Redis

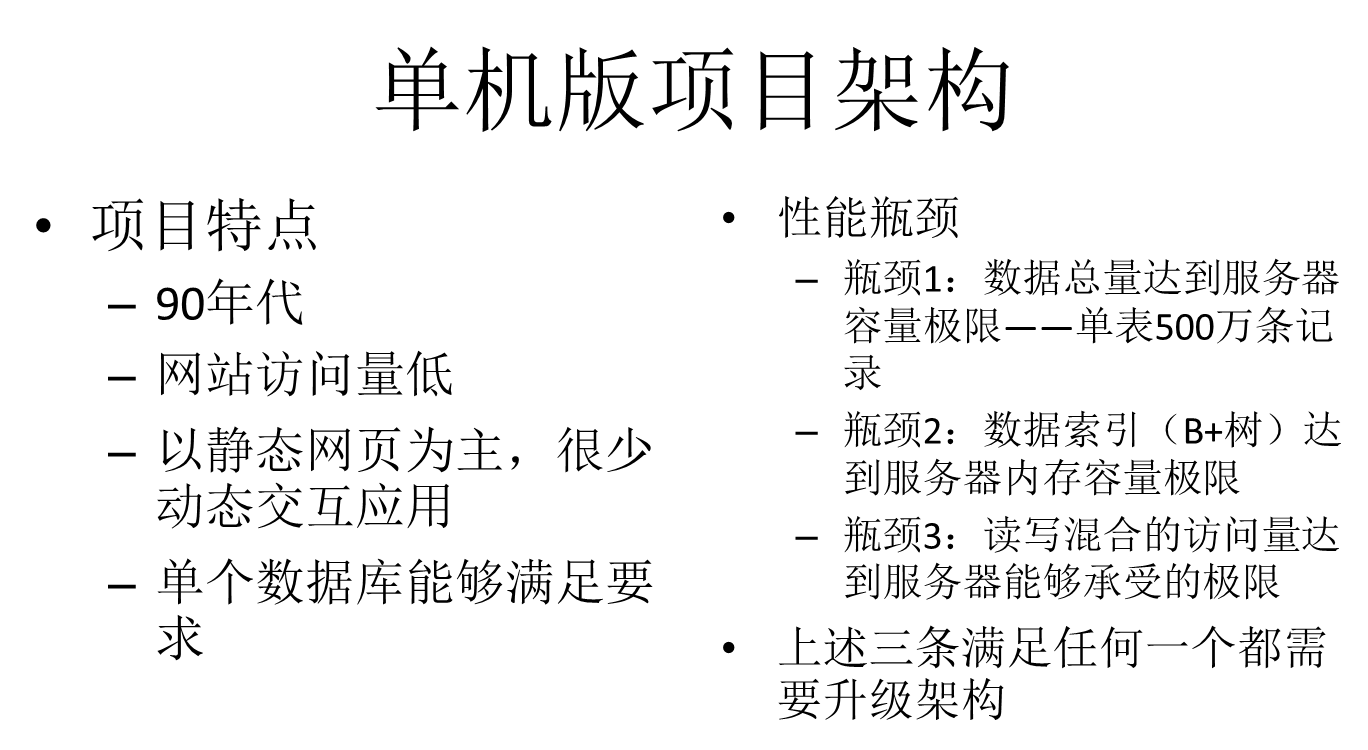
尚硅谷Java

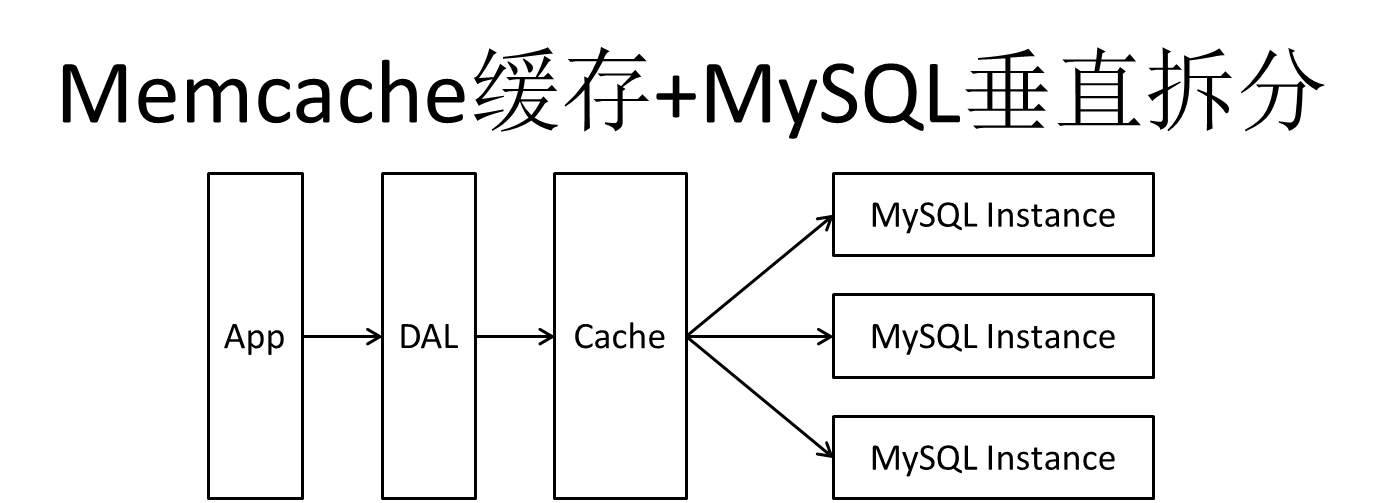
版本 V1.0

# 一、 Redis入门

## 1. 互联网项目架构演变

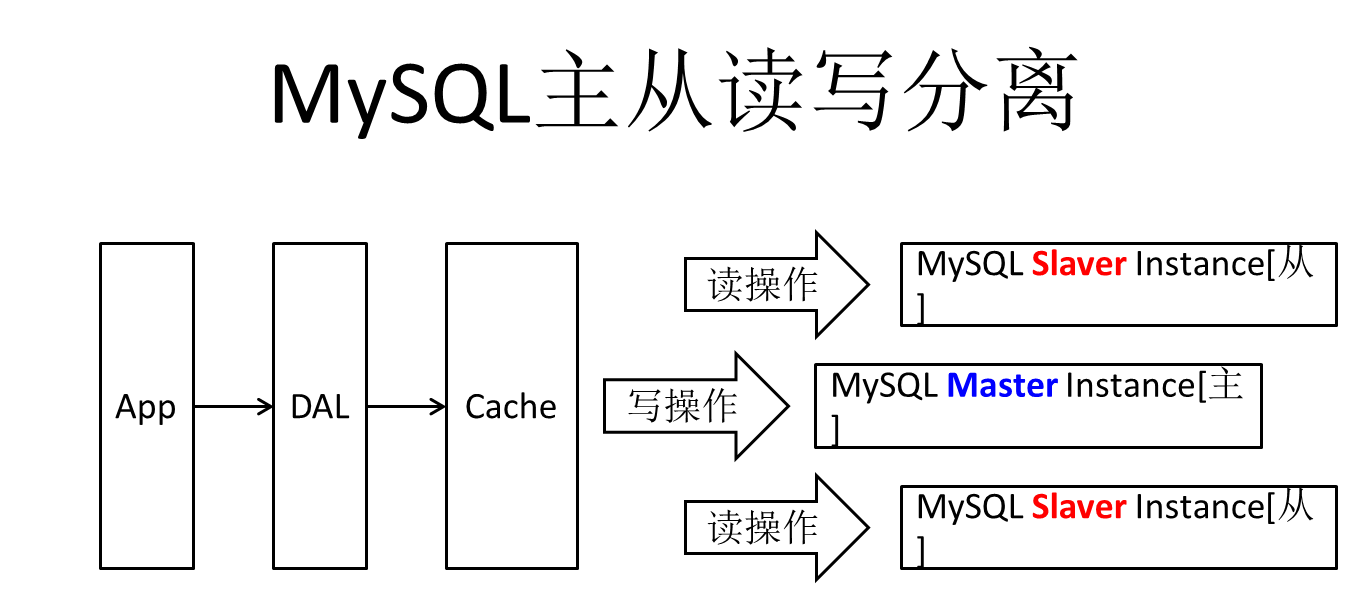




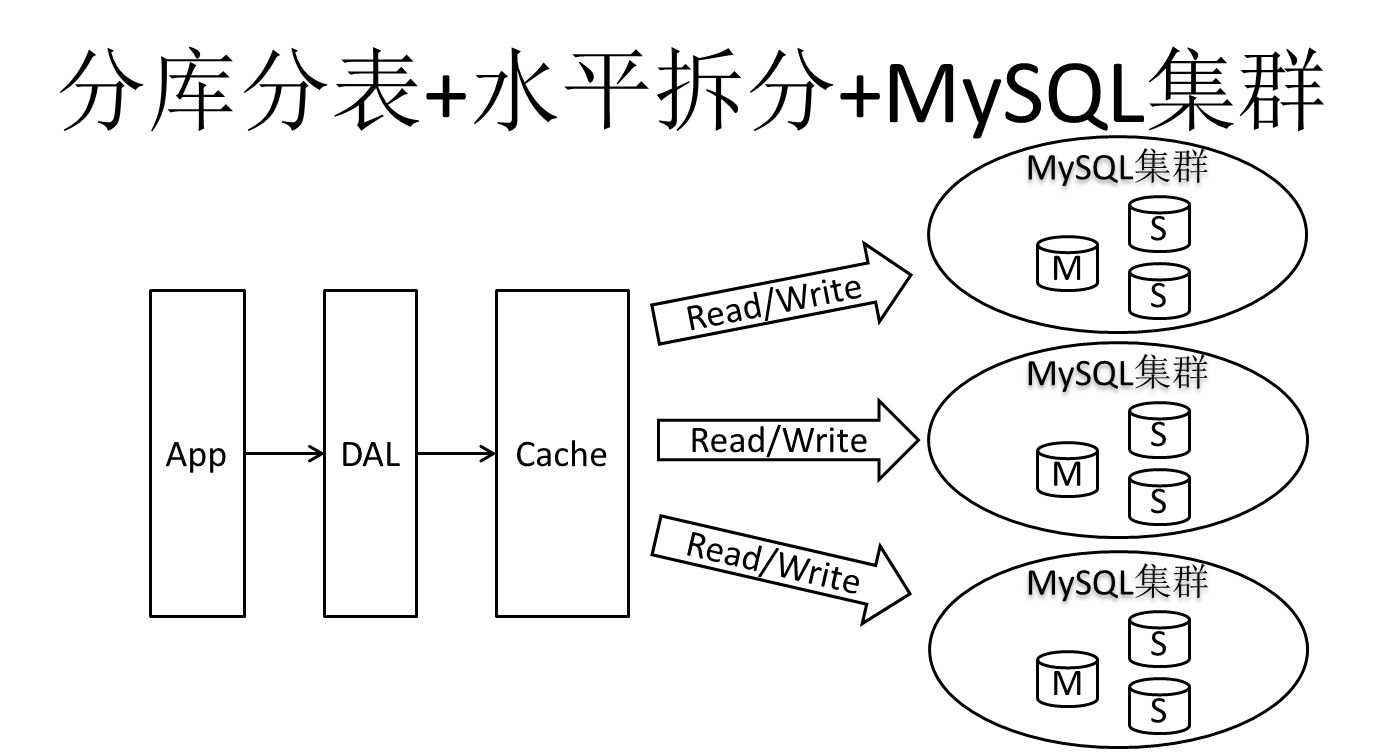


随着访问量上升，大部分使用MySQL架构的网站在数据库上都开始出现性能问题，Web程序不能再仅仅专注在功能上，同时也在追求性能。开始使用缓存技术缓解数据库压力，优化数据库的结构和索引。刚开始时比较流行的是通过文件缓存来缓解数据库压力，但是当访问量继续增大，文件缓存中的数据不能在多台Web服务器之间共享，大量的小文件IO也带来了比较高的IO压力。在这种情况下，Memcache就成了一款非常有效的解决方案。

Memcache作为一个独立的分布式缓存服务器，为多个Web服务器提供了一个共享的高性能缓存服务，在Memcache服务器上，又发展了根据hash算法来进行多台Memcache缓存服务的扩展，然后又出现了一致性hash来解决增加或减少缓存服务器导致重新hash带来的大量缓存失效问题。



由于数据库的写入压力增加，Memcached只能缓解数据库的读取压力。读写集中在一个数据库上让数据库不堪重负，大部分网站开始使用主从复制技术来达到读写分离，以提高读写性能和读库的可扩展性。Mysql的master-slave模式成为这个时候的网站标配了

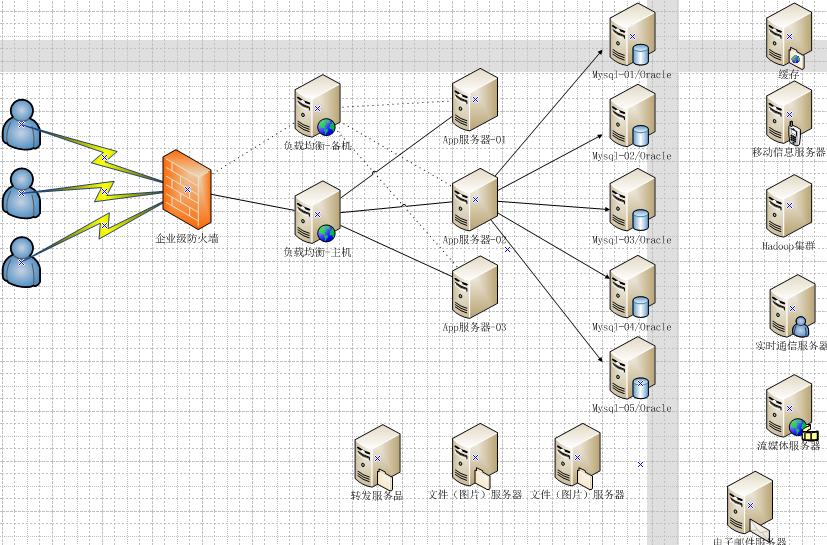


在Memcached的高速缓存，MySQL的主从复制，读写分离的基础之上，这时MySQL主库的写压力开始出现瓶颈，而数据量的持续猛增，由于MyISAM使用表锁，在高并发下会出现严重的锁问题，大量的高并发MySQL应用开始使用InnoDB引擎代替MyISAM。

同时，开始流行使用分表分库来缓解写压力和数据增长的扩展问题。这个时候，分表分库成了一个热门技术，是业界讨论的热门技术问题。也就在这个时候，MySQL推出了还不太稳定的表分区，这也给技术实力一般的公司带来了希望。虽然MySQL推出了MySQL Cluster集群，但性能也不能很好满足互联网的要求，只是在高可靠性上提供了非常大的保证。

MySQL数据库也经常存储一些大文本字段，导致数据库表非常的大，在做数据库恢复的时候就导致非常的慢，不容易快速恢复数据库。比如1000万4KB大小的文本就接近40GB的大小，如果能把这些数据从MySQL省去，MySQL将变得非常的小。关系数据库很强大，但是它并不能很好的应付所有的应用场景。MySQL的扩展性差（需要复杂的技术来实现），大数据下IO压力大，表结构更改困难，正是当前使用MySQL的开发人员面临的问题。

**现在的互联网架构**：



**目前互联网的新要求：3V和3高**

大数据时代的3V

Volume：海量，数据量极大

Variety：多样

数据类型：文本、图片、音频、视频……

终端设备：PC、移动端、嵌入式设备……

Velocity：实时

直播，金融证券……

互联网时代的3高

高可扩

不断优化现有的功能，不断开发新的功能；

高性能

不能让用户感觉到等待的时间；

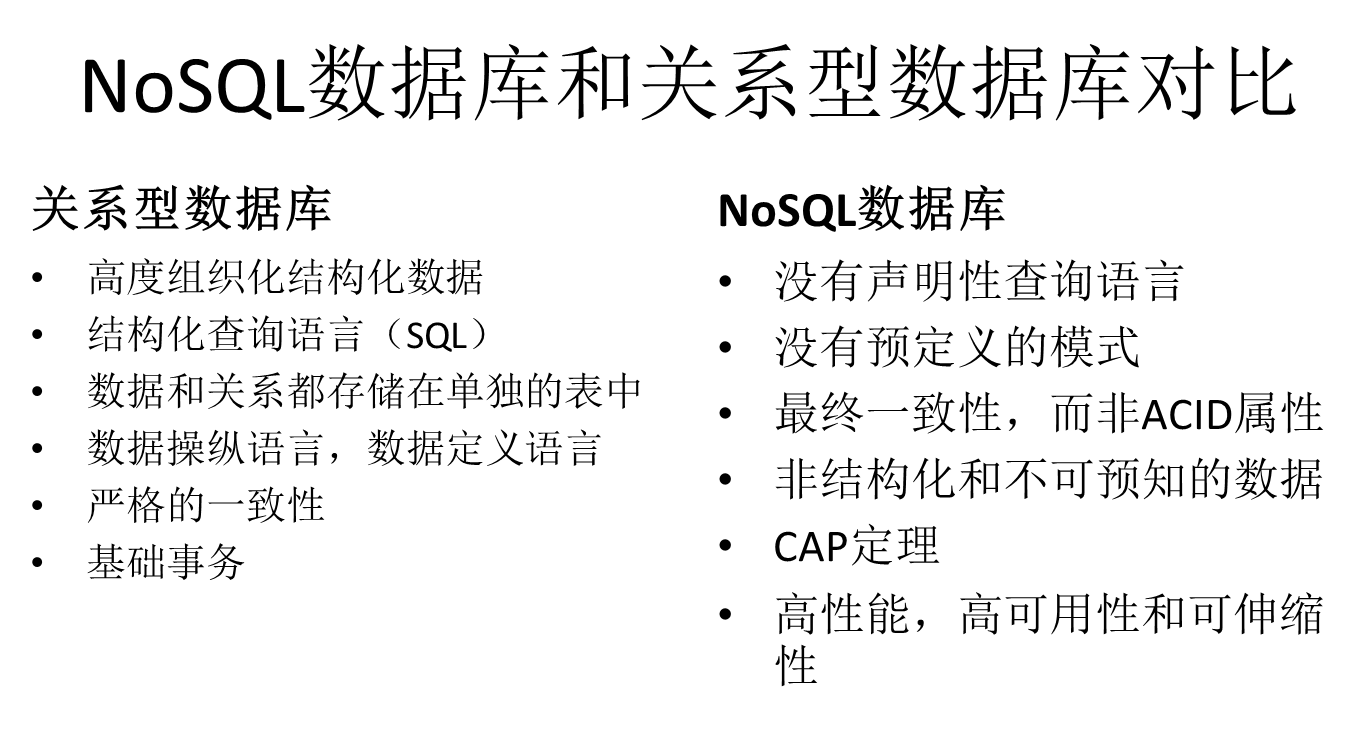
高并发

同时处理并发请求的能力，如双十一的秒杀、抢购火车票；

提升硬件，优化系统，优化项目，将费时的操作进入异步处理；

## 2. NoSQL

不仅仅是SQL—关系型数据库的强大助力



NoSQL数据库的优势

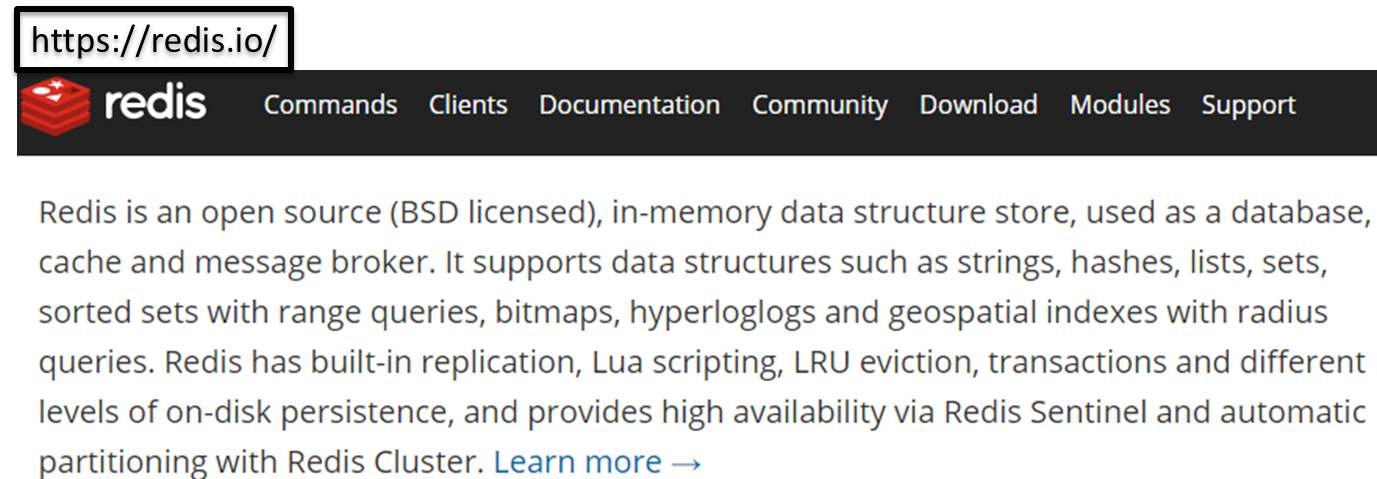
* 易扩展
  + NoSQL数据库种类繁多，但它们都有一个共通的特点：就是去除关系型数据库的“关系型”特点。数据之间无关系，这样就变得非常容易扩展，而相对应的来看：关系型数据库修改表结构非常困难。这就为项目架构设计提供了更大的扩展空间。
* 大数据量高性能
  + NoSQL数据库都具有非常高的读写性能，尤其在大数据量的情况下，表现同样优秀。这得益于NoSQL数据库中数据之间没有“关系”，数据库结构简单。
  + 从缓存角度来看，MySQL的Query Cache是表级别的粗粒度缓存，假设存储了100条数据，其中有一条数据修改了，整个缓存失效，效率很低。而NoSQL数据库的缓存是记录级的细粒度缓存，任何一条记录的修改都不影响其他记录，效率很高。
* 多样灵活的数据模型
  + NoSQL数据库无需事先为要存储的数据建立字段，随时可以存储自定义的数据格式。而在关系数据库里，增删字段是一件非常麻烦的事情。如果是非常大数据量的表，增减修改字段简直就是一个噩梦。

## 3. Redis

### 3.1 简介

**Redis**:Remote Dictionary Server(远程字典服务器)

官网：





### 3.2 安装

①将Redis的tar包上传到opt目录

②解压缩

③安装gcc环境

我们需要将源码编译后再安装，因此需要安装c语言的编译环境！不能直接make！



两种方式：

第一种：可以上网，yum install –y gcc-c++



第二种：不能上网，确保插入了安装光盘！

挂载安装光盘，然后进入Packages中，依次执行以下命令：

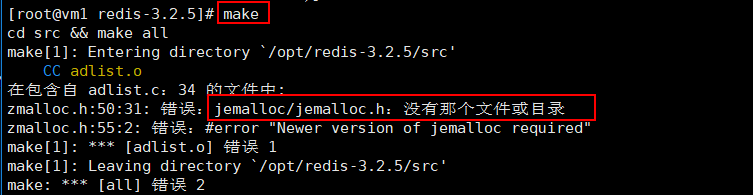
|  |
| --- |
| rpm -ivh mpfr-2.4.1-6.el6.x86\_64.rpm |
| rpm -ivh cpp-4.4.7-17.el6.x86\_64.rpm |
| rpm -ivh ppl-0.10.2-11.el6.x86\_64.rpm |
| rpm -ivh cloog-ppl-0.15.7-1.2.el6.x86\_64.rpm |
| rpm -ivh gcc-4.4.7-17.el6.x86\_64.rpm |

之后查看安装是否成功：rpm –qa|grep gcc



常见错误：在没有安装gcc环境下，如果执行了make，不会成功！安装环境后，第二次make有可能报错：

Jemalloc/jemalloc.h:没有那个文件



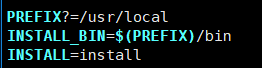
解决： 运行 make distclean之后再make

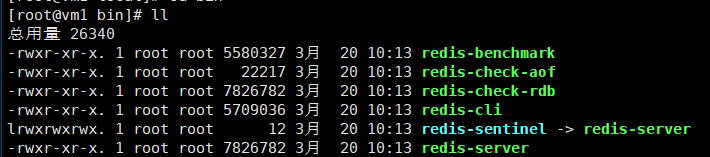
④编译，执行make命令！

⑤编译完成后，安装，执行make install命令！



⑥文件会被安装到 /usr/local/bin目录





⑦可以将redis的bin目录，加入到环境变量中

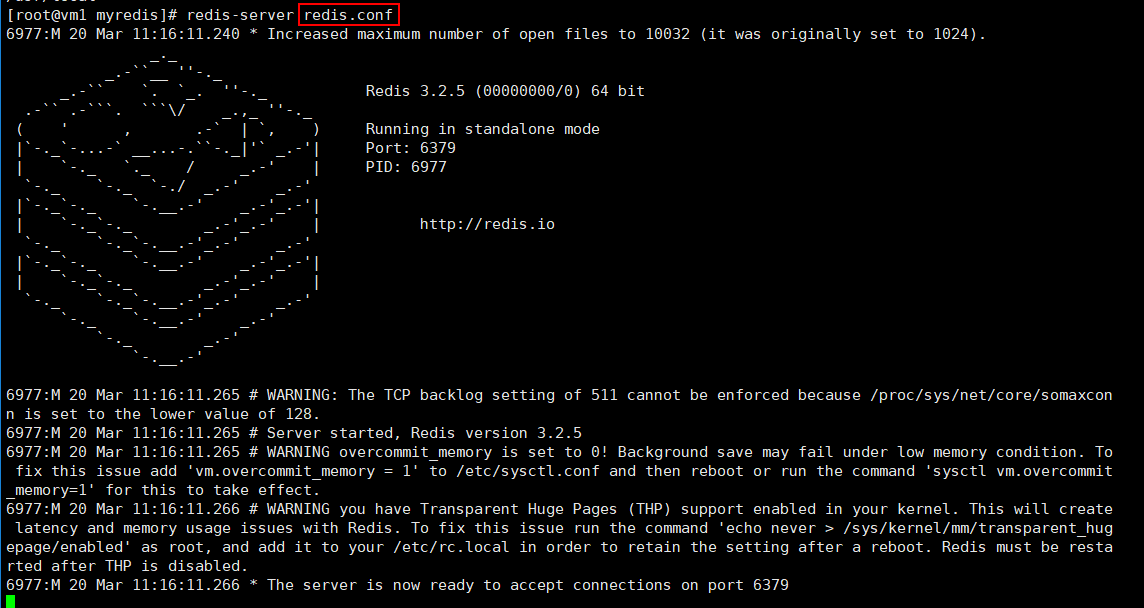
|  |  |
| --- | --- |
| bin目录常用命令 | |
| Redis-benchmark | 压力测试。标准是每秒80000次写操作，110000次读操作 (服务启动起来后执行,类似安兔兔跑分) |
| Redis-check-aof | 修复有问题的AOF文件 |
| Redis-check-dump | 修复有问题的dump.rdb文件 |
| Redis-sentinel | 启动哨兵，集群使用 |
| redis-server | 启动服务器 |
| redis-cli | 启动客户端 |

### 3.3 启动

#### 3.3.1 服务端启动

将配置文件，保留一份副本，进行启动。

命令： redis-server conf

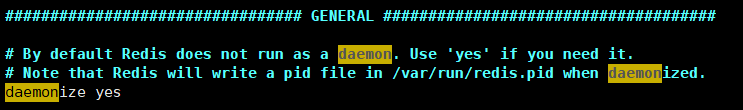


但是这样的话，发现命令窗口被占用了，很不方便。当然你可以再启动一个会话窗口。

解决：修改配置文件，改为守护进程，在后台运行

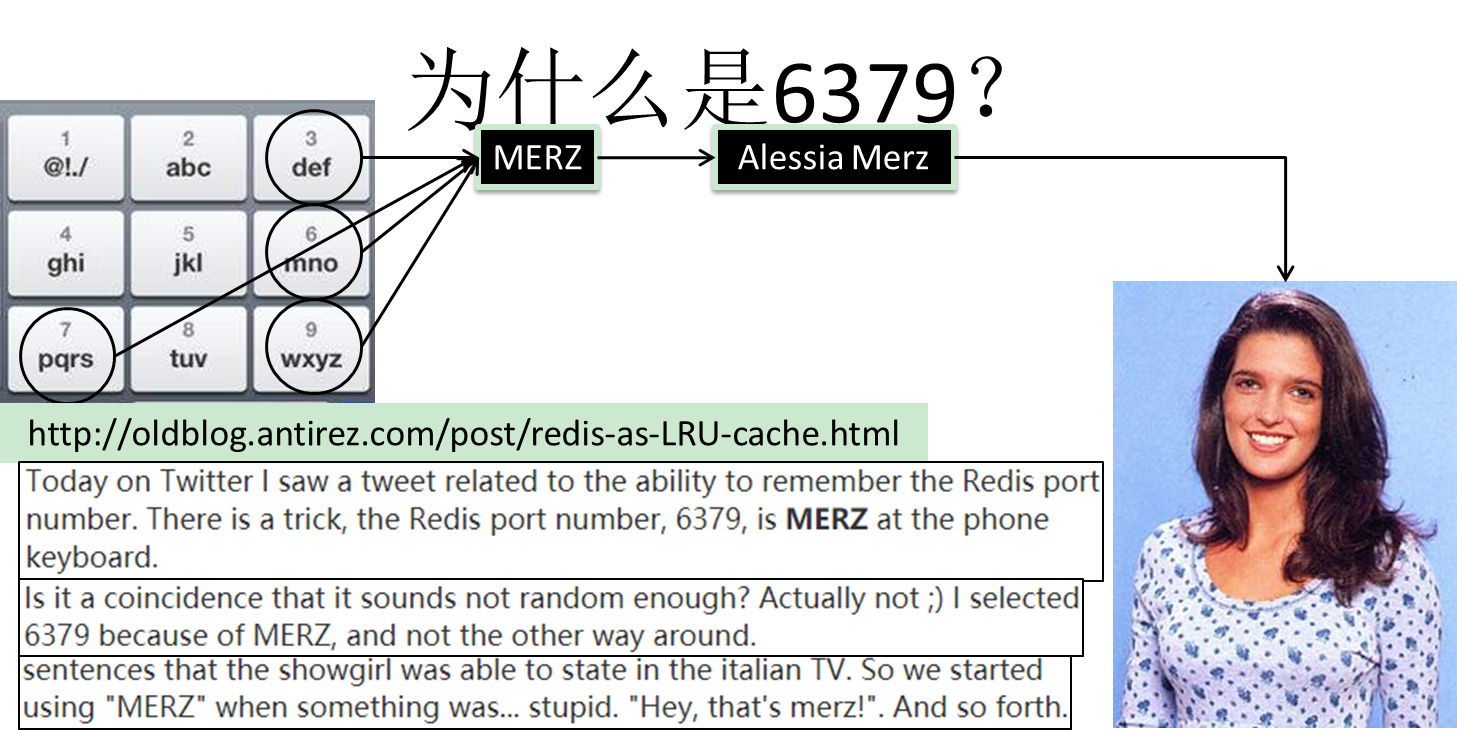
我们选择后台运行，怎么办？修改配置文件。

daemonize yes



后台启动后，查看服务： netstat –anp|grep 6379





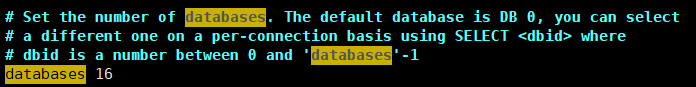
#### 3.3.2 客户端登录

|  |  |  |  |
| --- | --- | --- | --- |
| 命令 | 说明 | 举例 | 备注 |
| redis-cli | 启动客户端 | redis-cli –p 端口号  连接指定的端口号 | 直接执行的话，默认端口号就是6379； |
| ping | 测试联通 |  | 回复pong代表联通 |
| exit | 退出客户端 |  |  |
| redis-cli shutdown | 停止服务器 | redis-cli -h 127.0.0.1 -p 6379 shutdown  停止指定ip指定端口号的服务器 | redis是通过客户端发送停止服务器的命令 |

# 二、Redis基本操作

## 1. 数据库连接操作

|  |  |  |  |
| --- | --- | --- | --- |
| 命令 | 说明 | 举例 | 备注 |
| select <dbid> | 切换数据库 | select 1：切换到1号库 | 开启redis服务后，一共有16（0-15）个库，默认在0号库 |
| flushdb | 清空当前库 |  |  |
| dbsize | 查看数据库数据个数 |  |  |
| flushall | 通杀全部库 |  |  |



## 2. key的操作

Redis中的数据以键值对（key-value）为基本存储方式，其中key都是字符串。

|  |  |
| --- | --- |
| 表达式 | 描述 |
| KEYS pattern | 查询符合指定表达式的所有key，支持\*，？等 |
| TYPE key | 查看key对应值的类型 |
| EXISTS key | 指定的key是否存在，0代表不存在，1代表存在 |
| DEL key | 删除指定key |
| RANDOMKEY | 在现有的KEY中随机返回一个 |
| EXPIRE key seconds | 为键值设置过期时间，单位是秒，过期后key会被redis移除 |
| TTL key | 查看key还有多少秒过期，-1表示永不过期，-2表示已过期 |
| RENAME key newkey | 重命名一个key，NEWKEY不管是否是已经存在的都会执行，如果NEWKEY已经存在则会被覆盖 |
| RENAMENX key newkey | 只有在NEWKEY不存在时能够执行成功，否则失败 |

## 3. 常用五大数据类型

Redis中的数据以键值对（key-value）为基本存储方式，其中key都是字符串，这里探讨数据类型都是探讨value的类型。

|  |  |  |
| --- | --- | --- |
| key | value | |
| string | 字符串 |
| list | 可以重复的集合 |
| set | 不可以重复的集合 |
| hash | 类似于Map<String,String> |
| zset（sorted set） | 带分数的set |

## 4. String操作

String类型是Redis中最基本的类型，它是key对应的一个单一值。

二进制安全，不必担心由于编码等问题导致二进制数据变化。所以redis的string可以包含任何数据，比如jpg图片或者序列化的对象。

Redis中一个字符串值的最大容量是512M。

|  |  |
| --- | --- |
| SET key value | 添加键值对 |
| GET key | 查询指定key的值 |
| APPEND key value | 将给定的value追加到原值的末尾 |
| STRLEN key | 获取值的长度 |
| SETNX key value | 只有在 key 不存在时设置 key 的值 |
| INCR key | 指定key的值自增1，只对数字有效 |
| DECR key | 指定key的值自减1，只对数字有效 |
| INCRBY key num | 自增num |
| DECRBY key num | 自减num |
| MSET key1 value1 key2 value2… | 同时设置多个key-value对 |
| MGET key1 key2 | 同时获取一个或多个value |
| MSETNX key1 value1 key2 value2 | 当key不存在时，设置多个key-value对 |
| GETRANGE key起始索引 结束索引 | 获取指定范围的值，都是闭区间 |
| SETRANGE key起始索引 value | 从起始位置开始覆写指定的值 |
| GETSET key value | 以新换旧，同时获取旧值 |
| SETEX key 过期时间 value | 设置键值的同时，设置过期时间，单位秒 |

## 5. list操作

在Java中list 一般是单向链表，如常见的Arraylist，只能从一侧插入。

在Redis中，list是双向链表。可以从两侧插入。

可以简单理解为两端开口的，两端都可以进出。使用一个动画来演示。

常见操作：

遍历：遍历的时候，是从左往右取值；

删除：弹栈，POP；

添加：压栈，PUSH ；



Redis 列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素导列表的头部（左边）或者尾部（右边）。它的底层实际是个双向链表，对两端的操作性能很高，通过索引下标的操作中间的节点性能会较差。

|  |  |
| --- | --- |
| LPUSH/RPUSH key value1 value2… | 从左边/右边压入一个或多个值  头尾效率高，中间效率低 |
| LPOP/RPOP key | 从左边/右边弹出一个值  值在键在，值光键亡  弹出=返回+删除 |
| LRANGE key start stop | 查看指定区间的元素  正着数：0,1,2,3,...  倒着数：-1,-2,-3,... |
| LINDEX key index | 按照索引下标获取元素（从左到右） |
| LLEN key | 获取列表长度 |
| LINSERT key BEFORE|AFTER value newvalue | 在指定value的前后插入newvalue |
| LREM key n value | 从左边删除n个value |
| LSET key index value | 把指定索引位置的元素替换为另一个值 |
| LTRIM key start stop | 仅保留指定区间的数据 |
| RPOPLPUSH key1 key2 | 从key1右边弹出一个值，左侧压入到key2 |

## 6. set操作

set是无序的，且是不可重复的。

|  |  |
| --- | --- |
| SADD key member [member ...] | 将一个或多个 member 元素加入到集合 key 当中，已经存在于集合的 member 元素将被忽略。 |
| SMEMBERS key | 取出该集合的所有值 |
| SISMEMBER key value | 判断集合<key>是否为含有该<value>值，有返回1，没有返回0 |
| SCARD key | 返回集合中元素的数量 |
| SREM key member [member ...] | 从集合中删除元素 |
| SPOP key [count] | 从集合中随机弹出count个数量的元素，count不指定就弹出1个 |
| SRANDMEMBER key [count] | 从集合中随机返回count个数量的元素，count不指定就返回1个 |
| SINTER key [key ...] | 将指定的集合进行“交集”操作 |
| SINTERSTORE dest key [key ...] | 取交集，另存为一个set |
| SUNION key [key ...] | 将指定的集合执行“并集”操作 |
| SUNIONSTORE dest key [key ...] | 取并集，另存为set |
| SDIFF key [key ...] | 将指定的集合执行“差集”操作 |
| SDIFFSTORE dest key [key ...] | 取差集，另存为set |

## 7. hash操作

Hash数据类型的键值对中的值是“单列”的，不支持进一步的层次结构。

|  |  |
| --- | --- |
| key | field:value |
| "k01":"v01"  "k02":"v02"  "k03":"v03"  "k04":"v04"  "k05":"v05"  "k06":"v06"  "k07":"v07" |

从前到后的数据对应关系

* + JSON

stu:{"stu\_id":10,"stu\_name":"tom","stu\_age":30}

* + Java

public class Student {

private Integer stuId;//10

private String stuName;//"tom"

private Integer stuAge;//30

...

}

Student stu=new Student()’

stu.setStuId=10;

stu.setStuName=”tom”;

stu.setStuAge=30;

* + Redis hash

|  |  |  |
| --- | --- | --- |
| key | value(hash) | |
| stu | stu\_id | 10 |
| stu\_name | tom |
| stu\_age | 30 |

常用操作：

|  |  |
| --- | --- |
| HSET key field value | 为key中的field赋值value |
| HMSET key field value [field value ...] | 为指定key批量设置field-value |
| HSETNX key field value | 当指定key的field不存在时，设置其value |
| HGETALL key | 获取指定key的所有信息（field和value） |
| HKEYS key | 获取指定key的所有field |
| HVALS key | 获取指定key的所有value |
| HLEN key | 指定key的field个数 |
| HGET key field | 从key中根据field取出value |
| HMGET key field [field ...] | 为指定key获取多个filed的值 |
| HEXISTS key field | 指定key是否有field |
| HINCRBY key field increment | 为指定key的field加上增量increment |

## 8. zset操作

zset是一种特殊的set（sorted set），在保存value的时候，为每个value多保存了一个score信息。根据score信息，可以进行排序。

这个评分（score）被用来按照从最低分到最高分的方式排序集合中的成员。集合的成员是唯一的，但是评分可以是重复了

|  |  |
| --- | --- |
| ZADD key [score member ...] | 添加 |
| ZSCORE key member | 返回指定值的分数 |
| ZRANGE key start stop [WITHSCORES] | 返回指定区间的值，可选择是否一起返回scores |
| ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count] | 在分数的指定区间内返回数据，从小到大排列 |
| ZREVRANGEBYSCORE key max min [WITHSCORES] [LIMIT offset count] | 在分数的指定区间内返回数据，从大到小排列 |
| ZCARD key | 返回集合中所有的元素的数量 |
| ZCOUNT key min max | 统计分数区间内的元素个数 |
| ZREM key member | 删除该集合下，指定值的元素 |
| ZRANK key member | 返回该值在集合中的排名，从0开始 |
| ZINCRBY key increment value | 为元素的score加上增量 |

# 三、Redis配置文件

## 1.单位说明



1k和1kb是不同的；单位的大小写不敏感！

## 2. include



可以将公共的配置放入到一个公共的配置文件中，然后通过子配置文件引入父配置文件中的内容！

将配置按照模块分开！

## 3. network

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 备注 |
| bind | 限定访问的主机地址 | 如果没有bind，就是任意ip地址都可以访问。生产环境下，需要写自己应用服务器的ip地址。 |
| protected-mode | 安全防护模式 | 如果没有指定bind指令，也没有配置密码，那么保护模式就开启，只允许本机访问。 |
| port | 端口号 | 默认是6379 |
| tcp-backlog | 网络连接过程中，某种状态的队列的长度 | edis是单线程的，指定高并发时访问时排队的长度。超过后，就呈现阻塞状态。可以理解是一个请求到达后至到接受进程处理前的队列长度。（一般情况下是运维根据集群性能调控）高并发情况下，此值可以适当调高。 |
| timeout | 超时时间 | 默认永不超时 |
| tcp-keepalive | 对客户端的心跳检测间隔时间 |  |

## 4. general

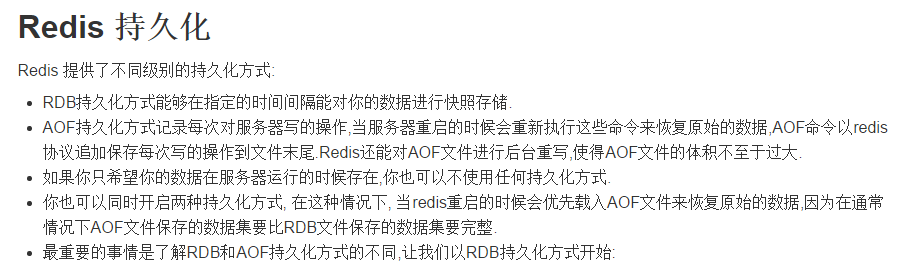
|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 备注 |
| daemonize | 是否为守护进程模式运行 | 守护进程模式可以在后台运行 |
| pidfile | 进程id文件保存的路径 | 配置PID文件路径，当redis作为守护进程运行的时候，它会把 pid 默认写到 /var/redis/run/redis\_6379.pid 文件里面 |
| loglevel | 定义日志级别 | debug（记录大量日志信息，适用于开发、测试阶段）  verbose（较多日志信息）  notice（适量日志信息，使用于生产环境）  warning（仅有部分重要、关键信息才会被记录） |
| logfile | 日志文件的位置 | 当指定为空字符串时，为标准输出，如果redis以守护进程模式运行，那么日志将会输出到/dev/null |
| syslog-enabled | 是否记录到系统日志 | 要想把日志记录到系统日志服务中，就把它改成 yes |
| syslog-ident | 设置系统日志的ID |  |
| syslog-facility | 指定系统日志设置 | 必须是 USER 或者是 LOCAL0-LOCAL7 之间的值 |
| databases | 设置数据库数量 |  |

## 5. 其他

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 备注 |
| requirepass | 设置密码 |  |
| maxclients | 最大连接数 |  |
| maxmemory | 最大占用多少内存 | 一旦占用内存超限，就开始根据缓存清理策略移除数据如果Redis无法根据移除规则来移除内存中的数据，或者设置了“不允许移除”，  那么Redis则会针对那些需要申请内存的指令返回错误信息，比如SET、LPUSH等。 |
| maxmemory-policy noeviction | 缓存清理策略 | （1）volatile-lru：使用LRU算法移除key，只对设置了过期时间的键  （2）allkeys-lru：使用LRU算法移除key  （3）volatile-random：在过期集合中移除随机的key，只对设置了过期时间的键  （4）allkeys-random：移除随机的key  （5）volatile-ttl：移除那些TTL值最小的key，即那些最近要过期的key  （6）noeviction：不进行移除。针对写操作，只是返回错误信息 |
| maxmemory-samples | 样本数 | 样本数越小，准确率越低，但是性能越好。LRU算法和最小TTL算法都并非是精确的算法，而是估算值，所以你可以设置样本的大小。一般设置3到7的数字。 |

# 四、持久化

Redis主要是工作在内存中。内存本身就不是一个持久化设备，断电后数据会清空。所以Redis在工作过程中，如果发生了意外停电事故，如何尽可能减少数据丢失。



## 1. RDB

### 1.1 RDB简介

RDB：在指定的时间间隔内将内存中的数据集快照写入磁盘，也就是行话讲的Snapshot快照，它恢复时是将快照文件直接读到内存里。

工作机制：每隔一段时间，就把内存中的数据保存到硬盘上的指定文件中。

RDB是默认开启的！

Redis会单独创建（fork）一个子进程来进行持久化，会先将数据写入到一个临时文件中，待持久化过程都结束了，再用这个临时文件替换上次持久化好的文件。整个过程中，主进程是不进行任何IO操作的，这就确保了极高的性能如果需要进行大规模数据的恢复，且对于数据恢复的完整性不是非常敏感，那RDB方式要比AOF方式更加的高效。

RDB的缺点是最后一次持久化后的数据可能丢失。

### 1.2 RDB保存策略

save 900 1 900 秒内如果至少有 1 个 key 的值变化，则保存

save 300 10 300 秒内如果至少有 10 个 key 的值变化，则保存

save 60 10000 60 秒内如果至少有 10000 个 key 的值变化，则保存

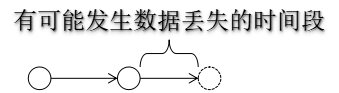
save “” 就是禁用RDB模式；

### 1.3 RDB常用属性配置

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 备注 |
| save | 保存策略 |  |
| dbfilename | RDB快照文件名 |  |
| dir | RDB快照保存的目录 | 必须是一个目录，不能是文件名。最好改为固定目录。默认为./代表执行redis-server命令时的当前目录！ |
| stop-writes-on-bgsave-error | 是否在备份出错时，继续接受写操作 | 如果用户开启了RDB快照功能，那么在redis持久化数据到磁盘时如果出现失败，默认情况下，redis会停止接受所有的写请求 |
| rdbcompression | 对于存储到磁盘中的快照，可以设置是否进行压缩存储。 | 如果是的话，redis会采用LZF算法进行压缩。如果你不想消耗CPU来进行压缩的话，  可以设置为关闭此功能，但是存储在磁盘上的快照会比较大。 |
| rdbchecksum | 是否进行数据校验 | 在存储快照后，我们还可以让redis使用CRC64算法来进行数据校验，但是这样做会增加大约10%的性能消耗，  如果希望获取到最大的性能提升，可以关闭此功能。 |

### 1.4 RDB数据丢失的情况

两次保存的时间间隔内，服务器宕机，或者发生断电问题。



### 1.5 RDB的触发

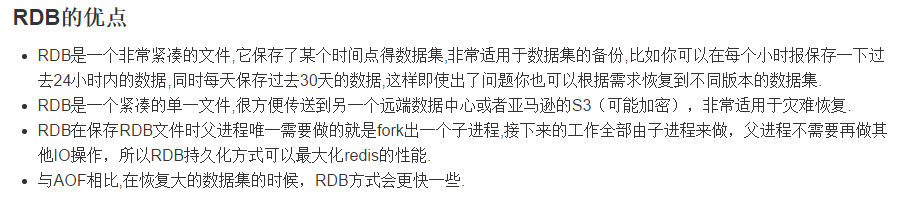
①基于自动保存的策略

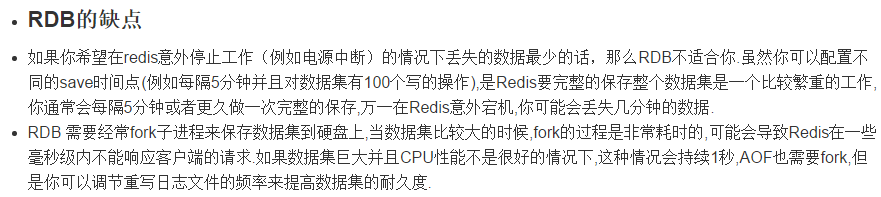
②执行save，或者bgsave命令！执行时，是阻塞状态。

③执行flushall命令，也会产生dump.rdb，但里面是空的，没有意义。

④当执行shutdown命令时，也会主动地备份数据。

### 1.6 RDB的优缺点





## 2. AOF

### 2.1 AOF简介

* AOF是以日志的形式来记录每个写操作，将每一次对数据进行修改，都把新建、修改数据的命令保存到指定文件中。Redis重新启动时读取这个文件，重新执行新建、修改数据的命令恢复数据。
* 默认不开启，需要手动开启
* AOF文件的保存路径，同RDB的路径一致。
* AOF在保存命令的时候，只会保存对数据有修改的命令，也就是写操作！
* 当RDB和AOF存的不一致的情况下，按照AOF来恢复。因为AOF是对RDB的补充。备份周期更短，也就更可靠。

### 2.2 AOF保存策略

appendfsync always：每次产生一条新的修改数据的命令都执行保存操作；效率低，但是安全！

appendfsync everysec：每秒执行一次保存操作。如果在未保存当前秒内操作时发生了断电，仍然会导致一部分数据丢失（即1秒钟的数据）。

appendfsync no：从不保存，将数据交给操作系统来处理。更快，也更不安全的选择。

推荐（并且也是默认）的措施为每秒 fsync 一次， 这种 fsync 策略可以兼顾速度和安全性。

### 2.3 AOF常用属性

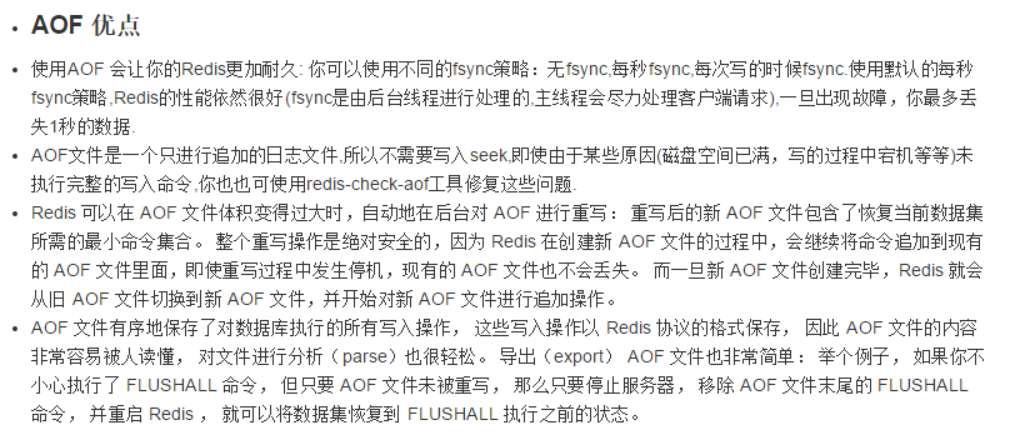
|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 备注 |
| appendonly | 是否开启AOF功能 | 默认是关闭的 |
| appendfilename | AOF文件名称 |  |
| appendfsync | AOF保存策略 | 官方建议everysec |
| no-appendfsync-on-rewrite | 在重写时，是否执行保存策略 | 执行重写，可以节省AOF文件的体积；而且在恢复的时候效率也更高。 |
| auto-aof-rewrite-percentage | 重写的触发条件 | 当目前aof文件大小超过上一次重写的aof文件大小的百分之多少进行重写 |
| auto-aof-rewrite-min-size | 设置允许重写的最小aof文件大小 | 避免了达到约定百分比但尺寸仍然很小的情况还要重写 |
| aof-load-truncated | 截断设置 | 如果选择的是yes，当截断的aof文件被导入的时候，会自动发布一个log给客户端然后load |

### 2.4 AOF文件的修复

如果AOF文件中出现了残余命令，会导致服务器无法重启。此时需要借助redis-check-aof工具来修复！

命令： redis-check-aof --fix 文件

### 2.5 AOF的优缺点



优点：

* 备份机制更稳健，丢失数据概率更低
* 可读的日志文本，通过操作AOF稳健，可以处理误操作



缺点：

* 比起RDB占用更多的磁盘空间
* 恢复备份速度要慢
* 每次读写都同步的话，有一定的性能压力
* 存在个别Bug，造成恢复不能

## 3. 备份建议

### 3.1 如何看待数据“绝对”安全

Redis作为内存数据库从本质上来说，如果不想牺牲性能，就不可能做到数据的“绝对”安全。

RDB和AOF都只是尽可能在兼顾性能的前提下降低数据丢失的风险，如果真的发生数据丢失问题，尽可能减少损失。

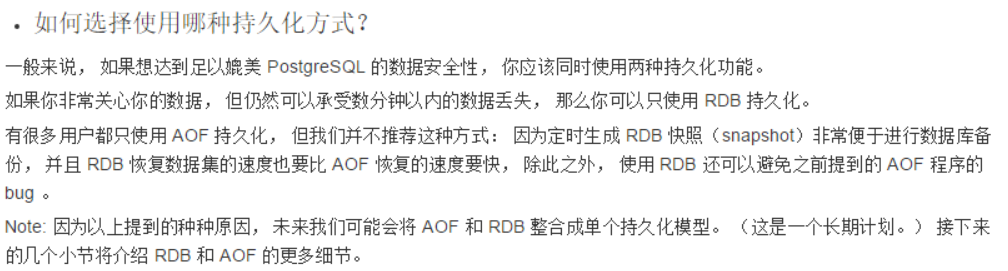
在整个项目的架构体系中，Redis大部分情况是扮演“二级缓存”角色。

二级缓存适合保存的数据

* 经常要查询，很少被修改的数据。
* 不是非常重要，允许出现偶尔的并发问题。
* 不会被其他应用程序修改。

如果Redis是作为缓存服务器，那么说明数据在MySQL这样的传统关系型数据库中是有正式版本的。数据最终以MySQL中的为准。

### 3.2 官方建议



官方推荐两个都用；如果对数据不敏感，可以选单独用RDB；不建议单独用AOF，因为可能出现Bug;如果只是做纯内存缓存，可以都不用

# 五、事务

## 1. 事务简介

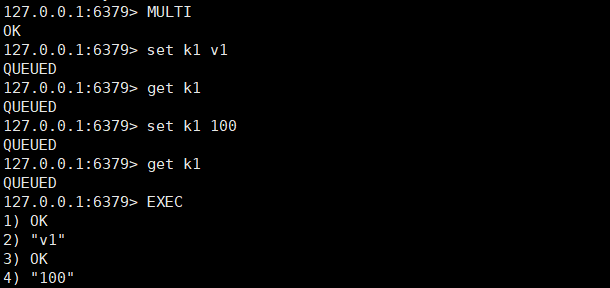
* Redis中事务，不同于传统的关系型数据库中的事务。
* Redis中的事务指的是一个单独的隔离操作。
* Redis的事务中的所有命令都会序列化、按顺序地执行且不会被其他客户端发送来的命令请求所打断。
* Redis事务的主要作用是串联多个命令防止别的命令插队

## 2. 事务常用命令

|  |  |
| --- | --- |
| MULTI | 标记一个事务块的开始 |
| EXEC | 执行事务中所有在排队等待的指令并将链接状态恢复到正常 当使用WATCH 时，只有当被监视的键没有被修改，且允许检查设定机制时，EXEC会被执行 |
| DISCARD | 刷新一个事务中所有在排队等待的指令，并且将连接状态恢复到正常。  如果已使用WATCH，DISCARD将释放所有被WATCH的key。 |
| WATCH | 标记所有指定的key 被监视起来，在事务中有条件的执行（乐观锁） |

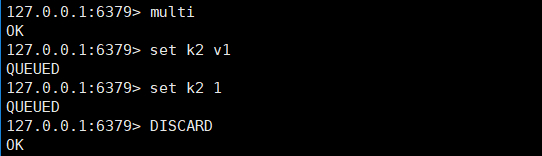
## 3. 事务的常见演示

### 3.1 简单组队



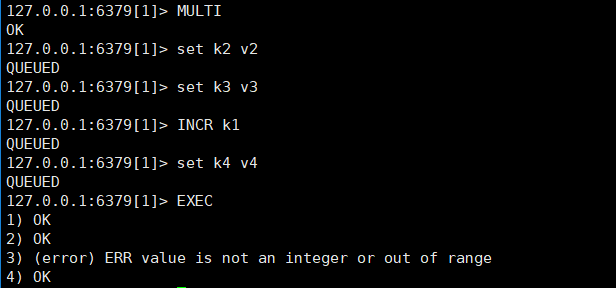
MULTI开启组队，EXEC依次执行队列中的命令。

DISCARD中途取消组队



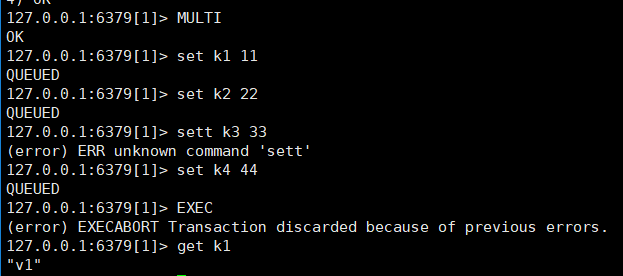
### 3.2 组队失败

#### 3.2.1 自作自受



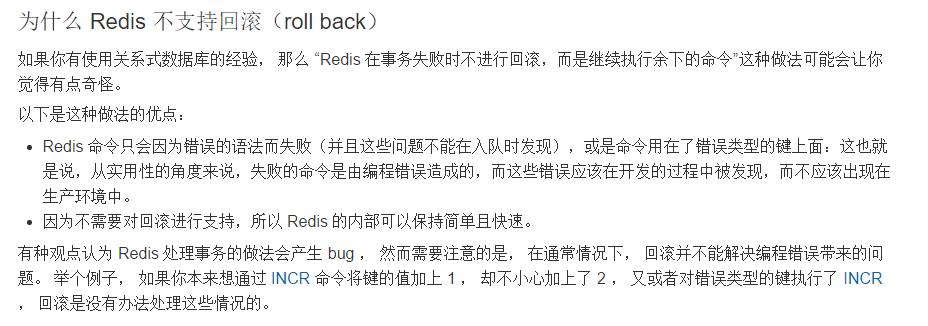
此种情况，语法符合规范，Redis只有在执行中，才可以发现错误。而在Redis中，并没有回滚机制，因此错误的命令，无法执行，正确的命令会全部执行！

#### 3.2.2 殃及池鱼



在编译的过程中，Redis检测出来了错误的语法命令，因此它认为这条组队，一定会发生错误，因此全体取消；

#### 3.3.3 官方说明



## 4. 锁



### 4.1 悲观锁

执行操作前假设当前的操作肯定（或有很大几率）会被打断（悲观）。基于这个假设，我们在做操作前就会把相关资源锁定，不允许自己执行期间有其他操作干扰。

Redis不支持悲观锁。Redis作为缓存服务器使用时，以读操作为主，很少写操作，相应的操作被打断的几率较少。不采用悲观锁是为了防止降低性能。

### 4.2 乐观锁

执行操作前假设当前操作不会被打断（乐观）。基于这个假设，我们在做操作前不会锁定资源，万一发生了其他操作的干扰，那么本次操作将被放弃。

## 5. Redis中的锁策略

* + Redis采用了乐观锁策略（通过watch操作）。乐观锁支持读操作，适用于多读少写的情况！
  + 在事务中，可以通过watch命令来加锁；使用 UNWATCH可以取消加锁；
  + 如果在事务之前，执行了WATCH（加锁），那么执行EXEC 命令或 DISCARD 命令后，锁对自动释放，即不需要再执行 UNWATCH 了

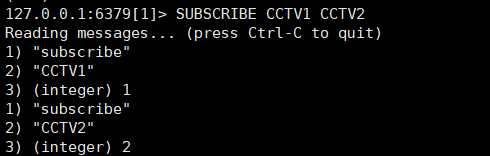
# 六、redis消息订阅

消息订阅是进程间的一种消息通信方式，即发送者（pub）发送消息，订阅者（sub）接收消息。Redis支持消息订阅机制。

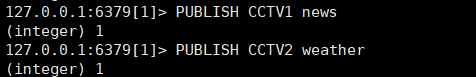
|  |  |  |
| --- | --- | --- |
| 命令 | 描述 | 举例 |
| SUBCRIBE [频道] | 订阅频道 |  |
| PUBLISH [频道][消息] | 向指定频道发布消息 |  |

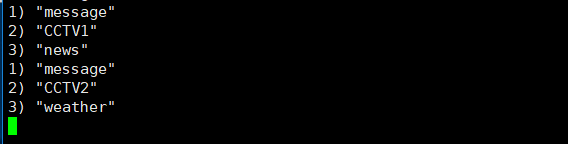
示例：

订阅者订阅消息后，此时控制台会处于阻塞状态，用于接收发布者发布的消息。



发布者发布消息：





# 七、主从复制

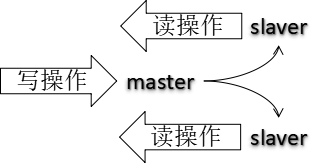
## 1. 主从简介

配置多台Redis服务器，以主机和备机的身份分开。主机数据更新后，根据配置和策略，自动同步到备机的master/salver机制，Master以写为主，Slave以读为主，二者之间自动同步数据。

目的：

读写分离提高Redis性能；

避免单点故障，容灾快速恢复

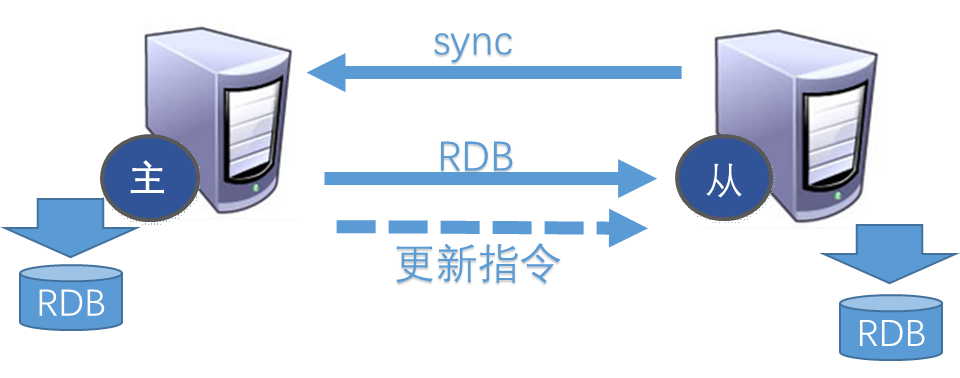


原理：

每次从机联通后，都会给主机发送sync指令，主机立刻进行存盘操作，发送RDB文件，给从机

从机收到RDB文件后，进行全盘加载。之后每次主机的写操作命令，都会立刻发送给从机，从机执行相同的命令来保证主从的数据一致！

注意：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，但是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，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），设置repl-diskless-sync yes。即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。



## 2. 主从准备

除非是不同的主机配置不同的Redis服务，否则在一台机器上面跑多个Redis服务，需要配置多个Redis配置文件。

①准备多个Redis配置文件，每个配置文件，需要配置以下属性

daemonize yes: 服务在后台运行

port：端口号

pidfile:pid保存文件

logfile：日志文件(如果没有指定的话，就不需要)

dump.rdb: RDB备份文件的名称

appendonly 关掉，或者是更改appendonly文件的名称。

样本：

include /root/redis\_repilication/redis.conf

port 6379

pidfile /var/run/redis\_6379.pid

dbfilename dump\_6379.rdb

②根据多个配置文件，启动多个Redis服务

原则是配从不配主。

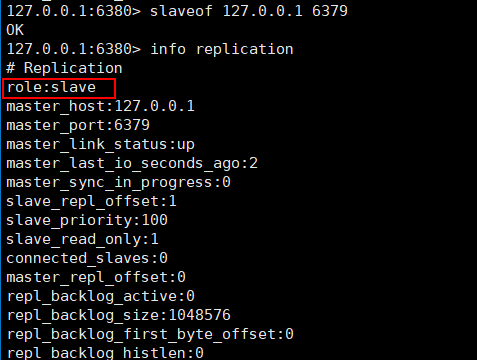
## 3. 主从建立

### 3.1 临时建立

原则：配从不配主。

配置：在从服务器上执行SLAVEOF ip:port命令；

查看：执行info replication命令；



### 3.2 永久建立

在从机的配置文件中，编写slaveof属性配置！

### 3.3 恢复身份

执行命令slaveof no one恢复自由身！

## 4. 主从常见问题

①从机是从头开始复制主机的信息，还是只复制切入以后的信息？

答：从头开始复制，即完全复制。

②从机是否可以写？

答：默认情况不能，可以通过修改配置文件，设置slave-read-only no.但是从机写入的数据是不能同步到主机，因此没用设置的必要！



③主机shutdown后，从机是上位还是原地待命？

答：原地待命

④主机又回来了后，主机新增记录，从机还能否顺利复制？

答：可以

⑤从机宕机后，重启，宕机期间主机的新增记录，从机是否会顺利复制？

答：可以

⑥其中一台从机down后重启，能否重认旧主？

答：不一定，看配置文件中是否配置了slaveof

⑦如果两台从机都从主机同步数据，此时主机的IO压力会增大，如何解决？

答：按照主---从（主）---从模式配置！

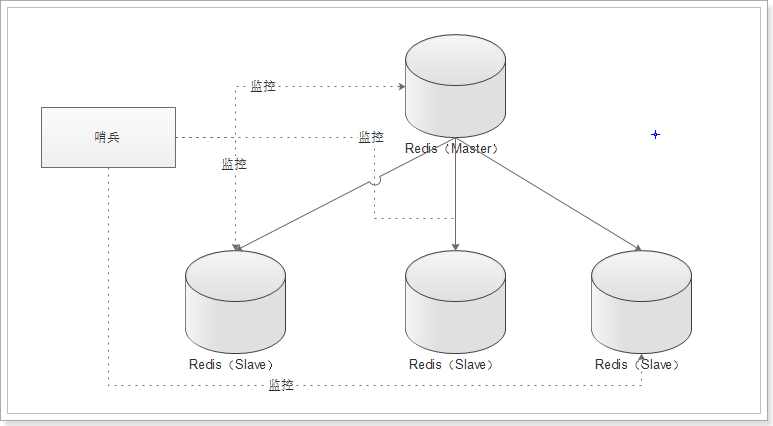
## 5. 哨兵模式

### 5.1 简介

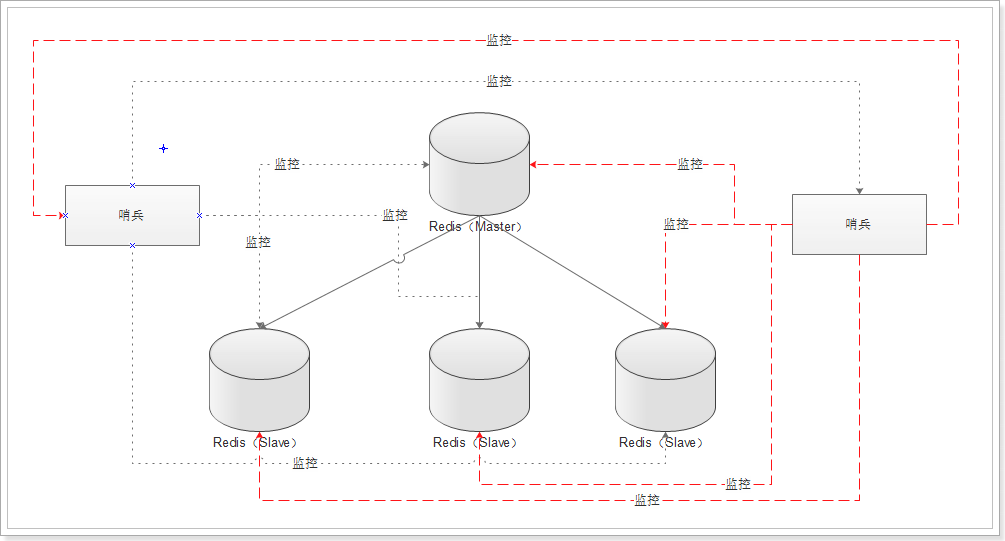
**作用**：

①主从状态检测

②如果Master异常，则会进行Master-Slave切换，将其中一个Slave作为Master，将之前的Master作为Slave



单个哨兵



多个哨兵

多个哨兵，不仅同时监控主从状态，且哨兵之间也互相监控！

**下线**：

①主观下线：Subjectively Down，简称 SDOWN，指的是当前 Sentinel 实例对某个redis服务器做出的下线判断。

②客观下线：Objectively Down， 简称 ODOWN，指的是多个 Sentinel 实例在对Master Server做出 SDOWN 判断，并且通过 SENTINEL is-master-down-by-addr 命令互相交流之后，得出的Master Server下线判断，然后开启failover.

**工作原理**：

①每个Sentinel以每秒钟一次的频率向它所知的Master，Slave以及其他 Sentinel 实例发送一个 PING 命令 ；

②如果一个实例（instance）距离最后一次有效回复 PING 命令的时间超过 down-after-milliseconds 选项所指定的值， 则这个实例会被 Sentinel 标记为主观下线；

③如果一个Master被标记为主观下线，则正在监视这个Master的所有 Sentinel 要以每秒一次的频率确认Master的确进入了主观下线状态；

④当有足够数量的 Sentinel（大于等于配置文件指定的值）在指定的时间范围内确认Master的确进入了主观下线状态， 则Master会被标记为客观下线 ；

⑤在一般情况下， 每个 Sentinel 会以每 10 秒一次的频率向它已知的所有Master，Slave发送 INFO 命令

⑥当Master被 Sentinel 标记为客观下线时，Sentinel 向下线的 Master 的所有 Slave 发送 INFO 命令的频率会从 10 秒一次改为每秒一次 ；

⑦若没有足够数量的 Sentinel 同意 Master 已经下线， Master 的主观下线状态就会被移除；

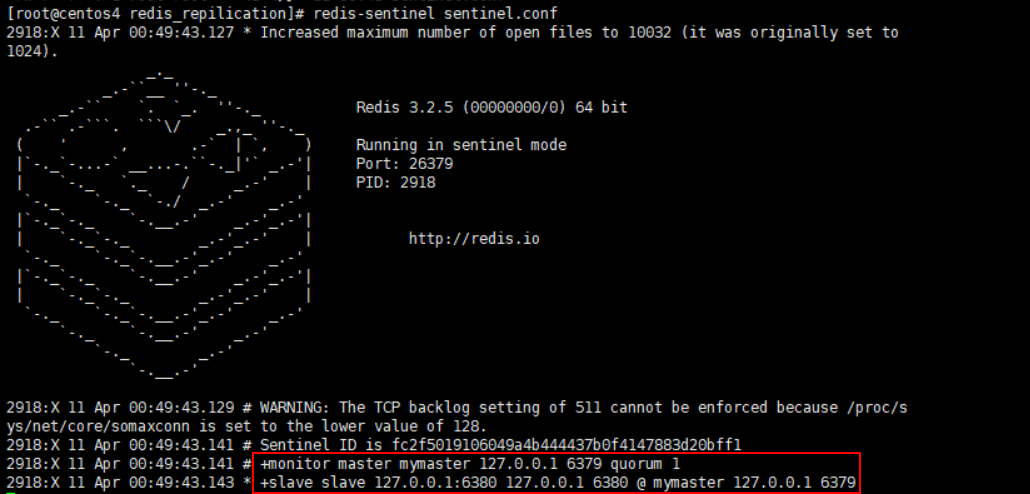
若 Master 重新向 Sentinel 的 PING 命令返回有效回复， Master 的客观下线状态就会被移除；

### 5.2 配置

哨兵模式需要配置哨兵的配置文件！

sentinel monitor mymaster 127.0.0.1 6379 1

启动哨兵：redis-sentinel sentinel.conf



### 5.3 主机宕机后

+sdown master mymaster 127.0.0.1 6379 【主观下线】

+odown master mymaster 127.0.0.1 6379 #quorum 1/1【客观下线】

……

+vote-for-leader 17818eb9240c8a625d2c8a13ae9d99ae3a70f9d2 1【选举leader】

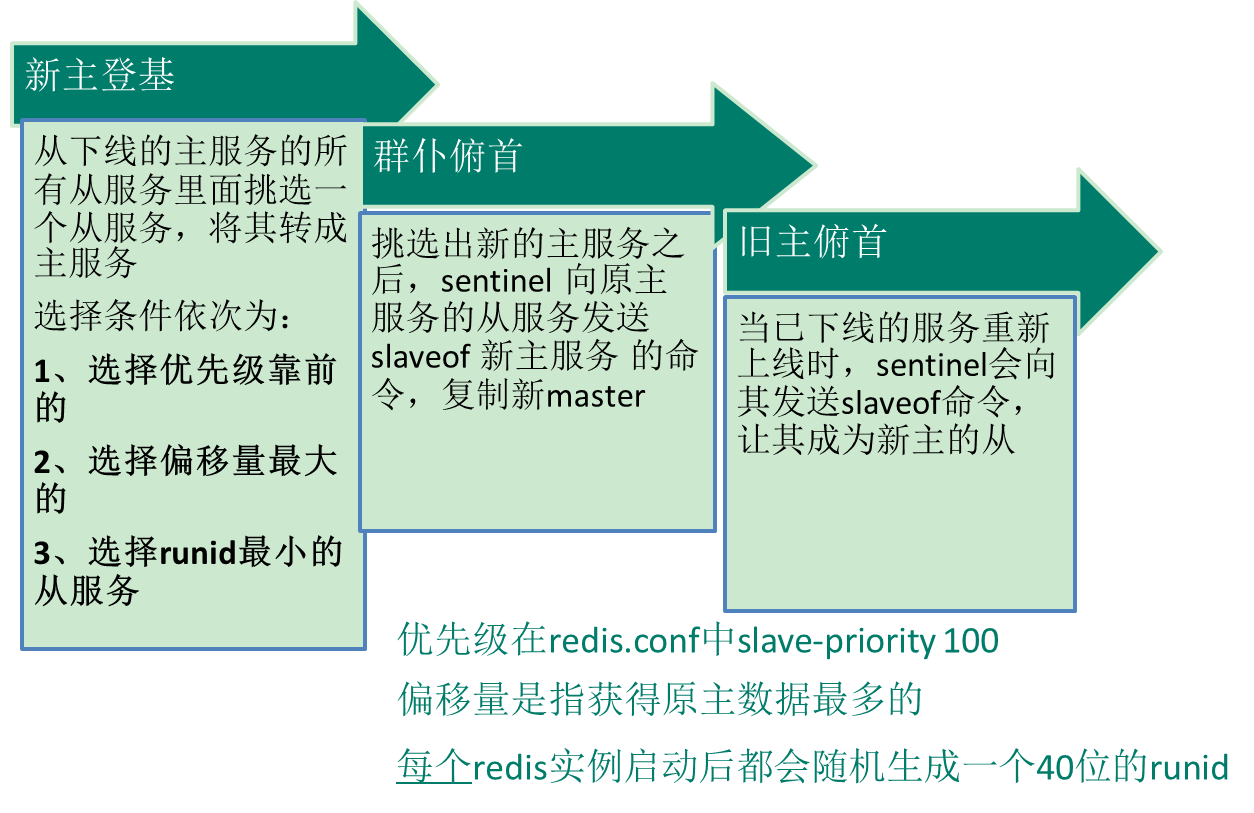
……

+failover-state-send-slaveof-noone slave 127.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6379【把一个从机设置为主机】

-------------挂掉的主机又重新启动---------------------

-sdown slave 127.0.0.1:6379 127.0.0.1 6379 @ mymaster 127.0.0.1 6381【离开主观下线状态】

+convert-to-slave slave 127.0.0.1:6379 127.0.0.1 6379 @ mymaster 127.0.0.1 6381【转换为从机】



### 5.4 哨兵模式使用Jedis

|  |
| --- |
| @Test  **public** **void** testSentinel() **throws** Exception {  Set<String> set = **new** HashSet<>();  // set中放的是哨兵的Ip和端口  set.add("192.168.6.10:26379");  GenericObjectPoolConfig poolConfig = **new** GenericObjectPoolConfig();  JedisSentinelPool jedisSentinelPool = **new** JedisSentinelPool("mymaster", set, poolConfig, 60000);  Jedis jedis = jedisSentinelPool.getResource();  String value = jedis.get("k7");  jedis.set("Jedis", "Jedis");  System.***out***.println(value);  } |

注意：使用哨兵模式启动的redis服务，必须使用真实的局域网ip！不要使用localhost和127.0.0.1!

# 八、Jedis

## 1. 简介

Jedis指通过Java连接Redis客户端。需要导入jar包：

|  |  |
| --- | --- |
| MySQL | Redis |
| Connection对象 | Jedis对象 |
| 连接池/数据源 | 连接池 |
| Connection对象每次用完需要关闭 | Jedis对象每次用完需要关闭 |

## 2. 配置

①导包：



②在所连接的Redis客户端的配置文件中：

* 注释掉bind 127.0.0.1
* 关闭保护模式，将ptotect-mode设置为no
* 禁用Linux防火墙，执行service iptables stop

③ 测试连接

@Test

**public** **void** testConnect() {

//连接指定的redis，需要ip地址和端口号

Jedis jedis=**new** Jedis("192.168.162.128", 6379);

String ping = jedis.ping();

System.***out***.println(ping);

}

## 3. 常用API

### 3.1 测试key

@Test

**public** **void** testKeys() {

//连接指定的redis，需要ip地址和端口号

Jedis jedis=**new** Jedis("192.168.162.128", 6379);

//获取所有key

Set<String> keys = jedis.keys("\*");

**for** (String key : keys) {

System.***out***.println(key);

}

//判断是否存在某个key

System.***out***.println("是否存在k2:"+jedis.exists("k2"));

//测试某个key的过期时间

System.***out***.println("k1的存活时间:"+jedis.ttl("k2"));

jedis.close();

}

### 3.2 测试string

@Test

**public** **void** testString() {

//连接指定的redis，需要ip地址和端口号

Jedis jedis=**new** Jedis("192.168.162.128", 6379);

System.***out***.println("获取K1的值："+jedis.get("k1"));

jedis.msetnx("k11","v12","k22","v22","k33","v33");

System.***out***.println(jedis.mget("k11","k22","k33"));

//关闭连接

jedis.close();

}

### 3.3 测试list

@Test

**public** **void** testList() {

//连接指定的redis，需要ip地址和端口号

Jedis jedis=**new** Jedis("192.168.162.128", 6379);

jedis.lpush("mylist", "1","2","3","4");

List<String> list = jedis.lrange("mylist", 0, -1);

**for** (String element : list) {

System.***out***.println(element);

}

//关闭连接

jedis.close();

}

### 3.5 测试set

@Test

**public** **void** testSet() {

//连接指定的redis，需要ip地址和端口号

Jedis jedis=**new** Jedis("192.168.162.128", 6379);

//添加元素

jedis.sadd("mySet", "Jack","Marry","Tom","Tony");

//删除指定元素

jedis.srem("mySet", "Tony");

//获取指定key的元素

Set<String> smembers = jedis.smembers("mySet");

**for** (String member : smembers) {

System.***out***.println(member);

}

//关闭连接

jedis.close();

}

### 3.5 测试hash

@Test

**public** **void** testHash() {

//连接指定的redis，需要ip地址和端口号

Jedis jedis=**new** Jedis("192.168.162.128", 6379);

jedis.hset("myHash", "username", "Jack");

jedis.hset("myHash", "password", "123123");

jedis.hset("myHash", "age", "11");

//将多个数据封装为一个map

Map<String, String> map=**new** HashMap<String, String>();

map.put("gender", "male");

map.put("department", "研发部");

//批量设置多个数据

jedis.hmset("myHash", map);

List<String> values = jedis.hmget("myHash","username","password");

**for** (String val : values) {

System.***out***.println(val);

}

//关闭连接

jedis.close();

}

### 3.6 测试zset

@Test

**public** **void** testZset() {

//连接指定的redis，需要ip地址和端口号

Jedis jedis=**new** Jedis("192.168.162.128", 6379);

jedis.zadd("myZset", 100, "math");

//将多个数据封装为一个map

Map<String, Double> subject=**new** HashMap<String, Double>();

subject.put("chinese", 88d);

subject.put("english", 86d);

//批量添加数据

jedis.zadd("myZset", subject);

Set<String> zset = jedis.zrange("myZset", 0, -1);

**for** (String val : zset) {

System.***out***.println(val);

}

//关闭连接

jedis.close();

}

## 4. 使用连接池

连接池的好处：节省每次连接redis服务带来的消耗，将创建好的连接实例反复利用；

### 4.1 连接池常用参数

|  |  |
| --- | --- |
| 参数 | 含义 |
| MaxTotal | 控制一个pool可分配多少个jedis实例，通过pool.getResource()来获取；如果赋值为-1，则表示不限制；如果pool已经分配了MaxTotal个jedis实例，则此时pool的状态为exhausted。 |
| maxIdle | 控制一个pool最多有多少个状态为idle(空闲)的jedis实例。 |
| MaxWaitMillis | 表示当borrow一个jedis实例时，最大的等待毫秒数，如果超过等待时间，则直接抛JedisConnectionException。 |
| testOnBorrow | 获得一个jedis实例的时候是否检查连接可用性（ping()）；如果为true，则得到的jedis实例均是可用的。 |

### 4.2 示例

@Test

**public** **void** testPool(){

//默认的连接池配置

GenericObjectPoolConfig poolConfig = **new** GenericObjectPoolConfig();

System.***out***.println(poolConfig);

JedisPool jedisPool=**new** JedisPool(poolConfig, "192.168.4.128", 6379,60000);

Jedis jedis = jedisPool.getResource();

String ping = jedis.ping();

System.***out***.println(ping);

//如果是从连接池中获取的，那么执行close方法只是将连接放回到池中

jedis.close();

jedisPool.close();}

# 九、案例

## 1. 手机验证码

需求：

①输入手机号，点击发送后随机生成6位数字码，2分钟有效

②输入验证码，点击验证，返回成功或失败

③每个手机号每天只能输入3次

## 2. 秒杀

### 2.1 实现思路



### 2.2 数据库连接池：

**public** **class** JedisPoolUtil {

**private** **static** **volatile** JedisPool *jedisPool* = **null**;

**private** JedisPoolUtil() {

}

**public** **static** JedisPool getJedisPoolInstance() {

**if** (**null** == *jedisPool*) {

**synchronized** (JedisPoolUtil.**class**) {

**if** (**null** == *jedisPool*) {

JedisPoolConfig poolConfig = **new** JedisPoolConfig();

poolConfig.setMaxTotal(200);//最大连接数

poolConfig.setMaxIdle(32);//最大空闲连接数

//获取连接时的最大等待毫秒数,如果超时就抛异常,默认-1

poolConfig.setMaxWaitMillis(100\*1000);

//连接耗尽时是否阻塞, false报异常,ture阻塞直到超时, 默认true

poolConfig.setBlockWhenExhausted(**true**);

//在获取连接的时候检查有效性, 默认false

poolConfig.setTestOnBorrow(**true**);

*jedisPool* = **new** JedisPool(poolConfig, "192.168.162.128", 6379, 60000 );

}

}

}

**return** *jedisPool*;

}

}

### 2.3. 使用ab进行压力测试

ab -c 10 -n 100 -p /root/postarg -T application/x-www-form-urlencoded <http://192.168.107.1:8080/seckill/doseckill>

参数含义：

-c:模拟多少个客户端

-n:测试几次

-T:内容类型

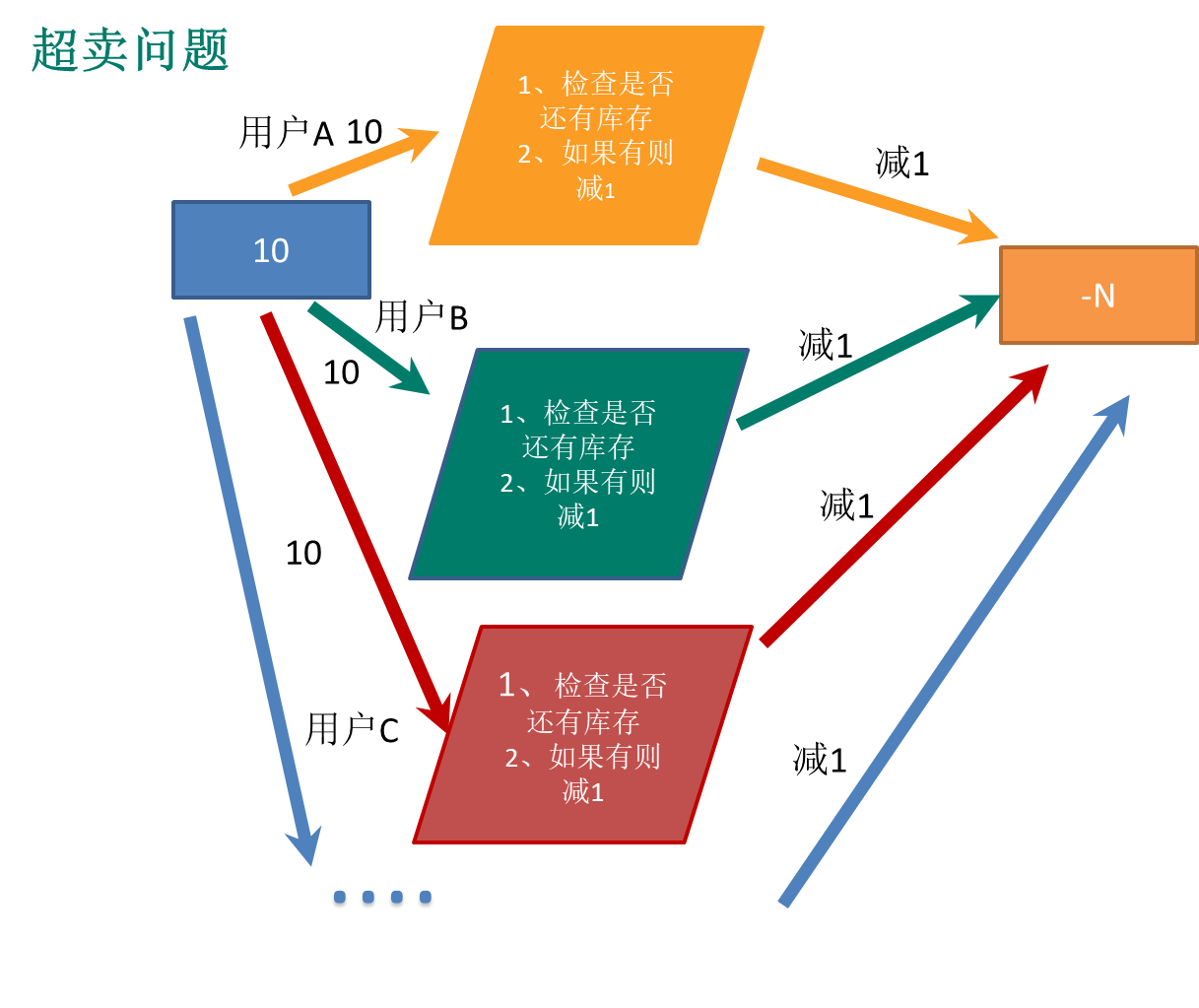
-P:参数文件（注意参数文件的参数要以&符号结尾，注意携带绝对路径）

postarg文件：

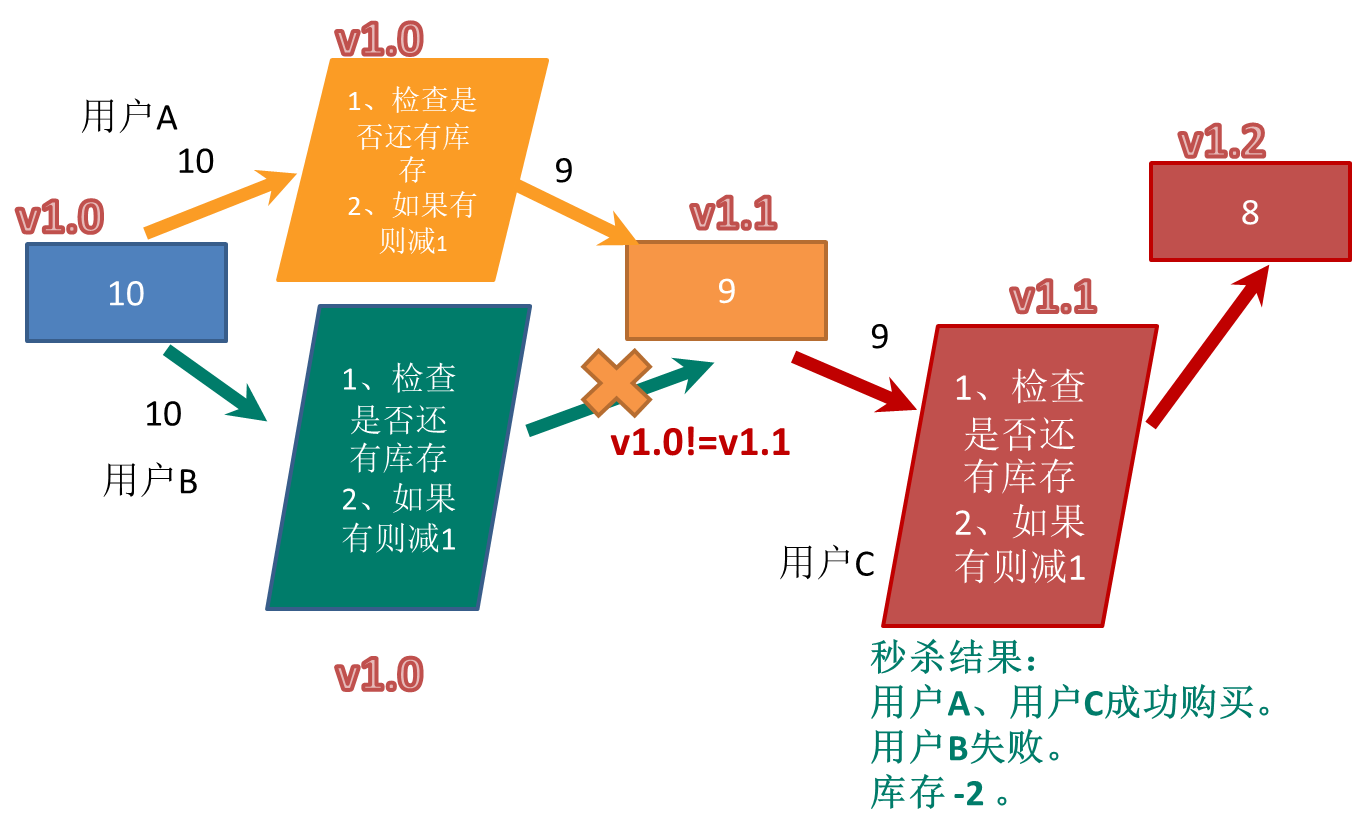
prodid=1001&

### 2.4 超卖问题





解决：使用乐观锁解决超卖！



### 2.5 秒杀富余问题

#### 2.5.1 压测脚本

ab -c 200 -n 2000 -p /root/postarg -T application/x-www-form-urlencoded http://192.168.107.1:8080/seckill/doseckill

#### 2.5.2 如何解决

使用lua脚本解决!

#### 2.5.3 Lua

Lua 是一个小巧的脚本语言，Lua脚本可以很容易的被C/C++ 代码调用，也可以反过来调用C/C++的函数，Lua并没有提供强大的库，一个完整的Lua解释器不过200k，所以Lua不适合作为开发独立应用程序的语言，而是作为嵌入式脚本语言



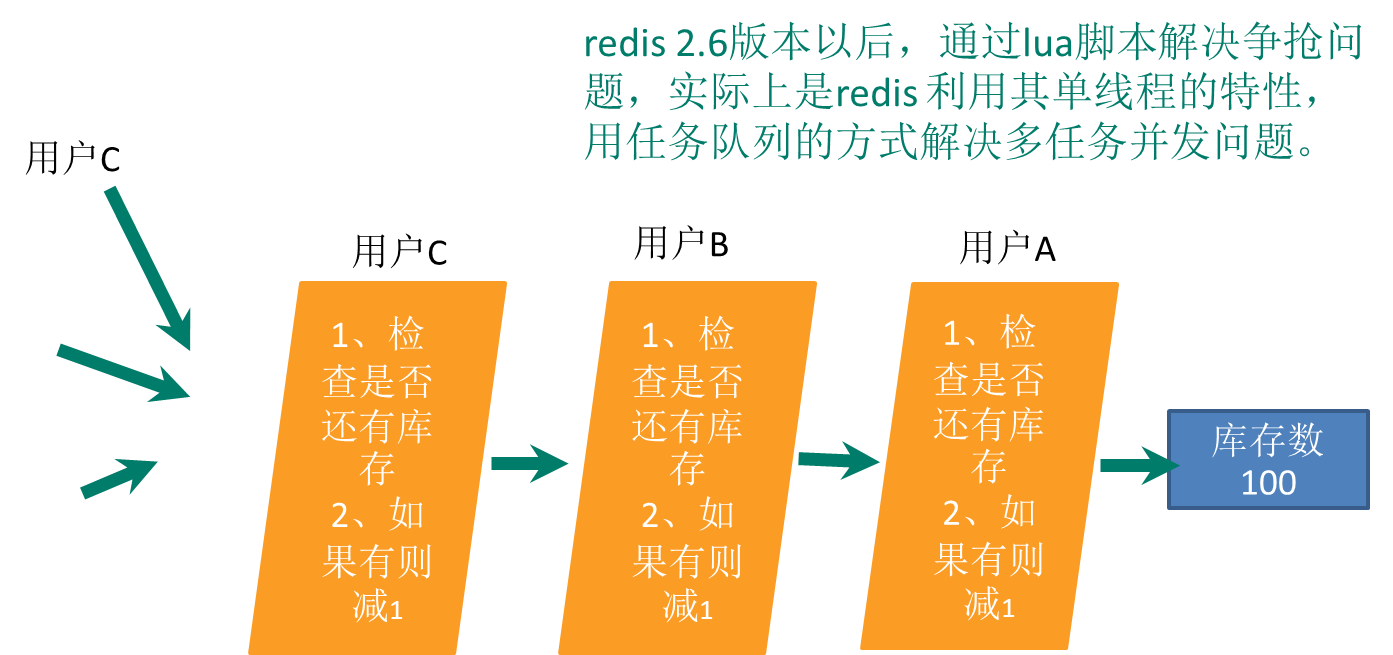
很多应用程序、游戏使用LUA作为自己的嵌入式脚本语言，以此来实现可配置性、可扩展性。这其中包括魔兽争霸地图、魔兽世界、博德之门、愤怒的小鸟等众多游戏插件或外挂。

LUA脚本在Redis中的优势:

将复杂的或者多步的redis操作，写为一个脚本，一次提交给redis执行，减少反复连接redis的次数。提升性能。

LUA脚本是类似redis事务，有一定的原子性，不会被其他命令插队，可以完成一些redis事务性的操作。

但是注意redis的lua脚本功能，只有在2.6以上的版本才可以使用。



# 十、Redis Cluster

## 1. 引入集群

问题：

* 容量不够，redis如何进行扩容？
* 并发写操作， redis如何分摊？

什么是集群：

Redis 集群实现了对Redis的水平扩容，即启动N个redis节点，将整个数据库分布存储在这N个节点中，每个节点存储总数据的1/N。

Redis 集群通过分区（partition）来提供一定程度的可用性（availability）： 即使集群中有一部分节点失效或者无法进行通讯， 集群也可以继续处理命令请求。

## 2. 创建集群

### 2.1 安装ruby环境：

本身redis集群的安装是很麻烦了，通过ruby工具，可以非常方便的将一系列命令打包为一个脚本！

**依次执行**在安装光盘下的Package目录(/media/CentOS\_6.8\_Final/Packages)下的rpm包：

|  |
| --- |
| rpm -ivh compat-readline5-5.2-17.1.el6.x86\_64.rpm |
| rpm -ivh ruby-libs-1.8.7.374-4.el6\_6.x86\_64.rpm |
| rpm -ivh ruby-1.8.7.374-4.el6\_6.x86\_64.rpm |
| rpm -ivh ruby-irb-1.8.7.374-4.el6\_6.x86\_64.rpm |
| rpm -ivh ruby-rdoc-1.8.7.374-4.el6\_6.x86\_64.rpm |
| rpm -ivh rubygems-1.3.7-5.el6.noarch.rpm |

也可以在联网状态下，执行yum安装，执行yum -y install ruby;

之后安装rubygem,rubygem是ruby的包管理框架。yum -y install rubygems

### 2.2 安装redis gem

redis-3.2.0.gem是一个通过ruby操作redis的插件！

拷贝redis-3.2.0.gem到/opt目录下，在opt目录下执行 gem install --local redis-3.2.0.gem；

### 2.3 制作6个redis配置文件

端口号分别是：6379,6380,6381,6389,6390,6391

注意：每个配置文件中需要指定

daemonize yes: 服务在后台运行

port：端口号

pidfile:pid保存文件

logfile：日志文件(如果没有指定的话，就不需要)

dump.rdb: RDB备份文件的名称

appendonly 关掉，或者是更改appendonly文件的名称。

cluster-enabled yes 打开集群模式

cluster-config-file nodes-6379.conf 设定节点配置文件名

cluster-node-timeout 15000 设定节点失联时间，超过该时间（毫秒），集群自动进行主从切换。

样例：

include /usr/local/myredis/rediscluster/redis\_base.conf

pidfile "/var/run/redis\_6379.pid"

port 6379

dbfilename "dump\_6379.rdb"

cluster-enabled yes

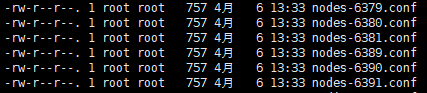
cluster-config-file nodes-6379.conf

cluster-node-timeout 15000

注意在创建集群的时候，初始化的时候，把所有节点的dump文件全部删掉。

### 2.4 开启集群

①首先依次启动6个节点，启动后，会在当前文件夹生成nodes-xxxx.conf文件



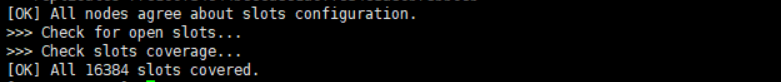
②配置集群

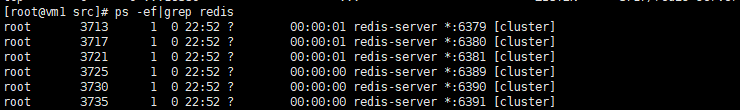
在/opt/redis-3.2.5/src目录下，执行命令：

./redis-trib.rb create --replicas 1 192.168.31.211:6379 192.168.31.211:6380 192.168.31.211:6381

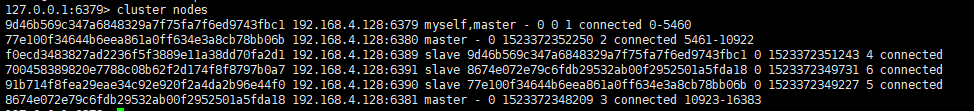
192.168.31.211:6389 192.168.31.211:6390 192.168.31.211:6391

注意，此处不要用127.0.0.1，请用真实IP地址！



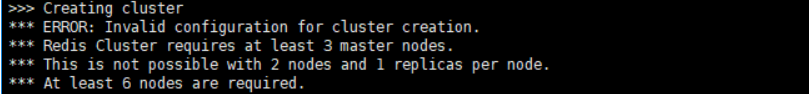


③之后登录到客户端，通过 cluster nodes 命令查看集群信息



④6个节点，为什么是三主三从？

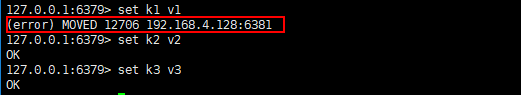
配置机器，至少需要6个节点，否则会报错：



命令create,代表创建一个集群。参数--replicas 1 表示我们希望为集群中的每个主节点创建一个从节点。一个集群至少要有**三个主节点**，分配原则尽量保证每个主数据库运行在不同的IP地址，每个从库和主库不在一个IP地址上。

### 2.5 slot

进入集群后，如果我们，直接写入数据，可能会看到报错信息：



这是因为，集群中多了slot(插槽)的设计。一个 Redis 集群包含 16384 个插槽（hash slot）， 数据库中的每个键都属于这 16384 个插槽的其中一个， 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插槽。 举个例子， 如果一个集群可以有主节点， 其中：

节点 A 负责处理 0 号至 5500 号插槽。

节点 B 负责处理 5501 号至 11000 号插槽。

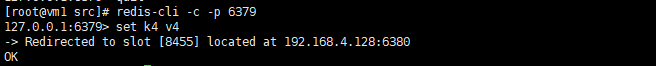
节点 C 负责处理 11001 号至 16383 号插槽。

### 2.6 集群中写入数据

#### 2.6.1 客户端重定向

①在redis-cli每次录入、查询键值，redis都会计算出该key应该送往的插槽，如果不是该客户端对应服务器插槽，redis会报错，并告知应前往的redis实例地址和端口。

②redis-cli客户端提供了 –c 参数实现自动重定向。如 redis-cli -c –p 6379 登入后，再录入、查询键值对可以自动重定向。



③每个slot可以存储一批键值对。

#### 2.6.2 如何多键操作

采用哈希算法后，会自动地分配slot，而 不在一个slot下的键值，是不能使用mget,mset等多键操作。



如果有需求，需要将一批业务数据一起插入呢？

解决：可以通过{}来定义组的概念，从而使key中{}内相同内容的键值对放到一个slot中去。



### 2.7 集群中读取数据

* CLUSTER KEYSLOT <key> 计算键 key 应该被放置在哪个槽上



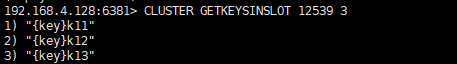
* CLUSTER COUNTKEYSINSLOT <slot> 返回槽 slot 目前包含的键值对数量



* CLUSTER KEYSLOT <key>:计算key应该放在哪个槽



* CLUSTER GETKEYSINSLOT <slot> <count> 返回 count 个 slot 槽中的键。

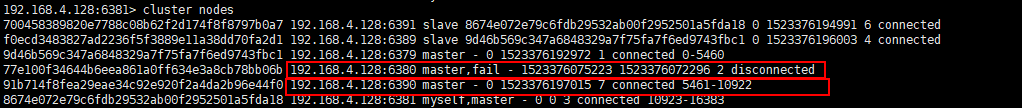


### 2.8 集群中故障恢复

问题1：如果主节点下线？从节点能否自动升为主节点？

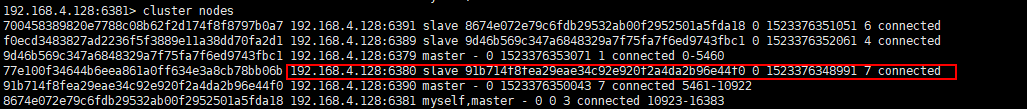
答：主节点下线，从节点自动升为主节点。





问题2：主节点恢复后，主从关系会如何？

主节点恢复后，主节点变为从节点！



问题3：如果所有某一段插槽的主从节点都宕掉，redis服务是否还能继续?

答：服务是否继续，可以通过redis.conf中的cluster-require-full-coverage参数(默认关闭)进行控制。

主从都宕掉，意味着有一片数据，会变成真空，没法再访问了！

如果无法访问的数据，是连续的业务数据，我们需要停止集群，避免缺少此部分数据，造成整个业务的异常。此时可以通过配置cluster-require-full-coverage为yes.

如果无法访问的数据，是相对独立的，对于其他业务的访问，并不影响，那么可以继续开启集群体提供服务。此时，可以配置cluster-require-full-coverage为no。

### 2.9 集群的Jedis开发

@Test

**public** **void** testCluster(){

Set<HostAndPort> jedisClusterNodes = **new** HashSet<HostAndPort>();

//Jedis Cluster will attempt to discover cluster nodes automatically

jedisClusterNodes.add(**new** HostAndPort("192.168.4.128", 6379));

JedisCluster jc = **new** JedisCluster(jedisClusterNodes);

jc.set("foo", "bar");

String value = jc.get("foo");

}

### 2.10 集群的优缺点

优点：

* 实现扩容
* 分摊压力
* 无中心配置相对简单

缺点：

* 多键操作是不被支持的
* 多键的**Redis事务是**不被支持的。lua脚本不被支持。
* 由于集群方案出现较晚，很多公司已经采用了其他的集群方案，而代理或者客户端分片的方案想要迁移至redis cluster，需要整体迁移而不是逐步过渡，复杂度较大。