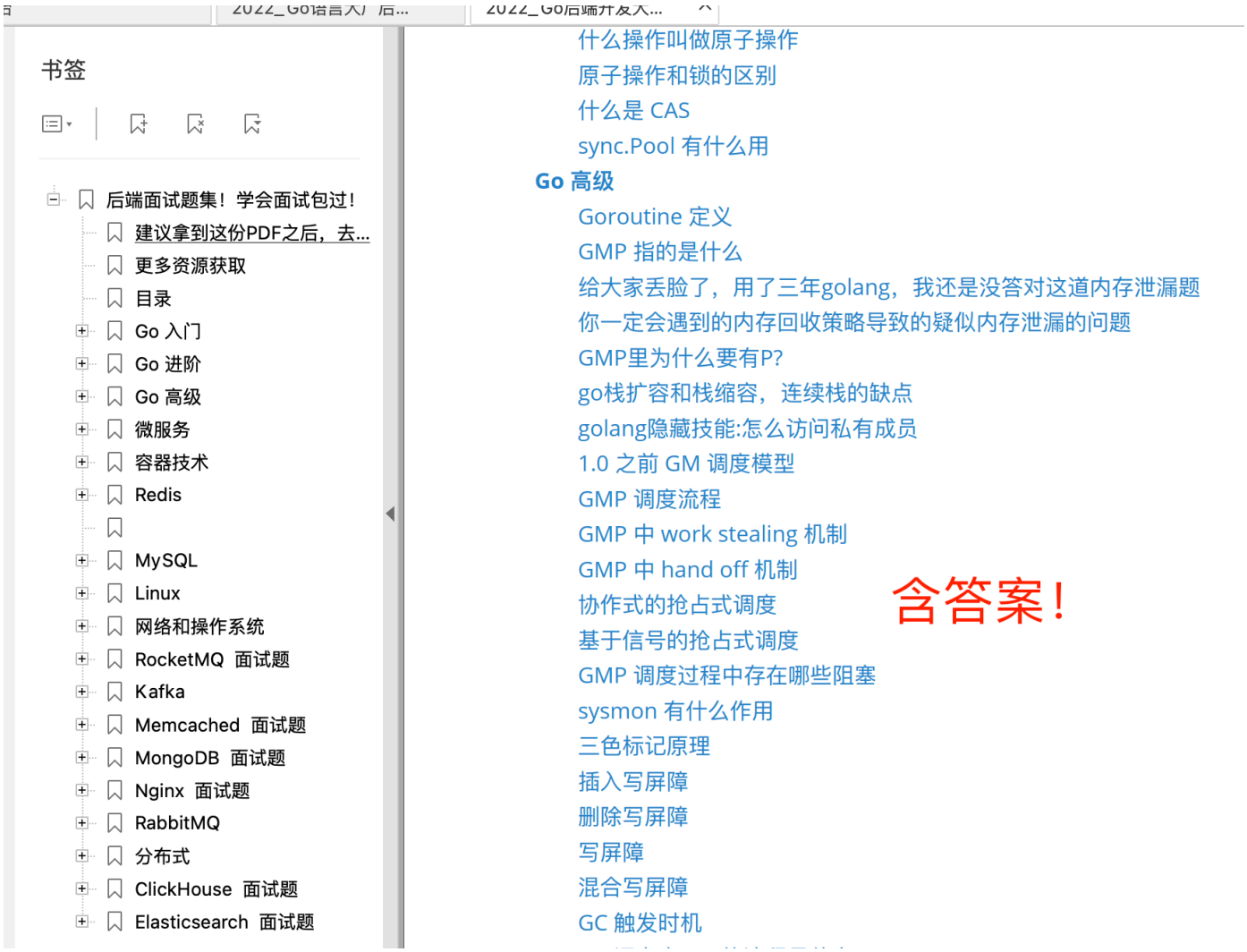
后端面试题集！学会面试包过！

大家好，我是小白。大家知道，我已经从事后端程序员好多年了，在学习和参与面试的过程中遇到过很多面试题。

最近花了些时间整理了一下后端程序员成长路线和高频面试题，这是一份涵盖大部分后端程序员所需要掌握的**核心知识**。

这些面试题从**能力模型上对标字节资深后端开发**，因为字节主要使用GO语言，因此语言方面的面试题，主要是面向go相关的题目。但除了语言这块，其他中间件和工程能力相关的面试题，**对于所有后端开发都适用！！**



如果不想要go相关的面试题，只要其他后端面试题，可以在后台回复【后端面试】。

面试题集PDF还会**不断迭代更新**，一般两个月更新一次，后续最新版本都会在我的个人公众号「**小白debug**」里第一时间发布！

建议拿到这份PDF之后，去关注「小白debug」，并重新获取一份最新的，因为你手上这份PDF可能已经过时了！！

在公众号里回复【**面试**】即可获得！！

如果大家看完电子书，觉得内容还不错，强烈希望大家能在**微信公众号**里搜索关注，并**星标**我的公众号，第一时间获取最新更新内容！

还有**技术划水【交流群】**，点击公众号右下角**【联系我】**或**扫描以下二维码**，备注"**进群**"，欢迎大家进群交流~。

如果大家对网络基础感兴趣的话，公众号里也有非常多相关的文章，也欢迎关注收看哈。**图解网络相关**的文章，已经重新整理成一份的电子书。**在公众号内回复【网络】，就可以获得整理的PDF电子书。**

同时我也为公众号粉丝准备了一些学习资料，关注公众号并回复**【视频】**、**【操作系统】**、**【网络】**即可获得相应学习资料！

另外，学习资料也会**不断更新**，后续会根据读者需求整理更多相关资源，有需要的读者也可以在公众号内留言。

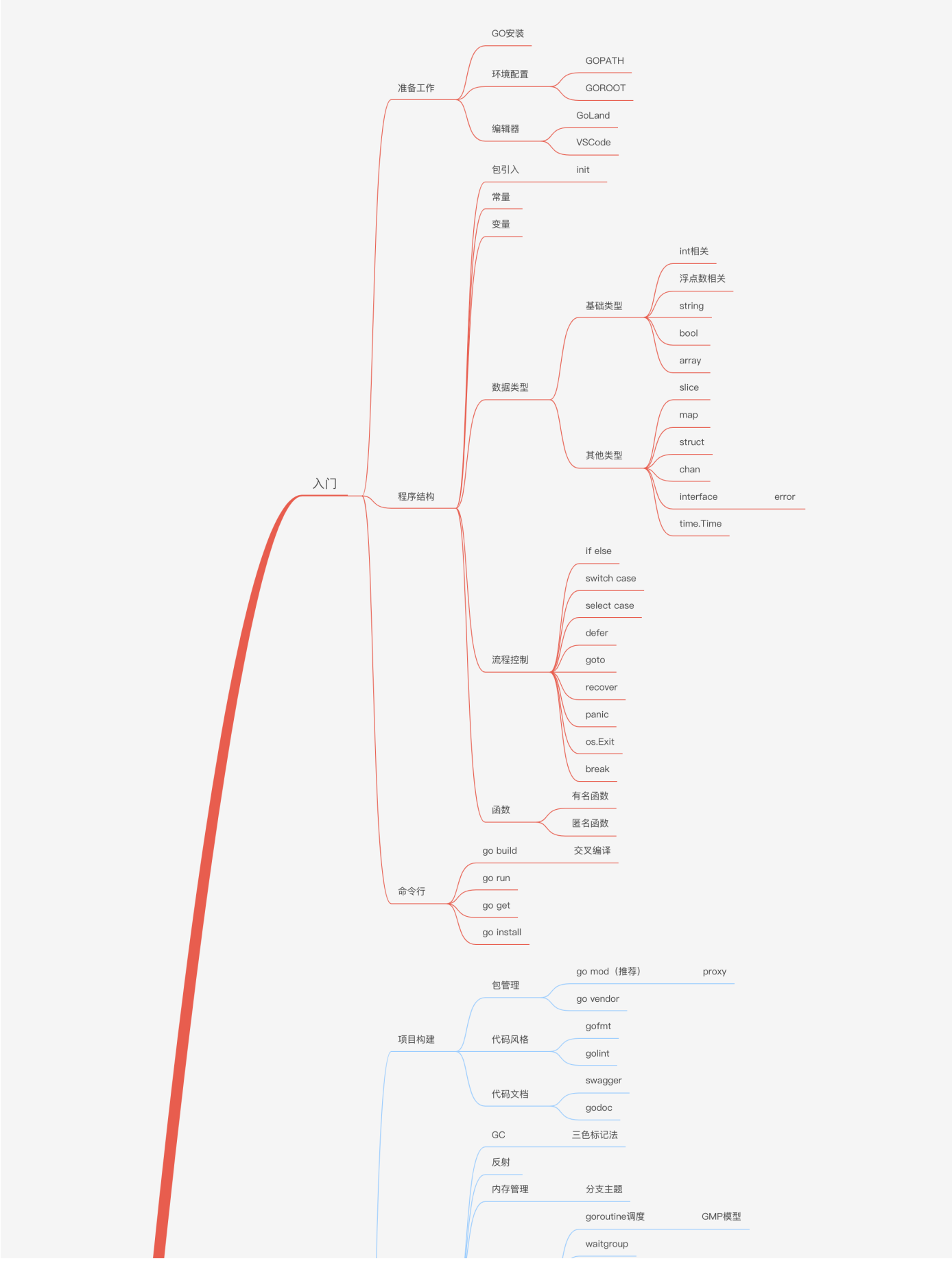
PDF内的题目非全部原创，主要来源于学习过程中不断收集到的面试题，初衷是帮助大家能更方便的备战面试。如侵犯到作者权益，可联系我删除。

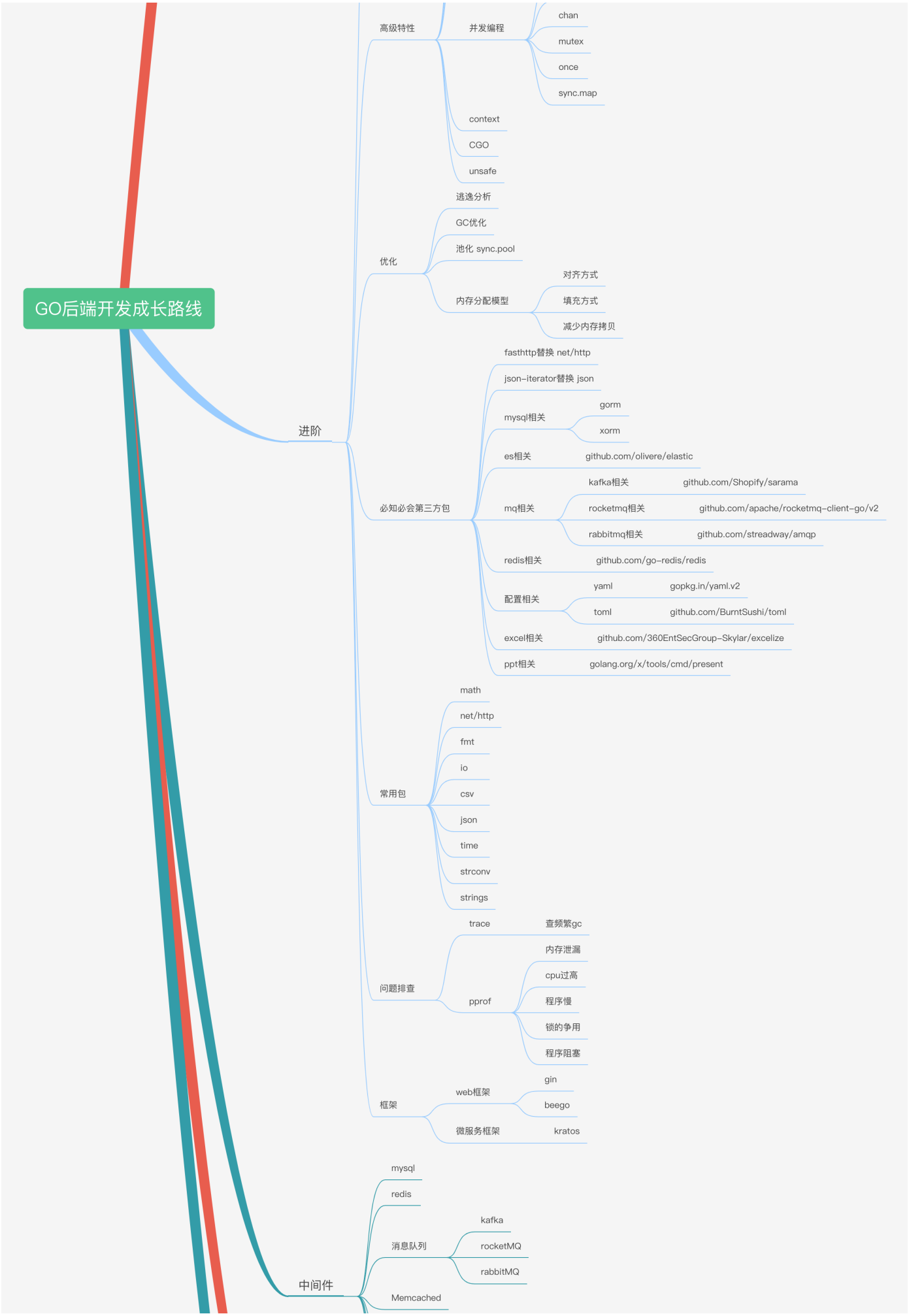
更多资源获取

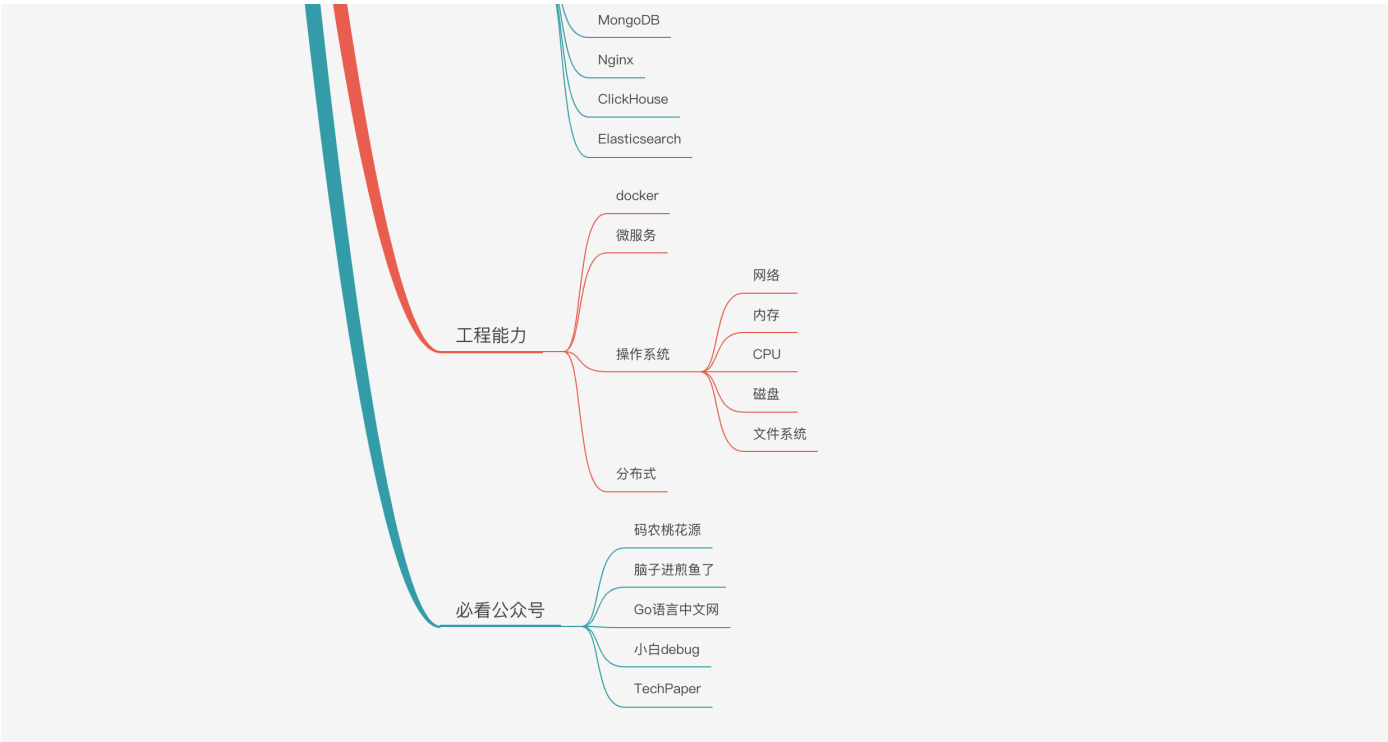
- 欢迎大家访问我的**博客** https://xiaobaidebug.top/ 。
- 在公众号内回复**【面试】**，可以获得整理的**面试PDF电子书**（含golang），适合快速备战面试查漏补缺。
- 在公众号内回复**【后端面试】**，可以获得整理的**面试PDF电子书**（不含golang），适合快速备战面试查漏补缺。
- 在公众号内回复**【网络】**，可以获得整理的**图解PDF电子书，图文并茂，通俗易懂**，非常适合巩固基础和进

阶。

- 在公众号内回复【**视频**】，可以获得整理的**视频教程**，内含**实战教程**，适合小白上路。




GO后端开发成长路线
进阶
高级特性
并发编程
chan
mutex
once
sync.map
context
CGO
unsafe
优化
逃逸分析
GC优化
池化 sync.pool
内存分配模型
对齐方式
填充方式
减少内存拷贝
必知必会第三方包
fasthttp替换 net/http
json-iterator替换 json
mysql相关
gorm
xorm
es相关
github.com/olivere/elastic
mq相关
kafka相关
github.com/Shopify/sarama
rocketmq相关
github.com/apache/rocketmq-client-go/v2
rabbitmq相关
github.com/streadway/amqp
redis相关
github.com/go-redis/redis
配置相关
yaml
gopkg.in/yaml.v2
toml
github.com/BurntSushi/toml
excel相关
github.com/360EntSecGroup-Skylar/excelize
ppt相关
golang.org/x/tools/cmd/present
常用包
math
net/http
fmt
io
csv
json
time
strconv
strings
问题排查
trace
查频繁gc
pprof
内存泄漏
cpu过高
程序慢
锁的争用
程序阻塞
框架
web框架
gin
beego
微服务框架
kratos
中间件
mysql
redis
消息队列
kafka
rocketMQ
rabbitMQ
Memcached



脑图持续不断更新中，在线查看地址
后续文章和内容会不断更新到 github项目 https://github.com/xiaobaiTech/golangFamily 中，欢迎关注。

目录

golang面试题：reflect（反射包）如何获取字段tag？为什么json包不能导出私有变量的tag?
昨天那个在for循环里append元素的同事，今天还在么?
go struct能不能比较
Go 支持什么形式的类型转换？将整数转换为浮点数。
Log包线程安全吗?
Goroutine和线程的区别?
下列哪一行会panic?
下列哪行代码会panic?
什么是 Goroutine？你如何停止它?
Golang中除了加Mutex锁以外还有哪些方式安全读写共享变量?
无缓冲 Chan 的发送和接收是否同步?
关于switch语句，下面说法正确的有?
下列Add函数定义正确的是?
关于 bool 变量 b 的赋值，下面错误的用法是?
关于变量的自增和自减操作，下面语句正确的是?
go语言的并发机制以及它所使用的CSP并发模型.
Golang 中常用的并发模型?
JSON 标准库对 nil slice 和 空 slice 的处理是一致的吗?
协程，线程，进程的区别。
关于协程，下列说法正确的有?
互斥锁，读写锁，死锁问题是怎么解决。
Golang的内存模型，为什么小对象多了会造成gc压力。
说下Go中的锁有哪些?三种锁，读写锁，互斥锁，还有map的安全的锁?
什么是channel，为什么它可以做到线程安全?
读写锁或者互斥锁读的时候能写吗?
怎么限制Goroutine的数量.
Channel是同步的还是异步的.
下列哪个类型可以使用 cap()函数?
Data Race问题怎么解决？能不能不加锁解决这个问题?
如何在运行时检查变量类型?
Go 两个接口之间可以存在什么关系?
关于map，下面说法正确的是?
关于同步锁，下面说法正确的是?
Go 当中同步锁有什么特点？作用是什么
Go 语言当中 Channel（通道）有什么特点，需要注意什么?
Go 语言当中 Channel 缓冲有什么特点?
关于channel，下面语法正确的是?
Go 语言中 cap 函数可以作用于那些内容?
go convey 是什么？一般用来做什么?
Go 语言当中 new 和 make 有什么区别吗?
Go 语言中 make 的作用是什么?
Printf(),Sprintf(),FprintF()都是格式化输出，有什么不同?
Go 语言当中数组和切片的区别是什么?
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
Go 语言当中数组和切片在传递的时候的区别是什么?
Go 语言是如何实现切片扩容的?
关于 channel 下面描述正确的是?
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?
Golang Slice 的底层实现

Go 进阶

- 解释一下 Dockerfile 的 ONBUILD 指令?
- 什么是 Docker Swarm?
- 如何在生产中监控 Docker?
- DevOps 有哪些优势?
- CI 服务有什么用途?
- 如何使用 Docker 技术创建与环境无关的容器系统?
- Dockerfile 配置文件中的 COPY 和 ADD 指令有什么不同?
- Docker 映像（image）是什么?
- Docker 容器（container）是什么?
- Docker 中心（hub）什么概念?
- 在任意给定时间点指出一个 Docker 容器可能存在的运行阶段?
- 有什么方法确定一个 Docker 容器运行状态?
- 在 Dockerfile 配置文件中最常用的指令有哪些?
- 什么类型的应用（无状态性或有状态性）更适合 Docker 容器技术?
- 解释基本 Docker 应用流程
- Docker Image 和 Docker Layer (层)有什么不同?
- 虚拟化技术是什么?
- 虚拟管理层（程序）是什么?
- Docker 群（Swarm）是什么?
- 在使用 Docker 技术的产品中如何监控其运行?
- 什么是孤儿卷及如何删除它?
- 什么是半虚拟化（Paravirtualization）?
- Docker 技术与虚拟机技术有何不同?
- 请解释一下 docerfile 配置文件中的 ONBUILD 指令的用途含义?
- 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?
- 在 Windows 系统上可以运行原生的 Docker 容器吗?
- 在非 Linux 操作系统平台上如何运行 Docker ?
- 容器化技术在底层的运行原理?
- 说说容器化技术与虚拟化技术的优缺点
- 如何使 Docker 适应多种运行环境?
- 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?

Redis

- 什么是 Redis?
- Redis 的数据类型?
- 使用 Redis 有哪些好处?
- Redis 相比 Memcached 有哪些优势?
- Memcache 与 Redis 的区别都有哪些?
- Redis 是单进程单线程的?
- 一个字符串类型的值能存储最大容量是多少?
- Redis 集群最大节点个数是多少?
- Reids 的特点
- 使用 Redis 有哪些好处?
- 为什么 edis 需要把所有数据放到内存中?
- Redis 的内存用完了会发生什么?
- Redis 的回收策略（淘汰策略）
- Redis 的持久化机制是什么？各自的优缺点?
- Redis 常见性能问题和解决方案：
- Redis 过期键的删除策略?
- Redis 的回收策略（淘汰策略）?

什么是交换空间?
什么是 Root 帐户
什么是 LILO?
什么是 BASH?
什么是 CLI?
什么是 GUI?
开源的优势是什么?
GNU 项目的重要性是什么?
绝对路径用什么符号表示? 当前目录、上层目录用什么表示? 主目录用什么表示? 切换目录用什么命令?
怎么查看当前进程? 怎么执行退出? 怎么查看当前路径?
怎么清屏? 怎么退出当前命令? 怎么执行睡眠? 怎么查看当前用户 id? 查看指定帮助用什么命令?
Ls 命令执行什么功能? 可以带哪些参数，有什么区别?
建立软链接(快捷方式)，以及硬链接的命令。
目录创建用什么命令? 创建文件用什么命令? 复制文件用什么命令?
查看文件内容有哪些命令可以使用?
随意写文件命令? 怎么向屏幕输出带空格的字符串，比如”hello world”?
终端是哪个文件夹下的哪个文件? 黑洞文件是哪个文件夹下的哪个命令?
移动文件用哪个命令? 改名用哪个命令?
复制文件用哪个命令? 如果需要连同文件夹一块复制呢? 如果需要有提示功能呢?
删除文件用哪个命令? 如果需要连目录及目录下文件一块删除呢? 删除空文件夹用什么命令?
Linux 下命令有哪几种可使用的通配符? 分别代表什么含义?
用什么命令对一个文件的内容进行统计? (行号、单词数、字节数)
Grep 命令有什么用? 如何忽略大小写? 如何查找不含该串的行?
Linux 中进程有哪几种状态? 在 ps 显示出来的信息中，分别用什么符号表示的?
怎么使一个命令在后台运行?
利用 ps 怎么显示所有的进程? 怎么利用 ps 查看指定进程的信息?
哪个命令专门用来查看后台任务?
把后台任务调到前台执行使用什么命令?把停下的后台任务在后台执行起来用什么命令?
终止进程用什么命令? 带什么参数?
怎么查看系统支持的所有信号?
搜索文件用什么命令? 格式是怎么样的?
查看当前谁在使用该主机用什么命令? 查找自己所在的终端信息用什么命令?
使用什么命令查看用过的命令列表?
使用什么命令查看磁盘使用空间? 空闲空间呢?
使用什么命令查看网络是否连通?
使用什么命令查看 ip 地址及接口信息?
查看各类环境变量用什么命令?
通过什么命令指定命令提示符?
查找命令的可执行文件是去哪查找的? 怎么对其进行设置及添加?
通过什么命令查找执行命令?
怎么对命令进行取别名?
du 和 df 的定义，以及区别?
awk 详解。
当你需要给命令绑定一个宏或者按键的时候，应该怎么做呢?
如果一个linux新手想要知道当前系统支持的所有命令的列表，他需要怎么做?
如果你的助手想要打印出当前的目录栈，你会建议他怎么做?
你的系统目前有许多正在运行的任务，在不重启机器的条件下，有什么方法可以把所有正在运行的进程移除呢?
bash shell 中的hash 命令有什么作用?
哪一个bash内置命令能够进行数学运算。

怎样一页一页地查看一个大文件的内容呢?
数据字典属于哪一个用户的?
怎样查看一个linux命令的概要与用法?假设你在/bin目录中偶然看到一个你从没见过的的命令，怎样才能知道它的作用和用法呢?
使用哪一个命令可以查看自己文件系统的磁盘空间配额呢?
说一下异步和非阻塞的区别?
滑动窗口的概念以及应用?
Epoll原理.
负载均衡原理是什么?
LVS相关了解.

网络和操作系统

进程和线程的区别?
协程与线程的区别?
并发和并行有什么区别?
进程与线程的切换流程?
为什么虚拟地址空间切换会比较耗时?
进程间通信方式有哪些?
进程间同步的方式有哪些?
线程同步的方式有哪些?
线程的分类?
什么是临界区，如何解决冲突?
什么是死锁?死锁产生的条件?
进程调度策略有哪几种?
进程有哪些状态?
什么是分页?
什么是分段?
分页和分段有什区别?
什么是交换空间?
页面替换算法有哪些?
什么是缓冲区溢出?有什么危害?
什么是虚拟内存?
讲一讲 IO 多路复用?
硬链接和软链接有什么区别?
中断的处理过程?
中断和轮询有什么区别?
请详细介绍一下 TCP 的三次握手机制，为什么要三次握手?
讲一讲 SYN 超时，洪泛攻击，以及解决策略
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?

RocketMQ 面试题

多个 MQ 如何选型?
为什么要使用 MQ?
RocketMQ 由哪些角色组成，每个角色作用和特点是什么?
RocketMQ 中的 Topic 和 JMS 的 queue 有什么区别?
RocketMQ 消费模式有几种?
Broker 如何处理拉取请求的?
RocketMQ 如何做负载均衡?
消息重复消费
RocketMQ 如何保证消息不丢失

Go 入门

与其他语言相比，使用 Go 有什么好处?

- 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。
- Golang 针对并发进行了优化，并且在规模上运行良好。
- 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。
- 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?

Golang 使用以下类型：

- Method
- Boolean
- Numeric
- String
- Array
- Slice
- Struct
- Pointer
- Function
- Interface
- Map
- Channel

Golang开发新手常犯的50个错误

https://blog.csdn.net/gezhonglei2007/article/details/52237582

关于整型切片的初始化，下面正确的是?

A. s := make([]int)

B. s := make([]int, 0)

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

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

答案

> B C D

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)
    }
}
```

答案

> 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

关于channel的特性，下面说法正确的是?

A. 给一个 nil channel 发送数据，造成永远阻塞

B. 从一个 nil channel 接收数据，造成永远阻塞

C. 给一个已经关闭的 channel 发送数据，引起 panic

D. 从一个已经关闭的 channel 接收数据，如果缓冲区中为空，则返回一个零值

答案

> A B C D

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)
}
```

答案

> Go语言中，常量无法寻址, 是不能进行取指针操作的

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)
    }
}
```

答案

> 012
>
> range 一个返回值时，这个值是下标，两个值时，第一个是下标，第二个是值，当 x 为 map时，第一个是key，第二个是value

关于无缓冲和有冲突的channel，下面说法正确的是?

A. 无缓冲的channel是默认的缓冲为1的channel；

B. 无缓冲的channel和有缓冲的channel都是同步的；

C. 无缓冲的channel和有缓冲的channel都是非同步的；

D. 无缓冲的channel是同步的，而有缓冲的channel是非同步的；

答案

> D

下列代码输出什么?

```
func Foo(x interface{}) {
    if x == nil {
        fmt.Println("empty interface")
        return
    }
    fmt.Println("non-empty interface")
}
func Test64(t *testing.T) {
    var x *int = nil
    Foo(x)
}
```

答案

> non-empty interface
>
> 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

关于select机制，下面说法正确的是?

A. select机制用来处理异步IO问题；

B. select机制最大的一条限制就是每个case语句里必须是一个IO操作；

C. golang在语言级别支持select关键字；

D. select关键字的用法与switch语句非常类似，后面要带判断条件；

答案

> A B C

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以下命令声明：

```
package <packagename>
```

您可以使用以下方法导入和导出包以重用导出的函数或类型：

```
import <packagename>
```

Golang 的标准包是 fmt，其中包含格式化和打印功能，如 Println().

关于字符串拼接,下列正确的是?

A. str := 'abc' + '123'

B. str := "abc" + "123"

C. str： = '123' + "abc"

D. fmt.Sprintf("abc%d", 123)

答案

> B D 双引号用来表示字符串 string，其实质是一个 byte 类型的数组，单引号表示 rune 类型。

连nil切片和空切片一不一样都不清楚？那BAT面试官只好让你回去等通知了。

https://mp.weixin.qq.com/s/sW4PD1MiaunURNDIU4BbQQ

golang面试题：字符串转成byte数组，会发生内存拷贝吗?

https://mp.weixin.qq.com/s/qmlPuGVISx8NYp2b9LrqnA

golang面试题：翻转含有中文、数字、英文字母的字符串

https://mp.weixin.qq.com/s/ssinnUM22PHPWRug8EzAkg

golang面试题：拷贝大切片一定比小切片代价大吗?

https://mp.weixin.qq.com/s/8Dp2eCYzDdBbxAG5-jNevQ

golang面试题：json包变量不加tag会怎么样?

https://mp.weixin.qq.com/s/bZlKV_BWSqc-qCa4DrsCbg

golang面试题：reflect（反射包）如何获取字段tag？为什么json包不能导出私有变量的tag?

https://mp.weixin.qq.com/s/P7TEx2mInwEktXTEE6JDWQ

昨天那个在for循环里append元素的同事，今天还在么？

https://mp.weixin.qq.com/s/SHxcspmiKyPwPBbhfVxsGA

go struct能不能比较

- 相同struct类型的可以比较
- 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)
```

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
```

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。

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也不罕见（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。

下列哪一行会panic?

```
func Test76(t *testing.T) {
    var x interface{}
    var y interface{} = []int{3, 5}
    _ = x == x
    _ = x == y
    _ = y == y
}
```

答案

> _ = y == y 会发生panic, 因为两个比较值的动态类型为同一个不可比较类型

下列哪行代码会panic?

```
func Test77(t *testing.T) {
    x := make([]int, 2, 10)
    _ = x[6:10]
    _ = x[6:]
    _ = x[2:]
}
```

答案

> _ = x[6:] 这一行会发生panic, 截取符号 [i:j],
> 如果 j 省略，默认是原切片或者数组的长度，x 的长度是 2，小于起始下标 6 ，所以 panic

什么是 Goroutine？你如何停止它?

一个 Goroutine 是一个函数或方法执行同时旁边其他任何够程采用了特殊的 Goroutine 线程。Goroutine 线程比标准线程更轻量级，大多数 Golang 程序同时使用数千个 g、Goroutine。

要创建 Goroutine，请 go 在函数声明之前添加关键字。

```
go f(x, y, z)
```

您可以通过向 Goroutine 发送一个信号通道来停止它。Goroutines 只能在被告知检查时响应信号，因此您需要在逻辑位置（例如 for 循环顶部）包含检查。

```
package main

func main(){

  quit := make(chan bool) go func(){
```

```
    for {
      select {
      case <-quit:
        return default:
        //…
      }
      }
  }()
  //…
  quit <- true
}
```

Golang中除了加Mutex锁以外还有哪些方式安全读写共享变量?

Golang中Goroutine 可以通过 Channel 进行安全读写共享变量。

无缓冲 Chan 的发送和接收是否同步?

```
ch := make(chan int)    无缓冲的channel由于没有缓冲发送和接收需要同步.
ch := make(chan int, 2) 有缓冲channel不要求发送和接收操作同步.
```

- channel无缓冲时，发送阻塞直到数据被接收，接收阻塞直到读到数据。
- channel有缓冲时，当缓冲满时发送阻塞，当缓冲空时接收阻塞。

关于switch语句，下面说法正确的有?

A. 条件表达式必须为常量或者整数；

B. 单个case中，可以出现多个结果选项；

C. 需要用break来明确退出一个case；

D. 只有在case中明确添加fallthrough关键字，才会继续执行紧跟的下一个case；

答案

> B D

下列Add函数定义正确的是?

```
func Test54(t *testing.T) {
    var a Integer = 1
    var b Integer = 2
    var i interface{} = &a
    sum := i.(*Integer).Add(b)
    fmt.Println(sum)
}
A.
type Integer int
func (a Integer) Add(b Integer) Integer {
        return a + b
}

B.
type Integer int
func (a Integer) Add(b *Integer) Integer {
        return a + *b
}

C.
type Integer int
func (a *Integer) Add(b Integer) Integer {
        return *a + b
}

D.
type Integer int
func (a *Integer) Add(b *Integer) Integer {
        return *a + *b
}
```

答案

> A C

关于 bool 变量 b 的赋值，下面错误的用法是?

A. b = true

B. b = 1

C. b = bool(1)

D. b = (1 == 2)

答案

> B C

关于变量的自增和自减操作，下面语句正确的是?

A.

```
i := 1
i++
```

B.

```
i := 1
j = i++
```

C.

```
i := 1
++i
```

D.

```
i := 1
i--
```

答案

> A D
>
> go 里面没有 ++i 和 --i

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：

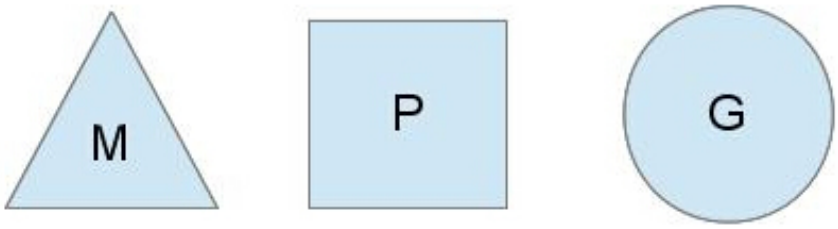
- 用户空间 避免了内核态和用户态的切换导致的成本。
- 可以由语言和框架层进行调度。
- 更小的栈空间允许创建大量的实例。

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.

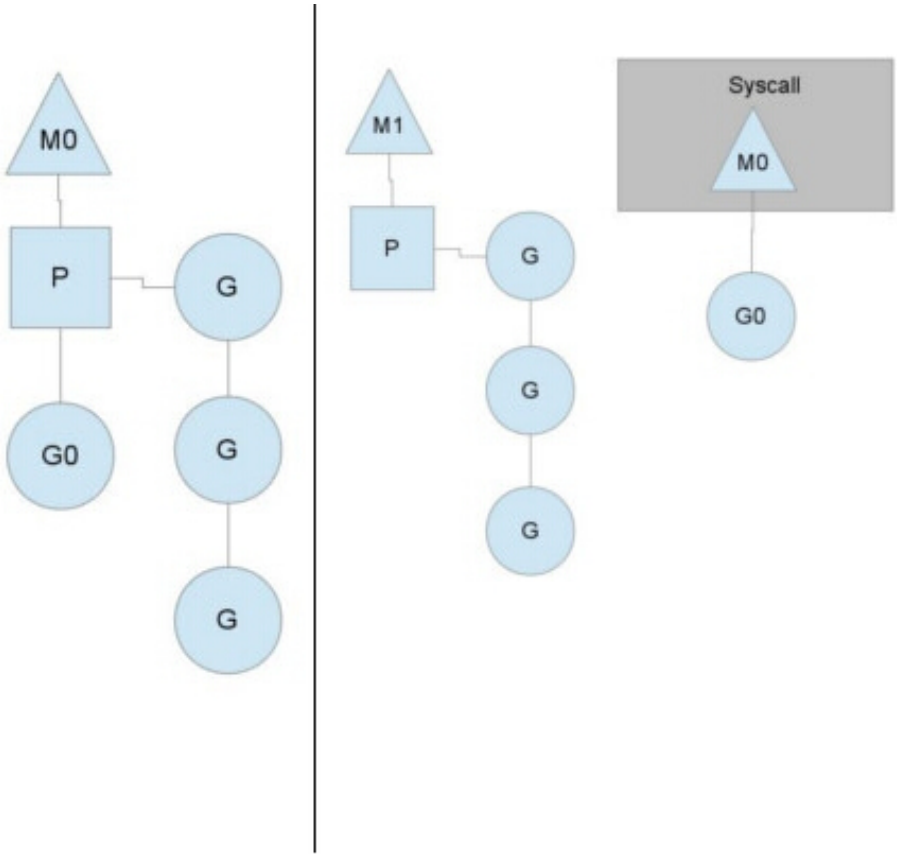
M（Machine）：对内核级线程的封装，数量对应真实的CPU数（真正干活的对象）.

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 中常用的并发模型?

Golang 中常用的并发模型有三种:

- 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

- 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:

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

```
func main(){
    var wg sync.WaitGroup
    var urls = []string{
        "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，不能被拷贝

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 给拷贝传递到了 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

- 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。

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

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

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

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

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

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

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

当我们查询或者处理一个空的列表的时候，这非常有用，它会告诉我们返回的是一个列表，但是列表内没有任何值。

总之，nil slice 和 empty slice是不同的东西,需要我们加以区分的.

协程，线程，进程的区别。

- 进程

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位。每个进程都有自己的独立内存空间，不同进程通过进程间通信来通信。由于进程比较重量，占据独立的内存，所以上下文进程间的切换开销（栈、寄存器、虚拟内存、文件句柄等）比较大，但相对比较稳定安全。

- 线程

线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。线程间通信主要通过共享内存，上下文切换很快，资源开销较少，但相比进程不够稳定容易丢失数据。

- 协程

协程是一种用户态的轻量级线程，协程的调度完全由用户控制。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，直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快。

关于协程，下列说法正确的有?

A. 协程和线程都可以实现程序的并发执行；

B. 线程比协程更轻量级；

C. 协程不存在死锁问题；

D. 通过 channel 来进行协程间的通信；

答案

> A D

互斥锁，读写锁，死锁问题是怎么解决。

- 互斥锁

互斥锁就是互斥变量mutex，用来锁住临界区的.

条件锁就是条件变量，当进程的某些资源要求不满足时就进入休眠，也就是锁住了。当资源被分配到了，条件锁打开，进程继续运行；读写锁，也类似，用于缓冲区等临界资源能互斥访问的。

- 读写锁

通常有些公共数据修改的机会很少，但其读的机会很多。并且在读的过程中会伴随着查找，给这种代码加锁会降低我们的程序效率。读写锁可以解决这个问题。

读写锁的行为

当前锁的状态	读锁请求	写锁请求
无锁	可以	可以
读锁	可以	阻塞
写锁	阻塞	阻塞

注意：写独占，读共享，写锁优先级高

- 死锁

一般情况下，如果同一个线程先后两次调用lock，在第二次调用时，由于锁已经被占用，该线程会挂起等待别的线程释放锁，然而锁正是被自己占用着的，该线程又被挂起而没有机会释放锁，因此就永远处于挂起等待状态了，这叫做死锁（Deadlock）。 另外一种情况是：若线程A获得了锁1，线程B获得了锁2，这时线程A调用lock试图获得锁2，结果是需要挂起等待线程B释放锁2，而这时线程B也调用lock试图获得锁1，结果是需要挂起等待线程A释放锁1，于是线程A和B都永远处于挂起状态了。

死锁产生的四个必要条件:

1. 互斥条件：一个资源每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，对已获得的资源保持不放。
3. 不剥夺条件:进程已获得的资源，在未使用完之前，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循环等待资源关系。 这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

a. 预防死锁

可以把资源一次性分配：（破坏请求和保持条件）

然后剥夺资源：即当某进程新的资源未满足时，释放已占有的资源（破坏不可剥夺条件）

资源有序分配法：系统给每类资源赋予一个编号，每一个进程按编号递增的顺序请求资源，释放则相反（破坏环路等待条件）

b. 避免死锁

预防死锁的几种策略，会严重地损害系统性能。因此在避免死锁时，要施加较弱的限制，从而获得 较满意的系统性能。由于在避免死锁的策略中，允许进程动态地申请资源。因而，系统在进行资源分配之前预先计算资源分配的安全性。若此次分配不会导致系统进入不安全状态，则将资源分配给进程；否则，进程等待。其中最具有代表性的避免死锁算法是银行家算法。

c. 检测死锁

首先为每个进程和每个资源指定一个唯一的号码,然后建立资源分配表和进程等待表.

d. 解除死锁

当发现有进程死锁后，便应立即把它从死锁状态中解脱出来，常采用的方法有.

e. 剥夺资源

从其它进程剥夺足够数量的资源给死锁进程，以解除死锁状态.

f. 撤消进程

可以直接撤消死锁进程或撤消代价最小的进程，直至有足够的资源可用，死锁状态.消除为止.所谓代价是指优先级、运行代价、进程的重要性和价值等。

Golang的内存模型，为什么小对象多了会造成gc压力。

通常小对象过多会导致GC三色法消耗过多的GPU。优化思路是，减少对象分配.

说下Go中的锁有哪些?三种锁，读写锁，互斥锁，还有map的安全的锁?

Go中的三种锁包括:互斥锁,读写锁,sync.Map的安全的锁.

- 互斥锁

Go并发程序对共享资源进行访问控制的主要手段，由标准库代码包中sync中的Mutex结构体表示。

```
//Mutex 是互斥锁， 零值是解锁的互斥锁， 首次使用后不得复制互斥锁。
type Mutex struct {
   state int32
   sema  uint32
}
```

sync.Mutex包中的类型只有两个公开的指针方法Lock和Unlock。

```
//Locker表示可以锁定和解锁的对象。
type Locker interface {
   Lock()
   Unlock()
}

//锁定当前的互斥量
//如果锁已被使用，则调用goroutine
//阻塞直到互斥锁可用。
func (m *Mutex) Lock()

//对当前互斥量进行解锁
//如果在进入解锁时未锁定m，则为运行时错误。
//锁定的互斥锁与特定的goroutine无关。
//允许一个goroutine锁定Mutex然后安排另一个goroutine来解锁它。
func (m *Mutex) Unlock()
```

声明一个互斥锁：

```
var mutex sync.Mutex
```

不像C或Java的锁类工具，我们可能会犯一个错误：忘记及时解开已被锁住的锁，从而导致流程异常。但Go由于存在defer，所以此类问题出现的概率极低。关于defer解锁的方式如下：

```
var mutex sync.Mutex
func Write()  {
   mutex.Lock()
   defer mutex.Unlock()
}
```

如果对一个已经上锁的对象再次上锁，那么就会导致该锁定操作被阻塞，直到该互斥锁回到被解锁状态.

```
fpackage main

import (
    "fmt"
    "sync"
    "time"
)

func main() {

    var mutex sync.Mutex
    fmt.Println("begin lock")
    mutex.Lock()
    fmt.Println("get locked")
    for i := 1; i <= 3; i++ {
        go func(i int) {
            fmt.Println("begin lock ", i)
            mutex.Lock()
            fmt.Println("get locked ", i)
        }(i)
    }

    time.Sleep(time.Second)
    fmt.Println("Unlock the lock")
    mutex.Unlock()
    fmt.Println("get unlocked")
    time.Sleep(time.Second)
}
```

我们在for循环之前开始加锁，然后在每一次循环中创建一个协程，并对其加锁，但是由于之前已经加锁了，所以这个for循环中的加锁会陷入阻塞直到main中的锁被解锁， time.Sleep(time.Second) 是为了能让系统有足够的时间运行for循环，输出结果如下：

```
> go run mutex.go
begin lock
get locked
begin lock  3
begin lock  1
begin lock  2
Unlock the lock
get unlocked
get locked  3
```

这里可以看到解锁后，三个协程会重新抢夺互斥锁权，最终协程3获胜。

互斥锁锁定操作的逆操作并不会导致协程阻塞，但是有可能导致引发一个无法恢复的运行时的panic，比如对一个未锁定的互斥锁进行解锁时就会发生panic。避免这种情况的最有效方式就是使用defer。

我们知道如果遇到panic，可以使用recover方法进行恢复，但是如果对重复解锁互斥锁引发的panic却是无用的（Go 1.8及以后）。

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    defer func() {
        fmt.Println("Try to recover the panic")
        if p := recover(); p != nil {
            fmt.Println("recover the panic : ", p)
        }
    }()
    var mutex sync.Mutex
    fmt.Println("begin lock")
    mutex.Lock()
    fmt.Println("get locked")
    fmt.Println("unlock lock")
    mutex.Unlock()
    fmt.Println("lock is unlocked")
    fmt.Println("unlock lock again")
    mutex.Unlock()
}
```

运行:

```
> go run mutex.go
begin lock
get locked
unlock lock
```

```
lock is unlocked
unlock lock again
fatal error: sync: unlock of unlocked mutex

goroutine 1 [running]:
runtime.throw(0x4bc1a8, 0x1e)
        /home/keke/soft/go/src/runtime/panic.go:617 +0x72 fp=0xc000084ea8
sp=0xc000084e78 pc=0x427ba2
sync.throw(0x4bc1a8, 0x1e)
        /home/keke/soft/go/src/runtime/panic.go:603 +0x35 fp=0xc000084ec8
sp=0xc000084ea8 pc=0x427b25
sync.(*Mutex).Unlock(0xc00001a0c8)
        /home/keke/soft/go/src/sync/mutex.go:184 +0xc1 fp=0xc000084ef0 sp=0xc000084ec8
pc=0x45f821
main.main()
        /home/keke/go/Test/mutex.go:25 +0x25f fp=0xc000084f98 sp=0xc000084ef0
pc=0x486c1f
runtime.main()
        /home/keke/soft/go/src/runtime/proc.go:200 +0x20c fp=0xc000084fe0
sp=0xc000084f98 pc=0x4294ec
runtime.goexit()
        /home/keke/soft/go/src/runtime/asm_amd64.s:1337 +0x1 fp=0xc000084fe8
sp=0xc000084fe0 pc=0x450ad1
exit status 2
```

这里试图对重复解锁引发的panic进行recover，但是我们发现操作失败，虽然互斥锁可以被多个协程共享，但还是建议将对同一个互斥锁的加锁解锁操作放在同一个层次的代码中。

- 读写锁

读写锁是针对读写操作的互斥锁，可以分别针对读操作与写操作进行锁定和解锁操作 。

读写锁的访问控制规则如下：

① 多个写操作之间是互斥的 ② 写操作与读操作之间也是互斥的 ③ 多个读操作之间不是互斥的

在这样的控制规则下，读写锁可以大大降低性能损耗。

在Go的标准库代码包中sync中的RWMutex结构体表示为:

```
// RWMutex是一个读/写互斥锁，可以由任意数量的读操作或单个写操作持有。
// RWMutex的零值是未锁定的互斥锁。
//首次使用后，不得复制RWMutex。
//如果goroutine持有RWMutex进行读取而另一个goroutine可能会调用Lock，那么在释放初始读锁之前，
goroutine不应该期望能够获取读锁定。
//特别是，这种禁止递归读锁定。 这是为了确保锁最终变得可用；阻止的锁定会阻止新读操作获取锁定。
type RWMutex struct {
   w           Mutex  //如果有待处理的写操作就持有
   writerSem   uint32 // 写操作等待读操作完成的信号量
   readerSem   uint32 //读操作等待写操作完成的信号量
   readerCount int32  // 待处理的读操作数量
   readerWait  int32  // number of departing readers
}
```

sync中的RWMutex有以下几种方法：

```
//对读操作的锁定
func (rw *RWMutex) RLock()
//对读操作的解锁
func (rw *RWMutex) RUnlock()
//对写操作的锁定
func (rw *RWMutex) Lock()
//对写操作的解锁
func (rw *RWMutex) Unlock()

//返回一个实现了sync.Locker接口类型的值，实际上是回调rw.RLock and rw.RUnlock.
func (rw *RWMutex) RLocker() Locker
```

Unlock方法会试图唤醒所有想进行读锁定而被阻塞的协程，而 RUnlock方法只会在已无任何读锁定的情况下，试图唤醒一个因欲进行写锁定而被阻塞的协程。若对一个未被写锁定的读写锁进行写解锁，就会引发一个不可恢复的panic，同理对一个未被读锁定的读写锁进行读写锁也会如此。

由于读写锁控制下的多个读操作之间不是互斥的，因此对于读解锁更容易被忽视。对于同一个读写锁，添加多少个读锁定，就必要有等量的读解锁，这样才能其他协程有机会进行操作。

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    var rwm sync.RWMutex
    for i := 0; i < 5; i++ {
        go func(i int) {
            fmt.Println("try to lock read ", i)
```

```
            rwm.RLock()
            fmt.Println("get locked ", i)
            time.Sleep(time.Second * 2)
            fmt.Println("try to unlock for reading ", i)
            rwm.RUnlock()
            fmt.Println("unlocked for reading ", i)
        }(i)
    }
    time.Sleep(time.Millisecond * 1000)
    fmt.Println("try to lock for writing")
    rwm.Lock()
    fmt.Println("locked for writing")
}
```

运行:

```
> go run rwmutex.go
try to lock read  0
get locked  0
try to lock read  4
get locked  4
try to lock read  3
get locked  3
try to lock read  1
get locked  1
try to lock read  2
get locked  2
try to lock for writing
try to unlock for reading  0
unlocked for reading  0
try to unlock for reading  2
unlocked for reading  2
try to unlock for reading  1
unlocked for reading  1
try to unlock for reading  3
unlocked for reading  3
try to unlock for reading  4
unlocked for reading  4
locked for writing
```

这里可以看到创建了五个协程用于对读写锁的读锁定与读解锁操作。在 rwm.Lock()种会对main中协程进行写锁定，但是for循环中的读解锁尚未完成，因此会造成mian中的协程阻塞。当for循环中的读解锁操作都完成后就会试图唤醒main中阻塞的协程，main中的写锁定才会完成。

- sync.Map安全锁

golang中的sync.Map是并发安全的，其实也就是sync包中golang自定义的一个名叫Map的结构体。

应用示例:

```
package main
import (
    "sync"
    "fmt"
)

func main() {
    //开箱即用
    var sm sync.Map
    //store 方法,添加元素
    sm.Store(1,"a")
    //Load 方法，获得value
    if v,ok:=sm.Load(1);ok{
        fmt.Println(v)
    }
    //LoadOrStore方法，获取或者保存
    //参数是一对key：value，如果该key存在且没有被标记删除则返回原先的value（不更新）和true；不存在则store，返回该value 和false
    if vv,ok:=sm.LoadOrStore(1,"c");ok{
        fmt.Println(vv)
    }
    if vv,ok:=sm.LoadOrStore(2,"c");!ok{
        fmt.Println(vv)
    }
    //遍历该map，参数是个函数，该函数参的两个参数是遍历获得的key和value，返回一个bool值，当返回false时，遍历立刻结束。
    sm.Range(func(k,v interface{})bool{
        fmt.Print(k)
        fmt.Print(":")
        fmt.Print(v)
        fmt.Println()
        return true
    })
}
```

运行：

```
a
a
c
1:a
2:c
```

sync.Map的数据结构:

```
 type Map struct {
    // 该锁用来保护dirty
    mu Mutex
    // 存读的数据，因为是atomic.value类型，只读类型，所以它的读是并发安全的
    read atomic.Value // readOnly
    //包含最新的写入的数据，并且在写的时候，会把read 中未被删除的数据拷贝到该dirty中，因为是普通的map存在并发安全问题，需要用到上面的mu字段。
    dirty map[interface{}]*entry
    // 从read读数据的时候，会将该字段+1，当等于len（dirty）的时候，会将dirty拷贝到read中（从而提升读的性能）。
    misses int
}
```

read的数据结构是：

```
type readOnly struct {
    m  map[interface{}]*entry
    // 如果Map.dirty的数据和m 中的数据不一样是为true
    amended bool
}
```

entry的数据结构：

```
type entry struct {
    //可见value是个指针类型，虽然read和dirty存在冗余情况（amended=false），但是由于是指针类型，存储的空间应该不是问题
    p unsafe.Pointer // *interface{}
}
```

Delete 方法:

```
func (m *Map) Delete(key interface{}) {
    read, _ := m.read.Load().(readOnly)
    e, ok := read.m[key]
    //如果read中没有，并且dirty中有新元素，那么就去dirty中去找
    if !ok && read.amended {
        m.mu.Lock()
        //这是双检查（上面的if判断和锁不是一个原子性操作）
        read, _ = m.read.Load().(readOnly)
        e, ok = read.m[key]
        if !ok && read.amended {
            //直接删除
            delete(m.dirty, key)
        }
        m.mu.Unlock()
    }
    if ok {
```

```
    //如果read中存在该key，则将该value 赋值nil（采用标记的方式删除！）
        e.delete()
    }
}

func (e *entry) delete() (hadValue bool) {
    for {
        p := atomic.LoadPointer(&e.p)
        if p == nil || p == expunged {
            return false
        }
        if atomic.CompareAndSwapPointer(&e.p, p, nil) {
            return true
        }
    }
}
```

Store 方法:

```
func (m *Map) Store(key, value interface{}) {
    // 如果m.read存在这个key，并且没有被标记删除，则尝试更新。
    read, _ := m.read.Load().(readOnly)
    if e, ok := read.m[key]; ok && e.tryStore(&value) {
        return
    }
    // 如果read不存在或者已经被标记删除
    m.mu.Lock()
    read, _ = m.read.Load().(readOnly)
    if e, ok := read.m[key]; ok {
    //如果entry被标记expunge，则表明dirty没有key，可添加入dirty，并更新entry
        if e.unexpungeLocked() {
            //加入dirty中
            m.dirty[key] = e
        }
        //更新value值
        e.storeLocked(&value)
        //dirty 存在该key，更新
    } else if e, ok := m.dirty[key]; ok {
        e.storeLocked(&value)
        //read 和dirty都没有，新添加一条
    } else {
     //dirty中没有新的数据，往dirty中增加第一个新键
        if !read.amended {
            //将read中未删除的数据加入到dirty中
            m.dirtyLocked()
            m.read.Store(readOnly{m: read.m, amended: true})
        }
        m.dirty[key] = newEntry(value)
    }
```

```
    m.mu.Unlock()
}

//将read中未删除的数据加入到dirty中
func (m *Map) dirtyLocked() {
    if m.dirty != nil {
        return
    }
    read, _ := m.read.Load().(readOnly)
    m.dirty = make(map[interface{}]*entry, len(read.m))
    //read如果较大的话，可能影响性能
    for k, e := range read.m {
    //通过此次操作，dirty中的元素都是未被删除的，可见expunge的元素不在dirty中
        if !e.tryExpungeLocked() {
            m.dirty[k] = e
        }
    }
}

//判断entry是否被标记删除，并且将标记为nil的entry更新标记为expunge
func (e *entry) tryExpungeLocked() (isExpunged bool) {
    p := atomic.LoadPointer(&e.p)
    for p == nil {
        // 将已经删除标记为nil的数据标记为expunged
        if atomic.CompareAndSwapPointer(&e.p, nil, expunged) {
            return true
        }
        p = atomic.LoadPointer(&e.p)
    }
    return p == expunged
}

//对entry 尝试更新
func (e *entry) tryStore(i *interface{}) bool {
    p := atomic.LoadPointer(&e.p)
    if p == expunged {
        return false
    }
    for {
        if atomic.CompareAndSwapPointer(&e.p, p, unsafe.Pointer(i)) {
            return true
        }
        p = atomic.LoadPointer(&e.p)
        if p == expunged {
            return false
        }
    }
}
```

```
//read里 将标记为expunge的更新为nil
func (e *entry) unexpungeLocked() (wasExpunged bool) {
    return atomic.CompareAndSwapPointer(&e.p, expunged, nil)
}

//更新entry
func (e *entry) storeLocked(i *interface{}) {
    atomic.StorePointer(&e.p, unsafe.Pointer(i))
}
```

因此，每次操作先检查read，因为read 并发安全，性能好些；read不满足，则加锁检查dirty，一旦是新的键值，dirty会被read更新。

Load方法:

Load方法是一个加载方法，查找key。

```
func (m *Map) Load(key interface{}) (value interface{}, ok bool) {
    //因read只读，线程安全，先查看是否满足条件
    read, _ := m.read.Load().(readOnly)
    e, ok := read.m[key]
    //如果read没有，并且dirty有新数据，那从dirty中查找，由于dirty是普通map，线程不安全，这个时候用到互斥锁了
    if !ok && read.amended {
        m.mu.Lock()
        // 双重检查
        read, _ = m.read.Load().(readOnly)
        e, ok = read.m[key]
        // 如果read中还是不存在，并且dirty中有新数据
        if !ok && read.amended {
            e, ok = m.dirty[key]
            // mssLocked () 函数是性能是sync.Map 性能得以保证的重要函数，目的讲有锁的dirty数据，替换到只读线程安全的read里
            m.missLocked()
        }
        m.mu.Unlock()
    }
    if !ok {
        return nil, false
    }
    return e.load()
}

//dirty 提升至read 关键函数，当misses 经过多次因为load之后，大小等于len（dirty）时候，讲dirty替换到read里，以此达到性能提升。
func (m *Map) missLocked() {
    m.misses++
    if m.misses < len(m.dirty) {
        return
    }
```

```
    //原子操作，耗时很小
    m.read.Store(readOnly{m: m.dirty})
    m.dirty = nil
    m.misses = 0
}
```

sync.Map是通过冗余的两个数据结构(read、dirty),实现性能的提升。为了提升性能，load、delete、store等操作尽量使用只读的read；为了提高read的key击中概率，采用动态调整，将dirty数据提升为read；对于数据的删除，采用延迟标记删除法，只有在提升dirty的时候才删除。

什么是channel，为什么它可以做到线程安全?

Channel是Go中的一个核心类型，可以把它看成一个管道，通过它并发核心单元就可以发送或者接收数据进行通讯(communication),Channel也可以理解是一个先进先出的队列，通过管道进行通信。

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

读写锁或者互斥锁读的时候能写吗?

Go中读写锁包括读锁和写锁，多个读线程可以同时访问共享数据；写线程必须等待所有读线程都释放锁以后，才能取得锁；同样的，读线程必须等待写线程释放锁后，才能取得锁，也就是说读写锁要确保的是如下互斥关系，可以同时读，但是读-写，写-写都是互斥的。

怎么限制Goroutine的数量.

在Golang中，Goroutine虽然很好，但是数量太多了，往往会带来很多麻烦，比如耗尽系统资源导致程序崩溃，或者CPU使用率过高导致系统忙不过来。所以我们可以限制下Goroutine的数量,这样就需要在每一次执行go之前判断goroutine的数量，如果数量超了，就要阻塞go的执行。第一时间想到的就是使用通道。每次执行的go之前向通道写入值，直到通道满的时候就阻塞了，

```
package main

import "fmt"

var ch chan  int

func elegance(){
    <-ch
    fmt.Println("the ch value receive",ch)
}

func main(){
    ch = make(chan int,5)
    for i:=0;i<10;i++{
```

```
        ch <-1
        fmt.Println("the ch value send",ch)
        go elegance()
        fmt.Println("the result i",i)
    }

}
```

运行:

```
> go run goroutine.go
the ch value send 0xc00009c000
the result i 0
the ch value send 0xc00009c000
the result i 1
the ch value send 0xc00009c000
the result i 2
the ch value send 0xc00009c000
the result i 3
the ch value send 0xc00009c000
the result i 4
the ch value send 0xc00009c000
the result i 5
the ch value send 0xc00009c000
the ch value receive 0xc00009c000
the result i 6
the ch value receive 0xc00009c000
the ch value send 0xc00009c000
the result i 7
the ch value send 0xc00009c000
the result i 8
the ch value send 0xc00009c000
the result i 9
the ch value send 0xc00009c000
the ch value receive 0xc00009c000
the ch value receive 0xc00009c000
the ch value receive 0xc00009c000
the result i 10
the ch value send 0xc00009c000
the result i 11
the ch value send 0xc00009c000
the result i 12
the ch value send 0xc00009c000
the result i 13
the ch value send 0xc00009c000
the ch value receive 0xc00009c000
the ch value receive 0xc00009c000
the ch value receive 0xc00009c000
the ch value receive 0xc00009c000
```

```
the result i 14
the ch value receive 0xc00009c000
> go run goroutine.go
the ch value send 0xc00007e000
the result i 0
the ch value send 0xc00007e000
the result i 1
the ch value send 0xc00007e000
the result i 2
the ch value send 0xc00007e000
the result i 3
the ch value send 0xc00007e000
the ch value receive 0xc00007e000
the result i 4
the ch value send 0xc00007e000
the ch value receive 0xc00007e000
the result i 5
the ch value send 0xc00007e000
the ch value receive 0xc00007e000
the result i 6
the ch value send 0xc00007e000
the result i 7
the ch value send 0xc00007e000
the ch value receive 0xc00007e000
the ch value receive 0xc00007e000
the ch value receive 0xc00007e000
the result i 8
the ch value send 0xc00007e000
the result i 9
```

这样每次同时运行的goroutine就被限制为5个了。但是新的问题于是就出现了，因为并不是所有的goroutine都执行完了，在main函数退出之后，还有一些goroutine没有执行完就被强制结束了。这个时候我们就需要用到sync.WaitGroup。使用WaitGroup等待所有的goroutine退出。

```
package main

import (
    "fmt"
    "runtime"
    "sync"
    "time"
)
// Pool Goroutine Pool
type Pool struct {
    queue chan int
    wg *sync.WaitGroup
}
// New 新建一个协程池
func NewPool(size int) *Pool{
```

```
    if size <=0{
        size = 1
    }
    return &Pool{
        queue:make(chan int,size),
        wg:&sync.WaitGroup{},
    }
}
// Add 新增一个执行
func (p *Pool)Add(delta int){
    // delta为正数就添加
    for i :=0;i<delta;i++{
        p.queue <-1
    }
    // delta为负数就减少
    for i:=0;i>delta;i--{
        <-p.queue
    }
    p.wg.Add(delta)
}
// Done 执行完成减一
func (p *Pool) Done(){
    <-p.queue
    p.wg.Done()
}
// Wait 等待Goroutine执行完毕
func (p *Pool) Wait(){
    p.wg.Wait()
}

func main(){
    // 这里限制5个并发
    pool := NewPool(5)
    fmt.Println("the NumGoroutine begin is:",runtime.NumGoroutine())
    for i:=0;i<20;i++{
        pool.Add(1)
        go func(i int) {
            time.Sleep(time.Second)
            fmt.Println("the NumGoroutine continue is:",runtime.NumGoroutine())
            pool.Done()
        }(i)
    }
    pool.Wait()
    fmt.Println("the NumGoroutine done is:",runtime.NumGoroutine())
}
```

运行:

```
the NumGoroutine begin is: 1
```

```
the NumGoroutine continue is: 6
the NumGoroutine continue is: 7
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 3
the NumGoroutine continue is: 2
the NumGoroutine done is: 1
```

其中，Go的GOMAXPROCS默认值已经设置为CPU的核数， 这里允许我们的Go程序充分使用机器的每一个CPU,最大程度的提高我们程序的并发性能。runtime.NumGoroutine函数在被调用后，会返回系统中的处于特定状态的Goroutine的数量。这里的特指是指Grunnable\Gruning\Gsyscall\Gwaition。处于这些状态的Groutine即被看做是活跃的或者说正在被调度。

这里需要注意下：垃圾回收所在Groutine的状态也处于这个范围内的话，也会被纳入该计数器。

Channel是同步的还是异步的.

Channel是异步进行的。

channel存在3种状态：

- nil，未初始化的状态，只进行了声明，或者手动赋值为nil
- active，正常的channel，可读或者可写
- closed，已关闭，千万不要误认为关闭channel后，channel的值是nil

下列哪个类型可以使用 cap()函数?

A. array

B. slice

C. map

D. channel

答案

> A B D
> array 返回数组的元素个数；
> slice 返回 slice 的最大容量；
> channel 返回 channel 的容量；

Data Race问题怎么解决？能不能不加锁解决这个问题?

同步访问共享数据是处理数据竞争的一种有效的方法.golang在1.1之后引入了竞争检测机制，可以使用 go run -race 或者 go build -race来进行静态检测。 其在内部的实现是,开启多个协程执行同一个命令， 并且记录下每个变量的状态.

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

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

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

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

如何在运行时检查变量类型?

类型开关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，和一个 defaultcase，如果没有一个 case 为真，则执行。

Go 两个接口之间可以存在什么关系?

如果两个接口有相同的方法列表，那么他们就是等价的，可以相互赋值。如果接口 A的方法列表是接口 B的方法列表的自己，那么接口 B可以赋值给接口 A。接口查询是否成功，要在运行期才能够确定。

关于map，下面说法正确的是?

A. map 反序列化时 json.unmarshal() 的入参必须为 map 的地址；

B. 在函数调用中传递 map，则子函数中对 map 元素的增加不会导致父函数中 map 的修改；

C. 在函数调用中传递 map，则子函数中对 map 元素的修改不会导致父函数中 map 的修改；

D. 不能使用内置函数 delete() 删除 map 的元素

答案

> A

关于同步锁，下面说法正确的是?

A. 当一个 goroutine 获得了 Mutex 后，其他 goroutine 就只能乖乖的等待，除非该 goroutine 释放这个 Mutex；

B. RWMutex 在读锁占用的情况下，会阻止写，但不阻止读；

C. RWMutex 在写锁占用情况下，会阻止任何其他 goroutine（无论读和写）进来，整个锁相当于由该 goroutine 独占；

D. Lock() 操作需要保证有 Unlock() 或 RUnlock() 调用与之对应；

答案

> A B C , 106

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 Gorouline（协程）就只能乖乖的等待，除非该 gorouline 释放了该 MutexRWMutex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?

如果给一个 nil 的 channel 发送数据，会造成永远阻塞如果从一个 nil 的 channel 中接收数据，也会造成永久爱阻塞给一个已经关闭的 channel 发送数据，会引起 pannic 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。

关于channel，下面语法正确的是?

A. var ch chan int

B. ch := make(chan int)

C. <- ch

D. ch <-

答案

> ABC

> 写 chan 时，<- 右端必须要有值

Go 语言中 cap 函数可以作用于那些内容?

cap 函数在讲引用的问题中已经提到，可以作用于的类型有：

- array(数组)
- slice(切片)
- channel(通道)

go convey 是什么？一般用来做什么?

- go convey 是一个支持 golang 的单元测试框架
- 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
- go convey 提供了丰富的断言简化测试用例的编写

Go 语言当中 new 和 make 有什么区别吗?

new的作用是初始化一个纸箱类型的指针 new函数是内建函数，函数定义：

```
func new(Type)*Type
```

- 使用 new函数来分配空间
- 传递给 new函数的是一个类型，而不是一个值
- 返回值是指向这个新非配的地址的指针

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。

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

虽然这三个函数，都是格式化输出，但是输出的目标不一样 Printf 是标准输出，一般是屏幕，也可以重定向。Sprintf()是把格式化字符串输出到指定的字符串中。 Fprintf()是吧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?

数组：数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，不指定也会根据处初始化对的自动推算出大小，不可改变数组是通过值传递的

切片：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。

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。

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. 切片是引用传递

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

关于 channel 下面描述正确的是?

A. 向已关闭的通道发送数据会引发 panic；

B. 从已关闭的缓冲通道接收数据，返回已缓冲数据或者零值；

C. 无论接收还是接收，nil 通道都会阻塞；

D. close() 可以用于只接收通道；

E. 单向通道可以转换为双向通道；

F. 不能在单向通道上做逆向操作（例如：只发送通道用于接收）；

答案

> A B C F

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：

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

- 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。
- 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。
- 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，可以**迭代以及垃圾回收优化**。切片本身并不是动态数组或者数组指针。它内部实现的数据结构通过指针引用

底层数组，设定相关属性将数据读写操作限定在指定的区域内。切片本身是一个只读对象，其工作机制类似数组指针的一种封装。切片对象非常小，是因为它是只有3 个字段的数据结构：

- 指向底层数组的指针
- 切片的长度
- 切片的容量

Golang Slice 的扩容机制，有什么注意点?

Go 中切片扩容的策略是这样的：

- 首先判断，如果新申请容量大于2 倍的旧容量，最终容量就是新申请的容量
- 否则判断，如果旧切片的长度小于1024，则最终容量就是旧容量的两倍
- 否则判断，如果旧切片长度大于等于1024，则最终容量从旧容量开始循环增加原来的1/4,直到最终容量大于等于新申请的容量
- 如果最终容量计算值溢出，则最终容量就是新申请容量

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)。

new() 与 make() 的区别

> new只初始化并返回指针，而make不仅仅要做初始化，还需要设置一些数组的长度、容量等

Golang Map 如何扩容

装载因子：count/2^B

触发条件：

1. 装填因子是否大于6.5
2. overflow bucket 是否太多

解决方法：

1. 双倍扩容：扩容采取了一种称为“渐进式”地方式，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成了链表，性能大大降低，此时哈希种子 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。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。

如果两个不同的 key 落在的同一个桶上，hash 冲突使用链表法接近，遍历 bucket 中的 key 如果当前处于 map进行了扩容，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。

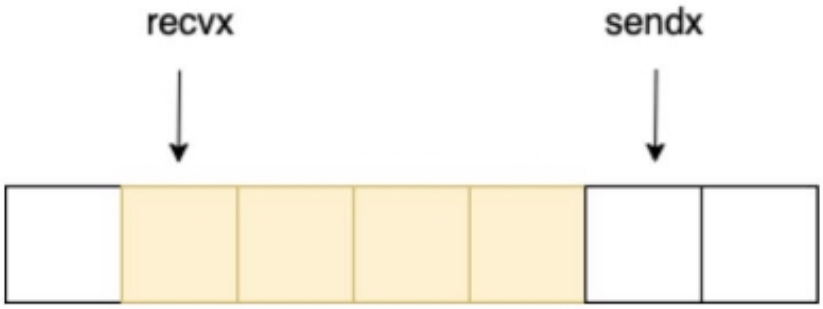
所以 channel 收发遵循先进先出 FIFO，分为有缓存和无缓存，channel 中大致有 buffer(当缓冲区大小部位0 时，是个 ring buffer)、sendx 和 recvx 收发的位置(ring buffer 记录实现)、sendq、recvq 当前 channel 因为缓冲区不足而阻塞的队列、使用双向链表存储、还有一个 mutex 锁控制并发、其他原属等。

Go 语言的 Channel 特性?

1. 给一个 nil channel 发送数据，造成永远阻塞
2. 从一个 nil channel 接收数据，造成永远阻塞
3. 给一个已经关闭的 channel 发送数据，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
5. 无缓冲的 channel 是同步的，而有缓冲的 channel 是非同步的
6. 关闭一个 nil channel 将会发生 panic

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：



hchan 中有两个与 buffer 相关的变量:recvx 和 sendx。其中 sendx 表示buffer 中可写的 index，recvx 表示 buffer 中可读的 index。从 recvx 到 sendx 之间的元素，表示已正常存放入 buffer 中的数据。

我们可以直接使用 buf[recvx]来读取到队列的第一个元素，使用 buf[sendx]= x 来将元素放到队尾。

Go 进阶

golang面试官：for select时，如果通道已经关闭会怎么样？如果只有一个case呢？

https://mp.weixin.qq.com/s/Oa3eExufo2Req_9IrDys-g

golang面试题：对已经关闭的的chan进行读写，会怎么样？为什么？

https://mp.weixin.qq.com/s/izbZ3JRqX6jI6Wn7bV6xNQ

golang面试题：对未初始化的的chan进行读写，会怎么样？为什么？

https://mp.weixin.qq.com/s/ixJu0wrGXsCcGzveCqnr6A

golang面试题：能说说uintptr和unsafe.Pointer的区别吗？

https://mp.weixin.qq.com/s/PSkz0zj-vqKzmIKa_b-xAA

golang 面试题：reflect（反射包）如何获取字段 tag？为什么 json 包不能导出私有变量的 tag？

https://mp.weixin.qq.com/s/WK9StkC3Jfy-o1dUqlo7Dg

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：

- 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。
- 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:

- 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。

1. 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。

- 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。

- 分代搜集（generation）

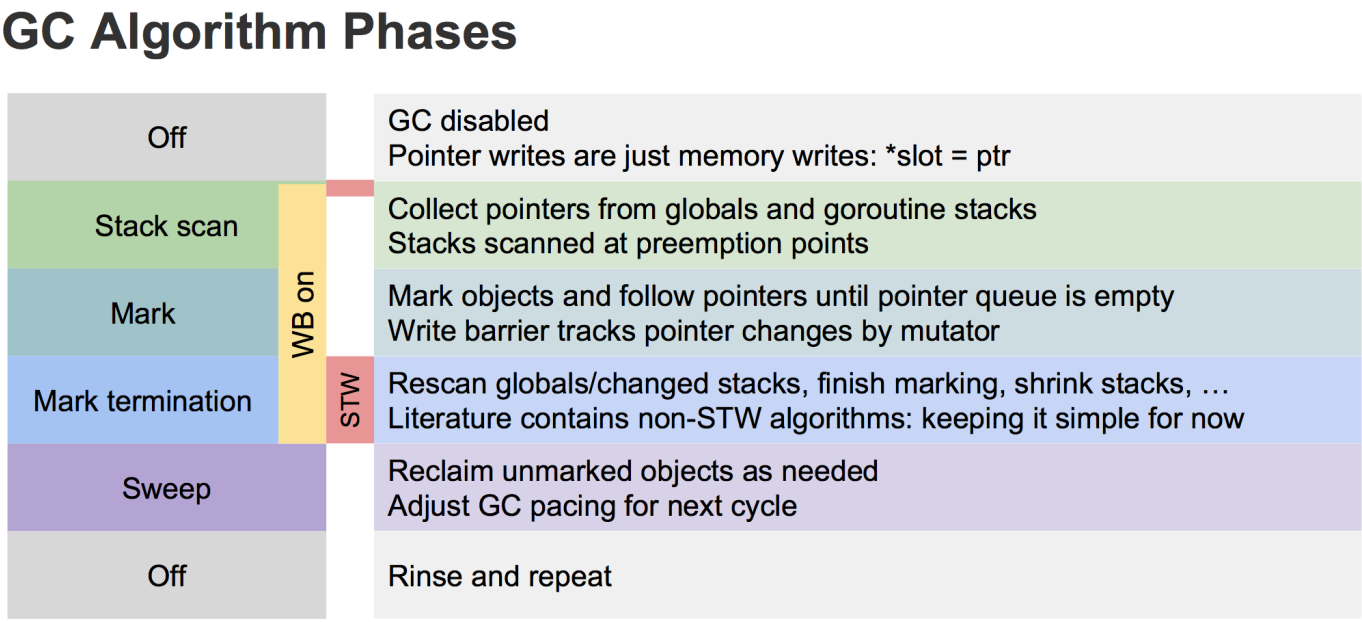
java的jvm 就使用的分代回收的思路。在面向对象编程语言中，绝大多数对象的生命周期都非常短。分代收集的基本思想是，将堆划分为两个或多个称为代（generation）的空间。新创建的对象存放在称为新生代（young generation）中（一般来说，新生代的大小会比 老年代小很多），随着垃圾回收的重复执行，生命周期较长的对象会被提升（promotion）到老年代中（这里用到了一个分类的思路，这个是也是科学思考的一个基本思路）。

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。

详细的Golang的GC介绍可以参看Golang垃圾回收.

Golang 中 Goroutine 如何调度?

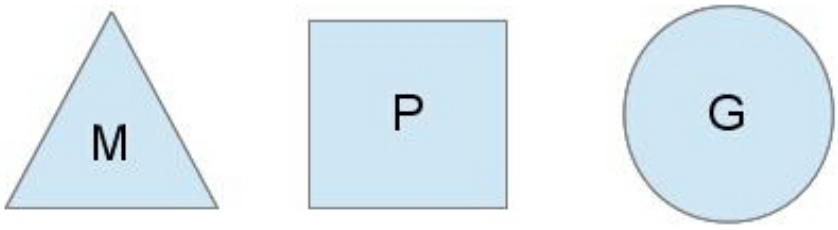
goroutine是Golang语言中最经典的设计，也是其魅力所在，goroutine的本质是协程，是实现并行计算的核心。goroutine使用方式非常的简单，只需使用go关键字即可启动一个协程，并且它是处于异步方式运行，你不需要等它运行完成以后在执行以后的代码。

```
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。

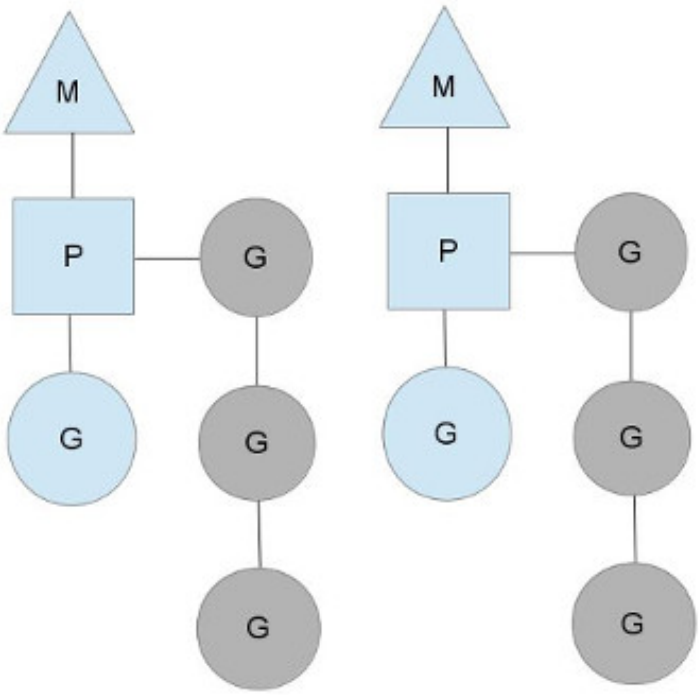
groutine能拥有强大的并发实现是通过GPM调度模型实现.



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）.

- 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
- 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。
- 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
- 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。

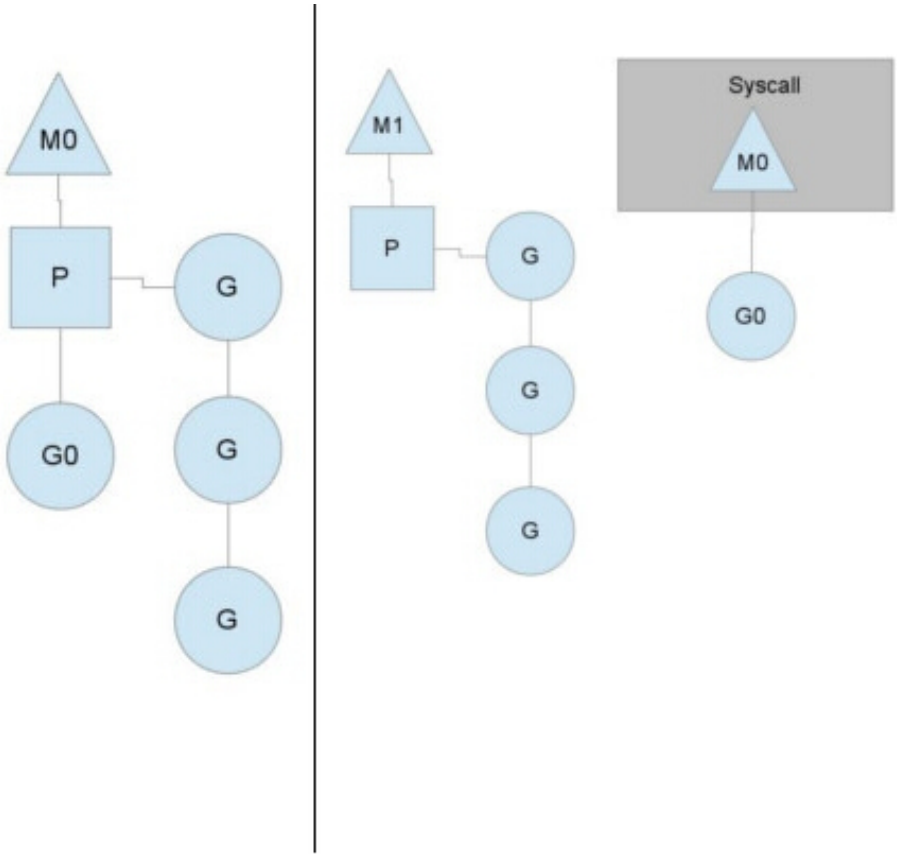
调度实现:



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。

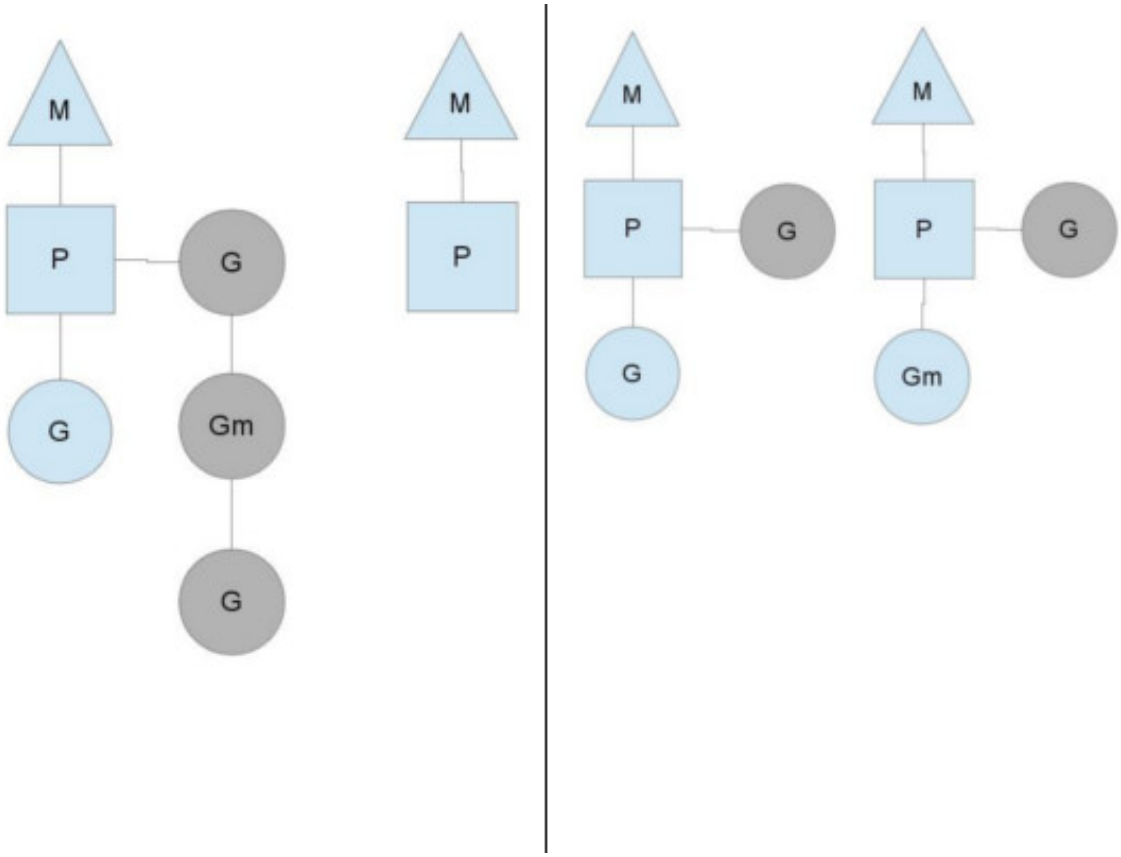
图中灰色的那些goroutine并没有运行，而是出于ready的就绪态，正在等待被调度。P维护着这个队列（称之为runqueue），Go语言里，启动一个goroutine很容易：go function 就行，所以每有一个go语句被执行，runqueue队列就在其末尾加入一个goroutine，在下一个调度点，就从runqueue中取出（如何决定取哪个goroutine？）一个goroutine执行。

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。

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。

下面这段代码有什么错误吗?

```
func funcMui(x,y int)(sum int,error){
    return x+y,nil
}
```

解析

> 第二个返回值没有命名,在函数有多个返回值时，只要有一个返回值有命名，
> 其他的也必须命名。如果有多个返回值必须加上括号()；
> 如果只有一个返回值且命名也必须加上括号()。
> 这里的第一个返回值有命名 sum，第二个没有命名，所以错误。

下面几段代码能否通过编译，如果能，输出什么?

```
func main() {
    list := new([]int)
    // 编译错误
    // new([]int) 之后的 list 是一个未设置长度的 *[]int 类型的指针
    // 不能对未设置长度的指针执行 append 操作。
    list = append(list, 1)
    fmt.Println(list)

    s1 := []int{1, 2, 3}
    s2 := []int{4, 5}
    // 编译错误，s2需要展开
    s1 = append(s1, s2)
    fmt.Println(s1)
}
```

下面能否通过编译?

```
func Test7(t *testing.T) {
    sn1 := struct {
        age  int
        name string
    }{age: 11, name: "qq"}
    sn2 := struct {
        age  int
        name string
    }{age: 11, name: "qq"}
    // true
    if sn1 == sn2 {
        fmt.Println("sn1 == sn2")
    }

    sm1 := struct {
        age int
        m   map[string]string
    }{age: 11, m: map[string]string{"a": "1"}}
    sm2 := struct {
        age int
        m   map[string]string
    }{age: 11, m: map[string]string{"a": "1"}}
    // 编译错误，含有map、slice类型的struct不能进行比较
    if sm1 == sm2 {
        fmt.Println("sm1 == sm2")
    }
}
```

通过指针变量 p 访问其成员变量 name，有哪几种方式?

A. p.name

B. (&p).name

C. (*p).name

D. p->name

答案

> AC

关于字符串连接，下面语法正确的是?

A. str := 'abc' + '123'

B. str := "abc" + "123"

C. str := '123' + "abc"

D. fmt.Sprintf("abc%d", 123)

答案

> BD
>
> golang单引号''中的内容表示单个字符（rune）,反引号``中的内容表示不可转义的字符串

关于iota，下面代码输出什么?

```
func Test10(t *testing.T) {
    const (
        x = iota
        _
        y
        z = "pi"
        k
        p = iota
        q
    )
    fmt.Println(x, y, z, k, p, q)
}
```

输出

> 0 2 pi pi 5 6

Mutex 几种状态

- mutexLocked —表示互斥锁的锁定状态；
- mutexWoken —表示从正常模式被从唤醒；
- mutexStarving —当前的互斥锁进入饥饿状态；
- 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，而是会和新请求锁的 goroutine 竞争锁的拥有。新请求锁的 goroutine 具有优势：它正在 CPU上执行，而且可能有好几个，所以刚刚唤醒的 goroutine 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。如果一个等待的 goroutine 超过1ms没有获取锁，那么它将会把锁转变为饥饿模式。

饥饿模式(公平锁)

为了解决了等待 G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 G(队头)，同时，饥饿模式下，新进来的 G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 g 一直抢不到锁的场景。饥饿模式的触发条件，当一个 G等待锁时间超过1 毫秒时，或者当前队列只剩下一个 g 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，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写锁之后释放通知读锁。同样的写锁也会等等待之前的读锁都释放完毕，才会开始进行后续的操作。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

- RWMutex 是单写多读锁，该锁可以加多个读锁或者一个写锁
- 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
- 写锁会阻止其他 goroutine（无论读和写）进来，整个锁由该 goroutine 独占
- 适用于读多写少的场景
- RWMutex 类型变量的零值是一个未锁定状态的互斥锁。
- RWMutex 在首次被使用之后就不能再被拷贝。
- RWMutex 的读锁或写锁在未锁定状态，解锁操作都会引发 panic。
- RWMutex 的一个写锁 Lock 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。
- RWMutex 的读锁不要用于递归调用，比较容易产生死锁。
- 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）。
- 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。
- 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）

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，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。

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

- 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。
- 每次 Add执行，请求计数器 v 加1，Done方法执行，请求计数器减1，v 为0 时通过信号量唤醒 Wait()。

下面代码输出什么?

```
func Test16(t *testing.T) {
    a := [2]int{5, 6}
    b := [3]int{5, 6}
    if a == b {
        fmt.Println("equal")
    } else {
        fmt.Println("not equal")
    }
}
```

A. compilation error

B. equal

C. not equal

答案

> A 编译错误
>
> 对于数组而言，一个数组是由数组中的值和数组的长度两部分组成的，如果两个数组长度不同，那么两个数组是属于
> 不同类型的，是不能进行比较的

什么是 sync.Once

- 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。
- 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。
- 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

一个或者多个操作在 CPU执行过程中不被中断的特性，称为原子性(atomicity)。这些操作对外表现成一个不可分割的整体，他们要么都执行，要么都不执行，外界不会看到他们只执行到一半的状态。而在现实世界中，CPU 不可能不中断的执行一系列操作，但如果我们在执行多个操作时，能让他们的中间状态对外不可见，那我们就可以宣城他们拥有了“不可分割”的原子性。

在 Go中，一条普通的赋值语句其实不是一个原子操作。列如，在32 位机器上写 int64 类型的变量就会有中间状态，因为他会被拆成两次写操作(MOV)——写低32 位和写高32 位。

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。

Goroutine 定义

Goroutine 是一个与其他 goroutines 并行运行在同一地址空间的 Go 函数或方法。一个运行的程序由一个或更多个 goroutine 组成。它与线程、协程、进程等不同。它是一个 goroutine”—— Rob Pike

Goroutines 在同一个用户地址空间里并行独立执行 functions，channels 则用于 goroutines 间的通信和同步访问控制。

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。

给大家丢脸了，用了三年golang，我还是没答对这道内存泄漏题

https://mp.weixin.qq.com/s/-agtdhlW7Yj7S88a0z7KHg

你一定会遇到的内存回收策略导致的疑似内存泄漏的问题

https://colobu.com/2019/08/28/go-memory-leak-i-dont-think-so/

GMP里为什么要有P?

https://mp.weixin.qq.com/s/SEE2TUeZQZ7W1BKkmnelAA

go栈扩容和栈缩容，连续栈的缺点

https://segmentfault.com/a/1190000019570427

golang隐藏技能:怎么访问私有成员

https://www.jianshu.com/p/7b3638b47845

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。

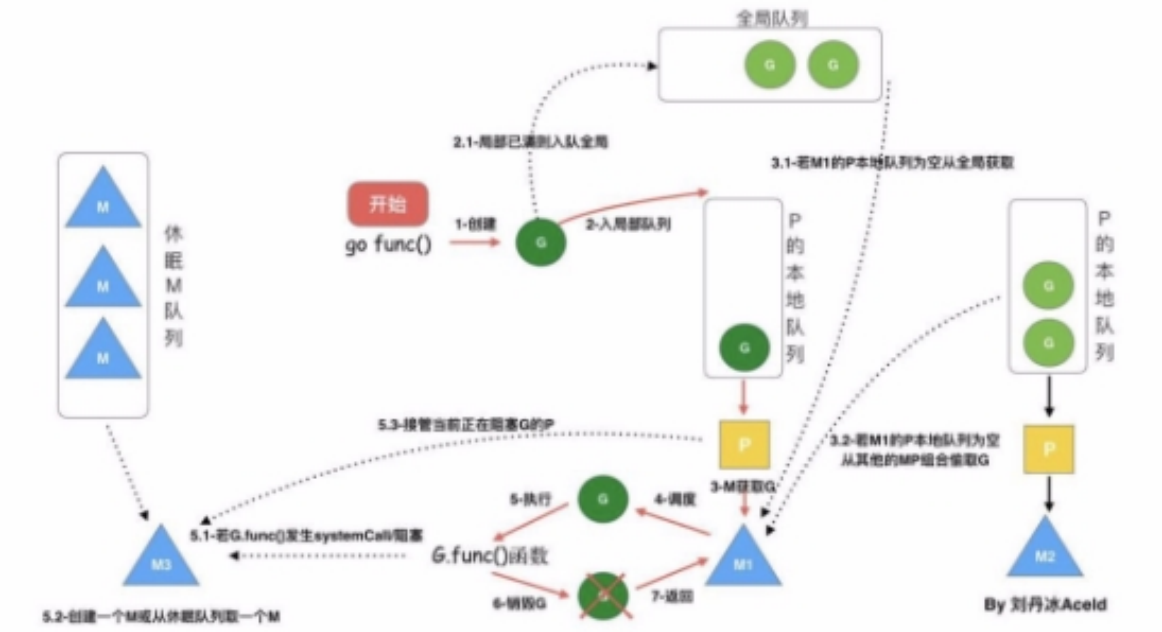
1．单一全局互斥锁(Sched.Lock)和集中状态存储

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



- 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)
- 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
- 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

- 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)。

- 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(流程5.3)

GMP 中 work stealing 机制

存到 P本地队列或者是全局队列。P此时去唤醒一个 M。P继续执行它的执行序。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。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，存在问题

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

- I/O，select
- block on syscall
- channel
- 等待锁
- 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

- 释放闲置超过5 分钟的 span 物理内存；
- 如果超过2 分钟没有垃圾回收，强制执行；
- 将长时间未处理的 netpoll 添加到全局队列；
- 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)；
- 收回因 syscall 长时间阻塞的 P；

golang面试题：怎么避免内存逃逸?

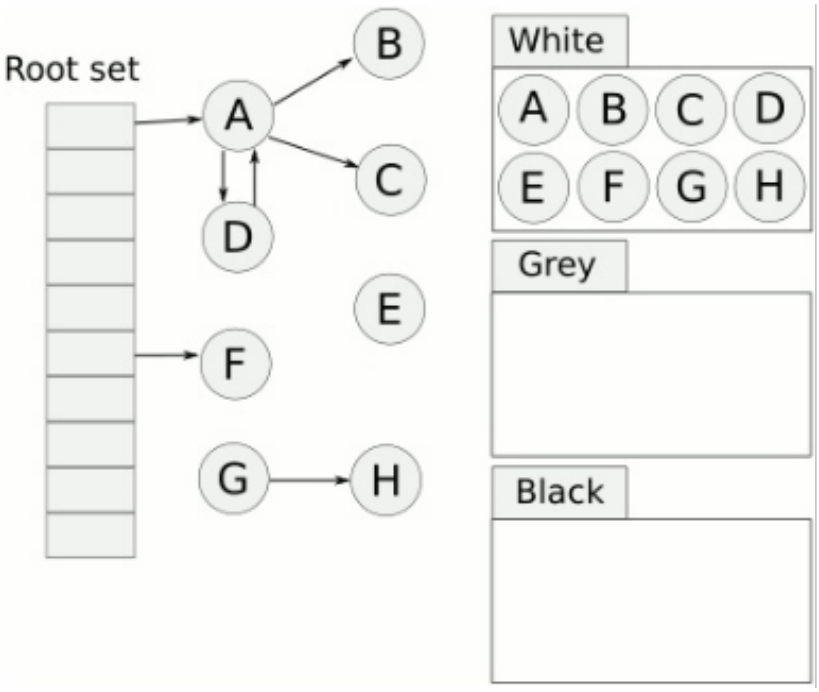
https://mp.weixin.qq.com/s/VzRTHz1JaDUvNRVB_yJa1A

golang面试题：简单聊聊内存逃逸?

https://mp.weixin.qq.com/s/wJmztRMB1ZAAIltyMcS0tw

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：

首先把所有的对象都放到白色的集合中

- 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
- 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
- 循环步骤3，知道灰色集合中没有对象
- 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

插入写屏障

golang 的回收没有混合屏障之前，一直是插入写屏障，由于栈赋值没有 hook 的原因，所以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，所以后面 golang 是引用混合写屏障解决这个问题。混合写屏障之后，就没有 STW。

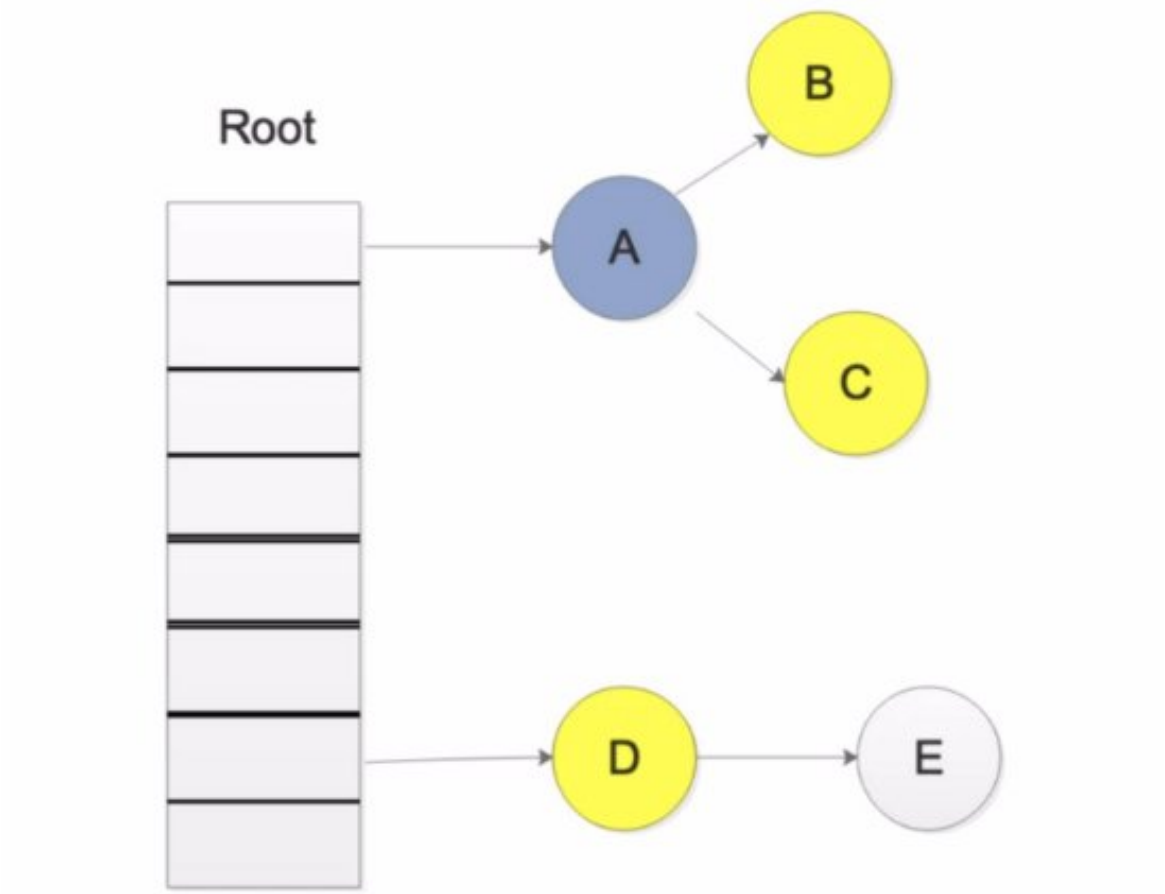
删除写屏障

goalng 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中被清理掉。

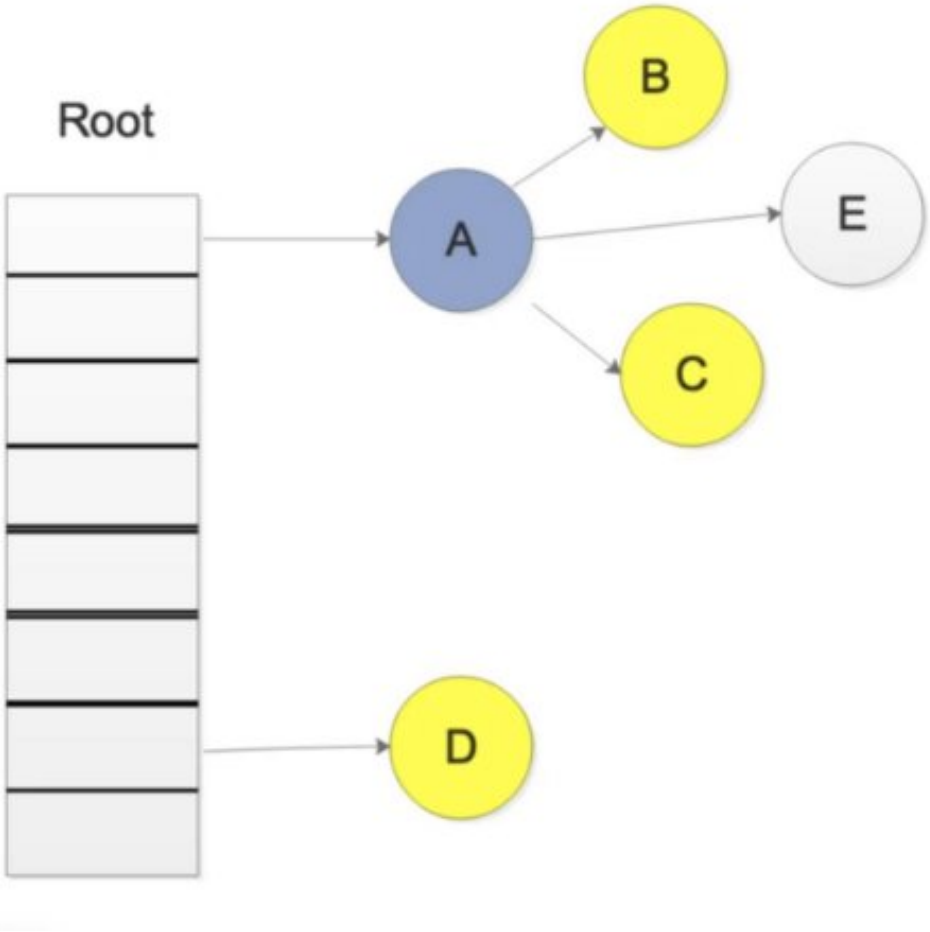
写屏障

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。

混合写屏障

- 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；
- 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；
- 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；
- 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 赋值器的工作的哈（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.由 GOGC 控制,默认100,即2 倍的关系,200 就是3 倍,

当 Go新创建的对象所占用的内存大小，除以上次 GC结束后保留下来的对象占用内存大小。

Go 语言中 GC 的流程是什么?

当前版本的 Go 以 STW 为界限，可以将 GC 划分为五个阶段：阶段说明赋值器状态 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 STWGCoff内存清扫阶段，将需要回收的内存归还到堆中，写屏障关闭并发 GCoff内存归还阶段，将过多的内存归还给操作系统，写屏障关闭并发。

GC 如何调优

通过 go tool pprof 和 go tool trace 等工具

- 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。
- 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。
- 需要时，增大 GOGC 的值，降低 GC 的运行频率。

Go语言的栈空间管理是怎么样的?

Go语言的运行环境（runtime）会在goroutine需要的时候动态地分配栈空间，而不是给每个goroutine分配固定大小的内存空间。这样就避免了需要程序员来决定栈的大小。

分块式的栈是最初Go语言组织栈的方式。当创建一个goroutine的时候，它会分配一个8KB的内存空间来给goroutine的栈使用。我们可能会考虑当这8KB的栈空间被用完的时候该怎么办?

为了处理这种情况，每个Go函数的开头都有一小段检测代码。这段代码会检查我们是否已经用完了分配的栈空间。如果是的话，它会调用`morestack`函数。`morestack`函数分配一块新的内存作为栈空间，并且在这块栈空间的底部填入各种信息（包括之前的那块栈地址）。在分配了这块新的栈空间之后，它会重试刚才造成栈空间不足的函数。这个过程叫做栈分裂（stack split）。

在新分配的栈底部，还插入了一个叫做`lessstack`的函数指针。这个函数还没有被调用。这样设置是为了从刚才造成栈空间不足的那个函数返回时做准备的。当我们从那个函数返回时，它会跳转到`lessstack`。`lessstack`函数会查看在栈底部存放的数据结构里的信息，然后调整栈指针（stack pointer）。这样就完成了从新的栈块到老的栈块的跳转。接下来，新分配的这个块栈空间就可以被释放掉了。

`分块式的栈`让我们能够按照需求来扩展和收缩栈的大小。 Go开发者不需要花精力去估计goroutine会用到多大的栈。创建一个新的goroutine的开销也不大。当 Go开发者不知道栈会扩展到多少大时，它也能很好的处理这种情况。

这一直是之前Go语言管理栈的的方法。但这个方法有一个问题。缩减栈空间是一个开销相对较大的操作。如果在一个循环里有栈分裂，那么它的开销就变得不可忽略了。一个函数会扩展，然后分裂栈。当它返回的时候又会释放之前分配的内存块。如果这些都发生在一个循环里的话，代价是相当大的。 这就是所谓的热分裂问题（hot split problem）。它是Go语言开发者选择新的栈管理方法的主要原因。新的方法叫做`栈复制法（stack copying）`。

栈复制法一开始和分块式的栈很像。当goroutine运行并用完栈空间的时候，与之前的方法一样，栈溢出检查会被触发。但是，不像之前的方法那样分配一个新的内存块并链接到老的栈内存块，新的方法会分配一个两倍大的内存块并把老的内存块内容复制到新的内存块里。这样做意味着当栈缩减回之前大小时，我们不需要做任何事情。栈的缩减没有任何代价。而且，当栈再次扩展时，运行环境也不需要再做任何事。它可以重用之前分配的空间。

栈的复制听起来很容易，但实际操作并非那么简单。存储在栈上的变量的地址可能已经被使用到。也就是说程序使用到了一些指向栈的指针。当移动栈的时候，所有指向栈里内容的指针都会变得无效。然而，指向栈内容的指针自身也必定是保存在栈上的。这是为了保证内存安全的必要条件。否则一个程序就有可能访问一段已经无效的栈空间了。

因为垃圾回收的需要，我们必须知道栈的哪些部分是被用作指针了。当我们移动栈的时候，我们可以更新栈里的指针让它们指向新的地址。所有相关的指针都会被更新。我们使用了垃圾回收的信息来复制栈，但并不是任何使用栈的函数都有这些信息。因为很大一部分运行环境是用C语言写的，很多被调用的运行环境里的函数并没有指针的信息，所以也就不能够被复制了。当遇到这种情况时，我们只能退回到分块式的栈并支付相应的开销。

这也是为什么现在运行环境的开发者正在用Go语言重写运行环境的大部分代码。无法用Go语言重写的部分（比如调度器的核心代码和垃圾回收器）会在特殊的栈上运行。这个特殊栈的大小由运行环境的开发者设置。

这些改变除了使栈复制成为可能，它也允许我们在将来实现并行垃圾回收。

另外一种不同的栈处理方式就是在虚拟内存中分配大内存段。由于物理内存只是在真正使用时才会被分配，因此看起来好似你可以分配一个大内存段并让操 作系统处理它。下面是这种方法的一些问题

首先，32位系统只能支持4G字节虚拟内存，并且应用只能用到其中的3G空间。由于同时运行百万goroutines的情况并不少见，因此你很可 能用光虚拟内存，即便我们假设每个goroutine的stack只有8K。

第二，然而我们可以在64位系统中分配大内存，它依赖于过量内存使用。所谓过量使用是指当你分配的内存大小超出物理内存大小时，依赖操作系统保证 在需要时能够分配出物理内存。然而，允许过量使用可能会导致一些风险。由于一些进程分配了超出机器物理内存大小的内存，如果这些进程使用更多内存 时，操作系统将不得不为它们补充分配内存。这会导致操作系统将一些内存段放入磁盘缓存，这常常会增加不可预测的处理延迟。正是考虑到这个原因，一 些新系统关闭了对过量使用的支持。

Goroutine和Channel的作用分别是什么?

进程是内存资源管理和cpu调度的执行单元。为了有效利用多核处理器的优势，将进程进一步细分，允许一个进程里存在多个线程，这多个线程还是共享同一片内存空间，但cpu调度的最小单元变成了线程。

那协程又是什么呢，以及与线程的差异性??

协程，可以看作是轻量级的线程。但与线程不同的是，线程的切换是由操作系统控制的，而协程的切换则是由用户控制的。

最早支持协程的程序语言应该是lisp方言scheme里的continuation（续延），续延允许scheme保存任意函数调用的现场，保存起来并重新执行。Lua,C#,python等语言也有自己的协程实现。

Go中的goroutinue就是协程,可以实现并行，多个协程可以在多个处理器同时跑。而协程同一时刻只能在一个处理器上跑（可以把宿主语言想象成单线程的就好了）。 然而,多个goroutine之间的通信是通过channel，而协程的通信是通过yield和resume()操作。

goroutine非常简单，只需要在函数的调用前面加关键字go即可，例如:

```
go elegance()
```

我们也可以启动5个goroutines分别打印索引。

```
func main() {
    for i:=1;i<5;i++ {
        go func(i int) {
            fmt.Println(i)
        }(i)
    }
    // 停歇5s，保证打印全部结束
    time.Sleep(5*time.Second)
}
```

在分析goroutine执行的随机性和并发性，启动了5个goroutine，再加上main函数的主goroutine，总共有6个goroutines。由于goroutine类似于"守护线程"，异步执行的,如果主goroutine不等待片刻，可能程序就没有输出打印了。

在Golang中channel则是goroutinues之间进行通信的渠道。

可以把channel形象比喻为工厂里的传送带,一头的生产者goroutine往传输带放东西,另一头的消费者goroutinue则从输送带取东西。channel实际上是一个有类型的消息队列,遵循先进先出的特点。

1. channel的操作符号

ch <- data 表示data被发送给channel ch；

data <- ch 表示从channel ch取一个值，然后赋给data。

1. 阻塞式channel

channel默认是没有缓冲区的，也就是说，通信是阻塞的。send操作必须等到有消费者accept才算完成。

应用示例:

```
func main() {
    ch1 := make(chan int)
    go pump(ch1) // pump hangs
    fmt.Println(<-ch1) // prints only 1
}

func pump(ch chan int) {
    for i:= 1; ; i++ {
        ch <- i
    }
}
```

在函数pump()里的channel在接受到第一个元素后就被阻塞了，直到主goroutinue取走了数据。最终channel阻塞在接受第二个元素，程序只打印 1。

没有缓冲(buffer)的channel只能容纳一个元素，而带有缓冲(buffer)channel则可以非阻塞容纳N个元素。发送数据到缓冲(buffer) channel不会被阻塞，除非channel已满；同样的，从缓冲(buffer) channel取数据也不会被阻塞，除非channel空了。

怎么查看Goroutine的数量?

GOMAXPROCS中控制的是未被阻塞的所有Goroutine,可以被Multiplex到多少个线程上运行,通过GOMAXPROCS可以查看Goroutine的数量。

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。

图1：微服务的蜂窝表示–微服务访谈问题

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

优势	说明
独立开发	所有微服务都可以根据各自的功能轻松开发
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?

- 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
- 组件化—微服务被视为可以轻松更换和升级的独立组件
- 业务能力—微服务非常简单，专注于单一功能
- 自治—开发人员和团队可以彼此独立工作，从而提高速度
- 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
- 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
- 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
- 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

微服务架构是什么样子的?

通常传统的项目体积庞大，需求、设计、开发、测试、部署流程固定。新功能需要在原项目上做修改。

但是微服务可以看做是对大项目的拆分，是在快速迭代更新上线的需求下产生的。新的功能模块会发布成新的服务组件，与其他已发布的服务组件一同协作。 服务内部有多个生产者和消费者，通常以http rest的方式调用，服务总体以一个（或几个）服务的形式呈现给客户使用。

微服务架构是一种思想对微服务架构我们没有一个明确的定义，但简单来说微服务架构是：

采用一组服务的方式来构建一个应用，服务独立部署在不同的进程中，不同服务通过一些轻量级交互机制来通信，例如 RPC、HTTP 等，服务可独立扩展伸缩，每个服务定义了明确的边界，不同的服务甚至可以采用不同的编程语言来实现，由独立的团队来维护。

Golang的微服务框架kit中有详细的微服务的例子,可以参考学习.

微服务架构设计包括：

1. 服务熔断降级限流机制 熔断降级的概念(Rate Limiter 限流器,Circuit breaker 断路器).
2. 框架调用方式解耦方式 Kit 或 Istio 或 Micro 服务发现(consul zookeeper kubeneters etcd) RPC调用框架.
3. 链路监控,zipkin和prometheus.
4. 多级缓存.
5. 网关 (kong gateway).
6. Docker部署管理 Kubenetters.
7. 自动集成部署 CI/CD 实践.
8. 自动扩容机制规则.
9. 压测 优化.
10. Trasport 数据传输(序列化和反序列化).
11. Logging 日志.
12. Metrics 指针对每个请求信息的仪表盘化.

微服务架构介绍详细的可以参考:

- Microservice Architectures

微服务架构如何运作?

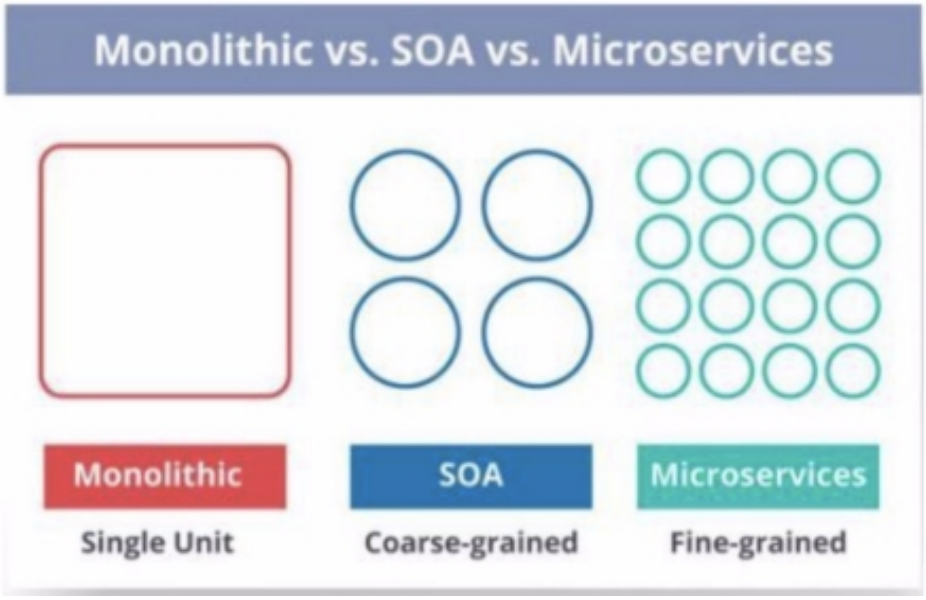
微服务架构具有以下组件：

- 客户端–来自不同设备的不同用户发送请求。
- 身份提供商–验证用户或客户身份并颁发安全令牌。
- API 网关–处理客户端请求。
- 静态内容–容纳系统的所有内容。
- 管理–在节点上平衡服务并识别故障。
- 服务发现–查找微服务之间通信路径的指南。
- 内容交付网络–代理服务器及其数据中心的分布式网络。
- 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?

微服务架构的优点微服务架构的缺点自由使用不同的技术增加故障排除挑战每个微服务都侧重于单一功能由于远程呼叫而增加延迟支持单个可部署单元增加了配置和其他操作的工作量允许经常发布软件难以保持交易安全确保每项服务的安全性艰难地跨越各种便捷跟踪数据多个服务是并行开发和部署的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?



单片 SOA 和微服务之间的比较–微服务访谈问题

- 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。
- 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。
- 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。

怎么做弹性扩缩容，原理是什么?

弹性伸缩（Auto Scaling）根据您的业务需求和伸缩策略，为您自动调整计算资源。您可设置定时、周期或监控策略，恰到好处地增加或减少CVM实例，并完成实例配置，保证业务平稳健康运行。在需求高峰期时，弹性伸缩自动增加CVM实例的数量，以保证性能不受影响；当需求较低时，则会减少CVM实例数量以降低成本。弹性伸缩既适合需求稳定的应用程序，同时也适合每天、每周、每月使用量不停波动的应用程序。

说一下中间件原理.

中间件（middleware）是基础软件的一大类，属于可复用软件的范畴。中间件处于操作系统软件与用户的应用软件的中间。中间件在操作系统、网络和数据库之上，应用软件的下层，总的作用是为处于自己上层的应用软件提供运行与开发的环境，帮助用户灵活、高效地开发和集成复杂的应用软件 IDC的定义是：中间件是一种独立的系统软件或服务程序，分布式应用软件借助这种软件在不同的技术之间共享资源，中间件位于客户机服务器的操作系统之上，管理计算资源和网络通信。

中间件解决的问题是：

在中间件产生以前，应用软件直接使用操作系统、网络协议和数据库等开发，这些都是计算机最底层的东西，越底层越复杂，开发者不得不面临许多很棘手的问题，如操作系统的多样性，繁杂的网络程序设计、管理，复杂多变的网络环境，数据分散处理带来的不一致性问题、性能和效率、安全，等等。这些与用户的业务没有直接关系，但又必须解决，耗费了大量有限的时间和精力。于是，有人提出能不能将应用软件所要面临的共性问题进行提炼、抽象，在操作系统之上再形成一个可复用的部分，供成千上万的应用软件重复使用。这一技术思想最终构成了中间件这类的软件。中间件屏蔽了底层操作系统的复杂性，使程序开发人员面对一个简单而统一的开发环境，减少程序设

计的复杂性，将注意力集中在自己的业务上，不必再为程序在不同系统软件上的移植而重复工作，从而大大减少了技术上的负担。

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。

- 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。
- 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。
- 配置管理：有时在各种环境中维护组件的配置变得困难。
- 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?

SOA 和微服务之间的主要区别如下：

SOA 微服务

遵循“尽可能多的共享”架构方法遵循“尽可能少分享”架构方法重要性在于“业务功能”重用重要性在于“有界背景”的概念它们有共同的治理和标准它们专注于人们的合作和其他选择的自由使用企业服务总线（ESB）进行通信简单的消息系统

它们支持多种消息协议它们使用轻量级协议，如 HTTP/REST 等

单线程，通常使用 Event Loop 功能进行非多线程，有跟多的开销来处理 I/O

锁定 I/O 处理

最大化应用程序服务可重用性专注于解耦

传统的关系数据库更常用现代关系数据库更常用

系统的变化需要修改整体系统的变化是创造一种新的服务

DevOps/Continuous Delivery 正在变得流

专注于 DevOps/持续交付行，但还不是主流

微服务有什么特点?

您可以列出微服务的特征，如下所示：

图7：微服务的特征–微服务访谈问题

什么是领域驱动设计?

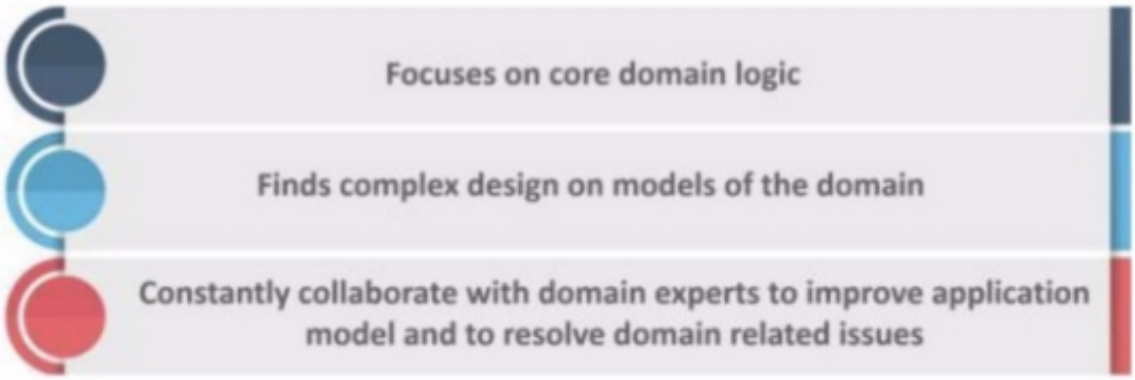


图8： DDD 原理–微服务面试问题

为什么需要域驱动设计（DDD）？

图9：我们需要 DDD 的因素–微服务面试问题

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。

- 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。
- 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。
- 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。

Docker 与虚拟机有何不同?

Docker 不是虚拟化方法。它依赖于实际实现基于容器的虚拟化或操作系统级虚拟化的其他工具。为此，Docker 最初使用 LXC 驱动程序，然后移动到libcontainer 现在重命名为 runc。Docker 主要专注于在应用程序容器内自动部署应用程序。应用程序容器旨在打包和运行单个服务，而系统容器则设计为运行多个进程，如虚拟机。因此，Docker 被视为容器化系统上的容器管理或应用程序部署工具。

- 容器不需要引导操作系统内核，因此可以在不到一秒的时间内创建容器。此功能使基于容器的虚拟化比其他虚拟化方法更加独特和可取。
- 由于基于容器的虚拟化为主机增加了很少或没有开销，因此基于容器的虚拟化具有接近本机的性能。
- 对于基于容器的虚拟化，与其他虚拟化不同，不需要其他软件。
- 主机上的所有容器共享主机的调度程序，从而节省了额外资源的需求。
- 与虚拟机映像相比，容器状态（Docker 或 LXC 映像）的大小很小，因此容器映像很容易分发。
- 容器中的资源管理是通过 cgroup 实现的。Cgroups 不允许容器消耗比分配给它们更多的资源。虽然主机的所有资源都在虚拟机中可见，但无法使用。这可以通过在容器和主机上同时运行 top 或 htop 来实现。所有环境的输出看起来都很相似。

什么是 Docker 镜像?

Docker 镜像是 Docker 容器的源代码，Docker 镜像用于创建容器。使用build 命令创建镜像。

什么是 Docker 容器?

Docker 容器包括应用程序及其所有依赖项，作为操作系统的独立进程运行。

Docker 容器有几种状态?

四种状态：运行、已暂停、重新启动、已退出。

Dockerfile 中最常见的指令是什么?

FROM：指定基础镜像

LABEL：功能是为镜像指定标签

RUN：运行指定的命令CMD：容器启动时要运行的命令

Dockerfile 中的命令 COPY 和 ADD 命令有什么区别?

COPY 与 ADD 的区别 COPY 的 SRC 只能是本地文件，其他用法一致。

解释一下 Dockerfile 的 ONBUILD 指令?

当镜像用作另一个镜像构建的基础时，ONBUILD 指令向镜像添加将在稍后执行的触发指令。如果要构建将用作构建其他镜像的基础的镜像（例如，可以使用特定于用户的配置自定义的应用程序构建环境或守护程序），这将非常有用。

什么是 Docker Swarm?

Docker Swarm 是 Docker 的本机群集。它将 Docker 主机池转变为单个虚拟Docker 主机。Docker Swarm 提供标准的 Docker API，任何已经与 Docker守护进程通信的工具都可以使用 Swarm 透明地扩展到多个主机。

如何在生产中监控 Docker?

Docker 提供 docker stats 和 docker 事件等工具来监控生产中的 Docker。我们可以使用这些命令获取重要统计数据的报告。

Docker 统计数据：当我们使用容器 ID 调用 docker stats 时，我们获得容器的CPU，内存使用情况等。它类似于 Linux 中的 top 命令。

Docker 事件：Docker 事件是一个命令，用于查看 Docker 守护程序中正在进

行的活动流。

一些常见的 Docker 事件：attach，commit，die，detach，rename，

destroy 等。我们还可以使用各种选项来限制或过滤我们感兴趣的事件。

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 映像（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?

一些最常用的指令如下：

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。

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 。

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。

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。

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 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。

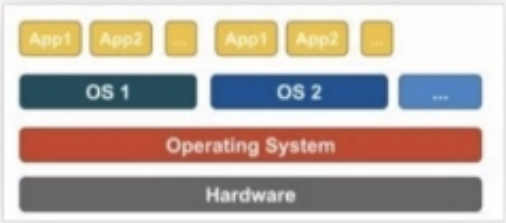
一般的 Docker 事件有：attach（关联），commit（提交），die（僵死）， detach（取消关联），rename（改名），destory（销毁）等。也可使用多个选项对事件记录筛选找到想要的事件信息。

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。在 Docker v。1.9 之前的版本中，删除这些孤儿卷存在很大问题。

什么是半虚拟化（Paravirtualization）?

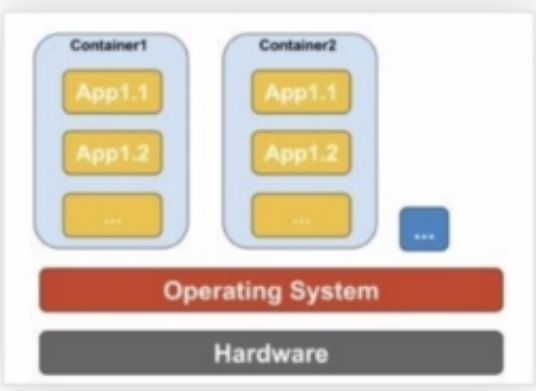
Paravirtualization，也称为第1 类虚拟机管理（层）程序，其直接在硬件或裸机（bare-metal）上运行，提供虚拟机直接使用物理硬件的服务，它帮助主机操作系统，虚拟化硬件和实际硬件进行协作以实现最佳性能。这种虚拟层管理技术的程序一般占用系统资源较小，其本身并不需要占用大量系统资源。



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。



- 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。
- 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。
- 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。
- 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。
- 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。
- 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?

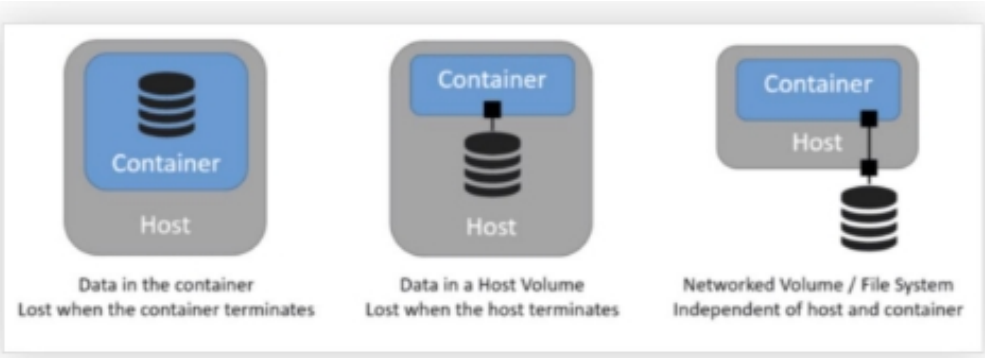
配置文件中的 ONBUILD 指令为创建的 Docker image （映像）加入在将来执行的指令（译注：在当前配置文件生成的映像中并不执行），用于在以这个创建的映像为基础的创建的子映像（image）中执行或定制。举例，以基映像创建自己的映像时，可定制创建特有的用户化的配置环境。

- 译注：由于原文较短，关于这个问题容易迷惑。译者认为，总体来说关键理解--以基础映像创建自有的映像过程中，基础映像中所有的创建层或指令是以整体或固化的方式导入自有映像中的，自有映像是不能对这个过程进行自有定制。而 ONBUILD 指令提供了将某些层从基础映像中剥离出来提供给之后以自有映像为基础映像派生新的映像的可定制途径。这对发布映像而普适在不同的运行环境定制非常有用。不当之处，请指正！）

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。



译注：

1 图中文字：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 图中文字：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 图中文字：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。

在 Windows 系统上可以运行原生的 Docker 容器吗?

在'Windows Server 2016'系统上，你可以运行 Windows 的原生容器，微软推出其映像是'Windows Nano Server'，一个轻量级的运行在容器中的 Windows 原生系统。您可以在其中布署基于.NET 的应用。

译注：结合 Docker 的基本技术原理，参考后面的问题26 和问题27，可推测，微软在系统内核上开发了对 Docker 的支持，支持其闭源系统的容器化虚拟技术。但译者认为， Windows 系统本就是闭源紧耦合的系统，好像你在本机上不装.NET 组件，各应用能很好运行似的。何必再弄个容器，浪费资源。这只是译者自己之孔见，想喷就喷！另： Windows Server 2016 版本之后的都可支持这种原生 Docker 技术，如 Windows Server 2018 版。

在非 Linux 操作系统平台上如何运行 Docker ?

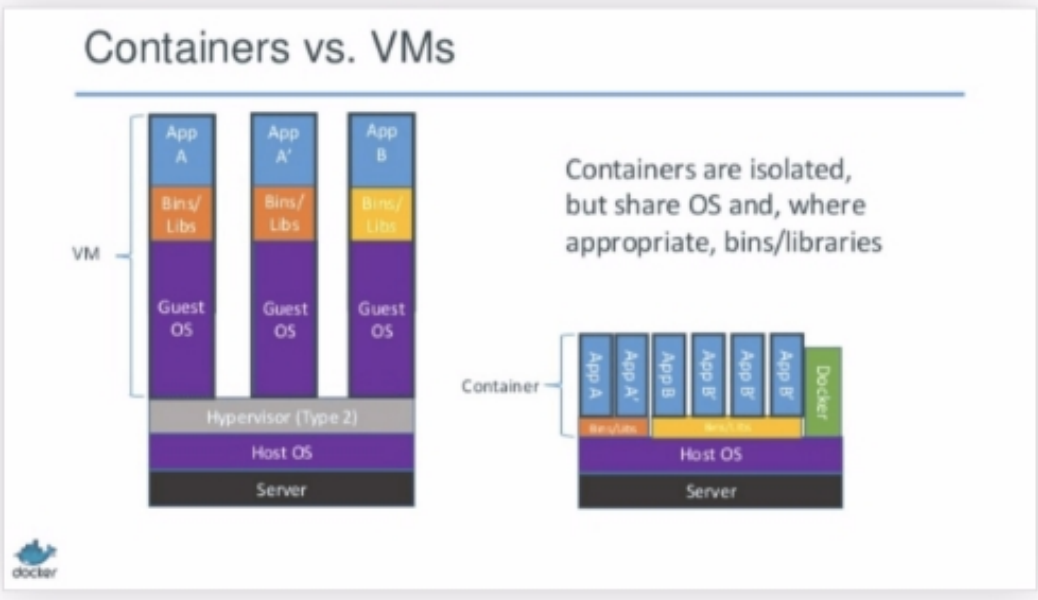
容器化虚拟技术概念可能来源于，在 Linux 内核版本2.6.24 上加入的对命名空间（ namespace）的技术支持特性。容器化进程加入其进程 ID 到其创建的每个进程上并且对每个进程中的系统级调用进行访问控制及审查。其本身是由系统级调用 clone ()克隆出来的进程，允许其创建属于自己命名空间的进程实例，而区别于之前的，归属与整个本机系统的进程实例。

如果上述在 Linux 系统内核上的技术实现成为可能，那么明显的问题是如何在非 Linux 系统上运行容器化的 Docker 。过去， Mac 和 Windows 系统上运行 Docker 容器都使用 Linux 虚拟机（VMs）技术， Docker 工具箱使用的容器运行在 Virtual Box 虚拟机上。现在，最新的情况是， Windows 平台上使用的是 Hyper-V 产品技术，Mac 平台上使用的是 Hypervisor.framework （框架）产品技术。

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。



图上文字说明：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

仅有下面的一些对比：

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。

docker-compose -f docker-com

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。

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：

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

Redis 优势：

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

- 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。
- 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?

- 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)
- 支持丰富数据类型，支持 string，list，set，Zset，hash 等
- 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
- 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?

- 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
- Redis 的速度比 Memcached 快很
- Redis 可以持久化其数据

Memcache 与 Redis 的区别都有哪些?

- 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。
- 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。
- 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?

答：512M

Redis 集群最大节点个数是多少?

16384 个。

Reids 的特点

Redis 本质上是一个 Key-Value 类型的内存数据库，很像 数据库统统加载在内存当中进行操作，定期通过异步操作把数据库数据 硬盘上进行保存。

Memcached，整个

flush 到

因为是纯内存操作，Redis 的性能非常出色，每秒可以处理超过 10 万次读写操 作，是已知性能最快的 Key-Value DB。

Redis 的出色之处不仅仅是性能，Redis 最大的魅力是支持保存多种数据结构， 此外单个 value 的最大限制是 1GB，不像 Memcached 只能保存 1MB 的数据， 因此 Redis 可以用来实现很多有用的功能。

比方说用他的 List 来做 FIFO 双向链表，实现一个轻量级的高性 能消息队列服 务，用他的 Set 可以做高性能的 tag 系统等等。另外 Redis 也可以对存入的 Key- Value 设置 expire 时间，因此也可以被当作一 个功能加强版的 Memcached 来

用。

Redis 的主要缺点是数据库容量受到物理内存的限制，不能用作海量数据的高性 能读写，因此 Redis 适合的场景主要局限在较小数据量的高性能操作和运算上。

使用 Redis 有哪些好处?

- 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 O(1)。

- 支持丰富数据类型，支持 string，list，set，sorted set，hash。
- 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。
- 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。

为什么 edis 需要把所有数据放到内存中?

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 的内存用完了会发生什么?

如果达到设置的上限， 返回。）或者你可以将 限时会冲刷掉旧的内容。

Redis 的写命令会返回错误信息（但是读命令还可以正常 Redis 当缓存来使用配置淘汰机制，当 Redis 达到内存上

Redis 的回收策略（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。

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、tt- 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：

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-lru。

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。

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：

- 只有一个文件 dump.rdb，方便持久化。

- 容灾性好，一个文件可以保存到安全的磁盘。
- 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)

- 相对于数据集大时，比 AOF 的启动效率更高。

缺点：

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：

- 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。
- 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。
- 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：

- AOF 文件比 RDB 文件大，且恢复速度慢。
- 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）？

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

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：

- 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

- 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?

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?

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?

- Redis Sentinal 着眼于高可用，在 master 宕机时会自动将 slave 提升为 master，继续提供服务。
- Redis Cluster 着眼于扩展性，在单个 redis 内存不足时，使用 Cluster 进行分片存储。

Redis 集群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 支持的 Java 客户端都有哪些？官方推荐用哪个?

Redisson、Jedis、lettuce 等等，官方推荐使用 Redisson。

Jedis 与 Redisson 对比有什么优缺点?

Jedis 是 Redis 的 Java 实现的客户端，其 API 提供了比较全面的 Redis 命令的支持；Redisson 实现了分布式和可扩展的 Java 数据结构，和 Jedis 相比，功能较为简单，不支持字符串操作，不支持排序、事务、管道、分区等 Redis 特性。

Redisson 的宗旨是促进使用者对 Redis 的关注分离，从而让使用者能够将精力更集中地放在处理业务逻辑上。

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232 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?

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。

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。

队列

Reids 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。

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。

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 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？我估计现在你很想把面试官一棒打死如果你手上有一根棒球棍的话，怎么问的这么详细。但是你很克制，然后神态自若的回答道：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。到这里，面试官暗地里已经对你竖起了大拇指。但是他不知道的是此刻你却竖起了中指，在椅子背后。

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？这时候你要给予惊讶的反馈：唉，是喔，这个锁就永远得不到释放了。紧接着你需要抓一抓自己得脑袋，故作思考片刻，好像接下来的结果是你主动思考出来的，然后回答：我记得 set 指令有非常复杂的参数，这个应该是可以同时把 setnx 和 expire 合成一条指令来用的！对方这时会显露笑容，心里开始默念：嗯，这小子还不错。

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 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。

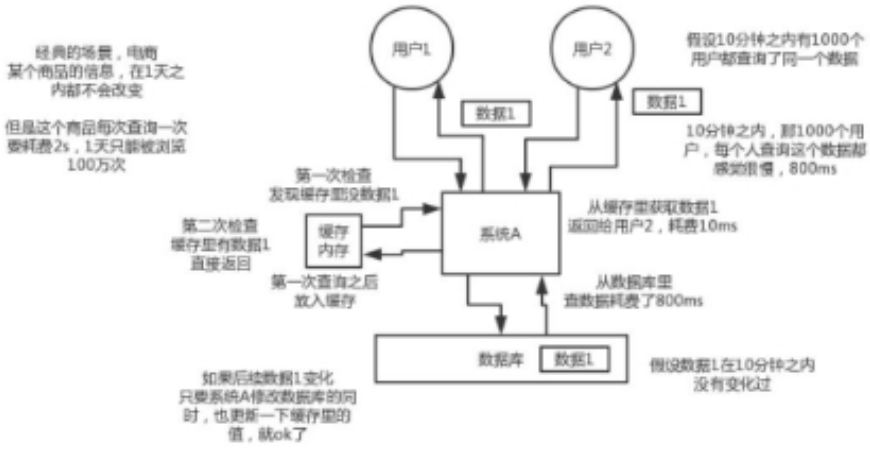
Memcached 与 Redis 的区别?

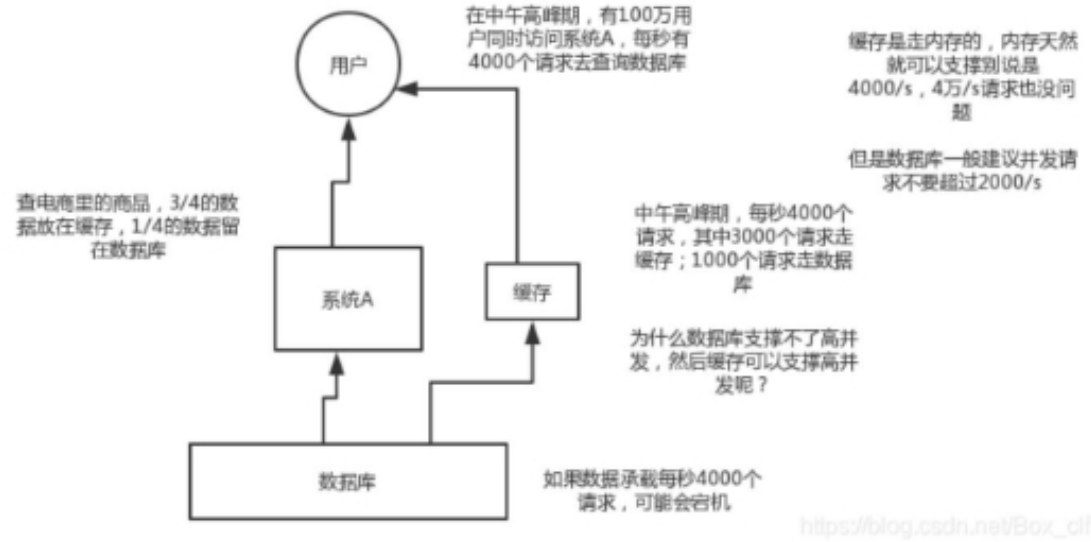
- Redis 不仅仅支持简单的 k/v 类型的数据，同时还提供 list，set，zset， hash 等数据结构的存储。而 memcache 只支持简单数据类型，需要客户端自 己处理复杂对象。
- Redis 支持数据的持久化， 可以将内存中的数据保持在磁盘中， 重启的时 候可以再次加载进行使用（ PS： 持久化在 rdb、aof）。

Redis 常见性能问题和解决方案：

- 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。
- 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。
- 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。
- 尽量避免在压力很大的主库上增加从。
- 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。

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

- 数据不一致
- 缓存雪崩
- 缓存穿透
- 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：

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。

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

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。

如何解决

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。

如果避免缓存"击穿"的问题?

缓存击穿

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：

- 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 。

- 和缓存“穿透“”的区别在于，这个 KEY 是真实存在对应的值的。

如何解决

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。

MySQL

隔离级别与锁的关系

回答这个问题，可以先阐述四种隔离级别，再阐述它们的实现原理。隔离级别就是依赖锁 和 MVCC 实现的。

实践中如何优化 MySQL?

最好是按照以下顺序优化：

- SQL 语句及索引的优化
- 数据库表结构的优化
- 系统配置的优化
- 硬件的优化

优化子查询的方法

- 用关联查询替代。
- 优化 GROUP BY 和 DISTINCT。
- 这两种查询据可以使用索引来优化，是最有效的优化方法。 - 关联查询中，使用标识列分组的效率更高。
- 如果不需要 ORDER BY，进行 GROUP BY 时加 ORDER BY NULL，MySQL 不会再进行文件排序。
- WITH ROLLUP 超级聚合，可以挪到应用程序处理。

前缀索引

- 语法：index(field(10))，使用字段值的前 10 个字符建立索引，默认是使用字段的全部内容建立索引。
- 前提：前缀的标识度高。比如密码就适合建立前缀索引，因为密码几乎各不相同。
- 实操的难度：在于前缀截取的长度。
- 我们可以利用 select count(*)/count(distinct left(password,prefixLen));，通过从调整 prefixLen 的值（从 1 自增）查看不同前缀长度的一个平均匹配度，接近 1 时就可 以了（表示一个密码的前 prefixLen 个字符几乎能确定唯一一条记录）。

MySQL 5.6 和 MySQL 5.7 对索引做了哪些优化?

- MySQL5.6 引入了索引下推优化，默认是开启的。
- 例子：user 表中（a,b,c）构成一个索引。
- select * from user where a='23' and b like '%eqw%' and c like 'dasd'。
- 解释：如果没有索引下推原则，则 MySQL 会通过 a='23' 先查询出一个对应的数据。然后返回到 MySQL 服务端。MySQL 服务端再基于两个 like 模糊查询来校验 and 查询出的数据是否符合条件。这个过程就设计到回表操作。
- 如果使用了索引下推技术，则 MySQL 会首先返回返回条件 a='23'的数据的索引，然后根据模糊查询的条件来校验索引行数据是否符合条件，如果符合条件，则直接根据 索引来定位对应的数据，如果不符合直接 reject 掉。因此，有了索引下推优化，可以 在有 like 条件的情况下，减少回表的次数。

MySQL 有关权限的表有哪几个呢?

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。

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。

table_priv 权限表：记录数据表级的操作权限。

columns_priv 权限表：记录数据列级的操作权限。

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 中都有哪些触发器?

MySQ- 数据库中有六种触发器： - Before Insert

- After Insert
- Before Update - After Update - Before Delete - After Delete

大表怎么优化？分库分表了是怎么做的？分表分库了有什么问题？ 有用到中间件么？他们的原理知道么?

当 MySQL 单表记录数过大时，数据库的 CRUD 性能会明显下降，一些常见的优化措施如 下：

- 限定数据的范围： 务必禁止不带任何限制数据范围条件的查询语句。比如：我们当用

户在查询订单历史的时候，我们可以控制在一个月的范围内。；

- 读/写分离： 经典的数据库拆分方案，主库负责写，从库负责读；
- 缓存： 使用 MySQL 的缓存，另外对重量级、更新少的数据可以考虑使用应用级别的

缓存； **还有就是通过分库分表的方式进行优化。主要有垂直分区、垂直分表、水平分区、水平分 表**

垂直分区

根据数据库里面数据表的相关性进行拆分。 例如，用户表中既有用户的登录信息又有 用户的基本信息，可以将用户表拆分成两个单独的表，甚至放到单独的库做分库。

简单来说垂直拆分是指数据表列的拆分，把一张列比较多的表拆分为多张表。 如下图 所示，这样来说大家应该就更容易理解了。

垂直拆分的优点：

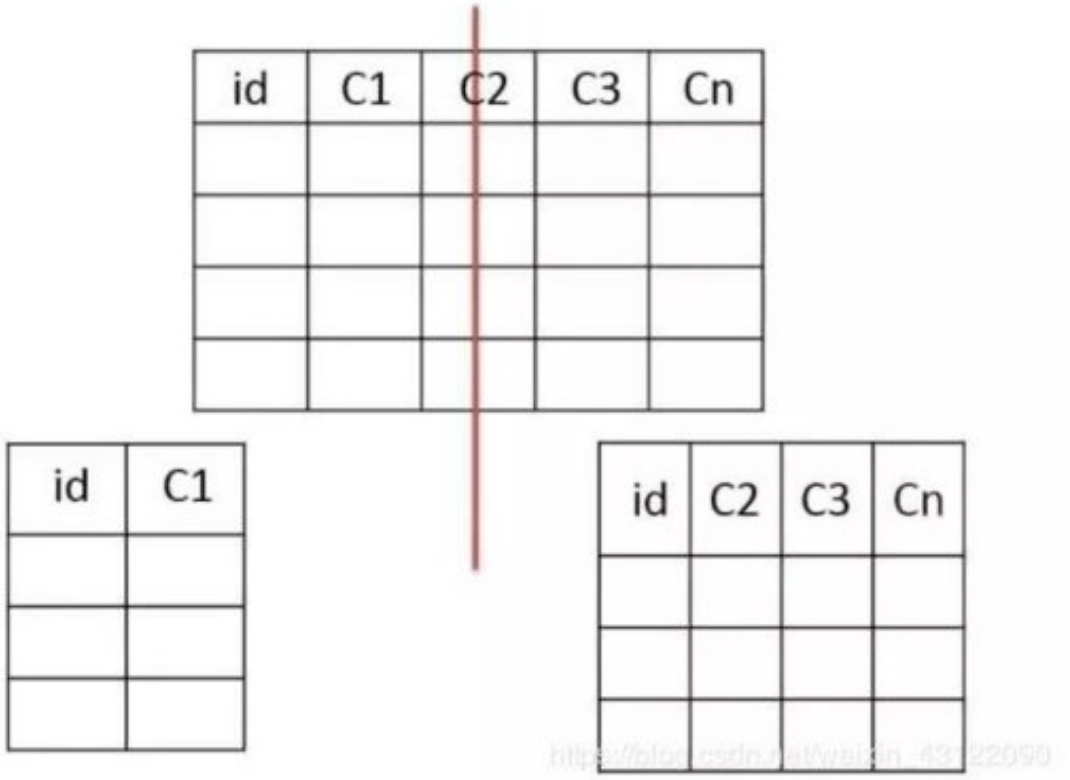
可以使得行数据变小，在查询时减少读取的 Block 数，减少 I/O 次数。此外，垂直分区可 以简化表的结构，易于维护。

垂直拆分的缺点：

主键会出现冗余，需要管理冗余列，并会引起 Join 操作，可以通过在应用层进行 Join 来 解决。此外，垂直分区会让事务变得更加复杂。

垂直分表

把主键和一些列放在一个表，然后把主键和另外的列放在另一个表中



适用场景

- 如果一个表中某些列常用，另外一些列不常用
- 可以使数据行变小，一个数据页能存储更多数据，查询时减少 I/O 次数

缺点

- 有些分表的策略基于应用层的逻辑算法，一旦逻辑算法改变，整个分表逻辑都会改

变，扩展性较差

- 对于应用层来说，逻辑算法增加开发成本
- 管理冗余列，查询所有数据需要 join 操作

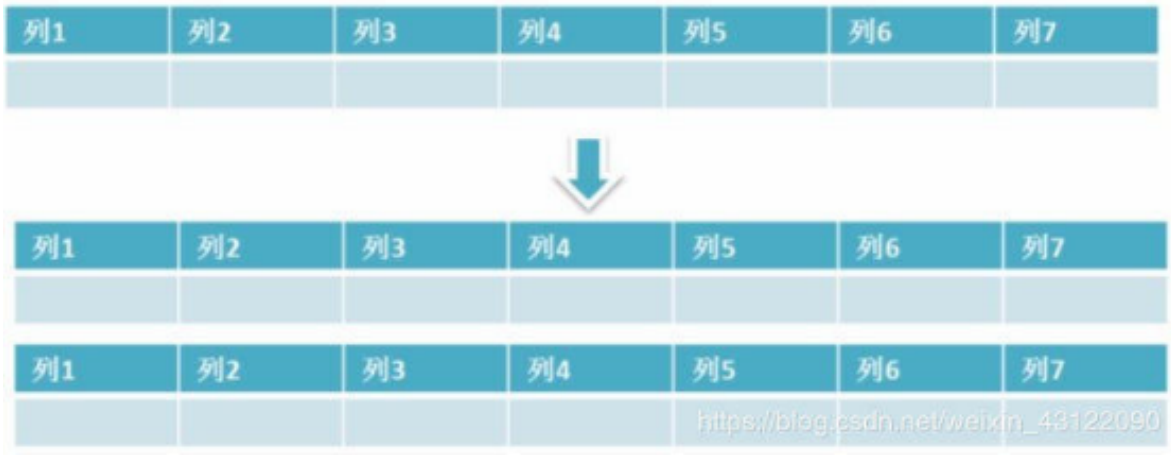
水平分区

- 保持数据表结构不变，通过某种策略存储数据分片。这样每一片数据分散到不同的表

或者库中，达到了分布式的目的。 水平拆分可以支撑非常大的数据量。

- 水平拆分是指数据表行的拆分，表的行数超过 200 万行时，就会变慢，这时可以把一

张的表的数据拆成多张表来存放。举个例子：我们可以将用户信息表拆分成多个用户 信息表，这样就可以避免单一表数据量过大对性能造成影响。



- 水品拆分可以支持非常大的数据量。需要注意的一点是

过大的问题，但由于表的数据还是在同一台机器上，其实对于提升 没有什么意义，所以 水平拆分最好分库 。

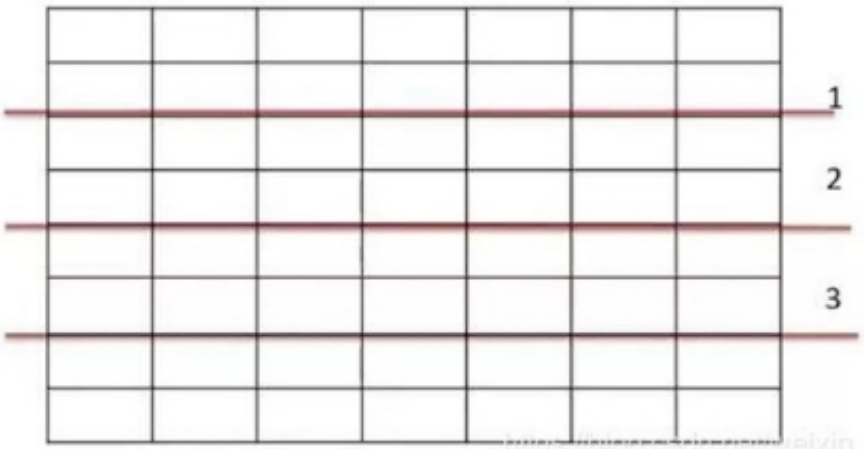
:分表仅仅是解决了单一表数据

MySQL 并发能力

- 水平拆分能够支持非常大的数据量存储，应用端改造也少，但 分片事务难以解决 ，

跨界点 Join 性能较差，逻辑复杂。

水平分表： 表很大，分割后可以降低在查询时需要读的数据和索引的页数，同时也降低了索引的层 数，提高查询次数。



适用场景

- 表中的数据本身就有独立性，例如表中分表记录各个地区的数据或者不同时期的数

据，特别是有些数据常用，有些不常用。

- 需要把数据存放在多个介质上。

水平切分的缺点

- 给应用增加复杂度，通常查询时需要多个表名，查询所有数据都需 UNION 操作。 - 在许多数据库应用中，这

种复杂度会超过它带来的优点，查询时会增加读一个索引层

的磁盘次数。

数据库分片的两种常见方案：

客户端代理：

分片逻辑在应用端，封装在 jar 包中，通过修改或者封装 JDBC 层来实现。当当网 的 Sharding-JDBC 、阿里的 TDDL是两种比较常用的实现。

中间件代理：

在应用和数据中间加了一个代理层。分片逻辑统一维护在中间件服务中。** 我们现在谈的 Mycat** 、360 的 Atlas、网易的 DDB 等等都是这种架构的实现。

分库分表后面临的问题

事务支持

分库分表后，就成了分布式事务了。如果依赖数据库本身的分布式事务管理功能去执行事 务，将付出高昂的性能代价； 如果由应用程序去协助控制，形成程序逻辑上的事务，又会 造成编程方面的负担。

跨库 join

只要是进行切分，跨节点 Join 的问题是不可避免的。但是良好的设计和切分却可以减少此 类情况的发生。解决这一问题的普遍做法是分两次查询实现。在第一次查询的结果集中找 出关联数据的 id,根据这些 id 发起第二次请求得到关联数据。 **数据迁移，容量规划，扩容等问题**

来自淘宝综合业务平台团队，它利用对 2 的倍数取余具有向前兼容的特性（如对 4 取余得 1 的数对 2 取余也是 1）来分配数据，避免了行级别的数据迁移，但是依然需要进行表级 别的迁移，同时对扩容规模和分表数量都有限制。总得来说，这些方案都不是十分的理

想，多多少少都存在一些缺点，这也从一个侧面反映出了 Sharding 扩容的难度。

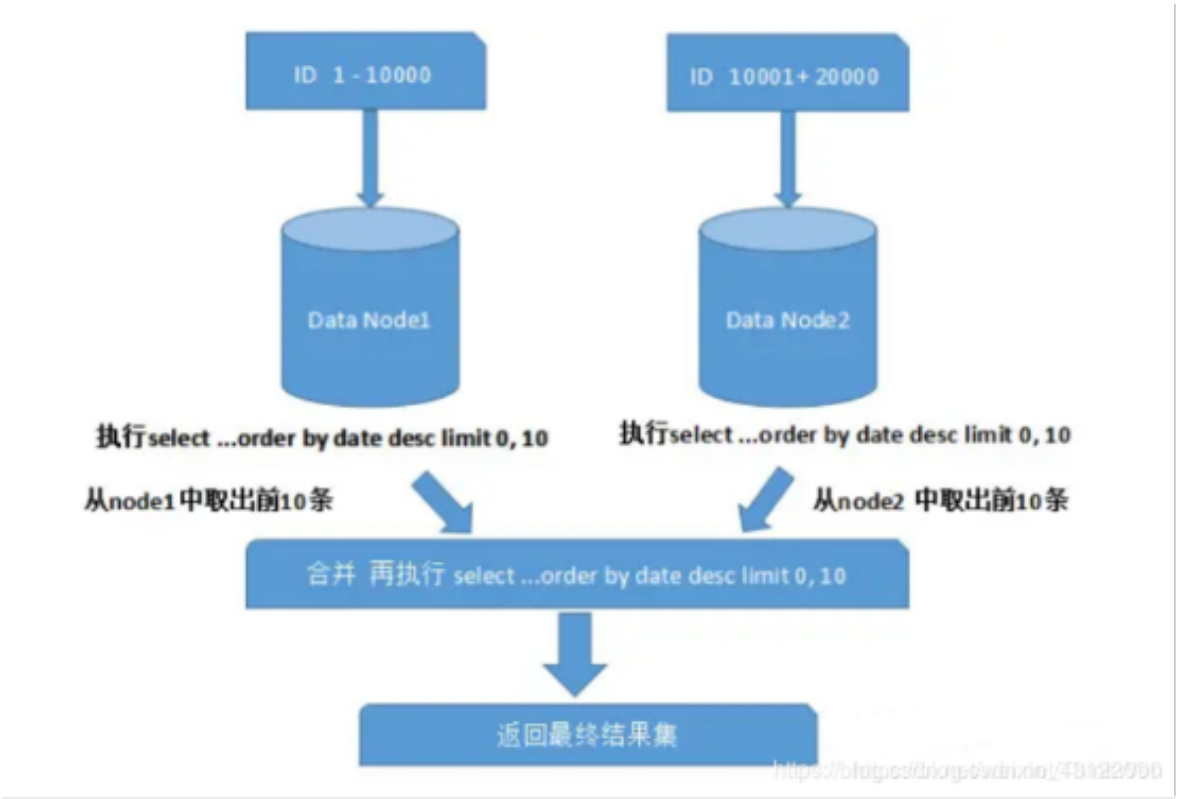
ID 问题

一旦数据库被切分到多个物理结点上，我们将不能再依赖数据库自身的主键生成机制。一 方面，某个分区数据库自生成的 ID 无法保证在全局上是唯一的；另一方面，应用程序在 插入数据之前需要先获得 ID,以便进行 SQL 路由、一些常见的主键生成策略

UUID 使用 UUID 作主键是最简单的方案，但是缺点也是非常明显的。由于 UUID 非常的 长，除占用大量存储空间外，最主要的问题是在索引上，在建立索引和基于索引进行查询 时都存在性能问题。 Twitter 的分布式自增 ID 算法 Snowflake 在分布式系统中，需要生 成全局 UID 的场合还是比较多的，twitter 的 snowflake 解决了这种需求，实现也还是很 简单的，除去配置信息，核心代码就是毫秒级时间 41 位 机器 ID 10 位 毫秒内序列 12 位。

跨分片的排序分页问题

一般来讲，分页时需要按照指定字段进行排序。当排序字段就是分片字段的时候，我们通 过分片规则可以比较容易定位到指定的分片，而当排序字段非分片字段的时候，情况就会 变得比较复杂了。为了最终结果的准确性，我们需要在不同的分片节点中将数据进行排序 并返回，并将不同分片返回的结果集进行汇总和再次排序，最后再返回给用户。如下图所 示：



B+ Tree 索引和 Hash 索引区别?

- hash 索引适合等值查询，但是无法进行范围查询。
- hash 索引没办法利用索引完成排序。
- hash 索引不支持多列联合索引的最左匹配规则。
- 如果有大量重复健值得情况下，hash 索引的效率会很低，因为哈希碰撞问题。

数据库索引的原理，为什么要用 B+树，为什么不用二叉树?

可以从几个维度去看这个问题，查询是否够快，效率是否稳定，存储数据多少，以及查找 磁盘次数，为什么不是二叉树，为什么不是平衡二叉树，为什么不是 B树，而偏偏是 B+

树呢?

为什么不是一般二叉树? 如果二叉树特殊化为一个链表，相当于全表扫描。平衡二叉树相比于二叉查找树来说，查找效率更稳定，总体的查找速度也更快。

为什么不是平衡二叉树呢? 我们知道，在内存比在磁盘的数据，查询效率快得多。如果树这种数据结构作为索引，那 我们每查找一次数据就需要从磁盘中读取一个节点，也就是我们说的一个磁盘块，但是平 衡二叉树可是每个节点只存储一个键值和数据的，如果是 B树，可以存储更多的节点数 据，树的高度也会降低，因此读取磁盘的次数就降下来啦，查询效率就快啦。 **那为什么不是 B 树而是 B+树呢?**

1）B+树非叶子节点上是不存储数据的，仅存储键值，而 B树节点中不仅存储键值，也会 存储数据。innodb 中页的默认大小是 16KB，如果不存储数据，那么就会存储更多的键 值，相应的树的阶数（节点的子节点树）就会更大，树就会更矮更胖，如此一来我们查找 数据进行磁盘的 IO 次数有会再次减少，数据查询的效率也会更快。 2）B+树索引的所有数据均存储在叶子节点，而且数据是按照顺序排列的，链表连着的。 那么 B+树使得范围查找，排序查找，分组查找以及去重查找变得异常简单。

据库三大范式是什么

- 第一范式：每个列都不可以再拆分。
- 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。
- 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：

- 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。
- db 权限表：记录各个帐号在各个数据库上的操作权限。
- table_priv 权限表：记录数据表级的操作权限。
- columns_priv 权限表：记录数据列级的操作权限。
- 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。

- 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。
- 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。
- 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

- 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，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
- 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。
- 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。
- 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?

- InnoDB索引是聚簇索引，MyISAM索引是非聚簇索引。
- 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。
- 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。
- 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

- 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。
- 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

- 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；
- 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。

- 可以通过 ALTER TABLE table_name ADD UNIQUE (column); 创建唯一索

引。

- 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。

- 可以通过 ALTER TABLE table_name ADD INDEX index_name (column);创建普通索引

- 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。

- 可以通过 ALTER TABLE table_name ADD FULLTEXT (column);创建全文索引。

MySQL 中有哪几种锁?

- 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。
- 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。
- 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?

SQL标准定义的四个隔离级别为：

- read uncommited：读到未提交数据
- read committed：脏读，不可重复读
- repeatable read：可重读
- serializable：串行事物

char 和 varchar 的区别?

- char 和 varchar 类型在存储和检索方面有所不同
- 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
- 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：

- “.frm”文件存储表定义
- 数据文件具有“.MYD”（MYData）扩展名
- 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?

- 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；
- 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；
- 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 IO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:

- 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%
- 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。

- 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）
- 然后删除其中无用数据（此过程需要不到两分钟）
- 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。
- 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

- 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
- 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。

- 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。

- 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 MyISAM 存储引擎。

MySQL会在下面这几种情况产生临时表。

- 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。
- 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。
- ORDER BY和 GROUPBY的子句不一样时也会产生临时表。
- DISTINCT 查询并且加上 ORDER BY时;
- 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
- FROM中的子查询;
- 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

- 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
- 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。
- 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 。
- 为每一张表设置一个 ID属性。
- 避免在 MHERE 字句中对字段进行 NULL
- 判断避免在 WHERE中使用!或>操作符
- 使用 BETWEEN AND 替代 IN
- 为搜索字段创建索引
- 选择正确的存储引擎，InnoDB、MyISAM、MEMORY等
- 使用 LIKE%abc%不会走索引，而使用 LIKE abc%会走索引。
- 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。
- 拆分大的 DELETE或 INSERT 语句
- 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。
- 字段设计尽可能使用 NOT NULL

- 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，MyISAM并发性比较差，使用的场景比较少主要特点是:

- 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，
- 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。
- 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。
- 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。
- 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。
- 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。
- 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

- 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。
- 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。
- 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。
- 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。
- 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。
- 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。

Mysql高可用方案有哪些?

Mysql高可用方案包括:

1. 主从复制方案

这是MySQL自身提供的一种高可用解决方案，数据同步方法采用的是MySQL replication技术。MySQL replication就是从服务器到主服务器拉取二进制日志文件，然后再将日志文件解析成相应的SQL在从服务器上重新执行一遍主服务器的操作，通过这种方式保证数据的一致性。为了达到更高的可用性，在实际的应用环境中，一般都是采用MySQL replication技术配合高可用集群软件keepalived来实现自动failover，这种方式可以实现95.000%的SLA。

1. MMM/MHA高可用方案

MMM提供了MySQL主主复制配置的监控、故障转移和管理的一套可伸缩的脚本套件。在MMM高可用方案中，典型的应用是双主多从架构，通过MySQL replication技术可以实现两个服务器互为主从，且在任何时候只有一个节点可以被写入，避免了多点写入的数据冲突。同时，当可写的主节点故障时，MMM套件可以立刻监控到，然后将服务自动切换到另一个主节点，继续提供服务，从而实现MySQL的高可用。

1. Heartbeat/SAN高可用方案

在这个方案中，处理failover的方式是高可用集群软件Heartbeat，它监控和管理各个节点间连接的网络，并监控集群服务，当节点出现故障或者服务不可用时，自动在其他节点启动集群服务。在数据共享方面，通过SAN（Storage Area Network）存储来共享数据，这种方案可以实现99.990%的SLA。

1. Heartbeat/DRBD高可用方案

这个方案处理failover的方式上依旧采用Heartbeat，不同的是，在数据共享方面，采用了基于块级别的数据同步软件DRBD来实现。DRBD是一个用软件实现的、无共享的、服务器之间镜像块设备内容的存储复制解决方案。和SAN网络不同，它并不共享存储，而是通过服务器之间的网络复制数据。

1. NDB CLUSTER高可用方案

国内用NDB集群的公司非常少，貌似有些银行有用。NDB集群不需要依赖第三方组件，全部都使用官方组件，能保证数据的一致性，某个数据节点挂掉，其他数据节点依然可以提供服务，管理节点需要做冗余以防挂掉。缺点是：管理和配置都很复杂，而且某些SQL语句例如join语句需要避免。

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。

- 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。
- 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。
- 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。
- 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。
- 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、电信部门等，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?

- 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。
- Linux 从 CPU 的角度出发，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：

- BASH命令区分大小写，而 DOS命令则不区分;
- 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
- 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?

了解即可

- 主机加电自检，加载 BIOS 硬件信息
- 读取 MBR 的引导文件(GRUB、LILO)
- 引导 Linux 内核
- 运行第一个进程 init (进程号永远为1)
- 进入相应的运行级别
- 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。

Linux 系统安装多个桌面环境有帮助吗?

通常，一个桌面环境，如 KDE或 Gnome，足以在没有问题的情况下运行。尽管系统允许从一个环境切换到另一个环境，但这对用户来说都是优先考虑的问题。有些程序在一个环境中工作而在另一个环境中无法工作，因此它也可以被视为选择使用哪个环境的一个因素。

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。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。

绝对路径用什么符号表示？当前目录、上层目录用什么表示？主目录用什么表示? 切换目录用什么命令?

绝对路径：如/etc/init.d
当前目录和上层目录：./ ../
主目录：~/
切换目录：cd

怎么查看当前进程？怎么执行退出？怎么查看当前路径?

查看当前进程：ps
执行退出：exit
查看当前路径：pwd

怎么清屏？怎么退出当前命令？怎么执行睡眠？怎么查看当前用户 id？查看指定帮助用什么命令?

清屏：clear
退出当前命令：ctrl+c 彻底退出
执行睡眠 ： ctrl+z 挂起当前进程fg 恢复后台
查看当前用户 id：”id“：查看显示目前登陆账户的 uid 和 gid 及所属分组及用户名
查看指定帮助：如 man adduser 这个很全 而且有例子；adduser --help 这个告诉你一些常用参数；info adduesr；

Ls 命令执行什么功能？可以带哪些参数，有什么区别?

ls 执行的功能：列出指定目录中的目录，以及文件
哪些参数以及区别：a 所有文件l 详细信息，包括大小字节数，可读可写可执行的权限等

建立软链接(快捷方式)，以及硬链接的命令。

软链接：ln -s slink source
硬链接：ln link source

目录创建用什么命令？创建文件用什么命令？复制文件用什么命令?

创建目录：mkdir
创建文件：典型的如 touch，vi 也可以创建文件，其实只要向一个不存在的文件输出，都会创建文件
复制文件：cp 7. 文件权限修改用什么命令？格式是怎么样的?
文件权限修改：chmod
格式如下：

$ chmod u+x file 给 file 的属主增加执行权限
$ chmod 751 file 给 file 的属主分配读、写、执行(7)的权限，给 file 的所在组分配读、执行(5)的权限，给其他用户分配执行(1)的权限
$ chmod u=rwx,g=rx,o=x file 上例的另一种形式
$ chmod =r file 为所有用户分配读权限
$ chmod 444 file 同上例
$ chmod a-wx,a+r file同上例
$ chmod -R u+r directory 递归地给 directory 目录下所有文件和子目录的属主分配读的权限

查看文件内容有哪些命令可以使用?

vi 文件名 #编辑方式查看，可修改
cat 文件名 #显示全部文件内容
more 文件名 #分页显示文件内容
less 文件名 #与 more 相似，更好的是可以往前翻页
tail 文件名 #仅查看尾部，还可以指定行数
head 文件名 #仅查看头部,还可以指定行数

随意写文件命令？怎么向屏幕输出带空格的字符串，比如”hello world”?

写文件命令：vi

向屏幕输出带空格的字符串:echo hello world

终端是哪个文件夹下的哪个文件？黑洞文件是哪个文件夹下的哪个命令?

终端 /dev/tty

黑洞文件 /dev/null

移动文件用哪个命令？改名用哪个命令?

mv mv

复制文件用哪个命令？如果需要连同文件夹一块复制呢？如果需要有提示功能呢?

cp cp -r？？？？

删除文件用哪个命令？如果需要连目录及目录下文件一块删除呢？删除空文件夹用什么命令?

rm rm -r rmdir

Linux 下命令有哪几种可使用的通配符？分别代表什么含义?

“？”可替代单个字符。

“*”可替代任意多个字符。

方括号“[charset]”可替代 charset 集中的任何单个字符，如[a-z]，[abABC]

用什么命令对一个文件的内容进行统计？(行号、单词数、字节数)

wc 命令 - c 统计字节数 - l 统计行数 - w 统计字数。

Grep 命令有什么用？如何忽略大小写？如何查找不含该串的行?

是一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹 配的行打印出来。
grep [stringSTRING] filename grep [^string] filename

Linux 中进程有哪几种状态？在 ps 显示出来的信息中，分别用什么符号表示的?

（1）、不可中断状态：进程处于睡眠状态，但是此刻进程是不可中断的。不可中断， 指进程不响应异步信号。
（2）、暂停状态/跟踪状态：向进程发送一个 SIGSTOP 信号，它就会因响应该信号 而进入 TASK_STOPPED 状态;当进程正在被跟踪时，它处于 TASK_TRACED 这个特殊的状态。
“正在被跟踪”指的是进程暂停下来，等待跟踪它的进程对它进行操作。

（3）、就绪状态：在 run_queue 队列里的状态

（4）、运行状态：在 run_queue 队列里的状态
（5）、可中断睡眠状态：处于这个状态的进程因为等待某某事件的发生（比如等待 socket 连接、等待信号量），而被挂起
（6）、zombie 状态（僵尸）：父亲没有通过 wait 系列的系统调用会顺便将子进程的尸体（task_struct）也释放掉
（7）、退出状态

D 不可中断 Uninterruptible（usually IO）
R 正在运行，或在队列中的进程
S 处于休眠状态
T 停止或被追踪
Z 僵尸进程
W 进入内存交换（从内核 2.6 开始无效）
X 死掉的进程

怎么使一个命令在后台运行?

一般都是使用 & 在命令结尾来让程序自动运行。(命令后可以不追加空格)

利用 ps 怎么显示所有的进程? 怎么利用 ps 查看指定进程的信息?

ps -ef (system v 输出)

ps -aux bsd 格式输出

ps -ef | grep pid

哪个命令专门用来查看后台任务?

job -l

把后台任务调到前台执行使用什么命令?把停下的后台任务在后台执行起来用什么命令?

把后台任务调到前台执行 fg

把停下的后台任务在后台执行起来 bg

终止进程用什么命令? 带什么参数?

kill [-s <信息名称或编号>][程序] 或 kill [-l <信息编号>]

kill-9 pid

怎么查看系统支持的所有信号?

kill -l

搜索文件用什么命令? 格式是怎么样的?

find <指定目录> <指定条件> <指定动作>

whereis 加参数与文件名

locate 只加文件名

find 直接搜索磁盘，较慢。

find / -name "string*"

查看当前谁在使用该主机用什么命令? 查找自己所在的终端信息用什么命令?

查找自己所在的终端信息：who am i

查看当前谁在使用该主机：who

使用什么命令查看用过的命令列表?

history

使用什么命令查看磁盘使用空间? 空闲空间呢?

df -hl
文件系统 容量 已用 可用 已用% 挂载点
Filesystem Size Used Avail Use% Mounted on /dev/hda2 45G 19G 24G 44% /
/dev/hda1 494M 19M 450M 4% /boot

使用什么命令查看网络是否连通?

netstat

使用什么命令查看 ip 地址及接口信息?

ifconfig

查看各类环境变量用什么命令?

查看所有 env
查看某个，如 home：env $HOME

通过什么命令指定命令提示符?

\u：显示当前用户账号
\h：显示当前主机名
\W：只显示当前路径最后一个目录
\w：显示当前绝对路径（当前用户目录会以~代替）
$PWD：显示当前全路径
$：显示命令行'$'或者'#'符号
#：下达的第几个命令
\d：代表日期，格式为week day month date，例如："MonAug1"
\t：显示时间为24小时格式，如：HH：MM：SS
\T：显示时间为12小时格式
\A：显示时间为24小时格式：HH：MM
\v：BASH的版本信息 如export PS1='[\u@\h\w#]$'

查找命令的可执行文件是去哪查找的? 怎么对其进行设置及添加?

whereis [-bfmsu][-B <目录>...][-M <目录>...][-S <目录>...][文件...]

补充说明：whereis 指令会在特定目录中查找符合条件的文件。这些文件的烈性应属于原始代码，二进制文件，或是帮助文件。

-b 只查找二进制文件。
-B<目录> 只在设置的目录下查找二进制文件。-f 不显示文件名前的路径名称。
-m 只查找说明文件。
-M<目录> 只在设置的目录下查找说明文件。-s 只查找原始代码文件。
-S<目录> 只在设置的目录下查找原始代码文件。-u 查找不包含指定类型的文件。
which 指令会在 PATH 变量指定的路径中，搜索某个系统命令的位置，并且返回第一个搜索结果。
-n 指定文件名长度，指定的长度必须大于或等于所有文件中最长的文件名。
-p 与-n 参数相同，但此处的包括了文件的路径。-w 指定输出时栏位的宽度。
-V 显示版本信息

通过什么命令查找执行命令?

which 只能查可执行文件

whereis 只能查二进制文件、说明文档，源文件等

怎么对命令进行取别名?

alias la='ls -a'

du 和 df 的定义，以及区别?

du 显示目录或文件的大小

df 显示每个<文件>所在的文件系统的信息，默认是显示所有文件系统。
（文件系统分配其中的一些磁盘块用来记录它自身的一些数据，如 i 节点，磁盘分布图，间接块，超级块等。这些数据对大多数用户级的程序来说是不可见的，通常称为 Meta Data。） du 命令是用户级的程序，它不考虑 Meta Data，而 df 命令则查看文件系统的磁盘分配图并考虑 Meta Data。
df 命令获得真正的文件系统数据，而 du 命令只查看文件系统的部分情况。

awk 详解。

awk '{pattern + action}' {filenames}
#cat /etc/passwd |awk -F ':' '{print $1"\t"$7}' //-F 的意思是以':'分隔 root /bin/bash
daemon /bin/sh 搜索/etc/passwd 有 root 关键字的所有行

#awk -F: '/root/' /etc/passwd root❌0:0:root:/root:/bin/bash

当你需要给命令绑定一个宏或者按键的时候，应该怎么做呢?

可以使用bind命令，bind可以很方便地在shell中实现宏或按键的绑定。

在进行按键绑定的时候，我们需要先获取到绑定按键对应的字符序列。

比如获取F12的字符序列获取方法如下：先按下Ctrl+V,然后按下F12 .我们就可以得到F12的字符序列 ^[[24~。

接着使用bind进行绑定。

[root@localhost ~]# bind '"\e[24~":"date"'

注意：相同的按键在不同的终端或终端模拟器下可能会产生不同的字符序列。

【附】也可以使用showkey -a命令查看按键对应的字符序列。

如果一个linux新手想要知道当前系统支持的所有命令的列表，他需要怎么做?

使用命令compgen -c，可以打印出所有支持的命令列表。

[root@localhost ~]$ compgen -c
l.
ll
ls
which
if
then
else
elif
fi
case

esac
for
select
while
until
do
done
…

如果你的助手想要打印出当前的目录栈，你会建议他怎么做?

使用Linux 命令dirs可以将当前的目录栈打印出来。

[root@localhost ~]# dirs
/usr/share/X11
【附】： 目录栈通过pushd popd 来操作。

你的系统目前有许多正在运行的任务，在不重启机器的条件下，有什么方法可以把所有正在运行的进程移除呢?

使用linux命令 'disown -r '可以将所有正在运行的进程移除。

bash shell 中的hash 命令有什么作用?

linux命令'hash'管理着一个内置的哈希表，记录了已执行过的命令的完整路径, 用该命令可以打印出你所使用过的命令以及执行的次数。

[root@localhost ~]# hash
hits command
2 /bin/ls
2 /bin/su

哪一个bash内置命令能够进行数学运算。

bash shell 的内置命令let 可以进行整型数的数学运算。

#! /bin/bash
…
…
let c=a+b
…
…

怎样一页一页地查看一个大文件的内容呢?

通过管道将命令”cat file_name.txt” 和 ’more’ 连接在一起可以实现这个需要.

[root@localhost ~]# cat file_name.txt | more

数据字典属于哪一个用户的?

数据字典是属于’SYS’用户的，用户‘SYS’ 和 ’SYSEM’是由系统默认自动创建的

怎样查看一个linux命令的概要与用法？假设你在/bin目录中偶然看到一个你从没见过的的命令，怎样才能知道它的作用和用法呢?

使用命令whatis 可以先出显示出这个命令的用法简要，比如，你可以使用whatis zcat 去查看‘zcat’的介绍以及使用简要。

[root@localhost ~]# whatis zcat
zcat [gzip] (1) – compress or expand files

使用哪一个命令可以查看自己文件系统的磁盘空间配额呢?

使用命令repquota 能够显示出一个文件系统的配额信息

【附】只有root用户才能够查看其它用户的配额。

说一下异步和非阻塞的区别?

- 异步和非阻塞的区别:

1. 异步：调用在发出之后，这个调用就直接返回，不管有无结果；异步是过程。
2. 非阻塞：关注的是程序在等待调用结果（消息，返回值）时的状态，指在不能立刻得到结果之前，该调用不会阻塞当前线程。

- 同步和异步的区别：

1. 步：一个服务的完成需要依赖其他服务时，只有等待被依赖的服务完成后，才算完成，这是一种可靠的服务序列。要么成功都成功，失败都失败，服务的状态可以保持一致。
2. 异步：一个服务的完成需要依赖其他服务时，只通知其他依赖服务开始执行，而不需要等待被依赖的服务完成，此时该服务就算完成了。被依赖的服务是否最终完成无法确定，一次它是一个不可靠的服务序列。

- 消息通知中的同步和异步：

1. 同步：当一个同步调用发出后，调用者要一直等待返回消息（或者调用结果）通知后，才能进行后续的执行。
2. 异步：当一个异步过程调用发出后，调用者不能立刻得到返回消息（结果）。在调用结束之后，通过消息回调来通知调用者是否调用成功。

- 阻塞与非阻塞的区别：

1. 阻塞：阻塞调用是指调用结果返回之前，当前线程会被挂起，一直处于等待消息通知，不能够执行其他业务，函数只有在得到结果之后才会返回。
2. 非阻塞：非阻塞和阻塞的概念相对应，指在不能立刻得到结果之前，该函数不会阻塞当前线程，而会立刻返回。

同步与异步是对应的，它们是线程之间的关系，两个线程之间要么是同步的，要么是异步的。

阻塞与非阻塞是对同一个线程来说的，在某个时刻，线程要么处于阻塞，要么处于非阻塞。

阻塞是使用同步机制的结果，非阻塞则是使用异步机制的结果。

滑动窗口的概念以及应用?

滑动窗口概念不仅存在于数据链路层，也存在于传输层，两者有不同的协议，但基本原理是相近的。其中一个重要区别是，一个是针对于帧的传送，另一个是字节数据的传送。

滑动窗口（Sliding window）是一种流量控制技术。早期的网络通信中，通信双方不会考虑网络的拥挤情况直接发送数据。由于大家不知道网络拥塞状况，同时发送数据，导致中间节点阻塞掉包，谁也发不了数据，所以就有了滑动窗口机制来解决此问题。参见滑动窗口如何根据网络拥塞发送数据仿真视频。

滑动窗口协议是用来改善吞吐量的一种技术，即容许发送方在接收任何应答之前传送附加的包。接收方告诉发送方在某一时刻能送多少包（称窗口尺寸）。

CP中采用滑动窗口来进行传输控制，滑动窗口的大小意味着接收方还有多大的缓冲区可以用于接收数据。发送方可以通过滑动窗口的大小来确定应该发送多少字节的数据。当滑动窗口为0时，发送方一般不能再发送数据报，但有两种情况除外，一种情况是可以发送紧急数据，例如，允许用户终止在远端机上的运行进程。另一种情况是发送方可以发送一个1字节的数据报来通知接收方重新声明它希望接收的下一字节及发送方的滑动窗口大小。

Epoll原理.

开发高性能网络程序时，windows开发者们言必称Iocp，linux开发者们则言必称Epoll。大家都明白Epoll是一种IO多路复用技术，可以非常高效的处理数以百万计的Socket句柄，比起以前的Select和Poll效率提高了很多。

先简单了解下如何使用C库封装的3个epoll系统调用。

```
int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event *events,int maxevents, int timeout);
```

使用起来很清晰，首先要调用`epoll_create`建立一个epoll对象。参数size是内核保证能够正确处理的最大句柄数，多于这个最大数时内核可不保证效果。 epoll_ctl可以操作上面建立的epoll，例如，将刚建立的`socket`加入到epoll中让其监控，或者把 epoll正在监控的某个socket句柄移出epoll，不再监控它等等。

`epoll_wait`在调用时，在给定的timeout时间内，当在监控的所有句柄中有事件发生时，就返回用户态的进程。

从调用方式就可以看到epoll相比select/poll的优越之处是,因为后者每次调用时都要传递你所要监控的所有socket给select/poll系统调用，这意味着需要将用户态的socket列表copy到内核态，如果以万计的句柄会导致每次都要copy几十几百KB的内存到内核态，非常低效。而我们调用`epoll_wait`时就相当于以往调用select/poll，但是这时却不用传递socket句柄给内核，因为内核已经在epoll_ctl中拿到了要监控的句柄列表。

所以，实际上在你调用`epoll_create`后，内核就已经在内核态开始准备帮你存储要监控的句柄了，每次调用`epoll_ctl`只是在往内核的数据结构里塞入新的socket句柄。

在内核里，一切皆文件。所以，epoll向内核注册了一个文件系统，用于存储上述的被监控socket。当你调用epoll_create时，就会在这个虚拟的epoll文件系统里创建一个file结点。当然这个file不是普通文件，它只服务于epoll。

epoll在被内核初始化时（操作系统启动），同时会开辟出epoll自己的内核高速cache区，用于安置每一个我们想监控的socket，这些socket会以红黑树的形式保存在内核cache里，以支持快速的查找、插入、删除。这个内核高速cache区，就是建立连续的物理内存页，然后在之上建立slab层，通常来讲，就是物理上分配好你想要的size的内存对象，每次使用时都是使用空闲的已分配好的对象。

```
static int __init eventpoll_init(void)  {
    ... ...

    /* Allocates slab cache used to allocate "struct epitem" items */
    epi_cache = kmem_cache_create("eventpoll_epi", sizeof(struct epitem),
            0, SLAB_HWCACHE_ALIGN|EPI_SLAB_DEBUG|SLAB_PANIC,
            NULL, NULL);

    /* Allocates slab cache used to allocate "struct eppoll_entry" */
    pwq_cache = kmem_cache_create("eventpoll_pwq",
            sizeof(struct eppoll_entry), 0,
            EPI_SLAB_DEBUG|SLAB_PANIC, NULL, NULL);
 ... ...
 }
```

epoll的高效就在于，当我们调用`epoll_ctl`往里塞入百万个句柄时，`epoll_wait`仍然可以飞快的返回，并有效的将发生事件的句柄给我们用户。这是由于我们在调用`epoll_create`时，内核除了帮我们在epoll文件系统里建了个file结点，在内核cache里建了个红黑树用于存储以后epoll_ctl传来的socket外，还会再建立一个list链表，用于存储准备就绪的事件，当epoll_wait调用时，仅仅观察这个list链表里有没有数据即可。有数据就返回，没有数据就sleep，等到timeout时间到后即使链表没数据也返回。所以，epoll_wait非常高效。

而且，通常情况下即使我们要监控百万计的句柄，大多一次也只返回很少量的准备就绪句柄而已，所以，epoll_wait仅需要从内核态copy少量的句柄到用户态而已，因此就会非常的高效！

然而,这个准备就绪list链表是怎么维护的呢？当我们执行epoll_ctl时，除了把socket放到epoll文件系统里file对象对应的红黑树上之外，还会给内核中断处理程序注册一个回调函数，告诉内核，如果这个句柄的中断到了，就把它放到准备就绪list链表里。所以，当一个socket上有数据到了，内核在把网卡上的数据copy到内核中后就来把socket插入到准备就绪链表里了。

如此，一个红黑树，一张准备就绪句柄链表，少量的内核cache，就帮我们解决了大并发下的socket处理问题。执行`epoll_create`时，创建了红黑树和就绪链表，执行epoll_ctl时，如果增加socket句柄，则检查在红黑树中是否存在，存在立即返回，不存在则添加到树干上，然后向内核注册回调函数，用于当中断事件来临时向准备就绪链表中插入数据。执行epoll_wait时立刻返回准备就绪链表里的数据即可。

最后看看epoll独有的两种模式LT和ET。无论是LT和ET模式，都适用于以上所说的流程。区别是，LT模式下，只要一个句柄上的事件一次没有处理完，会在以后调用epoll_wait时每次返回这个句柄，而ET模式仅在第一次返回。

当一个socket句柄上有事件时，内核会把该句柄插入上面所说的准备就绪list链表，这时我们调用`epoll_wait`，会把准备就绪的socket拷贝到用户态内存，然后清空准备就绪list链表，最后，`epoll_wait`需要做的事情，就是检查这些socket，如果不是ET模式（就是LT模式的句柄了），并且这些socket上确实有未处理的事件时，又把该句柄放回到刚刚清空的准备就绪链表了。所以，非ET的句柄，只要它上面还有事件，epoll_wait每次都会返回。而ET模式的句柄，除非有新中断到，即使socket上的事件没有处理完，也是不会每次从epoll_wait返回的。

因此epoll比select的提高实际上是一个用空间换时间思想的具体应用.对比阻塞IO的处理模型, 可以看到采用了多路复用IO之后, 程序可以自由的进行自己除了IO操作之外的工作, 只有到IO状态发生变化的时候由多路复用IO进行通知, 然后再采取相应的操作, 而不用一直阻塞等待IO状态发生变化,提高效率.

负载均衡原理是什么?

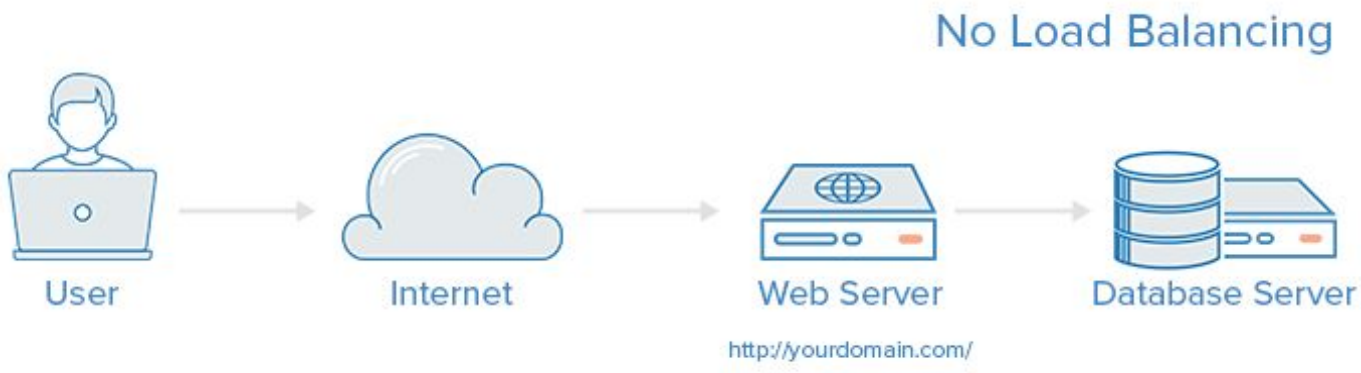
负载均衡Load Balance）是高可用网络基础架构的关键组件，通常用于将工作负载分布到多个服务器来提高网站、应用、数据库或其他服务的性能和可靠性。负载均衡，其核心就是网络流量分发，分很多维度。

负载均衡（Load Balance）通常是分摊到多个操作单元上进行执行，例如Web服务器、FTP服务器、企业关键应用服务器和其它关键任务服务器等，从而共同完成工作任务。

负载均衡是建立在现有网络结构之上，它提供了一种廉价有效透明的方法扩展网络设备和服务器的带宽、增加吞吐量、加强网络数据处理能力、提高网络的灵活性和可用性。

通过一个例子详细介绍:

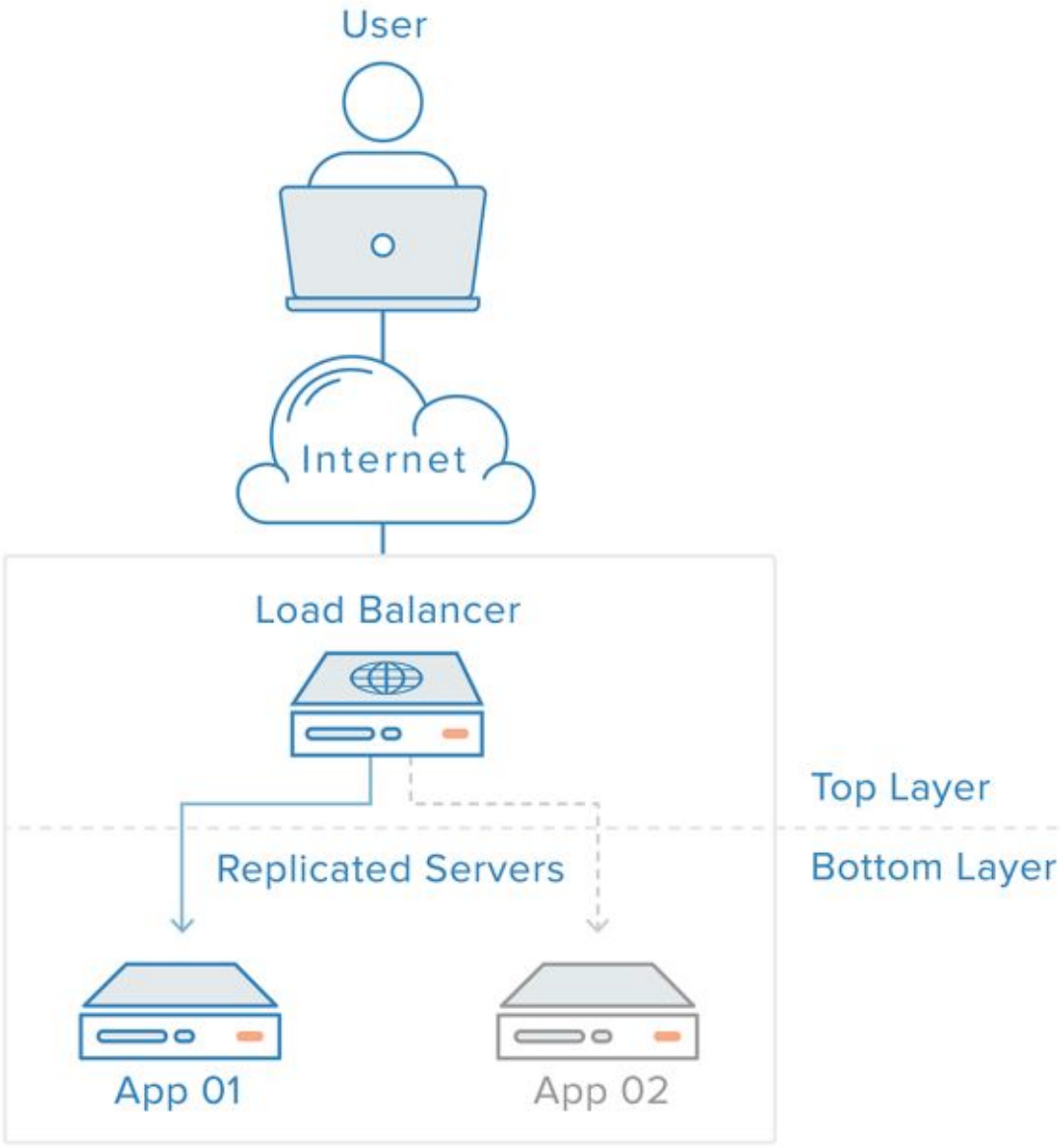
- 没有负载均衡 web 架构



在这里用户是直连到 web 服务器，如果这个服务器宕机了，那么用户自然也就没办法访问了。 另外，如果同时有很多用户试图访问服务器，超过了其能处理的极限，就会出现加载速度缓慢或根本无法连接的情况。

而通过在后端引入一个负载均衡器和至少一个额外的 web 服务器，可以缓解这个故障。 通常情况下，所有的后端服务器会保证提供相同的内容，以便用户无论哪个服务器响应，都能收到一致的内容。

- 有负载均衡 web 架构



用户访问负载均衡器，再由负载均衡器将请求转发给后端服务器。在这种情况下，单点故障现在转移到负载均衡器上了。 这里又可以通过引入第二个负载均衡器来缓解。

那么负载均衡器的工作方式是什么样的呢,负载均衡器又可以处理什么样的请求?

负载均衡器的管理员能主要为下面四种主要类型的请求设置转发规则：

- HTTP (七层)
- HTTPS (七层)
- TCP (四层)
- UDP (四层)

负载均衡器如何选择要转发的后端服务器?

负载均衡器一般根据两个因素来决定要将请求转发到哪个服务器。首先，确保所选择的服务器能够对请求做出响应，然后根据预先配置的规则从健康服务器池（healthy pool）中进行选择。

因为，负载均衡器应当只选择能正常做出响应的后端服务器，因此就需要有一种判断后端服务器是否健康的方法。为了监视后台服务器的运行状况，运行状态检查服务会定期尝试使用转发规则定义的协议和端口去连接后端服务器。 如果，服务器无法通过健康检查，就会从池中剔除，保证流量不会被转发到该服务器，直到其再次通过健康检查为止。

负载均衡算法

负载均衡算法决定了后端的哪些健康服务器会被选中。 其中常用的算法包括：

- Round Robin（轮询）：为第一个请求选择列表中的第一个服务器，然后按顺序向下移动列表直到结尾，然后循环。
- Least Connections（最小连接）：优先选择连接数最少的服务器，在普遍会话较长的情况下推荐使用。
- Source：根据请求源的 IP 的散列（hash）来选择要转发的服务器。这种方式可以一定程度上保证特定用户能连接到相同的服务器。

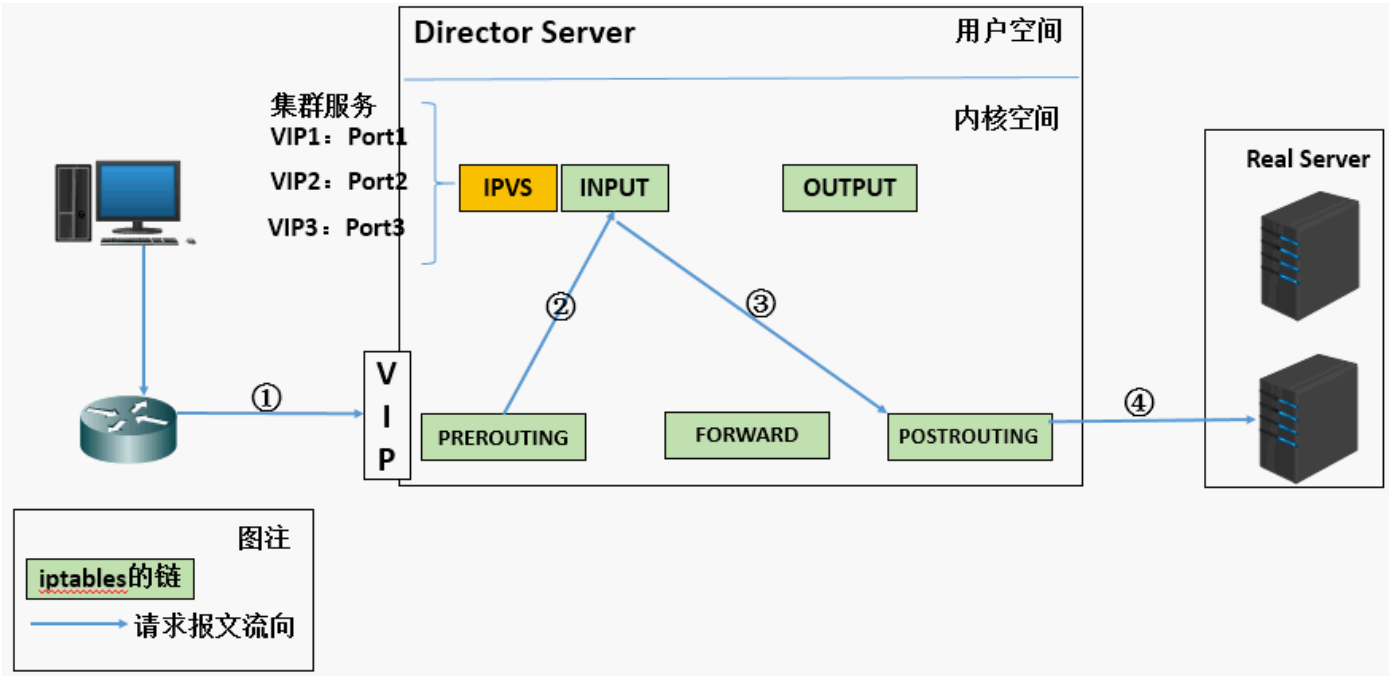
如果你的应用需要处理状态而要求用户能连接到和之前相同的服务器。可以通过 Source 算法基于客户端的 IP 信息创建关联，或者使用粘性会话（sticky sessions）。

除此之外，想要解决负载均衡器的单点故障问题，可以将第二个负载均衡器连接到第一个上，从而形成一个集群。

LVS相关了解.

LVS是 Linux Virtual Server 的简称，也就是Linux虚拟服务器。这是一个由章文嵩博士发起的一个开源项目，它的官方网站是LinuxVirtualServer现在 LVS 已经是 Linux 内核标准的一部分。使用 LVS 可以达到的技术目标是：通过 LVS 达到的负载均衡技术和 Linux 操作系统实现一个高性能高可用的 Linux 服务器集群，它具有良好的可靠性、可扩展性和可操作性。 从而以低廉的成本实现最优的性能。LVS 是一个实现负载均衡集群的开源软件项目，LVS架构从逻辑上可分为调度层、Server集群层和共享存储。

LVS的基本工作原理:



1. 当用户向负载均衡调度器（Director Server）发起请求，调度器将请求发往至内核空间
2. PREROUTING链首先会接收到用户请求，判断目标IP确定是本机IP，将数据包发往INPUT链
3. IPVS是工作在INPUT链上的，当用户请求到达INPUT时，IPVS会将用户请求和自己已定义好的集群服务进行比对，如果用户请求的就是定义的集群服务，那么此时IPVS会强行修改数据包里的目标IP地址及端口，并将新的数据包发往POSTROUTING链
4. POSTROUTING链接收数据包后发现目标IP地址刚好是自己的后端服务器，那么此时通过选路，将数据包最终

发送给后端的服务器

LVS的组成:

LVS 由2部分程序组成，包括 `ipvs` 和 `ipvsadm`。

1. ipvs(ip virtual server)：一段代码工作在内核空间，叫ipvs，是真正生效实现调度的代码。
2. ipvsadm：另外一段是工作在用户空间，叫ipvsadm，负责为ipvs内核框架编写规则，定义谁是集群服务，而谁是后端真实的服务器(Real Server)

详细的LVS的介绍可以参考LVS详解.

网络和操作系统

进程和线程的区别?

- 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。
- 切换：线程上下文切换比进程上下文切换要快得多。
- 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。
- 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?

- 线程和进程都是同步机制，而协程是异步机制。
- 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。
- 一个线程可以有多个协程，一个进程也可以有多个协程。
- 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。
- 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

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?

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。

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。（4）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。
- 7）SIGTERM：结束进程信号。shell 下执行 kill 进程 pid 发送该信号。
- 8）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。
- 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。
- 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。
- 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。
- 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：

- 管道：速度慢，容量有限；

- Socket：任何进程间都能通讯，但速度慢；
- 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；
- 信号量：不能传递复杂消息，只能用来同步；
- 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：

- 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。
- 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:

- 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；
- 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；
- 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。

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?

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。

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。**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：

- 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。

- 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：

- 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；
- 进入临界区的进程要在有限时间内退出。
- 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：（有一个条件不成立，则不会产生死锁）

- 互斥条件：一个资源一次只能被一个进程使用
- 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
- 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
- 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**：
- 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。
- 检测死锁并且恢复。
- 仔细地对资源进行动态分配，以避免死锁。
- 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?

- **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。
- **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。
- **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。

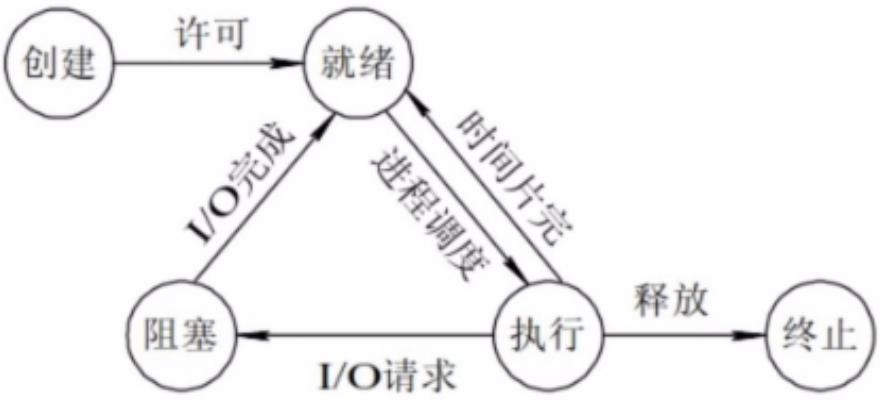
时间片轮转：将所有就绪进程按 FCFS的原则排成一个队列，每次调度时，把 CPU时间分配给队首进程，该进程可以执行一个时间片。当时间片用完时，由

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。

- 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。
- **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。



- 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。
- 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。
- 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。

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。

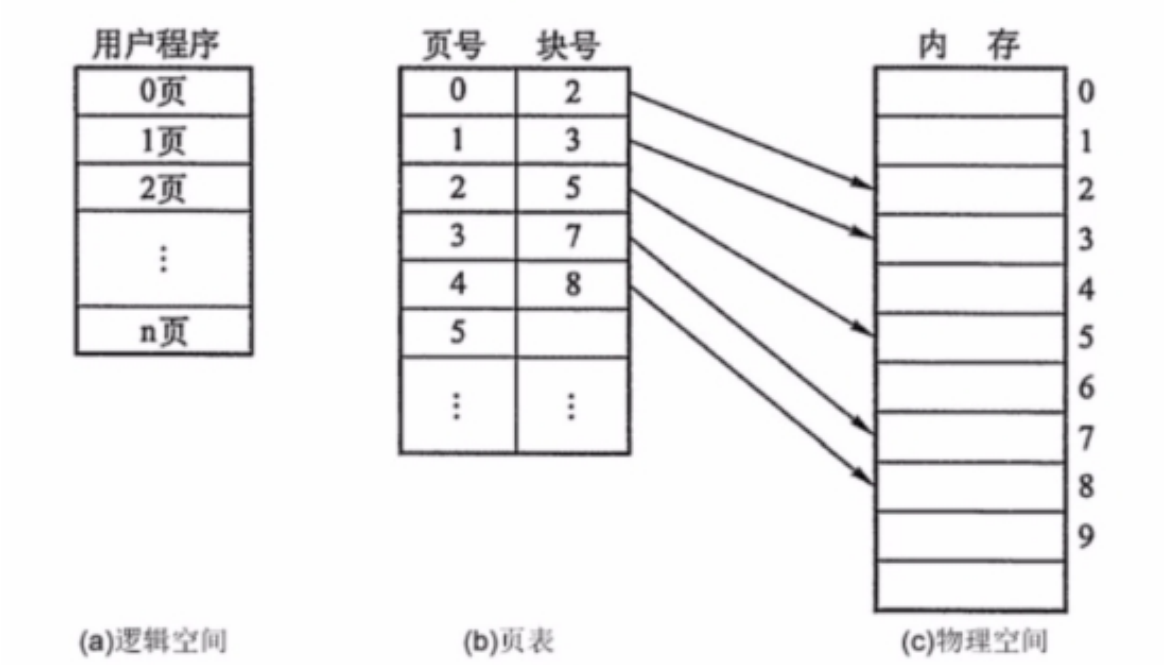
阻塞态→就绪态：则是等待的条件已满足，只需分配到处理器后就能运行。

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。

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。

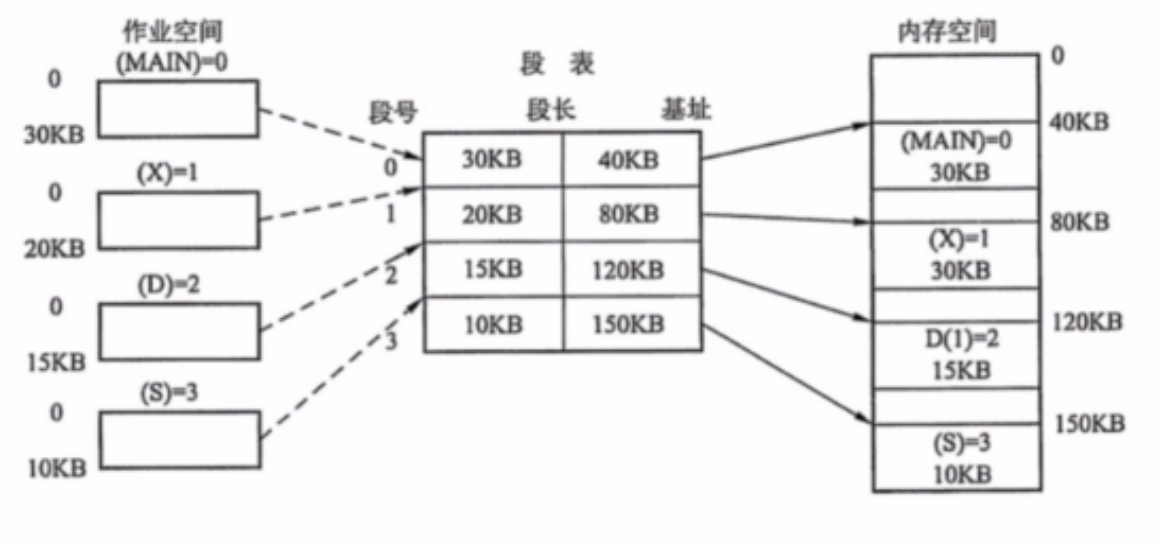
什么是分页?

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?

- 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。
- 分页的地址空间是一维地址空间，分段是二维的。
- 页的大小不可变，段的大小可以动态改变。
- 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：

- 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。
- 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：

- **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。
- **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。
- **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。
- **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：

- 程序崩溃，导致拒绝额服务
- 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：

- 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。
- 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。
- 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。
- 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。
- 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。
- 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?

- 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。
- 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?

- 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。
- 开中断：以便执行中断时能响应较高级别的中断请求。
- 中断处理
- 关中断：保证恢复现场时不被新中断打扰
- 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?

- 轮询：CPU对特定设备轮流询问。中断：通过特定事件提醒 CPU。
- 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。

RocketMQ 面试题

多个 MQ 如何选型?

MQ	描述
RabbitMQ	erlang 开发，对消息堆积的支持并不好，当大量消息积压的时候，会导致 RabbitMQ 的性能急剧下 降。每秒钟可以处理几万到十几万条消息。
RocketMQ	Java 开发，面向互联网集群化功能丰富，对在线业务的响应时延做了很多的优化，大多数情况下可以做 到毫秒级的响应，每秒钟大概能处理几十万条消息。
Kafka	Scala 开发，面向日志功能丰富，性能最高。当你的业务场景中，每秒钟消息数量没有那么多的时候， Kafka 的时延反而会比较高。所以，Kafka 不太适合在线业务场景。
ActiveMQ	Java 开发，简单，稳定，性能不如前面三个。小型系统用也可以，但是不推荐。推荐用互联网主流的。

为什么要使用 MQ?

因为项目比较大，做了分布式系统，所有远程服务调用请求都是同步执行经常出问题，所 以引入了 MQ。

作用	描述
解耦	系统耦合度降低，没有强依赖关系。
异步	不需要同步执行的远程调用可以有效提高响应时间。
削峰	请求达到峰值后，后端 service 还可以保持固定消费速率消费，不会被压垮。

RocketMQ 由哪些角色组成，每个角色作用和特点是什么?

角色	作用
Nameserver	无状态，动态列表；这是和 ZooKeeper 的重要区别之一。ZooKeeper 是有状态的。
Producer	消息生产者，负责发消息到 Broker。
Broker	就是 MQ 本身，负责收发消息、持久化消息等。
Consumer	消息消费者，负责从 Broker 上拉取消息进行消费，消费完进行 ack。

RocketMQ 中的 Topic 和 JMS 的 queue 有什么区别?

queue 就是来源于数据结构的 FIFO 队列。而 Topic 是个抽象的概念，每个 Topic 底层对 应 N 个 queue，而数据也真实存在 queue 上的。

RocketMQ 消费模式有几种?

消费模型由 Consumer 决定，消费维度为 Topic。

- 集群消费

1.一条消息只会被同 Group 中的一个 Consumer 消费

2.多个 Group 同时消费一个 Topic 时，每个 Group 都会有一个 Consumer 消费到数据。

- 广播消费

消息将对一个 Consumer Group 下的各个 Consumer 实例都消费一遍。即即使这些Consumer 属于同一个 Consumer Group，消息也会被 Consumer Group 中的每个 Consumer 都消费一次。

Broker 如何处理拉取请求的?

Consumer 首次请求 Broker。

- Broker 中是否有符合条件的消息
- 有
    - 响应 Consumer。
    - 等待下次 Consumer 的请求。
- 没有
    - DefaultMessageStore#ReputMessageService#run 方法。 - PullRequestHoldService 来 Hold 连接，每个 5s 执行一次检查pullRequestTable 有没有消息，有的话立即推送。
    - 每隔 1ms 检查 commitLog 中是否有新消息，有的话写入到pullRequestTable。
    - 当有新消息的时候返回请求。
    - 挂起 consumer 的请求，即不断开连接，也不返回数据。
    - 使用 consumer 的 offset。

RocketMQ 如何做负载均衡?

通过 Topic 在多 Broker 中分布式存储实现。

producer 端

发送端指定 message queue 发送消息到相应的 broker，来达到写入时的负载均衡

- 提升写入吞吐量，当多个 producer 同时向一个 broker 写入数据的时候，性能会下降 - 消息分布在多 broker 中，为负载消费做准备

默认策略是随机选择：

- producer 维护一个 index
- 每次取节点会自增
- index 向所有 broker 个数取余
- 自带容错策略

其他实现：

- SelectMessageQueueByHash
- hash 的是传入的 args
- SelectMessageQueueByRandom
- SelectMessageQueueByMachineRoom 没有实现

也可以自定义实现 MessageQueueSelector 接口中的 select 方法

```
MESSAGEQUEUE SELECT(FINAL LIST<MESSAGEQUEUE> MQS, FINAL MESSAGE MSG, FINAL OBJECT ARG);
```

consumer 端 采用的是平均分配算法来进行负载均衡。

其他负载均衡算法

平均分配策略(默认)(AllocateMessageQueueAveragely) 环形分配策略 (AllocateMessageQueueAveragelyByCircle) 手动配置分配策略 (AllocateMessageQueueByConfig) 机房分配策略 (AllocateMessageQueueByMachineRoom) 一致性哈希分配策略 (AllocateMessageQueueConsistentHash) 靠近机房策略 (AllocateMachineRoomNearby)

追问：当消费负载均衡 consumer 和 queue 不对等的时候会发生什么?

Consumer 和 queue 会优先平均分配，如果 Consumer 少于 queue 的个数，则会存在部 分 Consumer 消费多个 queue 的情况，如果 Consumer 等于 queue 的个数，那就是一 个 Consumer 消费一个 queue，如果 Consumer 个数大于 queue 的个数，那么会有部分 Consumer 空余出来，白白的浪费了。

消息重复消费

影响消息正常发送和消费的重要原因是网络的不确定性。

引起重复消费的原因

- ACK

正常情况下在 consumer 真正消费完消息后应该发送 ack，通知 broker 该消息 已正常消费，从 queue 中剔除

当 ack 因为网络原因无法发送到 broker，broker 会认为词条消息没有被消费， 此后会开启消息重投机制把消息再次投递到 consumer

- 消费模式

在 CLUSTERING模式下，消息在 broker 中会保证相同 group 的 consumer 消 费一次，但是针对不同 group 的 consumer 会推送多次

解决方案

- 数据库表

处理消息前，使用消息主键在表中带有约束的字段中 insert

- Map

单机时可以使用 map ConcurrentHashMap -> putIfAbsent guava cache - Redis

分布式锁搞起来。

RocketMQ 如何保证消息不丢失

首先在如下三个部分都可能会出现丢失消息的情况：

- Producer 端
- Broker 端
- Consumer 端

Producer 端如何保证消息不丢失

- 采取 send()同步发消息，发送结果是同步感知的。
- 发送失败后可以重试，设置重试次数。默认 3 次。 PRODUCER.SETRETRYTIMESWHENSENDFAILED(10);

集群部署，比如发送失败了的原因可能是当前 Broker 宕机了，重试的时候会发

送到其他 Broker 上。

Broker 端如何保证消息不丢失

- 修改刷盘策略为同步刷盘。默认情况下是异步刷盘的。 FLUSHDISKTYPE = SYNC_FLUSH
- 集群部署，主从模式，高可用。

Consumer 端如何保证消息不丢失

- 完全消费正常后在进行手动 ack 确认。

高吞吐量下如何优化生产者和消费者的性能?

开发

- 同一 group 下，多机部署，并行消费 - 单个 Consumer 提高消费线程个数 - 批量消费
- 消息批量拉取
- 业务逻辑批量处理

运维

- 网卡调优
- jvm 调优
- 多线程与 cpu 调优 - Page Cache

Kafka

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：

- 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。
- 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。
- 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：

- 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。
- 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：

- 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。
- 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：

- Producer（生产者）: 产生消息的一方。
- Consumer（消费者）: 消费消息的一方。
- 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。
- 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。
- 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 。这正如我上面所画的图一样。

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?

- 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）。
- 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?

- 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
- 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
- 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：

- 更改用户会员等级。
- 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 个 Topic 只对应一个 Partition。
- 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?

- 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；
- 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?

- 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。
- 通过索引信息可以快速定位 message 和确定 response 的最大大小。
- 通过 index 元数据全部映射到 memory，可以避免 segment file 的 IO 磁盘操作。
- 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。

是什么确保了 Kafka 中服务器的负载平衡?

由于领导者的主要角色是执行分区的所有读写请求的任务，而追随者被动地复制领导者。因 此，在领导者失败时，其中一个追随者接管了领导者的角色。基本上，整个过程可确保服务 器的负载平衡。

消费者 API 的作用是什么?

允许应用程序订阅一个或多个主题并处理生成给它们的记录流的 API，我们称之为消费者 API。

解释流 API 的作用?

一种允许应用程序充当流处理器的 API，它还使用一个或多个主题的输入流，并生成一个输 出流到一个或多个输出主题，此外，有效地将输入流转换为输出流，我们称之为流 API。

Kafka 为什么那么快?

- Cache Filesystem Cache PageCache 缓存。
- 顺序写：由于现代的操作系统提供了预读和写技术，磁盘的顺序写大多数情况下比随机

写内存还要快。

- Zero-copy 零拷技术减少拷贝次数。
- Batching of Messages 批量处理。合并小的请求，然后以流的方式进行交互，直顶网

络上限。

- Pul- 拉模式 使用拉模式进行消息的获取消费，与消费端处理能力相符。

Kafka 系统工具有哪些类型?

- Kafka 迁移工具：它有助于将代理从一个版本迁移到另一个版本。
- Mirror Maker：Mirror Maker 工具有助于将一个 Kafka 集群的镜像提供给另一个。 - 消费者检查:对于指定的主题集和消费者组，它显示主题，分区，所有者。

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。

Zookeeper 对于 Kafka 的作用是什么?

- Zookeeper 是一个开放源码的、高性能的协调服务，它用于 Kafka 的分布式应用。
- Zookeeper 主要用于在集群中不同节点之间进行通信。
- 在 Kafka 中，它被用于提交偏移量，因此如果节点在任何情况下都失败了，它都可以从 之前提交的偏移量中获取。
- 除此之外，它还执行其他活动，如: leader 检测、分布式同步、配置管理、识别新节点 何时离开或连接、集群、节点实时状态等等。

流 API 的作用是什么?

一种允许应用程序充当流处理器的 API，它还使用一个或多个主题的输入流，并生成一个输 出流到一个或多个输出主题，此外，有效地将输入流转换为输出流，我们称之为流 API。

Kafka 的流处理是什么意思?

连续、实时、并发和以逐记录方式处理数据的类型，我们称之为 Kafka 流处理。

Kafka 集群中保留期的目的是什么?

保留期限保留了 Kafka 群集中的所有已记录。它不会检查它们是否已被消耗。此外，可以通 过使用保留期的配置设置来丢弃记录，而且，它可以释放一些空间。

Memcached 面试题

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?

- 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。

- 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）。

- 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。

- 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）。

- Memcached 的简单限制就是键（key）和 Value 的限制。最大键长为 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 长度支 持到 512K。

- 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。

- 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 。

- 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：不能变长，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。

- 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。

什么是二进制协议，我该关注吗?

关于二进制最好的信息当然是二进制协议规范： 二进制协议尝试为端提供一个更有效的、可靠的协议，减少客户端/服务器端因 处理协议而产生的 CPU 时间。

根据 Facebook 的测试，解析 ASCII协议是 Memcached 中消耗 CPU 时间最 多的环节。所以，我们为什么不改进 ASCII协议呢?

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，用户在浏览其他页面时，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：

- 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。

- 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。

- session 共享

缓存 session，使用 Redis， Memcached。

- 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。

- 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。

- 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。
- 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。
- 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。

提到如何检查函数的源代码?

要检查没有任何括号的函数源代码，必须调用该函数。

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。

Nginx 面试题

Nginx 是如何实现高并发的?

如果一个 server 采用一个进程(或者线程)负责一个 request 的方式，那么进 程数就是并发数。那么显而易见的，就是会有很多进程在等待中。等什么？最 多的应该是等待网络传输。其缺点胖友应该也感觉到了，此处不述。

而 Nginx 的异步非阻塞工作方式正是利用了这点等待的时间。在需要等待的 时候，这些进程就空闲出来待命了。因此表现为少数几个进程就解决了大量的 并发问题。

Nginx 是如何利用的呢，简单来说：同样的 4 个进程，如果采用一个进程负 责一个 request 的方式，那么，同时进来 4 个 request 之后，每个进程就 负责其中一个，直至会话关闭。期间，如果有第 5 个 request 进来了。就无 法及时反应了，因为 4 个进程都没干完活呢，因此，一般有个调度进程，每当 新进来了一个 request ，就新开个进程来处理。

Nginx 不这样，每进来一个 request ，会有一个 worker 进程去处理。但不 是全程的处理，处理到什么程度呢？处理到可能发生阻塞的地方，比如向上游 （后端）服务器转发 request ，并等待请求返回。那么，这个处理的 worker 不会这么傻等着，他会在发送完请求后，注册一个事件：“如果 upstream 返 回了，告诉我一声，我再接着干”。于是他就休息去了。此时，如果再有 request 进来，他就可以很快再按这种方式处理。而一旦上游服务器返回了， 就会触发这个事件，worker 才会来接手，这个 request 才会接着往下走。 **这就是为什么说，Nginx 基于事件模型。**

由于 web server 的工作性质决定了每个 request 的大部份生命都是在网络 传输中，实际上花费在 server 机器上的时间片不多。这是几个进程就解决高 并发的秘密所在。即：

webserver 刚好属于网络 IO 密集型应用，不算是计算密集型。异步，非阻 塞，使用 epol- ，和大量细节处的优化，也正是 Nginx 之所以然的技术基 石。

请解释 Nginx 如何处理 HTTP 请求。

Nginx 使用反应器模式。主事件循环等待操作系统发出准备事件的信号，这样 数据就可以从套接字读取，在该实例中读取到缓冲区并进行处理。单个线程可 以提供数万个并发连接。

为什么要做动、静分离?

在我们的软件开发中，有些请求是需要后台处理的（如：.jsp,.do 等等），有些 请求是不需要经过后台处理的（如：css、html、jpg、js 等等），这些不需要 经过后台处理的文件称为静态文件，否则动态文件。因此我们后台处理忽略静 态文件，但是如果直接忽略静态文件的话，后台的请求次数就明显增多了。在 我们对资源的响应速度有要求的时候，应该使用这种动静分离的策略去解决

动、静分离将网站静态资源（HTML，JavaScript，CSS 等）与后台应用分开 部署，提高用户访问静态代码的速度，降低对后台应用访问。这里将静态资源 放到 nginx 中，动态资源转发到 tomcat 服务器中,毕竟 Tomcat 的优势是处理 动态请求。

nginx 是如何实现高并发的?

一个主进程，多个工作进程，每个工作进程可以处理多个请求，每进来一个 request，会有一个 worker 进程去处理。但不是全程的处理，处理到可能发生 阻塞的地方，比如向上游（后端）服务器转发 request，并等待请求返回。那 么，这个处理的 worker 继续处理其他请求，而一旦上游服务器返回了，就会 触发这个事件，worker 才会来接手，这个 request 才会接着往下走。由于 web server 的工作性质决定了每个 request 的大部份生命都是在网络传输中， 实际上花费在 server 机器上的时间片不多。这是几个进程就解决高并发的秘密 所在。即 webserver 刚好属于网络 IO 密集型应用，不算是计算密集型。

Nginx 静态资源?

静态资源访问，就是存放在 nginx 的 html 页面，我们可以自己编写。

Nginx 配置高可用性怎么配置?

当上游服务器(真实访问服务器)，一旦出现故障或者是没有及时相应的话，应

该直接轮训到下一台服务器，保证服务器的高可用。

Nginx 配置代码：

```
server {

     listen 80;

     server\_name www.lijie.com;cc  nginx 发送给上游服务器(真实访问的服 务器)超时时间

         proxy\_send\_timeout 1s;###

         nginx 接受上游服务器(真实访问的服务器)超时时间

         proxy\_read\_timeout 1s;

         index index.html index.htm;
```

```
    }

}
```

502 错误可能原因

- FastCGI 进程是否已经启动
- FastCGI worker 进程数是否不够
- FastCGI 执行时间过长
- fastcgi_connect_timeout 300; - fastcgi_send_timeout 300;
- fastcgi_read_timeout 300;
- FastCGI Buffer 不够
- nginx 和 apache 一样，有前端缓冲限制，可以调整缓冲参数 - fastcgi_buffer_size 32k;
- fastcgi_buffers 8 32k;
- Proxy Buffer 不够
- 如果你用了 Proxying，调整
- proxy_buffer_size 16k;
- proxy_buffers 4 16k;
- php 脚本执行时间过长
- 将 php-fpm.conf 的 0s 的 0s 改成一个时间

在 Nginx 中，解释如何在 URL 中保留双斜线?

要在 URL 中保留双斜线，就必须使用 语法:merge_slashes [on/off] 默认值: merge_slashes on

环境: http，server

merge_slashes_off;

Nginx 服务器上的 Master 和 Worker 进程分别是什么?

Master 进程：读取及评估配置和维持 ；Worker 进程：处理请求。

Nginx 的优缺点?

优点：

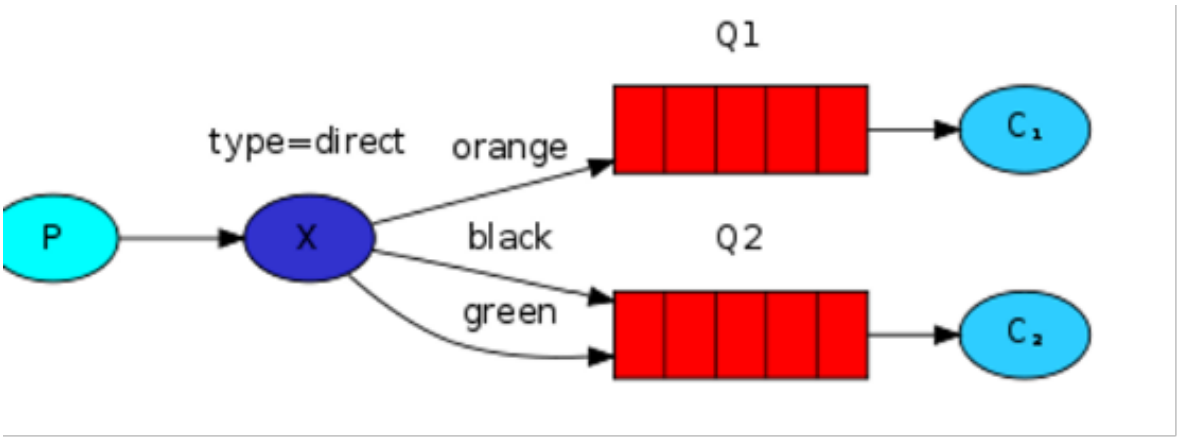
- 占内存小，可实现高并发连接，处理响应快。
- 可实现 HTTP 服务器、虚拟主机、方向代理、负载均衡。 - Nginx 配置简单。
- 可以不暴露正式的服务器 IP 地址。

缺点：

动态处理差，nginx 处理静态文件好,耗费内存少，但是处理动态页面则很鸡 肋，现在一般前端用 nginx 作为反向代理抗住压力。

RabbitMQ

RabbitMQ routing 路由模式



消息生产者将消息发送给交换机按照路由判断,路由是字符串(info) 当前产生的消息携 带路由字符(对象的方法)，交换机根据路由的 key，只能匹配上路由 key 对应的消息队列, 对应的消费者才能消费消息。

根据业务功能定义路由字符串。

从系统的代码逻辑中获取对应的功能字符串,将消息任务扔到对应的队列中。

业务场景：error 通知、EXCEPTION、错误通知的功能、传统意义的错误通知、客户 通知、利用 key 路由，可以将程序中的错误封装成消息传入到消息队列中，开发者可以自 定义消费者，实时接收错误。

消息怎么路由?

消息提供方->路由->一至多个队列消息发布到交换器时，消息将拥有一个路由键 （routing key），在消息创建时设定。通过队列路由键，可以把队列绑定到交换器上。消 息到达交换器后，RabbitMQ 会将消息的路由键与队列的路由键进行匹配（针对不同的交

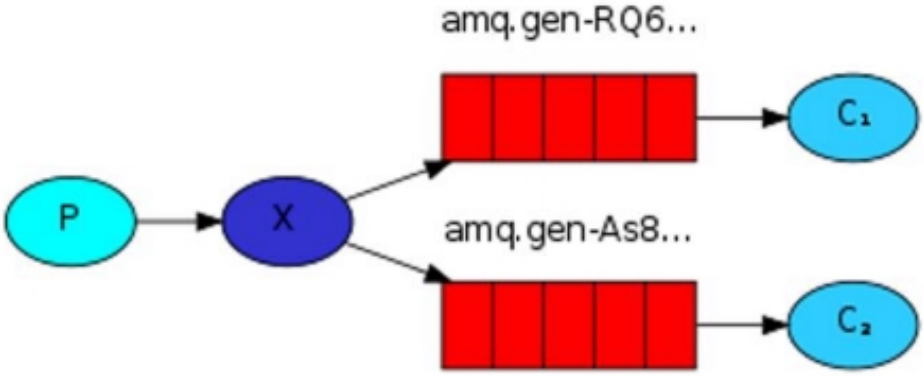
换器有不同的路由规则）。

常用的交换器主要分为一下三种：

- fanout：如果交换器收到消息，将会广播到所有绑定的队列上。
- direct：如果路由键完全匹配，消息就被投递到相应的队列。
- topic：可以使来自不同源头的消息能够到达同一个队列。 使用 topic 交换器时，可

以使用通配符。

RabbitMQ publish/subscribe 发布订阅(共享资源)



- 每个消费者监听自己的队列。
- 生产者将消息发给 broker，由交换机将消息转发到绑定此交换机的每个队列，每个绑

定交换机的队列都将接收到消息。

能够在地理上分开的不同数据中心使用 RabbitMQ cluster 么?

不能。

第一，你无法控制所创建的 queue 实际分布在 cluster 里的哪个 node 上（一般使用

HAProxy + cluster 模型时都是这样），这可能会导致各种跨地域访问时的常见问题。 第二，Erlang 的 OTP 通信框架对延迟的容忍度有限，这可能会触发各种超时，导致业务 疲于处理。

第三，在广域网上的连接失效问题将导致经典的"脑裂"问题，而 RabbitMQ 目前无法 处理（该问题主要是说 Mnesia）。

RabbitMQ 有那些基本概念?

- Broker：简单来说就是消息队列服务器实体。
- Exchange：消息交换机，它指定消息按什么规则，路由到哪个队列。
- Queue：消息队列载体，每个消息都会被投入到一个或多个队列。
- Binding：绑定，它的作用就是把 exchange 和 queue 按照路由规则绑定起来。
- Routing Key：路由关键字，exchange 根据这个关键字进行消息投递。
- VHost：vhost 可以理解为虚拟 broker ，即 mini-RabbitMQ server。其内部均含

有独立的 queue、exchange 和 binding 等，但最最重要的是，其拥有独立的权限 系统，可以做到 vhost 范围的用户控制。当然，从 RabbitMQ 的全局角度，vhost 可以作为不同权限隔离的手段（一个典型的例子就是不同的应用可以跑在不同的 vhost 中）。

- Producer：消息生产者，就是投递消息的程序。
- Consumer：消息消费者，就是接受消息的程序。
- Channel：消息通道，在客户端的每个连接里，可建立多个 channel，每个 channel

代表一个会话任务。

- 由 Exchange、Queue、RoutingKey 三个才能决定一个从 Exchange 到 Queue 的唯

一的线路。

什么情况下会出现 blackholed 问题?

blackholed 问题是指，向 exchange 投递了 message ，而由于各种原因导致该 message 丢失，但发送者却不知道。可导致 blackholed 的情况：1.向未绑定 queue 的 exchange 发送 message；2.exchange 以 binding_key key_A 绑定了 queue queue_A，但向该 exchange 发送 message 使用的 routing_key 却是 key_B。

什么是消费者 Consumer?

消费消息，也就是接收消息的一方。

消费者连接到 RabbitMQ 服务器，并订阅到队列上。消费消息时只消费消息体，丢弃标 签。

消息如何分发?

- 若该队列至少有一个消费者订阅，消息将以循环（round-robin）的方式发送给消费

者。每条消息只会分发给一个订阅的消费者（前提是消费者能够正常处理消息并进行 确认）。

- 通过路由可实现多消费的功能

Basic.Reject 的用法是什么?

该信令可用于 consumer 对收到的 message 进行 reject 。若在该信令中设置 requeue=true，则当 RabbitMQ server 收到该拒绝信令后，会将该 message 重新发 送到下一个处于 consume 状态的 consumer 处（理论上仍可能将该消息发送给当前 consumer）。若设置 requeue=false ，则 RabbitMQ server 在收到拒绝信令后，将直 接将该 message 从 queue 中移除。

另外一种移除 queue 中 message 的小技巧是，consumer 回复 Basic.Ack 但不对获 取到的 message 做任何处理。而 Basic.Nack 是对 Basic.Reject 的扩展，以支持一次 拒绝多条 message 的能力。

什么是 Binding 绑定?

通过绑定将交换器和队列关联起来，一般会指定一个 BindingKey,这样 RabbitMq 就知道 如何正确路由消息到队列了。

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：

- 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。
- 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

- 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。

分布式锁实现原理，用过吗?

在分析分布式锁的三种实现方式之前，先了解一下分布式锁应该具备哪些条件：

1. 在分布式系统环境下，一个方法在同一时间只能被一个机器的一个线程执行；
2. 高可用的获取锁与释放锁；
3. 高性能的获取锁与释放锁；
4. 具备可重入特性；
5. 具备锁失效机制，防止死锁；
6. 具备非阻塞锁特性，即没有获取到锁将直接返回获取锁失败。

分布式的CAP理论告诉我们"任何一个分布式系统都无法同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance），最多只能同时满足两项。"所以，很多系统在设计之初就要对这三者做出取舍。在互联网领域的绝大多数的场景中，都需要牺牲强一致性来换取系统的高可用性，系统往往只需要保证"最终一致性"，只要这个最终时间是在用户可以接受的范围内即可。

通常分布式锁以单独的服务方式实现，目前比较常用的分布式锁实现有三种：

- 基于数据库实现分布式锁。
- 基于缓存（redis，memcached，tair）实现分布式锁。
- 基于Zookeeper实现分布式锁。

尽管有这三种方案，但是不同的业务也要根据自己的情况进行选型，他们之间没有最好只有更适合！

- 基于数据库的实现方式

基于数据库的实现方式的核心思想是：在数据库中创建一个表，表中包含方法名等字段，并在方法名字段上创建唯一索引，想要执行某个方法，就使用这个方法名向表中插入数据，成功插入则获取锁，执行完成后删除对应的行数据释放锁。

创建一个表：

```
DROP TABLE IF EXISTS `method_lock`;
CREATE TABLE `method_lock` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT COMMENT '主键',
  `method_name` varchar(64) NOT NULL COMMENT '锁定的方法名',
  `desc` varchar(255) NOT NULL COMMENT '备注信息',
  `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`),
  UNIQUE KEY `uidx_method_name` (`method_name`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8 COMMENT='锁定中的方法';
```

想要执行某个方法，就使用这个方法名向表中插入数据：

```
INSERT INTO method_lock (method_name, desc) VALUES ('methodName', '测试的methodName');
```

因为我们对method_name做了唯一性约束，这里如果有多个请求同时提交到数据库的话，数据库会保证只有一个操作可以成功，那么我们就可以认为操作成功的那个线程获得了该方法的锁，可以执行方法体内容。

成功插入则获取锁，执行完成后删除对应的行数据释放锁：

```
delete from method_lock where method_name ='methodName';
```

注意：这里只是使用基于数据库的一种方法，使用数据库实现分布式锁还有很多其他的用法可以实现！

使用基于数据库的这种实现方式很简单，但是对于分布式锁应该具备的条件来说，它有一些问题需要解决及优化：

1、因为是基于数据库实现的，数据库的可用性和性能将直接影响分布式锁的可用性及性能，所以，数据库需要双机部署、数据同步、主备切换；

2、不具备可重入的特性，因为同一个线程在释放锁之前，行数据一直存在，无法再次成功插入数据，所以，需要在表中新增一列，用于记录当前获取到锁的机器和线程信息，在再次获取锁的时候，先查询表中机器和线程信息是否和当前机器和线程相同，若相同则直接获取锁；

3、没有锁失效机制，因为有可能出现成功插入数据后，服务器宕机了，对应的数据没有被删除，当服务恢复后一直获取不到锁，所以，需要在表中新增一列，用于记录失效时间，并且需要有定时任务清除这些失效的数据；

4、不具备阻塞锁特性，获取不到锁直接返回失败，所以需要优化获取逻辑，循环多次去获取。

5、在实施的过程中会遇到各种不同的问题，为了解决这些问题，实现方式将会越来越复杂；依赖数据库需要一定的资源开销，性能问题需要考虑。

- 基于Redis的实现方式

选用Redis实现分布式锁原因：

1. Redis有很高的性能；
2. Redis命令对此支持较好，实现起来比较方便

主要实现方式:

1. SET lock currentTime+expireTime EX 600 NX，使用set设置lock值，并设置过期时间为600秒，如果成功，则获取锁；

2. 获取锁后，如果该节点掉线，则到过期时间ock值自动失效；
3. 释放锁时，使用del删除lock键值；

使用redis单机来做分布式锁服务，可能会出现单点问题，导致服务可用性差，因此在服务稳定性要求高的场合，官方建议使用redis集群（例如5台，成功请求锁超过3台就认为获取锁），来实现redis分布式锁。详见RedLock。

优点:性能高，redis可持久化，也能保证数据不易丢失,redis集群方式提高稳定性。

缺点:使用redis主从切换时可能丢失部分数据。

- 基于ZooKeeper的实现方式

ZooKeeper是一个为分布式应用提供一致性服务的开源组件，它内部是一个分层的文件系统目录树结构，规定同一个目录下只能有一个唯一文件名。基于ZooKeeper实现分布式锁的步骤如下：

1. 创建一个目录mylock；
2. 线程A想获取锁就在mylock目录下创建临时顺序节点；
3. 获取mylock目录下所有的子节点，然后获取比自己小的兄弟节点，如果不存在，则说明当前线程顺序号最小，获得锁；
4. 线程B获取所有节点，判断自己不是最小节点，设置监听比自己次小的节点；
5. 线程A处理完，删除自己的节点，线程B监听到变更事件，判断自己是不是最小的节点，如果是则获得锁。

这里推荐一个Apache的开源库Curator，它是一个ZooKeeper客户端，Curator提供的InterProcessMutex是分布式锁的实现，acquire方法用于获取锁，release方法用于释放锁。

优点：具备高可用、可重入、阻塞锁特性，可解决失效死锁问题。

缺点：因为需要频繁的创建和删除节点，性能上不如Redis方式。

上面的三种实现方式，没有在所有场合都是完美的，所以，应根据不同的应用场景选择最适合的实现方式。

在分布式环境中，对资源进行上锁有时候是很重要的，比如抢购某一资源，这时候使用分布式锁就可以很好地控制资源。

Etcd怎么实现分布式锁?

首先思考下Etcd是什么？可能很多人第一反应可能是一个键值存储仓库，却没有重视官方定义的后半句，用于配置共享和服务发现。

```
A highly-available key value store for shared configuration and service discovery.
```

实际上，etcd 作为一个受到 ZooKeeper 与 doozer 启发而催生的项目，除了拥有与之类似的功能外，更专注于以下四点。

- 简单：基于 HTTP+JSON 的 API 让你用 curl 就可以轻松使用。
- 安全：可选 SSL 客户认证机制。
- 快速：每个实例每秒支持一千次写操作。
- 可信：使用 Raft 算法充分实现了分布式。

但是这里我们主要讲述Etcd如何实现分布式锁?

因为 Etcd 使用 Raft 算法保持了数据的强一致性，某次操作存储到集群中的值必然是全局一致的，所以很容易实现分布式锁。锁服务有两种使用方式，一是保持独占，二是控制时序。

- 保持独占即所有获取锁的用户最终只有一个可以得到。etcd 为此提供了一套实现分布式锁原子操作 CAS（CompareAndSwap）的 API。通过设置prevExist值，可以保证在多个节点同时去创建某个目录时，只有一个成功。而创建成功的用户就可以认为是获得了锁。
- 控制时序，即所有想要获得锁的用户都会被安排执行，但是获得锁的顺序也是全局唯一的，同时决定了执行顺序。etcd 为此也提供了一套 API（自动创建有序键），对一个目录建值时指定为POST动作，这样 etcd 会自动在目录下生成一个当前最大的值为键，存储这个新的值（客户端编号）。同时还可以使用 API 按顺序列出所有当前目录下的键值。此时这些键的值就是客户端的时序，而这些键中存储的值可以是代表客户端的编号。

在这里Ectd实现分布式锁基本实现原理为：

1. 在ectd系统里创建一个key
2. 如果创建失败，key存在，则监听该key的变化事件，直到该key被删除，回到1
3. 如果创建成功，则认为我获得了锁

应用示例:

```
package etcdsync

import (
    "fmt"
    "io"
    "os"
    "sync"
    "time"

    "github.com/coreos/etcd/client"
    "github.com/coreos/etcd/Godeps/_workspace/src/golang.org/x/net/context"
)

const (
    defaultTTL = 60
    defaultTry = 3
    deleteAction = "delete"
    expireAction = "expire"
)

// A Mutex is a mutual exclusion lock which is distributed across a cluster.
type Mutex struct {
    key    string
    id     string // The identity of the caller
    client client.Client
    kapi   client.KeysAPI
    ctx    context.Context
    ttl    time.Duration
    mutex  *sync.Mutex
    logger io.Writer
}

// New creates a Mutex with the given key which must be the same
// across the cluster nodes.
```

```
// machines are the ectd cluster addresses
func New(key string, ttl int, machines []string) *Mutex {
    cfg := client.Config{
        Endpoints:               machines,
        Transport:               client.DefaultTransport,
        HeaderTimeoutPerRequest: time.Second,
    }

    c, err := client.New(cfg)
    if err != nil {
        return nil
    }

    hostname, err := os.Hostname()
    if err != nil {
        return nil
    }

    if len(key) == 0 || len(machines) == 0 {
        return nil
    }

    if key[0] != '/' {
        key = "/" + key
    }

    if ttl < 1 {
        ttl = defaultTTL
    }

    return &Mutex{
        key:    key,
        id:     fmt.Sprintf("%v-%v-%v", hostname, os.Getpid(),
time.Now().Format("20060102-15:04:05.999999999")),
        client: c,
        kapi:   client.NewKeysAPI(c),
        ctx: context.TODO(),
        ttl: time.Second * time.Duration(ttl),
        mutex:  new(sync.Mutex),
    }
}

// Lock locks m.
// If the lock is already in use, the calling goroutine
// blocks until the mutex is available.
func (m *Mutex) Lock() (err error) {
    m.mutex.Lock()
    for try := 1; try <= defaultTry; try++ {
        if m.lock() == nil {
```

```
            return nil
        }

        m.debug("Lock node %v ERROR %v", m.key, err)
        if try < defaultTry {
            m.debug("Try to lock node %v again", m.key, err)
        }
    }
    return err
}

func (m *Mutex) lock() (err error) {
    m.debug("Trying to create a node : key=%v", m.key)
    setOptions := &client.SetOptions{
        PrevExist:client.PrevNoExist,
        TTL:      m.ttl,
    }
    resp, err := m.kapi.Set(m.ctx, m.key, m.id, setOptions)
    if err == nil {
        m.debug("Create node %v OK [%q]", m.key, resp)
        return nil
    }
    m.debug("Create node %v failed [%v]", m.key, err)
    e, ok := err.(client.Error)
    if !ok {
        return err
    }

    if e.Code != client.ErrorCodeNodeExist {
        return err
    }

    // Get the already node's value.
    resp, err = m.kapi.Get(m.ctx, m.key, nil)
    if err != nil {
        return err
    }
    m.debug("Get node %v OK", m.key)
    watcherOptions := &client.WatcherOptions{
        AfterIndex : resp.Index,
        Recursive:false,
    }
    watcher := m.kapi.Watcher(m.key, watcherOptions)
    for {
        m.debug("Watching %v ...", m.key)
        resp, err = watcher.Next(m.ctx)
        if err != nil {
            return err
        }
```

```
        m.debug("Received an event : %q", resp)
        if resp.Action == deleteAction || resp.Action == expireAction {
            return nil
        }
    }

}

// Unlock unlocks m.
// It is a run-time error if m is not locked on entry to Unlock.
//
// A locked Mutex is not associated with a particular goroutine.
// It is allowed for one goroutine to lock a Mutex and then
// arrange for another goroutine to unlock it.
func (m *Mutex) Unlock() (err error) {
    defer m.mutex.Unlock()
    for i := 1; i <= defaultTry; i++ {
        var resp *client.Response
        resp, err = m.kapi.Delete(m.ctx, m.key, nil)
        if err == nil {
            m.debug("Delete %v OK", m.key)
            return nil
        }
        m.debug("Delete %v falied: %q", m.key, resp)
        e, ok := err.(client.Error)
        if ok && e.Code == client.ErrorCodeKeyNotFound {
            return nil
        }
    }
    return err
}

func (m *Mutex) debug(format string, v ...interface{}) {
    if m.logger != nil {
        m.logger.Write([]byte(m.id))
        m.logger.Write([]byte(" "))
        m.logger.Write([]byte(fmt.Sprintf(format, v...)))
        m.logger.Write([]byte("\n"))
    }
}

func (m *Mutex) SetDebugLogger(w io.Writer) {
    m.logger = w
}
```

其实类似的实现有很多，但目前都已经过时，使用的都是被官方标记为deprecated的项目。且大部分接口都不如上述代码简单。 使用上，跟Golang官方sync包的Mutex接口非常类似，先New()，然后调用Lock()，使用完后调用Unlock()，就三个接口，就是这么简单。示例代码如下：

```
package main

import (
    "github.com/zieckey/etcdsync"
    "log"
)

func main() {
    //etcdsync.SetDebug(true)
    log.SetFlags(log.Ldate|log.Ltime|log.Lshortfile)
    m := etcdsync.New("/etcdsync", "123", []string{"http://127.0.0.1:2379"})
    if m == nil {
        log.Printf("etcdsync.NewMutex failed")
    }
    err := m.Lock()
    if err != nil {
        log.Printf("etcdsync.Lock failed")
    } else {
        log.Printf("etcdsync.Lock OK")
    }

    log.Printf("Get the lock. Do something here.")

    err = m.Unlock()
    if err != nil {
        log.Printf("etcdsync.Unlock failed")
    } else {
        log.Printf("etcdsync.Unlock OK")
    }
}
```

说说 ZooKeeper 一般都有哪些使用场景?

- 分布式协调：这个其实就是 zk 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 zk 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 zk 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 zk 那个节点的值，A立马就可以收到通知，完美解决。
- 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 zk 分布式锁，一个机器接收到了请求之后先获取 zk 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。
- 配置信息管理：zk 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 zk 么。

- 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 zk 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 zk 感知到切换到备份进程。

说说你们的分布式 session 方案是啥？怎么做的?

- Tomcat + redis

其实还挺方便的，就是使用 session 的代码跟以前一样，还是基于 tomcat 原生的 session 支持即可，然后就是用一个叫做 tomcat RedisSessionManager 的东西，让我们部署的 tomcat 都将 session 数据存储到 redis 即可.

- Spring Session + redis

分布式会话的这个东西重耦合在 tomcat，如果我要将 web容器迁移成 jetty，不能重新把 jetty 都配置一遍.

所以现在比较好用的还是基于 java 的一站式解决方案，使用 spring session 是一个很好的选择，给 spring session 配置基于 redis 来存储 session 数据，然后配置一个 spring session 的过滤器，这样的话，session 相关操作都会交

给 spring session 来管了。接着在代码中，就是用原生的 session 操作，就是直接基于 spring session 从 redis 中获取数据了。

分布式事务了解吗?

- 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）

- 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

- 本地消息表
- 可靠消息最终一致性方案
- 最大努力通知方案

那常见的分布式锁有哪些解决方案?

- Reids 的分布式锁，很多大公司会基于 Reidis 做扩展开发
- 基于 Zookeeper
- 基于数据库，比如 Mysql

ZK 和 Redis 的区别，各自有什么优缺点?

先说 Redis：

- Redis 只保证最终一致性，副本间的数据复制是异步进行（Set 是写，Get 是读，Reids 集群一般是读写分离架构，存在主从同步延迟情况），主从切换之后可能有部分数据没有复制过去可能会丢失锁情况，故强一致性要求的业务不推荐使用 Reids，推荐使用 zk。
- Redis 集群各方法的响应时间均为最低。随着并发量和业务数量的提升其响应时间会有明显上升（公有集群影响因素偏大），但是极限 qps 可以达到最大且基本无异常。

再说 ZK：

- 使用 ZooKeeper 集群，锁原理是使用 ZooKeeper 的临时节点，临时节点的生命周期在 Client 与集群的 Session 结束时结束。因此如果某个 Client 节点存在网络问题，与 ZooKeeper 集群断开连接，Session 超时同样会导致锁被错误的释放（导致被其他线程错误地持有），因此 ZooKeeper 也无法保证完全一致。
- ZK具有较好的稳定性；响应时间抖动很小，没有出现异常。但是随着并发量和业务数量的提升其响应时间和 qps 会明显下降。

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

- ## 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。

- ## 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。

- ## 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。

- ## 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：
- 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）
- 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）
- 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：

- 按功能划分数据库
- 分片（如开源的 Mycat、Amoeba等）。

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：

- 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
- 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。

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：

- 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。
- 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。
- 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）**
- 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。
- 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）
- 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”：

- 协调者节点向所有参与者节点发出”正式提交(commit)”的请求。
- 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。
- 参与者节点向协调者节点发送”完成”消息。
- 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”：

- 协调者节点向所有参与者节点发出”回滚操作(rollback)”的请求。
- 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。
- 参与者节点向协调者节点发送”回滚完成”消息。
- 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。
- 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。
- 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。

ClickHouse 面试题

什么是 ClickHouse?

ClickHouse 是近年来备受关注的开源列式数据库管理系统，主要用于数据分析 （OLAP）领域。通过向量化执行以及对 cpu 底层指令集（SIMD）的使用，它 可以对海量数据进行并行处理，从而加快数据的处理速度。ClickHouse 从 OLAP 场景需求出发，定制开发了一套全新的高效列式存储引擎，并且实现了数据有序 存储、主键索引、稀疏索引、数据 Sharding、数据 Partitioning、TTL、主备复 制等丰富功能。

ClickHouse 有哪些应用场景?

1. 绝大多数请求都是用于读访问的；
2. 数据需要以大批次（大于 1000 行）进行更新，而不是单行更新；
3. 数据只是添加到数据库，没有必要修改；
4. 读取数据时，会从数据库中提取出大量的行，但只用到一小部分列；
5. 表很“宽”，即表中包含大量的列；
6. 查询频率相对较低（通常每台服务器每秒查询数百次或更少）；
7. 对于简单查询，允许大约 50 毫秒的延迟；
8. 列的值是比较小的数值和短字符串（例如，每个 URL只有 60 个字节）；
9. 在处理单个查询时需要高吞吐量（每台服务器每秒高达数十亿行）；
10. 不需要事务；
11. 数据一致性要求较低；
12. 每次查询中只会查询一个大表。除了一个大表，其余都是小表；
13. 查询结果显著小于数据源。即数据有过滤或聚合。返回结果不超过单个服务 器内存。

ClickHouse 列式存储的优点有哪些?

- 当分析场景中往往需要读大量行但是少数几个列时，在行存模式下，数据按

行连续存储，所有列的数据都存储在一个 block 中，不参与计算的列在 IO 时也要全部读出，读取操作被严重放大。而列存模式下，只需要读取参与计 算的列即可，极大的减低了 IO cost，加速了查询。

- 同一列中的数据属于同一类型，压缩效果显著。列存往往有着高达十倍甚至

更高的压缩比，节省了大量的存储空间，降低了存储成本。

- 更高的压缩比意味着更小的 data size，从磁盘中读取相应数据耗时更短。 - 自由的压缩算法选择。不同列的数据具有不同的数据类型，适用的压缩算法

也就不尽相同。可以针对不同列类型，选择最合适的压缩算法。

- 高压缩比，意味着同等大小的内存能够存放更多数据，系统 cache 效果更好。

ClickHouse 的缺点是是什么?

- 不支持事务，不支持真正的删除/更新； - 不支持二级索引；
- join 实现与众不同；
- 不支持窗口功能；
- 元数据管理需要人为干预。

ClickHouse 的架构是怎样的?

ClickHouse 采用典型的分组式的分布式架构，其中：

- Shard。集群内划分为多个分片或分组（Shard 0 ... Shard N），通过 Shard

的线性扩展能力，支持海量数据的分布式存储计算。

- Node。每个 Shard 内包含一定数量的节点（Node，即进程），同一 Shard 内的节点互为副本，保障数据可靠。ClickHouse 中副本数可按需建设，且逻

辑上不同 Shard 内的副本数可不同。

- ZooKeeper Service。集群所有节点对等，节点间通过 ZooKeeper 服务进

行分布式协调。

ClickHouse 的逻辑数据模型?

从用户使用角度看，ClickHouse 的逻辑数据模型与关系型数据库有一定的相似： 一个集群包含多个数据库，一个数据库包含多张表，表用于实际存储数据。

ClickHouse 的核心特性?

- 列存储：列存储是指仅从存储系统中读取必要的列数据，无用列不读取，速

度非常快。ClickHouse 采用列存储，这对于分析型请求非常高效。一个典型 且真实的情况是，如果我们需要分析的数据有 50 列，而每次分析仅读取其 中的 5 列，那么通过列存储，我们仅需读取必要的列数据，相比于普通行

存，可减少 10 倍左右的读取、解压、处理等开销，对性能会有质的影响。

- 向量化执行：在支持列存的基础上，ClickHouse 实现了一套面向 向量化处

理 的计算引擎，大量的处理操作都是向量化执行的。相比于传统火山模型中 的逐行处理模式，向量化执行引擎采用批量处理模式，可以大幅减少函数调 用开销，降低指令、数据的 Cache Miss，提升 CPU 利用效率。并且 ClickHouse 可利用 SIMD 指令进一步加速执行效率。这部分是 ClickHouse 优于大量同类 OLAP 产品的重要因素。

- 编码压缩：由于 ClickHouse 采用列存储，相同列的数据连续存储，且底层

数据在存储时是经过排序的，这样数据的局部规律性非常强，有利于获得更 高的数据压缩比。此外，ClickHouse 除了支持 LZ### ZSTD 等通用压缩算 法外，还支持 Delta、DoubleDelta、Gorilla 等专用编码算法，用于进一步 提高数据压缩比。

- 多索引：列存用于裁剪不必要的字段读取，而索引则用于裁剪不必要的记录

读取。ClickHouse 支持丰富的索引，从而在查询时尽可能的裁剪不必要的 记录读取，提高查询性能。

使用 ClickHouse 时有哪些注意点?

分区和索引 分区粒度根据业务特点决定，不宜过粗或过细。一般选择按天分区，也可指定为

tuple()；以单表 1 亿数据为例，分区大小控制在 10-30 个为最佳。

必须指定索引列，clickhouse 中的索引列即排序列，通过 order by 指定，一般 在查询条件中经常被用来充当筛选条件的属性被纳入进来；可以是单一维度，也 可以是组合维度的索引；通常需要满足高级列在前、查询频率大的在前原则；还 有基数特别大的不适合做索引列，如用户表的 userid 字段；通常筛选后的数据 满足在百万以内为最佳。

数据采样策略 通过采用运算可极大提升数据分析的性能。

数据量太大时应避免使用 select * 操作，查询的性能会与查询的字段大小和数 量成线性变换；字段越少，消耗的 IO 资源就越少，性能就会越高。 千万以上数据集用 order by 查询时需要搭配 where 条件和 limit 语句一起使用。

如非必须不要在结果集上构建虚拟列，虚拟列非常消耗资源浪费性能，可以考虑 在前端进行处理，或者在表中构造实际字段进行额外存储。 不建议在高基列上执行 distinct 去重查询，改为近似去重 uniqCombined。 多表 Join 时要满足小表在右的原则，右表关联时被加载到内存中与左表进行比 较。

存储

ClickHouse 不支持设置多数据目录，为了提升数据 一个券组绑定多块物理磁盘提升读写性能；多数查询场景 硬盘快 2-3 倍。

io 性能，可以挂载虚拟券组，

SSD 盘会比普通机械

ClickHouse 的引擎有哪些?

ClickHouse 提供了大量的数据引擎，分为数据库引擎、表引擎，根据数据特点 及使用场景选择合适的引擎至关重要。

ClickHouse 引擎分类

在以下几种情况下，ClickHouse 使用自己的数据库引擎：

- 决定表存储在哪里以及以何种方式存储； - 支持哪些查询以及如何支持；
- 并发数据访问；
- 索引的使用；
- 是否可以执行多线程请求；
- 数据复制参数。

在所有的表引擎中，最为核心的当属 MergeTree 系列表引擎，这些表引擎拥有 最为强大的性能和最广泛的使用场合。对于非 MergeTree 系列的其他引擎而言， 主要用于特殊用途，场景相对有限。而 MergeTree 系列表引擎是官方主推的存 储引擎，支持几乎所有 ClickHouse 核心功能。

MergeTree 作为家族系列最基础的表引擎，主要有以下特点：

- 存储的数据按照主键排序：允许创建稀疏索引，从而加快数据查询速度； - 支持分区，可以通过 PRIMARY KEY语句指定分区字段；
- 支持数据副本；
- 支持数据采样。

建表引擎参数有哪些?

ENGINE：ENGINE = MergeTree()，MergeTree 引擎没有参数。

ORDER BY：order by 设定了分区内的数据按照哪些字段顺序进行有序保 存。

order by 是 MergeTree 中唯一一个必填项，甚至比 primary key 还重要，因 为当用户不设置主键的情况，很多处理会依照 order by 的字段进行处理。 要求：主键必须是 order by 字段的前缀字段。

如果 ORDER BY与 PRIMARY KEY不同，PRIMARY KEY必须是 ORDER BY的 前缀(为了保证分区内数据和主键的有序性)。

ORDER BY 决定了每个分区中数据的排序规则;

PRIMARY KEY 决定了一级索引(primary.idx);

ORDER BY 可以指代 PRIMARY KEY, 通常只用声明 ORDER BY 即可。

PARTITION BY：分区字段，可选。如果不填：只会使用一个分区。 分区目录：MergeTree 是以列文件+索引文件+表定义文件组成的，但是如果设 定了分区那么这些文件就会保存到不同的分区目录中。

PRIMARY KEY：指定主键，如果排序字段与主键不一致，可以单独指定主 键字段。否则默认主键是排序字段。可选。

SAMPLE BY：采样字段，如果指定了该字段，那么主键中也必须包含该字 段。比如 SAMPLE BY intHash32(UserID) ORDER BY (CounterID, EventDate, intHash32(UserID))。可选。

TTL：数据的存活时间。在 MergeTree 中，可以为某个列字段或整张表设置 TTL。当时间到达时，如果是列字段级别的 TTL，则会删除这一列的数据；如果 是表级别的 TTL，则会删除整张表的数据。可选。

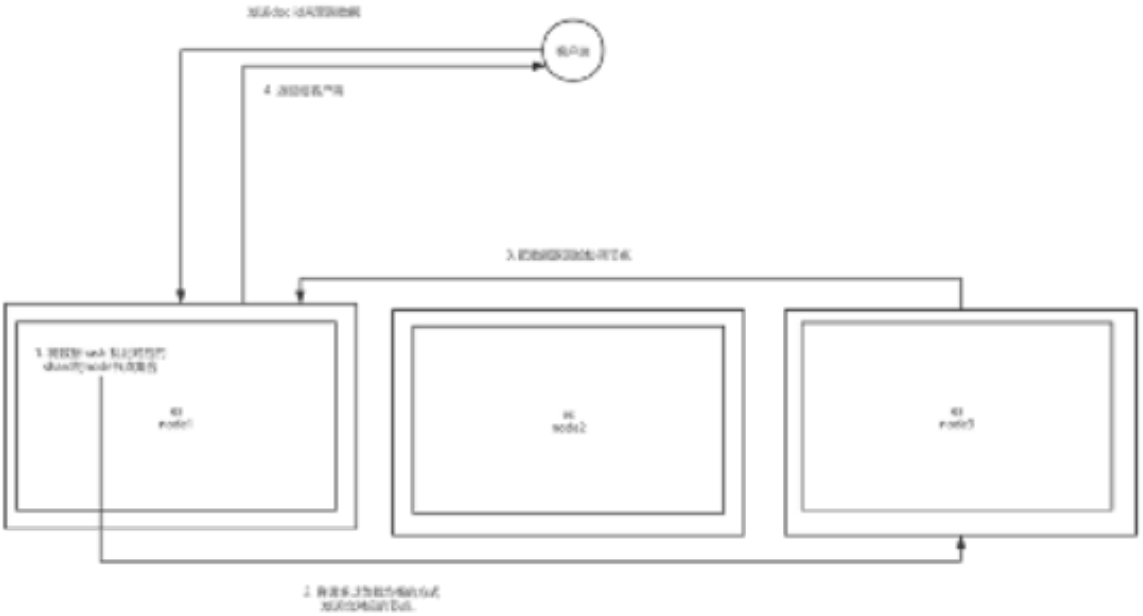
SETTINGS：额外的参数配置。可选。

Elasticsearch 面试题

Elasticsearch 读取数据

使用 RestFul API 向对应的 node 发送查询请求，根据 did 来判断在哪个 shard 上，返回的是 primary 和 replica 的 node 节点集合。 这样会负载均衡地把查询发送到对应节点，之后对应节点接收到请求，将

document 数据返回协调节点，协调节点把 document 返回给客户端。



您能解释一下 X-Pack for Elasticsearch 的功能和重要性吗?

X-Pack 是与 Elasticsearch 一起安装的扩展程序。

X-Pack 的各种功能包括安全性（基于角色的访问，特权/权限，角色和用户安 全性），监视，报告，警报等。

Elasticsearch 中的节点（比如共 20 个），其中的 10 个选了 一个 master，另外 10 个选了另一个 master，怎么办?

- 当集群 master 候选数量不小于 3 个时，可以通过设置最少投票通过数量

（discovery.zen.minimum_master_nodes）超过所有候选节点一半以上 来解决脑裂问题；

- 当候选数量为两个时，只能修改为唯一的一个 master 候选，其他作为data 节点，避免脑裂问题。

解释一下 Elasticsearch 集群中的 索引的概念？

Elasticsearch 集群可以包含多个索引，与关系数据库相比，它们相当于数据库 表。

你可以列出 Elasticsearch 各种类型的分析器吗?

Elasticsearch Analyzer 的类型为内置分析器和自定义分析器。

Standard Analyzer 标准分析器是默认分词器，如果未指定，则使用该分词器。 它基于 Unicode 文本分割算法，适用于大多数语言。

Whitespace Analyzer

基于空格字符切词。

Stop Analyzer

在 simple Analyzer 的基础上，移除停用词。

Keyword Analyzer 不切词，将输入的整个串一起返回。

自定义分词器的模板

自定义分词器的在 Mapping 的 Setting 部分设置：

```
PUT my\_custom\_index
{
    "settings":{
        "analysis":{
            "char\_filter":{

            },
            "tokenizer":{

            },
            "filter":{

            },
            "analyzer":{

            }
        }
    }
}
```

其中： “char_filter”:{},——对应字符过滤部分； “tokenizer”:{},——对应文本切分为分词部分；

“filter”:{},——对应分词后再过滤部分； “analyzer”:{}——对应分词器组成部分，其中会包含：1. 2. 3。

解释一下 Elasticsearch Node?

节点是 Elasticsearch 的实例。实际业务中，我们会说：ES 集群包含 3 个节 点、7 个节点。

这里节点实际就是：一个独立的 Elasticsearch 进程，一般将一个节点部署到 一台独立的服务器或者虚拟机、容器中。 不同节点根据角色不同，可以划分为：

主节点

帮助配置和管理在整个集群中添加和删除节点。

数据节点

存储数据并执行诸如 CRUD（创建/读取/更新/删除）操作，对数据进行搜索和 聚合的操作。

客户端节点（或者说：协调节点）

将集群请求转发到主节点，将与数据相 关的请求转发到数据节点。

摄取节点

用于在索引之前对文档进行预处理。

在安装 Elasticsearch 时，请说明不同的软件包及其重要性?

这个貌似没什么好说的，去官方文档下载对应操作系统安装包即可。 部分功能是收费的，如机器学习、高级别 kerberos 认证安全等选型要知悉。

Elasticsearch 在部署时，对 Linux 的设置有哪些优化方法?

关闭缓存 swap;

堆内存设置为：Min（节点内存/2, 32GB）;

设置最大文件句柄数;

线程池+队列大小根据业务需要做调整;

磁盘存储 raid 方式——存储有条件使用 RAID10，增加单节点性能以及避 免单节点存储故障。

请解释有关 Elasticsearch 的 NRT?

从文档索引（写入）到可搜索到之间的延迟默认一秒钟，因此 Elasticsearch 是 近实时（NRT）搜索平台。

也就是说：文档写入，最快一秒钟被索引到，不能再快了。 写入调优的时候，我们通常会动态调整：refresh_interval = 30s 或者更达 值，以使得写入数据更晚一点时间被搜索到。

elasticsearch 的 document 设计

在使用 es 时 避免使用复杂的查询语句（Join 、聚合），就是在建立索引时， 就根据查询语句建立好对应的元数据。