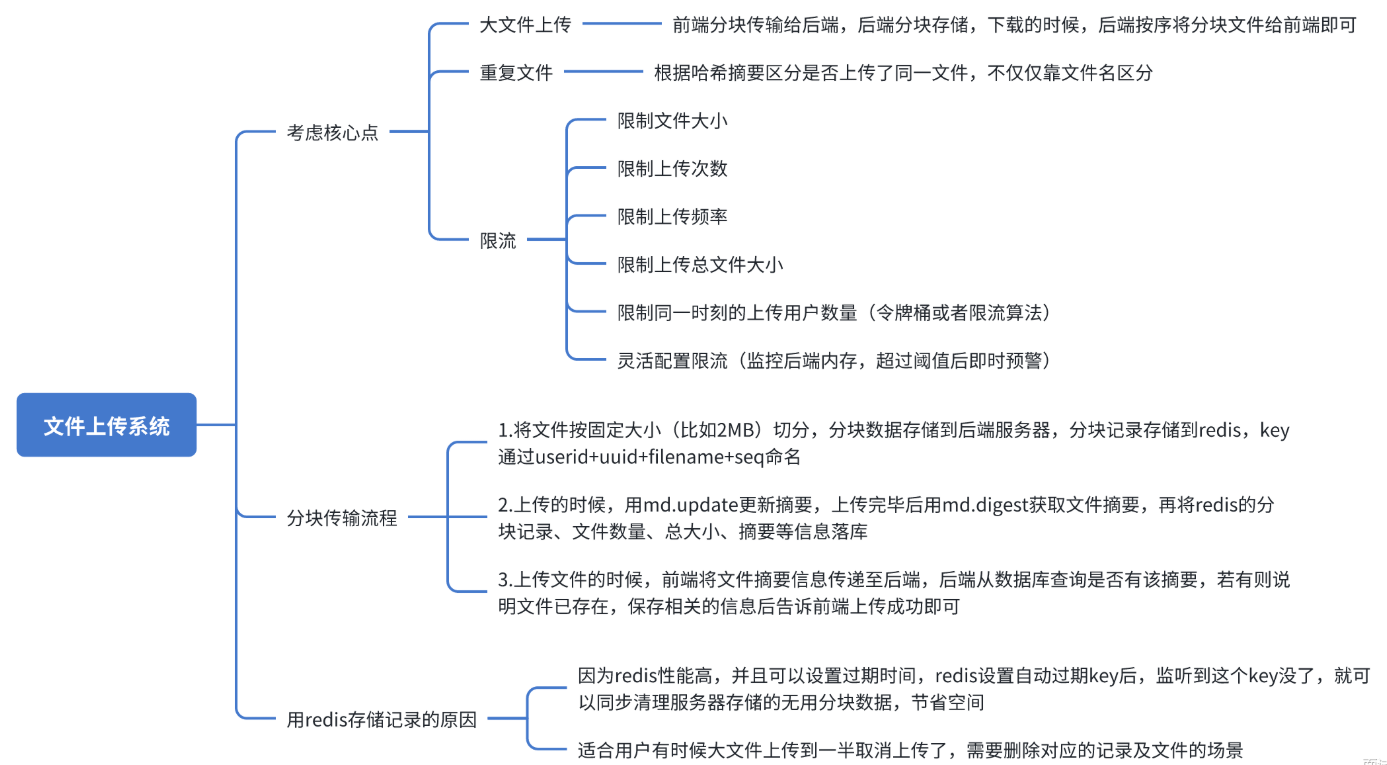
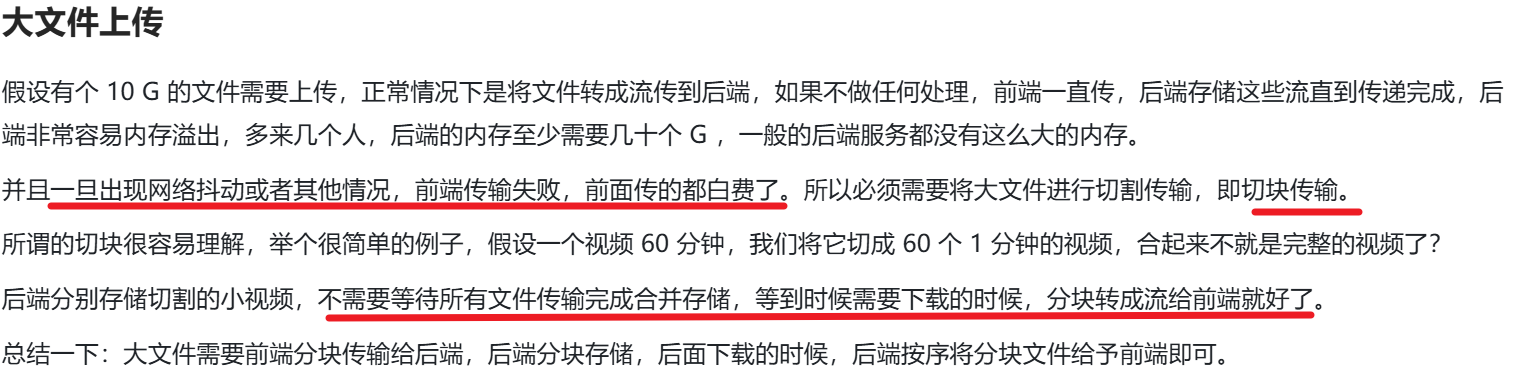
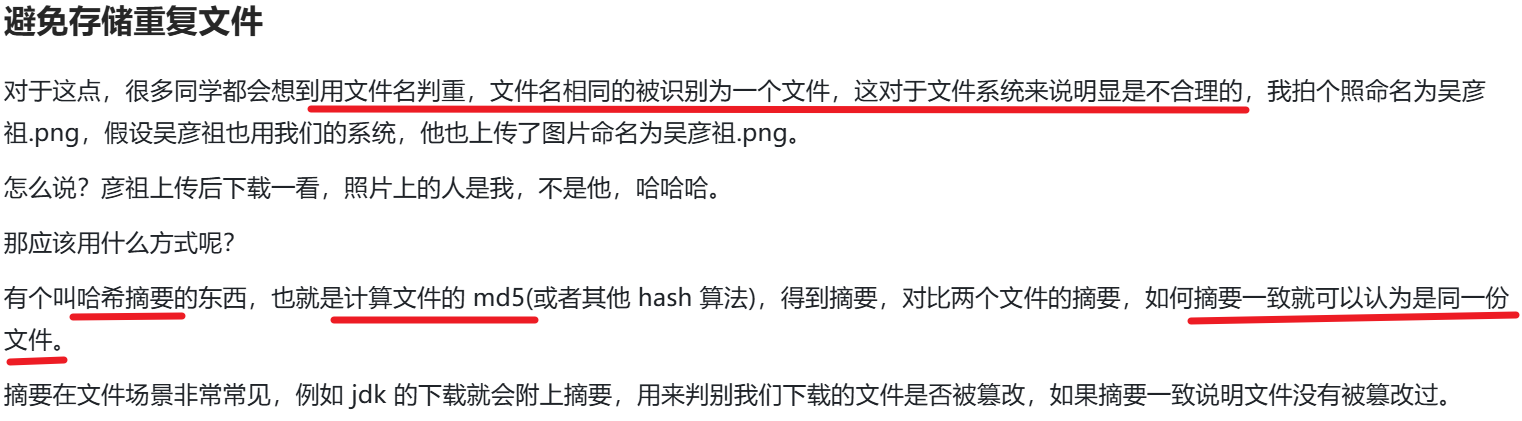
# 系统设计题

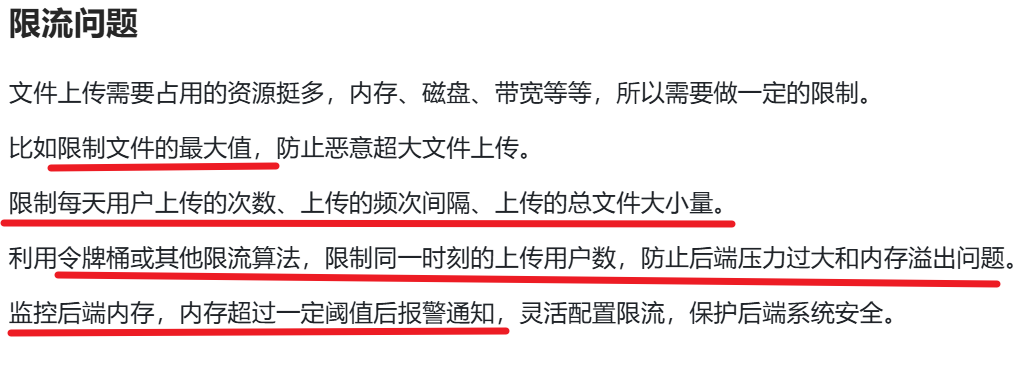
## 1 设计一个文件上传系统？

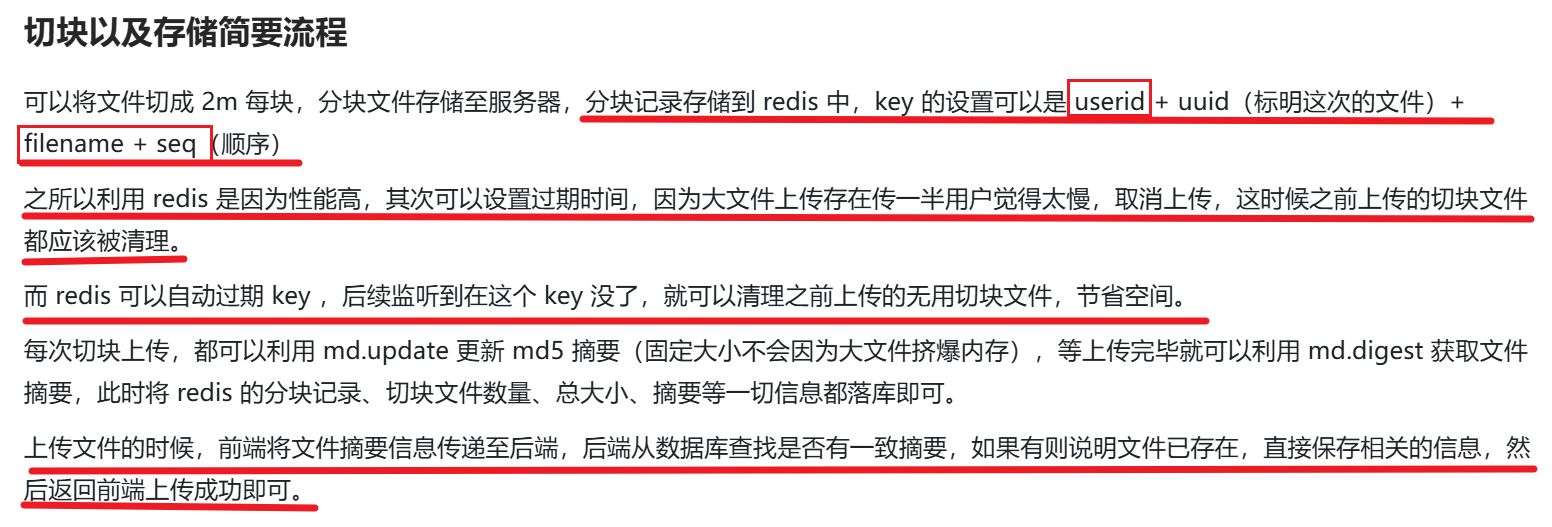






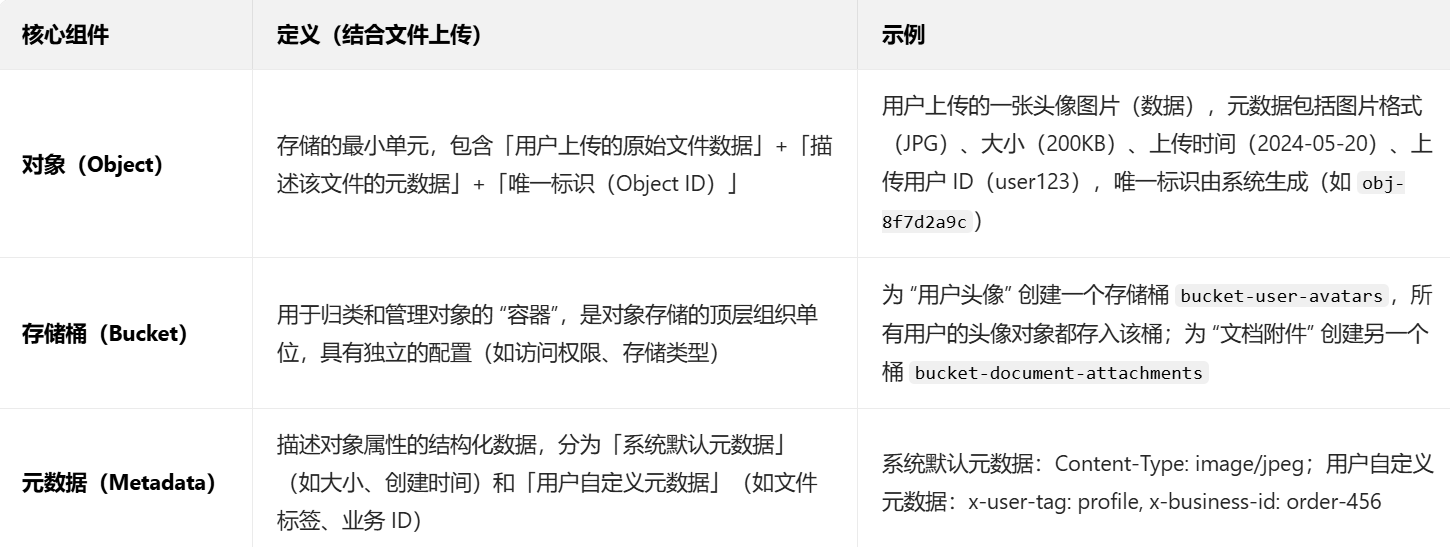






这里文件的存储是用对象存储

在文件上传系统中，**对象存储（Object Storage）**是一种专门用于存储、管理和访问海量非结构化数据（如用户上传的图片、视频、文档、压缩包等）的存储架构。





## 2 直播系统，高并发场景下弹幕/礼物/点赞的统计，直播间人气排序？

### 弹幕、点赞和礼物等在大用户量情况下的实时推送

**技术选型：**

**通信协议：**使用WebSocket协议，因为它是全双工通信，适合实时推送。

**分布式架构：**采用微服务架构，将不同的功能拆分成不同的服务，例如弹幕服务、点赞服务、礼物服务等，便于水平扩展。

**消息队列：**使用高性能的消息队列（如Kafka或RocketMQ）来缓冲和削峰填谷，确保系统稳定性。

**集群与负载均衡：**使用Nginx进行负载均衡，同时WebSocket服务器（如Netty）部署为集群，通过Redis等共享会话信息。

**缓存：**使用Redis来存储在线用户信息、热门直播间的弹幕等，提高读取速度。

**具体实现：**

**弹幕推送：**客户端通过WebSocket连接将弹幕发送到网关；网关层进行基础验证和限流，按房间ID分发到不同Kafka分区；消息处理服务消费Kafka消息，进行内容过滤和业务处理，然后广播服务将处理后的消息推送到同一房间的所有连接，再由统计服务异步更新各类计数指标；为了减轻服务器压力，可以采用批量推送的方式（例如每100毫秒打包一批弹幕发送）。

**点赞：**点赞可能不需要像弹幕那样实时，但也要保证最终实时性。点赞消息可以先写入消息队列，然后由处理服务消费，更新数据库中的点赞数，同时通过WebSocket推送更新信息给主播和观众。对于点赞这种高频操作，可以采用合并多次点赞再推送的方式，比如1秒内的点赞合并为一个消息推送。

**礼物：**由于礼物涉及余额的扣减，所以需要保证送礼事务的原子性；同时我们可以使用优先级队列，高价值的礼物优先处理，提升用户体验；在将礼物消息广播时，可以根据礼物的价值进行广播优化，特大礼物全站广播，普通礼物仅房间内广播。

**优化：**可以采用分级处理，热门直播间分配更多的资源，比如更多的消费者实例；对于大型直播平台，可以在不同地区部署边缘节点，使用户连接到最近的节点，降低延迟。

### 实现主播人气的排序

**人气计算：**主播的人气通常由多个指标组成，例如：实时在线人数、弹幕数量、点赞数量、礼物价值等，这些指标需要实时更新，并且按照一定的权重计算得出人气值。

**技术实现：**

**数据收集：**比如观看人数的统计，可以在用户进入和离开直播间时触发事件，然后使用消息队列去处理事件，然后用户的进入离开需要原子操作，避免并发问题，可以使用redis的**incr和decr**命令，弹幕点赞礼物这些，可以在消息处理的各环节埋点，进行数据采集；

从消息队列中获取数据，按直播间分组，根据预设的人气值计算模型，使用流处理框架，如Flink实时计算每个直播间的人气值；

将计算结果存储到Redis中（使用Sorted Set，以人气值作为分数，直播间ID作为成员），这样就可以实时获取排名。

**排序展示：**

前端定期（比如每10秒）从后端获取主播人气排名。后端从Redis的Sorted Set中获取前N名的主播信息返回给前端。

**注意事项：**

针对redis和mysql这种数据存储组件，需要设计主备切换的机制，实现自动故障转移；在系统压力大的时候可以关闭一些非核心功能，比如弹幕，降级保障核心功能可用；不同的业务使用独立的资源池，避免相互影响；基于历史的数据预测资源需求，提前进行扩容；对于历史人气值，可以设计冷热分离的数据库，把历史数据存储到冷数据库中，用于历史数据离线分析。

## 3 实时统计在线用户前十的主播系统设计?

首先是在线用户的统计方式，可以在**用户进入和离开直播间时触发事件**，然后使用**消息队列**去处理事件，实现流量的削峰填谷。然后用户的进入离开需要**原子操作，避免并发问题**，可以使用redis的**incr和decr**命令。然后可以考虑采用**心跳机制**检测用户是否真的在线，让**客户端每隔一段时间发送一个携带了主播ID的心跳包**，服务器通过检查心跳包来判断连接的存活状态。

其次数据存储问题，实时统计在线人数需要快速读写，传统的关系型数据库可能无法应对高并发，所以考虑使**用Redis+MySQL**。

再者，关于排名功能，**redis的有序集合**（Sorted Set）非常适合这个场景，因为它可以自动维护元素的分数排序，并且支持高效的排名查询。把**在线人数作为分数，直播间ID作为存储元素**，这样每次更新在线人数时，有序集合会自动调整顺序。然后用户可能希望点击排行榜上的热门主播去看这个主播的具体信息，所以可以**用Hash结构去存储主播的具体信息**，用于对单个主播的详细查询。针对排名更新的话，可以采用**定时任务**，比如**每隔3秒去刷新一次排名**，或者**房间人数变化超过阈值**的话就立即触发更新。

## 4 设计外卖系统？

首先是系统架构上，可以考虑采用分布式架构，把任务拆分成**用户服务、商家服务、订单服务、配送服务、支付服务**等。针对数据可以考虑采用**MySQL+Redis的存储结构**，用户下单后可能会频繁查询订单状态，因此可以把**订单状态缓存到redis**，缓存的过期时间可以设置成订单从创建到完成的时间。

外卖系统可能需要重点考虑几个功能，首先是**抢单**，可以基于地理位置进行分片，这里可以用redis的**GEO数据类型**，把订单推送到对应区域的订单池中，抢单的时候使用**分布式锁**来保证一个订单不会被重复抢。

针对商家，可能需要考虑**[库存不超卖](#_高并发场景保证优惠券领取不超卖，Redis扛不住怎么办？)**[的问题](#_高并发场景保证优惠券领取不超卖，Redis扛不住怎么办？)，这里可以考虑使用分布式锁+分段缓存的方式。

针对订单，一方面要**避免用户重复下单**，这个可以在**前端提交之后禁用按键**，在后端**针对每个订单可以基于雪花算法生成一个唯一id**保证操作的**幂等性**。一方面需要实现**超时未支付自动取消**，这里可以考虑采用**延迟队列实现**，比如借助RabbitMQ的[死信队列+消息存活时间](#_RabbitMQ怎样实现延迟队列？)来实现，比如把状态是未支付的订单路由到一个没有消费者的队列中，可以设置消息存活时间15分钟，超过15分钟消息还没有消费就转发给死信队列，执行订单取消的操作，并且**补偿预扣的库存**。定时方案的话，可以比如每1分钟扫描一次表，直接根据当前时间和超时时间进行筛选，比如超时取消时间是15分钟，当前是14点，那么13点45分之前创建的订单未付款其实就超时了，所以可以根据13点45分和未支付两个条件进行数据库查询，得到超时的订单，最后批量更新超时订单状态（一般采用分布式定时任务框架，比如xxljob等，如果涉及到多机任务并发执行提升效率的情况，需要对订单进行哈希取模分片，比如机器1处理订单号取模5余1的，机器2处理订单取模余2的）。定时任务的方案缺点就是，超时时间可能不准确，比如1分钟执行一次，那么相差时间可能就是一分钟。

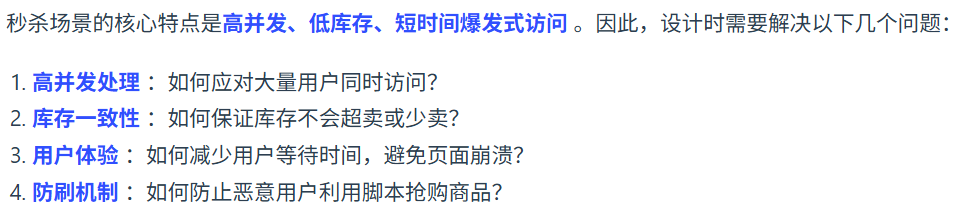
## 5 高并发场景保证优惠券领取不超卖，Redis扛不住怎么办？

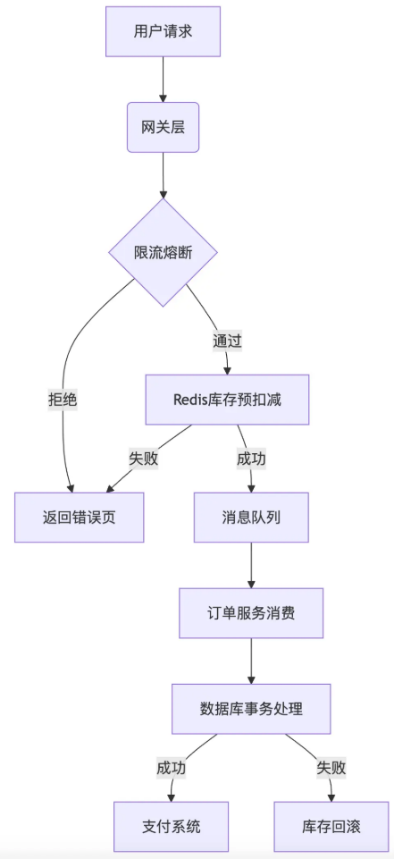
不超卖：首先的话可以直接在数据库层面解决，比如在更新库存的时候加**排他锁**，更新数据库库存的时候进行**库存数的检查**；这种方式的话在高并发场景下性能不太好；

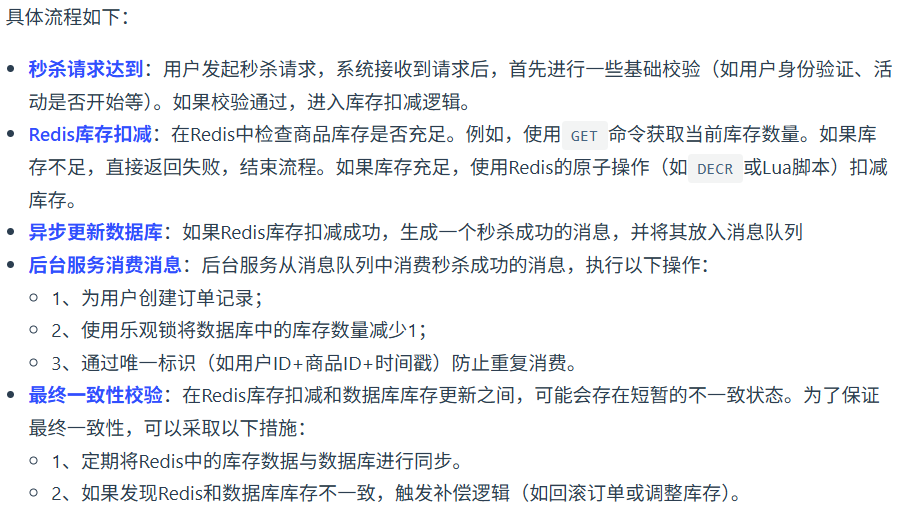
其次可以考虑**分布式锁结合分段缓存**的方式，把**库存分成多段进行存储**，比如库存100，可以在redis中分成5个库存key，用户下单时对用户id取余5，看落在哪个redis的key上，就去取哪个段的库存，这样就能同时处理5个请求；**当某个段的库存不足，就自动释放锁然后换到下一个分段库存尝试获取**；

Redis扛不住：可以考虑采用**redis分片集群**；或者**构建多级缓存**，把**最热门的部分券放到本地缓存**，大部分次热门的券放到redis，缓解redis的压力。

## 6 如何设计一个秒杀系统？（短时高并发）



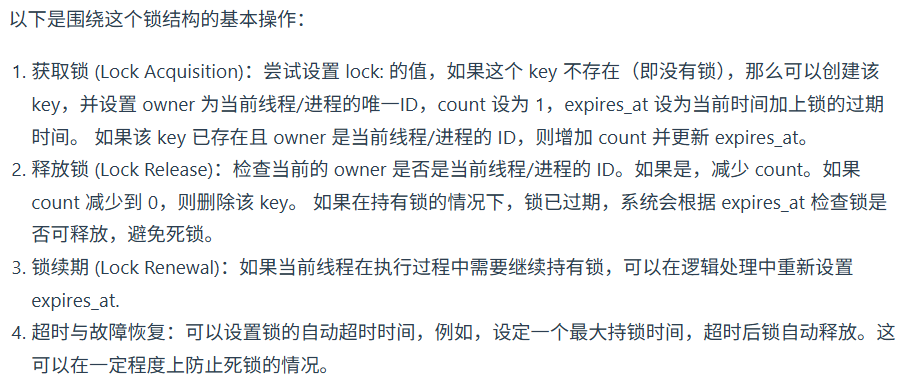


## 7 如果要做一个大流量的网站，单Redis无法承载怎么办

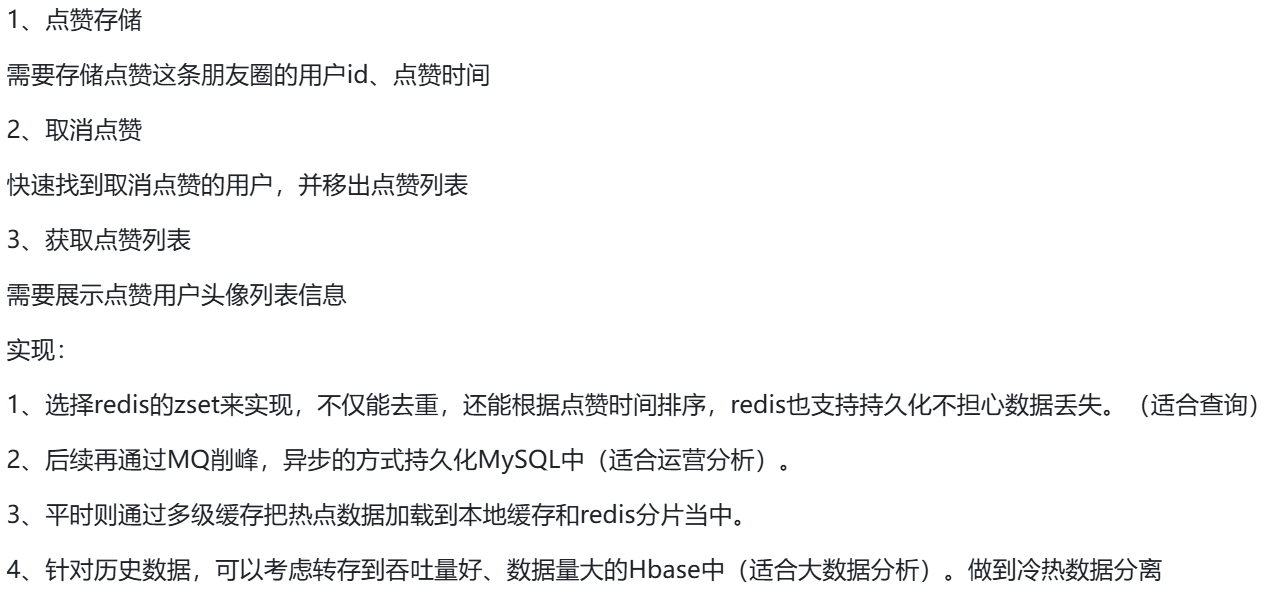


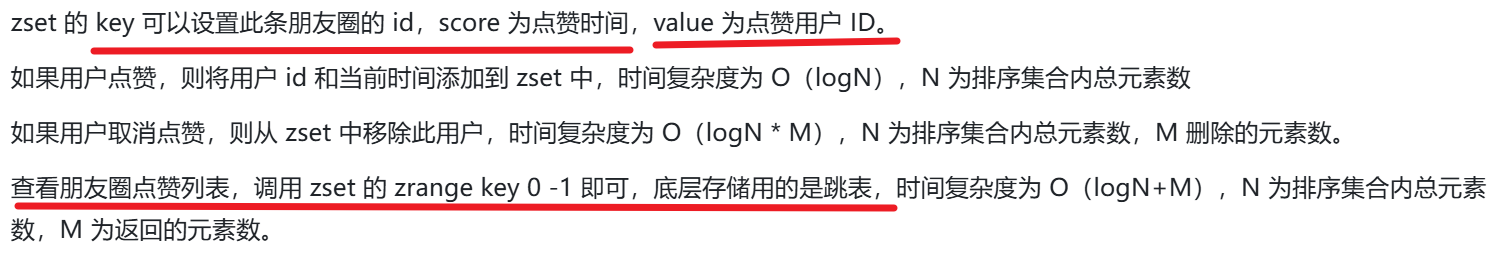
## 8 设计一个可重入锁？





## 9 朋友圈点赞功能的设计？

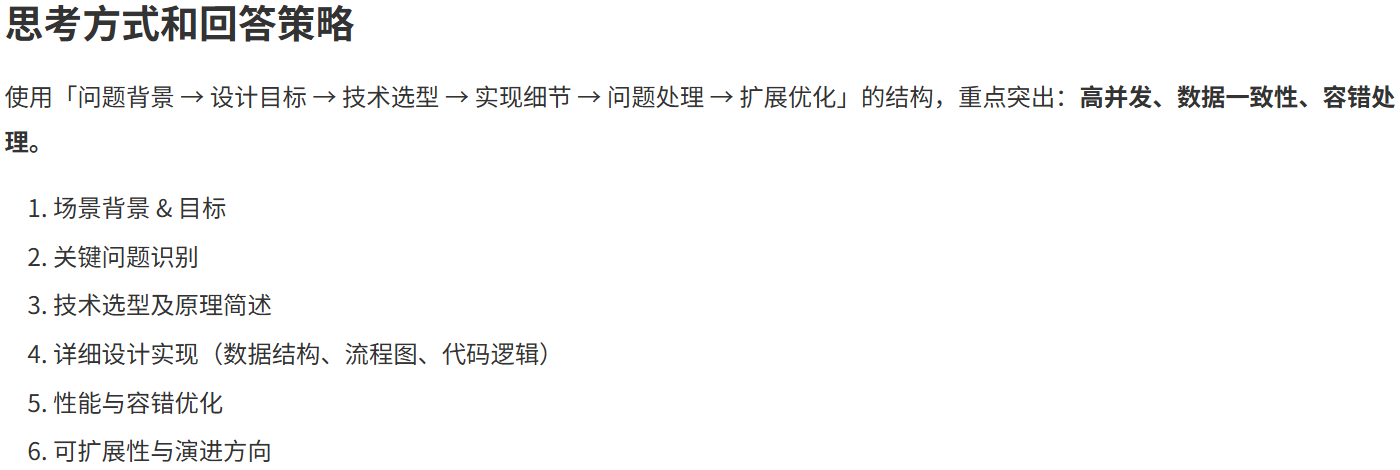




**系统设计总结：**

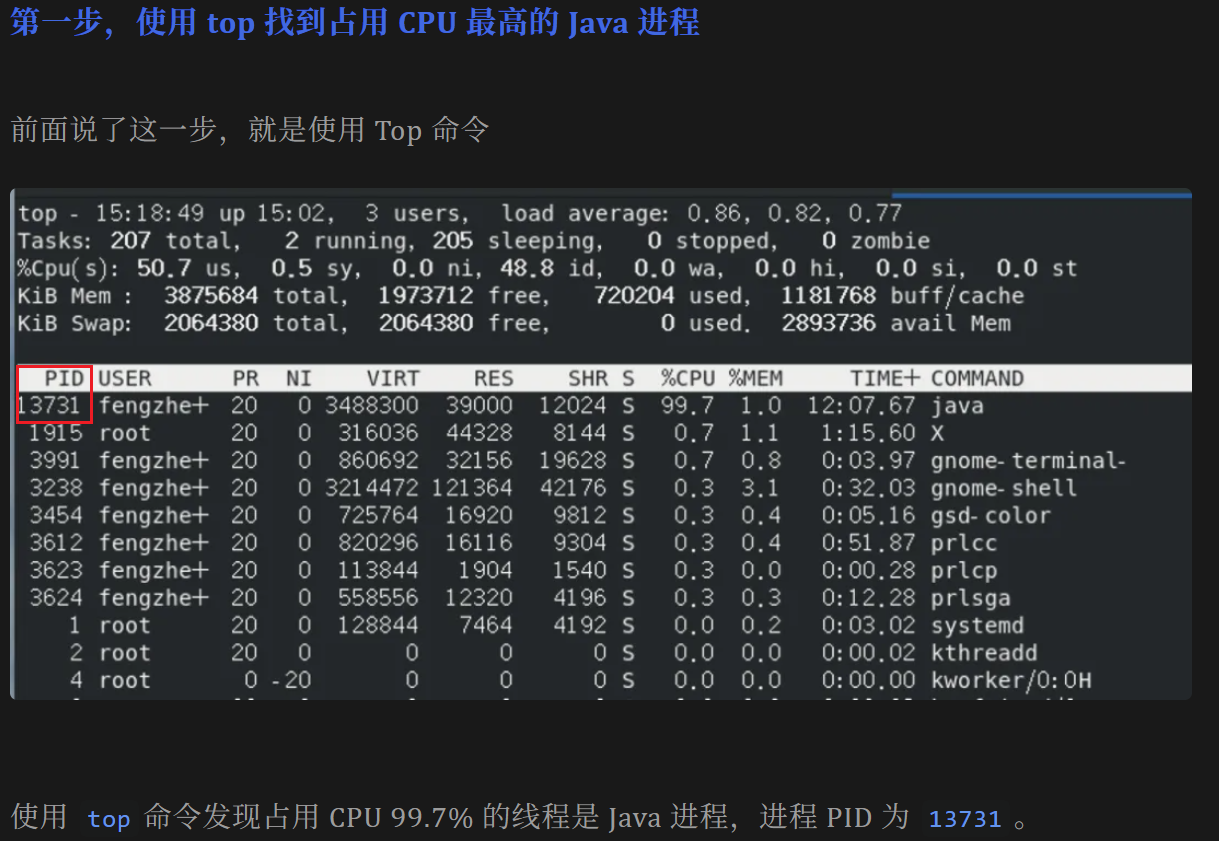




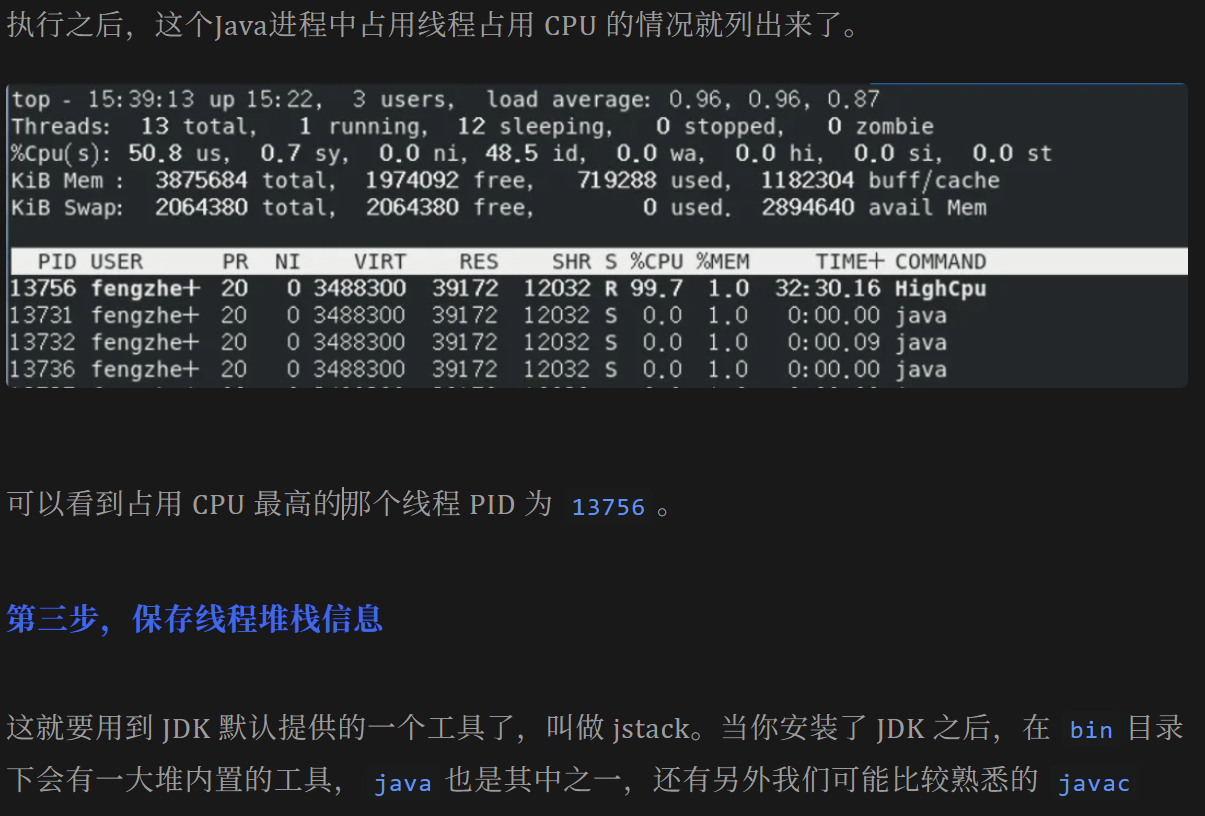


# 场景题

## 10 CPU 100%问题怎么排查？









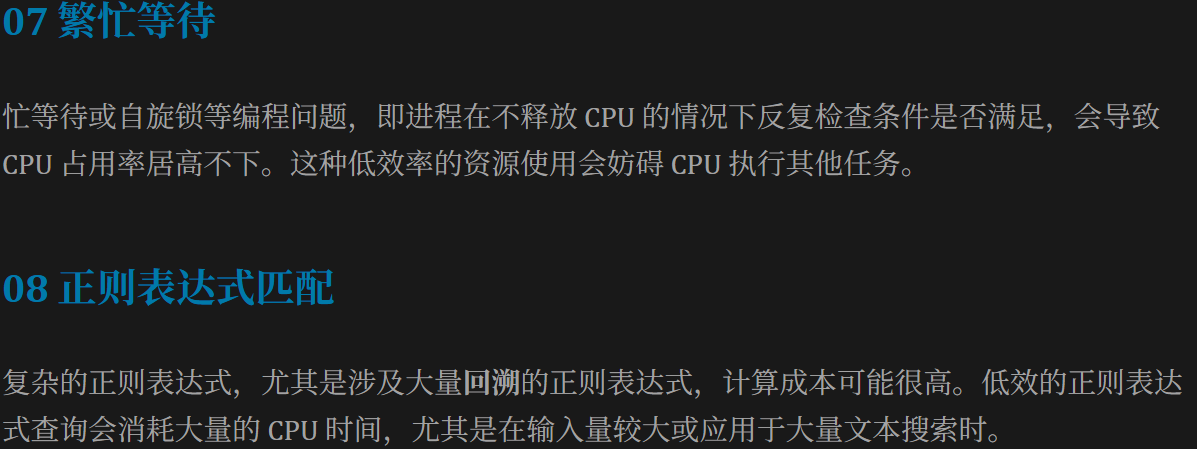




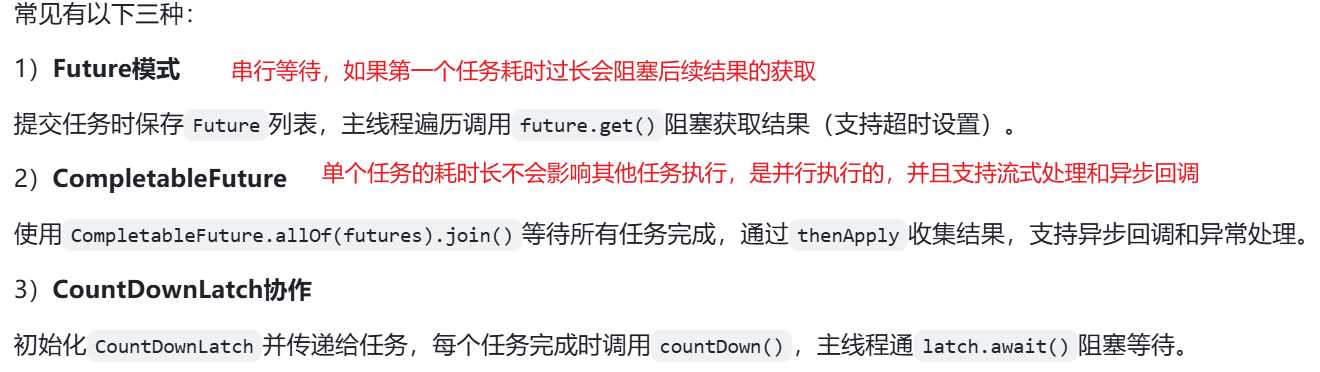
## 相关问题：导致CPU飙高的原因？



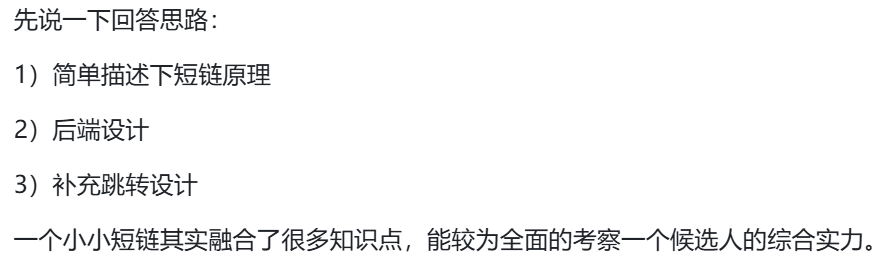


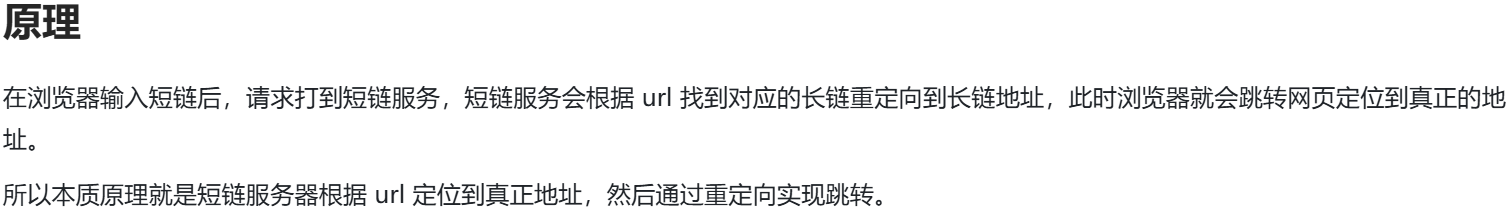


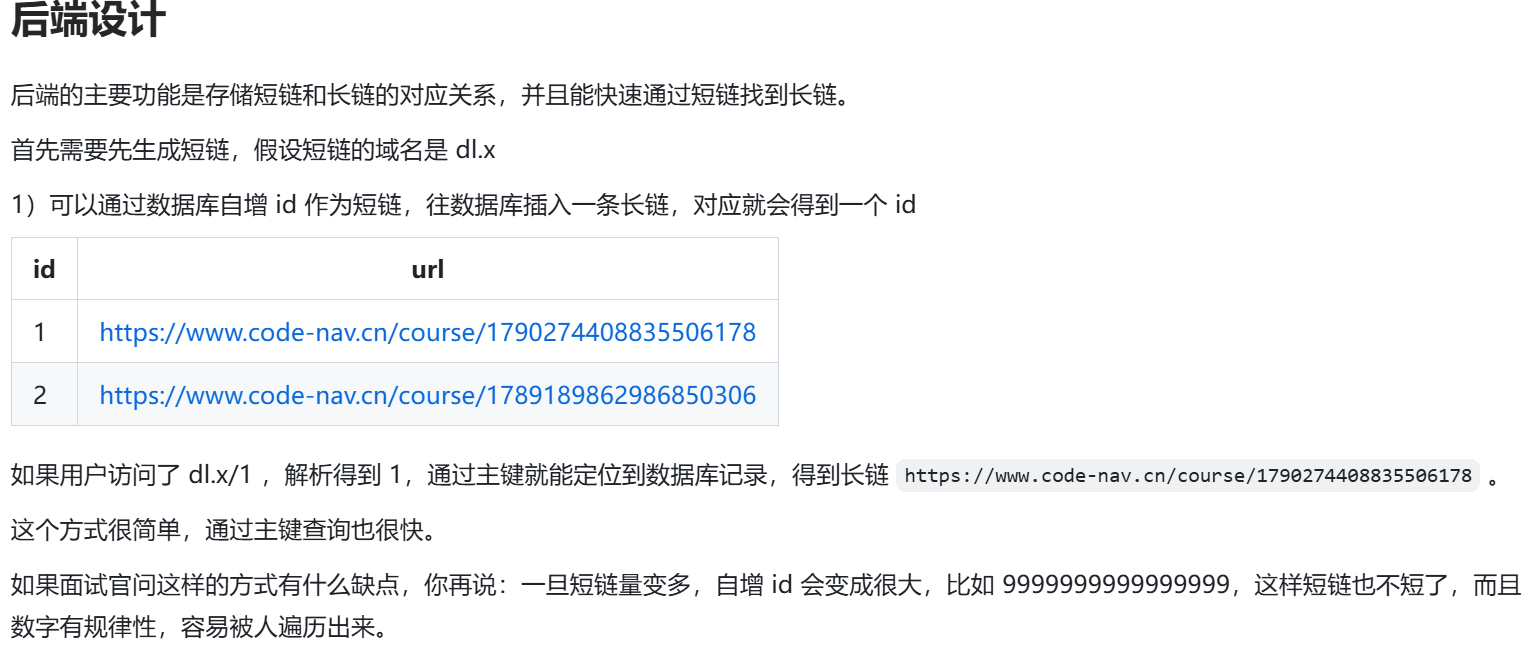
## 11 主线程如果得知线程池中多个并行任务的完成，并收集他们的结果？

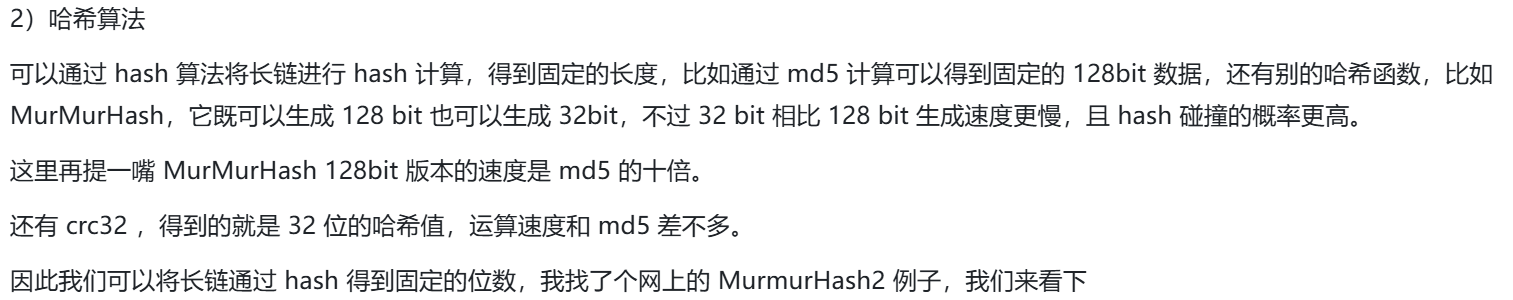


## 12 短链系统设计

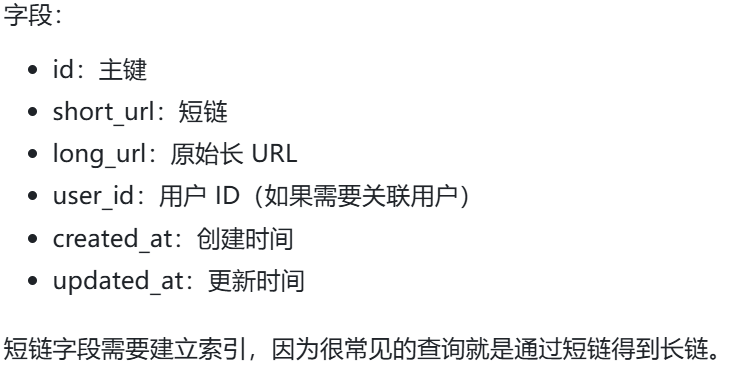


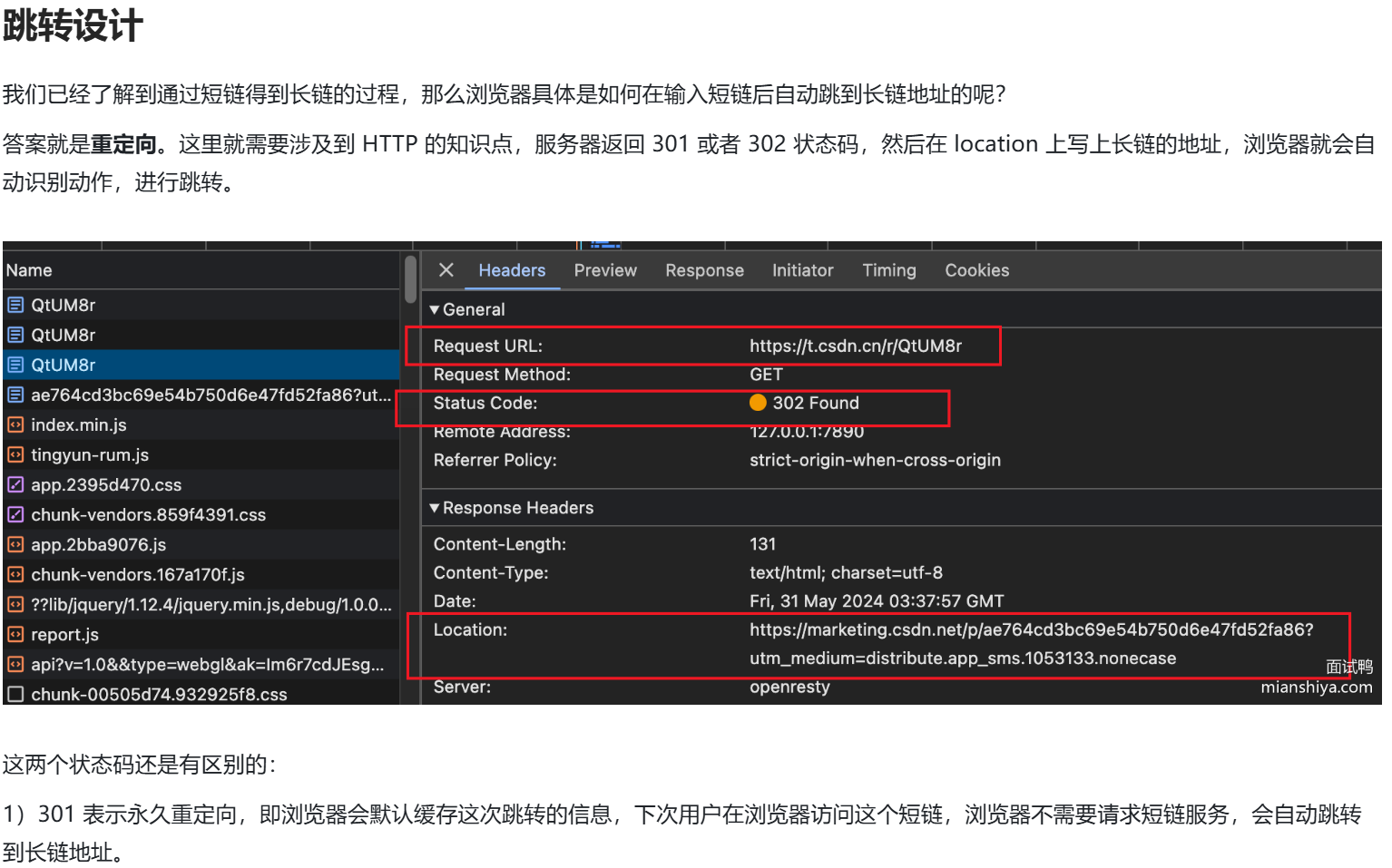


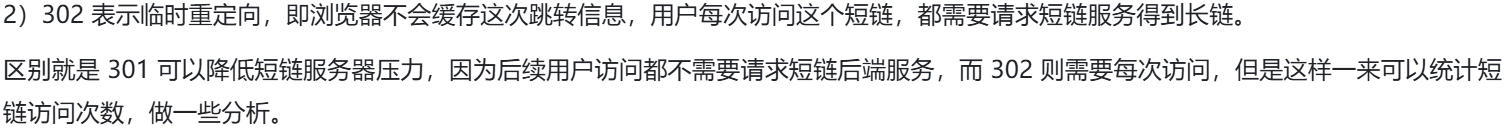


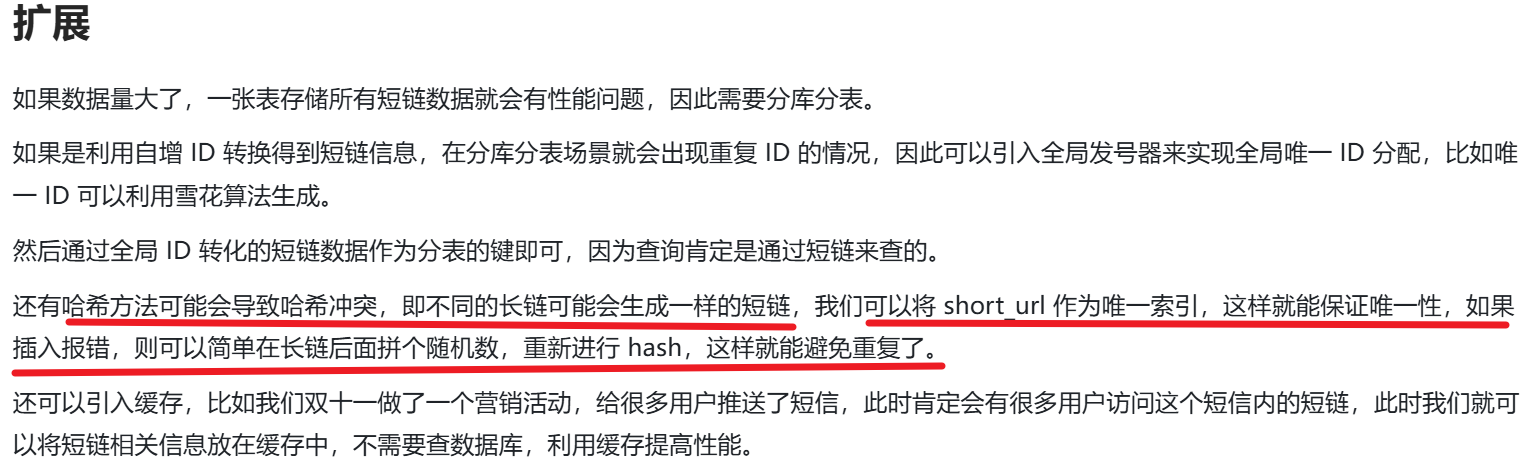












## 13 如何提高程序的性能？

**（1）代码层面**

* 根据场景选择合适的数据结构，比如，随机访问多的情况用ArrayList存储，频繁插入删除的用LinkedList存储；需要键值对且无序用HashMap；
* 对于集合，如果可以预估大小，尽量在创建的时候指定初始容量，避免频繁扩容带来的额外开销；
* 关注代码的时间复杂度，比如可以是哈希表查找代替遍历列表查找；
* 如果可以用基本数据类型的情况，多用基本数据类型而不是包装类，因为基本数据类型占用的内存比包装类对象占用的空间要小很多；
* 在循环或频繁拼接字符串的场景下，坚决不使用+操作符，而使用StringBuilder（线程不安全）或StringBuffer（线程安全，使用synchronized关键字修饰），因为它避免了创建大量临时的String对象；
* 对于不在使用的大对象引用显式置为null，有助于GC更早回收内存。

1. **并发和多线程层面**

* **利用线程池：**绝不手动创建线程，而是使用ThreadPoolExecutor，这样可以减少线程创建和销毁的开销，并有效控制系统资源；
* **使用并发集合：**如ConcurrentHashMap、CopyOnWriteArrayList等，它们在保证线程安全的同时，通常比使用synchronized包装的集合性能高得多；
* **减小锁粒度：**将一个大锁拆分成多个小锁，减少锁竞争。例如，用多个锁保护不同的数据段，而不是一个锁保护所有数据；
* **使用无锁编程：**在可能的情况下，使用java.util.concurrent.atomic包下的原子类（如AtomicInteger），它们基于CAS（Compare-And-Swap）操作，性能通常优于锁；
* **使用读写锁：**对于“读多写少”的场景，使用ReentrantReadWriteLock，允许多个读线程同时访问，提高并发性；
* **使用**volatile**关键字：**保证变量的可见性，避免线程从工作内存中读取过期的数据，是一种轻量级的同步机制。

**（3）JVM层面**

* **JVM参数调优：**

**堆内存设置：**通过-Xms和-Xmx设置合理的初始堆和最大堆大小，**将初始堆(**-Xms**)和最大堆(**-Xmx**)设置为相同值**。这样可以避免运行时动态调整堆大小带来的性能开销，同时防止在扩容时因内存不足而发生的停顿。

**选择垃圾收集器：**

高吞吐量场景：Parallel并行垃圾回收器。

低延迟响应场景：CMS（已废弃）或G1、ZGC。

**新生代与老年代比例：**根据对象生命周期特点，通过-XX:NewRatio进行调整。

* **编写对GC友好的代码：**

核心是**减少垃圾产生的数量**和**缩短对象的生命周期**。

避免创建生命周期很短的大对象，因为它们会直接进入老年代，增加Full GC的压力。

**（4）数据库层面**

* **SQL优化：**

**建立有效的索引：**这是最有效的手段。分析慢查询，为WHERE、ORDER BY、JOIN的字段建立索引。

**避免**SELECT \***：**只取需要的字段，减少网络传输和数据加载的开销。

**优化SQL语句：**避免在索引列上使用函数或计算，避免使用%开头的模糊查询。

**使用连接池：**如Druid，避免频繁创建和关闭数据库连接。

**（5）缓存层面**

**本地缓存：**对于数据量小、更新不频繁的数据，使用Caffeine，访问速度极快。

**分布式缓存：**对于大型应用，使用Redis等，缓解数据库压力，共享缓存数据。

**缓存策略：**合理设置过期时间、更新策略（如Cache-Aside、Write-Through）。

**（6）架构设计方面**

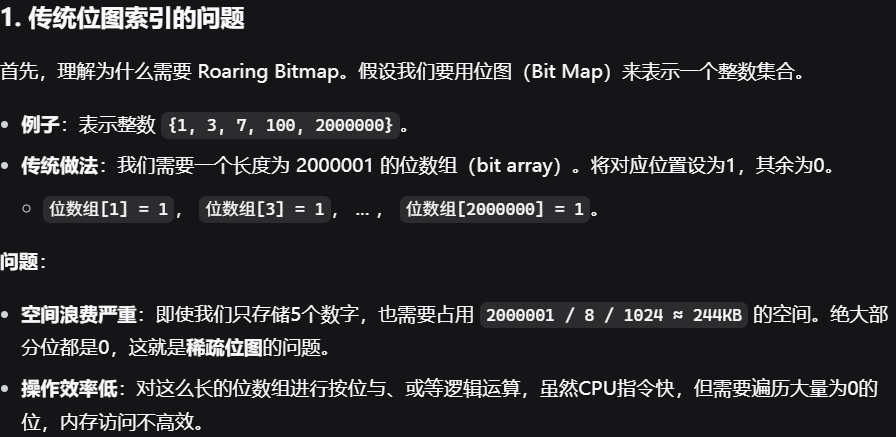
* **异步与虚拟线程思想：**

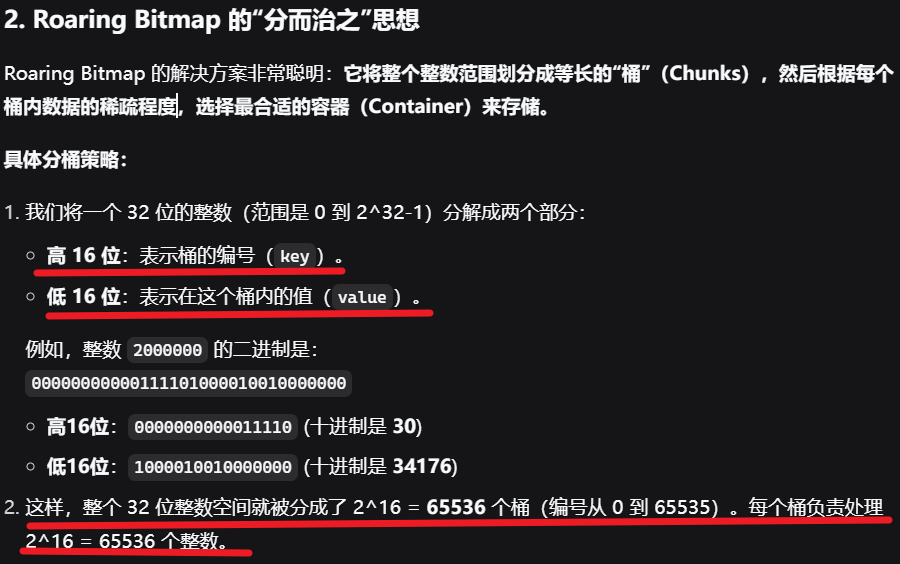
**异步处理：**对于非实时性任务（如发送邮件、记录日志），使用消息队列（如RabbitMQ、Kafka）进行异步解耦，快速响应用户请求。

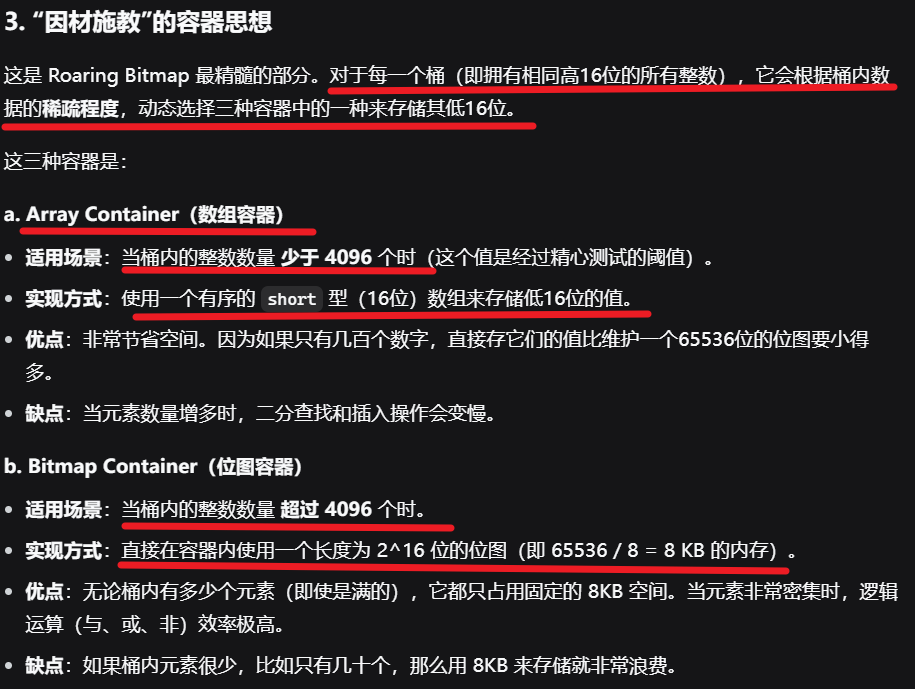
**Java 21虚拟线程：**传统线程是系统内核管理的，创建开销的成本较高，而java提出了虚拟线程的概念，每个线程只有几百个字节，创建开销成本极低，由JVM进行管理，多个虚拟线程承载在一个操作系统内核线程上，我们的IO操作实际运行在虚拟线程上，当某个虚拟线程IO阻塞，我们可以把这个虚拟线程挂起，从而释放他依赖的内核线程，然后可以马上换另一个虚拟线程来继续执行。

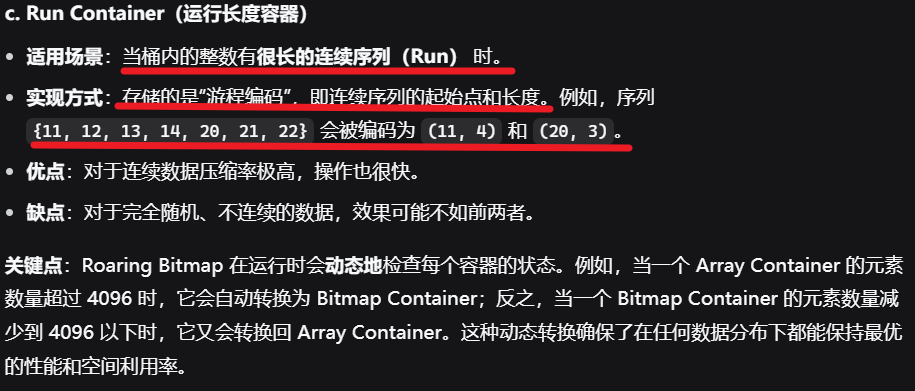
* **应用拆分与微服务：**将单体应用拆分为微服务，每个服务可以独立扩展，针对瓶颈服务进行针对性优化。
* **静态资源优化：**使用CDN加速静态资源（图片、CSS、JS）的访问。

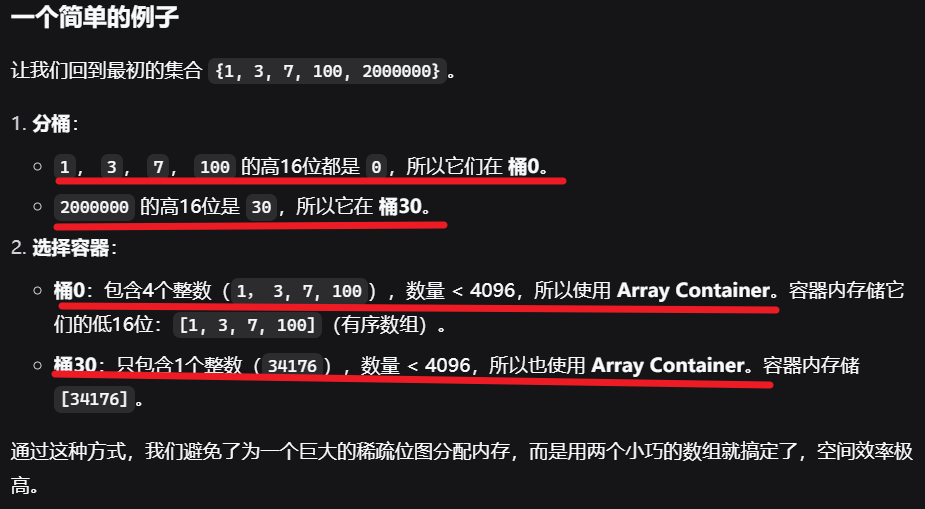
## 14 用BitMap来存储100个用户ID，但是这个ID范围很大，此时会有什么问题？











## 15 线上接口响应变慢怎么排查？

