并发系统稳定性、高可用的核心部分之一。

1. 为什么需要协程池?

   外界滥用（例如高并发场景）：如果无休止的开辟 Goroutine，虽然 Goroutine 是轻量级线程，但高并发下仍会触发调度器的频繁抢占，导致 CPU cache 失效和调度成本上升，带来上下文切换的开销。所以设计一个 Goroutine 池限制 Goroutine 的开辟个数在大型并发场景还是必要的。

   内部滥用（开发场景）：一个项目越复杂，交接次数越多，后续的接手者越不愿意修改主逻辑。而一旦有一些非主逻辑的业务，我们都倾向于开启独立代码分支逻辑，同时封装为独立的协程来完成。但是这种不断叠加分支逻辑、不断增加独立协程的方式本质上就是一种协程滥用，我们不断增加协程数，忽略了协程的本身开销和上下文切

换成本，很容易造成一个进程的 goroutine 数量过多，内存增加。不仅如此，这种做法还必须要保证分支代码质量。一个代码分支写的质量不行（比如没有设置 ctx 超时卡在 io 请求上），那么新启动的 goroutine 长时间无法释放，这就可能导致 goroutine 的泄露（**泄露**：不退出、不释放资源、不可控）。

2. 协程池的原理

   使用协程池来控制了协程数，一旦协程池满了之后，想新创建一个协程，只有两个结果：阻塞或失败（也就是返回错误）。还可能结合阻塞和失败，先阻塞，设置一个超时时间，当时间到了仍然阻塞则返回错误。

   新的协程太多，还需要设置超时时间，避免长期等待。其实协程池也使用 channel 来阻塞，把任务塞进 带缓存的 channel，如果缓存满了就阻塞。当然也可以返回 失败错误 。

3. ants（推荐）
   ants 是一个优秀的协程池库，它从原理上有以下优化：

   - 协程复用：创建协程是使用 `go func()` 来创建，协程的创建和销毁有比较大的开销（相对而言，在高并发场景）所以我们不再多次创建协程，而是复用协程。我们只需要把这个 `func()` 用闭包包起来，把它通过 channel 给现有的 goroutine 去执行。可以减少协程创建销毁的开销。
   - 动态调整 协程池的大小。
   - 指数退避自旋锁。在访问协程池共享资源（如可复用 worker 队列）时，为避免锁竞争造成调度阻塞，如果获取不到锁，会使用指数退避的原则，在指数时间后再次尝试。等待期间会把调度器让出。

golang 实现多并发请求（发送多个 get 请求）

在go 语言中其实有两种方法进行协程之间的通信。**一个是共享内存、一个是消息传递。**

1. 通过共享内存来通信。这是所有语言都通用的方式。设置一个原子读写的变量，多个线程互斥地去访问，就可以通过它来通信。这叫通过共享内存（也就是这个变量）来通信。
2. **通过通信来共享内存**，指的是 Go 使用 `channel` 在 Goroutine 之间传递变量的引用或值，从而 **避免多个 Goroutine 同时访问同一块内存**，实现了“数据所有权”的转移。因此称为通过通信（channel）来共享内存（即将对数据的访问权从一个线程转移到另一个）。

共享内存（互斥锁）

```go
//基本的 GET 请求
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "time"
    "sync"
```

```
    "runtime"
)

// 计数器
var counter int = 0

func httpget(lock *sync.Mutex){
    lock.Lock()
    counter++
    resp, err := http.Get("http://localhost:8000/rest/api/user")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer resp.Body.Close()
    body, err := ioutil.ReadAll(resp.Body)
    fmt.Println(string(body))
    fmt.Println(resp.StatusCode)
    if resp.StatusCode == 200 {
        fmt.Println("ok")
    }
    lock.Unlock()
}

func main() {
    start := time.Now()
    lock := &sync.Mutex{}
    for i := 0; i < 800; i++ {
        go httpget(lock)
    }
    for  {
        lock.Lock()
        c := counter
        lock.Unlock()
        runtime.Gosched()
        if c >= 800 {
            break
        }
    }
    end := time.Now()
    consume := end.Sub(start).Seconds()
    fmt.Println("程序执行耗时 (s): ", consume)
}
```

问题

我们可以看到共享内存的方式是可以做到并发，但是我们需要利用共享变量来进行协程的通信，也就需要使用互斥锁来确保数据安全性，导致代码啰嗦，复杂化，不易维护。我们后续使用 go 的消息传递方式避免这些问题。

消息传递（管道）

```go
//基本的 GET 请求
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "time"
)
// HTTP get 请求
func httpget(ch chan int){
    resp, err := http.Get("http://localhost:8000/rest/api/user")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer resp.Body.Close()
    body, err := ioutil.ReadAll(resp.Body)
    fmt.Println(string(body))
    fmt.Println(resp.StatusCode)
    if resp.StatusCode == 200 {
        fmt.Println("ok")
    }
    ch <- 1
}
// 主方法
func main() {
    start := time.Now()
    // 注意设置缓冲区大小要和开启协程的个人相等
    chs := make([]chan int, 2000)
    for i := 0; i < 2000; i++ {
        chs[i] = make(chan int)
        go httpget(chs[i])
    }
    for _, ch := range chs {
        <- ch
    }
    end := time.Now()
    consume := end.Sub(start).Seconds()
    fmt.Println("程序执行耗时 (s): ", consume)
}
```

总结：我们通过go 语言的管道 channel 来实现并发请求，能够解决如何避免传统共享内存实现并发的很多问题而且效率会高于共享内存的方法。

sync.pool

`sync.Pool` 是 Go 语言在标准库 `sync` 包中提供的一个类型，它主要用于存储和复用临时对象，以减少内存分配的开销，提高性能。以下是对 `sync.Pool` 的详细解析：

基本概念

`sync.Pool` 是一个**可以存储任意类型**的临时对象的集合。当你需要一个新的对象时，可以先从 `sync.Pool` 中尝试获取；如果 `sync.Pool` 中有可用的对象，则直接返回该对象；如果没有，则需要自行创建。使用完对象后，可以将其放回 `sync.Pool` 中，以供后续再次使用。

主要特点

1. **减少内存分配和垃圾回收（GC）压力**：通过复用已经分配的对象，`sync.Pool` 可以显著减少内存分配的次数，从而减轻 GC 的压力，提高程序的性能。
2. **并发安全**：`sync.Pool` 是 Goroutine 并发安全的，多个 Goroutine 可以同时从 `sync.Pool` 中获取和放回对象，而无需额外的同步措施。
3. **自动清理**：Go 的垃圾回收器在每次垃圾回收时，都会清除 `sync.Pool` 中的所有对象。因此，你不能假设一个对象被放入 `sync.Pool` 后就会一直存在。

使用场景

`sync.Pool` 适用于以下场景：

- 对象实例创建开销较大的场景，如数据库连接、大型数据结构等。
- 需要频繁创建和销毁临时对象的场景，如 HTTP 处理函数中频繁创建和销毁的请求上下文对象。

使用方法

1. **创建 Pool 实例**：首先，你需要创建一个 `sync.Pool` 的实例，并配置 `New` 方法。`New` 方法是一个无参函数，用于在 `sync.Pool` 中没有可用对象时创建一个新的对象。

```go
var pool = &sync.Pool{
    New: func() interface{} {
        return new(YourType) // 替换 YourType 为你的类型
    },
}
```

2. **获取对象**：使用 `Get` 方法从 `sync.Pool` 中获取对象。`Get` 方法会返回 `sync.Pool` 中已经存在的对象（如果存在的话），或者调用 `New` 方法创建一个新的对象。

```go
obj := pool.Get().(*YourType) // 替换 YourType 为你的类型，并进行类型断言
```

3. **使用对象**：获取到对象后，你可以像使用普通对象一样使用它。

4. **放回对象**：使用完对象后，使用 `Put` 方法将对象放回 `sync.Pool` 中，以供后续再次使用。

```
pool.Put(obj)
```

注意事项

1. **对象状态未知**：从 `sync.Pool` 中获取的对象的状态是未知的。因此，在使用对象之前，你应该将其重置到适当的初始状态。
2. **自动清理**：由于 Go 的垃圾回收器会清理 `sync.Pool` 中的对象，因此你不能依赖 `sync.Pool` 来长期存储对象。
3. **不适合所有场景**：`sync.Pool` 并不适合所有需要对象池的场景。特别是对于那些需要精确控制对象生命周期的场景，你可能需要实现自定义的对象池。

总的来说，`sync.Pool` 是 Go 语言提供的一个非常有用的工具，它可以帮助你减少内存分配和垃圾回收的开销，提高程序的性能。然而，在使用时需要注意其特性和局限，以免发生不可预见的问题。

十、GC 相关

1. 垃圾回收原理 - 地鼠文档
2. Golang 三色标记 + 混合写屏障 GC 模式全分析 - 地鼠文档
    - **算法**：Golang 采用三色标记清扫法进行垃圾回收，以减少 STW（Stop The World）的时间。**写屏障技术**被用来避免在并发标记过程中产生的误清扫问题。
    - **触发条件**：垃圾回收的触发条件包括内存分配达到一定比例、长时间未触发 GC、手动调用 runtime.GC () 等。

GC 算法有四种：

- **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）

- **节点复制**：节点复制也是基于追踪的算法。其将整个堆等分为两个半区（semi-space），一个包含现有数据，另一个包含已被废弃的数据。节点复制式垃圾收集从切换（flip）两个半区的角色开始，然后收集器在老的半区，也就是 Fromspace 中遍历存活的数据结构，在第一次访问某个单元时把它复制到新半区，也就是 Tospace 中去。在 Fromspace 中所有存活单元都被访问过之后，收集器在 Tospace 中建立一个存活数据结构的副本，用户程序可以重新开始运行了。
    - 优点：
        - 所有存活的数据结构都缩并地排列在 Tospace 的底部，这样就不会存在内存碎片的问题
        - 获取新内存可以简单地通过递增自由空间指针来实现。
    - 缺点：内存得不到充分利用，总有一半的内存空间处于浪费状态。
- **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

Go 垃圾回收机制的演变

Go 的 GC 回收有三次演进过程，Go V1.3 之前普通标记清除（mark and sweep）方法，整体过程需要启动 STW，效率极低。GoV1.5 三色标记法，堆空间启动写屏障，栈空间不启动，全部扫描之后，需要重新扫描一次栈 (需要 STW)，效率普通。GoV1.8 三色标记法，混合写屏障机制：栈空间不启动（全部标记成黑色），堆空间启用写屏障，整个过程不要 STW，效率高。

Go 1.3 及之前

- **Stop-the-World (STW)**: 在这些早期版本中，垃圾回收是全局停止的，即在进行垃圾回收时，所有应用程序的 goroutine 都会暂停。这种方式导致较长的停顿时间，对性能有显著影响。

Go 1.5

- **三色标记清除和并发标记**: 引入了并发标记阶段，使得标记过程可以与应用程序的执行并发进行。虽然依然有停顿，但停顿时间显著减少。
- **写屏障**: 实现了写屏障技术，确保在并发标记过程中对活动对象的准确追踪。

Go 1.8

- **Hybrid Barrier**: 引入**混合屏障机制**，结合了写屏障和标记终止，进一步减少了 STW 时间。
- **非阻塞回收**: 清除过程变为完全并发，减少了垃圾回收对应用程序的影响。

Go 1.9

- **Pacer 改进**: 对垃圾回收的节奏器（Pacer）进行了改进，使得垃圾回收周期更均匀，减少了突发性停顿。
- **Sweep Termination 改进**: 改进了清除终止阶段的并发性，进一步减少停顿时间。

Go 1.10

- **并行清除**: 增加了对并行清除的支持，多个清除操作可以并行执行，提高了清除效率。

Go 1.11

- **更好的内存分配器**: 优化了内存分配器，减少了垃圾回收压力。
- **对象再利用**: 增强了对对象再利用的支持，减少了内存分配次数，从而降低了垃圾回收频率。

Go 1.12

- **内存剖析器改进**: 引入新的内存剖析器，提供更精确的内存使用报告，帮助开发者更好地优化内存使用。

Go 1.13 - Go 1.15

- **进一步优化 Pacer**: 持续改进 Pacer，确保垃圾回收的平滑进行，减少对应用程序的影响。
- **优化并发性**: 增强了垃圾回收的并发性，进一步减少了停顿时间。

Go 1.16 及以后

- **内存使用优化**: 持续改进内存分配和垃圾回收算法，提高内存使用效率。
- **低延迟 GC**: 目标是在保持高吞吐量的同时，将垃圾回收的停顿时间进一步降低。
- **更智能的内存管理**: 引入更多智能化的内存管理策略，动态调整垃圾回收参数，以适应不同的工作负载。

总结

Go 语言的垃圾回收机制随着版本的演变不断优化，从早期的全局停止到现在的并发标记和清除，逐步减少了垃圾回收对应用程序性能的影响。未来的版本将继续致力于降低垃圾回收的停顿时间和提高内存管理效率，为开发者提供更高效、更稳定的运行环境。

三色标记法的流程

为什么需要三色标记?

三色标记的目的，主要是利用 Tracing GC (Tracing GC 是垃圾回收的一个大类，另外一个大类是引用计数) 做增量式垃圾回收，降低最大暂停时间。原生 Tracing GC 只有黑色和白色，没有中间的状态，这就要求 GC 扫描过程必须一次性完成，得到最后的黑色和白色对象。在前面增量式 GC 中介绍到了，这种方式会存在较大的暂停时间。

三色标记增加了中间状态灰色，增量式 GC 运行过程中，应用线程的运行可能改变了对象引用树，只要让黑色对象直接引用白色对象，GC 就可以增量式的运行，减少停顿时间。

什么是三色标记?

三色标记，通过字面意思我们就可以知道它由 3 种颜色组成：

1. 黑色 Black：表示对象是可达的，即使用中的对象，黑色是已经被扫描的对象。
2. 灰色 Gary：表示被黑色对象直接引用的对象，但还没对它进行扫描。
3. 白色 White：白色是对象的初始颜色，如果扫描完成后，对象依然还是白色的，说明此对象是垃圾对象。

三色标记规则

黑色不能指向白色对象。即黑色可以指向灰色，灰色可以指向白色。

三色标记法，主要流程如下：

三色标记算法是对标记阶段的改进，原理如下：

- **起初所有对象都是白色。**
- **从根出发扫描所有可达对象，标记为灰色，放入待处理队列。**
- **从队列取出灰色对象，将其引用对象标记为灰色放入队列，自身标记为黑色。**
- **重复 3，直到灰色对象队列为空。此时白色对象即为垃圾，进行回收。**

三色法标记主要是第一部分是扫描所有对象进行三色标记，标记为黑色、灰色和白色，标记完成后只有黑色和白色对象，黑色代表使用中对象，白色对象代表垃圾，灰色是白色过渡到黑色的中间临时状态，第二部分是清扫垃圾，即清理白色对象。

第一部分包含了栈扫描、标记和标记结束 3 个阶段。在栈扫描之前有 2 个重要的准备：STW（Stop The World）和开启写屏障（WB，Write Barrier）。

STW 是 Stop The World，指会暂停所有正在执行的用户线程 / 协程，进行垃圾回收的操作，在这之前会进行一些准备工作，比如开启 Write Barrier，把全局变量，以及每个 goroutine 中的 Root 对象 收集起来，Root 对象是标记扫描的源头，可以从 Root 对象依次索引到使用中的对象，STW 为垃圾对象的扫描和标记提供了必要的条件。

每个 P 都有一个 mcache，每个 mcache 都有 1 个 Span 用来存放 TinyObject，TinyObject 都是不包含指针的对象，所以这些对象可以直接标记为黑色，然后关闭 STW。

每个 P 都有 1 个进行扫描标记的 goroutine，可以进行并发标记，关闭 STW 后，这些 goroutine 就变成可运行状态，接收 Go Scheduler 的调度，被调度时执行 1 轮标记，它负责第 1 部分任务：栈扫描、标记和标记结束。

栈扫描阶段就是把前面搜集的 Root 对象找出来，标记为黑色，然后把它们引用的对象也找出来，标记为灰色，并且加入到 gcWork 队列，gcWork 队列保存了灰色的对象，每个灰色的对象都是一个 Work。

后面可以进入标记阶段，它是一个循环，不断的从 gcWork 队列中取出 work，所指向的对象标记为黑色，该对象指向的对象标记为灰色，然后加入队列，直到队列为空。然后进入标记结束阶段，再次开启 STW，不同的版本处理方式是不同的。

在 Go1.7 的版本是 Dijkstra 写屏障，这个写屏障只监控堆上指针数据的变动，由于成本原因，没有监控栈上指针的变动，由于应用 goroutine 和 GC 的标记 goroutine 都在运行，当栈上的指针指向的对象变更为白色对象时，这个白色对象应当标记为黑色，需要再次扫描全局变量和栈，以免释放这类不该释放的对象。

在 Go1.8 及以后的版本引入了混合写屏障，这个写屏障依然不监控栈上指针的变动，但是它的策略，使得无需再次扫描栈和全局变量，但依然需要 STW 然后进行一些检查。

标记结束阶段的最后会关闭写屏障，然后关闭 STW，唤醒熟睡已久的负责清扫垃圾的 goroutine。

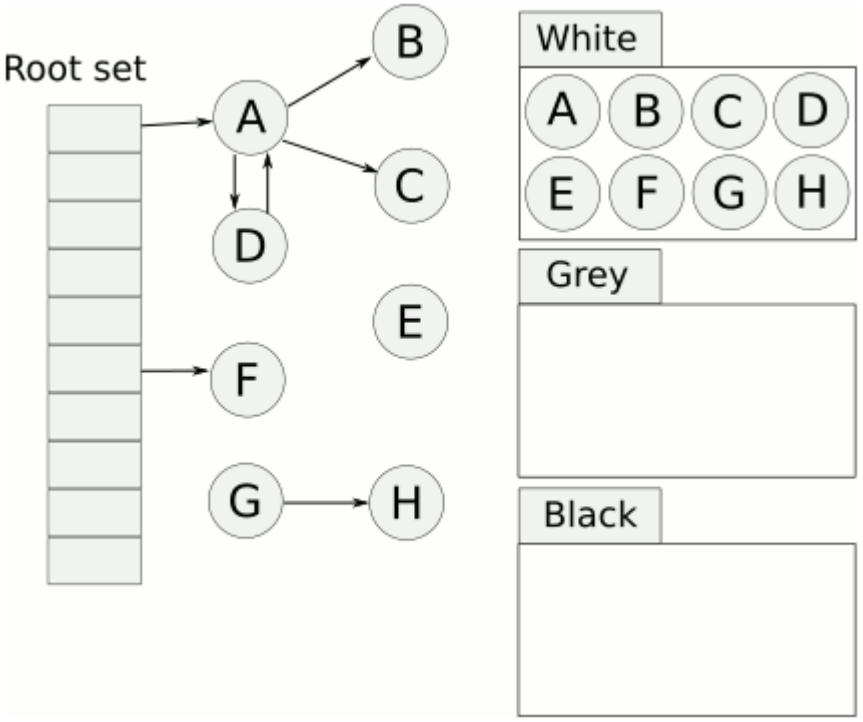
清扫 goroutine 是应用启动后立即创建的一个后台 goroutine，它会立刻进入睡眠，等待被唤醒，然后执行垃圾清理：把白色对象挨个清理掉，清扫 goroutine 和应用 goroutine 是并发进行的。清扫完成之后，它再次进入睡眠状态，等待下次被唤醒。

最后执行一些数据统计和状态修改的工作，并且设置好触发下一轮 GC 的阈值，把 GC 状态设置为 Off。

这些基本是 Go 垃圾回收的流程，但是在 go1.12 的源码稍微有一些不同，例如在标记结束后，就开始设置各种状态数据以及把 GC 状态成了 Off，在开启一轮 GC 时，会自动检测当前是否处于 Off，如果不是 Off，则当前 goroutine 会调用清扫函数，帮助清扫 goroutine 一起清扫 span，实际的 Go 垃圾回收流程以源码为准。

这里需要提下 go 的对象大小定义：

- 大对象是大于 32KB 的。
- 小对象 16KB 到 32KB 的。
- Tiny 对象指大小在 1Byte 到 16Byte 之间并且不包含指针的对象。



三色标记的一个明显好处是能够让用户程序和 mark 并发的进行。

混合写屏障规则是（GoV1.8）

在 Go 语言的垃圾回收（GC）机制中，对象的标记和写屏障的应用过程可以优化描述为以下步骤：

1. **GC 开始与根集合标记**：
    - GC 启动时，会先进行一次短暂的停顿（STW），以初始化 GC 的内部状态。
    - 在这次停顿中，所有活跃的对象（如全局变量、活跃栈帧中的指针等）被识别为根对象，并标记为灰色，表示它们需要被进一步扫描以确定其可达性。
2. **并发标记阶段**：
    - GC 与用户程序并发运行，从根集合开始，递归地扫描并标记所有可达的对象。
    - **插入写屏障**：当对象 A 新增一个指向对象 B 的指针时，如果对象 B 是白色（即未被标记），则将其标记为灰色。这确保了新增的引用不会导致对象 B 被错误地回收。
    - **删除写屏障**（在某些 Go 版本或实现中可能涉及，但具体行为可能有所不同）：当对象 A 删除一个指向对象 B 的指针时，如果对象 B 是灰色或白色，则将其重新标记为灰色（如果是白色，则直接标记为灰色；如果是灰色，则保持灰色状态）。这样做可以确保在后续扫描中，对象 B 仍然会被访问到，从而防止其被错误地回收。
3. **栈上对象的处理**：

    - 在 GC 期间，栈上创建的新对象最初是未标记的（即白色的），但由于它们是活跃对象，因此它们会很快被 GC 识别并处理。具体来说，当 GC 的标记器遍历到包含这些新对象的栈帧时，它们会被标记为灰色，并在后续的扫描过程中变为黑色。
4. **标记完成与清理**：
    - 当并发标记阶段完成足够的工作量或达到预定条件后，GC 会再次执行 STW，以完成剩余的标记工作，并准备进入清理阶段。
    - 清理阶段与用户程序并发进行，回收所有未被标记为可达（即黑色和灰色之外的对象）的内存。
5. **对象删除与可达性**：
    - 需要注意的是，对象被删除（即其引用被移除）并不直接导致其被标记为灰色或任何其他特定颜色。相反，对象的可达性是通过 GC 的扫描过程来确定的。如果一个对象在扫描过程中没有被任何可达对象引用，则它最终会被识别为不可达，并在清理阶段被回收。

综上所述，Go 语言的 GC 机制通过并发标记、写屏障和清理阶段来优化内存管理，减少 STW 时间，并提高程序的性能和响应速度。关于对象的标记和写屏障的具体行为，需要根据 Go 的当前版本和具体实现来准确理解。

Go V1.8 引入的混合写屏障（Hybrid Write Barrier）是一种优化垃圾回收（GC）性能的机制，它结合了插入写屏障（Insert Write Barrier）和删除写屏障（Delete Write Barrier）的优点，以减少垃圾回收过程中的停顿时间（STW，Stop The World）。混合写屏障规则可以概括如下：

插入写屏障规则

插入写屏障在对象 A 新增一个指向对象 B 的指针时触发。具体规则如下：

- **标记阶段**：当对象 A 新增一个指向对象 B 的指针时，如果对象 B 是白色（未被标记），则将其标记为灰色（表示其需要被进一步扫描）。这样做可以确保在标记过程中不会遗漏任何可达对象。
- **目的**：防止在并发标记过程中，由于新增的指针导致原本应该被回收的对象（白色对象）被错误地保留下来。

删除写屏障规则

删除写屏障在对象 A 删除一个指向对象 B 的指针时触发。具体规则如下：

- **标记阶段**：当对象 A 删除一个指向对象 B 的指针时，如果对象 B 是灰色或白色，则将其重新标记为灰色（如果是白色，则直接标记为灰色；如果是灰色，则保持灰色状态）。这样做可以确保在后续扫描中，对象 B 仍然会被访问到，从而防止其被错误地回收。
- **清除阶段**：在清除阶段开始时，所有在堆上的灰色对象都会被视为可达对象，因此不会被回收。删除写屏障确保了在并发修改指针的情况下，对象的可达性状态能够正确地被维护。

混合写屏障的优势

- **减少 STW 时间**：通过并发标记和写屏障机制，Go V1.8 能够显著减少垃圾回收过程中的 STW 时间，从而提高程序的并发性能和响应速度。
- **提高内存使用效率**：写屏障机制有助于更准确地识别垃圾对象，减少内存碎片的产生，提高内存的使用效率。
- **增强并发安全性**：在并发环境下，写屏障机制能够确保垃圾回收过程的安全性和正确性，防止由于并发修改导致的内存错误。

总之，Go V1.8 的混合写屏障规则通过结合插入写屏障和删除写屏障的优点，有效地优化了垃圾回收的性能和安全性，为 Go 语言的高并发特性提供了坚实的支撑。

GC 的触发时机?

初级必问，分为系统触发和主动触发。

1. 堆满。gcTriggerHeap：当所分配的堆大小达到阈值（由控制器计算的触发堆的大小）时，将会触发。
2. 超时。gcTriggerTime：当距离上一个 GC 周期的时间超过一定时间时，将会触发。时间周期以 runtime.forcegcperiod 变量为准，默认 2 分钟。
3. 启动。gcTriggerCycle：如果没有开启 GC，则启动 GC。
4. 手动。手动触发的 runtime.GC 方法。

十一、内存相关

1. 内存分配原理
2. 垃圾回收原理
3. 逃逸分析
4. Go 语言的内存模型及堆的分配管理

谈谈内存泄露，什么情况下内存会泄露？怎么定位排查内存泄漏问题?

泄露：不退出、不释放资源、不可控。

go 中的内存泄漏一般都是 goroutine 泄漏，就是 goroutine 没有被关闭，或者没有添加超时控制，让 goroutine 一只处于阻塞状态，不能被 GC。

内存泄露有下面一些情况

1. 如果 goroutine 在执行时**被阻塞**而无法退出，就会导致 goroutine 的内存泄漏，一个 goroutine 的最低栈大小为 2KB，在高并发的场景下，对内存的消耗也是非常恐怖的。
2. **互斥锁**未释放或者造成死锁会造成内存泄漏

3. **time.Ticker** 是每隔指定的时间就会向通道内写数据。作为循环触发器，必须调用 stop 方法才会停止，从而被 GC 掉，否则会一直占用内存空间。
4. 字符串的截取引发临时性的内存泄漏

```
func main () {
    var str0 = "12345678901234567890"
    str1 := str0 [:10]
}
```

切片虽然只使用了前面的 10 个字符，但它拥有数组的所有权（本质是 cap 容量等于数组长度），后面的字符虽然没有使用，但仍然无法 GC。

5. 切片截取引起父切片内存泄漏

```
func main () {
    var s0 = [] int {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
    s1 := s0 [:3]
}
```

同理，切片会导致父切片无法释放无用内存。

6. 函数数组传参引发内存泄漏【如果我们在函数传参的时候用到了数组传参，且这个数组够大（我们假设数组大小为 100 万，64 位机上消耗的内存约为 800w 字节，即 8MB 内存），或者该函数短时间内被调用 N 次，那么可想而知，会消耗大量内存，对性能产生极大的影响，如果短时间内分配大量内存，而又来不及 GC，那么就会产生临时性的内存泄漏，对于高并发场景相当可怕。】

内存泄漏排查方式

`pprof` 是 Go 官方内置的性能分析工具，支持分析 CPU、内存、goroutine、阻塞等运行时信息。它既适合线上调试，也非常适合在开发阶段检测性能瓶颈或排查协程泄露。

- runtime/pprof：采集程序（非 Server）的运行数据进行分析
- net/http/pprof：采集 HTTP Server 的运行时数据进行分析

输出方式：

- Report generation：报告生成
- Interactive terminal use：交互式终端使用
- Web interface：Web 界面

使用方式

1. 引入 `pprof` 包

```
import "runtime/pprof"
```

or

```
import _ "net/http/pprof"
import "net/http"
```

2. 启动一个调试服务。加在 `main()` 中，就能监听端口，暴露调试接口。

```
go func() {
    http.ListenAndServe("localhost:6060", nil)
}()
```

3. 打开功能

```
// 开启CPU性能分析：
pprof.StartCPUProfile(w io.Writer)
// 停止CPU性能分析：
pprof.StopCPUProfile()
```

4. 访问方式（通过浏览器或 `curl` 或写日志）

类型	地址	说明
goroutine	`http://localhost:6060/debug/pprof/goroutine?debug=2`	当前所有 goroutine 堆栈
heap	`http://localhost:6060/debug/pprof/heap`	内存使用情况
profile	`http://localhost:6060/debug/pprof/profile?seconds=30`	30 秒 CPU 性能分析
block	`http://localhost:6060/debug/pprof/block`	阻塞操作
threadcreate	`http://localhost:6060/debug/pprof/threadcreate`	线程创建信息

或者使用 `Curl`：

```
curl http://localhost:6060/debug/pprof/goroutine?debug=2
```

或者记录程序的堆栈信息

```
pprof.WriteHeapProfile(w io.Writer)
```

得到采样数据之后，使用go tool pprof工具进行内存性能分析。

5. 对这些数据进行分析，我们可以使用go tool pprof命令行工具。

go tool pprof最简单的使用方式为:

```
    go tool pprof [binary] [source]
```

其中：

- binary 是应用的二进制文件，用来解析各种符号；

- source 表示 profile 数据的来源，可以是本地的文件，也可以是 http 地址。

注意事项： 获取的 Profiling 数据是动态的，要想获得有效的数据，请保证应用处于较大的负载（比如正在生成中运行的服务，或者通过其他工具模拟访问压力）。否则如果应用处于空闲状态，得到的结果可能没有任何意义。

6. 典型用途场景

goroutine 泄露排查

访问：

```
http://localhost:6060/debug/pprof/goroutine?debug=2
```

- 看有没有数量激增、重复函数名、卡在 `<-chan`、`select{}`、`runtime.gopark` 等。

CPU 分析

```
go tool pprof http://localhost:6060/debug/pprof/profile?seconds=10
```

- 分析运行热点，优化算法或循环。

内存泄露排查

```
go tool pprof http://localhost:6060/debug/pprof/heap
```

- `inuse_objects`, `inuse_space` 等关键词帮助你找到没有释放的内存对象。

7. 示例代码

```go
package main

import (
    _ "net/http/pprof"
    "net/http"
    "fmt"
    "time"
)

func main() {
    go func() {
        http.ListenAndServe("localhost:6060", nil)
    }()

    for {
        go func() {
            time.Sleep(10 * time.Second)
```

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

    fmt.Println("Hello")
}
```

内存分析

1. Goroutine 分析
    - 访问：`/debug/pprof/goroutine?debug=2`
    - 查看当前所有协程状态（running、sleep、IO wait、chan 阻塞等）
    - 用于排查协程泄露或死锁
2. CPU 分析（profile）
    - 抓取：`/debug/pprof/profile?seconds=N`
    - 使用 `go tool pprof` 查看：
        - `flat`：函数自身 CPU 时间
        - `cum`：累计时间（含被调用者）
    - 用于查找 CPU 占用高的函数
3. 内存分析（heap 堆/ allocs 分配）
    - 抓取：`/debug/pprof/heap`，`/debug/pprof/allocs`
    - 查看对象分配与使用情况：
        - `inuse_objects` / `inuse_space`：当前在用内存
        - `total_alloc_objects` / `total_alloc_space`：累计分配
    - 用于排查内存泄露、高频分配等问题
4. 阻塞分析（block）
    - 抓取：`/debug/pprof/block`（需 `runtime.SetBlockProfileRate(1)`）
    - 查看锁、channel 等资源的阻塞位置与时间
5. 互斥锁分析（mutex）
    - 抓取：`/debug/pprof/mutex`（需 `runtime.SetMutexProfileFraction(1)`）
    - 查看锁竞争次数与等待时间，定位锁热点
6. 线程创建（threadcreate）
    - 抓取：`/debug/pprof/threadcreate`
    - 查看创建系统线程的调用栈，用于判断线程滥用
7. 跟踪分析（trace）

- 抓取：`/debug/pprof/trace?seconds=N`
- 使用 `go tool trace` 分析调度、GC、系统调用等全流程

8. 其他指标
    - **cmdline**：程序启动命令参数
    - **symbol**：地址与函数名映射，用于解析其它 profile 中的指针信息

各指标汇总表

类型	用途	默认开启	启用方法
goroutine	协程泄露/阻塞排查	✅	/debug/pprof/goroutine
heap / allocs	内存使用、泄露分析	✅	/debug/pprof/heap /allocs
threadcreate	系统线程数量与创建路径	✅	/debug/pprof/threadcreate
profile (CPU)	函数 CPU 占用分析	❌	加参数 seconds
block	阻塞位置（锁/chan）分析	❌	SetBlockProfileRate(1)
mutex	锁竞争与等待分析	❌	SetMutexProfileFraction(1)
trace	全面执行事件分析	❌	/debug/pprof/trace?seconds=N

golang 的内存逃逸吗？什么情况下会发生内存逃逸？（必问）

1. 本该分配到栈上的变量，跑到了堆上，这就是内存逃逸。
2. 栈是高地址到低地址，栈上的变量，函数结束后变量会跟着回收掉，不会有额外性能的开销。
3. 变量从栈逃逸到堆上，如果要回收掉，需要进行 GC，那么 GC 一定会带来额外的性能开销。编程语言不断优化 GC 算法，主要目的都是为了减少 GC 带来的额外性能开销，变量一旦逃逸会导致性能开销变大。

栈

- 分配速度快（非常快）
- 生命周期短 → 函数返回就没了
- **不需要 GC 管理**

堆

- 分配慢

- 生命周期长
- 需要 GC 回收

Go 编译器：

- 尽量把变量分配在栈上
- 只有**必须活得比函数长**，才分配到堆

内存逃逸的情况如下：

1. 方法内返回局部变量指针。
2. 向 channel 发送指针数据。
3. 在闭包中引用包外的值。
4. 在 slice 或 map 中存储指针。
5. 切片（扩容后）长度太大。
6. 在 interface 类型上调用方法。

逃逸并不是 bug！

只是：

- 堆分配比栈慢
- 需要 GC
- 频繁逃逸可能影响性能

所以 Go 性能调优时，最常看的就是**逃逸分析**。

请简述 Go 是如何分配内存的？

在 Go 中，内存分配采用了分层结构，主要包括四个组件：`mcache`、`mcentral`、`mheap` 和 `mspan`。

最上层是 `mcache`，它是每个线程私有的内存缓存，用于快速分配小对象，避免频繁加锁；当 `mcache` 中没有可用内存块时，会向中间层的 `mcentral` 请求。`mcentral` 维护了多个 `mspan`，每个 `mspan` 对应一种对象大小，用于集中管理和复用。

如果 `mcentral` 也无法满足请求，则会从全局的 `mheap` 分配内存，`mheap` 是整个 Go 堆管理的核心，负责向操作系统申请大块内存（称为 `arena`），并划分为多个 `mspan`。`mspan` 是管理若干页（page）的最小分配单元，负责承载实际的对象。

go 内存分配器

Channel 分配在栈上还是堆上？哪些对象分配在堆上，哪些对象分配在栈上?

Channel 被设计用来实现协程间通信的组件，其作用域和生命周期不可能仅限于某个函数内部，所以 golang 直接将其分配在堆上。

准确地说，你并不需要知道。Golang 中的变量只要被引用就一直会存活，存储在堆上还是栈上由内部实现决定，和具体的语法没有关系。

知道变量的存储位置确实和效率编程有关系。如果可能，Golang 编译器会将函数的局部变量分配到函数栈帧（stack frame）上。然而，如果编译器不能确保变量在函数 return 之后不再被引用，编译器就会将变量分配到堆上。而且，如果一个局部变量非常大，那么它也应该被分配到堆上而不是栈上。

当前情况下，如果一个变量被取地址，那么它就有可能被分配到堆上，然而，还要对这些变量做逃逸分析，如果函数 return 之后，变量不再被引用，则将其分配到栈上。

介绍一下大对象小对象，为什么小对象多了会造成 GC 压力?

小于等于 32k 的对象就是小对象，其它都是大对象。一般小对象通过 mspan 分配内存；大对象则直接由 mheap 分配内存。通常小对象过多会导致 GC 三色法消耗过多的 CPU。优化思路是，减少对象分配。

- 小对象：如果申请小对象时，发现当前内存空间不存在空闲块时，将会需要调用 nextFree 方法获取新的可用的对象，可能会触发 GC 行为。
- 大对象：如果申请大于 32k 以上的大对象时，可能会触发 GC 行为。

十二、编译

逃逸分析是怎么进行的

在编译原理中，分析指针动态范围的方法称之为逃逸分析。通俗来讲，当一个对象的指针被多个方法或线程引用时，我们称这个指针发生了逃逸。

Go 语言的逃逸分析是编译器执行静态代码分析后，对内存管理进行的优化和简化，它可以决定一个变量是分配到堆还栈上。

写过 C/C++ 的同学都知道，调用著名的 malloc 和 new 函数可以在堆上分配一块内存，这块内存的使用和销毁的责任都在程序员。一不小心，就会发生内存泄露。

Go 语言里，基本不用担心内存泄露了。虽然也有 new 函数，但是使用 new 函数得到的内存不一定就在堆上。堆和栈的区别对程序员“模糊化”了，当然这一切都是 Go 编译器在背后帮我们完成的。

Go 语言逃逸分析最基本的原则是：如果一个函数返回对一个变量的引用，那么它就会发生逃逸。

简单来说，编译器会分析代码的特征和代码生命周期，Go 中的变量只有在编译器可以证明在函数返回后不会再被引用的，才分配到栈上，其他情况下都是分配到堆上。

Go 语言里没有一个关键字或者函数可以直接让变量被编译器分配到堆上，相反，编译器通过分析代码来决定将变量分配到何处。

对一个变量取地址，可能会被分配到堆上。但是编译器进行逃逸分析后，如果考察到在函数返回后，此变量不会被引用，那么还是会被分配到栈上。

编译器会根据变量是否被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；

Go 的垃圾回收，让堆和栈对程序员保持透明。真正解放了程序员的双手，让他们可以专注于业务，“高效”地完成代码编写。把那些内存管理的复杂机制交给编译器，而程序员可以去享受生活。

如果变量都分配到堆上，堆不像栈可以自动清理。它会引起 Go 频繁地进行垃圾回收，而垃圾回收会占用比较大的系统开销（占用 CPU 容量的 25%）。

堆和栈相比，堆适合不可预知大小的内存分配。但是为此付出的代价是分配速度较慢，而且会形成内存碎片。栈内存分配则会非常快。栈分配内存只需要两个 CPU 指令：“PUSH”和“RELEASE”，分配和释放；而堆分配内存首先需要去找到一块大小合适的内存块，之后要通过垃圾回收才能释放。

通过逃逸分析，可以尽量把那些不需要分配到堆上的变量直接分配到栈上，堆上的变量少了，会减轻分配堆内存的开销，同时也会减少 GC 的压力，提高程序的运行速度。

GoRoot 和 GoPath 有什么用

GoRoot 是 Go 的安装路径。mac 或 unix 是在 `/usr/local/go` 路径上，来看下这里都装了些什么：

```
[qcrao@qcrao go]$ ls
api      bin   CONTRIBUTING.md  doc          lib      misc     pkg        robots.txt  test
AUTHORS  blog  CONTRIBUTORS     favicon.ico  LICENSE  PATENTS  README.md  src         VERSION
```

bin 目录下面：

```
[qcrao@qcrao go]$ ls pkg/
include      linux_amd64_dynlink  linux_amd64_shared              tool
linux_amd64  linux_amd64_race     linux_amd64_testcshared_shared
```

pkg 目录下面：

```
[qcrao@qcrao go]$ ls pkg/
include      linux_amd64_dynlink  linux_amd64_shared              tool
linux_amd64  linux_amd64_race     linux_amd64_testcshared_shared
[qcrao@qcrao go]$ ls pkg/linux_amd64
archive    context.a  encoding.a  hash     index     math    os         regexp     strings.a  text
bufio.a    crypto     errors.a    hash.a   internal  math.a  os.a       regexp.a   sync       time.a
bytes.a    crypto.a   expvar.a    html     io        mime    path       runtime    sync.a     unicode
cmd        database   flag.a      html.a   io.a      mime.a  path.a     runtime.a  syscall.a  unicode.a
compress   debug      fmt.a       image    log       net     plugin.a   sort.a     testing    vendor
container  encoding   go          image.a  log.a     net.a   reflect.a  strconv.a  testing.a
```

Go 工具目录如下，其中比较重要的有编译器 `compile`，链接器 `link`：

```
/usr/local/go/pkg/tool/linux_amd64
[qcrao@qcrao linux_amd64]$ ls
addr2line  buildid  compile  dist  fix   nm       pack   test2json  trace
asm        cgo      cover    doc   link  objdump  pprof  tour       vet
```

GoPath 的作用在于提供一个可以寻找 `.go` 源码的路径，它是一个工作空间的概念，可以设置多个目录。Go 官方要求，GoPath 下面需要包含三个文件夹：

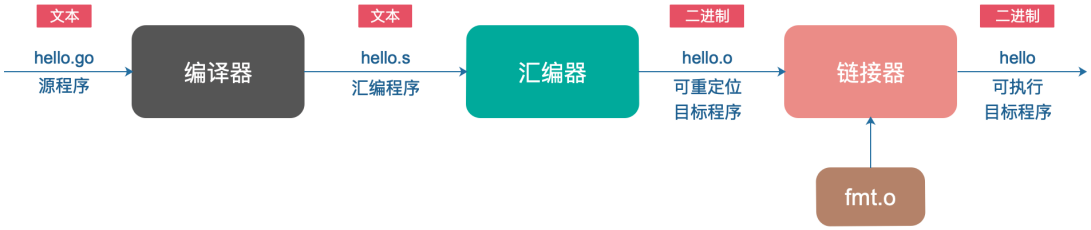
```
src
pkg
bin
```

src 存放源文件，pkg 存放源文件编译后的库文件，后缀为 `.a`；bin 则存放可执行文件。

Go 编译链接过程概述

Go 代码并不能直接运行，每条 Go 语句必须转化为一系列的低级机器语言指令，将这些指令打包到一起，并以二进制磁盘文件的形式存储起来，也就是可执行目标文件。

从源文件到可执行目标文件的转化过程：



完成以上各个阶段的就是 Go 编译系统。你肯定知道大名鼎鼎的 GCC（GNU Compile Collection），中文名为 GNU 编译器套装，它支持像 C，C++，Java，Python，Objective-C，Ada，Fortran，Pascal，能够为很多不同的机器生成机器码。

可执行目标文件可以直接在机器上执行。一般而言，先执行一些初始化的工作；找到 main 函数的入口，执行用户写的代码；执行完成后，main 函数退出；再执行一些收尾的工作，整个过程完毕。

在接下来的文章里，我们将探索 `编译` 和 `运行` 的过程。

Go 源码里的编译器源码位于 `src/cmd/compile` 路径下，链接器源码位于 `src/cmd/link` 路径下。

Go 编译相关的命令详解

和编译相关的命令主要是：

```
go build
go install
go run
```

go build

`go build` 用来编译指定 packages 里的源码文件以及它们的依赖包，编译的时候会到 `$GoPath/src/package` 路径下寻找源码文件。`go build` 还可以直接编译指定的源码文件，并且可以同时指定多个。

通过执行 `go help build` 命令得到 `go build` 的使用方法：

```
usage: go build [-o output] [-i] [build flags] [packages]
```

`-o` 只能在编译单个包的时候出现，它指定输出的可执行文件的名字。

`-i` 会安装编译目标所依赖的包，安装是指生成与代码包相对应的 `.a` 文件，即静态库文件（后面要参与链接），并且放置到当前工作区的 pkg 目录下，且库文件的目录层级和源码层级一致。

至于 build flags 参数，`build, clean, get, install, list, run, test` 这些命令会共用一套：

参数	作用
-a	强制重新编译所有涉及到的包，包括标准库中的代码包，这会重写 /usr/local/go 目录下的 `.a`
文件	
-n	打印命令执行过程，不真正执行
-p n	指定编译过程中命令执行的并行数，n 默认为 CPU 核数
-race	检测并报告程序中的数据竞争问题
-v	打印命令执行过程中所涉及到的代码包名称
-x	打印命令执行过程中所涉及到的命令，并执行
-work	打印编译过程中的临时文件夹。通常情况下，编译完成后会被删除

我们知道，Go 语言的源码文件分为三类：命令源码、库源码、测试源码。

> 命令源码文件：是 Go 程序的入口，包含 `func main ()` 函数，且第一行用 `package main` 声明属于 main 包。

> 库源码文件：主要是各种函数、接口等，例如工具类的函数。

> 测试源码文件：以 `_test.go` 为后缀的文件，用于测试程序的功能和性能。

注意，`go build` 会忽略 `*_test.go` 文件。

go install

`go install` 用于编译并安装指定的代码包及它们的依赖包。相比 `go build`，它只是多了一个“安装编译后的结果文件到指定目录”的步骤。
还是使用之前 hello-world 项目的例子，我们先将 pkg 目录删掉，在项目根目录执行：

```
go install src/main.go

或者

go install util
```

两者都会在根目录下新建一个 `pkg` 目录，并且生成一个 `util.a` 文件。并且，在执行前者的时候，会在 GOBIN 目录下生成名为 main 的可执行文件。所以，运行 `go install` 命令，库源码包对应的 `.a` 文件会被放置到 `pkg` 目录下，命令源码包生成的可执行文件会被放到 GOBIN 目录。

`go install` 在 GoPath 有多个目录的时候，会产生一些问题，具体可以去看郝林老师的 `Go 命令教程`，这里不展开了。

go run

`go run` 用于编译并运行命令源码文件。

Go 程序启动过程是怎样的

十三、框架

Gin

文档：https://gin-gonic.com/zh-cn/docs/introduction/

0. 特性

1. **快速**
    1. 基于 Radix 树的路由，小内存占用。没有反射。可预测的 API 性能。
2. **支持中间件**
    1. 传入的 HTTP 请求可以由一系列中间件和最终操作来处理。例如：Logger，Authorization，GZIP，最终操作 DB。
3. **Crash 处理**
    1. Gin 可以 catch 一个发生在 HTTP 请求中的 panic 并 recover 它。这样，你的服务器将始终可用。例如，你可以向 Sentry 报告这个 panic!
4. **JSON 验证**
    1. Gin 可以解析并验证请求的 JSON，例如检查所需值的存在。
5. **路由组**
    1. 更好地组织路由。是否需要授权，不同的 API 版本..... 此外，这些组可以无限制地嵌套而不会降低性能。
6. **错误管理**

1. Gin 提供了一种方便的方法来收集 HTTP 请求期间发生的所有错误。最终，中间件可以将它们写入日志文件，数据库并通过网络发送。

7. **内置渲染**
    1. Gin 为 JSON，XML 和 HTML 渲染提供了易于使用的 API。
8. **可扩展性**
    1. 新建一个中间件非常简单，去查看示例代码吧。

1. gin 目录结构

文档：https://blog.csdn.net/qq_34877350/article/details/107959381

```
├── gin
|   ├──  Router
|          └── router.go
|   ├──  Middlewares
|          └── corsMiddleware.go
|   ├──  Controllers
|          └── testController.go
|   ├──  Services
|          └── testService.go
|   ├──  Models
|          └── testModel.go
|   ├──  Databases
|          └── mysql.go
|   ├──  Sessions
|          └── session.go
└── main.go
```

- 使用 gorm 访问数据库
- gin 为项目根目录
- main.go 为入口文件
- Router 为路由目录
- Middlewares 为中间件目录
- Controllers 为控制器目录（MVC）
- Services 为服务层目录，这里把 DAO 逻辑也写入其中，如果分开也可以
- Models 为模型目录
- Databases 为数据库初始化目录
- Sessions 为 session 初始化目录
- 文件 引用顺序 大致如下：
- main.go(在 main 中关闭数据库) - router(Middlewares) - Controllers - Services(sessions) - Models - Databases

2. Gin 框架介绍及使用 - 李文周的博客

文档：https://www.liwenzhou.com/posts/Go/Gin_framework/#autoid-0-0-0

3. 源码

Gin 源码阅读与分析：
https://www.yuque.com/iveryimportantpig/huchao/zd24cb3z2bco5304

go-zero

文档：https://go-zero.dev/cn/docs/introduction

> go-zero 是一个集成了各种工程实践的 web 和 rpc 框架。通过弹性设计保障了大并发服务端的稳定性，经受了充分的实战检验。
> go-zero 包含极简的 API 定义和生成工具 goctl，可以根据定义的 api 文件一键生成 Go, iOS, Android, Kotlin, Dart, TypeScript, JavaScript 代码，并可直接运行。

使用 go-zero 的好处：

- 轻松获得支撑千万日活服务的稳定性
- 内建级联超时控制、限流、自适应熔断、自适应降载等微服务治理能力，无需配置和额外代码
- 微服务治理中间件可无缝集成到其它现有框架使用
- 极简的 API 描述，一键生成各端代码
- 自动校验客户端请求参数合法性
- 大量微服务治理和并发工具包

字节-CloudWeGo

文档：https://www.cloudwego.io/zh/docs/

HTTP-Hertz

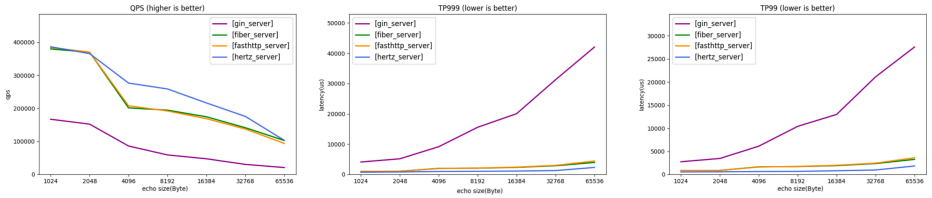
文档：https://www.cloudwego.io/zh/docs/hertz/overview/

> 是一个 Golang 微服务 HTTP 框架，在设计之初参考了其他开源框架 fasthttp、gin、echo 的优势，并结合字节跳动内部的需求，使其具有高易用性、高性能、高扩展性等特点，目前在字节跳动内部已广泛使用。如今越来越多的微服务选择使用 Golang，如果对微服务性能有要求，又希望框架能够充分满足内部的可定制化需求，Hertz 会是一个不错的选择。

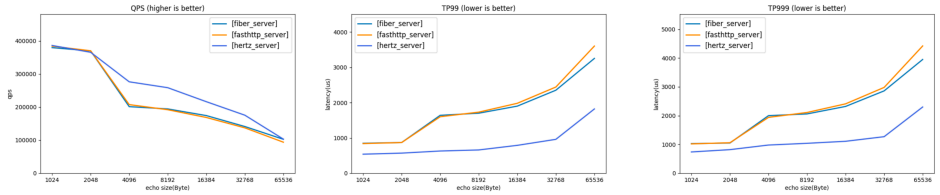
特点

- 高易用性在开发过程中，快速写出来正确的代码往往是更重要的。因此，在 Hertz 在迭代过程中，积极听取用户意见，持续打磨框架，希望为用户提供一个更好的使用体验，帮助用户更快的写出正确的代码。

- 高性能 Hertz 默认使用自研的高性能网络库 Netpoll，在一些特殊场景相较于 go net，Hertz 在 QPS、时延上均具有一定优势。关于性能数据，可参考下图 Echo 数据。四个框架的对比：



三个框架的对比：



关于详细的性能数据，可参考 https://github.com/cloudwego/hertz-benchmark。

- 高扩展性 Hertz 采用了分层设计，提供了较多的接口以及默认的扩展实现，用户也可以自行扩展。同时得益于框架的分层设计，框架的扩展性也会大很多。目前仅将稳定的能力开源给社区，更多的规划参考 RoadMap。
- 多协议支持 Hertz 框架原生提供 HTTP1.1. ALPN 协议支持。除此之外，由于分层设计，Hertz 甚至支持自定义构建协议解析逻辑，以满足协议层扩展的任意需求。
- 网络层切换能力 Hertz 实现了 Netpoll 和 Golang 原生网络库 间按需切换能力，用户可以针对不同的场景选择合适的网络库，同时也支持以插件的方式为 Hertz 扩展网络库实现。

RPC-Kitex

文档：https://www.cloudwego.io/zh/docs/kitex/overview/

> 字节跳动内部的 Golang 微服务 RPC 框架，具有**高性能**、**强可扩展**的特点，在字节内部已广泛使用。如果对微服务性能有要求，又希望定制扩展融入自己的治理体系，Kitex 会是一个不错的选择。

框架特点

- **高性能**使用自研的高性能网络库 Netpoll，性能相较 go net 具有显著优势。
- **扩展性**提供了较多的扩展接口以及默认扩展实现，使用者也可以根据需要自行定制扩展，具体见下面的框架扩展。
- **多消息协议**RPC 消息协议默认支持**Thrift**、**Kitex Protobuf**、**gRPC**。Thrift 支持 Buffered 和 Framed 二进制协议；Kitex Protobuf 是 Kitex 自定义的 Protobuf 消息协议，协议格式类似 Thrift；gRPC 是对 gRPC 消息协议的支持，可以与 gRPC 互通。除此之外，使用者也可以扩展自己的消息协议。

- **多传输协议**传输协议封装消息协议进行 RPC 互通，传输协议可以额外透传元信息，用于服务治理，Kitex 支持的传输协议有**TTHeader**、**HTTP2**。TTHeader 可以和 Thrift、Kitex Protobuf 结合使用；HTTP2 目前主要是结合 gRPC 协议使用，后续也会支持 Thrift。
- **多种消息类型**支持**PingPong**、**Oneway**、**双向 Streaming**。其中 Oneway 目前只对 Thrift 协议支持，双向 Streaming 只对 gRPC 支持，后续会考虑支持 Thrift 的双向 Streaming。
- **服务治理**支持服务注册/发现、负载均衡、熔断、限流、重试、监控、链路跟踪、日志、诊断等服务治理模块，大部分均已提供默认扩展，使用者可选择集成。
- **代码生成**Kitex 内置代码生成工具，可支持生成**Thrift**、**Protobuf**以及脚手架代码。

参考并致谢

1. 可可酱 可可酱：Golang 常见面试题
2. Bel_Ami 同学 golang 面试题 (从基础到高级)