# Redis单线程模型

原文：https://blog.csdn.net/y277an/article/details/98342442

## 概述

Redis基于Reactor模式开发了网络事件处理器，又叫文件事件处理器（file event handler）。

它的组成结构为4个部分：多个socket、IO多路复用程序、文件事件分派器、事件处理器（包括连接应答处理器、命令请求处理器、命令回复处理器）。

多个socket可能会并发产生不同的操作，每个操作对应不同的文件事件。但是IO多路复用程序同时监听多个socket，然后将产生事件的socket压入内存队列中。事件分派器每次从队列中取出一个socket，根据socket 的事件类型交给对应的事件处理器进行处理。因为文件事件分派器队列的消费是单线程的，所以Redis才叫单线程模型。

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

多个socket可能并发产生不同的操作，每个操作对应不同的文件事件，但是IO多路复用会监听多个socket，并将产生事件的socket放入队列中排队，每次从队列中取出一个socket给事件分派器，事件分派器把socket给对应的事件处理器。

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

要明白，进程通信是通过socket来完成的，不懂的同学可以先看看socket网络编程

**文件事件：**

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

当socket变可写时（客户端对redis执行read操作），socket会产生一个AE\_WRITEABLE事件。

IO多路复用程序可以同时监听AE\_READABLE和AE\_WRITEABLE两种事件，要是一个socket同时产生了这两种事件，文件事件分派器优先处理AE\_READABLE事件，然后才是AE\_WRITEABLE事件。

**文件事件分派器：**

如果是客户端要连接redis，那么会为socket关联连接应答处理器。

如果是客户端要写数据到redis，那么会为socket关联命令请求处理器。

如果是客户端要从redis读数据，那么会为socket关联命令回复处理器。

## 通信过程

**1）建立socket连接**

客户端socket01向Redis的ServerSocket请求建立连接，此时ServerSocket会产生一个AE\_READABLE事件，IO多路复用程序监听到ServerSocket产生的事件后，将该事件压入队列中(图中队列右边的ss)。文件事件分派器从队列中获取该事件，并交给连接应答处理器。连接应答处理器会创建一个能与客户端通信的socket01，并将该socket01的AE\_READABLE事件与命令请求处理器关联。

**2）发送set key value请求**

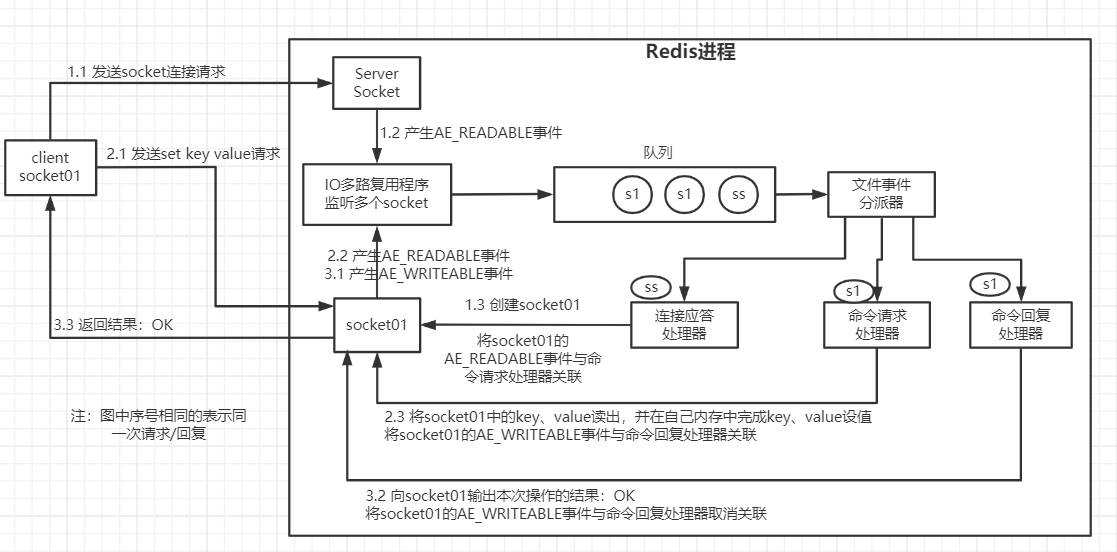
假设此时客户端发送set key value请求，此时Redis中的socket01会产生AE\_READABLE事件，IO多路复用程序将事件压入队列中(图中队列中间的s1)。文件事件分派器从队列中获取到该事件，由于前面socket01的AE\_READABLE 事件已经与命令请求处理器关联，因此事件分派器将事件交给命令请求处理器来处理。命令请求处理器读取socket01的 key value 并在自己内存中完成 key value 的设置。操作完成后，它会将 socket01 的 AE\_WRITABLE 事件与命令回复处理器关联。

**3）准备接收回复**

如果此时客户端准备好接收返回结果了，那么Redis中的 socket01 会产生一个 AE\_WRITABLE 事件，同样压入队列中(图中队列左边第一个s1)，事件分派器找到相关联的命令回复处理器，由命令回复处理器对 socket01 输入本次操作的一个结果，比如 ok，之后解除 socket01 的 AE\_WRITABLE 事件与命令回复处理器的关联。

**这样便完成了一次通信！**

客户端与Redis的一次通信过程：



# redis QPS性能测试

单节点TPS达到8万/秒，QPS达到10万/秒。

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。

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自带性能测试工具：redis-benchmark，可用它计算QPS。

格式：> redis-benchmark [option] [option value]

实例：100个客户端连接，发出10000个请求

> ./redis-benchmark -h 127.0.0.1 -p 6379 -c 100 -n 10000

实例：100个客户端连接，发出10000个请求，每个请求数据为100个字节的value

> ./redis-benchmark -h 127.0.0.1 -p 6379 -c 100 -n 10000 -d 100

**示例：测试服务端性能，50个并发连接，发出100000个请求，每个请求数据为2KB**

# 查看redis服务

[root@localhost bin]# pwd

/usr/local/bin

# 查看redis进程

[root@localhost bin]# ps -ef | grep redis

root 1051 1 0 12:24 ? 00:00:33 /usr/local/bin/redis-server 127.0.0.1:6379

root 1790 1489 0 14:06 pts/0 00:00:00 grep --color=auto redis

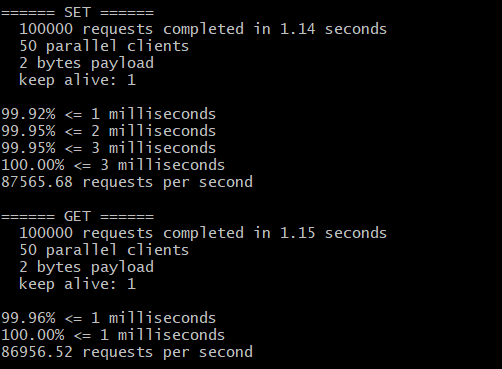
# 对该redis服务进行性能测试

[root@localhost bin]# ./redis-benchmark -h 127.0.0.1 -p 6379 -c 50 -n 100000 -d 2

说明：

会对ping、set、get、incr、lpush、rpush、lpop、rpop、sadd、hset、lrange分别进行性能测试。

我们关注最后一行：每秒87565.68个请求，即有8.7万QPS；这里的数据都只是测试数据，测出来的QPS不能代表实际生产的处理能力。



# 生产环境的redis部署

问题：需要了解公司的redis生产环境的部署架构，redis的主从架构？集群架构？用了哪种集群方案？有没有做高可用？有没有开启持久化确保数据恢复？线上redis给了几个G的内存？设置了哪些参数？压测后你们redis集群承载多少QPS？

描述：Redis Cluster，10台机器，5台机器部署了redis主节点，另外5台机器部署了redis从节点，每个主节点挂了一个从节点，5个节点对外提供读写服务，每个节点的读写高峰QPS可以达到每秒5万，5台机器最多是每秒25万读写请求。

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

每个主节点都挂了一个从节点，所以是高可用的，任何一个主节点宕机，都会自动故障迁移，redis从节点会自动变成主节点继续提供读写服务。

你往内存里写的是什么数据？每条数据的大小是多少？

商品详细数据，每条数据是10kb。100条数据是1mb，10万条数据是1g。

常驻内存的是200万条商品数据，占用内存是20g，仅仅不到总内存的50%。目前高峰期每秒就是3500左右的请求量。

# Redis的并发竞争问题和解决方案？

描述：多个客户端同时获取一个key，修改值后再写回去，但是顺序错了，数据就错了。如集群环境下多个子系统去set一个key，如某个key1，系统A设置key1=valueA，系统B设置key1=valueB，系统C设置key1=valueC。期望key1的顺序为valuaA->valueB->valueC。

解决1：基于zookeeper实现分布式锁。

每个系统通过zookeeper获取分布式锁，确保同一时间只能由一个系统在操作某个key，其它都不允许读和写。

解决2：若为自增/自减这类的操作，可使用redis自带的incr/desc命令

解决3：利用消息队列，将set方法变成串行化访问

解决4：基于lua实现cas乐观锁方案

1）set key value操作：KEYS[1]为key，ARGV[1]对应要比较的值，值相同则更新成 ARGV[2]，并返回 1，否则返回 0。

if redis.call("get", KEYS[1]) == ARGV[1] then

redis.call("set", KEYS[1], ARGV[2])

return 1

else

return 0

end

2）hset key field value操作：KEYS[1]为key，ARGV[1] 对应 Hash 的 field，ARGV[2]对应要比较的值，值相同则更新成 ARGV[3]，并返回 1，否则返回 0。

if redis.call("hget", KEYS[1], ARGV[1]) == ARGV[2] then

redis.call("hset", KEYS[1], ARGV[1], ARGV[3])

return 1

else

return 0

end

# 高并发场景中读请求长时间阻塞

描述：内存队列中可能会积压针对多个数据项的更新操作，因此需要根据自己的业务情况进行测试，可能需要部署多个服务，每个服务分担一些数据的更新操作。如果一个内存队列里积压了100个商品库存修改操作，每个库存修改操作要耗费10ms去完成，那么队列最后有一个读操作，可能等待10\*100=1000ms=1s后，才能得到数据，这个时候导致读请求的长时间阻塞。

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

其实根据之前的项目经验，一般来说，数据的写频率是很低的，因此实际上正常来说，在队列中积压的更新操作应该是很少的。像这种针对读高并发、读缓存架构的项目，一般来说写请求是非常少的，每秒的 QPS 能到几百就不错了。

我们来实际粗略测算一下。

如果一秒有 500 的写操作，如果分成 5 个时间片，每 200ms 就 100 个写操作，放到 20 个内存队列中，每个内存队列，可能就积压 5 个写操作。每个写操作性能测试后，一般是在 20ms 左右就完成，那么针对每个内存队列的数据的读请求，也就最多 hang 一会儿，200ms 以内肯定能返回了。

经过刚才简单的测算，我们知道，单机支撑的写 QPS 在几百是没问题的，如果写 QPS 扩大了 10 倍，那么就扩容机器，扩容 10 倍的机器，每个机器 20 个队列。

# Redis持久化

## 概述

redis是内存数据库，它把数据存储在内存中，这样加快读取速度的同时页对数据安全性产生新的问题，即当redis所在服务器发生宕机后，redis里的所有数据会丢失。

解决：redis提供了持久化功能-RDB和AOF，即将内存中的数据写入硬盘中。

两种持久化方式：

-1）RDB持久化：会将内存中的数据写入dump.rdb文件。Redis重启时将该文件的数据重新加载进内存。

如：save 900 1 #表示900秒内至少有1次写操作，则触发RDB

-2）AOF持久化：默认关闭，将所有的写操作追加到appendonly.aof文件中。Redis重启时将该文件的写指令从头到尾执行来还原数据集。

如：appendfsync everysec #每秒进行一次文件同步

附：两种都使用时，redis会优先使用AOF来还原数据集。

## RDB持久化

工作原理：满足一定规则时，会将内存中的数据全量生成一份副本存储到硬盘上（写入dump.rdb文件），这个过程称为“快照”。

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；或者可以手工调用save或bgsave命令。

一定规则如下：

在指定时间间隔内发生了指定次数的写操作（save、bgsave）；

执行flushall命令；

执行复制（replication）时。

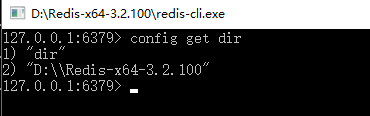
save与bgsave区别：

1）save：阻塞式持久化，redis的主进程把内存里的数据库状态写入到RDB文件中，在该文件创建结束之前redis不能处理任何命令请求。

2）bgsave：非阻塞式持久化，它fork出一个子进程专门把内存中的数据库状态写入RDB文件里，同时主进程还可以处理来自客户端的命令请求。这个内存使用率会翻倍！

通过rdb文件恢复数据：

若需要恢复数据，只需将备份文件 (dump.rdb) 移动到 redis 安装目录并启动服务即可，如果不知道Redis的安装目录可以在Redis客户端里面使用“config get dir”命令获得。



快照执行的过程：

1）redis使用fork函数复制一份当前进程（父进程）的副本（子进程）；

2）父进程继续处理来自客户端的请求，子进程开始将内存中的数据写入硬盘中的临时文件；

3）当子进程写完所有的数据后，用该临时文件替换旧的rdb文件，至此，一次快照操作完成。

注：rdb文件存储的是fork函数发生那一刻的内存数据。原因：fork函数发生，父进程和子进程共享同一块内存数据，当父进程需要修改其中的某片数据时，操作系统会将该片数据复制一份以保证子进程不受影响，因此fork之后修改的数据没有写入rdb文件。

配置RDB：

修改配置文件（redis/conf/redis.conf）：

--修改配置文件需要重启redis服务器才能生效

save 900 1 #表示900秒内至少有1次写操作，则触发RDB

save 300 10

save 60 10000

dbfilename dump.rdb #指定RDB的文件名称

dir /data/dbs/redis/6381 #指定RDB文件的存放目录

> redis-cli #重启redis服务器

> config get save #查看redis持久化配置

1) "save"

2) "900 1 300 10"

> config set save "2100 20" #修改redis持久化配置

RDB优点：

-1）适合大规模的数据恢复

-2）如果业务对数据完整性和一致性要求不高，rdb是很好的选择

RDB缺点：

-1）rdb并不健壮，当系统停止、或无意中redis被kill掉，最后写入redis的数据就会丢失。

-2）备份时占用内存翻倍。

## AOF持久化

工作原理：将所有的写操作追加到appendonly.aof文件中。Redis重启时将该文件的写指令从头到尾执行来还原数据集。

存储结构：内容是redis通讯协议(RESP )格式的命令文本存储。

触发时机：每当执行服务器（定时）任务、或调用flushAppendOnlyFile函数时执行aof写入保存。

具体步骤：

AOF的持久化是通过命令追加、文件写入和文件同步三个步骤实现。

-1）当redis开启AOF后，服务端每执行一次写操作（如set、sadd、rpush）就会把该条命令追加到一个单独的AOF缓冲区的末尾；

-2）然后调用fsync或fdatasync函数，把AOF缓冲区的数据写入AOF文件里；

-3）最后将文件同步。

配置AOF：

--修改配置文件（redis/conf/redis.conf）（需要重启redis服务器才能生效）：

appendonly yes #开启AOF，默认关闭

appendfsync everysec #AOF持久化策略，每秒进行一次文件同步。

appendfilename "appendonly.aof" # AOF文件名称

dir /data/dbs/redis/6381 # AOF文件存放目录

AOF持久化策略(appendfsync)：

如：appendfsync no

-1）always：每执行一个写操作进行一次文件同步，把内存缓冲区的aof缓冲数据真正写入aof文件里。

-2）everysec（默认）：每秒进行一次文件同步，把内存缓冲区的aof缓冲数据真正写入aof文件里。

-3）no：不同步，即当内存缓冲区的空间被填满或超过了设定的时限后系统自动执行文件同步。

配置重写触发机制：

> auto-aof-rewrite-percentage 100 #当前写入日志文件的大小占到初始日志文件大小的某个百分比时触发Rewrite

> auto-aof-rewrite-min-size 64mb #本次Rewrite最小的写入数据量

重写原理：aof机制是将写操作追加到aof文件中，这样会使得文件原来越大。因此redis在实现重写时，会fork出一条新进程读取内存中的数据，并重新写到一个临时文件中。并没有读取旧文件（内容过大）。最后替换旧的aof文件。

附：当aof文件的大小是上次rewrite后大小的2倍，且文件大于64M时触发。

通过aof文件恢复数据：

将appendonly.aof文件拷贝到redis中的bin目录下，重启redis服务器即可。

AOF机制中为什么会有同步操作？

答：为了提高文件的写效率，都会有一个写入策略，即当你往硬盘写入数据时，操作系统不是实时的将数据写入硬盘，而是先把数据暂时保存在一个内存缓冲区里，等到这个内存缓冲区被填满或超过了设定时限后才会真正地把缓冲区内的数据写入硬盘中。即当redis进行第二步文件写入时，从用户的角度看是已经把AOF缓冲区里的数据写入到AOF文件了，但对系统而言只不过是把AOF缓冲区的内容放到了另一个内存缓冲区里而已，之后redis还需要进行文件同步把该内存缓冲区里的数据真正写入硬盘上才算是完成了一次持久化。

## RDB和AOF比较

RDB和AOF比较：

-1）redis默认开启rdb持久化方式，在指定时间间隔内、发生指定次数的写操作，则将内存中的数据写入到磁盘中。

-2）rdb适合大规模的数据恢复（数据文件较小）。但rdb每次持久化都会内存翻倍。

-3）aof手动开启，默认每秒将写指令追加到aof文件中。

-4）aof的数据完整性比rdb高，但记录的内容过多时，在数据恢复中需要执行写指令，因而影响效率。

# Redis缓存问题

问：redis缓存和本地缓存用途？

## 缓存穿透

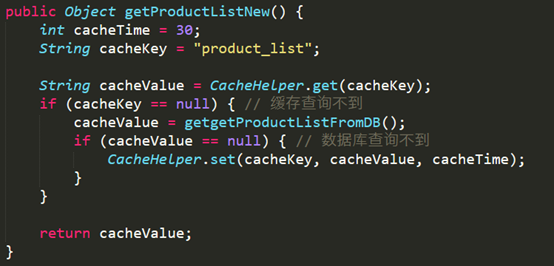
描述：访问不存在的数据，缓存不起作用，导致请求穿透到数据库，若存在大量这样的请求会导致数据库崩溃。

解决方案：

方案1：使用布隆过滤器：使用一个足够大的bitmap，用于存储可能访问的key，不存在的key直。

方案2：访问key未在DB查询有值，也将空值写入缓存，但设置较短的过期时间。这样第二次到缓存中获取就有值了，而不会继续访问数据库。

方案2伪代码：



说明：

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。

## 缓存雪崩

描述：当缓存服务器重启或大量缓存数据集中在某一时间段失效，所有原本应该访问缓存的请求都去访问数据库，造成瞬间DB请求量大、压力剧增，引起雪崩。

解决方案：

方案1：数据库限流（通过加锁或队列来控制操作数据库的线程数量）

方案2：设置过期标记更新缓存

方案3：缓存失效时间增加1~5分钟随机值，使得缓存不在同一时间失效。

另：做二级缓存，A1为原始缓存，A2为拷贝缓存。A1失效时，可以访问A2，A1缓存的失效时间较短，A2缓存的失效时间为长期。

方案1：服务层处理-方法加锁+双重校验

一般并发量不高时，使用最多的解决方案是加锁排队，伪代码如下：



说明：

加锁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，这时候1000个请求999个都在阻塞。

注：在分布式环境下还需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！

方案二：设置过期标记更新缓存

给每一个缓存数据增加相应的缓存标记，记录缓存是否失效，如果缓存标记失效则更新数据缓存。伪代码如下：



说明：

-1）缓存标记：记录缓存数据是否过期，若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（如标记缓存时间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。

## 缓存预热

描述：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。这样可避免在用户请求时，先查询数据库后，再将数据缓存的问题！

即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，上线时手动操作；

-2）数据量不大，可以在项目启动时自动进行加载；

-3）定时刷新缓存。

# Redis事务

## Redis事务特征

redis事务可以一次执行多个命令，并且带有以下三个特征：

1）批量操作在发送exec命令前被放入队列缓存；

2）收到exec命令后进入事务执行。若事务中任意命令执行失败，其余的命令依然被执行；

3）在事务执行过程，其它客户端提交的命令请求不会插入到事务执行命令序列中。

## Redis命令

一个事务从开始执行会经历三个阶段：开始事务、命令入队、执行事务。

与数据库事务区别：redis事务出错后，剩下的命令会继续执行，出错的数据不会回滚。

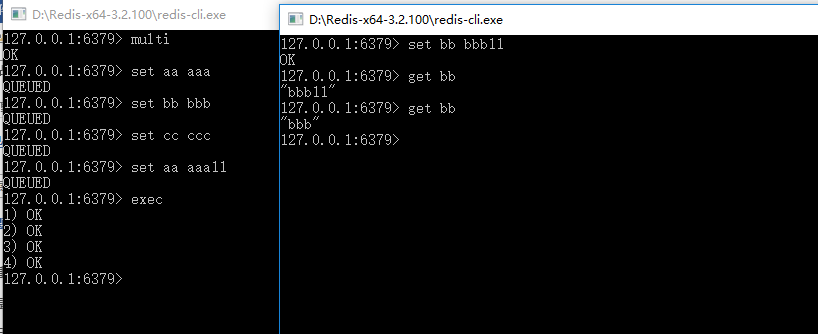
开启事务用multi命令，通过exec/discard命令来提交/回滚该事务内的所有操作。

当使用Append-Only模式时，Redis会通过调用系统函数write将该事务内的所有写操作在本次调用中全部写入磁盘。然而如果在写入的过程中出现系统崩溃，如电源故障导致的宕机，那么此时也许只有部分数据被写入到磁盘，而另外一部分数据却已经丢失。Redis服务器会在重新启动时执行一系列必要的一致性检测，一旦发现类似问题，就会立即退出并给出相应的错误提示。此时，我们就要充分利用Redis工具包中提供的redis-check-aof工具，该工具可以帮助我们定位到数据不一致的错误，并将已经写入的部分数据进行回滚。修复之后我们就可以再次重新启动Redis服务器了。

|  |  |
| --- | --- |
| **命令** | **说明** |
| multi | 标记事务的开始，其后执行的命令都将被存入命令队列，直到执行exec时，这些命令才会被原子的执行 |
| exec | 执行事务内命令队列中所有命令，同时将当前连接的状态恢复为正常状态，即非事务状态 |
| discard | 回滚事务队列中的所有命令，同时将当前连接的状态恢复为正常状态，即非事务状态 |
| watch key | 在multi命令执行之前，可以指定待监控的keys，然而再执行exec之前，如果被监控的keys发生修改，exec将放弃执行该事务队列中的所有命令。始终返回OK |
| unwatch | 取消当前事务中指定监控的Keys，如果执行了EXEC或DISCARD命令，则无需再手工执行该命令了，因为在此之后，事务中所有被监控的Keys都将自动取消。始终返回OK |

## 实战

实战1：



实战2：Java代码使用redis事务

1）stringRedisTemplate设置事务：

-1）stringRedisTemplate.setEnableTransactionSupport(true); //默认为false

-2）使用 SessionCallback, 在同一个 Redis Connection 中执行事务: 成功执行事务

2）测试：

@Test

public void testMultiSuccess() {

// 开启事务支持，在同一个 Connection 中执行命令

stringRedisTemplate.setEnableTransactionSupport(true);

stringRedisTemplate.multi();

stringRedisTemplate.opsForValue().set("name", "username");

stringRedisTemplate.opsForValue().set("age", "19");

System.out.println(stringRedisTemplate.exec()); // [true, true]

}

不开启Redis事务时使用stringRedisTemplate的multi()和exec()方法会报错：

\*错误用法：没有开启事务支持，事务执行会失败

抛出异常：Error in execution; nested exception is io.lettuce.core.RedisCommandExecutionException: ERR EXEC without MULTI

原因：enableTransactionSupport值默认为false，导致每一个RedisConnection都是重新获取的。所以执行的multi和exec方法不在同一Connection中

# Redis管道(pipe)

## Redis管道出现原因

redis客户端与redis服务端通信采用cs模式，每次交互都是完整的请求/响应模式。我们使用jedis或lettuce执行redis命令，每次都是建立socket连接，并等待返回。

每个命令底层建立tcp连接的时间是省不掉的，当需要对一组kv进行批量操作时，这组命令的耗时=sum(建立连接时间+发送命令、返回结果)，随着批量操作的key越多，时间累加呈线性增长。这样便出现了像redis连接池的技术，如JedisPool。

当需要对一组kv进行批量操作时，JedisPool池子里的connection连接、极端情况都被用完了，怎么办？答：需要等待JedisPool池里有可复用的connection才能继续执行。若在指定的等待时间内没有等到idle空闲连接，就报异常了。

如果能将多条命令“合并”到一起，进行一次网络IO，性能会提高不少吧。这就是今天的主角——Redis pipeline。

**Redis管道作用：**

一次请求/响应服务器能实现处理新的请求即使旧的请求还未被响应。这样就可以将多个命令发送到服务器，而不用等待回复，最后在一个步骤中读取该答复。

## Redis管道原理

当client使用pipeline发送命令时，redis服务端必须将部分请求放到队列中（使用内存），执行完毕后一次性发送结果。

客户端将多个命令缓存起来，缓冲区满了就发送（将多条命令打包发送）；有点像“请求合并”。服务端接受一组命令集合，切分后逐个执行返回。

## 实战

实战1：快速删除10w个key

实战2：通过pipe插入海量数据

> cat data.txt | redis-cli -h 127.0.0.1 -p 6379 -a 123456 --pipe

## Redis管道与Redis事务区别

pipeline选择客户端缓冲，multi选择服务端队列缓冲；

请求次数的不一致，multi需要每个命令都发送一次给服务端，pipeline所有命令一次性发送给服务端，请求次数相对于multi减少；

multi/exec可以保证原子性，而pipeline不保证原子性。

pipeline管道操作是需要客户端与服务端的支持，客户端将命令写入缓冲，最后再通过exec命令发送给服务端，服务端通过命令拆分，逐个执行返回结果。

# Redis Cluster集群

## 介绍

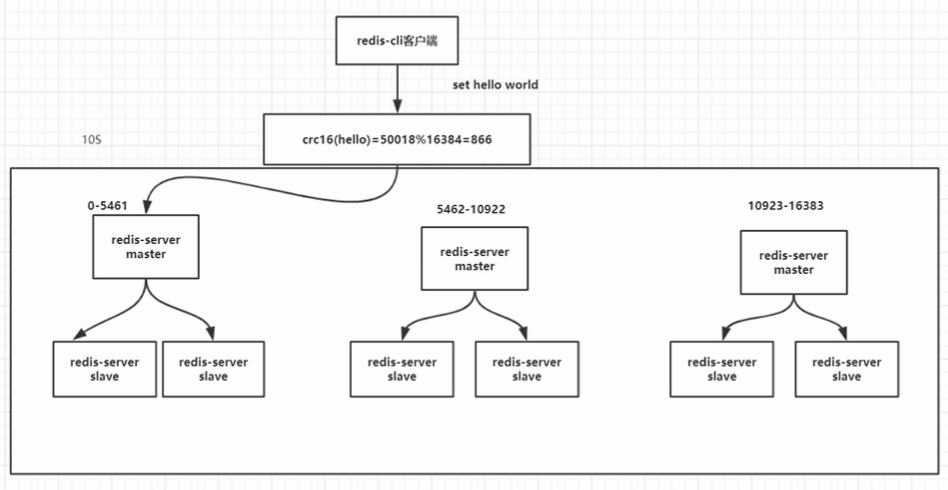
自动将数据进行分片，每个master上放一部分数据。

提供内置的高可用支持，部分master节点不可用时，其它master节点还是可以继续工作的。

redis cluster 功能强大，直接集成了 replication 和sentinel 的功能。

在 redis cluster 架构下，每个 redis 要放开两个端口号，比如一个是 6379，另外一个就是 加 1w 的端口号，比如 16379。16379 端口号是用来进行节点间通信的，也就是 cluster bus 的通信，用来进行故障检测、配置更新、故障转移授权。

cluster bus 用了另外一种。



## 节点间通信-gossip协议

redis cluster 节点间通信采用 二进制的协议-gossip协议，用于节点间进行高效的数据交换，占用更少的网络带宽和处理时间。

redis 维护集群元数据（节点信息、故障等等）采用gossip 协议，所有节点都持有一份元数据，不同的节点若出现了元数据的变更，就不断将元数据发送给其它的节点，让其它节点也进行元数据的变更。

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

gossip的通信端口：每个节点都有一个专门用于节点间通信的端口，就是自己提供服务的端口号+10000，比如7001，那么用于节点间通信的就是 17001 端口。每个节点每隔一段时间都会往另外几个节点发送 ping 消息，同时其它几个节点接收到 ping 之后返回 pong。

交换的信息：信息包括故障信息，节点的增加和删除，hash slot 信息等等。

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

1）meet：某个节点发送 meet 给新加入的节点，让新节点加入集群中，然后新节点就会开始与其它节点进行通信。redis-trib.rb add-node其实内部就是发送了一个 gossip meet 消息给新加入的节点，通知那个节点去加入我们的集群。

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

3）pong：返回ping和meeet，包括自己的状态和其他信息，也用于信息广播和更新。

4）fail：某个节点判断另一个节点fail之后，发送fail给其他节点，通知其他节点说，某个节点宕机啦。

## 哈希槽(hash slot)算法

常见的分布式系统寻址算法大致有三种：

简单hash取模算法（增加节点需要重建）

一致性hash算法（自动缓存迁移）+虚拟节点（自动负载均衡）

Redis cluster的哈希槽（hash slot）算法

Redis cluster有固定的16384个哈希槽（hash slot），对每个key计算CRC16值，然后对16384取模，可以获取key对应的哈希槽。

redis cluster 中每个 master 都会持有部分 slot，比如有 3 个 master，那么可能每个 master 持有5000 多个 hash slot。hash slot 让 node 的增加和移除很简单，增加一个 master，就将其他 master的 hash slot 移动部分过去，减少一个 master，就将它的 hash slot 移动到其他 master 上去。移动hash slot 的成本是非常低的。客户端的 api，可以对指定的数据，让他们走同一个 hash slot，通过 hash tag 来实现。

任何一台机器宕机，另外两个节点，不影响的。因为 key 找的是 hash slot，不是机器。

转移槽位：可以以下两种方式

》/usr/local/bin/redis-cli --cluster reshard 192.168.2.101:7002

》./redis-trib.rb reshard --from bdd63e1f522d78eb1bb2574b2461a7302e14944a  --to 1d204c88a14a76dc30abb05025135f7e850f2a5d  --slots 5461 --yes 192.168.112.36:6004

## 高可用性与主备切换原理

Redis Cluster的高可用原理，几乎跟哨兵是类似的。

**判断节点宕机：**

如果一个节点认为另外一个节点宕机，那么就是 pfail，主观宕机。如果多个节点都认为另外一个节点宕机了，那么就是 fail，客观宕机，跟哨兵的原理几乎一样，sdown，odown。

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

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

**从节点选举：**

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

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

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

从节点执行主备切换，从节点切换为主节点。