目　录

第八十二天
第八十一天
第八十天
第七十九天
第七十八天
第七十七天
第七十六天
第七十五天
第七十四天
第七十三天
第七十二天
第七十一天
第七十天
第六十九天
第六十八天
第六十七天
第六十六天
第六十五天
第六十四天
第六十三天
第六十二天
第六十一天
第六十天
第五十九天
第五十八天
第五十七天
第五十六天
第五十五天
第五十四天
第五十三天
第五十二天
第五十一天
第五十天

第四十九天
第四十八天
第四十七天
第四十六天
第四十五天
第四十四天
第四十三天
第四十二天
第四十一天
第四十天
第三十九天
第三十八天
第三十七天
第三十六天
第三十五天
第三十四天
第三十三天
redis知识点
第三十二天
第三十一天
第三十天
第二十九天
第二十八天
第二十七天
第二十六天
第二十五天
第二十四天
第二十三天
第二十二天
第二十一天
第二十天
第十九天
第十八天

第十七天
第十六天
第十五天
第十四天
第十三天
第十二天
第十一天
第十天
第九天
第八天
第七天
第六天
第五天
第四天
第三天
第二天
第一天

面试题

面试题

此专栏面试题转自公众号“Golang来啦”,下方有关注二维码。

面试经常被问到的问题

- 1、请你自我介绍一下你自己？
- 2、谈谈你对跳槽的看法？
- 3、工作中你难以和同事、上司相处，你该怎么办？
- 4、你对加班的看法？
- 5、你对薪资的要求？
- 6、你的职业规划？
- 7、你还有什么问题要问吗？
- 8、如果通过这次面试我们单位录用了你，但工作一段时间却发现你根本不适合这个职位，你怎么办？
- 9、在完成某项工作时，你认为领导要求的方式不是最好的，自己还有更好的方法，你应该怎么做？
- 10、如果你的工作出现失误，给本公司造成经济损失，你认为该怎么办？
- 11、你觉得你个性上最大的优点是什么？
- 12、说说你最大的缺点？
- 13、你对于我们公司了解多少？
- 14、请说出你选择这份工作的动机？
- 15、你最擅长的技术方向是什么？
- 16、你能为我们公司带来什么呢？
- 17、最能概括你自己的三个词是什么？
- 18、为什么要离职?
- 19、对工作的期望与目标何在？
- 20、就你申请的这个职位，你认为你还欠缺什么？

- 21、你通常如何处理别人的批评？
- 22、怎样对待自己的失败？
- 23、什么会让你有成就感？
- 24、你为什么愿意到我们公司来工作？
- 25、你和别人发生过争执吗？你是怎样解决的？
- 26、对这项工作，你有哪些可预见的困难？
- 27、如果我录用你，你将怎样开展工作？
- 28、在完成某项工作时，你认为领导要求的方式不是最好的，自己还有更好的方法，你应该怎么做？与上级意见不一是，你将怎么办？
- 29、你工作经验欠缺，如何能胜任这项工作？
- 30、您在前一家公司的离职原因是什么？
- 31、为了做好你工作份外之事，你该怎样获得他人的支持和帮助？
- 32、如果你在这次面试中没有被录用，你怎么打算？
- 33、谈谈你过去做过的成功案例？(工作中遇到什么问题)
- 34、如何安排自己的时间？会不会排斥加班？
- 35、这个职务的期许？
- 36、什么选择我们这家公司？
- 37、谈谈如何适应办公室工作的新环境？
- 38、工作中学习到了些什么？
- 39、除了本公司外，还应聘了哪些公司？
- 40、何时可以到职？

面试技术点

- go语言slice和map底层实现原理
- map底层实现原理：
- slice底层实现原理：
- map进行有序的排序
- 结构体不加tag可以转json字符串吗
- go语言tcp udp具体实现原理
- go语言协程调度原理，协程为什么快
- 你在开发过程中遇到了什么困难 以及怎么解决的
- go语言常见的一些算法（查找算法）
- mysql索引分为几种（b+tree hash表）
- redis比mysql好在哪里
- es具体的使用方法好在哪里
- 400 401 402 403 404状态吗
- slice扩容机制
- 欢迎补充.....

面试题汇总

1、go的调度
2、go struct能不能比较

因为是强类型语言，所以不同类型的结构不能作比较，但是同一类型的实例值是可以比较的，实例不可以比较，因为是指针类型

3、go defer（for defer），先进后出，后进先出

```
func b() {
    for i := 0; i < 4; i++ {
        defer fmt.Print(i)
    }
}
```

4、select可以用于什么，常用语gorotine的完美退出

golang 的 select 就是监听 IO 操作，当 IO 操作发生时，触发相应的动作
每个case语句里必须是一个IO操作，确切的说，应该是一个面向channel的IO操作

5、context包的用途

Context通常被译作上下文，它是一个比较抽象的概念，其本质，是【上下上下】存在上下层的传递，上会把内容传递给下。
在Go语言中，程序单元也就指的是Goroutine

6、client如何实现长连接

server是设置超时时间，for循环遍历的

7、主协程如何等其余协程完再操作

使用channel进行通信，context,select

8、slice，len，cap，共享，扩容

append函数，因为slice底层数据结构是，由数组、len、cap组成，所以，在使用append扩容时，会查看数组后面有没有连续内存快，有就在后面添加，没有就重新生成一个大的素组

9、map如何顺序读取

map不能顺序读取，是因为他是无序的，想要有序读取，首先的解决的问题就是，把 k e y 变为有序，所以可以把key放入切片，对切片进行排序，遍历切片，通过key取值。

10、实现set

```
type inter interface{}
type Set struct {
    m map[inter]bool
```

```
    sync.RWMutex
}

func New() *Set {
    return &Set{
    m: map[inter]bool{},
    }
}
func (s *Set) Add(item inter) {
    s.Lock()
    defer s.Unlock()
    s.m[item] = true
}
```

11、实现消息队列（多生产者，多消费者）

使用切片加锁可以实现

12、大文件排序

归并排序，分而治之,拆分为小文件，在排序

13、基本排序，哪些是稳定的
14、http get跟head

HEAD和GET本质是一样的，区别在于HEAD不含有呈现数据，而仅仅是HTTP头信息。有的人可能觉得这个方法没什么用，其实不是这样的。想象一个业务情景：欲判断某个资源是否存在，我们通常使用GET，但这里用HEAD则意义更加明确。

15、http 401,403

400 bad request，请求报文存在语法错误
401 unauthorized，表示发送的请求需要有通过 HTTP 认证的认证信息
403 forbidden，表示对请求资源的访问被服务器拒绝
404 not found，表示在服务器上没有找到请求的资源

16、http keep-alive

client发出的HTTP请求头需要增加Connection:keep-alive字段
Web-Server端要能识别Connection:keep-alive字段，并且在http的response里指定Connection:keep-alive字段，告诉client，我能提供keep-alive服务，并且"应允"client我暂时不会关闭socket连接

17、http能不能一次连接多次请求，不等后端返回

http本质上市使用socket连接，因此发送请求，接写入tcp缓冲，是可以多次进行的，这也是http是无状态的原因

18、tcp与udp区别，udp优点，适用场景

tcp传输的是数据流，而udp是数据包，tcp会进过三次握手，udp不需要

19、time-wait的作用

20、数据库如何建索引

２１、孤儿进程，僵尸进程
２２、死锁条件，如何避免
２３、linux命令，查看端口占用，cpu负载，内存占用，如何发送信号给一个进程
２４、git文件版本，使用顺序，merge跟rebase

２６、Slice与数组区别，Slice底层结构
２７、项目里的微信支付这块，在支付完微信通知这里，收到两次微信相同的支付通知，怎么防止重复消费（类似接口的幂等性），说了借助Redis或者数据库的事务
２８、项目里的消息推送怎么做的（业务有关）
２９、Go的反射包怎么找到对应的方法（这里忘记怎么问的，直接说不会，只用了DeepEqual，简单讲了DeepEqual）
３０、Redis基本数据结构
３１、Redis的List用过吗？底层怎么实现的？知道但是没用过，不知道怎么实现
３２、Mysql的索引有几种，时间复杂度
３３、InnoDb是表锁还是行锁，为什么（这里答不出来为什么，只说了行锁）
３４、Go的channel（有缓冲和无缓冲）
３５、退出程序时怎么防止channel没有消费完，这里一开始有点没清楚面试官问的，然后说了监听中断信号，做退出前的处理，然后面试官说不是这个意思，然后说发送前先告知长度，长度要是不知道呢？close channel下游会受到0值，可以利用这点（这里也有点跟面试官说不明白）
３６、用过什么消息中间件之类吗？没有
３７、有什么问题吗？评价？后面还有面试，后面再问吧

３８、生产者消费者模式，手写代码（Go直接使用channel实现很简单，还想着面试官会不会不让用channel实现，不用channel的可以使用数组加条件变量），channel缓冲长度怎么决定，怎么控制上游生产速度过快，这里没说出解决方案，只是简单说了channel长度可以与上下游的速度比例成线性关系，面试官说这是一种解决方案
３９、手写循环队列
４０、写的循环队列是不是线程安全，不是，怎么保证线程安全，加锁，效率有点低啊，然后面试官就提醒Go推崇原子操作和channel
４１、写完代码面试官说后面问的问题回答就可以，不知道的话没关系
４２、Linux会不会，只会几个命令，面试官就说一共也就一百多个命令
４３、TimeWait和CloseWait原因
４４、线段树了解吗？不了解，字典树？了解
４５、看过啥源码，nsq（Go的消息中间件），简单问了我里面的waitgroup包证明我看过
４６、sync.Pool用过吗，为什么使用，对象池，避免频繁分配对象（GC有关），那里面的对象是固定的吗？不清楚，没看过这个的源码
４７、有什么问题吗？评价？基础不错，Linux尚缺，Go的理解不够深入，高级数据结构不了解，优点是看源码
４８、后面面试官讲了他们做的东西，主要是广告部分，说日均数据量至少百万以上，多达上亿，高并发使用Go支撑，有微服务，服务治理，说我需要学的东西挺多的

４９、证明二叉树的叶子节点跟度数为2的节点的关系
５０、唯一索引和主键索引
５１、智能指针
５２、字符串解析为数字（考虑浮点型）

５３、单点登录，tcp粘包
５４、手写洗牌
５５、处理粘包断包实现，面试官以为是negle算法有关，解释了下negle跟糊涂窗口综合征有关，然后面试官觉得其他项目是crud就没问了
５６、goroutine调度用了什么系统调用，这个不会，面试官想从go问到操作系统，然后以为 *** 作系统基础不好，就问了操作系统问题
５７、进程虚拟空间分布，全局变量放哪里？答上来了，操作系统就不问了
５８、有没有网络编程，有，怎么看连接状态？netstat，有哪些？ESTABLISHED，LISTEN等等，有异常情况吗？TIME_WAIT很多，为什么？大量短链接
５９、几种基本排序算法说一下，问了堆的时间复杂度，稳定性，为什么不稳定
６０、 topk问题，海量数据topk（回答成切分多次加载内存，然后用维持k长度的有序链表，然后被说时间复杂度不好，提示说还是用堆，然后哦哦哦对）
最长连续字串和，这里我说的解决方案没用dp（对dp不熟），面试官一直引导我dp，还是不会
６１、什么是主键
６２、联合索引和唯一索引
６２、越多的索引越好吗？
６３、建立索引要注意什么？
６４、进程和线程区别？

65、死锁？
66、tcp三次握手
67、http，https
68、状态码401,301,302,201
69、项目我说只有一台机子，所以用的单机部署，面试官说单机也可以部署多个，有什么方法吗？我说docker，问docker有哪些网络，不熟，dockerfile关键字，只答几个。顺便扯了下nginx转发。
70、数据库隔离级别，提交读会造成什么
71、go调度
72、 goroutine泄漏有没有处理，设置timeout，select加定时器
73、mysql高可用的方案
74、进程线程区别
75、排序算法以及时间复杂度
76、怎么学习go
77、go的线程，给他讲了跟goroutine调度
78、io模型，同步阻塞，同步非阻塞，异步
79、cookie和session
实习项目
优缺点
同学的评价
兴趣爱好
有什么offer
80、接口kps测试
81、redis排行榜数据结构（跳跃表），查询时间复杂度
82、redis分布式，如何减少同步延迟
83、mysql能实现redis的功能吗
84、平时怎么学习？
85、看什么书？
86、兴趣爱好
87、看过google四篇分布式论文吗，没看过
89、cap理论，举例
90、LRU算法，LFU
91、讲讲怎么理解网络编程
92、go使用踩过什么坑（for range，数据库连接defer close）
93、go优缺点
95、go的值传递和引用
96、慢查询
97、为什么使用pg
98、redis的数据类型
99、所有左叶子节点的和
100、m个n大小的有序数组求并集，一开始是2路归并，求时间复杂度，后来在面试官提醒直接m路归并，求时间复杂度
101、static关键字，还有其他关键字吗
102、hash表设计，线程安全？
103、 线程自己独享什么
104、网络编程过程
105、select、epoll
106、看什么书
107、排行榜怎么实现
108、go的锁如何实现，用了什么cpu指令
109、go的runtime如何实现
110、看过sql的连接池实现吗
111、ctx包了解吗？有什么用？
112、go什么情况下会发生内存泄漏？（他说ctx没有cancel的时候，这个真不知道）
113、怎么实现协程完美退出？
114、智力题：1000瓶酒中有1瓶毒酒，10只老鼠，7天后毒性才发作，第8天要卖了，怎么求那瓶毒酒？
115、简单dp题，n * n矩阵从左上角到右下角有多少种走法（只限往下和往右走）
116、用channel实现定时器？（实际上是两个协程同步）
117、go为什么高并发好？讲了go的调度模型
118、操作系统内存管理？进程通讯，为什么共享存储区效率最高
119、实现一个hashmap，解决hash冲突的方法，解决hash倾斜的方法
120、怎么理解go的interface
121、100亿个数选top5，小根堆
122、数组和为n的数组对

123、最大连续子数组和
124、redis容灾，备份，扩容
125、跳跃表，为什么使用跳跃表而不使用红黑树
126、输入url后涉及什么
127、tcp怎么找到哪个套接字
128、ipc方式，共享存储区原理
130、进程虚拟空间布局
131、进程状态转换
132、线程的栈在哪里分配
133、多个线程读，一个线程写一个int32会不会有问题，int64呢（这里面试官后来说了要看数据总线的位数，32位的话写int32没问题，int64就有问题）
134、判断二叉树是否为满二叉树
135、lru实现
136、一个大整数（字符串形式表示的），移动字符求比它大的数中最小的
137、点赞系统设计

redis面试题

1.Redis 是一个基于内存的高性能key-value数据库。

2.Redis相比memcached有哪些优势：

- memcached所有的值均是简单的字符串，redis作为其替代者，支持更为丰富的数据类型
- redis的速度比memcached快很多
- redis可以持久化其数据

3.Redis是单线程

- redis利用队列技术将并发访问变为串行访问，消除了传统数据库串行控制的开销

4.Reids常用5种数据类型

- string，list，set，sorted set，hash

6.Reids6种淘汰策略：

- noeviction: 不删除策略, 达到最大内存限制时, 如果需要更多内存, 直接返回错误信息。大多数写命令都会导致占用更多的内存(有极少数会例外。
- **allkeys-lru:**所有key通用; 优先删除最近最少使用(less recently used ,LRU) 的 key。
- **volatile-lru:**只限于设置了 expire 的部分; 优先删除最近最少使用(less recently used ,LRU) 的 key。
- **allkeys-random:**所有key通用; 随机删除一部分 key。
- volatile-random: 只限于设置了 expire 的部分; 随机删除一部分 key。
- volatile-ttl: 只限于设置了 expire 的部分; 优先删除剩余时间(time to live,TTL) 短的key。

7.Redis的并发竞争问题如何解决?

单进程单线程模式，采用队列模式将并发访问变为串行访问。Redis本身没有锁的概念，Redis对于多个客户端连接并不存在竞争，利用setnx实现锁。

8.Redis是使用c语言开发的。

9.Redis前端启动命令

./redis-server

10.Reids支持的语言：

java、C、C#、C++、php、Node.js、Go等。

11.Redis 持久化方案：

Rdb 和 Aof

12.Redis 的主从复制

持久化保证了即使redis服务重启也不会丢失数据，因为redis服务重启后会将硬盘上持久化的数据恢复到内存中，但是当redis服务器的硬盘损坏了可能会导致数据丢失，如果通过redis的主从复制机制就可以避免这种单点故障，

13.Redis是单线程的，但Redis为什么这么快？

1、完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)；

2、数据结构简单，对数据操作也简单，Redis中的数据结构是专门进行设计的；

3、采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；

4、使用多路I/O复用模型，非阻塞IO；这里“多路”指的是多个网络连接，“复用”指的是复用同一个线程

5、使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求；

14.为什么Redis是单线程的？

Redis是基于内存的操作，CPU不是Redis的瓶颈，Redis的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且CPU不会成为瓶颈，那就顺理成章地采用单线程的方案了（毕竟采用多线程会有很多麻烦！）。

15.Redis info查看命令：info memory

16.Redis内存模型

- used_memory：Redis分配器分配的内存总量（单位是字节），包括使用的虚拟内存（即swap）；Redis分配器后面会介绍。used_memory_human只是显示更友好。
- used_memory_rss：Redis进程占据操作系统的内存（单位是字节），与top及ps命令看到的值是一致的；除了分配器分配的内存之外，used_memory_rss还包括进程运行本身需要的内存、内存碎片等，但是不包括虚拟内存。
- mem_fragmentation_ratio：内存碎片比率，该值是used_memory_rss / used_memory的比值。
- mem_allocator：Redis使用的内存分配器，在编译时指定；可以是 libc 、jemalloc或者tcmalloc，默认是jemalloc；截图中使用的便是默认的jemalloc。

17.Redis内存划分

- 数据

作为数据库，数据是最主要的部分；这部分占用的内存会统计在used_memory中。

- 进程本身运行需要的内存

Redis主进程本身运行肯定需要占用内存，如代码、常量池等等；这部分内存大约几兆，在大多数生产环境中与Redis数据占用的内存相比可以忽略。这部分内存不是由jemalloc分配，因此不会统计在used_memory中。

- 缓冲内存

缓冲内存包括客户端缓冲区、复制积压缓冲区、AOF缓冲区等；其中，客户端缓冲存储客户端连接的输入输出缓冲；复制积压缓冲用于部分复制功能；AOF缓冲区用于在进行AOF重写时，保存最近的写入命令。在了解相应功能之前，不需要知道这些缓冲的细节；这部分内存由jemalloc分配，因此会统计在used_memory中。

- 内存碎片

内存碎片是Redis在分配、回收物理内存过程中产生的。例如，如果对数据的更改频繁，而且数据之间的大小相差很大，可能导致redis释放的空间在物理内存中并没有释放，但redis又无法有效利用，这就形成了内存碎片。内存碎片不会统计在used_memory中。

18.Redis对象有5种类型

无论是哪种类型，Redis都不会直接存储，而是通过redisObject对象进行存储。

19.Redis没有直接使用C字符串

(即以空字符'\0'结尾的字符数组)作为默认的字符串表示，而是使用了SDS。SDS是简单动态字符串(Simple Dynamic String)的缩写。

20.Reidis的SDS在C字符串的基础上加入了free和len字段

21.Reids主从复制

复制是高可用Redis的基础，哨兵和集群都是在复制基础上实现高可用的。复制主要实现了数据的多机备份，以及对于读操作的负载均衡和简单的故障恢复。缺陷：故障恢复无法自动化；写操作无法负载均衡；存储能力受到单机的限制。

22.Redis哨兵

在复制的基础上，哨兵实现了自动化的故障恢复。缺陷：写操作无法负载均衡；存储能力受到单机的限制。

23.Reids持久化触发条件

RDB持久化的触发分为手动触发和自动触发两种。

24.Redis 开启AOF

Redis服务器默认开启RDB，关闭AOF；要开启AOF，需要在配置文件中配置：

appendonly yes

25.AOF常用配置总结

下面是AOF常用的配置项，以及默认值；前面介绍过的这里不再详细介绍。

- appendonly no：是否开启AOF
- appendfilename “appendonly.aof”：AOF文件名
- dir ./：RDB文件和AOF文件所在目录
- appendfsync everysec：fsync持久化策略
- no-appendfsync-on-rewrite no：AOF重写期间是否禁止fsync；如果开启该选项，可以减轻文件重写时CPU和硬盘的负载（尤其是硬盘），但是可能会丢失AOF重写期间的数据；需要在负载和安全性之间进行平衡
- auto-aof-rewrite-percentage 100：文件重写触发条件之一
- auto-aof-rewrite-min-size 64mb：文件重写触发提交之一
- aof-load-truncated yes：如果AOF文件结尾损坏，Redis启动时是否仍载入AOF文件

26.RDB和AOF的优缺点

RDB持久化

优点：RDB文件紧凑，体积小，网络传输快，适合全量复制；恢复速度比AOF快很多。当然，与AOF相比，RDB最重要的优点之一是对性能的影响相对较小。

缺点：RDB文件的致命缺点在于其数据快照的持久化方式决定了必然做不到实时持久化，而在数据越来越重要的今天，数据的大量丢失很多时候是无法接受的，因此AOF持久化成为主流。此外，RDB文件需要满足特定格式，兼容性差（如老版本的Redis不兼容新版本的RDB文件）。

AOF持久化

与RDB持久化相对应，AOF的优点在于支持秒级持久化、兼容性好，缺点是文件大、恢复速度慢、对性能影响大。

27.持久化策略选择

（1）如果Redis中的数据完全丢弃也没有关系（如Redis完全用作DB层数据的cache），那么无论是单机，还是主从架构，都可以不进行任何持久化。

（2）在单机环境下（对于个人开发者，这种情况可能比较常见），如果可以接受十几分钟或更多的数据丢失，选择RDB对Redis的性能更加有利；如果只能接受秒级别的数据丢失，应该选择AOF。

（3）但在多数情况下，我们都会配置主从环境，slave的存在既可以实现数据的热备，也可以进行读写分离分担Redis读请求，以及在master宕掉后继续提供服务。

28.redis缓存被击穿处理机制

使用mutex。简单地来说，就是在缓存失效的时候（判断拿出来的值为空），不是立即去load db，而是先使用缓存工具的某些带成功操作返回值的操作（比如Redis的SETNX或者Memcache的ADD）去set一个mutex key，当操作返回成功时，再进行load db的操作并回设缓存；否则，就重试整个get缓存的方法

29.Redis还提供的高级工具

像慢查询分析、性能测试、Pipeline、事务、Lua自定义命令、Bitmaps、HyperLogLog、发布/订阅、Geo等个性化功能。

30.Redis常用管理命令

```
# dbsize 返回当前数据库 key 的数量。
# info 返回当前 redis 服务器状态和一些统计信息。
# monitor 实时监听并返回redis服务器接收到的所有请求信息。
# shutdown 把数据同步保存到磁盘上，并关闭redis服务。
# config get parameter 获取一个 redis 配置参数信息。（个别参数可能无法获取）
# config set parameter value 设置一个 redis 配置参数信息。（个别参数可能无法获取）
# config resetstat 重置 info 命令的统计信息。（重置包括：keyspace 命中数、
# keyspace 错误数、 处理命令数，接收连接数、过期 key 数）
# debug object key 获取一个 key 的调试信息。
# debug segfault 制造一次服务器当机。
# flushdb 删除当前数据库中所有 key,此方法不会失败。小心慎用
# flushall 删除全部数据库中所有 key，此方法不会失败。小心慎用
```

31.Reids工具命令

```
#redis-server：Redis 服务器的 daemon 启动程序
#redis-cli：Redis 命令行操作工具。当然，你也可以用 telnet 根据其纯文本协议来操作
#redis-benchmark：Redis 性能测试工具，测试 Redis 在你的系统及你的配置下的读写性能
$redis-benchmark -n 100000 -c 50
#模拟同时由 50 个客户端发送 100000 个 SETs/GETs 查询
#redis-check-aof：更新日志检查
#redis-check-dump：本地数据库检查
```

32.为什么需要持久化？

由于Redis是一种内存型数据库，即服务器在运行时，系统为其分配了一部分内存存储数据，一旦服务器挂了，或者突然宕机了，那么数据库里面的数据将会丢失，为了使服务器即使突然关机也能保存数据，必须通过持久化的方式将数据从内存保存到磁盘中。

33.判断key是否存在

```
exists key +key名字
```

34.删除key

```
del key1 key2 ...
```

35.缓存和数据库间数据一致性问题

分布式环境下（单机就不用说了）非常容易出现缓存和数据库间的数据一致性问题，针对这一点的话，只能说，如果你的项目对缓存的要求是强一致性的，那么请不要使用缓存。我们只能采取合适的策略来降低缓存和数据库间数据不一致的概率，而无法保证两者间的强一致性。合适的策略包括 合适的缓存更新策略，更新数据库后要及时更新缓存、缓存失败时增加重试机制，例如MQ模式的消息队列。

36.布隆过滤器

bloomfilter就类似于一个hash set，用于快速判某个元素是否存在于集合中，其典型的应用场景就是快速判断一个key是否存在于某容器，不存在就直接返回。布隆过滤器的关键就在于hash算法和容器大小

37.缓存雪崩问题

存在同一时间内大量键过期（失效），接着来的一大波请求瞬间都落在了数据库中导致连接异常。

解决方案：

1、也是像解决缓存穿透一样加锁排队。

2、建立备份缓存，缓存A和缓存B，A设置超时时间，B不设值超时时间，先从A读缓存，A没有读B，并且更新A缓存和B缓存;

38.缓存并发问题

这里的并发指的是多个redis的client同时set key引起的并发问题。比较有效的解决方案就是把redis.set操作放在队列中使其串行化，必须的一个一个执行，具体的代码就不上了，当然加锁也是可以的，至于为什么不用redis中的事务，留给各位看官自己思考探究。

39.Redis分布式

redis支持主从的模式。原则：Master会将数据同步到slave，而slave不会将数据同步到master。Slave启动时会连接master来同步数据。

这是一个典型的分布式读写分离模型。我们可以利用master来插入数据，slave提供检索服务。这样可以有效减少单个机器的并发访问数量

40.读写分离模型

通过增加Slave DB的数量，读的性能可以线性增长。为了避免Master DB的单点故障，集群一般都会采用两台Master DB做双机热备，所以整个集群的读和写的可用性都非常高。读写分离架构的缺陷在于，不管是Master还是Slave，每个节点都必须保

存完整的数据，如果在数据量很大的情况下，集群的扩展能力还是受限于单个节点的存储能力，而且对于Write-intensive类型的应用，读写分离架构并不适合。

41.数据分片模型

为了解决读写分离模型的缺陷，可以将数据分片模型应用进来。

可以将每个节点看成都是独立的master，然后通过业务实现数据分片。

结合上面两种模型，可以将每个master设计成由一个master和多个slave组成的模型。

42. redis常见性能问题和解决方案：

Master最好不要做任何持久化工作，如RDB内存快照和AOF日志文件

如果数据比较重要，某个Slave开启AOF备份数据，策略设置为每秒同步一次

为了主从复制的速度和连接的稳定性，Master和Slave最好在同一个局域网内

尽量避免在压力很大的主库上增加从库

43.redis通讯协议

RESP 是redis客户端和服务端之前使用的一种通讯协议；RESP 的特点：实现简单、快速解析、可读性好

44.Redis分布式锁实现

先拿setnx来争抢锁，抢到之后，再用expire给锁加一个过期时间防止锁忘记了释放。**如果在setnx之后执行expire之前进程意外crash或者要重启维护了，那会怎么样？**set指令有非常复杂的参数，这个应该是可以同时把setnx和expire合成一条指令来用的！

45.Redis做异步队列

一般使用list结构作为队列，rpush生产消息，lpop消费消息。当lpop没有消息的时候，要适当sleep一会再重试。缺点：在消费者下线的情况下，生产的消息会丢失，得使用专业的消息队列如rabbitmq等。**能不能生产一次消费多次呢？**使用pub/sub主题订阅者模式，可以实现1:N的消息队列。

46.Redis中海量数据的正确操作方式

利用SCAN系列命令（SCAN、SSCAN、HSCAN、ZSCAN）完成数据迭代。

47.SCAN系列命令注意事项

- SCAN的参数没有key，因为其迭代对象是DB内数据；
- 返回值都是数组，第一个值都是下一次迭代游标；
- 时间复杂度：每次请求都是O(1)，完成所有迭代需要O(N)，N是元素数量；
- 可用版本：version >= 2.8.0；

48.Redis 管道 Pipeline

在某些场景下我们在一次操作中可能需要执行多个命令，而如果我们只是一个命令一个命令去执行则会浪费很多网络消耗时间，如果将命令一次性传输到 Redis中去再执行，则会减少很多开销时间。但是需要注意的是 pipeline中的命令并不是原子性执行的，也就是说管道中的命令到达 Redis服务器的时候可能会被其他的命令穿插

49.事务不支持回滚

50.手写一个 LRU 算法

```
class LRUCache<K, V> extends LinkedHashMap<K, V> {
    private final int CACHE_SIZE;

    /**
     * 传递进来最多能缓存多少数据
     *
     * @param cacheSize 缓存大小
     */
    public LRUCache(int cacheSize) {
        // true 表示让 linkedHashMap 按照访问顺序来进行排序，最近访问的放在头部，最老访问的放在尾部。
        super((int) Math.ceil(cacheSize / 0.75) + 1, 0.75f, true);
        CACHE_SIZE = cacheSize;
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        // 当 map中的数据量大于指定的缓存个数的时候，就自动删除最老的数据。
        return size() > CACHE_SIZE;
    }
}
```

51.多节点 Redis 分布式锁：Redlock 算法

获取当前时间（start）。

依次向 N 个 Redis节点请求锁。请求锁的方式与从单节点 Redis获取锁的方式一致。为了保证在某个 Redis节点不可用时该算法能够继续运行，获取锁的操作都需要设置超时时间，需要保证该超时时间远小于锁的有效时间。这样才能保证客户端在向某个 Redis节点获取锁失败之后，可以立刻尝试下一个节点。

计算获取锁的过程总共消耗多长时间（consumeTime = end - start）。如果客户端从大多数 Redis节点（>= N/2 + 1) 成功获取锁，并且获取锁总时长没有超过锁的有效时间，这种情况下，客户端会认为获取锁成功，否则，获取锁失败。

如果最终获取锁成功，锁的有效时间应该重新设置为锁最初的有效时间减去 consumeTime。

如果最终获取锁失败，客户端应该立刻向所有 Redis节点发起释放锁的请求。

52.Redis 中设置过期时间主要通过以下四种方式

expire key seconds：设置 key 在 n 秒后过期；
pexpire key milliseconds：设置 key 在 n 毫秒后过期；
expireat key timestamp：设置 key 在某个时间戳（精确到秒）之后过期；
pexpireat key millisecondsTimestamp：设置 key 在某个时间戳（精确到毫秒）之后过期；

53.Reids三种不同删除策略

定时删除：在设置键的过期时间的同时，创建一个定时任务，当键达到过期时间时，立即执行对键的删除操作

惰性删除：放任键过期不管，但在每次从键空间获取键时，都检查取得的键是否过期，如果过期的话，就删除该键，如果没有过期，就返回该键

定期删除：每隔一点时间，程序就对数据库进行一次检查，删除里面的过期键，至于要删除多少过期键，以及要检查多少个数据库，则由算法决定。

54.定时删除

优点：对内存友好，定时删除策略可以保证过期键会尽可能快地被删除，并释放国期间所占用的内存
缺点：对cpu时间不友好，在过期键比较多时，删除任务会占用很大一部分cpu时间，在内存不紧张但cpu时间紧张的情况下，将cpu时间用在删除和当前任务无关的过期键上，影响服务器的响应时间和吞吐量

55.定期删除

由于定时删除会占用太多cpu时间，影响服务器的响应时间和吞吐量以及惰性删除浪费太多内存，有内存泄露的危险，所以出现一种整合和折中这两种策略的定期删除策略。

定期删除策略每隔一段时间执行一次删除过期键操作，并通过限制删除操作执行的时长和频率来减少删除操作对CPU时间的影响。
定时删除策略有效地减少了因为过期键带来的内存浪费。

56.惰性删除

优点：对cpu时间友好，在每次从键空间获取键时进行过期键检查并是否删除，删除目标也仅限当前处理的键，这个策略不会在其他无关的删除任务上花费任何cpu时间。
缺点：对内存不友好，过期键过期也可能不会被删除，导致所占的内存也不会释放。甚至可能会出现内存泄露的现象，当存在很多过期键，而这些过期键又没有被访问到，这会可能导致它们会一直保存在内存中，造成内存泄露。

57.Reids 管理工具：Redis Manager 2.0

github地址

58.Redis常见的几种缓存策略

- Cache-Aside
- Read-Through
- Write-Through
- Write-Behind

59.Redis Module 实现布隆过滤器

Redis module 是Redis 4.0 以后支持的新的特性，这里很多国外牛逼的大学和机构提供了很多牛逼的Module 只要编译引入到Redis 中就能轻松的实现我们某些需求的功能。在Redis 官方Module 中有一些我们常见的一些模块，我们在这里就做一个简单的使用。

- neural-redis 主要是神经网络的机器学，集成到redis 可以做一些机器训练感兴趣的可以尝试
- RedisSearch 主要支持一些富文本的的搜索
- RedisBloom 支持分布式环境下的Bloom 过滤器

60.Redis 到底是怎么实现“附近的人”

使用方式

```
GEOADD key longitude latitude member [longitude latitude member ...]
```

将给定的位置对象（纬度、经度、名字）添加到指定的key。其中，key为集合名称，member为该经纬度所对应的对象。在实际运用中，当所需存储的对象数量过多时，可通过设置多key(如一个省一个key)的方式对对象集合变相做sharding，避免单集合数量过多。

成功插入后的返回值：

```
(integer) N
```

其中N为成功插入的个数。

转自：https://m.toutiao.com/is/JcBNWYs/

redis知识点

目录

学习计划

- Redis的介绍、优缺点、使用场景
- Linux中的安装
- 常用命令
- Redis各个数据类型及其使用场景
- Redis字符串（String）
- Redis哈希（Hash）
- Redis列表（List）
- Redis集合（Set）
- Redis有序集合（sorted set）
- Redis - 瑞士军刀
- 慢查询
- pipeline流水线
- 发布订阅
- bitmap
- HyperLogLog算法
- GEO
- Redis持久化，数据备份与恢复
- RDB
- AOF
- SpringBoot + Jedis + 1主2从3哨兵 实现Redis的高可用
- SpringBoot + Jedis + Redis Cluster代码案例
- 高可用
- 主从复制
- Redis Sentinel
- Redis Cluster
- Redis安全如何保证
- Redis性能测试
- Redis常用客户端总结
- Redis实战使用场景

学习笔记

Redis的介绍、优缺点、使用场景

- Redis是什么： 开源的，基于键值的存储服务系统，支持多种数据类型，性能高，功能丰富

特性（主要有8个特性）：

- 速度快：官方给出的结果是10W OPS，每秒10W的读写(为什么是10W，因为内存的相应时间是100纳秒-10万分之一秒)。数据存储在内存中；使用C语言开发；Redis使用单线程，减少上下文切换。本质原因是计算机存储介质的速度，内存比硬盘优几个数量级）。MemoryCache可以使用多核，性能上优于Redis。
- 持久化：Redis所有的数据保持在内存中，对数据的更新将异步地保存到磁盘上。断掉，宕机？ RDB快照/AOF日志模式来确保。MemoryCache不提供持久化
- 多种数据结构：Redis提供字符串，HashTable, 链表，集合，有序集合；另外新版本的redis提供BitMaps位图，HyperLogLog超小内存唯一值计数，GEORedis3.2提供的地理位置定位。相比memocache只提供字符串的key-value结构
- 支持多种编程语言：Java,PHP,Ruby,Lua,Node
- 功能丰富: 发布订阅，支持Lua脚本，支持简单事务，支持pipline来提高客户端的并发效率
- 简单：单机核心代码23000行，让开发者容易吃透和定制化；不依赖外部库；单线程模型
- 主从复制：主服务器的数据可以同步到从服务器上，为高可用提供可能
- 高可用、分布式：2.8版本后提供Redis-Sentinel支持高可用；3.0版本支持分布式
- 典型应用场景：
- 缓存系统：缓存一些数据减少对数据库的访问，提高响应速度
- 计数器：类似微博的转发数，评论数，incr/decr的操作是原子性的不会出错
- 消息队列系统：发布订阅的模型，消息队列不是很强
- 排行版： 提供的有序集合能提供排行版的功能，例如粉丝数，关注数
- 实时系统：利用位图实现布隆过滤器，秒杀等

安装

- Linux中安装

```
wget http://download.redis.io/releases/redis-5.0.7.tar.gz
tar -zxvf redis-5.0.7.tar.gz
mv redis-5.0.7 /usr/local/redis 不需要先创建/usr/local.redis文件夹
cd /usr/local/redis
make
make install
vi redis.conf
* bind 0.0.0.0 开发访问
* daemonize yes 设置后台运行
redis-server ./redis.conf 启动
redis-cli 进入命令行，进行简单的命令操作
vi redis.conf
> requirepass password 修改密码
redis-cli 再次进入cmd
> shutdown save 关闭redis，同时持久化当前数据
redis-server ./redis.conf 再次启动redis
redis-cli 进入命令行
> auth password
将redis配置成系统服务，redis/utils中自带命令，我们只需修改参数
/usr/local/redis/utils/./install_server.sh
[root~ utils]# ./install_server.sh
Welcome to the redis service installer
```

```
Please select the redis port for this instance: [6379] 默认端口不管
Selecting default: 6379
Please select the redis config file name [/etc/redis/6379.conf] /usr/local/redis/redis.conf 修改配置文件路径
Please select the redis log file name [/var/log/redis_6379.log] /usr/local/redis/redis.log 修改日志文件路径
Please select the data directory for this instance [/var/lib/redis/6379] /usr/local/redis/data 修改数据存储路径
Please select the redis executable path [/usr/local/bin/redis-server]
Selected config:
Port           : 6379
Config file    : /usr/local/redis/redis.conf
Log file       : /usr/local/redis/redis.log
Data dir       : /usr/local/redis/data
Executable     : /usr/local/bin/redis-server
Cli Executable : /usr/local/bin/redis-cli
chkconfig --list | grep redis 查看redis服务配置项
redis_6379      0:off   1:off   2:on    3:on    4:on    5:on    6:off
服务名是redis_6379
```

可执行文件说明

- redis-server: Redis服务器，启动Redis的
- redis-cli: Redis命令行客户端连接
- redis-benchmark: 对Redis做性能测试
- redis-check-aof: AOF文件修复工具
- redis-check-dump: RDB文件检查工具
- redis-sentinel: Sentinel服务器（2.8以后）
- 启动方式
- redis-server: 最简单的默认启动，使用redis的默认参数
- 动态参数启动：redis-server –port yourorderpoint
- 配置文件的方式： redis-server configpath
- 比较：
- 生产环境选择配置启动：单机多实例配置文件可以选择配置文件分开
- Redis客户端返回值
- 状态回复：ping->pong
- 错误恢复：执行错误的回复
- 整数回复：例如incr会返回一个整数
- 字符串回复： get
- 多行字符串回复：mget
- 常用配置
- daemonize: 是否是守护进程（y/n）
- port端口：默认是6379
- logfile：Redis系统日志
- dir:Redis工作目录
- 常用命令：在线练习http://try.redis.io/

```
redis-cli -h x.x.x.x -p x 连接
auth "password" 验证密码
redis-cli --raw可以避免中文乱码
exit 退出
select index 切换到指定的数据库
keys * 显示所有key，如果键值对多不建议使用，keys会遍历所有key，可以在从节点使用;时间复杂度O(N)
```

```
dbsize 算出所有的key的数量，只是数量;时间复杂度O(1)
exists key key是否存在，存在返回1，不存在返回0；时间复杂度O(1)
incr key 将key的值加一，是原子操作
decr key 将key的值加一，会出现复数，是原子操作
del key 删除key，删除成功返回1，失败返回0;时间复杂度O(1)
expire key seconds 设置过期时间，过期之后就不存在了；时间复杂度O(1)
ttl key 查看key剩余的过期时间，key不存在返回-2；key存在没设置过期时间返回-1；(TTL Time To Live)
persist key 去掉key的过期时间，再查看ttl key，返回值是-1，表示key存在并且没有设置过期时间
type key 查看类型；时间复杂度O(1)
config get * 获取配置信息
set key value插入值
sadd myset 1 2 3 4 插入set
get key获取值
del key删除key
cat redis.conf | grep -v "#" | grep -v "^$" 查看配置文件，去除所有的#，去除所有的空格
setnx key value #key不存在，才设置
set key value xx #可以存在，才设置
set key value [exporation EX seconds | PX milliseconds] [NX|EX]
mget key1 key2 key3 批量获取 1次mget=1次网络时间+n次命令时间;时间复杂度O(n)
mset key1 value1 key2 value2 批量插入；时间复杂度O(n)
n次get = n次网络时间 + n次命令时间，mget一次就能完成，省去大量的网络时间
getset key newvalue # set key newvalue并返回旧的value
append key value #将value追加到旧的value
strlen key #获取value的长度，中文占2个字节
incrbyfloat key 3.5 #增加key对应的值
set/get/del, incr(自增1)/decr(自减1)/incrby(incrby key n自增n)/decrby
getrange key start end #获取value从start到end的值
setrange key index value #设置指定下标为一个新的值
hset key field value #给key的field设置值
hget key field #获取key的field的值
hdel key field #删除key的field的值
hgetall key #获取key的所有值
hexists key field # 判断key的field是否存在
hlen key #获取key field的数量
hmset key field1 value1 field2 value2
hmget key field1 field2
hsetnx/hincrby/hdecry/hincrbyfloat
lpush key value1 value2...valueN #从左边插入
rpush key value1 value2...valueN #从右边插入
linsert key before|after value newValue
rinsert key before|after value newValue
lpop key #从左边弹出一个item
rpop key #从右边弹出一个item
lrem key count value #若count等于0或者不填，表示删除所有的value值相等的item;若count>0,表示从左到右删除最多count个value相等的item;若count<0,表示从右到左，删除最多Math.abs(count)个value相等的项
ltrim key start end #按照索引范围修剪列表，可以用来慢删除，因为全删除可能会阻塞redis
lrang key start end #获取key中从start到end的值
lindex key index #取第index的值
llen key #算出列表的长度
lset key index newValue #修改index的值为newValue
blpop key timeout #lpop阻塞版本，timeout是阻塞时间，timeout=0表示死等，lpop会立马返回，有时候数据更新不那么及时，或者消息队列中消息未及时处理，我们可以使用这个
brpop key timeout
lpush + LPOP = STACK
lpush + RPOP = QUEUE
lpush  + ltrim = 有序的集合
lpush + rpop = 消息队列
sadd key value #不支持插入重复元素，失败返回0
srem key element #删除集合中的element元素
smembers key #查看集合元素
sinter key1 key2 #取出相同：交集
sdiff key1 key2 #取出key1中key2没有的元素：差集
```

```
sunion key1 key2 #取出二者所有的元素：并集
sdiff|sinter|sunion store key #将结果存到key中，有时候计算一次耗时
scard key #计算集合大小
sismember key element #判断element是否在集合中
srandmember #返回所有元素，结果是无序的，小心使用，可能结果很大
smembers key #获取集合中的所有元素
spop key #从集合中随机弹出一个元素
scan
SADD = Tagging
SPOP/SRANDMEMBER = Random item
SADD + SINTER = Social Graph
zadd key score element #添加score和element O(logN)：使用xx和跳表的数据结构
zrem key element #删除元素
zscore key element #返回元素的分数
zincrby key increScore element #增加或减少元素分数
zcard key #返回元素的总个数
zrank key element #获取element的排名
zrange key start end [withscores] #返回指定索引范围内的升序元素
zrangebyscore key minScore maxScore [withscore] #返回分数在minScore和maxScore之间的元素
zcount key minScore maxScore #返回有序集合内在指定分数范围内的个数
zremrangebyrank key start end #删除指定排名内的元素
zremrangebyscore key minScore maxScore #删除指定分数内的元素
zrevrang/zrevrange/集合间的操作zsetunion
info replication 查看分片，能够获取到主从的数量和状态
config get databases 获取所有数据库
```

数据结构和内部编码

- Reids支持5中存储的数据格式： String, Hash, List, Set, Sorted Set

String

- redis 的 string 可以包含任何数据。比如jpg图片或者序列化的对象，最大能存储 512MB。
- 使用场景：缓存/计数器/分布式锁/Web集群session共享/分布式系统全局序号（不用每次都拿，一次拿1000个放到内存中）...
- 常用命令：
- 实战：实现分布式的id生成器，可以使用incr的思路，但是实际中会比这复杂

hash

- 是一个键值(key=>value)对集合。Redis hash 是一个 string 类型的 field 和 value 的映射表，hash 特别适合用于存储对象。
- 实战：统计用户主页的访问量， hincrby user:1:info pageview count
- Redis集群架构下不太适合

list

- Redis 列表是简单的字符串列表，按照插入顺序排序。列表最多可存储 232 - 1 元素 (4294967295, 每个列表可存储40多亿)。
- 实战：微博按时间顺序展示消息

set

- 是 string 类型的无序集合，不允许插入重复元素，插入重复元素失败返回0。集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是 O(1)。
- 实战：抽奖系统（量不是很大的时候）；like,star可以放到集合中；标签tag

zset

- 有序集合：有序且无重复元素，和 set 一样也是string类型元素的集合,且不允许重复的成员。不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。zset的成员是唯一的,但分数(score)却可以重复。
- 实战：排行榜

> Redis客户端： Java的Jedis(Socket通信)，Python的redis-py

瑞士军刀

> 慢查询

- 生命周期

两点说明：

1. 慢查询发生在第3阶段，比如keys *等这些需要扫描全表的操作
2. 客户端超时不一定慢查询，但慢查询是客户端超时的一个可能因素

两个配置

- slowlog-log-slower-than=n(微秒)：命令执行时间超过x微秒，会被丢到一个固定长度的慢查询queue中；n<0表示不配置
- slowlog-max-len: 先进先出的队列，固定长度，保存在内存中（重启redis会消失）

配置方法

默认值

- config get slowlog-max-len=128
- config get slowlog-log-slower-than=10000

修改配置文件重启

动态配置

- config set slowlog-max-len 1000
- config set slowlog-log-slower-than 1000

常用命令

- slowlog get [n]:获取慢查询队列
- slowlog len： 获取慢查询队列的长度
- slowlog reset: 清空慢查询队列

运维经验

- slowlog-max-len不要设置过大，默认10ms,通常设置1ms，根据QPS来设置
- slowlog-log-slower-than不要设置过小，通常设置1000左右
- 定期持久化慢查询

pipeline流水线（批量操作）

当遇到批量网络命令的时候，n次时间=n次网络时间+n次命令时间。举个例子，北京到上海的距离是1300公里，光速是3万公里/秒，假设光纤传输速度是光速的2/3，也就是万公里/秒，那么一次命令的传输时间是 1300/20000*2（来回）=13毫秒，什么是pipeline流水线，1次pipeline(n条命令)=1次网络时间+n次命令时间;pipeline命令在redis服务端会被拆分，因此pipeline命令不是一个原子的命令。注意每次pipeline携带数据量；pipeline每次只能作用在一个Redis节点上；M操作和pipeline的区别，M(mset)操作是redis的原生命令，是原子操作，pipeline不是原子操作

```
for(int i = 0; i < 10000; i++>) {
    jedis.hset(key, field, value); //1万次hset差不多要50秒
for(0->100) {
  Pipeline pipeline = jedis.pipelined();
  for(0->100) {
    pipeline.hset(key,field,value);
  }
  pipeline.syncAndReturnAll(); //拆分100次，每次100个命令，大概需要0.7秒
}
```

发布订阅：类似生产者消费者模型

- 角色：发布者（publisher)，频道(channel)，订阅者(subscriber); 发布者将消息发布到频道中，订阅者订阅相关的频道;
- API： publish/subscribe/unsubscribe
- publish channel message : publish sohu:tv “hello world”
- subscribe sohu:tv
- unsubscribe [channel]
- psubscribe [pattern] #订阅模式 sohu*

bitmap:位图：数据量很大的时候节省存储内存，数据量小了，不节省

hyperloglog（算法，数据结构）：

- 极小空间完成独立数量统计，本质是个string
- api: pfadd key element[s]:向hyperloglog添加元素 pfcount key[s]:计算hyperloglog的独立总数 pfmerge key1 key2合并

GEO: 3.2提供的用于计算地理位置信息；数据类型是zset，可以使用zset的删除命令

- 使用场景：微信摇一摇看附近好友
- api:
- geo key longitude latitude member #增加地理位置信息
- geopos key member[n] #获取地理位置信息
- geodist key member1 membe2 [unit] m米 km千米 mi英里 ft尺 获取两地位置的距离
- georadius #算出指定范围内的地址位置信息的集合，语法复杂了点
- 总结下Redis数据结构和类型的常见用法

类型	简介	特性
String	二进制安全	可以包含任何数据， 比如JPG图片或者月 一个键最大能存储5
Hash	键值对集合，即编程中的Map	适合存储对象， 并且可以向数据库中 一个属性 （Memcache中需 序列化成对象之后再 然后序列化回去）
List	双向链表	增删快， 提供了操作某一元素
Set	哈希表实现，元素不重复	添加/删除/修改的复 （1），为集合提供 差集的操作
Zset	将Set中的元素增加一个double类型的权重score， 按照score排序	数据插入集合就好
Hyperloglog	本质是string	极小空间完成独立数
GEO	数据类型是zset	存储地理位置信息， 并提供计算距离等操
Bitmap	位图	数据量很大的时候 数据量小了不节省

- String 简单动态字符串 Simple Dynamic String, SDS

```
Redis没有直接使用C语言的传统字符串表示，而是自己构建了一种名为简单动态字符串（Simple Dynamic String, SDS）的
抽象类型，并将SDS用作Redis的默认字符串表示。
每个sds.h/sdshdr结构表示一个SDS值：
struct sdshdr {
  int len; // 记录buf数组中已经使用的字节数量
  int free; // 记录buf数组中未使用字节的数量
```

```
    char buf[]; // 字节数组，用于保存字符串。SDS遵循C字符串以空字符结尾的惯例
}
```

Redis持久化

- 持久化的作用：redis所有数据保存在内存中，对数据的更新将异步地保存到磁盘上。
- 主流数据库持久化实现方式：快照（MySQL Dump/Redis RDB），写日志(MySQL Binlog/Redis AOF)

RDB:

- 创建RDB文件（二进制文件）到硬盘中，启动后载入RDB文件到内存

三种触发机制

- save(同步) - 会产生阻塞
- 文件策略：如存在老的RDB文件，新的替换老的，新的会先生成到一个临时文件
- bgsave(异步) - 不会阻塞
- 客户端执行bgsave之后，redis会使用linux的一个fork()命令生成主进程的一个子进程（fork的操作会执行一个内存页的拷贝，使用copy-on-write策略），子进程会创建RDB文件，创建完毕后将成功的消息返回给redis。fork()出来的子进程执行快的话不会阻塞主进程，否则也会阻塞redis，阻塞的实际点就是生成出来这个子进程。由于是异步，在创建的过程中还有其他命令在执行，如何保证RDB文件是最新的呢？在数据量大的时候bgsave才能突出优点。

命令savebgsaveIO类型同步异步阻塞是是（阻塞发生在fork子进程复杂度O(n)O(n)优点不会消耗额外内存不阻塞客户端命令缺点阻塞客户端命令需要fork,消耗内存

- 自动触发：多少秒内有多少changes会异步(bgsave)生成一个RDB文件，如60秒内有1W条changes，默认的规则，可以改；不是很好吧，无法控制频率；另外两条是900秒内有1条changes, 300秒内有10条changes；

配置

- dbfilename dump.rdb
- dir ./
- stop-writes-on-bgsave-error yes 当bgsave发生错误是停止写RDB文件
- rdbcompression yes 采用压缩格式
- rdbchecksum yes 采用校验和

其他不能忽视的点:

全量复制：debug reload；shutdown save会执行rdb文件的生成

AOF：

- RDB现存问题：耗时，耗性能(fork,IO)，不可控(突然宕机)
- AOF：redis中的cmd会先刷新到缓冲区，然后更具配置AOF的策略，异步存追加到AOF文件中，发生宕机后，可以通过-AOF恢复，基本上数据是完整的
- AOF的三种策略(配置的三种属性)
- always：来一条命令写一条；不丢失数据，IO开销较大
- everysec：每秒把缓冲区fsync到AOF文件；丢1秒数据
- no：操作系统决定什么时候把缓冲区同步到AOF就什么时候追加；不用配置，但是不可控，取决于操作系统

AOF重写

- 如果AOF文件很大的话，恢复会很慢，AOF的重写是优化一些命名，使其变成1条，对于过期数据没必要Log,本质是把过期的没有用的，重复的过滤掉，以此减少磁盘占用量，加速恢复。极端的例子，1亿次incr，实际只需要set counter n就够了
- 重写的两种方式
- bgrewriteaof：异步执行，redis fork出一个子进程，然后进行AOF重写
- AOF重写配置
- auto-aof-rewrite-min-size: AOF文件到达多大的时候才开始重写
- auto-aof-rewrite-percentage: AOF文件的增长率到达了多大才开始重写

统计

- aof_current_size AOF当前尺寸 字节
- aof_base_size AOF上次重启和重写的尺寸 字节，方便自动重写判断

重写触发机制(同时满足如下两条)

- aof_current_size > auto-aof-rewrite-min-size
- (aof_current_size - aof_base_size) / aof_base_size > auto-aof-rewrite-percentage

其他配置

- appendonly yes
- appendfilename “”
- appendfsync everysec
- dir /xx
- no-appendfsync-on-rewrite yes AOF在重启之后恢复，要权衡是否开启AOF日志追加的功能，这个时候IO很大，如果设置为yes，也就意味着在恢复之前的日志数据会丢失

RDB & AOF最佳策略：RDB优先于AOF先启用

- RDB：建议关掉，集中管理，在从节点开RDB
- AOF：建议开启，每秒刷盘
- 最佳策略：小分片（log文件分片）

常见问题

- fork操作：是一个同步操作，做一个内存页的拷贝；与内存量息息相关，内存越大，耗时越长；执行info命令，有个latest_fork_usec的值，看下上次fork执行耗时

- 进程外开销：

- CPU：RDB AOF文件生成，属于CPU密集型操作（不要和CPU密集型应用部署在一起，减少RDB AOF频率）；内存：fork内存开销；硬盘：IO开销大，选用SSD磁盘

- AOF追加阻塞：主线程将命令刷到AOF缓冲区，同步线程同步命令到硬盘，同时主线程会对比上次fsync的时间，如果大于2秒就阻塞主线程，否则不阻塞，主线程这么做是为了达到每秒刷盘的目的，让子线程完成AOF，以此来达到数据同步。

AOF发生阻塞怎么定位：redis日志/info persistence(aof_delayed_fsync累计阻塞次数,是累计，不好分清什么时候发生阻塞)

- 单机多实例部署

高可用

Redis主从复制

- 主从复制：单机故障/容量瓶颈/QPS瓶颈；一个master可以有多个slave，一个slave只能有一个master，数据必须是单流向，从master流向slave

复制的配置:

- 使用slaeof命令，在从redis中执行slave masterip:port使其成为master的从服务器，就能从master拉取数据了；执行slaveof no one清除掉不成为从节点，但是数据不清楚；
- 修改配置， slaveof ip port / slave-read-only yes(从节点只做都操作)；配置要更改的话，要重启，所以选择的时候谨慎

全量复制

- run_id(使用info server可以看到run_id),重启之后run_id就没有了，当从服务器去复制主服务器，主服务器run_id会在从服务器上做一个标识，当从服务器发现主服务器的run_id发生了变化，说明主服务器发生了变化（重启或者什么的），那么从服务器就要把主服务器的数据都同步过来
- 偏移量：部分复制中的一个依据，后面说
- 解析下上面的全量复制的过程，slave向master发送psync的命令要去master全量复制数据（PSYNC ，其中?表示我不知道master的runId啊，第一次连嘛，-1表示我都要，这时候slava咱啥也不知道），master大人收到了小弟的请求之后，大方的把自己的runId/offset发了过去，小弟收到后先存下来；在master大人把自个的信息发给小弟之后，立马投入了创建快照RDB的工作，一个bgsave命令立马开工，RDB生产了就发给slave；咦，细心的我们发现你这不对啊，你master创建快照到创建完成这之间新增的数据咋办，master吭吭了两声，我在开始快照的那一刻，后期的所有写命令都额外往buffer中存了一份，来保证我给你的是完整的，当我发送完RDB之后，立马给你发buffer；slave小弟内心对master大人产生了膜拜之情，收到了RDB/buffer之后，先把自己的老数据flush掉，然后load RDB,把最新的buffer刷一遍，分分钟让自己向master看齐。
- 开销：bgsave时间， RDB文件网络传输时间，从节点清空数据时间，从节点加载RDB的时间，可能的AOF重写时间
- 解释下上面的部分复制的过程，当遇到网络抖动，那这段时间内数据在slave上就会发生丢失，那么这些数据slave是不知道的，在2.8之前redis会重新做一次全量复制，但是很显然这样做开销很大，2.8之后提出部分复制的功能；当matster发现slave连接不上的时候，master在进行写操作的时候，也会往缓冲区写，等到下一次slave连上之后，slave会发送一条pysnc {offset}{runId}的命令，其中offset是slave自己的，相当于告诉master我的偏移量是多少，master判断slave的offset在缓冲区内（缓冲区有start/end offset）就向slave发送continue命令，然后把这部分数据发送给slave；当master发现slave这个offset偏移量很大的时候，也就意味着slave丢失了很多数据，那么就进行一次全量复制

故障处理:

- master/slave宕机的情况，主从模式没有实现故障的完全自动转移
- 常见问题：
- 读写分离：读流量分摊到从节点，可能遇到复制数据延迟，也可能读到过期的数据，从节点故障怎么办
- 主从配置不一致：主从maxmemory不一致，可能会丢失数据；主从内存不一致

- 规避全量复制：第一次不可避免：小主节点，低峰处理（夜间）；主节点重启后runId发生了变化
  规避复制风暴
- 单机主节点复制风暴，如果是1主N从，当master重启之后，所有的slave都会发生全量复制，可想而知这样非常容易造成redis服务的不可用

Redis-Sentinel

- 主从复制高可用？
- 手动故障转移，例如选出新的slave做master；写能力和存储能力受限；
- 架构说明
- Redis Sentinel是一个监控redis主从以及实施故障转移的工具，sentinel不是一个是多个的（会选举出一个master sentinel），这样可以保证sentinel的高可用和公平（不是一个sentinel判断不可用就不可用），可以把Redis Sentinel看成一个redis的额外进程，用来监控reids服务的可用与不可用；客户端不再记住redis的地址，而是记录sentinel的地址，sentinel知道谁是真的master;当多个sentinel发现并确认master有问题，sentinel内部会先选出来一个领导，让这个领导来完成故障的转移（因为执行slave no noe/new master这些命令只需要一个sentinel就够了），sentinel从slave中选举一个作为master，然后通知其他的slave去新的master获取数据。sentinel可以监控多套master-slave，
- 安装配置
- 配置开启主从节点
- 配置开启sentinel监控主节点（sentinel是特殊的redis）：sentinel默认端口是23679

```
sentinel monitor mymaster 127.0.0.1 7000 2 监控的主节点名字是mymaster，2表示2个sentinel觉得当前master有问题提才发生故障转移
sentinel down-after-milliseconds mymaster 30000 表示30秒不通之后就停掉master
sentinel parallel-syncs mymaster 1 表示每次并发的复制是1个在复制，这样可以减少master的压力
sentinel failover-timeout mymaster 180000 故障转移时间
```

- 实现原理: redis sentinel做失败判定和故障转移
- redis sentinel内部有三个定时任务

1. 每10秒每个sentinel对master和slave执行info：可以从replication中发现slave节点，确认主从关系
2. 每2秒每个sentinel通过master节点的channel交换信息（pub/sub): 什么意思呢，master节点上有个发布订阅的频道用于sentinel节点进行信息交换，利用的原理就是每个sentinel发布一个信息其他sentinel都可以收到这样一个原理，这个信息都包含当前sentinel节点的信息，以及它当前对master/slave做出的判断。这个频道是啥呢，1. 1. sentinel:hello，这个名字内部规定的
3. 每1秒每个sentinel对其他sentinel和redis执行ping操作-心跳检测，是失败判断的依据

主观下线和客观下线：

- sentinel monitor quorum是法定人数，有quorum个sentinel认为master不可用了那么master就会被客观下线
- sentinel down-after-milliseconds 一个sentinel如果在timeout毫秒内没收到master的回复就做主动下线的操作
- 主观下线：每个sentinel节点对redis节点失败都有自己的判断，这里的节点可以是master，也可以是slave
- 客观下线：所有的sentinel节点对某个redis节点认为失败的个数达到quorum个才下线

领导者选举

- 为啥要选领导者，因为只需要一个sentinel节点就能完成故障转移。怎么选举呢？
- 每个做完主观下线的sentinel节点（就是发现某个节点不可用了，并发出了自己的判断的节点）都会向其他sentinel节点发送sentinel is-master-down-by-addr命令，要求将自己设置为领导者。那么收到这个命令的sentinel如果在之前没有同意过其他sentinel的话，就会同意这个请求，否则拒绝，换句话说每个sentinel只有一个同意票，这个同意票给第一个问自

己的节点。好了，票发完了，如果这个sentinel节点发现自己拥有的票数超过sentinel集合半数并且操作quorum，那么它将成为领导者；如果此过程有多个sentinel节点成为领导者，那么过段时间将重新进行一次选举。领导者选举使用的是一个Raft算法，以上是抽象过程。所以sentinel的个数（>=3最好为奇数）和quorum的个数需要合理配置。

故障转移（sentinel领导者节点完成）

1. 从slave节点中选举一个“合适的”节点作为新的master节点
2. 对上面的slave节点执行slaveof no one命令让其成为master节点
3. 向剩余的slave节点发送命令，让它们成为新的master节点的slave节点，复制规则和parallel-syncs（允许并行复制的个数）参数有关。复制的过程master是做了优化的，只需要一个RDB的生成，然后同时向slave节点发送RDB和buffer，有一定的开销，特别是网络
4. 更新对原来master节点配置为slave，并保持对其“关注”，当其恢复后命令它去复制新的master节点

如何选择合适的slave的节点

1. 选择slave-priority(slave节点优先级)最高的slave节点，如果存在就返回，不存在则继续。一般不配置这个参数，什么情况下配置呢，当有一台slave节点的机器配置很高，我们希望当master挂了之后它能成为新的master时，做这个设置。
2. 选择复制偏移量最大的slave节点（复制的最完整），如果存在则返回，不存在则继续。
3. 选择runId最小的slave节点。runId最小就是最早的slave节点，假设它复制的最多。

运维和开发

- 节点运维：上下和下线
- 机器下线：机器因为不能用了或者过保等什么原因要换机器
- 机器性能不足：例如CPU、内存、硬盘、网络等
- 节点自生保障：例如服务不稳定等
- sentinel failover 主节点主动故障转移，已经选举了sentinel领导者，所以上述过程可以省略
- 从节点下线：临时下线还是永久下线。永久下线可能要清理掉一些配置文件，从节点下线的时候也要考虑读写分离的情况，因为这时候有可能正在读。
- 节点上线： 主节点执行sentinel failover进行替换，从节点slaveof即可，sentinel节点可以感知

JedisSentinelPool的实现 看源代码

- switch-master: 切换主节点
- convert-to-slave: 切换从节点
- sdown：主观下线

Redis-Cluster

为什么需要集群

- 并发量/
- 数据量： 业务需要500G怎么办，机器内存是16~256G
- 网络流量 Redis 3.0版本提供了分布式技术

数据分布

- 数据分区

两种方式

顺序分区

哈希分区

- 节点取余分区
- 新增节点之后基本所有数据要产生飘逸，一般产生多倍扩容节点，飘逸量少
- 一致性哈希分区
- 解决了哈希分区中新增节点造成书飘逸的问题
- 虚拟槽分区
- 好复杂没咋懂

搭建集群

原生安装(模拟3主3从)

1. 配置开启节点 enable-sync yes

```
port ${port}
daemonize yes
dir "/path"
dbfilename "dump-${port}.rdb"
logfile "${port}.log"
cluster-enabled yes
cluster-config-file nodes-${port}.conf #redis启动后这个nodes-port.conf会自动生成
```

开启命令 redis-server redis-7000.conf / redis-server redis-7001.conf /..., 此时开启的节点都是相互孤立的没有任何通信, 查看当前状态

```
127.0.0.1:7000> cluster info
cluster_state:fail
cluster_slots_assigned:0
cluster_slots_ok:0
cluster_slots_pfail:0
cluster_slots_fail:0
cluster_known_nodes:1
cluster_size:0
cluster_current_epoch:0
cluster_my_epoch:0
cluster_stats_messages_sent:0
cluster_stats_messages_received:0
```

显示当前是cluster状态 [外链图片转存失败,源站可能有防盗链机制,建议将图片保存下来直接

2. meet(节点的握手)

```
redis-cli -h 127.0.0.1 -p 7000 cluster meet 127.0.0.1 7001
[root@xx cluster]# redis-cli -p 7000 cluster meet 47.x.x.16 7001
OK
[root@xx cluster]# redis-cli -p 7000 cluster nodes
862e370d2342cf6bf883421003846e171770234e :7000@17000 myself,master - 0 0 0 connected
886b6e6c29f985f7f85acb1bf548d0937918eca3 4.x.x.6:7001@17001 handshake - 0 0 0 connected

redis-cli -h 127.0.0.1 -p 7000 cluster meet 127.0.0.1 7002
redis-cli -h 127.0.0.1 -p 7000 cluster meet 127.0.0.1 7003
redis-cli -h 127.0.0.1 -p 7000 cluster meet 127.0.0.1 7004
redis-cli -h 127.0.0.1 -p 7000 cluster meet 127.0.0.1 7005
```

这地方注意啊，手动配置的时候有个巨坑（坑了我1天）：在redis cluster架构中，每个redis要开发两个端口，比如一个是6379，那么另一个就是加10000之后的端口号，比如16379。16379端口是用来进行节点间通信的，也就是cluster bus集群总线，cluster bus的通信用来进行故障检测、配置更新、故障转移授权等操作。

3.指派槽

```
redis-cli -h 127.0.0.1 -p 7000 cluster addslots {0...5461}
redis-cli -h 127.0.0.1 -p 7001 cluster addslots {5462...10922}
redis-cli -h 127.0.0.1 -p 7002 cluster addslots {10923...16383}
```

4.主从关系的分配

```
redis-cli -h 127.0.0.1 -p 7003 cluster replicate ${node-id-7000}
redis-cli -h 127.0.0.1 -p 7004 cluster replicate ${node-id-7001}
redis-cli -h 127.0.0.1 -p 7005 cluster replicate ${node-id-7002}
```

```
常用命令
cluster notes 查看集群的node几点情况
cluster slots 查看slots的分布
cluster info 查看当前集群的状态，这个命令可以看到集群是否属于可用状态
redis-cli -c -p x 注意加-c，表示集群
```

- 开始的一串字符是nodeid, 整行表示的意识是我的nodeId是多少，哪个ip的那个端口，我是主还是从，是从的话从的谁（谁的nodeId），是主的话还能看到slots的分布情况
- 官方工具安装
- redis-trib.rb实现对redis集群的自动化安装
- 集群伸缩
- 客户端路由
- 集群原理
- 集群是怎么通信的：16379 端口号是用来进行节点间通信的，也就是 cluster bus 的东西，cluster bus 的通信，用来进行故障检测、配置更新、故障转移授权。cluster bus 用了另外一种二进制的协议，gossip 协议，用于节点间进行高效的数据交换，占用更少的网络带宽和处理时间。
- 开发运维常见问题

Redis事务

Redis 事务可以一次执行多个命令， 并且带有以下三个重要的保证：

- 批量操作在发送 EXEC 命令前被放入队列缓存。
- 收到 EXEC 命令后进入事务执行，事务中任意命令执行失败，其余的命令依然被执行。
- 在事务执行过程，其他客户端提交的命令请求不会插入到事务执行命令序列中。
- Redis事务从开始到执行会经历以下三个阶段：开始事务 -> 命令入队 -> 执行事务。单个 Redis 命令的执行是原子性的，但 Redis 没有在事务上增加任何维持原子性的机制，所以 Redis 事务的执行并不是原子性的。事务可以理解为一个打包的批量执行脚本，但批量指令并非原子化的操作，中间某条指令的失败不会导致前面已做指令的回滚，也不会造成后续的指令不做。这是官网上的说明 From redis docs on transactions: It’s important to note that even when a command fails, all the other commands in the queue are processed – Redis will not stop the processing of commands.
- Redis 通过监听一个 TCP 端口或者 Unix socket 的方式来接收来自客户端的连接，当一个连接建立后，Redis 内部会进行以下一些操作：
- 首先，客户端 socket 会被设置为非阻塞模式，因为 Redis 在网络事件处理上采用的是非阻塞多路复用模型。

- 然后为这个 socket 设置 TCP_NODELAY 属性，禁用 Nagle 算法
- 然后创建一个可读的文件事件用于监听这个客户端 socket 的数据发送
- Redis 管道技术可以在服务端未响应时，客户端可以继续向服务端发送请求，并最终一次性读取所有服务端的响应。管道技术最显著的优势是提高了 redis 服务的性能。

Redis 分区

- 分区是分割数据到多个Redis实例的处理过程，因此每个实例只保存key的一个子集。

分区的优势：

- 通过利用多台计算机内存的和值，允许我们构造更大的数据库。
- 通过多核和多台计算机，允许我们扩展计算能力；通过多台计算机和网络适配器，允许我们扩展网络带宽。

分区的不足：

- 涉及多个key的操作通常是不被支持的。举例来说，当两个set映射到不同的redis实例上时，你就不能对这两个set执行交集操作。
- 涉及多个key的redis事务不能使用。
- 当使用分区时，数据处理较为复杂，比如你需要处理多个rdb/aof文件，并且从多个实例和主机备份持久化文件。
- 增加或删除容量也比较复杂。redis集群大多数支持在运行时增加、删除节点的透明数据平衡的能力，但是类似于客户端分区、代理等其他系统则不支持这项特性。然而，一种叫做presharding的技术对此是有帮助的。
- 分区类型：Redis 有两种类型分区。 假设有4个Redis实例 R0，R1，R2，R3，和类似user:1，user:2这样的表示用户的多个key，对既定的key有多种不同方式来选择这个key存放在哪个实例中。也就是说，有不同的系统来映射某个key到某个Redis服务，关注+转发后，私信【Redis】获取300多页的Redis实战学习笔记。

范围分区

- 最简单的分区方式是按范围分区，就是映射一定范围的对象到特定的Redis实例。比如，ID从0到10000的用户会保存到实例R0，ID从10001到 20000的用户会保存到R1，以此类推。这种方式是可行的，并且在实际中使用，不足就是要有一个区间范围到实例的映射表。这个表要被管理，同时还需要各 种对象的映射表，通常对Redis来说并非是好的方法。

哈希分区

- 另外一种分区方法是hash分区。这对任何key都适用，也无需是object_name:这种形式，像下面描述的一样简单：用一个hash函数将key转换为一个数字，比如使用crc32 hash函数。对key foobar执行crc32(foobar)会输出类似93024922的整数。对这个整数取模，将其转化为0-3之间的数字，就可以将这个整数映射到4个Redis实例中的一个了。93024922 % 4 = 2，就是说key foobar应该被存到R2实例中。注意：取模操作是取除的余数，通常在多种编程语言中用%操作符实现。【当分区较多或发生变化的时候需要处理一些额外的情况】

其他

Redis设置port为6379的原因

关于I/O多路复用(又被称为“事件驱动”)，首先要理解的是，操作系统为你提供了一个功能，当你的某个socket可读或者可写的时候，它可以给你一个通知。这样当配合非阻塞的socket使用时，只有当系统通知我哪个描述符可读了，我才去执行read操作，可以保证每次read都能读到有效数据而不做纯返回-1和EAGAIN的无用功。

写操作类似。操作系统的这个功能通过select/poll/epoll/kqueue之类的系统调用函数来使用，这些函数都可以同时监视多个描述符的读写就绪状况，这样，多个描述符的I/O操作都能在一个线程内并发交替地顺序完成，这就叫I/O多路复用，这里的“复用”指的是复用同一个线程。

下面举一个例子，模拟一个tcp服务器处理30个客户socket。假设你是一个老师，让30个学生解答一道题目，然后检查学生做的是否正确，你有下面几个选择：

1. 第一种选择：按顺序逐个检查，先检查A，然后是B，之后是C、D。。。这中间如果有一个学生卡住，全班都会被耽误。这种模式就好比，你用循环挨个处理socket，根本不具有并发能力。

2. 第二种选择：你创建30个分身，每个分身检查一个学生的答案是否正确。 这种类似于为每一个用户创建一个进程或者线程处理连接。

3. 第三种选择，你站在讲台上等，谁解答完谁举手。这时C、D举手，表示他们解答问题完毕，你下去依次检查C、D的答案，然后继续回到讲台上等。此时E、A又举手，然后去处理E和A。。。 这种就是IO复用模型，Linux下的select、poll和epoll就是干这个的。将用户socket对应的fd注册进epoll，然后epoll帮你监听哪些socket上有消息到达，这样就避免了大量的无用操作。此时的socket应该采用非阻塞模式。

这样，整个过程只在调用select、poll、epoll这些调用的时候才会阻塞，收发客户消息是不会阻塞的，整个进程或者线程就被充分利用起来，这就是事件驱动，所谓的reactor模式。

什么时候适合用缓存

数据访问频率

- 访问频率高，适合用缓存，效果好
- 访问频率低，不建议使用，效果不佳

数据读写比例

- 读多写少，适合缓存，效果好
- 读少写多，不建议使用，效果不佳

数据一致性

- 一致性要求低，适合缓存，效果好
- 一致性要求高，不建议缓存，效果不佳

Redis中的缓存穿透、缓存雪崩、缓存击穿

- 缓存穿透

缓存穿透，是指查询一个数据库一定不存在的数据。正常的使用缓存流程大致是，数据查询先进行缓存查询，如果key不存在或者key已经过期，再对数据库进行查询，并把查询到的对象，放进缓存。如果数据库查询对象为空，则不放进缓存。

1. 参数传入对象主键ID
2. 根据key从缓存中获取对象
3. 如果对象不为空，直接返回
4. 如果对象为空，进行数据库查询
5. 如果从数据库查询出的对象不为空，则放入缓存（设定过期时间）

想象一下这个情况，如果传入的参数为-1，会是怎么样？这个-1，就是一定不存在的对象。就会每次都去查询数据库，而每次查询都是空，每次又都不会进行缓存。假如有恶意攻击，就可以利用这个漏洞，对数据库造成压力，甚至压垮数据库。即便是采用UUID，也是很容易找到一个不存在的KEY，进行攻击。

小编在工作中，会采用缓存空值的方式，也就是【代码流程】中第5步，如果从数据库查询的对象为空，也放入缓存，只是设定的缓存过期时间较短，比如设置为60秒。 redisTemplate.opsForValue().set(String.valueOf(goodsId), null, 60, TimeUnit.SECONDS); // 设置过期时间

解决方案

- 接口层增加校验，如用户鉴权校验，id做基础校验，id<=0的直接拦截；
- 从缓存取不到的数据，在数据库中也没有取到，这时也可以将key-value对写为key-null，缓存有效时间可以设置短点，如30秒（设置太长会导致正常情况也没法使用）。这样可以防止攻击用户反复用同一个id暴力攻击
- 缓存雪崩

缓存雪崩，是指在某一个时间段，缓存集中过期失效。

产生雪崩的原因之一，比如在写本文的时候，马上就要到双十二零点，很快就会迎来一波抢购，这波商品时间比较集中的放入了缓存，假设缓存一个小时。那么到了凌晨一点钟的时候，这批商品的缓存就都过期了。而对这批商品的访问查询，都落到了数据库上，对于数据库而言，就会产生周期性的压力波峰。

小编在做电商项目的时候，一般是采取不同分类商品，缓存不同周期。在同一分类中的商品，加上一个随机因子。这样能尽可能分散缓存过期时间，而且，热门类目的商品缓存时间长一些，冷门类目的商品缓存时间短一些，也能节省缓存服务的资源。

其实集中过期，倒不是非常致命，比较致命的缓存雪崩，是缓存服务器某个节点宕机或断网。因为自然形成的缓存雪崩，一定是在某个时间段集中创建缓存，那么那个时候数据库能顶住压力，这个时候，数据库也是可以顶住压力的。无非就是对数据库产生周期性的压力而已。而缓存服务节点的宕机，对数据库服务器造成的压力是不可预知的，很有可能瞬间就把数据库压垮。

解决方案：

- 缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生。
- 如果缓存数据库是分布式部署，将热点数据均匀分布在不同搞得缓存数据库中。
- 设置热点数据永远不过期。
- redis 持久化，一旦重启，自动从磁盘上加载数据，快速恢复缓存数据。
- 缓存击穿

缓存击穿，是指存在hot key，在不停的扛着大并发，大并发集中对这一个点进行访问，当这个key在失效的瞬间，持续的大并发就穿破缓存，直接请求数据库，就像在一个屏障上凿开了一个洞。

小编在做电商项目的时候，把这货就成为“爆款”。 其实，大多数情况下这种爆款很难对数据库服务器造成压垮性的压力。达到这个级别的公司没有几家的。所以，务实主义的小编，对主打商品都是早早的做好了准备，让缓存永不过期。即便某些商品自己发酵成了爆款，也是直接设为永不过期就好了。

解决方案

- 设置热点数据永远不过期。
- 加互斥锁，互斥锁参考代码如下：

说明：

1）缓存中有数据，直接走上述代码13行后就返回结果了

2）缓存中没有数据，第1个进入的线程，获取锁并从数据库去取数据，没释放锁之前，其他并行进入的线程会等待100ms，再重新去缓存取数据。这样就防止都去数据库重复取数据，重复往缓存中更新数据情况出现。另外看到一种方案，当查询缓存没有数据时，进而去数据库查找，即使数据库没有返回结果也要创建缓存，这样做是避免缓存击穿，但是要注意这个key设置的过期时间要短。

3）当然这是简化处理，理论上如果能根据key值加锁就更好了，就是线程A从数据库取key1的数据并不妨碍线程B取key2的数据，上面代码明显做不到这点。

百问

Redis番外篇

Redis 最开始的设计可能就是想做一个缓存来用。但是分布式环境复杂，暴露的问题可能比较多，所以 Redis 就要做集群。做集群后，可能和 Memcahed 效果类似了，我们要超越它，所以可能就有了多数据类型的存储结构。光做缓存，如何已宕机数据就丢失了。我们的口号是超越 Memcahed，所以我们要支持数据持久化。于是可能就有了 AOF 和 RDB，就可以当数据库来用来。这样 Redis 的高效可靠的设计，所以它又可以用来做消息中间件。这就是 Redis 的三大特点，可以用来做：缓存、数据库和消息中间件。

再来说说，Redis 如何设计成但进程单线程的？

根据官方的测试结果《How fast is Redis？》来看，在操作内存的情况下，CPU 并不能起到决定性的作用，反而可能带来一些其他问题。比如锁，CPU 切换带来的性能开销等。这一点我们可以根据官方的测试报告，提供的数据来证明。而且官方提供的数据是可以达到100000+的QPS（每秒内查询次数），这个数据并不比采用单进程多线程 Memcached 差！所以在基于内存的操作，CPU不是 Redis 瓶颈的情况下，官方采用来单进程单线程的设计。

Redis单线程为什么这么快？

快的原因有主要三点：

1. 纯内存操作：Redis是基于内存的，所有的命令都在内存中完成，内存的响应速度相比硬盘是非常快的，内存的响应时间大约是100纳秒，Redis官方给出的OPS是10W
2. 编程语言：Redis采用C语言编写，不依赖第三方类库，执行速度快
3. 线程模型：Redis使用单线程操作，避免了线程的切换和竞态消耗
4. 采用了非阻塞IO多路复用机制：多路I/O复用模型是利用 select、poll、epoll 可以同时监察多个流的 I/O 事件的能力，在空闲的时候，会把当前线程阻塞掉，当有一个或多个流有 I/O 事件时，就从阻塞态中唤醒，于是程序就会轮询一遍所有的流（epoll 是只轮询那些真正发出了事件的流），并且只依次顺序的处理就绪的流，这种做法就避免了大量的无用操作。这里“多路”指的是多个网络连接，“复用”指的是复用同一个线程。加上Redis自身的事件处理模型将epoll中的连接，读写，关闭都转换为了事件，不在I/O上浪费过多的时间
5. 由于是单线程，所以就存在一个顺序读写的问题，顺序读写比随机读写的速度快。
6. Redis的数据结构是经过专门的研究和设计的，所以操作起来简单且快。 最后，再说一点，Redis 是单进程和单线程的设计，并不是说它不能多进程多线程。比如备份时会 fork 一个新进程来操作；再比如基于 COW 原理的 RDB 操作就是多线程的。

Redis如何处理过期数据？Slave不能处理数据，那数据过期了怎么办？

1主2从的模式中，当master挂掉之后怎么办？

这种典型的模式就不上图了，master读写，slave1/2只读，当master挂掉之后，redis服务不可用，需要立马手动处理。两种处理方式，第一种是把master重新启动起来，不用改变现有的主从结构，缺点是什么呢，master重新启动并完成RDB/AOF的恢复是个耗时的过程，另外会造成slave1/2发生全量复制；第二种就是重新选举新的master，具体怎么做呢？选折其中一个slave，执行命令 slave no one来解除自己是从服务器的身份，使其称为一个master，注意的点是这个slave要改成读写模式；连到另一个slave，执行slave new master，让它去找master。整个过程是一个手动的过程，Redis Sentinel就是这样一个功能，自动完成切换，帅的一比。

> Redis分布式锁你真的会用吗？ https://www.xttblog.com/?p=4598

> Redis使用注意的点

1.由于是单线程模型，因此一次只运行一条命令
2.拒绝长（慢）命令：keys, flushall,flushdb, slow lua script, mutil/exec, operate big value(collection)

> Redis Sentinel和Redis Cluster的区别

1、sentinel

实现高可用，但是没有分区 监控，能持续监控Redis的主从实例是否正常工作； 通知，当被监控的Redis实例出问题时，能通过API通知系统管理员或其他程序； 自动故障恢复，如果主实例无法正常工作，Sentinel将启动故障恢复机制把一个从实例提升为主实例，其他的从实例将会被重新配置到新的主实例，且应用程序会得到一个更换新地址的通知。 Redis Sentinel是一个分布式系统，可以部署多个Sentinel实例来监控同一组Redis实例，它们通过Gossip协议来确定一个主实例宕机，通过Agreement协议来执行故障恢复和配置变更。

2、Redis Cluster特点如下：

- 所有的节点相互连接；
- 集群消息通信通过集群总线通信，，集群总线端口大小为客户端服务端口+10000，这个10000是固定值；
- 节点与节点之间通过二进制协议进行通信；
- 客户端和集群节点之间通信和通常一样，通过文本协议进行；
- 集群节点不会代理查询；

redis3.0以后推出了cluster，具有Sentinel的监控和自动Failover能力，同时提供一种官方的分区解决方案

Redis分布式锁/Redis的setnx命令如何设置key的失效时间（同时操作setnx和expire

Redis的setnx命令是当key不存在时设置key，但setnx不能同时完成expire设置失效时长，不能保证setnx和expire的原子性。我们可以使用set命令完成setnx和expire的操作，并且这种操作是原子操作。 下面是set命令的可选项：

```
set key value [EX seconds] [PX milliseconds] [NX|XX]
EX seconds：设置失效时长，单位秒
PX milliseconds：设置失效时长，单位毫秒
NX：key不存在时设置value，成功返回OK，失败返回(nil)
XX：key存在时设置value，成功返回OK，失败返回(nil)

案例：设置name=p7+，失效时长100s，不存在时设置
1.1.1.1:6379> set name p7+ ex 100 nx
```

> 假如Redis里面有1亿个key，其中有10w个key是以某个固定的已知的前缀开头的，如何将它们全部找出来？

可以使用keys [pattern]来列举出来，由于Redis是单线程的，在使用keys命令的时候会导致线程阻塞一段时间，我们也可以使用scan指令以无阻塞的方式取出来，但会有一定重复的概率，客户端去重就可以了。如果是主从模式的话，可以在从服务器执行keys命令，尽量不影响现有业务。

使用过Redis做异步队列么，你是怎么用的？

Redis如何实现延时队列？

RDB的原理是什么？

你给出两个词汇就可以了，fork和cow。fork是指redis通过创建子进程来进行RDB操作，cow指的是copy on write，子进程创建后，父子进程共享数据段，父进程继续提供读写服务，写脏的页面数据会逐渐和子进程分离开来。

Pipeline有什么好处，为什么要用pipeline？

可以将多次IO往返的时间缩减为一次，前提是pipeline执行的指令之间没有因果相关性。使用redis-benchmark进行压测的时候可以发现影响redis的QPS峰值的一个重要因素是pipeline批次指令的数目。

是否使用过Redis集群，集群的高可用怎么保证，集群的原理是什么？

Redis Sentinal 着眼于高可用，在master宕机时会自动将slave提升为master，继续提供服务。

Redis Cluster 着眼于扩展性，在单个redis内存不足时，使用Cluster进行分片存储。

Redis集群是如何通信的？

Redis集群采用gossip（流言）协议来通信，Gossip协议的主要职责就是信息交换。信息交换的载体就是节点彼此发送的Gossip消息，常用的Gossip消息可分为：ping消息、pong消息、meet消息、fail消息。集群中的每个节点都会单独开辟一个TCP通道，用于节点之间的彼此通信，通信端口是在基础端口的基础上加上1万，每个节点在固定周期内通过特定规则选择几个节点发送ping消息，接收到ping消息的节点用pong消息作为响应。

ping消息发送封装了自身节点和部分其他节点的状态数据，pong消息：当接收到ping、meet消息时，作为响应消息回复给发送方确认消息正常通信。pong消息内部封装了自身状态数据。节点也可以向集群内广播自身的pong消息来通知整个集群对自身状态进行更新，一段时间后整个集群达到了状态的一致性。

PS：定时任务默认每秒执行10次，每秒会随机选取5个节点找出最久没有通信的节点发送ping消息，用于保证Gossip信息交换的随机性

> Reids Key是如何寻址的？ 分布式寻址算法

hash 算法（大量缓存重建） 一致性 hash 算法（自动缓存迁移）+ 虚拟节点（自动负载均衡） redis cluster 的 hash slot 算法 hash 算法

来了一个 key，首先计算 hash 值，然后对节点数取模。然后打在不同的 master 节点上。一旦某一个 master 节点宕机，所有请求过来，都会基于最新的剩余 master 节点数去取模，尝试去取数据。这会导致大部分的请求过来，全部无法拿到有效的缓存，导致大量的流量涌入数据库。

一致性 hash 算法

一致性 hash 算法将整个 hash 值空间组织成一个虚拟的圆环，整个空间按顺时针方向组织，下一步将各个 master 节点（使用服务器的 ip 或主机名）进行 hash。这样就能确定每个节点在其哈希环上的位置。

来了一个 key，首先计算 hash 值，并确定此数据在环上的位置，从此位置沿环顺时针“行走”，遇到的第一个 master 节点就是 key 所在位置。

在一致性哈希算法中，如果一个节点挂了，受影响的数据仅仅是此节点到环空间前一个节点（沿着逆时针方向行走遇到的第一个节点）之间的数据，其它不受影响。增加一个节点也同理。

燃鹅，一致性哈希算法在节点太少时，容易因为节点分布不均匀而造成缓存热点的问题。为了解决这种热点问题，一致性 hash 算法引入了虚拟节点机制，即对每一个节点计算多个 hash，每个计算结果位置都放置一个虚拟节点。这样就实现了数据的均匀分布，负载均衡。

redis cluster 的 hash slot 算法

redis cluster 有固定的 16384 个 hash slot，对每个 key 计算 CRC16 值，然后对 16384 取模，可以获取 key 对应的 hash slot。

redis cluster 中每个 master 都会持有部分 slot，比如有 3 个 master，那么可能每个 master 持有 5000 多个 hash slot。hash slot 让 node 的增加和移除很简单，增加一个 master，就将其他 master 的 hash slot 移动部分过去，减少一个 master，就将它的 hash slot 移动到其他 master 上去。移动 hash slot 的成本是非常低的。客户端的 api，可以对指定的数据，让他们走同一个 hash slot，通过 hash tag 来实现。

如何解决DB和缓存一致性问题？

经典的使用场景是对于热点的数据读操作是从Redis中读取的，那么当数据发生写的操作该怎么办？因为写的操作要同时在数据库和缓存中进行，涉及到双写，那么就必然存在双写数据不一致的问题，如果业务的场景是要求强一致性，那就不要用缓存。那么我接下来就结合我的理解谈谈一些经典的使用场景下，如何尽可能的保证双写的一致性。

1. 读的时候先从Redis读取，如果Redis中没有，那么再去数据库中读，读完之后放回Redis，然后返回响应。写的时候先删除缓存，然后再去写入数据库，然后等待100毫秒再次删除缓存。这里有几点点需要说明： 为什么是先删除缓存而不是先更新数据？ 假设我们先更新数据，再删除缓存，那么存在数据更新成功，但是删除缓存失败的情况。先删除缓存，如果数据库写操作成功，下次读缓存时会从数据库获取新的数据并放入缓存；如果数据库写操作失败，那么再次读到的数据也是旧的数据，也保证了缓存的一致性。另外，有的缓存数据是经过计算的数据，不是单纯的某个字段的值，如果去更新的话会是一个复杂的过程，倒不如下次获取的时候去计算并放入缓存，这也是一个“懒”的思想。 为什么写入数据库后还要再次删除缓存?采用延时双删策略 我们在数据库写操作的时候是不能保证没有读的操作，特别是在高并发场景下，往往数据库写操作还没完成，就已经有读的操作完成并将修改前的数据放入缓存，这也就造成了缓存和数据库中的数据不一致的问题。那么解决这个问题有一下方法
2. 数据库写操作完成后再次删除缓存，这样出现不一致的时间就只会在数据库写操作和再次删除缓存这段时间内，如果业务能够容忍短时间的不一致，可以采用这个方法。
3. 如果业务对一致性要求较高，那么可以在第一次删除缓存后对后续的操作做串行处理，后续的所有操作都需要等待数据库写操作的完成。具体的代码思路可以是这样子，将请求放入JVM的一个Queue中，将请求积压在队列中，同步等待写操作

的完成。这个思路的实现有些复杂，还涉及到如果Queue中积累的请求过多该怎么处理，请求过多的话也就说明这是个热点key。

第二种方案：异步更新缓存(基于订阅binlog的同步机制) 技术整体思路： MySQL binlog增量订阅消费+消息队列+增量数据更新到redis 读Redis：热数据基本都在Redis 写MySQL:增删改都是操作MySQL 更新Redis数据：MySQ的数据操作binlog，来更新到Redis Redis更新 1）数据操作主要分为两大块： 一个是全量(将全部数据一次写入到redis) 一个是增量（实时更新）这里说的是增量,指的是mysql的update、insert、delate变更数据。 2）读取binlog后分析 ，利用消息队列,推送更新各台的redis缓存数据。 这样一旦MySQL中产生了新的写入、更新、删除等操作，就可以把binlog相关的消息推送至Redis，Redis再根据binlog中的记录，对Redis进行更新。 其实这种机制，很类似MySQL的主从备份机制，因为MySQL的主备也是通过binlog来实现的数据一致性。 这里可以结合使用canal(阿里的一款开源框架)，通过该框架可以对MySQL的binlog进行订阅，而canal正是模仿了mysql的slave数据库的备份请求，使得Redis的数据更新达到了相同的效果。 当然，这里的消息推送工具你也可以采用别的第三方：kafka、rabbitMQ等来实现推送更新Redis。 以上就是Redis和MySQL数据一致性详解。

> redis在什么情况下会变慢？

> 单线程的redis，如何知道要运行定时任务？

redis是单线程的，线程不但要处理定时任务，还要处理客户端请求，线程不能阻塞在定时任务或处理客户端请求上，那么，redis是如何知道何时该运行定时任务的呢？ Redis 的定时任务会记录在一个称为最小堆的数据结构中。这个堆中，最快要执行的任务排在堆的最上方。在每个循环周期，Redis 都会将最小堆里面已经到点的任务立即进行处理。处理完毕后，将最快要执行的任务还需要的时间记录下来，这个时间就是接下来处理客户端请求的最大时长，若达到了该时长，则暂时不处理客户端请求而去运行定时任务。

> Redis的五种数据类型分别是什么数据结构实现的？ Redis的字符串数据类型既可以存储字符串，又可以存储整数和浮点数，Redis在内部是怎么存储这些值的？ Redis的一部分命令只能对特定的数据类型执行操作，比如append只能对字符串执行，hset只能对hash执行，而另一部分命令却可以对所有数据类型执行，比如del, type, expire，不同的命令在执行时如何进行类型检查的？Redis在内部是否实现了一个类型系统？

Redis没有直接使用C语言的传统字符串表示，而是自己构建了一个名为简单动态字符串的抽象类型，并将SDS用作Redis的默认字符串表示。

```
struct sdshdr {
  // buf中已经占用空间的长度
  int len;
  // buf中剩余可用空间的长度
  int free;
  // 数据空间，值存在这里。
  char bu[];
}
```

转自：https://m.toutiao.com/is/e1JVgd7/

go小技巧

Go 箴言

- 不要通过共享内存进行通信，通过通信共享内存
- 并发不是并行
- 管道用于协调；互斥量（锁）用于同步
- 接口越大，抽象就越弱
- 利用好零值
- 空接口 `interface{}` 没有任何类型约束
- Gofmt 的风格不是人们最喜欢的，但 gofmt 是每个人的最爱
- 允许一点点重复比引入一点点依赖更好
- 系统调用必须始终使用构建标记进行保护
- 必须始终使用构建标记保护 Cgo
- Cgo 不是 Go
- 使用标准库的 `unsafe` 包，不能保证能如期运行
- 清晰比聪明更好
- 反射永远不清晰
- 错误是值
- 不要只检查错误，还要优雅地处理它们
- 设计架构，命名组件，（文档）记录细节
- 文档是供用户使用的
- 不要（在生产环境）使用 `panic()`

Author: Rob Pike
See more: https://go-proverbs.github.io/

Go 之禅

- 每个 package 实现单一的目的
- 显式处理错误
- 尽早返回，而不是使用深嵌套
- 让调用者处理并发（带来的问题）
- 在启动一个 goroutine 时，需要知道何时它会停止
- 避免 package 级别的状态
- 简单很重要
- 编写测试以锁定 package API 的行为
- 如果你觉得慢，先编写 benchmark 来证明
- 适度是一种美德
- 可维护性

Author: Dave Cheney
See more: https://the-zen-of-go.netlify.com/

代码

使用 `go fmt` 格式化

让团队一起使用官方的 Go 格式工具，不要重新发明轮子。
尝试减少代码复杂度。 这将帮助所有人使代码易于阅读。

多个 if 语句可以折叠成 switch

```
// NOT BAD
if foo() {
    // ...
} else if bar == baz {
    // ...
} else {
    // ...
}

// BETTER
switch {
case foo():
    // ...
case bar == baz:
    // ...
default:
    // ...
}
```

用 `chan struct{}` 来传递信号，`chan bool` 表达的不够清楚

当你在结构中看到 `chan bool` 的定义时，有时不容易理解如何使用该值，例如：

```
type Service struct {
    deleteCh chan bool // what does this bool mean?
}
```

但是我们可以将其改为明确的 `chan struct {}` 来使其更清楚：我们不在乎值（它始终是 `struct {}` ），我们关心可能发生的事件，例如：

```
type Service struct {
    deleteCh chan struct{} // ok, if event than delete something.
}
```

`30 * time.Second` 比 `time.Duration(30) * time.Second` 更好

你不需要将无类型的常量包装成类型，编译器会找出来。
另外最好将常量移到第一位：

```
// BAD
delay := time.Second * 60 * 24 * 60

// VERY BAD
delay := 60 * time.Second * 60 * 24

// GOOD
delay := 24 * 60 * 60 * time.Second
```

用 `time.Duration` 代替 `int64` + 变量名

```
// BAD
var delayMillis int64 = 15000

// GOOD
var delay time.Duration = 15 * time.Second
```

按类型分组 `const` 声明，按逻辑和/或类型分组 `var`

```
// BAD
const (
    foo = 1
    bar = 2
    message = "warn message"
)

// MOSTLY BAD
const foo = 1
const bar = 2
const message = "warn message"

// GOOD
const (
    foo = 1
    bar = 2
)

const message = "warn message"
```

这个模式也适用于 `var` 。

- 每个阻塞或者 IO 函数操作应该是可取消的或者至少是可超时的
- 为整型常量值实现 `Stringer` 接口
  - https://godoc.org/golang.org/x/tools/cmd/stringer
- 检查 `defer` 中的错误

```
 defer func() {
     err := ocp.Close()
     if err != nil {
         rerr = err
     }
 }()
```

- 不要在 `checkErr` 函数中使用 `panic()` 或 `os.Exit()`
- 仅仅在很特殊情况下才使用 panic, 你必须要去处理 error
- 不要给枚举使用别名，因为这打破了类型安全
  - https://play.golang.org/p/MGbeDwtXN3

```
 package main
 type Status = int
 type Format = int // remove `=` to have type safety

 const A Status = 1
 const B Format = 1
```

```
func main() {
    println(A == B)
}
```

- 如果你想省略返回参数，你最好表示出来
  - `_ = f()` 比 `f()` 更好
- 我们用 `a := []T{}` 来简单初始化 slice
- 用 range 循环来进行数组或 slice 的迭代
  - `for _, c := range a[3:7] {...}` 比 `for i := 3; i < 7; i++ {...}` 更好
- 多行字符串用反引号(`)
- 用 `_` 来跳过不用的参数

```
func f(a int, _ string) {}
```

- 如果你要比较时间戳，请使用 `time.Before` 或 `time.After` ，不要使用 `time.Sub` 来获得 duration (持续时间)，然后检查它的值。
- 带有上下文的函数第一个参数名为 `ctx` ，形如： `func foo(ctx Context, ...)`
- 几个相同类型的参数定义可以用简短的方式来进行

```
func f(a int, b int, s string, p string)
```

```
func f(a, b int, s, p string)
```

- 一个 slice 的零值是 nil
  - https://play.golang.org/p/pNT0d_Bunq

```
var s []int
fmt.Println(s, len(s), cap(s))
if s == nil {
fmt.Println("nil!")
}
// Output:
// [] 0 0
// nil!
```

    - https://play.golang.org/p/meTInNyxtk

```
var a []string
b := []string{}

fmt.Println(reflect.DeepEqual(a, []string{}))
fmt.Println(reflect.DeepEqual(b, []string{}))
// Output:
// false
// true
```

- 不要将枚举类型与 `<`, `>`, `<=` 和 `>=` 进行比较
  - 使用确定的值，不要像下面这样做:

```
value := reflect.ValueOf(object)
kind := value.Kind()
if kind >= reflect.Chan && kind <= reflect.Slice {
    // ...
}
```

- 用 `%+v` 来打印数据的比较全的信息
- 注意空结构 `struct{}`, 看 issue: https://github.com/golang/go/issues/23440
  - more: https://play.golang.org/p/9C0puRUstrP

```
func f1() {
    var a, b struct{}
    print(&a, "\n", &b, "\n") // Prints same address
    fmt.Println(&a == &b)     // Comparison returns false
}

func f2() {
    var a, b struct{}
    fmt.Printf("%p\n%p\n", &a, &b) // Again, same address
    fmt.Println(&a == &b)          // ...but the comparison returns true
}
```

- 包装错误： http://github.com/pkg/errors
  - 例如: `errors.Wrap(err, "additional message to a given error")`
- 在 Go 里面要小心使用 `range` :
  - `for i := range a` and `for i, v := range &a` ，都不是 `a` 的副本
  - 但是 `for i, v := range a` 里面的就是 `a` 的副本
  - 更多: https://play.golang.org/p/4b181zkB1O
- 从 map 读取一个不存在的 key 将不会 panic
  - `value := map["no_key"]` 将得到一个 0 值
  - `value, ok := map["no_key"]` 更好
- 不要使用原始参数进行文件操作
  - 而不是一个八进制参数 `os.MkdirAll(root, 0700)`
  - 使用此类型的预定义常量 `os.FileMode`
- 不要忘记为 `iota` 指定一种类型
  - https://play.golang.org/p/mZZdMaI92cI

```
    const (
        _ = iota
        testvar         // testvar 将是 int 类型
    )
```

vs

```
    type myType int
    const (
        _ myType = iota
        testvar         // testvar 将是 myType 类型
    )
```

不要在你不拥有的结构上使用 `encoding/gob`

在某些时候，结构可能会改变，而你可能会错过这一点。因此，这可能会导致很难找到 bug。

不要依赖于计算顺序，特别是在 return 语句中。

```
  // BAD
  return res, json.Unmarshal(b, &res)

  // GOOD
  err := json.Unmarshal(b, &res)
  return res, err
```

防止结构体字段用纯值方式初始化，添加 `_ struct {}` 字段：

```
type Point struct {
  X, Y float64
  _    struct{} // to prevent unkeyed literals
}
```

对于 `Point {X: 1, Y: 1}` 都可以，但是对于 `Point {1,1}` 则会出现编译错误：

```
./file.go:1:11: too few values in Point literal
```

当在你所有的结构体中添加了 `_ struct{}` 后，使用 `go vet` 命令进行检查，（原来声明的方式）就会提示没有足够的参数。

为了防止结构比较，添加 `func` 类型的空字段

```
  type Point struct {
    _ [0]func() // unexported, zero-width non-comparable field
    X, Y float64
  }
```

`http.HandlerFunc` 比 `http.Handler` 更好

用 `http.HandlerFunc` 你仅需要一个 func，`http.Handler` 需要一个类型。

移动 `defer` 到顶部

这可以提高代码可读性并明确函数结束时调用了什么。

JavaScript 解析整数为浮点数并且你的 int64 可能溢出

用 `json:"id,string"` 代替

```
type Request struct {
  ID int64 `json:"id,string"`
}
```

并发

- 以线程安全的方式创建单例（只创建一次）的最好选择是 `sync.Once`
  - 不要用 flags, mutexes, channels or atomics
- 永远不要使用 `select{}`，省略通道， 等待信号
- 不要关闭一个发送（写入）管道，应该由创建者关闭
  - 往一个关闭的 channel 写数据会引起 panic
- `math/rand` 中的 `func NewSource(seed int64) Source` 不是并发安全的，默认的 `lockedSource` 是并发安全的, see issue: https://github.com/golang/go/issues/3611
  - 更多: https://golang.org/pkg/math/rand/
- 当你需要一个自定义类型的 atomic 值时，可以使用 atomic.Value

性能

- 不要省略 `defer`
  - 在大多数情况下 200ns 加速可以忽略不计
- 总是关闭 http body `defer r.Body.Close()`
  - 除非你需要泄露 goroutine
- 过滤但不分配新内存

```
  b := a[:0]
  for _, x := range a {
      if f(x) {
        b = append(b, x)
      }
  }
```

为了帮助编译器删除绑定检查，请参见此模式 `_ = b [7]`

- `time.Time` 有指针字段 `time.Location` 并且这对 go GC 不好
  - 只有使用了大量的 `time.Time` 才（对性能）有意义，否则用 timestamp 代替
- `regexp.MustCompile` 比 `regexp.Compile` 更好
  - 在大多数情况下，你的正则表达式是不可变的，所以你最好在 `func init` 中初始化它
- 请勿在你的热点代码中过度使用 `fmt.Sprintf`. 由于维护接口的缓冲池和动态调度，它是很昂贵的。
  - 如果你正在使用 `fmt.Sprintf("%s%s", var1, var2)`，考虑使用简单的字符串连接。
  - 如果你正在使用 `fmt.Sprintf("%x", var)`，考虑使用 `hex.EncodeToString` or `strconv.FormatInt(var, 16)`
- 如果你不需要用它，可以考虑丢弃它，例如 `io.Copy(ioutil.Discard, resp.Body)`
  - HTTP 客户端的传输不会重用连接，直到body被读完和关闭。

```
res, _ := client.Do(req)
io.Copy(ioutil.Discard, res.Body)
defer res.Body.Close()
```

- 不要在循环中使用 defer，否则会导致内存泄露
  - 因为这些 defer 会不断地填满你的栈（内存）
- 不要忘记停止 ticker，除非你需要泄露 channel

```
ticker := time.NewTicker(1 * time.Second)
defer ticker.Stop()
```

- 用自定义的 marshaler 去加速 marshaler 过程
  - 但是在使用它之前要进行定制！例如：https://play.golang.org/p/SEm9Hvsi0r

```
func (entry Entry) MarshalJSON() ([]byte, error) {
    buffer := bytes.NewBufferString("{")
    first := true
    for key, value := range entry {
        jsonValue, err := json.Marshal(value)
        if err != nil {
            return nil, err
        }
        if !first {
            buffer.WriteString(",")
        }
        first = false
        buffer.WriteString(key + ":" + string(jsonValue))
    }
    buffer.WriteString("}")
    return buffer.Bytes(), nil
}
```

- `sync.Map` 不是万能的，没有很强的理由就不要使用它。
  - 了解更多: https://github.com/golang/go/blob/master/src/sync/map.go#L12
- 在 `sync.Pool` 中分配内存存储非指针数据
  - 了解更多: https://github.com/dominikh/go-tools/blob/master/cmd/staticcheck/docs/checks/SA6002
- 为了隐藏逃生分析的指针，你可以小心使用这个函数：：
  - 来源: https://go-review.googlesource.com/c/go/+/86976

```
// noescape hides a pointer from escape analysis.  noescape is
// the identity function but escape analysis doesn't think the
// output depends on the input. noescape is inlined and currently
// compiles down to zero instructions.
//go:nosplit
func noescape(p unsafe.Pointer) unsafe.Pointer {
    x := uintptr(p)
    return unsafe.Pointer(x ^ 0)
}
```

- 对于最快的原子交换，你可以使用这个 `m := (*map[int]int)(atomic.LoadPointer(&ptr))`
- 如果执行许多顺序读取或写入操作，请使用缓冲 I/O
  - 减少系统调用次数
- 有 2 种方法清空一个 map：
  - 重用 map 内存 （但是也要注意 m 的回收）

```
for k := range m {
  delete(m, k)
}
```

- 分配新的

```
m = make(map[int]int)
```

模块

- 如果你想在 CI 中测试 `go.mod` （和 `go.sum` ）是否是最新 https://blog.urth.org/2019/08/13/testing-go-mod-tidiness-in-ci/

构建

- 用这个命令 `go build -ldflags="-s -w" ...` 去掉你的二进制文件
- 拆分构建不同版本的简单方法
  - 用 `// +build integration` 并且运行他们 `go test -v --tags integration .`
- 最小的 Go Docker 镜像
  - https://twitter.com/bbrodriges/status/873414658178396160
  - `CGO_ENABLED=0 go build -ldflags="-s -w" app.go && tar C app | docker import - myimage:latest`
- run go format on CI and compare diff
  - 这将确保一切都是生成的和承诺的
- 用最新的 Go 运行 Travis-CI，用 `travis 1`
  - 了解更多：https://github.com/travis-ci/travis-build/blob/master/public/version-aliases/go.json
- 检查代码格式是否有错误 `diff -u <(echo -n) <(gofmt -d .)`

测试

- 测试名称 `package_test` 比 `package` 要好
- `go test -short` 允许减少要运行的测试数

```
func TestSomething(t *testing.T) {
  if testing.Short() {
    t.Skip("skipping test in short mode.")
  }
}
```

- 根据系统架构跳过测试

```
if runtime.GOARM == "arm" {
   t.Skip("this doesn't work under ARM")
}
```

- 用 `testing.AllocsPerRun` 跟踪你的内存分配
  - https://godoc.org/testing#AllocsPerRun
- 多次运行你的基准测试可以避免噪音。
  - `go test -test.bench=. -count=20`

工具

- 快速替换 `gofmt -w -l -r "panic(err) -> log.Error(err)" .`
- `go list` 允许找到所有直接和传递的依赖关系
  - `go list -f '{{ .Imports }}' package`
  - `go list -f '{{ .Deps }}' package`
- 对于快速基准比较，我们有一个 `benchstat` 工具。
  - https://godoc.org/golang.org/x/perf/cmd/benchstat
- go-critic linter 从这个文件中强制执行几条建议
- `go mod why -m <module>` 告诉我们为什么特定的模块在 `go.mod` 文件中。
- `GOGC=off go build ...` 应该会加快构建速度 source
- 内存分析器每 512KB 记录一次分配。你能通过 `GODEBUG` 环境变量增加比例，来查看你的文件的更多详细信息。
  - 来源：https://twitter.com/bboreham/status/1105036740253937664
- `go mod why -m <module>` 告诉我们为什么特定的模块是在 `go.mod` 文件中。

其他

- dump goroutines https://stackoverflow.com/a/27398062/433041

```
go func() {
   sigs := make(chan os.Signal, 1)
   signal.Notify(sigs, syscall.SIGQUIT)
   buf := make([]byte, 1<<20)
   for {
      <-sigs
      stacklen := runtime.Stack(buf, true)
      log.Printf("=== received SIGQUIT ===\n*** goroutine dump...\n%s\n*** end\n"  , buf[:stacklen])
   }
}()
```

- 在编译期检查接口的实现

```
var _ io.Reader = (*MyFastReader)(nil)
```

- len(nil) = 0
  - https://golang.org/pkg/builtin/#len
- 匿名结构很酷

```
var hits struct {
    sync.Mutex
    n int
}
hits.Lock()
hits.n++
hits.Unlock()
```

- `httputil.DumpRequest` 是非常有用的东西，不要自己创建
  - https://godoc.org/net/http/httputil#DumpRequest
- 获得调用堆栈，我们可以使用 `runtime.Caller`
  - https://golang.org/pkg/runtime/#Caller
- 要 marshal 任意的 JSON， 你可以 marshal 为 `map[string]interface{}{}`
- 配置你的 `CDPATH` 以便你能在任何目录执行 `cd github.com/golang/go`
  - 添加这一行代码到 `bashrc` (或者其他类似的) `export CDPATH=$CDPATH:$GOPATH/src`
- 从一个 slice 生成简单的随机元素
  - `[]string{"one", "two", "three"}[rand.Intn(3)]`

转自：https://github.com/cristaloleg/go-advice/blob/master/README_ZH.md

基础语法50问

1.使用值为 nil 的 slice、map会发生啥

允许对值为 nil 的 slice 添加元素，但对值为 nil 的 map 添加元素，则会造成运行时 panic。

```
// map 错误示例
func main() {
    var m map[string]int
    m["one"] = 1  // error: panic: assignment to entry in nil map
    // m := make(map[string]int)// map 的正确声明，分配了实际的内存
}

// slice 正确示例
func main() {
 var s []int
 s = append(s, 1)
}
```

2.访问 map 中的 key，需要注意啥

当访问 map 中不存在的 key 时，Go 则会返回元素对应数据类型的零值，比如 nil、’’ 、false 和 0，取值操作总有值返回，故不能通过取出来的值，来判断 key 是不是在 map 中。

检查 key 是否存在可以用 map 直接访问，检查返回的第二个参数即可。

```
// 错误的 key 检测方式
func main() {
 x := map[string]string{"one": "2", "two": "", "three": "3"}
 if v := x["two"]; v == "" {
  fmt.Println("key two is no entry") // 键 two 存不存在都会返回的空字符串
 }
}

// 正确示例
func main() {
 x := map[string]string{"one": "2", "two": "", "three": "3"}
 if _, ok := x["two"]; !ok {
  fmt.Println("key two is no entry")
 }
}
```

3.string 类型的值可以修改吗

不能，尝试使用索引遍历字符串，来更新字符串中的个别字符，是不允许的。

string 类型的值是只读的二进制 byte slice，如果真要修改字符串中的字符，将 string 转为 []byte 修改后，再转为 string 即可。

```
// 修改字符串的错误示例
func main() {
 x := "text"
 x[0] = "T"  // error: cannot assign to x[0]
 fmt.Println(x)
```

```
}


// 修改示例
func main() {
 x := "text"
 xBytes := []byte(x)
 xBytes[0] = 'T' // 注意此时的 T 是 rune 类型
 x = string(xBytes)
 fmt.Println(x) // Text
}
```

4.switch 中如何强制执行下一个 case 代码块

switch 语句中的 case 代码块会默认带上 break，但可以使用 fallthrough 来强制执行下一个 case 代码块。

```
func main() {
 isSpace := func(char byte) bool {
  switch char {
  case ' ': // 空格符会直接 break，返回 false // 和其他语言不一样
  // fallthrough // 返回 true
  case '\t':
   return true
  }
  return false
 }
 fmt.Println(isSpace('\t')) // true
 fmt.Println(isSpace(' ')) // false
}
```

5.你是如何关闭 HTTP 的响应体的

直接在处理 HTTP 响应错误的代码块中，直接关闭非 nil 的响应体；手动调用 defer 来关闭响应体。

```
// 正确示例
func main() {
 resp, err := http.Get("http://www.baidu.com")

    // 关闭 resp.Body 的正确姿势
    if resp != nil {
  defer resp.Body.Close()
 }

 checkError(err)
 defer resp.Body.Close()

 body, err := ioutil.ReadAll(resp.Body)
 checkError(err)

 fmt.Println(string(body))
}
```

6.你是否主动关闭过http连接，为啥要这样做

有关闭，不关闭会程序可能会消耗完 socket 描述符。有如下2种关闭方式：

- 直接设置请求变量的 Close 字段值为 true，每次请求结束后就会主动关闭连接。设置 Header 请求头部选项 Connection: close，然后服务器返回的响应头部也会有这个选项，此时 HTTP 标准库会主动断开连接

```
// 主动关闭连接
func main() {
 req, err := http.NewRequest("GET", "http://golang.org", nil)
 checkError(err)

 req.Close = true
 //req.Header.Add("Connection", "close") // 等效的关闭方式

 resp, err := http.DefaultClient.Do(req)
 if resp != nil {
  defer resp.Body.Close()
 }
 checkError(err)

 body, err := ioutil.ReadAll(resp.Body)
 checkError(err)

 fmt.Println(string(body))
}
```

你可以创建一个自定义配置的 HTTP transport 客户端，用来取消 HTTP 全局的复用连接。

```
func main() {
 tr := http.Transport{DisableKeepAlives: true}
 client := http.Client{Transport: &tr}

 resp, err := client.Get("https://golang.google.cn/")
 if resp != nil {
  defer resp.Body.Close()
 }
 checkError(err)

 fmt.Println(resp.StatusCode) // 200

 body, err := ioutil.ReadAll(resp.Body)
 checkError(err)

 fmt.Println(len(string(body)))
}
```

7.解析 JSON 数据时，默认将数值当做哪种类型

在 encode/decode JSON 数据时，Go 默认会将数值当做 float64 处理。

```
func main() {
    var data = []byte(`{"status": 200}`)
    var result map[string]interface{}

    if err := json.Unmarshal(data, &result); err != nil {
    log.Fatalln(err)
}
```

解析出来的 200 是 float 类型。

8.如何从 panic 中恢复

在一个 defer 延迟执行的函数中调用 recover ，它便能捕捉/中断 panic。

```
// 错误的 recover 调用示例
func main() {
 recover() // 什么都不会捕捉
 panic("not good") // 发生 panic，主程序退出
 recover() // 不会被执行
 println("ok")
}

// 正确的 recover 调用示例
func main() {
 defer func() {
  fmt.Println("recovered: ", recover())
 }()
 panic("not good")
}
```

9.简短声明的变量需要注意啥

- 简短声明的变量只能在函数内部使用
- struct 的变量字段不能使用 := 来赋值
- 不能用简短声明方式来单独为一个变量重复声明， := 左侧至少有一个新变量，才允许多变量的重复声明

10.range 迭代 map是有序的吗

无序的。Go 的运行时是有意打乱迭代顺序的，所以你得到的迭代结果可能不一致。但也并不总会打乱，得到连续相同的 5 个迭代结果也是可能的。

11.recover的执行时机

无，recover 必须在 defer 函数中运行。recover 捕获的是祖父级调用时的异常，直接调用时无效。

```
func main() {
    recover()
    panic(1)
}
```

直接 defer 调用也是无效。

```
func main() {
    defer recover()
    panic(1)
}
```

defer 调用时多层嵌套依然无效。

```
func main() {
    defer func() {
        func() { recover() }()
```

```
    }()
    panic(1)
}
```

必须在 defer 函数中直接调用才有效。

```
func main() {
    defer func() {
        recover()
    }()
    panic(1)
}
```

12.闭包错误引用同一个变量问题怎么处理

在每轮迭代中生成一个局部变量 i 。如果没有 i := i 这行，将会打印同一个变量。

```
func main() {
    for i := 0; i < 5; i++ {
        i := i
        defer func() {
            println(i)
        }()
    }
}
```

或者是通过函数参数传入 i 。

```
func main() {
    for i := 0; i < 5; i++ {
        defer func(i int) {
            println(i)
        }(i)
    }
}
```

13.在循环内部执行defer语句会发生啥

defer 在函数退出时才能执行，在 for 执行 defer 会导致资源延迟释放。

```
func main() {
    for i := 0; i < 5; i++ {
        func() {
            f, err := os.Open("/path/to/file")
            if err != nil {
                log.Fatal(err)
            }
            defer f.Close()
        }()
    }
}
```

func 是一个局部函数，在局部函数里面执行 defer 将不会有问题。

14.说出一个避免Goroutine泄露的措施

可以通过 context 包来避免内存泄漏。

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

    ch := func(ctx context.Context) <-chan int {
        ch := make(chan int)
        go func() {
            for i := 0; ; i++ {
                select {
                case <- ctx.Done():
                    return
                case ch <- i:
                }
            }
        } ()
        return ch
    }(ctx)

    for v := range ch {
        fmt.Println(v)
        if v == 5 {
            cancel()
            break
        }
    }
}
```

下面的 for 循环停止取数据时，就用 cancel 函数，让另一个协程停止写数据。如果下面 for 已停止读取数据，上面 for 循环还在写入，就会造成内存泄漏。

15.如何跳出for select 循环

通常在for循环中，使用break可以跳出循环，但是注意在go语言中，for select配合时，break 并不能跳出循环。

```
func testSelectFor2(chExit chan bool){
 EXIT:
    for  {
        select {
        case v, ok := <-chExit:
            if !ok {
                fmt.Println("close channel 2", v)
                break EXIT//goto EXIT2
            }

            fmt.Println("ch2 val =", v)
        }
    }

    //EXIT2:
    fmt.Println("exit testSelectFor2")
}
```

16.如何在切片中查找

go中使用 sort.searchXXX 方法，在排序好的切片中查找指定的方法，但是其返回是对应的查找元素不存在时，待插入的位置下标(元素插入在返回下标前)。

可以通过封装如下函数，达到目的。

```
func IsExist(s []string, t string) (int, bool) {
    iIndex := sort.SearchStrings(s, t)
    bExist := iIndex!=len(s) && s[iIndex]==t

    return iIndex, bExist
}
```

17.如何初始化带嵌套结构的结构体

go 的哲学是组合优于继承，使用 struct 嵌套即可完成组合，内嵌的结构体属性就像外层结构的属性即可，可以直接调用。

注意初始化外层结构体时，必须指定内嵌结构体名称的结构体初始化，如下看到 s1方式报错，s2 方式正确。

```
type stPeople struct {
    Gender bool
    Name string
}

type stStudent struct {
    stPeople
    Class int
}

//尝试4 嵌套结构的初始化表达式
//var s1 = stStudent{false, "JimWen", 3}
var s2 = stStudent{stPeople{false, "JimWen"}, 3}
fmt.Println(s2.Gender, s2.Name, s2.Class)
```

18.切片和数组的区别

数组是具有固定长度，且拥有零个或者多个，相同数据类型元素的序列。数组的长度是数组类型的一部分，所以[3]int 和 [4]int 是两种不同的数组类型。数组需要指定大小，不指定也会根据初始化的自动推算出大小，不可改变；数组是值传递。数组是内置类型，是一组同类型数据的集合，它是值类型，通过从0开始的下标索引访问元素值。在初始化后长度是固定的，无法修改其长度。

当作为方法的参数传入时将复制一份数组而不是引用同一指针。数组的长度也是其类型的一部分，通过内置函数len(array)获取其长度。数组定义：

```
var array [10]int

var array =[5]int{1,2,3,4,5}
```

切片表示一个拥有相同类型元素的可变长度的序列。切片是一种轻量级的数据结构，它有三个属性：指针、长度和容量。切片不需要指定大小；切片是地址传递；切片可以通过数组来初始化，也可以通过内置函数make()初始化 。初始化时len=cap,在追加元素时如果容量cap不足时将按len的2倍扩容。切片定义：

```
var slice []type = make([]type, len)
```

19.new和make的区别

new 的作用是初始化一个指向类型的指针 (*T) 。new 函数是内建函数，函数定义：func new(Type) *Type。使用 new 函数来分配空间。传递给 new 函数的是一个类型，不是一个值。返回值是指向这个新分配的零值的指针。

make 的作用是为 slice，map 或 chan 初始化并返回引用 (T)。make 函数是内建函数，函数定义：func make(Type, size IntegerType) Type；第一个参数是一个类型，第二个参数是长度；返回值是一个类型。

make(T, args) 函数的目的与 new(T) 不同。它仅仅用于创建 Slice, Map 和 Channel，并且返回类型是 T（不是T*）的一个初始化的（不是零值）的实例。

20.Printf()、Sprintf()、Fprintf()函数的区别用法是什么

都是把格式好的字符串输出，只是输出的目标不一样。

Printf()，是把格式字符串输出到标准输出（一般是屏幕，可以重定向）。Printf() 是和标准输出文件 (stdout) 关联的，Fprintf 则没有这个限制。
Sprintf()，是把格式字符串输出到指定字符串中，所以参数比printf多一个char*。那就是目标字符串地址。

Fprintf()，是把格式字符串输出到指定文件设备中，所以参数比 printf 多一个文件指针 FILE*。主要用于文件操作。Fprintf()是格式化输出到一个stream，通常是到文件。

21.说说go语言中的for循环

for 循环支持 continue 和 break 来控制循环，但是它提供了一个更高级的break，可以选择中断哪一个循环 for 循环不支持以逗号为间隔的多个赋值语句，必须使用平行赋值的方式来初始化多个变量。

22.Array 类型的值作为函数参数

在 C/C++ 中，数组（名）是指针。将数组作为参数传进函数时，相当于传递了数组内存地址的引用，在函数内部会改变该数组的值。

在 Go 中，数组是值。作为参数传进函数时，传递的是数组的原始值拷贝，此时在函数内部是无法更新该数组的。

```
// 数组使用值拷贝传参
func main() {
 x := [3]int{1,2,3}

 func(arr [3]int) {
  arr[0] = 7
  fmt.Println(arr) // [7 2 3]
 }(x)
 fmt.Println(x)    // [1 2 3] // 并不是你以为的 [7 2 3]
}
```

想改变数组，直接传递指向这个数组的指针类型。

```
// 传址会修改原数据
func main() {
 x := [3]int{1,2,3}

 func(arr *[3]int) {
  (*arr)[0] = 7
  fmt.Println(arr) // &[7 2 3]
 }(&x)
 fmt.Println(x) // [7 2 3]
}
```

直接使用 slice：即使函数内部得到的是 slice 的值拷贝，但依旧会更新 slice 的原始数据（底层 array）

```
// 错误示例
func main() {
```

```
 x := []string{"a", "b", "c"}
 for v := range x {
  fmt.Println(v) // 1 2 3
 }
}


// 正确示例
func main() {
 x := []string{"a", "b", "c"}
 for _, v := range x { // 使用 _ 丢弃索引
  fmt.Println(v)
 }
}
```

说。go语言中的for循

23.说说go语言中的switch语句

单个 case 中，可以出现多个结果选项。只有在 case 中明确添加 fallthrough关键字，才会继续执行紧跟的下一个 case。

24.说说go语言中有没有隐藏的this指针

方法施加的对象显式传递，没有被隐藏起来。

golang 的面向对象表达更直观，对于面向过程只是换了一种语法形式来表达方法施加的对象不需要非得是指针，也不用非得叫this。

25.go语言中的引用类型包含哪些

数组切片、字典(map)、通道（channel）、接口（interface）。

26.go语言中指针运算有哪些

可以通过“&”取指针的地址；可以通过“*”取指针指向的数据。

26.说说go语言的main函数
main 函数不能带参数；main 函数不能定义返回值。main 函数所在的包必须为 main 包；main 函数中可以使用 flag 包来获取和解析命令行参数。

27.go语言触发异常的场景有哪些

- 空指针解析
- 下标越界
- 除数为0
- 调用 panic 函数

28.说说go语言的beego框架

- beego 是一个 golang 实现的轻量级HTTP框架
- beego 可以通过注释路由、正则路由等多种方式完成 url 路由注入

- 可以使用 bee new 工具生成空工程，然后使用 bee run 命令自动热编译

29.说说go语言的goconvey框架

- goconvey 是一个支持 golang 的单元测试框架
- goconvey 能够自动监控文件修改并启动测试，并可以将测试结果实时输出到web界面
- goconvey 提供了丰富的断言简化测试用例的编写

30.GoStub的作用是什么

- GoStub 可以对全局变量打桩
- GoStub 可以对函数打桩
- GoStub 不可以对类的成员方法打桩
- GoStub 可以打动态桩，比如对一个函数打桩后，多次调用该函数会有不同的行为

31.go语言编程的好处是什么

- 编译和运行都很快。
- 在语言层级支持并行操作。
- 有垃圾处理器。
- 内置字符串和 maps。
- 函数是 go 语言的最基本编程单位。

32.说说go语言的select机制

- select 机制用来处理异步 IO 问题
- select 机制最大的一条限制就是每个 case 语句里必须是一个 IO 操作
- golang 在语言级别支持 select 关键字

33.解释一下go语言中的静态类型声明

静态类型声明是告诉编译器不需要太多的关注这个变量的细节。

静态变量的声明，只是针对于编译的时候, 在连接程序的时候，编译器还要对这个变量进行实际的声明。

34.go的接口是什么

- 在 go 语言中，interface 也就是接口，被用来指定一个对象。接口具有下面的要素:

- 一系列的方法
- 具体应用中并用来表示某个数据类型
- 在 go 中使用 interface 来实现多态

35.Go语言里面的类型断言是怎么回事

类型断言是用来从一个接口里面读取数值给一个具体的类型变量。类型转换是指转换两个不相同的数据类型。

36.go语言中局部变量和全局变量的缺省值是什么

全局变量的缺省值是与这个类型相关的零值。

37.go语言编程的好处是什么

- 编译和运行都很快。
- 在语言层级支持并行操作。
- 有垃圾处理器。
- 内置字符串和 maps。
- 函数是 go 语言的最基本编程单位。

38.解释一下go语言中的静态类型声明

静态类型声明是告诉编译器不需要太多的关注这个变量的细节。
静态变量的声明，只是针对于编译的时候，在连接程序的时候，编译器还要对这个变量进行实际的声明。

39.模块化编程是怎么回事

模块化编程是指把一个大的程序分解成几个小的程序。这么做的目的是为了减少程序的复杂度，易于维护，并且达到最高的效率。

码字不易，请不吝点赞，随手关注，更多精彩，自动送达。

40.Golang的方法有什么特别之处

函数的定义声明没有接收者。
方法的声明和函数类似，他们的区别是：方法在定义的时候，会在func和方法名之间增加一个参数，这个参数就是接收者，这样我们定义的这个方法就和接收者绑定在了一起，称之为这个接收者的方法。
Go语言里有两种类型的接收者：值接收者和指针接收者。使用值类型接收者定义的方法，在调用的时候，使用的其实是值接收者的一个副本，所以对该值的任何操作，不会影响原来的类型变量。——-相当于形式参数。

如果我们使用一个指针作为接收者，那么就会其作用了，因为指针接收者传递的是一个指向原值指针的副本，指针的副本，指向的还是原来类型的值，所以修改时，同时也会影响原来类型变量的值。

41.Golang可变参数

函数方法的参数，可以是任意多个，这种我们称之为可以变参数，比如我们常用的fmt.Println()这类函数，可以接收一个可变的参数。可以变参数，可以是任意多个。我们自己也可以定义可以变参数，可变参数的定义，在类型前加上省略号...即可。

```
func main() {
 print("1","2","3")
}


func print (a ...interface{}){
 for _,v:=range a{
  fmt.Print(v)
 }
 fmt.Println()
}
```

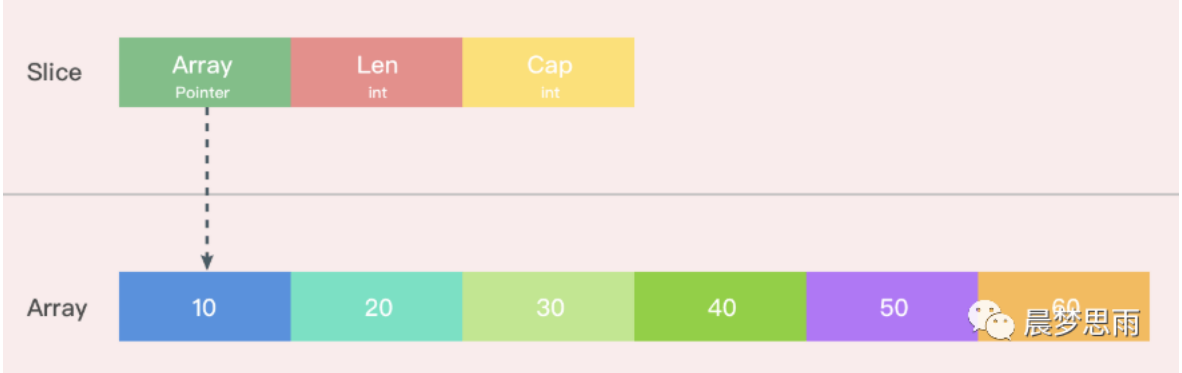
例子中我们自己定义了一个接受可变参数的函数，效果和fmt.Println()一样。可变参数本质上是一个数组，所以我们向使用数组一样使用它，比如例子中的 for range 循环。

42.Golang Slice的底层实现

切片是基于数组实现的，它的底层是数组，它自己本身非常小，可以理解为对底层数组的抽象。因为基于数组实现，所以它的底层的内存是连续分配的，效率非常高，还可以通过索引获得数据，可以迭代以及垃圾回收优化。
切片本身并不是动态数组或者数组指针。它内部实现的数据结构通过指针引用底层数组，设定相关属性将数据读写操作限定在指定的区域内。切片本身是一个只读对象，其工作机制类似数组指针的一种封装。
切片对象非常小，是因为它是只有3个字段的数据结构：

- 指向底层数组的指针
- 切片的长度
- 切片的容量

这3个字段，就是Go语言操作底层数组的元数据。



43.Golang Slice的扩容机制，有什么注意点

Go 中切片扩容的策略是这样的：

首先判断，如果新申请容量大于 2 倍的旧容量，最终容量就是新申请的容量。否则判断，如果旧切片的长度小于 1024，则最终容量就是旧容量的两倍。

否则判断，如果旧切片长度大于等于 1024，则最终容量从旧容量开始循环增加原来的 1/4，直到最终容量大于等于新申请的容量。如果最终容量计算值溢出，则最终容量就是新申请容量。

情况一：原数组还有容量可以扩容（实际容量没有填充完），这种情况下，扩容以后的数组还是指向原来的数组，对一个切片的操作可能影响多个指针指向相同地址的Slice。

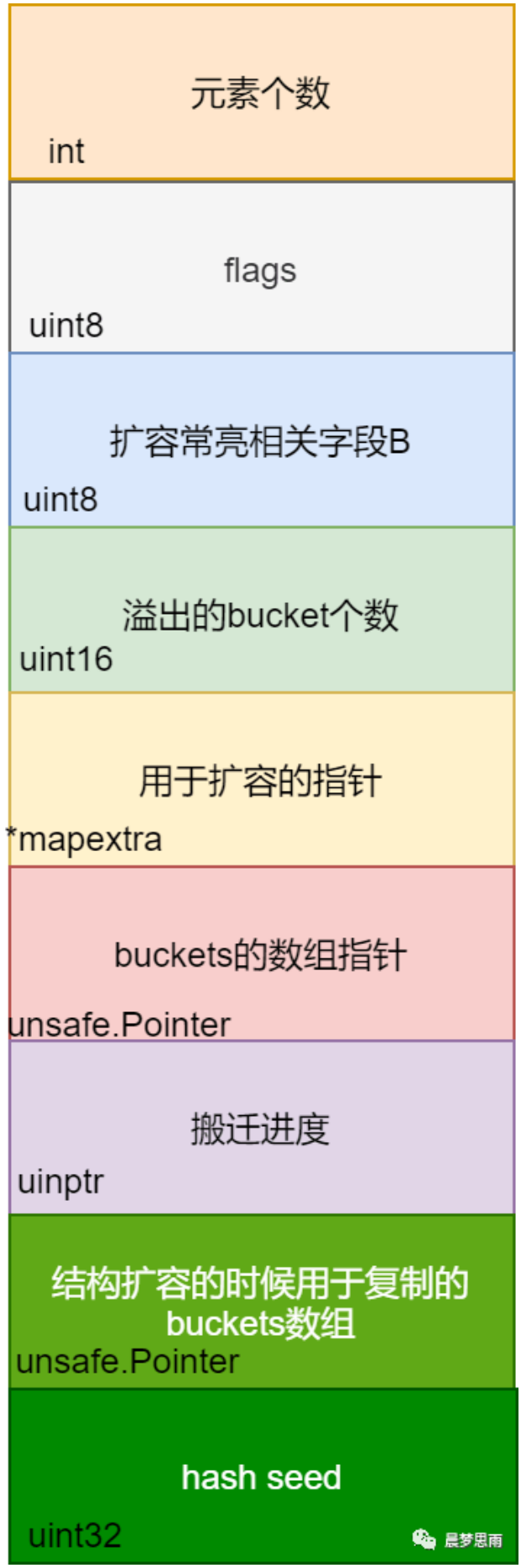
情况二：原来数组的容量已经达到了最大值，再想扩容， Go 默认会先开一片内存区域，把原来的值拷贝过来，然后再执行 append() 操作。这种情况丝毫不影响原数组。

要复制一个Slice，最好使用Copy函数。

44.Golang Map底层实现

Golang 中 map 的底层实现是一个散列表，因此实现 map 的过程实际上就是实现散表的过程。
在这个散列表中，主要出现的结构体有两个，一个叫hmap(a header for a go map)，一个叫bmap(a bucket for a Go map，通常叫其bucket)。

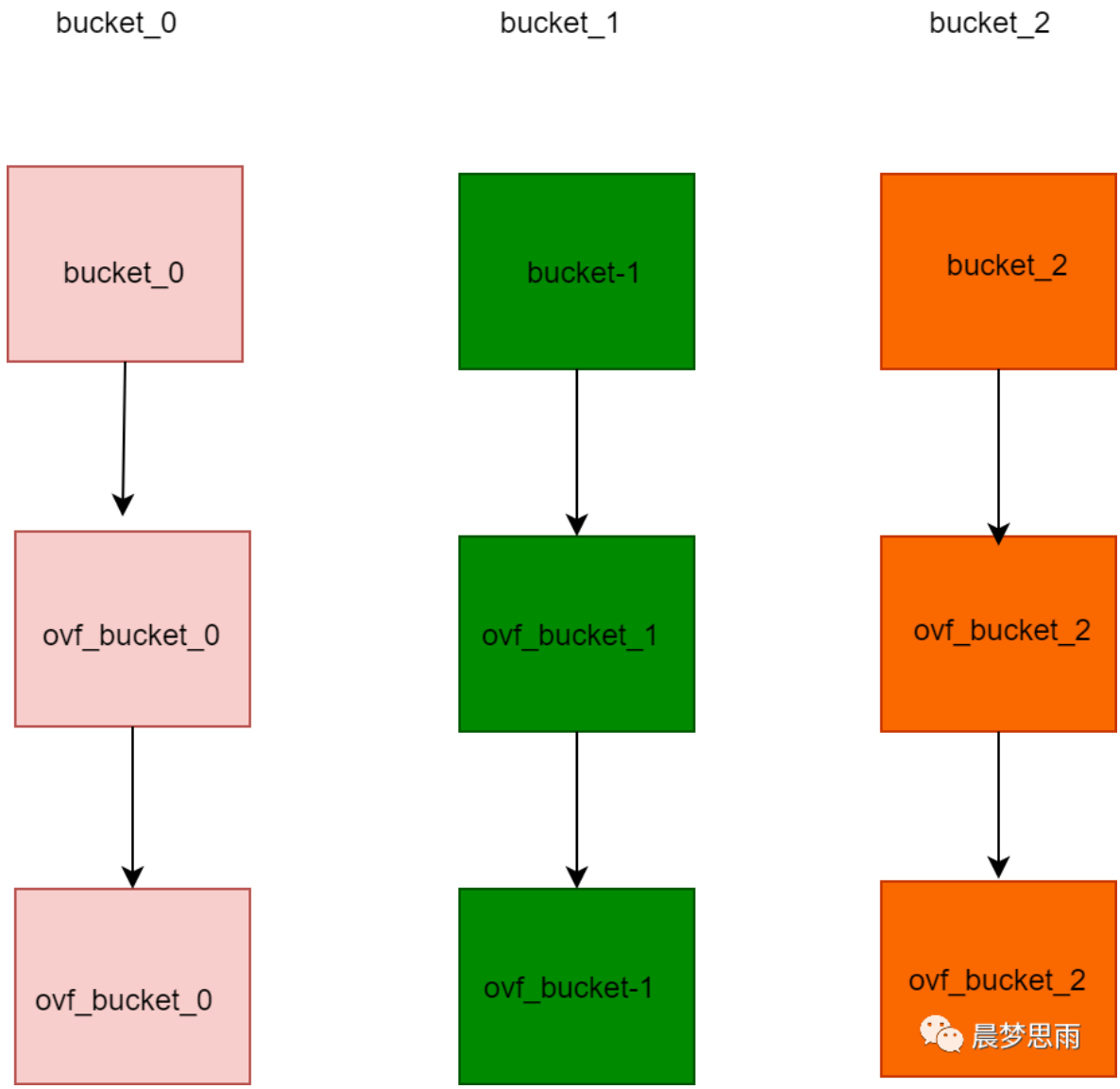
hmap如下所示：


元素个数
int
flags
uint8
扩容常亮相关字段B
uint8
溢出的bucket个数
uint16
用于扩容的指针
*mapextra
buckets的数组指针
unsafe.Pointer
搬迁进度
uinptr
结构扩容的时候用于复制的buckets数组
unsafe.Pointer
hash seed
uint32
晨梦思雨

图中有很多字段，但是便于理解 map 的架构，你只需要关心的只有一个，就是标红的字段：buckets 数组。Golang 的 map 中用于存储的结构是 bucket数组。而 bucket(即bmap)的结构是怎样的呢？
bucket：

相比于 hmap，bucket 的结构显得简单一些，标橙的字段依然是“核心”，我们使用的 map 中的 key 和 value 就存储在这里。

“高位哈希值”数组记录的是当前 bucket 中 key 相关的”索引”，稍后会详细叙述。还有一个字段是一个指向扩容后的 bucket 的指针，使得 bucket 会形成一个链表结构。
整体的结构应该是这样的：



Golang 把求得的哈希值按照用途一分为二：高位和低位。低位用于寻找当前 key属于 hmap 中的哪个 bucket，而高位用于寻找 bucket 中的哪个 key。
需要特别指出的一点是：map中的key/value值都是存到同一个数组中的。这样做的好处是：在key和value的长度不同的时候，可以消除padding带来的空间浪费。

Map 的扩容：当 Go 的 map 长度增长到大于加载因子所需的 map 长度时，Go 语言就会将产生一个新的 bucket 数组，然后把旧的 bucket 数组移到一个属性字段 oldbucket中。

注意：并不是立刻把旧的数组中的元素转义到新的 bucket 当中，而是，只有当访问到具体的某个 bucket 的时候，会把 bucket 中的数据转移到新的 bucket 中。

45. JSON 标准库对 nil slice 和 空 slice 的处理是一致的吗

首先 JSON 标准库对 nil slice 和 空 slice 的处理是不一致。

通常错误的用法，会报数组越界的错误，因为只是声明了slice，却没有给实例化的对象。

```
var slice []int
slice[1] = 0
```

此时slice的值是nil，这种情况可以用于需要返回slice的函数，当函数出现异常的时候，保证函数依然会有nil的返回值。

empty slice 是指slice不为nil，但是slice没有值，slice的底层的空间是空的，此时的定义如下：

```
slice := make([]int,0)
slice := []int{}
```

当我们查询或者处理一个空的列表的时候，这非常有用，它会告诉我们返回的是一个列表，但是列表内没有任何值。总之，nil slice 和 empty slice是不同的东西,需要我们加以区分的。

46.Golang的内存模型，为什么小对象多了会造成gc压力

通常小对象过多会导致 GC 三色法消耗过多的GPU。优化思路是，减少对象分配。

47.Data Race问题怎么解决？能不能不加锁解决这个问题

同步访问共享数据是处理数据竞争的一种有效的方法。

golang在 1.1 之后引入了竞争检测机制，可以使用 go run -race 或者 go build -race来进行静态检测。其在内部的实现是,开启多个协程执行同一个命令， 并且记录下每个变量的状态。

竞争检测器基于C/C++的ThreadSanitizer 运行时库，该库在Google内部代码基地和Chromium找到许多错误。这个技术在2012年九月集成到Go中，从那时开始，它已经在标准库中检测到42个竞争条件。现在，它已经是我们持续构建过程的一部分，当竞争条件出现时，它会继续捕捉到这些错误。

竞争检测器已经完全集成到Go工具链中，仅仅添加-race标志到命令行就使用了检测器。

```
$ go test -race mypkg    // 测试包
$ go run -race mysrc.go  // 编译和运行程序 $ go build -race mycmd
// 构建程序 $ go install -race mypkg // 安装程序
```

要想解决数据竞争的问题可以使用互斥锁sync.Mutex,解决数据竞争(Data race),也可以使用管道解决,使用管道的效率要比互斥锁高。

48.在 range 迭代 slice 时，你怎么修改值的

在 range 迭代中，得到的值其实是元素的一份值拷贝，更新拷贝并不会更改原来的元素，即是拷贝的地址并不是原有元素的地址。

```
func main() {
 data := []int{1, 2, 3}
 for _, v := range data {
  v *= 10  // data 中原有元素是不会被修改的
 }
 fmt.Println("data: ", data) // data:  [1 2 3]
}
```

如果要修改原有元素的值，应该使用索引直接访问。

```
func main() {
 data := []int{1, 2, 3}
```

```
 for i, v := range data {
  data[i] = v * 10
 }
 fmt.Println("data: ", data) // data:  [10 20 30]
}
```

如果你的集合保存的是指向值的指针，需稍作修改。依旧需要使用索引访问元素，不过可以使用 range 出来的元素直接更新原有值。

```
func main() {
 data := []*struct{ num int }{{1}, {2}, {3},}
 for _, v := range data {
  v.num *= 10 // 直接使用指针更新
 }
 fmt.Println(data[0], data[1], data[2]) // &{10} &{20} &{30}
}
```

49.nil interface 和 nil interface 的区别

虽然 interface 看起来像指针类型，但它不是。interface 类型的变量只有在类型和值均为 nil 时才为 nil如果你的 interface 变量的值是跟随其他变量变化的，与 nil 比较相等时小心。如果你的函数返回值类型是 interface，更要小心这个坑：

```
func main() {
   var data *byte
   var in interface{}

   fmt.Println(data, data == nil) // <nil> true
   fmt.Println(in, in == nil) // <nil> true

   in = data
   fmt.Println(in, in == nil) // <nil> false // data 值为 nil，但 in 值不为 nil
}

// 正确示例
func main() {
  doIt := func(arg int) interface{} {
  var result *struct{} = nil

  if arg > 0 {
  result = &struct{}{}
  } else {
  return nil // 明确指明返回 nil
  }

  return result
  }


  if res := doIt(-1); res != nil {
  fmt.Println("Good result: ", res)
  } else {
  fmt.Println("Bad result: ", res) // Bad result: <nil>
  }
}
```

50.select可以用于什么

常用语gorotine的完美退出。

golang 的 select 就是监听 IO 操作，当 IO 操作发生时，触发相应的动作每个case语句里必须是一个IO操作，确切的说，应该是一个面向channel的IO操作。

转自：公众号：晨梦思雨

运维面试题

简述 ETCD 及其特点？

etcd 是 CoreOS 团队发起的开源项目，是一个管理配置信息和服务发现（service discovery）的项目，它的目标是构建一个高可用的分布式键值（key-value）数据库，基于 Go 语言实现。

特点：

- 简单：支持 REST 风格的 HTTP+JSON API
- 安全：支持 HTTPS 方式的访问
- 快速：支持并发 1k/s 的写操作
- 可靠：支持分布式结构，基于 Raft 的一致性算法，Raft 是一套通过选举主节点来实现分布式系统一致性的算法。

简述 ETCD 适应的场景？

etcd 基于其优秀的特点，可广泛的应用于以下场景：

`服务发现（Service Discovery）`：服务发现主要解决在同一个分布式集群中的进程或服务，要如何才能找到对方并建立连接。本质上来说，服务发现就是想要了解集群中是否有进程在监听 udp 或 tcp 端口，并且通过名字就可以查找和连接。
`消息发布与订阅`：在分布式系统中，最适用的一种组件间通信方式就是消息发布与订阅。即构建一个配置共享中心，数据提供者在这个配置中心发布消息，而消息使用者则订阅他们关心的主题，一旦主题有消息发布，就会实时通知订阅者。通过这种方式可以做到分布式系统配置的集中式管理与动态更新。应用中用到的一些配置信息放到 etcd 上进行集中管理。
`负载均衡`：在分布式系统中，为了保证服务的高可用以及数据的一致性，通常都会把数据和服务部署多份，以此达到对等服务，即使其中的某一个服务失效了，也不影响使用。etcd 本身分布式架构存储的信息访问支持负载均衡。etcd 集群化以后，每个 etcd 的核心节点都可以处理用户的请求。所以，把数据量小但是访问频繁的消息数据直接存储到 etcd 中也可以实现负载均衡的效果。
`分布式通知与协调`：与消息发布和订阅类似，都用到了 etcd 中的 Watcher 机制，通过注册与异步通知机制，实现分布式环境下不同系统之间的通知与协调，从而对数据变更做到实时处理。
`分布式锁`：因为 etcd 使用 Raft 算法保持了数据的强一致性，某次操作存储到集群中的值必然是全局一致的，所以很容易实现分布式锁。锁服务有两种使用方式，一是保持独占，二是控制时序。
集群监控与 Leader 竞选：通过 etcd 来进行监控实现起来非常简单并且实时性强。

简述 HAProxy 及其特性？

HAProxy 是可提供高可用性、负载均衡以及基于 TCP 和 HTTP 应用的代理，是免费、快速并且可靠的一种解决方案。
HAProxy 非常适用于并发大（并发达 1w 以上）web 站点，这些站点通常又需要会话保持或七层处理。HAProxy 的运行模式使得它可以很简单安全的整合至当前的架构中，同时可以保护 web 服务器不被暴露到网络上。

HAProxy 的主要特性有：

- 可靠性和稳定性非常好，可以与硬件级的 F5 负载均衡设备相媲美；
- 最高可以同时维护 40000-50000 个并发连接，单位时间内处理的最大请求数为 20000 个，最大处理能力可达 10Git/s；
- 支持多达 8 种负载均衡算法，同时也支持会话保持；
- 支持虚机主机功能，从而实现 web 负载均衡更加灵活；
- 支持连接拒绝、全透明代理等独特的功能；
- 拥有强大的 ACL 支持，用于访问控制；
- 其独特的弹性二叉树数据结构，使数据结构的复杂性上升到了 0(1)，即数据的查寻速度不会随着数据条目的增加而速度有所下降；

- 支持客户端的 keepalive 功能，减少客户端与 haproxy 的多次三次握手导致资源浪费，让多个请求在一个 tcp 连接中完成；
- 支持 TCP 加速，零复制功能，类似于 mmap 机制；
- 支持响应池（response buffering）；
- 支持 RDP 协议；
- 基于源的粘性，类似 nginx 的 ip_hash 功能，把来自同一客户端的请求在一定时间内始终调度到上游的同一服务器；
- 更好统计数据接口，其 web 接口显示后端集群中各个服务器的接收、发送、拒绝、错误等数据的统计信息；
- 详细的健康状态检测，web 接口中有关于对上游服务器的健康检测状态，并提供了一定的管理功能；
- 基于流量的健康评估机制；
- 基于 http 认证；
- 基于命令行的管理接口；
- 日志分析器，可对日志进行分析。

简述 HAProxy 常见的负载均衡策略？

HAProxy 负载均衡策略非常多，常见的有如下 8 种：

- roundrobin：表示简单的轮询。
- static-rr：表示根据权重。
- leastconn：表示最少连接者先处理。
- source：表示根据请求的源 IP，类似 Nginx 的 IP_hash 机制。
- ri：表示根据请求的 URI。
- rl_param：表示根据 HTTP 请求头来锁定每一次 HTTP 请求。
- rdp-cookie(name)：表示根据据 cookie(name)来锁定并哈希每一次 TCP 请求。

简述负载均衡四层和七层的区别？

四层负载均衡器 也称为 4 层交换机，主要通过分析 IP 层及 TCP/UDP 层的流量实现基于 IP 加端口的负载均衡，如常见的 LVS、F5 等；

七层负载均衡器 也称为 7 层交换机，位于 OSI 的最高层，即应用层，此负载均衡器支持多种协议，如 HTTP、FTP、SMTP 等。7 层负载均衡器可根据报文内容，配合一定的负载均衡算法来选择后端服务器，即“内容交换器”。如常见的 HAProxy、Nginx。

简述 LVS、Nginx、HAproxy 的什么异同？

- 相同：三者都是软件负载均衡产品。
- 区别：
- LVS 基于 Linux 操作系统实现软负载均衡，而 HAProxy 和 Nginx 是基于第三方应用实现的软负载均衡；
- LVS 是可实现 4 层的 IP 负载均衡技术，无法实现基于目录、URL 的转发。而 HAProxy 和 Nginx 都可以实现 4 层和 7 层技术，HAProxy 可提供 TCP 和 HTTP 应用的负载均衡综合解决方案；
- LVS 因为工作在 ISO 模型的第四层，其状态监测功能单一，而 HAProxy 在状监测方面功能更丰富、强大，可支持端口、URL、脚本等多种状态检测方式；
- HAProxy 功能强大，但整体性能低于 4 层模式的 LVS 负载均衡。
- Nginx 主要用于 Web 服务器或缓存服务器。

简述 Heartbeat？

Heartbeat 是 Linux-HA 项目中的一个组件，它提供了心跳检测和资源接管、集群中服务的监测、失效切换等功能。
heartbeat 最核心的功能包括两个部分，心跳监测和资源接管。心跳监测可以通过网络链路和串口进行，而且支持冗余链路，它们之间相互发送报文来告诉对方自己当前的状态，如果在指定的时间内未收到对方发送的报文，那么就认为对方失效，这时需启动资源接管模块来接管运行在对方主机上的资源或者服务。

简述 Keepalived 及其工作原理？

Keepalived 是一个基于 VRRP 协议来实现的 LVS 服务高可用方案，可以解决静态路由出现的单点故障问题。

在一个 LVS 服务集群中通常有主服务器（MASTER）和备份服务器（BACKUP）两种角色的服务器，但是对外表现为一个虚拟 IP，主服务器会发送 VRRP 通告信息给备份服务器，当备份服务器收不到 VRRP 消息的时候，即主服务器异常的时候，备份服务器就会接管虚拟 IP，继续提供服务，从而保证了高可用性。

简述 Keepalived 体系主要模块及其作用？

keepalived 体系架构中主要有三个模块，分别是 core、check 和 vrrp。

- `core 模块`为 keepalived 的核心，负责主进程的启动、维护及全局配置文件的加载和解析。
- `vrrp 模块`是来实现 VRRP 协议的。
- `check`负责健康检查，常见的方式有端口检查及 URL 检查。

简述 Keepalived 如何通过健康检查来保证高可用？

Keepalived 工作在 TCP/IP 模型的第三、四和五层，即网络层、传输层和应用层。

- `网络层`，Keepalived 采用 ICMP 协议向服务器集群中的每个节点发送一个 ICMP 的数据包，如果某个节点没有返回响应数据包，则认为此节点发生了故障，Keepalived 将报告次节点失效，并从服务器集群中剔除故障节点。
- `传输层`，Keepalived 利用 TCP 的端口连接和扫描技术来判断集群节点是否正常。如常见的 web 服务默认端口 80，ssh 默认端口 22 等。Keepalived 一旦在传输层探测到相应端口没用响应数据返回，则认为此端口发生异常，从而将此端口对应的节点从服务器集群中剔除。
- `应用层`，可以运行 FTP、telnet、smtp、dns 等各种不同类型的高层协议，Keepalived 的运行方式也更加全面化和复杂化，用户可以通过自定义 Keepalived 的工作方式，来设定监测各种程序或服务是否正常，若监测结果与设定的正常结果不一致，将此服务对应的节点从服务器集群中剔除。

Keepalived 通过完整的健康检查机制，保证集群中的所有节点均有效从而实现高可用。

简述 LVS 的概念及其作用？

LVS 是 linux virtual server 的简写 linux 虚拟服务器，是一个虚拟的服务器集群系统，可以在 unix/linux 平台下实现负载均衡集群功能。

LVS 的主要作用是：通过 LVS 提供的负载均衡技术实现一个高性能、高可用的服务器群集。因此 LVS 主要可以实现：

- 把单台计算机无法承受的大规模的并发访问或数据流量分担到多台节点设备上分别处理，减少用户等待响应的时间，提升用户体验。
- 单个重负载的运算分担到多台节点设备上做并行处理，每个节点设备处理结束后，将结果汇总，返回给用户，系统处理能力得到大幅度提高。
- 7*24 小时的服务保证，任意一个或多个设备节点设备宕机，不能影响到业务。在负载均衡集群中，所有计算机节点都应该提供相同的服务，集群负载均衡获取所有对该服务的如站请求。

简述 LVS 的工作模式及其工作过程？

LVS 有三种负载均衡的模式，分别是 VS/NAT（nat 模式）、VS/DR（路由模式）、VS/TUN（隧道模式）。

- NAT 模式（VS-NAT）
- 原理：首先负载均衡器接收到客户的请求数据包时，根据调度算法决定将请求发送给哪个后端的真实服务器（RS）。然后负载均衡器就把客户端发送的请求数据包的目标 IP 地址及端口改成后端真实服务器的 IP 地址（RIP）。真实服务器响应完请求后，查看默认路由，把响应后的数据包发送给负载均衡器，负载均衡器在接收到响应包后，把包的源地址改成虚拟地址（VIP）然后发送回给客户端。
- 优点：集群中的服务器可以使用任何支持 TCP/IP 的操作系统，只要负载均衡器有一个合法的 IP 地址。
- 缺点：扩展性有限，当服务器节点增长过多时，由于所有的请求和应答都需要经过负载均衡器，因此负载均衡器将成为整个系统的瓶颈。
- IP 隧道模式（VS-TUN）
- 原理：首先负载均衡器接收到客户的请求数据包时，根据调度算法决定将请求发送给哪个后端的真实服务器（RS）。然后负载均衡器就把客户端发送的请求报文封装一层 IP 隧道（T-IP）转发到真实服务器（RS）。真实服务器响应完请求后，查看默认路由，把响应后的数据包直接发送给客户端，不需要经过负载均衡器。
- 优点：负载均衡器只负责将请求包分发给后端节点服务器，而 RS 将应答包直接发给用户。所以，减少了负载均衡器的大量数据流动，负载均衡器不再是系统的瓶颈，也能处理很巨大的请求量。
- 缺点：隧道模式的 RS 节点需要合法 IP，这种方式需要所有的服务器支持“IP Tunneling”。
- 直接路由模式（VS-DR）
- 原理：首先负载均衡器接收到客户的请求数据包时，根据调度算法决定将请求发送给哪个后端的真实服务器（RS）。然后负载均衡器就把客户端发送的请求数据包的目标 MAC 地址改成后端真实服务器的 MAC 地址（R-MAC）。真实服务器响应完请求后，查看默认路由，把响应后的数据包直接发送给客户端，不需要经过负载均衡器。
- 优点：负载均衡器只负责将请求包分发给后端节点服务器，而 RS 将应答包直接发给用户。所以，减少了负载均衡器的大量数据流动，负载均衡器不再是系统的瓶颈，也能处理很巨大的请求量。
- 缺点：需要负载均衡器与真实服务器 RS 都有一块网卡连接到同一物理网段上，必须在同一个局域网环境。

简述 LVS 调度器常见算法（均衡策略）？

LVS 调度器用的调度方法基本分为两类：

- 固定调度算法：rr，wrr，dh，sh
- rr：轮询算法，将请求依次分配给不同的 rs 节点，即 RS 节点中均摊分配。适合于 RS 所有节点处理性能接近的情况。
- wrr：加权轮训调度，依据不同 RS 的权值分配任务。权值较高的 RS 将优先获得任务，并且分配到的连接数将比权值低的 RS 更多。相同权值的 RS 得到相同数目的连接数。
- dh：目的地址哈希调度（destination hashing）以目的地址为关键字查找一个静态 hash 表来获得所需 RS。
- sh：源地址哈希调度（source hashing）以源地址为关键字查找一个静态 hash 表来获得需要的 RS。
- 动态调度算法：wlc，lc，lblc，lblcr
- wlc：加权最小连接数调度，假设各台 RS 的权值依次为 Wi，当前 tcp 连接数依次为 Ti，依次去 Ti/Wi 为最小的 RS 作为下一个分配的 RS。
- lc：最小连接数调度（least-connection），IPVS 表存储了所有活动的连接。LB 会比较将连接请求发送到当前连接最少的 RS。
- lblc：基于地址的最小连接数调度（locality-based least-connection）：将来自同一个目的地址的请求分配给同一台 RS，此时这台服务器是尚未满负荷的。否则就将这个请求分配给连接数最小的 RS，并以它作为下一次分配的首先考虑。

简述 LVS、Nginx、HAProxy 各自优缺点？

- Nginx 的优点：

- 工作在网络的 7 层之上，可以针对 http 应用做一些分流的策略，比如针对域名、目录结构。Nginx 正则规则比 HAProxy 更为强大和灵活。
- Nginx 对网络稳定性的依赖非常小，理论上能 ping 通就就能进行负载功能，LVS 对网络稳定性依赖比较大，稳定要求相对更高。
- Nginx 安装和配置、测试比较简单、方便，有清晰的日志用于排查和管理，LVS 的配置、测试就要花比较长的时间了。
- 可以承担高负载压力且稳定，一般能支撑几万次的并发量，负载度比 LVS 相对小些。
- Nginx 可以通过端口检测到服务器内部的故障，比如根据服务器处理网页返回的状态码、超时等等。
- Nginx 不仅仅是一款优秀的负载均衡器/反向代理软件，它同时也是功能强大的 Web 应用服务器。
- Nginx 作为 Web 反向加速缓存越来越成熟了，速度比传统的 Squid 服务器更快，很多场景下都将其作为反向代理加速器。
- Nginx 作为静态网页和图片服务器，这方面的性能非常优秀，同时第三方模块也很多。
- Nginx 的缺点：
- Nginx 仅能支持 http、https 和 Email 协议，这样就在适用范围上面小些。
- 对后端服务器的健康检查，只支持通过端口来检测，不支持通过 url 来检测。
- 不支持 Session 的直接保持，需要通过 ip_hash 来解决。
- LVS 的优点：
- 抗负载能力强、是工作在网络 4 层之上仅作分发之用，没有流量的产生。因此负载均衡软件里的性能最强的，对内存和 cpu 资源消耗比较低。
- LVS 工作稳定，因为其本身抗负载能力很强，自身有完整的双机热备方案。
- 无流量，LVS 只分发请求，而流量并不从它本身出去，这点保证了均衡器 IO 的性能不会收到大流量的影响。
- 应用范围较广，因为 LVS 工作在 4 层，所以它几乎可对所有应用做负载均衡，包括 http、数据库等。
- LVS 的缺点是：
- 软件本身不支持正则表达式处理，不能做动静分离。相对来说，Nginx/HAProxy+Keepalived 则具有明显的优势。
- 如果是网站应用比较庞大的话，LVS/DR+Keepalived 实施起来就比较复杂了。相对来说Nginx/HAProxy+Keepalived 就简单多了。
- HAProxy 的优点：
- HAProxy 也是支持虚拟主机的。
- HAProxy 的优点能够补充 Nginx 的一些缺点，比如支持 Session 的保持，Cookie 的引导，同时支持通过获取指定的 url 来检测后端服务器的状态。
- HAProxy 跟 LVS 类似，本身就只是一款负载均衡软件，单纯从效率上来讲 HAProxy 会比 Nginx 有更出色的负载均衡速度，在并发处理上也是优于 Nginx 的。
- HAProxy 支持 TCP 协议的负载均衡转发。

简述代理服务器的概念及其作用？

代理服务器是一个位于客户端和原始（资源）服务器之间的服务器，为了从原始服务器取得内容，客户端向代理服务器发送一个请求并指定目标原始服务器，然后代理服务器向原始服务器转交请求并将获得的内容返回给客户端。

其主要作用有：

- 资源获取：代替客户端实现从原始服务器的资源获取；
- 加速访问：代理服务器可能离原始服务器更近，从而起到一定的加速作用；
- 缓存作用：代理服务器保存从原始服务器所获取的资源，从而实现客户端快速的获取；
- 隐藏真实地址：代理服务器代替客户端去获取原始服务器资源，从而隐藏客户端真实信息。

简述高可用集群可通过哪两个维度衡量高可用性，各自含义是什么？

- RTO（Recovery Time Objective）：RTO 指服务恢复的时间，最佳的情况是 0，即服务立即恢复；最坏是无穷大，即服务永远无法恢复；
- RPO（Recovery Point Objective）：RPO 指指当灾难发生时允许丢失的数据量，0 意味着使用同步的数据，大于 0 意味着有数据丢失，如“RPO=1 d”指恢复时使用一天前的数据，那么一天之内的数据就丢失了。因此，恢复的最佳情况是 RTO = RPO = 0，几乎无法实现。

简述什么是 CAP 理论？

CAP 理论指出了在分布式系统中需要满足的三个条件，主要包括：

- Consistency（一致性）：所有节点在同一时间具有相同的数据；
- Availability（可用性）：保证每个请求不管成功或者失败都有响应；
- Partition tolerance（分区容错性）：系统中任意信息的丢失或失败不影响系统的继续运行。

CAP 理论的核心是：一个分布式系统不可能同时很好的满足一致性，可用性和分区容错性这三个需求，最多只能同时较好的满足两个。

简述什么是 ACID 理论？

- 原子性(Atomicity)：整体不可分割性，要么全做要不全不做；
- 一致性(Consistency)：事务执行前、后数据库状态均一致；
- 隔离性(Isolation)：在事务未提交前，它操作的数据，对其它用户不可见；
- 持久性 (Durable)：一旦事务成功，将进行永久的变更，记录与 redo 日志。

简述什么是 Kubernetes？

Kubernetes 是一个全新的基于容器技术的分布式系统支撑平台。是 Google 开源的容器集群管理系统（谷歌内部:Borg）。在 Docker 技术的基础上，为容器化的应用提供部署运行、资源调度、服务发现和动态伸缩等一系列完整功能，提高了大规模容器集群管理的便捷性。并且具有完备的集群管理能力，多层次的安全防护和准入机制、多租户应用支撑能力、透明的服务注册和发现机制、内建智能负载均衡器、强大的故障发现和自我修复能力、服务滚动升级和在线扩容能力、可扩展的资源自动调度机制以及多粒度的资源配额管理能力。

简述 Kubernetes 和 Docker 的关系？

- Docker 提供容器的生命周期管理和，Docker 镜像构建运行时容器。它的主要优点是将将软件/应用程序运行所需的设置和依赖项打包到一个容器中，从而实现了可移植性等优点。

- Kubernetes 用于关联和编排在多个主机上运行的容器。

简述 Kubernetes 中什么是 Minikube、Kubectl、Kubelet？

- `Minikube` 是一种可以在本地轻松运行一个单节点 Kubernetes 群集的工具。

- `Kubectl` 是一个命令行工具，可以使用该工具控制 Kubernetes 集群管理器，如检查群集资源，创建、删除和更新组件，查看应用程序。

- `Kubelet` 是一个代理服务，它在每个节点上运行，并使从服务器与主服务器通信。

简述 Kubernetes 常见的部署方式？

常见的 Kubernetes 部署方式有：

- kubeadm：也是推荐的一种部署方式；
- 二进制：
- minikube：在本地轻松运行一个单节点 Kubernetes 群集的工具。

简述 Kubernetes 如何实现集群管理？

在集群管理方面，Kubernetes 将集群中的机器划分为一个 Master 节点和一群工作节点 Node。其中，在 Master 节点运行着集群管理相关的一组进程 kube-apiserver、kube-controller-manager 和 kube-scheduler，这些进程实现了整个集群的资源管理、Pod 调度、弹性伸缩、安全控制、系统监控和纠错等管理能力，并且都是全自动完成的。

简述 Kubernetes 的优势、适应场景及其特点？

Kubernetes 作为一个完备的分布式系统支撑平台，其主要优势：

- 容器编排
- 轻量级
- 开源
- 弹性伸缩
- 负载均衡

Kubernetes 常见场景：

- 快速部署应用
- 快速扩展应用
- 无缝对接新的应用功能
- 节省资源，优化硬件资源的使用

Kubernetes 相关特点：

- 可移植: 支持公有云、私有云、混合云、多重云（multi-cloud）。
- 可扩展: 模块化,、插件化、可挂载、可组合。
- 自动化: 自动部署、自动重启、自动复制、自动伸缩/扩展。

简述 Kubernetes 的缺点或当前的不足之处？

Kubernetes 当前存在的缺点（不足）如下：

- 安装过程和配置相对困难复杂。
- 管理服务相对繁琐。
- 运行和编译需要很多时间。
- 它比其他替代品更昂贵。
- 对于简单的应用程序来说，可能不需要涉及 Kubernetes 即可满足。

简述 Kubernetes 相关基础概念？

- `master`：k8s 集群的管理节点，负责管理集群，提供集群的资源数据访问入口。拥有 Etcd 存储服务（可选），运行 Api Server 进程，Controller Manager 服务进程及 Scheduler 服务进程。

- `node`（worker）：Node（worker）是 Kubernetes 集群架构中运行 Pod 的服务节点，是 Kubernetes 集群操作的单元，用来承载被分配 Pod 的运行，是 Pod 运行的宿主机。运行 docker eninge 服务，守护进程 kunelet 及负载均衡器 kube-proxy。
- `pod`：运行于 Node 节点上，若干相关容器的组合。Pod 内包含的容器运行在同一宿主机上，使用相同的网络命名空间、IP 地址和端口，能够通过 localhost 进行通信。Pod 是 Kurbernetes 进行创建、调度和管理的最小单位，它提供了比容器更高层次的抽象，使得部署和管理更加灵活。一个 Pod 可以包含一个容器或者多个相关容器。
- `label`：Kubernetes 中的 Label 实质是一系列的 Key/Value 键值对，其中 key 与 value 可自定义。Label 可以附加到各种资源对象上，如 Node、Pod、Service、RC 等。一个资源对象可以定义任意数量的 Label，同一个 -Label 也可以被添加到任意数量的资源对象上去。Kubernetes 通过 Label Selector（标签选择器）查询和筛选资源对象。
- `Replication Controller`：Replication Controller 用来管理 Pod 的副本，保证集群中存在指定数量的 Pod 副本。集群中副本的数量大于指定数量，则会停止指定数量之外的多余容器数量。反之，则会启动少于指定数量个数的容器，保证数量不变。Replication Controller 是实现弹性伸缩、动态扩容和滚动升级的核心。
- `Deployment`：Deployment 在内部使用了 RS 来实现目的，Deployment 相当于 RC 的一次升级，其最大的特色为可以随时获知当前 Pod 的部署进度。
- `HPA`（Horizontal Pod Autoscaler）：Pod 的横向自动扩容，也是 Kubernetes 的一种资源，通过追踪分析 RC 控制的所有 Pod 目标的负载变化情况，来确定是否需要针对性的调整 Pod 副本数量。
- `Service`：Service 定义了 Pod 的逻辑集合和访问该集合的策略，是真实服务的抽象。Service 提供了一个统一的服务访问入口以及服务代理和发现机制，关联多个相同 Label 的 Pod，用户不需要了解后台 Pod 是如何运行。
- `Volume`：Volume 是 Pod 中能够被多个容器访问的共享目录，Kubernetes 中的 Volume 是定义在 Pod 上，可以被一个或多个 Pod 中的容器挂载到某个目录下。
- `Namespace`：Namespace 用于实现多租户的资源隔离，可将集群内部的资源对象分配到不同的 Namespace 中，形成逻辑上的不同项目、小组或用户组，便于不同的 Namespace 在共享使用整个集群的资源的同时还能被分别管理。

简述 Kubernetes 集群相关组件？

Kubernetes Master 控制组件，调度管理整个系统（集群），包含如下组件:

- `Kubernetes API Server`：作为 Kubernetes 系统的入口，其封装了核心对象的增删改查操作，以 RESTful API 接口方式提供给外部客户和内部组件调用，集群内各个功能模块之间数据交互和通信的中心枢纽。
- `Kubernetes Scheduler`：为新建立的 Pod 进行节点(node)选择(即分配机器)，负责集群的资源调度。
- `Kubernetes Controller`：负责执行各种控制器，目前已经提供了很多控制器来保证 Kubernetes 的正常运行。
- `Replication Controller`：管理维护 Replication Controller，关联 Replication Controller 和 - Pod，保证 Replication Controller 定义的副本数量与实际运行 Pod 数量一致。
- `Node Controller`：管理维护 Node，定期检查 Node 的健康状态，标识出(失效|未失效)的 Node 节点。
- `Namespace Controller`：管理维护 Namespace，定期清理无效的 Namespace，包括 Namesapce 下的 API 对象，比如 Pod、Service 等。
- `Service Controller`：管理维护 Service，提供负载以及服务代理。
- `EndPoints Controller`：管理维护 Endpoints，关联 Service 和 Pod，创建 Endpoints 为 Service 的后端，当 Pod 发生变化时，实时更新 Endpoints。
- `Service Account Controller`：管理维护 Service Account，为每个 Namespace 创建默认的 Service Account，同时为 Service Account 创建 Service Account Secret。
- `Persistent Volume Controller`：管理维护 Persistent Volume 和 Persistent Volume Claim，为新的 Persistent Volume Claim 分配 Persistent Volume 进行绑定，为释放的 Persistent Volume 执行清理回收。
- `Daemon Set Controller`：管理维护 Daemon Set，负责创建 Daemon Pod，保证指定的 Node 上正常的运行 Daemon Pod。

- `Deployment Controller`：管理维护 Deployment，关联 Deployment 和 Replication Controller，保证运行指定数量的 Pod。当 Deployment 更新时，控制实现 Replication Controller 和 Pod 的更新。
- `Job Controller`：管理维护 Job，为 Jod 创建一次性任务 Pod，保证完成 Job 指定完成的任务数目
- `Pod Autoscaler Controller`：实现 Pod 的自动伸缩，定时获取监控数据，进行策略匹配，当满足条件时执行 Pod 的伸缩动作。

简述 Kubernetes RC 的机制？

Replication Controller 用来管理 Pod 的副本，保证集群中存在指定数量的 Pod 副本。当定义了 RC 并提交至 Kubernetes 集群中之后，Master 节点上的 Controller Manager 组件获悉，并同时巡检系统中当前存活的目标 Pod，并确保目标 Pod 实例的数量刚好等于此 RC 的期望值，若存在过多的 Pod 副本在运行，系统会停止一些 Pod，反之则自动创建一些 Pod。

简述 Kubernetes Replica Set 和 Replication Controller 之间有什么区别？

Replica Set 和 Replication Controller 类似，都是确保在任何给定时间运行指定数量的 Pod 副本。不同之处在于 RS 使用基于集合的选择器，而 Replication Controller 使用基于权限的选择器。

简述 kube-proxy 作用？

kube-proxy 运行在所有节点上，它监听 apiserver 中 service 和 endpoint 的变化情况，创建路由规则以提供服务 IP 和负载均衡功能。简单理解此进程是 Service 的透明代理兼负载均衡器，其核心功能是将到某个 Service 的访问请求转发到后端的多个 Pod 实例上。

简述 kube-proxy iptables 原理？

Kubernetes 从 1.2 版本开始，将 iptables 作为 kube-proxy 的默认模式。iptables 模式下的 kube-proxy 不再起到 Proxy 的作用，其核心功能：通过 API Server 的 Watch 接口实时跟踪 Service 与 Endpoint 的变更信息，并更新对应的 iptables 规则，Client 的请求流量则通过 iptables 的 NAT 机制“直接路由”到目标 Pod。

简述 kube-proxy ipvs 原理？

IPVS 在 Kubernetes1.11 中升级为 GA 稳定版。IPVS 则专门用于高性能负载均衡，并使用更高效的数据结构（Hash 表），允许几乎无限的规模扩张，因此被 kube-proxy 采纳为最新模式。

在 IPVS 模式下，使用 iptables 的扩展 ipset，而不是直接调用 iptables 来生成规则链。iptables 规则链是一个线性的数据结构，ipset 则引入了带索引的数据结构，因此当规则很多时，也可以很高效地查找和匹配。

可以将 ipset 简单理解为一个 IP（段）的集合，这个集合的内容可以是 IP 地址、IP 网段、端口等，iptables 可以直接添加规则对这个“可变的集合”进行操作，这样做的好处在于可以大大减少 iptables 规则的数量，从而减少性能损耗。

简述 kube-proxy ipvs 和 iptables 的异同？

iptables 与 IPVS 都是基于 Netfilter 实现的，但因为定位不同，二者有着本质的差别：iptables 是为防火墙而设计的；IPVS 则专门用于高性能负载均衡，并使用更高效的数据结构（Hash 表），允许几乎无限的规模扩张。

与 iptables 相比，IPVS 拥有以下明显优势：

1. 为大型集群提供了更好的可扩展性和性能；
2. 支持比 iptables 更复杂的复制均衡算法（最小负载、最少连接、加权等）；
3. 支持服务器健康检查和连接重试等功能；
4. 可以动态修改 ipset 的集合，即使 iptables 的规则正在使用这个集合。

简述 Kubernetes 中什么是静态 Pod？

静态 pod 是由 kubelet 进行管理的仅存在于特定 Node 的 Pod 上，他们不能通过 API Server 进行管理，无法与 ReplicationController、Deployment 或者 DaemonSet 进行关联，并且 kubelet 无法对他们进行健康检查。静态 Pod 总

是由 kubelet 进行创建，并且总是在 kubelet 所在的 Node 上运行。

简述 Kubernetes 中 Pod 可能位于的状态？

- `Pending`：API Server 已经创建该 Pod，且 Pod 内还有一个或多个容器的镜像没有创建，包括正在下载镜像的过程。
- `Running`：Pod 内所有容器均已创建，且至少有一个容器处于运行状态、正在启动状态或正在重启状态。
- `Succeeded`：Pod 内所有容器均成功执行退出，且不会重启。
- `Failed`：Pod 内所有容器均已退出，但至少有一个容器退出为失败状态。
- `Unknown`：由于某种原因无法获取该 Pod 状态，可能由于网络通信不畅导致。

简述 Kubernetes 创建一个 Pod 的主要流程？

Kubernetes 中创建一个 Pod 涉及多个组件之间联动，主要流程如下：

1. 客户端提交 Pod 的配置信息（可以是 yaml 文件定义的信息）到 kube-apiserver。
2. Apiserver 收到指令后，通知给 controller-manager 创建一个资源对象。
3. Controller-manager 通过 api-server 将 pod 的配置信息存储到 ETCD 数据中心中。
4. Kube-scheduler 检测到 pod 信息会开始调度预选，会先过滤掉不符合 Pod 资源配置要求的节点，然后开始调度调优，主要是挑选出更适合运行 pod 的节点，然后将 pod 的资源配置单发送到 node 节点上的 kubelet 组件上。
5. Kubelet 根据 scheduler 发来的资源配置单运行 pod，运行成功后，将 pod 的运行信息返回给 scheduler，scheduler 将返回的 pod 运行状况的信息存储到 etcd 数据中心。

简述 Kubernetes 中 Pod 的重启策略？

Pod 重启策略（RestartPolicy）应用于 Pod 内的所有容器，并且仅在 Pod 所处的 Node 上由 kubelet 进行判断和重启操作。当某个容器异常退出或者健康检查失败时，kubelet 将根据 RestartPolicy 的设置来进行相应操作。

Pod 的重启策略包括 Always、OnFailure 和 Never，默认值为 Always。

- `Always`：当容器失效时，由 kubelet 自动重启该容器；
- `OnFailure`：当容器终止运行且退出码不为 0 时，由 kubelet 自动重启该容器；
- `Never`：不论容器运行状态如何，kubelet 都不会重启该容器。

同时 Pod 的重启策略与控制方式关联，当前可用于管理 Pod 的控制器包括 ReplicationController、Job、DaemonSet 及直接管理 kubelet 管理（静态 Pod）。

不同控制器的重启策略限制如下：

- RC 和 DaemonSet：必须设置为 Always，需要保证该容器持续运行；
- Job：OnFailure 或 Never，确保容器执行完成后不再重启；
- kubelet：在 Pod 失效时重启，不论将 RestartPolicy 设置为何值，也不会对 Pod 进行健康检查。

简述 Kubernetes 中 Pod 的健康检查方式？

对 Pod 的健康检查可以通过两类探针来检查：LivenessProbe 和 ReadinessProbe。

- `LivenessProbe 探针`：用于判断容器是否存活（running 状态），如果 LivenessProbe 探针探测到容器不健康，则 kubelet 将杀掉该容器，并根据容器的重启策略做相应处理。若一个容器不包含 LivenessProbe 探针，kubelet 认为该容器的 LivenessProbe 探针返回值用于是“Success”。
- `ReadineeProbe 探针`：用于判断容器是否启动完成（ready 状态）。如果 ReadinessProbe 探针探测到失败，则 Pod 的状态将被修改。Endpoint Controller 将从 Service 的 Endpoint 中删除包含该容器所在 Pod 的 Eenpoint。
- `startupProbe 探针`：启动检查机制，应用一些启动缓慢的业务，避免业务长时间启动而被上面两类探针 kill 掉。

简述 Kubernetes Pod 的 LivenessProbe 探针的常见方式？

kubelet 定期执行 LivenessProbe 探针来诊断容器的健康状态，通常有以下三种方式：

- `ExecAction`：在容器内执行一个命令，若返回码为 0，则表明容器健康。
- `TCPSocketAction`：通过容器的 IP 地址和端口号执行 TCP 检查，若能建立 TCP 连接，则表明容器健康。
- `HTTPGetAction`：通过容器的 IP 地址、端口号及路径调用 HTTP Get 方法，若响应的状态码大于等于 200 且小于 400，则表明容器健康。

简述 Kubernetes Pod 的常见调度方式？

Kubernetes 中，Pod 通常是容器的载体，主要有如下常见调度方式：

- Deployment 或 RC：该调度策略主要功能就是自动部署一个容器应用的多份副本，以及持续监控副本的数量，在集群内始终维持用户指定的副本数量。
- NodeSelector：定向调度，当需要手动指定将 Pod 调度到特定 Node 上，可以通过 Node 的标签（Label）和 Pod 的 nodeSelector 属性相匹配。
- NodeAffinity 亲和性调度：亲和性调度机制极大的扩展了 Pod 的调度能力，目前有两种节点亲和力表达：
- requiredDuringSchedulingIgnoredDuringExecution：硬规则，必须满足指定的规则，调度器才可以调度 Pod 至 Node 上（类似 nodeSelector，语法不同）。
- preferredDuringSchedulingIgnoredDuringExecution：软规则，优先调度至满足的 Node 的节点，但不强求，多个优先级规则还可以设置权重值。
- Taints 和 Tolerations（污点和容忍）：
- Taint：使 Node 拒绝特定 Pod 运行；
- Toleration：为 Pod 的属性，表示 Pod 能容忍（运行）标注了 Taint 的 Node。

简述 Kubernetes 初始化容器（init container）？

`init container` 的运行方式与应用容器不同，它们必须先于应用容器执行完成，当设置了多个 init container 时，将按顺序逐个运行，并且只有前一个 init container 运行成功后才能运行后一个 init container。当所有 init container 都成功运行后，Kubernetes 才会初始化 Pod 的各种信息，并开始创建和运行应用容器。

简述 Kubernetes deployment 升级过程？

- 初始创建 Deployment 时，系统创建了一个 ReplicaSet，并按用户的需求创建了对应数量的 Pod 副本。
- 当更新 Deployment 时，系统创建了一个新的 ReplicaSet，并将其副本数量扩展到 1，然后将旧 ReplicaSet 缩减为 2。
- 之后，系统继续按照相同的更新策略对新旧两个 ReplicaSet 进行逐个调整。
- 最后，新的 ReplicaSet 运行了对应个新版本 Pod 副本，旧的 ReplicaSet 副本数量则缩减为 0。

简述 Kubernetes deployment 升级策略？

在 Deployment 的定义中，可以通过 spec.strategy 指定 Pod 更新的策略，目前支持两种策略：Recreate（重建）和 RollingUpdate（滚动更新），默认值为 RollingUpdate。

- `Recreate`：设置 spec.strategy.type=Recreate，表示 Deployment 在更新 Pod 时，会先杀掉所有正在运行的 Pod，然后创建新的 Pod。
- `RollingUpdate`：设置 spec.strategy.type=RollingUpdate，表示 Deployment 会以滚动更新的方式来逐个更新 Pod。同时，可以通过设置 spec.strategy.rollingUpdate 下的两个参数（maxUnavailable 和 maxSurge）来控制滚动

更新的过程。

简述 Kubernetes DaemonSet 类型的资源特性？

DaemonSet 资源对象会在每个 Kubernetes 集群中的节点上运行，并且每个节点只能运行一个 pod，这是它和 deployment 资源对象的最大也是唯一的区别。

因此，在定义 yaml 文件中，不支持定义 replicas。

它的一般使用场景如下：

- 在去做每个节点的日志收集工作。
- 监控每个节点的的运行状态。

简述 Kubernetes 自动扩容机制？

Kubernetes 使用 Horizontal Pod Autoscaler（HPA）的控制器实现基于 CPU 使用率进行自动 Pod 扩缩容的功能。

HPA 控制器周期性地监测目标 Pod 的资源性能指标，并与 HPA 资源对象中的扩缩容条件进行对比，在满足条件时对 Pod 副本数量进行调整。

- **HPA 原理**

ubernetes 中的某个 Metrics Server（Heapster 或自定义 Metrics Server）持续采集所有 Pod 副本的指标数据。

HPA 控制器通过 Metrics Server 的 API（Heapster 的 API 或聚合 API）获取这些数据，基于用户定义的扩缩容规则进行计算，得到目标 Pod 副本数量。

当目标 Pod 副本数量与当前副本数量不同时，HPA 控制器就向 Pod 的副本控制器（Deployment、RC 或 ReplicaSet）发起 scale 操作，调整 Pod 的副本数量，完成扩缩容操作。

简述 Kubernetes Service 类型？

通过创建 Service，可以为一组具有相同功能的容器应用提供一个统一的入口地址，并且将请求负载分发到后端的各个容器应用上。其主要类型有：

- `ClusterIP`：虚拟的服务 IP 地址，该地址用于 Kubernetes 集群内部的 Pod 访问，在 Node 上 kube-proxy 通过设置的 iptables 规则进行转发；
- `NodePort`：使用宿主机的端口，使能够访问各 Node 的外部客户端通过 Node 的 IP 地址和端口号就能访问服务；
- `LoadBalancer`：使用外接负载均衡器完成到服务的负载分发，需要在 spec.status.loadBalancer 字段指定外部负载均衡器的 IP 地址，通常用于公有云。

简述 Kubernetes Service 分发后端的策略？

Service 负载分发的策略有：RoundRobin 和 SessionAffinity

- RoundRobin：默认为轮询模式，即轮询将请求转发到后端的各个 Pod 上。
- SessionAffinity：基于客户端 IP 地址进行会话保持的模式，即第 1 次将某个客户端发起的请求转发到后端的某个 Pod 上，之后从相同的客户端发起的请求都将被转发到后端相同的 Pod 上。

简述 Kubernetes Headless Service？

在某些应用场景中，若需要人为指定负载均衡器，不使用 Service 提供的默认负载均衡的功能，或者应用程序希望知道属于同组服务的其他实例。

Kubernetes 提供了 Headless Service 来实现这种功能，即不为 Service 设置 ClusterIP（入口 IP 地址），仅通过 Label Selector 将后端的 Pod 列表返回给调用的客户端。

简述 Kubernetes 外部如何访问集群内的服务？

对于 Kubernetes，集群外的客户端默认情况，无法通过 Pod 的 IP 地址或者 Service 的虚拟 IP 地址:虚拟端口号进行访问。

通常可以通过以下方式进行访问 Kubernetes 集群内的服务：

- 映射 Pod 到物理机：将 Pod 端口号映射到宿主机，即在 Pod 中采用 hostPort 方式，以使客户端应用能够通过物理机访问容器应用。
- 映射 Service 到物理机：将 Service 端口号映射到宿主机，即在 Service 中采用 nodePort 方式，以使客户端应用能够通过物理机访问容器应用。
- 映射 Sercie 到 LoadBalancer：通过设置 LoadBalancer 映射到云服务商提供的 LoadBalancer 地址。这种用法仅用于在公有云服务提供商的云平台上设置 Service 的场景。

简述 Kubernetes ingress？

Kubernetes 的 Ingress 资源对象，用于将不同 URL 的访问请求转发到后端不同的 Service，以实现 HTTP 层的业务路由机制。

Kubernetes 使用了 Ingress 策略和 Ingress Controller，两者结合并实现了一个完整的 Ingress 负载均衡器。

使用 Ingress 进行负载分发时，Ingress Controller 基于 Ingress 规则将客户端请求直接转发到 Service 对应的后端 Endpoint（Pod）上，从而跳过 kube-proxy 的转发功能，kube-proxy 不再起作用，全过程为：ingress controller + ingress 规则 —-> services。

同时当 Ingress Controller 提供的是对外服务，则实际上实现的是边缘路由器的功能。

简述 Kubernetes 镜像的下载策略？

K8s 的镜像下载策略有三种：Always、Never、IFNotPresent。

- `Always`：镜像标签为 latest 时，总是从指定的仓库中获取镜像。
- `Never`：禁止从仓库中下载镜像，也就是说只能使用本地镜像。
- `IfNotPresent`：仅当本地没有对应镜像时，才从目标仓库中下载。默认的镜像下载策略是：当镜像标签是 latest 时，默认策略是 Always；当镜像标签是自定义时（也就是标签不是 latest），那么默认策略是 IfNotPresent。

简述 Kubernetes 的负载均衡器？

负载均衡器是暴露服务的最常见和标准方式之一。

根据工作环境使用两种类型的负载均衡器，即内部负载均衡器或外部负载均衡器。

内部负载均衡器自动平衡负载并使用所需配置分配容器，而外部负载均衡器将流量从外部负载引导至后端容器。

简述 Kubernetes 各模块如何与 API Server 通信？

Kubernetes API Server 作为集群的核心，负责集群各功能模块之间的通信。

集群内的各个功能模块通过 API Server 将信息存入 etcd，当需要获取和操作这些数据时，则通过 API Server 提供的 REST 接口（用 GET、LIST 或 WATCH 方法）来实现，从而实现各模块之间的信息交互。

如 kubelet 进程与 API Server 的交互：每个 Node 上的 kubelet 每隔一个时间周期，就会调用一次 API Server 的 REST 接口报告自身状态，API Server 在接收到这些信息后，会将节点状态信息更新到 etcd 中。

如 kube-controller-manager 进程与 API Server 的交互：kube-controller-manager 中的 Node Controller 模块通过 API Server 提供的 Watch 接口实时监控 Node 的信息，并做相应处理。

如 kube-scheduler 进程与 API Server 的交互：Scheduler 通过 API Server 的 Watch 接口监听到新建 Pod 副本的信息后，会检索所有符合该 Pod 要求的 Node 列表，开始执行 Pod 调度逻辑，在调度成功后将 Pod 绑定到目标节点上。

简述 Kubernetes Scheduler 作用及实现原理？

Kubernetes Scheduler 是负责 Pod 调度的重要功能模块，Kubernetes Scheduler 在整个系统中承担了“承上启下”的重要功能，“承上”是指它负责接收 Controller Manager 创建的新 Pod，为其调度至目标 Node；“启下”是指调度完成后，目标 Node 上的 kubelet 服务进程接管后继工作，负责 Pod 接下来生命周期。

Kubernetes Scheduler 的作用是将待调度的 Pod（API 新创建的 Pod、Controller Manager 为补足副本而创建的 Pod 等）按照特定的调度算法和调度策略绑定（Binding）到集群中某个合适的 Node 上，并将绑定信息写入 etcd 中。

在整个调度过程中涉及**三个对象**：

分别是待调度 Pod 列表、可用 Node 列表，以及调度算法和策略。

Kubernetes Scheduler 通过调度算法调度为待调度 Pod 列表中的每个 Pod 从 Node 列表中选择一个最适合的 Node 来实现 Pod 的调度。

随后，目标节点上的 kubelet 通过 API Server 监听到 Kubernetes Scheduler 产生的 Pod 绑定事件，然后获取对应的 Pod 清单，下载 Image 镜像并启动容器。

简述 Kubernetes Scheduler 使用哪两种算法将 Pod 绑定到 worker 节点？

Kubernetes Scheduler 根据如下两种调度算法将 Pod 绑定到最合适的工作节点：

- `预选（Predicates）`：输入是所有节点，输出是满足预选条件的节点。kube-scheduler 根据预选策略过滤掉不满足策略的 Nodes。如果某节点的资源不足或者不满足预选策略的条件则无法通过预选。如“Node 的 label 必须与 Pod 的 Selector 一致”。
- `优选（Priorities）`：输入是预选阶段筛选出的节点，优选会根据优先策略为通过预选的 Nodes 进行打分排名，选择得分最高的 Node。例如，资源越富裕、负载越小的 Node 可能具有越高的排名。

简述 Kubernetes kubelet 的作用？

在 Kubernetes 集群中，在每个 Node（又称 Worker）上都会启动一个 kubelet 服务进程。

该进程用于处理 Master 下发到本节点的任务，管理 Pod 及 Pod 中的容器。

每个 kubelet 进程都会在 API Server 上注册节点自身的信息，定期向 Master 汇报节点资源的使用情况，并通过 cAdvisor 监控容器和节点资源。

简述 Kubernetes kubelet 监控 Worker 节点资源是使用什么组件来实现的？

kubelet 使用 cAdvisor 对 worker 节点资源进行监控。

在 Kubernetes 系统中，cAdvisor 已被默认集成到 kubelet 组件内，当 kubelet 服务启动时，它会自动启动 cAdvisor 服务，然后 cAdvisor 会实时采集所在节点的性能指标及在节点上运行的容器的性能指标。

简述 Kubernetes 如何保证集群的安全性？

Kubernetes 通过一系列机制来实现集群的安全控制，

主要有如下不同的维度：

- 基础设施方面：保证容器与其所在宿主机的隔离；
- 权限方面：
- 最小权限原则：合理限制所有组件的权限，确保组件只执行它被授权的行为，通过限制单个组件的能力来限制它的权限范围。
- 用户权限：划分普通用户和管理员的角色。
- 集群方面：
- API Server 的认证授权：Kubernetes 集群中所有资源的访问和变更都是通过 Kubernetes API Server 来实现的，因此需要建议采用更安全的 HTTPS 或 Token 来识别和认证客户端身份（Authentication），以及随后访问权限的授权（Authorization）环节。
- API Server 的授权管理：通过授权策略来决定一个 API 调用是否合法。对合法用户进行授权并且随后在用户访问时进行鉴权，建议采用更安全的 RBAC 方式来提升集群安全授权。
- 敏感数据引入 Secret 机制：对于集群敏感数据建议使用 Secret 方式进行保护。
- AdmissionControl（准入机制）：对 kubernetes api 的请求过程中，顺序为：先经过认证 & 授权，然后执行准入操作，最后对目标对象进行操作。

简述 Kubernetes 准入机制？

在对集群进行请求时，每个准入控制代码都按照一定顺序执行。

如果有一个准入控制拒绝了此次请求，那么整个请求的结果将会立即返回，并提示用户相应的 error 信息。

`准入控制`（AdmissionControl）准入控制本质上为一段准入代码，在对 kubernetes api 的请求过程中，顺序为：先经过认证 & 授权，然后执行准入操作，最后对目标对象进行操作。常用组件（控制代码）如下：

- `AlwaysAdmit`：允许所有请求
- `AlwaysDeny`：禁止所有请求，多用于测试环境。
- `ServiceAccount`：它将 serviceAccounts 实现了自动化，它会辅助 serviceAccount 做一些事情，比如如果 pod 没有 serviceAccount 属性，它会自动添加一个 default，并确保 pod 的 serviceAccount 始终存在。
- `LimitRanger`：观察所有的请求，确保没有违反已经定义好的约束条件，这些条件定义在 namespace 中 LimitRange 对象中。
  NamespaceExists：观察所有的请求，如果请求尝试创建一个不存在的 namespace，则这个请求被拒绝。

简述 Kubernetes RBAC 及其特点（优势）？

RBAC 是基于角色的访问控制，是一种基于个人用户的角色来管理对计算机或网络资源的访问的方法。

相对于其他授权模式，RBAC 具有如下优势：

- 对集群中的资源和非资源权限均有完整的覆盖。
- 整个 RBAC 完全由几个 API 对象完成， 同其他 API 对象一样， 可以用 kubectl 或 API 进行操作。
- 可以在运行时进行调整，无须重新启动 API Server。

简述 Kubernetes Secret 作用？

Secret 对象，主要作用是保管私密数据，比如密码、OAuth Tokens、SSH Keys 等信息。

将这些私密信息放在 Secret 对象中比直接放在 Pod 或 Docker Image 中更安全，也更便于使用和分发。

简述 Kubernetes Secret 有哪些使用方式？

创建完 secret 之后，可通过如下三种方式使用：

- 在创建 Pod 时，通过为 Pod 指定 Service Account 来自动使用该 Secret。
- 通过挂载该 Secret 到 Pod 来使用它。
- 在 Docker 镜像下载时使用，通过指定 Pod 的 spc.ImagePullSecrets 来引用它。

简述 Kubernetes PodSecurityPolicy 机制？

Kubernetes PodSecurityPolicy 是为了更精细地控制 Pod 对资源的使用方式以及提升安全策略。

在开启 PodSecurityPolicy 准入控制器后，Kubernetes 默认不允许创建任何 Pod，需要创建 PodSecurityPolicy 策略和相应的 RBAC 授权策略（Authorizing Policies），Pod 才能创建成功。

简述 Kubernetes PodSecurityPolicy 机制能实现哪些安全策略？

在 PodSecurityPolicy 对象中可以设置不同字段来控制 Pod 运行时的各种安全策略，常见的有：

- 特权模式：privileged 是否允许 Pod 以特权模式运行。
- 宿主机资源：控制 Pod 对宿主机资源的控制，如 hostPID：是否允许 Pod 共享宿主机的进程空间。
- 用户和组：设置运行容器的用户 ID（范围）或组（范围）。
- 提升权限：AllowPrivilegeEscalation：设置容器内的子进程是否可以提升权限，通常在设置非 root 用户（MustRunAsNonRoot）时进行设置。
- SELinux：进行 SELinux 的相关配置。

简述 Kubernetes 网络模型？

Kubernetes 网络模型中每个 Pod 都拥有一个独立的 IP 地址，并假定所有 Pod 都在一个可以直接连通的、扁平的网络空间中。

所以不管它们是否运行在同一个 Node（宿主机）中，都要求它们可以直接通过对方的 IP 进行访问。

设计这个原则的原因是，用户不需要额外考虑如何建立 Pod 之间的连接，也不需要考虑如何将容器端口映射到主机端口等问题。

同时为每个 Pod 都设置一个 IP 地址的模型使得同一个 Pod 内的不同容器会共享同一个网络命名空间，也就是同一个 Linux 网络协议栈。这就意味着同一个 Pod 内的容器可以通过 localhost 来连接对方的端口。

在 Kubernetes 的集群里，IP 是以 Pod 为单位进行分配的。

一个 Pod 内部的所有容器共享一个网络堆栈（相当于一个网络命名空间，它们的 IP 地址、网络设备、配置等都是共享的）。

简述 Kubernetes CNI 模型？

CNI 提供了一种应用容器的插件化网络解决方案，定义对容器网络进行操作和配置的规范，通过插件的形式对 CNI 接口进行实现。

CNI 仅关注在创建容器时分配网络资源，和在销毁容器时删除网络资源。在 CNI 模型中只涉及两个概念：容器和网络。

- 容器（Container）：是拥有独立 Linux 网络命名空间的环境，例如使用 Docker 或 rkt 创建的容器。容器需要拥有自己的 Linux 网络命名空间，这是加入网络的必要条件。
- 网络（Network）：表示可以互连的一组实体，这些实体拥有各自独立、唯一的 IP 地址，可以是容器、物理机或者其他网络设备（比如路由器）等。

对容器网络的设置和操作都通过插件（Plugin）进行具体实现，CNI 插件包括两种类型：

CNI Plugin 和 IPAM（IP Address Management）Plugin。

CNI Plugin 负责为容器配置网络资源，IPAM Plugin 负责对容器的 IP 地址进行分配和管理。

IPAM Plugin 作为 CNI Plugin 的一部分，与 CNI Plugin 协同工作。

简述 Kubernetes 网络策略？

为实现细粒度的容器间网络访问隔离策略，Kubernetes 引入 Network Policy。

Network Policy 的主要功能是对 Pod 间的网络通信进行限制和准入控制，设置允许访问或禁止访问的客户端 Pod 列表。

Network Policy 定义网络策略，配合策略控制器（Policy Controller）进行策略的实现。

简述 Kubernetes 网络策略原理？

Network Policy 的工作原理主要为：policy controller 需要实现一个 API Listener，监听用户设置的 Network Policy 定义，并将网络访问规则通过各 Node 的 Agent 进行实际设置（Agent 则需要通过 CNI 网络插件实现）。

简述 Kubernetes 中 flannel 的作用？

Flannel 可以用于 Kubernetes 底层网络的实现，主要作用有：

- 它能协助 Kubernetes，给每一个 Node 上的 Docker 容器都分配互相不冲突的 IP 地址。
- 它能在这些 IP 地址之间建立一个覆盖网络（Overlay Network），通过这个覆盖网络，将数据包原封不动地传递到目标容器内。

简述 Kubernetes Calico 网络组件实现原理？

Calico 是一个基于 BGP 的纯三层的网络方案，与 OpenStack、Kubernetes、AWS、GCE 等云平台都能够良好地集成。

Calico 在每个计算节点都利用 Linux Kernel 实现了一个高效的 vRouter 来负责数据转发。每个 vRouter 都通过 BGP 协议把在本节点上运行的容器的路由信息向整个 Calico 网络广播，并自动设置到达其他节点的路由转发规则。

Calico 保证所有容器之间的数据流量都是通过 **IP 路由的方式完成互联互通的。**

Calico 节点组网时可以直接利用数据中心的网络结构（L2 或者 L3），不需要额外的 NAT、隧道或者 Overlay Network，没有额外的封包解包，能够节约 CPU 运算，提高网络效率。

简述 Kubernetes 共享存储的作用？

Kubernetes 对于有状态的容器应用或者对数据需要持久化的应用，因此需要更加可靠的存储来保存应用产生的重要数据，以便容器应用在重建之后仍然可以使用之前的数据。因此需要使用共享存储。

简述 Kubernetes 数据持久化的方式有哪些？

Kubernetes 通过数据持久化来持久化保存重要数据，常见的方式有：

- EmptyDir（空目录）：没有指定要挂载宿主机上的某个目录，直接由 Pod 内保部映射到宿主机上。类似于 docker 中的 manager volume。
- 场景：
- 只需要临时将数据保存在磁盘上，比如在合并/排序算法中；
- 作为两个容器的共享存储。
- 特性：
- 同个 pod 里面的不同容器，共享同一个持久化目录，当 pod 节点删除时，volume 的数据也会被删除。
- emptyDir 的数据持久化的生命周期和使用的 pod 一致，一般是作为临时存储使用。
- Hostpath：将宿主机上已存在的目录或文件挂载到容器内部。类似于 docker 中的 bind mount 挂载方式。
- 特性：增加了 pod 与节点之间的耦合。
- PersistentVolume（简称 PV）：如基于 NFS 服务的 PV，也可以基于 GFS 的 PV。它的作用是统一数据持久化目录，方便管理。

简述 Kubernetes PV 和 PVC？

PV 是对底层网络共享存储的抽象，将共享存储定义为一种“资源”。

PVC 则是用户对存储资源的一个“申请”。

简述 Kubernetes PV 生命周期内的阶段？

某个 PV 在生命周期中可能处于以下 4 个阶段（Phaes）之一。

- Available：可用状态，还未与某个 PVC 绑定。
- Bound：已与某个 PVC 绑定。
- Released：绑定的 PVC 已经删除，资源已释放，但没有被集群回收。
- Failed：自动资源回收失败。

简述 Kubernetes 所支持的存储供应模式？

Kubernetes 支持两种资源的存储供应模式：静态模式（Static）和动态模式（Dynamic）。

- **静态模式**：集群管理员手工创建许多 PV，在定义 PV 时需要将后端存储的特性进行设置。
- **动态模式**：集群管理员无须手工创建 PV，而是通过 StorageClass 的设置对后端存储进行描述，标记为某种类型。此时要求 PVC 对存储的类型进行声明，系统将自动完成 PV 的创建及与 PVC 的绑定。

简述 Kubernetes CSI 模型？

Kubernetes CSI 是 Kubernetes 推出与容器对接的存储接口标准，存储提供方只需要基于标准接口进行存储插件的实现，就能使用 Kubernetes 的原生存储机制为容器提供存储服务。

CSI 使得存储提供方的代码能和 Kubernetes 代码彻底解耦，部署也与 Kubernetes 核心组件分离，显然，存储插件的开发由提供方自行维护，就能为 Kubernetes 用户提供更多的存储功能，也更加安全可靠。

CSI 包括 CSI Controller 和 CSI Node：

- CSI Controller 的主要功能是提供存储服务视角对存储资源和存储卷进行管理和操作。
- CSI Node 的主要功能是对主机（Node）上的 Volume 进行管理和操作。

简述 Kubernetes Worker 节点加入集群的过程？

通常需要对 Worker 节点进行扩容，从而将应用系统进行水平扩展。

主要过程如下：

- 在该 Node 上安装 Docker、kubelet 和 kube-proxy 服务；
- 然后配置 kubelet 和 kubeproxy 的启动参数，将 Master URL 指定为当前 Kubernetes 集群 Master 的地址，最后启动这些服务；
- 通过 kubelet 默认的自动注册机制，新的 Worker 将会自动加入现有的 Kubernetes 集群中；
- Kubernetes Master 在接受了新 Worker 的注册之后，会自动将其纳入当前集群的调度范围。

简述 Kubernetes Pod 如何实现对节点的资源控制？

Kubernetes 集群里的节点提供的资源主要是计算资源，计算资源是可计量的能被申请、分配和使用的基础资源。

当前 Kubernetes 集群中的计算资源主要包括 CPU、GPU 及 Memory。CPU 与 Memory 是被 Pod 使用的，因此在配置 Pod 时可以通过参数 CPU Request 及 Memory Request 为其中的每个容器指定所需使用的 CPU 与 Memory 量，Kubernetes 会根据 Request 的值去查找有足够资源的 Node 来调度此 Pod。

通常，一个程序所使用的 CPU 与 Memory 是一个动态的量，确切地说，是一个范围，跟它的负载密切相关：负载增加时，CPU 和 Memory 的使用量也会增加。

简述 Kubernetes Requests 和 Limits 如何影响 Pod 的调度？

当一个 Pod 创建成功时，Kubernetes 调度器（Scheduler）会为该 Pod 选择一个节点来执行。

对于每种计算资源（CPU 和 Memory）而言，每个节点都有一个能用于运行 Pod 的最大容量值。调度器在调度时，首先要确保调度后该节点上所有 Pod 的 CPU 和内存的 Requests 总和，不超过该节点能提供给 Pod 使用的 CPU 和 Memory 的最大容量值。

简述 Kubernetes Metric Service？

在 Kubernetes 从 1.10 版本后采用 Metrics Server 作为默认的性能数据采集和监控，主要用于提供核心指标（Core Metrics），包括 Node、Pod 的 CPU 和内存使用指标。

对其他自定义指标（Custom Metrics）的监控则由 Prometheus 等组件来完成。

简述 Kubernetes 中，如何使用 EFK 实现日志的统一管理？

在 Kubernetes 集群环境中，通常一个完整的应用或服务涉及组件过多，建议对日志系统进行集中化管理，通常采用 EFK 实现。

EFK 是 Elasticsearch、Fluentd 和 Kibana 的组合，其各组件功能如下：

- Elasticsearch：是一个搜索引擎，负责存储日志并提供查询接口；
- Fluentd：负责从 Kubernetes 搜集日志，每个 node 节点上面的 fluentd 监控并收集该节点上面的系统日志，并将处理过后的日志信息发送给 Elasticsearch；
- Kibana：提供了一个 Web GUI，用户可以浏览和搜索存储在 Elasticsearch 中的日志。

通过在每台 node 上部署一个以 DaemonSet 方式运行的 fluentd 来收集每台 node 上的日志。

Fluentd 将 docker 日志目录/var/lib/docker/containers 和/var/log 目录挂载到 Pod 中，然后 Pod 会在 node 节点的/var/log/pods 目录中创建新的目录，可以区别不同的容器日志输出，该目录下有一个日志文件链接到/var/lib/docker/contianers 目录下的容器日志输出。

简述 Kubernetes 如何进行优雅的节点关机维护？

由于 Kubernetes 节点运行大量 Pod，因此在进行关机维护之前，建议先使用 kubectl drain 将该节点的 Pod 进行驱逐，然后进行关机维护。

简述 Kubernetes 集群联邦？

Kubernetes 集群联邦可以将多个 Kubernetes 集群作为一个集群进行管理。

因此，可以在一个数据中心/云中创建多个 Kubernetes 集群，并使用集群联邦在一个地方控制/管理所有集群。

简述 Helm 及其优势？

`Helm` 是 Kubernetes 的软件包管理工具。类似 Ubuntu 中使用的 apt、Centos 中使用的 yum 或者 Python 中的 pip 一样。

Helm 能够将一组 K8S 资源打包统一管理, 是查找、共享和使用为 Kubernetes 构建的软件的最佳方式。

Helm 中通常每个包称为一个 Chart，一个 Chart 是一个目录（一般情况下会将目录进行打包压缩，形成 name-version.tgz 格式的单一文件，方便传输和存储）。

- Helm 优势

在 Kubernetes 中部署一个可以使用的应用，需要涉及到很多的 Kubernetes 资源的共同协作。

使用 helm 则具有如下优势：

- 统一管理、配置和更新这些分散的 k8s 的应用资源文件；
- 分发和复用一套应用模板；
- 将应用的一系列资源当做一个软件包管理。
- 对于应用发布者而言，可以通过 Helm 打包应用、管理应用依赖关系、管理应用版本并发布应用到软件仓库。
- 对于使用者而言，使用 Helm 后不用需要编写复杂的应用部署文件，可以以简单的方式在 Kubernetes 上查找、安装、升级、回滚、卸载应用程序。

简述 OpenShift 及其特性？

OpenShift 是一个容器应用程序平台，用于在安全的、可伸缩的资源上部署新应用程序，而配置和管理开销最小。

OpenShift 构建于 Red Hat Enterprise Linux、Docker 和 Kubernetes 之上，为企业级应用程序提供了一个安全且可伸缩的多租户操作系统，同时还提供了集成的应用程序运行时和库。

其主要特性：

- 自助服务平台：OpenShift 允许开发人员使用 Source-to-Image(S2I)从模板或自己的源代码管理存储库创建应用程序。系统管理员可以为用户和项目定义资源配额和限制，以控制系统资源的使用。
- 多语言支持：OpenShift 支持 Java、Node.js、PHP、Perl 以及直接来自 Red Hat 的 Ruby。OpenShift 还支持中间件产品，如 Apache httpd、Apache Tomcat、JBoss EAP、ActiveMQ 和 Fuse。
- 自动化：OpenShift 提供应用程序生命周期管理功能，当上游源或容器映像发生更改时，可以自动重新构建和重新部署容器。根据调度和策略扩展或故障转移应用程序。
- 用户界面：OpenShift 提供用于部署和监视应用程序的 web UI，以及用于远程管理应用程序和资源的 CLi。
- 协作：OpenShift 允许在组织内或与更大的社区共享项目。
- 可伸缩性和高可用性：OpenShift 提供了容器多租户和一个分布式应用程序平台，其中包括弹性，高可用性，以便应用程序能够在物理机器宕机等事件中存活下来。OpenShift 提供了对容器健康状况的自动发现和自动重新部署。

- 容器可移植性：在 OpenShift 中，应用程序和服务使用标准容器映像进行打包，组合应用程序使用 Kubernetes 进行管理。这些映像可以部署到基于这些基础技术的其他平台上。
- 开源：没有厂商锁定。
- 安全性：OpenShift 使用 SELinux 提供多层安全性、基于角色的访问控制以及与外部身份验证系统(如 LDAP 和 OAuth)集成的能力。
- 动态存储管理：OpenShift 使用 Kubernetes 持久卷和持久卷声明的方式为容器数据提供静态和动态存储管理
- 基于云(或不基于云)：可以在裸机服务器、活来自多个供应商的 hypervisor 和大多数 IaaS 云提供商上部署 OpenShift 容器平台。
- 企业级：Red Hat 支持 OpenShift、选定的容器映像和应用程序运行时。可信的第三方容器映像、运行时和应用程序由 Red Hat 认证。可以在 OpenShift 提供的高可用性的强化安全环境中运行内部或第三方应用程序。
- 日志聚合和 metrics：可以在中心节点收集、聚合和分析部署在 OpenShift 上的应用程序的日志信息。OpenShift 能够实时收集关于应用程序的度量和运行时信息，并帮助不断优化性能。
- 其他特性：OpenShift 支持微服务体系结构，OpenShift 的本地特性足以支持 DevOps 流程，很容易与标准和定制的持续集成/持续部署工具集成。

简述 OpenShift projects 及其作用？

OpenShift 管理 projects 和 users。

一个 projects 对 Kubernetes 资源进行分组，以便用户可以使用访问权限。还可以为 projects 分配配额，从而限制了已定义的 pod、volumes、services 和其他资源。

project 允许一组用户独立于其他组组织和管理其内容，必须允许用户访问项目。如果允许创建项目，用户将自动访问自己的项目。

简述 OpenShift 高可用的实现？

OpenShift 平台集群的高可用性(HA)有两个不同的方面：

OpenShift 基础设施本身的 HA(即主机)；以及在 OpenShift 集群中运行的应用程序的 HA。

默认情况下，OpenShift 为 master 节点提供了完全支持的本机 HA 机制。

对于应用程序或“pods”，如果 pod 因任何原因丢失，Kubernetes 将调度另一个副本，将其连接到服务层和持久存储。

如果整个节点丢失，Kubernetes 会为它所有的 pod 安排替换节点，最终所有的应用程序都会重新可用。pod 中的应用程序负责它们自己的状态，因此它们需要自己维护应用程序状态(如 HTTP 会话复制或数据库复制)。

简述 OpenShift 的 SDN 网络实现？

默认情况下，Docker 网络使用仅使用主机虚机网桥 bridge，主机内的所有容器都连接至该网桥。

连接到此桥的所有容器都可以彼此通信，但不能与不同主机上的容器通信。

为了支持跨集群的容器之间的通信，OpenShift 容器平台使用了软件定义的网络(SDN)方法。

软件定义的网络是一种网络模型，它通过**几个网络层的抽象来管理网络服务。**

SDN 将处理流量的软件(称为控制平面)和路由流量的底层机制(称为数据平面)解耦。SDN 支持控制平面和数据平面之间的通信。

在 OpenShift 中，可以为 pod 网络配置三个 SDN 插件:

- ovs-subnet：默认插件，子网提供了一个 flat pod 网络，其中每个 pod 可以与其他 pod 和 service 通信。
- ovs-multitenant：该为 pod 和服务提供了额外的隔离层。当使用此插件时，每个 project 接收一个惟一的虚拟网络 ID (VNID)，该 ID 标识来自属于该 project 的 pod 的流量。通过使用 VNID，来自不同 project 的 pod 不能与其他 project 的 pod 和 service 通信。
- ovs-network policy：此插件允许管理员使用 NetworkPolicy 对象定义自己的隔离策略。

cluster network 由 OpenShift SDN 建立和维护，它使用 Open vSwitch 创建 overlay 网络，master 节点不能通过集群网络访问容器，除非 master 同时也为 node 节点。

简述 OpenShift 角色及其作用？

OpenShift 的角色具有不同级别的访问和策略，包括集群和本地策略。

user 和 group 可以同时与多个 role 关联。

简述 OpenShift 支持哪些身份验证？

OpenShift 容器平台支持的其他认证类型包括:

- Basic Authentication (Remote)：一种通用的后端集成机制，允许用户使用针对远程标识提供者验证的凭据登录到 OpenShift 容器平台。用户将他们的用户名和密码发送到 OpenShift 容器平台，OpenShift 平台通过到服务器的请求验证这些凭据，并将凭据作为基本的 Auth 头传递。这要求用户在登录过程中向 OpenShift 容器平台输入他们的凭据。
- Request Header Authentication：用户使用请求头值(如 X-RemoteUser)登录到 OpenShift 容器平台。它通常与身份验证代理结合使用，身份验证代理对用户进行身份验证，然后通过请求头值为 OpenShift 容器平台提供用户标识。
- Keystone Authentication：Keystone 是一个 OpenStack 项目，提供标识、令牌、目录和策略服务。OpenShift 容器平台与 Keystone 集成，通过配置 OpenStack Keystone v3 服务器将用户存储在内部数据库中，从而支持共享身份验证。这种配置允许用户使用 Keystone 凭证登录 OpenShift 容器平台。
- LDAP Authentication：用户使用他们的 LDAP 凭证登录到 OpenShift 容器平台。在身份验证期间，LDAP 目录将搜索与提供的用户名匹配的条目。如果找到匹配项，则尝试使用条目的专有名称(DN)和提供的密码进行简单绑定。
- GitHub Authentication：GitHub 使用 OAuth，它允许与 OpenShift 容器平台集成使用 OAuth 身份验证来促进令牌交换流。这允许用户使用他们的 GitHub 凭证登录到 OpenShift 容器平台。为了防止使用 GitHub 用户 id 的未授权用户登录到 OpenShift 容器平台集群，可以将访问权限限制在特定的 GitHub 组织中。

简述什么是中间件？

`中间件` 是一种独立的系统软件或服务程序，分布式应用软件借助这种软件在不同的技术之间共享资源。

通常位于客户机/服务器的操作系统中间，是**连接两个独立应用程序或独立系统的软件**。

通过中间件实现两个不同系统之间的信息交换。

转自：https://blog.csdn.net/estarhao/article/details/114703958?spm=1001.2014.3001.5501

第一百零五天

第一百零五天

1.下面代码输出什么？请简要说明。

```
var c = make(chan int)
var a int

func f() {
    a = 1
    <-c
}
func main() {
    go f()
    c <- 0
    print(a)
}
```

- A. 不能编译；
- B. 输出 1；
- C. 输出 0；
- D. panic；

参考答案及解析：B。能正确输出，不过主协程会阻塞 f() 函数的执行。

2.下面代码输出什么？请简要说明。

```
type MyMutex struct {
    count int
    sync.Mutex
}

func main() {
    var mu MyMutex
    mu.Lock()
    var mu1 = mu
    mu.count++
    mu.Unlock()
    mu1.Lock()
    mu1.count++
    mu1.Unlock()
    fmt.Println(mu.count, mu1.count)
}
```

- A. 不能编译；
- B. 输出 1, 1；
- C. 输出 1, 2；
- D. fatal error；

参考答案及解析：D。加锁后复制变量，会将锁的状态也复制，所以 mu1 其实是已经加锁状态，再加锁会死锁。

第一百零四天

第一百零四天

1.关于同步锁，下面说法正确的是？

- A. 当一个 goroutine 获得了 Mutex 后，其他 goroutine 就只能乖乖的等待，除非该 goroutine 释放这个 Mutex；
- B. RWMutex 在读锁占用的情况下，会阻止写，但不阻止读；
- C. RWMutex 在写锁占用情况下，会阻止任何其他 goroutine（无论读和写）进来，整个锁相当于由该 goroutine 独占；
- D. Lock() 操作需要保证有 Unlock() 或 RUnlock() 调用与之对应；

参考答案及解析：ABC。

2.下面代码输出什么？请简要说明。

```
func main() {
    var wg sync.WaitGroup
    wg.Add(1)
    go func() {
        time.Sleep(time.Millisecond)
        wg.Done()
        wg.Add(1)
    }()
    wg.Wait()
}
```

- A. 不能编译；
- B. 无输出，正常退出；
- C. 程序 hang 住；
- D. panic；

参考答案及解析：D。WaitGroup 在调用 Wait() 之后不能再调用 Add() 方法的。

第一百零三天

第一百零三天

1.下面代码输出什么？

```
func main() {
    fmt.Println(doubleScore(0))
    fmt.Println(doubleScore(20.0))
    fmt.Println(doubleScore(50.0))
}
func doubleScore(source float32) (score float32) {
    defer func() {
        if score < 1 || score >= 100 {
            score = source
        }
    }()
    return source * 2
}
```

参考答案及解析：输出 0 40 50。知识点：defer 语句与返回值。函数的 return value 不是原子操作，而是在编译器中分解为两部分：返回值赋值 和 return。

2.下面代码输出什么？请简要说明。

```
var mu sync.RWMutex
var count int

func main() {
    go A()
    time.Sleep(2 * time.Second)
    mu.Lock()
    defer mu.Unlock()
    count++
    fmt.Println(count)
}
func A() {
    mu.RLock()
    defer mu.RUnlock()
    B()
}
func B() {
    time.Sleep(5 * time.Second)
    C()
}
func C() {
    mu.RLock()
    defer mu.RUnlock()
}
```

- A. 不能编译；
- B. 输出 1；
- C. 程序 hang 住；

- D. fatal error；

参考答案及解析：D。当写锁阻塞时，新的读锁是无法申请的（有效防止写锁饥饿），导致死锁。

第一百零二天

第一百零二天

1.ch := make(chan interface{}) 和 ch := make(chan interface{},1) 有什么区别？

参考答案及解析：第一个是声明无缓存通道，第二个是声明缓存为 1 的通道。无缓存通道需要一直有接收者接收数据，写操作才会继续，不然会一直阻塞；而缓冲为 1 则即使没有接收者也不会阻塞，因为缓冲大小是 1 ，只有当放第二个值的时候，第一个还没被人拿走，这时候才会阻塞。注意这两者还是有区别的。

2.下面的代码输出什么？请简要说明。

```
var mu sync.Mutex
var chain string

func main() {
    chain = "main"
    A()
    fmt.Println(chain)
}
func A() {
    mu.Lock()
    defer mu.Unlock()
    chain = chain + " --> A"
    B()
}

func B() {
    chain = chain + " --> B"
    C()
}

func C() {
    mu.Lock()
    defer mu.Unlock()
    chain = chain + " --> C"
}
```

- A. 不能编译；
- B. 输出 main –> A –> B –> C；
- C. 输出 main；
- D. fatal error；

参考答案即解析：D。使用 Lock() 加锁后，不能再继续对其加锁，直到利用 Unlock() 解锁后才能再加锁。

引自博客《鸟窝》 https://colobu.com/

第一百零一天

第一百零一天

1.关于循环语句，下面说法正确的有？

- A. 循环语句既支持 for 关键字，也支持 while 和 do-while；
- B. 关键字for的基本使用方法与C/C++中没有任何差异；
- C. for 循环支持 continue 和 break 来控制循环，但是它提供了一个更高级的 break，可以选择中断哪一个循环；
- D. for 循环不支持以逗号为间隔的多个赋值语句，必须使用平行赋值的方式来初始化多个变量；

参考答案及解析：CD。

2.下面代码的功能是从小到大找出 17 和 38 的 3 个公倍数，请问下面的代码有什么问题？

```
var ch chan int = make(chan int)

func generate() {
    for i := 17; i < 5000; i += 17 {
        ch <- i
        time.Sleep(1 * time.Millisecond)
    }
    close(ch)
}

func main() {
    timeout := time.After(800 * time.Millisecond)
    go generate()
    found := 0
    for {
        select {
        case i, ok := <-ch:
            if ok {
                if i%38 == 0 {
                    fmt.Println(i, "is a multiple of 17 and 38")
                    found++
                      if found == 3 {
                        break
                    }
                }
            } else {
                break
            }
        case <-timeout:
            fmt.Println("timed out")
            break
        }
    }
    fmt.Println("The end")
}
```

参考答案即解析：break 会跳出 select 块，但不会跳出 for 循环。这算是一个比较容易掉的坑。可以使用 break label 特性或者 goto 功能解决这个问题，这里使用 break label 作个示例。

```
var ch chan int = make(chan int)

func generate() {
    for i := 17; i < 5000; i += 17 {
        ch <- i
        time.Sleep(1 * time.Millisecond)
    }
    close(ch)
}

func main() {
    timeout := time.After(800 * time.Millisecond)
    go generate()
    found := 0
    MAIN_LOOP:
    for {
        select {
        case i, ok := <-ch:
            if ok {
                if i%38 == 0 {
                    fmt.Println(i, "is a multiple of 17 and 38")
                    found++
                    if found == 3 {
                        break MAIN_LOOP
                    }
                }
            } else {
                break MAIN_LOOP
            }
        case <-timeout:
            fmt.Println("timed out")
            break MAIN_LOOP
        }
    }
    fmt.Println("The end")
}
```

第一百天

第一百天

1.下面代码输出什么？

```
func main() {
    m := map[string]int{
        "G": 7, "A": 1,
        "C": 3, "E": 5,
        "D": 4, "B": 2,
        "F": 6, "I": 9,
        "H": 8,
    }
    var order []string
    for k, _ := range m {
        order = append(order, k)
    }
    fmt.Println(order)
}
```

参考答案即解析：按字母无序输出。知识点：遍历 map 是无序的。

2.下面的代码有什么问题？

```
type UserAges struct {
    ages map[string]int
    sync.Mutex
}

func (ua *UserAges) Add(name string, age int) {
    ua.Lock()
    defer ua.Unlock()
    ua.ages[name] = age
}

func (ua *UserAges) Get(name string) int {
    if age, ok := ua.ages[name]; ok {
        return age
    }
    return -1
}

func main() {
    count := 1000
    gw := sync.WaitGroup{}
    gw.Add(count * 3)
    u := UserAges{ages: map[string]int{}}
    add := func(i int) {
        u.Add(fmt.Sprintf("user_%d", i), i)
        gw.Done()
    }
    for i := 0; i < count; i++ {
        go add(i)
        go add(i)
```

```
    }
    for i := 0; i < count; i++ {
        go func(i int) {
            defer gw.Done()
            u.Get(fmt.Sprintf("user_%d", i))
        }(i)
    }
    gw.Wait()
    fmt.Println("Done")
}
```

参考答案即解析：在执行 Get() 方法时可能报错。知识点：读写锁。虽然可以使用 sync.Mutex 做写锁，但是 map 是并发读写不安全的。map 属于引用类型，并发读写时多个协程是通过指针访问同一个地址，即访问共享变量，此时同时读写资源存在竞争关系，会报错 “fatal error: concurrent map read and map write”。

第九十九天

第九十九天

1.下面代码能编译通过吗？

```
func main() {
    true := false
    fmt.Println(true)
}
```

参考答案即解析：编译通过。true 是预定义标识符可以用作变量名，但是不建议这么做。

2.下面的代码输出什么？

```
func watShadowDefer(i int) (ret int) {
    ret = i * 2
    if ret > 10 {
        ret := 10
        defer func() {
            ret = ret + 1
        }()
    }
    return
}

func main() {
    result := watShadowDefer(50)
    fmt.Println(result)
}
```

参考答案即解析：100。知识点：变量作用域和defer 返回值。

第九十八天

第九十八天

1.下面代码输出什么？

```
func main() {
    a := 1
    for i := 0;i<5;i++ {
        a := a + 1
        a = a * 2
    }
    fmt.Println(a)
}
```

参考答案及解析：1。知识点：变量的作用域。注意 for 语句的变量 a 是重新声明，它的作用范围只在 for 语句范围内。

2.下面的代码输出什么？

```
func test(i int) (ret int) {
    ret = i * 2
    if ret > 10 {
        ret := 10
        return
    }
    return
}

func main() {
    result := test(10)
    fmt.Println(result)
}
```

参考答案即解析：编译错误。知识点：变量的作用域。编译错误信息：ret is shadowed during return。

第九十七天

第九十七天

1.关于map，下面说法正确的是？

- A. map 反序列化时 json.unmarshal() 的入参必须为map的地址；
- B. 在函数调用中传递 map，则子函数中对 map 元素的增加不会导致父函数中 map 的修改；
- C. 在函数调用中传递 map，则子函数中对 map 元素的修改不会导致父函数中 map 的修改；
- D. 不能使用内置函数 delete() 删除 map 的元素；

参考答案及解析：A。知识点：map 的使用。可以查看《Go Map》

2.下面代码输出什么？请简要说明。

```
type Foo struct {
    val int
}

func (f Foo) Inc(inc int) {
    f.val += inc
}

func main() {
    var f Foo
    f.Inc(100)
    fmt.Println(f.val)
}
```

参考答案及解析：输出 0。使用值类型接收者定义的方法，调用的时候，使用的是值的副本，对副本操作不会影响的原来的值。如果想要在调用函数中修改原值，可以使用指针接收者定义的方法。

```
type Foo struct {
    val int
}

func (f *Foo) Inc(inc int) {
    f.val += inc
}

func main() {
    f := &Foo{}
    f.Inc(100)
    fmt.Println(f.val)  // 100
}
```

第九十六天

第九十六天

1.下面的代码输出什么？

```
type Point struct{ x, y int }

func main() {
    s := []Point{
        {1, 2},
        {3, 4},
    }
    for _, p := range s {
        p.x, p.y = p.y, p.x
    }
    fmt.Println(s)
}
```

参考答案及解析：输出 [{1 2} {3 4}]。知识点：for range 循环。range 循环的时候，获取到的元素值是副本，就比如这里的 p。修复代码示例：

```
type Point struct{ x, y int }

func main() {
    s := []*Point{
        &Point{1, 2},
        &Point{3, 4},
    }
    for _, p := range s {
        p.x, p.y = p.y, p.x
    }
    fmt.Println(*s[0])
    fmt.Println(*s[1])
}
```

2.下面的代码有什么隐患？

```
func get() []byte {
    raw := make([]byte, 10000)
    fmt.Println(len(raw), cap(raw), &raw[0])
    return raw[:3]
}

func main() {
    data := get()
    fmt.Println(len(data), cap(data), &data[0])
}
```

参考答案及解析：get() 函数返回的切片与原切片公用底层数组，如果在调用函数里面（这里是 main() 函数）修改返回的切片，将会影响到原切片。为了避免掉入陷阱，可以如下修改：

```
func get() []byte {
    raw := make([]byte, 10000)
    fmt.Println(len(raw), cap(raw), &raw[0])
    res := make([]byte, 3)
    copy(res, raw[:3])
    return res
}

func main() {
    data := get()
    fmt.Println(len(data), cap(data), &data[0])
}
```

第九十五天

第九十五天

1.下面代码输出什么？请简要说明。

```
type foo struct{ Val int }

type bar struct{ Val int }

func main() {
    a := &foo{Val: 5}
    b := &foo{Val: 5}
    c := foo{Val: 5}
    d := bar{Val: 5}
    e := bar{Val: 5}
    f := bar{Val: 5}
    fmt.Print(a == b, c == foo(d), e == f)
}
```

参考答案及解析：false true true。这道题唯一有疑问的地方就在第一个比较，Go 语言里没有引用变量，每个变量都占用一个惟一的内存位置，所以第一个比较输出 false。这个知识点在《Go 语言没有引用传递》有介绍。

2.下面代码输出什么？

```
func A() int {
    time.Sleep(100 * time.Millisecond)
    return 1
}

func B() int {
    time.Sleep(1000 * time.Millisecond)
    return 2
}

func main() {
    ch := make(chan int, 1)
    go func() {
        select {
        case ch <- A():
        case ch <- B():
        default:
            ch <- 3
        }
    }()
    fmt.Println(<-ch)
}
```

参考答案及解析：1、2随机输出。

第九十四天

第九十四天

1.下面这段代码输出什么？请简要说明。

```
func main() {
    a := 2 ^ 15
    b := 4 ^ 15
    if a > b {
        println("a")
    } else {
        println("b")
    }
}
```

参考答案及解析：a。Go 语言里面 ^ 表示按位异或，而不是求幂。

```
0010 ^ 1111 == 1101    (2^15 == 13)
0100 ^ 1111 == 1011    (4^15 == 11)
```

2.下面哪些函数不能通过编译？

```
func A(string string) string {
    return string + string
}

func B(len int) int {
    return len + len
}

func C(val, default string) string {
    if val == "" {
        return default
    }
    return val
}
```

参考答案及解析：C() 函数不能通过编译。C() 函数的 default 属于关键字。string 和 len 是预定义标识符，可以在局部使用。nil 也可以当做变量使用，不过不建议写这样的代码，可读性不好，小心被接手你代码的人胖揍。

```
var nil = new(int)

func main() {
    var p *int
    if p == nil {
        fmt.Println("p is nil")
    } else {
        fmt.Println("p is not nil")
    }
}
```

第九十三天

第九十三天

1.关于 main() 函数，下面说法正确的是？

- A.不能带参数；
- B.不能定义返回值；
- C.所在的包必须为 main 包；
- D.可以使用 flag 包来获取和解析命令行参数；

参考答案及解析：ABCD。

2.下面代码能编译通过吗？请简要说明。

```
type User struct {
    Name string
}

func (u *User) SetName(name string) {
    u.Name = name
    fmt.Println(u.Name)
}

type Employee User

func main() {
    employee := new(Employee)
    employee.SetName("Jack")
}
```

参考答案及解析：编译不通过。当使用 type 声明一个新类型，它不会继承原有类型的方法集。

第九十二天

第九十二天

1.下面代码输出什么？

```
var x int

func init() {
    x++
}

func main() {
    init()
    fmt.Println(x)
}
```

参考答案及解析：编译失败。init() 函数不能被其他函数调用，包括 main() 函数。

2.min() 函数是求两个数之间的较小值，能否在 该函数中添加一行代码将其功能补全。

```
func min(a int, b uint) {
    var min = 0
    fmt.Printf("The min of %d and %d is %d\n", a, b, min)
}

func main() {
   min(1225, 256)
}
```

参考答案即解析：利用 copy() 函数的功能：切片复制，并且返回两者长度的较小值。

```
func min(a int, b uint) {
    var min = 0
    min = copy(make([]struct{},a),make([]struct{},b))
    fmt.Printf("The min of %d and %d is %d\n", a, b, min)
}

func main() {
    min(1225, 256)
}
```

第九十一天

第九十一天

1.下面两段代码能否编译通过？请简要说明。

第一段：

```
func f() {}
func f() {}
func main() {}
```

第二段：

```
func init(){}
func init(){}
func main() {}
```

参考答案及解析：第二段代码能通过编译。除 init() 函数之外，一个包内不允许有其他同名函数。

2.下面代码有什么问题？请指出。

```
func (m map[string]string) Set(key string, value string) {
    m[key] = value
}

func main() {
    m := make(map[string]string)
    m.Set("A", "One")
}
```

参考答案及解析：Unnamed Type 不能作为方法的接收者。昨天我们讲过 Named Type 与 Unamed Type 的区别，就用 Named Type 来修复下代码：

```
type User map[string]string

func (m User) Set(key string, value string) {
    m[key] = value
}

func main() {
    m := make(User)
    m.Set("A", "One")
}
```

第九十天

第九十天

1.下面代码能通过编译吗？

```
type T int

func F(t T) {}

func main() {
    var q int
    F(q)
}
```

2.下面代码能通过编译吗？请简要说明。

```
type T []int

func F(t T) {}

func main() {
    var q []int
    F(q)
}
```

我们将这两道题目放到一块做一个解析，第一题不能通过编译，第二题可以通过编译。我们知道不同类型的值是不能相互赋值的，即使底层类型一样，所以第一题编译不通过；对于底层类型相同的变量可以相互赋值还有一个重要的条件，即至少有一个不是有名类型（named type）。

这是 Go 语言规范手册的原文：

```
"x's type V and T have identical underlying types and at least one of V or T is not a named type."
```

Named Type 有两类：

- 内置类型，比如 int, int64, float, string, bool 等；
- 使用关键字 type 声明的类型；

Unnamed Type 是基于已有的 Named Type 组合一起的类型，例如：struct{}、[]string、interface{}、map[string]bool 等。

第八十九天

第八十九天

1.下面代码能编译通过吗？请简要说明。

```
func main() {
    v := []int{1, 2, 3}
    for i, n := 0, len(v); i < n; i++ {
        v = append(v, i)
    }
    fmt.Println(v)
}
```

参考答案及解析：能编译通过，输出 [1 2 3 0 1 2]。for 循环开始的时候，终止条件只会计算一次。

2.下面代码输出什么？

```
type P *int
type Q *int

func main() {
    var p P = new(int)
    *p += 8
    var x *int = p
    var q Q = x
    *q++
    fmt.Println(*p, *q)
}
```

- A.8 8
- B.8 9
- C.9 9

参考答案及解析：C。指针变量指向相同的地址。

第八十八天

第八十八天

1.下面这段代码能通过编译吗？请简要说明。

```
func main() {
    m := make(map[string]int)
    m["foo"]++
    fmt.Println(m["foo"])
}
```

参考答案及解析：能通过编译。

上面的代码可以理解成：

```
func main() {
    m := make(map[string]int)
    v := m["foo"]
    v++
    m["foo"] = v
    fmt.Println(m["foo"])
}
```

2.下面的代码输出什么，请简要说明？

```
func Foo() error {
    var err *os.PathError = nil
    // ...
    return err
}

func main() {
    err := Foo()
    fmt.Println(err)
    fmt.Println(err == nil)
}
```

参考答案及解析：nil false。知识点：接口值与 nil 值。只有在值和动态类型都为 nil 的情况下，接口值才为 nil。Foo() 函数返回的 err 变量，值为 nil、动态类型为 `*os.PathError` ，与 nil（值为 nil，动态类型为 nil）显然是不相等。我们可以打印下变量 err 的详情：

```
fmt.Printf("%#v\n",err)    // (*os.PathError)(nil)
```

一个更合适的解决办法：

```
func Foo() (err error) {
    // ...
    return
}
```

```
func main() {
    err := Foo()
    fmt.Println(err)
    fmt.Println(err == nil)
}
```

第八十七天

第八十七天

1.关于协程，下面说法正确是()

- A.协程和线程都可以实现程序的并发执行；
- B.线程比协程更轻量级；
- C.协程不存在死锁问题；
- D.通过 channel 来进行协程间的通信；

参考答案及解析：AD。

2.在数学里面，有著名的勾股定理：

$$a^2 + b^2 = c^2$$

例如，有我们熟悉的组合（3，4，5）、（6、8、10）等。在 Go 语言中，下面代码输出 true：

```
    fmt.Println(3^2+4^2 == 5^2) // true
```

问题来了，下面代码输出什么，请简要说明。

```
func main() {
    fmt.Println(6^2+8^2 == 10^2)
}
```

参考答案及解析：false。在 Go 语言里面，^ 作为二元运算符时表示按位异或：对应位，相同为 0，相异为 1。所以第一段代码输出 true 是因为：

```
0011 ^ 0010 == 0001    (3^2 == 1)
0100 ^ 0010 == 0110    (4^2 == 6)
0101 ^ 0010 == 0111    (5^2 == 7)
```

1+6=7，这当然是相等的。你来试试分解下第二段代码的数学表达式。

第八十六天

第八十六天

1.n 是秒数，下面代码输出什么？

```
func main() {
    n := 43210
    fmt.Println(n/60*60, " hours and ", n%60*60, " seconds")
}
```

参考答案及解析：43200 hours and 600 seconds。知识点：运算符优先级。算术运算符 `*、/` 和 % 的优先级相同，从左向右结合。

修复代码如下：

```
func main() {
    n := 43210
    fmt.Println(n/(60*60), "hours and", n%(60*60), "seconds")
}
```

2.下面代码输出什么，为什么？

```
const (
    Century = 100
    Decade  = 010
    Year    = 001
)

func main() {
    fmt.Println(Century + 2*Decade + 2*Year)
}
```

参考答案及解析：118。知识点：进制数。Go 语言里面，八进制数以 0 开头，十六进制数以 0x 开头，所以 Decade 表示十进制的 8。

第八十五天

第八十五天

1.下面这段代码输出什么？请简要说明。

```
func main() {
    fmt.Println(strings.TrimRight("ABBA", "BA"))
}
```

参考答案及解析：输出空字符。这是一个大多数人遇到的坑，TrimRight() 会将第二个参数字符串里面所有的字符拿出来处理，只要与其中任何一个字符相等，便会将其删除。想正确地截取字符串，可以参考 TrimSuffix() 函数。

2.下面代码输出什么？

```
func main() {
    var src, dst []int
    src = []int{1, 2, 3}
    copy(dst, src)
    fmt.Println(dst)
}
```

参考答案及解析：输出 []。知识点：拷贝切片。copy(dst, src) 函数返回 len(dst)、len(src) 之间的最小值。如果想要将 src 完全拷贝至 dst，必须给 dst 分配足够的内存空间。

修复代码：

```
func main() {
    var src, dst []int
    src = []int{1, 2, 3}
    dst = make([]int, len(src))
    n := copy(dst, src)
    fmt.Println(n,dst)
}
```

或者直接使用 append()

```
func main() {
    var src, dst []int
    src = []int{1, 2, 3}
    dst = append(dst, src...)
    fmt.Println("dst:", dst)
}
```

详情请参考《非懂不可的Slice（二）》

第八十四天

第八十四天

1.函数执行时，如果由于 panic 导致了异常，则延迟函数不会执行。这一说法是否正确？

- A. true
- B. false

参考答案及解析：B。
由 panic 引发异常以后，程序停止执行，然后调用延迟函数（defer），就像程序正常退出一样。

2.下面代码输出什么？

```
func main() {
    a := [3]int{0, 1, 2}
    s := a[1:2]

    s[0] = 11
    s = append(s, 12)
    s = append(s, 13)
    s[0] = 21

    fmt.Println(a)
    fmt.Println(s)
}
```

参考答案及解析：

输出：

```
    [0 11 12]
    [21 12 13]
```

详情请参考《非懂不可的Slice（二）》

第八十三天

第八十三天

1.同级文件的包名不允许有多个，是否正确？

- A. true
- B. false

参考答案及解析：A。一个文件夹下只能有一个包，可以多个.go文件，但这些文件必须属于同一个包。

2.下面的代码有什么问题，请说明。

```
type data struct {
    name string
}

func (p *data) print() {
    fmt.Println("name:", p.name)
}

type printer interface {
    print()
}

func main() {
    d1 := data{"one"}
    d1.print()

    var in printer = data{"two"}
    in.print()
}
```

参考答案及解析：编译报错。

```
    cannot use data literal (type data) as type printer in assignment:
    data does not implement printer (print method has pointer receiver)
```

结构体类型 data 没有实现接口 printer。知识点：接口。

第八十二天

第八十二天

1.下面这段代码输出什么？

```
func main() {
    count := 0
    for i := range [256]struct{}{} {
        m, n := byte(i), int8(i)
        if n == -n {
            count++
        }
        if m == -m {
            count++
        }
    }
    fmt.Println(count)
}
```

参考答案及解析：4。知识点：数值溢出。当 i 的值为 0、128 是会发生相等情况，注意 byte 是 uint8 的别名。

引自《Go语言101》

2.下面代码输出什么？

```
const (
    azero = iota
    aone  = iota
)

const (
    info  = "msg"
    bzero = iota
    bone  = iota
)

func main() {
    fmt.Println(azero, aone)
    fmt.Println(bzero, bone)
}
```

参考答案及解析：0 1 1 2。知识点：iota 的使用。这道题易错点在 bzero、bone 的值，在一个常量声明代码块中，如果 iota 没出现在第一行，则常量的初始值就是非 0 值。

第八十一天

第八十一天

1.下面的代码输出什么？

```
func main() {
    var a []int = nil
    a, a[0] = []int{1, 2}, 9
    fmt.Println(a)
}
```

参考答案即解析：运行时错误。知识点：多重赋值。

多重赋值分为两个步骤，有先后顺序：

- 计算等号左边的索引表达式和取址表达式，接着计算等号右边的表达式；
- 赋值；

2.下面代码中的指针 p 为野指针，因为返回的栈内存在函数结束时会被释放？

```
type TimesMatcher struct {
    base int
}

func NewTimesMatcher(base int) *TimesMatcher  {
    return &TimesMatcher{base:base}
}

func main() {
    p := NewTimesMatcher(3)
    fmt.Println(p)
}
```

- A. false
- B. true

参考答案及解析：A。Go语言的内存回收机制规定，只要有一个指针指向引用一个变量，那么这个变量就不会被释放（内存逃逸），因此在 Go 语言中返回函数参数或临时变量是安全的。

第八十天

第八十天

1.定义一个包内全局字符串变量，下面语法正确的是？

- A. var str string
- B. str := “”
- C. str = “”
- D. var str = “”

参考答案及解析：AD。全局变量要定义在函数之外，而在函数之外定义的变量只能用 var 定义。短变量声明 := 只能用于函数之内。

2.下面的代码有什么问题？

```
func main() {

    wg := sync.WaitGroup{}

    for i := 0; i < 5; i++ {
        go func(wg sync.WaitGroup, i int) {
            wg.Add(1)
            fmt.Printf("i:%d\n", i)
            wg.Done()
        }(wg, i)
    }

    wg.Wait()

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

参考答案及解析：知识点：WaitGroup 的使用。存在两个问题：

- 在协程中使用 wg.Add()；
- 使用了 sync.WaitGroup 副本；

修复代码：

```
func main() {

    wg := sync.WaitGroup{}

    for i := 0; i < 5; i++ {
        wg.Add(1)
        go func(i int) {
            fmt.Printf("i:%d\n", i)
            wg.Done()
        }(i)
    }
```

```
    wg.Wait()

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

或者：

```
func main() {

    wg := &sync.WaitGroup{}

    for i := 0; i < 5; i++ {
        wg.Add(1)
        go func(wg *sync.WaitGroup,i int) {
            fmt.Printf("i:%d\n", i)
            wg.Done()
        }(wg,i)
    }

    wg.Wait()

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

第七十九天

第七十九天

1.interface{} 是可以指向任意对象的 Any 类型，是否正确？

- A. false
- B. true

参考答案及解析：B。

2.下面的代码有什么问题？

```
type ConfigOne struct {
    Daemon string
}

func (c *ConfigOne) String() string {
    return fmt.Sprintf("print: %v", c)
}

func main() {
    c := &ConfigOne{}
    c.String()
}
```

参考答案及解析：无限递归循环，栈溢出。知识点：类型的 String() 方法。如果类型定义了 String() 方法，使用 Printf()、Print() 、 Println() 、 Sprintf() 等格式化输出时会自动使用 String() 方法。

第七十八天

第七十八天

1.关于 switch 语句，下面说法正确的是？

- A. 单个 case 中，可以出现多个结果选项；
- B. 需要使用 break 来明确退出一个 case;
- C. 只有在 case 中明确添加 fallthrought 关键字，才会继续执行紧跟的下一个 case;
- D. 条件表达式必须为常量或者整数；

参考答案及解析：AC。

2.下面代码能编译通过吗？可以的话，输出什么？

```
func alwaysFalse() bool {
    return false
}

func main() {
    switch alwaysFalse()
    {
    case true:
        println(true)
    case false:
        println(false)
    }
}
```

参考答案及解析：可以编译通过，输出：true。知识点：Go 代码断行规则。

详情请查看：https://gfw.go101.org/article/line-break-rules.html

第七十七天

第七十七天

1.关于 cap 函数适用下面哪些类型？

- A. 数组；
- B. channel;
- C. map；
- D. slice；

参考答案即解析：ABD。cap() 函数的作用：

- arry 返回数组的元素个数；
- slice 返回 slice 的最大容量；
- channel 返回 channel 的容量；

2.下面代码输出什么？

```
func hello(num ...int) {
    num[0] = 18
}

func Test13(t *testing.T) {
    i := []int{5, 6, 7}
    hello(i...)
    fmt.Println(i[0])
}

func main() {
    t := &testing.T{}
    Test13(t)
}
```

- A. 18
- B. 5
- C. Compilation error

参考答案及解析：A。可变函数是指针传递。

第七十六天

第七十六天

1.下面这段代码输出什么？

```
type S1 struct{}

func (s1 S1) f() {
    fmt.Println("S1.f()")
}
func (s1 S1) g() {
    fmt.Println("S1.g()")
}

type S2 struct {
    S1
}

func (s2 S2) f() {
    fmt.Println("S2.f()")
}

type I interface {
    f()
}

func printType(i I) {

    fmt.Printf("%T\n", i)
    if s1, ok := i.(S1); ok {
        s1.f()
        s1.g()
    }
    if s2, ok := i.(S2); ok {
        s2.f()
        s2.g()
    }
}

func main() {
    printType(S1{})
    printType(S2{})
}
```

参考答案及解析：

```
main.S1
S1.f()
S1.g()
main.S2
S2.f()
S1.g()
```

知识点：类型断言，结构体嵌套。结构体 S2 嵌套了结构体 S1，S2 自己没有实现 g() ，调用的是 S1 的 g()。

2.下面的代码有什么问题？

```
func main() {
    var wg sync.WaitGroup
    wg.Add(1)
    go func() {
        fmt.Println("1")
        wg.Done()
        wg.Add(1)
    }()
    wg.Wait()
}
```

参考答案及解析：协程里面，使用 wg.Add(1) 但是没有 wg.Done()，导致 panic()。

第七十五天

第七十五天

1.下面的代码有什么问题，请说明？

```
func main() {
    f, err := os.Open("file")
    defer f.Close()
    if err != nil {
        return
    }

    b, err := ioutil.ReadAll(f)
    println(string(b))
}
```

参考答案及解析：defer 语句应该放在 if() 语句后面，先判断 err，再 defer 关闭文件句柄。

修复代码：

```
func main() {
    f, err := os.Open("file")
    if err != nil {
        return
    }
    defer f.Close()

    b, err := ioutil.ReadAll(f)
    println(string(b))
}
```

2.下面代码输出什么，为什么？

```
func f() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Printf("recover:%#v", r)
        }
    }()
    panic(1)
    panic(2)
}

func main() {
    f()
}
```

参考答案及解析：recover:1。知识点：panic、recover()。当程序 panic 时就不会往下执行，可以使用 recover() 捕获 panic 的内容。

第七十四天

第七十四天

1.下面代码有什么问题，请说明？

```
func main() {
    runtime.GOMAXPROCS(1)

    go func() {
        for i:=0;i<10 ;i++  {
            fmt.Println(i)
        }
    }()

    for {}
}
```

参考答案及解析：for {} 独占 CPU 资源导致其他 Goroutine 饿死。

可以通过阻塞的方式避免 CPU 占用，修复代码：

```
func main() {
    runtime.GOMAXPROCS(1)

    go func() {
        for i:=0;i<10 ;i++  {
            fmt.Println(i)
        }
        os.Exit(0)
    }()

    select {}
}
```

引自《Go语言高级编程》

2.假设 x 已声明，y 未声明，下面 4 行代码哪些是正确的。错误的请说明原因？

```
x, _ := f()  // 1
x, _ = f()  // 2
x, y := f()  // 3
x, y = f()  // 4
```

参考答案及解析：2、3正确。知识点：简短变量声明。使用简短变量声明有几个需要注意的地方：

- 只能用于函数内部；
- 短变量声明语句中至少要声明一个新的变量；

第七十三天

第七十三天

1.下面代码输出什么？

```
func test() []func() {
    var funs []func()
    for i := 0; i < 2; i++ {
        funs = append(funs, func() {
            println(&i, i)
        })
    }
    return funs
}

func main() {
    funs := test()
    for _, f := range funs {
        f()
    }
}
```

参考答案及解析：

```
0xc000018058 2
0xc000018058 2
```

知识点：闭包延迟求值。for 循环局部变量 i，匿名函数每一次使用的都是同一个变量。（说明：i 的地址，输出可能与上面的不一样）。

2.下面的代码能编译通过吗？可以的话输出什么，请说明？

```
var f = func(i int) {
    print("x")
}

func main() {
    f := func(i int) {
        print(i)
        if i > 0 {
            f(i - 1)
        }
    }
    f(10)
}
```

参考答案及解析：10x。这道题一眼看上去会输出 109876543210，其实这是错误的答案，这里不是递归。假设 main() 函数里为 f2()，外面的为 f1()，当声明 f2() 时，调用的是已经完成声明的 f1()。

看下面这段代码你应该会更容易理解一点：

```
var x = 23

func main() {
    x := 2*x - 4
    println(x)    // 输出:42
}
```

第七十二天

第七十二天

1.下面的代码输出什么，请说明。

```
type Slice []int

func NewSlice() Slice {
    return make(Slice, 0)
}
func (s *Slice) Add(elem int) *Slice {
    *s = append(*s, elem)
    fmt.Print(elem)
    return s
}
func main() {
    s := NewSlice()
    defer func() {
        s.Add(1).Add(2)
    }()
    s.Add(3)
}
```

参考答案及解析：312。对比昨天的第二题，本题的 s.Add(1).Add(2) 作为一个整体包在一个匿名函数中，会延迟执行。

2.下面的代码输出什么，请说明？

```
type Orange struct {
    Quantity int
}

func (o *Orange) Increase(n int) {
    o.Quantity += n
}

func (o *Orange) Decrease(n int) {
    o.Quantity -= n
}

func (o *Orange) String() string {
    return fmt.Sprintf("%#v", o.Quantity)
}

func main() {
    var orange Orange
    orange.Increase(10)
    orange.Decrease(5)
    fmt.Println(orange)
}
```

参考答案及解析：{5}。这道题容易忽视的点是，String() 是指针方法，而不是值方法，所以使用 Println() 输出时不会调用到 String() 方法。

可以这样修复：

```
func main() {
    orange := &Orange{}
    orange.Increase(10)
    orange.Decrease(5)
    fmt.Println(orange)
}
```

第七十一天

第七十一天

1.判断题：对变量x的取反操作是 `~x` ?

从参考答案及解析：错。Go 语言的取反操作是 `^` ，它返回一个每个 bit 位都取反的数。作用类似在 C、C#、Java 语言中中符号 `~` ，对于有符号的整数来说，是按照补码进行取反操作的（快速计算方法：对数 a 取反，结果为 -(a+1) ），对于无符号整数来说就是按位取反。

2.下面代码输出什么，请说明原因。

```
type Slice []int

func NewSlice() Slice {
    return make(Slice, 0)
}
func (s *Slice) Add(elem int) *Slice {
    *s = append(*s, elem)
    fmt.Print(elem)
    return s
}
func main() {
    s := NewSlice()
    defer s.Add(1).Add(2)
    s.Add(3)
}
```

参考答案及解析：132。这一题有两点需要注意：1.Add() 方法的返回值依然是指针类型 `*Slice` ，所以可以循环调用方法 Add()；2.defer 函数的参数（包括接收者）是在 defer 语句出现的位置做计算的，而不是在函数执行的时候计算的，所以 s.Add(1) 会先于 s.Add(3) 执行。

第七十天

第七十天

1.关于字符串连接，下面语法正确的是？

- A. str := ‘abc’ + ‘123’
- B. str := “abc” + “123”
- C. str ： = ‘123’ + “abc”
- D. fmt.Sprintf(“abc%d”, 123)

参考答案及解析：BD。知识点：单引号、双引号和字符串连接。在 Go 语言中，双引号用来表示字符串 string，其实质是一个 byte 类型的数组，单引号表示 rune 类型。

2.下面代码能编译通过吗？可以的话，输出什么？

```
func main() {

    println(DeferTest1(1))
    println(DeferTest2(1))
}

func DeferTest1(i int) (r int) {
    r = i
    defer func() {
        r += 3
    }()
    return r
}

func DeferTest2(i int) (r int) {
    defer func() {
        r += i
    }()
    return 2
}
```

参考答案及解析：43。具体解析请看《5 年 Gopher 都不知道的 defer 细节，你别再掉进坑里！》。

第六十九天

第六十九天

1.关于 slice 或 map 操作，下面正确的是？

- A.

```
var s []int
s = append(s,1)
```

- B.

```
var m map[string]int
m["one"] = 1
```

- C.

```
var s []int
s = make([]int, 0)
s = append(s,1)
```

- D.

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

参考答案及解析：ACD。

2.下面代码输出什么？

```
func test(x int) (func(), func()) {
    return func() {
        println(x)
        x += 10
    }, func() {
        println(x)
    }
}

func main() {
    a, b := test(100)
    a()
    b()
}
```

参考答案及解析：100 110。知识点：闭包引用相同变量。

第六十八天

第六十八天

1.下面代码有什么问题吗？

```
func main()  {

    for i:=0;i<10 ;i++  {
    loop:
        println(i)
    }
    goto loop
}
```

参考答案及解析：goto 不能跳转到其他函数或者内层代码。编译报错：

```
    goto loop jumps into block starting at
```

2.下面代码输出什么，请说明。

```
func main() {
    x := []int{0, 1, 2}
    y := [3]*int{}
    for i, v := range x {
        defer func() {
            print(v)
        }()
        y[i] = &v
    }
    print(*y[0], *y[1], *y[2])
}
```

参考答案及解析：22222。知识点：defer()、for-range。for-range 虽然使用的是 :=，但是 v 不会重新声明，可以打印 v 的地址验证下。

第六十七天

第六十七天

1.下面的代码输出什么？

```
type T struct {
    n int
}

func main() {
    ts := [2]T{}
    for i := range ts[:] {
        switch i {
        case 0:
            ts[1].n = 9
        case 1:
            fmt.Print(ts[i].n, " ")
        }
    }
    fmt.Print(ts)
}
```

参考答案及解析：9 [{0} {9}]。知识点：for-range 切片。for-range 切片时使用的是切片的副本，但不会复制底层数组，换句话说，此副本切片与原数组共享底层数组。

2.下面的代码输出什么？

```
type T struct {
    n int
}

func main() {
    ts := [2]T{}
    for i := range ts[:] {
        switch t := &ts[i]; i {
        case 0:
            t.n = 3;
            ts[1].n = 9
        case 1:
            fmt.Print(t.n, " ")
        }
    }
    fmt.Print(ts)
}
```

参考答案及解析：9 [{3} {9}]。知识点：for-range 切片。参考前几道题的解析，这道题的答案应该很明显。

引自《Go语言101》

第六十六天

第六十六天

1.下面的代码输出什么？

```
type T struct {
    n int
}

func main() {
    ts := [2]T{}
    for i, t := range ts {
        switch i {
        case 0:
            t.n = 3
            ts[1].n = 9
        case 1:
            fmt.Print(t.n, " ")
        }
    }
    fmt.Print(ts)
}
```

参考答案及解析：0 [{0} {9}]。知识点：for-range 循环数组。此时使用的是数组 ts 的副本，所以 t.n = 3 的赋值操作不会影响原数组。

2.下面的代码输出什么？

```
type T struct {
    n int
}

func main() {
    ts := [2]T{}
    for i, t := range &ts {
        switch i {
        case 0:
            t.n = 3
            ts[1].n = 9
        case 1:
            fmt.Print(t.n, " ")
        }
    }
    fmt.Print(ts)
}
```

参考答案及解析：9 [{0} {9}]。知识点：for-range 数组指针。for-range 循环中的循环变量 t 是原数组元素的副本。如果数组元素是结构体值，则副本的字段和原数组字段是两个不同的值。

均引自《Go语言101》

第六十五天

第六十五天

1.flag 是 bool 型变量，下面 if 表达式符合编码规范的是？

- A. if flag == 1
- B. if flag
- C. if flag == false
- D. if !flag

参考答案及解析：BCD。

2.下面的代码输出什么，请说明？

```
func main() {
    defer func() {
        fmt.Print(recover())
    }()
    defer func() {
        defer func() {
            fmt.Print(recover())
        }()
        panic(1)
    }()
    defer recover()
    panic(2)
}
```

参考答案及解析：12。相关知识点请看 第64天 题目解析。

第六十四天

第六十四天

1.下面列举的是 recover() 的几种调用方式，哪些是正确的？

- A.

```
func main() {
  recover()
  panic(1)
}
```

- B.

```
func main() {
  defer recover()
  panic(1)
}
```

- C.

```
func main() {
  defer func() {
     recover()
  }()
  panic(1)
}
```

- D.

```
func main() {
  defer func() {
     defer func() {
        recover()
     }()
  }()
  panic(1)
}
```

参考答案及解析：C。recover() 必须在 defer() 函数中直接调用才有效。上面其他几种情况调用都是无效的：直接调用 recover()、在 defer() 中直接调用 recover() 和 defer() 调用时多层嵌套。

2.下面代码输出什么，请说明？

```
func main() {
    defer func() {
        fmt.Print(recover())
    }()
    defer func() {
        defer fmt.Print(recover())
        panic(1)
    }()
```

```
    defer recover()
    panic(2)
}
```

参考答案及解析：21。recover() 必须在 defer() 函数中调用才有效，所以第 9 行代码捕获是无效的。在调用 defer() 时，便会计算函数的参数并压入栈中，所以执行第 6 行代码时，此时便会捕获 panic(2)；此后的 panic(1)，会被上一层的 recover() 捕获。所以输出 21。

引自《Go语言101》

第六十三天

第六十三天

1.下面选项正确的是？

- A. 类型可以声明的函数体内；
- B. Go 语言支持 ++i 或者 –i 操作；
- C. nil 是关键字；
- D. 匿名函数可以直接赋值给一个变量或者直接执行；

参考答案及解析：AD。

2.下面的代码输出什么？

```
func F(n int) func() int {
    return func() int {
        n++
        return n
    }
}

func main() {
    f := F(5)
    defer func() {
        fmt.Println(f())
    }()
    defer fmt.Println(f())
    i := f()
    fmt.Println(i)
}
```

参考答案及解析：768。知识点：匿名函数、defer()。defer() 后面的函数如果带参数，会优先计算参数，并将结果存储在栈中，到真正执行 defer() 的时候取出。

引自《Go语言101》

第六十二天

第六十二天

1.下面哪一行代码会编译出错，请说明。

```
func main() {
    nil := 123
    fmt.Println(nil)
    var _ map[string]int = nil
}
```

参考答案及解析：第 4 行，当前作用域中，预定义的 nil 被覆盖，此时 nil 是 int 类型值，不能赋值给 map 类型。

2.下面代码输出什么？

```
func main() {
    var x int8 = -128
    var y = x/-1
    fmt.Println(y)
}
```

参考答案及解析：-128。因为溢出。

第六十一天

第六十一天

1.下面这段代码输出什么？

```
func main() {
    var k = 1
    var s = []int{1, 2}
    k, s[k] = 0, 3
    fmt.Println(s[0] + s[1])
}
```

参考答案及解析：4。知识点：多重赋值。

- 多重赋值分为两个步骤，有先后顺序：
- 计算等号左边的索引表达式和取址表达式，接着计算等号右边的表达式；

赋值：

所以本例，会先计算 s[k]，等号右边是两个表达式是常量，所以赋值运算等同于 k, s[1] = 0, 3。

2.下面代码输出什么？

```
func main() {
    var k = 9
    for k = range []int{} {}
    fmt.Println(k)

    for k = 0; k < 3; k++ {
    }
    fmt.Println(k)


    for k = range (*[3]int)(nil) {
    }
    fmt.Println(k)
}
```

参考答案及解析：932。

第六十天

第六十天

1.下面哪一行代码会 panic，请说明原因？

```
package main

type T struct{}

func (*T) foo() {
}

func (T) bar() {
}

type S struct {
  *T
}

func main() {
  s := S{}
  _ = s.foo
  s.foo()
  _ = s.bar
}
```

参考答案及解析：第 19 行，因为 s.bar 将被展开为 `(*s.T).bar` ，而 s.T 是个空指针，解引用会 panic。

可以使用下面代码输出 s：

```
func main() {
    s := S{}
    fmt.Printf("%#v",s)    // 输出：main.S{T:(*main.T)(nil)}
}
```

引自：《Go语言101》

2.下面的代码有什么问题？

```
type data struct {
    sync.Mutex
}

func (d data) test(s string)  {
    d.Lock()
    defer d.Unlock()

    for i:=0;i<5 ;i++  {
        fmt.Println(s,i)
        time.Sleep(time.Second)
    }
}
```

```
func main() {

    var wg sync.WaitGroup
    wg.Add(2)
    var d data

    go func() {
        defer wg.Done()
        d.test("read")
    }()

    go func() {
        defer wg.Done()
        d.test("write")
    }()

    wg.Wait()
}
```

参考答案及解析：锁失效。将 Mutex 作为匿名字段时，相关的方法必须使用指针接收者，否则会导致锁机制失效。

修复代码：

```
func (d *data) test(s string)  {     // 指针接收者
    d.Lock()
    defer d.Unlock()

    for i:=0;i<5 ;i++  {
        fmt.Println(s,i)
        time.Sleep(time.Second)
    }
}
```

或者可以通过嵌入 `*Mutex` 来避免复制的问题，但需要初始化。

```
type data struct {
    *sync.Mutex     // *Mutex
}

func (d data) test(s string) {    // 值方法
    d.Lock()
    defer d.Unlock()

    for i := 0; i < 5; i++ {
        fmt.Println(s, i)
        time.Sleep(time.Second)
    }
}

func main() {

    var wg sync.WaitGroup
    wg.Add(2)

    d := data{new(sync.Mutex)}   // 初始化

    go func() {
        defer wg.Done()
```

```
        d.test("read")
    }()

    go func() {
        defer wg.Done()
        d.test("write")
    }()

    wg.Wait()
}
```

引自：《Go 语言学习笔记》· 同步

第五十九天

第五十九天

1.下面的代码输出什么？

```
type N int

func (n *N) test(){
    fmt.Println(*n)
}

func main()  {
    var n N = 10
    p := &n

    n++
    f1 := n.test

    n++
    f2 := p.test

    n++
    fmt.Println(n)

    f1()
    f2()
}
```

参考答案及解析：13 13 13。知识点：方法值。当目标方法的接收者是指针类型时，那么被复制的就是指针。

引自：《Go语言学习笔记》· 方法

2.下面哪一行代码会 panic，请说明原因？

```
package main

func main() {
  var m map[int]bool // nil
  _ = m[123]
  var p *[5]string // nil
  for range p {
    _ = len(p)
  }
  var s []int // nil
  _ = s[:]
  s, s[0] = []int{1, 2}, 9
}
```

参考答案及解析：第 12 行。因为左侧的 s[0] 中的 s 为 nil。

引自：《Go语言101》

第五十八天

第五十八天

1.下面的代码输出什么？

```
type T struct {
    x int
    y *int
}

func main() {

    i := 20
    t := T{10,&i}

    p := &t.x

    *p++
    *p--

    t.y = p

    fmt.Println(*t.y)
}
```

参考答案及解析：10。知识点：运算符优先级。如下规则：递增运算符 ++ 和递减运算符 – 的优先级低于解引用运算符 * 和取址运算符 &，解引用运算符和取址运算符的优先级低于选择器 . 中的属性选择操作符。

2.下面哪一行代码会 panic，请说明原因？

```
package main

func main() {
    x := make([]int, 2, 10)
    _ = x[6:10]
    _ = x[6:]
    _ = x[2:]
}
```

参考答案：第 6 行，截取符号 [i:j]，如果 j 省略，默认是原切片或者数组的长度，x 的长度是 2，小于起始下标 6 ，所以 panic。

两题均引自：《Go语言101》

第五十七天

第五十七天

1.下面哪一行代码会 panic，请说明原因？

```
package main

func main() {
  var x interface{}
  var y interface{} = []int{3, 5}
  _ = x == x
  _ = x == y
  _ = y == y
}
```

参考答案及解析：第 8 行。因为两个比较值的动态类型为同一个不可比较类型。

2.下面的代码输出什么？

```
var o = fmt.Print

func main() {
    c := make(chan int, 1)
    for range [3]struct{}{} {
        select {
        default:
            o(1)
        case <-c:
            o(2)
            c = nil
        case c <- 1:
            o(3)
        }
    }
}
```

参考答案及解析：321。第一次循环，写操作已经准备好，执行 o(3)，输出 3；第二次，读操作准备好，执行 o(2)，输出 2 并将 c 赋值为 nil；第三次，由于 c 为 nil，走的是 default 分支，输出 1。

两题均引自：《Go语言101》

第五十六天

第五十六天

1.下面的代码有什么问题？

```
package main

import "fmt"

func main() {
    s := make([]int, 3, 9)
    fmt.Println(len(s))
    s2 := s[4:8]
    fmt.Println(len(s2))
}
```

参考答案及解析：代码没问题，输出 3 4。从一个基础切片派生出的子切片的长度可能大于基础切片的长度。假设基础切片是 baseSlice，使用操作符 [low,high]，有如下规则：0 <= low <= high <= cap(baseSlice)，只要上述满足这个关系，下标 low 和 high 都可以大于 len(baseSlice)。

引自：《Go语言101》

2.下面代码输出什么？

```
type N int

func (n N) test(){
    fmt.Println(n)
}

func main()  {
    var n N = 10
    p := &n

    n++
    f1 := n.test

    n++
    f2 := p.test

    n++
    fmt.Println(n)

    f1()
    f2()
}
```

参考答案及解析：13 11 12。知识点：方法值。当指针值赋值给变量或者作为函数参数传递时，会立即计算并复制该方法执行所需的接收者对象，与其绑定，以便在稍后执行时，能隐式第传入接收者参数。

引自：《Go语言学习笔记》· 方法

第五十五天

第五十五天

1.关于 channel 下面描述正确的是？

- A. close() 可以用于只接收通道；
- B. 单向通道可以转换为双向通道；
- C. 不能在单向通道上做逆向操作（例如：只发送通道用于接收）；

参考答案及解析：C。

2.下面的代码有什么问题？

```
type T struct {
    n int
}

func getT() T {
    return T{}
}

func main() {
    getT().n = 1
}
```

参考答案及解析：编译错误：

```
    cannot assign to getT().n
```

直接返回的 `T{}` 无法寻址，不可直接赋值。

修复代码：

```
type T struct {
    n int
}

func getT() T {
    return T{}
}

func main() {
    t := getT()
    p := &t.n    // <=> p = &(t.n)
    *p = 1
    fmt.Println(t.n)
}
```

第五十四天

第五十四天

1.下面的代码有什么问题？

```
type N int
func (n N) value(){
    n++
    fmt.Printf("v:%p,%v\n",&n,n)
}

func (n *N) pointer(){
    *n++
    fmt.Printf("v:%p,%v\n",n,*n)
}


func main() {

    var a N = 25

    p := &a
    p1 := &p

    p1.value()
    p1.pointer()
}
```

参考答案及解析：编译错误：

```
calling method value with receiver p1 (type **N) requires explicit dereference
calling method pointer with receiver p1 (type **N) requires explicit dereference
```

不能使用多级指针调用方法。

2.下面的代码输出什么？

```
type N int

func (n N) test(){
    fmt.Println(n)
}

func main()  {
    var n N = 10
    fmt.Println(n)

    n++
    f1 := N.test
    f1(n)

    n++
    f2 := (*N).test
```

```
    f2(&n)
}
```

参考答案及解析：10 11 12。知识点：方法表达式。通过类型引用的方法表达式会被还原成普通函数样式，接收者是第一个参数，调用时显示传参。类型可以是 T 或 `*T` ，只要目标方法存在于该类型的方法集中就可以。

还可以直接使用方法表达式调用：

```
func main()  {
    var n N = 10

    fmt.Println(n)

    n++
    N.test(n)

    n++
    (*N).test(&n)
}
```

引自：《Go语言学习笔记》· 方法

第五十三天

第五十三天

1.关于 channel 下面描述正确的是？

- A. 向已关闭的通道发送数据会引发 panic；
- B. 从已关闭的缓冲通道接收数据，返回已缓冲数据或者零值；
- C. 无论接收还是接收，nil 通道都会阻塞；

参考答案及解析：ABC。

2.下面的代码有几处问题？请详细说明。

```
type T struct {
    n int
}

func (t *T) Set(n int) {
    t.n = n
}
func getT() T {
    return T{}
}

func main() {
    getT().Set(1)
}
```

参考答案及解析：有两处问题：

- 1.直接返回的 T{} 不可寻址；
- 2.不可寻址的结构体不能调用带结构体指针接收者的方法；

修复代码：

```
type T struct {
    n int
}

func (t *T) Set(n int) {
    t.n = n
}

func getT() T {
    return T{}
}

func main() {
```

```
    t := getT()
    t.Set(2)
    fmt.Println(t.n)
}
```

第五十二天

第五十二天

1.下面的代码有什么问题？

```
type X struct {}
func (x *X) test()  {
   println(x)
}
func main() {
    var a *X
    a.test()
    X{}.test()
}
```

参考答案及解析：X{} 是不可寻址的，不能直接调用方法。知识点：在方法中，指针类型的接收者必须是合法指针（包括 nil）,或能获取实例地址。

修复代码：

```
func main() {
    var a *X
    a.test()    // 相当于 test(nil)
    var x = X{}
    x.test()
}
```

引自：《Go语言学习笔记》· 方法

2.下面代码有什么不规范的地方吗？

```
func main() {
    x := map[string]string{"one":"a","two":"","three":"c"}
    if v := x["two"]; v == "" {
        fmt.Println("no entry")
    }
}
```

参考答案及解析：检查 map 是否含有某一元素，直接判断元素的值并不是一种合适的方式。最可靠的操作是使用访问 map 时返回的第二个值。

修复代码如下：

```
func main() {
    x := map[string]string{"one":"a","two":"","three":"c"}
    if _,ok := x["two"]; !ok {
        fmt.Println("no entry")
    }
}
```

引自：http://devs.cloudimmunity.com/gotchas-and-common-mistakes-in-go-golang/index.html

第五十一天

第五十一天

1.下面的代码能否正确输出？

```
func main() {
    var fn1 = func() {}
    var fn2 = func() {}
    if fn1 != fn2 {
        println("fn1 not equal fn2")
    }
}
```

参考答案及解析：编译错误

```
    invalid operation: fn1 != fn2 (func can only be compared to nil)
```

函数只能与 nil 比较。

2.下面代码输出什么？

```
type T struct {
    n int
}
func main() {
    m := make(map[int]T)
    m[0].n = 1
    fmt.Println(m[0].n)
}
```

- A. 1
- B. compilation error

参考答案及解析：B。编译错误：

```
    cannot assign to struct field m[0].n in map
```

map[key]struct 中 struct 是不可寻址的，所以无法直接赋值。

修复代码：

```
type T struct {
    n int
}
func main() {
    m := make(map[int]T)
    t := T{1}
    m[0] = t
```

```
    fmt.Println(m[0].n)
}
```

第五十天

第五十天

1.下面这段代码输出什么？

```
type T struct {
    ls []int
}
func foo(t T) {
    t.ls[0] = 100
}
func main() {
    var t = T{
        ls: []int{1, 2, 3},
    }
    foo(t)
    fmt.Println(t.ls[0])
}
```

- A. 1
- B. 100
- C. compilation error

参考答案及解析：B。调用 foo() 函数时虽然是传值，但 foo() 函数中，字段 ls 依旧可以看成是指向底层数组的指针。

2.下面代码输出什么？

```
func main() {
    isMatch := func(i int) bool {
        switch(i) {
        case 1:
        case 2:
            return true
        }
        return false
    }
    fmt.Println(isMatch(1))
    fmt.Println(isMatch(2))
}
```

参考答案及解析：false true。Go 语言的 switch 语句虽然没有”break”，但如果 case 完成程序会默认 break，可以在 case 语句后面加上关键字 fallthrough，这样就会接着走下一个 case 语句（不用匹配后续条件表达式）。或者，利用 case 可以匹配多个值的特性。

修复代码：

```
func main() {
    isMatch := func(i int) bool {
        switch(i) {
```

```
        case 1:
            fallthrough
        case 2:
            return true
        }
        return false
    }
    fmt.Println(isMatch(1))     // true
    fmt.Println(isMatch(2))     // true
    match := func(i int) bool {
        switch(i) {
        case 1,2:
            return true
        }
        return false
    }
    fmt.Println(match(1))       // true
    fmt.Println(match(2))       // true
}
```

第四十九天

第四十九天

1.下面代码输出什么？

```
func main() {
    var ch chan int
    select {
    case v, ok := <-ch:
        println(v, ok)
    default:
        println("default")
    }
}
```

参考答案及解析：default。ch 为 nil，读写都会阻塞。

2.下面这段代码输出什么？

```
type People struct {
    name string `json:"name"`
}
func main() {
    js := `{
        "name":"seekload"
    }`
    var p People
    err := json.Unmarshal([]byte(js), &p)
    if err != nil {
        fmt.Println("err: ", err)
        return
    }
    fmt.Println(p)
}
```

参考答案及解析：输出 **{}**。知识点：结构体访问控制，因为 name 首字母是小写，导致其他包不能访问，所以输出为空结构体。修复代码：

```
    type People struct {
        Name string `json:"name"`
    }
```

第四十八天

第四十八天

1.下面代码有什么问题？

```
type foo struct {
    bar int
}

func main() {
    var f foo
    f.bar, tmp := 1, 2
}
```

参考答案及解析：编译错误：

```
    non-name f.bar on left side of :=
```

:= 操作符不能用于结构体字段赋值。

2.下面的代码输出什么？

```
func main() {
    fmt.Println(~2)
}
```

参考答案及解析：编译错误。

```
    invalid character U+007E '~'
```

很多语言都是采用 ~ 作为按位取反运算符，Go 里面采用的是 ^ 。按位取反之后返回一个每个 bit 位都取反的数，对于有符号的整数来说，是按照补码进行取反操作的（快速计算方法：对数 a 取反，结果为 -(a+1) ），对于无符号整数来说就是按位取反。例如：

```
func main() {
    var a int8 = 3
    var b uint8 = 3
    var c int8 = -3

    fmt.Printf("^%b=%b %d\n", a, ^a, ^a) // ^11=-100 -4
    fmt.Printf("^%b=%b %d\n", b, ^b, ^b) // ^11=11111100 252
    fmt.Printf("^%b=%b %d\n", c, ^c, ^c) // ^-11=10 2
}
```

另外需要注意的是，如果作为二元运算符，^ 表示按位异或，即：对应位相同为 0，相异为 1。例如：

```
func main() {
    var a int8 = 3
    var c int8 = 5
```

```
    fmt.Printf("a: %08b\n", a)
    fmt.Printf("c: %08b\n", c)
    fmt.Printf("a^c: %08b\n", a ^ c)
}
```

给大家重点介绍下这个操作符 `&^` ，按位置零，例如： `z = x &^ y` ，表示如果 y 中的 bit 位为 1，则 z 对应 bit 位为 0，否则 z 对应 bit 位等于 x 中相应的 bit 位的值。

不知道大家发现没有，我们还可以这样理解或操作符 `|` ，表达式 `z = x | y` ，如果 y 中的 bit 位为 1，则 z 对应 bit 位为 1，否则 z 对应 bit 位等于 x 中相应的 bit 位的值，与 `&^` 完全相反。

```
    var x uint8 = 214
    var y uint8 = 92
    fmt.Printf("x: %08b\n", x)
    fmt.Printf("y: %08b\n", y)
    fmt.Printf("x | y: %08b\n", x | y)
    fmt.Printf("x &^ y: %08b\n", x &^ y)
```

输出：

```
    x: 11010110
    y: 01011100
    x | y: 11011110
    x &^ y: 10000010
```

第四十七天

第四十七天

1.下面的代码有什么问题？

```
func main() {
    data := []int{1,2,3}
    i := 0
    ++i
    fmt.Println(data[i++])
}
```

参考答案及解析：对于自增、自减，需要注意：

- 自增、自减不在是运算符，只能作为独立语句，而不是表达式；
- 不像其他语言，Go 语言中不支持 ++i 和 --i 操作；

表达式通常是求值代码，可作为右值或参数使用。而语句表示完成一个任务，比如 if、for 语句等。表达式可作为语句使用，但语句不能当做表达式。

修复代码：

```
func main() {
    data := []int{1,2,3}
    i := 0
    i++
    fmt.Println(data[i])
}
```

2.下面代码最后一行输出什么？请说明原因。

```
func main() {
    x := 1
    fmt.Println(x)
    {
        fmt.Println(x)
        i,x := 2,2
        fmt.Println(i,x)
    }
    fmt.Println(x)  // print ?
}
```

参考答案及解析：输出1。知识点：变量隐藏。使用变量简短声明符号 := 时，如果符号左边有多个变量，只需要保证至少有一个变量是新声明的，并对已定义的变量尽进行赋值操作。但如果出现作用域之后，就会导致变量隐藏的问题，就像这个例子一样。

这个坑很容易挖，但又很难发现。即使对于经验丰富的 Go 开发者而言，这也是一个非常常见的陷阱。

引自：http://devs.cloudimmunity.com/gotchas-and-common-mistakes-in-go-golang/

第四十六天

第四十六天

1.下面的代码有什么问题？

```
func main() {
    const x = 123
    const y = 1.23
    fmt.Println(x)
}
```

参考答案及解析：编译可以通过。知识点：常量。常量是一个简单值的标识符，在程序运行时，不会被修改的量。不像变量，常量未使用是能编译通过的。

2.下面代码输出什么？

```
const (
    x uint16 = 120
    y
    s = "abc"
    z
)
func main() {
    fmt.Printf("%T %v\n", y, y)
    fmt.Printf("%T %v\n", z, z)
}
```

参考答案及解析：知识点：常量。

输出：

```
    uint16 120
    string abc
```

常量组中如不指定类型和初始化值，则与上一行非空常量右值相同

3.下面代码有什么问题？

```
func main() {
    var x string = nil
    if x == nil {
        x = "default"
    }
}
```

参考答案及解析：将 nil 分配给 string 类型的变量。这是个大多数新手会犯的错误。修复代码：

```
func main() {
    var x string //defaults to "" (zero value)
```

```
    if x == "" {
        x = "default"
    }
}
```

引自：http://devs.cloudimmunity.com/gotchas-and-common-mistakes-in-go-golang/index.html

第四十五天

第四十五天

1.下面代码有什么错误？

```
func main() {
    one := 0
    one := 1
}
```

参考答案及解析：变量重复声明。不能在单独的声明中重复声明一个变量，但在多变量声明的时候是可以的，但必须保证至少有一个变量是新声明的。

修复代码：

```
func main() {
    one := 0
    one, two := 1,2
    one,two = two,one
}
```

引自：http://devs.cloudimmunity.com/gotchas-and-common-mistakes-in-go-golang/

2.下面代码有什么问题？

```
func main() {
    x := []int{
        1,
        2
    }
    _ = x
}
```

参考答案及解析：编译错误，第四行代码没有逗号。用字面量初始化数组、slice 和 map 时，最好是在每个元素后面加上逗号，即使是声明在一行或者多行都不会出错。

修复代码：

```
func main() {
    x := []int{    // 多行
        1,
        2,
    }
    x = x

    y := []int{3,4,} // 一行 no error
    y = y
}
```

引自：http://devs.cloudimmunity.com/gotchas-and-common-mistakes-in-go-golang/index.html

3.下面代码输出什么？

```
func test(x byte)  {
    fmt.Println(x)
}

func main() {
    var a byte = 0x11
    var b uint8 = a
    var c uint8 = a + b
    test(c)
}
```

参考答案及解析：34。与 rune 是 int32 的别名一样，byte 是 uint8 的别名，别名类型无序转换，可直接转换。

第四十四天

第四十四天

1.下面代码有什么问题？

```
func main() {
    m := make(map[string]int,2)
    cap(m)
}
```

参考答案及解析：问题：使用 cap() 获取 map 的容量。1.使用 make 创建 map 变量时可以指定第二个参数，不过会被忽略。2.cap() 函数适用于数组、数组指针、slice 和 channel，不适用于 map，可以使用 len() 返回 map 的元素个数。

引自：http://devs.cloudimmunity.com/gotchas-and-common-mistakes-in-go-golang/index.html

2.下面的代码有什么问题？

```
func main() {
    var x = nil
    _ = x
}
```

参考答案及解析：nil 用于表示 interface、函数、maps、slices 和 channels 的“零值”。如果不指定变量的类型，编译器猜不出变量的具体类型，导致编译错误。

修复代码：

```
func main() {
    var x interface{} = nil
    _ = x
}
```

3.下面代码能编译通过吗？

```
type info struct {
    result int
}
func work() (int,error) {
    return 13,nil
}
func main() {
    var data info
    data.result, err := work()
    fmt.Printf("info: %+v\n",data)
}
```

参考答案及解析：编译失败。

```
    non-name data.result on left side of :=
```

不能使用短变量声明设置结构体字段值，修复代码：

```
func main() {
    var data info
    var err error
    data.result, err = work() //ok
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(data)
}
```

引自：http://devs.cloudimmunity.com/gotchas-and-common-mistakes-in-go-golang/index.html

第四十三天

第四十三天

1.下面的代码有什么问题？

```
import (
    "fmt"
    "log"
    "time"
)
func main() {
}
```

参考答案及解析：导入的包没有被使用。如果引入一个包，但是未使用其中如何函数、接口、结构体或变量的话，代码将编译失败。

如果你真的需要引入包，可以使用下划线操作符， `_` ，来作为这个包的名字，从而避免失败。下划线操作符用于引入，但不使用。

我们还可以注释或者移除未使用的包。

修复代码：

```
import (
    _ "fmt"
    "log"
    "time"
)
var _ = log.Println
func main() {
    _ = time.Now
}
```

引自：http://devs.cloudimmunity.com/gotchas-and-common-mistakes-in-go-golang/

2.下面代码输出什么？

```
func main() {
    x := interface{}(nil)
    y := (*int)(nil)
    a := y == x
    b := y == nil
    _, c := x.(interface{})
    println(a, b, c)
}
```

- A. true true false
- B. false true true
- C. true true true

- D. false true false

参考答案及解析：D。知识点：类型断言。类型断言语法：i.(Type)，其中 i 是接口，Type 是类型或接口。编译时会自动检测 i 的动态类型与 Type 是否一致。但是，如果动态类型不存在，则断言总是失败

3.下面代码有几处错误的地方？请说明原因。

```
func main() {

    var s []int
    s = append(s,1)

    var m map[string]int
    m["one"] = 1
}
```

参考答案及解析：有 1 出错误，不能对 nil 的 map 直接赋值，需要使用 make() 初始化。但可以使用 append() 函数对为 nil 的 slice 增加元素。

修复代码：

```
func main() {
    var m map[string]int
    m = make(map[string]int)
    m["one"] = 1
}
```

第四十二天

第四十二天

1.请指出下面代码的错误？

```
package main

var gvar int

func main() {
    var one int
    two := 2
    var three int
    three = 3

    func(unused string) {
        fmt.Println("Unused arg. No compile error")
    }("what?")
}
```

参考答案及解析：变量 one、two 和 three 声明未使用。知识点：未使用变量。如果有未使用的变量代码将编译失败。但也有例外，函数中声明的变量必须要使用，但可以有未使用的全局变量。函数的参数未使用也是可以的。

如果你给未使用的变量分配了一个新值，代码也还是会编译失败。你需要在某个地方使用这个变量，才能让编译器愉快的编译。

修复代码：

```
func main() {
    var one int
    _ = one

    two := 2
    fmt.Println(two)

    var three int
    three = 3
    one = three

    var four int
    four = four
}
```

另一个选择是注释掉或者移除未使用的变量 。

引自：http://devs.cloudimmunity.com/gotchas-and-common-mistakes-in-go-golang/

2.下面代码输出什么？

```
type ConfigOne struct {
    Daemon string
}
```

```
func (c *ConfigOne) String() string {
    return fmt.Sprintf("print: %v", c)
}

func main() {
    c := &ConfigOne{}
    c.String()
}
```

参考答案及解析：运行时错误。如果类型实现 String() 方法，当格式化输出时会自动使用 String() 方法。上面这段代码是在该类型的 String() 方法内使用格式化输出，导致递归调用，最后抛错。

```
    runtime: goroutine stack exceeds 1000000000-byte limit
    fatal error: stack overflow
```

3.下面代码输出什么？

```
func main() {
    var a = []int{1, 2, 3, 4, 5}
    var r = make([]int, 0)

    for i, v := range a {
        if i == 0 {
            a = append(a, 6, 7)
        }

        r = append(r, v)
    }

    fmt.Println(r)
}
```

参考答案及解析：[1 2 3 4 5]。a 在 for range 过程中增加了两个元素
，len 由 5 增加到 7，但 for range 时会使用 a 的副本 a’ 参与循环，副本的 len 依旧是 5，因此 for range 只会循环 5 次，也就只获取 a 对应的底层数组的前 5 个元素。

第四十一天

第四十一天

1.下面代码编译能通过吗？

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

参考答案及解析：编译错误。

```
syntax error: unexpected semicolon or newline before {
```

Go 语言中，大括号不能放在单独的一行。

正确的代码如下：

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

引自：http://devs.cloudimmunity.com/gotchas-and-common-mistakes-in-go-golang/

2.下面这段代码输出什么？

```
var x = []int{2: 2, 3, 0: 1}

func main() {
    fmt.Println(x)
}
```

参考答案及解析：输出[1 0 2 3]，字面量初始化切片时候，可以指定索引，没有指定索引的元素会在前一个索引基础之上加一，所以输出[1 0 2 3]，而不是[1 3 2]。

3.下面这段代码输出什么？

```
func incr(p *int) int {
    *p++
    return *p
}
func main() {
    v := 1
    incr(&v)
    fmt.Println(v)
}
```

参考答案及解析：2。知识点：指针。p 是指针变量，指向变量 v， `*p++` 操作的意思是取出变量 v 的值并执行加一操作，所以 v 的最终值是 2。

第四十天

第四十天

1.关于select机制，下面说法正确的是?

- A. select机制用来处理异步IO问题；
- B. select机制最大的一条限制就是每个case语句里必须是一个IO操作；
- C. golang在语言级别支持select关键字；
- D. select关键字的用法与switch语句非常类似，后面要带判断条件；

参考答案及解析：ABC。

2.下面的代码有什么问题？

```
func Stop(stop <-chan bool) {
    close(stop)
}
```

参考答案及解析：有方向的 channel 不可以被关闭。

3.下面这段代码存在什么问题？

```
type Param map[string]interface{}

type Show struct {
    *Param
}

func main() {
    s := new(Show)
    s.Param["day"] = 2
}
```

参考答案及解析：存在两个问题：1.map 需要初始化才能使用；2.指针不支持索引。修复代码如下：

```
func main() {
    s := new(Show)
    // 修复代码
    p := make(Param)
    p["day"] = 2
    s.Param = &p
    tmp := *s.Param
    fmt.Println(tmp["day"])
}
```

第三十九天

第三十九天

1.关于无缓冲和有冲突的channel，下面说法正确的是？

- A. 无缓冲的channel是默认的缓冲为1的channel；
- B. 无缓冲的channel和有缓冲的channel都是同步的；
- C. 无缓冲的channel和有缓冲的channel都是非同步的；
- D. 无缓冲的channel是同步的，而有缓冲的channel是非同步的；

参考答案及解析：D。

2.下面代码是否能编译通过？如果通过，输出什么？

```
func Foo(x interface{}) {
    if x == nil {
        fmt.Println("empty interface")
        return
    }
    fmt.Println("non-empty interface")
}
func main() {
    var x *int = nil
   Foo(x)
}
```

参考答案及解析：non-empty interface 考点：interface 的内部结构，我们知道接口除了有静态类型，还有动态类型和动态值，当且仅当动态值和动态类型都为 nil 时，接口类型值才为 nil。这里的 x 的动态类型是 `*int` ，所以 x 不为 nil。

3.下面代码输出什么？

```
func main() {
    ch := make(chan int, 100)
    // A
    go func() {
        for i := 0; i < 10; i++ {
            ch <- i
        }
    }()
    // B
   go func() {
        for {
            a, ok := <-ch
            if !ok {
                fmt.Println("close")
                return
            }
            fmt.Println("a: ", a)
        }
```

```
    }()
    close(ch)
    fmt.Println("ok")
    time.Sleep(time.Second * 10)
}
```

参考答案及解析：程序抛异常。先定义下，第一个协程为 A 协程，第二个协程为 B 协程；当 A 协程还没起时，主协程已经将 channel 关闭了，当 A 协程往关闭的 channel 发送数据时会 panic，panic: send on closed channel。

第三十八天

第三十八天

1.关于异常的触发，下面说法正确的是？

- A. 空指针解析；
- B. 下标越界；
- C. 除数为0；
- D. 调用panic函数；

参考答案及解析：ABCD。

2.下面代码输出什么？

```
func main() {
    x := []string{"a", "b", "c"}
    for v := range x {
        fmt.Print(v)
    }
}
```

参考答案及解析：012。注意区别下面代码段：

```
func main() {
    x := []string{"a", "b", "c"}
    for _, v := range x {
        fmt.Print(v)     //输出 abc
    }
}
```

3.下面这段代码能否编译通过？如果通过，输出什么？

```
type User struct{}
 type User1 User
 type User2 = User

 func (i User1) m1() {
     fmt.Println("m1")
 }
 func (i User) m2() {
     fmt.Println("m2")
}

func main() {
    var i1 User1
    var i2 User2
    i1.m1()
```

```
    i2.m2()
}
```

参考答案及解析：能，输出m1 m2，第 2 行代码基于类型 User 创建了新类型 User1，第 3 行代码是创建了 User 的类型别名 User2，注意使用 = 定义类型别名。因为 User2 是别名，完全等价于 User，所以 User2 具有 User 所有的方法。但是 i1.m2() 是不能执行的，因为 User1 没有定义该方法。

第三十七天

第三十七天

1.关于channel的特性，下面说法正确的是？

- A. 给一个 nil channel 发送数据，造成永远阻塞
- B. 从一个 nil channel 接收数据，造成永远阻塞
- C. 给一个已经关闭的 channel 发送数据，引起 panic
- D. 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

参考答案及解析：ABCD。

2.下面代码有什么问题？

```
const i = 100
var j = 123

func main() {
    fmt.Println(&j, j)
    fmt.Println(&i, i)
}
```

参考答案及解析：编译报错cannot take the address of i。知识点：常量。常量不同于变量的在运行期分配内存，常量通常会被编译器在预处理阶段直接展开，作为指令数据使用，所以常量无法寻址。

3.下面代码能否编译通过？如果通过，输出什么？

```
func GetValue(m map[int]string, id int) (string, bool) {

    if _, exist := m[id]; exist {
        return "exist", true
    }
    return nil, false
}
func main() {
    intmap := map[int]string{
        1: "a",
        2: "b",
        3: "c",
    }

    v, err := GetValue(intmap, 3)
    fmt.Println(v, err)
}
```

参考答案及解析：不能通过编译。知识点：函数返回值类型。nil 可以用作 interface、function、pointer、map、slice 和 channel 的“空值”。但是如果不特别指定的话，Go 语言不能识别类型，所以会报错:cannot use nil as type string in return argument.

第三十六天

第三十六天

1.关于函数声明，下面语法正确的是？

- A. func f(a, b int) (value int, err error)
- B. func f(a int, b int) (value int, err error)
- C. func f(a, b int) (value int, error)
- D. func f(a int, b int) (int, int, error)

参考答案及解析：ABD。

2.关于整型切片的初始化，下面正确的是？

- A. s := make([]int)
- B. s := make([]int, 0)
- C. s := make([]int, 5, 10)
- D. s := []int{1, 2, 3, 4, 5}

参考答案及解析：BCD

3.下面代码会触发异常吗？请说明。

```
func main() {
    runtime.GOMAXPROCS(1)
    int_chan := make(chan int, 1)
    string_chan := make(chan string, 1)
    int_chan <- 1
    string_chan <- "hello"
    select {
    case value := <-int_chan:
        fmt.Println(value)
    case value := <-string_chan:
        panic(value)
    }
}
```

参考答案及解析：select 会随机选择一个可用通道做收发操作，所以可能触发异常，也可能不会。

第三十五天

第三十五天

1.关于 bool 变量 b 的赋值，下面错误的用法是？

- A. b = true
- B. b = 1
- C. b = bool(1)
- D. b = (1 == 2)

参考答案及解析：BC。

2.关于变量的自增和自减操作，下面语句正确的是？

```
A.
i := 1
i++

B.
i := 1
j = i++

C.
i := 1
++i

D.
i := 1
i--
```

参考答案及解析：AD。知识点：自增自减操作。i++ 和 i– 在 Go 语言中是语句，不是表达式，因此不能赋值给另外的变量。此外没有 ++i 和 –i。

3.关于GetPodAction定义，下面赋值正确的是

```
type Fragment interface {
        Exec(transInfo *TransInfo) error
}
type GetPodAction struct {
}
func (g GetPodAction) Exec(transInfo *TransInfo) error {
        ...
        return nil
}
```

- A. var fragment Fragment = new(GetPodAction)

- B. var fragment Fragment = GetPodAction
- C. var fragment Fragment = &GetPodAction{}
- D. var fragment Fragment = GetPodAction{}

参考答案及解析：ACD。

第三十四天

第三十四天

1.关于类型转化，下面选项正确的是？

A.

type MyInt int

var i int = 1

var j MyInt = i

B.

type MyInt int

var i int = 1

var j MyInt = (MyInt)i

C.

type MyInt int

var i int = 1

var j MyInt = MyInt(i)

D.

type MyInt int

var i int = 1

var j MyInt = i.(MyInt)

参考答案及解析：C。知识点：强制类型转化。

2.关于switch语句，下面说法正确的有?

- A. 条件表达式必须为常量或者整数；
- B. 单个case中，可以出现多个结果选项；
- C. 需要用break来明确退出一个case；
- D. 只有在case中明确添加fallthrough关键字，才会继续执行紧跟的下一个case；

参考答案及解析：BD。参考文章 条件语句和循环语句

3.如果 Add() 函数的调用代码为：

```
func main() {
    var a Integer = 1
    var b Integer = 2
    var i interface{} = &a
    sum := i.(*Integer).Add(b)
    fmt.Println(sum)
}
```

则Add函数定义正确的是()

```
A.
type Integer int
func (a Integer) Add(b Integer) Integer {
        return a + b
}

B.
type Integer int
func (a Integer) Add(b *Integer) Integer {
        return a + *b
}

C.
type Integer int
func (a *Integer) Add(b Integer) Integer {
        return *a + b
}

D.
type Integer int
func (a *Integer) Add(b *Integer) Integer {
        return *a + *b
}
```

参考答案及解析：AC。知识点：类型断言、方法集。

第三十三天

第三十三天

1.关于协程，下面说法正确是（）

- A. 协程和线程都可以实现程序的并发执行；
- B. 线程比协程更轻量级；
- C. 协程不存在死锁问题；
- D. 通过 channel 来进行协程间的通信；

参考答案及解析：AD。

2.关于循环语句，下面说法正确的有（）

- A. 循环语句既支持 for 关键字，也支持 while 和 do-while；
- B. 关键字 for 的基本使用方法与 C/C++ 中没有任何差异；
- C. for 循环支持 continue 和 break 来控制循环，但是它提供了一个更高级的 break，可以选择中断哪一个循环；
- D. for 循环不支持以逗号为间隔的多个赋值语句，必须使用平行赋值的方式来初始化多个变量；

参考答案及解析：CD。

3.下面代码输出正确的是？

```
func main() {
    i := 1
    s := []string{"A", "B", "C"}
    i, s[i-1] = 2, "Z"
    fmt.Printf("s: %v \n", s)
}
```

- A. s: [Z,B,C]
- B. s: [A,Z,C]

参考答案及解析：A。知识点：多重赋值。

多重赋值分为两个步骤，有先后顺序：

- 计算等号左边的索引表达式和取址表达式，接着计算等号右边的表达式；
- 赋值；

所以本例，会先计算 s[i-1]，等号右边是两个表达式是常量，所以赋值运算等同于 i, s[0] = 2, “Z”。

redis知识点

第三十二天

第三十二天

1.下面这段代码输出结果正确吗？

```
type Foo struct {
    bar string
}
func main() {
    s1 := []Foo{
        {"A"},
        {"B"},
        {"C"},
    }
    s2 := make([]*Foo, len(s1))
    for i, value := range s1 {
        s2[i] = &value
    }
    fmt.Println(s1[0], s1[1], s1[2])
    fmt.Println(s2[0], s2[1], s2[2])
}
输出：
{A} {B} {C}
&{A} &{B} &{C}
```

参考答案及解析：s2 的输出结果错误。s2 的输出是 &{C} &{C} &{C}，在 第 30 天 的答案解析第二题，我们提到过，for range 使用短变量声明(:=)的形式迭代变量时，变量 i、value 在每次循环体中都会被重用，而不是重新声明。所以 s2 每次填充的都是临时变量 value 的地址，而在最后一次循环中，value 被赋值为{c}。因此，s2 输出的时候显示出了三个 &{c}。

可行的解决办法如下：

```
for i := range s1 {
    s2[i] = &s1[i]
}
```

2.下面代码里的 counter 的输出值？

```
func main() {

    var m = map[string]int{
        "A": 21,
        "B": 22,
        "C": 23,
    }
    counter := 0
    for k, v := range m {
        if counter == 0 {
            delete(m, "A")
        }
        counter++
        fmt.Println(k, v)
    }
```

```
    fmt.Println("counter is ", counter)
}
```

- A. 2
- B. 3
- C. 2 或 3

参考答案及解析：C。for range map 是无序的，如果第一次循环到 A，则输出 3；否则输出 2。

第三十一天

第三十一天

1.下面这段代码输出什么？

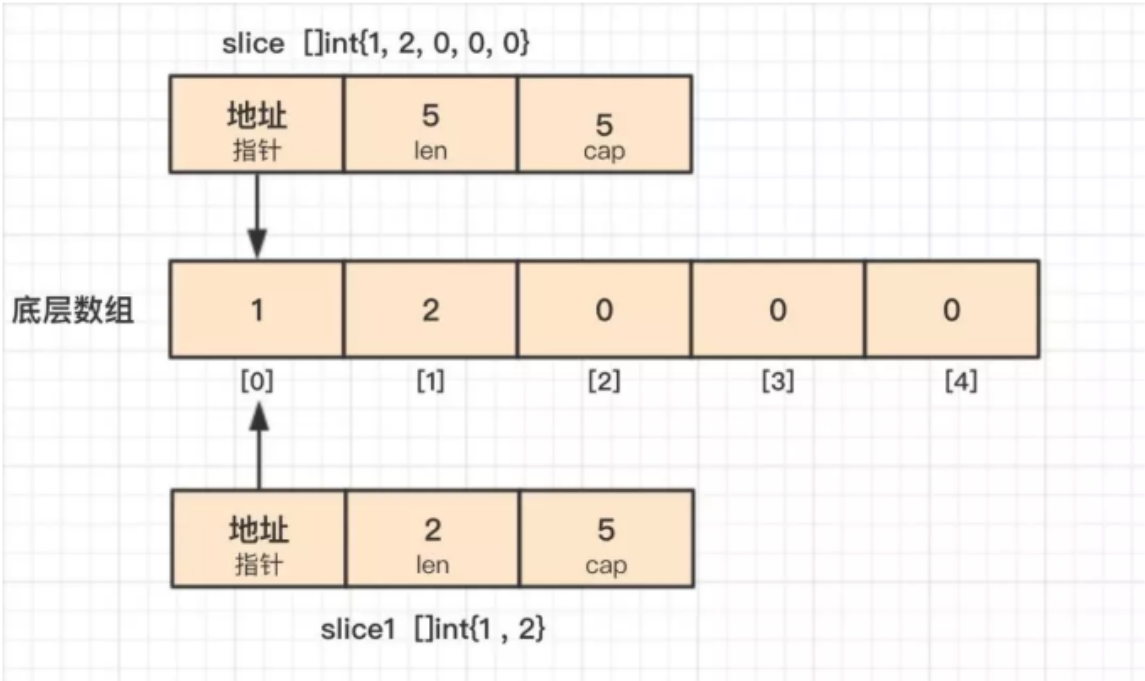
```
func change(s ...int) {
    s = append(s,3)
}

func main() {
    slice := make([]int,5,5)
    slice[0] = 1
    slice[1] = 2
    change(slice...)
    fmt.Println(slice)
    change(slice[0:2]...)
    fmt.Println(slice)
}
```

参考答案及解析：

```
[1 2 0 0 0]
[1 2 3 0 0]
```

知识点：可变函数、append()操作。Go 提供的语法糖...，可以将 slice 传进可变函数，不会创建新的切片。第一次调用 change() 时，append() 操作使切片底层数组发生了扩容，原 slice 的底层数组不会改变；第二次调用change() 函数时，使用了操作符[i,j]获得一个新的切片，假定为 slice1，它的底层数组和原切片底层数组是重合的，不过 slice1 的长度、容量分别是 2、5，所以在 change() 函数中对 slice1 底层数组的修改会影响到原切片。



2.下面这段代码输出什么？

```
func main() {
    var a = []int{1, 2, 3, 4, 5}
    var r [5]int

    for i, v := range a {
        if i == 0 {
            a[1] = 12
            a[2] = 13
        }
        r[i] = v
    }
    fmt.Println("r = ", r)
    fmt.Println("a = ", a)
}
```

参考答案及解析：

```
r =  [1 12 13 4 5]
a =  [1 12 13 4 5]
```

这道题是 第30天 的第二题的一个解决办法，这的 a 是一个切片，那切片是怎么实现的呢？切片在 go 的内部结构有一个指向底层数组的指针，当 range 表达式发生复制时，副本的指针依旧指向原底层数组，所以对切片的修改都会反应到底层数组上，所以通过 v 可以获得修改后的数组元素。

引自：https://tonybai.com/2015/09/17/7-things-you-may-not-pay-attation-to-in-go/

第三十天

第三十天

1.下面这段代码输出什么？

```
func f(n int) (r int) {
    defer func() {
        r += n
        recover()
    }()

    var f func()

    defer f()
    f = func() {
        r += 2
    }
    return n + 1
}

func main() {
    fmt.Println(f(3))
}
```

参考答案及解析：7。根据 5 年 Gopher 都不知道的 defer 细节，你别再掉进坑里！ 提到的“三步拆解法”，第一步执行r = n +1，接着执行第二个 defer，由于此时 f() 未定义，引发异常，随即执行第一个 defer，异常被 recover()，程序正常执行，最后 return。

此题引自知识星球《Go项目实战》。

2.下面这段代码输出什么？

```
func main() {
    var a = [5]int{1, 2, 3, 4, 5}
    var r [5]int

    for i, v := range a {
        if i == 0 {
            a[1] = 12
            a[2] = 13
        }
        r[i] = v
    }
    fmt.Println("r = ", r)
    fmt.Println("a = ", a)
}
```

参考答案及解析：

```
r =  [1 2 3 4 5]
a =  [1 12 13 4 5]
```

range 表达式是副本参与循环，就是说例子中参与循环的是 a 的副本，而不是真正的 a。就这个例子来说，假设 b 是 a 的副本，则 range 循环代码是这样的

```
for i, v := range b {
    if i == 0 {
        a[1] = 12
        a[2] = 13
    }
    r[i] = v
}
```

因此无论 a 被如何修改，其副本 b 依旧保持原值，并且参与循环的是 b，因此 v 从 b 中取出的仍旧是 a 的原值，而非修改后的值。

如果想要 r 和 a 一样输出，修复办法：

```
func main() {
    var a = [5]int{1, 2, 3, 4, 5}
    var r [5]int

    for i, v := range &a {
        if i == 0 {
            a[1] = 12
            a[2] = 13
        }
        r[i] = v
    }
    fmt.Println("r = ", r)
    fmt.Println("a = ", a)
}
```

输出：

```
r =  [1 12 13 4 5]
a =  [1 12 13 4 5]
```

修复代码中，使用 `*[5]int` 作为 range 表达式，其副本依旧是一个指向原数组 a 的指针，因此后续所有循环中均是 &a 指向的原数组亲自参与的，因此 v 能从 &a 指向的原数组中取出 a 修改后的值。

引自：https://tonybai.com/2015/09/17/7-things-you-may-not-pay-attation-to-in-go/

第二十九天

第二十九天

1.下面这段代码能否正常结束？

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

参考答案及解析：不会出现死循环，能正常结束。
循环次数在循环开始前就已经确定，循环内改变切片的长度，不影响循环次数。

2.下面这段代码输出什么？为什么？

```
func main() {

    var m = [...]int{1, 2, 3}

    for i, v := range m {
        go func() {
            fmt.Println(i, v)
        }()
    }

    time.Sleep(time.Second * 3)
}
```

参考答案及解析：

```
2 3
2 3
2 3
```

for range 使用短变量声明(:=)的形式迭代变量，需要注意的是，变量 i、v 在每次循环体中都会被重用，而不是重新声明。

各个 goroutine 中输出的 i、v 值都是 for range 循环结束后的 i、v 最终值，而不是各个goroutine启动时的i, v值。可以理解为闭包引用，使用的是上下文环境的值。

两种可行的 fix 方法:

1.使用函数传递

```
for i, v := range m {
    go func(i,v int) {
        fmt.Println(i, v)
    }(i,v)
}
```

2.使用临时变量保留当前值

```
for i, v := range m {
    i := i          // 这里的 := 会重新声明变量，而不是重用
    v := v
    go func() {
        fmt.Println(i, v)
    }()
}
```

引自：https://tonybai.com/2015/09/17/7-things-you-may-not-pay-attation-to-in-go/

第二十八天

第二十八天

1.下面的代码有什么问题？

```
func main() {
    fmt.Println([...]int{1} == [2]int{1})
    fmt.Println([]int{1} == []int{1})
}
```

参考答案及解析：有两处错误

- go 中不同类型是不能比较的，而数组长度是数组类型的一部分，所以 [...]int{1} 和 [2]int{1} 是两种不同的类型，不能比较；
- 切片是不能比较的；

2.下面这段代码输出什么？

```
var p *int

func foo() (*int, error) {
    var i int = 5
    return &i, nil
}

func bar() {
    //use p
    fmt.Println(*p)
}

func main() {
    p, err := foo()
    if err != nil {
        fmt.Println(err)
        return
    }
    bar()
    fmt.Println(*p)
}
```

- A. 5 5
- B. runtime error

参考答案及解析：B。知识点：变量作用域。问题出在操作符:=，对于使用:=定义的变量，如果新变量与同名已定义的变量不在同一个作用域中，那么 Go 会新定义这个变量。对于本例来说，main() 函数里的 p 是新定义的变量，会遮住全局变量 p，导致执行到bar()时程序，全局变量 p 依然还是 nil，程序随即 Crash。

正确的做法是将 main() 函数修改为：

```
func main() {
    var err error
    p, err = foo()
    if err != nil {
        fmt.Println(err)
        return
    }
    bar()
    fmt.Println(*p)
}
```

这道题目引自 Tony Bai 老师的一篇文章，原文讲的很详细，推荐。
https://tonybai.com/2015/01/13/a-hole-about-variable-scope-in-golang/

第二十七天

第二十七天

1.下面这段代码输出什么？

```
type Direction int

const (
    North Direction = iota
    East
    South
    West
)

func (d Direction) String() string {
    return [...]string{"North", "East", "South", "West"}[d]
}

func main() {
    fmt.Println(South)
}
```

参考答案及解析：South。知识点：iota 的用法、类型的 String() 方法。

根据 iota 的用法推断出 South 的值是 2；另外，如果类型定义了 String() 方法，当使用 fmt.Printf()、fmt.Print() 和 fmt.Println() 会自动使用 String() 方法，实现字符串的打印。

2.下面代码输出什么？

```
type Math struct {
    x, y int
}

var m = map[string]Math{
    "foo": Math{2, 3},
}

func main() {
    m["foo"].x = 4
    fmt.Println(m["foo"].x)
}
```

- A. 4
- B. compilation error

参考答案及解析：B，编译报错 cannot assign to struct field m[“foo”].x in map。错误原因：对于类似 X = Y的赋值操作，必须知道 X 的地址，才能够将 Y 的值赋给 X，但 go 中的 map 的 value 本身是不可寻址的。

有两个解决办法：

1.使用临时变量

```
type Math struct {
    x, y int
}

var m = map[string]Math{
    "foo": Math{2, 3},
}

func main() {
    tmp := m["foo"]
    tmp.x = 4
    m["foo"] = tmp
    fmt.Println(m["foo"].x)
}
```

2.修改数据结构

```
type Math struct {
    x, y int
}

var m = map[string]*Math{
    "foo": &Math{2, 3},
}

func main() {
    m["foo"].x = 4
    fmt.Println(m["foo"].x)
    fmt.Printf("%#v", m["foo"])   // %#v 格式化输出详细信息
}
```

第二十六天

第二十六天

1.下面这段代码输出什么？

```
const (
    a = iota
    b = iota
)
const (
    name = "name"
    c    = iota
    d    = iota
)
func main() {
    fmt.Println(a)
    fmt.Println(b)
    fmt.Println(c)
    fmt.Println(d)
}
```

参考答案及解析：0 1 1 2。知识点：iota 的用法。

iota 是 golang 语言的常量计数器，只能在常量的表达式中使用。

iota 在 const 关键字出现时将被重置为0，const中每新增一行常量声明将使 iota 计数一次。

推荐阅读：

https://studygolang.com/articles/2192

2.下面这段代码输出什么？为什么？

```
type People interface {
    Show()
}

type Student struct{}

func (stu *Student) Show() {

}

func main() {

    var s *Student
    if s == nil {
        fmt.Println("s is nil")
    } else {
        fmt.Println("s is not nil")
    }
    var p People = s
    if p == nil {
        fmt.Println("p is nil")
```

```
    } else {
        fmt.Println("p is not nil")
    }
}
```

参考答案及解析：s is nil 和 p is not nil。这道题会不会有点诧异，我们分配给变量 p 的值明明是 nil，然而 p 却不是 nil。记住一点，当且仅当动态值和动态类型都为 nil 时，接口类型值才为 nil。上面的代码，给变量 p 赋值之后，p 的动态值是 nil，但是动态类型却是 `*Student` ，是一个 nil 指针，所以相等条件不成立

第二十五天

第二十五天

1.下面这段代码输出什么？为什么？

```
func (i int) PrintInt ()  {
    fmt.Println(i)
}

func main() {
    var i int = 1
    i.PrintInt()
}
```

- A. 1
- B. compilation error

参考答案及解析：B。基于类型创建的方法必须定义在同一个包内，上面的代码基于 int 类型创建了 PrintInt() 方法，由于 int 类型和方法 PrintInt() 定义在不同的包内，所以编译出错。

解决的办法可以定义一种新的类型：

```
type Myint int

func (i Myint) PrintInt ()  {
    fmt.Println(i)
}

func main() {
    var i Myint = 1
    i.PrintInt()
}
```

2.下面这段代码输出什么？为什么？

```
type People interface {
    Speak(string) string
}

type Student struct{}

func (stu *Student) Speak(think string) (talk string) {
    if think == "speak" {
        talk = "speak"
    } else {
        talk = "hi"
    }
    return
}

func main() {
```

```
    var peo People = Student{}
    think := "speak"
    fmt.Println(peo.Speak(think))
}
```

- A. speak
- B. compilation error

参考答案及解析：B。编译错误 Student does not implement People (Speak method has pointer receiver)，值类型 Student 没有实现接口的 Speak() 方法，而是指针类型 `*Student` 实现该方法。

第二十四天

第二十四天

1.下面这段代码输出什么？

```
func main() {
    m := map[int]string{0:"zero",1:"one"}
    for k,v := range m {
        fmt.Println(k,v)
    }
}
```

参考答案及解析：

```
0 zero
1 one
// 或者
1 one
0 zero
```

map 的输出是无序的。

2.下面这段代码输出什么？

```
func main() {
    a := 1
    b := 2
    defer calc("1", a, calc("10", a, b))
    a = 0
    defer calc("2", a, calc("20", a, b))
    b = 1
}

func calc(index string, a, b int) int {
    ret := a + b
    fmt.Println(index, a, b, ret)
    return ret
}
```

参考答案及解析：

```
10 1 2 3
20 0 2 2
2 0 2 2
1 1 3 4
```

程序执行到 main() 函数三行代码的时候，会先执行 calc() 函数的 b 参数，即：calc(“10”,a,b)，输出：10 1 2 3，得到值 3，因为
defer 定义的函数是延迟函数，故 calc(“1”,1,3) 会被延迟执行；

程序执行到第五行的时候，同样先执行 calc(“20”,a,b) 输出：20 0 2 2 得到值 2，同样将 calc(“2”,0,2) 延迟执行；

程序执行到末尾的时候，按照栈先进后出的方式依次执行：calc(“2”,0,2)，calc(“1”,1,3)，则就依次输出：2 0 2 2，1 1 3 4。

第二十三天

第二十三天

1.下面这段代码输出什么？为什么？

```
func main() {

    s1 := []int{1, 2, 3}
    s2 := s1[1:]
    s2[1] = 4
    fmt.Println(s1)
    s2 = append(s2, 5, 6, 7)
    fmt.Println(s1)
}
```

参考答案及解析：

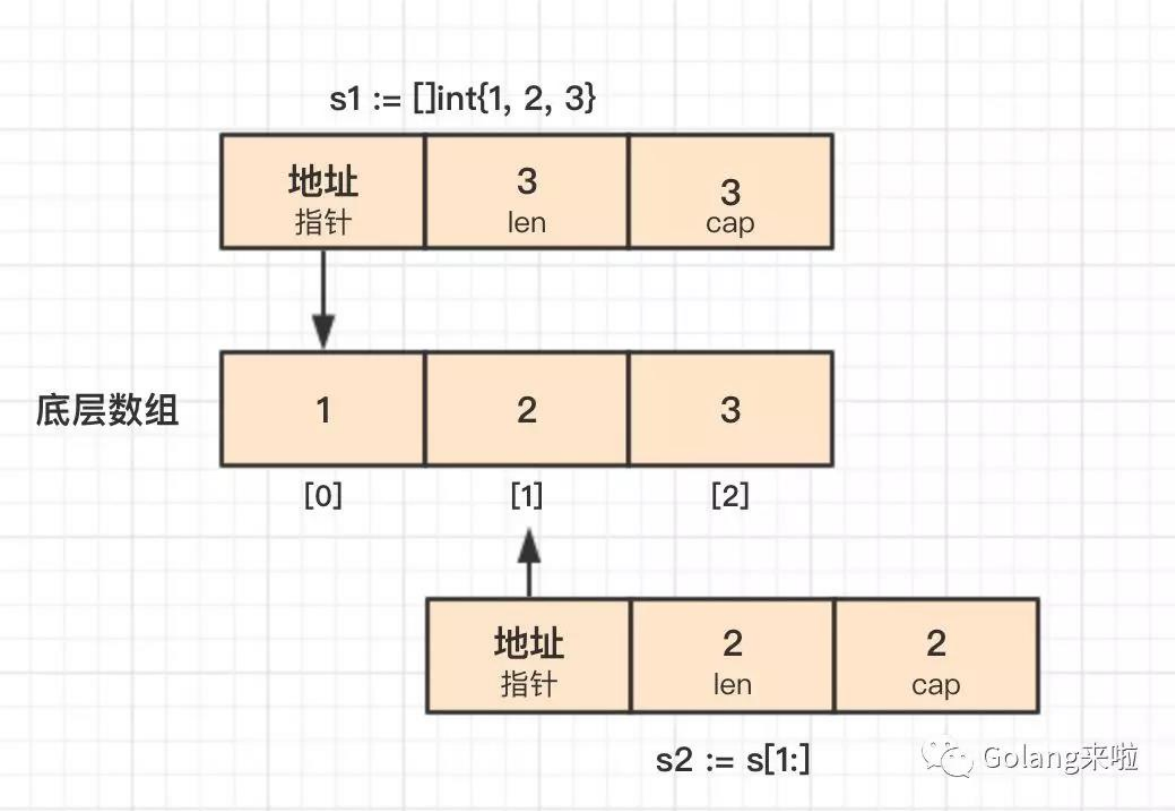
[1 2 4]

[1 2 4]

我们知道，golang 中切片底层的数据结构是数组。当使用 s1[1:] 获得切片 s2，和 s1 共享同一个底层数组，这会导致 s2[1] = 4 语句影响 s1。

而 append 操作会导致底层数组扩容，生成新的数组，因此追加数据后的 s2 不会影响 s1。

但是为什么对 s2 赋值后影响的却是 s1 的第三个元素呢？这是因为切片 s2 是从数组的第二个元素开始，s2 索引为 1 的元素对应的是 s1 索引为 2 的元素。



2.下面选项正确的是？

```
func main() {
    if a := 1; false {
    } else if b := 2; false {
    } else {
        println(a, b)
    }
}
```

- A. 1 2
- B. compilation error

参考答案及解析：A。知识点：代码块和变量作用域。

第二十二天

第二十二天

1.下面的代码有几处语法问题，各是什么？

```
package main
import (
    "fmt"
)
func main() {
    var x string = nil
    if x == nil {
        x = "default"
    }
    fmt.Println(x)
}
```

参考答案及解析：两个地方有语法问题。golang 的字符串类型是不能赋值 nil 的，也不能跟 nil 比较。

2.return 之后的 defer 语句会执行吗，下面这段代码输出什么？

```
var a bool = true
func main() {
    defer func(){
        fmt.Println("1")
    }()
    if a == true {
        fmt.Println("2")
        return
    }
    defer func(){
        fmt.Println("3")
    }()
}
```

参考答案及解析：2 1。defer 关键字后面的函数或者方法想要执行必须先注册，return 之后的 defer 是不能注册的， 也就不能执行后面的函数或方法

第二十一天

第二十一天

1.下面的两个切片声明中有什么区别？哪个更可取？

```
A. var a []int
B. a := []int{}
```

参考答案及解析：A 声明的是 nil 切片；B 声明的是长度和容量都为 0 的空切片。第一种切片声明不会分配内存，优先选择。

2. A、B、C、D 哪些选项有语法错误？

```
type S struct {
}

func f(x interface{}) {
}

func g(x *interface{}) {
}

func main() {
    s := S{}
    p := &s
    f(s) //A
    g(s) //B
    f(p) //C
    g(p) //D
}
```

参考答案及解析：BD。函数参数为 interface{} 时可以接收任何类型的参数，包括用户自定义类型等，即使是接收指针类型也用 interface{}，而不是使用 `*interface{}` 。

```
永远不要使用一个指针指向一个接口类型，因为它已经是一个指针。
```

3.下面 A、B 两处应该填入什么代码，才能确保顺利打印出结果？

```
type S struct {
    m string
}

func f() *S {
    return __  //A
}

func main() {
    p := __    //B
    fmt.Println(p.m) //print "foo"
}
```

参考答案及解析：

```
A. &S{"foo"}
B. *f() 或者 f()
```

f() 函数返回参数是指针类型，所以可以用 & 取结构体的指针；B 处，如果填 `*f()` ，则 p 是 S 类型；如果填 f()，则 p 是 `*S` 类型，不过都可以使用 p.m 取得结构体的成员。

第二十天

第二十天

1.下面这段代码正确的输出是什么？

```
func f() {
    defer fmt.Println("D")
    fmt.Println("F")
}

func main() {
    f()
    fmt.Println("M")
}
```

- A. F M D
- B. D F M
- C. F D M

参考答案及解析：C。被调用函数里的 defer 语句在返回之前就会被执行，所以输出顺序是 F D M。

2.下面代码输出什么？

```
type Person struct {
    age int
}

func main() {
    person := &Person{28}

    // 1.
    defer fmt.Println(person.age)

    // 2.
    defer func(p *Person) {
        fmt.Println(p.age)
    }(person)

    // 3.
    defer func() {
        fmt.Println(person.age)
    }()

    person = &Person{29}
}
```

参考答案及解析：29 28 28。这道题在第 19 天题目的基础上做了一点点小改动，前一题最后一行代码 person.age = 29 是修改引用对象的成员 age，这题最后一行代码 person = &Person{29} 是修改引用对象本身，来看看有什么区别。

1处.person.age 这一行代码跟之前含义是一样的，此时是将 28 当做 defer 函数的参数，会把 28 缓存在栈中，等到最后执行该 defer 语句的时候取出，即输出 28；

2处.defer 缓存的是结构体 Person{28} 的地址，这个地址指向的结构体没有被改变，最后 defer 语句后面的函数执行的时候取出仍是 28；

3处.闭包引用，person 的值已经被改变，指向结构体 Person{29}，所以输出 29.

由于 defer 的执行顺序为先进后出，即 3 2 1，所以输出 29 28 28。

第十九天

第十九天

1.下面代码段输出什么？

```
type Person struct {
    age int
}

func main() {
    person := &Person{28}

    // 1.
    defer fmt.Println(person.age)

    // 2.
    defer func(p *Person) {
        fmt.Println(p.age)
    }(person)

    // 3.
    defer func() {
        fmt.Println(person.age)
    }()

    person.age = 29
}
```

参考答案及解析：29 29 28。变量 person 是一个指针变量 。

1.person.age 此时是将 28 当做 defer 函数的参数，会把 28 缓存在栈中，等到最后执行该 defer 语句的时候取出，即输出 28；

2.defer 缓存的是结构体 Person{28} 的地址，最终 Person{28} 的 age 被重新赋值为 29，所以 defer 语句最后执行的时候，依靠缓存的地址取出的 age 便是 29，即输出 29；

3.闭包引用，输出 29；

又由于 defer 的执行顺序为先进后出，即 3 2 1，所以输出 29 29 28。

第十八天

第十八天

1.f1()、f2()、f3() 函数分别返回什么？

```
func f1() (r int) {
    defer func() {
        r++
    }()
    return 0
}

func f2() (r int) {
    t := 5
    defer func() {
        t = t + 5
    }()
    return t
}

func f3() (r int) {
    defer func(r int) {
        r = r + 5
    }(r)
    return 1
}
```

答案是 29 29 28
首先定义的局部变量person类型是一个指针
其次defer是先进后出结构，故defer执行顺序为3 2 1
3是匿名函数使用外部对象，而对象是指针，又在defer中，执行优先级为最低，故最外层代码修改以后，defer则会使用修改后的对象，故29
2是函数传递参数进去，因为是指针，同3
1看是一条语句，其实可以写成defer func(age int){fmt.Println(age)}(person.age) 因为传递的是执行到此初始person对象的age值，是而在此时age为28因为是值类型传递，所以输出为28

第十七天

第十七天

1.下面代码中，x 已声明，y 没有声明，判断每条语句的对错。

```
1. x, _ := f()
2. x, _ = f()
3. x, y := f()
4. x, y = f()
```

参考答案及解析：错、对、对、错。知识点：变量的声明。1.错，x 已经声明，不能使用 :=；2.对；3.对，当多值赋值时，:= 左边的变量无论声明与否都可以；4.错，y 没有声明。

2.下面代码输出什么？

```
func increaseA() int {
    var i int
    defer func() {
        i++
    }()
    return i
}

func increaseB() (r int) {
    defer func() {
        r++
    }()
    return r
}

func main() {
    fmt.Println(increaseA())
    fmt.Println(increaseB())
}
```

- A. 1 1
- B. 0 1
- C. 1 0
- D. 0 0

参考答案及解析：B。知识点：defer、返回值。注意一下，increaseA() 的返回参数是匿名，increaseB() 是具名。关于 defer 与返回值的知识点，后面我会写篇文章详细分析，到时候可以看下文章的讲解。

下面代码输出什么？

```
type A interface {
    ShowA() int
}
```

```
type B interface {
    ShowB() int
}

type Work struct {
    i int
}

func (w Work) ShowA() int {
    return w.i + 10
}

func (w Work) ShowB() int {
    return w.i + 20
}

func main() {
    var a A = Work{3}
    s := a.(Work)
    fmt.Println(s.ShowA())
    fmt.Println(s.ShowB())
}
```

- A. 13 23
- B. compilation error

参考答案及解析：A。知识点：类型断言。这道题可以和第 15 天的第三题 和第 16 天的第三题结合起来看

第十六天

第十六天

1.切片 a、b、c 的长度和容量分别是多少？

```
func main() {

    s := [3]int{1, 2, 3}
    a := s[:0]
    b := s[:2]
    c := s[1:2:cap(s)]
}
```

参考答案及解析：a、b、c 的长度和容量分别是 0 3、2 3、1 2。知识点：数组或切片的截取操作。截取操作有带 2 个或者 3 个参数，形如：[i:j] 和 [i:j:k]，假设截取对象的底层数组长度为 l。在操作符 [i:j] 中，如果 i 省略，默认 0，如果 j 省略，默认底层数组的长度，截取得到的切片长度和容量计算方法是 j-i、l-i。操作符 [i:j:k]，k 主要是用来限制切片的容量，但是不能大于数组的长度 l，截取得到的切片长度和容量计算方法是 j-i、k-i。

2.下面代码中 A B 两处应该怎么修改才能顺利编译？

```
func main() {
    var m map[string]int        //A
    m["a"] = 1
    if v := m["b"]; v != nil {  //B
        fmt.Println(v)
    }
}
```

参考答案及解析：

```
func main() {
    m := make(map[string]int)
    m["a"] = 1
    if v,ok := m["b"]; ok {
        fmt.Println(v)
    }
}
```

在 A 处只声明了map m ,并没有分配内存空间，不能直接赋值，需要使用 make()，都提倡使用 make() 或者字面量的方式直接初始化 map。

B 处，v,k := m[“b”] 当 key 为 b 的元素不存在的时候，v 会返回值类型对应的零值，k 返回 false。

3.下面代码输出什么？

```
type A interface {
    ShowA() int
}

type B interface {
    ShowB() int
```

```
}

type Work struct {
    i int
}

func (w Work) ShowA() int {
    return w.i + 10
}

func (w Work) ShowB() int {
    return w.i + 20
}

func main() {
    c := Work{3}
    var a A = c
    var b B = c
    fmt.Println(a.ShowB())
    fmt.Println(b.ShowA())
}
```

- A. 23 13
- B. compilation error

参考答案及解析：B。知识点：接口的静态类型。a、b 具有相同的动态类型和动态值，分别是结构体 work 和 {3}；a 的静态类型是 A，b 的静态类型是 B，接口 A 不包括方法 ShowB()，接口 B 也不包括方法 ShowA()，编译报错。看下编译错误：

```
a.ShowB undefined (type A has no field or method ShowB)
b.ShowA undefined (type B has no field or method ShowA)
```

第十五天

第十五天

1.下面代码下划线处可以填入哪个选项？

```
func main() {
    var s1 []int
    var s2 = []int{}
    if __ == nil {
        fmt.Println("yes nil")
    }else{
        fmt.Println("no nil")
    }
}
```

- A. s1
- B. s2
- C. s1、s2 都可以

参考答案及解析：A。知识点：nil 切片和空切片。nil 切片和 nil 相等，一般用来表示一个不存在的切片；空切片和 nil 不相等，表示一个空的集合。

2.下面这段代码输出什么？

```
func main() {
    i := 65
    fmt.Println(string(i))
}
```

- A. A
- B. 65
- C. compilation error

参考答案及解析：A。UTF-8 编码中，十进制数字 65 对应的符号是 A。

3.下面这段代码输出什么？

```
type A interface {
    ShowA() int
}

type B interface {
    ShowB() int
}

type Work struct {
```

```
    i int
}

func (w Work) ShowA() int {
    return w.i + 10
}

func (w Work) ShowB() int {
    return w.i + 20
}

func main() {
    c := Work{3}
    var a A = c
    var b B = c
    fmt.Println(a.ShowA())
    fmt.Println(b.ShowB())
}
```

参考答案及解析：13 23。知识点：接口。一种类型实现多个接口，结构体 Work 分别实现了接口 A、B，所以接口变量 a、b 调用各自的方法 ShowA() 和 ShowB()，输出 13、23。

第十四天

第十四天

1.下面代码输出什么？

```
func main() {
    str := "hello"
    str[0] = 'x'
    fmt.Println(str)
}
```

- A. hello
- B. xello
- C. compilation error

参考代码及解析：C。知识点：常量，Go 语言中的字符串是只读的。

2.下面代码输出什么？

```
func incr(p *int) int {
    *p++
    return *p
}

func main() {
    p :=1
    incr(&p)
    fmt.Println(p)
}
```

- A. 1
- B. 2
- C. 3

参考答案及解析：B。知识点：指针，incr() 函数里的 p 是 `*int` 类型的指针，指向的是 main() 函数的变量 p 的地址。第 2 行代码是将该地址的值执行一个自增操作，incr() 返回自增后的结果。

3.对 add() 函数调用正确的是（）

```
func add(args ...int) int {

    sum := 0
    for _, arg := range args {
        sum += arg
    }
```

```
    return sum
}
```

- A. add(1, 2)
- B. add(1, 3, 7)
- C. add([]int{1, 2})
- D. add([]int{1, 3, 7}...)

参考答案及解析：ABD。知识点：可变函数。

第十三天

第十三天

1.定义一个包内全局字符串变量，下面语法正确的是（）

- A. var str string
- B. str := “”
- C. str = “”
- D. var str = “”

参考答案及解析：AD。B 只支持局部变量声明；C 是赋值，str 必须在这之前已经声明；

2.下面这段代码输出什么?

```
func hello(i int) {
    fmt.Println(i)
}
func main() {
    i := 5
    defer hello(i)
    i = i + 10
}
```

参考答案及解析：5。这个例子中，hello() 函数的参数在执行 defer 语句的时候会保存一份副本，在实际调用 hello() 函数时用，所以是 5.

3.下面这段代码输出什么？

```
type People struct{}

func (p *People) ShowA() {
    fmt.Println("showA")
    p.ShowB()
}
func (p *People) ShowB() {
    fmt.Println("showB")
}

type Teacher struct {
    People
}

func (t *Teacher) ShowB() {
    fmt.Println("teacher showB")
}

func main() {
    t := Teacher{}
```

```
    t.ShowA()
}
```

参考答案及解析：

```
showA
showB
```

知识点：结构体嵌套。这道题可以结合第 12 天的第三题一起看，Teacher 没有自己 ShowA()，所以调用内部类型 People 的同名方法，需要注意的是第 5 行代码调用的是 People 自己的 ShowB 方法。

第十二天

第十二天

1.下面属于关键字的是（）

- A.func
- B.struct
- C.class
- D.defer

参考答案及解析：ABD。知识点：Go 语言的关键字。Go 语言有 25 个关键字，看下图：

```
break        default      func         interface    select
case         defer        go           map          struct
chan         else         goto         package      switch
const        fallthrough  if           range        type
continue     for          import       return       var
```

2.下面这段代码输出什么？

```
func main() {
    i := -5
    j := +5
    fmt.Printf("%+d %+d", i, j)
}
```

- A. -5 +5
- B. +5 +5
- C. 0 0

参考答案及解析：A。%d表示输出十进制数字，+表示输出数值的符号。这里不表示取反。

3.下面这段代码输出什么？

```
type People struct{}

func (p *People) ShowA() {
    fmt.Println("showA")
    p.ShowB()
}
```

```
func (p *People) ShowB() {
    fmt.Println("showB")
}

type Teacher struct {
    People
}

func (t *Teacher) ShowB() {
    fmt.Println("teacher showB")
}

func main() {
    t := Teacher{}
    t.ShowB()
}
```

参考答案及解析：teacher showB。知识点：结构体嵌套。在嵌套结构体中，People 称为内部类型，Teacher 称为外部类型；通过嵌套，内部类型的属性、方法，可以为外部类型所有，就好像是外部类型自己的一样。此外，外部类型还可以定义自己的属性和方法，甚至可以定义与内部相同的方法，这样内部类型的方法就会被“屏蔽”。这个例子中的 ShowB() 就是同名方法。

第十一天

第十一天

1.关于 cap() 函数的适用类型，下面说法正确的是()

- A. array
- B. slice
- C. map
- D. channel

参考答案及解析：ABD。知识点：cap()，cap() 函数不适用 map。

2.下面这段代码输出什么？

```
func main() {
    var i interface{}
    if i == nil {
        fmt.Println("nil")
        return
    }
    fmt.Println("not nil")
}
```

- A. nil
- B. not nil
- C. compilation error

参考答案及解析：A。当且仅当接口的动态值和动态类型都为 nil 时，接口类型值才为 nil。

3.下面这段代码输出什么？

```
func main() {
    s := make(map[string]int)
    delete(s, "h")
    fmt.Println(s["h"])
}
```

- A. runtime panic
- B. 0
- C. compilation error

参考答案及解析：B。删除 map 不存在的键值对时，不会报错，相当于没有任何作用；获取不存在的减值对时，返回值类型对应的零值，所以返回 0。

第十天

第十天

1.下面这段代码输出什么？

```
func main() {
    a := 5
    b := 8.1
    fmt.Println(a + b)
}
```

- A.13.1
- B.13
- C.compilation error

参考答案及解析：C。a 的类型是 int，b 的类型是 float，两个不同类型的数值不能相加，编译报错。

2.下面这段代码输出什么？

```
package main

import (
    "fmt"
)

func main() {
    a := [5]int{1, 2, 3, 4, 5}
    t := a[3:4:4]
    fmt.Println(t[0])
}
```

- A.3
- B.4
- C.compilation error

参考答案及解析：B。知识点：操作符 [i,j]。基于数组（切片）可以使用操作符 [i,j] 创建新的切片，从索引 i，到索引 j 结束，截取已有数组（切片）的任意部分，返回新的切片，新切片的值包含原数组（切片）的 i 索引的值，但是不包含 j 索引的值。i、j 都是可选的，i 如果省略，默认是 0，j 如果省略，默认是原数组（切片）的长度。i、j 都不能超过这个长度值。

假如底层数组的大小为 k，截取之后获得的切片的长度和容量的计算方法：长度：j-i，容量：k-i。

截取操作符还可以有第三个参数，形如 [i,j,k]，第三个参数 k 用来限制新切片的容量，但不能超过原数组（切片）的底层数组大小。截取获得的切片的长度和容量分别是：j-i、k-i。

所以例子中，切片 t 为 [4]，长度和容量都是 1。

3.下面这段代码输出什么？

```
func main() {
    a := [2]int{5, 6}
    b := [3]int{5, 6}
    if a == b {
        fmt.Println("equal")
    } else {
        fmt.Println("not equal")
    }
}
```

- A. compilation error
- B. equal
- C. not equal

参考答案及解析：A。Go 中的数组是值类型，可比较，另外一方面，数组的长度也是数组类型的组成部分，所以 a 和 b 是不同的类型，是不能比较的，所以编译错误。

第九天

第九天

1.关于channel，下面语法正确的是()

- A. var ch chan int
- B. ch := make(chan int)
- C. <- ch
- D. ch <-

参考答案及解析：ABC。A、B都是声明 channel；C 读取 channel；写 channel 是必须带上值，所以 D 错误。

2.下面这段代码输出什么？

```
type person struct {
    name string
}

func main() {
    var m map[person]int
    p := person{"mike"}
    fmt.Println(m[p])
}
```

- A.0
- B.1
- C.Compilation error

参考答案及解析：A。打印一个 map 中不存在的值时，返回元素类型的零值。这个例子中，m 的类型是 map[person]int，因为 m 中不存在 p，所以打印 int 类型的零值，即 0。

3.下面这段代码输出什么？

```
func hello(num ...int) {
    num[0] = 18
}

func main() {
    i := []int{5, 6, 7}
    hello(i...)
    fmt.Println(i[0])
}
```

- A.18

- B.5
- C.Compilation error

参考答案及解析：18。知识点：可变函数。

第八天

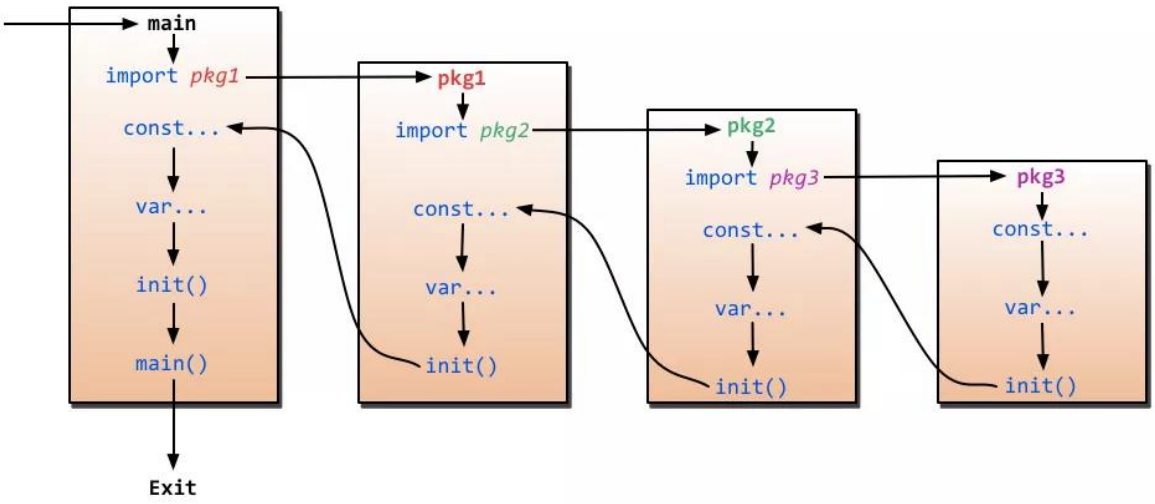
第八天

1.关于init函数，下面说法正确的是()

- A. 一个包中，可以包含多个 init 函数；
- B. 程序编译时，先执行依赖包的 init 函数，再执行 main 包内的 init 函数；
- C. main 包中，不能有 init 函数；
- D. init 函数可以被其他函数调用；

1.参考答案及解析：AB。关于 init() 函数有几个需要注意的地方：

- init() 函数是用于程序执行前做包的初始化的函数，比如初始化包里的变量等;
- 一个包可以出线多个 init() 函数,一个源文件也可以包含多个 init() 函数；
- 同一个包中多个 init() 函数的执行顺序没有明确定义，但是不同包的init函数是根据包导入的依赖关系决定的（看下图）；
- init() 函数在代码中不能被显示调用、不能被引用（赋值给函数变量），否则出现编译错误;
- 一个包被引用多次，如 A import B,C import B,A import C，B 被引用多次，但 B 包只会初始化一次；
- 引入包，不可出现死循环。即 A import B,B import A，这种情况编译失败；



2.下面这段代码输出什么以及原因？

```
func hello() []string {
    return nil
}

func main() {
    h := hello
    if h == nil {
        fmt.Println("nil")
    } else {
        fmt.Println("not nil")
    }
}
```

- A. nil
- B. not nil
- C. compilation error

答案及解析：B。这道题目里面，是将 hello() 赋值给变量 h，而不是函数的返回值，所以输出 not nil。

3.下面这段代码能否编译通过？如果可以，输出什么？

```
func GetValue() int {
    return 1
}

func main() {
    i := GetValue()
    switch i.(type) {
    case int:
        println("int")
    case string:
        println("string")
    case interface{}:
        println("interface")
    default:
        println("unknown")
    }
}
```

参考答案及解析：编译失败。考点：类型选择，类型选择的语法形如：i.(type)，其中 i 是接口，type 是固定关键字，需要注意的是，只有接口类型才可以使用类型选择。看下关于接口的文章。

第七天

第七天

1.关于字符串连接，下面语法正确的是？

- A. str := ‘abc’ + ‘123’
- B. str := “abc” + “123”
- str := ‘123’ + “abc”
- D. fmt.Sprintf(“abc%d”, 123)

参考答案及解析：BD。知识点：字符串连接。除了以上两种连接方式，还有 strings.Join()、buffer.WriteString()等。

2.下面这段代码能否编译通过？如果可以，输出什么？

```
const (
    x = iota
    _
    y
    z = "zz"
    k
    p = iota
)

func main()  {
    fmt.Println(x,y,z,k,p)
}
```

参考答案及解析：编译通过，输出：0 2 zz zz 5。知识点：iota 的使用。给大家贴篇文章，讲的很详细 https://www.cnblogs.com/zsy/p/5370052.html

3.下面赋值正确的是()

- A. var x = nil
- B. var x interface{} = nil
- C. var x string = nil
- D. var x error = nil

参考答案及解析：BD。知识点：nil 值。nil 只能赋值给指针、chan、func、interface、map 或 slice 类型的变量。强调下 D 选项的 error 类型，它是一种内置接口类型，看下方贴出的源码就知道，所以 D 是对的。

```
type error interface {
    Error() string
}
```

第六天

第六天

1.通过指针变量 p 访问其成员变量 name，有哪几种方式？

- A.p.name
- B.(&p).name
- C.(`*p`).name
- D.p->name

参考答案及解析：AC。& 取址运算符，* 指针解引用。

2.下面这段代码能否通过编译？如果通过，输出什么？

```
package main

import "fmt"

type MyInt1 int
type MyInt2 = int

func main() {
    var i int =0
    var i1 MyInt1 = i
    var i2 MyInt2 = i
    fmt.Println(i1,i2)
}
```

参考答案及解析：编译不通过，cannot use i (type int) as type MyInt1 in assignment。

这道题考的是类型别名与类型定义的区别。

第 5 行代码是基于类型 int 创建了新类型 MyInt1，第 6 行代码是创建了 int 的类型别名 MyInt2，注意类型别名的定义时 = 。所以，第 10 行代码相当于是将 int 类型的变量赋值给 MyInt1 类型的变量，Go 是强类型语言，编译当然不通过；而 MyInt2 只是 int 的别名，本质上还是 int，可以赋值。

第 10 行代码的赋值可以使用强制类型转化 var i1 MyInt1 = MyInt1(i).

第五天

第五天

1.下面这段代码能否通过编译？不能的话，原因是什么？如果通过，输出什么？

```
func main() {
    sn1 := struct {
        age  int
        name string
    }{age: 11, name: "qq"}
    sn2 := struct {
        age  int
        name string
    }{age: 11, name: "qq"}

    if sn1 == sn2 {
        fmt.Println("sn1 == sn2")
    }

    sm1 := struct {
        age int
        m   map[string]string
    }{age: 11, m: map[string]string{"a": "1"}}
    sm2 := struct {
        age int
        m   map[string]string
    }{age: 11, m: map[string]string{"a": "1"}}

    if sm1 == sm2 {
        fmt.Println("sm1 == sm2")
    }
}
```

参考答案及解析：编译不通过 invalid operation: sm1 == sm2

这道题目考的是结构体的比较，有几个需要注意的地方：

- 结构体只能比较是否相等，但是不能比较大小。
- 相同类型的结构体才能够进行比较，结构体是否相同不但与属性类型有关，还与属性顺序相关，sn3 与 sn1 就是不同的结构体；

```
    sn3:= struct {
            name string
            age  int
        }{age:11,name:"qq"}
```

- 如果 struct 的所有成员都可以比较，则该 struct 就可以通过 == 或 != 进行比较是否相等，比较时逐个项进行比较，如果每一项都相等，则两个结构体才相等，否则不相等；

那什么是可比较的呢，常见的有 bool、数值型、字符、指针、数组等，像切片、map、函数等是不能比较的。 具体可以参考 Go 说明文档。https://golang.org/ref/spec#Comparison_operators

第四天

第四天

1.下面这段代码能否通过编译，不能的话原因是什么；如果能，输出什么。

```
func main() {
    list := new([]int)
    list = append(list, 1)
    fmt.Println(list)
}
```

参考答案及解析：不能通过编译，new([]int) 之后的 list 是一个 `*[]int` 类型的指针，不能对指针执行 append 操作。可以使用 make() 初始化之后再用。同样的，map 和 channel 建议使用 make() 或字面量的方式初始化，不要用 new() 。

2.下面这段代码能否通过编译，如果可以，输出什么？

```
func main() {
    s1 := []int{1, 2, 3}
    s2 := []int{4, 5}
    s1 = append(s1, s2)
    fmt.Println(s1)
}
```

参考答案及解析：不能通过编译。append() 的第二个参数不能直接使用 slice，需使用 ... 操作符，将一个切片追加到另一个切片上：append(s1,s2...)。或者直接跟上元素，形如：append(s1,1,2,3)。

3.下面这段代码能否通过编译，如果可以，输出什么？

```
var(
    size := 1024
    max_size = size*2
)

func main() {
    fmt.Println(size,max_size)
}
```

参考答案及解析：不能通过编译。这道题的主要知识点是变量声明的简短模式，形如：x := 100。但这种声明方式有限制：

```
1. 必须使用显示初始化；
2. 不能提供数据类型，编译器会自动推导；
3. 只能在函数内部使用简短模式；
```

第三天

第三天

1.下面两段代码输出什么。

```
// 1.
func main() {
    s := make([]int, 5)
    s = append(s, 1, 2, 3)
    fmt.Println(s)
}

// 2.
func main() {
    s := make([]int,0)
    s = append(s,1,2,3,4)
    fmt.Println(s)
}
```

两段代码分别输出：

```
    [0 0 0 0 0 1 2 3]
    [1 2 3 4]
```

参考解析：这道题考的是使用 append 向 slice 添加元素，第一段代码常见的错误是 [1 2 3]，需要注意。

2.下面这段代码有什么缺陷

```
    func funcMui(x,y int)(sum int,error){
        return x+y,nil
    }
```

参考答案：第二个返回值没有命名。

参考解析：

在函数有多个返回值时，只要有一个返回值有命名，其他的也必须命名。如果有多个返回值必须加上括号()；如果只有一个返回值且命名也必须加上括号()。这里的第一个返回值有命名 sum，第二个没有命名，所以错误。

3.new() 与 make() 的区别

参考答案：

new(T) 和 make(T,args) 是 Go 语言内建函数，用来分配内存，但适用的类型不同。

new(T) 会为 T 类型的新值分配已置零的内存空间，并返回地址（指针），即类型为 `*T` 的值。换句话说就是，返回一个指针，该指针指向新分配的、类型为 T 的零值。适用于值类型，如数组、结构体等。

make(T,args) 返回初始化之后的 T 类型的值，这个值并不是 T 类型的零值，也不是指针 `*T` ，是经过初始化之后的 T 的引用。make() 只适用于 slice、map 和 channel.

第二天

第二天

下面这段代码输出什么，说明原因。

```
func main() {

    slice := []int{0,1,2,3}
    m := make(map[int]*int)

    for key,val := range slice {
        m[key] = &val
    }

    for k,v := range m {
        fmt.Println(k,"->",*v)
    }
}
```

直接给答案：

```
    0 -> 3
    1 -> 3
    2 -> 3
    3 -> 3
```

参考解析：这是新手常会犯的错误写法，for range 循环的时候会创建每个元素的副本，而不是元素的引用，所以 m[key] = &val 取的都是变量 val 的地址，所以最后 map 中的所有元素的值都是变量 val 的地址，因为最后 val 被赋值为3，所有输出都是3.

正确的写法:

```
func main() {

    slice := []int{0,1,2,3}
    m := make(map[int]*int)

    for key,val := range slice {
        value := val
        m[key] = &value
    }

    for k,v := range m {
        fmt.Println(k,"===>",*v)
    }
}
```

第一天

第一天

下面这段代码输出的内容

```
package main

import (
    "fmt"
)

func main() {
    defer_call()
}

func defer_call() {
    defer func() { fmt.Println("打印前") }()
    defer func() { fmt.Println("打印中") }()
    defer func() { fmt.Println("打印后") }()
    panic("触发异常")
}
```

看下答案，输出：

```
    打印后
    打印中
    打印前
    panic: 触发异常
```

参考解析：defer 的执行顺序是后进先出。当出现 panic 语句的时候，会先按照 defer 的后进先出的顺序执行，最后才会执行 panic