# Redis概述

## 什么是redis

Redis 是一个基于内存的高性能key-value数据库。

## Redis特点

Redis本质上是一个Key-Value类型的内存数据库，很像memcached，整个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flush到硬盘上进行保存。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。

# Redis与memcache比较

1、性能

都比较高，性能对我们来说应该都不是瓶颈

总体来讲，TPS方面redis和memcache差不多，要大于mongodb

2、操作的便利性

memcache数据结构单一

redis丰富一些，数据操作方面，redis更好一些，较少的网络IO次数

mongodb支持丰富的数据表达，索引，最类似关系型数据库，支持的查询语言非常丰富

3、内存空间的大小和数据量的大小

redis在2.0版本后增加了自己的VM特性，突破物理内存的限制；可以对key value设置过期时间（类似memcache）

memcache可以修改最大可用内存,采用LRU算法

mongoDB适合大数据量的存储，依赖操作系统VM做内存管理，吃内存也比较厉害，服务不要和别的服务在一起

4、可用性（单点问题）

对于单点问题，

redis，依赖客户端来实现分布式读写；主从复制时，每次从节点重新连接主节点都要依赖整个快照,无增量复制，因性能和效率问题，所以单点问题比较复杂；不支持自动sharding,需要依赖程序设定一致hash 机制。一种替代方案是，不用redis本身的复制机制，采用自己做主动复制（多份存储），或者改成增量复制的方式（需要自己实现），一致性问题和性能的权衡、Memcache本身没有数据冗余机制，也没必要；对于故障预防，采用依赖成熟的hash或者环状的算法，解决单点故障引起的抖动问题。

mongoDB支持master-slave,replicaset（内部采用paxos选举算法，自动故障恢复）,auto sharding机制，对客户端屏蔽了故障转移和切分机制。

5、可靠性（持久化）

对于数据持久化和数据恢复，

redis支持（快照、AOF）：依赖快照进行持久化，aof增强了可靠性的同时，对性能有所影响

memcache不支持，通常用在做缓存,提升性能；

6、数据一致性（事务支持）

Memcache 在并发场景下，用cas保证一致性

redis事务支持比较弱，只能保证事务中的每个操作连续执行

mongoDB不支持事务

7、数据分析

mongoDB内置了数据分析的功能(mapreduce),其他不支持

8、应用场景

redis：数据量较小的性能操作和运算上

memcache：用于在动态系统中减少数据库负载，提升性能;做缓存，提高性能（适合读多写少，对于数据量比较大，可以采用sharding）

MongoDB:主要解决海量数据的访问效率问题

# Redis支持的数据类型

Redis通过Key-Value的单值不同类型来区分, 以下是支持的类型:  
Strings  
Lists  
Sets 求交集、并集  
Sorted Set   
hashes

# Redis缺点

分析:大家用redis这么久，这个问题是必须要了解的，基本上使用redis都会碰到一些问题，常见的面试也会问这几个。

(一)缓存和数据库双写一致性问题

分析:一致性问题是分布式常见问题，还可以再分为最终一致性和强一致性。数据库和缓存双写，就必然会存在不一致的问题。答这个问题，先明白一个前提。就是如果对数据有强一致性要求，不能放缓存。我们所做的一切，只能保证最终一致性。另外，我们所做的方案其实从根本上来说，只能说降低不一致发生的概率，无法完全避免。因此，有强一致性要求的数据，不能放缓存。

首先，采取正确更新策略，先更新数据库，再删缓存。其次，因为可能存在删除缓存失败的问题，提供一个补偿措施即可，例如利用消息队列。

(二)缓存雪崩问题

一般中小型传统软件企业，很难碰到这个问题。如果有大并发的项目，流量有几百万左右。这两个问题一定要深刻考虑。

缓存雪崩，即缓存同一时间大面积的失效，这个时候又来了一波请求，结果请求都怼到数据库上，从而导致数据库连接异常。

解决方案:

① 给缓存的失效时间，加上一个随机值，避免集体失效。

② 使用互斥锁，但是该方案吞吐量明显下降了。

③ 双缓存。我们有两个缓存，缓存A和缓存B。缓存A的失效时间为20分钟，缓存B不设失效时间。自己做缓存预热操作。然后细分以下几个小点

I 从缓存A读数据库，有则直接返回

II A没有数据，直接从B读数据，直接返回，并且异步启动一个更新线程。

III 更新线程同时更新缓存A和缓存B。

(三)缓存击穿问题

一般中小型传统软件企业，很难碰到这个问题。如果有大并发的项目，流量有几百万左右。这两个问题一定要深刻考虑。

缓存穿透，即黑客故意去请求缓存中不存在的数据，导致所有的请求都怼到数据库上，从而数据库连接异常。

解决方案:

① 利用互斥锁，缓存失效的时候，先去获得锁，得到锁了，再去请求数据库。没得到锁，则休眠一段时间重试

② 采用异步更新策略，无论key是否取到值，都直接返回。value值中维护一个缓存失效时间，缓存如果过期，异步起一个线程去读数据库，更新缓存。需要做缓存预热(项目启动前，先加载缓存)操作。

③ 提供一个能迅速判断请求是否有效的拦截机制，比如，利用布隆过滤器，内部维护一系列合法有效的key。迅速判断出，请求所携带的Key是否合法有效。如果不合法，则直接返回。

(四)缓存的并发竞争问题

分析:这个问题大致就是，同时有多个子系统去set一个key。这个时候要注意什么呢？大家思考过么。需要说明一下，博主提前百度了一下，发现答案基本都是推荐用redis事务机制。博主不推荐使用redis的事务机制。因为我们的生产环境，基本都是redis集群环境，做了数据分片操作。你一个事务中有涉及到多个key操作的时候，这多个key不一定都存储在同一个redis-server上。因此，redis的事务机制，十分鸡肋。

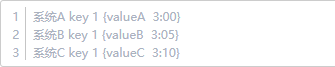
① 如果对这个key操作，不要求顺序

这种情况下，准备一个分布式锁，大家去抢锁，抢到锁就做set操作即可，比较简单。

② 如果对这个key操作，要求顺序

假设有一个key1,系统A需要将key1设置为valueA,系统B需要将key1设置为valueB,系统C需要将key1设置为valueC.

期望按照key1的value值按照 valueA-->valueB-->valueC的顺序变化。这种时候我们在数据写入数据库的时候，需要保存一个时间戳。假设时间戳如下



那么，假设这会系统B先抢到锁，将key1设置为{valueB 3:05}。接下来系统A抢到锁，发现自己的valueA的时间戳早于缓存中的时间戳，那就不做set操作了。以此类推。

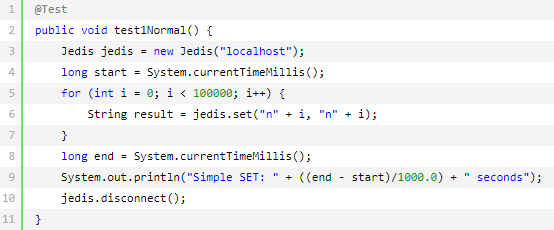
其他方法，比如利用队列，将set方法变成串行访问也可以。总之，灵活变通。

# Java相关的API



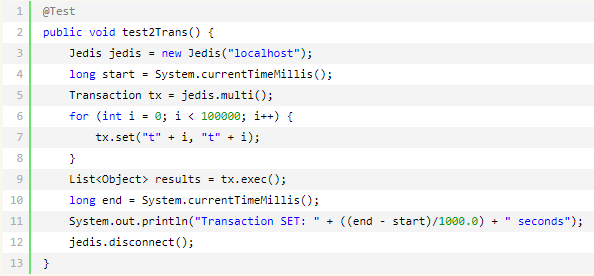
## Java调用redis几种方式

### 普通同步调用



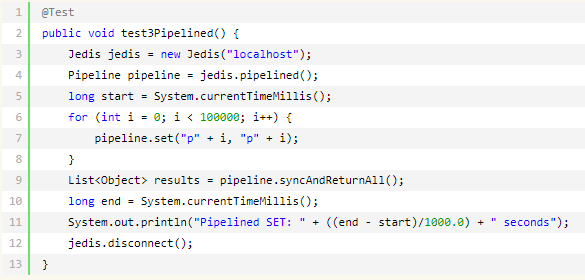
### 事务方式

redis的事务很简单，他主要目的是保障，一个client发起的事务中的命令可以连续的执行，而中间不会插入其他client的命令。

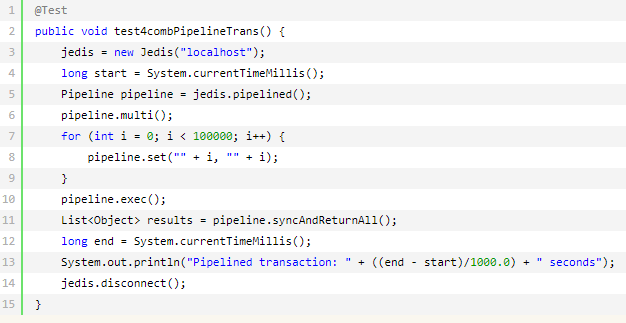


### 管道方式

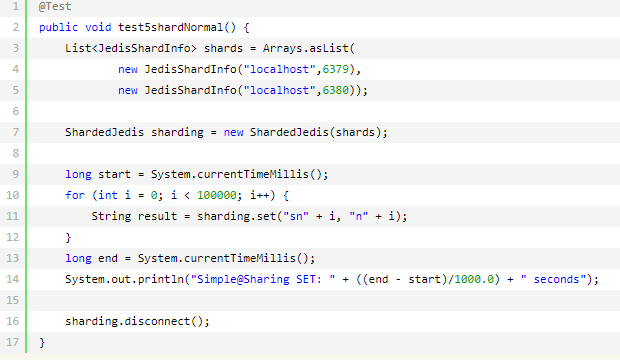
我们需要采用异步方式，一次发送多个指令，不同步等待其返回结果。这样可以取得非常好的执行效率。这就是管道。



### 管道中调用事务



### 分布式直连同步调用



# Redis线程模式

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。Redis使用多路复用I/O模式。多个网络连接使用一个线程，当无I/O事件，线程处于阻塞状态，当有I/O事件是，基于事件驱动，程序会轮询所有真正有I/O事件的流。

### 单进程单线程好处

1、代码更清晰，处理逻辑更简单

2、不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗

3、不存在多进程或者多线程导致的切换而消耗CPU

### 单进程单线程缺点

1、无法发挥多核CPU性能，不过可以通过在单机开多个Redis实例来完善；

# Redis部署模式

## 单节点模式

单节点模式是最简单的Redis模式，就是一个redis实例，如果只是自己测试缓存或者小程序，数据量很小，仅仅做一个小型的KEY/VALUE型数据库，完全足够。

## 主从模式

主从模式就是N个redis实例，可以是1主N从。

### 特点

1.一个Master可以有多个Slaves，可以是1主N从。

2.默认配置下，master节点可以进行读和写，slave节点只能进行读操作，写操作被禁止（readonly）。

3.不要修改配置让slave节点支持写操作，没有意义，原因一，写入的数据不会被同步到其他节点；原因二，当master节点修改同一条数据后，slave节点的数据会被覆盖掉。

4.slave节点挂了不影响其他slave节点的读和master节点的读和写，重新启动后会将数据从master节点同步过来。

5.master节点挂了以后，不影响slave节点的读，Redis将不再提供写服务，master节点启动后Redis将重新对外提供写服务。

6.特别说明：该种模式下，master节点挂了以后，slave不会竞选成为master。

对有密码的情况说明一下，当master节点设置密码时：

客户端访问master需要密码

启动slave需要密码，在配置中进行配置即可

客户端访问slave不需要密码

综上，客户端只需要配置一个密码参数，而redis配置文件中需要配置两个参数。

分别是：

Redis服务端配置文件：

masterauth "chrdw,hdhxt!"

requirepass "chrdw,hdhxt!"

客户端配置文件：

jedis-cluster.password=chrdw,hdhxt!

注意没有引号。

### 作用

主从模式的一个作用是备份数据，这样当一个节点损坏（指不可恢复的硬件损坏）时，数据因为有备份，可以方便恢复。

另一个作用是负载均衡，所有客户端都访问一个节点肯定会影响Redis工作效率，有了主从以后，查询操作就可以通过查询从节点来完成。

### Java启动

Jedis jedis=new Jedis("127.0.0.1", 56379);

jedis.close();

## 哨兵模式

sentinel的中文含义是哨兵、守卫。也就是说既然主从模式中，当master节点挂了以后，slave节点不能主动选举一个master节点出来，那么我就安排一个或多个sentinel来做这件事，当sentinel发现master节点挂了以后，sentinel就会从slave中重新选举一个master。

### 特点

1.sentinel模式是建立在主从模式的基础上，如果只有一个Redis节点，sentinel就没有任何意义。

2.当master节点挂了以后，sentinel会在slave中选择一个做为master，并修改它们的配置文件，其他slave的配置文件也会被修改，比如slaveof属性会指向新的master。

3.当master节点重新启动后，它将不再是master，而是作为slave接收新的master节点的同步数据

4.sentinel因为也是一个进程有挂掉的可能，所以sentinel也会启动多个形成一个sentinel集群。

5.当主从模式配置密码时，sentinel也会同步将配置信息修改到配置文件中，不需要担心。

6.一个sentinel或sentinel集群可以管理多个主从Redis。

7.sentinel最好不要和Redis部署在同一台机器，不然Redis的服务器挂了以后，sentinel也挂了。

8.sentinel监控的Redis集群都会定义一个master名字，这个名字代表Redis集群的master Redis。

### Java启动连接

Set<String> sentinels = new HashSet<>(

Arrays.asList("ip:port", "ip:port", "ip:port"));

GenericObjectPoolConfig poolConfig = new GenericObjectPoolConfig();

//此处对poolConfig进行设置

JedisSentinelPool pool = new JedisSentinelPool("yiwangzhibujian", sentinels, poolConfig);

Jedis jedis = pool.getResource();

//使用jedis进行操作

//使用完直接close即可,会自动判断，若单个连接则关闭，在连接池内则归还

jedis.close();

## 集群模式

cluster的出现是为了解决单机Redis容量有限的问题，将Redis的数据根据一定的规则分配到多台机器。

### 键值分配

一个 Redis Cluster包含16384（0~16383）个哈希槽，存储在Redis Cluster中的所有键都会被映射到这些slot中，当需要在 Redis 集群中放置一个 key-value时，redis 先对 key 使用 crc16 算法算出一个结果，然后把结果对 16384 取模，这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽，redis 会根据节点数量大致均等的将哈希槽映射到不同的节点。这种结构很容易添加或者删除节点，并且无论是添加删除或者修改某一个节点，都不会造成集群不可用的状态。

### 新增节点

1、启动创建的新redis节点

2、添加新节点

redis-cli --cluster add-node 127.0.0.1:7006 127.0.0.1:7000  第一个ip:port 为需要添加的节点ip和端口，第二个ip:port为当前集群中的节点和端口

3、分配哈希槽

新添加的节点是没有哈希曹的，所以并不能正常存储数据，需要给新添加的节点分配哈希曹；redis-cli --cluster reshard ip:port

### 删除节点

1：如果删除的节点是主节点，这里我们删除192.168.1.160:7006节点，这个节点有1000个哈希槽

首先要把节点中的哈希槽转移到其他节点中，执行下面的命令

cd /usr/local/redis3.0/src

./redis-trib.rb reshard 192.168.1.160:7000

系统会提示我们要移动多少哈希槽，这里移动1000个，因为192.168.1.160:7006节点有1000个哈希槽

然后系统提示我们输入要接收这些哈希槽的节点的ID，这里使用192.168.1.160:7001的节点ID

然后要我们选择从那些节点中转出哈希槽，这里一定要输入192.168.1.160:7006这个节点的ID，最后输入 done  表示输入完毕

最后一步，使用下面的命令把这个节点删除

cd /usr/local/redis3.0/src/

./redis-trib.rb del-node 192.168.1.160:7000  需要删除的节点ID

2：如果节点是从节点的，直接使用下面的命令删除即可。

cd /usr/local/redis3.0/src/

./redis-trib.rb del-node 192.168.1.160:7000 需要删除的节点ID

### Java启动连接



# Redis事务

事务的实现特征：  
　　1). 在事务中的所有命令都将会被串行化的顺序执行，事务执行期间，Redis不会再为其它客户端的请求提供任何服务，从而保证了事物中的所有命令被原子的执行。  
　　2). 和关系型数据库中的事务相比，在Redis事务中如果有某一条命令执行失败，其后的命令仍然会被继续执行。  
　　3). 我们可以通过MULTI命令开启一个事务，有关系型数据库开发经验的人可以将其理解为"BEGIN TRANSACTION"语句。在该语句之后执行的命令都将被视为事务之内的操作，最后我们可以通过执行EXEC/DISCARD命令来提交/回滚该事务内的所有操作。这两个Redis命令可被视为等同于关系型数据库中的COMMIT/ROLLBACK语句。  
　　4). 在事务开启之前，如果客户端与服务器之间出现通讯故障并导致网络断开，其后所有待执行的语句都将不会被服务器执行。然而如果网络中断事件是发生在客户端执行EXEC命令之后，那么该事务中的所有命令都会被服务器执行。  
　　5). 当使用Append-Only模式时，Redis会通过调用系统函数write将该事务内的所有写操作在本次调用中全部写入磁盘。然而如果在写入的过程中出现系统崩溃，如电源故障导致的宕机，那么此时也许只有部分数据被写入到磁盘，而另外一部分数据却已经丢失。

Redis服务器会在重新启动时执行一系列必要的一致性检测，一旦发现类似问题，就会立即退出并给出相应的错误提示。此时，我们就要充分利用Redis工具包中提供的redis-check-aof工具，该工具可以帮助我们定位到数据不一致的错误，并将已经写入的部

分数据进行回滚。修复之后我们就可以再次重新启动Redis服务器了。

# Redis分布式锁实现

## 单实例redis（不考虑容错性）

### 目标

首先，为了确保分布式锁可用，我们至少要确保锁的实现同时满足以下四个条件：

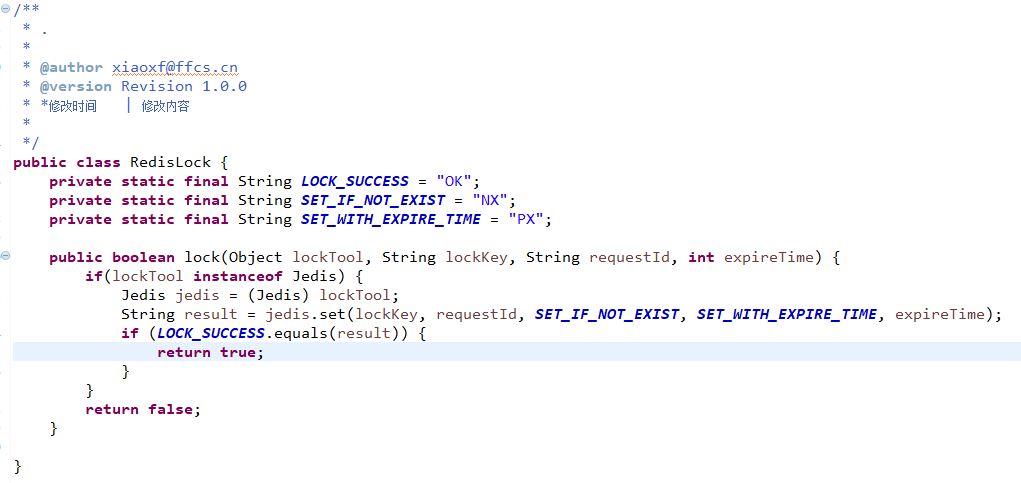
1、互斥性。在任意时刻，只有一个客户端能持有锁。

2、不会发生死锁。即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁。

3、具有容错性。只要大部分的Redis节点正常运行，客户端就可以加锁和解锁。

4、解铃还须系铃人。加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了。

### 加锁



我们加锁就一行代码：jedis.set(String key, String value, String nxxx, String expx, int time)，这个set()方法一共有五个形参：

第一个为key，我们使用key来当锁，因为key是唯一的。

第二个为value，我们传的是requestId，很多童鞋可能不明白，有key作为锁不就够了吗，为什么还要用到value？原因就是我们在上面讲到可靠性时，分布式锁要满足第四个条件解铃还须系铃人，通过给value赋值为requestId，我们就知道这把锁是哪个请求加的了，在解锁的时候就可以有依据。requestId可以使用UUID.randomUUID().toString()方法生成。

第三个为nxxx，这个参数我们填的是NX，意思是SET IF NOT EXIST，即当key不存在时，我们进行set操作；若key已经存在，则不做任何操作；

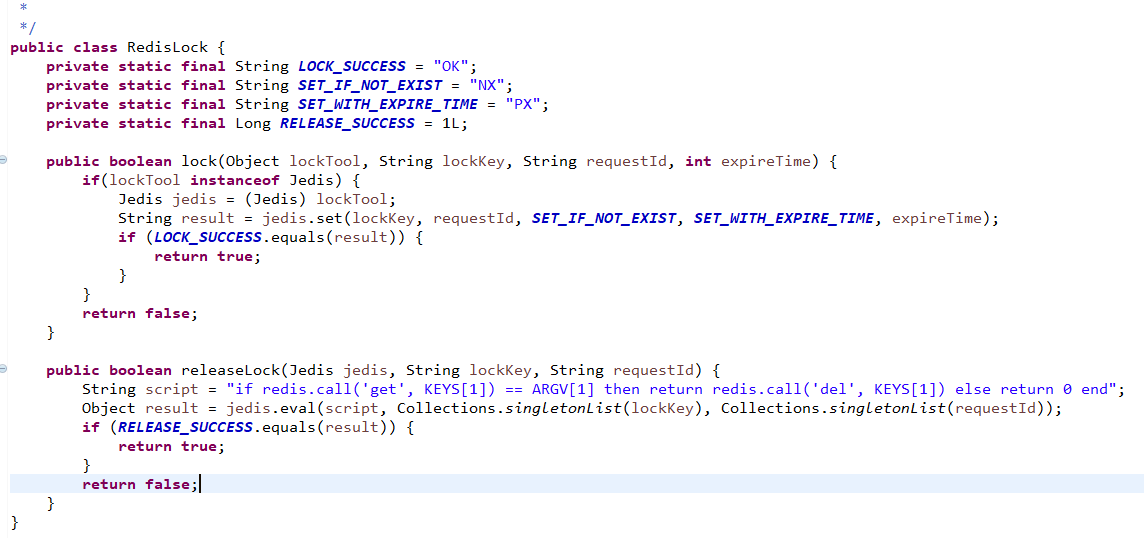
第四个为expx，这个参数我们传的是PX，意思是我们要给这个key加一个过期的设置，具体时间由第五个参数决定。

第五个为time，与第四个参数相呼应，代表key的过期时间。

总的来说，执行上面的set()方法就只会导致两种结果：1. 当前没有锁（key不存在），那么就进行加锁操作，并对锁设置个有效期，同时value表示加锁的客户端。2. 已有锁存在，不做任何操作。

心细的童鞋就会发现了，我们的加锁代码满足我们可靠性里描述的三个条件。首先，set()加入了NX参数，可以保证如果已有key存在，则函数不会调用成功，也就是只有一个客户端能持有锁，满足互斥性。其次，由于我们对锁设置了过期时间，即使锁的持有者后续发生崩溃而没有解锁，锁也会因为到了过期时间而自动解锁（即key被删除），不会发生死锁。最后，因为我们将value赋值为requestId，代表加锁的客户端请求标识，那么在客户端在解锁的时候就可以进行校验是否是同一个客户端。由于我们只考虑Redis单机部署的场景，所以容错性我们暂不考虑。

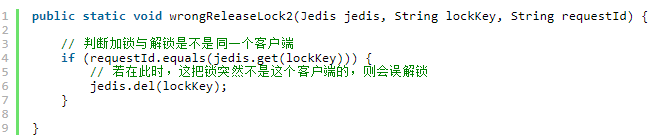
### 解锁



我们解锁只需要两行代码就搞定了！第一行代码，我们写了一个简单的Lua脚本代码，上一次见到这个编程语言还是在《[黑客与画家](http://www.amazon.cn/gp/product/B004WHZGZQ/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&tag=importnew-23&linkCode=as2&camp=536&creative=3200&creativeASIN=B004WHZGZQ)》里，没想到这次居然用上了。第二行代码，我们将Lua代码传到jedis.eval()方法里，并使参数KEYS[1]赋值为lockKey，ARGV[1]赋值为requestId。eval()方法是将Lua代码交给Redis服务端执行。

那么这段Lua代码的功能是什么呢？其实很简单，首先获取锁对应的value值，检查是否与requestId相等，如果相等则删除锁（解锁）。那么为什么要使用Lua语言来实现呢？因为要确保上述操作是原子性的。为什么判断加锁用户和解锁用户是同一个用户与del操作必须原子的呢？比如客户端A加锁，一段时间之后客户端A解锁，在执行jedis.del()之前，锁突然过期了，此时客户端B尝试加锁成功，然后客户端A再执行del()方法，则将客户端B的锁给解除了。

错误操作：



## Redisson

### 可重入锁

Redisson的分布式可重入锁RLock Java对象实现了java.util.concurrent.locks.Lock接口，同时还支持自动过期解锁。



### 公平锁

Redisson分布式可重入公平锁也是实现了java.util.concurrent.locks.Lock接口的一种RLock对象。在提供了自动过期解锁功能的同时，保证了当多个Redisson客户端线程同时请求加锁时，优先分配给先发出请求的线程。



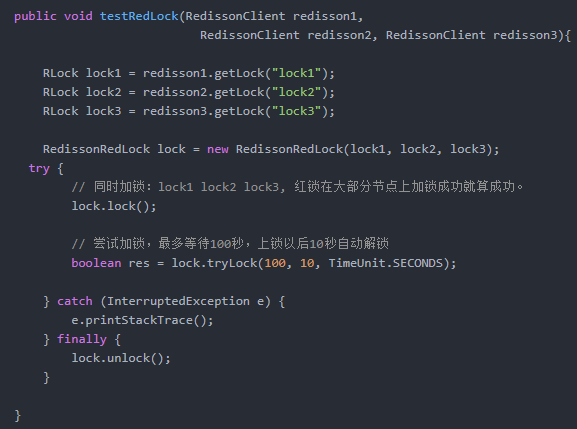
### 联锁

Redisson的RedissonMultiLock对象可以将多个RLock对象关联为一个联锁，每个RLock对象实例可以来自于不同的Redisson实例。



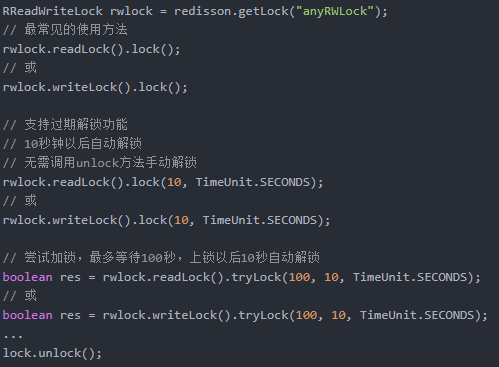
### 红锁

Redisson的RedissonRedLock对象实现了[Redlock](https://link.jianshu.com/?t=http://redis.cn/topics/distlock.html)介绍的加锁算法。该对象也可以用来将多个RLock对象关联为一个红锁，每个RLock对象实例可以来自于不同的Redisson实例。



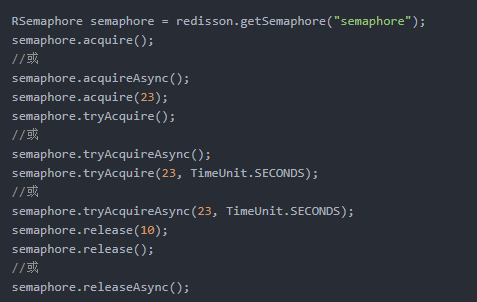
### 读写锁

Redisson的分布式可重入读写锁RReadWriteLock Java对象实现了java.util.concurrent.locks.ReadWriteLock接口。同时还支持自动过期解锁。该对象允许同时有多个读取锁，但是最多只能有一个写入锁。



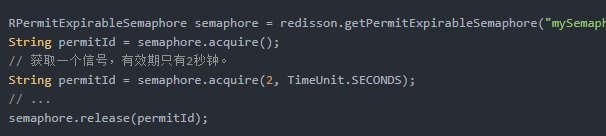
### 信号量

Redisson的分布式信号量（Semaphore）Java对象RSemaphore采用了与java.util.concurrent.Semaphore相似的接口和用法。



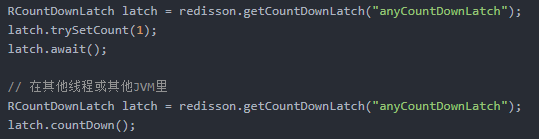
### 可过期信号量

Redisson的可过期性信号量（PermitExpirableSemaphore）实在RSemaphore对象的基础上，为每个信号增加了一个过期时间。每个信号可以通过独立的ID来辨识，释放时只能通过提交这个ID才能释放。



### 闭锁

Redisson的分布式闭锁（CountDownLatch）Java对象RCountDownLatch采用了与java.util.concurrent.CountDownLatch相似的接口和用法。



# Redis配置文件



# 运维

redis-cli -h <hostname> -p <port>

# [Redis 持久化之RDB和AOF](https://www.cnblogs.com/yiwangzhibujian/p/7067628.html)

Redis 持久化提供了多种不同级别的持久化方式:一种是RDB,另一种是AOF.

RDB 持久化可以在指定的时间间隔内生成数据集的时间点快照（point-in-time snapshot）。

AOF 持久化记录服务器执行的所有写操作命令，并在服务器启动时，通过重新执行这些命令来还原数据集。 AOF 文件中的命令全部以 Redis 协议的格式来保存，新命令会被追加到文件的末尾。 Redis 还可以在后台对 AOF 文件进行重写（rewrite），使得 AOF 文件的体积不会超出保存数据集状态所需的实际大小。Redis 还可以同时使用 AOF 持久化和 RDB 持久化。 在这种情况下， 当 Redis 重启时， 它会优先使用 AOF 文件来还原数据集， 因为 AOF 文件保存的数据集通常比 RDB 文件所保存的数据集更完整。你甚至可以关闭持久化功能，让数据只在服务器运行时存在。  
　　了解 RDB 持久化和 AOF 持久化之间的异同是非常重要的， 以下几个小节将详细地介绍这这两种持久化功能， 并对它们的相同和不同之处进行说明。

## RDB 的优点:

　　RDB 是一个非常紧凑（compact）的文件，它保存了 Redis 在某个时间点上的数据集。 这种文件非常适合用于进行备份： 比如说，你可以在最近的 24 小时内，每小时备份一次 RDB 文件，并且在每个月的每一天，也备份一个 RDB 文件。 这样的话，即使遇上问题，也可以随时将数据集还原到不同的版本。RDB 非常适用于灾难恢复（disaster recovery）：它只有一个文件，并且内容都非常紧凑，可以（在加密后）将它传送到别的数据中心，或者亚马逊 S3 中。RDB 可以最大化 Redis 的性能：父进程在保存 RDB 文件时唯一要做的就是 fork 出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无须执行任何磁盘 I/O 操作。RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。

## RDB 的缺点:

　　如果你需要尽量避免在服务器故障时丢失数据，那么 RDB 不适合你。 虽然 Redis 允许你设置不同的保存点（save point）来控制保存 RDB 文件的频率， 但是， 因为RDB 文件需要保存整个数据集的状态， 所以它并不是一个轻松的操作。 因此你可能会至少 5 分钟才保存一次 RDB 文件。 在这种情况下， 一旦发生故障停机， 你就可能会丢失好几分钟的数据。每次保存 RDB 的时候，Redis 都要 fork() 出一个子进程，并由子进程来进行实际的持久化工作。 在数据集比较庞大时， fork() 可能会非常耗时，造成服务器在某某毫秒内停止处理客户端； 如果数据集非常巨大，并且 CPU 时间非常紧张的话，那么这种停止时间甚至可能会长达整整一秒。 虽然 AOF 重写也需要进行 fork() ，但无论 AOF 重写的执行间隔有多长，数据的耐久性都不会有任何损失。

## AOF 的优点:

　　使用 AOF 持久化会让 Redis 变得非常耐久（much more durable）：你可以设置不同的 fsync 策略，比如无 fsync ，每秒钟一次 fsync ，或者每次执行写入命令时 fsync 。 AOF 的默认策略为每秒钟 fsync 一次，在这种配置下，Redis 仍然可以保持良好的性能，并且就算发生故障停机，也最多只会丢失一秒钟的数据（ fsync 会在后台线程执行，所以主线程可以继续努力地处理命令请求）。AOF 文件是一个只进行追加操作的日志文件（append only log）， 因此对 AOF 文件的写入不需要进行 seek ， 即使日志因为某些原因而包含了未写入完整的命令（比如写入时磁盘已满，写入中途停机，等等）， redis-check-aof 工具也可以轻易地修复这种问题。

Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写： 重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。 整个重写操作是绝对安全的，因为 Redis 在创建新 AOF 文件的过程中，会继续将命令追加到现有的 AOF 文件里面，即使重写过程中发生停机，现有的 AOF 文件也不会丢失。 而一旦新 AOF 文件创建完毕，Redis 就会从旧 AOF 文件切换到新 AOF 文件，并开始对新 AOF 文件进行追加操作。AOF 文件有序地保存了对数据库执行的所有写入操作， 这些写入操作以 Redis 协议的格式保存， 因此 AOF 文件的内容非常容易被人读懂， 对文件进行分析（parse）也很轻松。 导出（export） AOF 文件也非常简单： 举个例子， 如果你不小心执行了 FLUSHALL 命令， 但只要 AOF 文件未被重写， 那么只要停止服务器， 移除 AOF 文件末尾的 FLUSHALL 命令， 并重启 Redis ， 就可以将数据集恢复到 FLUSHALL 执行之前的状态。

## AOF 的缺点:

对于相同的数据集来说，AOF 文件的体积通常要大于 RDB 文件的体积。根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB 。 在一般情况下， 每秒 fsync 的性能依然非常高， 而关闭 fsync 可以让 AOF 的速度和 RDB 一样快， 即使在高负荷之下也是如此。 不过在处理巨大的写入载入时，RDB 可以提供更有保证的最大延迟时间（latency）。AOF 在过去曾经发生过这样的 bug ： 因为个别命令的原因，导致 AOF 文件在重新载入时，无法将数据集恢复成保存时的原样。 （举个例子，阻塞命令 BRPOPLPUSH 就曾经引起过这样的 bug 。） 测试套件里为这种情况添加了测试： 它们会自动生成随机的、复杂的数据集， 并通过重新载入这些数据来确保一切正常。 虽然这种 bug 在 AOF 文件中并不常见， 但是对比来说， RDB 几乎是不可能出现这种 bug 的。

## RDB 和 AOF ,我应该用哪一个？

一般来说,如果想达到足以媲美 PostgreSQL 的数据安全性， 你应该同时使用两种持久化功能。如果你非常关心你的数据,但仍然可以承受数分钟以内的数据丢失， 那么你可以只使用 RDB 持久化。有很多用户都只使用 AOF 持久化， 但我们并不推荐这种方式： 因为定时生成 RDB 快照（snapshot）非常便于进行数据库备份， 并且 RDB 恢复数据集的速度也要比 AOF 恢复的速度要快， 除此之外， 使用 RDB 还可以避免之前提到的 AOF 程序的 bug 。因为以上提到的种种原因， 未来我们可能会将 AOF 和 RDB 整合成单个持久化模型。 （这是一个长期计划。）

## RDB 快照:

　　在默认情况下， Redis 将数据库快照保存在名字为 dump.rdb 的二进制文件中。你可以对 Redis 进行设置， 让它在“ N 秒内数据集至少有 M 个改动”这一条件被满足时， 自动保存一次数据集。你也可以通过调用 SAVE 或者 BGSAVE ， 手动让 Redis 进行数据集保存操作。比如说， 以下设置会让 Redis 在满足“ 60 秒内有至少有 1000 个键被改动”这一条件时， 自动保存一次数据集：

save 60 1000

这种持久化方式被称为快照（snapshot）。

## 快照的运作方式:

当 Redis 需要保存 dump.rdb 文件时， 服务器执行以下操作：

1、Redis 调用 fork() ，同时拥有父进程和子进程。

2、子进程将数据集写入到一个临时 RDB 文件中。

3、当子进程完成对新 RDB 文件的写入时，Redis 用新 RDB 文件替换原来的 RDB 文件，并删除旧的 RDB 文件。

这种工作方式使得 Redis 可以从写时复制（copy-on-write）机制中获益。只进行追加操作的文件（append-only file，AOF）快照功能并不是非常耐久（durable）： 如果 Redis 因为某些原因而造成故障停机， 那么服务器将丢失最近写入、且仍未保存到快照中的那些数据。尽管对于某些程序来说， 数据的耐久性并不是最重要的考虑因素， 但是对于那些追求完全耐久能力（full durability）的程序来说， 快照功能就不太适用了。  
　　从 1.1 版本开始， Redis 增加了一种完全耐久的持久化方式： AOF 持久化。你可以通过修改配置文件来打开 AOF 功能：appendonly yes  
　　从现在开始， 每当 Redis 执行一个改变数据集的命令时（比如 SET）， 这个命令就会被追加到 AOF 文件的末尾。这样的话， 当 Redis 重新启时， 程序就可以通过重新执行 AOF 文件中的命令来达到重建数据集的目的。

## AOF 重写:

因为 AOF 的运作方式是不断地将命令追加到文件的末尾， 所以随着写入命令的不断增加， AOF 文件的体积也会变得越来越大。举个例子， 如果你对一个计数器调用了 100 次 INCR ， 那么仅仅是为了保存这个计数器的当前值， AOF 文件就需要使用 100 条记录（entry）。然而在实际上， 只使用一条 SET 命令已经足以保存计数器的当前值了， 其余 99 条记录实际上都是多余的。为了处理这种情况， Redis 支持一种有趣的特性： 可以在不打断服务客户端的情况下， 对 AOF 文件进行重建（rebuild）。执行 BGREWRITEAOF 命令， Redis 将生成一个新的 AOF 文件， 这个文件包含重建当前数据集所需的最少命令。

## AOF 有多耐久？

你可以配置 Redis 多久才将数据 fsync 到磁盘一次。有三个选项：

1、每次有新命令追加到 AOF 文件时就执行一次 fsync ：非常慢，也非常安全。

2、每秒 fsync 一次：足够快（和使用 RDB 持久化差不多），并且在故障时只会丢失 1 秒钟的数据。

3、从不 fsync ：将数据交给操作系统来处理。更快，也更不安全的选择。

推荐（并且也是默认）的措施为每秒 fsync 一次， 这种 fsync 策略可以兼顾速度和安全性。总是 fsync 的策略在实际使用中非常慢， 即使在 Redis 2.0 对相关的程序进行了改进之后仍是如此 —— 频繁调用 fsync 注定了这种策略不可能快得起来。

## 如果 AOF 文件出错了，怎么办？

服务器可能在程序正在对 AOF 文件进行写入时停机， 如果停机造成了 AOF 文件出错（corrupt）， 那么 Redis 在重启时会拒绝载入这个 AOF 文件， 从而确保数据的一致性不会被破坏。

当发生这种情况时， 可以用以下方法来修复出错的 AOF 文件：

1、为现有的 AOF 文件创建一个备份。

2、使用 Redis 附带的 redis-check-aof 程序，对原来的 AOF 文件进行修复。$ redis-check-aof --fix

3、（可选）使用 diff -u 对比修复后的 AOF 文件和原始 AOF 文件的备份，查看两个文件之间的不同之处。

4、重启 Redis 服务器，等待服务器载入修复后的 AOF 文件，并进行数据恢复。

## AOF 的运作方式

AOF 重写和 RDB 创建快照一样，都巧妙地利用了写时复制机制。以下是 AOF 重写的执行步骤:

1、Redis 执行 fork() ，现在同时拥有父进程和子进程。

2、子进程开始将新 AOF 文件的内容写入到临时文件。对于所有新执行的写入命令，父进程一边将它们累积到一个内存缓存中，一边将这些改动追加到现有 AOF 文件的末尾： 这样即使在重写的中途发生停机，现有的 AOF 文件也还是安全的。当子进程完成重写工作时，它给父进程发送一个信号，父进程在接收到信号之后，将内存缓存中的所有数据追加到新 AOF 文件的末尾。现在 Redis 原子地用新文件替换旧文件，之后所有命令都会直接追加到新 AOF 文件的末尾。

3、为最新的 dump.rdb 文件创建一个备份。将备份放到一个安全的地方。执行以下两条命令：

redis-cli> CONFIG SET appendonly yes

redis-cli> CONFIG SET save ""

确保命令执行之后，数据库的键的数量没有改变。

确保写命令会被正确地追加到 AOF 文件的末尾。  
　　步骤 3 执行的第一条命令开启了 AOF 功能： Redis 会阻塞直到初始 AOF 文件创建完成为止， 之后 Redis 会继续处理命令请求， 并开始将写入命令追加到 AOF 文件末尾。

步骤 3 执行的第二条命令用于关闭 RDB 功能。 这一步是可选的， 如果你愿意的话， 也可以同时使用 RDB 和 AOF 这两种持久化功能。  
　　别忘了在 redis.conf 中打开 AOF 功能！ 否则的话， 服务器重启之后， 之前通过 CONFIG SET 设置的配置就会被遗忘， 程序会按原来的配置来启动服务器。

## RDB 和 AOF 之间的相互作用

在版本号大于等于 2.4 的 Redis 中， BGSAVE 执行的过程中， 不可以执行 BGREWRITEAOF 。 反过来说， 在 BGREWRITEAOF 执行的过程中， 也不可以执行 BGSAVE 。

这可以防止两个 Redis 后台进程同时对磁盘进行大量的 I/O 操作。  
如果 BGSAVE 正在执行， 并且用户显示地调用 BGREWRITEAOF 命令， 那么服务器将向用户回复一个 OK 状态， 并告知用户， BGREWRITEAOF 已经被预定执行： 一旦 BGSAVE 执行完毕， BGREWRITEAOF 就会正式开始。当 Redis 启动时， 如果 RDB 持久化和 AOF 持久化都被打开了， 那么程序会优先使用 AOF 文件来恢复数据集， 因为 AOF 文件所保存的数据通常是最完整的。

## 备份 Redis 数据

Redis 对于数据备份是非常友好的， 因为你可以在服务器运行的时候对 RDB 文件进行复制： RDB 文件一旦被创建， 就不会进行任何修改。 当服务器要创建一个新的 RDB 文件时， 它先将文件的内容保存在一个临时文件里面， 当临时文件写入完毕时， 程序才使用  原子地用临时文件替换原来的 RDB 文件。这也就是说， 无论何时， 复制 RDB 文件都是绝对安全的。

**定时快照方式(snapshot)：**

该持久化方式实际是在Redis内部一个定时器事件，每隔固定时间去检查当前数据发生的改变次数与时间是否满足配置的持久化触发的条件，如果满足则通 过操作系统fork调用来创建出一个子进程，这个子进程默认会与父进程共享相同的地址空间，这时就可以通过子进程来遍历整个内存来进行存储操作，而主进程 则仍然可以提供服务，当有写入时由操作系统按照内存页(page)为单位来进行copy-on-write保证父子进程之间不会互相影响。

该持久化的主要缺点是定时快照只是代表一段时间内的内存映像，所以系统重启会丢失上次快照与重启之间所有的数据。

**基于语句追加方式(aof)：**

aof方式实际类似mysql的基于语句的binlog方式，即每条会使Redis内存数据发生改变的命令都会追加到一个log文件中，也就是说这个log文件就是Redis的持久化数据。

aof的方式的主要缺点是追加log文件可能导致体积过大，当系统重启恢复数据时如果是aof的方式则加载数据会非常慢，几十G的数据可能需要几小时才能加载完，当然这个耗时并不是因为磁盘文件读取速度慢，而是由于读取的所有命令都要在内存中执行一遍。另外由于每条命令都要写log,所以使用aof的方式，Redis的读写性能也会有所下降。

可以考虑将数据保存到不同的Redis实例中，每个实例的内存大小在2G左右，避免将鸡蛋放到一个篮子里，既可以减少缓存失效给系统带来的影响，又可以加快数据恢复的速度，不过同时也给系统设计带来了一定的复杂性。

# Redis回收策略

* 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

分析:这个问题其实相当重要，到底redis有没用到家，这个问题就可以看出来。比如你redis只能存5G数据，可是你写了10G，那会删5G的数据。怎么删的，这个问题思考过么？还有，你的数据已经设置了过期时间，但是时间到了，内存占用率还是比较高，有思考过原因么?

redis采用的是定期删除+惰性删除策略。

为什么不用定时删除策略?

定时删除,用一个定时器来负责监视key,过期则自动删除。虽然内存及时释放，但是十分消耗CPU资源。在大并发请求下，CPU要将时间应用在处理请求，而不是删除key,因此没有采用这一策略.

定期删除+惰性删除是如何工作的呢?

定期删除，redis默认每个100ms检查，是否有过期的key,有过期key则删除。需要说明的是，redis不是每个100ms将所有的key检查一次，而是随机抽取进行检查(如果每隔100ms,全部key进行检查，redis岂不是卡死)。因此，如果只采用定期删除策略，会导致很多key到时间没有删除。

于是，惰性删除派上用场。也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。

采用定期删除+惰性删除就没其他问题了么?

不是的，如果定期删除没删除key。然后你也没即时去请求key，也就是说惰性删除也没生效。这样，redis的内存会越来越高。那么就应该采用内存淘汰机制。

在redis.conf中有一行配置

# maxmemory-policy volatile-lru

该配置就是配内存淘汰策略的(什么，你没配过？好好反省一下自己)

1）noeviction：当内存不足以容纳新写入数据时，新写入操作会报错。应该没人用吧。

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。推荐使用，目前项目在用这种。

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。应该也没人用吧，你不删最少使用Key,去随机删。

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。这种情况一般是把redis既当缓存，又做持久化存储的时候才用。不推荐

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。依然不推荐

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。不推荐

ps：如果没有设置 expire 的key, 不满足先决条件(prerequisites); 那么 volatile-lru, volatile-random 和 volatile-ttl 策略的行为, 和 noeviction(不删除) 基本上一致。

# 面试总结

Redis有哪些数据结构？

字符串String、字典Hash、列表List、集合Set、有序集合SortedSet。

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：摁，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。

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，。

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。

Redis如何做持久化的？

bgsave做镜像全量持久化，aof做增量持久化。因为bgsave会耗费较长时间，不够实时，在停机的时候会导致大量丢失数据，所以需要aof来配合使用。在redis实例重启时，会使用bgsave持久化文件重新构建内存，再使用aof重放近期的操作指令来实现完整恢复重启之前的状态。

对方追问那如果突然机器掉电会怎样？取决于aof日志sync属性的配置，如果不要求性能，在每条写指令时都sync一下磁盘，就不会丢失数据。但是在高性能的要求下每次都sync是不现实的，一般都使用定时sync，比如1s1次，这个时候最多就会丢失1s的数据。

对方追问bgsave的原理是什么？你给出两个词汇就可以了，fork和cow。fork是指redis通过创建子进程来进行bgsave操作，cow指的是copy on write，子进程创建后，父子进程共享数据段，父进程继续提供读写服务，写脏的页面数据会逐渐和子进程分离开来。

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。

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。

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的I/O模型

非阻塞I/O，采用多路复用技术使得redis可以在单线程下处理多个请求