# 后端开发总结

听明白问题，思考清楚再回答  
听明白问题，思考清楚再回答  
听明白问题，思考清楚再回答

# 简历

2021-2022年的简历  
2022-2023年的简历

# 二、基础知识

## 2.1 计算机网络

学习书籍：

### 2.1.1 tcp,udp

### 2.1.2 http，http2.0,https

### 2.1.4 websocket

## 2.2 操作系统

## 2.3 数据结构

# 算法

# 四、golang语言

## 4.1 内存管理（内存分配，GC）

### GC

GoV1.3- 普通标记清除法，整体过程需要启动STW，效率极低。

GoV1.5- 三色标记法， 堆空间启动写屏障，栈空间不启动，全部扫描之后，需要重新扫描一次栈(需要STW)，效率普通。

GoV1.8-三色标记法，混合写屏障机制， 栈空间不启动，堆空间启动。解决了最后重新扫码一次栈需要进行STW的问题，效率较高。

**标记删除**   
先STW，然后再标记对象，未被标记的对象删除

**三色标记**  
先STW，创建的对象为白色

从根节点查询对象，查询到的对象为灰色

然后从灰色队列里面，往下遍历，灰色的下一个对象标记为灰色，自己修改为黑色，

一直循环遍历上面步骤3中的灰色，直到灰色队列里没有灰色，然后结束STW，再清楚白色对象内存；

**写屏障机制**

解决三色标记里  
白色被黑色对象引用  
白色对象链路可抵达的灰色对象关系被删除

导致误删的情况

**三色不变性**  
强三色不变性：不让黑色应用白色   插入写屏障实现，在黑色对象引用白色对象的时候，将白色对象修改为灰色

或者弱三色不变性：黑色指向的白色对象链路上必须要有灰色  删除写屏障，被删除的对象，如果自身为灰色或者白色，那么被标记为灰色

**写屏障机制短板：**

插入写屏障：结束时需要STW来重新扫描栈，标记栈上引用的白色对象的存活；

删除写屏障：回收精度低，本来该在本次GC中回收的垃圾，可能会逃过这个回合。

**混合写屏障机制**

GC开始阶段，将栈上可抵达的对象全部标记为黑色  
在GC期间，任何在栈上新建的对象，均为黑色

被删除和被添加的对象均标记为灰色

**触发方式**

1. 基于时间的周期性触发(cycle=2分钟)，由监控线程触发：主协程runtime.main函数会创建监控线程，监控线程执行sysmon函数，sysmon函数判断达到gc触发条件唤醒forcegchelper协程，将forcegchelper协程加入全局队列等待调度，forcegchelper协程调用gcStart函数
2. mallocgc触发：用户程序创建对象需要为其分配内存，最终会调用runtime.mallocgc函数，mallocgc判断达到gc触发条件，会调用gcStart函数
3. 手动触发：用户主动调用runtime.GC函数，GC函数调用gcStart函数

**注意：屏障技术是不在栈上应用的，因为要保证栈的运行效率**

## 4.2 调度器，go调度的流程

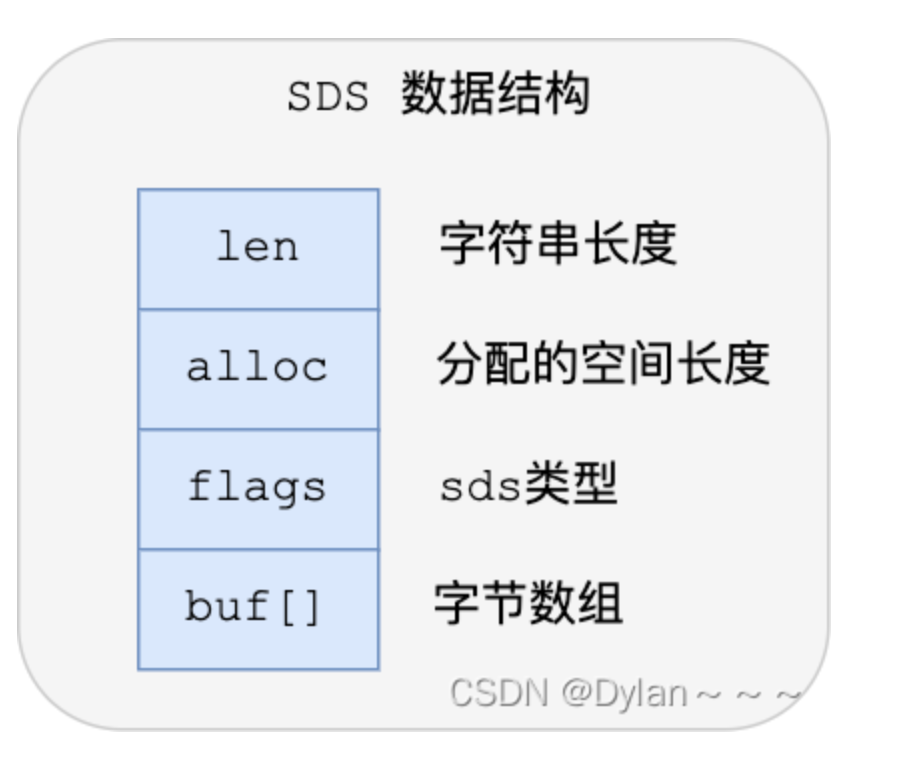
## 4.3 关键字

# 数据库

## 5.1 redis

面经内容：<https://zhuanlan.zhihu.com/p/594544077>  
<https://zhuanlan.zhihu.com/p/91539644>

### 5.1.1 redis的基本数据结构？

**string**：采用SDS结构实现，SDS的数据结构如下，类似于go的切片，当数组内存不足的时候，会自动扩容数组的大小  


list 列表 5.0以前采用

set 无需集合

zset 有序集合  
<https://blog.csdn.net/weixin_37598243/article/details/127625454>  
  
布隆过滤器：通过提前判断数据是否存在数据库，来防止使用者来恶意的攻击软件，比如用id为-1去疯狂查询软件。

### 5.1.2 reids的淘汰策略有哪些？

一般的剔除策略有 FIFO 淘汰最早数据、LRU 剔除最近最少使用、和 LFU 剔除最近使用频率最低的数据几种策略。

no\_eviction(4.0后默认):当内存限制达到并且客户端尝试执行会让更多内存被使用的命令，直接返回错误（大部分的写入指令，但DEL和几个例外）

allkeys-lru: 尝试回收最少使用的键（LRU），使得新添加的数据有空间存放。

volatile-lru: 尝试回收最少使用的键（LRU），但仅限于在过期集合的键,使得新添加的数据有空间存放。

allkeys-random: 回收随机的键使得新添加的数据有空间存放。

volatile-random: 回收随机的键使得新添加的数据有空间存放，但仅限于在过期集合的键。

volatile-ttl: 回收在过期集合的键，并且优先回收存活时间（TTL）较短的键,使得新添加的数据有空间存放。

### 5.1.3 redis的**Pipeline？**

Pipeline的作用是可以批量去执行redis的命令，减少redis客户端跟服务端的请求，

优缺点：

Pipeline 每批打包的命令不能过多，因为 Pipeline 方式打包命令再发送，那么 Redis 必须在处理完所有命令前先缓存起所有命令的处理结果。这样就有一个内存的消耗，可以将大量命令拆分为多个小的Pipeline命令完成

Pipeline 操作是非原子性的，如果要求原子性的，不推荐使用 Pipeline

### 5.1.4 **缓存穿透，缓存击穿，缓存雪崩？**

**缓存穿透**：缓存和数据库中都没有的数据，产生这个问题的原因可能是外部的恶意攻击，比如用用户id为-1去疯狂查询数据，导致db瘫痪,  
1.对接口的请求进行参数校验，如布隆过滤器来过滤请求等。

**缓存击穿：**缓存中没有但数据库中有的数据，就是某个热点数据失效时，大量针对这个数据的请求会穿透到数据源，

1.可以使用互斥锁更新，保证同一个进程中针对同一个数据不会并发请求到 DB，减小 DB 压力。

2.使用随机退避方式，失效时随机 sleep 一个很短的时间，再次查询，如果失败再执行更新。

3.针对多个热点 key 同时失效的问题，可以在缓存时使用固定时间加上一个小的随机数，避免大量热点 key 同一时刻失效

**缓存雪崩**：数据大批量到过期时间，而查询数据量巨大，引起数据库压力过大甚至down机，产生的原因是缓存挂掉，这时所有的请求都会穿透到 DB。

1.使用快速失败的熔断策略，减少 DB 瞬间压力；

2.使用主从模式和集群模式来尽量保证缓存服务的高可用

### 5.1.5 redis的优缺点

1. 支持多种缓存的数据结构  
2. 内存存储数据，查询速度快

3. 单线程，多路复用

### 5.1.5 redis高可用

通过哨兵集群来实现集群的高可用，

### 5.1.6 redis数据持久化

RDB:全量备份，内存快照，可以用于数据的容灾处理，性能高，恢复时间快 ；

AOF：增量备份，它是记录追加的日志文件，降低数据的丢失间隙 ；

4.0 后redis默认采用通过RDB和AOF

### 5.1.6 redis实现分布式锁？

加锁： SET NX PX + 校验唯一随机值

业务逻辑。。。

解锁： Lua脚本（判断加锁是否是本线程在解锁）

### 5.1.7 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。）或者你可以配置内存淘汰机制，当Redis达到内存上限时会冲刷掉旧的内容

## **5.2 mysql**

面经：[https://www.dandelioncloud.cn/article/details/1570040448557477889  
  
https://javabetter.cn/sidebar/sanfene/mysql.html](https://www.dandelioncloud.cn/article/details/1570040448557477889)

### 索引

#### 5.2.1索引有哪些优缺点？

增加查询的速度，但会降低增删改的速度，索引会落盘会增加磁盘的使用

#### 5.2.2 mysql有几种索引？

应用层面： 普通索引，唯一索引，主键索引，复合索引

键值关系: 聚簇索引和非聚簇索引   
聚簇索引的键值都在一颗树上，数据域存储在叶子节点中，非聚簇索引叶子节点需要二次查看才能查询到存储的数据

#### 5.2.3 索引怎么实现的？

可以通过哈希，B树，B+树

#### 5.2.4 为什么索引结构默认使用B+Tree，而不是B-Tree，Hash，二叉树，红黑树？

**哈希实现优缺点**:：等值查询数据非常的快，但是不支持范围查询和排序查询，还存在哈希碰撞问题

1. **tree实现优缺点**：等值查询比b+树要快，但是对于范围查询，排序等功能，速度不如b+树，因为每个叶子节点都有数据，范围查询的时候需要进行一次中序遍历来查询，而b+树数据都在叶子节点中，适合范围查询；

**二叉树：**树的高度不均匀不能自平衡，极端情况可能形成单链表，查找效率跟树的高度有关，在数的高度很大时，IO代价非常高

**红黑树**：数据量增大的情况下，数的高度会增加在，增大IO代价；

B+树的高度范围是1-3层，如果存储bigint主键索引，一页存储大小为1k的话，3层B+数存储的索引信息也是千万级别的数据了；

#### 5.2.5 知道聚簇索引和非聚簇索引？

聚簇索引: 索引信息和数据信息存储在一颗树上

非聚簇索引: 索引信息和数据信息存储在多颗树上

#### 5.2.6 知道回表和索引下推吗？

**回表：**就是指非聚簇索引在查询数据的时候，叶子节点存储的内容是指针等信息，需要进行二次查询才能查询到数据，这个过程叫做回表，

如果你查询的内容从索引数据能查询到的话，也不需要再回表了

**下推：**索引下推，举例 查询a=？,b like ‘%dfaf%’,c like ‘%dfaf%’的时候，a能匹配到索引，但是b，c匹配不到索引，如果不索引下推的话，会把符合a=？的数据全部扫到内存里面，然后再筛选bc，有索引下推的话，就不需要扫下来这些数据，在索引树进行筛选，然后再去拉数据；在5.6版本引入并默认开启的设置（SET optimizer\_switch = ‘index\_condition\_pushdown=off’）；

#### 5.2.7 讲讲Mysql的最左前缀原则？

多个查询的时候，通常把频繁查询的列放在开始，因为多个筛选的时候，会从左往右查询筛选内容；

#### 5.2.8 创建索引需要注意什么？

根据业务需求哈  
索引的字段越小越好；  
尽量扩展索引；  
创建索引尽量局分度高的；

#### 5.2.9 索引失效？

使用模糊like%faf

使用!=或者<>

查询语句有使用到函数

查询语句有运算逻辑

使用or

#### 5.2.9 使用是索引查询一定会提高查询速度吗？

不一定，当数据量小的时候，通过索引查询不一定会比直接查询快

### 基础

#### 5.2.10 Mysql支持哪些存储引擎？

innoDB,myIsam,Memory

innoDB,myIsam区别：

innoDB支持事务，外键，其他的不支持

innoDB，MyIsam的索引都是非聚簇索引，没有聚簇索引  
Innodb 不支持全文索引，而 MyISAM 支持全文索引，查询效率上 MyISAM 要高；

#### 5.2.11 varchar 和char的区别？

varchar 存储的字节内容是范围的，char存储的字节内容是固定的

在能确定字符串字节的大小的时候使用char，不确定的时候用varchar

#### 5.2.12 sql的执行流程？

管理链接，权限校验-->缓存判断-->语法语句分析-->语句优化-->执行，通过存储引擎拉去数据

#### 5.2.13 in和exists 区别？

in 内容为固定的值，exists 用于子查询

#### 5.2.14 drop、delete与truncate的区别

delete 逐行删除，可回滚  
teuncate 删除表列的数据但是还有结构 不可回滚

drop 删除表列的数据和结构 不可回滚

### 事务

#### 5.2.15 mysql的事务怎么实现的？

通过redolog，undolog来实现的

#### 5.2.16 事务日志内容？

redolog：重试日志，在事务执行过程中，开始写入，当发生机器宕机的时候，进行重试，来实现事务的持久性；

undolog：回滚日志，在事务执行前，写入修改值的原来值和修改值，当发生机器宕机的时候，可以来回滚数据，实现事务的原子性；

#### 5.2.17 事务的特性？

ACID

原子性（undolog），一致性，持久性（redolog），隔离性（MVCC实现）

#### 5.2.18 mysql的隔离级别？

读未提交  
读已提交

可重复读

串行执行

#### 5.2.19 知道脏读？幻读？不可重复读？

脏读：事务A,B事务A开始读一个值为1，然后事务B对值进行了修改，事务A在读这个值时，值不一致，产生脏读；

幻读：事务A,B事务A开始读>1的值，然后事务B创建了3条>1的值，事务A再读>1的时候，多出来几个值，想产生了幻觉，所以是幻读；

不可重复读：事务A,B事务A开始读一个值为1，然后事务B对值进行了修改兵并提交，事务A在读这个值时，值不一致，所以是不可重复读；

#### 5.2.20 事务是怎么做隔离的?

通过锁或者MVCC

### 日志

#### 5.2.21知道redo log和undo log ,binlog内容?

binlog:数据表结果或者数据表数据变动产生的二进制文件，主要的目的是恢复复制  
内容分为了：

row：记录每行数据的变更  
statement:只记录sql语句

#### 5.2.22 mvcc是怎么实现多版本控制的？

通过undo日志当快照版本，通过readView来进行控制事务可以看到数据内容

实现：

innodb的每行数据都有一个隐藏的回滚指针执行回滚的日志和事务id

readView会记录创建记录的事务id，最大的事务id，最小的事务id，事务id列表，

系统会根据隔离策略来决定什么时候查看readView，来控制本事务应该看到什么内容，

如 可重复读的隔离级别，就只在事务开始的查寻一次readView然后后面的查询都同这个readView

读已提交的隔离级别，每次查询都会寻一次readView

读可重复读的隔离级别串行读 脏读，不可重复读和幻读的问题  
读已提交的隔离级别 脏读

### 锁

#### 5.2.23 锁有哪些？

颗粒度：

页锁，表锁，行锁  
类别：

共享锁（s）和排他锁（x）

加锁的态度  
悲观锁和乐观锁  
乐观锁就是锁的逻辑在代码上

悲观锁就是直接屏蔽一切可能违反数据完整性的操作

#### 5.2.24 innoDB存储引擎行锁是怎么实现的？

InnoDB引擎的行锁是通过在索引上实现的。当事务要获取一行数据的锁时，首先会根据给定的条件使用索引定位到所需的行，然后在行上设置锁

#### 5.2.25死锁的条件？

死锁是指两个或多个进程在执行过程中，因为竞争资源或互斥关系导致彼此无法前进的一种状态。死锁发生的条件有以下四个：

1. 互斥条件（Mutual Exclusion）：一个资源每次只能被一个进程使用，即该资源同时只能由一个进程占用。

2. 请求并保持条件（Hold and Wait）：一个进程在申请新的资源的同时保持对已占有资源的占用。

3. 不可剥夺条件（No Preemption）：进程已获得的资源在末使用完之前不能被强行剥夺，只能由进程自己释放。

4. 循环等待条件（Circular Wait）：若干进程之间形成一种头尾相接的环形链，每个进程都在等待下一个进程所占有的资源。

当以上四个条件同时满足时，就会发生死锁。为了避免死锁的发生，可以采取一些措施，比如破坏其中一个条件或者引入死锁检测和恢复的机制。

### Mysql优化

#### 5.2.26 mysql优化你的流程是怎么的？

从日志里面拉去频发慢sql，然后在通过expain前，检查查询内容是否准确，查询条数是否满足最左前缀，然后通过expain去查看查询是否走索引。

# 服务架构

## 6.1 分布式

## 6.2 微服务

# 七、云原生

## 7.1 k8s

### 7.1.1 pod 如何对外提供访问？

hostPort：容器开一个端口，通过node的ip+端口访问

Service：它创建的pod会使用主机的ip，使用主机IP+端口访问

service的nodePort，LoadBalancer：通过kube-proxy实现；

ingress访问；

### 7.1.2 iptabel和ipvs的区别 ？

匹配pod的iptabel规则的方式不同  
iptable是时间复杂度O(n),ipvs的时间复杂度是O(1)  
并且ipvs的负载均衡策略更多

iptabel为了防火墙规则使用，ipvs为了高性能的负载均衡

### 7.1.3 固定pod的ip方法？

annotation固定     
cni.projectcalico.org/ipAddrs: "["10.224.0.20"]"

### 7.1.4 controller选主的流程？

采用etcd做一个分布式的乐观锁 lease，configmap，endpoint  
抢到锁的manager当做主master

### 7.1.5 创建一个pod的流程？

创建的请求到apiserver，然后经过apiserver的认证，鉴权，准入流程，然后创建的pod资源序列存储在etcd中

Scheduler  watch到pod创建的时间，然后对pod进行过滤筛选，然后对符合的调度node进行打分，选择打分最高的node，然后修改pod的标签，把绑定node的消息写入，

Kubelete  watch属于本节点的pod，然后获取到新增的pod，并且走创建流程

如果是通过控制器创建的pod，触发的点是controller，通过controller去控制pod资源对象的创建

### 7.1.6 ReplicaSet跟deployment的区别？

ReplicaSet的作用是去唯一创建pod达到预期值

deployment是创建ReplicaSet然后去控制pod达到预期值

deployment还有滚动，批量更新等策略

### 7.1.7  etcd raft算法内容?

选主流程：如果一个追随者在超时时间没有接受领导者的心跳时，就会去成为候选者去进行选主，如果有大多数的赞成，自己就会成为领导者。  
其他节点同意选主的条件是：对方的任期比自己大或者任期相等，日志项编号比自己大；

日志复制流程：领导者会去接受客户端的请求，并复制消息给其他节点，大多数节点如果提交成功的话，就返回成功给客户端；

一致性检查流程

### 7.1.8 etcd 是否会出现数据丢失，缺失?

会，  
多数节点故障：一共3个节点，一个写操作，如果2个节点返回成功，就认为成功了，但是2个节点直接宕机了，就会出现这个写操作丢失；（监控节点状态）

持久化配置错误：提交日志系统是成功的，但是读数据的时候失败，读取不了，导致数据缺失；（监控持久化路径访问权限，以及定时持久化）

### 7.1.9 label和annotation的区别?

Labels是用来筛选对象资源的，a

nnotation扩充对象的元数据，Labels有长度限制，annotation没有长度限制

### 7.1.10 **什么是静态Pod？**

静态 Pod 是k8s集群启动需要的容器pod，系统级别的pod

它只由kubelet管理 ，不由kube-apiserver修改控制

### 7.1.12 **简述Kubernetes Pod的常见调度方式？**

NodeSelector：指定特定的node调度

NodeAffinity亲和性调度：

污点

### 7.1.13 informer的内容？

每一个资源对象都有一个informer，它通过去watch去监控资源的创建，删除，更新等event，通过event的类型来进行对应执行对于控制器逻辑；

使用限流的工作队列包含的限时和基础的队列能力；在源码里他们是功能是叠加的；

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

预选，优选

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

cAdvisor

### 7.1.16 RBAC有哪些创建形式？

管理员创建角色，通过角色绑定，绑定主体和角色的关系，绑定好了，主体就通过角色设置的规则，去达到权限控制

主体有：

user，sa, group

角色有：

roler  针对一个命名空间权限点，clusterRoler 集群权限点

### 7.1.17 两个对象的selector 能直接对比吗？

不能

顺序不同，但是是同一个对象的情况

### 7.1.18 deltaFIFO 事件太多导致内存挂掉？

业务逻辑得支持重试

修改业务的逻辑增加时间处理的速度

以及修改限速配置

## 7.2 docker

overlay2   
https://blog.csdn.net/qq\_15770331/article/details/96699386

## 7.3 containerd

## 7.4 prometheus

### 7.4.1动态拉取targets怎么实现的？

每次执行hang的时候，会去执行reloader对于的ApplyConfig去更新配置，"scrape\_sd"对于的修改的config就是target的配置，ApplyConfig函数回去先去registerProviders，然后再startProvider

### **7.4.2怎么存储数据的，存储数据的流程？**

按label进行hash，找到serires的series map中对应的内存chunk

写入chunk和WAL

该chunk时间跨度是否大于2h或者样本数量是否大于120，如果大于，truncate该chunk写入mmap file，并重置该chunk内容

Mmap file是否超过4个，如果超过，那么重置索引为第五个文件，并修改WAL checkpoint点

### 7.4.3 alartManager

prometheus定时采集指标数据  
prometheus定时计算是否指标触发规则  
触发规则的指标告警状态转为pending，当持续时间超过for指定的时间后，转换为firing，并将告警发送到alertmanager  
alertmanager收到告警后，等待一段分组时间，到时间后发送告警；如果该分组又持续收到了告警，会等待一个分组告警间隔时间后，再次为该分组发送告警  
如果该告警一直存在，alertmanager会按照重发间隔来重复发送告警

# 八、消息队列

# **九、**rpc

## 9.1 rpc是什么？

rpc是远程程序调用，跟http一样是一个应用层的协议，但是他们的应用场景不一样，一个是程序内部调用，类比于，我去调用你的函数，一个是web的应用场景

## 9.2 protobuf是什么？跟json相比为什么快？

protobuf是二进制编码，跟json相比，它不需要解析文本数据，

protobuf跟json相比，protobuf的二进制数据的数据大小更小，在相同的带宽和传输速度下，传输的时间更短

# 十、场景题

## 9.1 一个外卖平台上有一个外卖单子，现在有多名骑手想接这一单，如何保证只有一个骑手可以接到单子？

可以通过分布式锁来实现一个外卖单子被一个骑手抢到

可以用redis的setnx 来实现或者数据库的update 来实现都是采用的乐观锁的实现方式

## 9.2 如何把****一个文件****快速下发到****100w****个服务器？

如果100台机器都有分发的功能，可以采用多叉树的模式去分发传递文件

跟ptp网络协议很类似，同龄人之间互相传递和下载文件

## 9.3给每个****组****分配不同的****IP段****，怎么设计一种结构使的****快速****得知****IP****是哪个组的?

用哈希表做一个映射表，o1的查询数据，key为ip，value为组名  
  
面试官问：如果ip的数据很多呢，不是key会存储很多？  
 如果一个组的ip是有顺序的范围的，就可以压缩数据范围了，把一个范围的ip到这个组存储一个map来使用

面试官问：如果全部都是无序的，不能归纳ip到组的关系呢?

内心（啊这，纯傻逼吧），那就专门有个存储这些数据的组件，比如我用数据库分库分表存储这堆数据，提供查询接口这样子~

## 9.4典型****TOPk****系列的问题：10亿个数，找出最大的10个。等(10万个数，输出从小到大？有十万个单词，找出重复次数最高十个？)

大数据的存算分离，算采用分布式算，可以用多态机器去采访数据，分布计算起topK的结果，然后再汇总每个计算的topK结果  
  
如果没有分布式的场景可以用  
大小堆去实现：分批次取数据，然后把topK的数据去更新大小堆的数据，然后依次处理完数据

时间复杂度：O(n\*logK)

## 9.5 让你设计一个****微信发红包的api****，你会怎么设计，不能有人领到的红包里面没钱，红包数值精确到分。

先根据总金额，需要分配的红包个数和分配规则去分配好红包金额  
提供一个分布式锁的api接口让用户去抢红包

抢到红包在去获取分配好的红包金额

## 9.6 分布式多个机器生成id，如何保证不重复?

雪花算法：根据机器ID 和序列号和时间戳去生成唯一id

## 9.7扫码登录是如何实现的？

它跟JTW的数据验证方式一样的，只是中间有一层把二维码转化为对于token的流程

## 9.8 分布式集群中如何保证线程安全？

分布式锁，版本控制

## 9.9 某网站/app首页每天会从10000个商家里面推荐50个商家置顶，每个商家有一个权值，你如何来推荐？第二天怎么更新推荐的商家？

## 9.10 如何设计一个本地缓存？需要考虑哪些方面？

存储结构分区加锁，确保并发更新数据安全  
支持多重淘汰机制  
内存上限控制

持久化

性能监控和调优 （比如每次触发淘汰机制的时候数据字段++，内存上限触发的时候数据++，缓存命中统计等等）

## 9.11 分布式事务的实现方式？

两阶段提交（强一致性方案）：通过一个协调者组件来协调多个事务是否回滚或者执行完毕，

这个方案会让每个参与者都阻塞住，要么同时全部成功，要么同时全部失败

在搞高并发的场景下就会有问题  
达到了

缺点：缺失了可用性

三阶段：是一样的，防止某个事务超时，给每个事务加了一个超时的状态

TCC(最终一致性的方案):  
它不阻塞每个参与者，比如A参与者执行成功了，直接就退出了，然后通知B参与者  
去执行自己的事务，如果B失败了，在重试执行自己的事务，如果还是不行，就在通知A去回滚自己的事务结果，通过补偿机制，回滚日志做

消息表或者消息队列实现：

因为这设计到通知A，B都需要被通知，所以可以加一个消息表或者消息队列来维护发送消息状态。

幂等问题：

就是同一个订单，不能被扣款多次，被称之为幂等性。

强一致性的场景：

秒杀系统

# 产品设计架构

## 11.1 MVC

model view controller   
简洁快速开发，使用于产品模型简单，业务场景不复杂的情况

## 11.2 DDD

**DDD （Domain-Driven Design）**领域驱动设计

它是一个软件开发的设计思想，需要去考虑领域区分  po，实体怎么设计等等

它把复杂的项目开发领域划分了一下，让业务模块更加的清晰，代码可读性更强

go语言的代码架构  
freedom

<https://github.com/8treenet/freedom/blob/master/example/base/README.md>

# 微服务

它是一个产品整体架构的设计思想

微服务的提出是解决传统大型单体应用程序在扩展性、可维护性和可管理性方面遇到的问题

优点：  
把一个大的产品拆分为了各个小的应用了，每个产品的独立性更强，产品实现方式多样化，提升每个模块的扩展性和高可用

缺点：

成本问题：微服务的数据库和计算资源可能会增加成本。

网络通信问题：服务之间的网络通信可能导致性能问题。