# **golang**基础

## 与其他语⾔相⽐，使⽤ **Go** 有什么好处？

1. 与其他作为学术实验开始的语⾔不同，Go 代码的设计是务实的。每个功能和语法决策都旨在让程 序员的⽣活 更轻松。

2. Golang 针对并发进⾏了优化，并且在规模上运⾏良好。

3. 由于单⼀的标准代码格式，Golang 通常被认为⽐其他语⾔更具可读性。

4. ⾃动垃圾收集明显⽐ Java 或 Python 更有效，因为它与程序同时执⾏。

## **Golang** 使⽤什么数据类型？

1. Method
2. Boolean
3. Numeric
4. String
5. Array
6. Slice
7. Struct
8. Pointer
9. Function
10. Interface
11. Map
12. Channe

## 整型切片有哪些初始化的方式

s:=make([]int,0)

s:= make([]int,5,10)

s:= []int{1,2,3,4,5}

**Go 当中同步锁有什么特点？作用是什么**

当一个 Goroutine（协程）获得了 Mutex 后，其他 Goroutine（协程）就只能乖 乖的等待，除非该 Goroutine 释放了该 Mutex。RWMutex 在读锁占用的情况下， 会阻止写，但不阻止读 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 同步锁的作用是保证资源在使用时的独有性，不会因为并发而导致数据错乱， 保证系统的稳定性。

## **Go** 语言当中 **Channel**（通道）有什么特点，需要注意什

**么？**

1. 如果给一个 nil 的 channel 发送数据，会造成永远阻塞。

2. 如果从一个 nil 的 channel 中接收数据，也会造成永久阻塞。

3. 给一个已经关闭的 channel 发送数据， 会引起 panic

4. 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。

## **Go** 语言当中 **Channel** 缓冲有什么特点？

无缓冲的 channel 是同步的，而有缓冲的 channel 是非同步的。

## **Go** 语言中 **cap** 函数可以作用于哪些内容？

1. array(数组)
2. slice(切片)
3. channel(通道)

## **Go Convey** 是什么？一般用来做什么？

1. go convey 是一个支持 Golang 的单元测试框架

2. go convey 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

3. go convey 提供了丰富的断言简化测试用例的编写 输入 non-empty interface

1. 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

2. 但是在此案例中，直接在main函数中的\*int类型的x的值是nil

## **Go** 语言当中 **new** 的作用是什么

new 创建一个该类型的实例，并且返回指向该实例的指针。new 函数是内建函 数，函数定义： func new(Type) \*Type

1. 使用 new 函数来分配空间

2. 传递给 new 函数的是一个类型，而不是一个值

3. 返回值是指向这个新分配的地址的指针

## **Go** 语言中 **make** 的作用是什么

make 的作用是为 slice, map or chan 的初始化 然后返回引用 make 函数是内 建函数，函数定义：

func make(Type, size IntegerType) Type

make(T, args)函数的目的和 new(T)不同 仅仅用于创建 slice, map, channel 而且返回类型是实例

**关于select机制**

1. select机制⽤来处理异步IO问题；

2. 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

3. golang在语⾔级别⽀持select关键字

## **Go** 程序中的包是什么

包(pkg)是 Go ⼯作区中包含 Go 源⽂件或其他包的⽬录。源⽂件中的每个函数、变量和类型都存储在链接包 中。每 个 Go 源⽂件都属于⼀个包，该包在⽂件顶部使⽤ package关键词声明。使用import导入包。

**Printf()，Sprintf()，FprintF() 都是格式化输出，有什么不**

**同**

1. Printf 是标准输出，一般是屏幕，也可以重定向

2. Sprintf()是把格式化字符串输出到指定的字符串中

3. Fprintf()是把格式化字符串输出到文件中

## **Go** 语言当中数组和切片的区别是什么？

1. 数组， 组固定长度。数组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，自动推算出大小， 大小不可改变。数组是通过值传递的

2. 切片， 可以改变长度。切片是轻量级的数据结构，三个属性，指针，长度，容量 不需要指定大小切 片是地址传递（引用传递）可以通过数组来初始化，也可以 通过内置函数 make()来初始化，初始 化的时候 len=cap，然后进行扩容

## 拷贝大切片一定比小切片代价大吗？

切片底层结构

type SliceHeader struct { Data uintptr

Len int

Cap int

}

所有切片的大小相同；三个字段（一个 uintptr，两个int）。切片中的第一个字是指向切片底层数组的指 针，这是切片的存储空间，第二个字段是切片的长度，第三个字段是容量。将一个 slice 变量分配给另一 个变量只会复制三个机器字。所以 拷贝大切片跟小切片的代价应该是一样的。

## **json**包变量不加**tag**会怎么样？

1. 如果变量首字母小写，则为private。无论如何不能转，因为json包里认为私有变量为不可导出的

2. 如果变量首字母大写，则为public，不加tag，可以正常转为json里的字段，json内字段名跟结构体 内字段原名一致

3. 如果变量首字母大写，则为public，加了tag，从struct转json的时候，json的字段名就是tag里的字 段名，原字段名已经没用

## **Log**包线程安全吗

Golang的标准库提供了log的机制，但是该模块的功能较为简单（看似简单，其实他有他的设计思路）。在输 出的 位置做了线程安全的保护

**Go 语言当中数组和切片在传递的时候的区别是什么？**

1. 数组是值传递

2. 切片看上去像是引用传递，但其实是值传递

## **Goroutine**和线程的区别**?**

从调度上看，goroutine的调度开销远远⼩于线程调度开销。 OS的线程由OS内核调度，每隔⼏毫秒，⼀ 个硬件时钟中断发到CPU，CPU调⽤⼀个调度器内核函数。这个函数暂 停当前正在运⾏的线程，把他的 寄存器信息保存到内存中，查看线程列表并决定接下来运⾏哪⼀个线程，再从内存 中恢复线程的注册表 信息，最后继续执⾏选中的线程。这种线程切换需要⼀个完整的上下⽂切换：即保存⼀个线程 的状态到 内存，再恢复另外⼀个线程的状态，最后更新调度器的数据结构。某种意义上，这种操作还是很慢的。 Go运⾏的时候包涵⼀个⾃⼰的调度器，这个调度器使⽤⼀个称为⼀个M:N调度技术，m个goroutine到n 个os线程 （可以⽤GOMAXPROCS来控制n的数量），Go的调度器不是由硬件时钟来定期触发的，⽽是 由特定的go语⾔结构 来触发的，他不需要切换到内核语境，所以调度⼀个goroutine⽐调度⼀个线程的 成本低很多。 从栈空间上，goroutine的栈空间更加动态灵活。 每个OS的线程都有⼀个固定⼤⼩的栈内 存，通常是2MB，栈内存⽤于保存在其他函数调⽤期间哪些正在执⾏或者 临时暂停的函数的局部变量。 这个固定的栈⼤⼩，如果对于goroutine来说，可能是⼀种巨⼤的浪费。作为对⽐ goroutine在⽣命周期 开始只有⼀个很⼩的栈，典型情况是2KB, 在go程序中，⼀次创建⼗万左右的goroutine也不 罕⻅

（2KB\*100,000=200MB）。⽽且goroutine的栈不是固定⼤⼩，它可以按需增⼤和缩⼩，最⼤限制可以

到 1GB。 goroutine没有⼀个特定的标识。 在⼤部分⽀持多线程的操作系统和编程语⾔中，线程有⼀个 独特的标识，通常是⼀个整数或者指针，这个特性可以 让我们构建⼀个线程的局部存储，本质是⼀个全 局的map，以线程的标识作为键，这样每个线程可以独⽴使⽤这个 map存储和获取值，不受其他线程⼲ 扰。 goroutine中没有可供程序员访问的标识，原因是⼀种纯函数的理念，不希望滥⽤线程局部存储导致

⼀个不健康的 超距作⽤，即函数的⾏为不仅取决于它的参数，还取决于运⾏它的线程标识。

## **Go** 语言是如何实现切片扩容的？

func main() {

arr := make([]int, 0)

for i := 0; i < 2000; i++ {

fmt.Println("len 为", len(arr), "cap 为", cap(arr)) arr = append(arr, i)

}

}

我们可以看下结果

依次是 0,1,2,4,8,16,32,64,128,256,512,1024

但到了 1024 之后,就变成了 1024,1280,1696,2304

每次都是扩容了四分之一左右

## 看下面代码的 **defer** 的执行顺序是什么？ **defer** 的作用和

**特 点是什么？**

defer 的作用是：

1. 你只需要在调用普通函数或方法前加上关键字 defer，就完成了 defer 所需要 的语法。

2. 当 defer 语句被执行时，跟在 defer 后面的函数会被延迟执行。

3. 直到 包含该 defer 语句的函数执行完毕时，defer 后的函数才会被执行，不论包含 defer 语句的函 数是通过 return 正常结束，还是由于 panic 导致的异常结束。

4. 你可以在一个函数中执行多条 defer 语句，它们的执行顺序与声明顺序相反。

defer 的常用场景：

1. defer 语句经常被用于处理成对的操作，如打开、关闭、连接、断开连接、 加锁、释放锁。

2. 通过 defer 机制，不论函数逻辑多复杂，都能保证在任何执行路径下，资 源被释放。

3. 释放资源的 defer 应该直接跟在请求资源的语句后

## **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 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：

1. 指向底层数组的指针

2. 切片的长度

3. 切片的容量

## **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：

1. 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

2. 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

3. 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

4. 如果最终容量计算值溢出，则最终容量就是新申请容量

## 扩容前后的 **Slice** 是否相同？

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。

## **Golang** 的参数传递、引用类型

Go 语言中所有的传参都是值传递（传值），都是一个副本，一个拷贝。因为拷 贝的内容有时候是非引 用类型（int、string、struct 等这些），这样就在函 数中就无法修改原内容数据；有的是引用类型（指 针、map、slice、chan 等这 些），这样就可以修改原内容数据。 Golang 的引用类型包括 slice、map 和 channel。它们有复杂的内部结构，除 了申请内存外，还需要初始化相关属性。内置函数 new 计算类 型大小，为其分 配零值内存，返回指针。而 make 会被编译器翻译成具体的创建函数，由其分 配内存和 初始化成员结构，返回对象而非指针

## **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)。

## **Golang Map** 如何扩容

1. 双倍扩容：扩容采取了一种称为“渐进式”的方式，原有的 key 并不会一 次性搬迁完毕，每次最多只 会搬迁 2 个 bucket

2. 等量扩容：重新排列，极端情况下，重新排列也解决不了，map 存储就会蜕 变成链表，性能大大 降低，此时哈希因子 hash0 的设置，可以降低此类极 端场景的发生

## **Golang Map** 查找

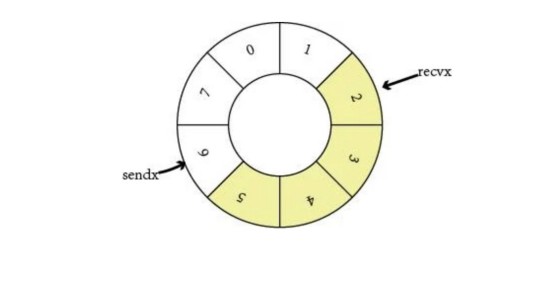
Go 语言中 map 采用的是哈希查找表，由一个 key 通过哈希函数得到哈希值，64 位系统中就生成一个 64bit 的哈希值，由这个哈希值将 key 对应存到不同的桶 （bucket）中，当有多个哈希映射到相同的的桶 中时，使用链表解决哈希冲 突。 细节：key 经过 hash 后共 64 位，根据 hmap 中 B 的值，计算它到 底要落在哪个 桶时，桶的数量为 2^B，如 B=5，那么用 64 位最后 5 位表示第几号桶，在用 hash 值的 高 8 位确定在 bucket 中的存储位置，当前 bmap 中的 bucket 未找 到，则查询对应的 overflow bucket，对应位置有数据则对比完整的哈希值， 确定是否是要查找的数据。如果当前 map 处于数据搬移状 态，则优先从 oldbuckets 查找。

**介绍一下 Channel**

Go 语言中，不要通过共享内存来通信，而要通过通信来实现内存共享。Go 的 CSP(Communicating Sequential Process)并发模型，中文可以叫做通信顺序进 程，是通过 goroutine 和 channel 来实现的。 channel 收发遵循先进先出 FIFO 的原则。分为有缓冲区和无缓冲区，channel 中包括 buffer、sendx 和 recvx 收发的位置(ring buffer 记录实现)、sendq、 recv。当 channel 因为缓冲区不足而阻塞了队列，则 使用双向链表存储

## **Channel** 的 **ring buffer** 实现

channel 中使用了 ring buffer（环形缓冲区) 来缓存写入的数据。ring buffer 有很多好处，而且非常适 合用来实现 FIFO 式的固定长度队列。 在 channel 中，ring buffer 的实现如下：



上图展示的是一个缓冲区为 8 的 channel buffer，recvx 指向最早被读取的数 据，sendx 指向再次写入

时插入的位置。

## **for**循环**select**时，如果通道已经关闭会怎么样？如果

**select中的case只有一个，又会怎么样？**

1. for循环select时，如果其中一个case通道已经关闭，则每次都会执行到这个case。

2. 如果select里边只有一个case，而这个case被关闭了，则会出现死循环。

# 代码分析

## 下⾯的代码是有问题的，请说明原因

type People struct { Name string

}

func (p \*People) String() string { return fmt.Sprintf("print: %v", p)

}

func main() {

p := &People{} p.String()

}

解析： 在golang中 String() string ⽅法实际上是实现了 String 的接⼝的，该接⼝定义在 fmt/print.go

中：

type Stringer interface { String() string

}

在使⽤ fmt 包中的打印⽅法时，如果类型实现了这个接⼝，会直接调⽤。⽽题⽬中打 印 p 的时候会直接 调⽤ p 实现的 String() ⽅法，然后就产⽣了循环调⽤

## 交替打印数字和字⺟

1. 问题描述

使⽤两个 goroutine 交替打印序列，⼀个 goroutine 打印数字， 另外⼀ 个 goroutine 打印字⺟， 最终 效果如下：

12AB34CD56EF78GH910IJ1112KL1314MN1516OP1718QR1920ST2122UV2324WX2526YZ2728

2. 解题思路

问题很简单，使⽤ channel 来控制打印的进度。使⽤两个 channel ，来分别控制数字和 字⺟的打印序 列， 数字打印完成后通过 channel 通知字⺟打印, 字⺟打印完成后通知数 字打印，然后周⽽复始的⼯ 作。

3. 源码参考

letter, number := make(chan bool), make(chan bool)

wait := sync.WaitGroup{} go func() {

i := 1

for {

select {

case <-number: fmt.Print(i) i++ fmt.Print(i) i++

letter <- true break

default:

break

}

}

}()

wait.Add(1)

go func(wait \*sync.WaitGroup) {

str := "ABCDEFGHIJKLMNOPQRSTUVWXYZ" i := 0

for {

select {

case <-letter:

if i >= strings.Count(str, "")-1 { wait.Done()

return

}

fmt.Print(str[i : i+1]) i++

if i >= strings.Count(str, "") { i = 0

}

fmt.Print(str[i : i+1]) i++

number <- true break

default:

break

}

}

}(&wait)

number <- true wait.Wait()

4. 源码解析

这⾥⽤到了两个 channel 负责通知，letter负责通知打印字⺟的goroutine来打印字⺟， number⽤来通 知打印数字的goroutine打印数字。 wait⽤来等待字⺟打印完成后退出循环。

## 下列代码是否会触发异常

func Test59(t \*testing.T) { runtime.GOMAXPROCS(1)

intChan := make(chan int, 1)

stringChan := make(chan string, 1) intChan <- 1

stringChan <- "hello"

select {

case value := <-intChan: fmt.Println(value)

case value := <-stringChan:

panic(value)

}

}

不⼀定，当两个chan同时有值时，select 会随机选择⼀个可⽤通道做收发操作

## 如何在运行时检查变量类型？

类型开关(Type Switch)是在运行时检查变量类型的最佳方式。类型开关按类型 而不是值来评估变量。每 个 Switch 至少包含一个 case 用作条件语句，如果没 有一个 case 为真，则执行 default。

var value interface{} value = "abcd"

switch value.(type) { case string:

fmt.Println("string") case int:

fmt.Println("int")

case bool:

fmt.Println("bool") default:

fmt.Println("default")

}

## 判断字符串中字符是否全都不同

1. 问题描述

请实现⼀个算法，确定⼀个字符串的所有字符【是否全都不同】。这⾥我们要求【不允 许使⽤额外的存 储结构】。 给定⼀个string，请返回⼀个bool值,true代表所有字符全都 不同，false代表存在相同的字 符。 保证字符串中的字符为【ASCII字符】。字符串的⻓ 度⼩于等于【3000】

2. 解题思路

这⾥有⼏个重点，第⼀个是 ASCII字符 ， ASCII字符 字符⼀共有256个，其中128个是常 ⽤字符，可以在 键盘上输⼊。128之后的是键盘上⽆法找到的。 然后是全部不同，也就是字符串中的字符没有重复的， 再次，不准使⽤额外的储存结 构，且字符串⼩于等于3000。 如果允许其他额外储存结构，这个题⽬很好 做。如果不允许的话，可以使⽤golang内置 的⽅式实现。

3. 源码参考

func isUniqueString(s string) bool { if strings.Count(s, "") > 3000 {

return false

}

for \_, v := range s { if v > 127 {

return false

}

if strings.Count(s, string(v)) > 1 { return false

}

}

return true

}

4. 源码解析

以上两种⽅法都可以实现这个算法。 第⼀个⽅法使⽤的是golang内置⽅法 strings.Count ,可以⽤来判断 在⼀个字符串中包含 的另外⼀个字符串的数量。 第⼆个⽅法使⽤的是golang内置⽅法 strings.Index 和 strings.LastIndex ，⽤来判断指 定字符串在另外⼀个字符串的索引未知，分别是第⼀次发现位置和最后 发现位置

## 翻转字符串

1. 问题描述

请实现⼀个算法，在不使⽤【额外数据结构和储存空间】的情况下，翻转⼀个给定的字 符串(可以使⽤单 个过程变量)。 给定⼀个string，请返回⼀个string，为翻转后的字符串。保证字符串的⻓度⼩于等于 5000。

2. 解题思路 翻转字符串其实是将⼀个字符串以中间字符为轴，前后翻转，即将str[len]赋值给str[0], 将str[0] 赋值

str[len]。

3. 源码参考：

func reverString(s string) (string, bool) { str := []rune(s)

l := len(str)

if l > 5000 {

return s, false

}

for i := 0; i < l/2; i++ {

str[i], str[l-1-i] = str[l-1-i], str[i]

}

return string(str), true

}

4. 源码解析：

以字符串⻓度的1/2为轴，前后赋值

## 翻转含有中文、数字、英文字母的字符串

1. 解题思路

解题思路 翻转字符串其实是将⼀个字符串以中间字符为轴，前后翻转

2. 源码参考

package main

import "fmt"

func main() {

src := "nick 今天天气很好啊！1234" dst := reverse([]rune(src))

fmt.Printf("%v\n", string(dst))

}

func reverse(s []rune) []rune { length := len(s)

for i := 0; i < length/2; i++ {

s[i], s[length-1-i] = s[length-1-i], s[i]

}

return s

}

3. 源码解析

rune关键字，从golang源码中看出，它是int32的别名（-2^31 ~ 2^31-1），比起byte（-128～127）， 可表示更多的字符。 由于rune可表示的范围更大，所以能处理一切字符，当然也包括中文字符。在平时计算中文字符，可用 rune。

因此将字符串转为rune的切片

## 判断两个给定的字符串排序后是否⼀致

1. 问题描述

给定两个字符串，请编写程序，确定其中⼀个字符串的字符重新排列后，能否变成另⼀ 个字符串。 这⾥ 规定【⼤⼩写为不同字符】，且考虑字符串重点空格。给定⼀个string s1和⼀个string s2，请返回⼀个 bool，代表两串是否重新排列后可相同。 保证两串的 ⻓度都⼩于等于5000。

2. 解题思路

⾸先要保证字符串⻓度⼩于5000。之后只需要⼀次循环遍历s1中的字符在s2是否都存 在即可。

3. 源码参考

func isRegroup(s1, s2 string) bool { sl1 := len([]rune(s1))

sl2 := len([]rune(s2))

if sl1 > 5000 || sl2 > 5000 || sl1 != sl2 { return false

}

for \_, v := range s1 {

if strings.Count(s1, string(v)) != strings.Count(s2, string(v)) { return false

}

}

return true

}

4. 源码解析

使⽤golang内置⽅法 strings.Count 来判断字符是否⼀致

## 字符串替换问题

1. 问题描述

请编写⼀个⽅法，将字符串中的空格全部替换为“%20”。 假定该字符串有⾜够的空间存 放新增的字符， 并且知道字符串的真实⻓度(⼩于等于1000)，同时保证字符串由【⼤⼩ 写的英⽂字⺟组成】。 给定⼀个 string为原始的串，返回替换后的string。

2. 解题思路 两个问题，第⼀个是只能是英⽂字⺟，第⼆个是替换空格。

3. 源码参考

func replaceBlank(s string) (string, bool) { if len([]rune(s)) > 1000 {

return s, false

}

for \_, v := range s {

if string(v) != " " && unicode.IsLetter(v) == false { return s, false

}

}

return strings.Replace(s, " ", "%20", -1), true

}

4. 源码解析

使⽤了golang内置⽅法 unicode.IsLetter 判断字符是否是字⺟，之后使⽤ strings.Replace 来替换 空格

## 下列代码有什么问题

const i = 100 var j = 123 func main() {

fmt.Println(&j, j) fmt.Println(&i, i)

}

go语言中，常量无法寻址，是不能进行取地址操作的

## 下列代码输出什么

func Test62(t \*testing.T) {

x := []string{"a", "b", "c"} for v := range x { fmt.Print(v)

}

}

对于切片，range一个返回值时，返回的是切片的下标即索引。返回两个值时，第一个是下标，第二个是 值。 对于集合，range一个返回值时，返回的是集合的key值。返回两个值时，第一个是key，第二个是值 因此，以上代码输出 012

## 下列代码输出什么

func main() {

var x \*int = nil Foo(x)

}

func Foo(x interface{}) { if x == nil {

fmt.Println("empty interface") return

}

fmt.Println(x)

fmt.Println("non-empty interface")

}

**机器⼈坐标问题**

1. 问题描述

有⼀个机器⼈，给⼀串指令，L左转 R右转，F前进⼀步，B后退⼀步，问最后机器⼈的 坐标，最开始， 机器⼈位于 0 0，⽅向为正Y。 可以输⼊重复指令n ： ⽐如 R2(LF) 这 个等于指令 RLFLF。 问最后机器⼈ 的坐标是多少？

2. 解题思路 这⾥的⼀个难点是解析重复指令。主要指令解析成功，计算坐标就简单了

3. 源码参考

package main

import (

"unicode"

)

const (

Left = iota Top

Right

Bottom

)

func main() {

println(move("R2(LF)", 0, 0, Top))

}

func move(cmd string, x0 int, y0 int, z0 int) (x, y, z int) { x, y, z = x0, y0, z0

repeat := 0 repeatCmd := ""

for \_, s := range cmd {

switch {

case unicode.IsNumber(s):

repeat = repeat\*10 + (int(s) - '0') case s == ')':

for i := 0; i < repeat; i++ {

x, y, z = move(repeatCmd, x, y, z)

}

repeat = 0 repeatCmd = ""

case repeat > 0 && s != '(' && s != ')': repeatCmd = repeatCmd + string(s)

case s == 'L':

z = (z + 1) % 4

case s == 'R':

z = (z - 1 + 4) % 4

case s == 'F': switch {

case z == Left || z == Right:

x = x - z + 1

case z == Top || z == Bottom: y = y - z + 2

}

case s == 'B': switch {

case z == Left || z == Right:

x = x + z - 1

case z == Top || z == Bottom: y = y + z - 2

}

}

}

return

}

4. 源码解析

这⾥使⽤三个值表示机器⼈当前的状况，分别是：x表示x坐标，y表示y坐标，z表示当 前⽅向。 L、R 命 令会改变值z，F、B命令会改变值x、y。 值x、y的改变还受当前的z值 影响。 如果是重复指令，那么将 重复次数和重复的指令存起来递归调⽤即可

## **nil**切片和空切片的区别

1. nil切片表示该切片尚未初始化，仅有一个类型声明，并不存在示例。引用数组指针地址为0。

2. 空切片表示该切片以及被初始化，有一个切实存在的对象，只是该切片的元素个数为0。引用的数 组指针地址也是一个存在的内存地址

例如：

package main

import (

"fmt" "reflect" "unsafe"

)

func main() {

var s1 []int

s2 := make([]int, 0)

s4 := make([]int, 0)

fmt.Printf("s1 pointer:%+v, s2 pointer:%+v, s4 pointer:%+v, \n", \* (\*reflect.SliceHeader)(unsafe.Pointer(&s1)), \*(\*reflect.SliceHeader) (unsafe.Pointer(&s2)), \*(\*reflect.SliceHeader)(unsafe.Pointer(&s4)))

fmt.Printf("%v\n", (\*(\*reflect.SliceHeader)(unsafe.Pointer(&s1))).Data == (\* (\*reflect.SliceHeader)(unsafe.Pointer(&s2))).Data)

fmt.Printf("%v\n", (\*(\*reflect.SliceHeader)(unsafe.Pointer(&s2))).Data == (\* (\*reflect.SliceHeader)(unsafe.Pointer(&s4))).Data)

}

切片数据结构

type SliceHeader struct {

Data uintptr //引用数组指针地址 Len int // 切片的目前使用长度 Cap int // 切片的容量

}

## 字符串转成**byte**切片，无内存拷贝的实现

1. 源码参考

package main

import (

"fmt" "reflect"

"unsafe"

)

func main() {

1. := "abc"

ssh := \*(\*reflect.StringHeader)(unsafe.Pointer(&a)) fmt.Printf("%+v\n", ssh)

1. := \*(\*[]byte)(unsafe.Pointer(&ssh)) fmt.Printf("%+v\n", b)

fmt.Printf("%+v\n", \*(\*reflect.SliceHeader)(unsafe.Pointer(&b)))

}

2. 源码分析

StringHeader 是字符串在go的底层结构

type StringHeader struct { Data uintptr

Len int

}

SliceHeader 是切片在go的底层结构

type SliceHeader struct { Data uintptr

Len int

Cap int

}

在底层转换二者，只需要把 StringHeader 的地址强转成 SliceHeader 就行。

unsafe.Pointer(&a)方法可以得到变量a的地址 (*reflect.StringHeader)(unsafe.Pointer(&a)) 可以把字符串a转成底层结构的形式 (*[]byte)(unsafe.Pointer(&ssh)) 可以把ssh底层结构体转成byte的切片的指针 再通过 \*转为指针指向的实际内容

## 如何获取结构体**tag**信息

tag 信息可以通过反射（reflect包）获取。 示例代码：

package main

import (

"fmt" "reflect"

)

type J struct {

a string //小写无tag

b string `json:"B"` //小写+tag

1. string //大写无tag
2. string `json:"DD" otherTag:"good"` //大写+tag

}

func printTag(stru interface{}) {

t := reflect.TypeOf(stru).Elem()

for i := 0; i < t.NumField(); i++ {

fmt.Printf("结构体内第%v个字段 %v 对应的json tag是 %v , 还有otherTag？ = %v

\n", i+1, t.Field(i).Name, t.Field(i).Tag.Get("json"), t.Field(i).Tag.Get("otherTag"))

}

}

func main() {

j := J{

a: "1",

b: "2",

C: "3",

D: "4",

}

printTag(&j)

}

## 下列代码会造成死循环吗？

package main

import "fmt"

func main() {

s := []int{1,2,3,4,5}

for \_, v:=range s { s =append(s, v)

fmt.Printf("len(s)=%v\n",len(s))

}

}

不会死循环，for range其实是golang的语法糖，在循环开始前会获取切片的长度 len(切片)，然后再执行 len(切片)次数的循环

## **uintptr**和**unsafe.Pointer**的区别

1. unsafe.Pointer只是单纯的通用指针类型，用于转换不同类型指针，它不可以参与指针运算；

2. uintptr是用于指针运算的，GC 不把 uintptr 当指针，也就是说 uintptr 无法持有对象， uintptr 类 型的目标会被回收；

3. unsafe.Pointer 可以和 普通指针 进行相互转换

4. unsafe.Pointer 可以和 uintptr 进行相互转换 示例代码：

package main

import ( "fmt" "unsafe"

)

type W struct { b int32

c int64

}

func main() {

var w \*W = new(W)

//这时w的变量打印出来都是默认值0，0 fmt.Println(w.b,w.c)

//现在我们通过指针运算给b变量赋值为10

b := unsafe.Pointer(uintptr(unsafe.Pointer(w)) + unsafe.Offsetof(w.b))

\*((\*int)(b)) = 10

//此时结果就变成了10，0 fmt.Println(w.b,w.c)

}

1. uintptr(unsafe.Pointer(w)) 获取了 w 的指针起始值
2. unsafe.Offsetof(w.b) 获取 b 变量的偏移量

3. 两个相加就得到了 b 的地址值，将通用指针 Pointer 转换成具体指针 ((\*int)(b))，通过 \* 符号取值， 然后赋值。*((*int)(b)) 相当于把 (\*int)(b) 转换成 int 了，最后对变量重新赋值成 10

## 知道**golang**的内存逃逸吗？什么情况下会发生内存逃逸？

golang程序变量会携带有一组校验数据，用来证明它的整个生命周期是否在运行时完全可知。如果变量 通过了这些校验，它就可以在栈上分配。否则就说它 逃逸 了，必须在堆上分配 能引起变量逃逸到堆上的典型情况

1. 在方法内把局部变量指针返回 局部变量原本应该在栈中分配，在栈中回收。但是由于返回时被外部 引用，因此其生命周期大于栈，则溢出

2. 发送指针或带有指针的值到 channel 中。 在编译时，是没有办法知道哪个 goroutine 会在 channel

上接收数据。所以编译器没法知道变量什么时候才会被释放

3. 在一个切片上存储指针或带指针的值。 一个典型的例子就是 []\*string 。这会导致切片的内容逃 逸。尽管其后面的数组可能是在栈上分配的，但其引用的值一定是在堆上

4. slice 的背后数组被重新分配了，因为 append 时可能会超出其容量( cap )。 slice 初始化的地方在 编译时是可以知道的，它最开始会在栈上分配。如果切片背后的存储要基于运行时的数据进行扩 充，就会在堆上分配

5. 在 interface 类型上调用方法。 在 interface 类型上调用方法都是动态调度的 —— 方法的真正实现 只能在运行时知道。想像一个 io.Reader 类型的变量 r , 调用 r.Read(b) 会使得 r 的值和切片b 的背 后存储都逃逸掉，所以会在堆上分配

代码示例：

package main import "fmt" type A struct {

s string

}

// 这是上面提到的 "在方法内把局部变量指针返回" 的情况

func foo(s string) \*A { a := new(A)

a.s = s

return a //返回局部变量a,在C语言中妥妥野指针，但在go则ok，但a会逃逸到堆

}

func main() {

1. := foo("hello")
2. := a.s + " world" c := b + "!"

fmt.Println(c)

}

执行go build -gcflags=-m main.go

go build -gcflags=-m main.go

# command-line-arguments

./main.go:7:6: can inline foo

./main.go:13:10: inlining call to foo

./main.go:16:13: inlining call to fmt.Println

/var/folders/45/qx9lfw2s2zzgvhzg3mtzkwzc0000gn/T/go- build409982591/b001/\_gomod\_.go:6:6: can inline init.0

./main.go:7:10: leaking param: s

./main.go:8:10: new(A) escapes to heap

./main.go:16:13: io.Writer(os.Stdout) escapes to heap

./main.go:16:13: c escapes to heap

./main.go:15:9: b + "!" escapes to heap

./main.go:13:10: main new(A) does not escape

./main.go:14:11: main a.s + " world" does not escape

./main.go:16:13: main []interface {} literal does not escape

<autogenerated>:1: os.(\*File).close .this does not escape

1. ./main.go:8:10: new(A) escapes to heap 说明 new(A) 逃逸了,符合上述提到的常见情况中的第一种

1. ./main.go:14:11: main a.s + " world" does not escape 说明 b 变量没有逃逸，因为它只在方法内 存在，会在方法结束时被回收
2. ./main.go:15:9: b + "!" escapes to heap 说明 c 变量逃逸，通过fmt.Println(a ...interface{})打印的 变量，都会发生逃逸

## 怎么避免内存逃逸

在runtime/stubs.go:133有个函数叫noescape。noescape可以在逃逸分析中隐藏一个指针。让这个指 针在逃逸分析中不会被检测为逃逸

// 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.

// USE CAREFULLY!

//go:nosplit

func noescape(p unsafe.Pointer) unsafe.Pointer { x := uintptr(p)

return unsafe.Pointer(x ^ 0)

}

示例代码

package main

import ( "unsafe"

)

type A struct { S \*string

}

func (f \*A) String() string { return \*f.S

}

type ATrick struct { S unsafe.Pointer

}

func (f \*ATrick) String() string { return \*(\*string)(f.S)

}

func NewA(s string) A { return A{S: &s}

}

func NewATrick(s string) ATrick {

return ATrick{S: noescape(unsafe.Pointer(&s))}

}

func noescape(p unsafe.Pointer) unsafe.Pointer { x := uintptr(p)

return unsafe.Pointer(x ^ 0)

}

func main() { s := "hello"

f1 := NewA(s)

f2 := NewATrick(s) s1 := f1.String() s2 := f2.String()

\_ = s1 + s2

}

执行go build -gcflags=-m main.go

go build -gcflags=-m main.go

# command-line-arguments

./main.go:11:6: can inline (\*A).String

./main.go:19:6: can inline (\*ATrick).String

./main.go:23:6: can inline NewA

./main.go:31:6: can inline noescape

./main.go:27:6: can inline NewATrick

./main.go:28:29: inlining call to noescape

./main.go:36:6: can inline main

./main.go:38:14: inlining call to NewA

./main.go:39:19: inlining call to NewATrick

./main.go:39:19: inlining call to noescape

./main.go:40:17: inlining call to (\*A).String

./main.go:41:17: inlining call to (\*ATrick).String

/var/folders/45/qx9lfw2s2zzgvhzg3mtzkwzc0000gn/T/go- build763863171/b001/\_gomod\_.go:6:6: can inline init.0

./main.go:11:7: leaking param: f to result ~r0 level=2

./main.go:19:7: leaking param: f to result ~r0 level=2

./main.go:24:16: &s escapes to heap

./main.go:23:13: moved to heap: s

./main.go:27:18: NewATrick s does not escape

./main.go:28:45: NewATrick &s does not escape

./main.go:31:15: noescape p does not escape

./main.go:38:14: main &s does not escape

./main.go:39:19: main &s does not escape

./main.go:40:10: main f1 does not escape

./main.go:41:10: main f2 does not escape

./main.go:42:9: main s1 + s2 does not escape

其中主要看中间一小段

./main.go:24:16: &s escapes to heap

./main.go:23:13: moved to heap: s

//这个是NewA中的，逃逸了

./main.go:27:18: NewATrick s does not escape // NewATrick里的s的却没逃逸

./main.go:28:45: NewATrick &s does not escape

1. 上段代码对A和ATrick同样的功能有两种实现：他们包含一个 string ，然后用 String() 方法返回这个 字符串。但是从逃逸分析看ATrick 版本没有逃逸

2. noescape() 函数的作用是遮蔽输入和输出的依赖关系。使编译器不认为 p 会通过 x 逃逸， 因为

uintptr() 产生的引用是编译器无法理解的

3. 内置的 uintptr 类型是一个真正的指针类型，但是在编译器层面，它只是一个存储一个 指针地址 的

int 类型。代码的最后一行返回 unsafe.Pointer 也是一个 int

1. noescape() 在 runtime 包中使用 unsafe.Pointer 的地方被大量使用。如果作者清楚被

unsafe.Pointer 引用的数据肯定不会被逃逸，但编译器却不知道的情况下，这是很有用的

5. 实际项目中不建议使用unsafe包！ 毕竟包的名字就叫做 unsafe, 而且源码中的注释也写明了 USE CAREFULLY!

## **go struct**能不能⽐较

1. 相同struct类型的可以⽐较

2. 不同struct类型的不可以⽐较,编译都不过，类型不匹配

package main import "fmt" func main() {

type A struct { a int

}

type B struct { a int

}

a := A{1}

//b := A{1}

b := B{1}

if a == b { fmt.Println("a == b")

}else{

fmt.Println("a != b")

}

}

// output

// command-line-arguments [command-line-arguments.test]

// ./.go:14:7: invalid operation: a == b (mismatched types A and B)

## 下⾯代码能运⾏吗？为什么

type Param map[string]interface{} type Show struct {

Param

}

func main1() { s := new(Show)

s.Param["RMB"] = 10000

}

解析 共发现两个问题：

1. main 函数不能加数字。

2. new 关键字⽆法初始化 Show 结构体中的 Param 属性，所以直接 对 s.Param 操作会出错

## 请说出下⾯代码存在什么问题

type student struct { Name string

}

func zhoujielun(v interface{}) { switch msg := v.(type) {

case \*student, student:

msg.Name

}

}

解析：

golang中有规定， switch type 的 case T1 ，类型列表只有⼀个，那么 v := m.(type) 中的 v 的类型就是

T1类型。

如果是 case T1, T2 ，类型列表中有多个，那 v 的类型还是多对应接⼝的类型，也就 是 m 的类型。 所以这⾥ msg 的类型还是 interface{} ，所以他没有 Name 这个字段，编译阶段就会 报错。具体解释

⻅： <https://golang.org/ref> /spec#Type\_switches

## **Go** ⽀持什么形式的类型转换？将整数转换为浮点数

Go ⽀持显式类型转换以满⾜其严格的类型要求。

i := 55 //int

j := 67.8 //float64

sum := i + int(j)//j is converted to int

## **Go** 语言当中值传递和地址传递（引用传递）如何运用？有

**什 么区别？举例说明**

1. 值传递只会把参数的值复制一份放进对应的函数，两个变量的地址不同， 不可相互修改。

2. 地址传递(引用传递)会将变量本身传入对应的函数，在函数中可以对该变 量进行值内容的修改。

## 写出打印的结果

type People struct {

name string `json:"name"`

}

func main() { js := `{ "name":"11"

}`

var p People

err := json.Unmarshal([]byte(js), &p) if err != nil {

fmt.Println("err: ", err) return

}

fmt.Println("people: ", p)

}

解析：

按照 golang 的语法，⼩写开头的⽅法、属性或 struct 是私有的，同样，在 json 解 码或转码的时候也⽆ 法上线私有属性的转换。 题⽬中是⽆法正常得到 People 的 name 值的。⽽且，私有属性 name 也不应 该加 json 的标签。

# 进阶**-**并发编程

## **go**语⾔的并发机制以及它所使⽤的**CSP**并发模型

CSP模型是上个世纪七⼗年代提出的,不同于传统的多线程通过共享内存来通信，CSP讲究的是“以通信的

⽅式来共享 内存”。⽤于描述两个独⽴的并发实体通过共享的通讯 channel(管道)进⾏通信的并发模型。 CSP中channel是第⼀ 类对象，它不关注发送消息的实体，⽽关注与发送消息时使⽤的channel。 Golang中channel 是被单独创建并且可以在进程之间传递，它的通信模式类似于 boss-worker 模式的，

⼀个实体 通过将消息发送到channel 中，然后⼜监听这个 channel 的实体处理，两个实体之间是匿名

的，这个就实现实体中 间的解耦，其中 channel 是同步的⼀个消息被发送到 channel 中，最终是⼀定要 被另外的实体消费掉的，在实现 原理上其实类似⼀个阻塞的消息队列。

Goroutine 是Golang实际并发执⾏的实体，它底层是使⽤协程(coroutine)实现并发，coroutine是⼀种 运⾏在⽤户 态的⽤户线程，类似于 greenthread，go底层选择使⽤coroutine的出发点是因为，它具有 以下特点 ：

1. ⽤户空间 避免了内核态和⽤户态的切换导致的成本。

2. 可以由语⾔和框架层进⾏调度。

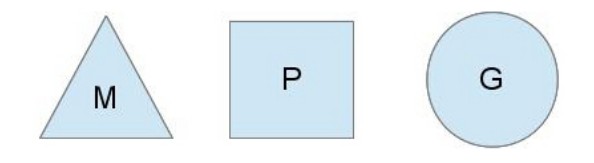
3. 更⼩的栈空间允许创建⼤量的实例

Golang中的Goroutine的特性:

Golang内部有三个对象： P对象(processor) 代表上下⽂（或者可以认为是cpu），M(work thread)代表

⼯作线 程，G对象（goroutine）. 正常情况下⼀个cpu对象启⼀个⼯作线程对象，线程去检查并执⾏ goroutine对象。碰到goroutine对象阻塞的时 候，会启动⼀个新的⼯作线程，以充分利⽤cpu资源。 所 有有时候线程对象会⽐处理器对象多很多

我们⽤如下图分别表示P、M、G:

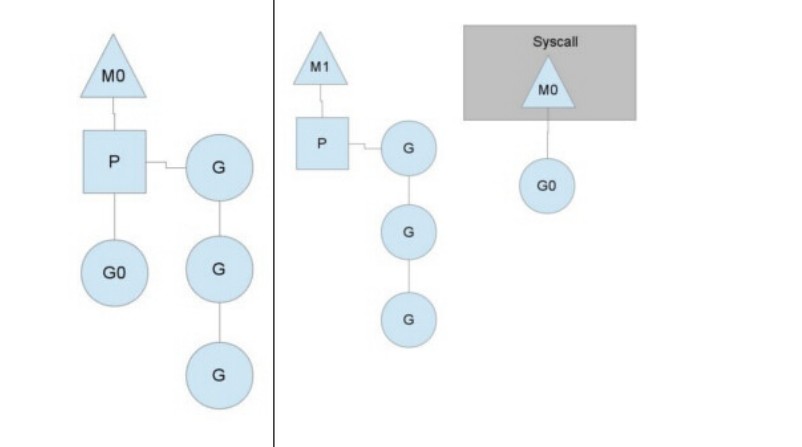


G（Goroutine） ：我们所说的协程，为⽤户级的轻量级线程，每个Goroutine对象中的sched保存着其 上下⽂信 息.

M（Machine） ：对内核级线程的封装，数量对应真实的CPU数（真正⼲活的对象）. P（Processor） ：即为G和M的调度对象，⽤来调度G和M之间的关联关系，其数量可通过 GOMAXPROCS()来设 置，默认为核⼼数.

在单核情况下，所有Goroutine运⾏在同⼀个线程（M0）中，每⼀个线程维护⼀个上下⽂（P），任何时 刻，⼀个 上下⽂中只有⼀个Goroutine，其他Goroutine在runqueue中等待。 ⼀个Goroutine运⾏完⾃

⼰的时间⽚后，让出上下⽂，⾃⼰回到runqueue中（如下图所示）。 当正在运⾏的G0阻塞的时候（可 以需要IO），会再创建⼀个线程（M1），P转到新的线程中去运⾏



当M0返回时，它会尝试从其他线程中“偷”⼀个上下⽂过来，如果没有偷到，会把Goroutine放到Global

runqueue 中去，然后把⾃⼰放⼊线程缓存中。 上下⽂会定时检查Global runqueue。 Golang是为并发⽽⽣的语⾔，Go语⾔是为数不多的在语⾔层⾯实现并发的语⾔；也正是Go语⾔的并发 特性，吸引 了全球⽆数的开发者。

Golang的CSP并发模型，是通过Goroutine和Channel来实现的。

Goroutine 是Go语⾔中并发的执⾏单位。有点抽象，其实就是和传统概念上的”线程“类似，可以理解为” 线程“。 Channel是Go语⾔中各个并发结构体(Goroutine)之前的通信机制。通常Channel，是各个 Goroutine之间通信的” 管道“，有点类似于Linux中的管道。 通信机制channel也很⽅便，传数据⽤channel <- data，取数据⽤<-channel。 在通信过程中，传数据channel <- data和取数据<-channel必然会成对出现，因为这边传，那边取，两个 goroutine之间才会实现通信。 ⽽且不管传还是取，必阻塞，直到另外的goroutine传或者取为⽌

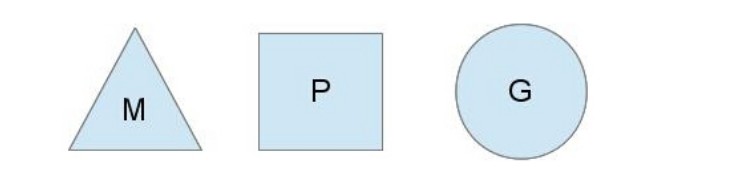
## **Golang** 中 **Goroutine** 如何调度

goroutine是Golang语⾔中最经典的设计，也是其魅⼒所在，goroutine的本质是协程，是实现并⾏计算 的核⼼。 goroutine使⽤⽅式⾮常的简单，只需使⽤go关键字即可启动⼀个协程，并且它是处于异步⽅ 式运⾏，你不需要等 它运⾏完成以后在执⾏以后的代码。

go func()//通过go关键字启动⼀个协程来运⾏函数 协程：

协程拥有⾃⼰的寄存器上下⽂和栈。协程调度切换时，将寄存器上下⽂和栈保存到其他地⽅，在切回来

的时候，恢 复先前保存的寄存器上下⽂和栈。 因此，协程能保留上⼀次调⽤时的状态（即所有局部状态 的⼀个特定组合），每 次过程重⼊时，就相当于进⼊上⼀次调⽤的状态，换种说法：进⼊上⼀次离开时 所处逻辑流的位置。 线程和进程的 操作是由程序触发系统接⼝，最后的执⾏者是系统；协程的操作执⾏ 者则是⽤户⾃身程序，goroutine也是协程。 groutine能拥有强⼤的并发实现是通过GPM调度模型实现.



Go的调度器内部有四个重要的结构：M，P，S，Sched，如上图所示（Sched未给出）.

1. M:M代表内核级线程，⼀个M就是⼀个线程，goroutine就是跑在M之上的；M是⼀个很⼤的结构，

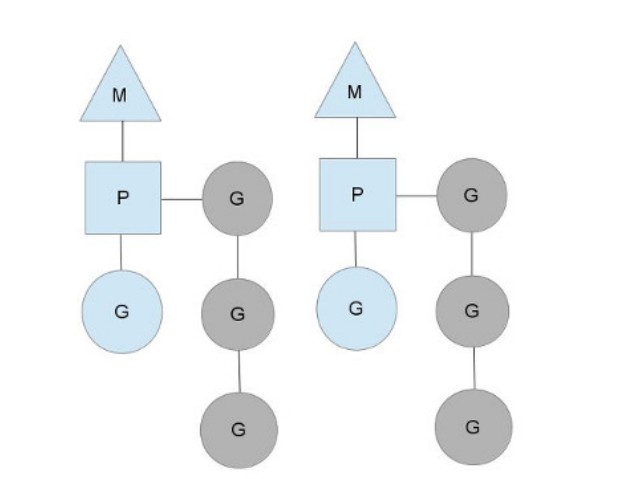
⾥⾯维护 ⼩对象内存cache（mcache）、当前执⾏的goroutine、随机数发⽣器等等⾮常多的信息

2. G:代表⼀个goroutine，它有⾃⼰的栈，instruction pointer和其他信息（正在等待的channel等

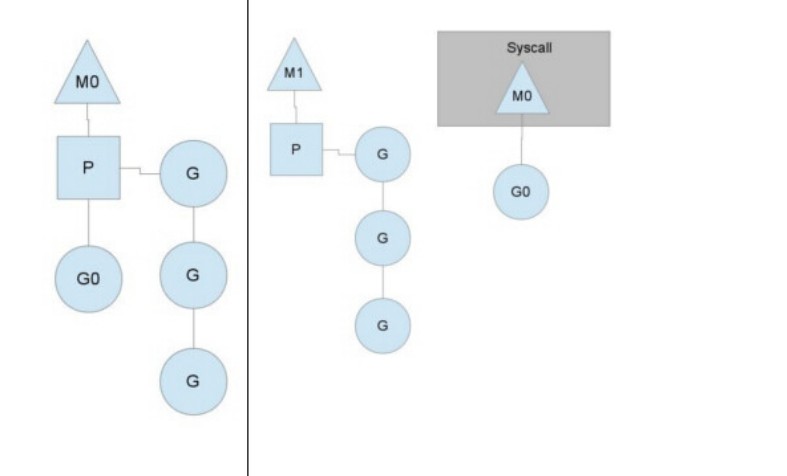
等），⽤于调 度。

3. P:P全称是Processor，处理器，它的主要⽤途就是⽤来执⾏goroutine的，所以它也维护了⼀个 goroutine队 列，⾥⾯存储了所有需要它来执⾏的goroutine

4. Sched：代表调度器，它维护有存储M和G的队列以及调度器的⼀些状态信息等 调度实现：

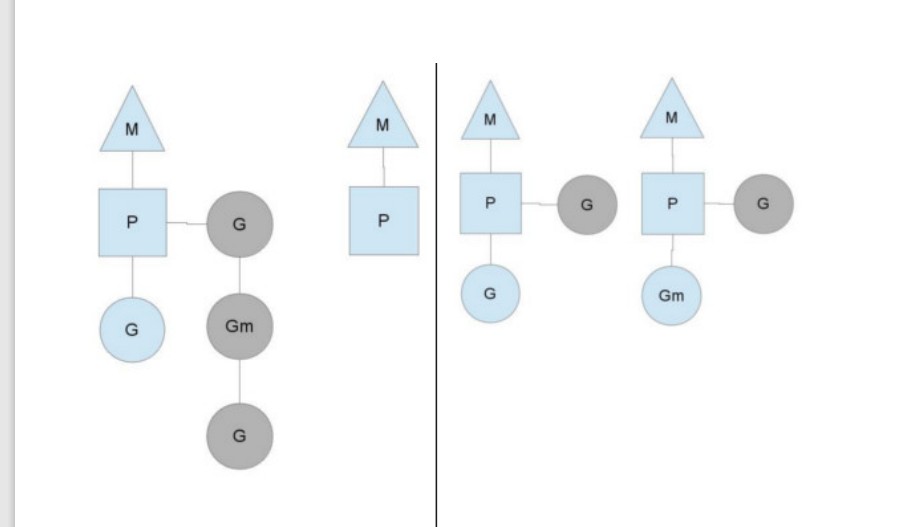


从上图中可以看到，有2个物理线程M，每⼀个M都拥有⼀个处理器P，每⼀个也都有⼀个正在运⾏的 goroutine。P 的数量可以通过GOMAXPROCS()来设置，它其实也就代表了真正的并发度，即有多少个 goroutine可以同时运⾏。 图中灰⾊的那些goroutine并没有运⾏，⽽是出于ready的就绪态，正在等待 被调度。P维护着这个队列（称之为 runqueue），Go语⾔⾥，启动⼀个goroutine很容易：go function 就⾏，所以每有⼀个go语句被执⾏， runqueue队列就在其末尾加⼊⼀个goroutine，在下⼀个调度点， 就从runqueue中取出（如何决定取哪个 goroutine？）⼀个goroutine执⾏。 当⼀个OS线程M0陷⼊阻 塞时，P转⽽在运⾏M1，图中的M1可能是正被创建，或者从线程缓存中取出



当MO返回时，它必须尝试取得⼀个P来运⾏goroutine，⼀般情况下，它会从其他的OS线程那⾥拿⼀个

P过来， 如 果没有拿到的话，它就把goroutine放在⼀个global runqueue⾥，然后⾃⼰睡眠（放⼊线程 缓存⾥）。所有的P也 会周期性的检查global runqueue并运⾏其中的goroutine，否则global runqueue 上的goroutine永远⽆法执⾏。 另⼀种情况是P所分配的任务G很快就执⾏完了（分配不均），这就导致 了这个处理器P很忙，但是其他的P还有任 务，此时如果global runqueue没有任务G了，那么P不得不从 其他的P⾥拿⼀些G来执⾏。



通常来说，如果P从其他的P那⾥要拿任务的话，⼀般就拿run queue的⼀半，这就确保了每个OS线程都 能充分的 使⽤。

## **Golang** 中常⽤的并发模型

Golang 中常⽤的并发模型有三种 ：

1. 通过channel通知实现并发控制

⽆缓冲的通道指的是通道的⼤⼩为0，也就是说，这种类型的通道在接收前没有能⼒保存任何值，它要求

发送 goroutine 和接收 goroutine 同时准备好，才可以完成发送和接收操作。 从上⾯⽆缓冲的通道定义 来看，发送 goroutine 和接收 gouroutine 必须是同步的，同时准备后，如果没有同时准 备好的话，先 执⾏的操作就会阻塞等待，直到另⼀个相对应的操作准备好为⽌。这种⽆缓冲的通道我们也称之为同 步 通道

func main() {

ch := make(chan struct{}) go func() {

fmt.Println("start working") time.Sleep(time.Second \* 1) ch <- struct{}{}

}()

<-ch fmt.Println("finished")

}

当主 goroutine 运⾏到 <-ch 接受 channel 的值的时候，如果该 channel 中没有数据，就会⼀直阻塞等 待，直到有 值。 这样就可以简单实现并发控制

2. 通过sync包中的WaitGroup实现并发控制

Goroutine是异步执⾏的，有的时候为了防⽌在结束mian函数的时候结束掉Goroutine，所以需要同步 等待，这个 时候就需要⽤ WaitGroup了，在 sync 包中，提供了 WaitGroup ，它会等待它收集的所有 goroutine 任务全部完成。在WaitGroup⾥主要有三个⽅法:

1. Add, 可以添加或减少 goroutine的数量. 2. Done, 相当于Add(-1).

3. Wait, 执⾏后会堵塞主线程，直到WaitGroup ⾥的值减⾄0.

在主 goroutine 中 Add(delta int) 索要等待goroutine 的数量。 在每⼀个 goroutine 完成后 Done() 表示 这⼀个 goroutine 已经完成，当所有的 goroutine 都完成后，在主 goroutine 中 WaitGroup 返回

func main(){

var wg sync.WaitGroup var urls = []string{

["http://www.golang.org/](http://www.golang.org/)", "<http://www.google.com/>",

}

for \_, url := range urls { wg.Add(1)

go func(url string) {

defer wg.Done() http.Get(url)

}(url)

}

wg.Wait()

}

在Golang官⽹中对于WaitGroup介绍是 A WaitGroup must not be copied after first use ,在 WaitGroup 第⼀次使⽤后，不能被拷⻉

应⽤示例:

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", i)

wg.Done()

}(wg, i)

}

wg.Wait() fmt.Println("exit")

}

运⾏:

i:1i:3i:2i:0i:4fatal error: all goroutines are asleep - deadlock! goroutine 1 [semacquire]:

sync.runtime\_Semacquire(0xc000094018)

/home/keke/soft/go/src/runtime/sema.go:56 +0x39 sync.(\*WaitGroup).Wait(0xc000094010)

/home/keke/soft/go/src/sync/waitgroup.go:130 +0x64 main.main()

/home/keke/go/Test/wait.go:17 +0xab

exit status 2

它提示所有的 goroutine 都已经睡眠了，出现了死锁。这是因为 wg 给拷⻉传递到了 goroutine 中，导 致只有 Add 操作，其实 Done操作是在 wg 的副本执⾏的。 因此 Wait 就死锁了。 这个第⼀个修改⽅式: 将匿名函数中 wg 的传⼊类型改为 \*sync.WaitGrou,这样就能引⽤到正确的WaitGroup了。 这 个第⼆个 修改⽅式:将匿名函数中的 wg 的传⼊参数去掉，因为Go⽀持闭包类型，在匿名函数中可以直接使⽤外⾯ 的 wg 变量

3. 在Go 1.7 以后引进的强⼤的Context上下⽂，实现并发控制

通常,在⼀些简单场景下使⽤ channel 和 WaitGroup 已经⾜够了，但是当⾯临⼀些复杂多变的⽹络并发 场景下 channel 和 WaitGroup 显得有些⼒不从⼼了。 ⽐如⼀个⽹络请求 Request，每个 Request 都需 要开启⼀个 goroutine 做⼀些事情，这些 goroutine ⼜可能会开启其他的 goroutine，⽐如数据库和RPC 服务。 所以我们需要 ⼀种可以跟踪 goroutine 的⽅案，才可以达到控制他们的⽬的，这就是Go语⾔为 我们提供的 Context，称之为上 下⽂⾮常贴切，它就是goroutine 的上下⽂。 它是包括⼀个程序的运⾏ 环境、现场和快照等。每个程序要运⾏时， 都需要知道当前程序的运⾏状态，通常Go 将这些封装在⼀ 个 Context ⾥，再将它传给要执⾏的 goroutine

context 包主要是⽤来处理多个 goroutine 之间共享数据，及多个 goroutine 的管理。 context 包的核

⼼是 struct Context，接⼝声明如下：

// A Context carries a deadline, cancelation signal, and request-scoped values

// across API boundaries. Its methods are safe for simultaneous use by multiple

// goroutines.

type Context interface {

// Done returns a channel that is closed when this `Context` is canceled

// or times out.

Done() <-chan struct{}

// Err indicates why this Context was canceled, after the Done channel

// is closed. Err() error

// Deadline returns the time when this Context will be canceled, if any. Deadline() (deadline time.Time, ok bool)

// Value returns the value associated with key or nil if none.

Value(key interface{}) interface{}

}

Done() 返回⼀个只能接受数据的channel类型，当该context关闭或者超时时间到了的时候，该channel 就会有⼀个 取消信号 Err() 在Done() 之后，返回context 取消的原因。 Deadline() 设置该context cancel 的时间点 Value() ⽅法允许 Context 对象携带request作⽤域的数据，该数据必须是线程安全的。 Context 对象是线程安全的，你可以把⼀个 Context 对象传递给任意个数的 gorotuine，对它执⾏ 取消 操作时， 所有 goroutine 都会接收到取消信号。 ⼀个 Context 不能拥有 Cancel ⽅法，同时我们也只能 Done channel 接收数据。 其中的原因是⼀致的：接收取消 信号的函数和发送信号的函数通常不是⼀ 个。 典型的场景是：⽗操作为⼦操作操作启动 goroutine，⼦操作也就不能取消⽗操作

## 并发编程概念是什么？

并⾏是指两个或者多个事件在同⼀时刻发⽣；并发是指两个或多个事件在同⼀时间间隔发⽣。 并⾏是在 不同实体上的多个事件，并发是在同⼀实体上的多个事件。在⼀台处理器上“同时”处理多个任务，在多台 处理器上同时处理多个任务。如hadoop分布式集群 并发偏重于多个任务交替执⾏，⽽多个任务之间有 可能还是串⾏的。⽽并⾏是真正意义上的“同时执⾏”。 并发编程是指在⼀台处理器上“同时”处理多个任 务。并发是在同⼀实体上的多个事件。多个事件在同⼀时间间隔发 ⽣。并发编程的⽬标是充分的利⽤处 理器的每⼀个核，以达到最⾼的处理性能。

## **Mutex** 几种状态

1. mutexLocked — 表示互斥锁的锁定状态；

2. mutexWoken — 表示从正常模式被从唤醒；

3. mutexStarving — 当前的互斥锁进入饥饿状态；

4. waitersCount — 当前互斥锁上等待的 Goroutine 个数；

## **Mutex** 正常模式和饥饿模式

正常模式（非公平锁）

正常模式下，所有等待锁的 goroutine 按照 FIFO（先进先出）顺序等待。唤醒 的 goroutine 不会直接 拥有锁，而是会和新请求 goroutine 竞争锁。新请求的 goroutine 更容易抢占：因为它正在 CPU 上执 行，所以刚刚唤醒的 goroutine 20 有很大可能在锁竞争中失败。在这种情况下，这个被唤醒的 goroutine 会加入 到等待队列的前面。

饥饿模式（公平锁）

为了解决了等待 goroutine 队列的长尾问题 饥饿模式下，直接由 unlock 把锁交给等待队列中排在第一 位的 goroutine (队 头)，同时，饥饿模式下，新进来的 goroutine 不会参与抢锁也不会进入自旋状 态， 会直接进入等待队列的尾部。这样很好的解决了老的 goroutine 一直抢不 到锁的场景。 饥饿模式的触发 条件：当一个 goroutine 等待锁时间超过 1 毫秒时，或者当前 队列只剩下一个 goroutine 的时候， Mutex 切换到饥饿模式。

总结

对于两种模式，正常模式下的性能是最好的，goroutine 可以连续多次获取 锁，饥饿模式解决了取锁公 平的问题，但是性能会下降，这其实是性能和公平 的一个平衡模式。

## **Mutex** 允许自旋的条件

1. 锁已被占用，并且锁不处于饥饿模式。

2. 积累的自旋次数小于最大自旋次数（active\_spin=4）。

3. CPU 核数大于 1。

4. 有空闲的 P。

5. 当前 Goroutine 所挂载的 P 下，本地待运行队列为空。

## **RWMutex** 实现

通过记录 readerCount 读锁的数量来进行控制，当有一个写锁的时候，会将读 锁数量设置为负数 1<<30。

目的是让新进入的读锁等待之前的写锁释放通知读 锁。同样的当有写锁进行抢占时，也会等待之前的读锁都释 放完毕，才会开始 21 进行后续的操作。 而等写锁释放完之后，会将值重新加上 1<<30, 并通知刚才 新进 入的读锁（rw.readerSem），两者互相限制。

**RWMutex 注意事项**

1. RWMutex 是单写多读锁，该锁可以加多个读锁或者一个写锁

2. 读锁占用的情况下会阻止写，不会阻止读，多个 Goroutine 可以同时获取 读锁 3. 写锁会阻止其他 Goroutine（无论读和写）进来，整个锁由该 Goroutine 独占 4. 适用于读多写少的场景

5. RWMutex 类型变量的零值是一个未锁定状态的互斥锁

6. RWMutex 在首次被使用之后就不能再被拷贝

7. RWMutex 的读锁或写锁在未锁定状态，解锁操作都会引发 panic

8. RWMutex 的一个写锁去锁定临界区的共享资源，如果临界区的共享资源已 被（读锁或写锁）锁 定，这个写锁操作的 goroutine 将被阻塞直到解锁

9. RWMutex 的读锁不要用于递归调用，比较容易产生死锁

10. RWMutex 的锁定状态与特定的 goroutine 没有关联。一个 goroutine 可 以 RLock（Lock），另一 个 goroutine 可以 RUnlock（Unlock）

11. 写锁被解锁后，所有因操作锁定读锁而被阻塞的 goroutine 会被唤醒，并 都可以成功锁定读锁

12. 读锁被解锁后，在没有被其他读锁锁定的前提下，所有因操作锁定写锁而 被阻塞的 Goroutine，其 中等待时间最长的一个 Goroutine 会被唤醒

## **Cond** 是什么

Cond 实现了一种条件变量，可以使用在多个 Reader 等待共享资源 ready 的场 景（如果只有一读一 写，一个锁或者 channel 就搞定了） 22 每个 Cond 都会关联一个 Lock（\*sync.Mutex or

\*sync.RWMutex），当修改条 件或者调用 Wait 方法时，必须加锁，保护 condition。

## **Broadcast** 和 **Signal** 区别

func (c \*Cond) Broadcast()

Broadcast 会唤醒所有等待 c 的 goroutine。 调用 Broadcast 的时候，可以加锁，也可以不加锁。

func (c \*Cond) Signal()

Signal 只唤醒 1 个等待 c 的 goroutine。 调用 Signal 的时候，可以加锁，也可以不加锁。

## **Cond** 中 **Wait** 使用

func (c \*Cond) Wait()

Wait()会自动释放 c.L 锁，并挂起调用者的 goroutine。之后恢复执行， Wait()会在返回时对 c.L 加锁。 除非被 Signal 或者 Broadcast 唤醒，否则 Wait()不会返回。 由于 Wait()第一次恢复时，C.L 并没有加 锁，所以当 Wait 返回时，调用者通常 并不能假设条件为真。如下代码：。 取而代之的是, 调用者应该在 循环中调用 Wait。（简单来说，只要想使用 condition，就必须加锁。）

c.L.Lock()

for !condition() { c.Wait()

}

... make use of condition ... c.L.Unlock()

## **WaitGroup** 用法

一个 WaitGroup 对象可以等待一组协程结束。使用方法是：

1. main 协程通过调用 wg.Add(delta int) 设置 worker 协程的个数，然后创 建 worker 协程；

2. worker 协程执行结束以后，都要调用 wg.Done()；

3. main 协程调用 wg.Wait() 且被 block，直到所有 worker 协程全部执行结束 后返回。

## **WaitGroup** 实现原理

1. WaitGroup 主要维护了 2 个计数器，一个是请求计数器 v，一个是等待计数 器 w，二者组成一个

64bit 的值，请求计数器占高 32bit，等待计数器占低 32bit。

2. 每次 Add 执行，请求计数器 v 加 1，Done 方法执行，等待计数器减 1，v 为 0 时通过信号量唤醒

Wait()

## 什么是 **sync.Once**

1. Once 可以用来执行且仅仅执行一次动作，常常用于单例对象的初始化场 景。

2. Once 常常用来初始化单例资源，或者并发访问只需初始化一次的共享资 源，或者在测试的时候初 始化一次测试资源。

3. sync.Once 只暴露了一个方法 Do，你可以多次调用 Do 方法，但是只有第 一次调用 Do 方法时 f 参 数才会执行，这里的 f 是一个无参数无返回值 的函数。

## 什么操作叫做原子操作

原子操作即是进行过程中不能被中断的操作，针对某个值的原子操作在被进行 的过程中，CPU 绝不会再 去进行其他的针对该值的操作。为了实现这样的严谨 性，原子操作仅会由一个独立的 CPU 指令代表和完 成。原子操作是无锁的，常 常直接通过 CPU 指令直接实现。 事实上，其它同步技术的实现常常依赖于 原 子操作。

## 原子操作和锁的区别

原子操作由底层硬件支持，而锁则由操作系统的调度器实现。 锁应当用来保护一段逻辑，对于一个变量 更新的保护。 原子操作通常执行上会更有效率，并且更能利用计算机多核的优势，如果要更 新的是一个 复合对象，则应当使用 atomic.Value 封装好的实现

## 什么是 **CAS**

CAS 的全称为 Compare And Swap，直译就是比较交换。是一条 CPU 的原子指 令，其作用是让 CPU 先进行比较两个值是否相等，然后原子地更新某个位置的 值，其实现方式是给予硬件平台的汇编指令， 在 intel 的 CPU 中，使用的 cmpxchg 指令，就是说 CAS 是靠硬件实现的，从而在硬件层面提升效率。 简述过程是这样：

假设包含 3 个参数内存位置(V)、预期原值(A)和新值(B)。V 表示要更新变量的 值，E 表示预期值，N 表 示新值。仅当 V 值等于 E 值时，才会将 V 的值设为 N， 如果 V 值和 E 值不同，则说明已经有其他线程在 做更新，则当前线程什么都不 做，最后 CAS 返回当前 V 的真实值。CAS 操作时抱着乐观的态度进行的， 它总 是认为自己可以成功完成操作。基于这样的原理，CAS 操作即使没有锁，也可 以发现其他线程对于 当前线程的干扰。

## **sync.Pool** 有什么用

对于很多需要重复分配、回收内存的地方，sync.Pool 是一个很好的选择。频 繁地分配、回收内存会给 GC 带来一定的负担，严重的时候会引起 CPU 的毛 刺。而 sync.Pool 可以将暂时将不用的对象缓存起 来，待下次需要的时候直 接使用，不用再次经过内存分配，复用对象的内存，减轻 GC 的压力，提升系 统的性能。

# 进阶**-Go Runtime**

## **Goroutine** 定义

Golang 在语言级别支持协程，称之为 Goroutine。Golang 标准库提供的所有 系统调用操作(包括所有 的同步 I/O 操作)，都会出让 CPU 给其他 Goroutine。这让 Goroutine 的切换管理不依赖于系统的线程 和进程，也不依 赖于 CPU 的核心数量，而是交给 Golang 的运行时统一调度。

## **GMP** 指的是什么

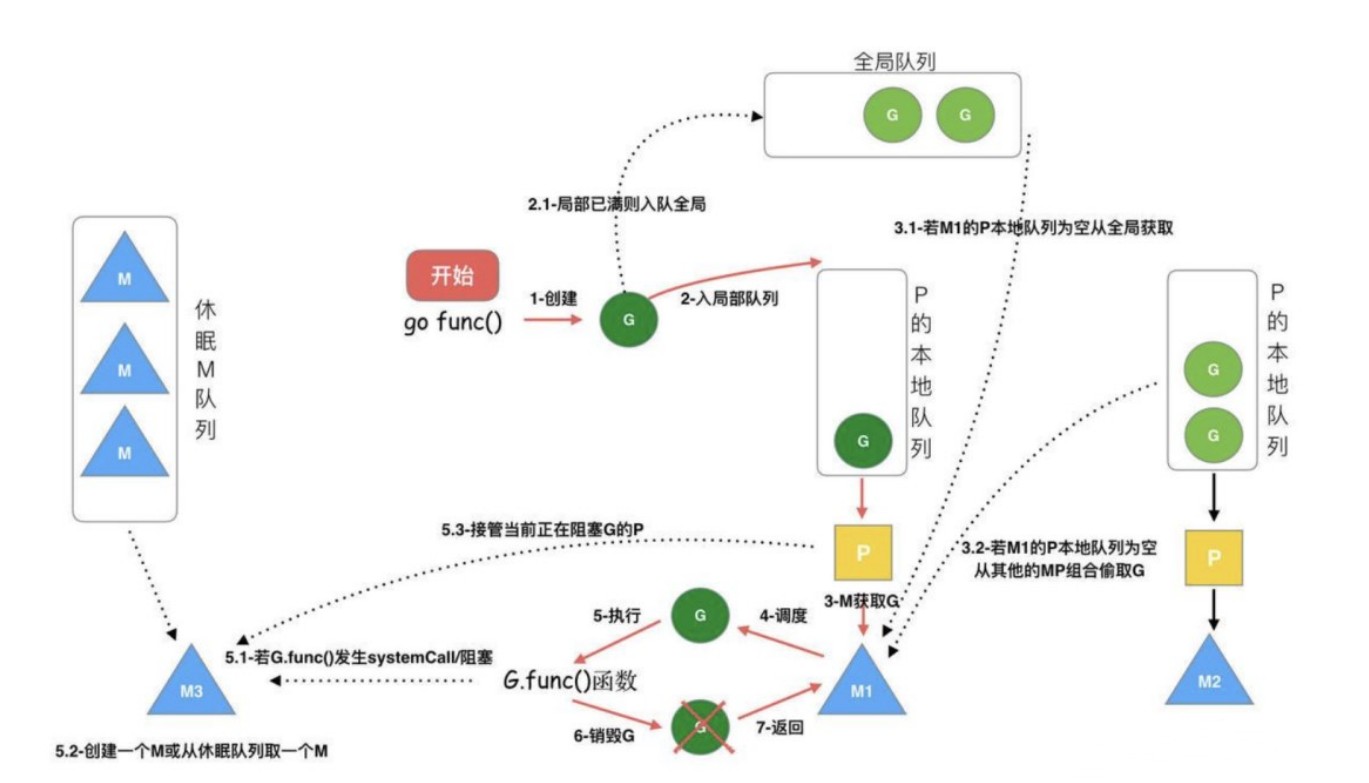
G（Goroutine）：我们所说的协程，为用户级的轻量级线程，每个 Goroutine 对象中的 sched 保存着 其上下文信息。

M（Machine）：对内核级线程的封装，数量对应真实的 CPU 数（真正干活的对 象）。 P（Processor）：即为 G 和 M 的调度对象，用来调度 G 和 M 之间的关联关系， 其数量可通过 GOMAXPROCS()来设置，默认为核心数。

## **1.0** 之前 **GM** 调度模型

调度器把 G 都分配到 M 上，不同的 G 在不同的 M 并发运行时，都需要向系统申 请资源，比如堆栈内 存等，因为资源是全局的，就会因为资源竞争照成很多性 能损耗。为了解决这一的问题 go 从 1.1 版本 引入，在运行时系统的时候加入 p 对象，让 P 去管理这个 G 对象，M 想要运行 G，必须绑定 P，才能运 行 P 所管理 的 G。 GM 调度存在的问题： 1．单一全局互斥锁（Sched.Lock）和集中状态存储 2． Goroutine 传递问题（M 经常在 M 之间传递”可运行”的 goroutine） 3．每个 M 做内存缓存，导致内存 占用过高，数据局部性较差 4．频繁 syscall 调用，导致严重的线程阻塞/解锁，加剧额外的性能损耗。

## **GMP** 调度流程



1. 每个 P 有个局部队列，局部队列保存待执行的 goroutine（流程 2），当 M 绑定的 P 的的局部队列 已经满了之后就会把 goroutine 放到全局队列（流 程 2-1）

2. 每个 P 和一个 M 绑定，M 是真正的执行 P 中 goroutine 的实体（流程 3）， M 从绑定的 P 中的局 部队列获取 G 来执行

3. 当 M 绑定的 P 的局部队列为空时，M 会从全局队列获取到本地队列来执行 G （流程 3.1），当从 全局队列中没有获取到可执行的 G 时候，M 会从其他 P 的局部队列中偷取 G 来执行（流程 3.2）， 这种从其他 P 偷的方式称为 work stealing

4. 当 G 因系统调用（syscall）阻塞时会阻塞 M，此时 P 会和 M 解绑即 hand off，并寻找新的 idle 的

M，若没有 idle 的 M 就会新建一个 M（流程 5.1）

5. 当 G 因 channel 或者 network I/O 阻塞时，不会阻塞 M，M 会寻找其他 runnable 的 G；当阻塞 的 G 恢复后会重新进入 runnable 进入 P 队列等待执 行

## **GMP** 中 **work stealing** 机制

获取 P 本地队列，当从绑定 P 本地 runq 上找不到可执行的 g，尝试从全局链 表中拿，再拿不到从 netpoll 和事件池里拿，最后会从别的 P 里偷任务。P 此时去唤醒一个 M。P 继续执行其它的程序。M 寻 找是否有空闲的 P，如果有则 将该 G 对象移动到它本身。接下来 M 执行一个调度循环（调用 G 对象-> 执行-> 清理线程→继续找新的 Goroutine 执行）

## **GMP** 中 **hand off** 机制

当本线程 M 因为 G 进行的系统调用阻塞时，线程释放绑定的 P，把 P 转移给其 他空闲的 M 执行。 细 节：当发生上线文切换时，需要对执行现场进行保护，以便下次被调度执行 时进行现场恢复。Go 调度 器 M 的栈保存在 G 对象上，只需要将 M 所需要的寄存 器（SP、PC 等）保存到 G 对象上就可以实现现 场保护。当这些寄存器数据被保 护起来，就随时可以做上下文切换了，在中断之前把现场保存起来。如 果此时 G 任务还没有执行完，M 可以将任务重新丢到 P 的任务队列，等待下一次被调度 执行。当再次被 调度执行时，M 通过访问 G 的 vdsoSP、vdsoPC 寄存器进行现场 恢复（从上次中断位置继续执行）。

## 协作式的抢占式调度

在 1.14 版本之前，程序只能依靠 Goroutine 主动让出 CPU 资源才能触发调 度。这种方式存在问题有：

1. 某些 Goroutine 可以长时间占用线程，造成其它 Goroutine 的饥饿

2. 垃圾回收需要暂停整个程序（Stop-the-world，STW），最长可能需要几分 钟的时间，导致整个程 序无法工作

## 基于信号的抢占式调度

在任何情况下，Go 运行时并行执行（注意，不是并发）的 goroutines 数量是 小于等于 P 的数量的。为 了提高系统的性能，P 的数量肯定不是越小越好，所 以官方默认值就是 CPU 的核心数，设置的过小的 话，如果一个持有 P 的 M， 由于 P 当前执行的 G 调用了 syscall 而导致 M 被阻塞，那么此时关键点： GO 的调度器是迟钝的，它很可能什么都没做，直到 M 阻塞了相当长时间以 后，才会发现有一个 P/M 被 syscall 阻塞了。然后，才会用空闲的 M 来强这 个 P。通过 sysmon 监控实现的抢占式调度，最快在 20us，最慢在 10-20ms 才 会发现有一个 M 持有 P 并阻塞了。操作系统在 1ms 内可以完成很多次线程 调 度（一般情况 1ms 可以完成几十次线程调度），Go 发起 IO/syscall 的时候执 行该 G 的 M 会阻塞然 后被 OS 调度走，P 什么也不干，sysmon 最慢要 10-20ms 才能发现这个阻塞，说不定那时候阻塞已经 结束了，这样宝贵的 P 资源就这么 被阻塞的 M 浪费了。

## **GMP** 调度过程中存在哪些阻塞

1. I/O，select
2. block on syscall
3. channel 4. 等待锁

5. runtime.Gosched()

## **Sysmon** 有什么作用

Sysmon 也叫监控线程，变动的周期性检查，好处

1. 释放闲置超过 5 分钟的 span 物理内存；

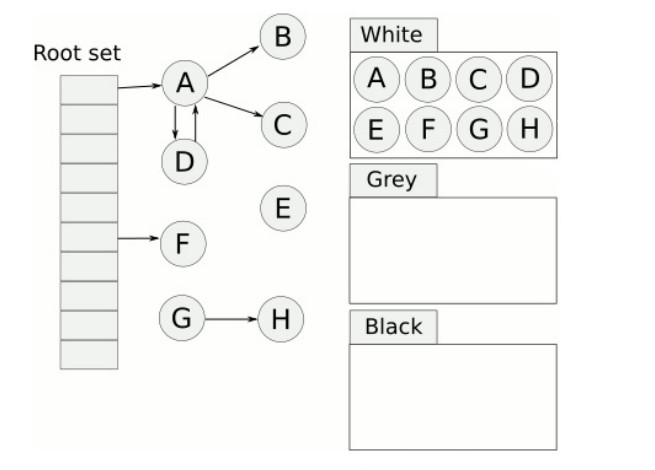
2. 如果超过 2 分钟没有垃圾回收，强制执行；

3. 将长时间未处理的 netpoll 添加到全局队列；

4. 向长时间运行的 G 任务发出抢占调度（超过 10ms 的 g，会进行 retake）； 5. 收回因 syscall 长时间阻塞的 P；

## 三色标记原理

我们首先看一张图，大概就会对 三色标记法有一个大致的了解：



原理：

1. 首先把所有的对象都放到白色的集合中

2. 从根节点开始遍历对象，遍历到的白色对象从白色集合中放到灰色集合中

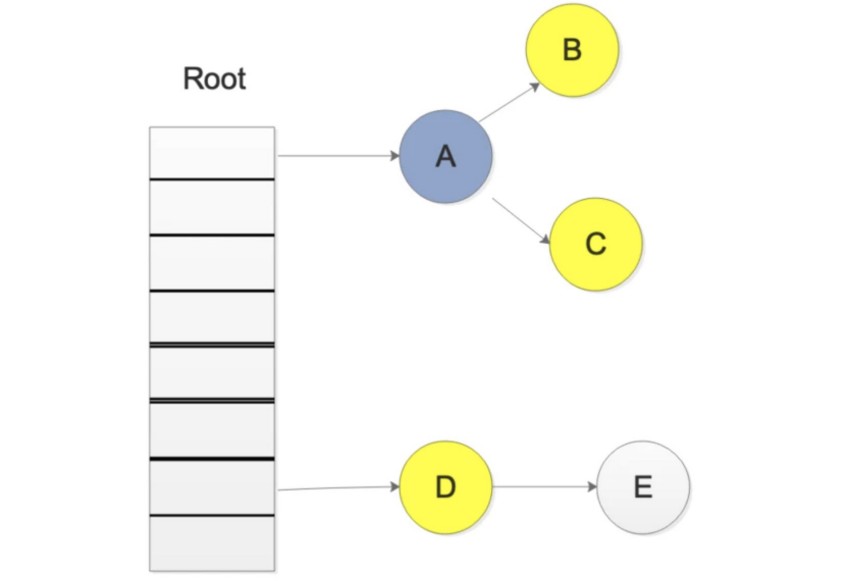
3. 遍历灰色集合中的对象，把灰色对象引用的白色集合的对象放入到灰色集 合中，同时把遍历过的灰 色集合中的对象放到黑色的集合中

4. 循环步骤 3，知道灰色集合中没有对象

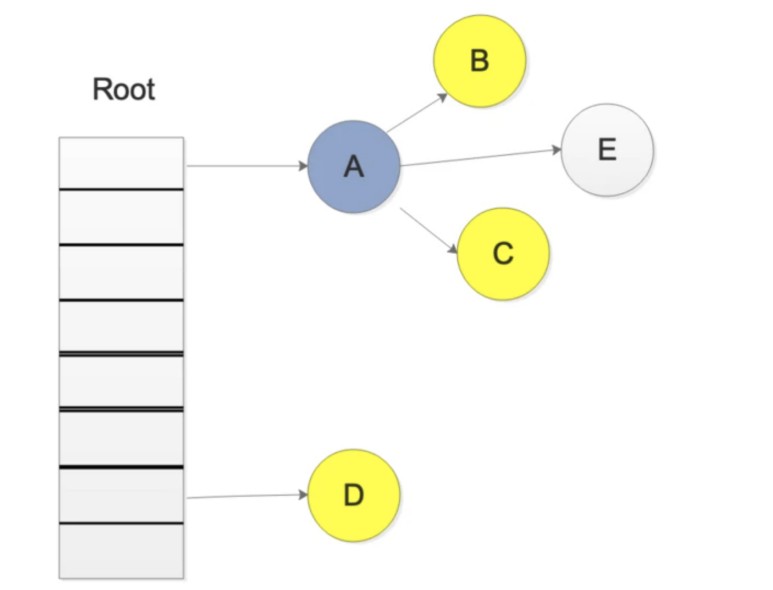
5. 步骤 4 结束后，白色集合中的对象就是不可达对象，也就是垃圾，进行回收

## 写屏障

Go 在进行三色标记的时候并没有 STW，也就是说，此时的对象还是可以进行修 改。 那么我们考虑一 下，下面的情况。



我们在进行三色标记中扫描灰色集合中，扫描到了对象 A，并标记了对象 A 的 所有引用，这时候，开始 扫描对象 D 的引用，而此时，另一个 goroutine 修改 了 D->E 的引用，变成了如下图所示



这样会不会导致 E 对象就扫描不到了，而被误认为 为白色对象，也就是垃圾 写屏障就是为了解决这样

的问题，引入写屏障后，在上述步骤后，E 会被认为 是存活的，即使后面 E 被 A 对象抛弃，E 会被在下

一轮的 GC 中进行回收，这一 轮 GC 中是不会对对象 E 进行回收的

## 插入写屏障

Go GC 在混合写屏障之前，一直是插入写屏障，由于栈赋值没有 hook 的原 因，栈中没有启用写屏障， 所以有 STW。Golang 的解决方法是：只是需要在结 束时启动 STW 来重新扫描栈。这个自然就会导致整 个进程的赋值器卡顿。

## 删除写屏障

Golang 没有这一步，Golang 的内存写屏障是由插入写屏障到混合写屏障过渡 的。简单介绍一下，一个 对象即使被删除了最后一个指向它的指针也依旧可以 活过这一轮，在下一轮 GC 中才被清理掉

## 混合写屏障

1. 混合写屏障继承了插入写屏障的优点，起始无需 STW 打快照，直接并发扫 描垃圾即可；

2. 混合写屏障继承了删除写屏障的优点，赋值器是黑色赋值器，GC 期间，任 何在栈上创建的新对 象，均为黑色。扫描过一次就不需要扫描了，这样就 消除了插入写屏障时期最后 STW 的重新扫描 栈；

3. 混合写屏障扫描精度继承了删除写屏障，比插入写屏障更低，随着带来的 是 GC 过程全程无 STW；

4. 混合写屏障扫描栈虽然没有 STW，但是扫描某一个具体的栈的时候，还是 要停止这个 goroutine 赋值器的工作（针对一个 goroutine 栈来说，是 暂停扫的，要么全灰，要么全黑哈，原子状态切 换）

## **GC** 触发时机

主动触发：调用 runtime.GC

被动触发： 使用系统监控，该触发条件由 runtime.forcegcperiod 变量控制，默认为 2 分 钟。当超过 两分钟没有产生任何 GC 时，强制触发 GC。 使用步调（Pacing）算法，其核心思想是控制内存增长的比 例。如 Go 的 GC 是一种比例 GC, 下一次 GC 结束时的堆大小和上一次 GC 存活堆大小成比例. 。

## **Go** 语言中 **GC** 的流程是什么？

Go1.14 版本以 STW 为界限，可以将 GC 划分为五个阶段： GCMark 标记准备阶段，为并发标记做准备 工作，启动写屏障 STWGCMark 扫描标记阶段，与赋值器并发执行，写屏障开启并发 GCMarkTermination 标记终止阶段，保证一个周期内标记任务完成，停止写屏 障 GCoff 内存清扫阶 段，将需要回收的内存归还到堆中，写屏障关闭 GCoff 内存归还阶段，将过多的内存归还给操作系统， 写屏障关闭。

## **GC** 如何调优

通过 go tool pprof 和 go tool trace 等工具

1. 控制内存分配的速度，限制 Goroutine 的数量，从而提高赋值器对 CPU 的利用率。

2. 减少并复用内存，例如使用 sync.Pool 来复用需要频繁创建临时对象，例 如提前分配足够的内存来 降低多余的拷贝。

3. 需要时，增大 GOGC 的值，降低 GC 的运行频率。

## **Golang GC** 时会发⽣什么**?**

⾸先我们先来了解下垃圾回收.什么是垃圾回收？ 内存管理是程序员开发应⽤的⼀⼤难题。传统的系统级编程语⾔（主要指C/C++）中，程序开发者必须对 内存⼩⼼的进⾏管理操作，控制内存的申请及释放。因为稍有不慎，就可能产⽣内存泄露问题，这种问 题不易发现并且难以定位，⼀直成为困扰程序开发者的噩梦。如何解决这个头疼的问题呢？ 过去⼀般采⽤两种办法：

1. 内存泄露检测⼯具。这种⼯具的原理⼀般是静态代码扫描，通过扫描程序检测可能出现内存泄露的 代码段。然

⽽检测⼯具难免有疏漏和不⾜，只能起到辅助作⽤。

2. 智能指针。这是 c++ 中引⼊的⾃动内存管理⽅法，通过拥有⾃动内存管理功能的指针对象来引⽤对 象，是程 序员不⽤太关注内存的释放，⽽达到内存⾃动释放的⽬的。这种⽅法是采⽤最⼴泛的做法，但是对 程序开发者 有⼀定的学习成本（并⾮语⾔层⾯的原⽣⽀持），⽽且⼀旦有忘记使⽤的场景依然⽆法避免内存泄 露。 为了解决这个问题，后来开发出来的⼏乎所有新语⾔（java，python，php等等）都引⼊了语⾔层

⾯的⾃动内存管

理 – 也就是语⾔的使⽤者只⽤关注内存的申请⽽不必关⼼内存的释放，内存释放由虚拟机（virtual machine）或运

⾏时（runtime）来⾃动进⾏管理。⽽这种对不再使⽤的内存资源进⾏⾃动回收的⾏为就被称为垃 圾回收。

常⽤的垃圾回收的⽅法:

1. 引⽤计数（reference counting） 这是最简单的⼀种垃圾回收算法，和之前提到的智能指针异曲同⼯。对每个对象维护⼀个引⽤计 数，当引⽤该对象的对象被销毁或更新时被引⽤对象的引⽤计数⾃动减⼀，当被引⽤对象被创建或 被赋值给其他对象时引⽤计数⾃动加⼀。当引⽤计数为0时则⽴即回收对象。 这种⽅法的优点是实现简单，并且内存的回收很及时。这种算法在内存⽐较紧张和实时性⽐较⾼的 系统中使⽤的⽐较⼴泛，如ios cocoa框架，php，python等。 但是简单引⽤计数算法也有明显的缺点

1. 频繁更新引⽤计数降低了性能。

⼀种简单的解决⽅法就是编译器将相邻的引⽤计数更新操作合并到⼀次更新；还有⼀种⽅法是 针对频繁发⽣的临时变量引⽤不进⾏计数，⽽是在引⽤达到0时通过扫描堆栈确认是否还有临 时对象引⽤⽽决定是否释放。等等还有很多其他⽅法，具体可以参考这⾥。

2. 循环引⽤。 当对象间发⽣循环引⽤时引⽤链中的对象都⽆法得到释放。最明显的解决办法是避免产⽣循环 引⽤，如cocoa引⼊ 了strong指针和weak指针两种指针类型。或者系统检测循环引⽤并主动打破循环链。当然这 也增加了垃圾回收的复杂度。

1. 标记-清除（mark and sweep）

标记-清除（mark and sweep）分为两步，标记从根变量开始迭代得遍历所有被引⽤的对象，对能 够通过应⽤遍历访问到的对象都进⾏标记为“被引⽤”；标记完成后进⾏清除操作，对没有标记过的 内存进⾏回收（回收同时可能伴有碎⽚整理操作）。这种⽅法解决了引⽤计数的不⾜，但是也有⽐ 较明显的问题：每次启动垃圾回收都会暂停当前所有的正常代码执⾏，回收是系统响应能⼒⼤⼤降 低！当然后续也出现了很多mark&sweep算法的变种（如三⾊标记法）优化了这个问题。

1. 分代搜集（generation）

java的jvm 就使⽤的分代回收的思路。在⾯向对象编程语⾔中，绝⼤多数对象的⽣命周期都⾮常 短。分代收集的基 本思想是，将堆划分为两个或多个称为代（generation）的空间。新创建的对象存放在称为新⽣代

（young

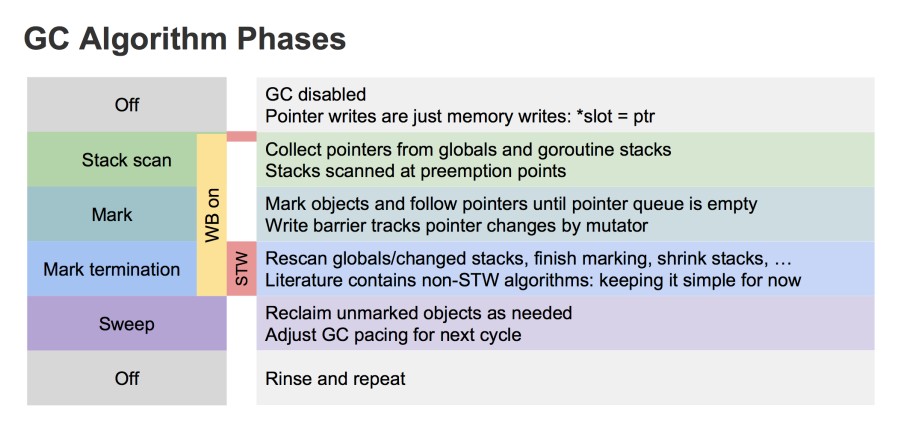
generation）中（⼀般来说，新⽣代的⼤⼩会⽐ ⽼年代⼩很多），随着垃圾回收的重复执⾏，⽣命 周期较⻓的对象会被提升（promotion）到⽼年代中（这⾥⽤到了⼀个分类的思路，这个是也是科 学思考的⼀个基本思路）。因此，新⽣代垃圾回收和⽼年代垃圾回收两种不同的垃圾回收⽅式应运

⽽⽣，分别⽤于对各⾃空间中的对象执⾏垃圾回收。新⽣代垃圾回收的速度⾮常快，⽐⽼年代快⼏

个数量级，即使新⽣代垃圾回收的频率更⾼，执⾏效率也仍然⽐⽼年代垃圾回收强，这是因为⼤多 数对象的⽣命周期都很短，根本⽆需提升到⽼年代

Golang GC 时会发⽣什么?

Golang 1.5后，采取的是“⾮分代的、⾮移动的、并发的、三⾊的”标记清除垃圾回收算法。 golang 中的 gc 基本上是标记清除的过程：



gc的过程⼀共分为四个阶段： 1. 栈扫描（开始时STW）

2. 第⼀次标记（并发）

3. 第⼆次标记（STW）

4. 清除（并发） 整个进程空间⾥申请每个对象占据的内存可以视为⼀个图，初始状态下每个内存对象都是⽩⾊标记。

1. 先STW，做⼀些准备⼯作，⽐如 enable write barrier。然后取消STW，将扫描任务作为多个并发 的

goroutine⽴即⼊队给调度器，进⽽被CPU处理

2. 第⼀轮先扫描root对象，包括全局指针和 goroutine 栈上的指针，标记为灰⾊放⼊队列

3. 第⼆轮将第⼀步队列中的对象引⽤的对象置为灰⾊加⼊队列，⼀个对象引⽤的所有对象都置灰并加

⼊队列后， 这个对象才能置为⿊⾊并从队列之中取出。循环往复，最后队列为空时，整个图剩下的⽩⾊内存空 间即不可到

达的对象，即没有被引⽤的对象；

4. 第三轮再次STW，将第⼆轮过程中新增对象申请的内存进⾏标记（灰⾊），这⾥使⽤了write barrier（写屏

障）去记录

Golang gc 优化的核⼼就是尽量使得 STW(Stop The World) 的时间越来越短。

# 微服务

## 您对微服务有何了解？

微服务，又称微服务架构，是一种架构风格，它将应用程序构建为以业务领域 为模型的小型自治服务集 合。

通俗地说，你必须看到蜜蜂如何通过对齐六角形蜡细胞来构建它们的蜂窝状 物。他们最初从使用各种材 料的小部分开始，并继续从中构建一个大型蜂箱。 这些细胞形成图案，产生坚固的结构，将蜂窝的特定 部分固定在一起。 这里，每个细胞独立于另一个细胞，但它也与其他细胞相关。这意味着对一个 细胞的 损害不会损害其他细胞，因此，蜜蜂可以在不影响完整蜂箱的情况下重 建这些细胞



请参考上图。这里，每个六边形形状代表单独的服务组件。与蜜蜂的工作类 似，每个敏捷团队都使用可 用的框架和所选的技术堆栈构建单独的服务组件。 就像在蜂箱中一样，每个服务组件形成一个强大的微 服务架构，以提供更好的 36 可扩展性。此外，敏捷团队可以单独处理每个服务组件的问题，而对整个应 用 程序没有影响或影响最小。

## 说说微服务架构的优势

1. 独立开发 所有微服务都可以根据各自的功能轻松开发

2. 独立部署 根据他们所提供的服务，可以在任何应用中单独部署

3. 故障隔离 即使应用中的一个服务不起作用，系统仍然继续运行

4. 混合技术栈 可以用不同的语言和技术来构建同一应用程序的不同服务

5. 粒度缩放 各个组件可根据需要进行扩展，无需将所有组件融合到一起

## 微服务有哪些特点

1. 解耦—系统内的服务很大程度上是分离的。因此，整个应用程序可以轻松 构建，更改和扩展

2. 组件化—微服务被视为可以轻松更换和升级的独立组件

3. 业务能力—微服务非常简单，专注于单一功能

4. 自治—开发人员和团队可以彼此独立工作，从而提高速度

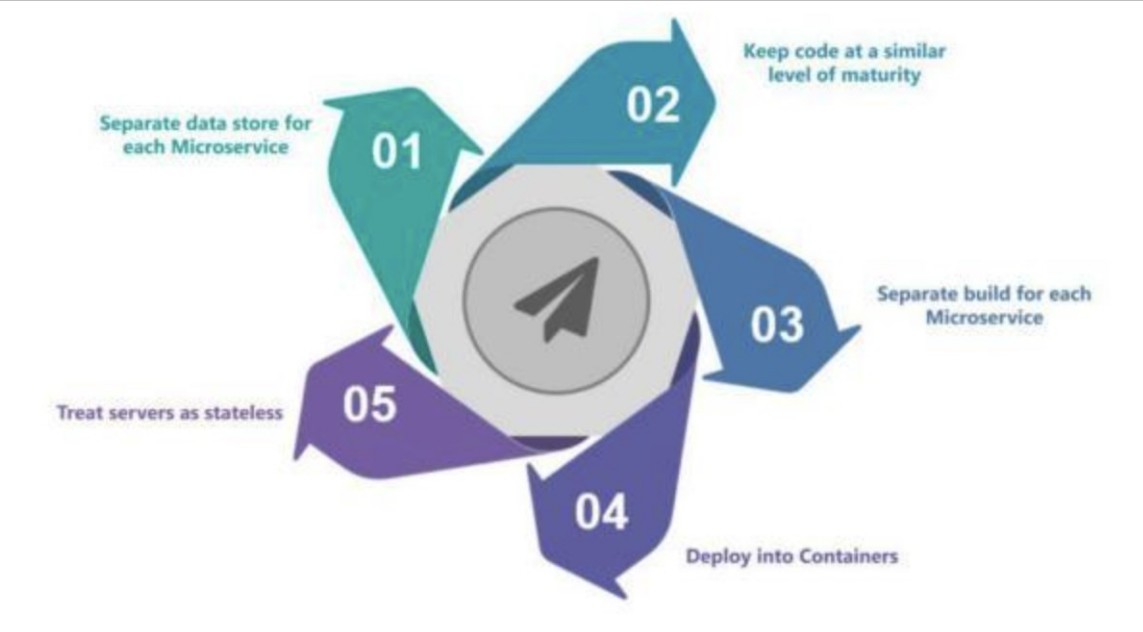
5. 持续交付—通过软件创建，测试和批准的系统自动化，允许频繁发布软件

6. 责任—微服务不关注应用程序作为项目。相反，他们将应用程序视为他们 负责的产品

7. 分散治理—重点是使用正确的工具来做正确的工作。这意味着没有标准化 模式或任何技术模式。开 发人员可以自由选择最有用的工具来解决他们的 问题

8. 敏捷—微服务支持敏捷开发。任何新功能都可以快速开发并再次丢弃

## 设计微服务的最佳实践是什么



**微服务架构如何运作**

微服务架构具有以下组件：

1. 客户端 – 来自不同设备的不同用户发送请求。

2. 身份提供商 – 验证用户或客户身份并颁发安全令牌。

3. API 网关 – 处理客户端请求。

4. 静态内容 – 容纳系统的所有内容。

5. 管理 – 在节点上平衡服务并识别故障。

6. 服务发现 – 查找微服务之间通信路径的指南。

7. 网络 – 代理服务器及其数据中心的分布式网络。

8. 远程服务 – 启用驻留在 IT 设备网络上的远程访问信息

## 微服务架构的优缺点是什么

微服务架构的优点 ：

1. 自由使用不同的技术

2. 每个微服务都侧重于单一功能

3. 支持单个可部署单元

4. 允许经常发布软件

5. 确保每项服务的安全性

6. 多个服务是并行开发和部署的 微服务架构的缺点

1. 增加故障排除挑战

2. 由于远程调用而增加延迟

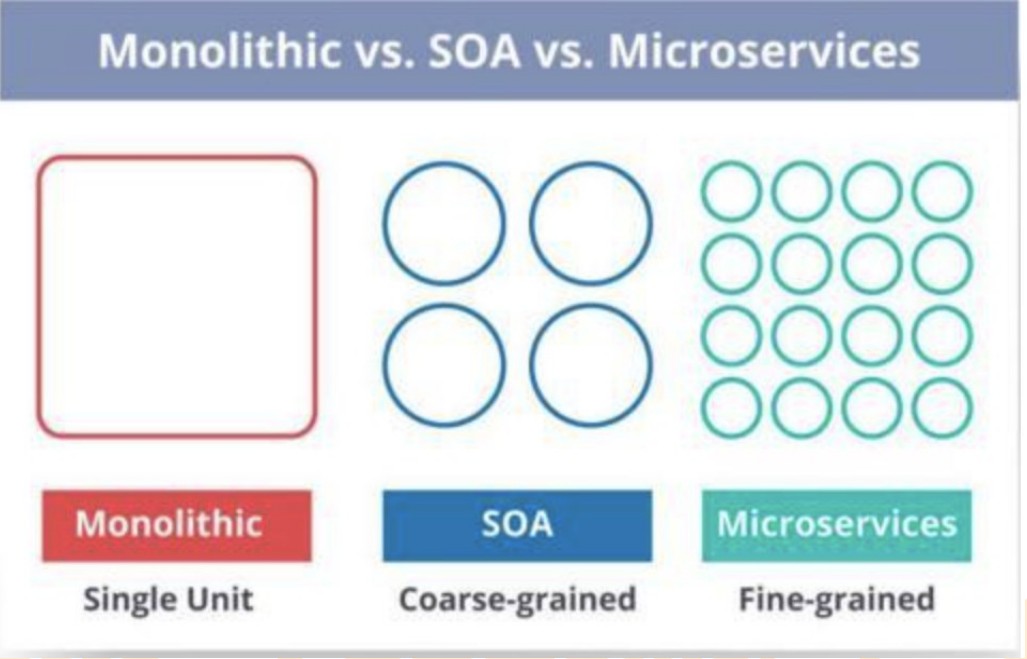
3. 增加了配置和其他操作的工作量

4. 难以保持交易安全

5. 艰难地跨越各种便捷跟踪数据

6. 难以在服务之间进行编码

## 单片，**SOA** 和微服务架构有什么区别？



1. 单片架构类似于大容器，其中应用程序的所有软件组件组装在一起并紧密 封装。

2. 一个面向服务的架构是一种相互通信服务的集合。通信可以涉及简单的数 据传递，也可以涉及两个 或多个协调某些活动的服务。

3. 微服务架构是一种架构风格，它将应用程序构建为以业务域为模型的小型 自治服务集合。

## 在使用微服务架构时，您面临哪些挑战？

开发一些较小的微服务听起来很容易，但开发它们时经常遇到的挑战如下。

1. 自动化组件：难以自动化，因为有许多较小的组件。因此，对于每个组 件，我们必须遵循 Build，

Deploy 和 Monitor 的各个阶段。

2. 易感性：将大量组件维护在一起变得难以部署，维护，监控和识别问题。 它需要在所有组件周围具 有很好的感知能力。 、

3. 配置管理：有时在各种环境中维护组件的配置变得困难。

4. 调试：很难找到错误的每一项服务。维护集中式日志记录和仪表板以调试 问题至关重要。

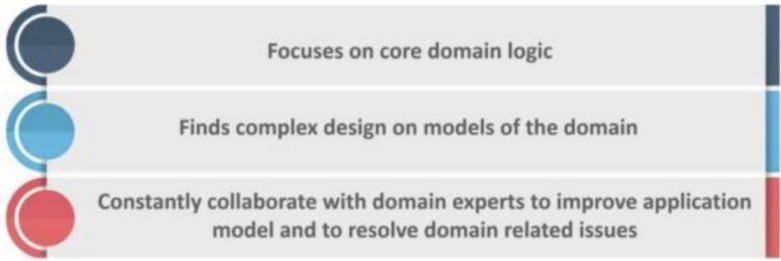
## **SOA** 和微服务架构之间的主要区别是什么？



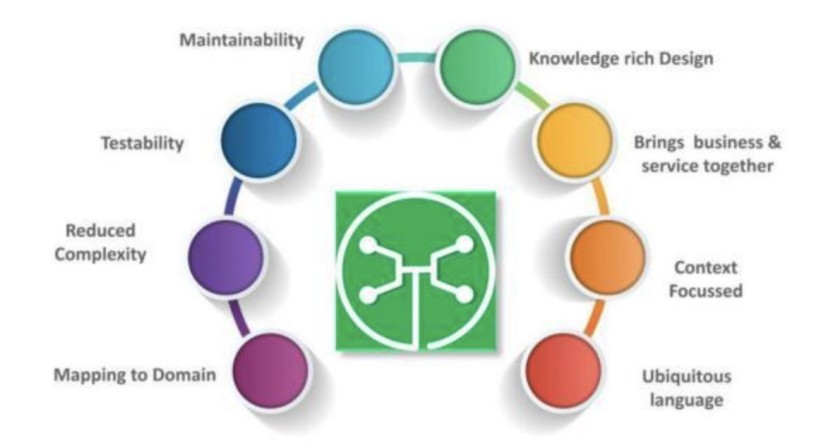
**微服务有什么特点**



**什么是领域驱动设计**



**为什么需要域驱动设计（DDD）**



**什么是无所不在的语言**

如果您必须定义泛在语言（UL），那么它是特定域的开发人员和用户使用的通 用语言，通过该语言可以 轻松解释域。 无处不在的语言必须非常清晰，以便它将所有团队成员放在同一页面上，并以 机器可以理 解的方式进行翻译

## 什么是凝聚力

模块内部元素所属的程度被认为是凝聚力

## 什么是耦合

组件之间依赖关系强度的度量被认为是耦合。一个好的设计总是被认为具有高 内聚力和低耦合性。

## 、什么是 **REST / RESTful** 以及它的用途是什么？

Representational State Transfer（REST）/ RESTful Web 服务是一种帮助计 算机系统通过 Internet 进 行通信的架构风格。这使得微服务更容易理解和实 现。 微服务可以使用或不使用 RESTful API 实现，但 使用 RESTful API 构建松散 耦合的微服务总是更容易。

## 什么是不同类型的微服务测试

在使用微服务时，由于有多个微服务协同工作，测试变得非常复杂。因此，测 试分为不同的级别

1. 在底层，我们有面向技术的测试，如单元测试和性能测试。这些是完全自 动化的。

2. 在中间层面，我们进行了诸如压力测试和可用性测试之类的探索性测试。

3. 在顶层， 我们的验收测试数量很少。这些验收测试有助于利益相关者理解 和验证软件功能

# 容器技术

## 为什么需要 **DevOps**

在当今，软件开发公司在软件新版本发布方面，多尝试通过发布一系列以小的 特性改变集为目标的新软 件版本，代替发布一个大特性改变集的新软件版本的 方式。这种方式有许多优点，诸如，快速的客户反 馈，软件质量的保证等。也 会获得较高的客户满意度评价。完成这样的软件发布模式，开发公司需要做 到：

增加软件布署的频率 降低新发布版本的失败率 缩短修复缺陷的交付时间 加快解决版本冲突的问题

DevOps 满足所有这些需求且帮助公司高质完成软件无缝交付的目标。

## **Docker** 是什么？

Docker 是一个容器化平台，它包装你所有开发环境依赖成一个整体，像一个容 器。保证项目开发，如 开发、测试、发布等各生产环节都可以无缝工作在不同 的平台 Docker 容器：将一个软件包装在一个完 整的文件系统中，该文件系统包含运行 所需的一切：代码，运行时，系统工具，系统库等。可以安装在 服务器上的任 何东西。 这保证软件总是运行在相同的运行环境，无需考虑基础环境配置的改变。

## **DevOps** 有哪些优势？

技术优势: 持续的软件交付能力 修复问题变得简单 更快得解决问题 商业优势: 更快交付的特性 更稳定的操作系统环境

更多时间可用于创造价值 (而不是修复 / 维护)

## **CI** 服务有什么用途？

CI （Continuous Integration）-- 持续集成服务 -- 主要用于整合团队开发 中不同开发者提交到开发仓库 中的项目代码变化，并即时整合编译，检查整合 编译错误的服务。它需要一天中多次整合编译代码的能 力，若出现整合错误， 可以优异地准确定位提交错误源

## 如何使用 **Docker** 技术创建与环境无关的容器系统？

Docker 技术有三中主要的技术途径辅助完成此需求： 存储卷（Volumes）

环境变量（Environment variable）注入

只读（Read-only）文件系统

## **Dockerfile** 配置文件中的 **COPY** 和 **ADD** 指令有什么不

**同？**

虽然 ADD 和 COPY 功能相似，推荐 COPY

那是因为 COPY 比 ADD 更直观易懂。 COPY 只是将本地文件拷入容器这么简 单，而 ADD 有一些其它 特性功能（诸如，本地归档解压和支持远程网址访问 等），这些特性在指令本身体现并不明显。因此， 有必要使用 ADD 指令的最好 例子是需要在本地自动解压归档文件到容器中的情况，如 ADD rootfs.tar.xz 。

## **Docker** 镜像（**image**）是什么？

Docker image 是 Docker 容器的源。换言之，Docker images 用于创建 Docker 容器（containers）。 映像（Images）通过 Docker build 命令创建， 当 run 映像时，它启动成一个 容器（container）进 程。 做好的映像由于可 能非常庞大，常注册存储在诸如 registry.hub.docker.com 这样的公共平台 上。 映像常被分层设计，每层可单独成为一个小映像，由多层小映像再构成大 映像，这样碎片化的设计为了 使映像在互联网上共享时，最小化传输数据需 求。

## **Docker** 容器（**container**）是什么

Docker containers -- Docker 容器 -- 是包含其所有运行依赖环境，但与其 它容器共享操作系统内核的 应用，它运行在独立的主机操作系统用户空间进程 中。Docker 容器并不紧密依赖特定的基础平台：可运 行在任何配置的计算机， 任何平台以及任何云平台上

## **Docker** 中心（**hub**）什么概念

Docker hub 是云基础的 Docker 注册服务平台，它允许用户进行访问 Docker 中心资源库，创建自己的 Docker 映像并测试，推送并存储创建好的 Docker 映像，连接 Docker 云平台将已创建好的指定 Docker 映像布署到本地主机等 任务。它提供了一个查找发现 Docker 映像，发布 Docker 映像及控制变化升 级 的资源中心，成为用户组或团队协作开发中保证自动化开发流程的有效技术 途径。

## 在任意给定时间点指出一个 **Docker** 容器可能存在的运行阶

**段**

在任意时间点，一个 Docker 容器可能存在以下运行阶段 ：

运行中（Running） 已暂停（Paused） 重启中（Restarting） 已退出（Exited）

## 有什么方法确定一个 **Docker** 容器运行状态

使用如下命令行命令确定一个 Docker 容器的运行状态

docker ps –a

这将列表形式输出运行在主机上的所有 Docker 容器及其运行状态。从这个列 表中很容易找到想要的容 器及其运行状态

## 在 **Dockerfile** 配置文件中最常用的指令有哪些？

1. FROM：使用 FROM 为后续的指令建立基础映像。在所有有效的 Dockerfile 中， FROM 是第一条 指令。

2. LABEL：LABEL 指令用于组织项目映像，模块，许可等。在自动化布署方面 LABEL 也有很大用途。 在 LABEL 中指定一组键值对，可用于程序化配置或布署 Docker 。

3. RUN：RUN 指令可在映像当前层执行任何命令并创建一个新层，用于在映像层中 添加功能层，也 许最来的层会依赖它。

4. CMD：使用 CMD 指令为执行的容器提供默认值。在 Dockerfile 文件中，若添 加多个 CMD 指令， 只有最后的 CMD 指令运行

## 什么类型的应用（无状态性或有状态性）更适合 **Docker** 容

**器技术**

对于 Docker 容器创建无状态性（Stateless）的应用更可取。通过从应用项目 中将与状态相关的信息及 配置提取掉，我们可以在项目环境外建立不依赖项目 环境的 Docker 容器。这样，我们可以在任意产品 中运行同一容器，只需根据 产品需要像问 & 答（QA）一样给其配置环境即可。 这帮助我们在不同场景 重 用相同的 Docker 映像。另外，使用 无状态性（Stateless）容器应用相比有 状态性（Stateful）容器 应用更具伸缩性，也容易创建

## 解释基本 **Docker** 应用流程

初始，所有都有赖于 Dockerfile 配置文件。Dockerfile 配置文件就是创建 Docker image (映像) 的源代 码。 48 一旦 Dockerfile 配置好了，就可以创建（build）并生成 'image（映像） ' ，'image' 就是 Dockerfile 配置文件中 「源代码」的「编译」版本。 一旦有了 'image' ，就可以在 registry（注册中 心） 发布它。 'registry' 类似 git 的资源库 -- 你可以推送你的映像（image），也可取回库中的映像

（image）。 之后，你就可以使用 image 去启动运行 'containers（容器）'。运行中的容 器在许多方

面，与虚拟机非常相似，但容器的运行不需要虚拟管理软件的运 行。

## **Docker Image** 和 **Docker Layer** （层**)** 有什么不同？

Image：一个 Docker Image 是由一系列 Docker 只读层（read-only Layer） 创建出来的。 Layer：在 Dockerfile 配置文件中完成的一条配置指令，即表示一个 Docker 层（Layer）。 如下 Dockerfile 文件包 含 4 条指令，每条指令创建一个层（Layer）

FROM ubuntu:15.04 COPY . /app

RUN make /app

CMD python /app/app.py

重点，每层只对其前一层进行一（某）些进化

## 虚拟化技术是什么？

最初的构想，virtualisation（虚拟化） 被认为是逻辑划分大型主机使得多个 应用可以并行运行的一种技 术方案。然而，随着技术公司及开源社区的推进， 现实发生了戏剧性的转变，以致产生了以一种或某种 方式操作特权指令可以在 单台基于 x86 硬件的系统上同时运行多个（种）操作系统的技术。 实质的效果 是，虚拟化技术允许你在一个硬件平台下运行 2 个完全不同的操作 系统。每个客户操作系统可完成像系 统自检、启动、载入系统内核等像在独立 硬件上的一切动作。同时也具备坚实的安全基础，例如，客户 操作系统不能获 取完全访问主机或其它客户系统的权限，及其它涉及安全，可能把系统搞坏的 操作。 基

于对客户操作系统虚拟硬件、运行环境模拟方法的不同，对虚拟化技术进行 分类，主要的有如下 3 种虚

拟化技术种类 ： 全模拟（Emulation） 半虚拟（Paravirtualization）

基于容器的虚拟化（Container-based virtualization）

## 虚拟管理层（程序）是什么

hypervisor -- 虚拟管理层（程序）-- 负责创建客户虚拟机系统运行所需虚拟 硬件环境。它监管客户虚拟 操作系统的运行，并为客户系统提供必要的运行资 源，保证客户虚拟系统的运行。虚拟管理层（程序） 驻留在物理主机系统和虚 拟客户系统之间，为虚拟客户系统提供必要的虚拟服务。如何理解它，它侦听 运行在虚拟机中的客户操作系统的操作并在主机操作系统中模拟客户操作系统 所需硬件资源请求。满足 客户机的运行需求

虚拟化技术的快速发展，主要在云平台，由于在虚拟管理程序的帮助下，可允 许在单台物理服务器上生 成多个虚拟服务器，驱动着虚拟化技术快速发展及广 泛应用。诸如，Xen，VMware，KVM 等，以及商 业化的处理器硬件生产厂商也加 入在硬件层面支持虚拟化技术的支持。诸如，Intel 的 VT 和 AMD-V

## **Docker** 集群（**Swarm**）是什么

Docker Swarm -- Docker 群 -- 是原生的 Docker 集群服务工具。它将一群 Docker 主机集成为单一一个 虚拟 Docker 主机。利用一个 Docker 守护进程， 通过标准的 Docker API 和任何完善的通讯工具， Docker Swarm 提供透明地将 Docker 主机扩散到多台主机上的服务

## 在使用 **Docker** 技术的产品中如何监控其运行

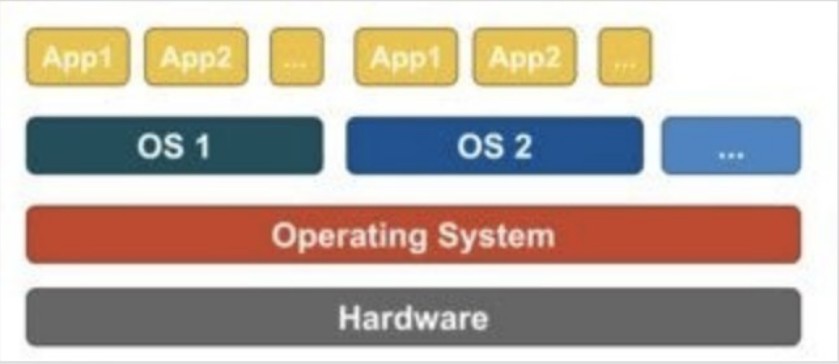
Docker 在产品中提供如 运行统计和 Docker 事件的工具。可以通过这些工具 命令获取 Docker 运行状 况的统计信息或报告。 Docker stats ： 通过指定的容器 id 获取其运行统计信息，可获得容器对 CPU， 内存使用情况等的统计信息，类似 Linux 系统中的 top 命令。 Docker events ：Docker 事件是一个命 令，用于观察显示运行中的 Docker 一 系列的行为活动。 一般的 Docker 事件有：attach（关联）， commit（提交），die（僵死）， detach（取消关联），rename（改名），destory（销毁）等。也可 使用多个选 项对事件记录筛选找到想要的事件信息

## 什么是孤儿卷及如何删除它

孤儿卷是未与任何容器关联的卷。 docker volume rm 删除指定的卷 docker volume prune 删除所有孤儿卷

## 什么是半虚拟化（**Paravirtualization**）

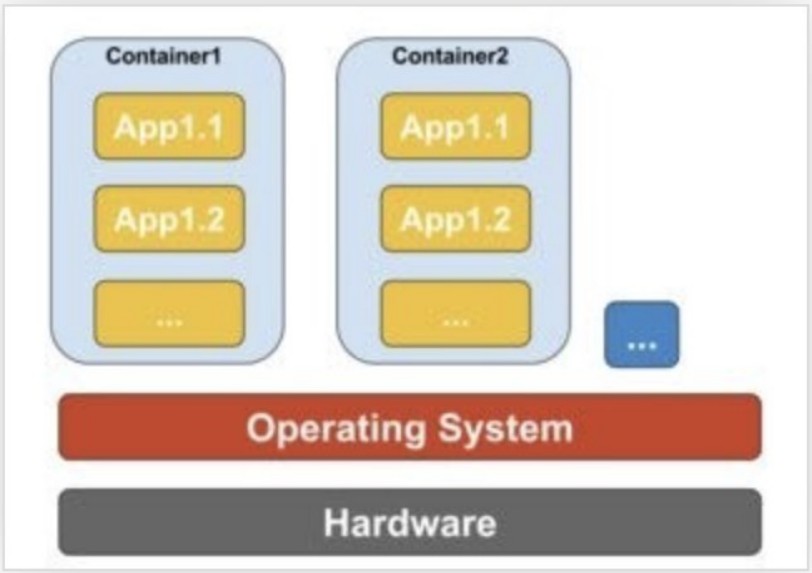
Paravirtualization，也称为第 1 类虚拟机管理（层）程序，其直接在硬件或 裸机（bare-metal）上运 行，提供虚拟机直接使用物理硬件的服务，它帮助主 机操作系统，虚拟化硬件和实际硬件进行协作以实 现最佳性能。这种虚拟层管 理技术的程序一般占用系统资源较小，其本身并不需要占用大量系统资源。



这种虚拟层管理程序有 Xen, KVM 等

## **Docker** 技术与虚拟机技术有何不同？

Docker 不是严格意义上的虚拟化硬件的技术。它依赖 container-based virtualization（基于容器的虚 拟化） 的技术实现工具，或可以认为它是操作 系统用户运行级别的虚拟化。因此， Docker 最初使用 LXC 驱动它，后来移至 由 libcontainer 基础库驱动它，现已更名为 runc 。 Docker 主要致力于应 用容 器内的应用程序的自动化部署。应用容器设计用于包装和运行单一服务， 而操作系统设计用于运行多进 程任务，提供多种运算服务的能力。如虚拟机中 等同完全操作系统的能力。因此，Docker 被认为是容器 化系统上管理容器及应 用容器化的布署工具



1. 与虚拟机不同，容器无需启动操作系统内核，因此，容器可在不到 1 秒钟 时间内运行起来。这个特 性，使得容器化技术比其它虚拟化技术更具有独 特性及可取性。

2. 由于容器化技术很少或几乎不给主机系统增加负载，因此，基于容器的虚 拟化技术具有近乎原生的

性能表现。

3. 基于容器的虚拟化，与其他硬件虚拟化不同，运行时不需要其他额外的虚 拟管理层软件。

4. 主机上的所有容器共享主机操作系统上的进程调度，从而节省了额外的资 源的需求。

5. 与虚拟机 image 相比，容器（Docker 或 LXC images）映像较小， 因 此，容器映像易于分发。

6. 容器中的资源分配由 Cgroups 实现。 Cgroup 不会让容器占用比给它们分 配的更多的资源。但 是，现在其它的虚拟化技术，对于虚拟机，主机的所 有资源都可见，但无法使用。这可以通过在容 器和主机上同时运行 top 或 htop 来观察到。在两个环境中的输出看起来相同。

## 请解释一下 **docerfile** 配置文件中的 **ONBUILD** 指令的用途

**含义**

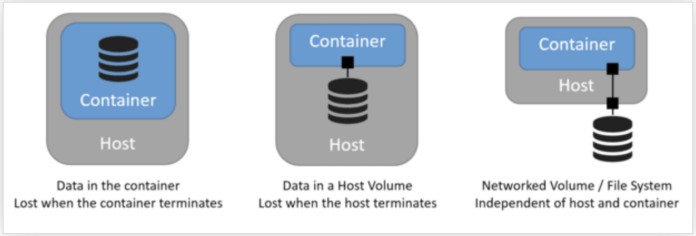
配置文件中的 ONBUILD 指令为创建的 Docker image （映像）加入在将来执行 的指令，即使用了带有

ONBUILD指令的镜像作为基础镜像时，会执行基础镜像的ONBUILD指令

## 有否在创建有状态性的 **Docker** 应用的较好实践？ 最适合

**的 场景有什么**

有状态性 Docker 应用的问题关键在于状态数据保存在哪儿的问题。 若所有数 据保存在容器内， 当更 新软件版本或想将 Docker 容器移到其它机器上时， 找回这些在运行中产生的状态数据将非常困难。 您 需要做的是将这些表达运行状态的数据保存在永久卷中。参考如下 3 种模 式：



1. 数据保存在容器中，当容器停止运行时，运行状态数据丢失！

2. 数据保存在主机卷（Host Volume）中，当主机停机时，运行状 态数据将无法访问

3. 数据保存在网络文件系统卷中，数据访问不依赖容器的运行与主 机的运行 若您使用：docker run - v hostFolder:/containerfolder 命令运行您的容 器， 容器运行中任何对 /containerfolder 目录下数 据的改变， 将永久保存 在主机的 hostfolder 目录下。 使用网络文件系统（nfs）与此类似。 那样 您 就可以运行您的容器在任何主机上且其运行状态数据被保存在网络文件系统 上。

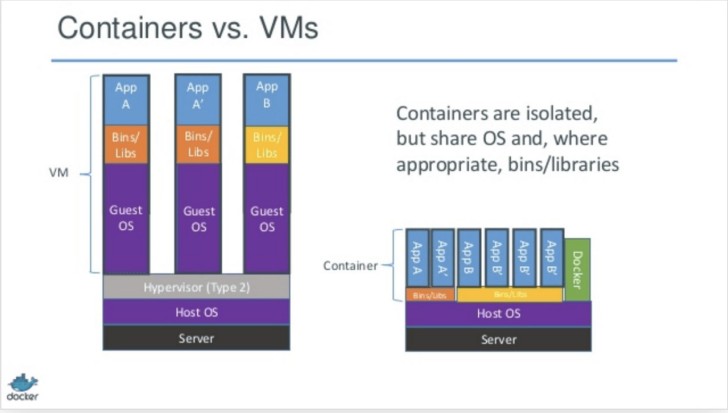
## 容器化技术在底层的运行原理？

2006 年前后， 人们，包括一些谷歌的雇员， 在 Linux 内核级别上实现了一 种新的名为 命名空间

（namespace） 的技术（实际上这种概念在 FreeBSD 系 统上由来已久）。我们知道，操作系统的一个 功能就是进程共享公共资源， 诸 如，网络和硬盘空间等。 但是，如果一些公共资源被包装在一个命名空 间中， 只允许属于这个命名空间中的进程访问又如何呢？ 也就是说，可以分配一大块 硬盘空间给命名空 间 X 供其使用，但是，命名空间 Y 中的进程无法看到或访 问这部分资源。 同样地， 命名空间 Y 中分配 的资源，命名空间 X 中的进程 也无法访问。当然， X 中的进程无法与 Y 中的进程进行交互。这提供了某 种 对公共资源的虚拟化和隔离的技术。 这就是 Docker 技术的底层工作原理： 每个容器运行在它自己的 命名空间中， 但是，确实与其它运行中的容器共用相同的系统内核。 隔离的产生是由于系统 内核清楚地

知道命名空间及其中的进程，且这些进程调用系统 API 时，内核保 证进程只能访问属于其命名空间中的

资源



运行中的容器是隔离的。准确地说， 各容器共享操作系统内 核及操作系统 API。

## 说说容器化技术与虚拟化技术的优缺点

不能像虚拟机那样在容器上运行与主机完全不同的操作系统。 然而， 可以在 容器上运行不同的 Linux 发布版，由于容器共享系统内核的缘故。容器的隔离 性没有虚拟机那么健壮。事实上， 在早期容器化技 术实现上，存在某种方法使 客户容器可接管整个主机系统。 也可看到，载入新容器并运行，并不会像虚 拟机那样装载一个新的操作系统进 来。 所有的容器共享同一系统内核， 这也就是容器被认为非常轻量化 的原因。 同样的原因，不像虚拟机， 你不须为容器预分配大量的内存空间， 因为它不 是运行新的整个 的操作系统。 这使得在一个操作系统主机上，可以同时运行成 百上千个容器应用， 在运行完整操作系统 的虚拟机上，进行这么多的并行沙箱 实验是不可能的

## 如何使 **Docker** 适应多种运行环境

您必然想改变您的 Docker 应用配置以更适应现实运行环境的变化。下面包含 一些修改建议： 移除应用 代码中对任何固定存储卷的绑定，由于代码驻留在容器内部，而不能 从外部进行修正。 绑定应用端口到 主机上的不同端口 差异化设置环境变量 （例如： 减少日志冗余或者使能发电子邮件） 设定重启策略

（例如： restart: always ）， 避免长时间宕机 加入额外的服务（例如： log aggregator） 由于以上原 因， 您更需要一个 Compose 配置文件，大概叫 production.yml ，它配置了恰当的产品整合服务。 这 个配置文件只需包含您 选择的合适的原始 Compose 配置文件中，你改动的部分

## 为什么 **Docker compose** 采取的是并不等待前面依赖服务

**项 的容器启动就绪后再启动的组合容器启动策略？**

Docker 的 Compose 配置总是以依赖启动序列来启动或停止 Compose 中的服务 容器， 依赖启动序列 是由 Compose 配置文件中的 depends\_on ， links ， volumes\_from 和 network\_mode: "service : ..." 等这些配置指令所确定 的。 然而， Compose 启动中， 各容器的启动并不等待其依赖容器（这必定是你 整 个应用中的某个依赖的服务或应用）启动就绪后才启动。使用这种策略较好的 理由如下： 等待一个数据库服务（举例）就绪这样的问题， 在大型分布式系统中仅是相比 其它大问题的某些小问 题。 在实际发布产品运维中， 您的数据库服务会由于 各种原因，或者迁移宿主机导致其不可访问。 您 发布的产品需要有应对这样状 况的弹性。 掌控这些， 开发设计您的应用， 使其在访问数据库失效的情

况下， 能够试图 重连数据库， 直至其连接到数据库为止。 最佳的解决方案是在您的应用代码中检查是

否有应对意外的发生，无论是任何 原因导致的启动或连接失效都应考虑在内

## 什么是**k8s**？说出你的理解

K8s是kubernetes的简称，其本质是一个开源的容器编排系统，主要用于管理容器化的应用，其目标是 让部署容器化的应用简单并且高效（powerful）,Kubernetes提供了应用部署，规划，更新，维护的一种 机制。 说简单点：k8s就是一个编排容器的系统，一个可以管理容器应用全生命周期的工具，从创建应用，应用 的部署，应用提供服务，扩容缩容应用，应用更新，都非常的方便，而且还可以做到故障自愈，所以， k8s是一个非常强大的容器编排系统

## **k8s**的组件有哪些，作用分别是什么

k8s主要由master节点和node节点构成。master节点负责管理集群，node节点是容器应用真正运行的 地方。

master节点包含的组件有：kube-api-server、kube-controller-manager、kube-scheduler、etcd。 node节点包含的组件有：kubelet、kube-proxy、container-runtime。

kube-api-server：以下简称api-server，api-server是k8s最重要的核心组件之一，它是k8s集群管理的统 一访问入口，提供了RESTful API接口, 实现了认证、授权和准入控制等安全功能；api-server还是其他组 件之间的数据交互和通信的枢纽，其他组件彼此之间并不会直接通信，其他组件对资源对象的增、删、 改、查和监听操作都是交由api-server处理后，api-server再提交给etcd数据库做持久化存储，只有api- server才能直接操作etcd数据库，其他组件都不能直接操作etcd数据库，其他组件都是通过api-server间 接的读取，写入数据到etcd。

kube-controller-manager：以下简称controller-manager，controller-manager是k8s中各种控制器的 的管理者，是k8s集群内部的管理控制中心，也是k8s自动化功能的核心；controller-manager内部包含 replication controller、node controller、deployment controller、endpoint controller等各种资源对 象的控制器，每种控制器都负责一种特定资源的控制流程，而controller-manager正是这些controller的 核心管理者。

kube-scheduler：以下简称scheduler，scheduler负责集群资源调度，其作用是将待调度的pod通过一 系列复杂的调度算法计算出最合适的node节点，然后将pod绑定到目标节点上。shceduler会根据pod的 信息，全部节点信息列表，过滤掉不符合要求的节点，过滤出一批候选节点，然后给候选节点打分，选 分最高的就是最佳节点，scheduler就会把目标pod安置到该节点。

Etcd：etcd是一个分布式的键值对存储数据库，主要是用于保存k8s集群状态数据，比如，pod，service 等资源对象的信息；etcd可以是单个也可以有多个，多个就是etcd数据库集群，etcd通常部署奇数个实 例，在大规模集群中，etcd有5个或7个节点就足够了；另外说明一点，etcd本质上可以不与master节点 部署在一起，只要master节点能通过网络连接etcd数据库即可。

kubelet：每个node节点上都有一个kubelet服务进程，kubelet作为连接master和各node之间的桥梁， 负责维护pod和容器的生命周期，当监听到master下发到本节点的任务时，比如创建、更新、终止pod 等任务，kubelet 即通过控制docker来创建、更新、销毁容器；

每个kubelet进程都会在api-server上注册本节点自身的信息，用于定期向master汇报本节点资源的使用 情况。

kube-proxy：kube-proxy运行在node节点上，在Node节点上实现Pod网络代理，维护网络规则和四层 负载均衡工作，kube-proxy会监听api-server中从而获取service和endpoint的变化情况，创建并维护路 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。

container-runtime：容器运行时环境，即运行容器所需要的一系列程序，目前k8s支持的容器运行时有

很多，如docker、rkt或其他，比较受欢迎的是docker，但是新版的k8s已经宣布弃用docker

## **kubelet**的功能、作用是什么？

kubelet部署在每个node节点上的，它主要有2个功能：

1、节点管理。kubelet启动时会向api-server进行注册，然后会定时的向api-server汇报本节点信息状 态，资源使用状态等，这样master就能够知道node节点的资源剩余，节点是否失联等等相关的信息了。 master知道了整个集群所有节点的资源情况，这对于 pod 的调度和正常运行至关重要。 2、pod管理。kubelet负责维护node节点上pod的生命周期，当kubelet监听到master的下发到自己节 点的任务时，比如要创建、更新、删除一个pod，kubelet 就会通过CRI（容器运行时接口）插件来调用 不同的容器运行时来创建、更新、删除容器；常见的容器运行时有docker、containerd、rkt等等这些容 器运行时，我们最熟悉的就是docker了，但在新版本的k8s已经弃用docker了，k8s1.24版本中已经使用 containerd作为容器运行时了。 3、容器健康检查。pod中可以定义启动探针、存活探针、就绪探针等3种，我们最常用的就是存活探 针、就绪探针，kubelet 会定期调用容器中的探针来检测容器是否存活，是否就绪，如果是存活探针，则 会根据探测结果对检查失败的容器进行相应的重启策略；

4、Metrics Server资源监控。在node节点上部署Metrics Server用于监控node节点、pod的CPU、内 存、文件系统、网络使用等资源使用情况，而kubelet则通过Metrics Server获取所在节点及容器的上的 数据。

## **kube-api-server**的端口是多少？各个**pod**是如何访问

**kube-api-server的？**

kube-api-server的端口是8080和6443，前者是http的端口，后者是https的端口，以我本机使用 kubeadm安装的k8s为例：

在命名空间的kube-system命名空间里，有一个名称为kube-api-master的pod，这个pod就是运行着 kube-api-server进程，它绑定了master主机的ip地址和6443端口，但是在default命名空间下，存在一 个叫kubernetes的服务，该服务对外暴露端口为443，目标端口6443，这个服务的ip地址是clusterip地 址池里面的第一个地址，同时这个服务的yaml定义里面并没有指定标签选择器，也就是说这个 kubernetes服务所对应的endpoint是手动创建的，该endpoint也是名称叫做kubernetes，该endpoint 的yaml定义里面代理到master节点的6443端口，也就是kube-api-server的IP和端口。这样一来，其他 pod访问kube-api-server的整个流程就是：pod创建后嵌入了环境变量，pod获取到了kubernetes这个 服务的ip和443端口，请求到kubernetes这个服务其实就是转发到了master节点上的6443端口的kube- api-server这个pod里面

## **k8s**中命名空间的作用是什么

namespace是kubernetes系统中的一种非常重要的资源，namespace的主要作用是用来实现多套环境 的资源隔离，或者说是多租户的资源隔离。 k8s通过将集群内部的资源分配到不同的namespace中，可以形成逻辑上的隔离，以方便不同的资源进 行隔离使用和管理。不同的命名空间可以存在同名的资源，命名空间为资源提供了一个作用域。 可以通过k8s的授权机制，将不同的namespace交给不同的租户进行管理，这样就实现了多租户的资源 隔离，还可以结合k8s的资源配额机制，限定不同的租户能占用的资源，例如CPU使用量、内存使用量等 等来实现租户可用资源的管理

## **k8s**提供了大量的**REST**接口，其中有一个是**Kubernetes Proxy API**接口，简述一下这个**Proxy**接口的作用，已经怎

**么使用**

kubernetes proxy api接口，从名称中可以得知，proxy是代理的意思，其作用就是代理rest请求； Kubernets API server 将接收到的rest请求转发到某个node上的kubelet守护进程的rest接口，由该 kubelet进程负责响应。我们可以使用这种Proxy接口来直接访问某个pod，这对于逐一排查pod异常问题 很有帮助。

## **pod**是什么

在kubernetes的世界中，k8s并不直接处理容器，而是使用多个容器共存的理念，这组容器就叫做pod。 pod是k8s中可以创建和管理的最小单元，是资源对象模型中由用户创建或部署的最小资源对象模型，其 他的资源对象都是用来支撑pod对象功能的，比如，pod控制器就是用来管理pod对象的，service或者 imgress资源对象是用来暴露pod引用对象的，persistentvolume资源是用来为pod提供存储等等，简而 言之，k8s不会直接处理容器，而是pod，pod才是k8s中可以创建和管理的最小单元，也是基本单元。

## **pod**的原理是什么

在微服务的概念里，一般的，一个容器会被设计为运行一个进程，除非进程本身产生子进程，这样，由 于不能将多个进程聚集在同一个单独的容器中，所以需要一种更高级的结构将容器绑定在一起，并将它 们作为一个单元进行管理，这就是k8s中pod的背后原理。

## **pod**有什么特点？

1、每个pod就像一个独立的逻辑机器，k8s会为每个pod分配一个集群内部唯一的IP地址，所以每个pod 都拥有自己的IP地址、主机名、进程等； 2、一个pod可以包含1个或多个容器，1个容器一般被设计成只运行1个进程，1个pod只可能运行在单个 节点上，即不可能1个pod跨节点运行，pod的生命周期是短暂，也就是说pod可能随时被消亡（如节点 异常，pod异常等情况）； 2、每一个pod都有一个特殊的被称为"根容器"的pause容器，也称info容器，pause容器对应的镜像属于 k8s平台的一部分，除了pause容器，每个pod还包含一个或多个跑业务相关组件的应用容器； 3、一个pod中的容器共享network命名空间；

4、一个pod里的多个容器共享pod IP，这就意味着1个pod里面的多个容器的进程所占用的端口不能相 同，否则在这个pod里面就会产生端口冲突；既然每个pod都有自己的IP和端口空间，那么对不同的两个 pod来说就不可能存在端口冲突； 5、应该将应用程序组织到多个pod中，而每个pod只包含紧密相关的组件或进程；

6、pod是k8s中扩容、缩容的基本单位，也就是说k8s中扩容缩容是针对pod而言而非容器。

## **pause**容器作用是什么

每个pod里运行着一个特殊的被称之为pause的容器，也称根容器，而其他容器则称为业务容器；创建 pause容器主要是为了为业务容器提供 Linux命名空间，共享基础：包括 pid、icp、net 等，以及启动 init 进程，并收割僵尸进程；这些业务容器共享pause容器的网络命名空间和volume挂载卷，当pod被创 建时，pod首先会创建pause容器，从而把其他业务容器加入pause容器，从而让所有业务容器都在同一 个命名空间中，这样可以就可以实现网络共享。pod还可以共享存储，在pod级别引入数据卷volume， 业务容器都可以挂载这个数据卷从而实现持久化存储

## **pod**的重启策略有哪些

pod重启容器策略是指针对pod内所有容器的重启策略，不是重启pod，其可以通过restartPolicy字段配 置pod重启容器的策略，如下:

Always: 当容器终止退出后，总是重启容器，默认策略就是Always。

OnFailure: 当容器异常退出，退出状态码非0时，才重启容器。

Never: 当容器终止退出，不管退出状态码是什么，从不重启容器。

## **pod**的镜像拉取策略有哪几种

pod镜像拉取策略可以通过imagePullPolicy字段配置镜像拉取策略，主要有3中镜像拉取策略，如下： IfNotPresent: 默认值，镜像在node节点宿主机上不存在时才拉取。

Always: 总是重新拉取，即每次创建pod都会重新从镜像仓库拉取一次镜像。

Never: 永远不会主动拉取镜像，仅使用本地镜像，需要你手动拉取镜像到node节点，如果node节点不 存在镜像则pod启动失败。

## **pod**的存活探针有哪几种？

kubernetes可以通过存活探针检查容器是否还在运行，可以为pod中的每个容器单独定义存活探针， kubernetes将定期执行探针，如果探测失败，将杀死容器，并根据restartPolicy策略来决定是否重启容 器，kubernetes提供了3种探测容器的存活探针，如下： httpGet：通过容器的IP、端口、路径发送http 请求，返回200-400范围内的状态码表示成功。 exec：在容器内执行shell命令，根据命令退出状态码是否为0进行判断，0表示健康，非0表示不健康。 TCPSocket：与容器的IP、端口建立TCP Socket链接，能建立则说明探测成功，不能建立则说明探测失 败。

grpc: 通过容器的IP、容器的端口向服务发送gRPC请求，返回serving则表示成功

## 存活探针的属性参数有哪几个？

initialDelaySeconds：表示在容器启动后延时多久秒才开始探测； periodSeconds：表示执行探测的频率，即间隔多少秒探测一次，默认间隔周期是10秒，最小1秒； timeoutSeconds：表示探测超时时间，默认1秒，最小1秒，表示容器必须在超时时间范围内做出响应， 否则视为本次探测失败； successThreshold：表示最少连续探测成功多少次才被认定为成功，默认是1，对于liveness必须是1， 最小值是1； failureThreshold：表示连续探测失败多少次才被认定为失败，默认是3，连续3次失败，k8s 将根据pod 重启策略对容器做出决定；

注意：定义存活探针时，一定要设置initialDelaySeconds属性，该属性为初始延时，如果不设置，默认 容器启动时探针就开始探测了，这样可能会存在 应用程序还未启动就绪，就会导致探针检测失败，k8s就会根据pod重启策略杀掉容器然后再重新创建容 器的莫名其妙的问题。

在生产环境中，一定要定义一个存活探针。

## **pod**的就绪探针有哪几种？

我们知道，当一个pod启动后，就会立即加入service的endpoint ip列表中，并开始接收到客户端的链接 请求，假若此时pod中的容器的业务进程还没有初始化完毕，那么这些客户端链接请求就会失败，为了解 决这个问题，kubernetes提供了就绪探针来解决这个问题的。 在pod中的容器定义一个就绪探针，就绪探针周期性检查容器，如果就绪探针检查失败了，说明该pod还 未准备就绪，不能接受客户端链接，则该pod将从endpoint列表中移除，被剔除了service就不会把请求 分发给该pod，然后就绪探针继续检查，如果随后容器就绪，则再重新把pod加回endpoint列表。k8s提

供了3种就绪探针，如下：

httpGet：通过容器的IP、容器的端口以及路径来发送http get请求，返回200-400范围内的状态码表示 请求成功。 exec：在容器内执行shell命令，它根据shell命令退出状态码是否为0进行判断，0表示健康，非0表示不 健康。

TCPSocket：通过容器的IP、端口建立TCP Socket链接，能正常建立链接，则说明探针成功，不能正常 建立链接，则探针失败。

grpc: 通过容器的IP、容器的端口向服务发送gRPC请求，返回serving则表示成功

## 就绪探针的属性参数有哪些

initialDelaySeconds：延时秒数，即容器启动多少秒后才开始探测，不写默认容器启动就探测； periodSeconds ：执行探测的频率（秒），默认为10秒，最低值为1；

timeoutSeconds ：超时时间，表示探测时在超时时间内必须得到响应，负责视为本次探测失败，默认为

1秒，最小值为1；

failureThreshold ：连续探测失败的次数，视为本次探测失败，默认为3次，最小值为1次；

successThreshold ：连续探测成功的次数，视为本次探测成功，默认为1次，最小值为1次；

## 就绪探针与存活探针区别是什么

两者作用不一样，存活探针是将检查失败的容器杀死，创建新的启动容器来保持pod正常工作； 就绪探针是，当就绪探针检查失败，并不重启容器，而是将pod移出endpoint，就绪探针确保了service 中的pod都是可用的，确保客户端只与正常的pod交互并且客户端永远不会知道系统存在问题

## 简单讲一下 **pod**创建过程

情况一、如果面试官问的是使用kubectl run命令创建的pod，可以这样说：

#注意：kubectl run 在旧版本中创建的是deployment，但在新的版本中创建的是pod则其创建过程不涉 及deployment

如果是单独的创建一个pod，则其创建过程是这样的： 1、首先，用户通过kubectl或其他api客户端工具提交需要创建的pod信息给apiserver； 2、apiserver验证客户端的用户权限信息，验证通过开始处理创建请求生成pod对象信息，并将信息存入 etcd，然后返回确认信息给客户端； 3、apiserver开始反馈etcd中pod对象的变化，其他组件使用watch机制跟踪apiserver上的变动； 4、scheduler发现有新的pod对象要创建，开始调用内部算法机制为pod分配最佳的主机，并将结果信息 更新至apiserver； 5、node节点上的kubelet通过watch机制跟踪apiserver发现有pod调度到本节点，尝试调用docker启动 容器，并将结果反馈apiserver；

6、apiserver将收到的pod状态信息存入etcd中。 至此，整个pod创建完毕。

情况二、如果面试官说的是使用deployment来创建pod，则可以这样回答： 1、首先，用户使用kubectl create命令或者kubectl apply命令提交了要创建一个deployment资源请 求；

2、api-server收到创建资源的请求后，会对客户端操作进行身份认证，在客户端的~/.kube文件夹下，已 经设置好了相关的用户认证信息，这样api-server会知道我是哪个用户，并对此用户进行鉴权，当api- server确定客户端的请求合法后，就会接受本次操作，并把相关的信息保存到etcd中，然后返回确认信 息给客户端。 3、apiserver开始反馈etcd中过程创建的对象的变化，其他组件使用watch机制跟踪apiserver上的变 动。

4、controller-manager组件会监听api-server的信息，controller-manager是有多个类型的，比如 Deployment Controller, 它的作用就是负责监听Deployment，此时Deployment Controller发现有新的

deployment要创建，那么它就会去创建一个ReplicaSet，一个ReplicaSet的产生，又被另一个叫做

ReplicaSet Controller监听到了，紧接着它就会去分析ReplicaSet的语义，它了解到是要依照ReplicaSet 的template去创建Pod, 它一看这个Pod并不存在，那么就新建此Pod，当Pod刚被创建时，它的 nodeName属性值为空，代表着此Pod未被调度。 5、调度器Scheduler组件开始介入工作，Scheduler也是通过watch机制跟踪apiserver上的变动，发现 有未调度的Pod，则根据内部算法、节点资源情况，pod定义的亲和性反亲和性等等，调度器会综合的选 出一批候选节点，在候选节点中选择一个最优的节点，然后将pod绑定该该节点，将信息反馈给api- server。 6、kubelet组件布署于Node之上，它也是通过watch机制跟踪apiserver上的变动，监听到有一个Pod应 该要被调度到自身所在Node上来，kubelet首先判断本地是否在此Pod，如果不存在，则会进入创建Pod 流程，创建Pod有分为几种情况，第一种是容器不需要挂载外部存储，则相当于直接docker run把容器 启动，但不会直接挂载docker网络，而是通过CNI调用网络插件配置容器网络，如果需要挂载外部存 储，则还要调用CSI来挂载存储。kubelet创建完pod，将信息反馈给api-server，api-servier将pod信息 写入etcd。

7、Pod建立成功后，ReplicaSet Controller会对其持续进行关注，如果Pod因意外或被我们手动退出， ReplicaSet Controller会知道，并创建新的Pod，以保持replicas数量期望值。

## 简单描述一下**pod**的终止过程

1、用户向apiserver发送删除pod对象的命令； 2、apiserver中的pod对象信息会随着时间的推移而更新，在宽限期内（默认30s），pod被视为dead； 3、将pod标记为terminating状态； 4、kubectl在监控到pod对象为terminating状态了就会启动pod关闭过程； 5、endpoint控制器监控到pod对象的关闭行为时将其从所有匹配到此endpoint的server资源endpoint 列表中删除； 6、如果当前pod对象定义了preStop钩子处理器，则在其被标记为terminating后会意同步的方式启动执 行；

7、pod对象中的容器进程收到停止信息；

8、宽限期结束后，若pod中还存在运行的进程，那么pod对象会收到立即终止的信息； 9、kubelet请求apiserver将此pod资源的宽限期设置为0从而完成删除操作，此时pod对用户已不可见。

## **pod**的生命周期有哪几种

Pending（挂起）：API server已经创建pod，但是该pod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（运行中）：Pod内所有的容器已经创建，且至少有一个容器处于运行状态、正在启动括正在重 启状态；

Succeed（成功）：Pod内所有容器均已退出，且不会再重启；

Failed（失败）：Pod内所有容器均已退出，且至少有一个容器为退出失败状态 Unknown（未知）：某于某种原因apiserver无法获取该pod的状态，可能由于网络通行问题导致；

## **pod**一致处于**pending**状态一般有哪些情况，怎么排查？

一个pod一开始创建的时候，它本身就是会处于pending状态，这时可能是正在拉取镜像，正在创建容器 的过程。

如果等了一会发现pod一直处于pending状态，那么我们可以使用kubectl describe命令查看一下pod的 Events详细信息。一般可能会有这么几种情况导致pod一直处于pending状态： 1、调度器调度失败。Scheduer调度器无法为pod分配一个合适的node节点。而这又会有很多种情况， 比如，node节点处在cpu、内存压力，导致无节点可调度；pod定义了资源请求，没有node节点满足资 源请求；node节点上有污点而pod没有定义容忍；pod中定义了亲和性或反亲和性而没有节点满足这些 亲和性或反亲和性；以上是调度器调度失败的几种情况。

2、pvc、pv无法动态创建。如果因为pvc或pv无法动态创建，那么pod也会一直处于pending状态，比如

要使用StatefulSet 创建redis集群，因为粗心大意，定义的storageClassName名称写错了，那么会造成 无法创建pvc，这种情况pod也会一直处于pending状态，或者，即使pvc是正常创建了，但是由于某些异 常原因导致动态供应存储无法正常创建pv，那么这种情况pod也会一直处于pending状态。

## **pod**的初始化容器是干什么的？

init container，初始化容器用于在启动应用容器之前完成应用容器所需要的前置条件，初始化容器本质 上和应用容器是一样的，但是初始化容器是仅允许一次就结束的任务，初始化容器具有两大特征： 1、初始化容器必须运行完成直至结束，若某初始化容器运行失败，那么kubernetes需要重启它直到成 功完成； 2、初始化容器必须按照定义的顺序执行，当且仅当前一个初始化容器成功之后，后面的一个初始化容器 才能运行；

## **pod**的资源请求、限制如何定义？

pod的资源请求、资源限制可以直接在pod中定义，主要包括两块内容，limits，限制pod能使用的最大 cpu和内存，requests，pod启动时申请的cpu和内存。

resources:

limits: cpu: 2

memory: 2G

requests: cpu: 1

memory: 500G

#资源配额

#限制最大资源，上限

#CPU限制，单位是code数

#内存最大限制

#请求资源（最小，下限）

#CPU请求，单位是code数

#内存最小请求

## **pod**的定义中有个**command**和**args**参数，这两个参数不会

**和docker镜像的entrypointc冲突吗**

不会。 在pod中定义的command参数用于指定容器的启动命令列表，如果不指定，则默认使用Dockerfile打包 时的启动命令，args参数用于容器的启动命令需要的参数列表；

特别说明：

kubernetes中的command、args其实是实现覆盖dockerfile中的ENTRYPOINT的功能的。当 1、如果command和args均没有写，那么使用Dockerfile的配置； 2、如果command写了但args没写，那么Dockerfile默认的配置会被忽略，执行指定的command； 3、如果command没写但args写了，那么Dockerfile中的ENTRYPOINT的会被执行，使用当前args的参 数；

4、如果command和args都写了，那么Dockerfile会被忽略，执行输入的command和args。

## 标签及标签选择器是什么，如何使用

标签是键值对类型，标签可以附加到任何资源对象上，主要用于管理对象，查询和筛选。标签常被用于 标签选择器的匹配度检查，从而完成资源筛选；一个资源可以定义一个或多个标签在其上面。

标签选择器，标签要与标签选择器结合在一起，标签选择器允许我们选择标记有特定标签的资源对象子 集，如pod，并对这些特定标签的pod进行查询，删除等操作。 标签和标签选择器最重要的使用之一在于，在deployment中，在pod模板中定义pod的标签，然后在 deployment定义标签选择器，这样就通过标签选择器来选择哪些pod是受其控制的，service也是通过标 签选择器来关联哪些pod最后其服务后端pod

## **service**是如何与**pod**关联的

答案是通过标签选择器，每一个由deployment创建的pod都带有标签，这样，service就可以定义标签选 择器来关联哪些pod是作为其后端了，就是这样，service就与pod管联在一起了。

## **service**的域名解析格式、**pod**的域名解析格式

service的DNS域名表示格式为..svc.，servicename是service的名称，namespace是service所处的命名 空间，clusterdomain是k8s集群设置的域名后缀，一般默认为 cluster.local，一般的，我们不会去改k8s 集群设置的域名后缀，同时，当pod要链接的svc处于同一个命名空间时，可以省略以及后面的.svc不 写，这样，就可以有下面三种方式来表示svc的域名

svc-nginx.default.svc.cluster.local svc-nginx.default

<clusterdomain> svc-nginx

命名空间省略不写

#完整的写法

#带命名空间写法,省略了后面的.svc.

#如果pod与svc在同一个命名空间，可以将

pod的DNS域名格式为：..pod. ，其中，pod-ip需要使用-将ip直接的点替换掉，namespace为pod所在 的命名空间，clusterdomain是k8s集群设置的域名后缀，一般默认为 cluster.local ，如果没有改变k8s 集群默认的域名后缀，则可以省略该后缀不写。除此之外，其他的均不可省略，这一点与svc域名有所不 同。

#进入default命名空间的busybox pod里面，测试下载文件

kubectl -n default exec -it deployment-busybox-567674bd67-lmrgw -- sh

wget 10-244-166-167.helm.pod.cluster.local:80

名空间里的IP为10.244.166.167的pod wget 10-244-166-167.helm.pod:80 域名后缀默认省略了

wget 10-244-166-143.default.pod:80 default命名空间里的IP为10.244.166.143的pod

wget 10-244-166-143.default:80

关键字

wget 10-244-166-143:80

空间和pod关键字

#可以正常下载，这里下载的是helm命

#可以正常下载，这里把k8s集群设置的

#可以正常下载，这里下载的是

#报错了，错误写法，说明不能省略pod

#报错了，错误写法，说明不能省略命名

对于StatefulSet创建的pod，statefulset.spec.serviceName字段解释如下： 也就是说StatefulSet创建的pod，其pod的域名为：pod-specific- string.serviceName.default.svc.cluster.local，而pod-specific-string就是pod的名称。 例如：redis-sts-0.redis-svc.default.svc.cluster.local:6379,redis-sts-1.redis-

svc.default.svc.cluster.local:6379,redis-sts-2.redis-svc.default.svc.cluster.local:6379,redis-sts- 3.redis-svc.default.svc.cluster.local:6379,redis-sts-4.redis-svc.default.svc.cluster.local:6379,redis- sts-5.redis-svc.default.svc.cluster.local:6379，pod里面的应用程序就可以拿这串字符串去连接Redis集

群了

## **service**的类型有哪几种

ClusterIP：表示service仅供集群内部使用，默认值就是ClusterIP类型 NodePort：表示service可以对外访问应用，会在每个节点上暴露一个端口，这样外部浏览器访问地址 为：任意节点的IP：NodePort就能连上service了 LoadBalancer：表示service对外访问应用，这种类型的service是公有云环境下的service，此模式需要 外部云厂商的支持，需要有一个公网IP地址

ExternalName：这种类型的service会把集群外部的服务引入集群内部，这样集群内直接访问service就

可以间接的使用集群外部服务了 一般情况下，service都是ClusterIP类型的，通过ingress接入的外部流量。

## 一个应用**pod**是如何发现**service**的，或者说，**pod**里面的

**容器用于是如何连接service的？**

有两种方式，一种是通过环境变量，另一种是通过service的dns域名方式。 1、环境变量：当pod被创建之后，k8s系统会自动为容器注入集群内有效的service名称和端口号等信息 为环境变量的形式，这样容器应用直接通过取环境变量值就能访问service了，如curl http://${WEBAPP\_SERVICE\_HOST}:{WEBAPP\_SERVICE\_PORT}

2、DNS方式：使用dns域名解析的前提是k8s集群内有DNS域名解析服务器，默认k8s中会有一个

CoreDNS作为k8s集群的默认DNS服务器提供域名解析服务器；service的DNS域名表示格式为..svc.， servicename是service的名称，namespace是service所处的命名空间，clusterdomain是k8s集群设置 的域名后缀，一般默认为 cluster.local ，这样容器应用直接通过service域名就能访问service了，如wget [http://svc-deployment-nginx.default.svc.cluster.local:80](http://svc-deployment-nginx.default.svc.cluster.local/)，另外，service的port端口如果定义了名 称，那么port也可以通过DNS进行解析，格式为：*.*...svc.

## 如何创建一个**service**代理外部的服务，或者换句话来说， 在**k8s**集群内的应用如何访问外部的服务，如数据库服务，

**缓存服务等?**

可以通过创建一个没有标签选择器的service来代理集群外部的服务。 1、创建service时不指定selector标签选择器，但需要指定service的port端口、端口的name、端口协议 等，这样创建出来的service因为没有指定标签选择器就不会自动创建endpoint； 2、手动创建一个与service同名的endpoint，endpoint中定义外部服务的IP和端口，endpoint的名称一 定要与service的名称一样，端口协议也要一样，端口的name也要与service的端口的name一样，不然 endpoint不能与service进行关联。 完成以上两步，k8s会自动将service和同名的endpoint进行关联，这样，k8s集群内的应用服务直接访问 这个service就可以相当于访问外部的服务了。

## **service**、**endpoint**、**kube-proxys**三种的关系是什么？

service：在kubernetes中，service是一种为一组功能相同的pod提供单一不变的接入点的资源。当 service被建立时，service的IP和端口不会改变，这样外部的客户端（也可以是集群内部的客户端）通过 service的IP和端口来建立链接，这些链接会被路由到提供该服务的任意一个pod上。通过这样的方式， 客户端不需要知道每个单独提供服务的pod地址，这样pod就可以在集群中随时被创建或销毁。 endpoint：service维护一个叫endpoint的资源列表，endpoint资源对象保存着service关联的pod的ip和 端口。从表面上看，当pod消失，service会在endpoint列表中剔除pod，当有新的pod加入，service就 会将pod ip加入endpoint列表；但是正在底层的逻辑是，endpoint的这种自动剔除、添加、更新pod的 地址其实底层是由endpoint controller控制的，endpoint controller负责监听service和对应的pod副本 的变化，如果监听到service被删除，则删除和该service同名的endpoint对象，如果监听到新的service 被创建或者修改，则根据该service信息获取得相关pod列表，然后创建或更新service对应的endpoint对 象，如果监听到pod事件，则更新它所对应的service的endpoint对象。

kube-proxy：kube-proxy运行在node节点上，在Node节点上实现Pod网络代理，维护网络规则和四层 负载均衡工作，kube-proxy会监听api-server中从而获取service和endpoint的变化情况，创建并维护路 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

## 无头**service**和普通的**service**有什么区别，无头**service**使

**用场景是什么**

无头service没有cluster ip，在定义service时将 service.spec.clusterIP：None，就表示创建的是无头 service。 普通的service是用于为一组后端pod提供请求连接的负载均衡，让客户端能通过固定的service ip地址来 访问pod，这类的pod是没有状态的，同时service还具有负载均衡和服务发现的功能。普通service跟我 们平时使用的nginx反向代理很相识。

但是，试想这样一种情况，有6个redis pod ,它们相互之间要通信并要组成一个redis集群，不在需要所谓 的service负载均衡，这时无头service就是派上用场了，无头service由于没有cluster ip，kube-proxy就 不会处理它也就不会对它生成规则负载均衡，无头service直接绑定的是pod 的ip。无头service仍会有标 签选择器，有标签选择器就会有endpoint资源。 使用场景：无头service一般用于有状态的应用场景，如Kaka集群、Redis集群等，这类pod之间需要相互 通信相互组成集群，不在需要所谓的service负载均衡。

## **deployment**怎么扩容或缩容？

直接修改pod副本数即可，可以通过下面的方式来修改pod副本数： 1、直接修改yaml文件的replicas字段数值，然后kubectl apply -f xxx.yaml来实现更新； 2、使用kubectl edit deployment xxx 修改replicas来实现在线更新；

3、使用kubectl scale --replicas=5 deployment/deployment-nginx命令来扩容缩容。

## **deployment**的更新升级策略有哪些？

deployment的升级策略主要有两种。

1、Recreate 重建更新：这种更新策略会杀掉所有正在运行的pod，然后再重新创建的pod； 2、rollingUpdate 滚动更新：这种更新策略，deployment会以滚动更新的方式来逐个更新pod，同时通 过设置滚动更新的两个参数maxUnavailable、maxSurge来控制更新的过程。

## **deployment**的滚动更新策略有两个特别主要的参数，解释

**一下它们是什么意思**

maxUnavailable：最大不可用数，maxUnavailable用于指定deployment在更新的过程中不可用状态的 pod的最大数量，maxUnavailable的值可以是一个整数值，也可以是pod期望副本的百分比，如25%， 计算时向下取整。 maxSurge：最大激增数，maxSurge指定deployment在更新的过程中pod的总数量最大能超过pod副本 数多少个，maxUnavailable的值可以是一个整数值，也可以是pod期望副本的百分比，如25%，计算时 向上取整。

## **deployment**更新的命令有哪些

可以通过三种方式来实现更新deployment。 1、直接修改yaml文件的镜像版本，然后kubectl apply -f xxx.yaml来实现更新； 2、使用kubectl edit deployment xxx 实现在线更新；

3、使用kubectl set image deployment/nginx busybox=busybox nginx=nginx:1.9.1 命令来更新

## 简述一下**deployment**的更新过程

eployment是通过控制replicaset来实现，由replicaset真正创建pod副本，每更新一次deployment，都 会创建新的replicaset，下面来举例deployment的更新过程：

假设要升级一个nginx-deployment的版本镜像为nginx:1.9，deployment的定义滚动更新参数如下：

replicas: 3

deployment.spec.strategy.type: RollingUpdate maxUnavailable：25%

maxSurge：25%

通过计算我们得出，3\*25%=0.75，maxUnavailable是向下取整，则maxUnavailable=0，maxSurge是 向上取整，则maxSurge=1，所以我们得出在整个deployment升级镜像过程中，不管旧的pod和新的 pod是如何创建消亡的，pod总数最大不能超过3+maxSurge=4个，最大pod不可用数3- maxUnavailable=3个。

现在具体讲一下deployment的更新升级过程：

使用kubectl set image deployment/nginx nginx=nginx:1.9 --record 命令来更新； 1、deployment创建一个新的replaceset，先新增1个新版本pod，此时pod总数为4个，不能再新增了， 再新增就超过pod总数4个了；旧=3，新=1，总=4； 2、减少一个旧版本的pod，此时pod总数为3个，这时不能再减少了，再减少就不满足最大pod不可用数

3个了；旧=2，新=1，总=3；

3、再新增一个新版本的pod，此时pod总数为4个，不能再新增了；旧=2，新=2，总=4；

4、减少一个旧版本的pod，此时pod总数为3个，这时不能再减少了；旧=1，新=2，总=3；

5、再新增一个新版本的pod，此时pod总数为4个，不能再新增了；旧=1，新=3，总=4；

6、减少一个旧版本的pod，此时pod总数为3个，更新完成，pod都是新版本了；旧=0，新=3，总=3；

## **deployment**的回滚使用什么命令

在升级deployment时kubectl set image 命令加上 --record 参数可以记录具体的升级历史信息，使用 kubectl rollout history deployment/deployment-nginx 命令来查看指定的deployment升级历史记录， 如果需要回滚到某个指定的版本，可以使用kubectl rollout undo deployment/deployment-nginx --to- revision=2 命令来实现。

## 讲一下都有哪些存储卷，作用分别是什么**?**

|  |  |  |
| --- | --- | --- |
| **卷** | **作用** | **常用场景** |
| emptyDir | 用于存储临时数据的简单空目录 | 一个pod中的多个容器需要共享彼此的数 据 ，emptyDir的数据随着容器的消亡也 会销毁 |
| hostPath | 用于将目录从工作节点的文件系统 挂载到pod中 | 不常用，缺点是，pod的调度不是固定 的，也就是当pod消失后deployment重 新创建一个pod，而这pod如果不是被调 度到之前pod的节点，那么该pod就不能 访问之前的数据 |
| configMap | 用于将非敏感的数据保存到键值对 中，使用时可以使用作为环境变 量、命令行参数arg，存储卷被 pods挂载使用 | 将应用程序的不敏感配置文件创建为 configmap卷，在pod中挂载configmap 卷，可是实现热更新 |

|  |  |  |
| --- | --- | --- |
| **卷** | **作用** | **常用场景** |
| secret | 主要用于存储和管理一些敏感数 据，然后通过在 Pod 的容器里挂 载 Volume 的方式或者环境变量的 方式访问到这些 Secret 里保存的 信息了，pod会自动解密Secret 的 信息 | 将应用程序的账号密码等敏感信息通过  secret卷的形式挂载到pod中使用 |
| downwardApi | 主要用于暴露pod元数据，如pod 的名字 | pod中的应用程序需要指定pod的name 等元数据，就可以通过downwardApi 卷 的形式挂载给pod使用 |
| projected | 这是一种特殊的卷，用于将上面这 些卷一次性的挂载给pod使用 | 将上面这些卷一次性的挂载给pod使用 |
| pvc | pvc是存储卷声明 | 通常会创建pvc表示对存储的申请，然后 在pod中使用pvc |
| 网络存储卷 | pod挂载网络存储卷，这样就能将 数据持久化到后端的存储里 | 常见的网络存储卷有nfs存储、glusterfs 卷、ceph rbd存储卷 |

**pv的访问模式有哪几种**

ReadWriteOnce，简写：RWO 表示，只仅允许单个节点以读写方式挂载； ReadOnlyMany，简写：ROX 表示，可以被许多节点以只读方式挂载； ReadWriteMany，简写：RWX 表示，可以被多个节点以读写方式挂载；

## **pv**的回收策略有哪几种

主要有2中回收策略：retain 保留、delete 删除。 Retain：保留，该策略允许手动回收资源，当删除PVC时，PV仍然存在，PV被视为已释放，管理员可以 手动回收卷。 Delete：删除，如果Volume插件支持，删除PVC时会同时删除PV，动态卷默认为Delete，目前支持 Delete的存储后端包括AWS EBS，GCE PD，Azure Disk，OpenStack Cinder等。

## 在**pv**的生命周期中，一般有几种状态

pv一共有4中状态，分别是： 创建pv后，pv的的状态有以下4种：Available（可用）、Bound（已绑定）、Released（已释放）、 Failed（失败）

Available，表示pv已经创建正常，处于可用状态； Bound，表示pv已经被某个pvc绑定，注意，一个pv一旦被某个pvc绑定，那么该pvc就独占该pv，其他 pvc不能再与该pv绑定；

Released，表示pvc被删除了，pv状态就会变成已释放； Failed，表示pv的自动回收失败；

## 存储类的资源回收策略

主要有2中回收策略，delete 删除，默认就是delete策略、retain 保留。 Retain：保留，该策略允许手动回收资源，当删除PVC时，PV仍然存在，PV被视为已释放，管理员可以 手动回收卷。 Delete：删除，如果Volume插件支持，删除PVC时会同时删除PV，动态卷默认为Delete，目前支持 Delete的存储后端包括AWS EBS，GCE PD，Azure Disk，OpenStack Cinder等。

## 怎么使一个**node**脱离集群调度，比如要停机维护单又不能

**影响业务应用**

crodon背后的原理其实就是打污点 使用kubectl drain 命令

## **pv**存储空间不足怎么扩容

一般的，我们会使用动态分配存储资源，在创建storageclass时指定参数 allowVolumeExpansion： true，表示允许用户通过修改pvc申请的存储空间自动完成pv的扩容，当增大pvc的存储空间时，不会重 新创建一个pv，而是扩容其绑定的后端pv。这样就能完成扩容了。但是allowVolumeExpansion这个特 性只支持扩容空间不支持减少空间。

## **k8s**生产中遇到什么特别影响深刻的问题吗，问题排查解决

**思路是怎么样的？**

前端的lb负载均衡服务器上的keepalived出现过脑裂现象。

1、当时问题现象是这样的，vip同时出现在主服务器和备服务器上，但业务上又没受到影响；

2、这时首先去查看备服务器上的keepalived日志，发现有日志信息显示凌晨的时候备服务器出现了vrrp 协议超时，所以才导致了备服务器接管了vip；查看主服务器上的keepalived日志，没有发现明显的报错 信息，继续查看主服务器和备服务器上的keepalived进程状态，都是running状态的；查看主服务器上检 测脚本所检测的进程，其进程也是正常的，也就是说主服务器根本没有成功执行检测脚本（成功执行检 查脚本是会kill掉keepalived进程，脚本里面其实就是配置了检查nginx进程是否存活，如果检查到nginx 不存活则kill掉keepalived，这样来实现备服务器接管vip）； 3、排查服务器上的防火墙、selinux，防火墙状态和selinux状态都是关闭着的； 4、使用tcpdump工具在备服务器上进行抓取数据包分析，分析发现，现在确实是备接管的vip，也确实 是备服务器也在对外发送vrrp心跳包，所以现在外部流量应该都是流入备服务器上的vip； 5、怀疑：主服务器上设置的vrrp心跳包时间间隔太长，以及检测脚本设置的检测时间设置不合理导致该 问题； 6、修改vrrp协议的心跳包时间间隔，由原来的2秒改成1秒就发送一次心跳包；检测脚本的检测时间也修 改短一点，同时还修改检测脚本的检测失败的次数，比如连续检测2次失败才认定为检测失败； 7、重启主备上的keepalived，现在keepalived是正常的，主服务器上有vip，备服务器上没有vip； 8、持续观察：第二天又发现keepalived出现过脑裂现象，vip又同时出现在主服务器和备服务器上，又 是凌晨的时候备服务器显示vrrp心跳包超时，所以才导致备服务器接管了vip； 9、同样的时间，都是凌晨，vrrp协议超时；很奇怪，很有理由怀疑是网络问题，询问第三方厂家上层路 由器是否禁止了vrrp协议，第三方厂家回复，没有禁止vrrp协议；

10、百度、看官方文档求解；

11、百度、看官网文档得知，keepalived有2种传播模式，一种是组播模式，一种是单播模式， keepalived默认在组播模式下工作，主服务器会往主播地址224.0.0.18发送心跳包，当局域网内有多个 keepalived实例的时候，如果都用主播模式，会存在冲突干扰的情况，所以官方建议使用单播模式通 信，单播模式就是点对点通行，即主向备服务器一对一的发送心跳包； 12、将keepalived模式改为单播模式，继续观察，无再发生脑裂现象。问题得以解决。

答：测试环境二进制搭建etcd集群，etcd集群出现2个leader的现象。

1、问题现象就是：刚搭建的k8s集群，是测试环境的，搭建完成之后发现，使用kubectl get nodes 显示 没有资源，kubectl get namespace 一会能正常显示全部的命名空间，一会又显示不了命名空间，这种 奇怪情况。 2、当时经验不是很足，第一点想到的是不是因为网络插件calico没装导致的，但是想想，即使没有安装 网络插件，最多是node节点状态是notready，也不可能是没有资源发现呀； 3、然后想到etcd数据库，k8s的资源都是存储在etcd数据库中的；

4、查看etcd进程服务的启动状态，发现etcd服务状态是处于running状态，但是日志有大量的报错信 息，日志大概报错信息就是集群节点的id不匹配，存在冲突等等报错信息； 5、使用etcdctl命令查看etcd集群的健康状态，发现集群是health状态，但是居然显示有2个leader，这 很奇怪（当初安装etcd的时候其实也只是简单看到了集群是健康状态，然后没注意到有2个leader，也没 太关注etcd服务进程的日志报错信息，以为etcd集群状态是health状态就可以了） 6、现在etcd出现了2个leader，肯定是存在问题的； 7、全部检测一遍etcd的各个节点的配置文件，确认配置文件里面各个参数配置都没有问题，重启etcd集 群，报错信息仍未解决，仍然存在2个leader； 8、尝试把其中一个leader节点踢出集群，然后再重新添加它进入集群，仍然是报错，仍然显示有2个 leader； 9、尝试重新生成etcd的证书，重新颁发etcd的证书，问题仍然存在，仍然显示有2个leader；日志仍是 报错集群节点的id不匹配，存在冲突； 10、计算etcd命令的MD5值，确保各个节点的etcd命令是相同的，确保在scp传输的时候没有损耗等 等，问题仍未解决；

11、无解，请求同事，架构师介入帮忙排查问题，仍未解决； 12、删除全部etcd相关的文件，重新部署etcd集群，etcd集群正常了，现在只有一个leader，使用命令 kubectl get nodes 查看节点，也能正常显示了； 13、最终问题的原因也没有定位出来，只能怀疑是环境问题了，由于是刚部署的k8s测试环境，etcd里面 没有数据，所以可以删除重新创建etcd集群，如果是线上环境的etcd集群出现这种问题，就不能随便删 除etcd集群了，必须要先进行数据备份才能进行其他方法的处理。

# **redis**

## 什么是 **Redis?**

Redis 是完全开源免费的，遵守 BSD 协议，是一个高性能的 key-value 数据 库。

Redis 与其他 key - value 缓存产品有以下三个特点：

1. Redis 支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时 候可以再次加载进行使 用。

2. Redis 不仅仅支持简单的 key-value 类型的数据，同时还提供 list， set，zset，hash 等数据结构的 存储。

3. Redis 支持数据的备份，即 master-slave 模式的数据备份。 Redis 优势：

4. 性能极高 – Redis 能读的速度是 110000 次/s,写的速度是 81000 次 /s 。

5. 丰富的数据类型 – Redis 支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型 操作。

6. 原子 – Redis 的所有操作都是原子性的，意思就是要么成功执行要么失 败完全不执行。单个操作是 原子性的。多个操作也支持事务，即原子性， 通过 MULTI 和 EXEC 指令包起来。

7. 丰富的特性 – Redis 还支持 publish/subscribe, 通知, key 过期等等 特性。

## **Redis** 与其他 **key-value** 存储有什么不同？

1. Redis 有着更为复杂的数据结构并且提供对他们的原子性操作，这是一个 不同于其他数据库的进化 路径。Redis 的数据类型都是基于基本数据结构 的同 时对程序员透明，无需进行额外的抽象。

2. Redis 运行在内存中但是可以持久化到磁盘，所以在对不同数据集进行高 速读写时需要权衡内存， 因为数据量不能大于硬件内存。在内存数据库方 面的 另一个优点是，相比在磁盘上相同的复杂的数 据结构，在内存中操作 起来非常 简单，这样 Redis 可以做很多内部复杂性很强的事情。同时，在 磁盘格式方面 他们是紧凑的以追加的方式产生的，因为他们并不需要进行 随机访问

## **Redis** 的数据类型？

Redis 支持五种数据类型：string（字符串），hash（哈希），list（列 表），set（集合）及 zsetsorted set：有序集合)。 我们实际项目中比较常用的是 string，hash 如果你是 Redis 中高级用户， 还 需要加上下面几种数据结构 HyperLogLog、Geo、Pub/Sub。 如果你说还玩过 Redis Module，像 BloomFilter，RedisSearch，Redis-ML， 面试官得眼睛就开始发亮了。

## 使用 **Redis** 有哪些好处？

1. 速度快，因为数据存在内存中，类似于 HashMap，HashMap 的优势就是查 找和操作的时间复杂 度都是 O1)

2. 支持丰富数据类型，支持 string，list，set，Zset，hash 等

3. 支持事务，操作都是原子性，所谓的原子性就是对数据的更改要么全部执 行，要么全部不执行

4. 丰富的特性：可用于缓存，消息，按 key 设置过期时间，过期后将会自动删除

## **Redis** 相比 **Memcached** 有哪些优势？

1. 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

1. Redis 的速度比 Memcached 快很

3. Redis 可以持久化其数据

## **Memcache** 与 **Redis** 的区别都有哪些？

1. 存储方式 Memecache 把数据全部存在内存之中，断电后会挂掉，数据不 能超过内存大小。 Redis

有部份存在硬盘上，这样能保证数据的持久性。

2. 数据支持类型 Memcache 对数据类型支持相对简单。 Redis 有复杂的数 据类型。

3. 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

## **Redis** 是单进程单线程的

Redis 是单进程单线程的，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

## 一个字符串类型的值能存储最大容量是多少

513M

## **Redis** 的持久化机制是什么？各自的优缺点？

Redis 提供两种持久化机制 RDB 和 AOF 机制：

RDBRedis DataBase)持久化方式：

是指用数据集快照的方式半持久化模式)记录 Redis 数据库的所有键值对,在某 个时间点将数据写入一个 临时文件，持久化结束后，用这个临时文件替换上次 持久化的文件，达到数据恢复。

优点：

1. 只有一个文件 dump.rdb，方便持久化。

2. 容灾性好，一个文件可以保存到安全的磁盘。

3. 性能最大化，fork 子进程来完成写操作，让主进程继续处理命令，所以 是 IO 最大化。使用单独子 进程来进行持久化，主进程不会进行任何 IO 操 作，保证了 Redis 的高性能)

4. 相对于数据集大时，比 AOF 的启动效率更高。 缺点：

1. 数据安全性低。RDB 是间隔一段时间进行持久化，如果持久化之间 Redis 发生 故障，会发生数据 丢失。所以这种方式更适合数据要求不严谨的时候。

AOFAppend-only file)持久化方式：

是指所有的命令行记录以 Redis 命令请求协议的格式完全持久化存储)保存为 aof 文件。 优点：

1. 数据安全，aof 持久化可以配置 appendfsync 属性，有 always，每进行 一次命令操作就记录到

aof 文件中一次。

2. 通过 append 模式写文件，即使中途服务器宕机，可以通过 redis□check-aof 工具解决数据一致性 问题。

3. AOF 机制的 rewrite 模式。AOF 文件没被 rewrite 之前（文件过大时会 对命令进行合并重写），可 以删除其中的某些命令（比如误操作的 flushall）)

缺点：

1. AOF 文件比 RDB 文件大，且恢复速度慢。

2. 数据集大的时候，比 rdb 启动效率低

## **Redis** 常见性能问题和解决方案：

1. Master 最好不要写内存快照，如果 Master 写内存快照，save 命令调度 rdbSave 函数，会阻塞主 线程的工作，当快照比较大时对性能影响是非常大 的，会间断性暂停服务

2. 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 （

3. 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

4. 尽量避免在压力很大的主库上增加从

5. 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1<- Slave2 <- Slave3… 这样的结构方便解决单点故障问题，实现 Slave 对 Master 的替换。如果 Master 挂了，可以立刻启 用 Slave1 做 Master，其 他不变。

## **Redis** 过期键的删除策略？ 、

1. 定时删除:在设置键的过期时间的同时，创建一个定时器 timer). 让定时 器在键的过期时间来临时， 立即执行对键的删除操作。

2. 惰性删除:放任键过期不管，但是每次从键空间中获取键时，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。

3. 定期删除:每隔一段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。

## **Redis** 的回收策略（淘汰策略）**?**

1. volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选 最近最少使用的数据淘汰

2. volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选 将要过期的数据淘汰

1. volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任 意选择数据淘汰

4. allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘 汰

5. allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰

6. no-enviction（驱逐）：禁止驱逐数据 注意这里的 6 种机制，volatile 和 allkeys 规定了是对已设置 过期时间的数 据集淘汰数据还是从全部数据集淘汰数据，后面的 lru、ttl 以及 random 是 三种不同 的淘汰策略，再加上一种 no-enviction 永不回收的策略。

## 使用策略规则：

1. 如果数据呈现幂律分布，也就是一部分数据访问频率高，一部分数据访问 频率低，则使用 allkeys- lr

2. 如果数据呈现平等分布，也就是所有的数据访问频率都相同，则使用 allkeys-random

## 为什么 **Redis** 需要把所有数据放到内存中？

1. Redis 为了达到最快的读写速度将数据都读到内存中，并通过异步的方式 将数 据写入磁盘。所以 Redis 具有快速和数据持久化的特征。如果不将数据放 在 内存中，磁盘 I/O 速度为严重影响 Redis 的性能。在内存越来越便宜的今 天， Redis 将会越来越受欢迎。如果设置了最大使用的内存，则数 据已有记录 数达 到内存限值后不能继续插入新值。

## **Redis** 的同步机制了解么？

1. Redis 可以使用主从同步，从从同步。第一次同步时，主节点做一次 bgsave， 并同时将后续修改 操作记录到内存 buffer，待完成后将 rdb 文件全 量同步到 复制节点，复制节点接受完成后将 rdb 镜像加载到内存。加载完成 后，再通 知主节点将期间修改的操作记录同步到复制节点进行重放就完 成了同步过程。

## **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，集群的原理是什么？

1. 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。 2. 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** 集群方案什么情况下会导致整个集群不可用？

有 A，B，C 三个节点的集群,在没有复制模型的情况下,如果节点 B 失败了， 那么整个集群就会以为缺少

5501-11000 这个范围的槽而不可用。

## **Redis** 如何设置密码及验证密码？

设置密码：config set requirepass 123456

授权密码：auth 123456

## 说说 **Redis** 哈希槽的概念？

Redis 集群没有使用一致性 hash,而是引入了哈希槽的概念，Redis 集群 有 16384 个哈希槽，每个 key

通过 CRC16 校验后对 16384 取模来决定放置 哪 个槽，集群的每个节点负责一部分 hash 槽。

## **Redis** 集群的主从复制模型是怎样的？

为了使在部分节点失败或者大部分节点无法通信的情况下集群仍然可用， 所以 集群使用了主从复制模型, 每个节点都会有 N-1 个复制品。

## **Redis** 集群会有写操作丢失吗？为什么？

Redis 并不能保证数据的强一致性，这意味这在实际中集群在特定的条件 下可 能会丢失写操作。

## **Redis** 集群之间是如何复制的？

异步复制

## **Redis** 集群最大节点个数是多少？

16384 个。

## **Redis** 集群如何选择数据库？

Redis 集群目前无法做数据库选择，默认在 0 数据库。

## 怎么测试 **Redis** 的连通性

使用 ping 命令。

## 怎么理解 **Redis** 事务？

1. 事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执 行。事务在执行的过程 中，不会被其他客户端发送来的命令请求所打断。

2. 事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执 行。

## **Redis** 事务相关的命令有哪几个？

MULTI、EXEC、DISCARD、WATCH

## **Redis key** 的过期时间和永久有效分别怎么设置？

EXPIRE 和 PERSIST 命令。

## **Redis** 如何做内存优化？

尽可能使用散列表（hashes），散列表（是说散列表里面存储的数少）使用的 内存非常小，所以你应该 尽可能的将你的数据模型抽象到一个散列表里面。比 如你的 web 系统中有一个用户对象，不要为这个用 户的名称，姓氏，邮箱，密 码设置单独的 key,而是应该把这个用户的所有信息存储到一张散列表里面。

## **Redis** 回收进程如何工作的？

一个客户端运行了新的命令，添加了新的数据。Redi 检查内存使用情况， 如 果大于 maxmemory 的限 制, 则根据设定好的策略进行回收。一个新的命令被 执 行，等等。所以我们不断地穿越内存限制的边 界，通过不断达到边界然后不 断 地回收回到边界以下。如果一个命令的结果导致大量内存被使用（例如 很大 的 集合的交集保存到一个新的键），不用多久内存限制就会被这个内存使用量超 越。

## 都有哪些办法可以降低 **Redis** 的内存使用情况呢？

如果你使用的是 32 位的 Redis 实例，可以好好利用 Hash,list,sorted set,set 等集合类型数据，因为通常 情况下很多小的 Key-Value 可以用更紧凑 的方式存放到一起。

## **Redis** 的内存用完了会发生什么？

如果达到设置的上限，Redis 的写命令会返回错误信息（但是读命令还可 以正 常返回。）或者你可以将

Redis 当缓存来使用配置淘汰机制，当 Redis 达到 内存上限时会冲刷掉旧的内容。

## 一个 **Redis** 实例最多能存放多少的 **keys**？

List、Set、 Sorted Set 他们最多能存放多少元素？ 理论上 Redis 可以处理多达 2的32次方 的 keys，并 且在实际中进行了测试，每个实 例至少存放了 2 亿 5 千万的 keys。我们正在测试一些较大的值。任何 list、 set、和 sorted set 都可以放 232 个元素。换句话说，Redis 的存储极限是 系统中的可用内存值。

## **MySQL** 里有 **2000w** 数据，**Redis** 中只存 **20w** 的数据，如

**何保证 Redis 中的数据都是热点数据？**

Redis 内存数据集大小上升 到一定大小的时候，就会施行数据淘汰策略。 相关知识：Redis 提供 6 种数 据淘汰策略：

volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选 最近最少使用的数据淘汰 volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选 将要过期的数据淘汰 volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任 意选择数据淘汰 allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘 汰

allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰 no-enviction（驱逐）：禁止驱逐数据

## **Redis** 最适合的场景？

1. 会话缓存（Session Cache） 最常用的一种使用 Redis 的情景是会话缓存（session cache）。用 Redis 缓 存会话比其他存储（如 Memcached）的优势在于：Redis 提供持久化。当维护 一个不是 严格要求一致性的缓存时，如果用户的购物车信息全部丢失，大部分 人都会不高兴的，现在，他们 还会这样吗？ 幸运的是，随着 Redis 这些年的 改进，很容易找到怎么恰当的使用 Redis 来缓存会 话的文档。甚至广为人知的 商业平台 Magento 也提供 Redis 的插件。

2. 全页缓存（FPC） 除基本的会话 token 之外，Redis 还提供很简便的 FPC 平台。回到一致性问 题，

即使重启了 Redis 实例，因为有磁盘的持久化，用户也不会看到页面加载 速度的下降，这是一个极 大改进，类似 PHP 本地 FPC。 再次以 Magento 为 例，Magento 提供一个插件来使用 Redis 作为 全页缓存后端。 此外，对 WordPress 的用户来说，Pantheon 有一个非常好的插件 wp-redis，这 个插件 能帮助你以最快速度加载你曾浏览过的页面。

3. 队列 Redis 在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得 Redis 能作为一个很好 的消息队列平台来使用。Redis 作为队列使用的操作， 就类似于本地程序语言（如 Python）对 list 的 push/pop 操作。 如果你快 速的在 Google 中搜索“Redis queues”，你马上就能找到大量的开源 项目， 这些项目的目的就是利用 Redis 创建非常好的后端工具，以满足各种队列需 求。例如， Celery 有一个后台就是使用 Redis 作为 broker，你可以从这里去 查看。

4. 排行榜/计数器 Redis 在内存中对数字进行递增或递减的操作实现的非常好。集合（Set）和有 序集 合（Sorted Set）也使得我们在执行这些操作的时候变的非常简单，Redis 只是正好提供了这两种 数据结构。所以，我们要从排序集合中获取到排名最靠 前的 10 个用户–我们称之为 “user\_scores”，我们只需要像下面一样执行即 可： 当然，这是假定你是根据你用户的分数做递增 的排序。如果你想返回用户 及用户的分数，你需要这样执行： ZRANGE user\_scores 0 10 WITHSCORES Agora Games 就是一个很好的例子，用 Ruby 实现的，它的排行榜就是使用 Redis 来存储数据的，你可以在这里看到。

5. 发布/订阅 最后（但肯定不是最不重要的）是 Redis 的发布/订阅功能。发布/订阅的使用 场景确实 非常多。我已看见人们在社交网络连接中使用，还可作为基于发布/订 阅的脚本触发器，甚至用 Redis 的发布/订阅功能来建立聊天系统！

## 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，如果将它们全部找出来？**

使用 keys 指令可以扫出指定模式的 key 列表。 对方接着追问：如果这个 Redis 正在给线上的业务提供 服务，那使用 keys 指 令会有什么问题？ 这个时候你要回答 Redis 关键的一个特性：Redis 的单线程 的。keys 指令会 导致线程阻塞一段时间，线上服务会停顿，直到指令执行完毕，服务才能恢 复。这个时 候可以使用 scan 指令，scan 指令可以无阻塞的提取出指定模式的 70 key 列表，但是会有一定的重复概 率，在客户端做一次去重就可以了，但是整 体所花费的时间会比直接用 keys 指令长。

## 如果有大量的 **key** 需要设置同一时间过期，一般需要注意

**什 么？**

如果大量的 key 过期时间设置的过于集中，到过期的那个时间点，Redis 可能 会出现短暂的卡顿现象。 一般需要在时间上加一个随机值，使得过期时间 分散 一些。

## 使用过 **Redis** 做异步队列么，你是怎么用的？

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。

如果对方追问可不可以不 用 sleep 呢？

list 还有个指令叫 blpop，在没有消息的时候，它会阻塞住直 到消息到来。 如果对方追问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。

如果对方追问 pub/sub 有什么缺点？ 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。 如果对方追问 Redis 如何实现延时队列？

使用 sortedset，拿时间 戳作为 score，消息内容作为 key 调用 zadd 来生产消息，消费者用

zrangebyscore 指令获取 N 秒之前的数据轮询进行处理。

## 使用过 **Redis** 分布式锁么，它是什么回事 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。 这时候对方会告诉你 说你回答得不错，然后接着问如果在 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？可以同时把 setnx 和 expire 合成一条指令来用的！

# **mysql**

## 据库三大范式是什么

1. 第一范式：每个列都不可以再拆分。

2. 第二范式：在第一范式的基础上，非主键列完全依赖于主键，而不能是依 赖于主键的一部分。

3. 第三范式：在第二范式的基础上，非主键列只依赖于主键，不依赖于其他 非主键。 在设计数据库结 构的时候，要尽量遵守三范式，如果不遵守，必须有足够 的理由。比如性能。事实上我们经常会为 了性能而妥协数据库的设计。

## **MySQL** 有关权限的表都有哪几个？

MySQL 服务器通过权限表来控制用户对数据库的访问，权限表存放在 MySQL 数 据库里，由 mysql\_install\_db 脚本初始化。这些权限表分别 user，db， table\_priv，columns\_priv 和 host。下面 分别介绍一下这些表的结构和内容：

1. user 权限表：记录允许连接到服务器的用户帐号信息，里面的权限是全局 级的。

2. db 权限表：记录各个帐号在各个数据库上的操作权限。

3. table\_priv 权限表：记录数据表级的操作权限。

4. columns\_priv 权限表：记录数据列级的操作权限。

5. host 权限表：配合 db 权限表对给定主机上数据库级操作权限作更细致的控 制。这个权限表不受

GRANT 和 REVOKE 语句的影响。

## **MySQL** 的 **Binlog** 有有几种录入格式？分别有什么区别？

有三种格式，statement，row 和 mixed。

1. statement 模式下，每一条会修改数据的 SQL 都会记录在 Binlog 中。不需 要记录每一行的变化， 减少了 Binlog 日志量，节约了 IO，提高性能。由于 sql 的执行是有上下文的，因此在保存的时候 需要保存相关的信息，同时还 有一些使用了函数之类的语句无法被记录复制。

2. row 级别下，不记录 SQL 语句上下文相关信息，仅保存哪条记录被修改。记 录单元为每一行的改 动，基本是可以全部记下来但是由于很多操作，会导 致大量行的改动(比如 alter table)，因此这种 模式的文件保存的信息太 多，日志量太大。

3. mixed，一种折中的方案，普通操作使用 statement 记录，当无法使用 statement 的时候使用 row

## **MySQL** 存储引擎 **MyISAM** 与 **InnoDB** 区别

1. 锁粒度方面：由于锁粒度不同，InnoDB 比 MyISAM 支持更高的并发;InnoDB 的锁粒度为行锁、 MyISAM 的锁粒度为表锁、行锁需要对每一行进行加锁， 73 所以锁的开销更大，但是能解决脏读 和不可重复读的问题，相对来说也更 容易发生死锁

2. 可恢复性上：由于 InnoDB 是有事务日志的，所以在产生由于数据库崩溃等 条件后，可以根据日志 文件进行恢复。而 MyISAM 则没有事务日志。

3. 查询性能上:MylSAM 要优于 InnoDB 因为 InnoDB 在查询过程中，是需要维护 数据缓存，而且查

询过程是先定位到行所在的数据块，然后在从数据块中 定位到要查找的行;而 MyISAM 可以直接定 位到数据所在的内存地址，可以 直接找到数据。

4. 表结构文件上:MyISAM 的表结构文件包括:frm(表结构定义),.MYI(索 引),.MYD(数据);而 InnoDB 的表 数据文件为:ibd 和 frm(表结构定义)。

## **MyISAM** 索引与 **InnoDB** 索引的区别？

1. InnoDB 索引是聚簇索引，MyISAM 索引是非聚簇索引。

2. InnoDB 的主键索引的叶子节点存储着行数据，因此主键索引非常高效。

3. MyISAM 索引的叶子节点存储的是行数据地址，需要再寻址一次才能得到数 据。

4. InnoDB 非主键索引的叶子节点存储的是主键和其他带索引的列数据，因此 查询时做到覆盖索引会 非常高效。

## 什么是索引

索引是一种特殊的文件(InnoDB 数据表上的索引是表空间的一个组成部分)，它 们包含着对数据表里所 有记录的引用指针。 索引是一种数据结构。数据库索引，是数据库管理系统中一个排序的数据结 构，以 协助快速查询、更新数据库表中数据。索引的实现通常使用 B 树及其变 种 B+树。 更通俗的说，索引就 相当于目录。为了方便查找书中的内容，通过对内容建立 索引形成目录。索引是一个文件，它是要占据 物理空间的。

## 索引有哪些优缺点

索引的优点

1. 可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

2. 通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性 能。 索引的缺点

1. 时间方面：创建索引和维护索引要耗费时间，具体地，当对表中的数据进 行增加、删除和修改的时 候，索引也要动态的维护，会降低增/改/删的执 行效率；

2. 空间方面：索引需要占物理空间

## 索引有哪几种类型？

主键索引:

数据列不允许重复，不允许为 NULL，一个表只能有一个主键。 唯一索引:

数据列不允许重复，允许为 NULL 值，一个表允许多个列创建唯一索引。

1. 可以通过 ALTER TABLE table\_name ADD UNIQUE (column); 创建唯一索 引。
2. 可以通过 ALTER TABLE table\_name ADD UNIQUE (column1,column2); 创 建唯一组合索引。 普通索引:

基本的索引类型，没有唯一性的限制，允许为 NULL 值。

1. 可以通过 ALTER TABLE table\_name ADD INDEX index\_name (column);创建 普通索引
2. 可以通过 ALTER TABLE table\_name ADD INDEX index\_name(column1, column2, column3);创建 组合索引。

全文索引：

是目前搜索引擎使用的一种关键技术。

1. 可以通过 ALTER TABLE table\_name ADD FULLTEXT (column);创建全文索引。

## **MySQL** 中有哪几种锁？

1. 表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概 率最高，并发度最低。

2. 行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概 率最低，并发度也最高。

3. 页面锁：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界 于表锁和行锁之间，并发 度一般。

## **MySQL** 中 **InnoDB** 支持的四种事务隔离级别名称，以及逐

**级 之间的区别？**

1. SQL 标准定义的四个隔离级别为：

1. read uncommited：读到未提交数据
2. read committed：脏读，不可重复读
3. repeatable read：可重读
4. serializable：串行事物

## **char** 和 **varchar** 的区别？

1. char 和 varchar 类型在存储和检索方面有所不同

2. char 列长度固定为创建表时声明的长度，长度值范围是 1 到 255

3. 当 char 值被存储时，它们被用空格填充到特定长度，检索 char 值时需删 除尾随空格。

## 主键和候选键有什么区别？

表格的每一行都由主键唯一标识,一个表只有一个主键。 主键也是候选键。按照惯例，候选键可以被指定 为主键，并且可以用于任何外 键引用

## 如何在 **Unix** 和 **MySQL** 时间戳之间进行转换？

UNIX\_TIMESTAMP 是从 MySQL 时间戳转换为 Unix 时间戳的命令 FROM\_UNIXTIME 是从 Unix 时间戳 转换为 MySQL 时间戳的命令

## **MyISAM** 表类型将在哪里存储，并且还提供其存储格式？

每个 MyISAM 表格以三种格式存储在磁盘上：

1. “.frm”文件 存储表定义

2. 数据文件具有“.MYD”（MYData）扩展名

3. 索引文件具有“.MYI”（MYIndex）扩展名

## **MySQL** 里记录货币用什么字段类型好

NUMERIC 和 DECIMAL 类型被 MySQL 实现为同样的类型，这在 SQL92 标准允许。 他们被用于保存 值，该值的准确精度是极其重要的值，例如与金钱有关的数 据。当声明一个类是这些类型之一时，精度 和规模的能被(并且通常是)指定。 例如： salary DECIMAL(9,2) 在这个例子中，9(precision)代表将被用 于存储值的总的小数位数，而 2(scale)代表将被用于存储小数点后的位数。 因此，在这种情况下，能被 存储在 salary 列中的值的范围是从-9999999.99 到 9999999.99。

## 创建索引时需要注意什么？

1. 非空字段：应该指定列为 NOT NULL，除非你想存储 NULL。在 MySQL 中，含 有空值的列很难进 行查询优化，因为它们使得索引、索引的统计信息以及 比较运算更加复杂。应该用 0、一个特殊的 值或者一个空串代替空值；

2. 取值离散大的字段：（变量各个取值之间的差异程度）的列放到联合索引 的前面，可以通过

count()函数查看字段的差异值，返回值越大说明字段的 唯一值越多字段的离散程度高；

3. 索引字段越小越好：数据库的数据存储以页为单位一页存储的数据越多一 次 I/O 操作获取的数据越 大效率越高

## 使用索引查询一定能提高查询的性能吗？为什么

通常，通过索引查询数据比全表扫描要快。但是我们也必须注意到它的代价。 索引需要空间来存储，也 需要定期维护， 每当有记录在表中增减或索引列被修 改时，索引本身也会被修改。 这意味着每条记录的 INSERT，DELETE，UPDATE 将为此多付出 4，5 次的磁盘 I/O。 因为索引需要额外的存储空间和处理， 那 些不必要的索引反而会使查询反应时间变慢。使用索引查询不一定能提高查询 性能，索引范围查询 (INDEX RANGE SCAN)适用于两种情况:

1. 基于一个范围的检索，一般查询返回结果集小于表中记录数的 30%

2. 基于非唯一性索引的检索

## 百万级别或以上的数据如何删除

关于索引：由于索引需要额外的维护成本，因为索引文件是单独存在的文件,所 以当我们对数据的增加,修 改,删除,都会产生额外的对索引文件的操作,这些操 作需要消耗额外的 IO,会降低增/改/删的执行效率。所 以，在我们删除数据库 百万级别数据的时候，查询 MySQL 官方手册得知删除数据的速度和创建的索引 数量是成正比的。

1. 所以我们想要删除百万数据的时候可以先删除索引（此时大概耗时三分多 钟）

2. 然后删除其中无用数据（此过程需要不到两分钟）

3. 删除完成后重新创建索引(此时数据较少了)创建索引也非常快，约十分钟 左右。

4. 与之前的直接删除绝对是要快速很多，更别说万一删除中断,一切删除会回 滚。那更是坑了。

## 什么是最左前缀原则？什么是最左匹配原则

顾名思义，就是最左优先，在创建多列索引时，要根据业务需求，where 子句 中使用最频繁的一列放在 最左边。 最左前缀匹配原则，非常重要的原则，MySQL 会一直向右匹配直到遇到范围查 询(>、<、 between、like)就停止匹配，比如 a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d 是用不到索引的，如果建立 (a,b,d,c)的索引则都可以用到，a,b,d 的顺序可以任意调整。 =和 in 可以乱 序，比如 a = 1 and b = 2 and c = 3 建立(a,b,c)索引可以任 意顺序，MySQL 的查询优化器会帮你优化成 索引可以识别的形式。

## 什么是聚簇索引？何时使用聚簇索引与非聚簇索引

1. 聚簇索引：将数据存储与索引放到了一块，找到索引也就找到了数据

2. 非聚簇索引：将数据存储于索引分开结构，索引结构的叶子节点指向了数 据的对应行，myisam 通 过 key\_buffer 把索引先缓存到内存中，当需要访问 数据时（通过索引访问数据），在内存中直接 搜索索引，然后通过索引找 到磁盘相应数据，这也就是为什么索引不在 key buffer 命中时，速度慢 的 原因。

## **MySQL** 连接器

首先需要在 MySQL 客户端登陆才能使用，所以需要 个连接器 来连接用户和 MySQL 数据库，我们 一般 是使用 mysql-u 用户名-p 密码

来进行 MySQL 登陆，和服务端建立连接。在完成 TCP 握手后，连接器会根据你 输入的用户名和密码验 证你的登录身份。如果用户名或者密码错误，MySQL 就 会提示 Access denied for user，来结束执行。 如果登录成功后，MySQL 会根 据权限表中的记录来判定你的权限。

## **MySQL** 查询缓存

连接完成后，你就可以执行 SQL 语句了，这行逻辑就会来到第二步:查询缓存。 MySQL 在得到一个执行 请求后，会首先去查询缓存 中查找，是否执行过这条 SQL 语句，之前执行过的语句以及结果会以 key- value 对的形式，被直接放在 内存中。key 是查询语句，value 是查询的结果。 如果通过 key 能够查找 到这条 SQL 语句，就直接妾返回 SQL 的执行结果。如果语句不在查询缓存中，就会继续后面的执行阶 段。执行完成后，执行结果 就会被放入查询缓存中。 可以看到，如果查询命中缓存，MySQL 不需要执 行后面的复杂操作，就可以直 接返回结果，效率会很高。

## **MySQL** 分析器

如果没有命中查询，就开始执行真正的 SQL 语句。

1. 首先，MySQL 会根据你写的 SQL 语句进行解析，分析器会先做词法分析，你 写的 SQL 就是由多个 字符串和空格组成的一条 SQL 语句，MySQL 需要识别出 里面的字符串是什么，代表什么。

2. 然后进行语法分析，根据词法分析的结果，语法分析器会根据语法规则， 判断你输入的这个 SQL 语句是否满足 MySQL 语法。如果 SQL 语句不正确， 就会提示 You have an error in your SQL syntax

## **MySQL** 优化器

经过分析器的词法分析和语法分析后，你这条 SQL 就合法了，MySQL 就知道你 要做什么了。但是在执 行前，还需要进行优化器的处理，优化器会判断你使用 了哪种索引，使用了何种连接，优化器的作用就 是确定效率最高的执行方案

## **MySQL** 执行器

MySQL 通过分析器知道了你的 SQL 语句是否合法，你想要做什么操作，通过优 化器知道了该怎么做效 率最高，然后就进入了执行阶段，开始执行这条 SQL 语 句在执行阶段，MySQL 首先会判断你有没有执 行这条语句的权限，没有权限的 话，就会返回没有权限的错误。如果有权限，就打开表继续执行。打开 表的时候，执行器就会根据表的引擎定义，去使用这个引擎提供的接口。对于有索引的表，执行的逻辑 也差不多。

## 什么是临时表，何时删除临时表？

什么是临时表?MySQL 在执行 SQL 语句的过程中 通常会临时创建一些存储中间 结果集的表，临时 表只 对当前连接可见，在连接关闭时，临时表会被删除并 释放所有表空间。 临时表分为两种:一种是内存临时 表，一种是磁盘临时表，什么区别呢?内存临 时表使用的是 MEMORY 存储引擎，而临时表采用的是 MylSAM 存储引擎。 MySQL 会在下面这几种情况产生临时表。

1. 使用 UNION 查询:UNION 有两种，一种是 UNION，一种是 UNION ALL，它们 都用于联合查询;区 别是使用 UNION 会去掉两个表中的重复数据，相当于对 结果集做了一下 去重(distinct)。使用 UNIONALL，则不会排重，返回所有 的行。使用 UNION 查询会产生临时表。

2. 使用 TEMPTABLE 算法或者是 UNION 查询中的视图。TEMPTABLE 算法是一种创 建临时表的算 法，它是将结果放置到临时表中，意味这要 MySQL 要先创建 好一个临时表，然后将结果放到临时 表中去，然后再使用这个临时表进行 相应的查询。

3. ORDER BY 和 GROUPBY 的子句不一样时也会产生临时表。

4. DISTINCT 查询并且加上 ORDER BY 时;

5. SQL 中用到 SQL\_SMALL\_RESULT 选项时;如果查询结果比较小的时候，可以加 上 SQL SMALL RESULT 来优化，产生临时表

6. FROM 中的子查询;

7. EXPLAIN 查看执行计划结果的 Extra 列中，如果使用 Using Temporary 就 表示会用到临时表

## 谈谈 **SQL** 优化的经验

1. 查询语句无论是使用哪种判断条件等于、小于、大于，WHERE 左侧的条件 查询字段不要使用函数 或者表达式

2. 使用 EXPLAIN 命令优化你的 SELECT 查询，对于复杂、效率低的 SQL 语 句，我们通常是使用

explainsql 来分析这条 SQL 语句，这样方便我们分 析，进行优化。

3. 当你的 SELECT 查询语句只需要使用一条记录时，要使用 LIMIT 1。不要 直接使用 SELECT\*，而应 该使用具体需要查询的表字段，因为使用 EXPLAIN 进行分析时，SELECT"使用的是全表扫描，也就 是 type =all 。

4. 为每一张表设置一个 ID 属性。

5. 避免在 MHERE 字句中对字段进行 NULL

6. 判断避免在 WHERE 中使用!或>操作符

7. 使用 BETWEEN AND 替代 IN 8. 为搜索字段创建索引

9. 选择正确的存储引擎，InnoDB、MyISAM、MEMORY 等

10. 使用 LIKE%abc%不会走索引，而使用 LIKE abc%会走索引。

11. 对于枚举类型的字段(即有固定罗列值的字段)，建议使用 ENUM 而不是 VARCHAR，如性别、星 期、类型、类别等。

12. 拆分大的 DELETE 或 INSERT 语句

13. 选择合适的字段类型，选择标准是尽可能小、尽可能定长、尽可能使用整 数。

14. 字段设计尽可能使用 NOT NULL

15. 进行水平切割或者垂直分割

## 什么叫外链接？

外连接分为三种，分别是是左外连接(LEFT OUTER J0IN 或 LEFT JOIN 右外连 接(RIGHT OUTER JOIN 或 RIC GHT JOIN、全外连接(FULL OUTER JOIN 或 FULLJOIN)。 左外连接:又称为左连接，这种连接方式会 显示左表不符合条件的数据行，右边 不符合条件的数据行直接显示 NULL。 右外连接:也被称为右连接， 他与左连接相对，这种连接方式会显示右表不符合 条件的数据行，左表不符合条件的数据行直接显示

NULL

## 什么叫内链接？

结合两个表中相同的字段，返回关联字段相符的记录就是内链接

## 使用 **union** 和 **union all** 时需要注意些什么？

通过 union 连接的 SQL 分别单独取出的列数必须相同。 使用 union 时，多个相等的行将会被合并，由 于合升比较耗时，一般不直接使 用 union 进行合并，而是通常采用 union all 进行合并。

## **MyISAM** 存储引擎的特点

在 5.1 版本之前，MyISAM 是 MySQL 的默认存储引擎，MylSAM 并发性比较差，使 用的场景比较少主 要特点是:

1. 不支持事务操作，ACID 的特性也就不存在了，这一设计是为了性能和效率 考虑的，

2. 不支持外键操作，如果强行增加外键，MySQL 不会报错，只不过外键不起作 用。

3. MyISAM 默认的锁粒度是表级锁，所以并发性能比较差，加锁比较快，锁冲 突比较少，不太容易发 生死锁的情况。

4. MyISAM 会在磁盘上存储三个文件，文件名和表名相同，扩展名分别是 frm(存储表定义)、 MYD(MYData，存储数据)、MYI(MyIndex，存储索引)。 这里需要特别注意的是 MyISAM 只缓存 索引文件，并不缓存数据文件。

5. MyISAM 支持的索引类型有全局索引(Full-Text)、B-Tree 索引、R-Tree 索 引 □ Full-Text 索引:它的 出现是为了解决针对文本的模糊查询效率较低的 问题。 □ B-Tree 索引:所有的索引节点都按照平衡 树的数据结构来存储，所有的 索引数据节点都在叶节点 □ R-Tree 索引:它的存储方式和 B-Tree 索引 有一些区别，主要设计用于 存储空间和多维数据的字段做索引目前的 MySQL 版本仅支持 geometry 类型的字段作索引，相对于 BTREE,RTREE 的优势在于范围查找。

6. 数据库所在主机如果宕机，MyISAM 的数据文件容易损坏，而且难以恢复。

7. 增删改查性能方面:SELECT 性能较高，适用于查询较多的情况

## **InnoDB** 存储引擎的特点

自从 MySQL5.1 之后，默认的存储引擎变成了 InnoDB 存储引擎，相对于 MylSAM，InnoDB 存储引擎 有了较大的改变，它的主要特点是

1. 支持事务操作，具有事务 ACID 隔离特性，默认的隔离级别是可重复读 (repetable-read)、通过 MVCC(并发版本控制)来实现的。能够解决 脏读 和 不可重复读 的问题。 InnoDB 支持外键操作。

2. InnoDB 默认的锁粒度行级锁，并发性能比较好，会发生死锁的情况。

3. 和 MyISAM 一样的是，InnoDB 存储引擎也有 frm 文件存储表结构定义，但是 不同的是，InnoDB 的表数据与索引数据是存储在一起的，都位于 B+数的叶 子节点上，而 MylSAM 的表数据和索引数 据是分开的。

4. InnoDB 有安全的日志文件，这个日志文件用于恢复因数据库崩溃或其他情 况导致的数据丢失问 题，保证数据的一致性。

5. InnoDB 和 MylSAM 支持的索引类型相同，但具体实现因为文件结构的不同有 很大差异。

6. 增删改查性能方面，果执行大量的增删改操作，推荐使用 InnoDB 存储引 擎，它在删除操作时是对 行删除，不会重建表。

# **linux**

## 什么是 **Linux**

Linux 是⼀套免费使⽤和⾃由传播的类 Unix 操作系统，是⼀个基于 POSIX和 Unix 的多⽤户、多任务、

⽀持多线程 和多 CPU的操作系统。它能运⾏主要的 Unix ⼯具软件、应⽤程序和⽹络协议。它⽀持32 位 和64 位硬件。Linux 继承了 Unix 以⽹络为核⼼的设计思想，是⼀个性能稳定的多⽤户⽹络操作系统

## **Unix** 和 **Linux** 有什么区别

Linux 和 Unix 都是功能强⼤的操作系统，都是应⽤⼴泛的服务器操作系统，有很多相似之处，甚⾄有⼀ 部分⼈错 误地认为 Unix 和 Linux 操作系统是⼀样的，然⽽，事实并⾮如此，以下是两者的区别。

1. 开源性：Linux 是⼀款开源操作系统，不需要付费，即可使⽤；Unix 是⼀款对源码实⾏知识产权保 护的传统 商业软件，使⽤需要付费授权使⽤。

2. 跨平台性：Linux 操作系统具有良好的跨平台性能，可运⾏在多种硬件平台上；Unix 操作系统跨平 台性能较 弱，⼤多需与硬件配套使⽤。

3. 可视化界⾯：Linux 除了进⾏命令⾏操作，还有窗体管理系统；Unix 只是命令⾏下的系统。

4. 硬件环境：Linux 操作系统对硬件的要求较低，安装⽅法更易掌握；Unix 对硬件要求⽐较苛刻，按 照难度较 ⼤。

5. ⽤户群体：Linux 的⽤户群体很⼴泛，个⼈和企业均可使⽤；Unix 的⽤户群体⽐较窄，多是安全性 要求⾼的 ⼤型企业使⽤，如银⾏、电信部⻔等，或者 Unix 硬件⼚商使⽤，如 Sun等。相⽐于 Unix 操作系统，Linux 操作系统更受⼴⼤计算机爱好者的喜爱，主要原因是 Linux 操作系统具有 Unix 操 作系统的全部功能，并且能 够在普通 PC计算机上实现全部的 Unix 特性，开源免费的特性，更容易 普及使⽤

## 什么是 **Linux** 内核？

Linux 系统的核⼼是内核。内核控制着计算机系统上的所有硬件和软件，在必要时分配硬件，并根据需要 执⾏软 件。 系统内存管理 应⽤程序管理 硬件设备管理 ⽂件系统管理

## **Linux** 的基本组件是什么？

就像任何其他典型的操作系统⼀样，Linux 拥有所有这些组件：内核，shell 和 GUI，系统实⽤程序和应

⽤程序。 Linux ⽐其他操作系统更具优势的是每个⽅⾯都附带其他功能，所有代码都可以免费下载

## **Linux** 的体系结构

从⼤的⽅⾯讲，Linux 体系结构可以分为两块： ⽤户空间(User Space)：⽤户空间⼜包括⽤户的应⽤程序 (User Applications)、C 库(C Library)。 内核空间(Kernel Space)：内核空间⼜包括系统调⽤接⼝ (System Call Interface)、内核(Kernel)、平台架构相 关的代码(Architecture - Dependent Kernel Code)。 为什么 Linux 体系结构要分为⽤户空间和内核空间的原因？

1. 现代 CPU 实现了不同的⼯作模式，不同模式下 CPU 可以执⾏的指令和访问的寄存器不同。

2. Linux 从 CPU 的⻆度出发，为了保护内核的安全，把系统分成了两部分。⽤户空间和内核空间是程 序执⾏的 两种不同的状态，我们可以通过两种⽅式完成⽤户空间到内核空间的转移：1）系统调

⽤；2）硬件中断。

## **BASH** 和 **DOS** 之间的基本区别是什么？

BASH和 DOS控制台之间的主要区别在于3 个⽅⾯：

1. BASH命令区分⼤⼩写，⽽ DOS命令则不区分;

2. 在 BASH下，/ character 是⽬录分隔符，\作为转义字符。在 DOS下，/⽤作命令参数分隔符，\是

⽬录分隔符

3. DOS遵循命名⽂件中的约定，即8 个字符的⽂件名后跟⼀个点，扩展名为3 个字符。BASH没有遵循 这样的惯 例

## **Linux** 开机启动过程？

1. 主机加电自检，加载 BIOS 硬件信息

1. 读取 MBR 的引导文件(GRUB、LILO)
2. 引导 Linux 内核

4. 运行第一个进程 init (进程号永远为 1 )

5. 进入相应的运行级别

6. 运行终端，输入用户名和密码

## **Linux** 系统缺省的运行级别？

关机 单机用户模式

字符界面的多用户模式(不支持网络) 字符界面的多用户模式

未分配使用 图形界面的多用户模式 重启

## **Linux** 使用的进程间通信方式？

管道(pipe)、流管道(s\_pipe)、有名管道(FIFO) 信号(signal)

消息队列

共享内存 信号量 套接字(socket)

## **Linux** 有哪些系统日志文件？

比较重要的是 /var/log/messages 日志文件 该日志文件是许多进程日志文件的汇总，从该文件可以看出任何入侵企图或成 功的入侵。 另外，如果胖 友的系统里有 ELK 日志集中收集，它也会被收集进 去。

## 什么是交换空间

交换空间是 Linux 使用的一定空间，用于临时保存一些并发运行的程序。当 RAM 没有足够的内存来容 纳正在执行的所有程序时，就会发生这种情况

## 什么是 **Root** 帐户

root 帐户就像一个系统管理员帐户，允许你完全控制系统。你可以在此处创建 和维护用户帐户，为每个 帐户分配不同的权限。每次安装 Linux 时都是默认帐 户

## 什么是 **LILO**？

LILO 是 Linux 的引导加载程序。它主要用于将 Linux 操作系统加载到主内存 中，以便它可以开始运行。

## 什么是 **BASH**？

BASH 是 Bourne Again SHell 的缩写。它由 Steve Bourne 编写，作为原始 Bourne Shell（由/ bin / sh 表示）的替代品。它结合了原始版本的 Bourne Shell 的所有功能，以及其他功能，使其更容易使用。从 那以后，它已被改编 为运行 Linux 的大多数系统的默认 shell。

## 什么是 **CLI**？

命令行界面（英语：command-line interface，缩写]：CLI）是在图形用户界 面得到普及之前使用最为 广泛的用户界面，它通常不支持鼠标，用户通过键盘 输入指令，计算机接收到指令后，予以执行。也有 人称之为字符用户界面 （CUI）。 通常认为，命令行界面（CLI）没有图形用户界面（GUI）那么方便用 户操作。 因为，命令行界面的软件通常需要用户记忆操作的命令，但是，由于其本身的 特点，命令行界 面要较图形用户界面节约计算机系统的资源。在熟记命令的前 提下，使用命令行界面往往要较使用图形 用户界面的操作速度要快。所以，图 形用户界面的操作系统中，都保留着可选的命令行界面

## 什么是 **GUI**？

图形用户界面（Graphical User Interface，简称 GUI，又称图形用户接口） 是指采用图形方式显示的计 算机操作用户界面。 91 图形用户界面是一种人与计算机通信的界面显示格式，允许用户使用鼠标等输 入设备操纵屏幕上的图标或菜单选项，以选择命令、调用文件、启动程序或执 行其它一些日常任务。与 通过键盘输入文本或字符命令来完成例行任务的字符 界面相比，图形用户界面有许多优点

## 开源的优势是什么？

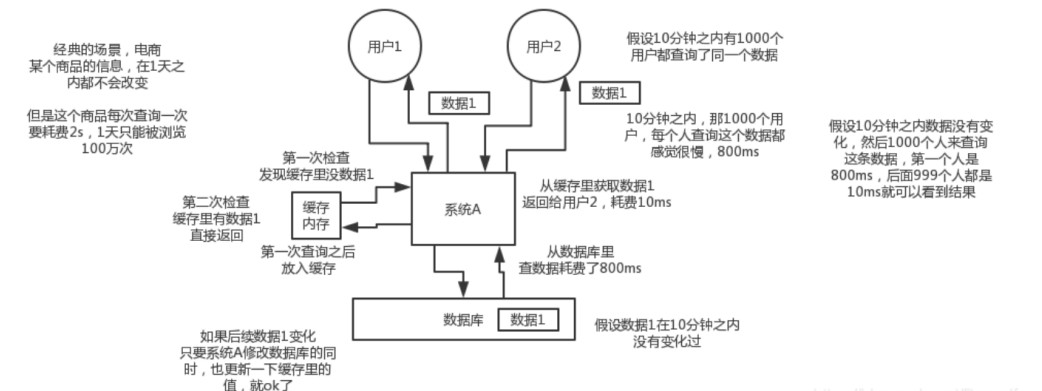
开源允许你将软件（包括源代码）免费分发给任何感兴趣的人。然后，人们可 以添加功能，甚至可以调 试和更正源代码中的错误。它们甚至可以让它运行得 更好，然后再次自由地重新分配这些增强的源代 码。这最终使社区中的每个人 受益

## **GNU** 项目的重要性是什么

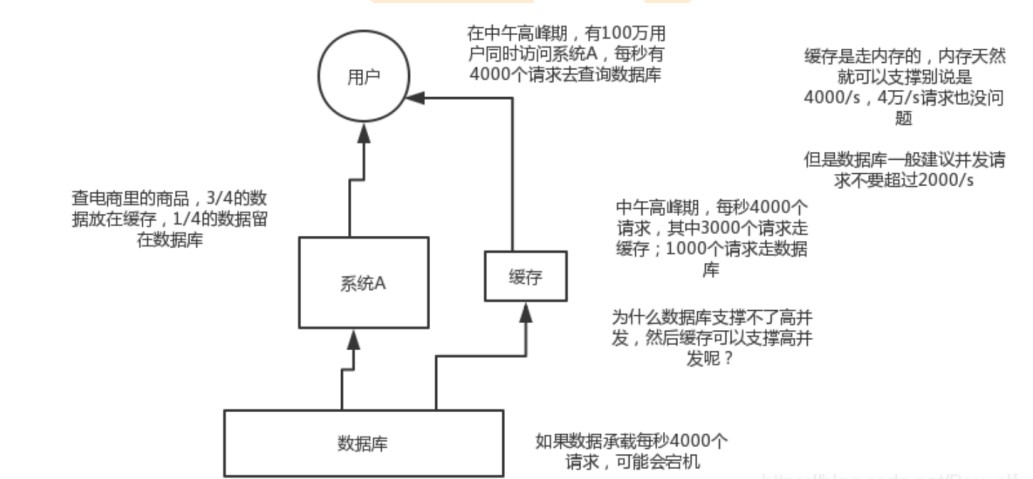
这种所谓的自由软件运动具有多种优势，例如可以自由地运行程序以及根据你 的需要自由学习和修改程 序。它还允许你将软件副本重新分发给其他人，以及 自由改进软件并将其发布给公众

**缓存**

## 缓存如何实现高性能？



**缓存如何实现高并发**



**Redis 和 Memcached 的区别**

Redis 拥有更多的数据结构和丰富的数据操作 Redis 内存利用率高于 Memcached Redis 是单线程， Memcached 是多线程，在存储大数据的情况下，Redis 比 Memcached 稍有逊色 Memcached 没有原 生的集群模式，Redis 官方支持 Redis Cluster 集群模式

## 用缓存可能出现的问题

1. 数据不一致

2. 缓存雪崩

3. 缓存穿透

4. 缓存并发竞争

## 当查询缓存报错，怎么提高可用性？

缓存可以极大的提高查询性能，但是缓存数据丢失和缓存不可用不能影响应用 的正常工作。因此，一般 情况下，如果缓存出现异常，需要手动捕获这个异 常，并且记录日志，并且从数据库查询数据返回给用 户，而不应该导致业务不 可用

## 如果避免缓存**”**穿透**”**的问题？

缓存穿透，是指查询一个一定不存在的数据，由于缓存是不命中时被动写，并 且处于容错考虑，如果从 DB 查不到数据则不写入缓存，这将导致这个不存在 的数据每次请求都要到 DB 去查询，失去了缓存的意 义。 如何解决

有两种方案可以解决 ：

1. 方案一，缓存空对象。 当从 DB 查询数据为空，我们仍然将这个空结果进行缓存，具体的值需要使 用 特殊的标识，能和真正缓存的数据区分开。另外，需要设置较短的过期时间， 一般建议不要超过 5 分钟。

2. 方案二，BloomFilter 布隆过滤器。 在缓存服务的基础上，构建 BloomFilter 数据结构，在

BloomFilter 中存储 对应的 KEY 是否存在，如果存在，说明该 KEY 对应的值不为空

## 如何避免缓存**“**雪崩**”**的问题

缓存雪崩，是指缓存由于某些原因无法提供服务( 例如，缓存挂掉了 )，所有 请求全部达到 DB 中，导致

DB 负荷大增，最终挂掉的情况。

预防和解决缓存雪崩的问题，可以从以下多个方面进行共同着手。 1）缓存高可用：通过搭建缓存的高可 用，避免缓存挂掉导致无法提供服务的 情况，从而降低出现缓存雪崩的情况。假设我们使用 Redis 作为 缓存，则可以 使用 Redis Sentinel 或 Redis Cluster 实现高可用。 2）本地缓存：如果使用本地缓存时， 即使分布式缓存挂了，也可以将 DB 查 询到的结果缓存到本地，避免后续请求全部到达 DB 中。如果我 们使用 JVM ， 则可以使用 Ehcache、Guava Cache 实现本地缓存的功能。

## 如何 避免缓存**“**击穿**”**的问题

缓存击穿，是指某个极度“热点”数据在某个时间点过期时，恰好在这个时间 点对这个 KEY 有大量的并发 请求过来，这些请求发现缓存过期一般都会从 DB 加载数据并回设到缓存，但是这个时候大并发的请求 可能会瞬间 DB 压垮。 对于一些设置了过期时间的 KEY ，如果这些 KEY 可能会在某些时间点被超高 并 发地访问，是一种非常“热点”的数据。这个时候，需要考虑这个问题。 区别：

1. 和缓存“雪崩“”的区别在于，前者针对某一 KEY 缓存，后者则是很多 KEY 。

2. 和缓存“穿透“”的区别在于，这个 KEY 是真实存在对应的值的 有两种方案可以解决：

1. 方案一，使用互斥锁。请求发现缓存不存在后，去查询 DB 前，使用分布 式锁，保证有且只有一个 线程去查询 DB ，并更新到缓存。

2. 方案二，手动过期。缓存上从不设置过期时间，功能上将过期时间存在 KEY 对应的 VALUE 里。流 程如下：

1. 获取缓存。通过 VALUE 的过期时间，判断是否过期。如果未过期，则 直接返回；如果已过 期，继续往下执行。

2. 通过一个后台的异步线程进行缓存的构建，也就是“手动”过期。通过 后台的异步线程，保证有 且只有一个线程去查询 DB。

3. 同时，虽然 VALUE 已经过期，还是直接返回。通过这样的方式，保证 服务的可用性，虽然损 失了一定的时效性。

## 什么是缓存预热？如何实现缓存预热

缓存预热

在刚启动的缓存系统中，如果缓存中没有任何数据，如果依靠用户请求的方式 重建缓存数据，那么对数 据库的压力非常大，而且系统的性能开销也是巨大 的。 此时，最好的策略是启动时就把热点数据加载 好。这样，用户请求时，直接读 取的就是缓存的数据，而无需去读取 DB 重建缓存数据。举个例子，热 门的或 者推荐的商品，需要提前预热到缓存中

如何实现

一般来说，有如下几种方式来实现： 数据量不大时，项目启动时，自动进行初始化。 写个修复数据脚 本，手动执行该脚本。 写个管理界面，可以手动点击，预热对应的数据到缓存中。

## 缓存数据的淘汰策略有哪些？

除了缓存服务器自带的缓存自动失效策略之外，我们还可以根据具体的业务需 求进行自定义的“手动”缓 存淘汰，常见的策略有两种： 1、定时去清理过期的缓存。 2、当有用户请求过来时，再判断这个请求所 用到的缓存是否过期，过期的话 就去底层系统得到新数据并更新缓存。 两者各有优劣，第一种的缺点是 维护大量缓存的 key 是比较麻烦的，第二种 的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对 比较复杂！ 具体用哪种方案，大家可以根据自己的应用场景来权衡。

# 网络和操作系统

## 进程和线程的区别？

1. 调度：进程是资源管理的基本单位，线程是程序执行的基本单位。

2. 切换：线程上下文切换比进程上下文切换要快得多。

3. 拥有资源： 进程是拥有资源的一个独立单位，线程不拥有系统资源，但是 可以访问隶属于进程的资 源。

4. 系统开销： 创建或撤销进程时，系统都要为之分配或回收系统资源，如内 存空间，I/O 设备等，OS 所付出的开销显著大于在创建或撤销线程时的开 销，进程切换的开销也远大于线程切换的开销。

## 协程与线程的区别？

1. 线程和进程都是同步机制，而协程是异步机制。

2. 线程是抢占式，而协程是非抢占式的。需要用户释放使用权切换到其他协 程，因此同一时间其实只 有一个协程拥有运行权，相当于单线程的能力。

3. 一个线程可以有多个协程，一个进程也可以有多个协程。

4. 协程不被操作系统内核管理，而完全是由程序控制。线程是被分割的 CPU 资源，协程是组织好的代 码流程，线程是协程的资源。但协程不会直接使 用线程，协程直接利用的是执行器关联任意线程或 线程池。

5. 协程能保留上一次调用时的状态。

## 并发和并行有什么区别？

并发就是在一段时间内，多个任务都会被处理；但在某一时刻，只有一个任务 在执行。单核处理器可以 做到并发。比如有两个进程 A 和 B，A 运行一个时间 98 片之后，切换到 B，B 运行一个时间片之后又切 换到 A。因为切换速度足够 快，所以宏观上表现为在一段时间内能同时运行多个程序。 并行就是在同一 时刻，有多个任务在执行。这个需要多核处理器才能完成，在 微观上就能同时执行多条指令，不同的程 序被放到不同的处理器上运行，这个 是物理上的多个进程同时进行。

## 进程与线程的切换流程

进程切换分两步：

1. 切换页表以使用新的地址空间，一旦去切换上下文，处理器中所有已经缓存 的内存地址一瞬间都作 废了。

2. 切换内核栈和硬件上下文。 对于 linux 来说，线程和进程的最大区别就在于地址空间，对于线程切 换，第 1 步是不需要做的，第 2 步是进程和线程切换都要做的。 因为每个进程都有自己的虚拟地址 空间，而线程是共享所在进程的虚拟地址空 间的，因此同一个进程中的线程进行线程切换时不涉及 虚拟地址空间的转换

## 为什么虚拟地址空间切换会比较耗时？

进程都有自己的虚拟地址空间，把虚拟地址转换为物理地址需要查找页表，页 表查找是一个很慢的过 程，因此通常使用 Cache 来缓存常用的地址映射，这样 可以加速页表查找，这个 Cache 就是 TLB

（translation Lookaside Buffer， TLB 本质上就是一个 Cache，是用来加速页表查找的）。 由于每个进 程都有自己的虚拟地址空间，那么显然每个进程都有自己的页表， 那么当进程切换后页表也要进行切 换，页表切换后 TLB 就失效了，Cache 失效 导致命中率降低，那么虚拟地址转换为物理地址就会变慢， 表现出来的就是程 序运行会变慢，而线程切换则不会导致 TLB 失效，因为线程无需切换地址空 间，因此 我们通常说线程切换要比较进程切换块，原因就在这里

## 进程间通信方式有哪些？

1. 管道：管道这种通讯方式有两种限制，一是半双工的通信，数据只能单向流 动，二是只能在具有亲 缘关系的进程间使用。进程的亲缘关系通常是指父子进 程关系。 管道可以分为两类：匿名管道和命 名管道。匿名管道是单向的，只能在有亲缘 关系的进程间通信；命名管道以磁盘文件的方式存在， 可以实现本机任意两个 进程通信。

2. 信号：信号是一种比较复杂的通信方式，信号可以在任何时候发给某一进程， 而无需知道该进程的 状态 Linux 系统中常用信号：

1. SIGHUP：用户从终端注销，所有已启动进程都将收到该进程。系统缺省状 态下对该信号的处 理是终止进程。

2. SIGINT：程序终止信号。程序运行过程中，按 Ctrl+C 键将产生该信号。

3. SIGQUIT：程序退出信号。程序运行过程中，按 Ctrl+\键将产生该信 号。

1. SIGBUS 和 SIGSEGV：进程访问非法地址。

5. SIGFPE：运算中出现致命错误，如除零操作、数据溢出等。

6. SIGKILL：用户终止进程执行信号。shell 下执行 kill -9 发送该信号。

1. SIGTERM：结束进程信号。shell 下执行 kill 进程 pid 发送该信号。
2. SIGALRM：定时器信号。

9. SIGCLD：子进程退出信号。如果其父进程没有忽略该信号也没有处理该信 号，则子进程退出 后将形成僵尸进程

3. 信号量：信号量是一个计数器，可以用来控制多个进程对共享资源的访 问。它常作为一种锁机制， 防止某进程正在访问共享资源时，其他进程也 访问该资源。因此，主要作为进程间以及同一进程内 不同线程之间的同步 手段

4. 消息队列：消息队列是消息的链接表，包括 Posix 消息队列和 System V 消 息队列。有足够权限的 进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号 承载信息量少，管道只能 承载无格式字节流以及缓冲区大小受限等缺点。

5. 共享内存：共享内存就是映射一段能被其他进程所访问的内存，这段共享 内存由一个进程创建，但

多个进程都可以访问。共享内存是最快的 IPC 方 式，它是针对其他进程间通信方式运行效率低而专 门设计的。它往往与其 他通信机制，如信号量，配合使用，来实现进程间的同步和通信

6. Socket：与其他通信机制不同的是，它可用于不同机器间的进程通信 优缺点：

1. 管道：速度慢，容量有限；

2. Socket：任何进程间都能通讯，但速度慢；

3. 消息队列：容量受到系统限制，且要注意第一次读的时候，要考虑上一次 没有读完数据的问题；

4. 信号量：不能传递复杂消息，只能用来同步；

5. 共享内存区：能够很容易控制容量，速度快，但要保持同步，比如一个进 程在写的时候，另一个进 程要注意读写的问题，相当于线程中的线程安 全，当然，共享内存区同样可以用作线程间通讯，不 过没这个必要，线程 间本来就已经共享了同一进程内的一块内存。

## 进程间同步的方式有哪些

临界区： 通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访 问。 优点：保证在某一时刻只有一个线程能访问数据的简便办法。

缺点：虽然临界区同步速度很快，但却只能用来同步本进程内的线程，而不可 用来同步多个进程中的线 程

互斥量：

为协调共同对一个共享资源的单独访问而设计的。互斥量跟临界区很相似，比 临界区复杂，互斥对象只 有一个，只有拥有互斥对象的线程才具有访问资源的 权限。 优点：使用互斥不仅仅能够在同一应用程序不同线程中实现资源的安全共享， 而且可以在不同应用程序 的线程之间实现对资源的安全共享。

缺点：

1. 互斥量是可以命名的，也就是说它可以跨越进程使用，所以创建互斥量需 要的资源更多，所以如果 只为了在进程内部是用的话使用临界区会带来速 度上的优势并能够减少资源占用量

2. 通过互斥量可以指定资源被独占的方式使用，但如果有下面一种情况通过 互斥量就无法处理，比如 现在一位用户购买了一份三个并发访问许可的数 据库系统，可以根据用户购买的访问许可数量来决 定有多少个线程/进程能 同时进行数据库操作，这时候如果利用互斥量就没有办法完成这个要求， 信号量对象可以说是一种资源计数器

信号量：

为控制一个具有有限数量用户资源而设计。它允许多个线程在同一时刻访问同 一资源，但是需要限制在 同一时刻访问此资源的最大线程数目。互斥量是信号 量的一种特殊情况，当信号量的最大资源数=1 就是 互斥量了。 优点：适用于对 Socket（套接字）程序中线程的同步

缺点:

1. 信号量机制必须有公共内存，不能用于分布式操作系统，这是它最大的弱 点；

2. 信号量机制功能强大，但使用时对信号量的操作分散， 而且难以控制，读 写和维护都很困难，加重 了程序员的编码负担；

3. 核心操作 P-V 分散在各用户程序的代码中，不易控制和管理，一旦错误， 后果严重，且不易发现和 纠正。

事件：

用来通知线程有一些事件已发生，从而启动后继任务的开始。 102 优点：事件对象通过通知操作的方式 来保持线程的同步，并且可以实现不同进 程中的线程同步操作。

## 线程同步的方式有哪些

1. 临界区：当多个线程访问一个独占性共享资源时，可以使用临界区对象。拥有 临界区的线程可以访 问被保护起来的资源或代码段，其他线程若想访问，则被 挂起，直到拥有临界区的线程放弃临界区 为止，以此达到用原子方式操 作共享 资源的目的。

2. 事件：事件机制，则允许一个线程在处理完一个任务后，主动唤醒另外一个线 程执行任务。 互斥 量：互斥对象和临界区对象非常相似，只是其允许在进程间使用，而临界 区只限制与同一进程的各 个线程之间使用，但是更节省资源，更有效率。

3. 信号量：当需要一个计数器来限制可以使用某共享资源的线程数目时，可以使 用“信号量”对象 区 别：

1. 互斥量与临界区的作用非常相似，但互斥量是可以命名的，也就是说互斥 量可以跨越进程使 用，但创建互斥量需要的资源更多，所以如果只为了在 进程内部是用的话使用临界区会带来 速度上的优势并能够减少资源占用 量 。因为互斥量是跨进程的互斥量一旦被创建，就可以通 过名字打开它。

2. 互斥量，信号量，事件都可以被跨越进程使用来进行同步数据操作。

## 线程的分类

从线程的运行空间来说，分为用户级线程（user-level thread, ULT）和内核 级线程（kernel-level,

KLT）

内核级线程：这类线程依赖于内核，又称为内核支持的线程或轻量级进程。无 论是在用户程序中的线程 还是系统进程中的线程，它们的创建、撤销和切换都 由内核实现。比如英特尔 i5-8250U 是 4 核 8 线 程，这里的线程就是内核级线程 用户级线程：它仅存在于用户级中，这种线程是不依赖于操作系统核心的。应 用进程利用线程库来完成 其创建和管理，速度比较快，操作系统内核无法感知 用户级线程的存在。

## 什么是临界区，如何解决冲突？

每个进程中访问临界资源的那段程序称为临界区，一次仅允许一个进程使用的 资源称为临界资源。 解决 冲突的办法 ：

1. 如果有若干进程要求进入空闲的临界区，一次仅允许一个进程进入，如已 有进程进入自己的临界 区，则其它所有试图进入临界区的进程必须等待；

2. 进入临界区的进程要在有限时间内退出。

3. 如果进程不能进入自己的临界区，则应让出 CPU，避免进程出现“忙等”现 象。

## 什么是死锁？死锁产生的条件？

什么是死锁：

在两个或者多个并发进程中，如果每个进程持有某种资源而又等待其它进程释 放它或它们现在保持着的 资源，在未改变这种状态之前都不能向前推进，称这 一组进程产生了死锁。通俗的讲就是两个或多个进 程无限期的阻塞、相互等待 的一种状态。

死锁产生的四个必要条件

1. 互斥条件：一个资源一次只能被一个进程使用

2. 请求与保持条件：一个进程因请求资源而阻塞时，对已获得资源保持不放

3. 不剥夺条件：进程获得的资源，在未完全使用完之前，不能强行剥夺

4. 循环等待条件：若干进程之间形成一种头尾相接的环形等待资源关系 如何处理死锁问题

1. 忽略该问题。例如鸵鸟算法，该算法可以应用在极少发生死锁的的情况 下。为什么叫鸵鸟算法呢，

因为传说中鸵鸟看到危险就把头埋在地底下， 可能鸵鸟觉得看不到危险也就没危险了吧。跟掩耳盗 铃有点像。

2. 检测死锁并且恢复。

3. 仔细地对资源进行动态分配，以避免死锁。

4. 通过破除死锁四个必要条件之一，来防止死锁产生。

## 进程调度策略有哪几种

1. 先来先服务：非抢占式的调度算法，按照请求的顺序进行调度。有利于长 作业，但不利于短作业， 因为短作业必须一直等待前面的长作业执行完毕 才能执行，而长作业又需要执行很长时间，造成了 短作业等待时间过长。 另外，对 I/O 密集型进程也不利，因为这种进程每次进行 I/O 操作之后又 得 重新排队

2. 短作业优先：非抢占式的调度算法，按估计运行时间最短的顺序进行调 度。长作业有可能会饿死， 处于一直等待短作业执行完毕的状态。因为如 果一直有短作业到来，那么长作业永远得不到调度。

3. 最短剩余时间优先：最短作业优先的抢占式版本，按剩余运行时间的顺序 进行调度。 当一个新的作 业到达时，其整个运行时间与当前进程的剩余时 间作比较。如果新的进程需要的时间更少，则挂起 当前进程，运行新的进 程。否则新的进程等待

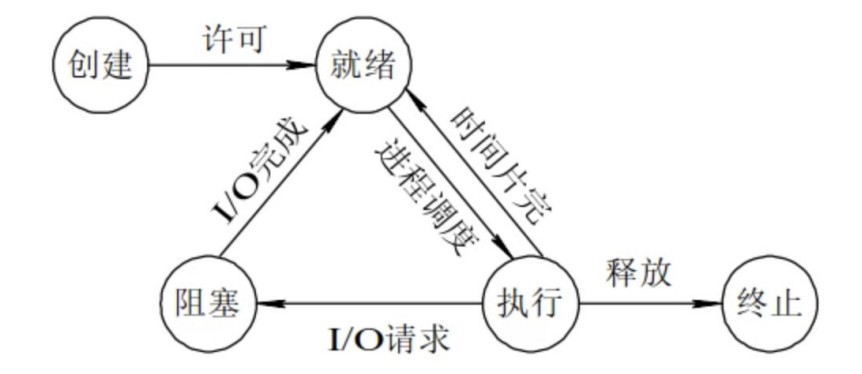
4. 时间片轮转：将所有就绪进程按 FCFS 的原则排成一个队列，每次调度时，把 CPU 时间分配给队首 进程，该进程可以执行一个时间片。当时间片用完时，由 计时器发出时钟中断，调度程序便停止该 进程的执行，并将它送往就绪队列的 末尾，同时继续把 CPU 时间分配给队首的进程。

1. 时间片轮转算法的效率和时间片的大小有很大关系：因为进程切换都要保 存进程的信息并且 载入新进程的信息，如果时间片太小，会导致进程切换 得太频繁，在进程切换上就会花过多 时间。 而如果时间片过长，那么实时 性就不能得到保证

5. 优先级调度：为每个进程分配一个优先级，按优先级进行调度。为了防止 低优先级的进程永远等不 到调度，可以随着时间的推移增加等待进程的优 先级

## 进程有哪些状态

进程一共有 5 种状态，分别是创建、就绪、运行（执行）、终止、阻塞。



1. 运行状态就是进程正在 CPU 上运行。在单处理机环境下，每一时刻最多只 有一个进程处于运行状 态。

2. 就绪状态就是说进程已处于准备运行的状态，即进程获得了除 CPU 之外的 一切所需资源，一旦得 到 CPU 即可运行。

3. 阻塞状态就是进程正在等待某一事件而暂停运行，比如等待某资源为可用 或等待 I/O 完成。即使

CPU 空闲，该进程也不能运行。

4. 运行态→阻塞态：往往是由于等待外设，等待主存等资源分配或等待人工干预 而引起的。

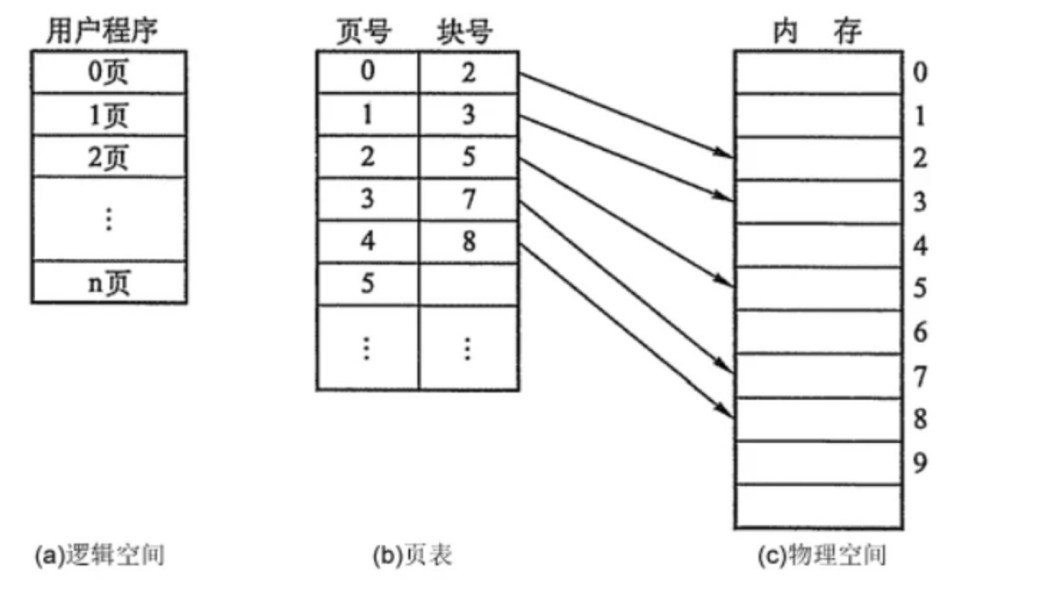
5. 阻塞态→就绪态：则是等待的条件已满足，只需分配到处理器后就能运行。

6. 运行态→就绪态：不是由于自身原因，而是由外界原因使运行状态的进程让出 处理器，这时候就变 成就绪态。例如时间片用完，或有更高优先级的进程来抢 占处理器等。

7. 就绪态→运行态：系统按某种策略选中就绪队列中的一个进程占用处理器，此 时就变成了运行态

## 什么是分页？

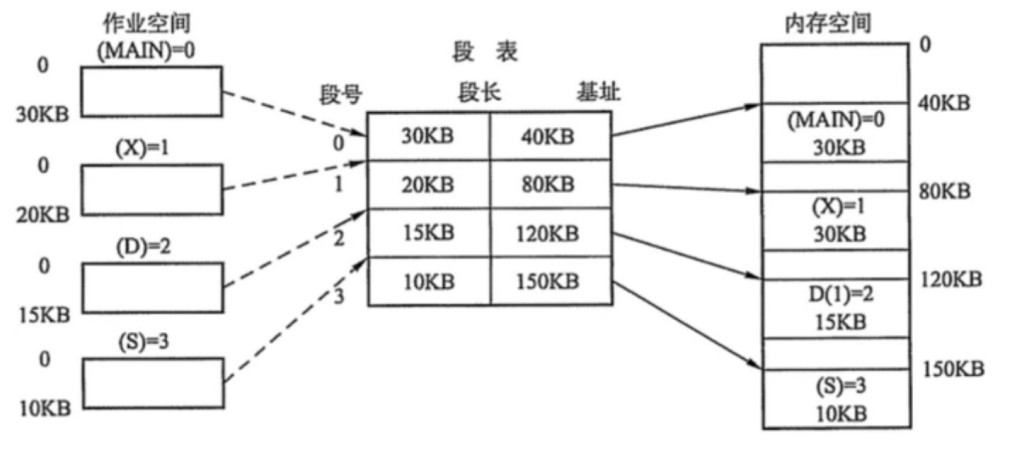
把内存空间划分为大小相等且固定的块，作为主存的基本单位。因为程序数据 存储在不同的页面中，而 页面又离散的分布在内存中，因此需要一个页表来记 录映射关系，以实现从页号到物理块号的映射。 访 问分页系统中内存数据需要两次的内存访问 (一次是从内存中访问页表，从 中找到指定的物理块号，加上 页内偏移得到实际物理地址；第二次就是根据第 一次得到的物理地址访问内存取出数据)。



## 什么是分段

分页是为了提高内存利用率，而分段是为了满足程序员在编写代码的时候的一 些逻辑需求(比如数据共 享，数据保护，动态链接等)。 分段内存管理当中，地址是二维的，一维是段号，二维是段内地址；其中 每个 段的长度是不一样的，而且每个段内部都是从 0 开始编址的。由于分段管理 中，每个段内部是连续 内存分配，但是段和段之间是离散分配的，因此也存在 一个逻辑地址到物理地址的映射关系，相应的就

是段表机制。



## 分页和分段有什区别

1. 分页对程序员是透明的，但是分段需要程序员显式划分每个段。

2. 分页的地址空间是一维地址空间，分段是二维的。

3. 页的大小不可变，段的大小可以动态改变。

4. 分页主要用于实现虚拟内存，从而获得更大的地址空间；分段主要是为了使程序和数据可以被划分 为逻辑上独立的地址空间并且有助于共享和保 护。

## 什么是交换空间

操作系统把物理内存(physical RAM)分成一块一块的小内存，每一块内存被称 为页(page)。当内存资源 不足时，Linux 把某些页的内容转移至硬盘上的一块 空间上，以释放内存空间。硬盘上的那块空间叫做 交换空间(swap space),而这 一过程被称为交换(swapping)。物理内存和交换空间的总容量就是虚拟内存 的 可用容量。

用途：

1. 物理内存不足时一些不常用的页可以被交换出去，腾给系统。

2. 程序启动时很多内存页被用来初始化，之后便不再需要，可以交换出去

## 页面替换算法有哪些

在程序运行过程中，如果要访问的页面不在内存中，就发生缺页中断从而将该 页调入内存中。此时如果 内存已无空闲空间，系统必须从内存中调出一个页面 到磁盘对换区中来腾出空间。

包括以下算法：

1. 最佳算法：所选择的被换出的页面将是最长时间内不再被访问，通常可以 保证获得最低的缺页率。 这是一种理论上的算法，因为无法知道一个页面 多长时间不再被访问。

2. 先进先出：选择换出的页面是最先进入的页面。该算法将那些经常被访问 的页面也被换出，从而使 缺页率升高。

3. LRU：虽然无法知道将来要使用的页面情况，但是可以知道过去使用页面 的情况。LRU 将最近最久 未使用的页面换出。为了实现 LRU，需要在内存中 维护一个所有页面的链表。当一个页面被访问 时，将这个页面移到链表表 头。这样就能保证链表表尾的页面是最近最久未访问的。因为每次访问 都 需要更新链表，因此这种方式实现的 LRU 代价很高。

4. 时钟算法：时钟算法使用环形链表将页面连接起来，再使用一个指针指向 最老的页面。它将整个环

形链表的每一个页面做一个标记，如果标记是 0， 那么暂时就不会被替换，然后时钟算法遍历整个 环，遇到标记为 1 的就替 换，否则将标记为 0 的标记为 1

## 什么是缓冲区溢出？有什么危害？

缓冲区溢出是指当计算机向缓冲区填充数据时超出了缓冲区本身的容量，溢出 的数据覆盖在合法数据 上。 危害有以下两点：

1. 程序崩溃，导致拒绝额服务

2. 跳转并且执行一段恶意代码 造成缓冲区溢出的主要原因是程序中没有仔细检查用户输入

## 什么是虚拟内存

虚拟内存就是说，让物理内存扩充成更大的逻辑内存，从而让程序获得更多的 可用内存。虚拟内存使用 部分加载的技术，让一个进程或者资源的某些页面加 载进内存，从而能够加载更多的进程，甚至能加载 比内存大的进程，这样看起 来好像内存变大了，这部分内存其实包含了磁盘或者硬盘，并且就叫做虚拟 内存

## 讲一讲 **IO** 多路复用？

IO 多路复用是指内核一旦发现进程指定的一个或者多个 IO 条件准备读取，它 就通知该进程。IO 多路复 用适用如下场合：

1. 当客户处理多个描述字时（一般是交互式输入和网络套接口），必须使用 I/O 复用。

2. 当一个客户同时处理多个套接口时，而这种情况是可能的，但很少出现。

3. 如果一个 TCP 服务器既要处理监听套接口，又要处理已连接套接口，一般 也要用到 I/O 复用。

4. 如果一个服务器即要处理 TCP，又要处理 UDP，一般要使用 I/O 复用。

5. 如果一个服务器要处理多个服务或多个协议，一般要使用 I/O 复用。

6. 与多进程和多线程技术相比，I/O 多路复用技术的最大优势是系统开销小， 系统不必创建进程/线 程，也不必维护这些进程/线程，从而大大减小了系 统的开销

## 硬链接和软链接有什么区别

1. 硬链接就是在目录下创建一个条目，记录着文件名与 inode 编号，这个 inode 就是源文件的 inode。删除任意一个条目，文件还是存在，只要引用 数量不为 0。但是硬链接有限制，它不能跨 越文件系统，也不能对目录进行 链接

2. 符号链接文件保存着源文件所在的绝对路径，在读取时会定位到源文件 上，可以理解为 Windows 的快捷方式。当源文件被删除了，链接文件就打 不开了。因为记录的是路径，所以可以为目录建立 符号链接

## 中断的处理过程

1. 保护现场：将当前执行程序的相关数据保存在寄存器中，然后入栈。

2. 开中断：以便执行中断时能响应较高级别的中断请求。

3. 中断处理

4. 关中断：保证恢复现场时不被新中断打扰

5. 恢复现场：从堆栈中按序取出程序数据，恢复中断前的执行状态

## 中断和轮询有什么区别？

1. 轮询：CPU 对特定设备轮流询问。中断：通过特定事件提醒 CPU。

2. 轮询：效率低等待时间长，CPU 利用率不高。中断：容易遗漏问题，CPU 利 用率不高

## 请详细介绍一下 **TCP** 的三次握手机制，为什么要三次握

**手？**

在讲三次握手之前首先要介绍 TCP 报文中两个重要的字段：一个是序号字段， 另一个是确认号字段，这 两个字段将在握手阶段以及整个信息传输过程起到重 要作用。

第一步：客户端 TCP 向服务端的 TCP 发送一个不带额外数据的特殊 TCP 报文 段，该报文段的 SYN 标志 位会被置 1，所以把它称为 SYN 报文段。这时客户端 会选取一个初始序列号（假设为 client\_num）， 并将此编号放置在序号字段中。 该报文段会被封装在一个 IP 数据报中发送给服务器。 第二步：服务器接收到 SYN 报文段后，会为该 TCP 分配缓存和变量，并发送 允许连接的确认报文。在 允许连接的报文中，SYN 标志位仍被置为 1，确认号字 段填的是 client\_num + 1 的值。最后服务端也会 选取一个 server\_num 存放到 序号字段中，这个报文段称为 SYNACK 报文段。

第三步：在接收到 SYNACK 报文段后，客户端最后也要向服务端发送一个确认 报文，这个报文和前两个 不一样，SYN 标志位置 0，在确认号字段中填上 server\_num + 1 的值，并且这个报文段可以携带数据。 一旦完成这 3 个步骤， 客户端和服务器之间就可以相互发送包含数据的报文了。 如果不是三次握手，二 次两次的话，服务器就不知道客户端是否接收到了自己 的 SYNACK 报文段，从而无法建立连接；四次握 手就显得多余了。

## 讲一讲 **SYN** 超时，洪泛攻击，以及解决策略

什么 SYN 是洪泛攻击？ 在 TCP 的三次握手机制的第一步中，客户端会向服务 器发送 SYN 报文段。服务 器接收到 SYN 报文段后会为该 TCP 分配缓存和变量， 如果攻击分子大量地往服务器发送 SYN 报文段， 服务器的连接资源终将被耗尽， 导致内存溢出无法继续服务。 解决策略： 当服务器接受到 SYN 报文段 时，不直接为该 TCP 分配资源，而只 是打开一个半开的套接字。接着会使用 SYN 报文段的源 Id，目的 Id，端口号 以及只有服务器自己知道的一个秘密函数生成一个 cookie，并把 cookie 作为 序列号响应给 客户端。 如果客户端是正常建立连接，将会返回一个确认字段为 cookie + 1 的报文段。 接下来服务器会 根据确认报文的源 Id，目的 Id，端口号以及秘密函数计算出一 个结果，如果结果的值 + 1 等于确认字段 的值，则证明是刚刚请求连接的客户 端，这时候才为该 TCP 分配资源 这样一来就不会为恶意攻击的

SYN 报文段分配资源空间，避免了攻击

## 详细介绍一下 **TCP** 的四次挥手机制，为什么要有

**TIME\_WAIT 状态，为什么需要四次握手？服务器出现了大**

**量 CLOSE\_WAIT 状态如何解决？**

当客户端要服务器断开连接时，客户端 TCP 会向服务器发送一个特殊的报文段， 该报文段的 FIN 标志 位会被置 1，接着服务器会向客户端发送一个确认报文段。 然后服务器也会客户端发送一个 FIN 标志位 为 1 的终止报文段，随后客户端回 送一个确认报文段，服务器立即断开连接。客户端等待一段时间后也 断开连接。 其实四次挥手的过程是很容易理解的，由于 TCP 协议是全双工的，也就是说客 户端和服务端 都可以发起断开连接。两边各发起一次断开连接的申请，加上各 自的两次确认，看起来就像执行了四次 挥手。 为什么要有 TIME\_WAIT 状态？因为客户端最后向服务器发送的确认 ACK 是有 可能丢失的，当出 现超时，服务端会再次发送 FIN 报文段，如果客户端已经关 闭了就收不到了。还有一点是避免新旧连接 混杂。 大量 CLOSE\_WAIT 表示程序出现了问题，对方的 socket 已经关闭连接，而我 方忙于读或写没有 及时关闭连接，需要检查代码，特别是释放资源的代码，或 者是处理请求的线程配置

# 消息队列

## **Kafka** 是什么？主要应用场景有哪些？

Kafka 是一个分布式流式处理平台。 流平台具有三个关键功能：

1. 消息队列：发布和订阅消息流，这个功能类似于消息队列，这也是 Kafka 也被归类为消息队列的原 因。

2. 容错的持久方式存储记录消息流：Kafka 会把消息持久化到磁盘，有效避 免了消息丢失的风险。

3. 流式处理平台：在消息发布的时候进行处理，Kafka 提供了一个完整的流 式处理类库。

Kafka 主要有两大应用场景：

1. 消息队列：建立实时流数据管道，以可靠地在系统或应用程序之间获取数 据。

2. 数据处理：构建实时的流数据处理程序来转换或处理数据流

## 和其他消息队列相比，**Kafka** 的优势在哪里？

我们现在经常提到 Kafka 的时候就已经默认它是一个非常优秀的消息队列了， 我们也会经常拿它跟

RocketMQ、RabbitMQ 对比。我觉得 Kafka 相比其他消息 队列主要的优势如下 :

1. 极致的性能 ：基于 Scala 和 Java 语言开发，设计中大量使用了批量处 理和异步的思想，最高可以 每秒处理千万级别的消息

2. 生态系统兼容性无可匹敌 ：Kafka 与周边生态系统的兼容性是最好的没 有之一，尤其在大数据和流 计算领域

实际上在早期的时候 Kafka 并不是一个合格的消息队列，早期的 Kafka 在消 息队列领域就像是一个衣衫 褴褛的孩子一样，功能不完备并且有一些小问题比 如丢失消息、不保证消息可靠性等等。当然，这也和 LinkedIn 最早开发 Kafka 用于处理海量的日志有很大关系，哈哈哈，人家本来最开始就不是为了 作为消 息队列滴，谁知道后面误打误撞在消息队列领域占据了一席之地

## 什么是 **Producer**、**Consumer**、**Broker**、**Topic**、

**Partition？**

Kafka 将生产者发布的消息发送到 Topic（主题） 中，需要这些消息的消费 者可以订阅这些 Topic（主 题）。Kafka 比较重要的几个概念：

1. Producer（生产者） : 产生消息的一方。

2. Consumer（消费者） : 消费消息的一方。

3. Broker（代理） : 可以看作是一个独立的 Kafka 实例。多个 Kafka Broker 组成一个 Kafka Cluster。

4. Topic（主题） : Producer 将消息发送到特定的主题，Consumer 通过订 阅特定的 Topic(主题) 来 消费消息。

5. Partition（分区） : Partition 属于 Topic 的一部分。一个 Topic 可 以有多个 Partition ，并且同一

Topic 下的 Partition 可以分布在不同 的 Broker 上，这也就表明一个 Topic 可以横跨多个 Broker

。

## **Kafka** 的多副本机制了解吗？

Kafka 为分区（Partition）引入了多副本（Replica）机制。分区 （Partition）中的多个副本之间会有一 个叫做 leader 的家伙，其他副本称为 follower。我们发送的消息会被发送到 leader 副本，然后 follower 副本才 能从 leader 副本中拉取消息进行同步。 生产者和消费者只与 leader 副本交互。你可以 理解为其他副本只是 leader 副本的拷贝，它们的存在只是为了保证消息存储的安全性。当 leader 副本

发 生故障时会从 follower 中选举出一个 leader,但是 follower 中如果有和 leader 同步程度达不到要求的 参加不了 leader 的竞选

## **Kafka** 的多分区（**Partition**）以及多副本（**Replica**）机制

**有 什么好处呢**

1. Kafka 通过给特定 Topic 指定多个 Partition, 而各个 Partition 可以 分布在不同的 Broker 上, 这样 便能提供比较好的并发能力（负载均 衡）。

2. Partition 可以指定对应的 Replica 数, 这也极大地提高了消息存储的安 全性, 提高了容灾能力，不过 也相应的增加了所需要的存储空间。

## **Zookeeper** 在 **Kafka** 中的作用知道吗？

1. Broker 注册 ：在 Zookeeper 上会有一个专门用来进行 Broker 服务器 列表记录的节点。每个 Broker 在启动时，都会到 Zookeeper 上进行注 册，即到 /brokers/ids 下创建属于自己的节点。每 个 Broker 就会将 自己的 IP 地址和端口等信息记录到该节点中去

2. Topic 注册 ： 在 Kafka 中，同一个 Topic 的消息会被分成多个分区并 将其分布在多个 Broker 上， 这些分区信息及与 Broker 的对应关系也都 是由 Zookeeper 在维护。比如我创建了一个名字为 my- topic 的主题并且 它有两个分区，对应到 zookeeper 中会创建这些文件夹： /brokers/topics/my- topic/Partitions/0、/brokers/topics/my□topic/Partitions/1

3. 负载均衡 ：上面也说过了 Kafka 通过给特定 Topic 指定多个 Partition, 而各个 Partition 可以分布 在不同的 Broker 上, 这样便能 提供比较好的并发能力。 对于同一个 Topic 的不同 Partition， Kafka 会尽力将这些 Partition 分布到不同的 Broker 服务器上。当生产者产生 消息后也会尽量投递 到不同 Broker 的 Partition 里面。当 Consumer 消 费的时候，Zookeeper 可以根据当前的 Partition 数量以及 Consumer 数 量来实现动态负载均衡。

## **Kafka** 如何保证消息的消费顺序？

我们在使用消息队列的过程中经常有业务场景需要严格保证消息的消费顺序， 比如我们同时发了 2 个消 息，这 2 个消息对应的操作分别对应的数据库操作 是：

1. 更改用户会员等级。

2. 根据会员等级计算订单价格。

假如这两条消息的消费顺序不一样造成的最终结果就会截然不同。 Kafka 中 Partition(分区)是真正保存 消息的地方，我们发送的消息都被放在 了这里。而我们的 Partition(分区) 又存在于 Topic(主题) 这个概 念中，并 且我们可以给特定 Topic 指定多个 Partition。 每次添加消息到 Partition(分区) 的时候都会采 用尾加法，如上图所示。 Kafka 只能为我们保证 Partition(分区) 中的消息有序。

消息在被追加到 Partition(分区)的时候都会分配一个特定的偏移量 （offset）。Kafka 通过偏移量

（offset）来保证消息在分区内的顺序性。 所以，我们就有一种很简单的保证消息消费顺序的方法：1 个 Topic 只对应一 个 Partition。这样当然可以解决问题，但是破坏了 Kafka 的设计初衷。 Kafka 中发送 1 条消息的时候，可以指定 topic, partition, key,data（数 据） 4 个参数。如果你发送消息的时候指定了 Partition 的话，所有消息都 会被发送到指定的 Partition。并且，同一个 key 的消息可以保证只发送到

同 一个 partition，这个我们可以采用表/对象的 id 来作为 key。

总结一下，对于如何保证 Kafka 中消息消费的顺序，有了下面两种方法：

1. 1 个 Topic 只对应一个 Partition。
2. 发送消息的时候指定 key/Partition

## **Kafka** 如何保证消息不丢失？

生产者丢失消息的情况

生产者(Producer) 调用 send 方法发送消息之后，消息可能因为网络问题并没 有发送过去。 所以，我们 不能默认在调用 send 方法发送消息之后消息发送成功了。为了确定 消息是发送成功，我们要判断消息 发送的结果。但是要注意的是 Kafka 生产者 (Producer) 使用 send 方法发送消息实际上是异步的操作， 我们可以通 过 get()方法获取调用结果，但是这样也让它变为了同步操作

消费者丢失消息的情况

我们知道消息在被追加到 Partition(分区)的时候都会分配一个特定的偏移量 （offset）。偏移量（offset) 表示 Consumer 当前消费到的 Partition(分区) 的所在的位置。Kafka 通过偏移量（offset）可以保证消 息在分区内的顺序 性。 当消费者拉取到了分区的某个消息之后，消费者会自动提交了 offset。自动提 交 的话会有一个问题，试想一下，当消费者刚拿到这个消息准备进行真正消费 的时候，突然挂掉了，消息 实际上并没有被消费，但是 offset 却被自动提交 了。 解决办法也比较粗暴，我们手动关闭自动提交 offset，每次在真正消费完消息 之后再自己手动提交 offset 。 但是，细心的朋友一定会发现，这样会带 来 消息被重新消费的问题。比如你刚刚消费完消息之后，还没提交 offset，结果 自己挂掉了，那么这个 消息理论上就会被消费两次

## **Kafka** 判断一个节点是否还活着有那两个条件？

1. 节点必须可以维护和 ZooKeeper 的连接，Zookeeper 通过心跳机制检查每 个节点的连接；

2. 如果节点是个 follower,他必须能及时的同步 leader 的写操作，延时不能太久

## **producer** 是否直接将数据发送到 **broker** 的 **leader**（主节

**点）**

producer 直接将数据发送到 broker 的 leader(主节点)，不需要在多个节点 进行分发，为了 帮助 producer 做到这点，所有的 Kafka 节点都可以及时的告知:哪些节点是 活动的，目标 topic 目标分区的 leader 在哪。这样 producer 就可以直接将消息发送到目 的地了

## **Kafka consumer** 是否可以消费指定分区消息吗？

Kafa consumer 消费消息时，向 broker 发出"fetch"请求去消费特定分区的消 息，consumer 指定消息 在日志中的偏移量（offset），就可以消费从这个位置 开始的消息，customer 拥有了 offset 的控制权， 可以向后回滚去重新消费之 前的消息，这是很有意义的

## **Kafka** 高效文件存储设计特点是什么

1. Kafka 把 topic 中一个 parition 大文件分成多个小文件段，通过多个小 文件段，就容易定期清除或 删除已经消费完文件，减少磁盘占用。

2. 通过索引信息可以快速定位 message 和确定 response 的最大大小。

1. 通过 index 元数据全部映射到 memory，可以避免 segment file 的 IO 磁盘操作。

4. 通过索引文件稀疏存储，可以大幅降低 index 文件元数据占用空间大小。

## **partition** 的数据如何保存到硬盘

topic 中的多个 partition 以文件夹的形式保存到 broker，每个分区序号从 0 递增，且消息有序。 Partition 文件下有多个 segment（xxx.index，xxx.log） segment 文件里的 大小和配置文件大小一致 可以根据要求修改，默认为 1g。 如果大小大于 1g 时，会滚动一个新的 segment 并且以上一个 segment 最后 一条消息的偏移量命名

## **kafka** 生产数据时数据的分组策略是怎样的

生产者决定数据产生到集群的哪个 partition 中，每一条消息都是以（key， value）格式，Key 是由生 产者发送数据传入，所以生产者（key）决定了数据 产生到集群的哪个 partition

## **consumer** 是推还是拉？

customer 应该从 brokes 拉取消息还是 brokers 将消息推送到 consumer，也 就是 pull 还 push。在这 方面，Kafka 遵循了一种大部分消息系统共同的传统 的设计：producer 将消息推送到 broker， consumer 从 broker 拉取消息。

push 模式，将消息推送到下游的 consumer。这样做有好处也有坏处：由 broker 决定消息推送的速 率，对于不同消费速率的 consumer 就不太好处理 了。消息系统都致力于让 consumer 以最大的速率最 快速的消费消息，但不幸 的是，push 模式下，当 broker 推送的速率远大于 consumer 消费的速率时， consumer 恐怕就要崩溃了。最终 Kafka 还是选取了传统的 pull 模式

## **Kafka** 维护消费状态跟踪的方法有什么？

大部分消息系统在 broker 端的维护消息被消费的记录：一个消息被分发到 consumer 后 broker 就马上 进行标记或者等待 customer 的通知后进行标记。 这样也可以在消息在消费后立马就删除以减少空间占 用

# 分布式

## 分布式服务接口的幂等性如何设计

所谓幂等性，就是说一个接口，多次发起同一个请求，你这个接口得保证结果 是准确得。比如不能多扣 款。不能多插入一条数据，不能将统计值多加了 1， 这就是幂等性。 其实保证幂等性主要是三点：

1. 对于每个请求必须有一个唯一的标识，举个例子：订单支付请求，肯定得 包含订单 ID，一个订单 ID 最多支付一次

2. 每次处理完请求之后，必须有一个记录标识这个请求处理过了，比如说常 见得方案是再 MySQL 中 记录个状态啥得，比如支付之前记录一条这个订单 得支付流水，而且支付流水采用 order id 作为唯 一键（unique key）。只 有成功插入这个支付流水，才可以执行实际得支付扣款

3. 每次接收请求需要进行判断之前是否处理过得逻辑处理，比如说，如果有 一个订单已经支付了，就 已经有了一条支付流水，那么如果重复发送这个 请求，则此时先插入支付流水，order id 已经存在 了，唯一键约束生效， 报错插入不进去得。然后你就不用再扣款了

## 分布式系统中的接口调用如何保证顺序性

可以接入 MQ，如果是系统 A 使用多线程处理的话，可以使用内存队列，来保证 顺序性，如果你要

100%的顺序性，当然可以使用分布式锁来搞，会影响系统的 并发性

## 说说 **ZooKeeper** 一般都有哪些使用场景

1. 分布式协调：这个其实就是 ZooKeeper 很经典的一个用法，简单来说，就 好比，你系统 A 发送个 请求到 MQ，然后 B 消费了之后处理。那 A 系统如何 指导 B 系统的处理结果？用 ZooKeeper 就可 以实现分布式系统之间的协调 工作。A 系统发送请求之后可以在 ZooKeeper 上对某个节点的值注 册个监听 器，一旦 B 系统处理完了就修改 ZooKeeper 那个节点的值，A 立马就可以收 到通知，完 美解决

2. 分布所锁：对某一个数据联系发出两个修改操作，两台机器同时收到请 求，但是只能一台机器先执 行另外一个机器再执行，那么此时就可以使用 ZooKeeper 分布式锁，一个机器接收到了请求之后 先获取 ZooKeeper 上的一 把分布式锁，就是可以去创建一个 znode，接着执行操作，然后另外一 个机 器也尝试去创建那个 znode，结果发现自己创建不了，因为被别人创建了， 那只能等着，等 等一个机器执行完了自己再执行

3. 配置信息管理：ZooKeeper 可以用作很多系统的配置信息的管理，比如 Kafka，storm 等等很多分 布式系统都会选用 zk 来做一些元数据，配置信息 的管理，包括 Dubbo 注册中心不也支持 ZooKeeper 么

4. HA 高可用性：这个应该是很常见的，比如 hdfs，yarn 等很多大数据系统， 都选择基于 ZooKeeper 来开发 HA 高可用机制，就是一个重要进程一般会主 备两个，主进程挂了立马通过 ZooKeeper 感知到切换到备份进程。

## 分布式事务了解吗？

1. XA 方案/两阶段提交方案 第一个阶段（先询问） 第二个阶段（再执行）

2. TCC 方案 TCC 的全程是：Try、Confirm、Cancel 这个其实是用到了补偿的概念，分为了三个阶段 Try 阶段：这个阶段说的是对各个服务的资源做检测以及对资源进行锁定 或者预留 Confirm 阶段： 这个阶段说的是在各个服务中执行实际的操作 Cancel 阶段：如果任何一个服务的业务方法执行出 错，那么这里就需要进 行补偿，就是执行已经成功的业务逻辑的回滚操作

3. 本地消息表

4. 可靠消息最终一致性方案

5. 最大努力通知方案

## 那常见的分布式锁有哪些解决方案

1. Reids 的分布式锁，很多大公司会基于 Reidis 做扩展开发

2. 基于 etcd

3. 基于数据库，比如 MySQL

## **MySQL** 如何做分布式锁

方法一： 利用 MySQL 的锁表，创建一张表，设置一个 UNIQUE KEY 这个 KEY 就是要锁 的 KEY，所以 同一个 KEY 在 MySQL 表里只能插入一次了，这样对锁的竞争就交 给了数据库，处理同一个 KEY 数据库 保证了只有一个节点能插入成功，其他节 点都会插入失败。 DB 分布式锁的实现：通过主键 id 的唯一性 进行加锁，说白了就是加锁的形式 是向一张表中插入一条数据，该条数据的 ID 就是一把分布式锁，例如 当一次请 求插入了一条 ID 为 1 的数据，其他想要进行插入数据的并发请求必须等第一次 请求执行完成 后删除这条 ID 为 1 的数据才能继续插入，实现了分布式锁的功 能

方法二： 使用流水号+时间戳做幂等操作，可以看作是一个不会释放的锁

## 你了解业界哪些大公司的分布式锁框架

1. Google:Chubby

Chubby 是一套分布式协调系统，内部使用 Paxos 协调 Master 与 Replicas。 Chubby lock service 被

应用在 GFS, BigTable 等项目中，其首要设计目标是 高可靠性，而不是高性能。 Chubby 被作为粗粒度 锁使用，例如被用于选主。持有锁的时间跨度一般为小时 或天，而不是秒级。 Chubby 对外提供类似于 文件系统的 API，在 Chubby 创建文件路径即加锁操作。 Chubby 使用 Delay 和 SequenceNumber 来 优化锁机制。Delay 保证客户端异常释放锁时，Chubby 仍认为该客户端一直持有锁。Sequence number 指锁的持有者 向 Chubby 服务端请求一个序号（包括几个属性），然后之后在需要使用锁的时 候将该序号一并发给 Chubby 服务器，服务端检查序号的合法性，包括 number 是否有效等。

1. 京东 SharkLock

SharkLock 是基于 Redis 实现的分布式锁。锁的排他性由 SETNX 原语实现，使 用 timeout 与续租机制 实现锁的强制释放

1. 蚂蚁金服 SOFAJRaft-RheaKV 分布式锁

RheaKV 是基于 SOFAJRaft 和 RocksDB 实现的嵌入式、分布式、高可用、强一 致的 KV 存储类库。 RheaKV 对外提供 lock 接口，为了优化数据的读写，按不同的存储类型，提供 不同的锁特性。RheaKV 提供 wathcdog 调度器来控制锁的自动续租机制，避免 锁在任务完成前提前释放，和锁永不释放造成死 锁

1. Netflix: Curator

Curator 是 ZooKeeper 的客户端封装，其分布式锁的实现完全由 ZooKeeper 完 成。 在 ZooKeeper 创 建 EPHEMERAL\_SEQUENTIAL 节点视为加锁，节点的 EPHEMERAL 特性保证了锁持有者与 ZooKeeper 断开时强制释放锁；节点的 SEQUENTIAL 特性避免了加锁较多时的惊群效应

## 请讲一下你对 **CAP** 理论的理解

在理论计算机科学中，CAP 定理（CAP theorem），又被称作布鲁尔定理 （Brewer’s theorem），它 指出对于一个分布式计算系统来说，不可能同时满 足以下三点：

1. Consistency（一致性） 指数据在多个副本之间能够保持一致的特性 （严格的一致性）

2. Availability（可用性） 指系统提供的服务必须一直处于可用的状态， 每次请求都能获取到非错的响 应（不保证获取的数据为最新数据）

3. Partition tolerance（分区容错性） 分布式系统在遇到任何网络分区故 障的时候，仍然能够对外提 供满足一致性和可用性的服务，除非整个网络 环境都发生了故障

Spring Cloud 在 CAP 法则上主要满足的是 A 和 P 法则，Dubbo 和 Zookeeper 在 CAP 法则主要满足的 是 C 和 P 法则。 CAP 仅适用于原子读写的 NOSQL 场景中，并不适合数据库系统。现在的分布式 系统具 有更多特性比如扩展性、可用性等等，在进行系统设计和开发时，我们 不应该仅仅局限在 CAP 问题上。 现实生活中，大部分人解释这一定律时，常常简单的表述为：“一致性、可用 性、分区容忍性三者你只能 同时达到其中两个，不可能同时达到”。实际上这 是一个非常具有误导性质的说法，而且在 CAP 理论诞 生 12 年之后，CAP 之父也 在 2012 年重写了之前的论文。 当发生网络分区的时候，如果我们要继续服 务，那么强一致性和可用性只能 2 选 1。也就是说当网络分区之后 P 是前提，决定了 P 之后才有 C 和 A 的选择。 也就是说分区容错性（Partition tolerance）我们是必须要实现的。

## 请讲一下你对 **BASE** 理论的理解

BASE 理论由 eBay 架构师 Dan Pritchett 提出，在 2008 年上被分表为论文，并 且 eBay 给出了他们在 实践中总结的基于 BASE 理论的一套新的分布式事务解决 方案。 BASE 是 Basically Available（基本可 用） 、Soft-state（软状态） 和 Eventually Consistent（最终一致性） 三个短语的缩写。BASE 理论是 对 CAP 中一致性和可用性权衡的结果，其来源于对大规模互联网系统分布式实践的总 结，是基于 CAP 定理逐步演化而来的，它大大降低了我们对系统的要求。 BASE 理论的核心思想是即使无法做到强一致 性，但每个应用都可以根据自身业 务特点，采用适当的方式来使系统达到最终一致性。也就是牺牲数据

的一致性 来满足系统的高可用性，系统中一部分数据不可用或者不一致时，仍需要保持 系统整体“主要

可用”。 针对数据库领域，BASE 思想的主要实现是对业务数据进行拆分，让不同的数据 分布在不同的机 器上，以提升系统的可用性，当前主要有以下两种做法：

1. 按功能划分数据库

1. 分片（如开源的 MyCat、Amoeba 等）

## 请讲一下 **BASE** 理论的三要素

基本可用

基本可用是指分布式系统在出现不可预知故障的时候，允许损失部分可用性。 但是，这绝不等价于系统 不可用。 比如：

1. 响应时间上的损失：正常情况下，一个在线搜索引擎需要在 0.5 秒之内返 回给用户相应的查询结 果，但由于出现故障，查询结果的响应时间增加了 1~2 秒

2. 系统功能上的损失：正常情况下，在一个电子商务网站上进行购物的时 候，消费者几乎能够顺利完 成每一笔订单，但是在一些节日大促购物高峰 的时候，由于消费者的购物行为激增，为了保护购物 系统的稳定性，部分 消费者可能会被引导到一个降级页面

软状态

软状态指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响 系统的整体可用性，即允 许系统在不同节点的数据副本之间进行数据同步的过 程存在延时

最终一致性

强调的是系统中所有的数据副本，在经过一段时间的同步后，最终能够达到一 个一致的状态。因此，最 终一致性的本质是需要系统保证最终数据能够达到一 致，而不需要实时保证系统数据的强一致性

## 分布式与集群的区别是什么

分布式：一个业务分拆多个子业务，部署在不同的服务器上 集群：同一个业务，部署在多个服务器上。 比如之前做电商网站搭的 redis 集 群以及 solr 集群都是属于将 Redis 服务器提供的缓存服务以及 solr 服 务器提 供的搜索服务部署在多个服务器上以提高系统性能、并发量解决海量存储问 题。

## 请说一下对两阶段提交协议的理解

分布式系统的一个难点是如何保证架构下多个节点在进行事务性操作的时候保 持一致性。为实现这个目 的，二阶段提交算法的成立基于以下假设：

1. 该分布式系统中，存在一个节点作为协调者(Coordinator)，其他节点作为 参与者(Cohorts)。且节 点之间可以进行网络通信。

2. 所有节点都采用预写式日志，且日志被写入后即被保持在可靠的存储设备 上，即使节点损坏不会导 致日志数据的消失。

3. 所有节点不会永久性损坏，即使损坏后仍然可以恢复 第一阶段（投票阶段）

1. 协调者节点向所有参与者节点询问是否可以执行提交操作(vote)，并开始 等待各参与者节点的响 应。

2. 参与者节点执行询问发起为止的所有事务操作，并将 Undo 信息和 Redo 信 息写入日志。（注意： 若成功这里其实每个参与者已经执行了事务操作）

3. 各参与者节点响应协调者节点发起的询问。如果参与者节点的事务操作实 际执行成功，则它返回一 个”同意”消息；如果参与者节点的事务操作实 际执行失败，则它返回一个”中止”消息

第二阶段（提交执行阶段）

当协调者节点从所有参与者节点获得的相应消息都为”同意”：

1. 协调者节点向所有参与者节点发出”正式提交(commit)”的请求。

2. 参与者节点正式完成操作，并释放在整个事务期间内占用的资源。

3. 参与者节点向协调者节点发送”完成”消息。

4. 协调者节点受到所有参与者节点反馈的”完成”消息后，完成事务 如果任一参与者节点在第一阶段返回的响应消息为”中止”：

1. 协调者节点向所有参与者节点发出”回滚操作(rollback)”的请求。

2. 参与者节点利用之前写入的 Undo 信息执行回滚，并释放在整个事务期间内 占用的资源。

3. 参与者节点向协调者节点发送”回滚完成”消息。

4. 协调者节点受到所有参与者节点反馈的”回滚完成”消息后，取消事务

## 请讲一下对 **TCC** 协议的理解

Try Confirm Cancel

Try：尝试待执行的业务 ，这个过程并未执行业务，只是完成所有业务的 一致性检查，并预留好执行所 需的全部资源。

1. Confirm：执行业务，这个过程真正开始执行业务，由于 Try 阶段已经完成 了一致性检查，因此本 过程直接执行，而不做任何检查。并且在执行的过 程中，会使用到 Try 阶段预留的业务资源。

2. Cancel：取消执行的业务，若业务执行失败，则进入 Cancel 阶段，它会释 放所有占用的业务资 源，并回滚 Confirm 阶段执行的操作。

# **Memcached**

## **Memcached** 的多线程是什么？如何使⽤它们？

线程就是定律（threads rule）！在 Steven Grimm 和 Facebook 的努⼒下， Memcached 1.2 及更⾼ 版本拥有了 多线程模式。多线程模式允许 Memcached 能够充分利⽤多个 CPU，并在 CPU 之间共享所 有的缓存数据。 Memcached 使⽤⼀种简单的锁机制来保证数据更新操作的互斥。相⽐在同⼀ 个物理机 器上运⾏多个 Memcached 实例，这种⽅式能够更有效地处理 multi gets。 如果你的系统负载并不重， 也许你不需要启⽤多线程⼯作模式。如果你在运⾏ ⼀个拥有⼤规模硬件的、庞⼤的⽹ 站，你将会看到多 线程的好处。 简单地总结⼀下：命令解析（Memcached 在这⾥花了⼤部分时间）可以运⾏ 在 多线程模 式下。Memcached 内部对数据的操作是基于很多全局锁的（因此 这部分⼯作不是多线程的）。未来对 多 线程模式的改进，将移除⼤量的全局锁， 提⾼ Memcached 在负载极⾼的场景下的性能。

## **Memcached** 是什么，有什么作⽤

Memcached 是⼀个开源的，⾼性能的内存绶存软件，从名称上看 Mem 就是 内存的意思，⽽ Cache 就 是缓存的 意思。Memcached 的作⽤：通过在事先规 划好的内存空间中临时绶存数据库中的各类数据， 以达到减少业务对数 据库的 直接⾼并发访问，从⽽达到提升数据库的访问性能，加速⽹站集群动态应⽤ 服 务的能⼒

## **Memcached** 与 **Redis** 的区别？

1. Redis 不仅仅⽀持简单的 K/V类型的数据，同时还提供 list，set，zset， hash 等数据结构的存储。

⽽ memcache 只⽀持简单数据类型，需要客户 端⾃⼰处理复杂对象。

2. Redis ⽀持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候 可以再次加载进⾏使⽤

（PS：持久化在 rdb、aof）。

3. 由于 Memcache 没有持久化机制，因此宕机所有缓存数据失效。Redis 配 置为持久化，宕机重启 后，将⾃动加载宕机时刻的数据到缓存系统中。具 有更好的灾备机制。

4. Memcache 可以使⽤ Magent 在客户端进⾏⼀致性 hash 做分布式。 Redis ⽀持在服务器端做分布 式（PS:Twemproxy/Codis/Redis-cluster 多 种分布式实现⽅式）。

5. Memcached 的简单限制就是键（key）和 Value 的限制。最⼤键⻓为 250 个字符。可以接受的储 存数据不能超过 1MB（可修改配置⽂件变⼤），因 为这是典型 slab 的最⼤值，不适合虚 拟机使

⽤。⽽ Redis 的 Key ⻓度⽀ 持到 512K。

6. Redis 使⽤的是单线程模型，保证了数据按顺序提交。Memcache 需要使 ⽤ cas 保证数据⼀致性。 CAS（Check and Set）是⼀个确保并发⼀致性的 机制，属于“乐观锁”范畴；原理很简 单：拿版本 号，操作，对⽐版本 号，如果⼀致就操作，不⼀致就放弃任何操作。

7. CPU 利⽤：由于 Redis 只使⽤单核，⽽ Memcached 可以使⽤多核，所以 平均每⼀个核上 Redis 在存储⼩数据时⽐ Memcached 性能更⾼。⽽在 100k 以上的数据中，Memcached 性能要 ⾼于 Redis 。

8. Memcached 内存管理：使⽤ Slab Allocation。原理相当简单，预先分配 ⼀系列⼤⼩固定的组，然 后根据数据⼤⼩选择最合适的块存储。避免了内 存碎⽚。（缺点：不能变⻓，浪费了⼀定 空间） Memcached 默认情况下下 ⼀个 slab 的最⼤值为前⼀个的 1.25 倍。

9. Redis 内存管理： Redis 通过定义⼀个数组来记录所有的内存分配情况， Redis 采⽤的是包装的 malloc/free，相较于 Memcached 的内存 管理⽅ 法来说，要简单很多。由于 malloc ⾸先 以链表 的⽅式搜索已管理的内存 中可⽤的空间分配，导致内存碎⽚⽐较多。

## 如果缓存数据在导出导⼊之间过期了，你⼜怎么处理这些数

**据呢**

因此，批量导出导⼊数据并不像你想象中的那么有⽤。不过在⼀个场景倒是很 有⽤。如果你有⼤量的从不变化

的数 据，并且希望缓存很快热（warm）起 来，批量导⼊缓存数据是很有帮助的。虽然这个场景并不典型，但 却经常发 ⽣，因此我们会考虑在将来实现批量导出导⼊的功能。 如果⼀个 Memcached 节点 down 了让你 很痛苦，那么你还会陷⼊其他很多麻 烦。你的系统太脆弱了。你需要做 ⼀些优化⼯作。⽐如处理”惊群”问题

（⽐ 如 Memcached 节点都失效了，反复的查询让你的数据库不堪重负…这个 问题 在 FAQ 的其他提到 过），或者优化不好的查询。记住，Memcached 并不是你 逃避优化查询的借⼝

## 如何实现集群中的 **session** 共享存储

Session 是运⾏在⼀台服务器上的，所有的访问都会到达我们的唯⼀服务器 上，这样我们可以根据客户 端传来的 sessionID，来获取 session，或在对应 Session 不存在的情况下（session ⽣命周期到了/⽤户 第⼀次登录），创 建⼀ 个新的 Session；但是，如果我们在集群环境下，假设我们有两台服务器 A， B，⽤户的请求会由 Nginx 服务器进 ⾏转发（别的⽅案也是同理），⽤户登录 时，Nginx 将请求转发⾄ 服务器 A 上，A 创建了新的 session，并将 SessionID 返回给客户端，⽤户在浏览其他⻚⾯时，客户端验 证登录状态， Nginx 将请求转发⾄服务器 B，由于 B 上并没有对应客户端发来 sessionId 的 session， 所以会重新创建⼀个新的 session，并且再将这个新的 sessionID 返 回给客户端，这样，我们可以想象

⼀下，⽤户每⼀次操作都有 1/2 的概 率进⾏ 再次的登录，这样不仅对⽤户体验特别差，还会让服务器上 的 session 激增， 加⼤服务器的运⾏压⼒。 为了解决集群环境下的 seesion 共享问题，共有 4 种解决⽅ 案：

1. 粘性 session 粘性 session 是指 Ngnix 每次都将同⼀⽤户的所有请求转发⾄同⼀台服务器 上，即 将⽤户与服务器绑定

2. 服务器 session 复制 即每次 session 发⽣变化时，创建或者修改，就⼴播给所有集群中的服务器，

使所有的服务器上的 session 相同。

1. session 共享 缓存 session，使⽤ Redis， Memcached。

4. session 持久化 将 session 存储⾄数据库中，像操作数据⼀样才做 session。

## **Memcached** 和 **MySQL** 的 **query cache** 相⽐，有什么优

**缺点**

把 Memcached 引⼊应⽤中，还是需要不少⼯作量的。MySQL 有个使⽤⽅便 的 query cache，可以⾃ 动地缓存 SQL 查询的结果，被缓存的 SQL 查询可以 被反复地快速执⾏。Memcached 与之相⽐，怎么 样呢？MySQL 的 query cache 是集中式的，连接到该 query cache 的 MySQL 服务器都会受益

1. 当你修改表时，MySQL 的 query cache 会⽴刻被刷新（flush）。存储⼀ 个 Memcached item 只 需要很少的时间，但是当写操作很频繁时，MySQL 的 query cache 会经常让所有缓存数据 都失效
2. 在多核 CPU 上，MySQL 的 query cache 会遇到扩展问题（scalability issues）。在多核 CPU 上，

query cache 会增加⼀个全局锁（global lock）, 由于需要刷新更多的缓存数据，速度会变得更慢

3. 在 MySQL 的 query cache 中，我们是不能存储任意的数据的（只能是 SQL 查询结果）。⽽利⽤ Memcached，我们可以搭建出各种⾼效的缓存。⽐ 如，可以执⾏多个独⽴的查询，构建出⼀个⽤ 户对象 （user object），然后将 ⽤户对象缓存到 Memcached 中。⽽ query cache 是 SQL 语句级 别的，不可 能做 到这⼀点。在⼩的⽹站中，query cache 会有所帮助，但随着⽹站规模的 增加， query cache 的弊将⼤于 利。

4. query cache 能够利⽤的内存容量受到 MySQL 服务器空闲内存空间的限 制。给数据库服务器增加 更多的内存 来缓存数据，固然是很好的。但是，有了 Memcached，只要你有空闲的内存，都可以

⽤来增加 Memcached 集群的规 模，然后你就可以缓存更多的数据

## **Memcached** 是原⼦的吗？

所有的被发送到 Memcached 的单个命令是完全原⼦的。如果你针对同⼀份数 据同时发送了⼀个 set 命 令和⼀个 get 命令，它们不会影响对⽅。它们将被串 ⾏化、先后执⾏。即使在多线程模 式，所有的命令 都是原⼦的，除⾮程序有 bug。 命令序列不是原⼦的。如果你通过 get 命令获取了⼀个 item，修改了 它，然后 想把它 set 回 Memcached，我们 不保证这个 item 没有被其他进程 （process，未必是操作 系统中的进程）操作过。在并发的情况下，你也可能 覆 写了⼀个被其他进程 set 的 item。 Memcached 1.2.5 以及更⾼版本，提供了 gets 和 cas 命令，它们可以解决上 ⾯的问题。如果你使⽤ gets 命令查 询 某个 key 的 item，Memcached 会给你 返回该 item 当前值的唯⼀标识。如果你覆写了这个 item 并想 把它写回 到 Memcached 中，你可以通过 cas 命令把那个唯⼀标识⼀起发送给 Memcached。如果该 item 存放在 Memcached 中的唯⼀标识与你提供的⼀ 致，你的写操作将会成功。如果另⼀个进程在这期 间也修改了这个 item，那 么该 item 存放在 Memcached 中的唯⼀标识将会改变，你的写操作就会失 败

## **Memcached** 能够更有效地使⽤内存吗

Memcache 客户端仅根据哈希算法来决定将某个 key 存储在哪个节点上，⽽不 考虑节点的内存⼤⼩。 因此，你可 以在不同的节点上使⽤⼤⼩不等的缓存。但 是⼀般都是这样做的：拥有较多内存的节点上可 以运⾏多个 Memcached 实 例，每个实例使⽤的内存跟其他节点上的实例相同

## **Memcached** 的内存分配器是如何⼯作的？为什么不适⽤

**malloc/free？ 为何要使⽤ slabs？**

实际上，这是⼀个编译时选项。默认会使⽤内部的 slab 分配器。你确实确实应 该使⽤内建的 slab 分配 器。最早的 时候，Memcached 只使⽤ malloc/free 来管理内存。然⽽，这种⽅式不能与 OS 的内存管 理以前很好地⼯作。反 复地 malloc/free 造成了内存碎⽚，OS 最终花费⼤量的时间去查找连续的内存块 来 满⾜ malloc 的请求，⽽不是运 ⾏ Memcached 进程。如果你不同意，当然可 以使⽤ malloc。 slab 分配器就是为了解决这个问题⽽⽣的。内存被分配并划分成 chunks，⼀ 直被重复使⽤。因为内存被划分 成⼤ ⼩不等的 slabs，如果 item 的⼤⼩与被选 择存放它的 slab 不是很合适的话，就会浪费⼀些内存。 Steven Grimm 正在这 ⽅⾯已经做出了有效的改进

# **MongoDB** ⾯试题

## **ObjectID** 有哪些部分组成

⼀共有四部分组成:时间戳、客户端 ID、客户进程 ID、三个字节的增量计数 器

## 当我试图更新⼀个正在被迁移的块**(chunk)**上的⽂档时会发

**⽣什 么**

更新操作会⽴即发⽣在旧的分⽚(shard)上,然后更改才会在所有权转移 (ownership transfers)前复制到 新的分⽚ 上

## 为什么要在 **MongoDB** 中使⽤分析器

MongoDB 中包括了⼀个可以显示数据库中每个操作性能特点的数据库分析 器。通过这个分析器你可以 找到⽐预期 慢的查询(或写操作);利⽤这⼀信息,⽐如, 可以确定是否需要添加索引。

## 解释⼀下 **MongoDB** 中的索引是什么

索引是 MongoDB 中的特殊结构，它以易于遍历的形式存储⼀⼩部分数据集。 索引按索引中指定的字段 的值排序，存储特定字段或⼀组字段的值

## 什么是集合（表）

集合就是⼀组 MongoDB ⽂档。它相当于关系型数据库（RDBMS）中的表这 种概念。集合位于单独的

⼀个数据 库中。⼀个集合内的多个⽂档可以有多个不 同的字段。⼀般来说，集合中的⽂档都有着相同或 相关的⽬的。

## 什么是 **NoSQL** 数据库？**NoSQL** 和 **RDBMS** 有什么区别？

**在哪 些情况下 使⽤和不使⽤ NoSQL 数据库？**

NoSQL 是⾮关系型数据库，NoSQL = Not Only SQL。 关系型数据库采⽤的结构化的数据，NoSQL 采⽤ 的是键值 对的⽅式存储数据。 在处理⾮结构化/半结构化的⼤数据时；在⽔平⽅向上进⾏扩展时，随时 应对动 态增加的数据 项时可以优先考虑。 使⽤ NoSQL 数据库。 在考虑数据库的成熟度；⽀持；分析和 商业智能；管理 及专业性等问题时，应优先考虑关 系型数据库

## 提及插⼊⽂档的命令语法是什么？

⽤于插⼊⽂档的命令语法是 database.collection.insert（⽂档）

## 如果在⼀个分⽚（**shard**）停⽌或者很慢的时候**,**我发起⼀个

**查询 会怎样**

如果⼀个分⽚（shard）停⽌了，除⾮查询设置了“partial”选项,否则查询会 返回⼀个错误。如果⼀个分

⽚ （shard）响应很慢，MongoDB 则会等待它的 响应

## 如何执⾏事务**/**加锁？

因为 MongoDB 设计就是轻量⾼性能，所以没有传统的锁和复杂的事务的回滚