### 走一遍

systemctl stop firewalld.service #停止firewall

systemctl disable firewalld.service #禁止firewall开机启动

firewall-cmd --state #查看默认防火墙状态（关闭后显示notrunning，开启后显示running）

# 只要问到缓存，上来第一个问题，肯定能是先问问你项目哪里用了缓存？为啥要用？不用行不行？如果用了以后可能会有什么不良的后果？

### （1）在项目中缓存是如何使用的？

这个，你结合你自己项目的业务来，你如果用了那恭喜你，你如果没用那不好意思，你硬加也得加一个场景吧

### （2）为啥在项目里要用缓存呢？

用缓存，主要是俩用途，高性能和高并发

1）高性能

假设这么个场景，你有个操作，一个请求过来，吭哧吭哧你各种乱七八糟操作mysql，半天查出来一个结果，耗时600ms。但是这个结果可能接下来几个小时都不会变了，或者变了也可以不用立即反馈给用户。那么此时咋办？

缓存啊，折腾600ms查出来的结果，扔缓存里，一个key对应一个value，下次再有人查，别走mysql折腾600ms了。直接从缓存里，通过一个key查出来一个value，2ms搞定。性能提升300倍。

这就是所谓的高性能。

就是把你一些复杂操作耗时查出来的结果，如果确定后面不咋变了，然后但是马上还有很多读请求，那么直接结果放缓存，后面直接读缓存就好了。

2）高并发

mysql这么重的数据库，压根儿设计不是让你玩儿高并发的，虽然也可以玩儿，但是天然支持不好。mysql单机支撑到2000qps也开始容易报警了。

所以要是你有个系统，高峰期一秒钟过来的请求有1万，那一个mysql单机绝对会死掉。你这个时候就只能上缓存，把很多数据放缓存，别放mysql。缓存功能简单，说白了就是key-value式操作，单机支撑的并发量轻松一秒几万十几万，支撑高并发so easy。单机承载并发量是mysql单机的几十倍。

3）所以你要结合这俩场景考虑一下，你为啥要用缓存？

一般很多同学项目里没啥高并发场景，那就别折腾了，直接用高性能那个场景吧，就思考有没有可以缓存结果的复杂查询场景，后续可以大幅度提升性能，优化用户体验，有，就说这个理由，没有？？那你也得编一个出来吧，不然你不是在搞笑么

### （3）用了缓存之后会有啥不良的后果？

呵呵。。。你要是没考虑过这个问题，那你就尴尬了，面试官会觉得你头脑简单，四肢也不发达。你别光是傻用一个东西，多考虑考虑背后的一些事儿。

常见的缓存问题有仨（当然其实有很多，我这里就说仨，你能说出来也可以了）

1）缓存与数据库双写不一致

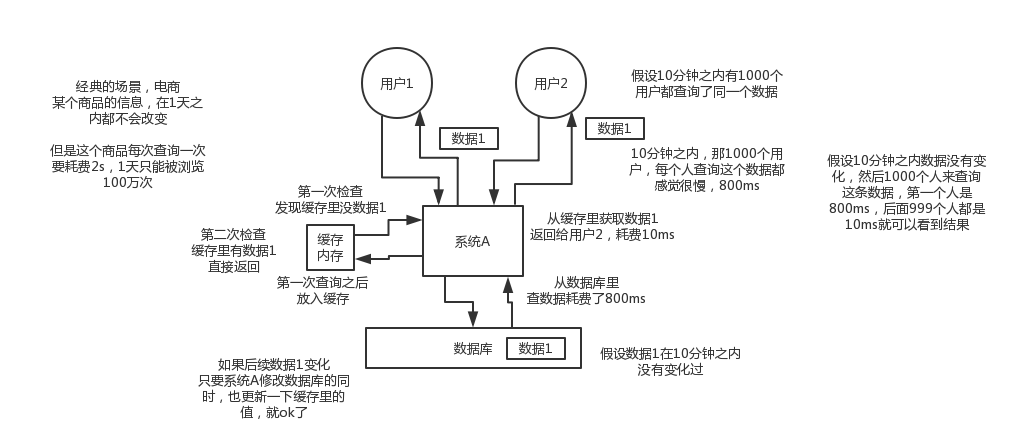
2）缓存雪崩

3）缓存穿透

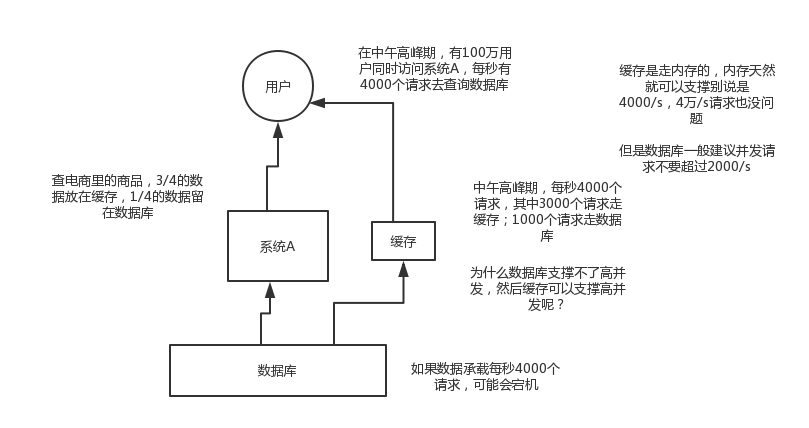
4）缓存并发竞争

这仨问题是常见面试题，后面我要讲，大家看到后面自然就知道了，但是人要是问你，你至少自己能说出来，并且给出对应的解决方案

缓存如何实现高性能?



缓存如何实现高并发?



# redis和memcached有什么区别？redis的线程模型是什么？为什么单线程的redis比多线程的memcached效率要高得多（为什么redis是单线程的但是还可以支撑高并发）？

还有可能面试官会问问你redis和memcached的区别，不过说实话，最近这两年，我作为面试官都不太喜欢这么问了，memched是早些年各大互联网公司常用的缓存方案，但是现在近几年基本都是redis，没什么公司用memcached了

### （1）redis和memcached有啥区别

这个事儿吧，你可以比较出N多个区别来，但是我还是采取redis作者给出的几个比较吧

1）Redis支持服务器端的数据操作：Redis相比Memcached来说，拥有更多的数据结构和并支持更丰富的数据操作，通常在Memcached里，你需要将数据拿到客户端来进行类似的修改再set回去。这大大增加了网络IO的次数和数据体积。在Redis中，这些复杂的操作通常和一般的GET/SET一样高效。所以，如果需要缓存能够支持更复杂的结构和操作，那么Redis会是不错的选择。

~~2）内存使用效率对比：使用简单的key-value存储的话，Memcached的内存利用率更高，而如果Redis采用hash结构来做key-value存储，由于其组合式的压缩，其内存利用率会高于Memcached。~~

~~3）性能对比：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，虽然Redis最近也在存储大数据的性能上进行优化，但是比起Memcached，还是稍有逊色。~~

4）集群模式：memcached没有原生的集群模式，需要依靠客户端来实现往集群中分片写入数据；但是redis目前是原生支持cluster模式的，redis官方就是支持redis cluster集群模式的，比memcached来说要更好

### （2）redis的线程模型

1）文件事件处理器

redis基于reactor模式开发了网络事件处理器，这个处理器叫做文件事件处理器，file event handler。这个文件事件处理器，是单线程的，redis才叫做单线程的模型，采用IO多路复用机制同时监听多个socket，根据socket上的事件来选择对应的事件处理器来处理这个事件。

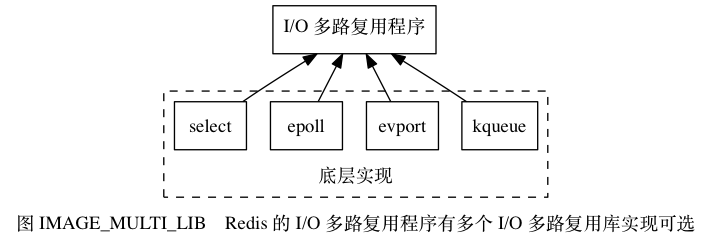
如果被监听的socket准备好执行accept、read、write、close等操作的时候，跟操作对应的文件事件就会产生，这个时候文件事件处理器就会调用之前关联好的事件处理器来处理这个事件。

文件事件处理器是单线程模式运行的，但是通过IO多路复用机制监听多个socket，可以实现高性能的网络通信模型，又可以跟内部其他单线程的模块进行对接，保证了redis内部的线程模型的简单性。

文件事件处理器的结构包含4个部分：多个socket，IO多路复用程序，文件事件分派器，事件处理器（命令请求处理器、命令回复处理器、连接应答处理器，等等）。

多个socket可能并发的产生不同的操作，每个操作对应不同的文件事件，但是IO多路复用程序会监听多个socket，但是会将socket放入一个队列中排队，每次从队列中取出一个socket给事件分派器，事件分派器把socket给对应的事件处理器。

然后一个socket的事件处理完之后，IO多路复用程序才会将队列中的下一个socket给事件分派器。文件事件分派器会根据每个socket当前产生的事件，来选择对应的事件处理器来处理。



2）文件事件

当socket变得可读时（比如客户端对redis执行write操作，或者close操作），或者有新的可以应答的socket出现时（客户端对redis执行connect操作），socket就会产生一个AE\_READABLE事件。

当socket变得可写的时候（客户端对redis执行read操作），socket会产生一个AE\_WRITABLE事件。

IO多路复用程序可以同时监听AE\_REABLE和AE\_WRITABLE两种事件，要是一个socket同时产生了AE\_READABLE和AE\_WRITABLE两种事件，那么文件事件分派器优先处理AE\_REABLE事件，然后才是AE\_WRITABLE事件。

3）文件事件处理器

如果是客户端要连接redis，那么会为socket关联连接应答处理器

如果是客户端要写数据到redis，那么会为socket关联命令请求处理器

如果是客户端要从redis读数据，那么会为socket关联命令回复处理器

4）客户端与redis通信的一次流程

在redis启动初始化的时候，redis会将连接应答处理器跟AE\_READABLE事件关联起来，接着如果一个客户端跟redis发起连接，此时会产生一个AE\_READABLE事件，然后由连接应答处理器来处理跟客户端建立连接，创建客户端对应的socket，同时将这个socket的AE\_READABLE事件跟命令请求处理器关联起来。

当客户端向redis发起请求的时候（不管是读请求还是写请求，都一样），首先就会在socket产生一个AE\_READABLE事件，然后由对应的命令请求处理器来处理。这个命令请求处理器就会从socket中读取请求相关数据，然后进行执行和处理。

接着redis这边准备好了给客户端的响应数据之后，就会将socket的AE\_WRITABLE事件跟命令回复处理器关联起来，当客户端这边准备好读取响应数据时，就会在socket上产生一个AE\_WRITABLE事件，会由对应的命令回复处理器来处理，就是将准备好的响应数据写入socket，供客户端来读取。

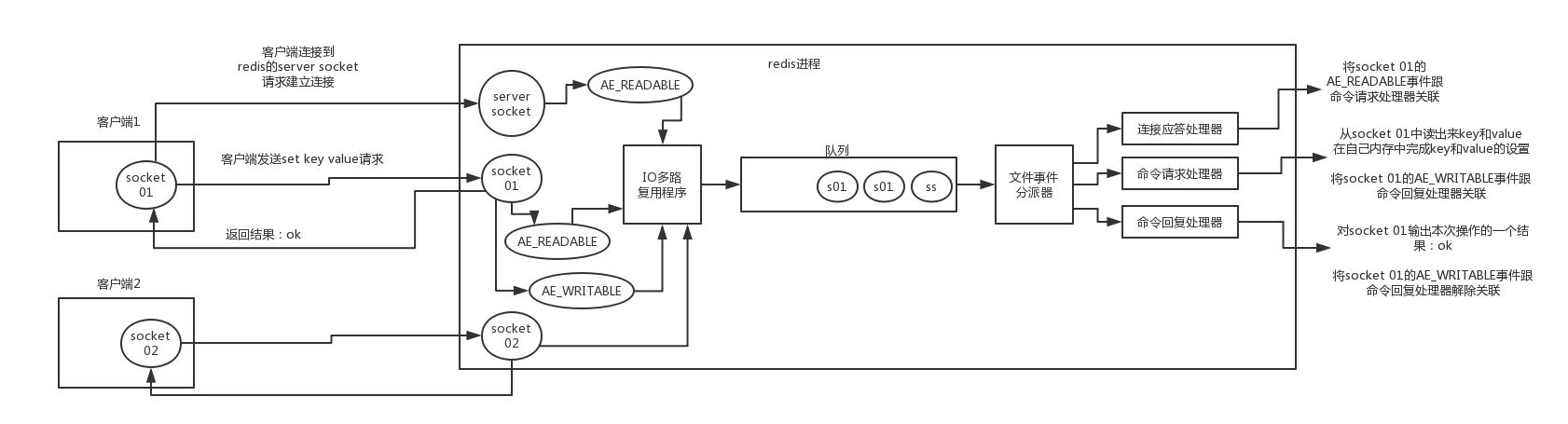
命令回复处理器写完之后，就会删除这个socket的AE\_WRITABLE事件和命令回复处理器的关联关系。

### （3）为啥redis单线程模型也能效率这么高？

1）纯内存操作

2）核心是基于非阻塞的IO多路复用机制

3）单线程反而避免了多线程的频繁上下文切换问题（百度）



# redis的过期策略都有哪些？内存淘汰机制都有哪些？手写一下LRU代码实现？

1）老师啊，我往redis里写的数据怎么没了？

之前有同学问过我，说我们生产环境的redis怎么经常会丢掉一些数据？写进去了，过一会儿可能就没了。我的天，同学，你问这个问题就说明redis你就没用对啊。redis是缓存，你给当存储了是吧？

啥叫缓存？用内存当缓存。内存是无限的吗，内存是很宝贵而且是有限的，磁盘是廉价而且是大量的。可能一台机器就几十个G的内存，但是可以有几个T的硬盘空间。redis主要是基于内存来进行高性能、高并发的读写操作的。

那既然内存是有限的，比如redis就只能用10个G，你要是往里面写了20个G的数据，会咋办？当然会干掉10个G的数据，然后就保留10个G的数据了。那干掉哪些数据？保留哪些数据？当然是干掉不常用的数据，保留常用的数据了。

所以说，这是缓存的一个最基本的概念，数据是会过期的，要么是你自己设置个过期时间，要么是redis自己给干掉。

set key value 过期时间（1小时）

set进去的key，1小时之后就没了，就失效了

2）老师，我的数据明明都过期了，怎么还占用着内存啊？

还有一种就是如果你设置好了一个过期时间，你知道redis是怎么给你弄成过期的吗？什么时候删除掉？如果你不知道，之前有个学员就问了，为啥好多数据明明应该过期了，结果发现redis内存占用还是很高？那是因为你不知道redis是怎么删除那些过期key的。

redis 内存一共是10g，你现在往里面写了5g的数据，结果这些数据明明你都设置了过期时间，要求这些数据1小时之后都会过期，结果1小时之后，你回来一看，redis机器，怎么内存占用还是50%呢？5g数据过期了，我从redis里查，是查不到了，结果过期的数据还占用着redis的内存。

如果你连这个问题都不知道，上来就懵了，回答不出来，那线上你写代码的时候，想当然的认为写进redis的数据就一定会存在，后面导致系统各种漏洞和bug，谁来负责？

|  |
| --- |
| 过期键的删除策略  如果一个键是过期的，那它到了过期时间之后是不是马上就从内存中被被删除呢？？如果不是，那过期后到底什么时候被删除呢？？  其实有三种不同的删除策略： （1）：立即删除。在设置键的过期时间时，创建一个回调事件，当过期时间达到时，由时间处理器自动执行键的删除操作。 （2）：惰性删除。键过期了就过期了，不管。每次从dict字典中按key取值时，先检查此key是否已经过期，如果过期了就删除它，并返回nil，如果没过期，就返回键值。 （3）：定时删除。每隔一段时间，对expires字典进行检查，删除里面的过期键。  可以看到，第二种为被动删除，第一种和第三种为主动删除，且第一种实时性更高。下面对这三种删除策略进行具体分析。  立即删除  立即删除能保证内存中数据的最大新鲜度，因为它保证过期键值会在过期后马上被删除，其所占用的内存也会随之释放。但是**立即删除对cpu是最不友好的**。因为删除操作会占用cpu的时间，如果刚好碰上了cpu很忙的时候，比如正在做交集或排序等计算的时候，就会给cpu造成额外的压力。  而且目前redis事件处理器对时间事件的处理方式--无序链表，查找一个key的时间复杂度为O(n),所以并不适合用来处理大量的时间事件。  惰性删除  惰性删除是指，某个键值过期后，此键值不会马上被删除，而是等到下次被使用的时候，才会被检查到过期，此时才能得到删除。所以惰性删除的缺点很明显:**浪费内存**。dict字典和expires字典都要保存这个键值的信息。  举个例子，对于一些按时间点来更新的数据，比如log日志，过期后在很长的一段时间内可能都得不到访问，这样在这段时间内就要拜拜浪费这么多内存来存log。**这对于性能非常依赖于内存大小的redis来说，是比较致命的**。  定时删除  从上面分析来看，立即删除会短时间内占用大量cpu，惰性删除会在一段时间内浪费内存，所以定时删除是一个折中的办法。 定时删除是：每隔一段时间执行一次删除操作，并通过限制删除操作执行的时长和频率，来减少删除操作对cpu的影响。另一方面定时删除也有效的减少了因惰性删除带来的内存浪费。  redis使用的策略  redis使用的过期键值删除策略是：惰性删除加上定期删除，两者配合使用。 |

### （1）设置过期时间

我们set key的时候，都可以给一个expire time，就是过期时间，指定这个key比如说只能存活1个小时？10分钟？这个很有用，我们自己可以指定缓存到期就失效。

如果假设你设置一个一批key只能存活1个小时，那么接下来1小时后，redis是怎么对这批key进行删除的？

答案是：定期删除+惰性删除

所谓定期删除，指的是redis默认是每隔100ms就随机抽取一些设置了过期时间的key，检查其是否过期，如果过期就删除。假设redis里放了10万个key，都设置了过期时间，你每隔几百毫秒，就检查10万个key，那redis基本上就死了，cpu负载会很高的，消耗在你的检查过期key上了。注意，这里可不是每隔100ms就遍历所有的设置过期时间的key，那样就是一场性能上的灾难。实际上redis是每隔100ms随机抽取一些key来检查和删除的。

但是问题是，定期删除可能会导致很多过期key到了时间并没有被删除掉，那咋整呢？所以就是惰性删除了。这就是说，在你获取某个key的时候，redis会检查一下 ，这个key如果设置了过期时间那么是否过期了？如果过期了此时就会删除，不会给你返回任何东西。

并不是key到时间就被删除掉，而是你查询这个key的时候，redis再懒惰的检查一下

通过上述两种手段结合起来，保证过期的key一定会被干掉。

很简单，就是说，你的过期key，靠定期删除没有被删除掉，还停留在内存里，占用着你的内存呢，除非你的系统去查一下那个key，才会被redis给删除掉。

但是实际上这还是有问题的，如果定期删除漏掉了很多过期key，然后你也没及时去查，也就没走惰性删除，此时会怎么样？如果大量过期key堆积在内存里，导致redis内存块耗尽了，咋整？

答案是：走内存淘汰机制。

### （2）内存淘汰

如果redis的内存占用过多的时候，此时会进行内存淘汰，有如下一些策略：

redis 10个key，现在已经满了，redis需要删除掉5个key

1个key，最近1分钟被查询了100次

1个key，最近10分钟被查询了50次

1个key，最近1个小时倍查询了1次

1）noeviction：当内存不足以容纳新写入数据时，新写入操作会报错，这个一般没人用吧，实在是太恶心了

2）allkeys-lru：当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的key（这个是最常用的）

3）allkeys-random：当内存不足以容纳新写入数据时，在键空间中，随机移除某个key，这个一般没人用吧，为啥要随机，肯定是把最近最少使用的key给干掉啊

4）volatile-lru：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的key（这个一般不太合适）

5）volatile-random：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个key

6）volatile-ttl：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的key优先移除

### （3）要不你手写一个LRU算法？

我确实有时会问这个，因为有些候选人如果确实过五关斩六将，前面的问题都答的很好，那么其实让他写一下LRU算法，可以考察一下编码功底

你可以现场手写最原始的LRU算法，那个代码量太大了，我觉得不太现实

|  |
| --- |
| **public** **class** LRUCache<K, V> **extends** LinkedHashMap<K, V> {    **private** **final** **int** CACHE\_SIZE;  // 这里就是传递进来最多能缓存多少数据  **public** LRUCache(**int** cacheSize) {  **super**((**int**) Math.*ceil*(cacheSize / 0.75) + 1, 0.75f, **true**); // 这块就是设置一个hashmap的初始大小，同时最后一个true指的是让linkedhashmap按照访问顺序来进行排序，最近访问的放在头，最老访问的就在尾  CACHE\_SIZE = cacheSize;  }  @Override  **protected** **boolean** removeOldestEntry(Map.Entry oldest) {  **return** size() > CACHE\_SIZE; // 这个意思就是说当map中的数据量大于指定的缓存个数的时候，就自动删除最老的数据  }  } |

我给你看上面的代码，是告诉你最起码你也得写出来上面那种代码，不求自己纯手工从底层开始打造出自己的LRU，但是起码知道如何利用已有的jdk数据结构实现一个java版的LRU

# 怎么保证redis是高并发以及高可用的？

如何保证Redis的高并发和高可用？

redis的主从复制原理能介绍一下么？

redis的哨兵原理能介绍一下么？

其实问这个问题，主要是考考你，redis单机能承载多高并发？如果单机扛不住如何扩容抗更多的并发？redis会不会挂？既然redis会挂那怎么保证redis是高可用的？

其实针对的都是项目中你肯定要考虑的一些问题，如果你没考虑过，那确实你对生产系统中的问题思考太少。

就是如果你用redis缓存技术的话，肯定要考虑如何用redis来加多台机器，保证redis是高并发的，还有就是如何让Redis保证自己不是挂掉以后就直接死掉了，redis高可用

我这里会选用我之前讲解过这一块内容，redis高并发、高可用、缓存一致性

redis高并发：主从架构，一主多从，一般来说，很多项目其实就足够了，单主用来写入数据，单机几万QPS，多从用来查询数据，多个从实例可以提供每秒10万的QPS。

redis高并发的同时，还需要容纳大量的数据：一主多从，每个实例都容纳了完整的数据，比如redis主就10G的内存量，其实你就最对只能容纳10g的数据量。如果你的缓存要容纳的数据量很大，达到了几十g，甚至几百g，或者是几t，那你就需要redis集群，而且用redis集群之后，可以提供可能每秒几十万的读写并发。

redis高可用：如果你做主从架构部署，其实就是加上哨兵就可以了，就可以实现，任何一个实例宕机，自动会进行主备切换。

### redis如何通过读写分离来承载读请求QPS超过10万+？

1、redis高并发跟整个系统的高并发之间的关系

redis，你要搞高并发的话，不可避免，要把底层的缓存搞得很好

mysql，高并发，做到了，那么也是通过一系列复杂的分库分表，订单系统，事务要求的，QPS到几万，比较高了

要做一些电商的商品详情页，真正的超高并发，QPS上十万，甚至是百万，一秒钟百万的请求量,光是redis是不够的，但是redis是整个大型的缓存架构中，支撑高并发的架构里面，非常重要的一个环节

首先，你的底层的缓存中间件，缓存系统，必须能够支撑的起我们说的那种高并发，其次，再经过良好的整体的缓存架构的设计（多级缓存架构、热点缓存），支撑真正的上十万，甚至上百万的高并发

2、redis不能支撑高并发的瓶颈在哪里？

单机

3、如果redis要支撑超过10万+的并发，那应该怎么做？

单机的redis几乎不太可能说QPS超过10万+，除非一些特殊情况，比如你的机器性能特别好，配置特别高，物理机，维护做的特别好，而且你的整体的操作不是太复杂

单机在几万

读写分离，一般来说，对缓存，一般都是用来支撑读高并发的，写的请求是比较少的，可能写请求也就一秒钟几千，一两千

大量的请求都是读，一秒钟二十万次读

读写分离

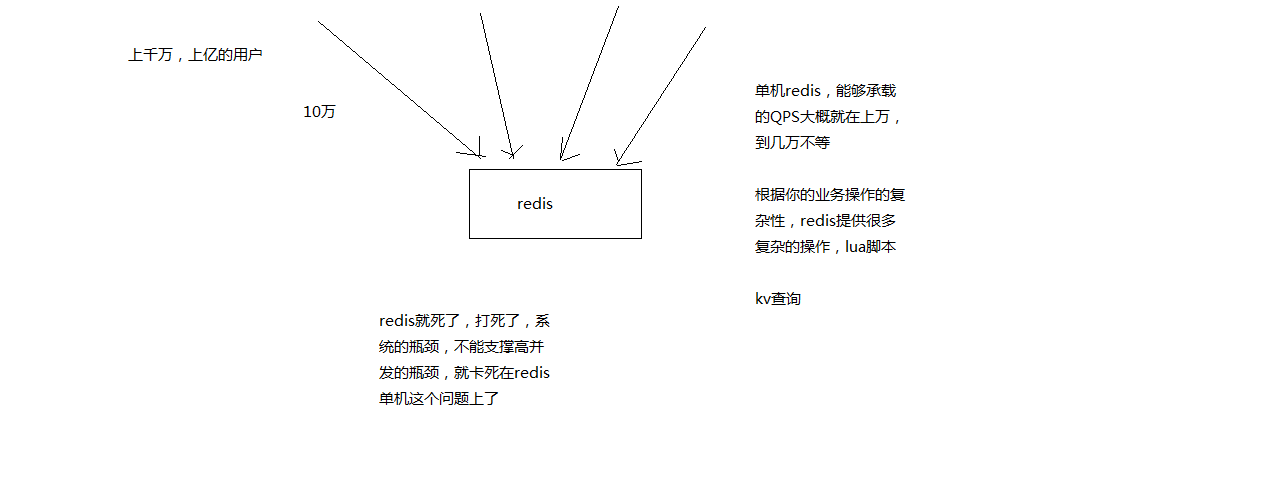
主从架构 -> 读写分离 -> 支撑10万+读QPS的架构

4、接下来要讲解的一个topic

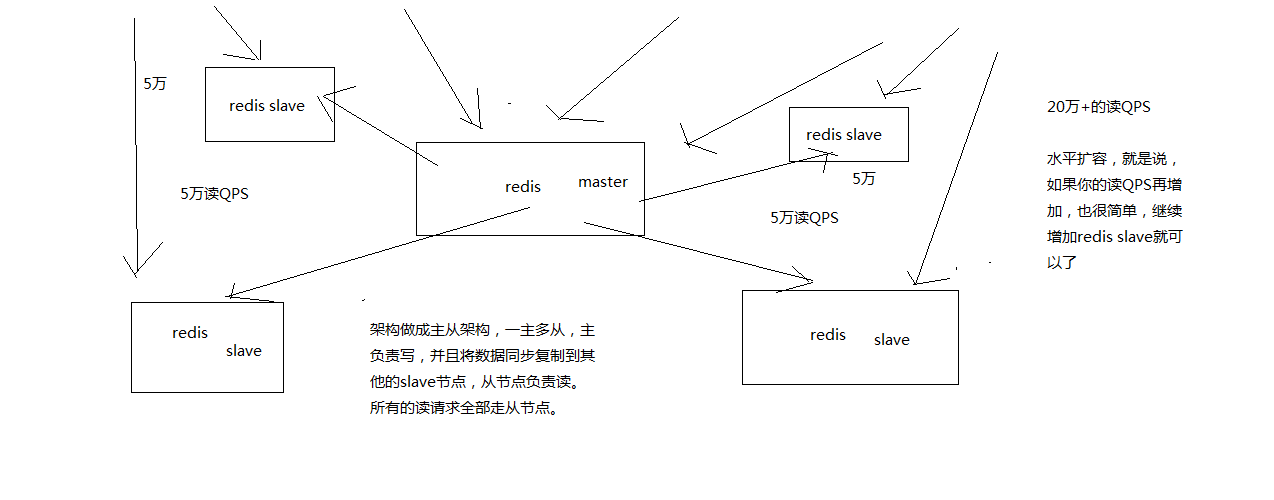
redis replication

redis主从架构 -> 读写分离架构 -> 可支持水平扩展的读高并发架构

redis单机的瓶颈



redis主从实现读写分离支撑10万+的高并发



### redis replication以及master持久化对主从架构的安全意义

课程大纲

1、图解redis replication基本原理

2、redis replication的核心机制

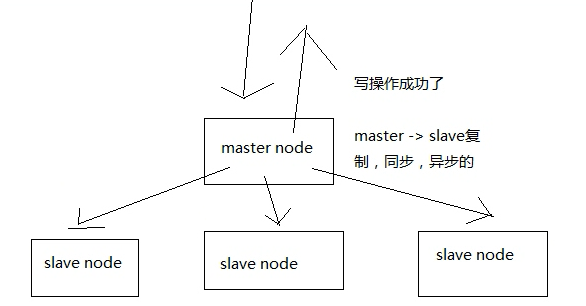
3、master持久化对于主从架构的安全保障的意义

redis replication -> 主从架构 -> 读写分离 -> 水平扩容支撑读高并发

redis replication的最最基本的原理，铺垫

------------------------------------------------------------------------

1、图解redis replication基本原理



------------------------------------------------------------------------

2、redis replication的核心机制

（1）redis采用异步方式复制数据到slave节点，不过redis 2.8开始，slave node会周期性地确认自己每次复制的数据量

（2）一个master node是可以配置多个slave node的

（3）slave node也可以连接其他的slave node

（4）slave node做复制的时候，是不会block master node的正常工作的

（5）slave node在做复制的时候，也不会block对自己的查询操作，它会用旧的数据集来提供服务; 但是复制完成的时候，需要删除旧数据集，加载新数据集，这个时候就会暂停对外服务了

（6）slave node主要用来进行横向扩容，做读写分离，扩容的slave node可以提高读的吞吐量

slave，高可用性，有很大的关系

------------------------------------------------------------------------

3、master持久化对于主从架构的安全保障的意义

如果采用了主从架构，那么建议必须开启master node的持久化！

不建议用slave node作为master node的数据热备，因为那样的话，如果你关掉master的持久化，可能在master宕机重启的时候数据是空的，然后可能一经过复制，salve node数据也丢了

master -> RDB和AOF都关闭了 -> 全部在内存中

master宕机，重启，是没有本地数据可以恢复的，然后就会直接认为自己IDE数据是空的

master就会将空的数据集同步到slave上去，所有slave的数据全部清空

100%的数据丢失

master节点，必须要使用持久化机制

第二个，master的各种备份方案，要不要做，万一说本地的所有文件丢失了; 从备份中挑选一份rdb去恢复master; 这样才能确保master启动的时候，是有数据的

即使采用了后续讲解的高可用机制，slave node可以自动接管master node，但是也可能sentinal还没有检测到master failure，master node就自动重启了，还是可能导致上面的所有slave node数据清空故障

### redis主从复制原理、断点续传、无磁盘化复制、过期key处理

1、主从架构的核心原理

当启动一个slave node的时候，它会发送一个PSYNC命令给master node

如果这是slave node重新连接master node，那么master node仅仅会复制给slave部分缺少的数据; 否则如果是slave node第一次连接master node，那么会触发一次full resynchronization

开始full resynchronization的时候，master会启动一个后台线程，开始生成一份RDB快照文件，同时还会将从客户端收到的所有写命令缓存在内存中。RDB文件生成完毕之后，master会将这个RDB发送给slave，slave会先写入本地磁盘，然后再从本地磁盘加载到内存中。然后master会将内存中缓存的写命令发送给slave，slave也会同步这些数据。

slave node如果跟master node有网络故障，断开了连接，会自动重连。master如果发现有多个slave node都来重新连接，仅仅会启动一个rdb save操作，用一份数据服务所有slave node。

2、主从复制的断点续传

从redis 2.8开始，就支持主从复制的断点续传，如果主从复制过程中，网络连接断掉了，那么可以接着上次复制的地方，继续复制下去，而不是从头开始复制一份

master node会在内存中常见一个backlog，master和slave都会保存一个replica offset还有一个master id，offset就是保存在backlog中的。如果master和slave网络连接断掉了，slave会让master从上次的replica offset开始继续复制

但是如果没有找到对应的offset，那么就会执行一次resynchronization

3、无磁盘化复制

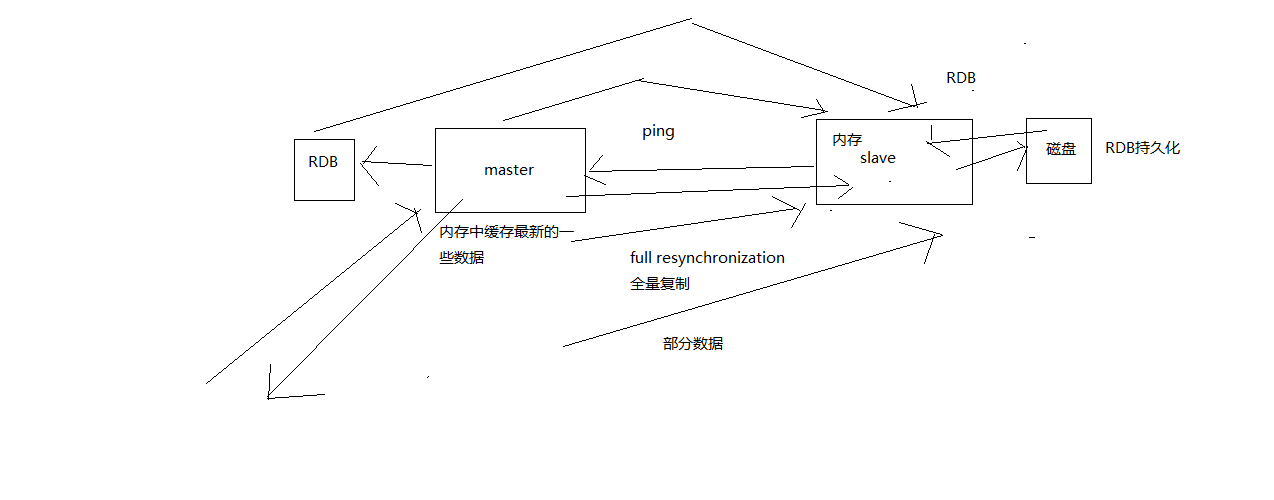
master在内存中直接创建rdb，然后发送给slave，不会在自己本地落地磁盘了

repl-diskless-sync

repl-diskless-sync-delay，等待一定时长再开始复制，因为要等更多slave重新连接过来

4、过期key处理

slave不会过期key，只会等待master过期key。如果master过期了一个key，或者通过LRU淘汰了一个key，那么会模拟一条del命令发送给slave。



### redis replication的完整流运行程和原理的再次深入剖析

1、复制的完整流程

（1）slave node启动，仅仅保存master node的信息，包括master node的host和ip，但是复制流程没开始master host和ip是从哪儿来的，redis.conf里面的slaveof配置的

（2）slave node内部有个定时任务，每秒检查是否有新的master node要连接和复制，如果发现，就跟master node建立socket网络连接

（3）slave node发送ping命令给master node

（4）口令认证，如果master设置了requirepass，那么salve node必须发送masterauth的口令过去进行认证

（5）master node第一次执行全量复制，将所有数据发给slave node

（6）master node后续持续将写命令，异步复制给slave node

2、数据同步相关的核心机制

指的就是第一次slave连接master的时候，执行的全量复制，那个过程里面你的一些细节的机制

（1）master和slave都会维护一个offset

master会在自身不断累加offset，slave也会在自身不断累加offset

slave每秒都会上报自己的offset给master，同时master也会保存每个slave的offset

这个倒不是说特定就用在全量复制的，主要是master和slave都要知道各自的数据的offset，才能知道互相之间的数据不一致的情况

（2）backlog

master node有一个backlog，默认是1MB大小

master node给slave node复制数据时，也会将数据在backlog中同步写一份

backlog主要是用来做全量复制中断候的增量复制的

（3）master run id

info server，可以看到master run id

如果根据host+ip定位master node，是不靠谱的，如果master node重启或者数据出现了变化，那么slave node应该根据不同的run id区分，run id不同就做全量复制

如果需要不更改run id重启redis，可以使用redis-cli debug reload命令

（4）psync

从节点使用psync从master node进行复制，psync runid offset

master node会根据自身的情况返回响应信息，可能是FULLRESYNC runid offset触发全量复制，可能是CONTINUE触发增量复制

3、全量复制

（1）master执行bgsave，在本地生成一份rdb快照文件

（2）master node将rdb快照文件发送给salve node，如果rdb复制时间超过60秒（repl-timeout），那么slave node就会认为复制失败，可以适当调节大这个参数

（3）对于千兆网卡的机器，一般每秒传输100MB，6G文件，很可能超过60s

（4）master node在生成rdb时，会将所有新的写命令缓存在内存中，在salve node保存了rdb之后，再将新的写命令复制给salve node

（5）client-output-buffer-limit slave 256MB 64MB 60，如果在复制期间，内存缓冲区持续消耗超过64MB，或者一次性超过256MB，那么停止复制，复制失败

（6）slave node接收到rdb之后，清空自己的旧数据，然后重新加载rdb到自己的内存中，同时基于旧的数据版本对外提供服务

（7）如果slave node开启了AOF，那么会立即执行BGREWRITEAOF，重写AOF

rdb生成、rdb通过网络拷贝、slave旧数据的清理、slave aof rewrite，很耗费时间

如果复制的数据量在4G~6G之间，那么很可能全量复制时间消耗到1分半到2分钟

4、增量复制

（1）如果全量复制过程中，master-slave网络连接断掉，那么salve重新连接master时，会触发增量复制

（2）master直接从自己的backlog中获取部分丢失的数据，发送给slave node，默认backlog就是1MB

（3）msater就是根据slave发送的psync中的offset来从backlog中获取数据的

5、heartbeat

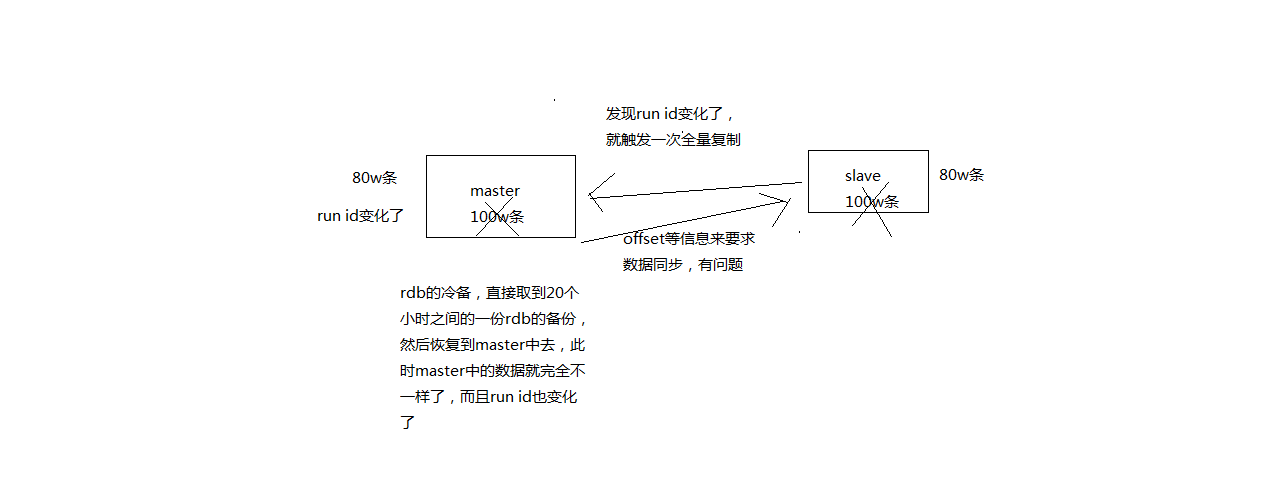
主从节点互相都会发送heartbeat信息

master默认每隔10秒发送一次heartbeat，salve node每隔1秒发送一个heartbeat

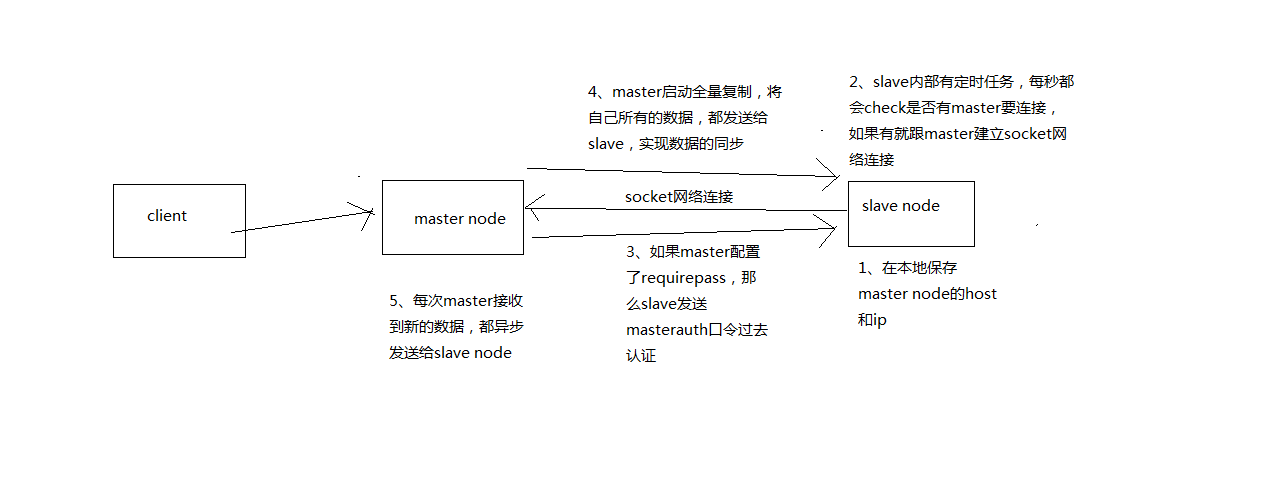
6、异步复制

master每次接收到写命令之后，先在内部写入数据，然后异步发送给slave node

master run id的作用



复制的完整的基本流程



### redis主从架构下如何才能做到99.99%的高可用性？

1、什么是99.99%高可用？

架构上，高可用性，99.99%的高可用性

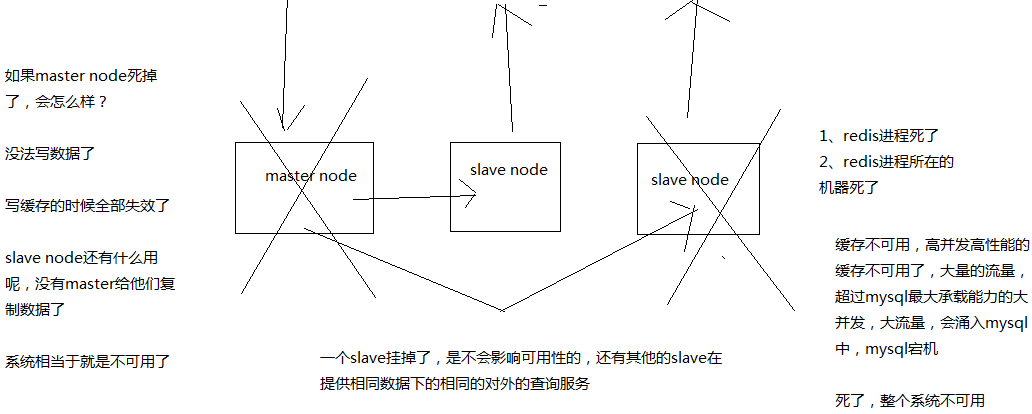
讲的学术，99.99%，公式，系统可用的时间 / 系统故障的时间，365天，在365天 \* 99.99%的时间内，你的系统都是可以哗哗对外提供服务的，那就是高可用性，99.99%

系统可用的时间 / 总的时间 = 高可用性，然后会对各种时间的概念，说一大堆解释

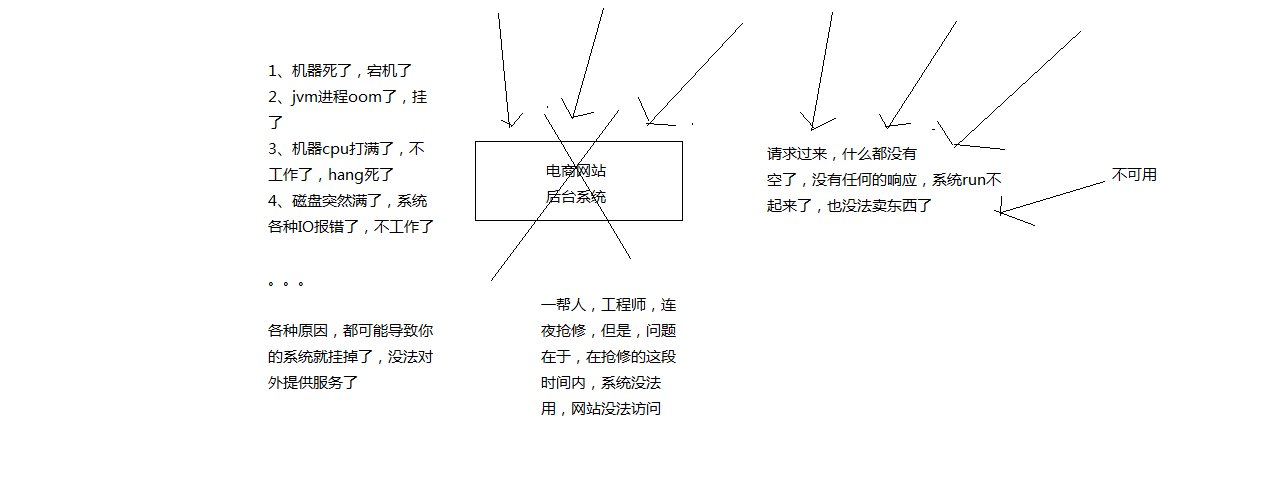
2、redis不可用是什么？单实例不可用？主从架构不可用？不可用的后果是什么？

3、redis怎么才能做到高可用？

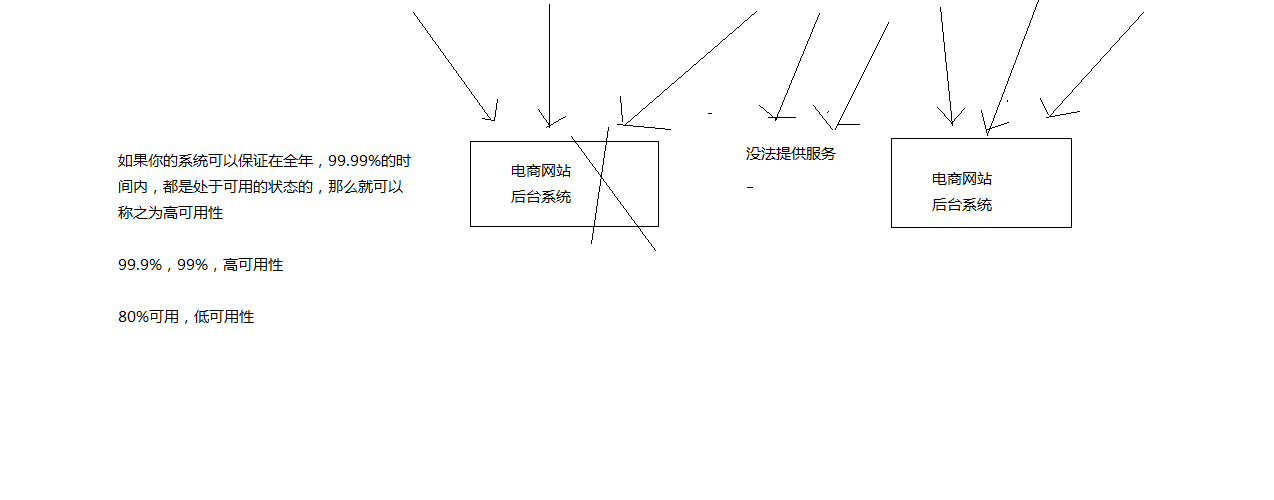
redis的不可用



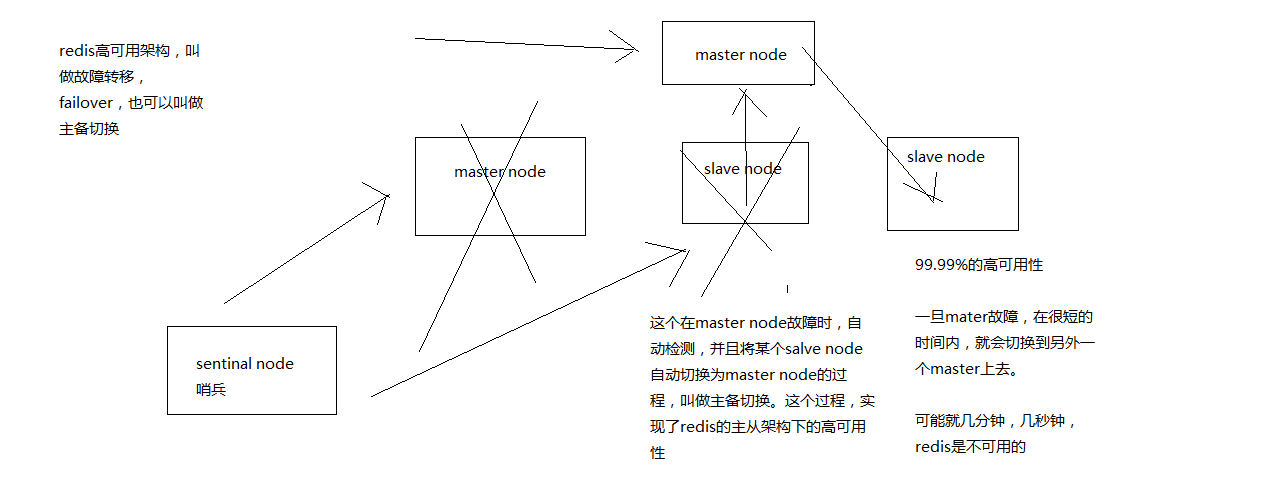
系统处于不可用是什么意思



什么是99.99%高可用性



redis基于哨兵的高可用性



### redis哨兵架构的相关基础知识的讲解

1、哨兵的介绍

sentinal，中文名是哨兵

哨兵是redis集群架构中非常重要的一个组件，主要功能如下

（1）集群监控，负责监控redis master和slave进程是否正常工作

（2）消息通知，如果某个redis实例有故障，那么哨兵负责发送消息作为报警通知给管理员

（3）故障转移，如果master node挂掉了，会自动转移到slave node上

（4）配置中心，如果故障转移发生了，通知client客户端新的master地址

哨兵本身也是分布式的，作为一个哨兵集群去运行，互相协同工作

（1）故障转移时，判断一个master node是宕机了，需要大部分的哨兵都同意才行，涉及到了分布式选举的问题

（2）即使部分哨兵节点挂掉了，哨兵集群还是能正常工作的，因为如果一个作为高可用机制重要组成部分的故障转移系统本身是单点的，那就很坑爹了

目前采用的是sentinal 2版本，sentinal 2相对于sentinal 1来说，重写了很多代码，主要是让故障转移的机制和算法变得更加健壮和简单

2、哨兵的核心知识

（1）哨兵至少需要3个实例，来保证自己的健壮性

（2）哨兵 + redis主从的部署架构，是不会保证数据零丢失的，只能保证redis集群的高可用性

（3）对于哨兵 + redis主从这种复杂的部署架构，尽量在测试环境和生产环境，都进行充足的测试和演练

3、为什么redis哨兵集群只有2个节点无法正常工作？

哨兵集群必须部署2个以上节点

如果哨兵集群仅仅部署了个2个哨兵实例，quorum=1

+----+ +----+

| M1 |--------| R1 |

| S1 | | S2 |

+----+ +----+

Configuration: quorum = 1

master宕机，s1和s2中只要有1个哨兵认为master宕机就可以还行切换，同时s1和s2中会选举出一个哨兵来执行故障转移

同时这个时候，需要majority(票数)，也就是大多数哨兵都是运行的，2个哨兵的majority票数就是2（2的majority=2，3的majority=2，5的majority=3，4的majority=2），2个哨兵都运行着，就可以允许执行故障转移

但是如果整个M1和S1运行的机器宕机了，那么哨兵只有1个了，此时就没有majority来允许执行故障转移，虽然另外一台机器还有一个R1，但是故障转移不会执行

4、经典的3节点哨兵集群

+----+

| M1 |

| S1 |

+----+

|

+----+ | +----+

| R2 |----+---| R3 |

| S2 | | S3 |

+----+ +----+

Configuration: quorum = 2，majority

如果M1所在机器宕机了，那么三个哨兵还剩下2个，S2和S3可以一致认为master宕机，然后选举出一个来执行故障转移

同时3个哨兵的majority是2，所以还剩下的2个哨兵运行着，就可以允许执行故障转移

### redis哨兵主备切换的数据丢失问题：异步复制、集群脑裂

课程大纲

1、两种数据丢失的情况

2、解决异步复制和脑裂导致的数据丢失

------------------------------------------------------------------

1、两种数据丢失的情况

主备切换的过程，可能会导致数据丢失

（1）异步复制导致的数据丢失

因为master -> slave的复制是异步的，所以可能有部分数据还没复制到slave，master就宕机了，此时这些部分数据就丢失了

（2）脑裂导致的数据丢失

脑裂，也就是说，某个master所在机器突然脱离了正常的网络，跟其他slave机器不能连接，但是实际上master还运行着

此时哨兵可能就会认为master宕机了，然后开启选举，将其他slave切换成了master

这个时候，集群里就会有两个master，也就是所谓的脑裂

此时虽然某个slave被切换成了master，但是可能client还没来得及切换到新的master，还继续写向旧master的数据可能也丢失了

因此旧master再次恢复的时候，会被作为一个slave挂到新的master上去，自己的数据会清空，重新从新的master复制数据

------------------------------------------------------------------

2、解决异步复制和脑裂导致的数据丢失

min-slaves-to-write 1

min-slaves-max-lag 10

要求至少有1个slave，数据复制和同步的延迟不能超过10秒

如果说一旦所有的slave，数据复制和同步的延迟都超过了10秒钟，那么这个时候，master就不会再接收任何请求了

上面两个配置可以减少异步复制和脑裂导致的数据丢失

（1）减少异步复制的数据丢失

有了min-slaves-max-lag这个配置，就可以确保说，一旦slave复制数据和ack延时太长，就认为可能master宕机后损失的数据太多了，那么就拒绝写请求，这样可以把master宕机时由于部分数据未同步到slave导致的数据丢失降低的可控范围内

（2）减少脑裂的数据丢失

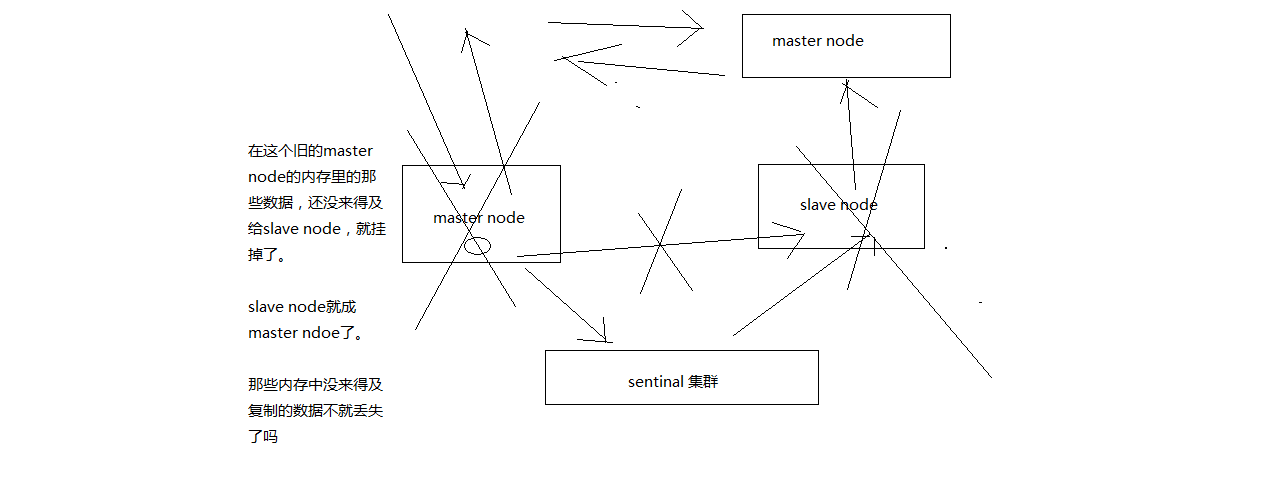
如果一个master出现了脑裂，跟其他slave丢了连接，那么上面两个配置可以确保说，如果不能继续给指定数量的slave发送数据，而且slave超过10秒没有给自己ack消息，那么就直接拒绝客户端的写请求

这样脑裂后的旧master就不会接受client的新数据，也就避免了数据丢失

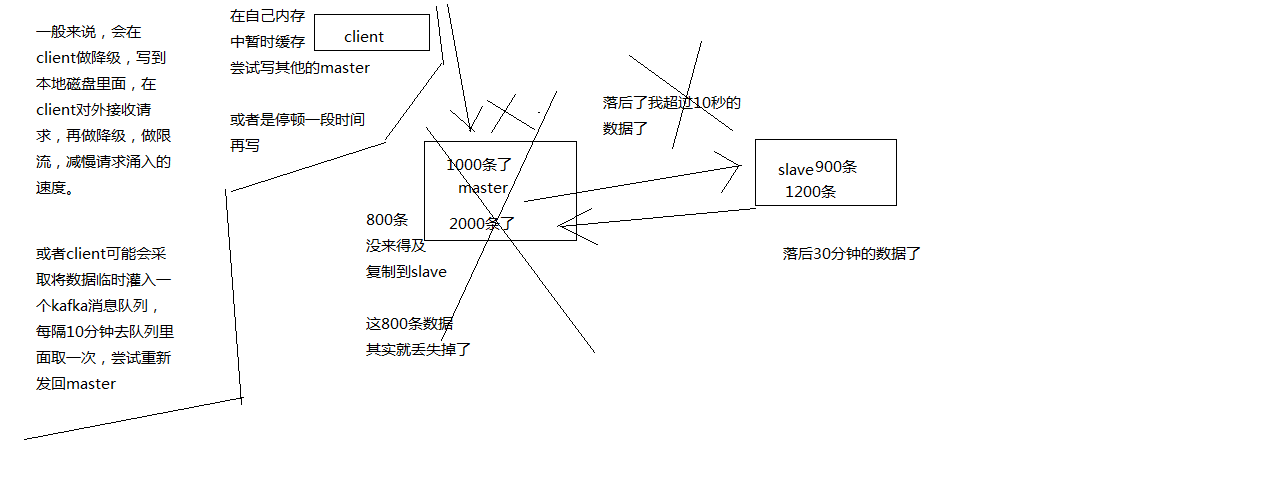
上面的配置就确保了，如果跟任何一个slave丢了连接，在10秒后发现没有slave给自己ack，那么就拒绝新的写请求

因此在脑裂场景下，最多就丢失10秒的数据

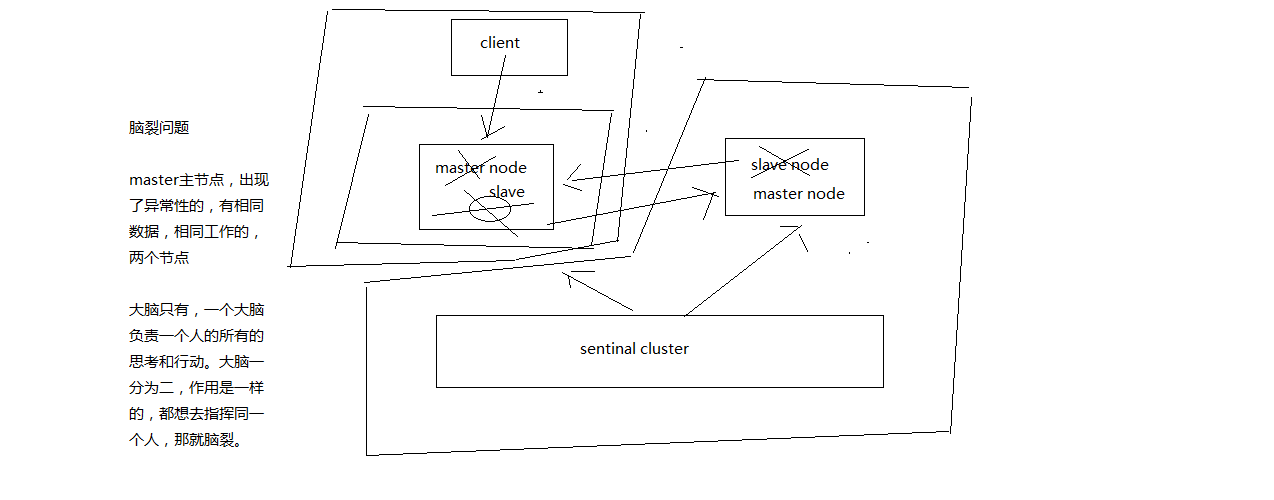
异步复制导致的数据丢失问题



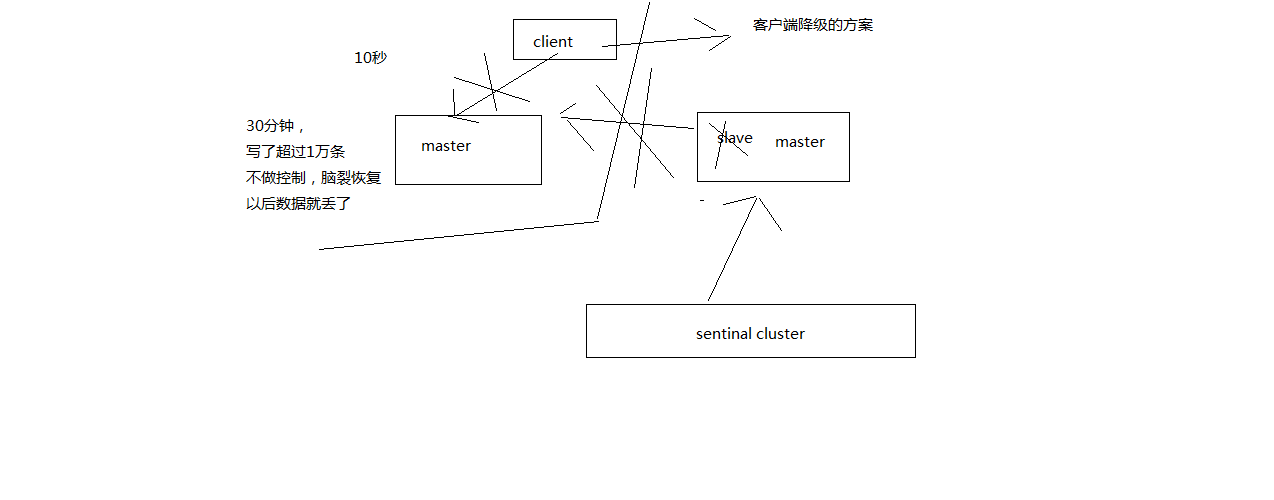
异步复制导致数据丢失如何降低损失



集群脑裂导致的数据丢失问题



脑裂导致数据丢失的问题如何降低损失



### redis哨兵的多个核心底层原理的深入解析（包含slave选举算法）

1. sdown和odown转换机制

sdown和odown两种失败状态

sdown是主观宕机，就一个哨兵如果自己觉得一个master宕机了，那么就是主观宕机

odown是客观宕机，如果quorum数量的哨兵都觉得一个master宕机了，那么就是客观宕机

sdown达成的条件很简单，如果一个哨兵ping一个master，超过了is-master-down-after-milliseconds指定的毫秒数之后，就主观认为master宕机

sdown到odown转换的条件很简单，如果一个哨兵在指定时间内，收到了quorum指定数量的其他哨兵也认为那个master是sdown了，那么就认为是odown了，客观认为master宕机

2、哨兵集群的自动发现机制

哨兵互相之间的发现，是通过redis的pub/sub系统实现的，每个哨兵都会往\_\_sentinel\_\_:hello这个channel里发送一个消息，这时候所有其他哨兵都可以消费到这个消息，并感知到其他的哨兵的存在

每隔两秒钟，每个哨兵都会往自己监控的某个master+slaves对应的\_\_sentinel\_\_:hello channel里发送一个消息，内容是自己的host、ip和runid还有对这个master的监控配置

每个哨兵也会去监听自己监控的每个master+slaves对应的\_\_sentinel\_\_:hello channel，然后去感知到同样在监听这个master+slaves的其他哨兵的存在

每个哨兵还会跟其他哨兵交换对master的监控配置，互相进行监控配置的同步

3、slave配置的自动纠正

哨兵会负责自动纠正slave的一些配置，比如slave如果要成为潜在的master候选人，哨兵会确保slave在复制现有master的数据; 如果slave连接到了一个错误的master上，比如故障转移之后，那么哨兵会确保它们连接到正确的master上

4、slave->master选举算法

如果一个master被认为odown了，而且majority哨兵都允许了主备切换，那么某个哨兵就会执行主备切换操作，此时首先要选举一个slave来

会考虑slave的一些信息

（1）跟master断开连接的时长

（2）slave优先级

（3）复制offset

（4）run id

如果一个slave跟master断开连接已经超过了down-after-milliseconds的10倍，外加master宕机的时长，那么slave就被认为不适合选举为master

(down-after-milliseconds \* 10) + milliseconds\_since\_master\_is\_in\_SDOWN\_state

接下来会对slave进行排序

（1）按照slave优先级进行排序，slave priority越低，优先级就越高

（2）如果slave priority相同，那么看replica offset，哪个slave复制了越多的数据，offset越靠后，优先级就越高

（3）如果上面两个条件都相同，那么选择一个run id比较小的那个slave

5、quorum和majority

每次一个哨兵要做主备切换，首先需要quorum数量的哨兵认为odown，然后选举出一个哨兵来做切换，这个哨兵还得得到majority哨兵的授权，才能正式执行切换

如果quorum < majority，比如5个哨兵，majority就是3，quorum设置为2，那么就3个哨兵授权就可以执行切换

但是如果quorum >= majority，那么必须quorum数量的哨兵都授权，比如5个哨兵，quorum是5，那么必须5个哨兵都同意授权，才能执行切换

6、configuration epoch

哨兵会对一套redis master+slave进行监控，有相应的监控的配置

执行切换的那个哨兵，会从要切换到的新master（salve->master）那里得到一个configuration epoch，这就是一个version号，每次切换的version号都必须是唯一的

如果第一个选举出的哨兵切换失败了，那么其他哨兵，会等待failover-timeout时间，然后接替继续执行切换，此时会重新获取一个新的configuration epoch，作为新的version号

7、configuraiton传播

哨兵完成切换之后，会在自己本地更新生成最新的master配置，然后同步给其他的哨兵，就是通过之前说的pub/sub消息机制

这里之前的version号就很重要了，因为各种消息都是通过一个channel去发布和监听的，所以一个哨兵完成一次新的切换之后，新的master配置是跟着新的version号的

其他的哨兵都是根据版本号的大小来更新自己的master配置的

# 怎么保证redis挂掉之后再重启数据可以进行恢复？

redis的持久化有哪几种方式？

不同的持久化机制都有什么优缺点？

持久化机制具体底层是如何实现的？

redis如果仅仅只是将数据缓存在内存里面，如果redis宕机了，再重启，内存里的数据就全部都弄丢了啊。。。。。。你必须得用redis的持久化机制，将数据写入内存的同时，异步的慢慢的将数据写入磁盘文件里，进行持久化

如果redis宕机了，重启启动，自动从磁盘上加载之前持久化的一些数据，就可以了，也许会丢失少许数据，但是至少不会将所有数据都弄丢

这个其实一样，针对的都是redis的生产环境可能遇到的一些问题，就是redis要是挂了再重启，内存里的数据不就全丢了？能不能重启的时候把数据给恢复了？

### redis持久化机对于生产环境中的灾难恢复的意义

课程大纲

1、故障发生的时候会怎么样

2、如何应对故障的发生

很多同学，自己也看过一些redis的资料和书籍，当然可能也看过一些redis视频课程

所有的资料，其实都会讲解redis持久化，但是有个问题，我到目前为止，没有看到有人很仔细的去讲解，redis的持久化意义

redis的持久化，RDB，AOF，区别，各自的特点是什么，适合什么场景

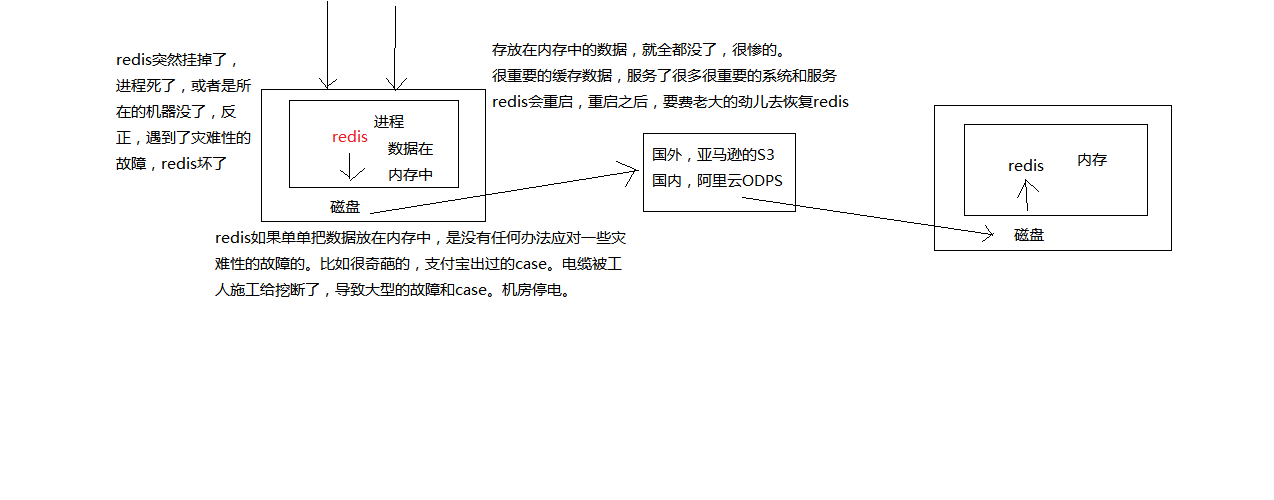
redis的企业级的持久化方案是什么，是用来跟哪些企业级的场景结合起来使用的？？？

redis持久化的意义，在于故障恢复

比如你部署了一个redis，作为cache缓存，当然也可以保存一些较为重要的数据

如果没有持久化的话，redis遇到灾难性故障的时候，就会丢失所有的数据

如果通过持久化将数据搞一份儿在磁盘上去，然后定期比如说同步和备份到一些云存储服务上去，那么就可以保证数据不丢失全部，还是可以恢复一部分数据回来的



### 图解分析redis的RDB和AOF两种持久化机制的工作原理

课程大纲

1、RDB和AOF两种持久化机制的介绍

2、RDB持久化机制的优点

3、RDB持久化机制的缺点

4、AOF持久化机制的优点

5、AOF持久化机制的缺点

6、RDB和AOF到底该如何选择

我们已经知道对于一个企业级的redis架构来说，持久化是不可减少的

企业级redis集群架构：海量数据、高并发、高可用

持久化主要是做灾难恢复，数据恢复，也可以归类到高可用的一个环节里面去

比如你redis整个挂了，然后redis就不可用了，你要做的事情是让redis变得可用，尽快变得可用

重启redis，尽快让它对外提供服务，但是就像上一讲说，如果你没做数据备份，这个时候redis启动了，也不可用啊，数据都没了

很可能说，大量的请求过来，缓存全部无法命中，在redis里根本找不到数据，这个时候就死定了，缓存雪崩问题，所有请求，没有在redis命中，就会去mysql数据库这种数据源头中去找，一下子mysql承接高并发，然后就挂了

mysql挂掉，你都没法去找数据恢复到redis里面去，redis的数据从哪儿来？从mysql来。。。

具体的完整的缓存雪崩的场景，还有企业级的解决方案，到后面讲

如果你把redis的持久化做好，备份和恢复方案做到企业级的程度，那么即使你的redis故障了，也可以通过备份数据，快速恢复，一旦恢复立即对外提供服务

redis的持久化，跟高可用，是有关系的，企业级redis架构中去讲解

redis持久化：RDB，AOF

-------------------------------------------------------------------------------------

1、RDB和AOF两种持久化机制的介绍

RDB持久化机制，对redis中的数据执行周期性的持久化

AOF机制对每条写入命令作为日志，以append-only的模式写入一个日志文件中，在redis重启的时候，可以通过回放AOF日志中的写入指令来重新构建整个数据集

如果我们想要redis仅仅作为纯内存的缓存来用，那么可以禁止RDB和AOF所有的持久化机制

通过RDB或AOF，都可以将redis内存中的数据给持久化到磁盘上面来，然后可以将这些数据备份到别的地方去，比如说阿里云，云服务

如果redis挂了，服务器上的内存和磁盘上的数据都丢了，可以从云服务上拷贝回来之前的数据，放到指定的目录中，然后重新启动redis，redis就会自动根据持久化数据文件中的数据，去恢复内存中的数据，继续对外提供服务

如果同时使用RDB和AOF两种持久化机制，那么在redis重启的时候，会使用AOF来重新构建数据，因为AOF中的数据更加完整

-------------------------------------------------------------------------------------

2、RDB持久化机制的优点

（1）RDB会生成多个数据文件，每个数据文件都代表了某一个时刻中redis的数据，这种多个数据文件的方式，非常适合做冷备，可以将这种完整的数据文件发送到一些远程的安全存储上去，比如说Amazon的S3云服务上去，在国内可以是阿里云的ODPS分布式存储上，以预定好的备份策略来定期备份redis中的数据

（2）RDB对redis对外提供的读写服务，影响非常小，可以让redis保持高性能，因为redis主进程只需要fork一个子进程，让子进程执行磁盘IO操作来进行RDB持久化即可

（3）相对于AOF持久化机制来说，直接基于RDB数据文件来重启和恢复redis进程，更加快速

-------------------------------------------------------------------------------------

3、RDB持久化机制的缺点

（1）如果想要在redis故障时，尽可能少的丢失数据，那么RDB没有AOF好。一般来说，RDB数据快照文件，都是每隔5分钟，或者更长时间生成一次，这个时候就得接受一旦redis进程宕机，那么会丢失最近5分钟的数据

（2）RDB每次在fork子进程来执行RDB快照数据文件生成的时候，如果数据文件特别大，可能会导致对客户端提供的服务暂停数毫秒，或者甚至数秒

-------------------------------------------------------------------------------------

4、AOF持久化机制的优点

（1）AOF可以更好的保护数据不丢失，一般AOF会每隔1秒，通过一个后台线程执行一次fsync操作，最多丢失1秒钟的数据

（2）AOF日志文件以append-only模式写入，所以没有任何磁盘寻址的开销，写入性能非常高，而且文件不容易破损，即使文件尾部破损，也很容易修复

（3）AOF日志文件即使过大的时候，出现后台重写操作，也不会影响客户端的读写。因为在rewrite log的时候，会对其中的指导进行压缩，创建出一份需要恢复数据的最小日志出来。再创建新日志文件的时候，老的日志文件还是照常写入。当新的merge后的日志文件ready的时候，再交换新老日志文件即可。

（4）AOF日志文件的命令通过非常可读的方式进行记录，这个特性非常适合做灾难性的误删除的紧急恢复。比如某人不小心用flushall命令清空了所有数据，只要这个时候后台rewrite还没有发生，那么就可以立即拷贝AOF文件，将最后一条flushall命令给删了，然后再将该AOF文件放回去，就可以通过恢复机制，自动恢复所有数据

-------------------------------------------------------------------------------------

5、AOF持久化机制的缺点

（1）对于同一份数据来说，AOF日志文件通常比RDB数据快照文件更大

（2）AOF开启后，支持的写QPS会比RDB支持的写QPS低，因为AOF一般会配置成每秒fsync一次日志文件，当然，每秒一次fsync，性能也还是很高的

（3）以前AOF发生过bug，就是通过AOF记录的日志，进行数据恢复的时候，没有恢复一模一样的数据出来。所以说，类似AOF这种较为复杂的基于命令日志/merge/回放的方式，比基于RDB每次持久化一份完整的数据快照文件的方式，更加脆弱一些，容易有bug。不过AOF就是为了避免rewrite过程导致的bug，因此每次rewrite并不是基于旧的指令日志进行merge的，而是基于当时内存中的数据进行指令的重新构建，这样健壮性会好很多。

-------------------------------------------------------------------------------------

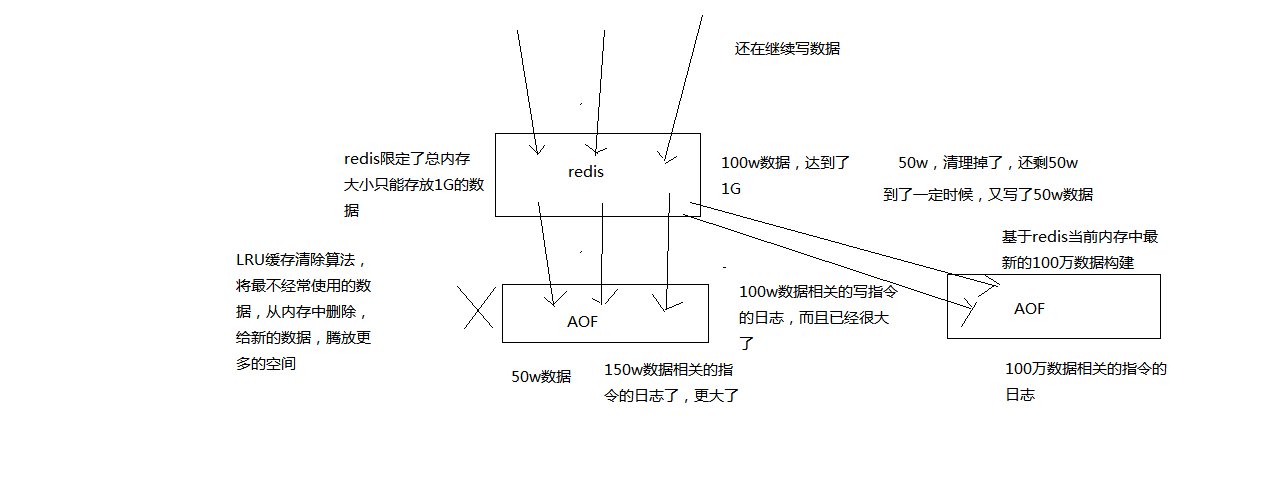
6、RDB和AOF到底该如何选择

（1）不要仅仅使用RDB，因为那样会导致你丢失很多数据

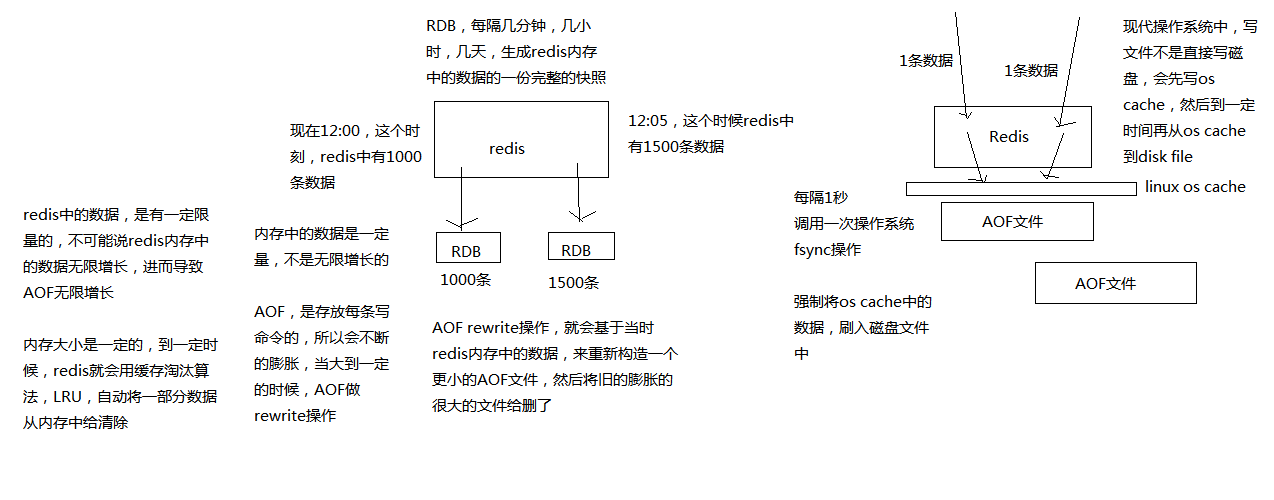
（2）也不要仅仅使用AOF，因为那样有两个问题，第一，你通过AOF做冷备，没有RDB做冷备，来的恢复速度更快; 第二，RDB每次简单粗暴生成数据快照，更加健壮，可以避免AOF这种复杂的备份和恢复机制的bug

（3）综合使用AOF和RDB两种持久化机制，用AOF来保证数据不丢失，作为数据恢复的第一选择; 用RDB来做不同程度的冷备，在AOF文件都丢失或损坏不可用的时候，还可以使用RDB来进行快速的数据恢复

AOF rewrite原理剖析

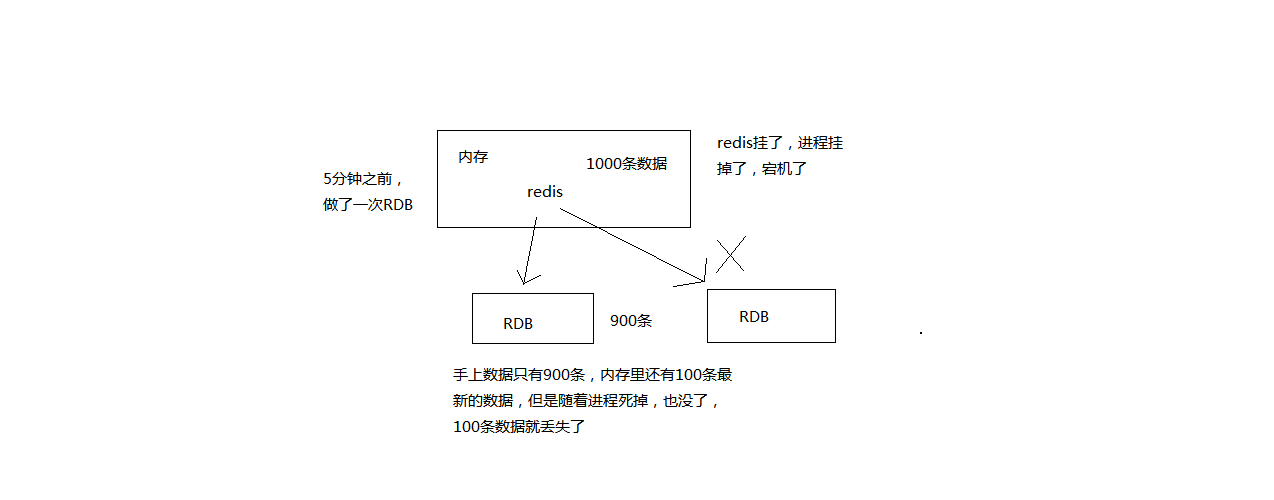


RDB和AOF的介绍



### redis的RDB和AOF两种持久化机制的优劣势对比

RDB丢失数据的问题



# 你能聊聊redis cluster集群模式的原理吗？

redis集群模式的工作原理能说一下么？

在集群模式下，redis的key是如何寻址的？

分布式寻址都有哪些算法？

了解一致性hash算法吗？

在以前，如果前几年的时候，一般来说，redis如果要搞几个节点，每个节点存储一部分的数据，得借助一些中间件来实现，比如说有codis，或者twemproxy，都有。有一些redis中间件，你读写redis中间件，redis中间件负责将你的数据分布式存储在多台机器上的redis实例中。

这两年，redis不断在发展，redis也不断的有新的版本，redis cluster，redis集群模式，你可以做到在多台机器上，部署多个redis实例，每个实例存储一部分的数据，同时每个redis实例可以挂redis从实例，自动确保说，如果redis主实例挂了，会自动切换到redis从实例顶上来。

现在redis的新版本，大家都是用redis cluster的，也就是redis原生支持的redis集群模式，那么面试官肯定会就redis cluster对你来个几连炮。要是你没用过redis cluster，正常，以前很多人用codis之类的客户端来支持集群，但是起码你得研究一下redis cluster吧。

### redis如何在保持读写分离+高可用的架构下，还能横向扩容支撑1T+海量数据

1、单机redis在海量数据面前的瓶颈

2、怎么才能够突破单机瓶颈，让redis支撑海量数据？

3、redis的集群架构

redis cluster

支撑N个redis master node，每个master node都可以挂载多个slave node

读写分离的架构，对于每个master来说，写就写到master，然后读就从mater对应的slave去读

高可用，因为每个master都有salve节点，那么如果mater挂掉，redis cluster这套机制，就会自动将某个slave切换成master

redis cluster（多master + 读写分离 + 高可用）

我们只要基于redis cluster去搭建redis集群即可，不需要手工去搭建replication复制+主从架构+读写分离+哨兵集群+高可用

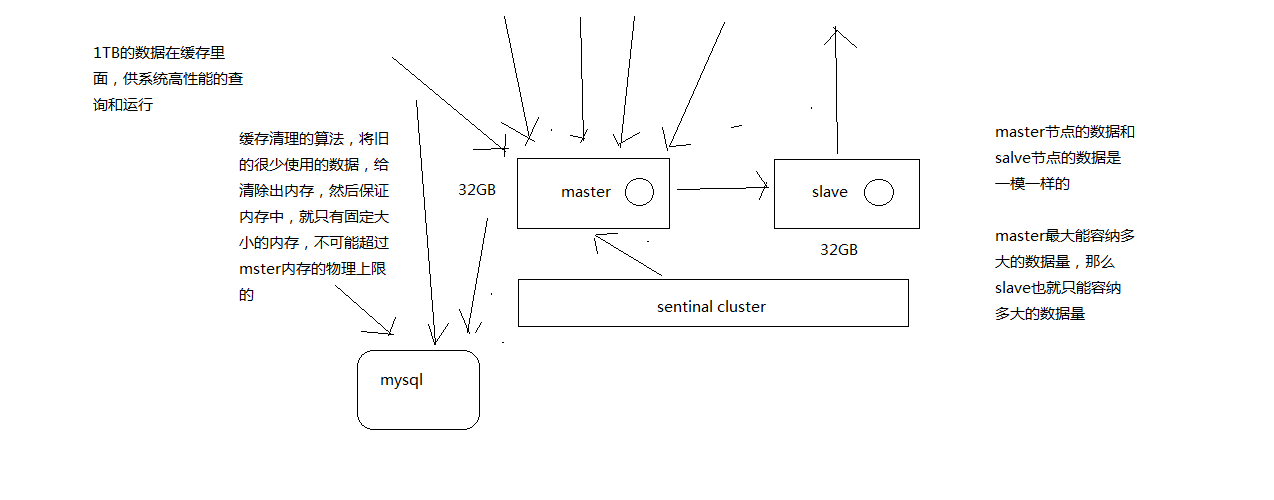
4、redis cluster vs. replication + sentinal

如果你的数据量很少，主要是承载高并发高性能的场景，比如你的缓存一般就几个G，单机足够了

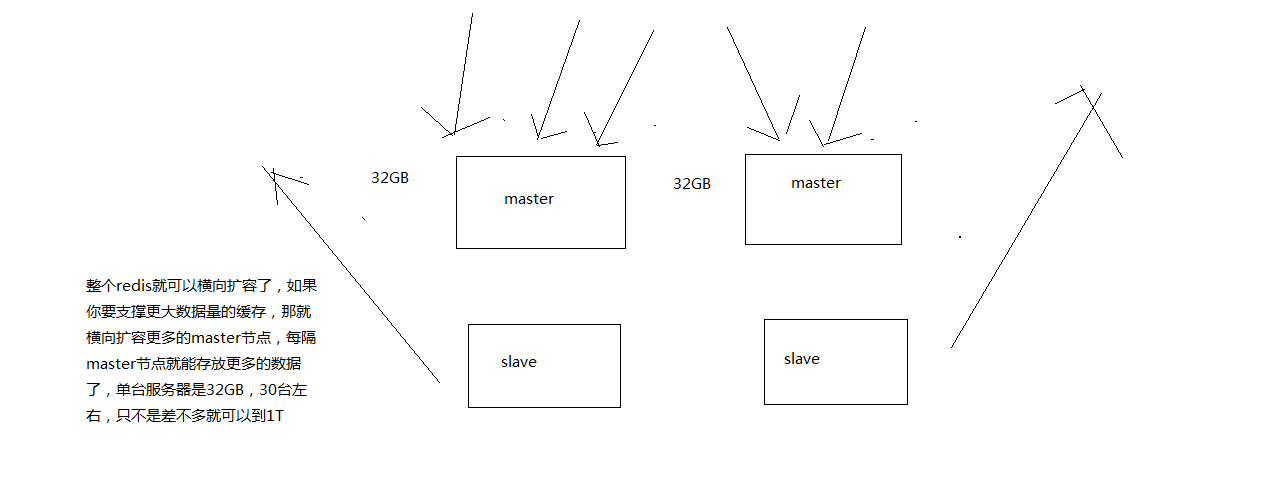
replication，一个mater，多个slave，要几个slave跟你的要求的读吞吐量有关系，然后自己搭建一个sentinal集群，去保证redis主从架构的高可用性，就可以了

redis cluster，主要是针对海量数据+高并发+高可用的场景，海量数据，如果你的数据量很大，那么建议就用redis cluster

redis单master架构的容量的瓶颈问题

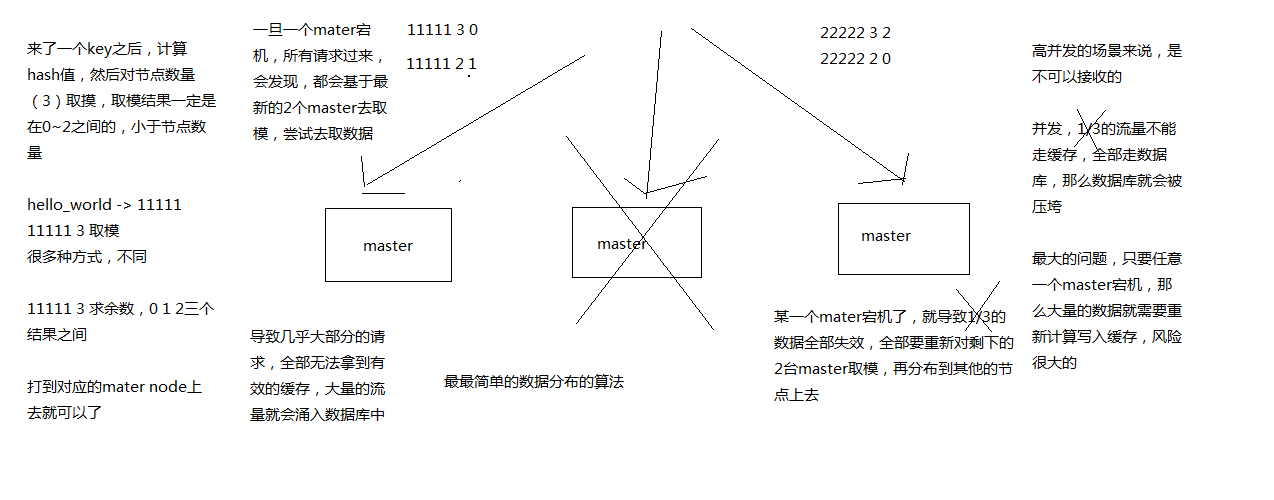


redis如何通过master横向扩容支撑1T+数据量

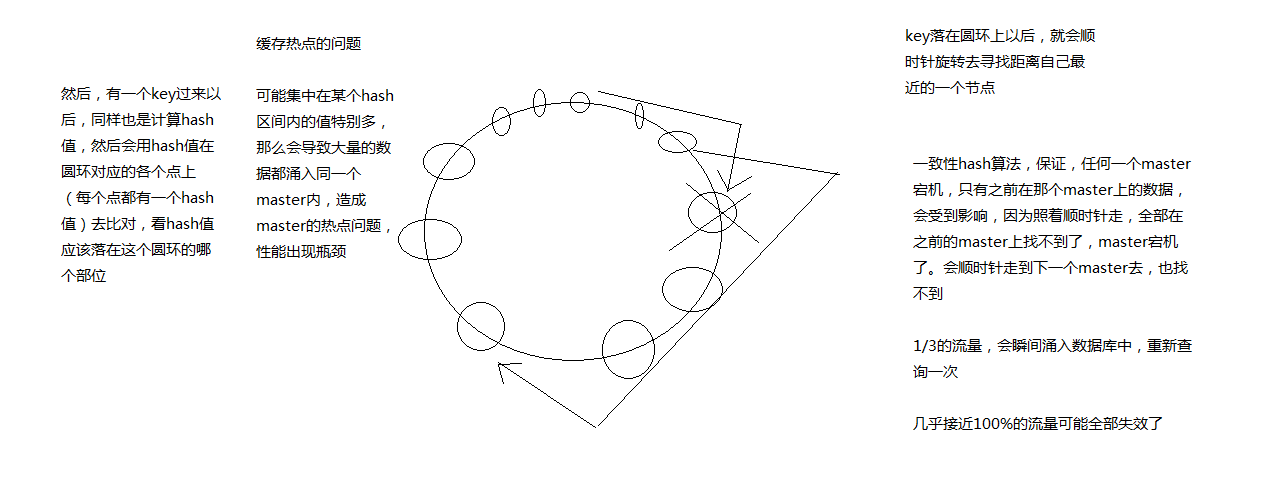


### 数据分布算法：hash+一致性hash+redis cluster的hash slot

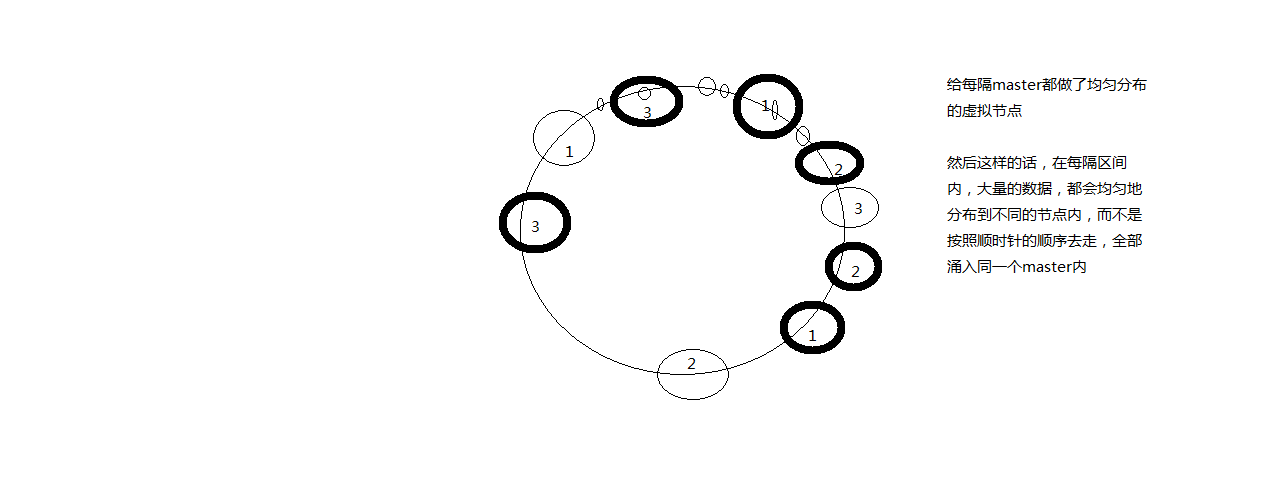
最老土的hash算法以及弊端



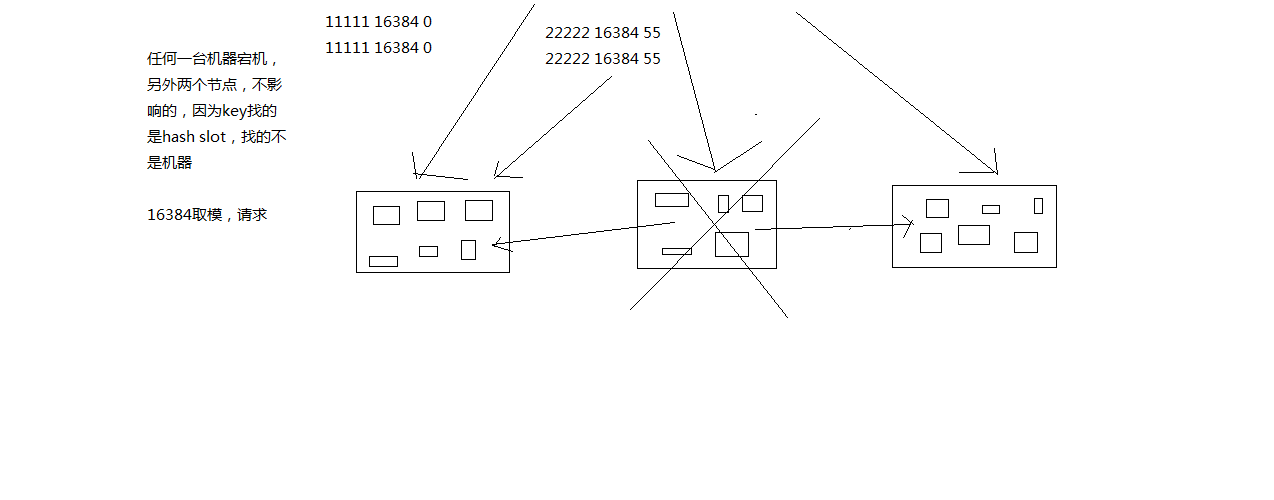
一致性hash算法的讲解和优点



一致性hash算法的虚拟节点实现负载均衡



redis cluster hash slot算法



### redis cluster的核心原理分析：gossip通信、jedis smart定位、主备切换

一、节点间的内部通信机制

1、基础通信原理

（1）redis cluster节点间采取gossip协议进行通信

跟集中式不同，不是将集群元数据（节点信息，故障，等等）集中存储在某个节点上，而是互相之间不断通信，保持整个集群所有节点的数据是完整的

维护集群的元数据用得，集中式，一种叫做gossip

集中式：好处在于，元数据的更新和读取，时效性非常好，一旦元数据出现了变更，立即就更新到集中式的存储中，其他节点读取的时候立即就可以感知到; 不好在于，所有的元数据的跟新压力全部集中在一个地方，可能会导致元数据的存储有压力

gossip：好处在于，元数据的更新比较分散，不是集中在一个地方，更新请求会陆陆续续，打到所有节点上去更新，有一定的延时，降低了压力; 缺点，元数据更新有延时，可能导致集群的一些操作会有一些滞后

我们刚才做reshard，去做另外一个操作，会发现说，configuration error，达成一致

（2）10000端口

每个节点都有一个专门用于节点间通信的端口，就是自己提供服务的端口号+10000，比如7001，那么用于节点间通信的就是17001端口

每隔节点每隔一段时间都会往另外几个节点发送ping消息，同时其他几点接收到ping之后返回pong

（3）交换的信息

故障信息，节点的增加和移除，hash slot信息，等等

2、gossip协议

gossip协议包含多种消息，包括ping，pong，meet，fail，等等

meet: 某个节点发送meet给新加入的节点，让新节点加入集群中，然后新节点就会开始与其他节点进行通信

redis-trib.rb add-node

其实内部就是发送了一个gossip meet消息，给新加入的节点，通知那个节点去加入我们的集群

ping: 每个节点都会频繁给其他节点发送ping，其中包含自己的状态还有自己维护的集群元数据，互相通过ping交换元数据

每个节点每秒都会频繁发送ping给其他的集群，ping，频繁的互相之间交换数据，互相进行元数据的更新

pong: 返回ping和meet，包含自己的状态和其他信息，也可以用于信息广播和更新

fail: 某个节点判断另一个节点fail之后，就发送fail给其他节点，通知其他节点，指定的节点宕机了

3、ping消息深入

ping很频繁，而且要携带一些元数据，所以可能会加重网络负担

每个节点每秒会执行10次ping，每次会选择5个最久没有通信的其他节点

当然如果发现某个节点通信延时达到了cluster\_node\_timeout / 2，那么立即发送ping，避免数据交换延时过长，落后的时间太长了

比如说，两个节点之间都10分钟没有交换数据了，那么整个集群处于严重的元数据不一致的情况，就会有问题

所以cluster\_node\_timeout可以调节，如果调节比较大，那么会降低发送的频率

每次ping，一个是带上自己节点的信息，还有就是带上1/10其他节点的信息，发送出去，进行数据交换

至少包含3个其他节点的信息，最多包含总节点-2个其他节点的信息

-------------------------------------------------------------------------------------------------------

二、面向集群的jedis内部实现原理

开发，jedis，redis的java client客户端，redis cluster，jedis cluster api

jedis cluster api与redis cluster集群交互的一些基本原理

1、基于重定向的客户端

redis-cli -c，自动重定向

（1）请求重定向

客户端可能会挑选任意一个redis实例去发送命令，每个redis实例接收到命令，都会计算key对应的hash slot

如果在本地就在本地处理，否则返回moved给客户端，让客户端进行重定向

cluster keyslot mykey，可以查看一个key对应的hash slot是什么

用redis-cli的时候，可以加入-c参数，支持自动的请求重定向，redis-cli接收到moved之后，会自动重定向到对应的节点执行命令

（2）计算hash slot

计算hash slot的算法，就是根据key计算CRC16值，然后对16384取模，拿到对应的hash slot

用hash tag可以手动指定key对应的slot，同一个hash tag下的key，都会在一个hash slot中，比如set mykey1:{100}和set mykey2:{100}

（3）hash slot查找

节点间通过gossip协议进行数据交换，就知道每个hash slot在哪个节点上

2、smart jedis

（1）什么是smart jedis

基于重定向的客户端，很消耗网络IO，因为大部分情况下，可能都会出现一次请求重定向，才能找到正确的节点

所以大部分的客户端，比如java redis客户端，就是jedis，都是smart的

本地维护一份hashslot -> node的映射表，缓存，大部分情况下，直接走本地缓存就可以找到hashslot -> node，不需要通过节点进行moved重定向

（2）JedisCluster的工作原理

在JedisCluster初始化的时候，就会随机选择一个node，初始化hashslot -> node映射表，同时为每个节点创建一个JedisPool连接池

每次基于JedisCluster执行操作，首先JedisCluster都会在本地计算key的hashslot，然后在本地映射表找到对应的节点

如果那个node正好还是持有那个hashslot，那么就ok; 如果说进行了reshard这样的操作，可能hashslot已经不在那个node上了，就会返回moved

如果JedisCluter API发现对应的节点返回moved，那么利用该节点的元数据，更新本地的hashslot -> node映射表缓存

重复上面几个步骤，直到找到对应的节点，如果重试超过5次，那么就报错，JedisClusterMaxRedirectionException

jedis老版本，可能会出现在集群某个节点故障还没完成自动切换恢复时，频繁更新hash slot，频繁ping节点检查活跃，导致大量网络IO开销

jedis最新版本，对于这些过度的hash slot更新和ping，都进行了优化，避免了类似问题

（3）hashslot迁移和ask重定向

如果hash slot正在迁移，那么会返回ask重定向给jedis

jedis接收到ask重定向之后，会重新定位到目标节点去执行，但是因为ask发生在hash slot迁移过程中，所以JedisCluster API收到ask是不会更新hashslot本地缓存

已经可以确定说，hashslot已经迁移完了，moved是会更新本地hashslot->node映射表缓存的

-------------------------------------------------------------------------------------------------------

三、高可用性与主备切换原理

redis cluster的高可用的原理，几乎跟哨兵是类似的

1、判断节点宕机

如果一个节点认为另外一个节点宕机，那么就是pfail，主观宕机

如果多个节点都认为另外一个节点宕机了，那么就是fail，客观宕机，跟哨兵的原理几乎一样，sdown，odown

在cluster-node-timeout内，某个节点一直没有返回pong，那么就被认为pfail

如果一个节点认为某个节点pfail了，那么会在gossip ping消息中，ping给其他节点，如果超过半数的节点都认为pfail了，那么就会变成fail

2、从节点过滤

对宕机的master node，从其所有的slave node中，选择一个切换成master node

检查每个slave node与master node断开连接的时间，如果超过了cluster-node-timeout \* cluster-slave-validity-factor，那么就没有资格切换成master

这个也是跟哨兵是一样的，从节点超时过滤的步骤

3、从节点选举

哨兵：对所有从节点进行排序，slave priority，offset，run id

每个从节点，都根据自己对master复制数据的offset，来设置一个选举时间，offset越大（复制数据越多）的从节点，选举时间越靠前，优先进行选举

所有的master node开始slave选举投票，给要进行选举的slave进行投票，如果大部分master node（N/2 + 1）都投票给了某个从节点，那么选举通过，那个从节点可以切换成master

从节点执行主备切换，从节点切换为主节点

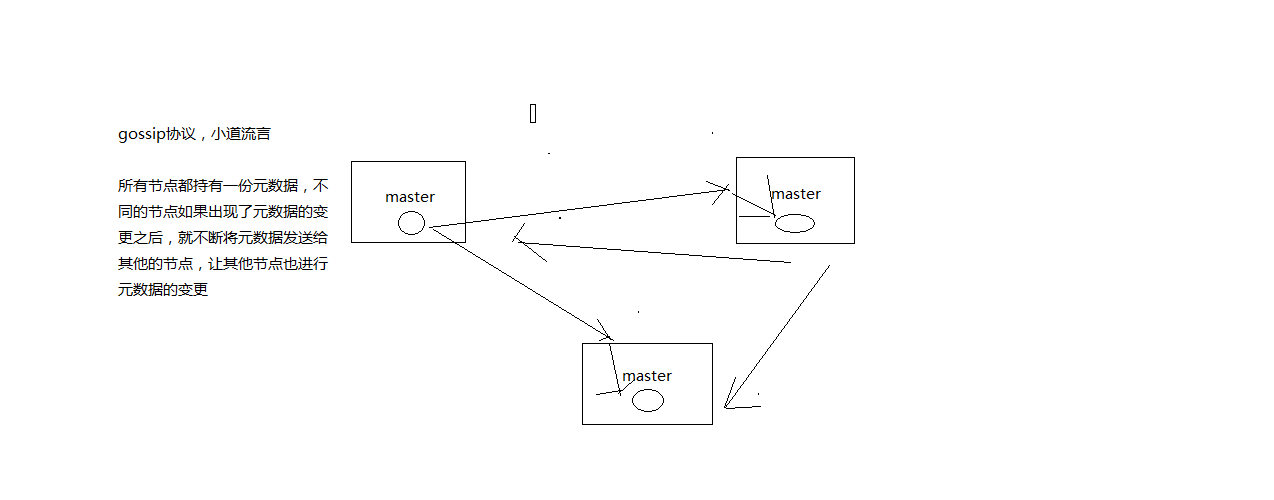
4、与哨兵比较

整个流程跟哨兵相比，非常类似，所以说，redis cluster功能强大，直接集成了replication和sentinal的功能

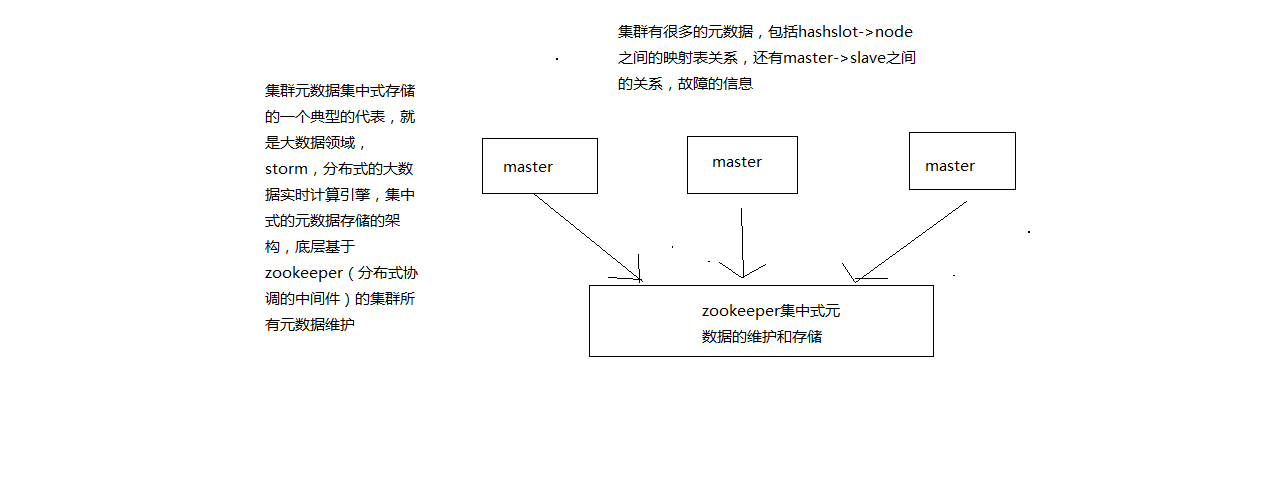
没有办法去给大家深入讲解redis底层的设计的细节，核心原理和设计的细节，那个除非单独开一门课，redis底层原理深度剖析，redis源码

对于咱们这个架构课来说，主要关注的是架构，不是底层的细节，对于架构来说，核心的原理的基本思路，是要梳理清晰的

gossip协议维护集群元数据



集中式的集群元数据存储和维护



# 你能说说我们一般如何应对缓存雪崩以及穿透问题吗？

了解什么是redis的雪崩和穿透？

redis崩溃之后会怎么样？

系统该如何应对这种情况？

如何处理redis的穿透？

其实这是问到缓存必问的，因为缓存雪崩和穿透，那是缓存最大的两个问题，要么不出现，一旦出现就是致命性的问题。所以面试官一定会问你。

缓存雪崩发生的现象

缓存雪崩的事前事中事后的解决方案

事前：redis高可用，主从+哨兵，redis cluster，避免全盘崩溃

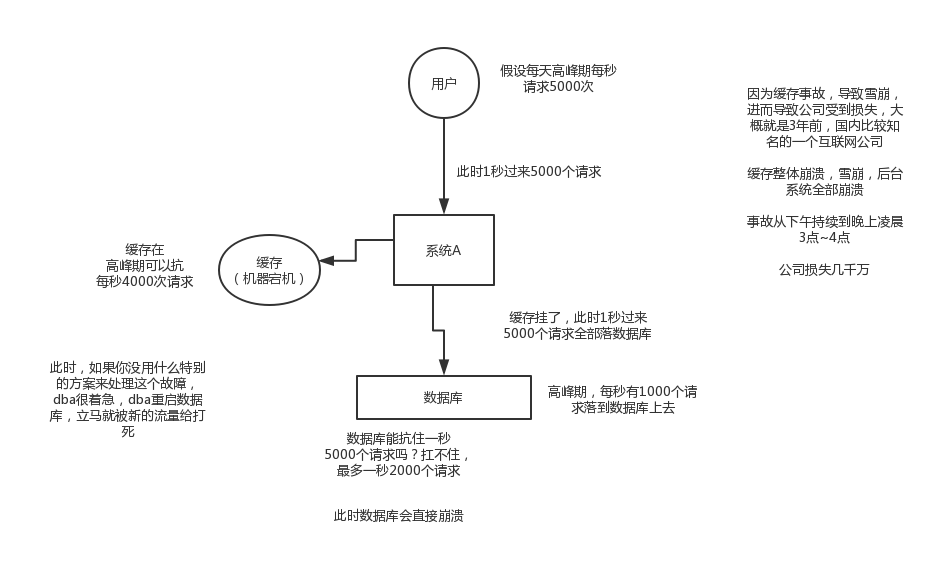
事中：本地ehcache缓存 + hystrix限流&降级，避免MySQL被打死

事后：redis持久化，快速恢复缓存数据

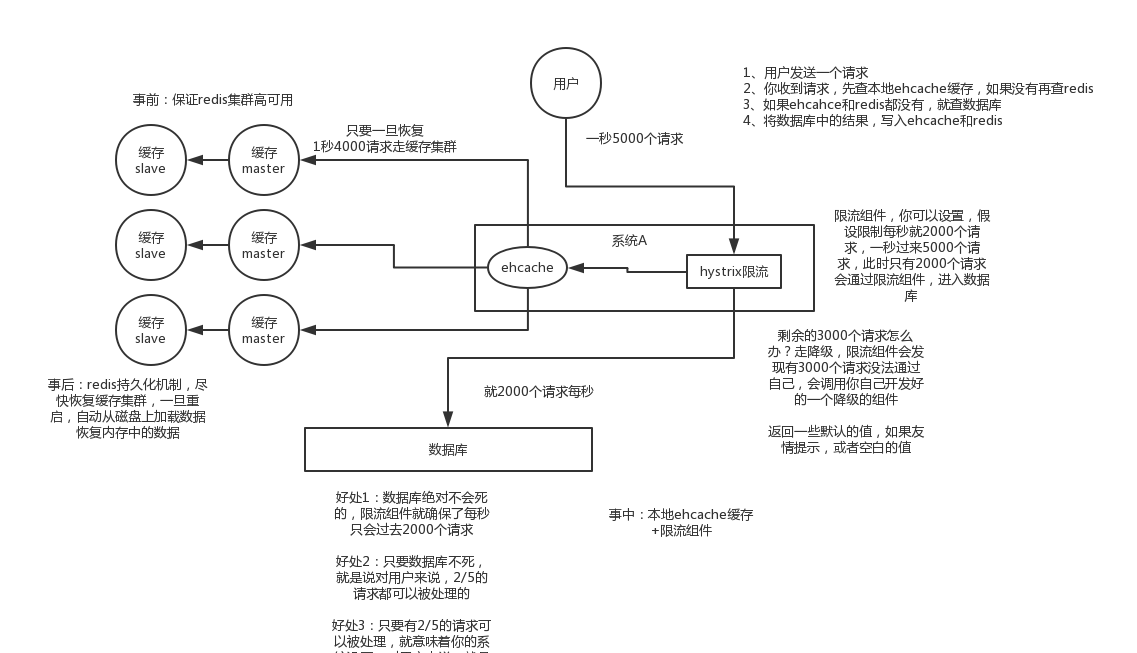
缓存穿透的现象

缓存穿透的解决方法

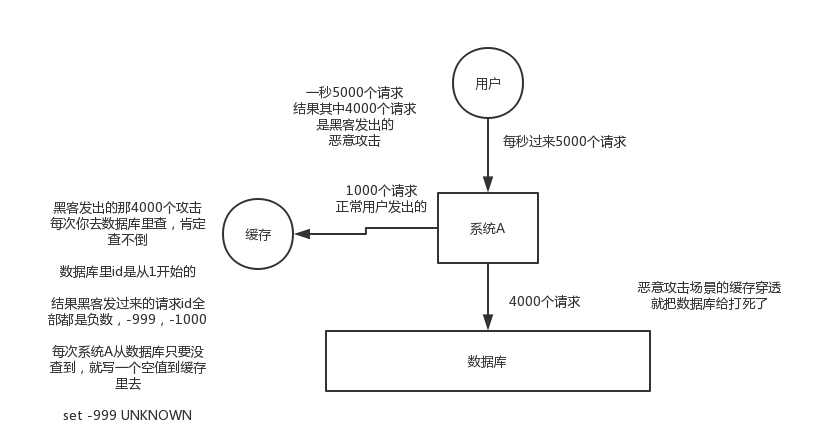
01\_缓存雪崩现象



02\_如何解决缓存雪崩



03\_缓存穿透现象以及解决方案



# 如何保证缓存与数据库双写时的数据一致性？

如何保证缓存与数据库的双写一致性？

你只要用缓存，就可能会涉及到缓存与数据库双存储双写，你只要是双写，就一定会有数据一致性的问题，那么你如何解决一致性问题？

一般来说，就是如果你的系统不是严格要求缓存+数据库必须一致性的话，缓存可以稍微的跟数据库偶尔有不一致的情况，最好不要做这个方案，读请求和写请求串行化，串到一个内存队列里去，这样就可以保证一定不会出现不一致的情况

串行化之后，就会导致系统的吞吐量会大幅度的降低，用比正常情况下多几倍的机器去支撑线上的一个请求。

### Cache Aside Pattern缓存+数据库读写模式的分析

最经典的缓存+数据库读写的模式，cache aside pattern

1、Cache Aside Pattern

（1）读的时候，先读缓存，缓存没有的话，那么就读数据库，然后取出数据后放入缓存，同时返回响应

（2）更新的时候，先删除缓存，然后再更新数据库

2、为什么是删除缓存，而不是更新缓存呢？

原因很简单，很多时候，复杂点的缓存的场景，因为缓存有的时候，不简单是数据库中直接取出来的值

商品详情页的系统，修改库存，只是修改了某个表的某些字段，但是要真正把这个影响的最终的库存计算出来，可能还需要从其他表查询一些数据，然后进行一些复杂的运算，才能最终计算出

现在最新的库存是多少，然后才能将库存更新到缓存中去

比如可能更新了某个表的一个字段，然后其对应的缓存，是需要查询另外两个表的数据，并进行运算，才能计算出缓存最新的值的

更新缓存的代价是很高的

是不是说，每次修改数据库的时候，都一定要将其对应的缓存去跟新一份？也许有的场景是这样的，但是对于比较复杂的缓存数据计算的场景，就不是这样了

如果你频繁修改一个缓存涉及的多个表，那么这个缓存会被频繁的更新，频繁的更新缓存

但是问题在于，这个缓存到底会不会被频繁访问到？？？

举个例子，一个缓存涉及的表的字段，在1分钟内就修改了20次，或者是100次，那么缓存跟新20次，100次; 但是这个缓存在1分钟内就被读取了1次，有大量的冷数据

28法则，黄金法则，20%的数据，占用了80%的访问量

实际上，如果你只是删除缓存的话，那么1分钟内，这个缓存不过就重新计算一次而已，开销大幅度降低

每次数据过来，就只是删除缓存，然后修改数据库，如果这个缓存，在1分钟内只是被访问了1次，那么只有那1次，缓存是要被重新计算的，用缓存才去算缓存

其实删除缓存，而不是更新缓存，就是一个lazy计算的思想，不要每次都重新做复杂的计算，不管它会不会用到，而是让它到需要被使用的时候再重新计算

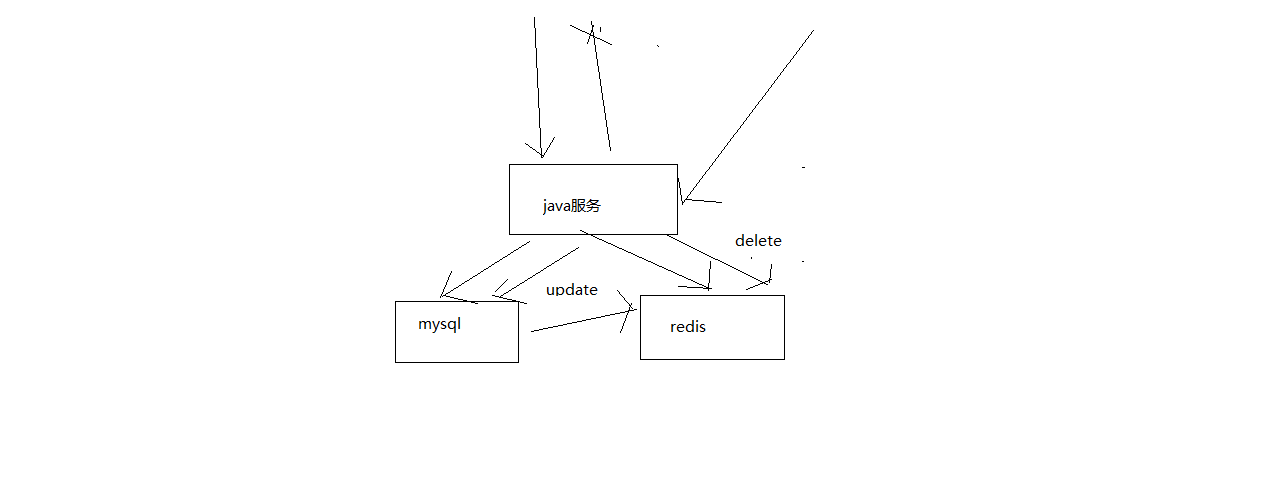
mybatis，hibernate，懒加载，思想

查询一个部门，部门带了一个员工的list，没有必要说每次查询部门，都里面的1000个员工的数据也同时查出来啊

80%的情况，查这个部门，就只是要访问这个部门的信息就可以了

先查部门，同时要访问里面的员工，那么这个时候只有在你要访问里面的员工的时候，才会去数据库里面查询1000个员工

cache aside pattern



### 高并发场景下的缓存+数据库双写不一致问题分析与解决方案设计

马上开始去开发业务系统

从哪一步开始做，从比较简单的那一块开始做，实时性要求比较高的那块数据的缓存去做

实时性比较高的数据缓存，选择的就是库存的服务

库存可能会修改，每次修改都要去更新这个缓存数据; 每次库存的数据，在缓存中一旦过期，或者是被清理掉了，前端的nginx服务都会发送请求给库存服务，去获取相应的数据

库存这一块，写数据库的时候，直接更新redis缓存

实际上没有这么的简单，这里，其实就涉及到了一个问题，数据库与缓存双写，数据不一致的问题

围绕和结合实时性较高的库存服务，把数据库与缓存双写不一致问题以及其解决方案，给大家讲解一下

数据库与缓存双写不一致，很常见的问题，大型的缓存架构中，第一个解决方案

大型的缓存架构全部讲解完了以后，整套架构是非常复杂，架构可以应对各种各样奇葩和极端的情况

也有一种可能，不是说，来讲课的就是超人，万能的

讲课，就跟写书一样，很可能会写错，也可能有些方案里的一些地方，我没考虑到

也可能说，有些方案只是适合某些场景，在某些场景下，可能需要你进行方案的优化和调整才能适用于你自己的项目

大家觉得对这些方案有什么疑问或者见解，都可以找我，沟通一下

如果的确我觉得是我讲解的不对，或者有些地方考虑不周，那么我可以在视频里补录，更新到网站上面去

多多包涵

1、最初级的缓存不一致问题以及解决方案

问题：先修改数据库，再删除缓存，如果删除缓存失败了，那么会导致数据库中是新数据，缓存中是旧数据，数据出现不一致

解决思路

先删除缓存，再修改数据库，如果删除缓存成功了，如果修改数据库失败了，那么数据库中是旧数据，缓存中是空的，那么数据不会不一致

因为读的时候缓存没有，则读数据库中旧数据，然后更新到缓存中

2、比较复杂的数据不一致问题分析

数据发生了变更，先删除了缓存，然后要去修改数据库，此时还没修改

一个请求过来，去读缓存，发现缓存空了，去查询数据库，查到了修改前的旧数据，放到了缓存中

数据变更的程序完成了数据库的修改

完了，数据库和缓存中的数据不一样了。。。。

3、为什么上亿流量高并发场景下，缓存会出现这个问题？

只有在对一个数据在并发的进行读写的时候，才可能会出现这种问题

其实如果说你的并发量很低的话，特别是读并发很低，每天访问量就1万次，那么很少的情况下，会出现刚才描述的那种不一致的场景

但是问题是，如果每天的是上亿的流量，每秒并发读是几万，每秒只要有数据更新的请求，就可能会出现上述的数据库+缓存不一致的情况

高并发了以后，问题是很多的

4、数据库与缓存更新与读取操作进行异步串行化

更新数据的时候，根据数据的唯一标识，将操作路由之后，发送到一个jvm内部的队列中

读取数据的时候，如果发现数据不在缓存中，那么将重新读取数据+更新缓存的操作，根据唯一标识路由之后，也发送同一个jvm内部的队列中

一个队列对应一个工作线程

每个工作线程串行拿到对应的操作，然后一条一条的执行

这样的话，一个数据变更的操作，先执行，删除缓存，然后再去更新数据库，但是还没完成更新

此时如果一个读请求过来，读到了空的缓存，那么可以先将缓存更新的请求发送到队列中，此时会在队列中积压，然后同步等待缓存更新完成

这里有一个优化点，一个队列中，其实多个更新缓存请求串在一起是没意义的，因此可以做过滤，如果发现队列中已经有一个更新缓存的请求了，那么就不用再放个更新请求操作进去了，直接等待前面的更新操作请求完成即可

待那个队列对应的工作线程完成了上一个操作的数据库的修改之后，才会去执行下一个操作，也就是缓存更新的操作，此时会从数据库中读取最新的值，然后写入缓存中

如果请求还在等待时间范围内，不断轮询发现可以取到值了，那么就直接返回; 如果请求等待的时间超过一定时长，那么这一次直接从数据库中读取当前的旧值

5、高并发的场景下，该解决方案要注意的问题

（1）读请求长时阻塞

由于读请求进行了非常轻度的异步化，所以一定要注意读超时的问题，每个读请求必须在超时时间范围内返回

该解决方案，最大的风险点在于说，可能数据更新很频繁，导致队列中积压了大量更新操作在里面，然后读请求会发生大量的超时，最后导致大量的请求直接走数据库

务必通过一些模拟真实的测试，看看更新数据的频繁是怎样的

另外一点，因为一个队列中，可能会积压针对多个数据项的更新操作，因此需要根据自己的业务情况进行测试，可能需要部署多个服务，每个服务分摊一些数据的更新操作

如果一个内存队列里居然会挤压100个商品的库存修改操作，每隔库存修改操作要耗费10ms区完成，那么最后一个商品的读请求，可能等待10 \* 100 = 1000ms = 1s后，才能得到数据

这个时候就导致读请求的长时阻塞

一定要做根据实际业务系统的运行情况，去进行一些压力测试，和模拟线上环境，去看看最繁忙的时候，内存队列可能会挤压多少更新操作，可能会导致最后一个更新操作对应的读请求，会hang多少时间，如果读请求在200ms返回，如果你计算过后，哪怕是最繁忙的时候，积压10个更新操作，最多等待200ms，那还可以的

如果一个内存队列可能积压的更新操作特别多，那么你就要加机器，让每个机器上部署的服务实例处理更少的数据，那么每个内存队列中积压的更新操作就会越少

其实根据之前的项目经验，一般来说数据的写频率是很低的，因此实际上正常来说，在队列中积压的更新操作应该是很少的

针对读高并发，读缓存架构的项目，一般写请求相对读来说，是非常非常少的，每秒的QPS能到几百就不错了

一秒，500的写操作，5份，每200ms，就100个写操作

单机器，20个内存队列，每个内存队列，可能就积压5个写操作，每个写操作性能测试后，一般在20ms左右就完成

那么针对每个内存队列中的数据的读请求，也就最多hang一会儿，200ms以内肯定能返回了

写QPS扩大10倍，但是经过刚才的测算，就知道，单机支撑写QPS几百没问题，那么就扩容机器，扩容10倍的机器，10台机器，每个机器20个队列，200个队列

大部分的情况下，应该是这样的，大量的读请求过来，都是直接走缓存取到数据的

少量情况下，可能遇到读跟数据更新冲突的情况，如上所述，那么此时更新操作如果先入队列，之后可能会瞬间来了对这个数据大量的读请求，但是因为做了去重的优化，所以也就一个更新缓存的操作跟在它后面

等数据更新完了，读请求触发的缓存更新操作也完成，然后临时等待的读请求全部可以读到缓存中的数据

（2）读请求并发量过高

这里还必须做好压力测试，确保恰巧碰上上述情况的时候，还有一个风险，就是突然间大量读请求会在几十毫秒的延时hang在服务上，看服务能不能抗的住，需要多少机器才能抗住最大的极限情况的峰值

但是因为并不是所有的数据都在同一时间更新，缓存也不会同一时间失效，所以每次可能也就是少数数据的缓存失效了，然后那些数据对应的读请求过来，并发量应该也不会特别大

按1:99的比例计算读和写的请求，每秒5万的读QPS，可能只有500次更新操作

如果一秒有500的写QPS，那么要测算好，可能写操作影响的数据有500条，这500条数据在缓存中失效后，可能导致多少读请求，发送读请求到库存服务来，要求更新缓存

一般来说，1:1，1:2，1:3，每秒钟有1000个读请求，会hang在库存服务上，每个读请求最多hang多少时间，200ms就会返回

在同一时间最多hang住的可能也就是单机200个读请求，同时hang住

单机hang200个读请求，还是ok的

1:20，每秒更新500条数据，这500秒数据对应的读请求，会有20 \* 500 = 1万

1万个读请求全部hang在库存服务上，就死定了

（3）多服务实例部署的请求路由

可能这个服务部署了多个实例，那么必须保证说，执行数据更新操作，以及执行缓存更新操作的请求，都通过nginx服务器路由到相同的服务实例上

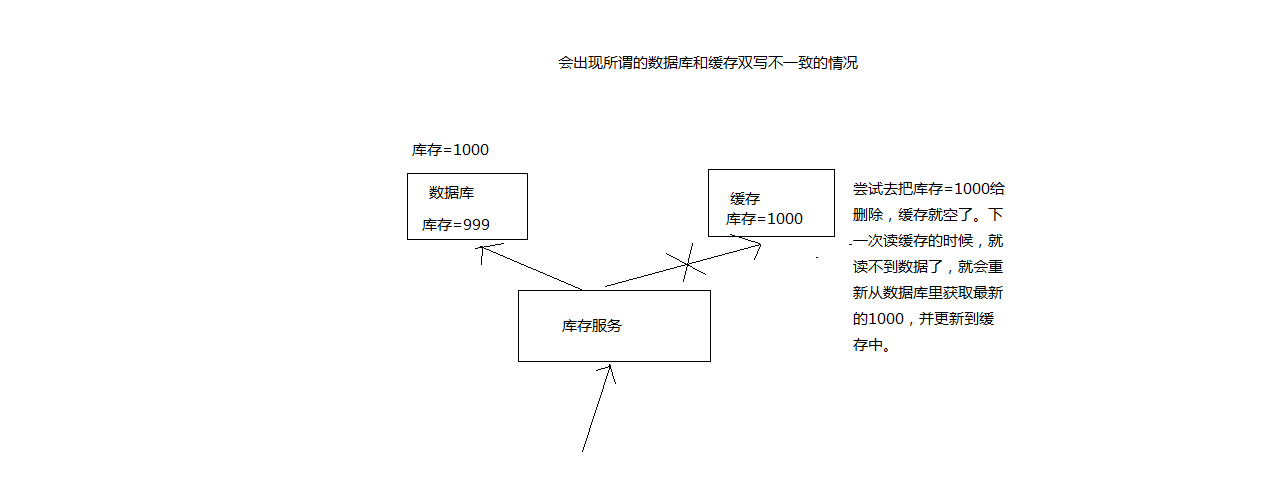
（4）热点商品的路由问题，导致请求的倾斜

万一某个商品的读写请求特别高，全部打到相同的机器的相同的队列里面去了，可能造成某台机器的压力过大

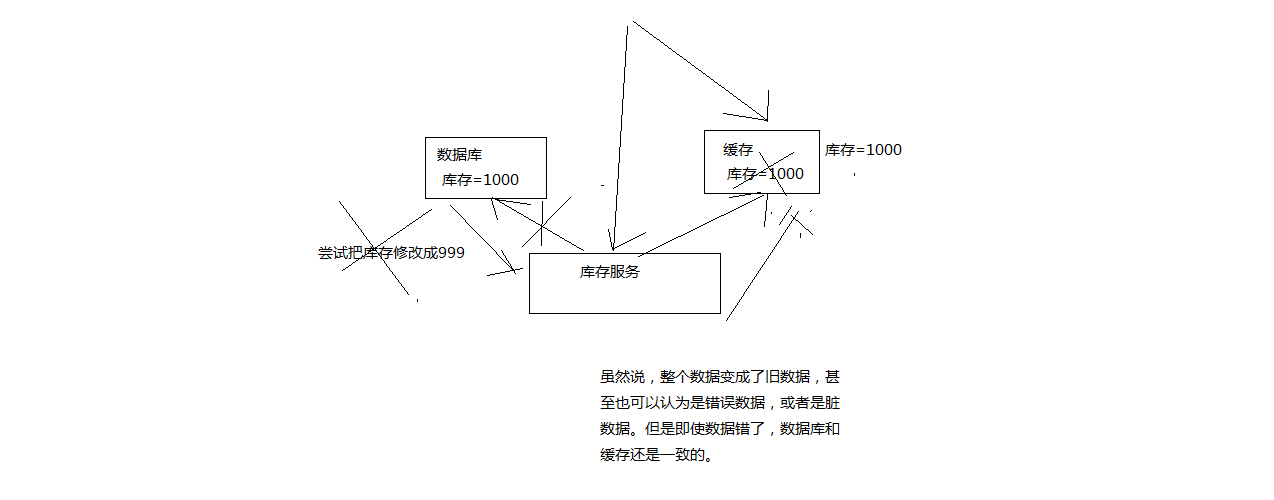
就是说，因为只有在商品数据更新的时候才会清空缓存，然后才会导致读写并发，所以更新频率不是太高的话，这个问题的影响并不是特别大

但是的确可能某些机器的负载会高一些

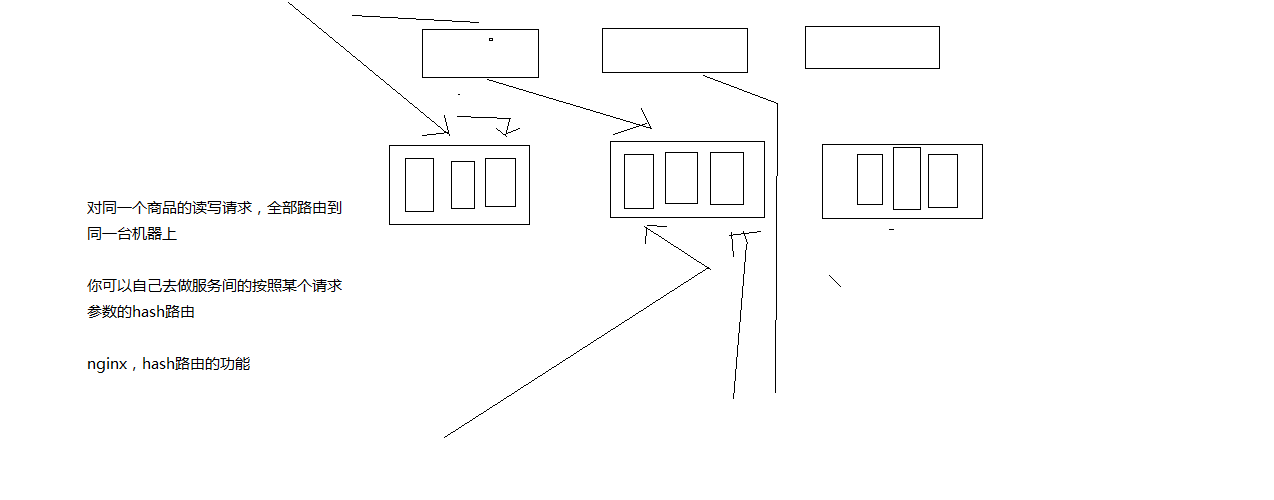
最初级的数据库+缓存双写不一致问题



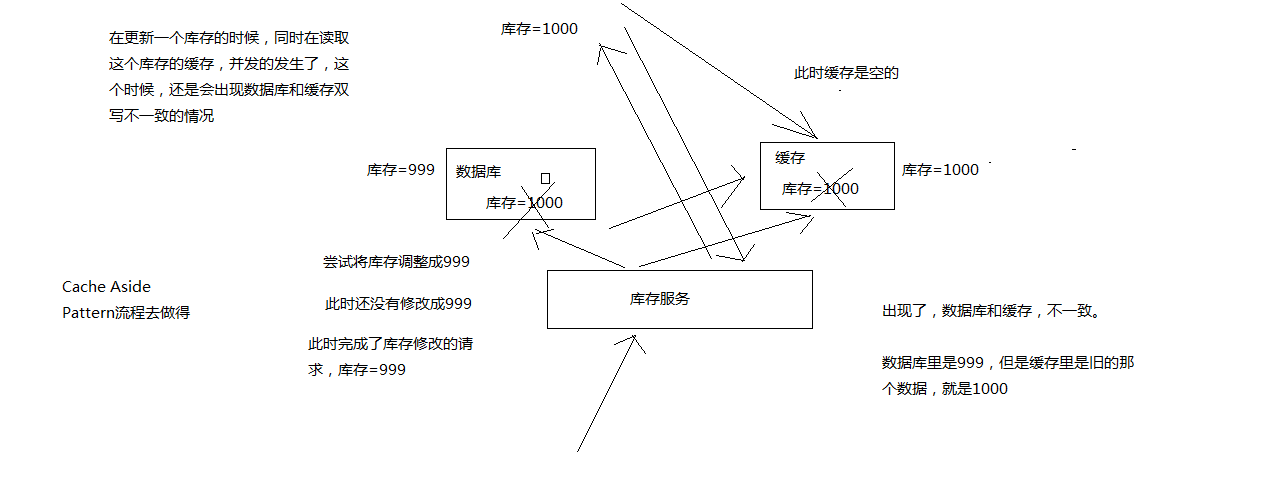
最初级的数据库+缓存双写不一致问题的解决方案



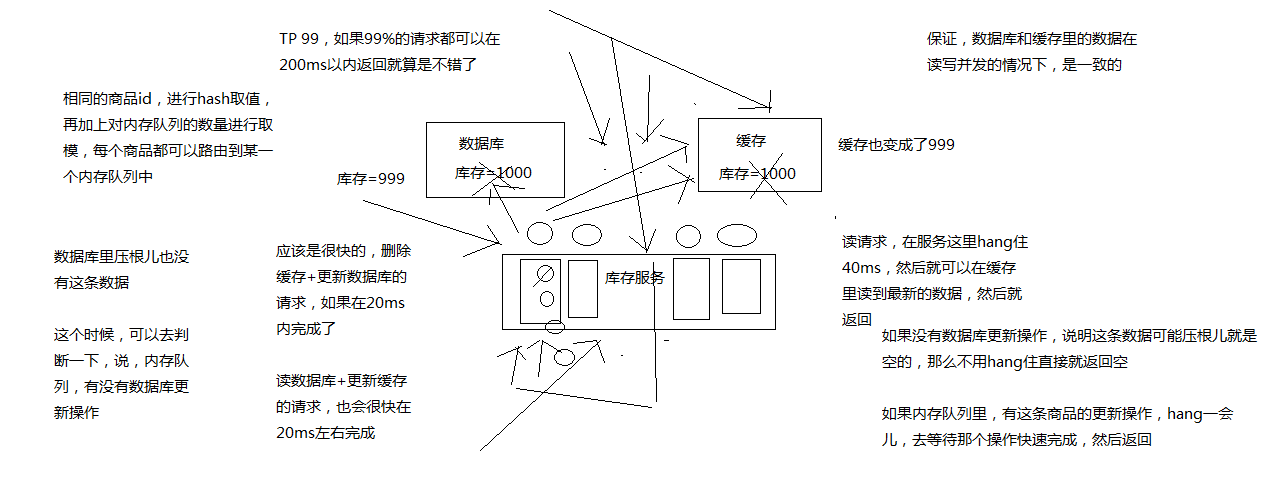
机器级别的请求路由问题



读写并发的时候复杂的数据库+缓存双写不一致的场景



复杂的数据库+缓存双写一致保障方案



### 你能说说redis的并发竞争问题该如何解决吗？

redis的并发竞争问题是什么？

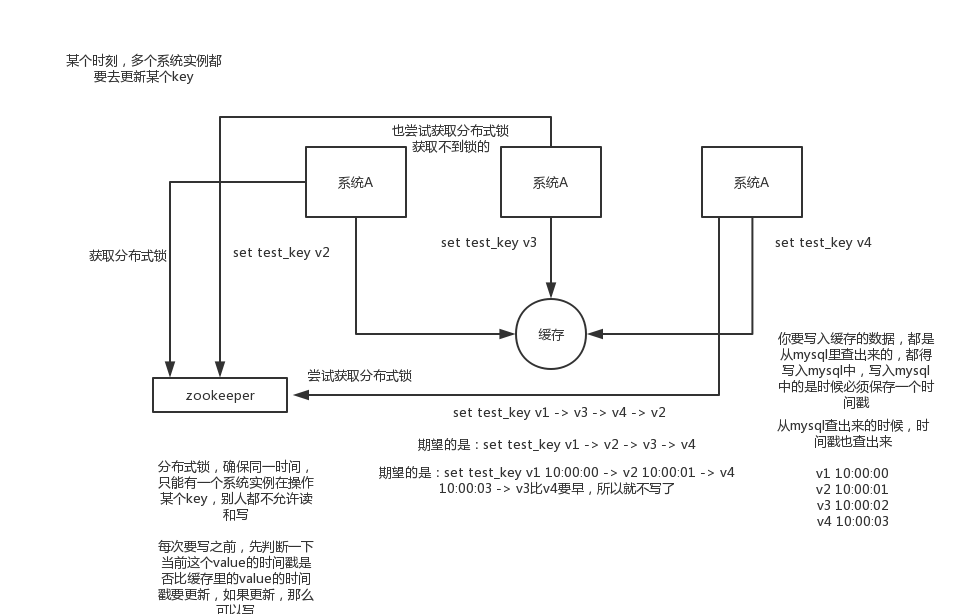
如何解决这个问题？

了解Redis事务的CAS方案吗？

这个也是线上非常常见的一个问题，就是多客户端同时并发写一个key，可能本来应该先到的数据后到了，导致数据版本错了。或者是多客户端同时获取一个key，修改值之后再写回去，只要顺序错了，数据就错了。

而且redis自己就有天然解决这个问题的CAS类的乐观锁方案

01\_redis并发竞争问题以及解决方案



# 你们公司生产环境的redis集群的部署架构是什么样的？

生产环境中的redis是怎么部署的？

看看你了解不了解你们公司的redis生产集群的部署架构，如果你不了解，那么确实你就很失职了，你的redis是主从架构？集群架构？用了哪种集群方案？有没有做高可用保证？有没有开启持久化机制确保可以进行数据恢复？线上redis给几个G的内存？设置了哪些参数？压测后你们redis集群承载多少QPS？

兄弟，这些你必须是门儿清的，否则你确实是没好好思考过

redis cluster，10台机器，5台机器部署了redis主实例，另外5台机器部署了redis的从实例，每个主实例挂了一个从实例，5个节点对外提供读写服务，每个节点的读写高峰qps可能可以达到每秒5万，5台机器最多是25万读写请求/s。

机器是什么配置？32G内存+8核CPU+1T磁盘，但是分配给redis进程的是10g内存，一般线上生产环境，redis的内存尽量不要超过10g，超过10g可能会有问题。

5台机器对外提供读写，一共有50g内存。

因为每个主实例都挂了一个从实例，所以是高可用的，任何一个主实例宕机，都会自动故障迁移，redis从实例会自动变成主实例继续提供读写服务

你往内存里写的是什么数据？每条数据的大小是多少？商品数据，每条数据是10kb。100条数据是1mb，10万条数据是1g。常驻内存的是200万条商品数据，占用内存是20g，仅仅不到总内存的50%。

目前高峰期每秒就是3500左右的请求量

比如我们吧，大型的公司，其实基础架构的team，会负责缓存集群的运维

# 分布式缓存相关面试题的回答技巧总结

说实话，这一套东西基本构成了缓存这块你必须知道的基础性的知识，如果你不知道，那么说明你有点失职，确实平时没好好积累。

因为这些问题确实不难，如果我往深了问，可以问的很细，结合项目扣的很细，比如你们公司线上系统高峰QPS 3000？那请求主要访问哪些接口？redis抗了多少请求？mysql抗了多少请求？你到底是怎么实现高并发的？咱们聊聊redis的内核吧，看看你对底层了解的多么？如果要缓存几百GB的数据会有什么坑该这么弄？如果缓存出现热点现象该这么处理？某个value特别大把网卡给打死了怎么办？等等等等，可以深挖的东西其实有很多。。。。。

但是如果你掌握好了这套东西的回答，那么你在面试的时候，如果面试官没有全都问到，你可以自己主动合盘脱出。比如你可以说，我们线上的缓存，做了啥啥机制，防止雪崩、防止穿透、保证双写时的数据一致性、保证并发竞争时的数据一致性，我们线上咋部署的，啥架构，怎么玩儿的。这套东西你可以自己说出来，展示一下你对缓存这块的掌握。