**9-Go**

1. **chan实现原理**

* chan实现了一个环形队列，队列长度由通道大小决定
* 被阻塞的线程会被挂在channel的等待队列，等待被唤醒。等待队列中有两个队列，一个是接收队列，一个是发送队列，一般情况下两个至少一个为空，除非一个线程下一边读一边写。
* 一个channel只允许同时间被一个goroutine读写
* 写入流程：首先如果等待队列中的接收队列中有G等待，说明没有缓冲区，或者缓冲区为空，此时会将数据写入其中一个G，再唤醒G，结束流程。如果缓冲区有空闲位置，直接写入后结束流程。如果没有空余位置，将发送数据写入G，然后将G加入发送队列，进入睡眠，等待被唤醒。
* 读取流程：如果发送队列有G等待，说明没有缓冲区，或者缓冲区为空，此时读取一个G中的数据然后将G唤醒。如果缓冲有数据，直接读取。如果没有数据，将G挂入接收队列，然后睡眠，等待被唤醒。
* 关闭channel：会把recvq中的G全部唤醒，数据全部置为nil。将sendq的G都唤醒，但是这些G会报出异常。出现异常的场景包括：关闭返回值为nil的channel，关闭已经关闭的channel，向已经关闭的channel写数据。
* 用法：（1）读写分离（2）通过select控制多个channel，实现控制线程 （2）range持续读出数据。

2. **slice实现原理**

* slice底层使用了数组来实现，底层容量不足的时候自动重新分配生成新的slice。
* make可以指定slice的长度和容量
* 用数组创建slice的时候，slice将与原数组公用一部分内存。数组切片后面的都为切片预留长度。如果对slice进行扩容的时候，slice空间不够，他会先进行操作，等到空间不够的时候新建一个空间，然后把原有的数据复制过来。扩容大小规定，如果原来容量小于1024，则扩容2倍，如果大于1024则扩容1.25倍。
* 使用copy拷贝两个切片，但是长度不等，长的往小的copy，只会复制小的最长长度的数据过去。
* 要点：（1）千万不要多个slice操作一个array，防止读写冲突（2）切片拷贝要确定拷贝个数（3）一开始确定好cap大小来预留空间（4）每个切片底层都是指向了一个数组，函数传递不会拷贝切片，切片只是个结构体

3. **map实现原理**

map底层使用了哈希表来实现，一个哈希表中有多个哈希表节点，称为bucket。bucket中第一个是哈希值，第二个是八个键值对，第三个是哈希冲突时候链接地址。哈希表通过负载因子决定是否扩充哈希表，还有一个等量扩容，是将访问量高的哈希地址冲突又比较多，将这部分重新整理，在bucket不变的情况下重新编排位置。

map线程不安全，可以使用 sync.map，它是线程安全的

sync.Map的设计目标是为了提供一个高效的并发安全的map，特别是在读多写少的场景下。它的设计考虑了以下目标：

1. 最小化锁的争用：通过使用只读和读写分离的数据结构，减少了锁的争用。
2. 延迟删除：通过标记删除而不是立即删除来提高性能。
3. 动态调整：根据实际的使用模式动态调整内部数据结构，以优化性能。

尽可能无锁化： 要实现并发安全，很难做到[无锁化](https://zhida.zhihu.com/search?q=%E6%97%A0%E9%94%81%E5%8C%96&zhida_source=entity&is_preview=1)。但是为了提升性能，应该尽可能使用[原子操作](https://zhida.zhihu.com/search?q=%E5%8E%9F%E5%AD%90%E6%93%8D%E4%BD%9C&zhida_source=entity&is_preview=1)，最大化减少锁的使用。

4. **struct实现原理**

struct中包含了tag字段，用来映射其它文件字段的映射，比如json等

5. **iota实现原理**

const中第一个数据必须指定表达式，如果后面都没有，则后面const继承第一个数据的表达式，iota在const出现位置后，按照语句行数递增

6. **string实现原理**

byte切片直接通过string()可以转换为字符串。string通过[]byte()转换为切片,string不允许修改

7. **defer**

* defer函数的参数遵从函数参数传递的值传递和引用传递
* 将函数中的变量直接在defer中使用，会导致变量的改变，包括返回值，因为return不是原子操作
* 延迟函数按照栈的原则，先进后出
* defer会有陷阱，recover没有直接在defer函数中调用，会导致recover不会被调用，同时panic的参数为nil或者协程没有发生panic都不会导致recover被调用。

8. **select**

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

9. **range**

* 遍历过程中可以视情况放弃接收index或value，可以一定程度上提升性能
* 遍历channel时，如果channel中没有数据，可能会阻塞
* 尽量避免遍历过程中修改原数据，遍历前就已经确定遍历次数，不可以修改

10. **mutex**

* 加锁后立即使用defer对其解锁，可以有效的避免死锁。
* 加锁和解锁最好出现在同一个层次的代码块中，比如同一个函数。
* 重复解锁会引起panic，应避免这种操作的可能性。
* Locked: 表示该Mutex是否已被锁定，0：没有锁定 1：已被锁定。
* Woken: 表示是否有协程已被唤醒，0：没有协程唤醒 1：已有协程唤醒，正在加锁过程中。
* Starving：表示该Mutex是否处于饥饿状态，0：没有饥饿 1：饥饿状态，说明有协程阻塞了超过1ms。
* Waiter: 表示阻塞等待锁的协程个数，协程解锁时根据此值来判断是否需要释放信号量。

11. **rwmutex**

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

12. **Goroutine**

在Go语言中，并不直接使用传统的线程模型，而是使用称为“goroutine”的概念。Goroutine是由Go运行时环境管理的轻量级线程。

1. 轻量级:

Goroutines比传统的操作系统线程更轻量，它们占用的内存更少，启动速度更快。

Go运行时可以在很少的操作系统线程上调度成千上万的goroutines。

1. 动态栈:

Goroutines拥有动态栈，这意味着它们的栈大小可以根据需要增长和缩小，这与固定大小的线程栈形成对比。

1. 调度:

Goroutines是由Go运行时的调度器（scheduler）调度的，而不是由操作系统直接调度。

这个调度器在用户态运行，使用了称为M:N调度（多个goroutines映射到较少的操作系统线程）的技术。

13. **GMP模型**

* **G**：G相当于操作系统改的进程控制块，它包括函数执行的指令和参数，任务对象，线程上下文切换，字段保护和字段的寄存器。
* **M**：M是一个线程，每个M都有一个线程的栈，如果没有给线程的栈分配内存，操作系统会给线程的栈分配默认内存。M的数量可以用SetMaxThreads()来决定。
* **P**：P是处理器的一个抽象概念，当P有任务，需要创建或者唤醒一个系统线程去处理它队列中的任务。P决定同时执行的任务数量，GOMAXPROCS可以限制数量。P有两种队列，全局队列用来平衡不同P的任务数量，所有M共享P的全局队列。本地队列是没有锁的，没有数据竞争问题，处理速度比较快。P和M的数量没有关系。
* **调度过程**：首先创建一个G对象，然后G被保存在P的本地队列或者全局队列。这时P会唤醒一个M。P按照它的执行顺序继续执行任务。M寻找一个空闲的P，如果找得到，将G移动到它自己。然后M执行一个调度循环：调用G对象->执行->清理线程->继续寻找Goroutine。M执行过程中上下文切换会随时发生，让发生切换，任务现场需要被保护，这样下一次调度执行可以进行线程恢复，M的栈保存在G对象。如果G对象没有被执行，M可以将G重新放到P的调度队列，等待下一次的调度执行。当调度执行，M可以通过G的寄存器现场恢复。
* **线程清理**：G调度为了实现P/M的绑定，线程清理就是释放P上的G，使得其它的G能够被调度。主动释放：当执行G任务时候发生了系统调用，M会阻塞，调度器会设置一个超时时间来释放P。被动释放：当系统调用发生，监控程序需要扫描处于阻塞状态的P/M。如果超时，P资源会回收，P被安排给队列中的其它G任务。
* **调度策略**：调度策略有三种：队列轮转，周期性查看队列中的G，让每个G执行一小段时间，防止被饿死。任务窃取，当前P没有任务的时候，从其它P的队列上获得G。另一种是任务转移，当M阻塞，将P转移到其它空闲的M。
* **go func(){}执行过程**：首先创建一个新的goroutine，让G保存在P的本地队列，如果满了就保存在全局队列。G在M上运行，每个M绑定一个P。如果P的本地队列没有G，M会从全局或者其它本地队列窃取G。当M阻塞，会将M从P中解除，把G运行在其它空闲的M或者新建M。当M恢复，会尝试获得一个空闲的P，如果没有空闲的P，M会休眠，G会放到全局队列。

14. **内存分配**

* Golang程序启动时申请一大块内存，并划分成spans、bitmap、arena区域
* arena区域按页划分成一个个小块
* span管理一个或多个页
* mcentral管理多个span供线程申请使用
* mcache作为线程私有资源，资源来源于mcentral
* 每个mcentral对象只管理特定的class规格的span。事实上每种class都会对应一个mcentral,这个mcentral的集合存放于mheap数据结构中

内存分配过程：

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

15. **垃圾回收**

**业界常见的垃圾回收算法有以下几种：**

* 引用计数：对每个对象维护一个引用计数，当引用该对象的对象被销毁时，引用计数减1，当引用计数器为0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。
* Mark-Sweep算法执行过程可以分成标记（Mark）和清除（Sweep）两个阶段：
* **Mark阶段：** 从根对象开始，对内存对象进行图遍历，对所有可达的对象进行标记。
* **Sweep阶段：** 对Mark阶段未被标记的内存对象进行回收，回收完毕后重置所有内存对象的标记以便下一轮‘标记-清除’
* 优点：解决了引用计数的缺点。
* 缺点：需要STW，即要暂时停掉程序运行。
* 代表语言：Golang（其采用三色标记法）
* 分代收集：按照对象生命周期长短划分不同的代空间，生命周期长的放入老年代，而短的放入新生代，不同代有不同的回收算法和回收频率。
* 优点：回收性能好
* 缺点：算法复杂
* 代表语言： JAVA

**三色标记法（标记-清除，GO）**：

三色标记算法将所有的内存对象抽象为黑、白、灰三类。

* **白色：** 初始色，如果标记阶段结束还是白色，则该内存对象将被回收。
* **灰色：** 活跃的对象，存在于标记阶段**中间状态**，自身及其下游需要被进一步扫描、标记。所有的根对象在标记开始时被全部置为灰色。
* **黑色：** 活跃的对象，已被标记完成，不会再**通过该对象**对其下游对象扫描标记。

**三色标记法的基本过程**

1. 垃圾回收开始前，所有的内存对象都是白色；
2. 垃圾回收开始时，所有根对象被标记会灰色；
3. 从灰色对象集合，选取一个灰色对象，将其下游白色对象置灰，将自己置为黑色；
4. 重复第3步骤，直至灰色对象集合为空。

**三色标记法本身仍然不能保证并发和增量执行的正确性。**在并发和增量执行的场景下，

活跃对象（白色）被错误回收的必要条件：

1. 不存在从灰色对象到达该白色对象的路径。（该白色对象在标记阶段不会再被扫描到）
2. 存在从黑色对象到达该白色对象的路径。（否则结合1的话，该白色对象本来就是一个需要被回收的垃圾对象）

但是三色标记法让并发和增量执行的实现变得容易。回到<标记-清除>中的异常例子。

因为活跃对象（白色）被错误回收的两个条件都是必要条件，要保证活跃对象都被发现，我们可以两个条件都破坏，也可以只破坏一个条件。由此，就衍生了两种三色不变性：

* **强三色不变性：**不存在黑色对象对白色对象的直接引用。（隐含了一层意思：如果该对象是活跃对象，那么必然存在从灰色对象到该对象的路径）。强三色不变性破坏了两个必要条件。
* **弱三色不变性：**被黑色对象直接引用的白色对象，必须要能够通过一些灰色对象可达。破坏了"不存在从灰色对象到达该白色对象的路径。"这个必要条件。

Go语言的三色不变性通过引入屏障技术来实现。

内存屏障技术是一种CPU屏障指令，用于让CPU和编译器对屏障指令前后的内存操作的执行顺序进行强制约束。现在处理器为了优化性能，经常会改变程序指令的执行顺序，而内存屏障技术则是用来对一些内存操作的执行顺序进行强制约束。**内存屏障指令前的内存操作一定会在内存屏障指令后的操作之前执行。**

内存屏障技术本身并不能用来保证三色不变性。但是我们可以利用内存屏障技术在内存操作之前执行特定的（用于保护三色不变性）的指令（执行特定代码）（就像Hook一样）。

**垃圾回收触发时机**

* 内存分配量达到阀值触发GC，阀值 = 上次GC内存分配量 \* 内存增长率，内存增长率由环境变量GOGC控制，默认为100
* 定期触发GC，默认情况下，最长2分钟触发一次GC
* 程序代码中也可以使用runtime.GC()来手动触发GC

**GC性能优化**

* GC性能与对象数量负相关，对象越多GC性能越差，对程序影响越大。
* 所以GC性能优化的思路之一就是减少对象分配个数，比如对象复用或使用大对象组合多个小对象等等。
* 另外，由于内存逃逸现象，有些隐式的内存分配也会产生，也有可能成为GC的负担。

16. **逃逸分析**

所谓逃逸分析（Escape analysis）是指由编译器决定内存分配的位置，不需要程序员指定。

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

逃逸场景

* 指针逃逸，局部变量进行new
* 栈空间不足逃逸，分配数组或者结构体太大
* 动态类型逃逸，interface类型数据
* 闭包引用对象逃逸

逃逸总结：

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

17. **并发控制**

* channel一般用于协程之间的通信，channel也可以用于并发控制。比如主协程启动N个子协程，主协程等待所有子协程退出后再继续后续流程，这种场景下channel也可轻易实现。
* WaitGroup通常用于等待一组“工作协程”结束的场景，其内部维护两个计数器，这里把它们称为“工作协程”计数器和“坐等协程”计数器，
* Add(delta int)方法用于增加“工作协程”计数，通常在启动新的“工作协程”之前调用；
* Done()方法用于减少“工作协程”计数，每次调用递减1，通常在“工作协程”内部且在临近返回之前调用；
* Wait()方法用于增加“坐等协程”计数，通常在所有”工作协程”全部启动之后调用；
* Done()方法除了负责递减“工作协程”计数以外，还会在“工作协程”计数变为0时检查“坐等协程”计数器并把“坐等协程”唤醒。需要注意的是，Done()方法递减“工作协程”计数后，如果“工作协程”计数变成负数时，将会触发panic，这就要求Add()方法调用要早于Done()方法。此外，通过Add()方法累加的“工作协程”计数要与实际需要等待的“工作协程”数量一致，否则也会触发panic。当“工作协程”计数多于实际需要等待的“工作协程”数量时，“坐等协程”可能会永远无法被唤醒而产生列锁，此时，Go运行时检测到死锁会触发panic。当“工作协程”计数小于实际需要等待的“工作协程”数量时，Done()会在“工作协程”计数变为负数时触发panic。
* Golang context是Golang应用开发常用的并发控制技术，它与WaitGroup最大的不同点是context对于派生goroutine有更强的控制力，它可以控制多级的goroutine。类似树形结构，总的概念就是父类节点取消，子类节点就会被取消，消息传递的时候优先选择子类节点，如果没有再往父类节点搜索。节点有以下形式：（1）WithCancel()创建可以取消的上下文（2）WithDeadline()和Withtimeout()创建可以使用过期时间的上下文（3）WithValue创建可以携带字段的上下文

18. **反射机制**

反射提供一种让程序检查自身结构的能力

19. **测试**

* 单元测试：文件名以test结尾，测试函数头部加上Test，参数为 t \*testing.T
* 性能测试：文件名以test结尾，测试函数头部加上Benchmark，参数为 b \*testing.B，测试的时候要加上参数 -bench=.
* 并发测试：通过t.run（string,func）来实现，func参数也必须是\*testing.T。函数中需要指定t.Parallel()才会进行并发。否则函数会阻塞。还可以将run用run进行包裹。

20. **切片增加**

* append 会改变切片的地址
* append 可以追加nil值

21. **循环变量绑定**

在for循环使用range的时候，获取的变量都是引用状态，如果是不在函数中传递过去或者创建新的变量来接收，很有可能在并发状态变量会被改变，因为变量的地址不会改变。需要显示绑定或者函数传递。

22. **go中的引用类型**

go 中四种引用类型有 slice， channel， function， map