## 简介

<http://bboyjing.github.io/categories/Redis/>

Redis是一个基于内存，也可以基于磁盘持久化的非关系型数据库。和memcached相似，但是其支持更复杂的数据结构，如list set sorted set，同时redis具有持久化的功能。

Redis将数据存储在内存中，这样可以有高速的读写性能，并且通过异步的方式将数据写入磁盘。如果数据不在内存中，磁盘IO速度会是一个瓶颈。

key value store（键值存储）：是一个以键值形式存储的数据库，用来作为唯一的存储系统，同时借助sentinel实现一定意义上的高可用。

Memory cached（内存缓存）：是一个把数据存储在内存中的高速缓存，在应用中用来实现高效的响应用户请求。

Data structure server（数据结构服务）：支持对复杂数据库结构的高速操作，如list string hash set stored等，提供某种特殊业务操作。

## Redis优势

1 丰富的操作 如hash list set stored sets

2 内建复制(replication)及集群(cluster)

3 就地更新操作，而无需停机重启生效

4 支持持久化缓存，常用的有RDB和AOF

## 内存回收

C语言不具备自动内存回收功能，所以redis在自己的对象系统中构建了一个引用计数技术实现内存回收机制。通过这一机制，程序可以通过跟踪对象的引用计数信息，在适当时候自动释放对象并进行内存回收。

在创建一个新对象时，引用计数的值会被初始化为1

当对象被一个新程序使用时，它的引用计数值会被增一

当对象不再被一个程序使用时，它的引用计数值会被减一

当对象的引用计数值变为0时，对象所占用的内存会被释放

## 对象共享

对象的引用计数还有对象共享的作用。

多个键共享同一个值对象需要两个操作：

1 将数据库键的值指针指向一个现有的值对象

2 将被共享的值对象引用计数增1

当将一个共享对象设置为键的值对象时，程序需要先检查给定的共享对象和键想创建的目标对象是否完全相同，只有在共享对象和目标对象完全相同的情况下，程序才会将共享对象用作键的值对象。而一个共享对象保存的值越复杂，验证共享对象和目标对象是否相同所需的复杂度就会越高，消耗的cpu时间也会越多。

如果共享对象是保存整数值的字符串对象，那么验证操作的复杂度O(1)

如果共享对象是保存字符串值的字符串对象，那么验证操作的复杂度O(N)

如果共享对象是包含了多个值（或者对象）对象，比如列表对象或者哈希对象，那么验证操作的复杂度将会是O(N2)

尽管共享对象可以节约更多的内存，但受到cpu时间的限制，redis只对包含整数值的字符串对象进行共享。

## 工作原理

Redis工作时，将启动一个fork函数创建一个父进程，复制当前进程，存为副本，父进程接受并处理客户端请求，而子进程则将内存中的数据文件写入磁盘中的临时文件，当子进程完成所有的写入操作时，会将原来的文件替换为最新生成的临时文件。

## redis持久性介绍

redis的持久性有两种：RDB(Redis database)和AOF(append only file)。RDB和AOF可以同时使用，但是BGSAVE和BGWRITEAOF不会同时执行，在redis服务器启动用于恢复数据时，将会优先使用AOF。持久的功能是用于恢复，但是持久的功能不能取代备份，还应该制定备份策略，对redis进行数据库备份，保证数据的完整性。

RDB持久化方式能够在指定的时间间隔能对你的数据进行快照存储。   
AOF持久化方式记录每次对服务器写的操作，当服务器重启的时候会重新执行这些命令来恢复原始的数据，AOF命令以redis协议追加保存每次写的操作到文件末尾。Redis还能对AOF文件进行后台重写，使得AOF文件的体积不至于过大。   
如果你只希望你的数据在服务器运行的时候存在，你也可以不使用任何持久化方式。   
你也可以同时开启两种持久化方式,，在这种情况下，当redis重启的时候会优先载入AOF文件来恢复原始的数据,因为在通常情况下AOF文件保存的数据集要比RDB文件保存的数据集要完整。

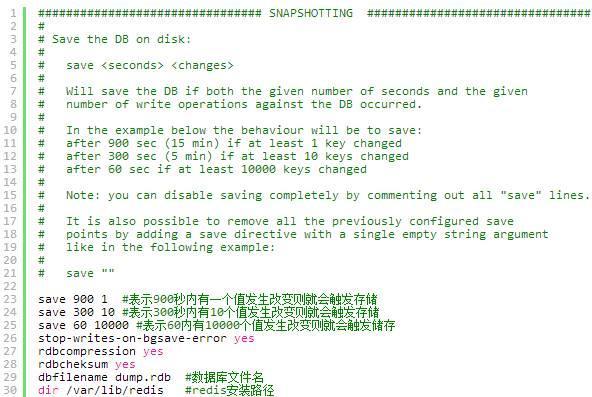
Redis默认开启的RDB存储方式。

RDB：在指定的时间段内将内存中的数据快照到磁盘中。当redis启动时，会根据快照文件来将数据恢复到内存中。（默认的）

在redis内部有一个定时器，每隔固定时间去检查当前数据发生改变的次数与时间是否满足配置的持久性触发的条件，如果满足则通过操作系统启动一个fork函数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。当持久化过程结束了，就会使用这个刚刚生成的临时文件替换上次的临时文件。

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。适应于需要大规模的数据恢复，且对数据恢复不是很敏感

redis.conf文件：



AOF：是以日志的形式来记录每个写操作，将redis执行过的所有写指令记录下来（不记录读操作）。（记录会导致数据发生修改的指令）

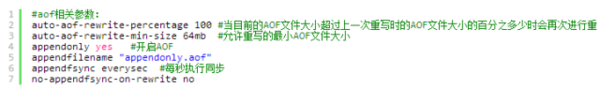
redis主进程通过fork创建子进程，子进程根据redis内存中的数据库重构后将此存储于临时文件中，父进程接受客户端的请求，并会把这些请求中的操作继续**追加**至原来的AOF文件，额外的这些新的写请求还会被放置于一个缓冲队列中，父进程把缓冲中的命令写到临时文件中，子进程重写完成会通知父进程，父进程用临时文件替换原来的AOF老文件。

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，会读取该文件重新构建数据，换言之，redis 重启的话就根据日志文件的内容将写指令从前到后执行一次以完成数据的恢复工作。 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。ROF文件远大于RDB文件。

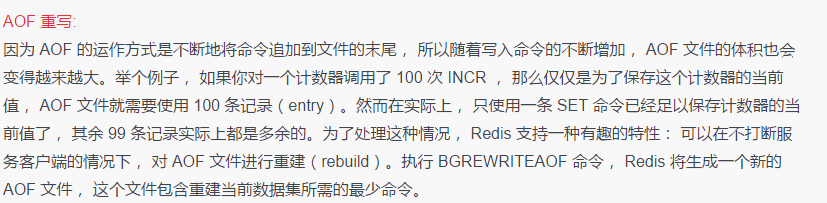
优点：每秒同步，每次修改同步。

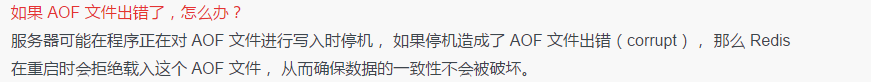
AOF对日志文件的写入操作时采用追加的模式进行，因此写入的过程中如果发生断电，机器宕机等情况发生，也不会对已存在数据文件造成破坏。

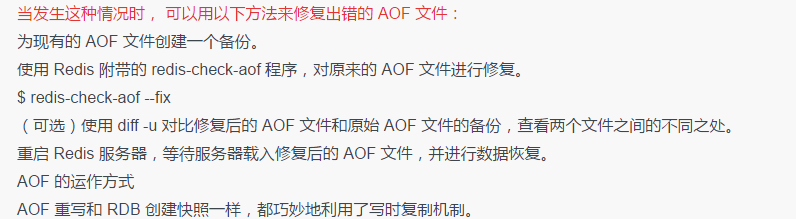


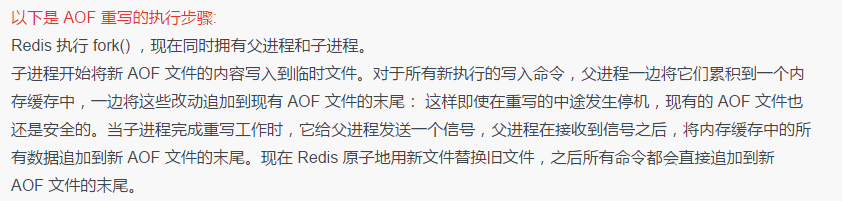
AOF采用文件追加的方式，文件会越来越大，为避免出现此种情况，新增了重写机制。当AOF文件的大小超过所设定的阈值时，redis就会启动aof文件的内容压缩。

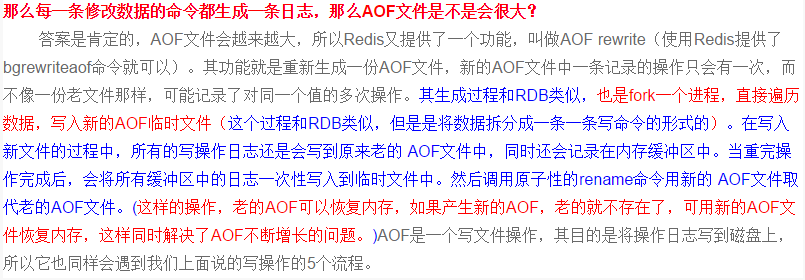
重写原理：AOF文件持续增长而过大时，会fork出一个新的进程来将文件重写（也就是先写临时文件最后再rename），遍历新进程的内存中数据，每条记录有一条set语句。重写AOF文件的操作，并没有读取旧的AOF文件，而是将整个内存中的数据库内容用命令的方式重写了一个新的AOF文件。













同时开启两种方式，优先使用AOF方式。

优化AOF文件：

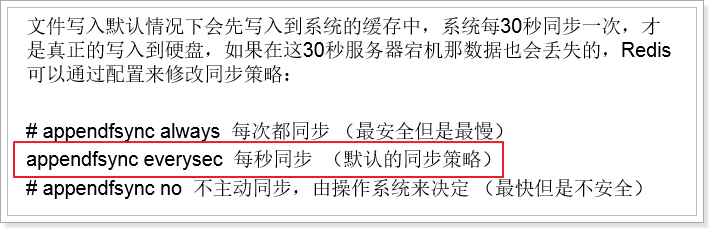


目的：去除数据的中间执行过程，保留最终数据命令即可。

Aof文件重写策略



文件同步策略



注意：即使每秒做文件同步也可能导致数据丢失。

## 出现的问题

有Redis线上运维经验的人会发现Redis在物理内存使用比较多，但还没有超过实际物理内存总容量时就会发生不稳定甚至崩溃的问题，有人认为是基于快照方式持久化的fork系统调用造成内存占用加倍而导致的，这种观点是不准确的，因为fork用的copy-on-write机制是基于操作系统页这个单位的，也就是只有有写入的脏页会被复制，但是一般你的系统不会在短时间内所有的页都发生了写入而导致复制，那么是什么原因导致Redis崩溃的呢？

答案是Redis的持久化使用了Buffer IO造成的，所谓Buffer IO是指Redis对持久化文件的写入和读取操作都会使用物理内存的Page Cache,而大多数数据库系统会使用Direct IO来绕过这层Page Cache并自行维护一个数据的Cache，而当Redis的持久化文件过大(尤其是快照文件)，并对其进行读写时，磁盘文件中的数据都会被加载到物理内 存中作为操作系统对该文件的一层Cache,而这层Cache的数据与Redis内存中管理的数据实际是重复存储的，虽然内核在物理内存紧张时会做 Page Cache的剔除工作，但内核很可能认为某块Page Cache更重要，而让你的进程开始Swap ,这时你的系统就会开始出现不稳定或者崩溃了。我们的经验是当你的Redis物理内存使用超过内存总容量的3/5时就会开始比较危险了。

## BGSAVE和BGrewriteaof阻塞问题

Redis有两种方式的持久化，分别是: 快照方式(Snapshotting)和AOF方式。如果一个Resdis服务器采用了AOF持久化方式（或者AOF和Snapshotting共存），当执行bgsave或者bgrewriteaof操作时，会导致Redis被阻塞。此时，Redis无法提供任何读取或者写入操作。（注意：Redis的主进程是单线程的工作模式哦~）

以bgrewriteaof为例，从字面上来理解，bgwriteaof是在后台由子进程来执行操作的，按理说是不会影响到Redis的主进程响应读写请求。

原理确实如此，Redis不希望AOF重写会造成服务器无法处理请求，所以Redis决定将AOF重写程序放到（后台）子进程中执行，这样处理的最大好处是：

1、子进程进行AOF重写期间，主进程可以继续处理命令请求；

2、子进程带有主进程的数据副本，使用子进程而不是线程，可以避免在锁的情况下，保证数据的安全性。

子进程对AOF进程重写，并且持久化的存储设备（如硬盘），应该来说是不会影响到主进程正常提供服务的。   可是，问题就出在硬盘上。

Redis开启了AOF持久化，往往会在配置文件中设置appendfsync everysec参数，该参数表示主进程会每秒调用fsync()函数，执行同步操作将数据同步到硬盘。如果此时子进程也在写入硬盘，会导致主进程的fsync()/write()操作被阻塞，从而导致Redis主进程被阻塞。

解决方法便是设置 no-appendfsync-on-rewrite yes，在子进程进行AOF重写和写入硬盘时，主进程不调用fsync()操作。值得注意的是，即使主进程不进行fsync()操作，操作系统内核也会根据自己的算法在适当的时机将数据写入到硬盘，这同样会导致IO阻塞。

一个服务器往往会跑多个Redis实例，虽然Redis A并没有进行rewriteaof，主进程进行fsync()，其他实例如果正在rewriteaof，同样会导致Redis A的fsync()阻塞。



## RDB压缩



压缩：

优点：减少磁盘存储空间

缺点：消耗CPU资源

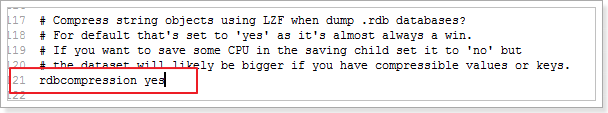
不压缩：

优点：不消耗CPU资源

缺点：占用磁盘空间多

如何选择？ 看需求、看服务器资源情况。

Redis默认是开启压缩的，配置：



## 大数据存在的问题

当redis中存储的数据量只有几个GB时，使用快照保存数据是没问题的，redis会创建子进程并将数据保存到磁盘中，生成快照的时间很快。但是随着redis占用的内存越来越多，bgsave在创建子进程时耗费的时间也会越来越多。如果redis占用的内存量有几十GB，并且剩余的空闲内存并不多，或者redis运行在虚拟机上，那么执行bgsave可能会导致系统长时间停顿。也可能引发系统大量的使用虚拟内存，从而导致redis性能降低甚至无法使用。

为了防止redis因创建子进程而出现停顿，可以考虑关闭 自动保存，转而通过手动触发bgsave或者save进行持久化。手动发送bgsave也会产生停顿，但是用户可以通过手动bgsave命令控制停顿出现的时间。Save命令会阻塞redis直至快照生成完毕，但是它不需要创建子进程，所以不会像bgsave一样因为创建子进程而导致redis停顿，因为没有子进程在争抢资源，所以save创建快照的速度会比bgsave创建的速度快。

## Redis时延分析

所以，Redis的事件循环是在单线程中进行的，因此要确保快速的进行事件处理，这样事件循环中的后续任务才不会被阻塞。

在Redis中影响时延的场景主要有三种：

* 耗时长的命令造成的阻塞；
* fork产生的阻塞；
* 持久化造成的阻塞；

### 耗时长的命令造成的阻塞

耗时长的命令主要包括keys、sort、smembers等。

对于keys命令，它是用于查找所有符合给定模式pattern的key，时间复杂度为O(N)， N为数据库中key的数量。当数据库中的个数达到千万时，这个命令会造成读写线程阻塞数秒。类似的命令有sunion sort等操作。

解决办法：

将处理快的请求和处理慢的请求分离，否则慢的影响到了快的；因为redis中的操作都是基于内存的，并且epoll是非阻塞的，这样可以把这些操作放置在一个单线程中完成，而对于持久化、AOF重写以及Master-slave同步数据这些耗时的操作就创建一个新进程来处理。   
同样对于keys这样的耗时操作，可将它分离出去，比如单独使用一个redis从节点专门用于keys、sort等耗时操作，通常这些查询一般不会是线上的实时业务，对时延要求不高。

**对于smembers命令**，它是用于获取集合全集，时间复杂度为O(N)，N为集合中的数量。如果一个集合中保存了千万级的数据，一次性取回也会造成事件处理线程的长时间阻塞。

**解决方案：**  
和sort、keys等命令不一样，smembers是线上实时应用场景中使用频率非常高的一个命令，可以从设计层面来考虑： 可以控制集合的数量。比如原来使用一个键来存储一年的记录，数据量大，我们可以使用12个键来分别保存12个月的记录，或者365个键来保存每一天的记录，将集合的规模控制在可接受的范围。

如果不容易将集合划分为多个子集合，而坚持用一个大集合来存储，那么在取集合的时候可以考虑使用SRANDMEMBER key [count]随机返回集合中的指定数量，当然如果要遍历集合中的所有元素，这个命令就不适合了。

**对于save命令，**使用事件处理线程进行数据的持久化，当数据量大的时候，会造成线程长时间阻塞，整个redis被block。save阻塞了事件处理的线程，我们甚至无法使用redis-cli查看当前的系统状态，造成“何时保存结束，目前保存了多少”这样的信息都无从得知。

**解决方案：**  
通过使用bgsave代替save来进行持久化。

### fork产生的阻塞

在Redis中可以通过新建一个子进程来执行耗时的操作，比如用于数据持久化保存bgsave命令： 开启RDB持久化后，当达到持久化的阈值，redis会fork一个新的子进程来做持久化，采用了操作系统的copy-on-wirte写时复制策略，子进程与父进程共享Page。如果父进程的Page（每页4K）有修改，父进程自己创建那个Page的副本，不会影响到子进程。  
另外在fork子进程时，虽然可共享的数据内容不需要复制，但会复制之前进程空间的内存页表，如果内存空间有40G（考虑每个页表条目8个字节），那么页表大小就有80M，这个复制会消耗大量时间。

类似的，以下这些操作都会fork子进程：

* Master向Slave首次同步数据：当Master节点接收到Slave节点的syn同步请求，会fork一个新的子进程，将内存数据dump到文件上，然后再同步到Slave节点中；
* AOF日志重写：使用AOF持久化方式，做AOF文件重写操作会创建新的进程去完成；

解决方案：

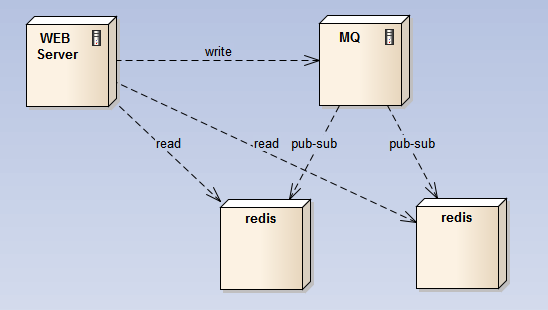
解决大内存页表复制时带来的影响：

1 控制每个redis实例的最大内存量， 从内存量上控制fork的时延； 一般建议不超过20G，根据自己服务器性能来确定（内存越大，持久化的时间越长，复制页表的时间越长，对事件循环的阻塞就延长） 。

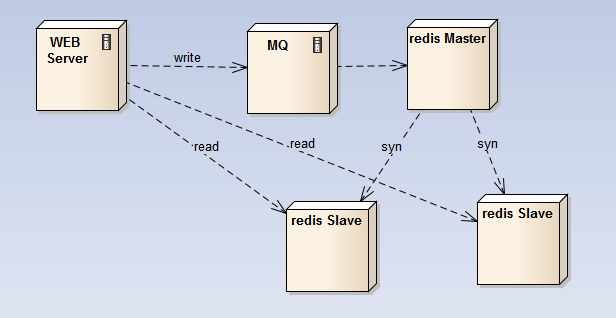
2 使用大内存页，默认内存页使用4KB，这样当使用40G的内存时，页表就有80M，而将每个内存页扩大到4M，页表就只有80K，这样复制页表几乎没有阻塞，同时也会提高快速页表缓冲TLB的命中率，但大内存页也有问题，在写时复制时，只要一个页快中任何一个元素被修改，这个页块都需要复制一份（COW机制的粒度是页面），这样在写时复制期间，会耗用更多的内存空间。

3 避免fork新进程，不使用持久化，不在主节点上进行查询：

只用单机，不开持久化，不挂slave节点（但这样的方案只适合缓存）,没有新的进程产生；另外为实现高可用，可以在写redis的前端挂上一个消息队列，在消息队列中使用pub-sub来做分发，保证每个写操作至少落到2个节点上；因为所有节点中数据相同，只需要用一个节点做持久化，并且该节点不对外提供查询。



master-slave：在主节点上开持久化，主节点不对外提供查询，查询由slave节点提供，从节点不提供持久化；这样所有fork耗时的操作都在主节点上，而查询请求由slave节点提供； 为解决主节点down机，因为主节点不具有可替代性，redis集群对外就只提供读，待主节点启动后，再继续更新操作；对于之前的更新操作，可以用MQ缓存起来，等主节点恢复后完成故障期间的写请求。



如果使用官方的Sentinel将从升级为主，整体实现就相对复杂了；需要更改可用从的ip配置，将其从可查询结点中剔除，让前端的查询负载不再落在新主上；然后，才能放开sentinel的切换操作，这个前后关系需要保证；

### 持久化造成的阻塞

执行持久化（AOF/RDB）对系统性能有较大影响，特别是服务器节点上还有其它读写磁盘的操作时（比如应用服务和redis服务部署在相同节点上，应用服务实时记录进出报日志）；应尽可能避免在IO密集节点上开Redis持久化。

**子进程持久化时的write和父进程的fsync冲突造成的阻塞。**

在开启了AOF持久化的节点上，当子进程执行AOF重写或者RDB持久化时，出现了Redis查询卡顿甚至长时间阻塞的问题,，此时Redis无法提供任何读写操作：

原因分析： Redis 服务设置了appendfsync everysec，主进程每秒钟便会调用fsync(),，要求内核将数据”确实”写到存储硬件里，但由于服务器正在进行大量IO操作, 导致主进程 fsync操作被阻塞,，最终导致Redis主进程阻塞。

**解决方案：**  
设置 no-appendfsync-on-rewrite yes， 在子进程执行AOF重写时,，主进程不调用fsync()操作；注意，即使进程不调用 fsync(),，系统内核也会根据自己的**算法**在适当的时机将数据写到硬盘（**Linux**默认最长不超过30秒）。这个设置带来的问题是当出现故障时，最长可能丢失超过30秒的数据，而不再是1秒。

**子进程进行AOF重写时，系统的sync造成主进程write阻塞：**

* 有大量IO操作write， 但未主动调用同步操作，造成kernel buffer中有大量脏数据；
* 系统同步时，sync的同步时间过长，造成redis的写aof日志write操作阻塞；
* 造成单线程的redis的下一个事件无法处理，整个redis阻塞。

**解决方案：**  
控制系统sync调用的时间，需要同步的数据多时，耗时就长；缩小这个耗时，控制每次同步的数据量；通过配置按比例（vm.dirty\_background\_ratio）或按值（vm.dirty\_bytes）设置sync的调用阈值（一般设置为32M同步一次）；2.6.12以后，AOF rewrite 32M时会主动调用fdatasync。

另外，Redis当发现当前正在写的文件有在执行fdatasync时，就先不调用write，只存在cache里，免得被block。但如果已经超过两秒都还是这个样子，则会强行执行write，即使redis会被block住。

**AOF重写完成后合并数据时造成的阻塞**：在bgrewriteaof过程中，所有新来的写入请求依然会被写入旧的AOF文件，同时放到AOF buffer中，当rewrite完成后，会在主线程把这部分内容合并到临时文件中之后才rename成新的AOF文件，所以rewrite过程中会不断打印"Background AOF buffer size: 80 MB， Background AOF buffer size: 180 MB"，要监控这部分的日志。这个合并的过程是阻塞的，如果产生了280MB的buffer，在100MB/s的传统硬盘上，Redis就要阻塞2.8秒。

**解决方案：**  
将硬盘设置的足够大，将AOF重写的阈值调高，保证高峰期间不会触发重写操作，在闲时使用crontab 调用AOF重写命令。

## Fork和COW机制

### 进程简介

进程的数据结构：

进程由三部分组成:PCB、程序段和数据段。

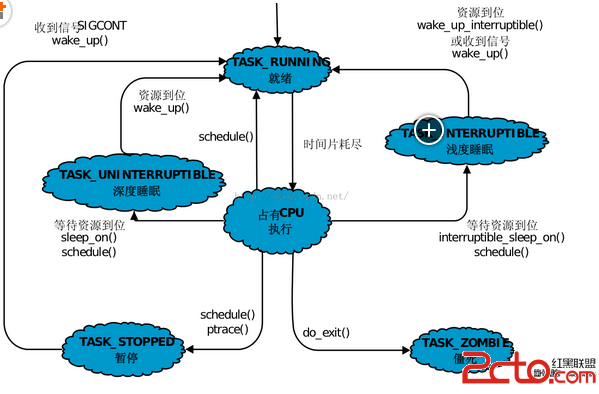
进程控制块PCB：用于描述进程情况及控制进程运行所需的全部信息。

代码段：是进程中能被进程调度程序在CPU上执行的程序代码段。

数据段：一个进程的数据段，可以是进程对应的程序加工处理的原始数据，也可以是程序执行后产生的中间或最终数据

进程和程序区别：进程是动态的，而程序是静态的。（进程存在的唯一标志：PCB，CPU通过PCB来控制进程） 进程的生命周期相对短暂，而程序是永久的。一个进程只能对应一个程序，一个程序可以对应多个进程。

进程三种状态：就绪 运行 阻塞 阻塞以后不能直接执行，必须进入就绪状态。



每个进程都会被分配一个独一无二的数字编号，称为 进程标识PID 2到32768

进程0：linux引导中创建的第一个进程，完成加载系统后，演变为进程调度 交换和存储管理进程 空闲进程

进程1：init进程，由0进程创建，完成系统的初始化，是系统中所有其他用户进程的祖先进程

创建进程大体步骤：

1 为新创建的进程分配一个内部标识，在内核中建立进程的结构

2 复制父进程的环境

3 为进程分配资源，包括进程映像需要的所有元素（程序 数据 用户栈）

4 复制父进程地址空间的内容到该进程地址空间中

5 置该进程的状态为就绪，插入就绪队列

进程撤销：进程终止时，操作系统做以下工作

1 关闭软中断：因为进程即将终止而不再处理任何软中断信号

2 回收资源：释放进程分配的所有资源，如关闭已经打开的文件，释放进程相应的数据结构

3 写记账信息：将进程在运行过程中所产生的记账数据（其中包括进程运行时的各种统计信息）记录到一个全局记账文件中

4 置该进程为僵死状态：向父进程发送子进程死的软中断信号，将终止信息status送到指定的存储单元中

5 转进程调度：因此此时cpu已经释放，需要由进程调度进程cpu再分配

### Fork系统调用

复制一个进程镜像

使用fork函数得到的子进程从父进程的继承了整个进程的地址空间，包括：进程上下文、进程堆栈、内存信息、打开的文件描述符、信号控制设置、进程优先级、进程组号、当前工作目录、根目录、资源限制、控制终端等。

子进程与父进程的区别：

1、父进程设置的锁，子进程不继承

2、各自的进程ID: 父子进程ID不同

3、子进程的未决警告被清除；

4、子进程的未决信号集设置为空集;

Fork函数有两个返回值，如果成功创建一个子进程，则对于父进程则返回子进程ID，对于子进程则返回0. 如果创建失败，则返回-1. 也就是说，一次调用两次返回。两次返回，是在各自的进程空间中返回的。（用户可以通过返回值判断哪个是父进程哪个是子进程）

子进程和父进程有各自的内存空间。（fork：代码段 数据段 堆栈段 PCB进程块的copy）

子进程对count的改变不会影响到父进程，因为它们有自己的数据段。

对于每个进程来说，都有正文段 数据段 堆和栈四个部分。内核会为这四个部分提供相应的物理内存，与进程的虚拟地址空间相映射。

### COW机制

在Linux程序中，fork()会产生一个和父进程完全相同的子进程，但子进程在此后多会exec系统调用，出于效率考虑，Linux中引入了“写时复制“技术，**也就是只有进程空间的各段的内容要发生变化时，才会将父进程的内容复制一份给子进程。**

（对于fork来讲，有一个很讨厌的东西叫exec系列的系统调用，它会勾引子进程另起炉灶。如果创建子进程就要内存拷贝的的话，一执行exec，辛辛苦苦拷贝的内存又被完全放弃了。由于fork()后会产生一个和父进程完全相同的子进程，但子进程在此后多会exec系统调用，处于效率考虑，linux中引入了“写时复制技术-Copy-On-Write”。）

那么子进程的物理空间没有代码，怎么去取指令执行exec系统调用呢？

 在fork之后exec之前两个进程用的是相同的物理空间（内存区），子进程的代码段、数据段、堆栈都是指向父进程的物理空间，也就是说，两者的虚拟空间不同，但其对应的物理空间是同一个。当父子进程中有更改相应段的行为发生时，再为子进程相应的段分配物理空间，如果不是因为exec，内核会给子进程的数据段、堆栈段分配相应的物理空间（至此两者有各自的进程空间，互不影响），而代码段继续共享父进程的物理空间（两者的代码完全相同）。而如果是因为exec，由于两者执行的代码不同，子进程的代码段也会分配单独的物理空间。

现在有一个父进程P1，这是一个主体，那么它是有灵魂也就身体的。现在在其虚拟地址空间（有相应的数据结构表示）上有：正文段，数据段，堆，栈这四个部分，相应的，内核要为这四个部分分配各自的物理块。即：正文段块，数据段块，堆块，栈块。

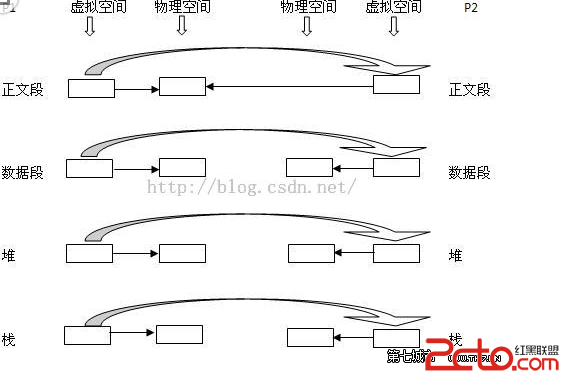
1. 现在P1用fork()函数为进程创建一个子进程P2，

内核：

 （1）复制P1的正文段，数据段，堆，栈这四个部分，注意是其内容相同。

 （2）为这四个部分分配物理块，P2的：正文段－＞P1的正文段的物理块，其实就是不为P2分配正文段块，让P2的正文段指向P1的正文段块，数据段－＞P2自己的数据段块（为其分配对应的块），堆－＞P2自己的堆块，栈－＞P2自己的栈块。

如下图所示：从左到右大的方向箭头表示复制内容。

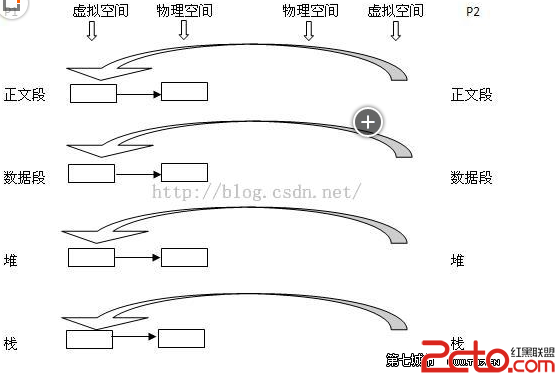


**p1调用fork()创建子进程p2时，p2拥有p1的副本，包括虚拟地址空间和相映射的物理内存，除了p1和p2共享正文代码段之外，其它的都是在物理级别上互相独立的。（但是有了COW机制后，就不会这样做了。对于数据段 堆栈，刚开始时，也是和正文段一样，都会指向父进程的物理空间，也就是说两者的虚拟空间不同，但是对应同一个物理空间）**

2.写时复制技术：内核只为新生成的子进程创建虚拟空间结构，它们复制于来自父进程的虚拟空间结构，但是不为这些段分配物理内存，它们共享父进程的物理空间，**当父子进程中有更改相应段的行为发生时，再为子进程相应的段分配物理空间。**

3. vfork()：这个做法更加火爆，内核连子进程的虚拟地址空间结构也不创建了，直接共享了父进程的虚拟空间，当然了，这种做法就顺水推舟的共享了父进程的物理空间。

父、子进程共享同一段区域，两者都进行了修改，修改操作并没有发生副本的生成，而是修改同一个内存区域。另外有一点，vfork总是保证让子进程先执行，在子进程调用exec或者exit之后父进程才能被调度执行。假设有一种情况是，子进程在调用上面两个函数之前依赖于父进程的进一步动作，则会导致死锁，因为父进程在等待子进程独立，而子进程独立需要父进程执行。



### 总结

传统的fork()系统调用直接把所有的资源复制给新创建的进程。这种实现过于简单并且效率低下，因为它拷贝的数据也许并不共享，更糟的情况是，如果新进程打算立即执行一个新的映像，那么所有的拷贝都将前功尽弃。Linux的fork()使用写时拷贝（copy-on-write）页实现。写时拷贝是一种可以推迟甚至免除拷贝数据的技术。内核此时并不复制整个进程地址空间，而是让父进程和子进程共享同一个拷贝。只有在需要写入的时候，数据才会被复制，从而使各个进程拥有各自的拷贝。也就是说，资源的复制只有在需要写入的时候才进行，在此之前，只是以只读方式共享。这种技术使地址空间上的页的拷贝被推迟到实际发生写入的时候。在页根本不会被写入的情况下{举例来说:fork()后立即调用exec()}它们就无需复制了。fork()的实际开销就是复制父进程的页表以及给子进程创建惟一的进程描述符。在一般情况下，进程创建后都会马上运行一个可执行的文件，这种优化可以避免拷贝大量根本就不会被使用的数据（地址空间里常常包含数十兆的数据）。由于Unix强调进程快速执行的能力，所以这个优化是很重要的。这里补充一点：Linux COW与exec没有必然联系。

## Redis主从复制

一个master可以有多个slave，支持链式复制，master以非阻塞的方式同步数据至slave。

启动一个slave后，slave会向主发送同步命令，请求同步主库上的数据，master将启动一个后台的子进程，将数据快照保存至在数据文件中，把数据文件发送给slave，slave将数据文件保存至本地中，在本地重建数据库后载入内存，同步完成。

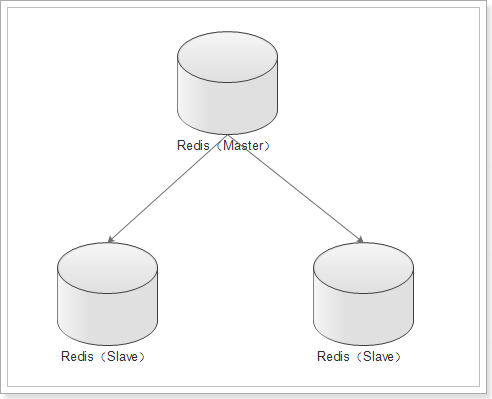
**可以将主库的持久化功能关闭，由从库负责持久化操作。减轻主库压力。**

### 主从复制好处

1 避免单点故障

2构建读写分离架构，满足读多写少的应用场景

### 主从架构



数据从主库同步到从库

### 设置主从

在redis中设置主从有2种方式：

1. 在redis.conf中设置slaveof
   1. slaveof <masterip> <masterport>
2. 使用redis-cli客户端连接到redis服务，执行slaveof命令
   1. slaveof <masterip> <masterport>

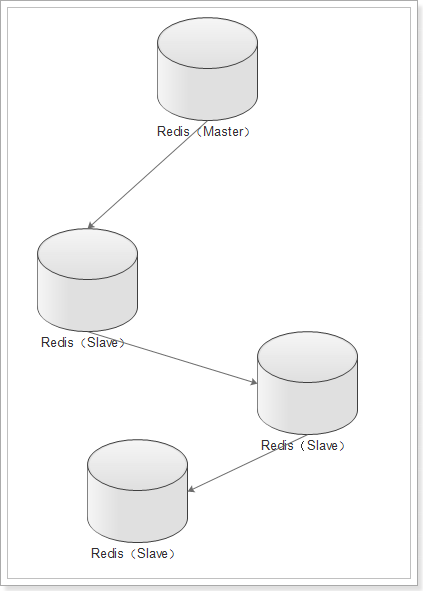
第二种方式在重启后将失去主从复制关系。也就是说是临时的主从。

查看主从信息：INFO replication

默认时，从库是只读的，只能在主库中写入数据。

可以在配置文件中开启非只读：slave-read-only no

### 主从从架构



如果一个主库连接了很多从库，那么在进行复制时，压力很大。这里相当于引入了一个中间层，由主库负责一个或者几个较少的从库复制，由从库进行更多的复制，减轻了主库的压力。

### 复制原理

1. 当从库和主库建立MS关系后，会向主数据库发送SYNC命令；
2. 主库接收到SYNC命令后会开始在后台保存快照（RDB持久化过程），并将期间接收到的写命令缓存起来；
3. 当快照完成后，主Redis会将快照文件和所有缓存的写命令发送给从Redis；
4. 从Redis接收到后，会载入快照文件并且执行收到的缓存的命令；
5. 之后，主Redis每当接收到写命令时就会将命令发送从Redis，从而保证数据的一致；

### 无磁盘复制

通过前面的复制过程我们了解到，主库接收到SYNC的命令时会执行RDB过程，即使在配置文件中禁用RDB持久化也会生成，那么如果主库所在的服务器磁盘IO性能较差，那么这个复制过程就会出现瓶颈，（主库将数据写到磁盘，并将数据发给从库）。庆幸的是，Redis在2.8.18版本开始实现了无磁盘复制功能（不过该功能还是处于试验阶段）。

原理：

Redis在与从数据库进行复制初始化时将不会将快照存储到磁盘，而是直接通过网络发送给从数据库，避免了IO性能差问题。

开启无磁盘复制：repl-diskless-sync yes

### 架构中出现宕机怎么办

如果在主从复制架构中出现宕机的情况，需要分情况看：

1. 从Redis宕机
   1. 这个相对而言比较简单，在Redis中从库重新启动后会自动加入到主从架构中，自动完成同步数据；
   2. 问题？ 如果从库在断开期间，主库的变化不大，从库再次启动后，主库依然会将所有的数据做RDB操作吗？还是增量更新？（从库有做持久化的前提下）
      1. 不会的，因为在Redis2.8版本后就实现了，主从断线后恢复的情况下实现增量复制。（从库在进行同步时，主库会记录从库的偏移量，则在下一次同步时，直接从之前记录的那个偏移量之后开始进行同步就可以了）

（如果是通过命令方式设置主从，那么此时从库中之前的数据会被清除，再进行复制）

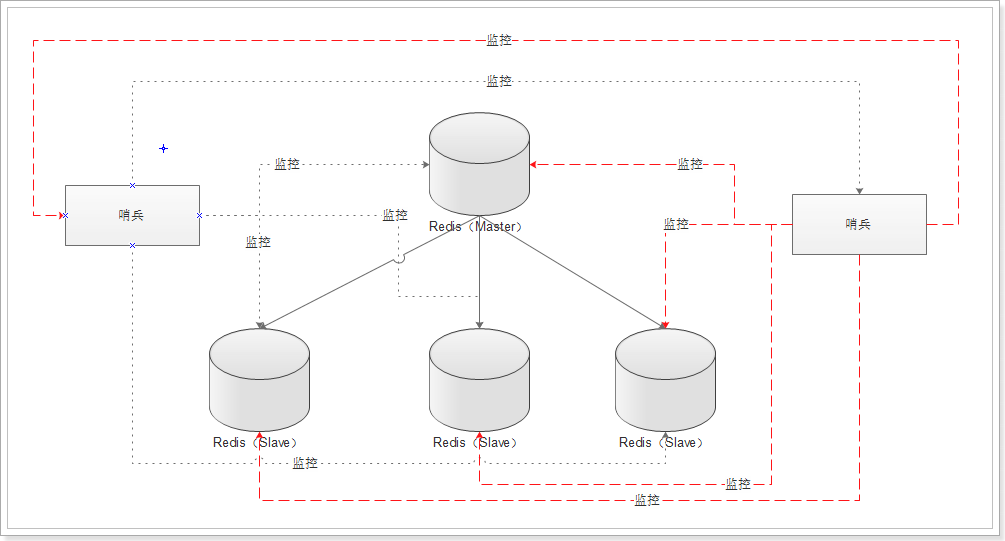
1. 主Redis宕机
   1. 这个相对而言就会复杂一些，需要以下2步才能完成
      1. 第一步，在从数据库中执行SLAVEOF NO ONE命令，断开主从关系并且提升为主库继续服务；（需要把其他的从库设置为当前库的从库）
      2. 第二步，将主库重新启动后，执行SLAVEOF命令，将其设置为其他库的从库，这时数据就能更新回来；
   2. 这个手动完成恢复的过程其实是比较麻烦的并且容易出错，有没有好办法解决呢？当前有的，Redis提供的哨兵（sentinel）的功能。

### 哨兵（sentinel）

顾名思义，哨兵的作用就是对Redis的系统的运行情况的监控，它是一个独立进程。它的功能有2个：

1. 监控主数据库和从数据库是否运行正常；
2. 主数据出现故障后自动将从数据库转化为主数据库；

#### 原理



多个哨兵，不仅同时监控主从数据库，而且哨兵之间互为监控。

#### 配置哨兵

动哨兵进程首先需要创建哨兵配置文件：

vim sentinel.conf

输入内容：

sentinel monitor taotaoMaster 127.0.0.1 6379 1

说明：

taotaoMaster：监控主数据库的名称，自定义即可，可以使用大小写字母和“.-\_”符号

127.0.0.1：监控的主数据库的IP

6379：监控的主数据库的端口

1：最低通过票数

启动哨兵进程：

redis-sentinel ./sentinel.conf

**哨兵无需配置slave，只需要指定master，哨兵会自动发现slave**

从库宕机后，哨兵的控制台会输出

2989:X 05 Jun 20:09:33.509 # +sdown slave 127.0.0.1:6380 127.0.0.1 6380 @ taotaoMaster 127.0.0.1 6379

此时如果将从库启动，从库会重新加入主从架构中。

#### 主库宕机

哨兵控制台会输出：

2989:X 05 Jun 20:16:50.300 # +sdown master taotaoMaster 127.0.0.1 6379 说明master服务已经宕机

2989:X 05 Jun 20:16:50.300 # +odown master taotaoMaster 127.0.0.1 6379 #quorum 1/1

2989:X 05 Jun 20:16:50.300 # +new-epoch 1

2989:X 05 Jun 20:16:50.300 # +try-failover master taotaoMaster 127.0.0.1 6379 开始恢复故障

2989:X 05 Jun 20:16:50.304 # +vote-for-leader 9059917216012421e8e89a4aa02f15b75346d2b7 1 投票选举哨兵leader，现在就一个哨兵所以leader就自己（超过半数以上，所以一般是奇数）

2989:X 05 Jun 20:16:50.304 # +elected-leader master taotaoMaster 127.0.0.1 6379 选中leader

2989:X 05 Jun 20:16:50.304 # +failover-state-select-slave master taotaoMaster 127.0.0.1 6379 选中其中的一个slave当做master

2989:X 05 Jun 20:16:50.357 # +selected-slave slave 127.0.0.1:6381 127.0.0.1 6381 @ taotaoMaster 127.0.0.1 6379 选中6381

2989:X 05 Jun 20:16:50.357 \* +failover-state-send-slaveof-noone slave 127.0.0.1:6381 127.0.0.1 6381 @ taotaoMaster 127.0.0.1 6379 发送slaveof no one命令

2989:X 05 Jun 20:16:50.420 \* +failover-state-wait-promotion slave 127.0.0.1:6381 127.0.0.1 6381 @ taotaoMaster 127.0.0.1 6379 等待升级master

2989:X 05 Jun 20:16:50.515 # +promoted-slave slave 127.0.0.1:6381 127.0.0.1 6381 @ taotaoMaster 127.0.0.1 6379 升级6381为master

2989:X 05 Jun 20:16:50.515 # +failover-state-reconf-slaves master taotaoMaster 127.0.0.1 6379

2989:X 05 Jun 20:16:50.566 \* +slave-reconf-sent slave 127.0.0.1:6380 127.0.0.1 6380 @ taotaoMaster 127.0.0.1 6379

2989:X 05 Jun 20:16:51.333 \* +slave-reconf-inprog slave 127.0.0.1:6380 127.0.0.1 6380 @ taotaoMaster 127.0.0.1 6379

2989:X 05 Jun 20:16:52.382 \* +slave-reconf-done slave 127.0.0.1:6380 127.0.0.1 6380 @ taotaoMaster 127.0.0.1 6379

2989:X 05 Jun 20:16:52.438 # +failover-end master taotaoMaster 127.0.0.1 6379 故障恢复完成

2989:X 05 Jun 20:16:52.438 # +switch-master taotaoMaster 127.0.0.1 6379 127.0.0.1 6381 主数据库从6379转变为6381

2989:X 05 Jun 20:16:52.438 \* +slave slave 127.0.0.1:6380 127.0.0.1 6380 @ taotaoMaster 127.0.0.1 6381 添加6380为6381的从库

2989:X 05 Jun 20:16:52.438 \* +slave slave 127.0.0.1:6379 127.0.0.1 6379 @ taotaoMaster 127.0.0.1 6381 添加6379为6381的从库

2989:X 05 Jun 20:17:22.463 # +sdown slave 127.0.0.1:6379 127.0.0.1 6379 @ taotaoMaster 127.0.0.1 6381 发现6379已经宕机，等待6379的恢复

#### 配置多个哨兵

### Redis主从的特点

1、redis使用异步复制，从服务器会以每秒一次的频率向主服务器报告复制流的处理进度

　　2、一个主服务器可以有多个从服务器，从服务器也可以有自己的从服务器(级联复制)

　　3、复制功能不会阻塞主服务器，即使一个或多个从服务器正在进行初次同步，主服务器也可以继续处理命令请求

　　4、复制功能可以用于数据冗余，也可以通过让多个从服务器处理只读命令请求来提升扩展性

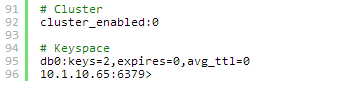
5、Redis从节点默认为只读，无须手动配置,redis的主从集群可以实现分担压力的效果，但是无法做到高可用，**如果master宕掉，服务就不可用了，所以使用redis的sentinel可以实现HA的功能**

### 主从配置

编辑/etc/redis.conf配置文件，将bind改为本机IP地址，重启服务即可。



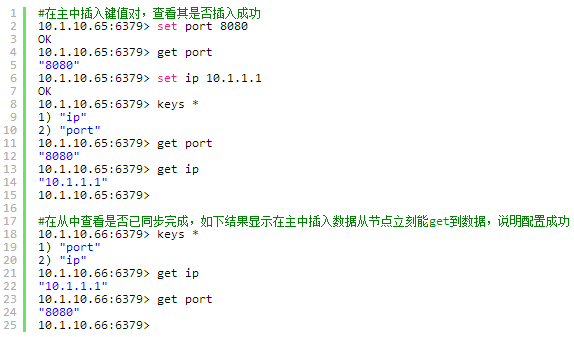
主节点配置相关参数：

从节点配置相关参数：



验证：



## Redis高可用管理工具sentinel

Sentinel是一个管理redis实例的工具，它可以对现有的redis进行监控、通知、故障自动转移，sentinel不断的检测redis实例是否可以正常的工作，通过API向其他程序报告redis的状态，如redis master不能工作，则会自动启动故障转移进程，将其中一个slave提升为master，其他slave将从新设置新的master服务器，而故障的master再次启动后会被sentinel自动降级为slave。



Sentinel作用如下：

　　1、监控：sentinel会不断的检查你的主服务器和从服务器是否运行正常

　　2、当被监控的某个redis服务器出现问题时，sentinel可以通过API向管理员或者其他应用程序发送通知

　　3、故障自动转移：当一个主服务器不能正常工作时，sentinel会开始一次自动故障转移操作，他会将其中一个从服务器升级为新的主服务器，并将其他从服务器改为复制新的主服务器；当客户端试图连接失效的主服务器时，集群也会向客户端返回新主服务器的地址，使得集群可以使用新主服务器代替失效服务器。

redis sentinel在监控redis实例时有两种redis宕机状态S\_DOWN和O\_DOWN：

　　S\_DOWN:当sentinel在指定的超时时间内没有收到一个正确的ping回复值，则认为是S\_DOWN

　　O\_DOWN:O\_DOWN的条件是有足够多的sentinel认为该redis实例是S\_DOWN。

　　注意：O\_DOWN只能是发生在主服务器，sentinel和其他从服务器不会发生O\_DOWN

## Redis命令

redis的存储结构是键值对，key只能是string类型，value支持5种数据类型，分别为字符串string 字符串列表list 字符串集合set 有序字符串集合sorted set和哈希hash

注意：1 key不要太长，不要超过1024字节，这不仅消耗内存，而且会降低查找效率

2 key不要太短，否则可读性会降低

3 在一个项目中，key最好使用统一的命名模式，如user:10000:passwd

### 基本命令

Keys \* 查找所有的键 keys速度非常快，但是在一个大的数据库上执行，仍然可能造成性能问题，因为redis是单进程单线程的。

Exists key 查看某个键是否存在

Del key 删除某个键

Type key 查看某个键的类型

Help

### String

可以存储任何形式的字符串，包括二进制数据 json化的对象 字节数组等。一个字符串类型的键允许存储的数据最大容量是512MB。

语法set key value 设置key=value

get key 获取key的value

incr key 将key的value自动加1（给数字字符类型自动加1）（将数字字符类型自动转换为integer类型，然后再加1）

decr key 将key的value自动减1

incrby key 10 对给定的key对应的value加10

decrby key 10 对给定的key 对应的value减10

append key value 给键对应值的后面追加value 返回的值是总长度

mset key1 value1 key2 value2…. 批量设置键值对

mget key1 key2… 获取多个键对应的值

expire key seconds 设置过期时间

ttl key 查看键的生存时间（剩下的时间）返回值-2表示key不存在 -1表示没有生存时间，永久生效

persist key 清除生存时间（重新设置值也会清除生存时间）

### Hash

Redis hash是一个string类型的field和value的映射表。一个key对应多个field，一个field对应一个value。将一个对象存储为hash类型，较于每个字段都存储为string

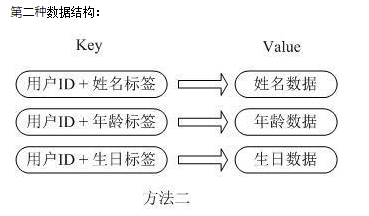
类型更能节省内存。

存储数据结构分析：

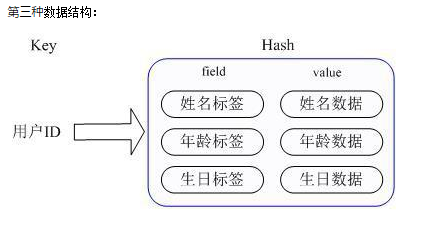


使用一个key获取一个对象，必须使用反序列化。

缺点：占用IO资源。



缺点：用户id，数据冗余。资源浪费。



redis存储结构：key是用户id，value是hash类型的数据。

命令：

hset user username wy 给user中的username属性设置一个值

hget user username 获取user中的username属性的值

hdel user username 删除user中的username属性

hsetnx user username wyy 如果user中的username属性值已经存在，则不会覆盖，否则设置值

hmset user password 123 age 11 同时设置多个值

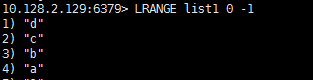
hmget user username age password 同时获取多个值

### List

是按照插入顺序进行排序的字符串链表，可以在其头部和尾部添加新的元素。在插入时如果该键不存在，redis将为该键创建一个新的链表。如果链表中所有的元素都被移除，那么该键也会从数据库中删除。List中可以包含的最大元素数量是4294967295。2^32-1

命令：

Lpush mylist a b c d 给list类型数据结构设置多个值（先进去的最后出来）



Lrange mylist 0 -1 获取mylist集合中所有的值 0代表链表的开头 -1代表链表的结束位置

Lpop mylist 将链表集合的头指针元素弹出

Lrem mylist 3 a 删除链表中前3个等于a的值

Lset mylist 2 s 设置链表位置2处的值 覆盖原值

Linsert mylist after s b 在链表s元素后面插入一个b

### Set

Redis的Set是string类型的无序集合。集合成员是唯一的，这就意味着集合中不能出现重复的数据。Redis 中 集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。

集合中最大的成员数为 2^32 - 1 (4294967295, 每个集合可存储40多亿个成员)。

命令：

Sadd myset a b c

Smembers myset 获取集合myset中值

Srem myset a b 删除myset中元素

Smove myset myset1 c 将集合myset中的元素c移动到集合myset1中

### Sorted set

有序集合

给set集合中每一个元素都设置一个得分，根据得分排序。

Set集合不允许重复，得分可以重复。

设置得分语法：ZADD key score member [score] [member]

zadd mysset 1 one 2 two 12 three 9 four 10 five 集合中添加5个元素，为每个元素设置一个得分

zcount mysset 1 10 获取分数1到10的元素（闭区间）

zcount mysset (1 10 左边是开区间，即不含1

zcount mysset –inf +inf 获取所有元素 –inf最低值 +inf最高值

zrange mysset 0 -1 withscores 获取集合中所有元素及其分数

zrangebyscore mysset 1 10 withscores limit 2 2 根据分数大小获取元素 limit分页获取值

## redis单进程单线程

Redis快的主要原因是：

1. 完全基于内存
2. 数据结构简单，对数据操作也简单（采用单线程，避免了不必要的上下文切换和竞争条件）
3. 使用I/O多路复用模型

多路 I/O 复用模型是利用select、poll、epoll可以同时监察多个流的 I/O 事件的能力，在空闲的时候，会把当前线程阻塞掉，当有一个或多个流有I/O事件时，就从阻塞态中唤醒，于是程序就会轮询一遍所有的流（epoll是只轮询那些真正发出了事件的流），并且只依次顺序的处理就绪的流，这种做法就避免了大量的无用操作。这里“多路”指的是多个网络连接，“复用”指的是复用同一个线程。采用多路 I/O 复用技术可以让单个线程高效的处理多个连接请求（尽量减少网络IO的时间消耗），且Redis在内存中操作数据的速度非常快（内存内的操作不会成为这里的性能瓶颈），主要以上两点造就了Redis具有很高的吞吐量。

单进程单线程好处：

1 代码更清晰 处理逻辑简单

2 不需要考虑各种锁的问题，不存在加锁释放锁的操作，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资源

单进程单线程弊端：

无法发挥多核CPU优势，不过可以在单机上开启多个redis实例来改善。

（Nginx是多进程单线程 Memcached是单进程多线程模型）

### Select poll epoll

Select缺点：

1 每次调用select时，都需要把fd（文件句柄）集合从用户态拷贝到内核态，这个开销在fd很多时会很大。

2 每次调用select后，都需要在内核遍历传递进来的所有fd，这个开销在fd很多时也比较大。

3 fd数量有限制，默认是1024

Poll解决了select的第三个缺点，fd数量不受限制，但是失去了select的跨平台特性。

Epoll：解决了select的三个缺点，是目前最好的一种IO多路复用解决方案。内核调用回调函数，把就绪的fd放入就绪链表中，并唤醒epoll\_wait，epoll\_wait只需要遍历就绪链表即可，而select和poll是遍历所有的fd。

select和poll都只提供了一个函数——select或者poll函数。而epoll提供了三个函数，epoll\_create,epoll\_ctl和epoll\_wait，epoll\_create是创建一个epoll句柄；epoll\_ctl是注册要监听的事件类型；epoll\_wait则是等待事件的产生。

对于第一个缺点，epoll的解决方案在epoll\_ctl函数中。每次注册新的事件到epoll句柄中时（在epoll\_ctl中指定EPOLL\_CTL\_ADD），会把所有的fd拷贝进内核，而不是在epoll\_wait的时候重复拷贝。epoll保证了每个fd在整个过程中只会拷贝一次。

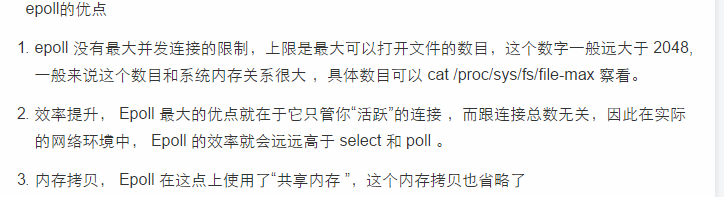
对于第二个缺点，epoll的解决方案不像select或poll一样每次都把current轮流加入fd对应的设备等待队列中，而只在epoll\_ctl时把current挂一遍（这一遍必不可少）并为每个fd指定一个回调函数，当设备就绪，唤醒等待队列上的等待者时，就会调用这个回调函数，而这个回调函数会把就绪的fd加入一个就绪链表）。epoll\_wait的工作实际上就是在这个就绪链表中查看有没有就绪的fd（利用schedule\_timeout()实现睡一会，判断一会的效果，和select实现中的第7步是类似的）。

对于第三个缺点，epoll没有这个限制，它所支持的FD上限是最大可以打开文件的数目，这个数字一般远大于2048,举个例子,在1GB内存的机器上大约是10万左右，具体数目可以cat /proc/sys/fs/file-max察看,一般来说这个数目和系统内存关系很大。

总结：

（1）select，poll实现需要自己不断轮询所有fd集合，直到设备就绪，期间可能要睡眠和唤醒多次交替。而epoll其实也需要调用epoll\_wait不断轮询就绪链表，期间也可能多次睡眠和唤醒交替，但是它是设备就绪时，调用回调函数，把就绪fd放入就绪链表中，并唤醒在epoll\_wait中进入睡眠的进程。虽然都要睡眠和交替，但是select和poll在“醒着”的时候要遍历整个fd集合，而epoll在“醒着”的时候只要判断一下就绪链表是否为空就行了，这节省了大量的CPU时间。这就是回调机制带来的性能提升。

（2）select，poll每次调用都要把fd集合从用户态往内核态拷贝一次，并且要把current往设备等待队列中挂一次，而epoll只要一次拷贝，而且把current往等待队列上挂也只挂一次（在epoll\_wait的开始，注意这里的等待队列并不是设备等待队列，只是一个epoll内部定义的等待队列）。这也能节省不少的开销。



## Redis并发问题

Redis是单进程单线程的，redis本身不会有并发问题（采用队列模式，将并发访问变为串行访问，命令是一个接一个执行）。但是在客户端发送多个请求时，我们自己操作的时候，仍然可能发生并发问题。

比如两个连接对键num=1进行操作，先取出num的值，然后再加1，最后的结果不一定是3，可能是2.

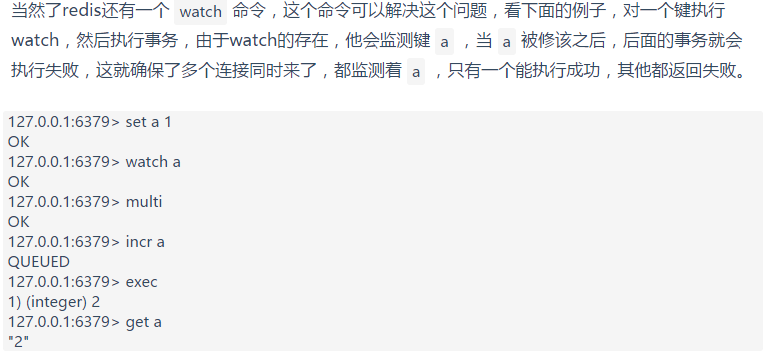
redis中也是有事务的，不过这个事务没有mysql中的完善，只保证了一致性和隔离性，不满足原子性和持久性。redis事务使用multi、exec命令.

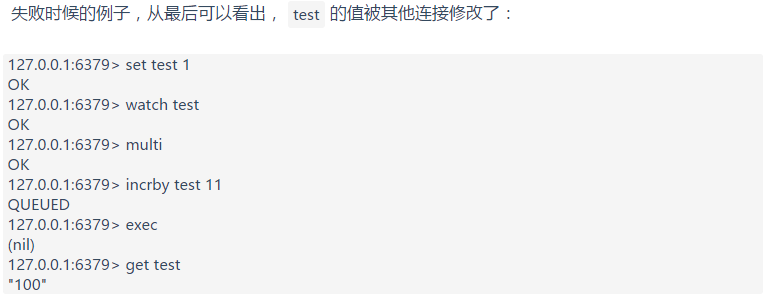
原子性，redis会将事务中的所有命令执行一遍，哪怕是中间有执行失败也不会回滚。kill信号、宿主机宕机等导致事务执行失败，redis也不会进行重试或者回滚。

持久性，redis事务的持久性依赖于redis所使用的持久化模式，遗憾的是各种持久化模式也都不是持久化的。

隔离性，redis是单进程，开启事务之后，会执行完当前连接的所有命令直到遇到exec命令，才处理其他连接的命令。  
一致性，看了文档，觉得挺扯的，但是貌似说的没有问题。

redis中的事务不支持原子性，所以解决不了上面的问题。





其他的解决方案：redis中命令是满足原子性的，因此在值为阿拉伯数字的时候，我可以将get和set命令修改为incr或者incrby来解决这个问题

Redis本身没有锁的概念，Redis对于多个客户端连接并不存在竞争，但是在Jedis客户端对Redis进行并发访问时会发生连接超时、数据转换错误、阻塞、客户端关闭连接等问题，这些问题均是由于客户端连接混乱造成。对此有2种解决方法：  
1.客户端角度，为保证每个客户端间正常有序与Redis进行通信，对连接进行池化，同时对客户端读写Redis操作采用内部锁synchronized。  
2.服务器角度，利用setnx实现锁。

语法：

SETNX key value  
功能：  
将 key 的值设为 value ，当且仅当 key 不存在；若给定的 key 已经存在，则 SETNX 不做任何动作。

时间复杂度：  
O(1)  
返回值：  
设置成功，返回 1 。  
设置失败，返回 0 。

模式：将 SETNX 用于加锁(locking)  
  
SETNX 可以用作加锁原语(locking primitive)。比如说，要对关键字(key) foo 加锁，客户端可以尝试以下方式：  
  
SETNX lock.foo <current Unix time + lock timeout + 1>  
  
如果 SETNX 返回 1 ，说明客户端已经获得了锁， key 设置的unix时间则指定了锁失效的时间。之后客户端可以通过 DEL lock.foo 来释放锁。  
  
如果 SETNX 返回 0 ，说明 key 已经被其他客户端上锁了。如果锁是非阻塞(non blocking lock)的，我们可以选择返回调用，或者进入一个重试循环，直到成功获得锁或重试超时(timeout)。

但是已经证实仅仅使用SETNX加锁带有竞争条件，在特定的情况下会造成错误。  
  
处理死锁(deadlock)  
  
上面的锁算法有一个问题：如果因为客户端失败、崩溃或其他原因导致没有办法释放锁的话，怎么办？  
  
这种状况可以通过检测发现——因为上锁的 key 保存的是 unix 时间戳，假如 key 值的时间戳小于当前的时间戳，表示锁已经不再有效。  
  
但是，当有多个客户端同时检测一个锁是否过期并尝试释放它的时候，我们不能简单粗暴地删除死锁的 key ，再用 SETNX 上锁，因为这时竞争条件(race condition)已经形成了：  
  
*C1 和 C2 读取 lock.foo 并检查时间戳， SETNX 都返回 0 ，因为它已经被 C3 锁上了，但 C3 在上锁之后就崩溃(crashed)了。  
C1 向 lock.foo 发送 DEL 命令。  
C1 向 lock.foo 发送 SETNX 并成功。  
C2 向 lock.foo 发送 DEL 命令。  
C2 向 lock.foo 发送 SETNX 并成功。  
出错：因为竞争条件的关系，C1 和 C2 两个都获得了锁。*

幸好，以下算法可以避免以上问题。来看看我们聪明的 C4 客户端怎么办：  
  
*C4 向 lock.foo 发送 SETNX 命令。  
因为崩溃掉的 C3 还锁着 lock.foo ，所以 Redis 向 C4 返回 0 。  
C4 向 lock.foo 发送 GET 命令，查看 lock.foo 的锁是否过期。如果不，则休眠(sleep)一段时间，并在之后重试。  
另一方面，如果 lock.foo 内的 unix 时间戳比当前时间戳老，C4 执行以下命令：  
GETSET lock.foo <current Unix timestamp + lock timeout + 1>*  
  
因为 GETSET 的作用，C4 可以检查看 GETSET 的返回值，确定 lock.foo 之前储存的旧值仍是那个过期时间戳，如果是的话，那么 C4 获得锁。  
如果其他客户端，比如 C5，比 C4 更快地执行了 GETSET 操作并获得锁，那么 C4 的 GETSET 操作返回的就是一个未过期的时间戳(C5 设置的时间戳)。C4 只好从第一步开始重试。  
注意，即便 C4 的 GETSET 操作对 key 进行了修改，这对未来也没什么影响。

这里假设锁key对应的value没有实际业务意义，否则会有问题，而且其实其value也确实不应该用在业务中。  
  
为了让这个加锁算法更健壮，获得锁的客户端应该常常检查过期时间以免锁因诸如 DEL 等命令的执行而被意外解开，因为客户端失败的情况非常复杂，不仅仅是崩溃这么简单，还可能是客户端因为某些操作被阻塞了相当长时间，紧接着 DEL 命令被尝试执行(但这时锁却在另外的客户端手上)。

**GETSET命令**

语法：

GETSET key value  
  
功能：  
将给定 key 的值设为 value ，并返回 key 的旧值(old value)。当 key 存在但不是字符串类型时，返回一个错误。  
时间复杂度：  
O(1)  
返回值：  
返回给定 key 的旧值；当 key 没有旧值时，也即是， key 不存在时，返回 nil 。

## Redis中的事务

见redis设计与实现

事务开始——命令入队——事务执行

Multi 开启一个事务 通过在客户端状态的flag是属性中打开REDIS\_MULTI标识来完成的。

如果客户端发送的是exec discard watch multi命令，那么服务器会立即执行这个命令，如果是其他命令，则会将这个命令放入一个事务队列中，然后向客户端回复queued。

每个redis客户端都有自己的事务状态，这个事务状态保存在客户端状态的mstate属性中。

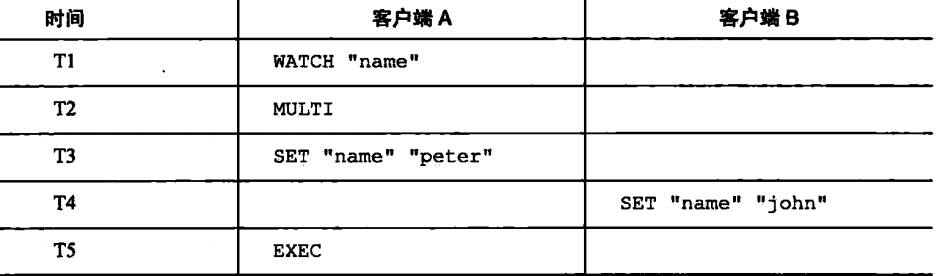
事务状态包含一个事务队列，以及一个已入队命令的计数器。（或者说是事务队列的长度）

事务队列以先进先出的方式保存入队的命令。

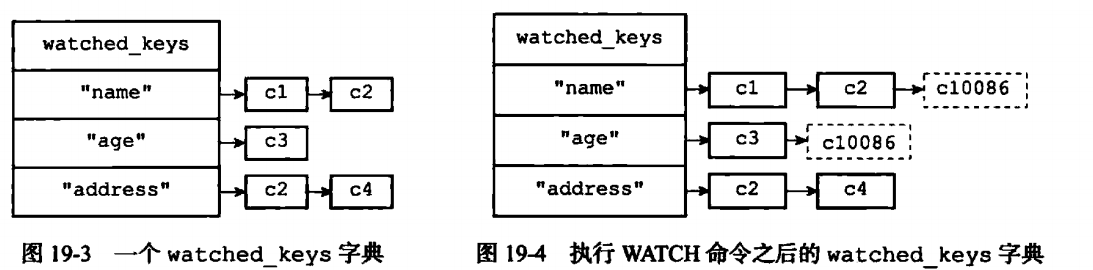
当客户端发送exec命令时，事务队列中的所有命令会被依次执行。

Watch命令是一个乐观锁。可以在exex执行之前，监视任意数量的数据库键，并在exec命令执行时，检查监视的键是否至少有一个已经被修改过了，如果被修改了，则服务器将拒绝执行事务，并向客户端返回代表事务执行失败的空回复。

下面例子将会执行失败



Watch实现原理：每个redis数据库都保存着一个watched\_keys字典，这个字典的键是某个被watch命令监视的数据库键，值是一个链表，链表中记录了所有监视相应数据库键的客户端。



监视机制的出发：所有对数据库进行修改的命令，如set lpush sadd zrem等，在执行之后都会调用multi.c/touchWatchKey函数对watched\_keys字典进行查看，查看是否有客户端正在监视刚刚被修改过的数据库键，如果有的话，那么touchWatchKey函数会将监视被修改键的客户端的redis\_dirty\_cas标识打开，表示该客户端的事务安全性已经被破坏。

判断事务是都安全：当服务器接受到exec命令时，服务器会根据这个客户端是否打开了redis\_dirty\_cas标识来决定是否执行事务。

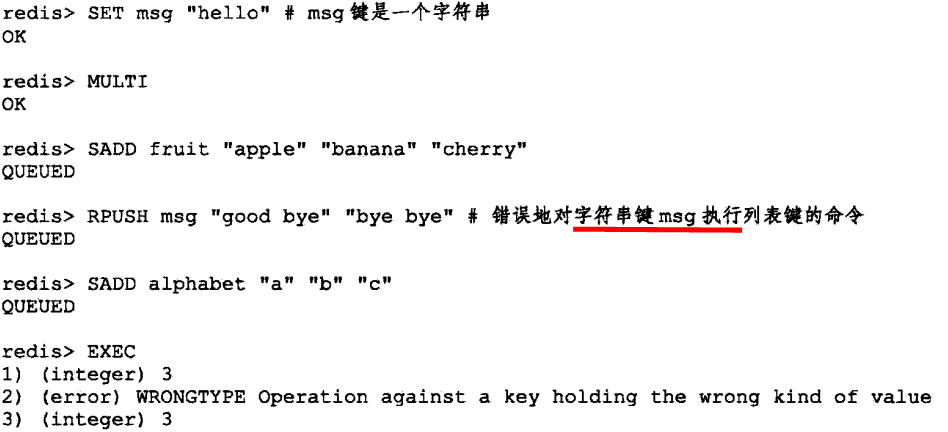
### Redis中事务的特性

原子性：数据库将事务中的多个操作当作一个整体来执行，要么全部执行，要么一个也不执行。Redis的事务具有原子性。

一种是事务中的所有事务都会被执行。一种是因为**命令入队出错而被服务器拒绝执行**，事务中的所有命令都不会被执行。



但是redis事务不支持事务回滚。即使事务队列中的某个命令在执行期间出现了错误，整个事务也会继续执行下去，直至将事务队列中的所有命令都执行完毕。



出错命令的后续命令会继续执行，并且之前的命令不会被影响。

一致性

数据库在执行事务之前是一致的，在事务执行之后，无论事务成功与否，数据库应该也是一致的。

Redis如何保证一致性？

1 入队错误。如果一个事务在入队命令过程中，出现了命令不存在，或者命令格式不正确等情况，那么redis会拒绝执行这个事务。

2 执行错误。这些命令只有在实际执行时才会触发。即使执行过程中出现了错误，服务器不会中断事务的执行。

因为在事务执行过程中，出错的命令会被服务器识别出来，并进行相应的错误处理，所以这些出错命令不会对数据库做出任何修改，也不会对事务一致性产生任何影响。

3 服务器停机 可以根据rdb或者aof文件来恢复数据。

隔离性：即使数据库中有多个事务并发执行，各个事务之间也不会互相影响，并且在并发状态下执行的事务和串行执行的事务产生的结果完全相同。

因为redis使用单进程单线程的方式来执行事务，并且保证在事务执行期间不会对事务进行中断，因此redis事务总是以串行方式运行的，并且具有隔离性。

持久性：事务执行完成之后的结果可以进行持久化。

当redis处于无持久化的内存模式下运行时，事务不具有持久性。

Rdb模式下，只有在满足特定的保存条件时，才会触发bgsave操作，对数据库进行保存操作，并且异步执行bgsave，不能保证事务数据在第一时间保存在磁盘中，不具有持久性。

Aof模式下，并且appendfsync选项为always时，程序总会在执行一条命令之后调用同步函数，将命令数据保存在磁盘中，具有持久性。如果appendfsync为其他，不具有持久性。

**不论redis处于何种模式，在事务中的最后加上save命令，总是可以保证事务的持久性。**

## IO模型

常见的IO模型：

1 同步阻塞IO：即传统的IO模型

2 同步非阻塞IO（NIO）：默认创建的socket都是阻塞的，非阻塞要求socket被设置为NONBLOCK。

3 IO多路复用：即经典的Reactor设计模式，有时也称为异步阻塞IO。Java中的selector和linux中的epoll都是这种模型。

4 异步IO：即经典的Proactor设计模式，也称为异步非阻塞IO

同步和异步描述的是用户线程和内核的交互方式。同步是指用户线程发起IO请求之后需要等待或者轮询内核IO操作完成之后才能继续执行。而异步是指用户线程发起IO请求后仍继续执行，当内核IO操作完成之后会通知用户线程，或者调用用户线程注册的回调函数。

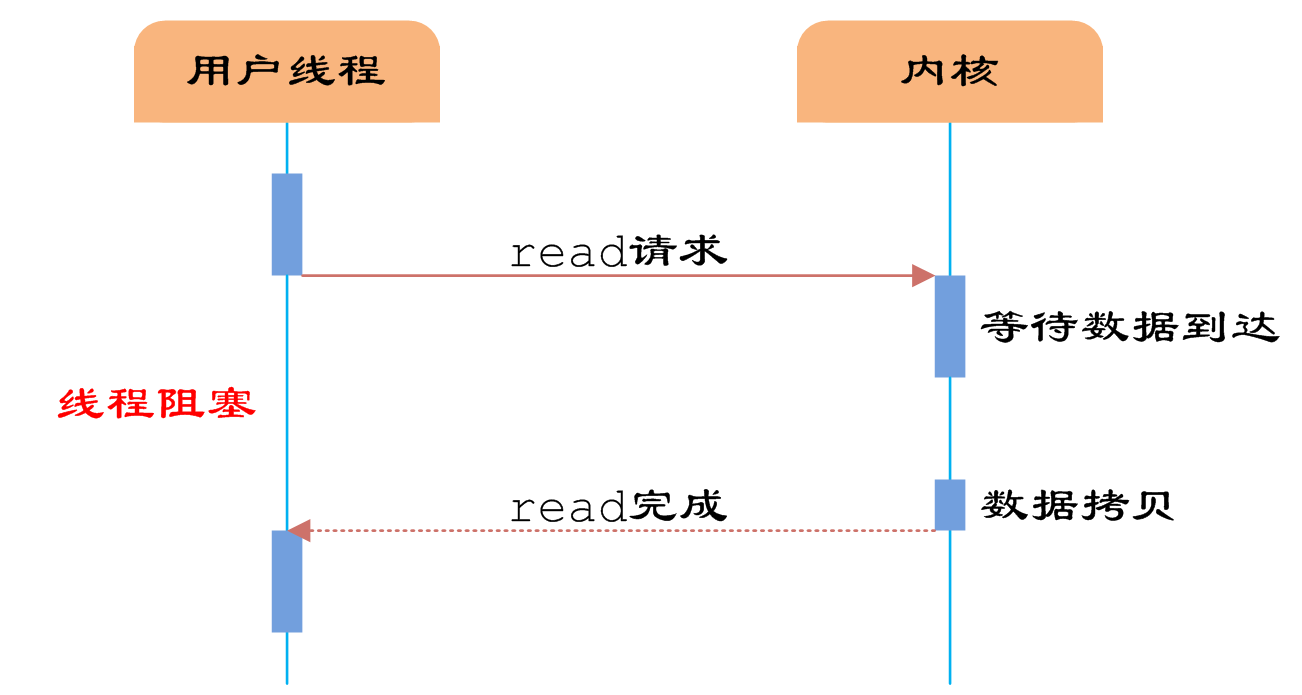
阻塞和非阻塞描述的是用户线程调用内核IO操作的方式：阻塞是指IO操作需要彻底完成后才能返回到用户空间。而非阻塞是指IO操作被调用之后立即返回给用户一个状态值，无需等到IO操作彻底完成。

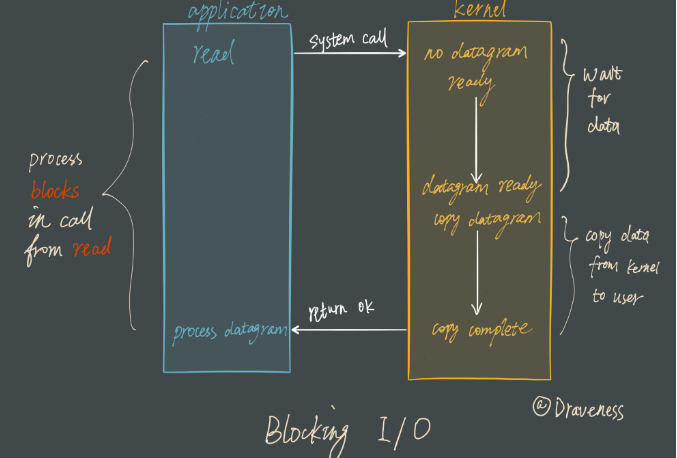
对于socket编程来说：

第一步是等待数据从网络传输到本地。当数据包到达时，数据将会从网络层拷贝到内核的缓存中。第二不是从内核中把数据拷贝到程序的数据区中。（也就是内核态到用户态数据的拷贝）

### 同步阻塞IO

用户线程在内核进行IO操作时被阻塞。





用户线程通过系统调用read发起IO读操作，由用户空间转到内核空间。内核等到数据包到达后，然后将接收的数据拷贝到用户空间，完成read操作。

用户线程使用同步阻塞IO模型的伪代码描述为：

{

read(socket, buffer);

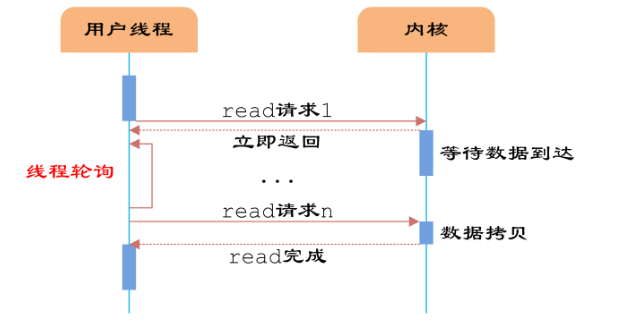
process(buffer);

}

即用户需要等待read将socket中的数据读取到buffer后，才继续处理接收的数据。整个IO请求的过程中，用户线程是被阻塞的，这导致用户在发起IO请求时，不能做任何事情，对CPU的资源利用率不够。

### 同步非阻塞IO

将socket设置为NONBLOCK。这样做用户线程可以在发起IO请求后可以立即返回。



由于socket是非阻塞的方式，因此用户线程发起IO请求时立即返回。但并未读取到任何数据，用户线程需要不断地发起IO请求，直到数据到达后，才真正读取到数据，继续执行。

用户线程使用同步非阻塞IO模型的伪代码描述为：

{

while(read(socket, buffer) != SUCCESS)

;

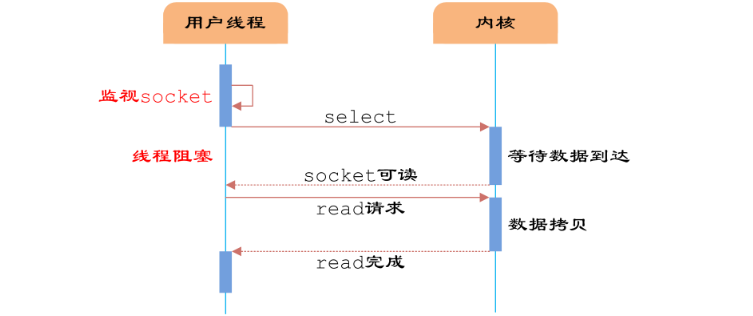
process(buffer);

}

即用户需要不断地调用read，尝试读取socket中的数据，直到读取成功后，才继续处理接收的数据。整个IO请求的过程中，虽然用户线程每次发起IO请求后可以立即返回，但是为了等到数据，仍需要不断地轮询、重复请求，消耗了大量的CPU的资源。一般很少直接使用这种模型，而是在其他IO模型中使用非阻塞IO这一特性。

### IO多路复用

IO多路复用模型是建立在内核提供的多路分离函数select基础之上的，使用select函数可以避免同步非阻塞IO模型中轮询等待的问题



用户首先将需要进行IO操作的socket添加到select中，然后阻塞等待select系统调用返回。当数据到达时，socket被激活，select函数返回。用户线程正式发起read请求，读取数据并继续执行。

从流程上来看，使用select函数进行IO请求和同步阻塞模型没有太大的区别，甚至还多了添加监视socket，以及调用select函数的额外操作，效率更差。但是，使用select以后最大的优势是用户可以在一个线程内同时处理多个socket的IO请求。用户可以注册多个socket，然后不断地调用select读取被激活的socket，即可达到在**同一个线程内同时处理多个IO请求的目的**。而在同步阻塞模型中，必须通过多线程的方式才能达到这个目的。

用户线程使用select函数的伪代码描述为：

{

select(socket);

while(1) {

sockets = select();

for(socket in sockets) {

if(can\_read(socket)) {

read(socket, buffer);

process(buffer);

}

}

}

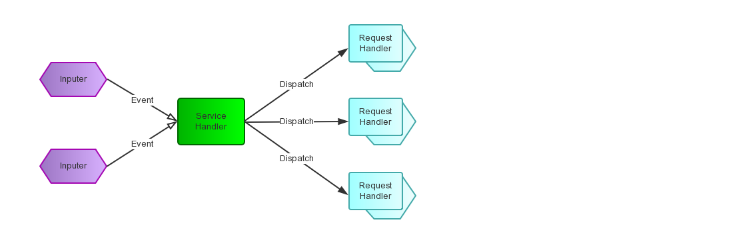
}

其中while循环前将socket添加到select监视中，然后在while内一直调用select获取被激活的socket，一旦socket可读，便调用read函数将socket中的数据读取出来。

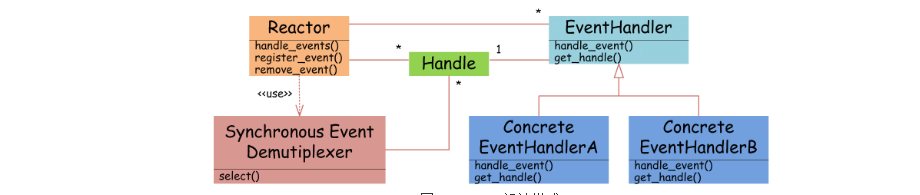
然而，使用select函数的优点并不仅限于此。虽然上述方式允许单线程内处理多个IO请求，但是每个IO请求的过程还是阻塞的（在select函数上阻塞），平均时间甚至比同步阻塞IO模型还要长。如果用户线程只注册自己感兴趣的socket或者IO请求，然后去做自己的事情，等到数据到来时再进行处理，则可以提高CPU的利用率。

**IO多路复用模型使用了Reactor设计模式实现了这一机制。**

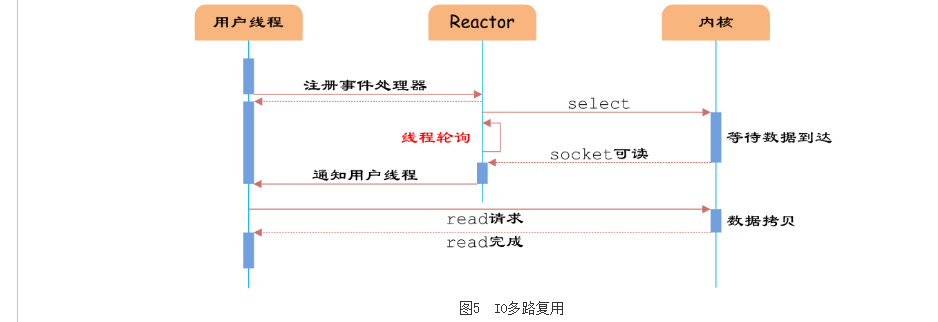
Reactor模式：基于事件驱动的，有一个或者多个并发输入源，有一个ServiceHandler，有多个RequestHandlers，这个Service Handler会同步的将输入请求（Event）多路复用的分发给相应的RequestHandler。



<http://www.blogjava.net/DLevin/archive/2015/09/02/427045.html>



EventHandler抽象类表示IO事件处理器，它拥有IO文件句柄Handle（通过get\_handle获取），以及对Handle的操作handle\_event（读/写等）。继承于EventHandler的子类可以对事件处理器的行为进行定制。Reactor类用于管理EventHandler（注册、删除等），并使用handle\_events实现事件循环，不断调用同步事件多路分离器（一般是内核）的多路分离函数select，只要某个文件句柄被激活（可读/写等），select就返回（阻塞），handle\_events就会调用与文件句柄关联的事件处理器的handle\_event进行相关操作。



通过Reactor的方式，可以将用户线程轮询IO操作状态的工作统一交给handle\_events事件循环进行处理。用户线程注册事件处理器之后可以继续执行做其他的工作（异步），而Reactor线程负责调用内核的select函数检查socket状态。当有socket被激活时，则通知相应的用户线程（或执行用户线程的回调函数），执行handle\_event进行数据读取、处理的工作。由于select函数是阻塞的，因此多路IO复用模型也被称为异步阻塞IO模型。注意，这里的所说的阻塞是指select函数执行时线程被阻塞，而不是指socket。一般在使用IO多路复用模型时，socket都是设置为NONBLOCK的，不过这并不会产生影响，因为用户发起IO请求时，数据已经到达了，用户线程一定不会被阻塞。

用户线程使用IO多路复用模型的伪代码描述为：

void UserEventHandler::handle\_event() {

if(can\_read(socket)) {

read(socket, buffer);

process(buffer);

}

}

{

Reactor.register(new UserEventHandler(socket));

}

用户需要重写EventHandler的handle\_event函数进行读取数据、处理数据的工作，用户线程只需要将自己的EventHandler注册到Reactor即可。Reactor中handle\_events事件循环的伪代码大致如下。

Reactor::handle\_events() {

while(1) {

sockets = select();

for(socket in sockets) {

get\_event\_handler(socket).handle\_event();

}

}

}

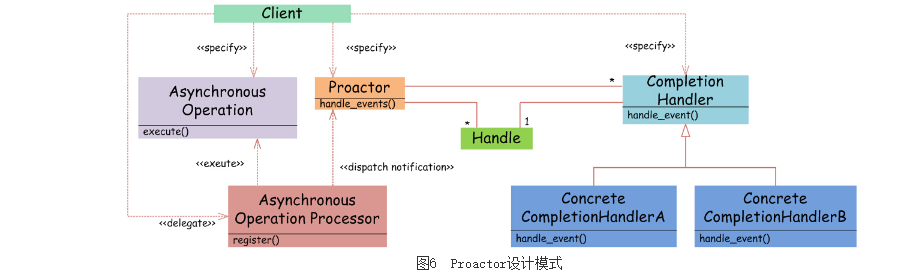
事件循环不断地调用select获取被激活的socket，然后根据获取socket对应的EventHandler，执行器handle\_event函数即可。

IO多路复用是最常使用的IO模型，但是其异步程度还不够“彻底”，因为它使用了会阻塞线程的select系统调用。因此IO多路复用只能称为异步阻塞IO，而非真正的异步IO。

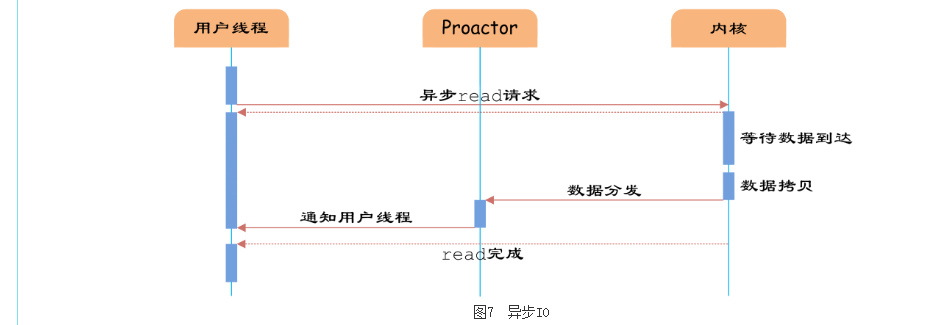
### 异步IO

“真正”的异步IO需要操作系统更强的支持。在IO多路复用模型中，事件循环将文件句柄的状态事件通知给用户线程，由用户线程自行读取数据、处理数据。而在异步IO模型中，当用户线程收到通知时，数据已经被内核读取完毕，并放在了用户线程指定的缓冲区内，内核在IO完成后通知用户线程直接使用即可。

异步IO模型使用了Proactor设计模式实现了这一机制。



Proactor模式和Reactor模式在结构上比较相似，不过在用户（Client）使用方式上差别较大。Reactor模式中，用户线程通过向Reactor对象注册感兴趣的事件监听，然后事件触发时调用事件处理函数。而Proactor模式中，用户线程将AsynchronousOperation（读/写等）、Proactor以及操作完成时的CompletionHandler注册到AsynchronousOperationProcessor。AsynchronousOperationProcessor使用Facade模式提供了一组异步操作API（读/写等）供用户使用，当用户线程调用异步API后，便继续执行自己的任务。AsynchronousOperationProcessor 会开启独立的内核线程执行异步操作，实现真正的异步。当异步IO操作完成时，AsynchronousOperationProcessor将用户线程与AsynchronousOperation一起注册的Proactor和CompletionHandler取出，然后将CompletionHandler与IO操作的结果数据一起转发给Proactor，Proactor负责回调每一个异步操作的事件完成处理函数handle\_event。虽然Proactor模式中每个异步操作都可以绑定一个Proactor对象，但是一般在操作系统中，Proactor被实现为Singleton模式，以便于集中化分发操作完成事件。



异步IO模型中，用户线程直接使用内核提供的异步IO API发起read请求，且发起后立即返回，继续执行用户线程代码。不过此时用户线程已经将调用的AsynchronousOperation和CompletionHandler注册到内核，然后操作系统开启独立的内核线程去处理IO操作。当read请求的数据到达时，由内核负责读取socket中的数据，并写入用户指定的缓冲区中。最后内核将read的数据和用户线程注册的CompletionHandler分发给内部Proactor，Proactor将IO完成的信息通知给用户线程（一般通过调用用户线程注册的完成事件处理函数），完成异步IO。

用户线程使用异步IO模型的伪代码描述为：

void UserCompletionHandler::handle\_event(buffer) {

process(buffer);

}

{

aio\_read(socket, new UserCompletionHandler);

}

用户需要重写CompletionHandler的handle\_event函数进行处理数据的工作，参数buffer表示Proactor已经准备好的数据，用户线程直接调用内核提供的异步IO API，并将重写的CompletionHandler注册即可。

相比于IO多路复用模型，异步IO并不十分常用，不少高性能并发服务程序使用IO多路复用模型+多线程任务处理的[**架构**](http://lib.csdn.net/base/architecture)基本可以满足需求。况且目前操作系统对异步IO的支持并非特别完善，更多的是采用IO多路复用模型模拟异步IO的方式（IO事件触发时不直接通知用户线程，而是将数据读写完毕后放到用户指定的缓冲区中）。Java7之后已经支持了异步IO。

## Memcached和redis

Memcache被称为**分布式**缓存服务器，但服务端并没有提供分布式功能，其分布式主要体现在客户端，对于Server端，仅仅是部署多个Memcache Server组成的集群，每个Server独立维护自己的数据，节点相互之间没有任何通信。Memcache的分布式是在客户端通过**一致性哈希算法**实现的，即将要存储的数据分布到某个特定的Server上存储，后续读取查询使用同样的Hash算法即可定位。

最简单的Hash算法：targetServer=serverList[hash(key)%serverList.size]。算法简单，而且具有不错的随机分布特性。但问题也很明显，Server的总数不能轻易变化。因为如果增加/减少Server的数量，对原先存储的所有Key的后续查询都将定位到别的Server上，导致大部分Cache都不能被命中而失效。

一致性哈希算法解决了这个问题。相对于对Server总数取模的算法，一致性Hash算法除了计算Key的Hash值外，还会计算每个Server对应的Hash值，然后将这些Hash值映射到有限的值域上,0~(2^32）-1。通过寻找Hash值大于Hash(key)的最小Server作为存储该Key的目标Server。如果找不到，则直接把具有最小Hash值的Server作为目标Server。   
   但是当Server数量很少的时候，很可能他们在环中的分布不是特别均匀，进而导致Cache不能均匀分布到所有的Server上。   
  这时就需要引入**虚拟节点（virtual node）**，一个物理节点对应多个虚拟节点，可以根据每个物理Server的负载能力，赋予不同的权重，根据权重为物理Server分配不同数量的虚拟节点。当为Cache定位目标Server时，如果定位到虚拟节点上，就表示Cache真正的存储位置是在该虚拟节点代表的物理Server上。   
  虚拟节点的Hash计算可以采用对应物理节点IP地址加数字后缀的方式。如，Node1的IP为192.168.0.1，Node1的Hash值是Hash（”192.168.0.1”）。Node1的虚拟节点Node1-1,Node1-2的Hash值是Hash(“192.168.0.1#1”),Hash(“192.168.0.1#2”)。   
恩，很不错的算法。

而在Redis2.8中，提供Sentinel作为Redis的HA解决方案：由一个或多个Sentinel实例组成的Sentinel系统可以监视任意多个主服务器，以及这些主服务器下的所有从服务器，并在被监视的主服务器进入下线状态时，自动将下线主服务器属下的某个从服务器升级为新的主服务器，然后由新的主服务器继续处理命令请求。

Sentinel作为一个Observer,为了避免单点问题，就需要多个Sentinel，这时就需要选出领头羊Sentinel…系统的复杂性大大增加。

在Redis3.0 Realease版中提供了集群方案：通过分片(sharding)的方式来保存集群中的键值对，集群的整个**数据库**被分为16384个槽(slot),数据库中的每个键会映射到这16384个槽中，集群中的每个节点负责处理被指派的槽，可以处理0个或最多16384个槽。集群中的每个节点都是平等的，没有中心，没有代理。每个节点通过gossip协议相互通讯掌握其他节点的健康状况和数据信息。   
  客户端直接访问集群节点。如果客户端请求的Key所在的槽正好被指派给了当前节点，那么当前节点直接执行这个命令；如果Key所在的槽并没有被指派给当前节点，那么会路由(redirect)到Key所在的节点进行处理。   
  节点和单机数据库的一个区别是，节点只能使用0号数据库，而单机Redis服务器则没有这一限制。   
  Redis集群的重新分片操作可以将任意数量已经指派给某个节点的槽改为指派给另外一个节点，并且相关槽所属的键值对也会从源节点移动到目标节点。重新分片操作可以在线进行，并且源节点和目标节点都可以继续处理命令请求。   
  Redis集群中的节点分为主节点（master）和从节点(slave),主节点用于处理槽，从节点复制主节点，在主节点下线时，代替主节点继续处理命令请求。

## Redis比memcached快

* Libevent。和Memcached不同，Redis并没有选择libevent。Libevent为了迎合通用性造成代码庞大(目前Redis代码还不到libevent的1/3)及牺牲了在特定平台的不少性能。Redis用libevent中两个文件修改实现了自己的epoll event loop。业界不少开发者也建议Redis使用另外一个libevent高性能替代libev，但是作者还是坚持Redis应该小巧并去依赖的思路。一个印象深刻的细节是编译Redis之前并不需要执行./configure。
* CAS问题。CAS是Memcached中比较方便的一种防止竞争修改资源的方法。CAS实现需要为每个cache key设置一个隐藏的cas token，cas相当value版本号，每次set会token需要递增，因此带来CPU和内存的双重开销，虽然这些开销很小，但是到单机10G+ cache以及QPS上万之后这些开销就会给双方相对带来一些细微性能差别。

## Redis集群

即使有了主从复制，每个数据库都要保存整个集群中的所有数据，容易形成木桶效应。

使用Jedis实现了分片集群，是由客户端控制哪些key数据保存到哪个数据库中，（各个redis数据库并不知道自己处于集群环境中）如果在水平扩容时就必须手动进行数据迁移，而且需要将整个集群停止服务，这样做非常不好的。

（一致性哈希算法）

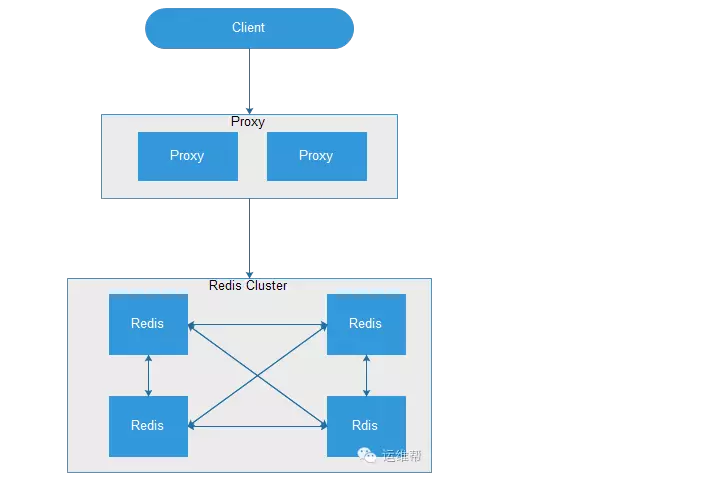
Redis3.0版本的一大特性就是集群（Cluster），接下来我们一起学习集群。

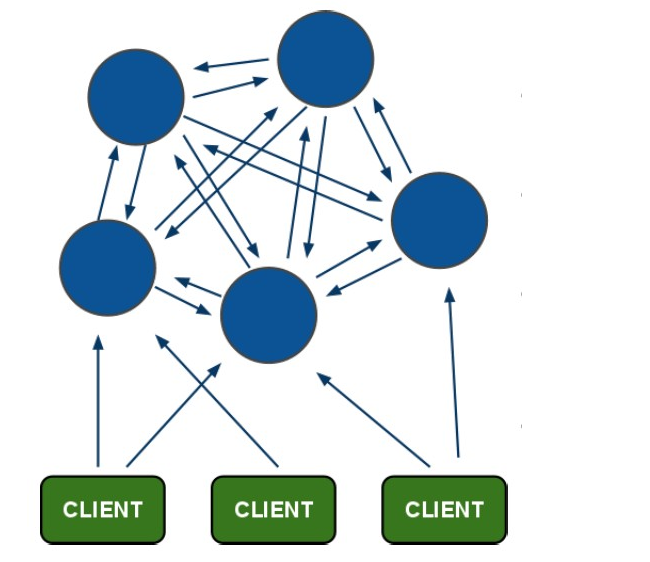
3.0版本之前，redis是不支持集群的。如果自己做集群，需要一个中间件，这个中间件负责将需要存入redis的key通过一套算法计算得一个值（hash算法），然后根据这个值找到对应的redis节点，将这些数据存在这个redis的节点中。

在取值的时候，同样先将key进行计算，得出对应的值，然后去找对应的redis节点，从对应的节点中取出对应的值。

缺点：hash计算需要在系统中进行，增加了系统的负担。这个集群中，如果某个节点挂了，其他节点无法知道，而且也不容易对每个节点进行负载均衡。

3.0版本之后，redis支持集群部署。





Redis Cluster是一个高性能高可用的分布式系统。由多个Redis实例组成的整体，数据按照Slot存储分布在多个Redis实例上，通过Gossip协议来进行节点之间通信。

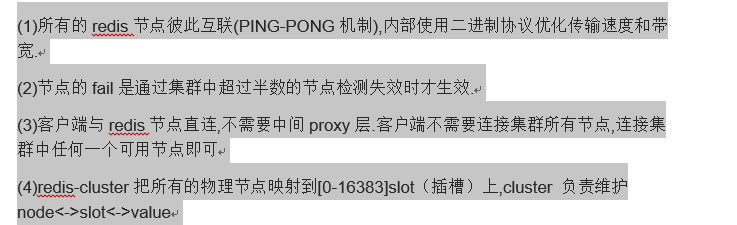
每一个蓝色的圈都代表着一个redis的服务器节点。它们任何两个节点之间都是相互连通的。客户端可以与任何一个节点相连接，然后就可以访问集群中的任何一个节点，对其进行存取和其他操作。

集群中各个节点是同等地位，没有主次之分，通过Gossip协议进行通信。这样每个节点就可以很容易的知道集群的变化，如节点的新增和删除等。

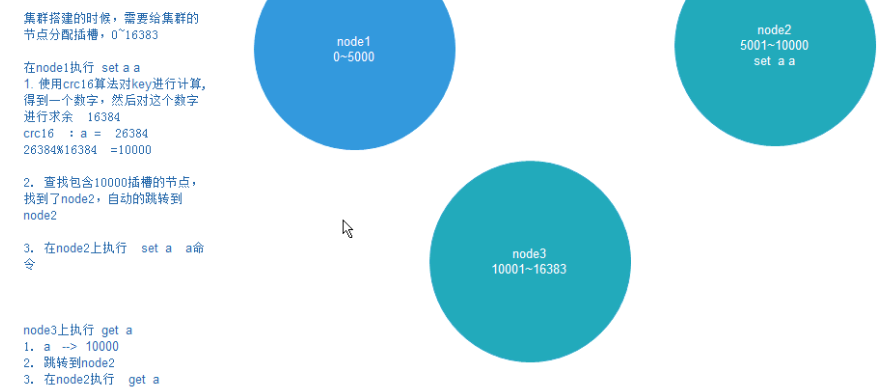
Gossip协议：Gossip是一种去中心化、容错并保证最终一致性的协议。（详解见Gossip）

Redis Cluster功能特点如下：

1. 所有的节点相互连接
2. 集群消息通信通过集群总线通信，，集群总线端口大小为客户端服务端口+10000，这个10000是固定值
3. 节点与节点之间通过二进制协议进行通信
4. 客户端和集群节点之间通信和通常一样，通过文本协议进行
5. 集群节点不会代理查询
6. 数据按照Slot存储分布在多个Redis实例上
7. 集群节点挂掉会自动故障转移
8. 可以相对平滑扩/缩容节点



在redis的每一个节点上，都有这么两个东西，一个是插槽（slot）可以理解为是一个可以存储两个数值的一个变量，这个变量的取值范围是：0-16383。还有一个就是cluster。这个cluster理解为是一个集群管理的插件。当我们的存取的key到达的时候，redis会根据crc16的算法得出一个结果，然后把结果对 16384 求余数，这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽，通过这个值，去找到对应的插槽所对应的节点，然后直接自动跳转到这个对应的节点上进行存取操作。



为了保证集群中节点的高可用，需要为每个节点有至少一个备用的redis服务。这个备用的redis称为从节点（slave）。

每一个节点都存有集群中所有主节点和从节点的信息。

它们之间通过互相的ping-pong判断是否节点可以连接上。如果有一半以上的节点去ping一个节点的时候没有回应，集群就认为这个节点宕机了，然后去连接它的备用节点。如果某个节点和所有从节点全部挂掉，我们集群就进入faill状态。还有就是如果有一半以上的主节点宕机，那么整个集群不可用。这就是**redis的投票机制**，具体原理如下图所示：

(1)投票过程是集群中所有master参与,如果半数以上master节点与master节点通信超时(cluster-node-timeout),认为当前master节点挂掉.

(2):什么时候整个集群不可用(cluster\_state:fail)?

 a:如果集群任意master挂掉,且当前master没有slave.集群进入fail状态,也可以理解成集群的slot映射[0-16383]不完整时进入fail状态. ps : redis-3.0.0.rc1加入cluster-require-full-coverage参数,默认关闭,打开集群兼容部分失败.

 b:如果集群超过半数以上master挂掉，无论是否有slave，集群进入fail状态.

### 设计的原则

1. **性能**：这是Redis赖以生存的看家本领，增加集群功能后当然不能对性能产生太大影响，所以Redis采取了P2P而非Proxy方式、异步复制、客户端重定向等设计，而牺牲了部分的一致性、使用性。
2. **水平扩展**：集群的最重要能力当然是扩展，文档中称可以线性扩展到1000结点。
3. **可用性**：在Cluster推出之前，可用性要靠Sentinel保证。有了集群之后也自动具有了Sentinel的监控和自动Failover能力。

由增加节点和删除节点的过程，可知道数据是存储在插槽中，节点负责管理插槽，因此不管节点个数怎么变化，槽位是一定的，数据的命中率依然会很高，不会出现之前在客户端进行分片，扩展性差的问题。

### 架构变化与CAP理论

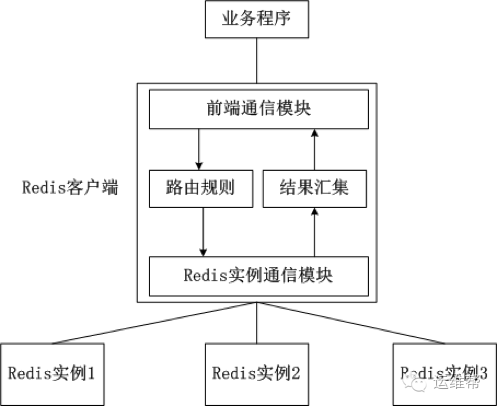
Redis Cluster集群功能推出已经有一段时间了。在单机版的Redis中，每个Master之间是没有任何通信的，所以我们一般在Jedis客户端或者Codis这样的代理中做Pre-sharding。按照CAP理论来说，**单机版的Redis属于保证CP(Consistency & Partition-Tolerancy)而牺牲A(Availability)**，也就说Redis能够保证所有用户看到相同的数据（一致性，因为Redis不自动冗余数据）和网络通信出问题时，暂时隔离开的子系统能继续运行（分区容忍性，因为Master之间没有直接关系，不需要通信），但是不保证某些结点故障时，所有请求都能被响应（可用性，某个Master结点挂了的话，那么它上面分片的数据就无法访问了）。

有了Cluster功能后，**Redis从一个单纯的NoSQL内存数据库变成了分布式NoSQL数据库，CAP模型也从CP变成了AP**。也就是说，通过自动分片和冗余数据，Redis具有了真正的分布式能力，某个结点挂了的话，因为数据在其他结点上有备份，所以其他结点顶上来就可以继续提供服务，保证了Availability。然而，也正因为这一点，Redis无法保证曾经的强一致性了。这也是CAP理论要求的，三者只能取其二。

### 集群方案

#### 客户端分片

客户端分片是把分片的逻辑放在Redis客户端实现，通过Redis客户端预先定义好的路由规则，把对Key的访问转发到不同的Redis实例中，最后把返回结果汇集。

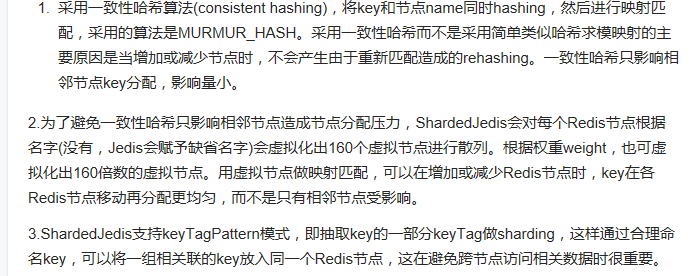
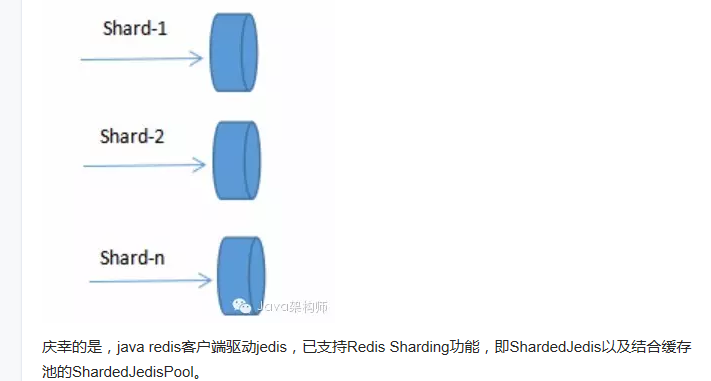


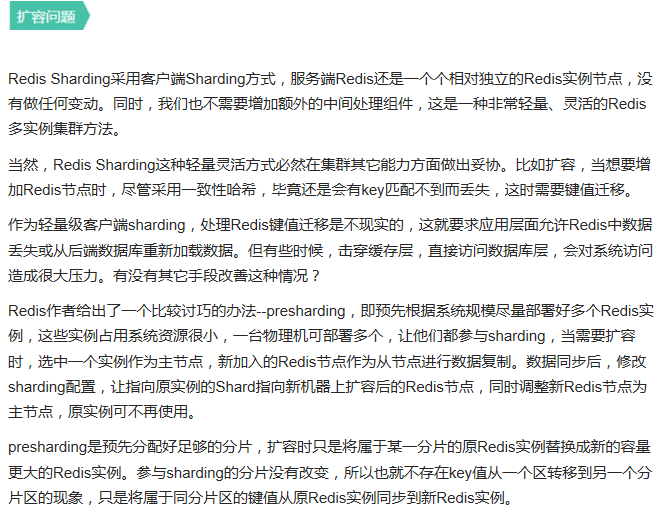
优点：所有的逻辑都是可控的，不依赖于第三方分布式中间件。

缺点：1 这是静态的分片方案，在增加和减少Redis实例的数量，需要手动调整分片的程序。

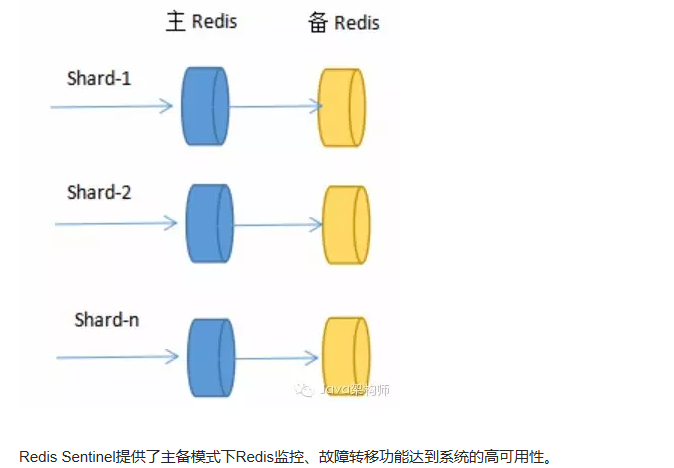
2 在不同的客户端程序中，维护相同的分片逻辑成本巨大。例如，系统中有两套业务系统共用一套Redis集群，一套业务系统用**Java**实现，另一套业务系统用**PHP**实现。为了保证分片逻辑的一致性，在Java客户端中实现的分片逻辑也需要在PHP客户端实现一次。相同的逻辑在不同的系统中分别实现，这种设计本来就非常糟糕，而且需要耗费巨大的开发成本保证两套业务系统分片逻辑的一致性。

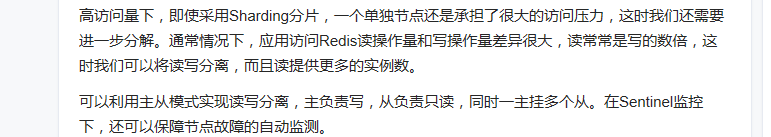
3 不能平滑地水平扩容，扩容/缩容时，必须手动调整分片程序，出现故障不能自动转移，难以运维

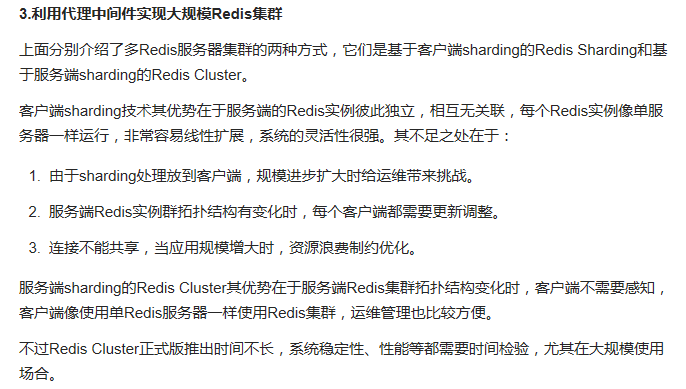


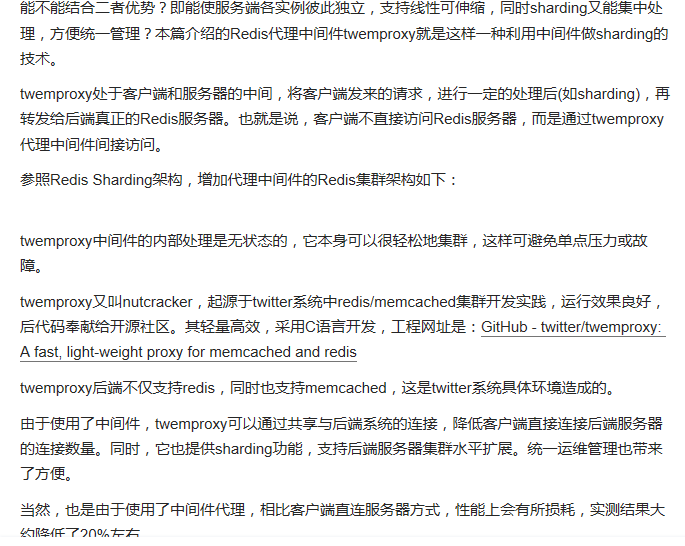








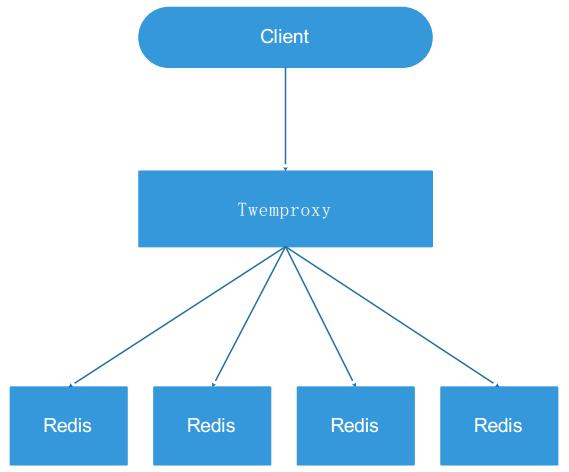


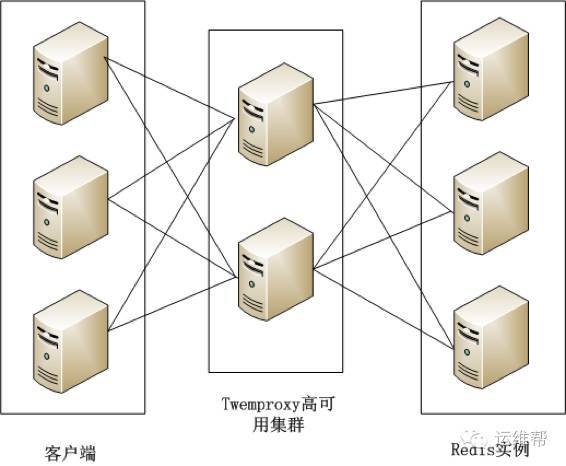


#### Twemproxy

Twemproxy是由Twitter开源的Redis代理，其基本原理是：Redis客户端把请求发送到Twemproxy，Twemproxy根据路由规则发送到正确的Redis实例，最后Twemproxy把结果汇集返回给客户端。

Twemproxy通过引入一个代理层，将多个Redis实例进行统一管理，使Redis客户端只需要在Twemproxy上进行操作，而不需要关心后面有多少个Redis实例，从而实现了Redis集群。





　Twemproxy的优点如下。

* 客户端像连接Redis实例一样连接Twemproxy，不需要改任何的代码逻辑。
* 支持无效Redis实例的自动删除。
* Twemproxy与Redis实例保持连接，减少了客户端与Redis实例的连接数。

　　Twemproxy有如下不足。

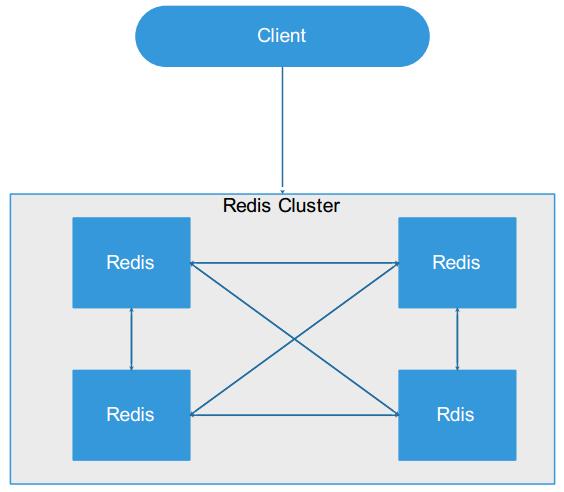
* 由于Redis客户端的每个请求都经过Twemproxy代理才能到达Redis服务器，这个过程中会产生性能损失。
* 没有友好的监控管理后台界面，不利于运维监控。
* 最大的问题是Twemproxy无法平滑地增加Redis实例。对于运维人员来说，当因为业务需要增加Redis实例时工作量非常大。

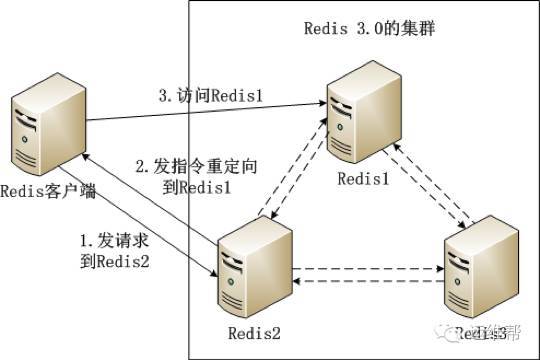
　　Twemproxy作为最被广泛使用、最久经考验、稳定性最高的Redis代理，在业界被广泛使用。

#### Redis3.0集群

Redis 3.0集群采用了P2P的模式，完全去中心化。Redis把所有的Key分成了16384个slot，每个Redis实例负责其中一部分slot。集群中的所有信息（节点、端口、slot等），都通过节点之间定期的数据交换而更新。

Redis客户端在任意一个Redis实例发出请求，如果所需数据不在该实例中，通过重定向命令引导客户端访问所需的实例。





（1） Redis客户端在Redis2实例上访问某个数据。

（2） 在Redis2内发现这个数据是在Redis3这个实例中，给Redis客户端发送一个重定向的

（3） Redis客户端收到重定向命令后，访问Redis3实例获取所需的数据。

* **优点**

a. 无中心节点

b. 数据按照 Slot 存储分布在多个 Redis 实例上

c. 平滑的进行扩容/缩容节点

d. 自动故障转移(节点之间通过 Gossip 协议交换状态信息,进行投票机制完成 Slave 到 Master 角

色的提升)

e. 降低运维成本，提高了系统的可扩展性和高可用性

* **缺点**

a. 严重依赖外部 Redis-Trib

b. 缺乏监控管理

c. 需要依赖 Smart Client(连接维护, 缓存路由表, MultiOp 和 Pipeline 支持)

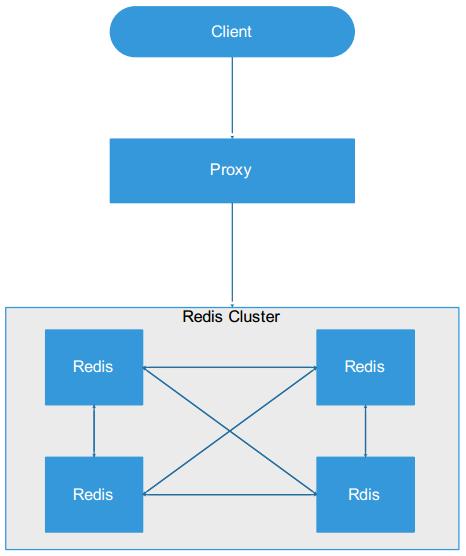
d. Failover 节点的检测过慢，不如“中心节点 ZooKeeper”及时

e. Gossip 消息的开销

f. 无法根据统计区分冷热数据

g. Slave“冷备”，不能缓解读压力

#### Proxy+Redis Cluster



优点：

Smart Client：

a. 相比于使用代理，减少了一层网络传输的消耗，效率较高。

b. 不依赖于第三方中间件，实现方法和代码自己掌控，可随时调整。

Proxy：

a. 提供一套 HTTP Restful 接口，隔离底层存储。对客户端完全透明，跨语言调用。

b. 升级维护较为容易，维护 Redis Cluster，只需要平滑升级 Proxy。

c. 层次化存储，底层存储做冷热异构存储。

d. 权限控制，Proxy 可以通过秘钥控制白名单，把一些不合法的请求都过滤掉。并

且也可以控制用户请求的超大 Value 进行控制，和过滤。

e. 安全性，可以屏蔽掉一些危险命令，比如 Keys、Save、Flush All 等。

f. 容量控制，根据不同用户容量申请进行容量限制。

g. 资源逻辑隔离，根据不同用户的 Key 加上前缀，来进行资源隔离。

h. 监控埋点，对于不同的接口进行埋点监控等信息。

缺点：

Smart Client：

a. 客户端的不成熟，影响应用的稳定性，提高开发难度。

b. MultiOp 和 Pipeline 支持有限。

c. 连接维护，Smart 客户端对连接到集群中每个结点 Socket 的维护。

Proxy：

a. 代理层多了一次转发，性能有所损耗。

b．进行扩容/缩容时候对运维要求较高，而且难以做到平滑的扩缩容

### 集群部署

<http://blog.csdn.net/dc_726/article/details/48552531>

修改配置文件redis.conf

* **绑定地址**：bind 192.168.XXX.XXX。不能绑定到127.0.0.1或localhost，否则指导客户端重定向时会报”Connection refused”的错误。
* **开启Cluster**：cluster-enabled yes
* **集群配置文件**：cluster-config-file nodes-7000.conf。这个配置文件不是要我们去配的，而是Redis运行时保存配置的文件，所以我们也不可以修改这个文件。
* **集群超时时间**：cluster-node-timeout 15000。结点超时多久则认为它宕机了。
* **槽是否全覆盖**：cluster-require-full-coverage no。默认是yes，**只要有结点宕机导致16384个槽没全被覆盖，整个集群就全部停止服务**，所以一定要改为no
* 后台运行：daemonize yes
* 输出日志：logfile “./redis.log”
* 监听端口：port 7000

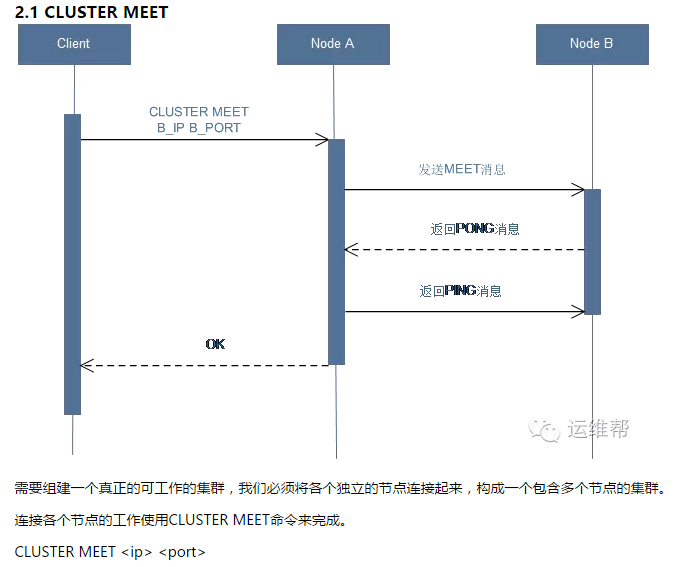
根据我们的集群规模，拷贝出来几份同样的配置文件，唯一不同的就是监听端口，可以依次改为7001、7002… 因为Redis Cluster如果数据冗余是1的话，至少要3个Master和3个Slave，所以我们拷贝出6个实例的配置文件。为了避免相互影响，为6个实例的配置文件建立独立的文件夹。



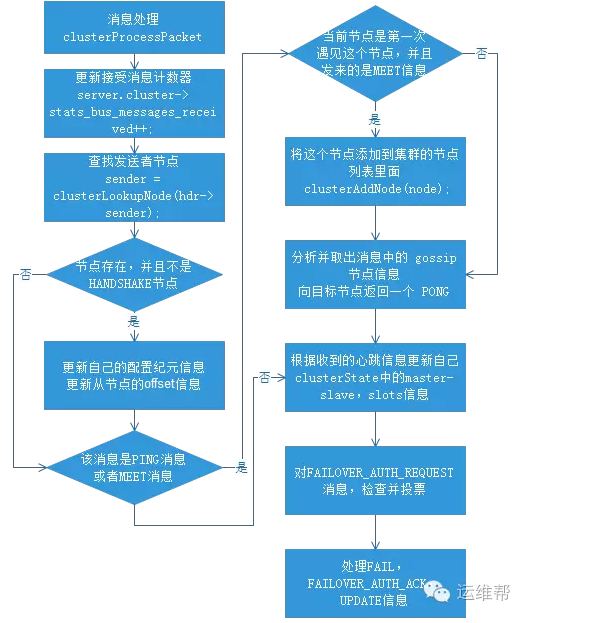
启动6个redis实例，此时还没有组建成集群。

用redis-trb.rb管理脚本建立起集群。可以看到，redis-trib默认用前3个实例作为Master，后3个作为Slave。因为Redis基于Master-Slave做数据备份，而非像Cassandra或Hazelcast一样不区分结点角色，自动复制并分配Slot的位置到各个结点。

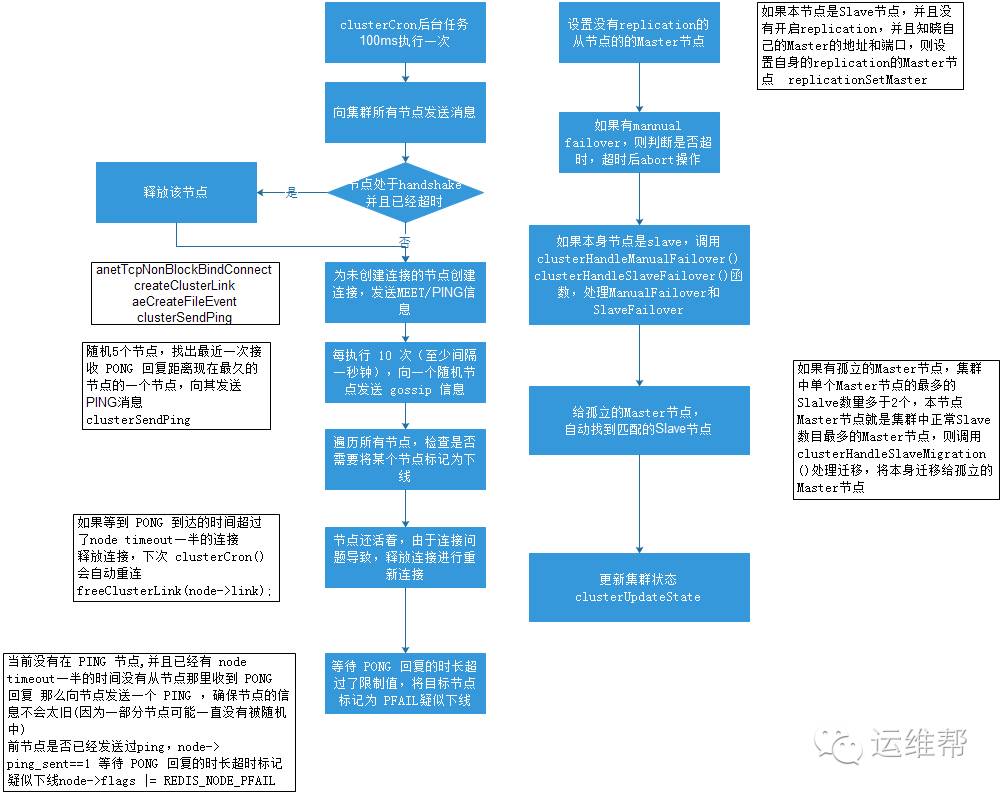
### 集群通信

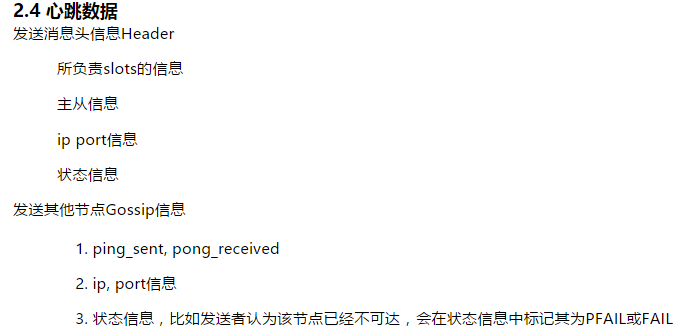
在最开始的时候，每个redis实例自己是一个集群，通过cluster meet让各个节点握手。

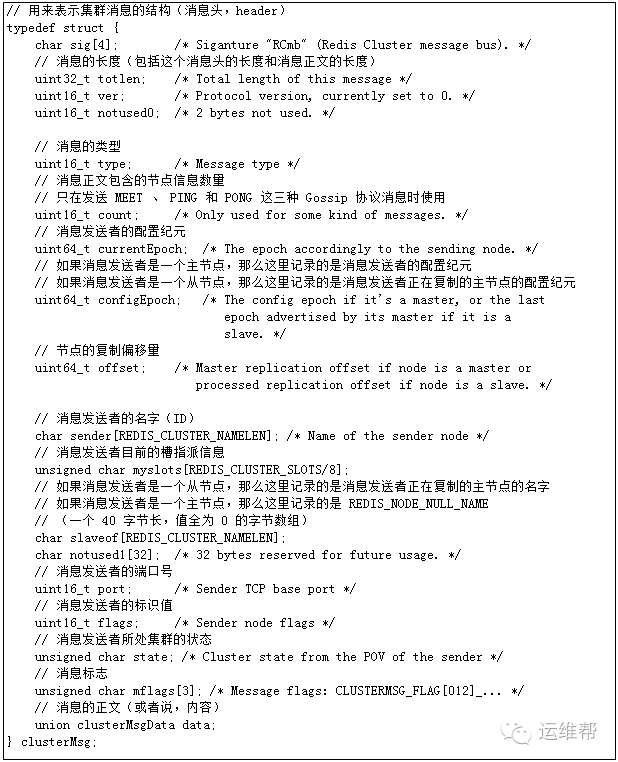
1. 节点 A 会为节点 B 创建一个 clusterNode 结构，并将该结构添加到自己的 clusterState.nodes 字典里面。
2. 节点A根据CLUSTER MEET命令给定的IP地址和端口号，向节点B发送一条MEET消息。
3. 节点B接收到节点A发送的MEET消息，节点B会为节点A创建一个clusterNode结构，并将该结构添加到自己的clusterState.nodes字典里面。
4. 节点B向节点A返回一条PONG消息。
5. 节点A将受到节点B返回的PONG消息，通过这条PONG消息节点A可以知道节点B已经成功的接收了自己发送的MEET消息。
6. 之后，节点A将向节点B返回一条PING消息。
7. 节点B将接收到的节点A返回的PING消息，通过这条PING消息节点B可以知道节点A已经成功的接收到了自己返回的PONG消息，握手完成。
8. 之后，节点A会将节点B的信息通过Gossip协议传播给集群中的其他节点，让其他节点也与节点B进行握手，最终，经过一段时间后，节点B会被集群中的所有节点认识。



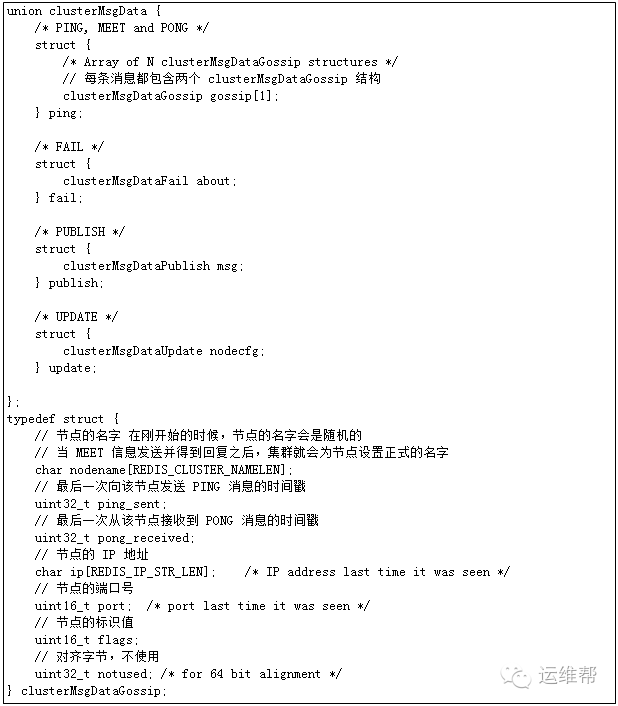








clusterMsg结构的currentEpoch、sender、myslots等属性记录了发送者自身的节点信息，接收者会根据这些信息，在自己的clusterState.nodes字典里找到发送者对应的clusterNode结构，并对结构进行更新。

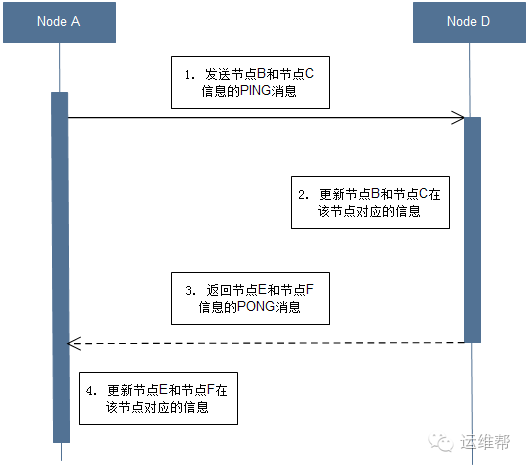


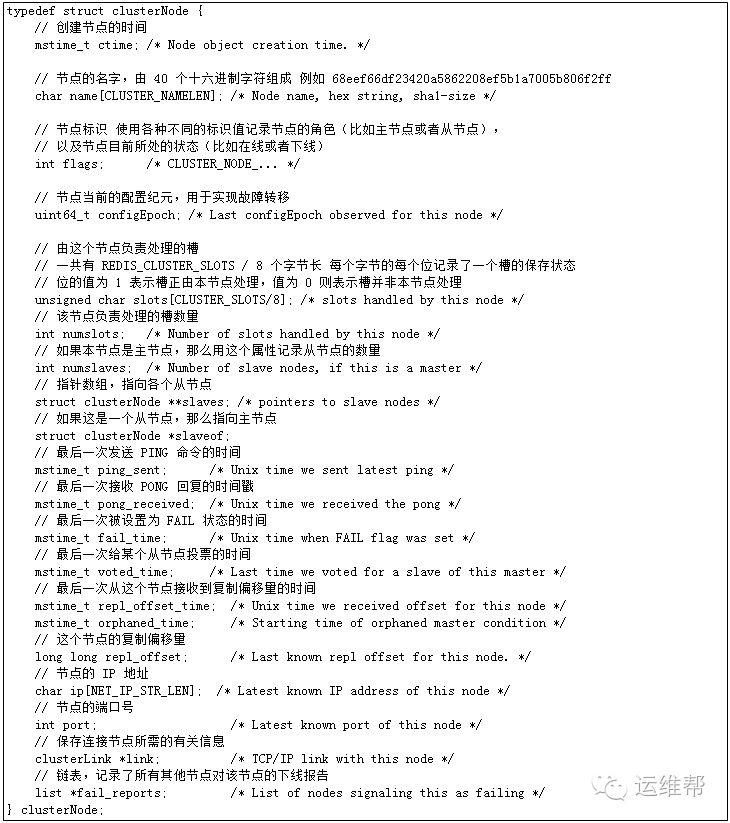
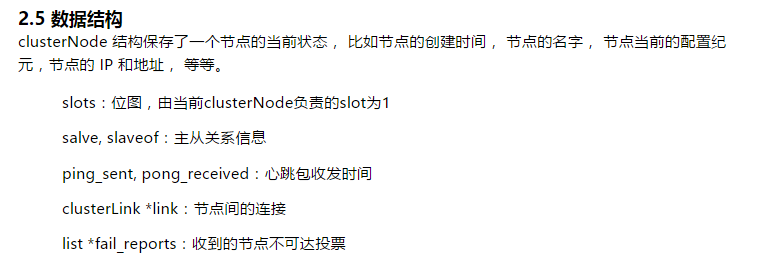
Redis集群中的各个节点通过Gossip协议来交换各自关于不同节点的状态信息，其中Gossip协议由MEET、PING、PONG三种消息实现，这三种消息的正文都由两个clusterMsgDataGossip结构组成。

每次发送MEET、PING、PONG消息时，发送者都从自己的已知节点列表中随机选出两个节点(可以是主节点或者从节点),并将这两个被选中节点的信息分别保存到两个结构中。

当接收者收到消息时，接收者会访问消息正文中的两个结构，并根据自己是否认识clusterMsgDataGossip结构中记录的被选中节点进行操作：

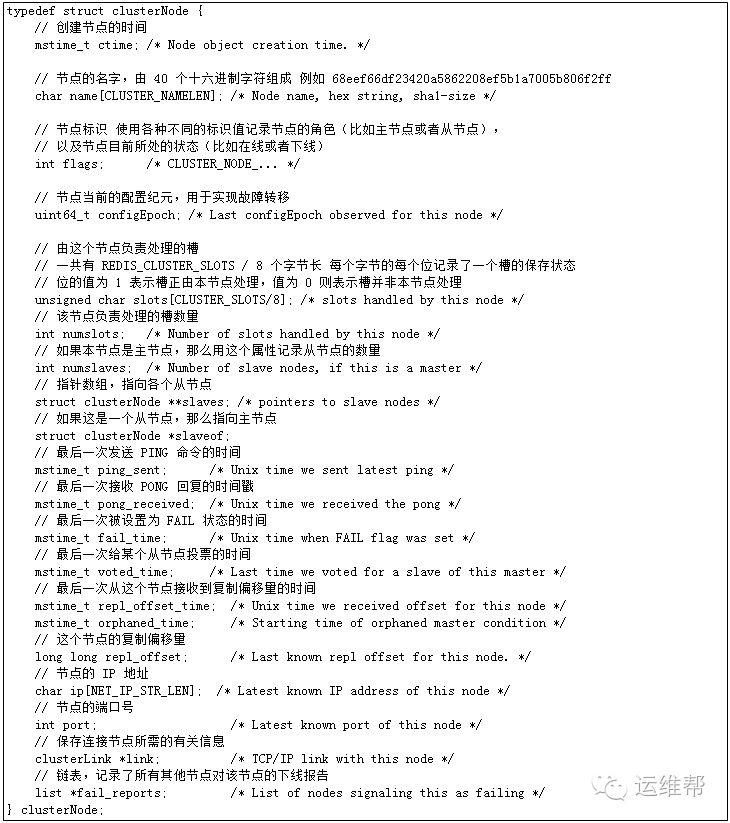
1. 如果被选中节点不存在于接收者的已知节点列表，那么说明接收者是第一次接触到被选中节点，接收者将根据结构中记录的IP地址和端口号等信息，与被选择节点进行握手。
2. 如果被选中节点已经存在于接收者的已知节点列表，那么说明接收者之前已经与被选中节点进行过接触，接收者将根据clusterMsgDataGossip结构记录的信息，对被选中节点对应的clusterNode结构进行更新。



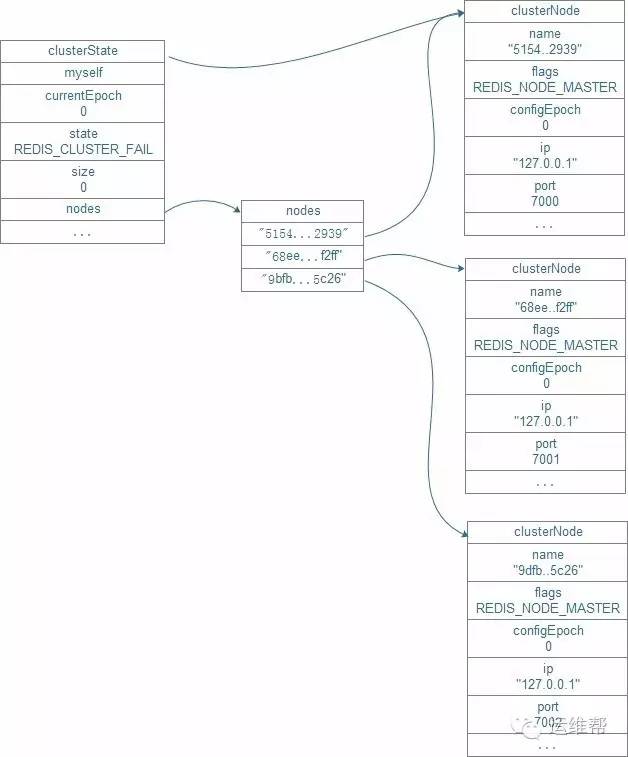
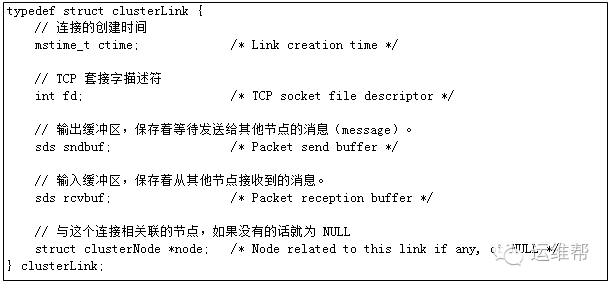


clusterState 结构记录了在当前节点的集群目前所处的状态。

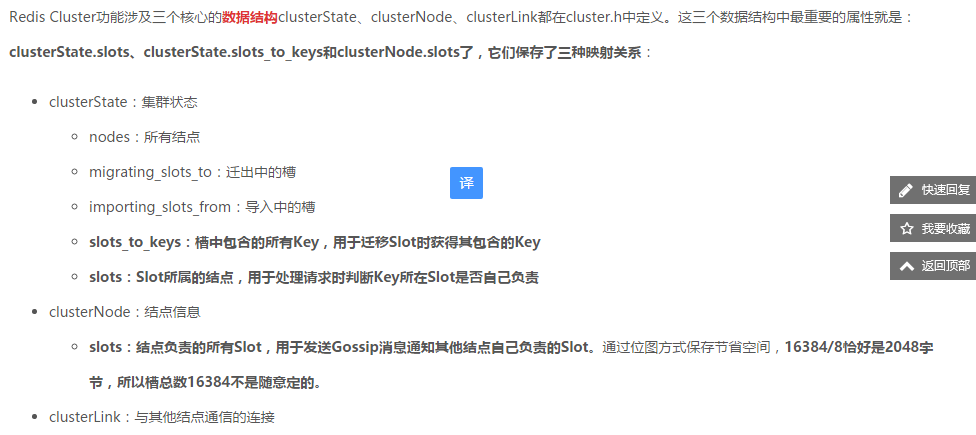
1. myself：指针指向自己的clusterNode
2. currentEpoch：当前节点的最大epoch，可能在心跳包的处理中更新
3. nodes：当前节点记录的所有节点，为clusterNode指针数组
4. slots：slot与clusterNode指针映射关系
5. migrating\_slots\_to,importing\_slots\_from：记录slots的迁移信息
6. failover\_auth\_time,failover\_auth\_count,failover\_auth\_sent,failover\_auth\_rank,failover\_auth\_epoch：Failover相关信息



clusterLink 结构保存了连接节点所需的有关信息， 比如套接字描述符， 输入缓冲区和输出缓冲区。



### 内部数据结构



// 集群状态，每个节点都保存着一个这样的状态，记录了它们眼中的集群的样子。

// 另外，虽然这个结构主要用于记录集群的属性，但是为了节约资源，

// 有些与节点有关的属性，比如 slots\_to\_keys 、 failover\_auth\_count

// 也被放到了这个结构里面。

typedef struct clusterState {

...

// 指向当前节点的指针

clusterNode \*myself; /\* This node \*/

// 集群当前的状态：是在线还是下线

int state; /\* REDIS\_CLUSTER\_OK, REDIS\_CLUSTER\_FAIL, ... \*/

// 集群节点名单（包括 myself 节点）

// 字典的键为节点的名字，字典的值为 clusterNode 结构

dict \*nodes; /\* Hash table of name -> clusterNode structures \*/

// 记录要从当前节点迁移到目标节点的槽，以及迁移的目标节点

// migrating\_slots\_to[i] = NULL 表示槽 i 未被迁移

// migrating\_slots\_to[i] = clusterNode\_A 表示槽 i 要从本节点迁移至节点 A

clusterNode \*migrating\_slots\_to[REDIS\_CLUSTER\_SLOTS];

// 记录要从源节点迁移到本节点的槽，以及进行迁移的源节点

// importing\_slots\_from[i] = NULL 表示槽 i 未进行导入

// importing\_slots\_from[i] = clusterNode\_A 表示正从节点 A 中导入槽 i

clusterNode \*importing\_slots\_from[REDIS\_CLUSTER\_SLOTS];

// 负责处理各个槽的节点

// 例如 slots[i] = clusterNode\_A 表示槽 i 由节点 A 处理

clusterNode \*slots[REDIS\_CLUSTER\_SLOTS];

// 跳跃表，表中以槽作为分值，键作为成员，对槽进行有序排序

// 当需要对某些槽进行区间（range）操作时，这个跳跃表可以提供方便

// 具体操作定义在 db.c 里面

zskiplist \*slots\_to\_keys;

...

} clusterState;

// 节点状态

struct clusterNode {

...

// 节点标识

// 使用各种不同的标识值记录节点的角色（比如主节点或者从节点），

// 以及节点目前所处的状态（比如在线或者下线）。

int flags; /\* REDIS\_NODE\_... \*/

// 由这个节点负责处理的槽

// 一共有 REDIS\_CLUSTER\_SLOTS / 8 个字节长

// 每个字节的每个位记录了一个槽的保存状态

// 位的值为 1 表示槽正由本节点处理，值为 0 则表示槽并非本节点处理

// 比如 slots[0] 的第一个位保存了槽 0 的保存情况

// slots[0] 的第二个位保存了槽 1 的保存情况，以此类推

unsigned char slots[REDIS\_CLUSTER\_SLOTS/8]; /\* slots handled by this node \*/

// 指针数组，指向各个从节点

struct clusterNode \*\*slaves; /\* pointers to slave nodes \*/

// 如果这是一个从节点，那么指向主节点

struct clusterNode \*slaveof; /\* pointer to the master node \*/

...

};

/\* clusterLink encapsulates everything needed to talk with a remote node. \*/

// clusterLink 包含了与其他节点进行通讯所需的全部信息

typedef struct clusterLink {

...

// TCP 套接字描述符

int fd; /\* TCP socket file descriptor \*/

// 与这个连接相关联的节点，如果没有的话就为 NULL

struct clusterNode \*node; /\* Node related to this link if any, or NULL \*/

...

} clusterLink;

### 数据分布及槽信息

#### 概念

Redis Cluster中有一个16384长度的槽的概念，他们的编号为0、1、2、3……16382、16383。这个槽是一个虚拟的槽，并不是真正存在的。正常工作的时候，Redis Cluster中的每个Master节点都会负责一部分的槽，当有某个key被映射到某个Master负责的槽，那么这个Master负责为这个key提供服务，至于哪个Master节点负责哪个槽，这是可以由用户指定的，也可以在初始化的时候自动生成（redis-trib.rb脚本）。这里值得一提的是，在Redis Cluster中，只有Master才拥有槽的所有权，如果是某个Master的slave，这个slave只负责槽的使用，但是没有所有权。

#### 数据分片

在Redis Cluster中，拥有16384个slot，这个数是固定的，存储在Redis Cluster中的所有的键都会被映射到这些slot中。数据库中的每个键都属于这 16384 个哈希槽的其中一个，集群使用公式 CRC16(key) % 16384 来计算键 key 属于哪个槽，其中 CRC16(key) 语句用于计算键 key 的 CRC16 校验和。集群中的每个节点负责处理一部分哈希槽。

#### 节点的槽指派信息

clusterNode结构的slots属性和numslot属性记录了节点负责处理那些槽：

struct clusterNode {

           //…

           unsignedchar slots[16384/8];

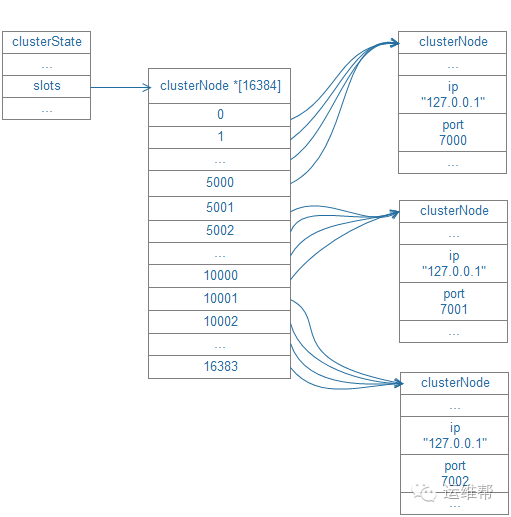
};

Slots属性是一个二进制位数组(bitarray)，这个数组的长度为16384/8=2048个字节，共包含16384个二进制位。

Master节点用bit来标识对于某个槽自己是否拥有。比如对于编号为1的槽，Master只要判断序列的第二位（索引从0开始）是不是为1即可。时间复杂度为O（1）。

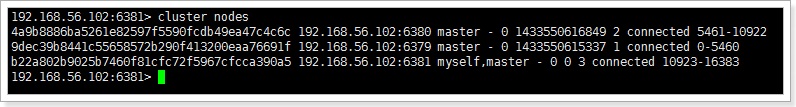
#### 集群所有槽的指派信息

通过将所有槽的指派信息保存在clusterState.slots数组里面，程序要检查槽i是否已经被指派，又或者取得负责处理槽i的节点，只需要访问clusterState.slots[i]的值即可，复杂度仅为O（1）。



#### 插槽的分配

通过cluster nodes命令可以查看当前集群的信息



该信息反映出了集群中的每个节点的id、身份、连接数、插槽数等。

当我们执行set abc 123命令时，redis是如何将数据保存到集群中的呢？执行步骤：

1. 接收命令set abc 123
2. 通过key（abc）计算出插槽值，然后根据插槽值找到对应的节点。（abc的插槽值为：7638）
3. 重定向到该节点执行命令

整个Redis提供了16384个插槽，也就是说集群中的每个节点分得的插槽数总和为16384。

./redis-trib.rb 脚本实现了是将16384个插槽平均分配给了N个节点。

**注意：如果插槽数有部分是没有指定到节点的，那么这部分插槽所对应的key将不能使用。**

Redis集群可以支持16384个节点，每个节点负责一个插槽。

#### 插槽和key关系

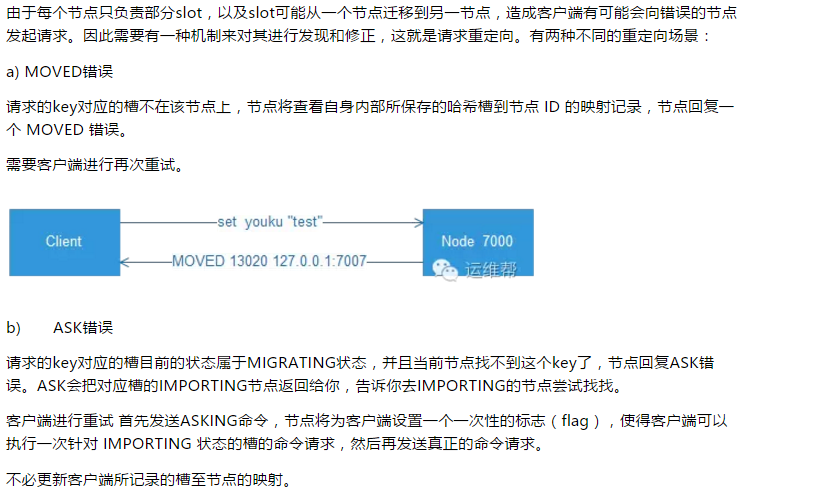
计算key的插槽值：

key的有效部分使用CRC16算法计算出哈希值，再将哈希值对16384取余，得到插槽值。

什么是有效部分？

1. 如果key中包含了{符号，且在{符号后存在}符号，并且{和}之间至少有一个字符，则有效部分是指{和}之间的部分；
   1. key={hello}\_tatao的有效部分是hello
2. 如果不满足上一条情况，整个key都是有效部分；
   1. key=hello\_taotao的有效部分是全部

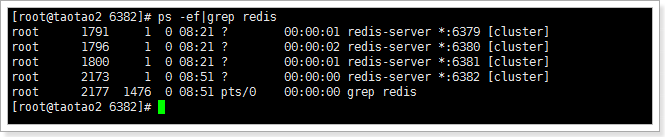
#### 请求重定向





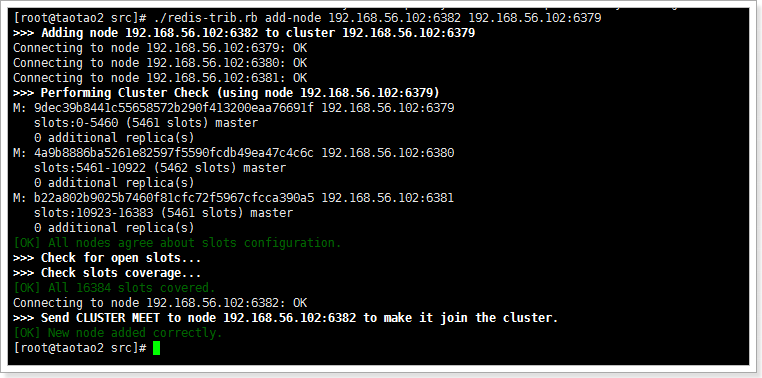
### 新增集群节点

再开启一个实例的端口为6382

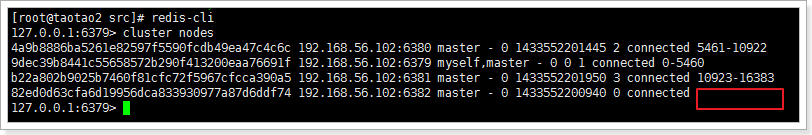


执行脚本：

./redis-trib.rb add-node 192.168.56.102:6382 192.168.56.102:6379



添加成功 查看集群信息



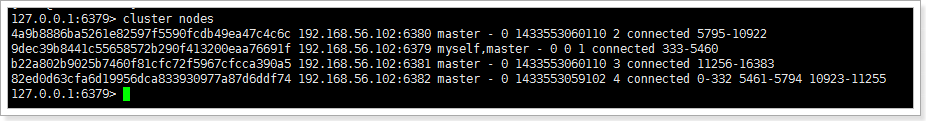
发现没有插槽数。

接下来需要给6382这个服务分配插槽，将6379的一部分（1000个）插槽分配给6382：





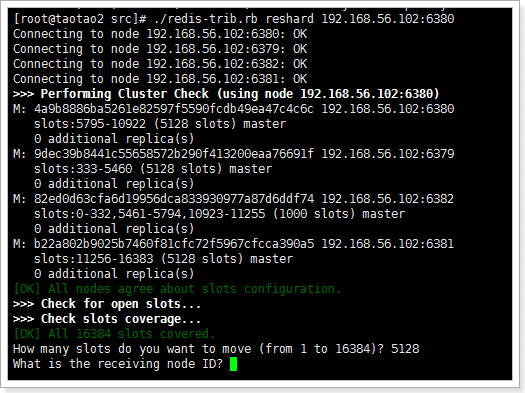
查看节点情况



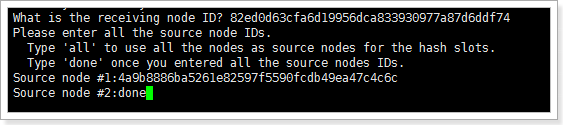
### 删除集群中的节点

想要删除集群节点中的某一个节点，需要严格执行2步：

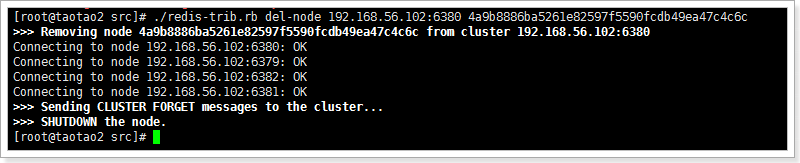
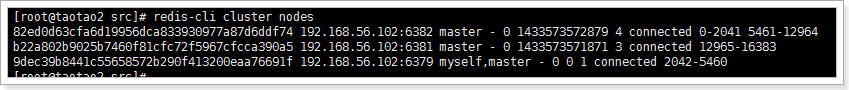
1. 将这个节点上的所有插槽转移到其他节点上；
   1. 假设我们想要删除6380这个节点
   2. 执行脚本：./redis-trib.rb reshard 192.168.56.102:6380
   3. 选择需要转移的插槽的数量，因为3380有5128个，所以转移5128个



* 1. 输入转移的节点的id，我们转移到6382节点：82ed0d63cfa6d19956dca833930977a87d6ddf7
  2. 输入插槽来源id，也就是6380的id
  3. 输入done，开始转移



* 1. 查看集群信息，可以看到6380节点已经没有插槽了。

1. **使用redis-trib.rb删除节点**
   1. ./redis-trib.rb del-node 192.168.56.102:6380 4a9b8886ba5261e82597f5590fcdb49ea47c4c6c
   2. del-node host:port node\_id
   3. 
   4. 查看集群信息，可以看到已经没有6380这个节点了。  
      

### 数据迁移

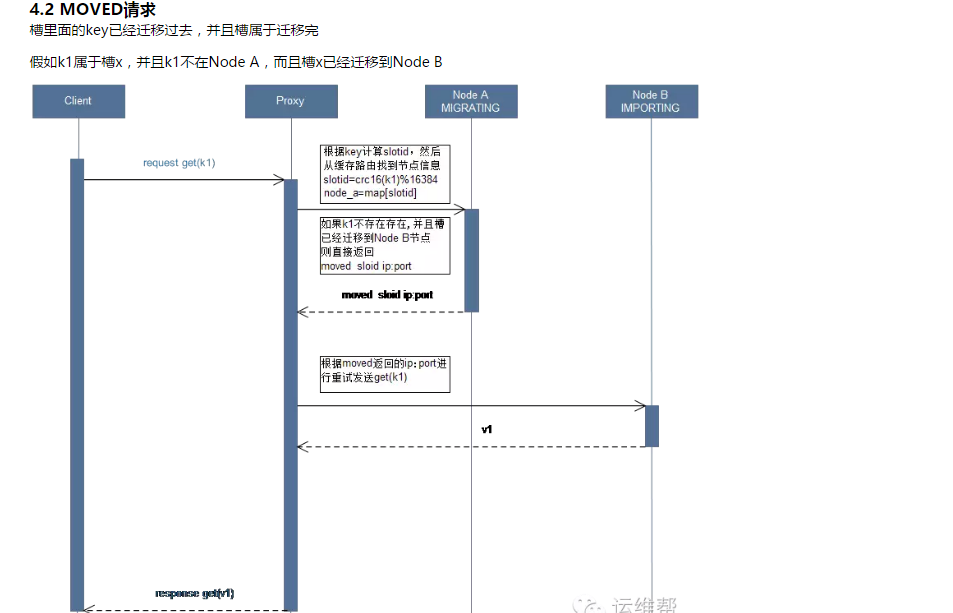
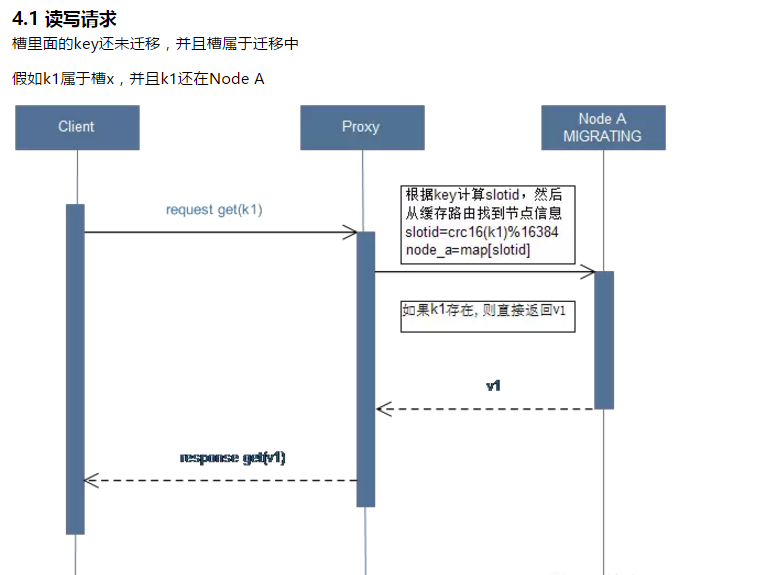
槽x从Node A向Node B迁移时，Node A和Node B都会有这个槽x，Node A上槽x的状态设置为MIGRATING，Node B上槽x的状态被设置为IMPORTING。

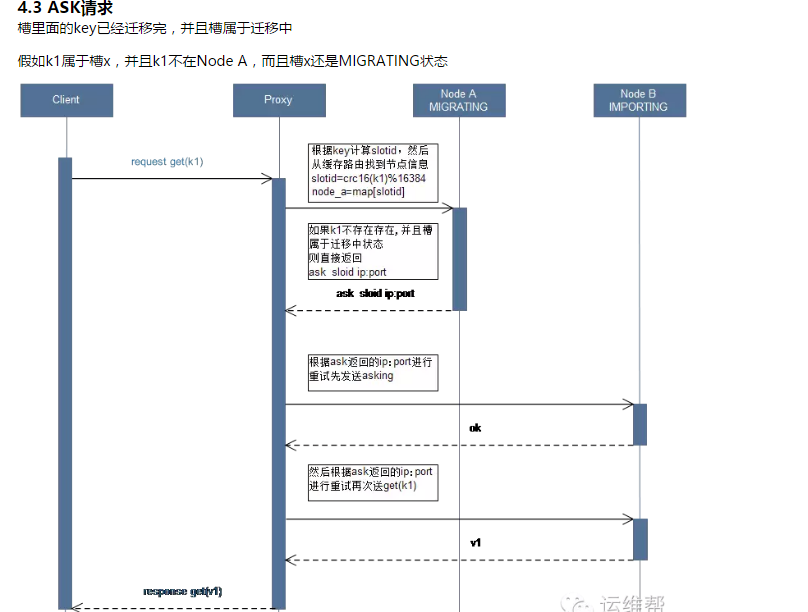
**MIGRATING状态**

1. 如果key存在则成功处理
2. 如果key不存在，则返回客户端ASK，客户端根据ASK首先发送ASKING命令到目标节点，然后发送请求的命令到目标节点
3. 当key包含多个命令，
   1. 如果都存在则成功处理
   2. 如果都不存在，则返回客户端ASK
   3. 如果一部分存在，则返回客户端TRYAGAIN，通知客户端稍后重试，这样当所有的key都    迁移完毕的时候客户端重试请求的时候回得到ASK，然后经过一次重定向就可以获取这批键
4. 此时不刷新客户端中node的映射关系

**IMPORTING状态**

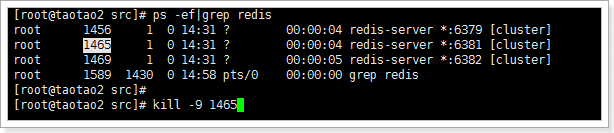
1. 如果key不在该节点上，会被MOVED重定向，刷新客户端中node的映射关系
2. 如果是ASKING命令则命令会被执行，key不在迁移的节点已经被迁移到目标的节点
3. Key不存在则新建

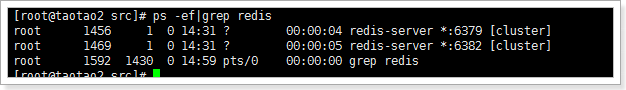




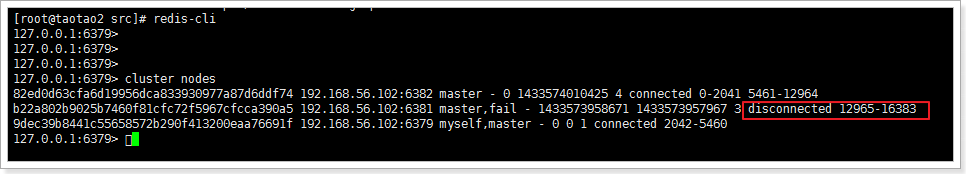
### 通信故障

如果集群中的某一节点宕机会出现什么状况？我们这里假设6381宕机。





尝试连接下集群，并且查看集群信息，发现6381的节点断开连接：



我们尝试执行set命令，结果发现无法执行：

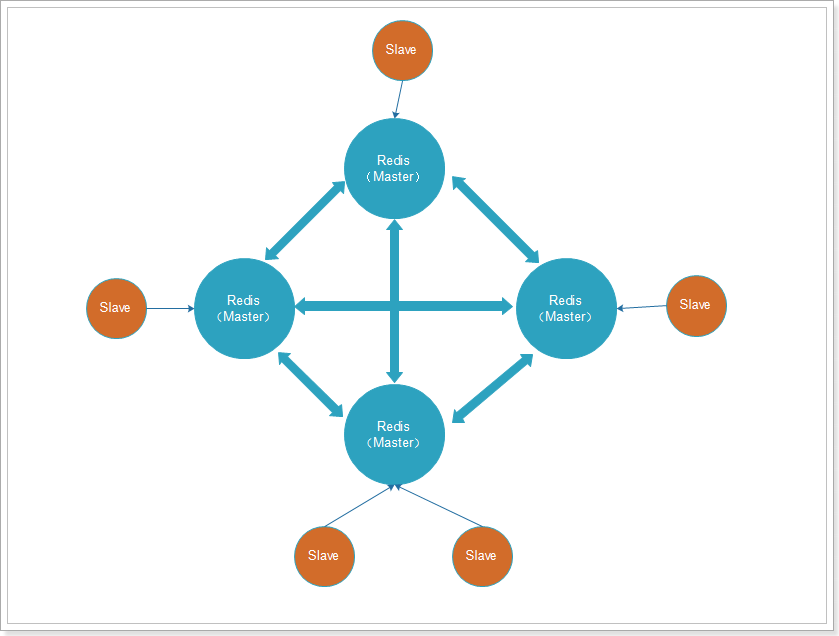


因为插槽数不完整，整个集群不可用。。有点弱。。。

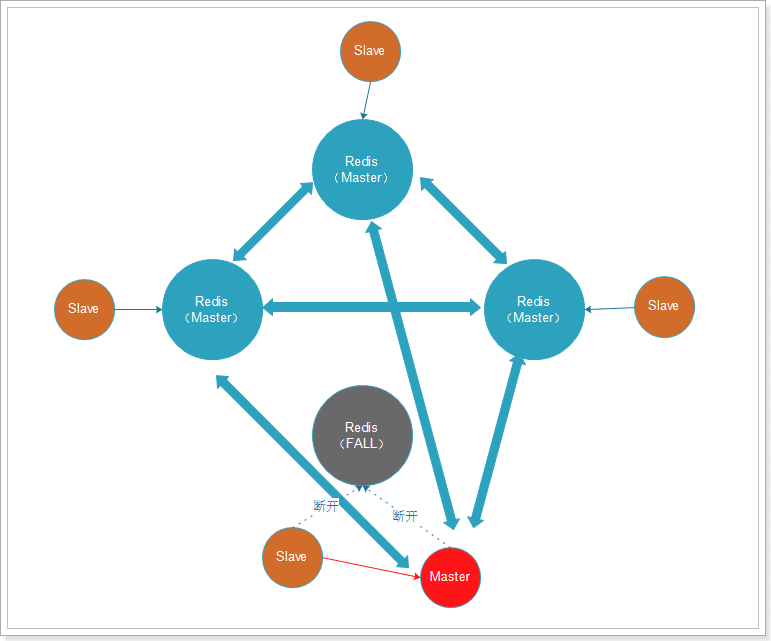
#### 故障机制

1. 集群中的每个节点都会定期的向其它节点发送PING命令，并且通过有没有收到回复判断目标节点是否下线；
2. 集群中每一秒就会随机选择几个节点，然后选择其中最久没有响应的节点放PING命令；
3. 如果一定时间内目标节点都没有响应，那么该节点就认为目标节点**疑似下线**；
4. 当集群中的节点超过半数认为该目标节点疑似下线，那么该节点就会被标记为**下线**；
5. 当集群中的任何一个节点下线，就会导致插槽区有空档，不完整，那么该集群将不可用；
6. 如何解决上述问题？
   1. 在Redis集群中可以使用主从模式实现某一个节点的高可用
   2. 当该节点（master）宕机后，集群会将该节点的从数据库（slave）转变为（master）继续完成集群服务；

#### 集群中的主从复制架构



出现故障:

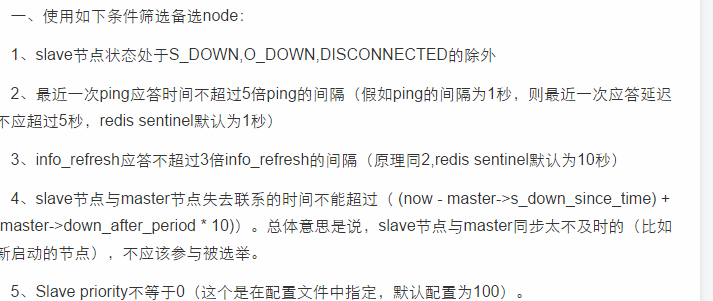


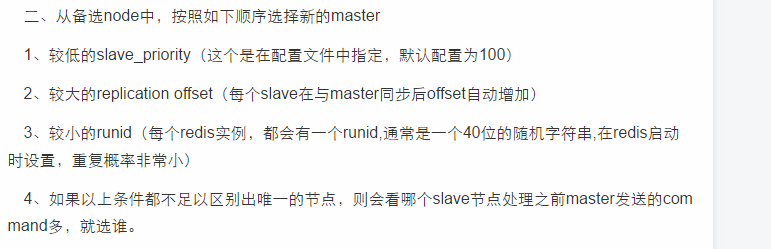
启动6个redis实例后，创建集群：

创建集群，指定了从库数量为1，创建顺序为主库（3个）、从库（3个）：

./redis-trib.rb create --replicas 1 192.168.56.102:6379 192.168.56.102:6380 192.168.56.102:6381 192.168.56.102:6479 192.168.56.102:6480 192.168.56.102:6481

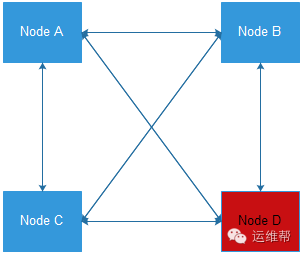
#### 投票选举机制





#### 故障检测

集群中的每个节点都会定期地向集群中的其他节点发送PING消息，以此交换各个节点状态信息，检测各个节点状态：在线状态、疑似下线状态PFAIL、已下线状态FAIL。



当主节点A通过消息得知主节点B认为主节点D进入了疑似下线(PFAIL)状态时,

主节点A会在自己的clusterState.nodes字典中找到主节点D所对应的clusterNode结构，并将主节点B的下线报告(failure report)添加到clusterNode结构的fail\_reports链表中

struct clusterNode {

           //...

           //记录所有其他节点对该节点的下线报告

           list\*fail\_reports;

           //...

};

每个下线报告由一个clusterNodeFailReport结构：

struct clusterNodeFailReport  {

           //报告目标节点已经下线的节点

           structclusterNode \*node;

           //最后一次从node节点收到下线报告的时间

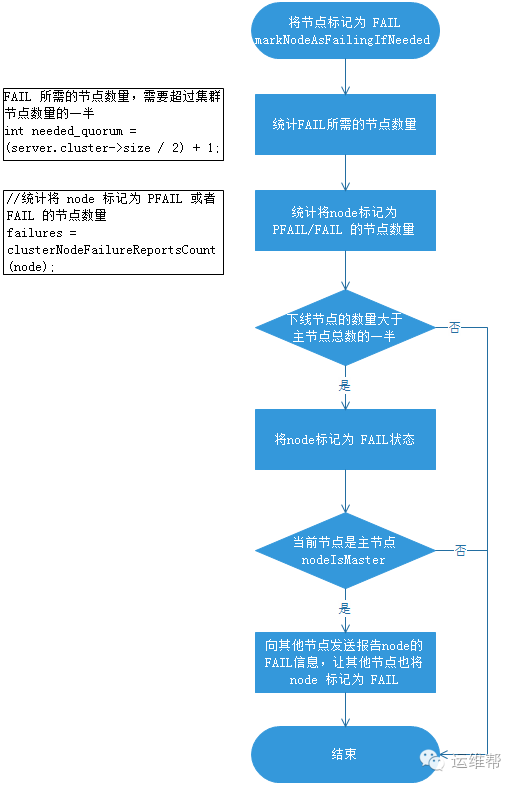
           mstime\_ttime;

}typedef clusterNodeFailReport;

如果集群里面，半数以上的主节点都将主节点D报告为疑似下线，那么主节点D将被标记为已下线(FAIL)状态，将主节点D标记为已下线的节点会向集群广播主节点D的FAIL消息，所有收到FAIL消息的节点都会立即更新nodes里面主节点D状态标记为已下线。

将 node 标记为 FAIL 需要满足以下两个条件：

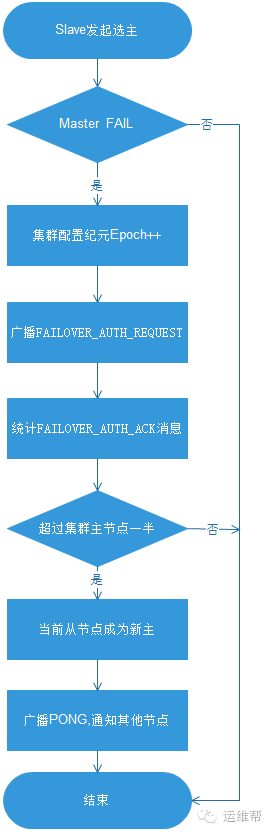
1. 有半数以上的主节点将 node 标记为 PFAIL 状态。
2. 当前节点也将 node 标记为 PFAIL 状态。



#### 多个从节点选主

**选新主的过程基于Raft协议选举方式来实现的**

1. 当从节点发现自己的主节点进行已下线状态时，从节点会广播一条CLUSTERMSG\_TYPE\_FAILOVER\_AUTH\_REQUEST消息，要求所有收到这条消息，并且具有投票权的主节点向这个从节点投票
2. 如果一个主节点具有投票权，并且这个主节点尚未投票给其他从节点，那么主节点将向要求投票的从节点返回一条，CLUSTERMSG\_TYPE\_FAILOVER\_AUTH\_ACK消息，表示这个主节点支持从节点成为新的主节点
3. 每个参与选举的从节点都会接收CLUSTERMSG\_TYPE\_FAILOVER\_AUTH\_ACK消息,并根据自己收到了多少条这种消息来统计自己获得了多少主节点的支持
4. 如果集群里有N个具有投票权的主节点，那么当一个从节点收集到大于等于集群N/2+1张支持票时，这个从节点就成为新的主节点
5. 如果在一个配置纪元没有从能够收集到足够的支持票数，那么集群进入一个新的配置纪元，并再次进行选主，直到选出新的主节点为止



#### 故障转移

当从节点发现自己的主节点变为已下线(FAIL)状态时，便尝试进Failover，以期成为新的主。

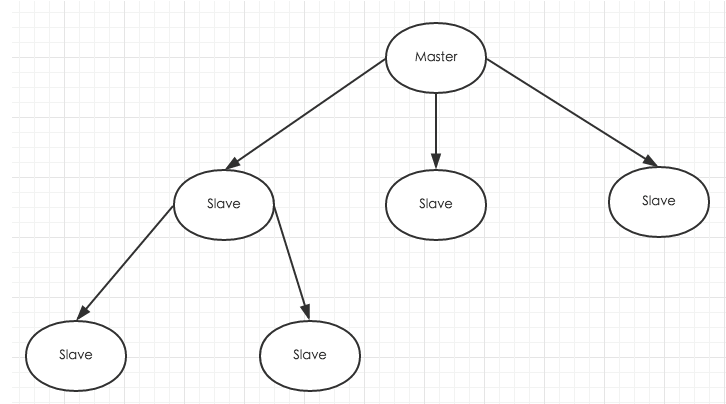
以下是故障转移的执行步骤：

1. 从下线主节点的所有从节点中选中一个从节点
2. 被选中的从节点执行SLAVEOF NO NOE命令，成为新的主节点
3. 新的主节点会撤销所有对已下线主节点的槽指派，并将这些槽全部指派给自己
4. 新的主节点对集群进行广播PONG消息，告知其他节点已经成为新的主节点
5. 新的主节点开始接收和处理槽相关的请求

### 使用集群注意的事项

1. **多键的命令操作（如MGET、MSET），如果每个键都位于同一个节点，则可以正常支持，否则会提示错误。**
2. **集群中的节点只能使用0号数据库，如果执行SELECT切换数据库会提示错误。**

## Redis主从同步

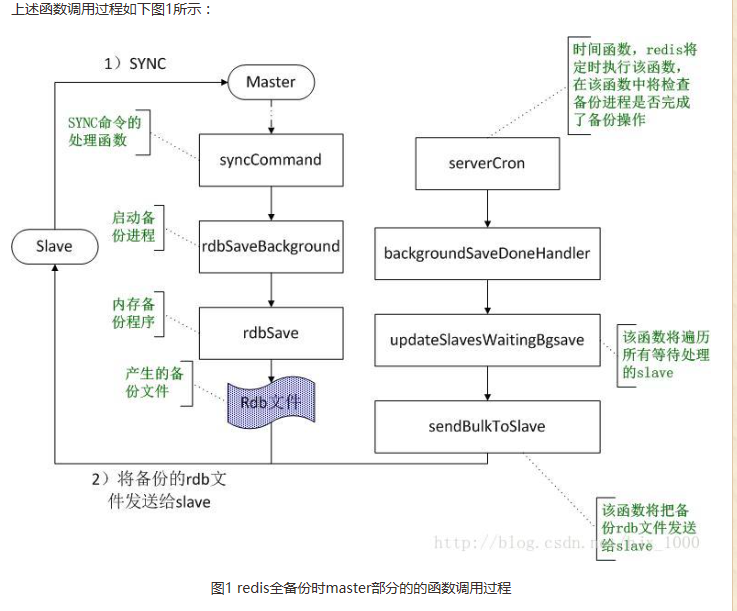


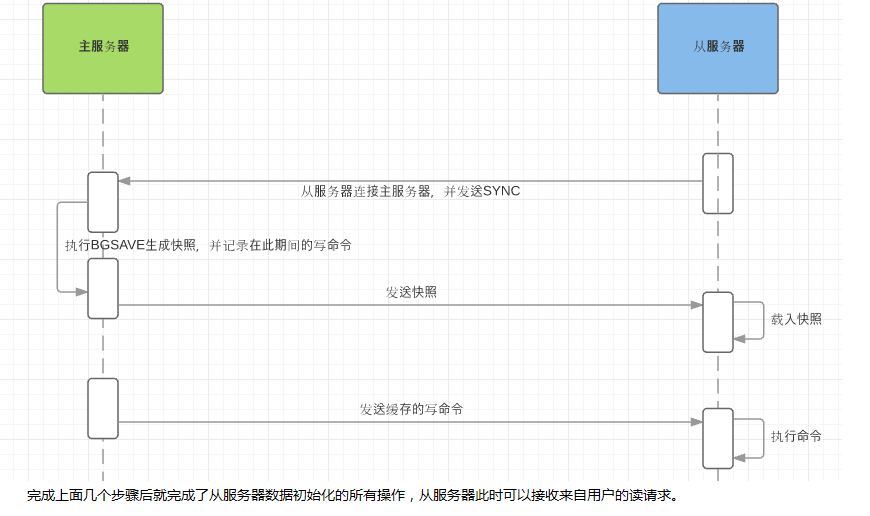
Redis主从复制可以根据是否是全量分为全量同步和增量同步。

**1 全量同步**

Redis全量复制一般发生在Slave初始化阶段，这时Slave需要将Master上的所有数据都复制一份。具体步骤如下：   
　　1）从服务器连接主服务器，发送SYNC命令；   
　　2）主服务器接收到SYNC命名后，开始执行BGSAVE命令生成RDB文件并使用缓冲区记录此后执行的所有写命令；   
　　3）主服务器BGSAVE执行完后，向所有从服务器发送快照文件，并在发送期间继续记录被执行的写命令；   
　　4）从服务器收到快照文件后丢弃所有旧数据，载入收到的快照；   
　　5）主服务器快照发送完毕后开始向从服务器发送缓冲区中的写命令；   
　　6）从服务器完成对快照的载入，开始接收命令请求，并执行来自主服务器缓冲区的写命令；

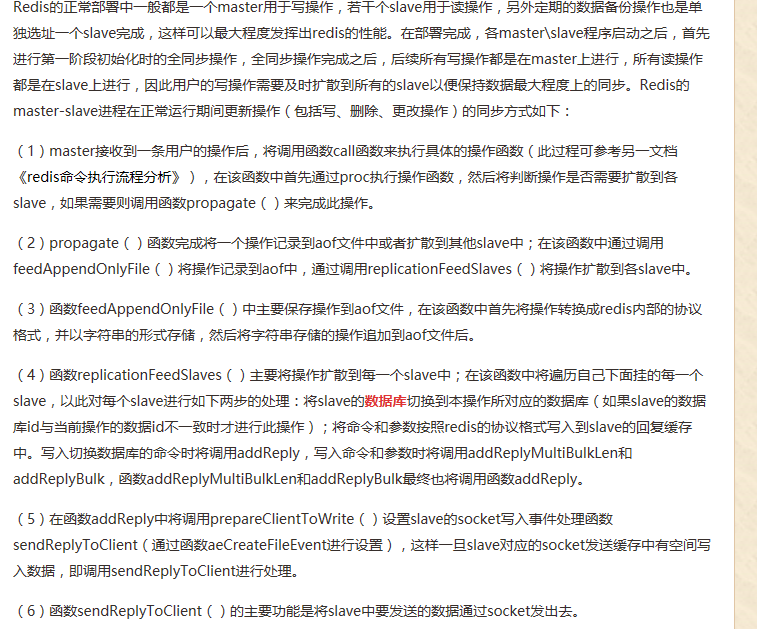


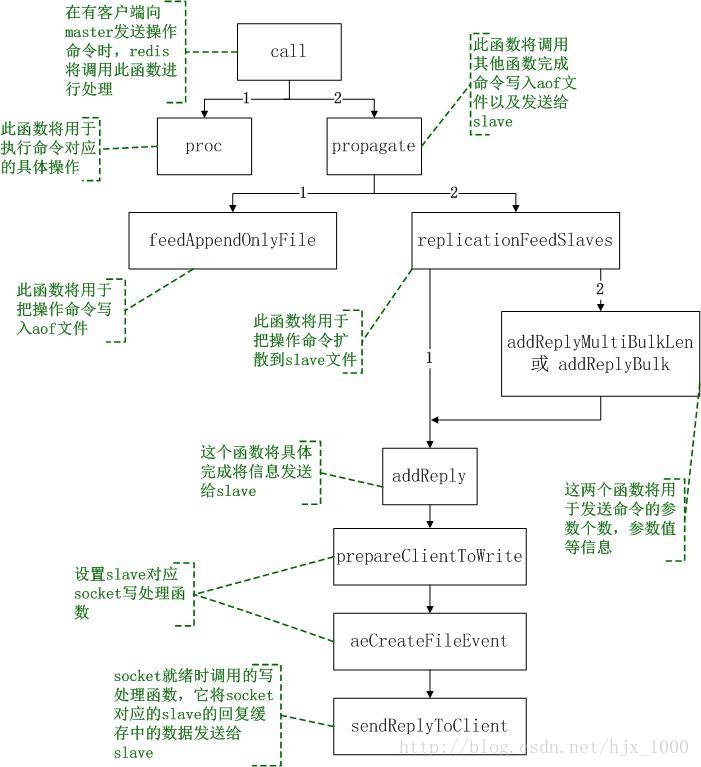




**2 增量同步**

Redis增量复制是指Slave初始化后开始正常工作时主服务器发生的写操作同步到从服务器的过程。   
增量复制的过程主要是主服务器每执行一个写命令就会向从服务器发送相同的写命令，从服务器接收并执行收到的写命令。

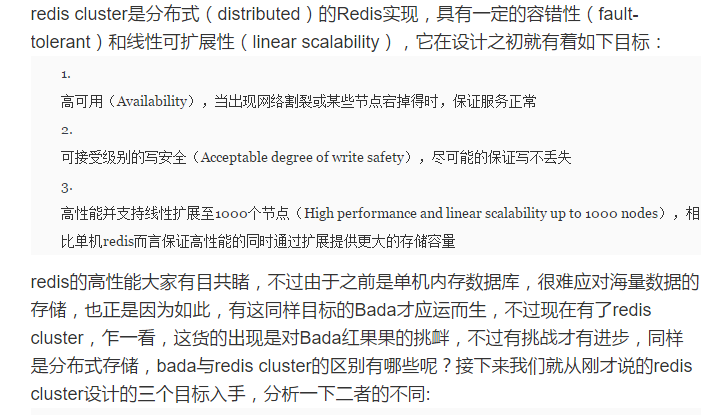


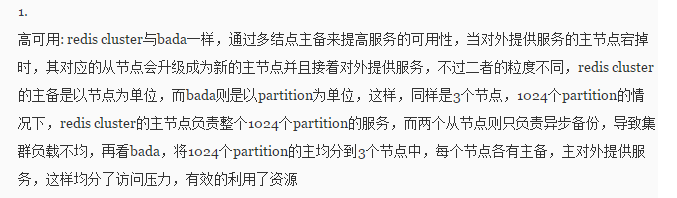


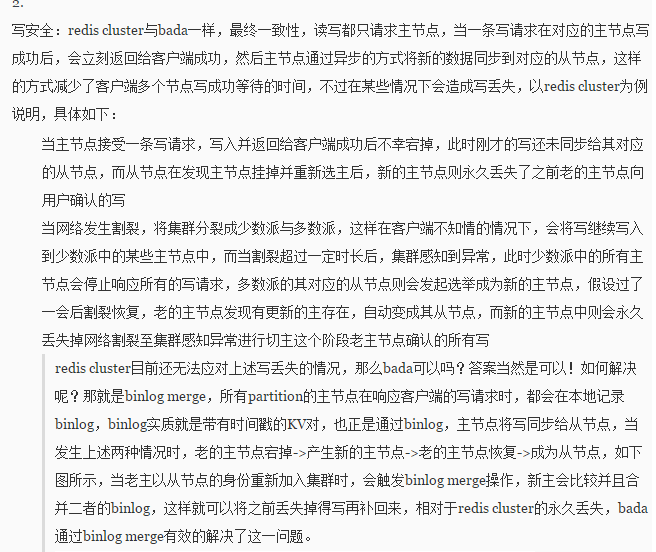
**3 Redis主从同步策略**

　　主从刚刚连接的时候，进行全量同步；全同步结束后，进行增量同步。当然，如果有需要，slave 在任何时候都可以发起全量同步。redis 策略是，无论如何，首先会尝试进行增量同步，如不成功，要求从机进行全量同步。

## redis比较









## Gossip协议

Gossip协议是基于一种叫做SWIM的协议（ **S** calable **W** eakly-consistent **I** nfection-style Process Group **M** embership Protocol）。SWIM是一种无中心的分布式协议，各个节点之间通过p2p实现信息交流同步各节点状态的方法。这是一种弱一致性的实现。

SWIM协议给每个进程组成员在本地维护一个成员表，记录该组存活的进程。该协议通过失效检测器（Failure Detector）和传播组件（Dissemination Component）来完成工作。

SWIM的失效检测器会检测失效的节点并将失效节点的更新信息发送给传播组件。SWIM的传播组件通过多播（multicast）的形式将失效信息传播给组内的其他成员。

协议的可扩展性体现在：新成员的加入和退出也以同样的方式进行多播通信。而在基本的时间周期内进行失效检测能够保证在限定的时间范围内完成完备性检查，即每个失效的进程都能最终被检测到（最终一致性）。通过多播方式传输协议消的问题在于效率不好也不可靠，通过在ping和ack消息中捎带成员更新信息能够降低丢包率和减少传输时延。这种传播方式被称为可传导的方式（Infection-style）。

在Gossip协议中，一个节点状态发生变化，并像临近节点发送更新信息，对于节点状态变化的信息随机发送给b个节点，最终，信息能够到达所有的节点。

Gossip的交互模式分为两种：Anti-entropy和Rumor mongering。

* Anti-entropy：每个节点周期性地随机选择其他节点，然后通过相互交换自己的所有数据来消除两者之间的差异。
* Rumor mongering：当一个节点有来新信息后，该节点变成活跃状态，并周期性地联系其他节点向其发送新信息。

每个节点维护一个自己的信息表 <key, (value, version)> ，即属性的值以及版本号；和一个记录其他节点的信息表 <node, <key, (value, version)>> 。每个节点和系统中的某个节点相互配对成为peer。而节点的信息交换方式主要有3种。

Push：拥有状态新信息的节点随机选择联系节点并想起发送自己得到信息。

Pull：发起信息交换的节点随机选择联系节点并从对方获取信息。

Push-Pull混合模式：发起信息交换的节点向选择的节点发送信息。

Gossip完成了状态的同步。即使有的节点因宕机而重启或者有新节点加入，但经过一段时间后，这些节点的状态也会与其他节点达成一致。也就是说，Gossip天然具有分布式容错的优点。

应用：

除了改善SWIM协议中的多播方式，Gossip还在很多地方有应用：

* 数据库复制：基于Gossip实现分布数据管理的一般思路是：一个节点实现数据更新，通过Gossip算法将更新传播导其他节点。
* 聚合计算：在无中心的系统中，没有中心节点存储全局信息。通过Gossip应用到分布环境下的聚合计算中来保证系统的发送消息的容错。

总之，Gossip简单、高效，同时具有很好的可扩展性和鲁棒性，非常适合大规模、动态、资源受限的网络环境。

## 分布式环境中的CAP理论 BASE理论

### CAP

一个分布式系统不可能同时满足一致性Consistency 可用性Availability和分区容错性Partition tolerance这三个基本需求，最多只能同时满足其中两项。

一致性Consistency：数据一致更新，所有数据变动都是同步的

可用性Availability：好的响应性能

分区容错性Partition tolerance：可靠性

关系型数据库的ACID模型是具有高一致性和可用性，很难进去分区。

#### 一致性

在分布式环境下，一致性是指数据在多个副本之间能否保持一致的特性。在一致性的需求下，当一个系统在数据一致的状态下执行更新操作后，应该保证系统的数据仍然处于一直的状态。

对于一个将数据副本分布在不同分布式节点上的系统来说，如果对第一个节点的数据进行了更新操作并且更新成功后，却没有使得第二个节点上的数据得到相应的更新，于是在对第二个节点的数据进行读取操作时，获取的依然是老数据（或称为脏数据），这就是典型的分布式数据不一致的情况。在分布式系统中，如果能够做到针对一个数据项的更新操作执行成功后，所有的用户都可以读取到其最新的值，那么这样的系统就被认为具有强一致性。

**强一致性**：它要求系统写入什么，读出来的也会是什么，用户体验好，但实现起来往往对系统的性能影响大。

**弱智一致性**：这种一致性级别约束了系统在写入成功后，不承诺立即可以读到写入的值，也不久承诺多久之后数据能够达到一致，但会尽可能地保证到某个时间级别（比如秒级别）后，数据能够达到一致状态。

**最终一致性**：最终一致性是弱一致性的一个特例，系统会保证在一定时间内，能够达到一个数据一致的状态。这里之所以将最终一致性单独提出来，是因为它是弱一致性中非常推崇的一种一致性模型，也是业界在大型分布式系统的数据一致性上比较推崇的模型。

#### 可用性

可用性是指系统提供的服务必须一直处于可用的状态，对于用户的每一个操作请求总是能够在有限的时间内返回结果。这里的重点是"有限时间内"和"返回结果"。

"有限时间内"是指，对于用户的一个操作请求，系统必须能够在指定的时间内返回对应的处理结果，如果超过了这个时间范围，那么系统就被认为是不可用的。另外，"有限的时间内"是指系统设计之初就设计好的运行指标，通常不同系统之间有很大的不同，无论如何，对于用户请求，系统必须存在一个合理的响应时间，否则用户便会对系统感到失望。

"返回结果"是可用性的另一个非常重要的指标，它要求系统在完成对用户请求的处理后，返回一个正常的响应结果。正常的响应结果通常能够明确地反映出队请求的处理结果，即成功或失败，而不是一个让用户感到困惑的返回结果。

#### 分区容错性

分区容错性约束了一个分布式系统具有如下特性：**分布式系统在遇到任何网络分区故障的时候，仍然需要能够保证对外提供满足一致性和可用性的服务，除非是整个网络环境都发生了故障**。

网络分区是指在分布式系统中，不同的节点分布在不同的子网络（机房或异地网络）中，由于一些特殊的原因导致这些子网络出现网络不连通的状况，但各个子网络的内部网络是正常的，从而导致整个系统的网络环境被切分成了若干个孤立的区域。需要注意的是，组成一个分布式系统的每个节点的加入与退出都可以看作是一个特殊的网络分区。

另外一种解释，当网络发生异常时，导致分布式系统中部分节点之间的网络延时不断增大，最终导致组成分布式系统的所有节点中，只有部分节点之间能够正常通信，而另外一些节点则不能。将这个现象称为网络分区。当网络分区出现时，分布式系统会出现局部小集群，在极端情况下，这些局部小集群会独立完成原本需要整个分布式系统才能完成的功能，包括对数据的事物处理，这就对分布式一致性提出了非常大的挑战。

（通信异常：从集中式向分布式演变过程中，必然引入网络因素，由于网络本身不可靠，因此也引入了额外的问题。分布式系统需要在各个节点之间进行网络通信，每次网络通信都会伴随着网络不可用的风险（网络或者路由器等硬件设备不可用），这都会导致分布式系统无法顺利完成一次网络通信。）

例子：一个分布式系统需要有3个备份，其分区容错性为2，即要求必须有2个备份才能继续提供服务。因此如果其中一个备份的机器节点不可达，由于分区容错性的存在，其可以继续提供服务，并且分布式系统会根据情况等待该分区重新连接，恢复失去连接时的新增数据，之后再提供服务。或者由于节点宕机，会重新产生一个分区，并同步所有的数据（时间较长），等数据同步完成后再开始进行服务。如果一个分布式系统不允许分区容错性，那么只要一个备份数据不可用，整个服务都不可用，这是不能接受的。

CAP是不可能同时满足。对于一个分布式系统来说，p这个属性是必备的。（由于网络问题可能导致某个节点和其他节点失去联系，这时候就形成了Partition，也就是说由于网络问题将系统中的节点隔离成了2个区域，互相无法知道对方的状态。这在分布式系统中非常常见）

当一个分布式系统满足分区容错性的情况下，如果选择可用性A，那么失去联系的那个节点依然可以向系统提供服务，但是它的数据不能保证是同步的。如果选择一致性C，那么必须等待失去联系的节点恢复过来，在这个过程中，那个节点是不允许对外提供服务的，这时候系统就处于不可用状态，失去了A。

微博可以用CP

OA ERP系统用AP

|  |  |
| --- | --- |
| CA | 放弃分区容错性，加强一致性和可用性，其实就是传统的单机数据库的选择 |

AP 放弃一致性（这里说的一致性是强一致性），追求分区容错性和可用性，这是很多分布式系统设计时的选择，例如很多NoSQL系统就是如此

|  |  |
| --- | --- |
| CP | 放弃可用性，追求一致性和分区容错性，基本不会选择，网络问题会直接让整个系统不可用 |

需要明确的一点是，对于一个分布式系统而言，分区容错性是一个最基本的要求。因为既然是一个分布式系统，那么分布式系统中的组件必然需要被部署到不同的节点，否则也就无所谓分布式系统了，因此必然出现子网络。而对于分布式系统而言，网络问题又是一个必定会出现的异常情况，因此分区容错性也就成为了一个分布式系统必然需要面对和解决的问题。因此系统架构师往往需要把精力花在如何根据业务特点在C（一致性）和A（可用性）之间寻求平衡。

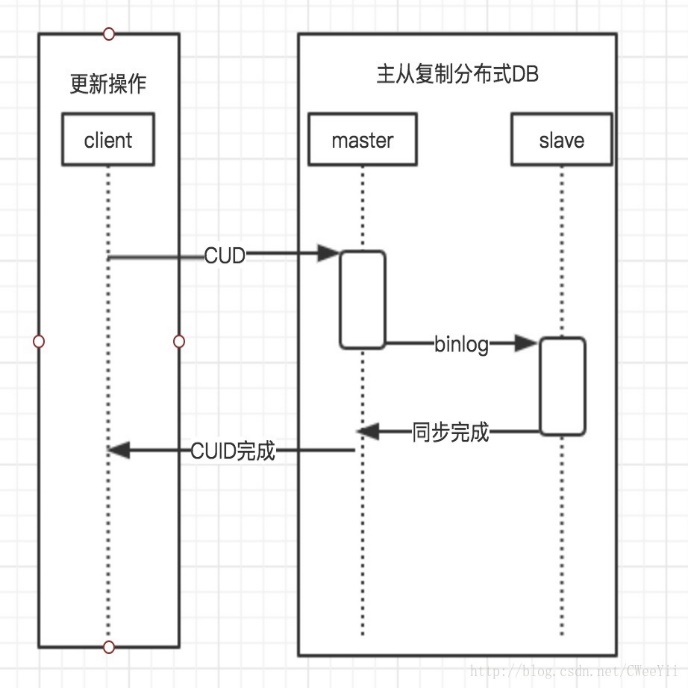
#### 分类

**1 单机数据可（CA）类型**

由于数据库的是以事务为单位进行执行的而事务具有ACID的特性，因此能够保证数据时刻都是同步的。并且只要整个网络不瘫痪（及单机数据库不宕机）都能保证服务是可用的。因此单机数据库系统是具有CA特性的。   
在数据库中主要是通过记录log日志（redo日志和undo日志）来保证数据的一致性。通过乐观锁（快照读，根据事务的不同执行时间，提供不同版本的快照数据，达到读不加锁，提高服务的可用性），对于需要加锁悲观锁的操作（update delete alter select for update)通过当前读及对要处理的记录加锁(gap锁)来使得整个操作串行化的目的。

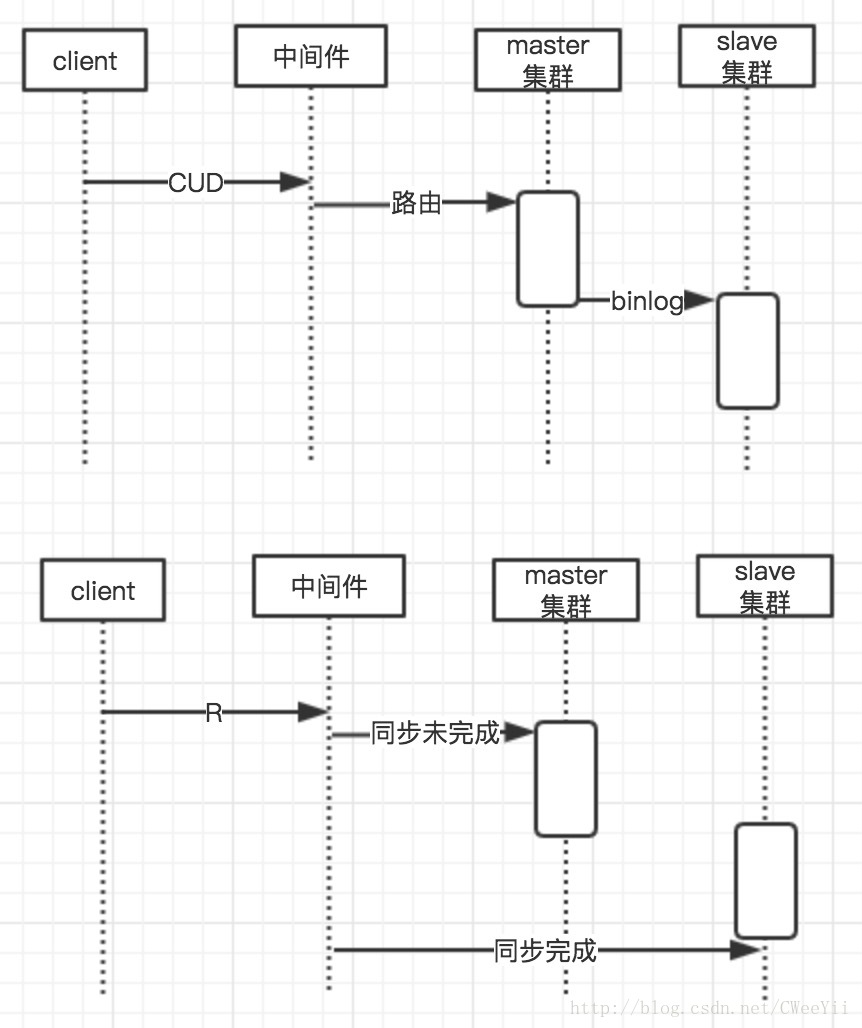
**2 CP类型系统（分布式数据库）**

**分布式数据库（半同步复制)**：主从同步延迟引起的数据不一致，可以通过消除主从延迟或者对有延迟的数据读主库来达到数据的强一致性。   
半同步复制：在客户端请求时，等从库同步完成后，再返回。



方案优点：利用数据库原生功能，比较简单   
方案缺点：主库的写请求时延会增长，吞吐量会降低。极端情况下，网络分区的出现，造成整个服务不可用。

**分布式数据库（数据库中间件）**：由中间件根据主从同步的完成情况来决定是查主库还是从库



方案优点：在中间件性能不存在问题的情况下，吞吐量比半同步复制高很多倍，略低于主从类型数据的写性能。   
方案缺点：对中间件性能要求很高，所有的请求都需要经过中间件进行路由，容易造成单点故障，当出现分区时所有请求都路由到主库，容易造成主库宕机。并且读写性能都有略微下降。

**分布式数据库（缓存更新记录）**：预估主从延迟的最大时间作为缓存过期时间，对于刚更新的数据，将key缓存到缓存中，当读请求时，先查缓存，命中查主库，未命中查从库。   
方案优点：写性能同直接写主库相比性能下降很少。有较高的吞吐量。   
方案缺点：读写都要先操作缓存数据，会对性能有所影响。过期时间难以保证，可能出现不是强一致性的情况。   
上述类型的数据库方案都是牺牲了可用性来保证数据的强一致性。

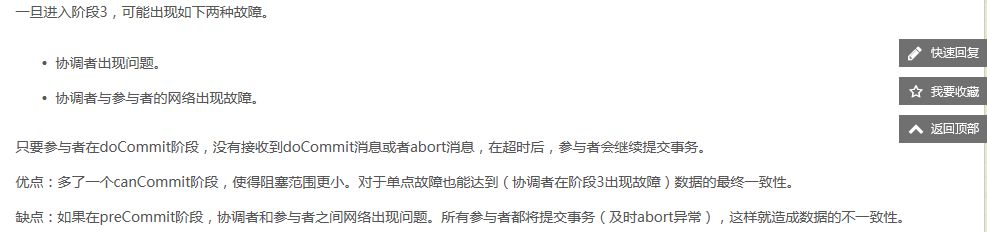
**分段提交协议（zookeeper)**

zookeeper通过实现分段提交协议（zab协议）来使得其具有强一致性和分区容错性的特点，其大量用于解决分布式中间件在遇到脑裂时的主-备选举和数据恢复的功能。

在分布式系统中，每个单机系统都可以通过本地事务来准确的知道自己处理的结果是成功或者失败。因此对于分布式系统我们可以提供一个协调者来统一调度分布式节点（参与者）的事务，并根据分布式节点（参与者）的处理结果来决定各个分布式节点的事务是否真正的提交，根据这种思路提出了两阶段提交和三阶段提交的分布式事务协议。







**3 AP类型的分布式系统**

AP类型是最常见的分布式系统，其只需要在一定时间内能够保证数据的最终一致性。大部分的中间件的实现都保证了AP特性，如redis、主从数据库等。

一主多从-读写分离（atlas）分布式数据库   
在一主多从、读写分离的分布式数据库中，主从的同步是通过binlog同步来实现，而binlog的dump过程是单线程的，并且大部分的同步方式都是行级同步，因此当存在大量的写操作时，主库提交的事务，在从库生效会一定的延迟，但是其保证一定会在从库生效。这样的主从数据库就保证了服务的可用性（读从，主从故障切换）的基础上，同时满足了最终一致性的要求，并且其会根据主库中事务提交的顺序（单线程的）在从库生效，其也满足了顺序一致性。

### BASE理论

BASE是Basically Available（基本可用）、Soft state（软状态）和Eventually consistent（最终一致性）三个短语的缩写。BASE理论是对CAP中一致性和可用性权衡的结果，其来源于对大规模互联网系统分布式实践的总结，是基于CAP定理逐步演化而来的。BASE理论的核心思想是：**即使无法做到强一致性，但每个应用都可以根据自身业务特点，采用适当的方式来使系统达到最终一致性**。接下来看一下BASE中的三要素：

**1、基本可用**

基本可用是指分布式系统在出现不可预知故障的时候，允许损失部分可用性----注意，这绝不等价于系统不可用。比如：

（1）响应时间上的损失。正常情况下，一个在线搜索引擎需要在0.5秒之内返回给用户相应的查询结果，但由于出现故障，查询结果的响应时间增加了1~2秒

（2）系统功能上的损失：正常情况下，在一个电子商务网站上进行购物的时候，消费者几乎能够顺利完成每一笔订单，但是在一些节日大促购物高峰的时候，由于消费者的购物行为激增，为了保护购物系统的稳定性，部分消费者可能会被引导到一个降级页面

**2、软状态**

软状态指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时。（状态可以有一段时间不同步）

**3、最终一致性**

最终一致性强调的是所有的数据副本，在经过一段时间的同步之后，最终都能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。

总的来说，BASE理论面向的是大型高可用可扩展的分布式系统，和传统的事务ACID特性是相反的，**它完全不同于ACID的强一致性模型，而是通过牺牲强一致性来获得可用性，并允许数据在一段时间内是不一致的，但最终达到一致状态**。但同时，在实际的分布式场景中，不同业务单元和组件对数据一致性的要求是不同的，因此在具体的分布式系统架构设计过程中，ACID特性和BASE理论往往又会结合在一起。

BASE思想的主要实现：1 按功能划分的数据库 2 sharding碎片

## Redis减少内存占用

### 短结构

Redis为列表 集合 散列和有序集合提供了一组配置选项，这些选项可以让redis以更节约内存的方式存储较短的结构。因此要保证压缩列表的长度不能太大，否则会影响性能。

#### Ziplist（压缩列表）——列表 有序集合 散列

通常情况下的存储方式

列表——双端链表或者ziplist

散列——hashtable 或者 ziplist

集合——hashtable 或者 整数集合

字符串对象——long或者embstr或者raw动态字符串

有序集合——ziplist或者跳跃表和字典

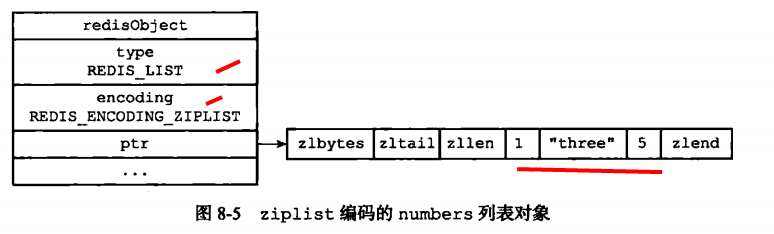
（有序集合使用跳跃表，这样在查询某个值时的时间复杂度为O(logn)，最差为O(n)。同时，每个节点都会有一个跨度，这样在查询某个值时，可以获得这个值的排位是多少。

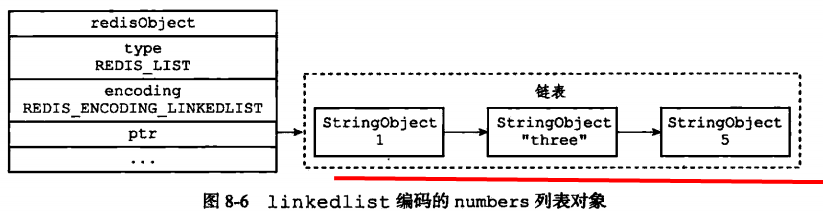
当列表 散列 有序集合的长度较短或者体积叫小时，redis会采用ziplist的紧凑存储方式来存储这些结构。

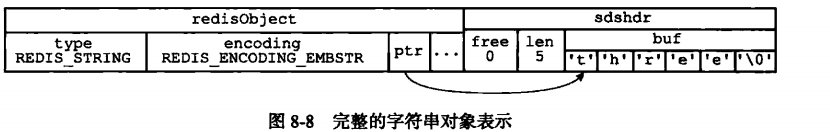
Ziplist是列表 散列 有序集合三种不同类型的对象的一种非结构化表示，以序列化方式存储数据，这些序列化数据每次被读取的时候都需要进行解码，每次写入的时候也要进行编码。

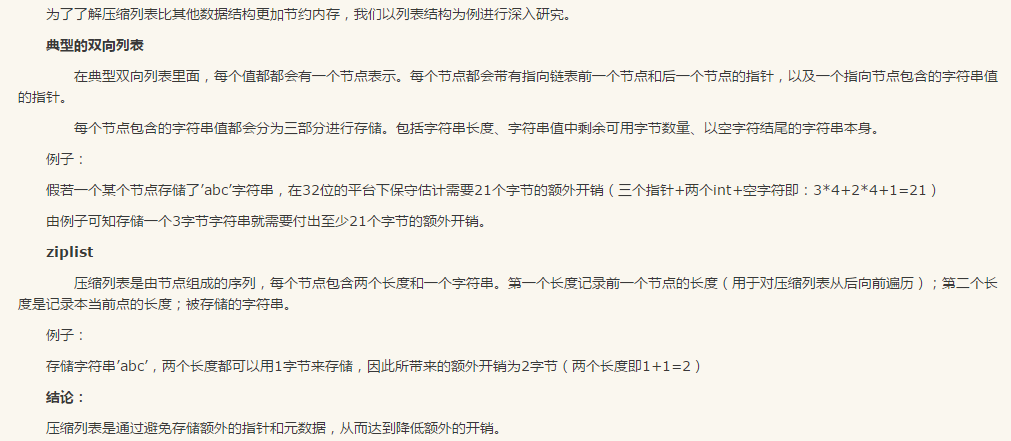
双向链表和压缩列表区别：

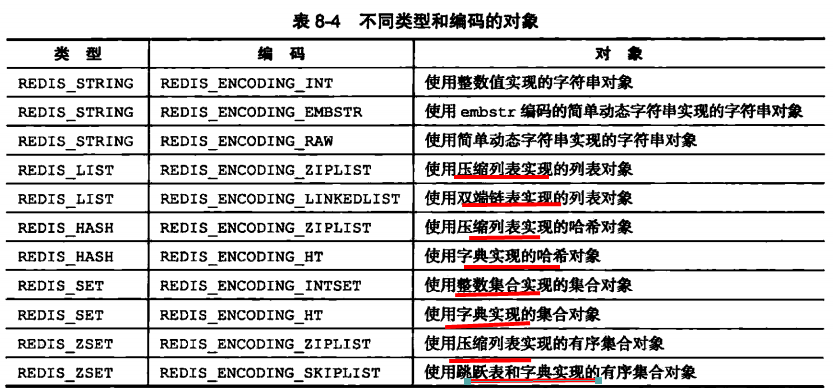
（结构不同，对于压缩列表，每个压缩列表节点保存了一个列表的元素，而linkedlist编码的对象使用双端链表实现，每个双端链表节点都保存了一个字符串对象，每个字符串对象都保存了一个列表元素）





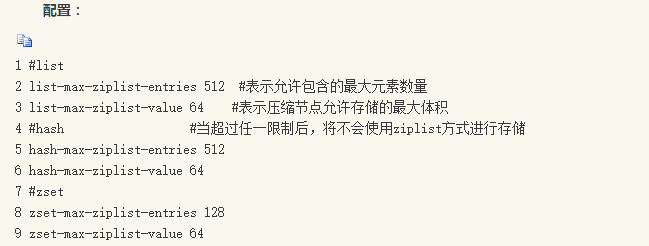






在列表对象包含的元素比较少时，redis使用压缩列表作为列表对象的底层实现。因为压缩列表比双端链表更节约内存，并且在元素数量较少时，在内存中以连续块方式保存的压缩列表比起双端链表可以更快地被载入到缓存中。随着列表中对象包含的元素越来越多，使用压缩列表来保存元素的优势逐渐消失，对象就会将底层的实现从压缩列表转向功能更强 也更适合保存大量元素的双端链表上面。（因为使用压缩列表时，如果要涉及到修改添加等操作时，需要将压缩列表进行解码，操作之后，再次进行编码）

列表对象保存的所有字符串元素的长度都小于64字节时，且列表对象保存的元素数量小于512个时，使用ziplist编码，否则使用linkedlist编码。



有序集合为什么同时使用跳跃表和字典来实现？

如果只采用字典来实现，则可以以O(1)时间复杂度查找成员的分值（键存储成员，值存储分值）。但是字典是以无序的方式来保存集合元素，所以在进行范围操作如zrank zrange时，程序需要对所有元素进行排序，需要至少O(nlogn)时间复杂度，以及额外的O(n)内存空间。（需要创建一个数组来保存排序后的元素）。

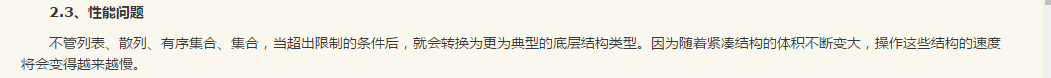
如果只使用跳跃表来实现有序集合，那么跳跃表执行范围操作的优点会保留，但是没有字典，那么根据成员查找分值的操作会变为O(logn).

#### 整数集合

前提：集合中包含的所有member都可以被解析为十进制整数。

以有序数组的方式存储集合不仅可以降低内存消耗，还可以提升集合操作的执行速度。





### 分片结构

Redis的分片（Sharding或者Partitioning）技术是指将数据分散到多个Redis实例中的方法，分片之后，每个redis拥有一部分原数据集的子集。在数据量非常大时，这种技术能够将数据量分散到若干主机的redis实例上，进而减轻单台redis实例的压力。分片技术能够以更易扩展的方式使用多台计算机的存储能力（这里主要指内存的存储能力）和计算能力：

（1）从存储能力的角度，分片技术通过使用多台计算机的内存来承担更大量的数据，如果没有分片技术，那么redis的存储能力将受限于单台主机的内存大小。

（2） 从计算能力的角度，分片技术通过将计算任务分散到多核或者多台主机中，能够充分利用多核、多台主机的计算能力。

**采用分片算法**

采用分片之后，数据将被分散到4个redis实例中，对数据的操作也被分散到每个redis实例中，此时单台主机的压力将大大减轻。

分片的部署，即实例2中分片算法被放在哪里？是在分片时需要首先考虑的问题，分片部署方式一般分为以下三种：  
（1）在客户端做分片；这种方式在客户端确定要连接的redis实例，然后直接访问相应的redis实例；  
（2）在代理中做分片；这种方式中，客户端并不直接访问redis实例，它也不知道自己要访问的具体是哪个redis实例，而是由代理转发请求和结果；其工作过程为：客户端先将请求发送给代理，代理通过分片算法确定要访问的是哪个redis实例，然后将请求发送给相应的redis实例，redis实例将结果返回给代理，代理最后将结果返回给客户端。  
（3）在redis服务器端做分片；这种方式被称为“查询路由”，在这种方式中客户端随机选择一个redis实例发送请求，如果所请求的内容不再当前redis实例中它会负责将请求转交给正确的redis实例，也有的实现中，redis实例不会转发请求，而是将正确redis的信息发给客户端，由客户端再去向正确的redis实例发送请求。

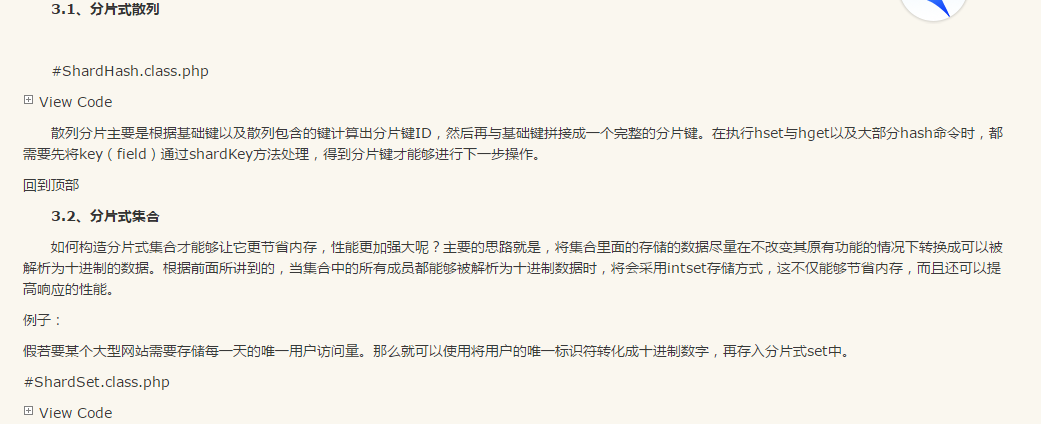
上面主要描述了分片的优点，当然分片的存在也有缺陷，例如：  
（1） 通常无法支持涉及多键的操作；在redis中有很多一次操作多个key的操作，例如求集合交集的SINTER操作，该操作将涉及到多个键，而这多个键有可能被分片到不同的redis实例中，此时就无法执行这种操作。  
（2）Redis的事务操作中涉及多个键时也不能用；  
（3）分片将导致数据处理更加复杂；例如在分片过程中，随着redis实例的增加，数据备份等操作都将会变得更加复杂。  
（4）Redis目前不支持动态分片操作，扩容和缩容操作都会比较复杂，尤其分片操作部署在客户端时，需要重新配置和启动客户端。在使用过程中缩容用的不多，扩容可以采用后面介绍的预分片策略来缓解此问题。

**预分片技术**

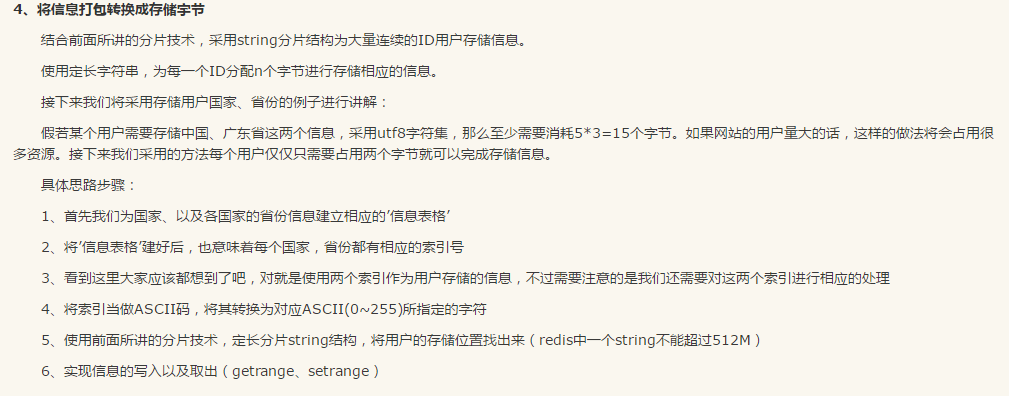
因为使用了一致性哈稀进行分片，那么不同的key分布到不同的Redis-Server上，当我们需要扩容时，需要增加机器到分片列表中，这时候会使得同样的key算出来落到跟原来不同的机器上，这样如果要取某一个值，会出现取不到的情况。而且在正常运营环境中，一般所存储的数据会逐渐增加，可能今天只要10个redis实例就能应付，但是到了一年以后就需要50个redis实例才能支撑，因此，redis的扩容是经常用到的功能，在redis的分布式部署中，有预分片技术是非常好用的方法之一；

预分片技术是指在开始时就启动足够多的redis实例（例如32或64个，估计一下够以后扩展用就行了），等到后续需要扩容的时候，只需要将其中一部分的redis实例转移到新增加的机子上即可，在redis实例迁移过程中使用redis的复制功能可以最大限度的降低redis的停工时间甚至可以做到没有停工时间。由于redis实例是轻量级的进程，而且占用内存较少，这里指单纯的空的redis实例，一个空的redis实例大约占用1M的内存；因此，这种方式即不会占用太多系统开销，又便于实现；  
  
Redis的预分片技术可以按照以下步骤进行实例迁移操作：  
（1）在新机子上启动新的redis实例；  
（2）将新redis实例作为slave将原redis实例作为master，将数据从原redis实例迁移到新redis实例上；  
（3）停止客户端（分片操作在客户端上时）或代理服务器（分片操作在代理上）  
（4）更新客户端或者代理服务器中的配置信息，去掉被迁移的原redis实例的ip和端口等信息，加上新启动redis实例的IP地址和端口；  
（5）向新启动的redis发送SLAVEOF NOONE命令，终止新redis实例对原redis实例的从属关系；  
（6）重启客户端程序或者代理程序，此时它们将会使用新的redis实例；

（7）关掉被迁移走数据的原redis实例；



### 打包存储二进制位和字节



## Redis---数据库

Redis默认支持16个数据库，编号为0-15.

Redis数据库的作用只是提供一个简单的命名空间，以便将不同用途的数据分开，在同一个服务器上进行分库不会带来任何效率上的提升。

如果使用多库，那么每次在读写操作时，都要使用select命令选择具体的库。可以将不同的业务存储在不同的数据库，比如用户和产品存放在两个数据库中。

Redis不支持自定义数据库的名字，每个数据库以编号命名，redis也不支持为不同的数据库设置不同的访问密码，所以一个客户端要么可以访问全部数据库，要么连一个数据库也没有访问权限。多个数据库之间并不是完全隔离的，通过flushall可以清空一个redis实例中所有数据库中的数据。不同的数据库不适宜存储不同应用程序的数据，可以使用0号数据库存储生产环境中的数据，使用1号数据库存储测试环境中的数据。不同的应用应该使用不同的redis实例去存储。 每个数据库都有属于自己的空间，在不同的数据库中可以有相同的key，而不用担心发生冲突。Flushdb只会清除当前数据库中的数据。

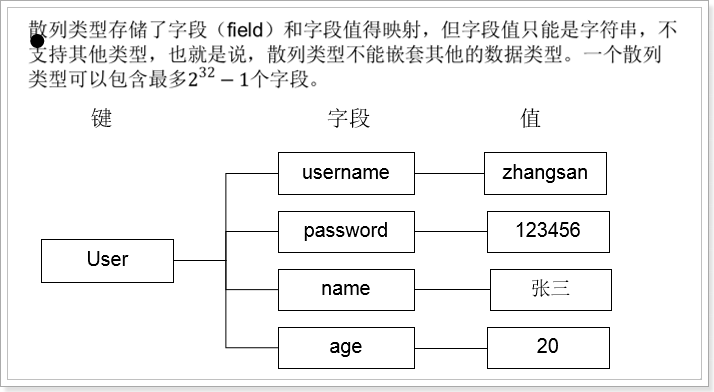
建议使用默认的db0数据库，不同的业务数据采用不同key区分，如User:XX Product:XX

建议采用多redis实例方法，一个新的空闲实例占用1w内存，开销很小。而且可以避免同时持久化写入操作。

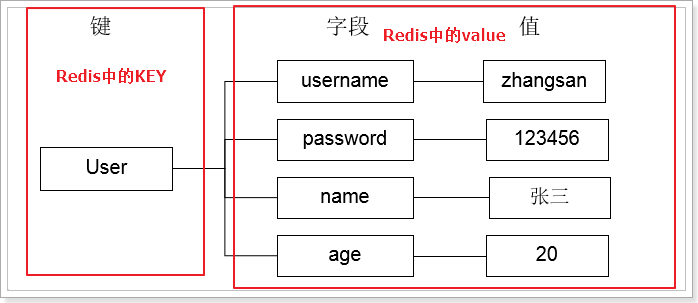
## Redis中的数据结构

### Hash详解

#### 基本结构



如果要更新age，只需要把age拿出来即可，不需要把其他的也取出。



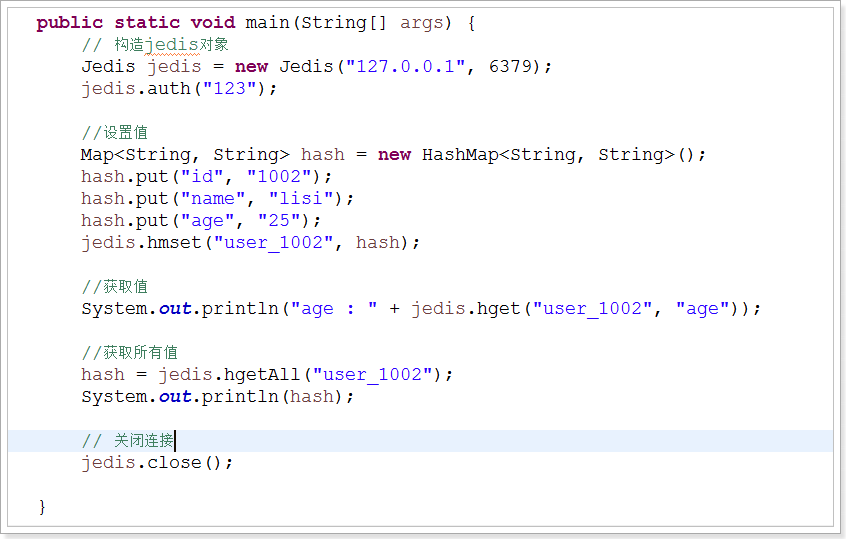
Key—field---value

Key对应多个field，一个field对应一个value

#### 基本命令



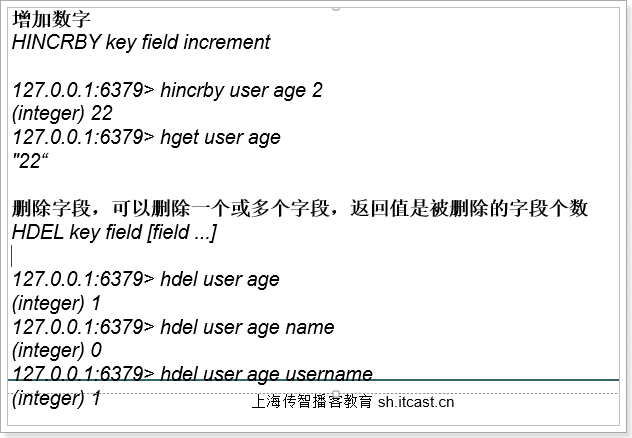
#### jedis操作hash结构



#### 判断字段是否存在



#### 增长和删除



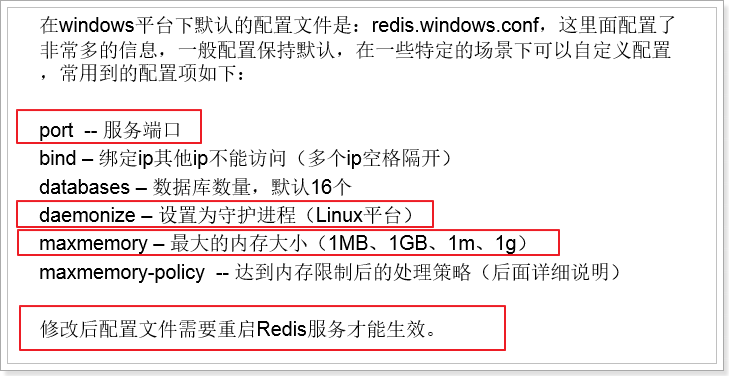
#### 获取字段名或字段值

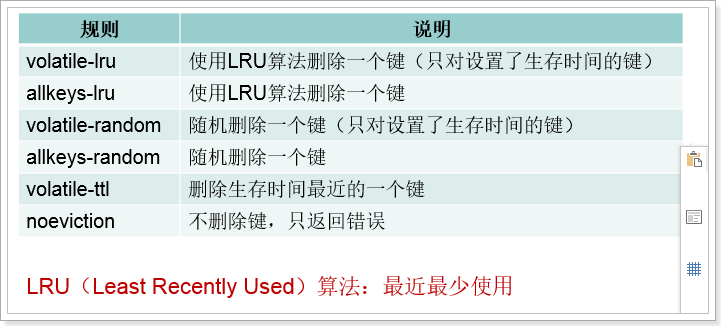


#### 获取字段数量



## redis配置文件





## Redis过期键删除策略

有三种删除策略：

**定时删除**：在设置键的过期时间同时，创建一个定时器，让定时器在键的过期时间来临时，立即执行对键的删除操作。

保证过期的键会尽可能快地被删除，并释放过期键占用的内存。

缺点是，在过期键比较多时，删除过期键这一行为可能会占用一部分cpu时间，在内存不紧张但cpu紧张情况下，会对服务器的响应时间和吞吐量造成影响。

惰性删除：每次从键空间获取值时，检查取得键是否过期，如果过期则删除该键，否则返回。

优点是不会在删除其他无关的过期键上花费任何CPU时间。缺点是，如果一个键已经过期，而这个键仍然保留在数据库中，那么只要这个过期键不被删除，它所占用的内存就不会释放。

如果数据库中存在非常多的过期的键，而这些过期的键又恰好没有被访问到的话，那么它们永远也不会被删除。这个可以看成内存泄露——无用的垃圾占用了大量内存。

定期删除：每隔一段时间，程序就对数据库进行一次检查，删除里面的过期键。至于要删除多少过期键，以及要检查多少个数据库，则由算法决定。定时删除会占用太多的cpu时间，影响服务器的响应时间和吞吐量。惰性删除浪费太多内存，有内存泄露危险。

定期删除策略的难点是确定删除操作执行的时长和频率。如果删除操作执行的很频繁，或者执行时间很长，定期删除就会退化为定时删除，以至于将cpu时间过多的消耗在删除过期键上。如果删除操作执行的太少，或者执行的时间太短，定期删除就会和惰性删除一样，出现浪费内存的情况。

因此，服务器需要合理的设置删除操作的执行时长和执行频率。

Redis中的过期删除策略：使用惰性删除和定期删除两种策略。通过配合这两种策略，服务器可以很好的在合理使用cpu时间和避免浪费内存空间之间取得平衡。

惰性删除策略的实现：

所有读写数据库的redis命令在执行之前都会对输入键进行检查：如果键已经过期，那么将会把输入键从数据库中删除，否则不做任何操作。

定期删除策略的实现：redis会周期性调用某个函数，该函数会在规定时间内，多次遍历服务器中的各个数据库，从数据库的expires字典中随机检查一部分键的过期时间，并删除其中的过期键。