**Redis:**

**数据类型：**

key是字符串，value有5种数据类型，Stringhash,list,set,zset

通用操作：keys \*

**1、String类型**

增：set key value     例如：set  name  zhangsan  
​  
删：del key     例如：del  name  
​  
改：set  name  lisi  例如：set name lisi  
​  
查：get name     例如  get name  
​

**2、hash**

增：hset    key     subkey subvalue;设置一个键值对  
​  
增加多个键值对：hmset  key subkey1 subvalue1  subkey2 subvalue2......  
​  
删：hdel key  subkey ;删除一个子键  
​  
删除全部：del key   
​  
改：hset key  subkey  subvalue ;修改一个键值对  
​  
查：hget key subkey;获取一个子键的值  
​  
获取多个：hmget key subkey1  subkey2  
​  
获取所有属性以属性值:  hgetall key:获取指定key值的所有信息   
​

**3、list**

增：lpush key member1 member2.....   :从左边开始插入  
​  
rpush key member1 member2....;从右边开始插入  
​  
删：lpop  key :从左边弹出一个  
​  
rpop key :从左边弹出一个  
​  
改：lpush key member1 member2.....  ；在原先的基础上从左边开始追加  
​  
rpush key member1 member2....; 在原先的基础上从右边开始追加  
​  
查：lrange key 0 -1 :查询全部  
​  
lrange key startIndex endIndex;查看  
​

**4、set**

增：sadd key member1 member2 ;设置值  
​  
删：srem key member1 member2;删除值  
​  
改：先删除，再添加  
​  
查：smembers key ;查看值  
​

**5、zset**

增：zadd key score1 value1 score2 value2...: 设置添加  
​  
删：zrem key value1 value2... :删除指定成员  
​  
改：先删后加  
​  
查：zcard key :展示元素的长度  
​  
zrange key 0 -1 ;查询全部的值  
​  
zscore key value :获取成员的数字  
​  
zcard key :展示元素的长度  
​

**扩展：**

keys \*:查询所有的key  
exists key:判断是否有指定的key 若有返回1,否则返回0  
rename key 新key:重命名  
type key:判断一个key的类型  
del key :删除  
select index:切换库  
move key 指定数据库: 将当前库的数据移动到指定库中  
dbsize: 返回当前库中有多少个key  
flushdb:清空当前数据库数据  
flushall:清空当前实例下所有的数据库数据

**面试题：**

**什么是Redis持久化？Redis有哪几种持久化方式？优缺点是什么？**

**1、快照（snapshots）**

　　缺省情况情况下，Redis把数据快照存放在磁盘上的二进制文件中，文件名为dump.rdb。你可以配置Redis的持久化策略，例如数据集中每N秒钟有超过M次更新，就将数据写入磁盘；或者你可以手工调用命令SAVE或BGSAVE。

　　工作原理

　　． Redis forks.

　　． 子进程开始将数据写到临时RDB文件中。

　　． 当子进程完成写RDB文件，用新文件替换老文件。

　　． 这种方式可以使Redis使用copy-on-write技术。

**2、AOF**

　　快照模式并不十分健壮，当系统停止，或者无意中Redis被kill掉，最后写入Redis的数据就会丢失。这对某些应用也许不是大问题，但对于要求高可靠性的应用来说，

　　Redis就不是一个合适的选择。

　　Append-only文件模式是另一种选择。

　　你可以在配置文件中打开AOF模式

redis提供了两种持久化的方式，分别是RDB（Redis DataBase）和AOF（Append Only File）。  
RDB，简而言之，就是在不同的时间点，将redis存储的数据生成快照并存储到磁盘等介质上；  
AOF，则是换了一个角度来实现持久化，那就是将redis执行过的所有写指令记录下来，在下次redis重新启动时，只要把这些写指令从前到后再重复执行一遍，就可以实现数据恢复了。  
PS: RDB和AOF两种方式也可以同时使用，在这种情况下，如果redis重启的话，则会优先采用AOF方式来进行数据恢复，这是因为AOF方式的数据恢复完整度更高。  
​  
​

**RDB方式：**

**AOF:**

**Redis主要有哪些功能？**

**哨兵（Sentinel）和复制（Replication）**

Redis服务器毫无征兆的罢工是个麻烦事，如何保证备份的机器是原始服务器的完整备份呢？这时候就需要哨兵和复制。

Sentinel可以管理多个Redis服务器，它提供了监控，提醒以及自动的故障转移的功能，Replication则是负责让一个Redis服务器可以配备多个备份的服务器。

Redis也是利用这两个功能来保证Redis的高可用的

**事务**

很多情况下我们需要一次执行不止一个命令，而且需要其同时成功或者失败。redis对事务的支持也是源自于这部分需求，即支持一次性按顺序执行多个命令的能力，并保证其原子性。

**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。

**Redis为什么是单线程的？**

多线程处理会涉及到锁，而且多线程处理会涉及到线程切换而消耗CPU。因为CPU不是Redis的瓶颈，Redis的瓶颈最有可能是机器内存或者网络带宽。单线程无法发挥多核CPU性能，不过可以通过在单机开多个Redis实例来解决。

**使用Redis的优势？**

1.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)

\2. 支持丰富数据类型，支持string，list，set，sorted set，hash

3.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

\4. 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单点吞吐量**

单点TPS达到8万/秒，QPS达到10万/秒，补充下TPS和QPS的概念

**1.QPS: 应用系统每秒钟最大能接受的用户访问量**

每秒钟处理完请求的次数，注意这里是处理完，具体是指发出请求到服务器处理完成功返回结果。可以理解在server中有个counter，每处理一个请求加1，1秒后counter=QPS。

**2.TPS： 每秒钟最大能处理的请求数**

每秒钟处理完的事务次数，一个应用系统1s能完成多少事务处理，一个事务在分布式处理中，可能会对应多个请求，对于衡量单个接口服务的处理能力，用QPS比较合理。

**Redis相比memcached有哪些优势？**

1.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型

2.Redis的速度比memcached快很多

3.Redis可以持久化其数据

4.Redis支持数据的备份，即master-slave模式的数据备份。

**Redis有哪几种数据淘汰策略？**

在Redis中，允许用户设置最大使用内存大小server.maxmemory，当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。

1.volatile-lru:从已设置过期的数据集中挑选最近最少使用的淘汰

2.volatile-ttr:从已设置过期的数据集中挑选将要过期的数据淘汰

3.volatile-random:从已设置过期的数据集中任意挑选数据淘汰

4.allkeys-lru:从数据集中挑选最近最少使用的数据淘汰

5.allkeys-random:从数据集中任意挑选数据淘汰

6.noenviction:禁止淘汰数据

redis淘汰数据时还会同步到aof

**Redis集群方案应该怎么做？都有哪些方案？**

1.twemproxy

2.codis，目前用的最多的集群方案，基本和twemproxy一致的效果，但它支持在 节点数量改变情况下，旧节点数据可恢复到新hash节点。

3.Redis cluster3.0自带的集，特点在于他的分布式算法不是一致性hash，而是hash槽的概念，以及自身支持节点设置从节点。

**Redis读写分离模型**

通过增加Slave DB的数量，读的性能可以线性增长。为了避免Master DB的单点故障，集群一般都会采用两台Master DB做双机热备，所以整个集群的读和写的可用性都非常高。

读写分离架构的缺陷在于，不管是Master还是Slave，每个节点都必须保存完整的数据，如果在数据量很大的情况下，集群的扩展能力还是受限于单个节点的存储能力，而且对于Write-intensive类型的应用，读写分离架构并不适合。

**Redis数据分片模型**

为了解决读写分离模型的缺陷，可以将数据分片模型应用进来。

可以将每个节点看成都是独立的master，然后通过业务实现数据分片。

结合上面两种模型，可以将每个master设计成由一个master和多个slave组成的模型。

**如何选择合适的持久化方式？**

\1. Redis主要提供了两种持久化机制：**RDB和AOF；**

**2.RDB**

默认开启，会按照配置的指定时间将内存中的数据快照到磁盘中，创建一个dump.rdb文件，Redis启动时再恢复到内存中。

Redis会单独创建fork()一个子进程，将当前父进程的数据库数据复制到子进程的内存中，然后由子进程写入到临时文件中，持久化的过程结束了，再用这个临时文件替换上次的快照文件，然后子进程退出，内存释放。

需要注意的是，每次快照持久化都会将主进程的数据库数据复制一遍，导致内存开销加倍，若此时内存不足，则会阻塞服务器运行，直到复制结束释放内存；都会将内存数据完整写入磁盘一次，所以如果数据量大的话，而且写操作频繁，必然会引起大量的磁盘I/O操作，严重影响性能，并且最后一次持久化后的数据可能会丢失；

**3.AOF**

以日志的形式记录每个写操作（读操作不记录），只需追加文件但不可以改写文件，Redis启动时会根据日志从头到尾全部执行一遍以完成数据的恢复工作。包括flushDB也会执行。

主要有两种方式触发：有写操作就写、每秒定时写（也会丢数据）。

因为AOF采用追加的方式，所以文件会越来越大，针对这个问题，新增了重写机制，就是当日志文件大到一定程度的时候，会fork出一条新进程来遍历进程内存中的数据，每条记录对应一条set语句，写到临时文件中，然后再替换到旧的日志文件（类似rdb的操作方式）。默认触发是当aof文件大小是上次重写后大小的一倍且文件大于64M时触发。

当两种方式同时开启时，数据恢复Redis会优先选择AOF恢复。一般情况下，只要使用默认开启的RDB即可，因为相对于AOF，RDB便于进行数据库备份，并且恢复数据集的速度也要快很多。

开启持久化缓存机制，对性能会有一定的影响，特别是当设置的内存满了的时候，更是下降到几百reqs/s。所以如果只是用来做缓存的话，可以关掉持久化。

**Redis常见性能问题和解决方案？**

(1) Master最好不要做任何持久化工作，如RDB内存快照和AOF日志文件

(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次

(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内

(4) 尽量避免在压力很大的主库上增加从库

(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。

**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，当需要在 Redis 集群中放置一个 key-value 时，根据 CRC16(key) mod 16384的值，决定将一个key放到哪个桶中。

**Redis集群最大节点个数是多少？**

Redis集群预分好16384个桶(哈希槽)

**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。

**Redis回收使用的是什么算法？**

LRU算法

**Redis有哪些适合的场景？**

1）Session共享(单点登录)

2）页面缓存

3）队列

4）排行榜/计数器

5）发布/订阅

**缓存雪崩、缓存穿透、缓存预热、缓存更新、缓存降级：**

**一、缓存雪崩**

缓存雪崩我们可以简单的理解为：由于原有缓存失效，新缓存未到期间(例如：我们设置缓存时采用了相同的过期时间，在同一时刻出现大面积的缓存过期)，所有原本应该访问缓存的请求都去查询数据库了，而对数据库CPU和内存造成巨大压力，严重的会造成数据库宕机。从而形成一系列连锁反应，造成整个系统崩溃。

**缓存正常从Redis中获取，示意图如下：**

**缓存失效瞬间示意图如下：**

缓存失效时的雪崩效应对底层系统的冲击非常可怕！大多数系统设计者考虑用加锁或者队列的方式保证来保证不会有大量的线程对数据库一次性进行读写，从而避免失效时大量的并发请求落到底层存储系统上。还有一个简单方案就时讲缓存失效时间分散开，比如我们可以在原有的失效时间基础上增加一个随机值，比如1-5分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

以下简单介绍两种实现方式的伪代码：

（1）碰到这种情况，一般并发量不是特别多的时候，使用最多的解决方案是加锁排队，伪代码如下：

//伪代码  
public object GetProductListNew() {  
   int cacheTime = 30;  
   String cacheKey = "product\_list";  
   String lockKey = cacheKey;  
​  
   String cacheValue = CacheHelper.get(cacheKey);  
   if (cacheValue != null) {  
       return cacheValue;  
  } else {  
       synchronized(lockKey) {  
           cacheValue = CacheHelper.get(cacheKey);  
           if (cacheValue != null) {  
               return cacheValue;  
          } else {  
           //这里一般是sql查询数据  
               cacheValue = GetProductListFromDB();   
               CacheHelper.Add(cacheKey, cacheValue, cacheTime);  
          }  
      }  
       return cacheValue;  
  }  
}  
​

加锁排队只是为了减轻数据库的压力，并没有提高系统吞吐量。假设在高并发下，缓存重建期间key是锁着的，这是过来1000个请求999个都在阻塞的。同样会导致用户等待超时，这是个治标不治本的方法！

注意：加锁排队的解决方式分布式环境的并发问题，有可能还要解决分布式锁的问题；线程还会被阻塞，用户体验很差！因此，在真正的高并发场景下很少使用！

（2）还有一个解决办法解决方案是：给每一个缓存数据增加相应的缓存标记，记录缓存的是否失效，如果缓存标记失效，则更新数据缓存，实例伪代码如下：

//伪代码

public object GetProductListNew() {

int cacheTime = 30;

String cacheKey = "product\_list";

//缓存标记

String cacheSign = cacheKey + "\_sign";

String sign = CacheHelper.Get(cacheSign);

//获取缓存值

String cacheValue = CacheHelper.Get(cacheKey);

if (sign != null) {

return cacheValue; //未过期，直接返回

} else {

CacheHelper.Add(cacheSign, "1", cacheTime);

ThreadPool.QueueUserWorkItem((arg) -> {

//这里一般是 sql查询数据

cacheValue = GetProductListFromDB();

//日期设缓存时间的2倍，用于脏读

CacheHelper.Add(cacheKey, cacheValue, cacheTime \* 2);

});

return cacheValue;

}

}

解释说明：

1、缓存标记：记录缓存数据是否过期，如果过期会触发通知另外的线程在后台去更新实际key的缓存；

2、缓存数据：它的过期时间比缓存标记的时间延长1倍，例：标记缓存时间30分钟，数据缓存设置为60分钟。 这样，当缓存标记key过期后，实际缓存还能把旧数据返回给调用端，直到另外的线程在后台更新完成后，才会返回新缓存。

关于缓存崩溃的解决方法，这里提出了三种方案：使用锁或队列、设置过期标志更新缓存、为key设置不同的缓存失效时间，还有一各被称为“二级缓存”的解决方法，有兴趣的读者可以自行研究。

**二、缓存穿透**

缓存穿透是指用户查询数据，在数据库没有，自然在缓存中也不会有。这样就导致用户查询的时候，在缓存中找不到，每次都要去数据库再查询一遍，然后返回空（相当于进行了两次无用的查询）。这样请求就绕过缓存直接查数据库，这也是经常提的缓存命中率问题。

有很多种方法可以有效地解决缓存穿透问题，最常见的则是采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。

另外也有一个更为简单粗暴的方法，如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。通过这个直接设置的默认值存放到缓存，这样第二次到缓冲中获取就有值了，而不会继续访问数据库，这种办法最简单粗暴！

//伪代码

public object GetProductListNew() {

int cacheTime = 30;

String cacheKey = "product\_list";

String cacheValue = CacheHelper.Get(cacheKey);

if (cacheValue != null) {

return cacheValue;

}

cacheValue = CacheHelper.Get(cacheKey);

if (cacheValue != null) {

return cacheValue;

} else {

//数据库查询不到，为空

cacheValue = GetProductListFromDB();

if (cacheValue == null) {

//如果发现为空，设置个默认值，也缓存起来

cacheValue = string.Empty;

}

CacheHelper.Add(cacheKey, cacheValue, cacheTime);

return cacheValue;

}

}

把空结果，也给缓存起来，这样下次同样的请求就可以直接返回空了，即可以避免当查询的值为空时引起的缓存穿透。同时也可以单独设置个缓存区域存储空值，对要查询的key进行预先校验，然后再放行给后面的正常缓存处理逻辑。

**三、缓存预热**

缓存预热这个应该是一个比较常见的概念，相信很多小伙伴都应该可以很容易的理解，缓存预热就是系统上线后，将相关的缓存数据直接加载到缓存系统。这样就可以避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据！

解决思路：

1、直接写个缓存刷新页面，上线时手工操作下；

2、数据量不大，可以在项目启动的时候自动进行加载；

3、定时刷新缓存；

**四、缓存更新**

除了缓存服务器自带的缓存失效策略之外（Redis默认的有6中策略可供选择），我们还可以根据具体的业务需求进行自定义的缓存淘汰，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。

**五、缓存降级**

当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

降级的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的（如加入购物车、结算）。

在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅；从而梳理出哪些必须誓死保护，哪些可降级；比如可以参考日志级别设置预案：

（1）一般：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级；

（2）警告：有些服务在一段时间内成功率有波动（如在95~100%之间），可以自动降级或人工降级，并发送告警；

（3）错误：比如可用率低于90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承受的最大阀值，此时可以根据情况自动降级或者人工降级；

（4）严重错误：比如因为特殊原因数据错误了，此时需要紧急人工降级。

**六、总结**

这些都是实际项目中，可能碰到的一些问题，也是面试的时候经常会被问到的知识点，实际上还有很多很多各种各样的问题，文中的解决方案，也不可能满足所有的场景，相对来说只是对该问题的入门解决方法。一般正式的业务场景往往要复杂的多，应用场景不同，方法和解决方案也不同，由于上述方案，考虑的问题并不是很全面，因此并不适用于正式的项目开发，但是可以作为概念理解入门，具体解决方案要根据实际情况来确定

**（1）、**

**什么是Redis持久化？Redis有哪几种持久化方式？优缺点是什么？**

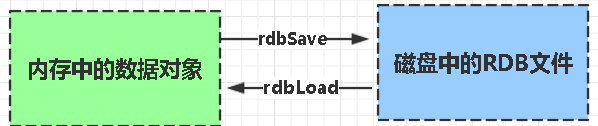
持久化就是把内存的数据写到磁盘中去，防止服务宕机了内存数据丢失。

Redis 提供了两种持久化方式:RDB（默认） 和AOF

**RDB：**

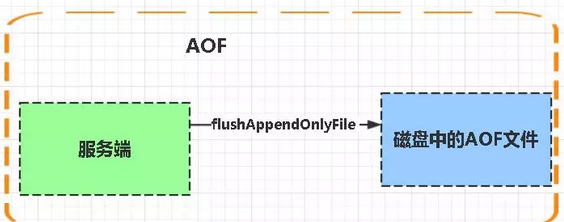
rdb是Redis DataBase缩写

功能核心函数rdbSave(生成RDB文件)和rdbLoad（从文件加载内存）两个函数



**AOF:**

Aof是Append-only file缩写



每当执行服务器(定时)任务或者函数时flushAppendOnlyFile 函数都会被调用， 这个函数执行以下两个工作

aof写入保存：

WRITE：根据条件，将 aof\_buf 中的缓存写入到 AOF 文件

SAVE：根据条件，调用 fsync 或 fdatasync 函数，将 AOF 文件保存到磁盘中。

**存储结构:**

内容是redis通讯协议(RESP )格式的命令文本存储。

**比较**：

1、aof文件比rdb更新频率高，优先使用aof还原数据。

2、aof比rdb更安全也更大

3、rdb性能比aof好

4、如果两个都配了优先加载AOF

**刚刚上面你有提到redis通讯协议(RESP )，能解释下什么是RESP？有什么特点？（可以看到很多面试其实都是连环炮，面试官其实在等着你回答到这个点，如果你答上了对你的评价就又加了一分）**

**Redis的协议规范是 Redis Serialization Protocol (Redis序列化协议)**

**RESP 是redis客户端和服务端之前使用的一种通讯协议；**

该协议是用于与Redis服务器通信的，用的较多的是Redis-cli通过pipe与Redis服务器联系；

协议如下：

客户端以规定格式的形式发送命令给服务器；

服务器在执行最后一条命令后，返回结果。

**RESP 的特点：实现简单、快速解析、可读性好**

Redis在TCP端口6379上监听到来的连接，客户端连接到来时，Redis服务器为此创建一个TCP连接。在客户端与服务器端之间传输的每个Redis命令或者数据都以\r\n结尾。

Redis接收由不同参数组成的命令。一旦收到命令，将会立刻被处理，并回复给客户端。

**客户端和服务器端发送命令进行通信的格式(类型)：5种类型**

**Simple Strings**

**状态回复（或者单行回复）以“+”开始以“\r\n”结尾的单行字符串形式**。例如：

"+OK\r\n"

客户端库将在“+”后面返回所有数据，正如上例中字符串“OK”一样。

**Errors**

**错误回复发送类似于状态回复。唯一的不同是第一个字节用“-”代替“+**”。

错误回复仅仅在一些意料之外的事情发生时发送，例如：如果你试图执行一个操作来应付错误的数据类型，或者如果命令不存在等等。所以当收到一个错误回复时，客户端将会出现一个异常。

eg: "-Error unknow command 'foobar'\r\n"

**Integers**

这种回复类型只是用CRLF结尾字符串来表示整型，用一个字节的“：”作为前缀。例如：“：0\r\n”，或者“:1000\r\n”是整型回复。

eg: ":1000\r\n"

**Bulk Strings**

批量回复被服务器用于返回一个单二进制安全字符串。

C: GET mykey

S: $6\r\nfoobar\r\n

服务器发送第一行回复，该行以“$”开始后面跟随实际要发送的字节数，随后是CRLF，然后发送实际数据，随后是2个字节的额外数据用于最后的CRLF。服务器发送的准确序列如下：

"$6\r\nfoobar\r\n"

如果请求的值不存在，批量回复将使用特殊的值-1来作为数据长度，例如：

C: GET nonexistingkey

S: $-1 //代表的是nil

当请求的对象不存在时，客户端库API不会返回空字符串，而会返回空对象。例如：Ruby库返回‘nil’，而C库返回NULL（或者在回复的对象里设置指定的标志）等等。

**Arrays**

像命令LRNGE需要返回多个值（列表的每个元素是一个值，而LRANGE需要返回多于一个单元素）。使用多批量写是有技巧的，用一个初始行作为前缀来指示多少个批量写紧随其后。批量回复的第一个字节总是\*，例如：

C: LRANGE mylist 0 3

s: \*4

s: $3

s: foo

s: $3

s: bar

s: $5

s: Hello

s: $5

s: World

正如您可以看到的多批量回复是以完全相同的格式使用Redis统一协议将命令发送给服务器。

服务器发送的第一行是\*4\r\n，用于指定紧随着4个批量回复。然后传送每个批量写。

**如果指定的键不存在，则该键被认为是持有一个空的列表，且数值0被当作多批量计数值来发送**，例如：

C: LRANGE nokey 0 1

S: \*0

**当BLPOP命令超时时，它返回nil多批量回复。这种类型多批量回复的计数器是-1，且值被当作nil来解释。**例如：

C: BLPOP key 1

S: \*-1

当这种情况发生时，客户端库API将返回空nil对象，且不是一个空列表。这必须有别于空列表和错误条件（例如：BLPOP命令的超时条件）。

**多批量回复中的Nil元素**

多批量回复的单元素长度可能是-1，为了发出信号这个元素被丢失且不是空字符串。这种情况发送在SORT命令时，此时使用GET模式选项且指定的键丢失。一个多批量回复包含一个空元素的例子如下：

S: \*3

S: $3

S: foo

S: $-1

S: $3

S: bar

第二个元素是空。客户端库返回如下:

["foo",nil,"bar"]

**请求：**

客户端发出的请求：set mykey myvalue

\*3

$3

SET

$5

mykey

$7

myvalue

上面的命令看上去像是单引号字符串，所以可以在查询中看到每个字节的准确值：

"\*3\r\n$3\r\nSET\r\n$5\r\nmykey\r\n$7\r\nmyvalue\r\n"

在Redis的回复中也使用这样的格式,

**回复：**

Redis用不同的回复类型回复命令。它可能从服务器发送的第一个字节开始校验回复类型：

* 用单行回复，回复的第一个字节将是“+”
* 错误消息，回复的第一个字节将是“-”
* 整型数字，回复的第一个字节将是“:”
* 批量回复，回复的第一个字节将是“$”
* 多个批量回复，回复的第一个字节将是“\*”

**Redis管道技术:**

在我们Redis中，客户端和Redis使用的是一个TCP协议进行连接，不论是客户端向Redis发送命令，还是Redis向客户端返回命令的一个执行结果，都需进行网络传输。这两个部分的总耗时我们就称为“往返时延”，就是发送命令和命令的一个返回结果这两个总共的耗时。

在执行多个命令时，每个命令都需要等待上一条命令执行完成，并且收到Redis返回结果之后才能执行。Redis中的命令都是串行执行的。就算第二条命令不需要上一条命令的执行结果，它也需要等待第一条命令执行完成之后才能执行下一条。

现在，Redis底层通过这个协议对管道提供了一个支持，管道就是可以一次性发送多个命令，并且在执行完成后一次性将管道中的这些命令这些结果返回，所以当一组命令中，每一条命令都不依赖于我们之前的命令的时候，我们就可以考虑用管道一起来发出。接着减少客户端与Redis的通讯次数，也能降低我们“往返时延”这样的一个效果。

**切换数据库：**

SELECT:选择数据库（数据库索引默认是从0开始的）

SELECT 9（选择数据库9，我们可以看到提示符就变成了127.0.0.1:6379[9]>

**Redis排序：**

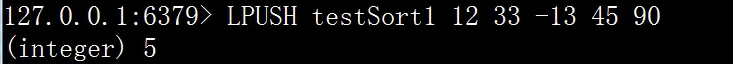
按照键值从小到大或者从大到小的顺序排序

SORT key（默认是升序）

SORT key DESC（降序）

**对数值进行排序**

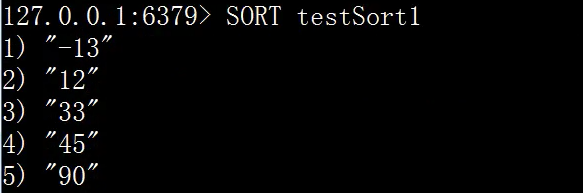
LPUSH testSort1 12 33 -13 45 90(创建一个新列表testSort1，并向里面添加12、33、-13、45、90这五个元素。）



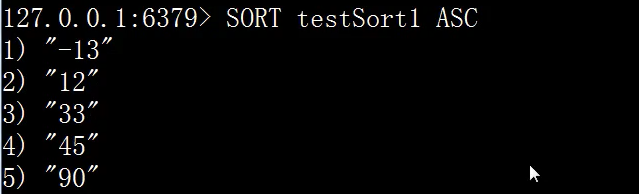
LRANGE testSort1 0 -1（获取列表testSort1的所有元素，我们可以看到返回的元素并没有按照从大到小或者是从小到大的顺序排列。）



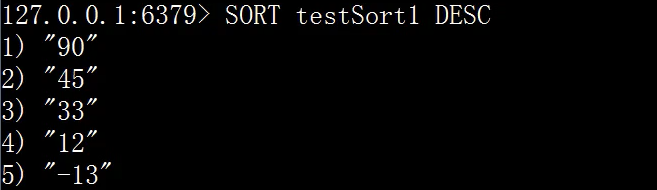
SORT testSort1 (返回的元素就变成了一个升序的效果,默认是升序)



SORT testSort1 ASC(返回的元素也是一个升序的效果)

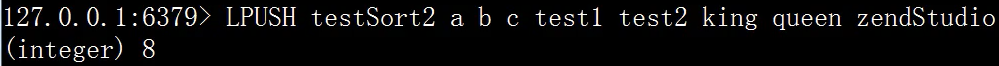


SORT testSort1 DESC( 返回的元素就变成了一个降序的效果)

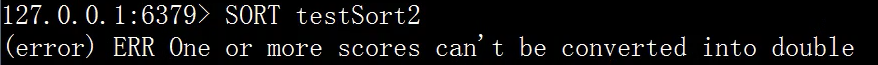


**对字母进行排序**

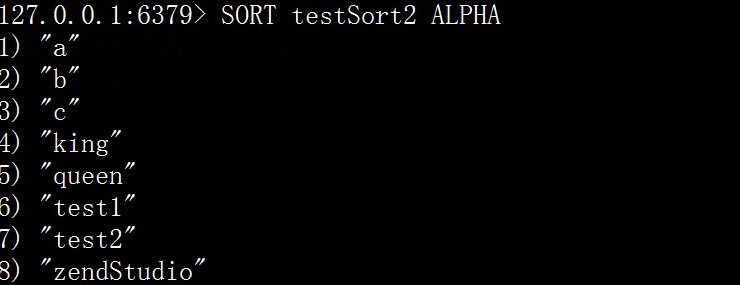
LPUSH testSort2 a b c test1 test2 king queen zendStudio（再创建一个新列表testSort2 ，并向里面添加a、b、c test1、test2 、king 、queen 、zendStudio这八个元素。）



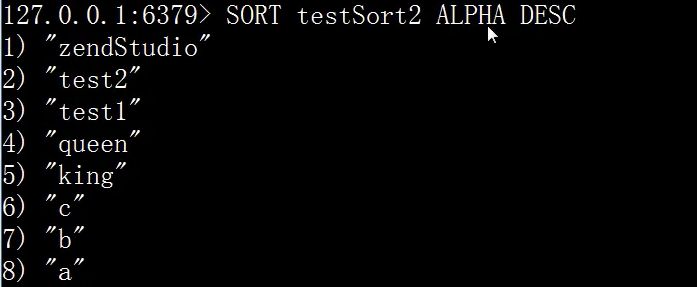
SORT testSort2（这时候就会返回一个错误，表示它不能转成数字再来进行排序）



SORT testSort2 ALPHA（将字母或字符串按ASCALL码从小到大来排序，默认升序）



SORT testSort2 ALPHA DESC(将字母或字符串按ASCALL码从大到小来排序）



**注意：对于汉字，我们redis只能识别UTF-8它也是按照我们这个ASCALL来排序**

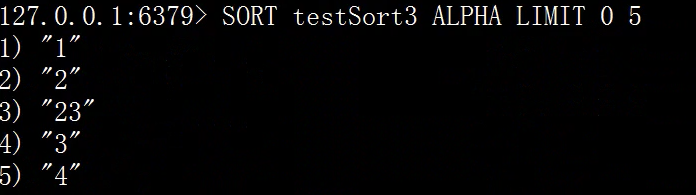
**使用LIMIT限制返回结果**

**注意：LIMIT 后的两个参数，第一个代表偏移量，第二个代表你要返回的值的个数是多少。**

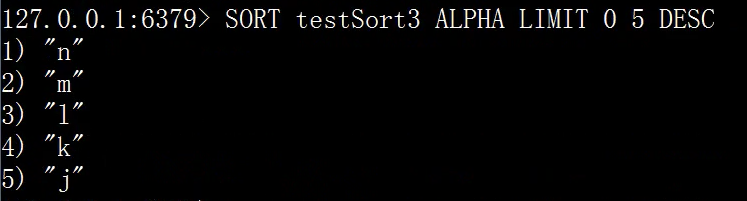
LPUSH testSort3 a b c d e f g h i j k l m n 1 2 3 4 6 8 900 23（创建一个新列表testSort3 ，并一次性向里面添加a、b、c、d、e、f、g、h、i、j、k、l、m、n、1、2、3、4、6、8、900、23这22个元素。）



SORT testSort3 ALPHA LIMIT 0 5（返回前5个结果）



SORT testSort3 ALPHA LIMIT 0 5 DESC（返回后5个结果并按降序排列）

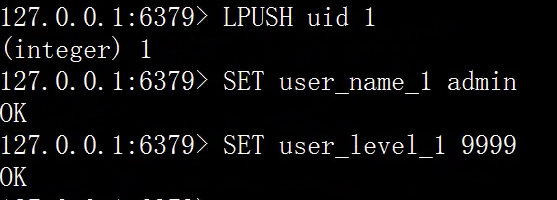


**根据外部key的权重进行排序**

LPUSH uid 1

SET user\_name\_1 admin

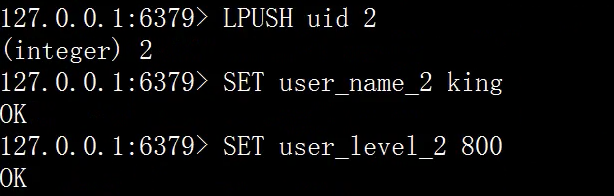
SET user\_level\_1 9999



LPUSH uid 2

SET user\_name\_2 king

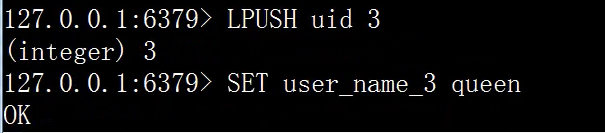
SET user\_level\_2 800



LPUSH uid 3

SET user\_name\_3 queen

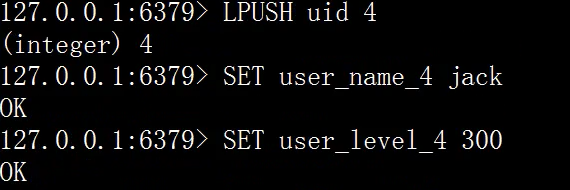
SET user\_level\_3 600



LPUSH uid 4

SET user\_name\_4 jack

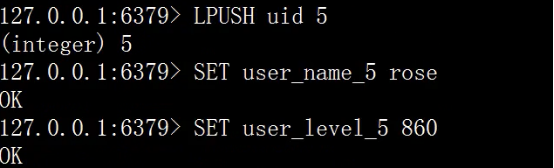
SET user\_level\_4 300



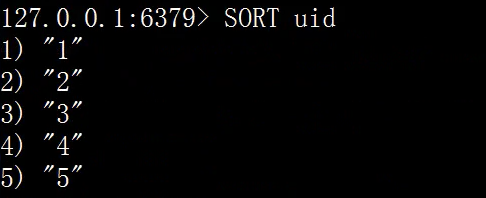
LPUSH uid 5

SET user\_name\_5 rose

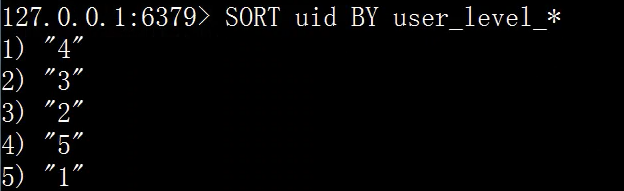
SET user\_level\_5 860



SORT uid （默认是按照uid从小到大的顺序进行排列）



SORT uid BY user\_level*\*（按照一个外部的权重，根据user\_level*\*也就是用户的级别来排序。）



**Redis发布订阅模式：**

**发布订阅模式 ：**

通过PUBLISH/SUBSCRIBE命令来实现发布和订阅模式

**发布订阅模式的原理 ：**

包含两个角色，一个是发布者,一个是订阅者

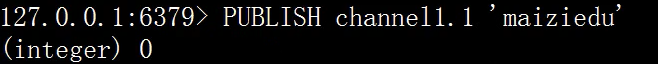
订阅者可以订阅一个或者多个频道channel

发布者可以向指定频道发布信息

**通过PUBLISH发布消息：**

PUBLISH channel message

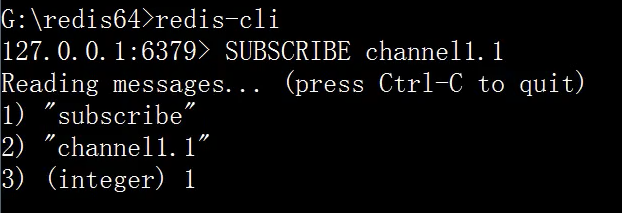
PUBLISH channel1.1 'maiziedu'（指定发布消息channel1.1频道发布信息'maiziedu'，返回的值代表客户端订阅该频道的数量）



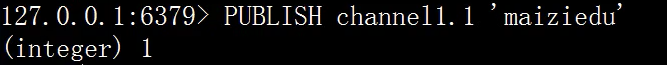
**通过SUBSCRIBE命令订阅频道：**

SUBSCRIBE channel ...

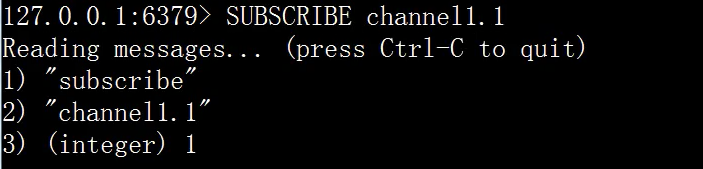
打开另一个客户端来订阅channel1.1这个频道



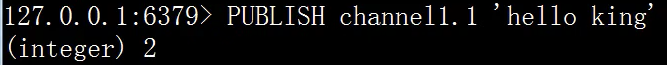
此时，再通过PUBLISH发布消息，返回1表示已经有一个客户端订阅这个channel1.1的频道。



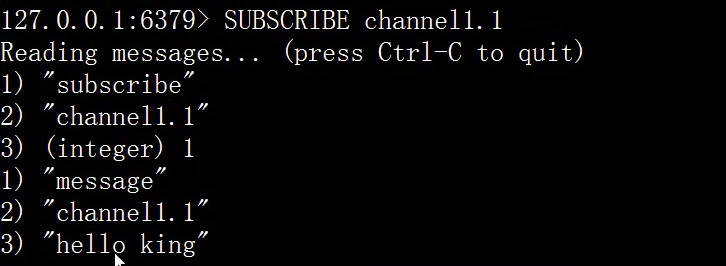
我们再打开一个客户端，订阅channel1.1的频道，返回的第一个值代表消息类型，subscribe代表订阅成功，第二三个值可能会因为第一个值而发生改变，含义也不同。第二个值代表你订阅的频道的名称，第三个代表当前订阅客户端的数量。



此时，再通过PUBLISH发布消息，返回2表示已经有两个客户端订阅这个channel1.1的频道。

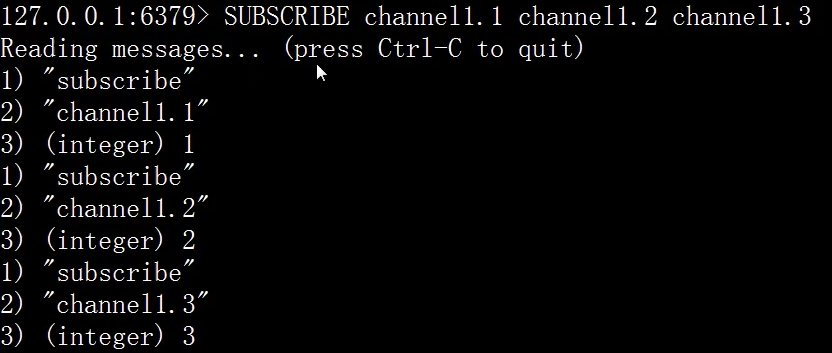


PUBLISH channel1.1 （此时就接到了我们之后发送的这个命令hello king）



注意：发布出去消息是不会被持久化，只有客户端订阅这个频道之后才能收到之后发送消息

SUBSCRIBE channel1.1 channel1.2 channel1.3（这个客户端订阅了三个频道）

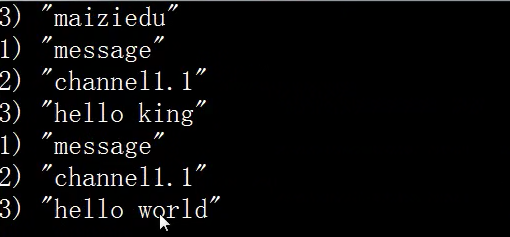


PUBLISH channel1.1 ‘hello world'（指定发布消息channel1.1频道发布信息‘hello world'，返回的值4代表有4个客户端订阅该频道的数量）

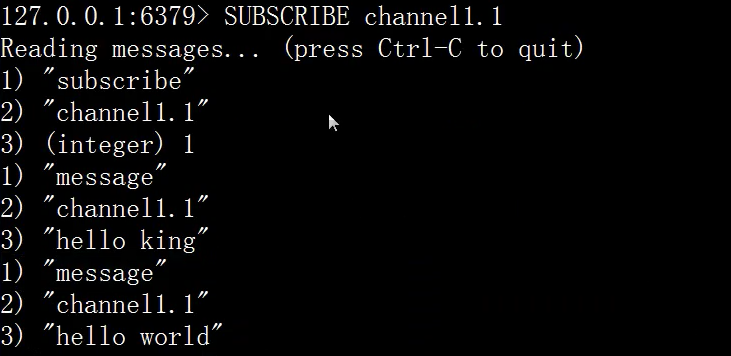


我们可以看到这4个客户端都收到了‘hello world’

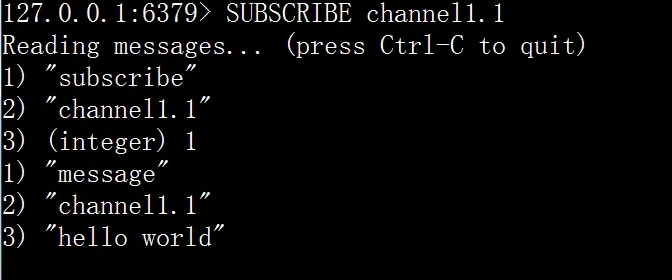
第一个客户端



第二个客户端

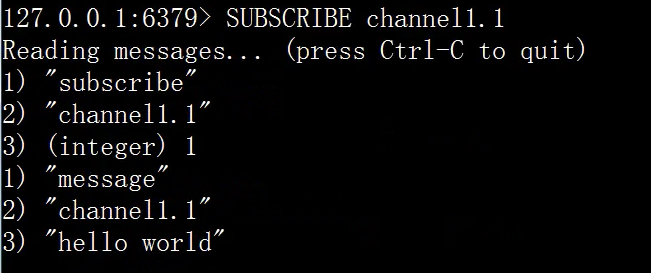


第三个客户端，Message代表的是接收到的消息



第四个客户端





**UNSUBSCRIBE 取消某个频道订阅，但是在REDIS客户端中做了限制，没有办法进行验证：**

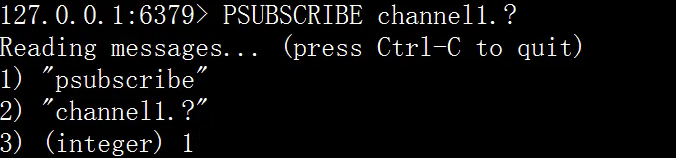
UNSUBSCRIBE [channel]

UNSUBSCRIBE channel1.1 （取消channel1.1频道订阅）

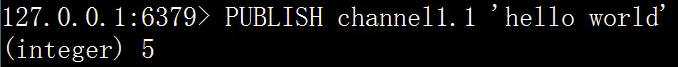


**PSUBSCRIBE 订阅指定的规则：**

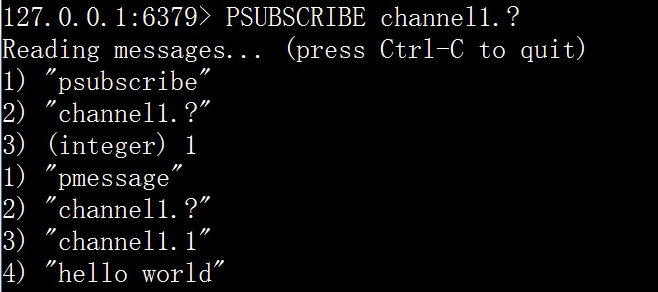
PSUBSCRIBE channel1.？



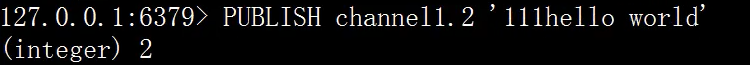
再打开另一个客户端，通过PUBLISH在channel1.1来发布消息



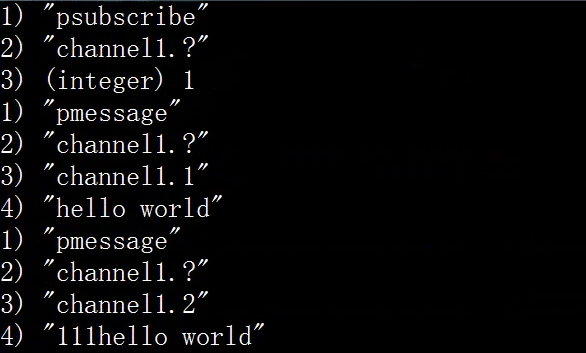
这样，这里就会收到channel1.1发布的消息



再打开客户端，通过PUBLISH在channel1.2来发布消息



这样，这里就会收到channel1.2发布的消息



UNPSUBSCRIBE

**Redis事务：**

**MULTI、 EXEC、DISCARD 和 WATCH 是 Redis 事务相关的命令。事务可以一次执行多个命令， 并且带有以下两个重要的保证：**

* 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。
* 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。

**MULTI 命令用于开启一个事务，它总是返回 OK 。**

MULTI 执行之后， 客户端可以继续向服务器发送任意多条命令， 这些命令不会立即被执行， 而是被放到一个队列中， 当 EXEC命令被调用时， 所有队列中的命令才会被执行。

**另一方面， 通过调用 DISCARD ， 客户端可以清空事务队列， 并放弃执行事务。**

**EXEC 命令的回复是一个数组**， **数组中的每个元素都是执行事务中的命令所产生的回复。** 其中， 回复元素的先后顺序和命令发送的先后顺序一致。

**当客户端处于事务状态时， 所有传入的命令都会返回一个内容为 QUEUED 的状态回复（status reply）， 这些被入队的命令将在 EXEC 命令被调用时执行。**

以下是一个事务例子， 它原子地增加了 foo 和 bar 两个键的值：

> MULTI

OK

> INCR foo

QUEUED

> INCR bar

QUEUED

> EXEC

1) (integer) 1

2) (integer) 1

**EXEC 命令负责触发并执行事务中的所有命令：**

* 如果客户端在使用 MULTI 开启了一个事务之后，却因为断线而没有成功执行 EXEC ，那么事务中的所有命令都不会被执行。
* 另一方面，如果客户端成功在开启事务之后执行 EXEC ，那么事务中的所有命令都会被执行。

当使用 AOF 方式做持久化的时候， Redis 会使用单个 write(2) 命令将事务写入到磁盘中。

**然而，如果 Redis 服务器因为某些原因被管理员杀死，或者遇上某种硬件故障，那么可能只有部分事务命令会被成功写入到磁盘中。**

如果 Redis 在重新启动时发现 AOF 文件出了这样的问题，那么它会退出，并汇报一个错误。

使用redis-check-aof程序可以修复这一问题：它会移除 AOF 文件中不完整事务的信息，确保服务器可以顺利启动。

从 2.2 版本开始，Redis 还可以通过乐观锁（optimistic lock）实现 CAS （check-and-set）操作，

**事务中的错误**

使用事务时可能会遇上以下两种错误：

* **事务在执行** [**EXEC**](http://www.redis.cn/commands/exec.html) **之前，入队的命令可能会出错**。比如说，命令可能会产生语法错误（参数数量错误，参数名错误，等等），或者其他更严重的错误，比如内存不足（如果服务器使用 maxmemory 设置了最大内存限制的话）。

**对于发生在** [**EXEC**](http://www.redis.cn/commands/exec.html) **执行之前的错误：**

客户端以前的做法是检查命令入队所得的返回值：如果命令入队时返回 QUEUED ，那么入队成功；否则，就是入队失败。如果有命令在入队时失败，那么大部分客户端都会停止并取消这个事务。

不过，**从 Redis 2.6.5 开始**，服务器会对命令入队失败的情况进行记录，并在客户端调用 [EXEC](http://www.redis.cn/commands/exec.html) 命令时，拒绝执行并自动放弃这个事务。

* **命令可能在** [**EXEC**](http://www.redis.cn/commands/exec.html) **调用之后失败。**举个例子，事务中的命令可能处理了错误类型的键，比如将列表命令用在了字符串键上面，诸如此类。

**至于那些在** [**EXEC**](http://www.redis.cn/commands/exec.html) **命令执行之后所产生的错误:**

并没有对它们进行特别处理： 即使事务中有某个/某些命令在执行时产生了错误， 事务中的其他命令仍然会继续执行。

MULTI

+OK

SET a 3

+QUEUED

LPOP a

+QUEUED

EXEC

\*2

+OK

-ERR Operation against a key holding the wrong kind of value

EXEC 返回两条bulk-string-reply第一条是 OK ，而第二条是 -ERR 。 至于怎样用合适的方法来表示事务中的错误， 则是由客户端自己决定的。

**最重要的是记住这样一条， 即使事务中有某条/某些命令执行失败了， 事务队列中的其他命令仍然会继续执行 —— Redis 不会停止执行事务中的命令。**

**Redis 不支持回滚（roll back）**

如果你有使用关系式数据库的经验， 那么 “Redis 在事务失败时不进行回滚，而是继续执行余下的命令”这种做法可能会让你觉得有点奇怪。

以下是这种做法的优点：

* Redis 命令只会因为错误的语法而失败（并且这些问题不能在入队时发现），或是命令用在了错误类型的键上面：这也就是说，从实用性的角度来说，失败的命令是由编程错误造成的，而这些错误应该在开发的过程中被发现，而不应该出现在生产环境中。
* 因为不需要对回滚进行支持，所以 Redis 的内部可以保持简单且快速。

有种观点认为 Redis 处理事务的做法会产生 bug ， 然而需要注意的是， 在通常情况下， 回滚并不能解决编程错误带来的问题。 举个例子， 如果你本来想通过 [INCR](http://www.redis.cn/commands/incr.html) 命令将键的值加上 1 ， 却不小心加上了 2 ， 又或者对错误类型的键执行了 [INCR](http://www.redis.cn/commands/incr.html) ， 回滚是没有办法处理这些情况的。

**放弃事务**

**当执行** [**DISCARD**](http://www.redis.cn/commands/discard.html) **命令时， 事务会被放弃， 事务队列会被清空， 并且客户端会从事务状态中退出：**

**使用 check-and-set 操作实现乐观锁**

[**WATCH**](http://www.redis.cn/commands/watch.html) **命令可以为 Redis 事务提供 check-and-set （CAS）行为。**

**被** [**WATCH**](http://www.redis.cn/commands/watch.html) **的键会被监视，并会发觉这些键是否被改动过了。 如果有至少一个被监视的键在** [**EXEC**](http://www.redis.cn/commands/exec.html) **执行之前被修改了， 那么整个事务都会被取消，** [**EXEC**](http://www.redis.cn/commands/exec.html) **返回[nil-reply](http://www.redis.cn/topics/protocol.html" \l "nil-reply)来表示事务已经失败。**

举个例子， 假设我们需要原子性地为某个值进行增 1 操作（假设 [INCR](http://www.redis.cn/commands/incr.html) 不存在）。

首先我们可能会这样做：

val = GET mykey

val = val + 1

SET mykey $val

上面的这个实现在只有一个客户端的时候可以执行得很好。 但是， 当多个客户端**同时**对同一个键进行这样的操作时， 就会产生竞争条件。举个例子， 如果客户端 A 和 B 都读取了键原来的值， 比如 10 ， 那么两个客户端都会将键的值设为 11 ， 但正确的结果应该是 12 才对。

有了 [WATCH](http://www.redis.cn/commands/watch.html) ， 我们就可以轻松地解决这类问题了：

WATCH mykey

val = GET mykey

val = val + 1

MULTI

SET mykey $val

EXEC

使用上面的代码， **如果在** [**WATCH**](http://www.redis.cn/commands/watch.html) **执行之后，** [**EXEC**](http://www.redis.cn/commands/exec.html) **执行之前， 有其他客户端修改了 mykey 的值， 那么当前客户端的事务就会失败**。 程序需要做的， 就是不断重试这个操作， 直到没有发生碰撞为止。

这种形式的锁被称作乐观锁， 它是一种非常强大的锁机制。 并且因为大多数情况下， 不同的客户端会访问不同的键， 碰撞的情况一般都很少， 所以通常并不需要进行重试。

[WATCH](http://www.redis.cn/commands/watch.html) 使得 [EXEC](http://www.redis.cn/commands/exec.html) 命令需要有条件地执行： **事务只能在所有被监视键都没有被修改的前提下执行， 如果这个前提不能满足的话，事务就不会被执行。**

[**WATCH**](http://www.redis.cn/commands/watch.html) **命令可以被调用多次。 对键的监视从** [**WATCH**](http://www.redis.cn/commands/watch.html) **执行之后开始生效， 直到调用** [**EXEC**](http://www.redis.cn/commands/exec.html) **为止。**

用户还可以在单个 [WATCH](http://www.redis.cn/commands/watch.html) 命令中监视任意多个键， 就像这样：

redis> WATCH key1 key2 key3

OK

**取消监视：**

* 当 [EXEC](http://www.redis.cn/commands/exec.html) 被调用时， 不管事务是否成功执行， 对所有键的监视都会被取消。
* 另外， 当客户端断开连接时， 该客户端对键的监视也会被取消。
* 使用无参数的 [UNWATCH](http://www.redis.cn/commands/unwatch.html) 命令可以手动取消对所有键的监视。

**在Redis中，实现高可用的技术主要包括持久化、复制、哨兵和集群:**

* **持久化**：持久化是最简单的高可用方法，有时甚至不被归为高可用的手段，主要作用是数据备份，即将数据存储在硬盘，保证数据不会因进程退出而丢失。
* **复制：**复制是高可用Redis的基础，哨兵和集群都是在复制基础上实现高可用的。复制主要实现了数据的多机备份以及对于读操作的负载均衡和简单的故障恢复。缺陷是故障恢复无法自动化、写操作无法负载均衡、存储能力受到单机的限制。
* **哨兵：**在复制的基础上，哨兵实现了自动化的故障恢复。缺陷是写操作无法负载均衡、存储能力受到单机的限制。
* **集群：**通过集群，Redis解决了写操作无法负载均衡以及存储能力受到单机限制的问题，实现了较为完善的高可用方案。

**Redis持久化概述**

**持久化的功能**：**Redis是内存数据库，数据都是存储在内存中，为了避免进程退出导致数据的永久丢失，需要定期将Redis中的数据以某种形式（数据或命令）从内存保存到硬盘。当下次Redis重启时，利用持久化文件实现数据恢复。**除此之外，为了进行灾难备份，可以将持久化文件拷贝到一个远程位置。

**Redis持久化分为RDB持久化和AOF持久化，前者将当前数据保存到硬盘，后者则是将每次执行的写命令保存到硬盘（类似于MySQL的Binlog）**。由于AOF持久化的实时性更好，即当进程意外退出时丢失的数据更少，因此AOF是目前主流的持久化方式

**RDB持久化:**

**RDB持久化是将当前进程中的数据生成快照保存到硬盘（因此也称作快照持久化）**，**保存的文件后缀是rdb；当Redis重新启动时，可以读取快照文件恢复数据**

**RDB持久化的触发分为手动触发和自动触发两种。**

**1、手动触发**

**save命令和bgsave命令都可以生成RDB文件**

**save命令**会阻塞Redis服务器进程，直到RDB文件创建完毕为止，在Redis服务器阻塞期间，服务器不能处理任何命令请求。

而**bgsave命令**会创建一个子进程，由子进程来负责创建RDB文件，父进程(即Redis主进程)则继续处理请求。

bgsave命令执行过程中，只有fork子进程时会阻塞服务器，而对于save命令，整个过程都会阻塞服务器，因此save已基本被废弃，线上环境要杜绝save的使用，后文中也将只介绍bgsave命令。此外，在自动触发RDB持久化时，Redis也会选择bgsave而不是save来进行持久化。下面介绍自动触发RDB持久化的条件。

**2）自动触发**

**save m n**

自动触发最常见的情况是在配置文件中通过save m n，指定当m秒内发生n次变化时，会触发bgsave。 例如，查看Redis的默认配置文件(Linux下为Redis根目录下的redis.conf)，可以看到如下配置信息：

其中save 900 1的含义是：当时间到900秒时，如果Redis数据发生了至少1次变化，则执行bgsave；save 300 10和save 60 10000同理。当三个save条件满足任意一个时，都会引起bgsave的调用。

除了save m n以外，还有一些其他情况会触发bgsave：

* **在主从复制场景下，如果从节点执行全量复制操作，则主节点会执行bgsave命令，并将rdb文件发送给从节点；**
* **执行shutdown命令时，自动执行rdb持久化，如下图所示：**

**bgSave的执行流程：**

前面介绍了触发bgsave的条件，下面将说明bgsave命令的执行流程，如下图所示

* Redis父进程首先判断：当前是否在执行save，或bgsave/bgrewriteaof（后面会详细介绍该命令）的子进程，如果在执行则bgsave命令直接返回。bgsave/bgrewriteaof 的子进程不能同时执行，主要是基于性能方面的考虑：两个并发的子进程同时执行大量的磁盘写操作，可能引起严重的性能问题。
* 父进程执行fork操作创建子进程，这个过程中父进程是阻塞的，Redis不能执行来自客户端的任何命令；
* 父进程fork后，bgsave命令返回”Background saving started”信息并不再阻塞父进程，并可以响应其他命令；
* 子进程创建RDB文件，根据父进程内存快照生成临时快照文件，完成后对原有文件进行原子替换；
* 子进程发送信号给父进程表示完成，父进程更新统计信息。

**当 Redis 需要保存 dump.rdb 文件时， 服务器执行以下操作:**

* **Redis 调用forks. 同时拥有父进程和子进程。**
* **子进程将数据集写入到一个临时 RDB 文件中。**
* **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。**

**RDB文件：RDB文件是经过压缩的二进制文件，dbfilename dump.rdb：RDB文件名。**

**启动时加载：**

RDB文件的载入工作是在服务器启动时自动执行的，并没有专门的命令。但是由于AOF的优先级更高，因此当AOF开启时，Redis会优先载入AOF文件来恢复数据；只有当AOF关闭时，才会在Redis服务器启动时检测RDB文件，并自动载入。服务器载入RDB文件期间处于阻塞状态，直到载入完成为止。

**Redis载入RDB文件时，会对RDB文件进行校验，如果文件损坏，则日志中会打印错误，Redis启动失败。**

**AOF持久化：**

**AOF持久化(即Append Only File持久化)，则是将Redis执行的每次写命令记录到单独的日志文件中（有点像MySQL的binlog）；当Redis重启时再次执行AOF文件中的命令来恢复数据。与RDB相比，AOF的实时性更好，因此已成为主流的持久化方案。**

**开启AOF：Redis服务器默认开启RDB，关闭AOF；要开启AOF，需要在配置文件中配置：**

appendOnly yes

**执行流程**

由于需要记录Redis的每条写命令，因此AOF不需要触发，下面介绍AOF的执行流程。

**AOF的执行流程包括：**

* 命令追加(append)：将Redis的写命令追加到缓冲区aof\_buf；
* 文件写入(write)和文件同步(sync)：根据不同的同步策略将aof\_buf中的内容同步到硬盘；
* 文件重写(rewrite)：定期重写AOF文件，达到压缩的目的。

**1) 命令追加(append)**

**Redis先将写命令追加到缓冲区，而不是直接写入文件，主要是为了避免每次有写命令都直接写入硬盘，导致硬盘IO成为Redis负载的瓶颈。**

命令追加的格式是Redis命令请求的协议格式，它是一种纯文本格式，具有兼容性好、可读性强、容易处理、操作简单避免二次开销等优点；具体格式略。在AOF文件中，除了用于指定数据库的select命令（如select 0 为选中0号数据库）是由Redis添加的，其他都是客户端发送来的写命令

**2) 文件写入(write)和文件同步(sync)**

Redis提供了多种AOF缓存区的同步文件策略，策略涉及到操作系统的write函数和fsync函数，说明如下：

为了提高文件写入效率，在现代操作系统中，当用户调用write函数将数据写入文件时，操作系统通常会将数据暂存到一个内存缓冲区里，当缓冲区被填满或超过了指定时限后，才真正将缓冲区的数据写入到硬盘里。这样的操作虽然提高了效率，但也带来了安全问题：如果计算机停机，内存缓冲区中的数据会丢失；因此系统同时提供了fsync、fdatasync等同步函数，可以强制操作系统立刻将缓冲区中的数据写入到硬盘里，从而确保数据的安全性。

AOF缓存区的同步文件策略由参数appendfsync控制，各个值的含义如下：

* **always**：命令写入aof\_buf后立即调用系统fsync操作同步到AOF文件，fsync完成后线程返回。这种情况下，每次有写命令都要同步到AOF文件，硬盘IO成为性能瓶颈，Redis只能支持大约几百TPS写入，严重降低了Redis的性能；即便是使用固态硬盘（SSD），每秒大约也只能处理几万个命令，而且会大大降低SSD的寿命。
* **no：**命令写入aof\_buf后调用系统write操作，不对AOF文件做fsync同步；同步由操作系统负责，通常同步周期为30秒。这种情况下，文件同步的时间不可控，且缓冲区中堆积的数据会很多，数据安全性无法保证。
* **everysec：**命令写入aof\_buf后调用系统write操作，write完成后线程返回；fsync同步文件操作由专门的线程每秒调用一次。everysec是前述两种策略的折中，是性能和数据安全性的平衡，因此是Redis的默认配置，也是我们推荐的配置。

**3) 文件重写(rewrite)**

随着时间流逝，Redis服务器执行的写命令越来越多，AOF文件也会越来越大；过大的AOF文件不仅会影响服务器的正常运行，也会导致数据恢复需要的时间过长。

文件重写是指定期重写AOF文件，减小AOF文件的体积。需要注意的是，AOF重写是把Redis进程内的数据转化为写命令，同步到新的AOF文件，不会对旧的AOF文件进行任何读取、写入操作。

关于文件重写需要注意的另一点是：对于AOF持久化来说，文件重写虽然是强烈推荐的，但并不是必须的。即使没有文件重写，数据也可以被持久化并在Redis启动的时候导入。因此在一些实现中，会关闭自动的文件重写，然后通过定时任务在每天的某一时刻定时执行。

**文件重写之所以能够压缩AOF文件，原因在于：**

* 过期的数据不再写入文件
* 无效的命令不再写入文件：如有些数据被重复设值(set mykey v1, set mykey v2)、有些数据被删除了(sadd myset v1, del myset)等等
* 多条命令可以合并为一个：如sadd myset v1, sadd myset v2, sadd myset v3可以合并为sadd myset v1 v2 v3。不过为了防止单条命令过大造成客户端缓冲区溢出，对于list、set、hash、zset类型的key，并不一定只使用一条命令；而是以某个常量为界将命令拆分为多条。这个常量在redis.h/REDIS\_AOF\_REWRITE\_ITEMS\_PER\_CMD中定义，不可更改，3.0版本中值是64。

通过上述内容可以看出，由于重写后AOF执行的命令减少了，文件重写既可以减少文件占用的空间，也可以加快恢复速度。

**文件重写的流程:**

文件重写流程如下图所示：

关于文件重写的流程，有两点需要特别注意：

* 重写由父进程fork子进程进行；
* 重写期间Redis执行的写命令，需要追加到新的AOF文件中，为此Redis引入了aof\_rewrite\_buf缓存。

对照上图，文件重写的流程如下：

* Redis父进程首先判断当前是否存在正在执行 bgsave/bgrewriteaof的子进程，如果存在则bgrewriteaof命令直接返回，如果存在bgsave命令则等bgsave执行完成后再执行。前面曾介绍过，这个主要是基于性能方面的考虑。
* 父进程执行fork操作创建子进程，这个过程中父进程是阻塞的。
* 父进程fork后，bgrewriteaof命令返回”Background append only file rewrite started”信息并不再阻塞父进程，并可以响应其他命令。Redis的所有写命令依然写入AOF缓冲区，并根据appendfsync策略同步到硬盘，保证原有AOF机制的正确。

由于fork操作使用写时复制技术，子进程只能共享fork操作时的内存数据。由于父进程依然在响应命令，因此Redis使用AOF重写缓冲区(图中的aof\_rewrite\_buf)保存这部分数据，防止新AOF文件生成期间丢失这部分数据。也就是说，bgrewriteaof执行期间，Redis的写命令同时追加到aof\_buf和aof\_rewirte\_buf两个缓冲区。

* 子进程根据内存快照，按照命令合并规则写入到新的AOF文件。
* 子进程写完新的AOF文件后，向父进程发信号，父进程更新统计信息，具体可以通过info persistence查看。

父进程把AOF重写缓冲区的数据写入到新的AOF文件，这样就保证了新AOF文件所保存的数据库状态和服务器当前状态一致。

使用新的AOF文件替换老文件，完成AOF重写。

**AOF 重写和 RDB 创建快照一样，都巧妙地利用了写时复制机制:**

* + **Redis 执行 fork() ，现在同时拥有父进程和子进程。**
  + **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。**
  + **搞定！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。**

**AOF的配置：**

* appendonly no：是否开启AOF
* appendfilename "appendonly.aof"：AOF文件名
* dir ./：RDB文件和AOF文件所在目录
* appendfsync everysec：fsync持久化策略

**下面分场景来讨论持久化策略的选择，下面的讨论也只是作为参考，实际方案可能更复杂更具多样性：**

1）如果Redis中的数据完全丢弃也没有关系（如Redis完全用作DB层数据的cache），那么无论是单机，还是主从架构，都可以不进行任何持久化。

2）在单机环境下（对于个人开发者，这种情况可能比较常见），如果可以接受十几分钟或更多的数据丢失，选择RDB对Redis的性能更加有利；如果只能接受秒级别的数据丢失，应该选择AOF。

3）但在多数情况下，我们都会配置主从环境，slave的存在既可以实现数据的热备，也可以进行读写分离分担Redis读请求，以及在master宕掉后继续提供服务。

在这种情况下，一种可行的做法是：

* master：完全关闭持久化（包括RDB和AOF），这样可以让master的性能达到最好；
* slave：关闭RDB，开启AOF（如果对数据安全要求不高，开启RDB关闭AOF也可以），并定时对持久化文件进行备份（如备份到其他文件夹，并标记好备份的时间）；然后关闭AOF的自动重写，然后添加定时任务，在每天Redis闲时（如凌晨12点）调用bgrewriteaof。

这里需要解释一下，为什么开启了主从复制，可以实现数据的热备份，还需要设置持久化呢？因为在一些特殊情况下，主从复制仍然不足以保证数据的安全，例如：

* master和slave进程同时停止：考虑这样一种场景，如果master和slave在同一栋大楼或同一个机房，则一次停电事故就可能导致master和slave机器同时关机，Redis进程停止；如果没有持久化，则面临的是数据的完全丢失。
* master误重启：考虑这样一种场景，master服务因为故障宕掉了，如果系统中有自动拉起机制（即检测到服务停止后重启该服务）将master自动重启，由于没有持久化文件，那么master重启后数据是空的，slave同步数据也变成了空的；如果master和slave都没有持久化，同样会面临数据的完全丢失。需要注意的是，即便是使用了哨兵（关于哨兵后面会有文章介绍）进行自动的主从切换，也有可能在哨兵轮询到master之前，便被自动拉起机制重启了。因此，应尽量避免“自动拉起机制”和“不做持久化”同时出现。

4）异地灾备：上述讨论的几种持久化策略，针对的都是一般的系统故障，如进程异常退出、宕机、断电等，这些故障不会损坏硬盘。但是对于一些可能导致硬盘损坏的灾难情况，如火灾地震，就需要进行异地灾备。

例如对于单机的情形，可以定时将RDB文件或重写后的AOF文件，通过scp拷贝到远程机器，如阿里云、AWS等；对于主从的情形，可以定时在master上执行bgsave，然后将RDB文件拷贝到远程机器，或者在slave上执行bgrewriteaof重写AOF文件后，将AOF文件拷贝到远程机器上。

一般来说，由于RDB文件文件小、恢复快，因此灾难恢复常用RDB文件；异地备份的频率根据数据安全性的需要及其它条件来确定，但最好不要低于一天一次。

**fork阻塞：CPU的阻塞：**

在Redis的实践中，众多因素限制了Redis单机的内存不能过大，例如：

* 当面对请求的暴增，需要从库扩容时，Redis内存过大会导致扩容时间太长；
* 当主机宕机时，切换主机后需要挂载从库，Redis内存过大导致挂载速度过慢；
* 以及持久化过程中的fork操作，下面详细说明。

父进程通过fork操作可以创建子进程；子进程创建后，父子进程共享代码段，不共享进程的数据空间，但是子进程会获得父进程的数据空间的副本。在操作系统fork的实际实现中，基本都采用了写时复制技术，即在父/子进程试图修改数据空间之前，父子进程实际上共享数据空间；但是当父/子进程的任何一个试图修改数据空间时，操作系统会为修改的那一部分(内存的一页)制作一个副本。

虽然fork时，子进程不会复制父进程的数据空间，但是会复制内存页表（页表相当于内存的索引、目录）；父进程的数据空间越大，内存页表越大，fork时复制耗时也会越多。

在Redis中，无论是RDB持久化的bgsave，还是AOF重写的bgrewriteaof，都需要fork出子进程来进行操作。如果Redis内存过大，会导致fork操作时复制内存页表耗时过多；而Redis主进程在进行fork时，是完全阻塞的，也就意味着无法响应客户端的请求，会造成请求延迟过大

**AOF追加阻塞：硬盘的阻塞**

前面提到过，在AOF中，如果AOF缓冲区的文件同步策略为everysec，则：在主线程中，命令写入aof\_buf后调用系统write操作，write完成后主线程返回；fsync同步文件操作由专门的文件同步线程每秒调用一次。

这种做法的问题在于，如果硬盘负载过高，那么fsync操作可能会超过1s；如果Redis主线程持续高速向aof\_buf写入命令，硬盘的负载可能会越来越大，IO资源消耗更快；如果此时Redis进程异常退出，丢失的数据也会越来越多，可能远超过1s。

为此，Redis的处理策略是这样的：主线程每次进行AOF会对比上次fsync成功的时间；如果距上次不到2s，主线程直接返回；如果超过2s，则主线程阻塞直到fsync同步完成。因此，如果系统硬盘负载过大导致fsync速度太慢，会导致Redis主线程的阻塞；此外，使用everysec配置，AOF最多可能丢失2s的数据，而不是1s。

**主从复制概述**

**1、主从复制**

**主从复制是指将一台Redis服务器的数据，复制到其它的Redis服务器。**前**者称为主节点(master)，后者称为从节点(slave)；数据的复制是单向的，只能由主节点到从节点。**

默认情况下，每台Redis服务器都是主节点，且一个主节点可以有多个从节点（或没有从节点），但一个从节点只能有一个主节点。

**主从复制的作用主要包括：**

* **数据冗余：**主从复制实现了数据的热备份，是持久化之外的一种数据冗余方式。
* **故障恢复：**当主节点出现问题时，可以由从节点提供服务，实现快速的故障恢复，但实际上是一种服务的冗余。
* **负载均衡：**在主从复制的基础上，配合读写分离，可以由主节点提供写服务，由从节点提供读服务（即写Redis数据时应用连接主节点，读Redis数据时应用连接从节点），分担服务器负载；尤其是在写少读多的场景下，通过多个从节点分担读负载，可以大大提高Redis服务器的并发量。
* **高可用基石：**除了上述作用以外，主从复制还是哨兵和集群能够实施的基础，因此说主从复制是Redis高可用的基础。

**二、如何使用主从复制**

为了更直观的理解主从复制，在介绍其内部原理之前，先说明我们需要如何操作才能开启主从复制。

**1、建立复制：**

需要注意，主从复制的开启，完全是在从节点发起的，不需要我们在主节点做任何事情。

从节点开启主从复制，有3种方式：

* **配置文件**

在从服务器的配置文件中加入：slaveof<masterip><masterport>

* **启动命令**

redis-server启动命令后加入：slaveof<masterip><masterport>

* **客户端命令**

Redis服务器启动后直接通过客户端执行命令：slaveof<masterip><masterport>，则该Redis实例成为从节点。

上述3种方式是等效的，下面以客户端命令的方式为例，看一下当执行了slaveof后，Redis主节点和从节点的变化。

**主从复制实例：slaveof <masterIp> <masterport>**

**准备工作：启动两个节点**

方便起见，实验所使用的主从节点是在一台机器上的不同Redis实例，其中主节点监听6379端口，从节点监听6380端口；从节点监听的端口号可以在配置文件中修改：

启动后可以看到：

两个Redis节点启动后（分别称为6379节点和6380节点），默认都是主节点。

**建立复制** :

此时在6380节点执行slaveof命令，使之变为从节点：

**观察效果**

**下面验证一下，在主从复制建立后，主节点的数据会复制到从节点中。**

* 首先在从节点中查询一个不存在的key：
* 然后在主节点中增加这个key：
* 此时在从节点中再次查询这个key，会发现主节点的操作已经同步至从节点：
* 然后在主节点删除这个key：
* 此时从节点中再次查询这个key，会发现主节点的操作已经同步至从节点：

**断开复制:slaveof no one**

通过slaveof<masterip><masterport>命令建立主从复制关系以后，可以通过slaveof no one断开。需要注意的是，从节点断开复制后，不会删除已有的数据，只是不再接受主节点新的数据变化。

从节点执行slaveof no one后，打印日志如下所示，可以看出断开复制后，从节点又变回为主节点：

主节点打印日志如下：

**主从复制的原理：**

**1、连接建立阶段**

**该阶段的主要作用是在主从节点之间建立连接，为数据同步做好准备。**

**步骤1：保存主节点信息**

从节点服务器内部维护了两个字段，即masterhost和masterport字段，用于存储主节点的ip和port信息。

需要注意的是，slaveof是异步命令，从节点完成主节点ip和port的保存后，向发送slaveof命令的客户端直接返回OK，实际的复制操作在这之后才开始进行。

这个过程中，可以看到从节点打印日志如下：

**步骤2：建立socket连接**

从节点每秒1次调用复制定时函数replicationCron()，如果发现了有主节点可以连接，便会根据主节点的ip和port，创建socket连接。如果连接成功，则：

从节点为该socket建立一个专门处理复制工作的文件事件处理器，负责后续的复制工作，如接收RDB文件、接收命令传播等。

主节点接收到从节点的socket连接后（即accept之后），为该socket创建相应的客户端状态，并将从节点看做是连接到主节点的一个客户端，后面的步骤会以从节点向主节点发送命令请求的形式来进行。

这个过程中，从节点打印日志如下：

**步骤3：发送ping命令**

从节点成为主节点的客户端之后，发送ping命令进行首次请求，目的是检查socket连接是否可用以及主节点当前是否能够处理请求。

从节点发送ping命令后，可能出现3种情况：

* 返回pong：说明socket连接正常，且主节点当前可以处理请求，复制过程继续。
* 超时：一定时间后从节点仍未收到主节点的回复，说明socket连接不可用，则从节点断开socket连接，并重连。
* 返回pong以外的结果：如果主节点返回其它结果，如正在处理超时运行的脚本，说明主节点当前无法处理命令，则从节点断开socket连接，并重连。

在主节点返回pong情况下，从节点打印日志如下：

**步骤4：身份验证**

如果从节点中设置了masterauth选项，则从节点需要向主节点进行身份验证；没有设置该选项则不需要验证。从节点进行身份验证是通过向主节点发送auth命令进行的，auth命令的参数即为配置文件中的masterauth的值。

如果主节点设置密码的状态，与从节点masterauth的状态一致（一致是指都存在，且密码相同，或者都不存在），则身份验证通过，复制过程继续；如果不一致，则从节点断开socket连接，并重连。

**步骤5：发送从节点端口信息**

身份验证后，从节点会向主节点发送其监听的端口号（前述例子中为6380），主节点将该信息保存到该从节点对应的客户端的slave\_listening\_port字段中；该端口信息除了在主节点中执行info Replication时显示以外，没有其他作用。

**2、数据同步阶段**

**主从节点之间的连接建立以后，便可以开始进行数据同步，该阶段可以理解为从节点数据的初始化。具体执行的方式是：从节点向主节点发送psync命令（Redis2.8以前是sync命令），开始同步。**

**数据同步阶段是主从复制最核心的阶段，根据主从节点当前状态的不同，可以分为全量复制和部分复制，下面会有一部分专门讲解这两种复制方式以及psync命令的执行过程，这里不再详述。**

**需要注意的是，在数据同步阶段之前，从节点是主节点的客户端，主节点不是从节点的客户端；而到了这一阶段及以后，主从节点互为客户端。原因在于：在此之前，主节点只需要响应从节点的请求即可，不需要主动发请求，而在数据同步阶段和后面的命令传播阶段，主节点需要主动向从节点发送请求（如推送缓冲区中的写命令），才能完成复制。**

**3、命令传播阶段**

**数据同步阶段完成后，主从节点进入命令传播阶段。在这个阶段主节点将自己执行的写命令发送给从节点，从节点接收命令并执行，从而保证主从节点数据的一致性。**

在命令传播阶段，除了发送写命令，主从节点还维持着心跳机制：PING和REPLCONF ACK。由于心跳机制的原理涉及部分复制，因此将在介绍了部分复制的相关内容后单独介绍该心跳机制。

**延迟与不一致**

需要注意的是，命令传播是异步的过程，即主节点发送写命令后并不会等待从节点的回复。因此实际上主从节点之间很难保持实时的一致性，延迟在所难免。数据不一致的程度，与主从节点之间的网络状况、主节点写命令的执行频率、以及主节点中的repl-disable-tcp-nodelay配置等有关。

**repl-disable-tcp-nodelay no：**该配置作用于命令传播阶段，控制主节点是否禁止与从节点的TCP\_NODELAY。默认no，即不禁止TCP\_NODELAY。当设置为yes时，TCP会对包进行合并从而减少带宽，但是发送的频率会降低，从节点数据延迟增加，一致性变差；具体发送频率与Linux内核的配置有关，默认配置为40ms。当设置为no时，TCP会立马将主节点的数据发送给从节点，带宽增加但延迟变小。

一般来说，只有当应用对Redis数据不一致的容忍度较高，且主从节点之间网络状况不好时，才会设置为yes；多数情况使用默认值no。

**四、全量复制和部分复制**

接下来我们来探讨一下数据同步阶段的全量复制和部分复制。

在Redis2.8以前，从节点向主节点发送sync命令请求同步数据，此时的同步方式是全量复制；在Redis2.8及以后，从节点可以发送psync命令请求同步数据，此时根据主从节点当前状态的不同，同步方式可能是全量复制或部分复制。后文介绍以Redis2.8及以后版本为例。

* **全量复制：**用于初次复制或其它无法进行部分复制的情况，将主节点中的所有数据都发送给从节点，是一个非常重型的操作。
* **部分复制：**用于网络中断等情况后的复制，只将中断期间主节点执行的写命令发送给从节点，与全量复制相比更加高效。需要注意的是，如果网络中断时间过长，造成主节点没有能够完整地保存中断期间执行的写命令，则无法进行部分复制，仍使用全量复制。

**1、全量复制：**

Redis通过psync命令进行全量复制的过程如下：

* 从节点判断无法进行部分复制，向主节点发送全量复制的请求；或从节点发送部分复制的请求，但主节点判断无法进行全量复制。具体判断过程需要在讲述了部分复制原理后再介绍。
* 主节点收到全量复制的命令后，执行bgsave，在后台生成RDB文件，并使用一个缓冲区（称为复制缓冲区）记录从现在开始执行的所有写命令。
* 主节点的bgsave执行完成后，将RDB文件发送给从节点；从节点首先清除自己的旧数据，然后载入接收的RDB文件，将数据库状态更新至主节点执行bgsave时的数据库状态。
* 主节点将前述复制缓冲区中的所有写命令发送给从节点，从节点执行这些写命令，将数据库状态更新至主节点的最新状态。
* 如果从节点开启了AOF，则会触发bgrewriteaof的执行，从而保证AOF文件更新至主节点的最新状态。

下面是执行全量复制时，主从节点打印的日志，可以看出日志内容与上述步骤是完全对应的：

* 主节点的打印日志
* 从节点打印日志

其中，有几点需要注意：

* 从节点接收了来自主节点的89260个字节的数据；
* 从节点在载入主节点的数据之前要先将老数据清除；
* 从节点在同步完数据后，调用了bgrewriteaof。

通过全量复制的过程可以看出，全量复制是非常重型的操作：

* 主节点通过bgsave命令fork子进程进行RDB持久化，该过程是非常消耗CPU、内存(页表复制)、硬盘IO的。
* 主节点通过网络将RDB文件发送给从节点，对主从节点的带宽都会带来很大消耗。
* 从节点清空老数据、载入新RDB文件的过程是阻塞的，无法响应客户端的命令。如果从节点执行bgrewriteaof，也会带来额外的消耗。

**2、部分复制：**

由于全量复制在主节点数据量较大时效率太低，因此Redis2.8开始提供部分复制，用于处理网络中断时的数据同步。

部分复制的实现，依赖于三个重要的概念：

**复制偏移量**

主节点和从节点分别维护一个复制偏移量（offset），代表的是主节点向从节点传递的字节数；主节点每次向从节点传播N个字节数据时，主节点的offset增加N；从节点每次收到主节点传来的N个字节数据时，从节点的offset增加N。

offset用于判断主从节点的数据库状态是否一致：如果二者offset相同，则一致；如果offset不同，则不一致，此时可以根据两个offset找出从节点缺少的那部分数据。例如，如果主节点的offset是1000，而从节点的offset是500，那么部分复制就需要将offset为501-1000的数据传递给从节点，而offset为501-1000的数据存储的位置，就是下面要介绍的复制积压缓冲区。

**复制积压缓冲区**

复制积压缓冲区是由主节点维护的、固定长度的、先进先出(FIFO)队列，默认大小1MB；当主节点开始有从节点时创建，其作用是备份主节点最近发送给从节点的数据。注意，无论主节点有一个还是多个从节点，都只需要一个复制积压缓冲区。

在命令传播阶段，主节点除了将写命令发送给从节点，还会发送一份给复制积压缓冲区，作为写命令的备份；除了存储写命令，复制积压缓冲区中还存储了其中的每个字节对应的复制偏移量(offset) 。由于复制积压缓冲区定长且先进先出，所以它保存的是主节点最近执行的写命令；时间较早的写命令会被挤出缓冲区。

由于该缓冲区长度固定且有限，因此可以备份的写命令也有限，当主从节点offset的差距过大超过缓冲区长度时，将无法执行部分复制，只能执行全量复制。反过来说，为了提高网络中断时部分复制执行的概率，可以根据需要增大复制积压缓冲区的大小(通过配置repl-backlog-size)来设置；例如如果网络中断的平均时间是60s，而主节点平均每秒产生的写命令(特定协议格式)所占的字节数为100KB，则复制积压缓冲区的平均需求为6MB，保险起见，可以设置为12MB，来保证绝大多数断线情况都可以使用部分复制。

从节点将offset发送给主节点后，主节点根据offset和缓冲区大小决定能否执行部分复制：

* 如果offset偏移量之后的数据，仍然都在复制积压缓冲区里，则执行部分复制；
* 如果offset偏移量之后的数据已不在复制积压缓冲区中（数据已被挤出），则执行全量复制。

**服务器运行ID(runid)**

每个Redis节点(无论主从)，在启动时都会自动生成一个随机ID(每次启动都不一样)，由40个随机的十六进制字符组成；runid用来唯一识别一个Redis节点。 通过info server命令，可以查看节点的runid：

主从节点初次复制时，主节点将自己的runid发送给从节点，从节点将这个runid保存起来；当断线重连时，从节点会将这个runid发送给主节点；主节点根据runid判断能否进行部分复制：

* 如果从节点保存的runid与主节点现在的runid相同，说明主从节点之前同步过，主节点会继续尝试使用部分复制(到底能不能部分复制还要看offset和复制积压缓冲区的情况)；
* 如果从节点保存的runid与主节点现在的runid不同，说明从节点在断线前同步的Redis节点并不是当前的主节点，只能进行全量复制。

**psync命令的执行(包同步)：**

在了解了复制偏移量、复制积压缓冲区、节点运行id之后，本节将介绍psync命令的参数和返回值，从而说明psync命令执行过程中，主从节点是如何确定使用全量复制还是部分复制的。

psync命令的执行过程可以参见下图：

图片来源：《Redis设计与实现》

（1）首先，从节点根据当前状态，决定如何调用psync命令：

* 如果从节点之前未执行过slaveof或最近执行了slaveof no one，则从节点发送命令为psync ? -1，向主节点请求全量复制；
* 如果从节点之前执行了slaveof，则发送命令为psync<runid><offset>，其中runid为上次复制的主节点的runid，offset为上次复制截止时从节点保存的复制偏移量。

（2）主节点根据收到的psync命令及当前服务器状态来决定执行全量复制还是部分复制：

* 如果主节点版本低于Redis2.8，则返回-ERR回复，此时从节点重新发送sync命令执行全量复制；
* 如果主节点版本够新、runid与从节点发送的runid相同，且从节点发送的offset之后的数据在复制积压缓冲区中都存在，则回复+CONTINUE，表示将进行部分复制，从节点等待主节点发送其缺少的数据即可；
* 如果主节点版本够新，但runid与从节点发送的runid不同，或从节点发送的offset之后的数据已不在复制积压缓冲区中(在队列中被挤出了)，则回复+FULLRESYNC<runid><offset>，表示要进行全量复制。其中runid表示主节点当前的runid，offset表示主节点当前的offset，从节点保存这两个值，以备使用。

**部分复制演示：**

在下面的演示中，网络中断几分钟后恢复，断开连接的主从节点进行了部分复制。为了便于模拟网络中断，本例中的主从节点在局域网中的两台机器上。

**网络中断**

网络中断一段时间后，主节点和从节点都会发现失去了与对方的连接（关于主从节点对超时的判断机制，后面会有说明）；此后，从节点便开始执行对主节点的重连，由于此时网络还没有恢复，重连失败，从节点会一直尝试重连。

主节点日志如下：

从节点日志如下：

**网络恢复**

网络恢复后，从节点连接主节点成功，并请求进行部分复制，主节点接收请求后，二者进行部分复制以同步数据。

主节点日志如下：

从节点日志如下：

**五、命令传播阶段心跳机制**

在命令传播阶段，除了发送写命令，主从节点还维持着心跳机制：PING和REPLCONF ACK。心跳机制对于主从复制的超时判断、数据安全等有作用。

**1、主→从：PING**

每隔指定的时间，主节点会向从节点发送PING命令，这个PING命令的作用，主要是为了让从节点进行超时判断。

PING发送的频率由repl-ping-slave-period参数控制，单位是秒，默认值是10s。

关于该PING命令究竟是由主节点发给从节点，还是相反，有一些争议。因为在Redis的官方文档中，对该参数的注释说明是从节点向主节点发送PING命令，如下图所示：

但是根据该参数的名称(含有ping-slave)及代码实现，我认为该PING命令是主节点发给从节点的。相关代码如下：

**2、从→主：REPLCONF ACK**

在命令传播阶段，从节点会向主节点发送REPLCONF ACK命令，频率是每秒1次；命令格式为：REPLCONF ACK{offset}，其中offset指从节点保存的复制偏移量。

REPLCONF ACK命令的作用包括：

**实时监测主从节点网络状态**

该命令会被主节点用于复制超时的判断。此外，在主节点中使用info Replication，可以看到其从节点的状态中的lag值，代表的是主节点上次收到该REPLCONF ACK命令的时间间隔，在正常情况下，该值应该是0或1，如下图所示：

**检测命令丢失**

从节点发送了自身的offset，主节点会与自己的offset对比，如果从节点数据缺失（如网络丢包），主节点会推送缺失的数据（这里也会利用复制积压缓冲区）。注意，offset和复制积压缓冲区，不仅可以用于部分复制，也可以用于处理命令丢失等情形；区别在于前者是在断线重连后进行的，而后者是在主从节点没有断线的情况下进行的。

**辅助保证从节点的数量和延迟**

Redis主节点中使用min-slaves-to-write和min-slaves-max-lag参数，来保证主节点在不安全的情况下不会执行写命令。

所谓不安全，是指从节点数量太少，或延迟过高。例如min-slaves-to-write和min-slaves-max-lag分别是3和10，含义是如果从节点数量小于3个，或所有从节点的延迟值都大于10s，则主节点拒绝执行写命令。而这里从节点延迟值的获取，就是通过主节点接收到REPLCONF ACK命令的时间来判断的，即前面所说的info Replication中的lag值。

**六、应用中的问题**

**1、读写分离及其中的问题**

在主从复制基础上实现的读写分离，可以实现Redis的读负载均衡：由主节点提供写服务，由一个或多个从节点提供读服务（多个从节点既可以提高数据冗余程度，也可以最大化读负载能力）；在读负载较大的应用场景下，可以大大提高Redis服务器的并发量。

下面介绍在使用Redis读写分离时，需要注意的问题：

**延迟与不一致问题**

前面已经讲到，由于主从复制的命令传播是异步的，延迟与数据的不一致不可避免。如果应用对数据不一致的接受程度较低，可能的优化措施包括：

* 优化主从节点之间的网络环境（如在同机房部署）；
* 监控主从节点延迟（通过offset）判断，如果从节点延迟过大，通知应用不再通过该从节点读取数据；
* 使用集群同时扩展写负载和读负载等。

在命令传播阶段以外的其它情况下，从节点的数据不一致可能更加严重，例如连接在数据同步阶段，或从节点失去与主节点的连接时等。

从节点的slave-serve-stale-data参数便与此有关，它控制这种情况下从节点的表现：如果为yes（默认值），则从节点仍能够响应客户端的命令；如果为no，则从节点只能响应info、slaveof等少数命令。该参数的设置与应用对数据一致性的要求有关；如果对数据一致性要求很高，则应设置为no。

**数据过期问题**

在单机版Redis中，存在两种删除策略：

* **惰性删除：**服务器不会主动删除数据，只有当客户端查询某个数据时，服务器判断该数据是否过期，如果过期则删除。
* **定期删除：**服务器执行定时任务删除过期数据，但是考虑到内存和CPU的折中（删除会释放内存，但是频繁的删除操作对CPU不友好），该删除的频率和执行时间都受到了限制。

在主从复制场景下，为了主从节点的数据一致性，从节点不会主动删除数据，而是由主节点控制从节点中过期数据的删除。由于主节点的惰性删除和定期删除策略都不能保证主节点及时对过期数据执行删除操作，因此，当客户端通过Redis从节点读取数据时，很容易读取到已经过期的数据。

Redis3.2中，从节点在读取数据时，增加了对数据是否过期的判断：如果该数据已过期，则不返回给客户端。将Redis升级到3.2可以解决数据过期问题。

**故障切换问题**

在没有使用哨兵的读写分离场景下，应用针对读和写分别连接不同的Redis节点；当主节点或从节点出现问题而发生更改时，需要及时修改应用程序读写Redis数据的连接；连接的切换可以手动进行，或者自己写监控程序进行切换，但前者响应慢、容易出错，后者实现复杂，成本都不算低。

**总结**

**在使用读写分离之前，可以考虑其他方法增加Redis的读负载能力，如尽量优化主节点（减少慢查询、减少持久化等其他情况带来的阻塞等）提高负载能力；使用Redis集群同时提高读负载能力和写负载能力等。如果使用读写分离，可以使用哨兵，使主从节点的故障切换尽可能自动化，并减少对应用程序的侵入。**

**2、复制超时问题：**

主从节点复制超时是导致复制中断的最重要的原因之一，本小节单独说明超时问题，下一小节说明其它会导致复制中断的问题。

**超时判断意义**

在复制连接建立过程中及之后，主从节点都有机制判断连接是否超时，其意义在于：

* 如果主节点判断连接超时，其会释放相应从节点的连接，从而释放各种资源，否则无效的从节点仍会占用主节点的各种资源（输出缓冲区、带宽、连接等）；此外连接超时的判断可以让主节点更准确的知道当前有效从节点的个数，有助于保证数据安全（配合前面讲到的min-slaves-to-write等参数）。
* 如果从节点判断连接超时，则可以及时重新建立连接，避免与主节点数据长期的不一致。

**判断机制**

主从复制超时判断的核心，在于repl-timeout参数，该参数规定了超时时间的阈值（默认60s），对于主节点和从节点同时有效；主从节点触发超时的条件分别如下：

**（1）主节点：**每秒1次调用复制定时函数replicationCron()，在其中判断当前时间距离上次收到各个从节点REPLCONF ACK的时间，是否超过了repl-timeout值，如果超过了则释放相应从节点的连接。

**（2）\***\*从节点：\*\*从节点对超时的判断同样是在复制定时函数中判断，基本逻辑是：

* 如果当前处于连接建立阶段，且距离上次收到主节点的信息的时间已超过repl-timeout，则释放与主节点的连接；
* 如果当前处于数据同步阶段，且收到主节点的RDB文件的时间超时，则停止数据同步，释放连接；
* 如果当前处于命令传播阶段，且距离上次收到主节点的PING命令或数据的时间已超过repl-timeout值，则释放与主节点的连接。

**需要注意的坑**

下面介绍与复制阶段连接超时有关的一些实际问题：

**数据同步阶段：**在主从节点进行全量复制bgsave时，主节点需要首先fork子进程将当前数据保存到RDB文件中，然后再将RDB文件通过网络传输到从节点。如果RDB文件过大，主节点在fork子进程+保存RDB文件时耗时过多，可能会导致从节点长时间收不到数据而触发超时。此时从节点会重连主节点，然后再次全量复制，再次超时，再次重连……这是个悲伤的循环。为了避免这种情况的发生，除了注意Redis单机数据量不要过大，另一方面就是适当增大repl-timeout值，具体的大小可以根据bgsave耗时来调整。

**命令传播阶段：**如前所述，在该阶段主节点会向从节点发送PING命令，频率由repl-ping-slave-period控制；该参数应明显小于repl-timeout值(后者至少是前者的几倍)。否则，如果两个参数相等或接近，网络抖动导致个别PING命令丢失，此时恰巧主节点也没有向从节点发送数据，那从节点很容易判断超时。

**慢查询导致的阻塞：**如果主节点或从节点执行了一些慢查询（如keys \*或者对大数据的hgetall等），导致服务器阻塞；阻塞期间无法响应复制连接中对方节点的请求，可能导致复制超时。

**3、复制中断问题：**

主从节点超时是复制中断的原因之一，除此之外，还有其它情况可能导致复制中断，其中最主要的是复制缓冲区溢出问题。

**复制缓冲区溢出**

前面曾提到过，在全量复制阶段，主节点会将执行的写命令放到复制缓冲区中，该缓冲区存放的数据包括了以下几个时间段内主节点执行的写命令：bgsave生成RDB文件、RDB文件由主节点发往从节点、从节点清空老数据并载入RDB文件中的数据。

当主节点数据量较大，或者主从节点之间网络延迟较大时，可能导致该缓冲区的大小超过了限制，此时主节点会断开与从节点之间的连接；这种情况可能引起全量复制→复制缓冲区溢出导致连接中断→重连→全量复制→复制缓冲区溢出导致连接中断……的循环。

复制缓冲区的大小由client-output-buffer-limit slave{hard limit}{soft limit}{soft seconds}配置，默认值为client-output-buffer-limit slave 256MB 64MB 60，其含义是：如果buffer大于256MB，或者连续60s大于64MB，则主节点会断开与该从节点的连接。该参数是可以通过config set命令动态配置的（即不重启Redis也可以生效）。

当复制缓冲区溢出时，主节点打印日志如下所示：

需要注意的是，复制缓冲区是客户端输出缓冲区的一种，主节点会为每一个从节点分别分配复制缓冲区；而复制积压缓冲区则是一个主节点只有一个，无论它有多少个从节点。

**4、各场景下复制的选择及优化技巧：**

在介绍了Redis复制的种种细节之后，现在我们可以来总结一下，在下面常见的场景中，何时使用部分复制，以及需要注意哪些问题。

**第一次建立复制：**

此时全量复制不可避免，但仍有几点需要注意：如果主节点的数据量较大，应该尽量避开流量的高峰期，避免造成阻塞；如果有多个从节点需要建立对主节点的复制，可以考虑将几个从节点错开，避免主节点带宽占用过大。此外，如果从节点过多，也可以调整主从复制的拓扑结构，由一主多从结构变为树状结构（中间的节点既是其主节点的从节点，也是其从节点的主节点）；但使用树状结构应该谨慎——主节点的直接从节点减少，降低了主节点的负担，但是多层从节点的延迟增大，数据一致性变差，且结构复杂，维护相当困难。

**主节点重启**

主节点重启可以分为两种情况来讨论，一种是故障导致宕机，另一种则是有计划的重启。

主节点宕机

主节点宕机重启后，Runid会发生变化，因此不能进行部分复制，只能全量复制。实际上在主节点宕机的情况下，应进行故障转移处理，将其中的一个从节点升级为主节点，其它从节点从新的主节点进行复制；且故障转移应尽量的自动化，后面文章将要介绍的哨兵便可以进行自动的故障转移。

安全重启：debug reload

在一些场景下，可能希望对主节点进行重启，例如主节点内存碎片率过高，或者希望调整一些只能在启动时调整的参数。如果使用普通的手段重启主节点，会使得runid发生变化，可能导致不必要的全量复制。

为了解决这个问题，Redis提供了debug reload的重启方式：重启后，主节点的runid和offset都不受影响，避免了全量复制。

如下图所示，debug reload重启后runid和offset都未受影响：

但debug reload是一柄双刃剑：它会清空当前内存中的数据，重新从RDB文件中加载，这个过程会导致主节点的阻塞，因此也需要谨慎。

**从节点重启**

从节点宕机重启后，其保存的主节点的runid会丢失，因此即使再次执行slaveof，也无法进行部分复制。

**网络中断**

如果主从节点之间出现网络问题，造成短时间内网络中断，可以分为多种情况讨论：

* 网络问题时间极为短暂，只造成了短暂的丢包，主从节点都没有判定超时（未触发repl-timeout）。此时只需要通过REPLCONF ACK来补充丢失的数据即可。
* 网络问题时间很长，主从节点判断超时（触发了repl-timeout），且丢失的数据过多，超过了复制积压缓冲区所能存储的范围。此时主从节点无法进行部分复制，只能进行全量复制。为了尽可能避免这种情况发生，应该根据实际情况适当调整复制积压缓冲区的大小；此外及时发现并修复网络中断，也可以减少全量复制。
* 介于前述两种情况之间，主从节点判断超时，且丢失的数据仍然都在复制积压缓冲区中。此时主从节点可以进行部分复制。

**5、复制相关的配置**

这一节总结一下与复制有关的配置，说明这些配置的作用、起作用的阶段，以及配置方法等；通过了解这些配置，一方面加深对Redis复制的了解，另一方面掌握这些配置的方法，可以优化Redis的使用，少走坑。

配置大致可以分为主节点相关配置、从节点相关配置以及与主从节点都有关的配置，下面分别说明。

**与主从节点都有关的配置**

首先介绍最特殊的配置，它决定了该节点是主节点还是从节点：

* slaveof<masterip><masterport>：Redis启动时起作用；作用是建立复制关系，开启了该配置的Redis服务器在启动后成为从节点。该注释默认注释掉，即Redis服务器默认都是主节点。
* repl-timeout 60：与各个阶段主从节点连接超时判断有关，见前面的介绍。

**主节点相关配置**

* **repl-diskless-sync no：**作用于全量复制阶段，控制主节点是否使用diskless复制（无盘复制）。所谓diskless复制，是指在全量复制时，主节点不再先把数据写入RDB文件，而是直接写入slave的socket中，整个过程中不涉及硬盘；diskless复制在磁盘IO很慢而网速很快时更有优势。需要注意的是，截至Redis3.0，diskless复制处于实验阶段，默认是关闭的。
* **repl-diskless-sync-delay 5：**该配置作用于全量复制阶段，当主节点使用diskless复制时，该配置决定主节点向从节点发送之前停顿的时间，单位是秒；只有当diskless复制打开时有效，默认5s。之所以设置停顿时间，是基于以下两个考虑：a.向slave的socket的传输一旦开始，新连接的slave只能等待当前数据传输结束，才能开始新的数据传输；b.多个从节点有较大的概率在短时间内建立主从复制。
* **client-output-buffer-limit slave 256MB 64MB 60：**与全量复制阶段主节点的缓冲区大小有关，见前面的介绍。
* **repl-disable-tcp-nodelay no：**与命令传播阶段的延迟有关，见前面的介绍。
* **masterauth <master-password>：**与连接建立阶段的身份验证有关，见前面的介绍。
* **repl-ping-slave-period 10：**与命令传播阶段主从节点的超时判断有关，见前面的介绍。
* **repl-backlog-size 1mb：**复制积压缓冲区的大小，见前面的介绍。
* **repl-backlog-ttl 3600：**当主节点没有从节点时，复制积压缓冲区保留的时间，这样当断开的从节点重新连进来时，可以进行全量复制（默认3600s）。如果设置为0，则永远不会释放复制积压缓冲区。
* **min-slaves-to-write 3与min-slaves-max-lag 10：**规定了主节点的最小从节点数目及对应的最大延迟，见前面的介绍。

**从节点相关配置**

* **slave-serve-stale-data yes：**与从节点数据陈旧时是否响应客户端命令有关，见前面的介绍。
* **slave-read-only yes：**从节点是否只读；默认是只读的。由于从节点开启写操作容易导致主从节点的数据不一致，因此该配置尽量不要修改。

**6、单机内存大小限制**

我之前在[《Redis高可用详解：持久化技术及方案选择》](https://mp.weixin.qq.com/s?__biz=MzI4NTA1MDEwNg==&mid=2650769300&idx=1&sn=49a11efa1a6ee605fceaddf240a55c40&scene=21#wechat_redirect)中，讲到了fork操作对Redis单机内存大小的限制。实际上在Redis的使用中， 限制单机内存大小的因素非常多，下面总结一下在主从复制中，单机内存过大可能造成的影响：

**切主：**当主节点宕机时，一种常见的容灾策略是将其中一个从节点提升为主节点，并将其它从节点挂载到新的主节点上，此时这些从节点只能进行全量复制。如果Redis单机内存达到10GB，一个从节点的同步时间在几分钟的级别；如果从节点较多，恢复的速度会更慢；如果系统的读负载很高，而这段时间从节点无法提供服务，会对系统造成很大的压力。

**从库扩容：**如果访问量突然增大，此时希望增加从节点分担读负载，如果数据量过大，从节点同步太慢，难以及时应对访问量的暴增。

**缓冲区溢出：**切主和从库扩容都是从节点可以正常同步的情形（虽然慢），但是如果数据量过大，造成全量复制阶段主节点的复制缓冲区溢出，从而导致复制中断，则主从节点的数据同步会全量复制→复制缓冲区溢出导致复制中断→重连→全量复制→复制缓冲区溢出导致复制中断……的循环。

**超时：**如果数据量过大，全量复制阶段主节点fork+保存RDB文件耗时过大，从节点长时间接收不到数据触发超时，主从节点的数据同步同样可能陷入全量复制→超时导致复制中断→重连→全量复制→超时导致复制中断……的循环。

此外，主节点单机内存除了绝对量不能太大，其占用主机内存的比例也不应过大：最好只使用50%-65%的内存，留下30%-45%的内存用于执行bgsave命令和创建复制缓冲区等。

7、info Replication

在Redis客户端通过info Replication可以查看与复制相关的状态， 对于了解主从节点的当前状态，以及解决出现的问题都会有帮助。

主节点：

从节点：

对于从节点，上半部分展示的是其作为从节点的状态，从connectd\_slaves开始，展示的是其作为潜在的主节点的状态。

info Replication中展示的大部分内容在文章中都已经讲述，这里不再详述。

**七、总结**

下面回顾一下本文的主要内容：

**主从复制的作用：**宏观的了解主从复制是为了解决什么样的问题，即数据冗余、故障恢复、读负载均衡等。

**主从复制的操作：**即slaveof命令。

**主从复制的原理：**主从复制包括了连接建立阶段、数据同步阶段、命令传播阶段；其中数据同步阶段，有全量复制和部分复制两种数据同步方式；命令传播阶段，主从节点之间有PING和REPLCONF ACK命令互相进行心跳检测。

**应用中的问题：**包括读写分离的问题（数据不一致、数据过期、故障切换等） 、复制超时问题 、复制中断问题等，然后总结了主从复制相关的配置，其中repl-timeout、client-output-buffer-limit slave等对解决Redis主从复制中出现的问题可能会有帮助。

主从复制虽然解决或缓解了数据冗余、故障恢复、读负载均衡等问题，但其缺陷仍很明显：故障恢复无法自动化；写操作无法负载均衡；存储能力受到单机的限制；这些问题的解决，需要哨兵和集群的帮助，我将在后面的文章中介绍，欢迎关注。

**实现故障恢复自动化：详解Redis哨兵技术**

**一、作用和架构**

**1.作用:**

在介绍哨兵之前，首先从宏观角度回顾一下Redis实现高可用相关的技术。它们包括：持久化、复制、哨兵和集群，其主要作用和解决的问题是：

* **持久化：**持久化是最简单的高可用方法（有时甚至不被归为高可用的手段），主要作用是数据备份，即将数据存储在硬盘，保证数据不会因进程退出而丢失。
* **复制：**复制是高可用Redis的基础，哨兵和集群都是在复制基础上实现高可用的。复制主要实现了数据的多机备份，以及对于读操作的负载均衡和简单的故障恢复。缺陷是故障恢复无法自动化；写操作无法负载均衡；存储能力受到单机的限制。
* **哨兵：**在复制的基础上，哨兵实现了自动化的故障恢复。缺陷是写操作无法负载均衡；存储能力受到单机的限制。
* **集群：**通过集群，Redis解决了写操作无法负载均衡，以及存储能力受到单机限制的问题，实现了较为完善的高可用方案。

Redis Sentinel，即Redis哨兵，在Redis 2.8版本开始引入。哨兵的核心功能是主节点的自动故障转移。下面是Redis官方文档对于哨兵功能的描述：

* **监控（Monitoring）：**哨兵会不断地检查主节点和从节点是否运作正常。
* **自动故障转移（Automatic Failover）：**当主节点不能正常工作时，哨兵会开始自动故障转移操作，它会将失效主节点的其中一个从节点升级为新的主节点，并让其他从节点改为复制新的主节点。
* **配置提供者（Configuration Provider）：**客户端在初始化时，通过连接哨兵来获得当前Redis服务的主节点地址。
* **通知（Notification）：**哨兵可以将故障转移的结果发送给客户端。

其中，监控和自动故障转移功能，使得哨兵可以及时发现主节点故障并完成转移；而配置提供者和通知功能，则需要在与客户端的交互中才能体现。

**2.架构：**

典型的哨兵架构图如下所示：

它由两部分组成：

* **哨兵节点：**哨兵系统由一个或多个哨兵节点组成，哨兵节点是特殊的Redis节点，不存储数据。
* **数据节点：**主节点和从节点都是数据节点。

**4.总结**

哨兵系统的搭建过程，有几点需要注意：

* 哨兵系统中的主从节点，与普通的主从节点并没有什么区别，故障发现和转移是由哨兵来控制和完成的。
* 哨兵节点本质上是Redis节点。
* 每个哨兵节点，只需要配置监控主节点，便可以自动发现其他的哨兵节点和从节点。
* 在哨兵节点启动和故障转移阶段，各个节点的配置文件会被重写（Config Rewrite）。
* 本章的例子中，一个哨兵只监控了一个主节点；实际上，一个哨兵可以监控多个主节点，通过配置多条sentinel monitor即可实现。

**1.哨兵节点支持的命令**

哨兵节点作为运行在特殊模式下的Redis节点，其支持的命令与普通的Redis节点不同。在运维中，我们可以通过这些命令查询或修改哨兵系统；不过更重要的是，哨兵系统要实现故障发现、故障转移等各种功能，离不开哨兵节点之间的通信，而通信的很大一部分是通过哨兵节点支持的命令来实现的。下面介绍哨兵节点支持的主要命令：

**基础查询：**

通过这些命令，可以查询哨兵系统的拓扑结构、节点信息、配置信息等。

* info sentinel：获取监控的所有主节点的基本信息。
* sentinel masters：获取监控的所有主节点的详细信息。
* sentinel master mymaster：获取监控的主节点mymaster的详细信息。
* sentinel slaves mymaster：获取监控的主节点mymaster的从节点的详细信息。
* sentinel sentinels mymaster：获取监控的主节点mymaster的哨兵节点的详细信息。
* sentinel get - master - addr - by- name mymaster：获取监控的主节点mymaster的地址信息，前文已有介绍。
* sentinel is-master-down-by-addr：哨兵节点之间可以通过该命令询问主节点是否下线，从而对是否客观下线做出判断。

**增加/移除对主节点的监控：**

sentinel monitor mymaster2 192.168.92.128 16379 2：与部署哨兵节点时配置文件中的sentinel monitor功能完全一样，不再详述。

sentinel remove mymaster2：取消当前哨兵节点对主节点mymaster2的监控。

**强制故障转移：**

sentinel failover mymaster：该命令可以强制对mymaster执行故障转移，即便当前的主节点运行完好；例如，如果当前主节点所在机器即将报废，便可以提前通过failover命令进行故障转移。

**基本原理:**

关于哨兵的原理，关键是了解以下几个概念：

**定时任务：**每个哨兵节点维护了3个定时任务。定时任务的功能分别如下：通过向主从节点发送info命令获取最新的主从结构；通过发布订阅功能获取其他哨兵节点的信息；通过向其他节点发送ping命令进行心跳检测，判断是否下线。

**主观下线：**在心跳检测的定时任务中，如果其他节点超过一定时间没有回复，哨兵节点就会将其进行主观下线。顾名思义，主观下线的意思是一个哨兵节点“主观地”判断下线；与主观下线相对应的是客观下线。

**客观下线：**哨兵节点在对主节点进行主观下线后，会通过sentinel is-master-down-by-addr命令询问其他哨兵节点该主节点的状态；如果判断主节点下线的哨兵数量达到一定数值，则对该主节点进行客观下线。

需要特别注意的是，客观下线是主节点才有的概念；如果从节点和哨兵节点发生故障，被哨兵主观下线后，不会再有后续的客观下线和故障转移操作。

**选举领导者哨兵节点：**当主节点被判断客观下线以后，各个哨兵节点会进行协商，选举出一个领导者哨兵节点，并由该领导者节点对其进行故障转移操作。

监视该主节点的所有哨兵都有可能被选为领导者，选举使用的算法是Raft算法；Raft算法的基本思路是先到先得：即在一轮选举中，哨兵A向B发送成为领导者的申请，如果B没有同意过其他哨兵，则会同意A成为领导者。选举的具体过程这里不做详细描述，一般来说，哨兵选择的过程很快，谁先完成客观下线，一般就能成为领导者。

**故障转移：**选举出的领导者哨兵，开始进行故障转移操作，该操作大体可以分为3个步骤：

* 在从节点中选择新的主节点：选择的原则是，首先过滤掉不健康的从节点；然后选择优先级最高的从节点（由slave-priority指定）；如果优先级无法区分，则选择复制偏移量最大的从节点；如果仍无法区分，则选择runid最小的从节点。
* 更新主从状态：通过slaveof no one命令，让选出来的从节点成为主节点；并通过slaveof命令让其他节点成为其从节点。
* 将已经下线的主节点（即6379）设置为新的主节点的从节点，当6379重新上线后，它会成为新的主节点的从节点。

**配置与实践建议**

**1.配置**

下面介绍与哨兵相关的几个配置。

**配置1：**sentinel monitor {masterName} {masterIp} {masterPort} {quorum}

sentinel monitor是哨兵最核心的配置，在前文讲述部署哨兵节点时已说明，其中：masterName指定了主节点名称，masterIp和masterPort指定了主节点地址，quorum是判断主节点客观下线的哨兵数量阈值：当判定主节点下线的哨兵数量达到quorum时，对主节点进行客观下线。建议取值为哨兵数量的一半加1。

**配置2：**sentinel down-after-milliseconds {masterName} {time}

sentinel down-after-milliseconds与主观下线的判断有关：哨兵使用ping命令对其他节点进行心跳检测，如果其他节点超过down-after-milliseconds配置的时间没有回复，哨兵就会将其进行主观下线。该配置对主节点、从节点和哨兵节点的主观下线判定都有效。

down-after-milliseconds的默认值是30000，即30s；可以根据不同的网络环境和应用要求来调整：值越大，对主观下线的判定会越宽松，好处是误判的可能性小，坏处是故障发现和故障转移的时间变长，客户端等待的时间也会变长。例如，如果应用对可用性要求较高，则可以将值适当调小，当故障发生时尽快完成转移；如果网络环境相对较差，可以适当提高该阈值，避免频繁误判。

**配置3：**sentinel parallel - syncs {masterName} {number}

sentinel parallel-syncs与故障转移之后从节点的复制有关：它规定了每次向新的主节点发起复制操作的从节点个数。例如，假设主节点切换完成之后，有3个从节点要向新的主节点发起复制；如果parallel-syncs=1，则从节点会一个一个开始复制；如果parallel-syncs=3，则3个从节点会一起开始复制。

parallel-syncs取值越大，从节点完成复制的时间越快，但是对主节点的网络负载、硬盘负载造成的压力也越大；应根据实际情况设置。例如，如果主节点的负载较低，而从节点对服务可用的要求较高，可以适量增加parallel-syncs取值。parallel-syncs的默认值是1。

**配置4：**sentinel failover - timeout {masterName} {time}

sentinel failover-timeout与故障转移超时的判断有关，但是该参数不是用来判断整个故障转移阶段的超时，而是其几个子阶段的超时，例如如果主节点晋升从节点时间超过timeout，或从节点向新的主节点发起复制操作的时间（不包括复制数据的时间）超过timeout，都会导致故障转移超时失败。

failover-timeout的默认值是180000，即180s；如果超时，则下一次该值会变为原来的2倍。

**配置5：**除上述几个参数外，还有一些其他参数，如安全验证相关的参数，这里不做介绍。

**2.实践建议**

* 哨兵节点的数量应不止一个。一方面增加哨兵节点的冗余，避免哨兵本身成为高可用的瓶颈；另一方面减少对下线的误判。此外，这些不同的哨兵节点应部署在不同的物理机上。
* 哨兵节点的数量应该是奇数，便于哨兵通过投票做出“决策”：领导者选举的决策、客观下线的决策等。
* 各个哨兵节点的配置应一致，包括硬件、参数等；此外，所有节点都应该使用ntp或类似服务，保证时间准确、一致。
* 哨兵的配置提供者和通知客户端功能，需要客户端的支持才能实现，如前文所说的Jedis；如果开发者使用的库未提供相应支持，则可能需要开发者自己实现。
* 当哨兵系统中的节点在Docker（或其他可能进行端口映射的软件）中部署时，应特别注意端口映射可能会导致哨兵系统无法正常工作，因为哨兵的工作基于与其他节点的通信，而Docker的端口映射可能导致哨兵无法连接到其他节点。例如，哨兵之间互相发现，依赖于它们对外宣称的IP和port，如果某个哨兵A部署在做了端口映射的Docker中，那么其他哨兵使用A宣称的port无法连接到A。

**六、总结**

本文首先介绍了哨兵的作用：监控、故障转移、配置提供者和通知；然后讲述了哨兵系统的部署方法，以及通过客户端访问哨兵系统的方法；再然后简要说明了哨兵实现的基本原理；最后给出了关于哨兵实践的一些建议。

在主从复制的基础上，哨兵引入了主节点的自动故障转移，进一步提高了Redis的高可用性；但是哨兵的缺陷同样很明显：哨兵无法对从节点进行自动故障转移，在读写分离场景下，从节点故障会导致读服务不可用，需要我们对从节点做额外的监控、切换操作。

此外，哨兵仍然没有解决写操作无法负载均衡、及存储能力受到单机限制的问题；这些问题的解决需要使用集群，欢迎关注社群后续内容。