Redis详解

---- 老孙

课程目标：

1、Redis概述

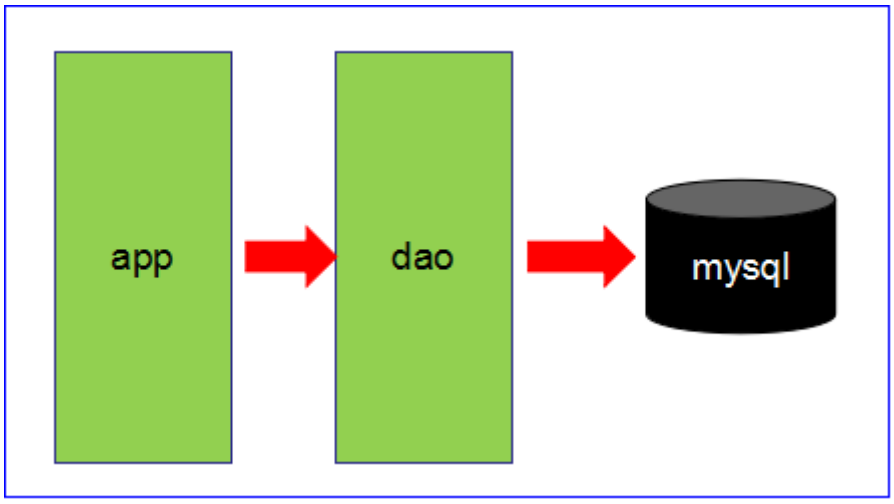
2、下载与安装

3、使用Redis

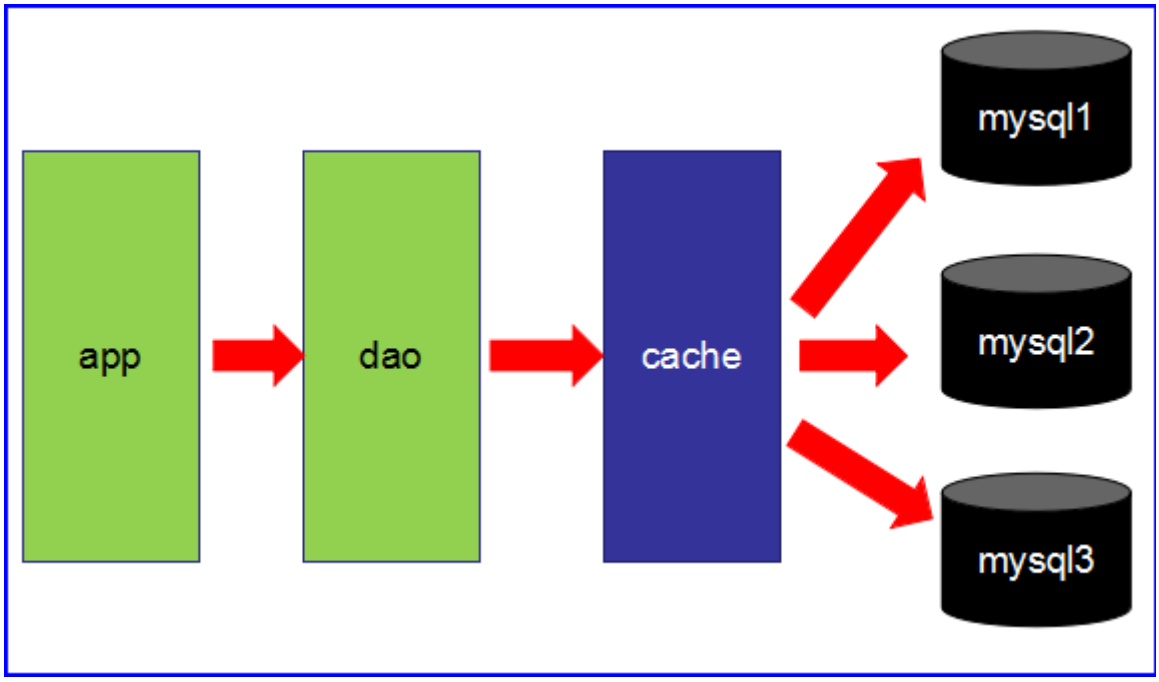
1. 概述

1.1 互联网架构的演变历程

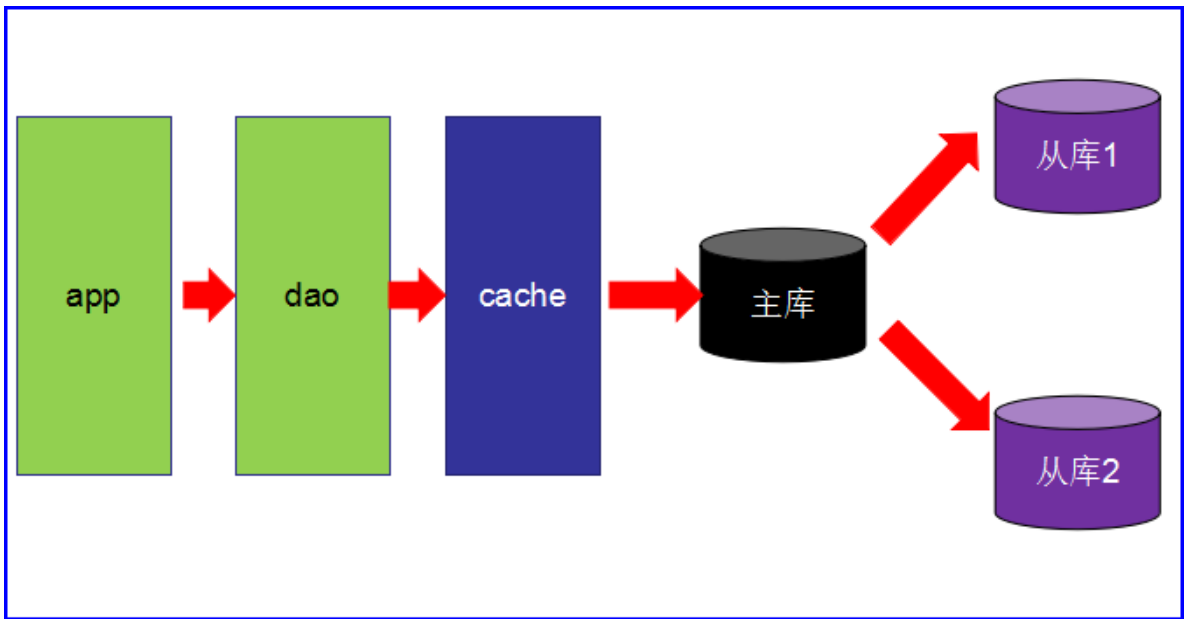
- 第1阶段
  - 数据访问量不大，简单的架构即可搞定！



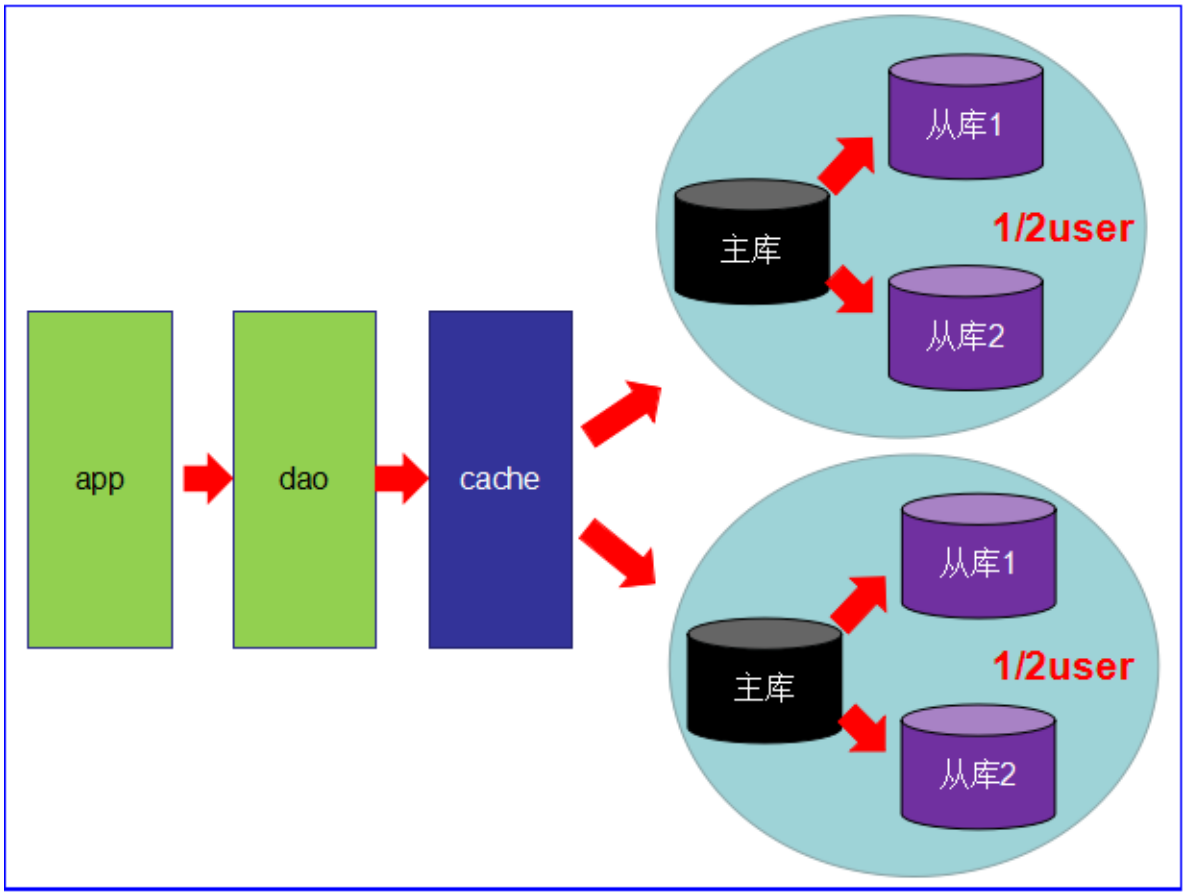
- 第2阶段
  - 数据访问量大，使用缓存技术来缓解数据库的压力。
  - 不同的业务访问不同的数据库



- 第3阶段
  - 主从读写分离。
  - 之前的缓存确实能够缓解数据库的压力，但是写和读都集中在一个数据库上，压力又来了。
  - 一个数据库负责写，一个数据库负责读。分工合作。愉快！
  - 让master（主数据库）来响应事务性**（增删改）**操作，让slave（从数据库）来响应非事务性**（查询）**操作，然后再采用主从复制来把master上的事务性操作同步到slave数据库中
  - mysql的master/slave就是网站的标配！



- 第4阶段
  - mysql的主从复制，读写分离的基础上，mysql的主库开始出现瓶颈
  - 由于MyISAM使用表锁，所以并发性能特别差
  - 分库分表开始流行，mysql也提出了表分区，虽然不稳定，但我们看到了希望
  - 开始吧，mysql集群



1.2 Redis入门介绍

- 互联网需求的3高
  - 高并发，高可扩，高性能
- Redis 是一种运行速度很快，并发性能很强，并且运行在内存上的NoSql（not only sql）数据库
- NoSQL数据库 和 传统数据库 相比的优势
  - 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，增加字段简直就是一个噩梦
- **Redis的常用使用场景**
  - **缓存**，毫无疑问这是Redis当今最为人熟知的使用场景。在提升服务器性能方面非常有效；一些频繁被访问的数据，经常被访问的数据如果放在关系型数据库，每次查询的开销都会很大，而放在redis中，因为redis 是放在内存中的可以很高效的访问
  - **排行榜**，在使用传统的关系型数据库（mysql oracle 等）来做这个事儿，非常的麻烦，而利用Redis的SortSet(有序集合)数据结构能够简单的搞定；
  - **计算器/限速器**，利用Redis中原子性的自增操作，我们可以统计类似用户点赞数、用户访问数等，这类操作如果用MySQL，频繁的读写会带来相当大的压力；限速器比较典型的使用场景是限制某个用户访问某个API的频率，常用的有抢购时，防止用户疯狂点击带来不必要的压力；
  - **好友关系**，利用集合的一些命令，比如求交集、并集、差集等。可以方便搞定一些共同好友、共同爱好之类的功能；
  - **简单消息队列**，除了Redis自身的发布/订阅模式，我们也可以利用List来实现一个队列机制，比如：到货通知、邮件发送之类的需求，不需要高可靠，但是会带来非常大的DB压力，完全可以用List来完成异步解耦；

- **Session共享**，以jsp为例，默认Session是保存在服务器的文件中，如果是集群服务，同一个用户过来可能落在不同机器上，这就会导致用户频繁登陆；采用Redis保存Session后，无论用户落在那台机器上都能够获取到对应的Session信息。

1.3 Redis/Memcache/MongoDB对比

都是nosql数据库的著名代表

1.3.1 Redis和Memcache

- Redis和Memcache都是内存数据库。不过memcache还可用于缓存其他东西，例如图片、视频等等。
- memcache 数据结构单一kv，redis 更丰富一些，还提供 list，set， hash 等数据结构的存储，有效的减少网络 IO 的次数
- 虚拟内存–Redis当物理内存用完时，可以将一些很久没用到的value交换到磁盘
- **存储数据安全**–memcache挂掉后，数据没了（没有持久化机制）；redis可以定期保存到磁盘（持久化）
- 灾难恢复–memcache挂掉后，数据不可恢复; redis数据丢失后可以通过RBD或AOF恢复

1.3.2 Redis和MongoDB

- redis和mongodb并不是竞争关系，更多的是一种**协作共存**的关系。
- mongodb本质上还是硬盘数据库，在复杂查询时仍然会有大量的资源消耗，而且在处理复杂逻辑时仍然要不可避免地进行多次查询。
- 这时就需要redis或Memcache这样的内存数据库来作为中间层进行缓存和加速。
- 比如在某些复杂页面的场景中，整个页面的内容如果都从mongodb中查询，可能要几十个查询语句，耗时很长。如果需求允许，则可以把整个页面的对象缓存至redis中，定期更新。这样mongodb和redis就能很好地协作起来

1.4 分布式数据库CAP原理

1.4.1 CAP简介

- 传统的关系型数据库事务具备ACID：
  - A：原子性
  - C：一致性
  - I：独立性
  - D：持久性
- 分布式数据库的CAP:
  - C（Consistency）：强一致性
    - “all nodes see the same data at the same time”,即更新操作成功并返回客户端后，**所有节点在同一时间的数据完全一致**，这就是分布式的一致性。一致性的问题在并发系统中不可避免，对于客户端来说，一致性指的是并发访问时更新过的数据如何获取的问题。从服务端来看，则是更新如何复制分布到整个系统，以保证数据最终一致。
  - A（Availability）：高可用性
    - 可用性指“Reads and writes always succeed”，即**服务一直可用**，而且要是正常的响应时间。好的可用性主要是指系统能够很好的为用户服务，不出现用户操作失败或者访问超时等用户体验不好的情况。

- P（Partition tolerance）：分区容错性
  - 即分布式系统在遇到**某节点或网络分区故障时，仍然能够对外提供满足一致性或可用性的服务**。
  - 分区容错性要求能够使应用虽然是一个分布式系统，而看上去却好像是在一个可以运转正常的整体。比如现在的分布式系统中有某一个或者几个机器宕掉了，其他剩下的机器还能够正常运转满足系统需求，对于用户而言并没有什么体验上的影响。

1.4.2 CAP理论

- CAP理论提出就是针对分布式数据库环境的，所以，P这个属性必须容忍它的存在，而且是必须具备的。
- 因为P是必须的，那么我们需要选择的就是A和C。
- 大家知道，在分布式环境下，为了保证系统可用性，通常都采取了复制的方式，避免一个节点损坏，导致系统不可用。那么就出现了每个节点上的数据出现了很多个副本的情况，而数据从一个节点复制到另外的节点时需要时间和要求网络畅通的，所以，当P发生时，也就是无法向某个节点复制数据时，这时候你有两个选择：
  - 选择可用性 A，此时，那个失去联系的节点依然可以向系统提供服务，不过它的数据就不能保证是同步的了（失去了C属性）。
  - 选择一致性C，为了保证数据库的一致性，我们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，这时候系统处于不可用状态(失去了A属性)。
- 最常见的例子是读写分离，某个节点负责写入数据，然后将数据同步到其它节点，其它节点提供读取的服务，当两个节点出现通信问题时，你就面临着选择A（继续提供服务，但是数据不保证准确），C（用户处于等待状态，一直等到数据同步完成）。

1.4.3 CAP总结

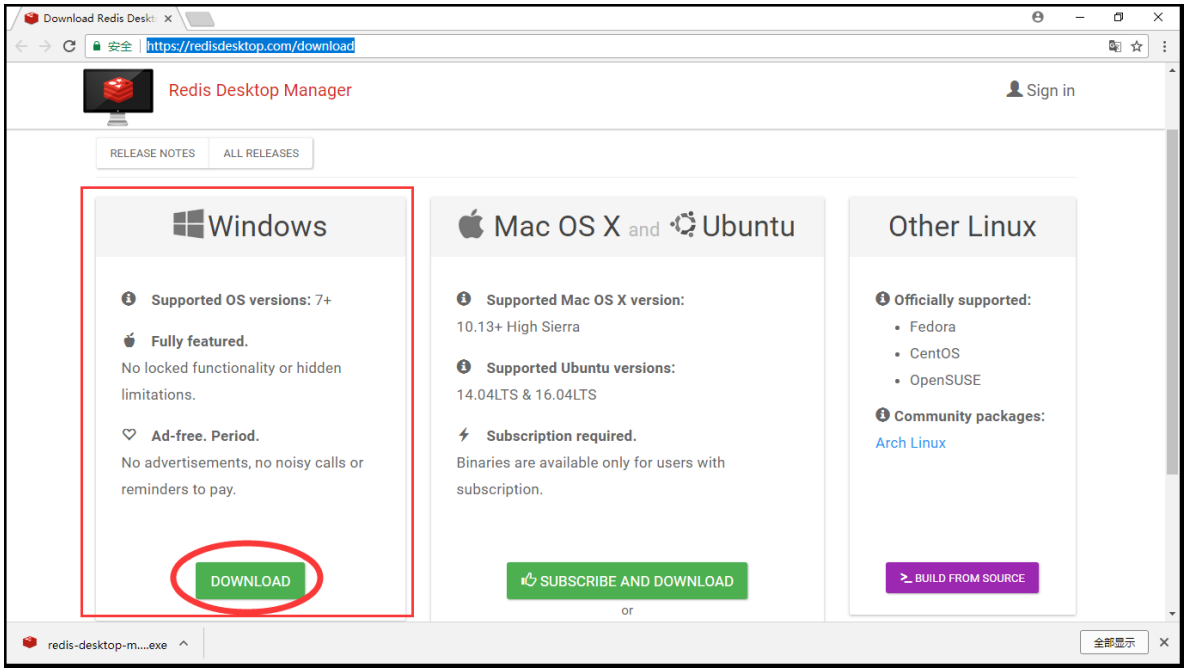
- 分区是常态，不可避免，三者不可共存
- **可用性和一致性是一对冤家**
  - 一致性高，可用性低
  - 一致性低，可用性高
- 因此，根据 CAP 原理将 NoSQL 数据库分成了满足 CA 原则、满足 CP 原则和满足 AP 原则三 大类：
  - CA - 单点集群，满足一致性，可用性的系统，通常在可扩展性上不太强大。
  - CP - 满足一致性，分区容忍性的系统，通常性能不是特别高。
  - AP - 满足可用性，分区容忍性的系统，通常可能对一致性要求低一些。

2. 下载与安装

2.1 下载

- redis：http://www.redis.net.cn/

- 图形工具：https://redisdesktop.com/download



2.2 安装

虽然可以在安装在windows操作系统，但是官方不推荐，所以我们一如既往的安装在linux上

1. 上传tar.gz包，并解压

```
tar -zxvf redis-5.0.4.tar.gz
```

2. 安装gcc（必须有网络）

```
yum -y install gcc
```

忘记是否安装过，可以使用 `gcc -v` 命令查看gcc版本，如果没有安装过，会提示命令不存在

3. 进入redis目录，进行编译

```
make
```

4. 编译之后，开始安装

```
make install
```

2.3 安装后的操作

2.3.1 后台运行方式

- redis默认不会使用后台运行，如果你需要，修改配置文件daemonize=yes，当你后台服务启动的时候，会写成一个进程文件运行。

```
vim /opt/redis-5.0.4/redis.conf
```

```
daemonize yes
```

- 以配置文件的方式启动

```
cd /usr/local/bin
redis-server /opt/redis-5.0.4/redis.conf
```

2.3.2 关闭数据库

- 单实例关闭

```
redis-cli shutdown
```

- 多实例关闭

```
redis-cli -p 6379 shutdown
```

2.3.3 常用操作

- 检测6379端口是否在监听

```
netstat -lntp | grep 6379
```

> 端口为什么是6379？
>
> 6379在是手机按键上MERZ对应的号码，
> 而MERZ取自意大利歌女Alessia Merz的名字。
> MERZ长期以来被antirez（redis作者）及其朋友当作愚蠢的代名词。

- 检测后台进程是否存在

```
ps -ef|grep redis
```

2.3.4 连接redis并测试

```
redis-cli
ping
```

2.3.5 HelloWorld

```
# 保存数据
set k1 china
# 获取数据
get k1
```

2.3.6 测试性能

- 先 ctrl+c，退出redis客户端

```
redis-benchmark
```

- 执行命令后，命令不会自动停止，需要我们手动ctrl+c停止测试

```
[root@localhost bin]# redis-benchmark
====== PING_INLINE ======
  100000 requests completed in 1.80 seconds    # 1.8秒处理了10万个请求，性能要看笔记本的配置高低
  50 parallel clients
  3 bytes payload
  keep alive: 1

87.69% <= 1 milliseconds
99.15% <= 2 milliseconds
99.65% <= 3 milliseconds
99.86% <= 4 milliseconds
99.92% <= 5 milliseconds
99.94% <= 6 milliseconds
99.97% <= 7 milliseconds
100.00% <= 7 milliseconds
55524.71 requests per second    # 每秒处理的请求数量
```

2.3.7 默认16个数据库

```
vim /opt/redis-5.0.4/redis.conf
```

```
127.0.0.1:6379> get k1                  # 查询k1
"china"
127.0.0.1:6379> select 16               # 切换16号数据库
(error) ERR DB index is out of range    # 数据库的下标超出了范围
127.0.0.1:6379> select 15               # 切换15号数据库
OK
127.0.0.1:6379[15]> get k1              # 查询k1
(nil)
127.0.0.1:6379[15]> select 0            # 切换0号数据库
OK
127.0.0.1:6379> get k1                  # 查询k1
"china"
```

2.3.8 数据库键的数量

```
dbsize
```

redis在linux支持命令补全（tab）

2.3.9 清空数据库

- 清空当前库

```
flushdb
```

- 清空所有（16个）库，慎用！！

```
flushall
```

2.3.10 模糊查询（key）

模糊查询keys命令，有三个通配符：

- *：通配任意多个字符
  - 查询所有的键

    ```
    keys *
    ```

  - 模糊查询k开头，后面随便多少个字符

    ```
    keys k*
    ```

  - 模糊查询e为最后一位，前面随便多少个字符

    ```
    keys *e
    ```

  - 双 * 模式，匹配任意多个字符：查询包含k的键

```
keys *k*
```

- ? : 通配单个字符
  - 模糊查询k字头，并且匹配一个字符

    ```
    keys k?
    ```

  - 你只记得第一个字母是k，他的长度是3

    ```
    keys k??
    ```

- [] : 通配括号内的某一个字符
  - 记得其他字母，就第二个字母可能是a或e

    ```
    keys r[ae]dis
    ```

2.3.11 键（key）

- exists key：判断某个key是否存在

```
127.0.0.1:6379> exists k1
(integer) 1   # 存在
127.0.0.1:6379> exists y1
(integer) 0   # 不存在
```

- move key db：移动（剪切，粘贴）键到几号库

```
127.0.0.1:6379> move x1 8   # 将x1移动到8号库
(integer) 1   # 移动成功
127.0.0.1:6379> exists x1   # 查看当前库中是否存在x1
(integer) 0   # 不存在（因为已经移走了）
127.0.0.1:6379> select 8    # 切换8号库
OK
127.0.0.1:6379[8]> keys *   # 查看当前库中的所有键
1) "x1"
```

- ttl key：查看键还有多久过期（-1永不过期，-2已过期）
- time to live 还能活多久

```
127.0.0.1:6379[8]> ttl x1
(integer) -1    # 永不过期
```

- expire key 秒：为键设置过期时间（生命倒计时）

```
127.0.0.1:6379[8]> set k1 v1        # 保存k1
OK
127.0.0.1:6379[8]> ttl k1           # 查看k1的过期时间
(integer) -1    # 永不过期
127.0.0.1:6379[8]> expire k1 10     # 设置k1的过期时间为10秒（10秒后自动销毁）
(integer) 1     # 设置成功
127.0.0.1:6379[8]> get k1           # 获取k1
"v1"
127.0.0.1:6379[8]> ttl k1           # 查看k1的过期时间
(integer) 2     # 还有2秒过期
127.0.0.1:6379[8]> get k1
(nil)
127.0.0.1:6379[8]> keys *           # 从内存中销毁了
(empty list or set)
```

- type key：查看键的数据类型

```
127.0.0.1:6379[8]> type k1
string          # k1的数据类型是会string字符串
```

3. 使用Redis

3.1 五大数据类型

- 操作文档：http://redisdoc.com/

3.1.1 字符串String

- set/get/del/append/strlen

```
127.0.0.1:6379> set k1 v1       # 保存数据
OK
127.0.0.1:6379> set k2 v2       # 保存数据
OK
127.0.0.1:6379> keys *
1) "k1"
2) "k2"
127.0.0.1:6379> del k2          # 删除数据k2
(integer) 1
127.0.0.1:6379> keys *
1) "k1"
127.0.0.1:6379> get k1          # 获取数据k1
```

```
"v1"
127.0.0.1:6379> append k1 abc   # 往k1的值追加数据abc
(integer) 5                     # 返回值的长度（字符数量）
127.0.0.1:6379> get k1
"v1abc"
127.0.0.1:6379> strlen k1       # 返回k1值的长度（字符数量）
(integer) 5
```

- incr/decr/incrby/decrby：加减操作，操作的必须是数字类型
  - incr：意思是increment，增加
  - decr：意思是decrement，减少

```
127.0.0.1:6379> set k1 1        # 初始化k1的值为1
OK
127.0.0.1:6379> incr k1         # k1自增1（相当于++）
(integer) 2
127.0.0.1:6379> incr k1
(integer) 3
127.0.0.1:6379> get k1
"3"
127.0.0.1:6379> decr k1         # k1自减1（相当于--）
(integer) 2
127.0.0.1:6379> decr k1
(integer) 1
127.0.0.1:6379> get k1
"1"
127.0.0.1:6379> incrby k1 3     # k1自增3（相当于+=3）
(integer) 4
127.0.0.1:6379> get k1
"4"
127.0.0.1:6379> decrby k1 2     # k1自减2（相当于-=2）
(integer) 2
127.0.0.1:6379> get k1
"2"
```

- getrange/setrange：类似between...and...
  - range：范围

```
127.0.0.1:6379> set k1 abcdef       # 初始化k1的值为abcdef
OK
127.0.0.1:6379> get k1
"abcdef"
127.0.0.1:6379> getrange k1 0 -1    # 查询k1全部的值
"abcdef"
127.0.0.1:6379> getrange k1 0 3     # 查询k1的值，范围是下标0~下标3（包含0和3，共
返回4个字符）
"abcd"
127.0.0.1:6379> setrange k1 1 xxx   # 替换k1的值，从下标1开始提供为xxx
(integer) 6
127.0.0.1:6379> get k1
"axxxef"
```

- setex/setnx
  - **set** with **ex**pir：添加数据的同时设置生命周期

```
127.0.0.1:6379> setex k1 5 v1        # 添加k1 v1数据的同时，设置5秒的声明周期
OK
127.0.0.1:6379> get k1
"v1"
127.0.0.1:6379> get k1
(nil)                                # 已过期，k1的值v1自动销毁
```

  - **set** if **n**ot e**x**ist：添加数据的时候判断是否已经存在，防止已存在的数据被覆盖掉

```
127.0.0.1:6379> setnx k1 sun
(integer) 0              # 添加失败，因为k1已经存在
127.0.0.1:6379> get k1
"laosun"
127.0.0.1:6379> setnx k2 sun
(integer) 1              # k2不存在，所以添加成功
```

- mset/mget/msetnx
- m：**more**更多

```
127.0.0.1:6379> set k1 v1 k2 v2              # set不支持一次添加多条数据
(error) ERR syntax error
127.0.0.1:6379> mset k1 v1 k2 v2 k3 v3       # mset可以一次添加多条数据
OK
127.0.0.1:6379> keys *
1) "k1"
2) "k2"
3) "k3"
127.0.0.1:6379> mget k2 k3                   # 一次获取多条数据
1) "v2"
2) "v3"
127.0.0.1:6379> msetnx k3 v3 k4 v4           # 一次添加多条数据时，如果添加的数据中有已经存在的，则失败
(integer) 0
127.0.0.1:6379> msetnx k4 v4 k5 v5           # 一次添加多条数据时，如果添加的数据中都不存在的，则成功
(integer) 1
```

- getset：先get后set

```
127.0.0.1:6379> getset k6 v6
(nil)            # 因为没有k6，所以get为null，然后将k6v6的值添加到数据库
127.0.0.1:6379> keys *
1) "k4"
2) "k1"
3) "k2"
```

```
4) "k3"
5) "k5"
6) "k6"
127.0.0.1:6379> get k6
"v6"
127.0.0.1:6379> getset k6 vv6   # 先获取k6的值，然后修改k6的值为vv6
"v6"
127.0.0.1:6379> get k6
"vv6"
```

3.1.2 列表List

push和pop，类似机枪AK47：push，压子弹，pop，射击出子弹

- lpush/rpush/lrange
  - l：left 自左向右→添加（从上往下添加）
  - r：right 自右向左←添加（从下往上添加）

```
127.0.0.1:6379> lpush list01 1 2 3 4 5          # 从上往下添加
(integer) 5
127.0.0.1:6379> keys *
1) "list01"
127.0.0.1:6379> lrange list01 0 -1              # 查询list01中的全部数据0表示开始，-1表示结尾
1) "5"
2) "4"
3) "3"
4) "2"
5) "1"
127.0.0.1:6379> rpush list02 1 2 3 4 5          # 从下往上添加
(integer) 5
127.0.0.1:6379> lrange list02 0 -1
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
```

- lpop/rpop：移除第一个元素（上左下右）

```
127.0.0.1:6379> lpop list02             # 从左（上）边移除第一个元素
"1"
127.0.0.1:6379> rpop list02             # 从右（下）边移除第一个元素
"5"
```

- lindex：根据下标查询元素（从左向右，自上而下）

```
127.0.0.1:6379> lrange list01 0 -1
1) "5"
2) "4"
3) "3"
4) "2"
5) "1"
127.0.0.1:6379> lindex list01 2     # 从上到下数，下标为2的值
"3"
127.0.0.1:6379> lindex list01 1     # 从上到下数，下标为1的值
"4"
```

- llen：返回集合长度

```
127.0.0.1:6379> llen list01
(integer) 5
```

- lrem：删除n个value

```
127.0.0.1:6379> lpush list01 1 2 2 3 3 3 4 4 4 4
(integer) 10
127.0.0.1:6379> lrem list01 2 3     # 从list01中移除2个3
(integer) 2
127.0.0.1:6379> lrange list01 0 -1
1) "4"
2) "4"
3) "4"
4) "4"
5) "3"
6) "2"
7) "2"
8) "1"
```

- ltrim：截取指定范围的值，别的全扔掉
  - ltrim key begindex endindex

```
127.0.0.1:6379> lpush list01 1 2 3 4 5 6 7 8 9
(integer) 9
127.0.0.1:6379> lrange list01 0 -1
1) "9"  # 下标0
2) "8"  # 下标1
3) "7"  # 下标2
4) "6"  # 下标3
5) "5"  # 下标4
6) "4"  # 下标5
7) "3"  # 下标6
8) "2"  # 下标7
9) "1"  # 下标8
127.0.0.1:6379> ltrim list01 3 6        # 截取下标3~6的值，别的全扔掉
OK
127.0.0.1:6379> lrange list01 0 -1
```

```
1) "6"
2) "5"
3) "4"
4) "3"
```

- rpoplpush：从一个集合搞一个元素到另一个集合中（右出一个，左进一个）

```
127.0.0.1:6379> rpush list01 1 2 3 4 5
(integer) 5
127.0.0.1:6379> lrange list01 0 -1
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
127.0.0.1:6379> rpush list02 1 2 3 4 5
(integer) 5
127.0.0.1:6379> lrange list02 0 -1
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
127.0.0.1:6379> rpoplpush list01 list02     # list01右边出一个，从左进入到
list02的第一个位置
"5"
127.0.0.1:6379> lrange list01 0 -1
1) "1"
2) "2"
3) "3"
4) "4"
127.0.0.1:6379> lrange list02 0 -1
1) "5"
2) "1"
3) "2"
4) "3"
5) "4"
6) "5"
```

- lset：改变某个下标的某个值
- lset key index value

```
127.0.0.1:6379> lrange list02 0 -1
1) "5"
2) "1"
3) "2"
4) "3"
5) "4"
6) "5"
127.0.0.1:6379> lset list02 0 x     # 将list02中下标为0的元素修改成x
OK
127.0.0.1:6379> lrange list02 0 -1
1) "x"
```

```
2) "1"
3) "2"
4) "3"
5) "4"
6) "5"
```

- linsert：插入元素（指定某个元素之前/之后）
- linsert key before/after oldvalue newvalue

```
127.0.0.1:6379> lrange list02 0 -1
1) "x"
2) "1"
3) "2"
4) "3"
5) "4"
6) "5"
127.0.0.1:6379> linsert list02 before 2 java        # 从左边进入，在list02中的2元素之前插入java
(integer) 7
127.0.0.1:6379> lrange list02 0 -1
1) "x"
2) "1"
3) "java"
4) "2"
5) "3"
6) "4"
7) "5"
127.0.0.1:6379> linsert list02 after 2 redis        # 从左边进入，在list02中的2元素之后插入redis
(integer) 8
127.0.0.1:6379> lrange list02 0 -1
1) "x"
2) "1"
3) "java"
4) "2"
5) "redis"
6) "3"
7) "4"
8) "5"
```

- 性能总结：类似添加火车皮一样，头尾操作效率高，中间操作效率惨；

3.1.3 集合Set

和java中的set特点类似，不允许重复

- sadd/smembers/sismember：添加/查看/判断是否存在

```
127.0.0.1:6379> sadd set01 1 2 2 3 3 3  # 添加元素（自动排除重复元素）
(integer) 3
127.0.0.1:6379> smembers set01          # 查询set01集合
1) "1"
2) "2"
3) "3"
127.0.0.1:6379> sismember set01 2
(integer) 1     # 存在
127.0.0.1:6379> sismember set01 5
(integer) 0     # 不存在
```

  - 注意：1和0可不是下标，而是布尔。1：true存在，2：false不存在

- scard：获得集合中的元素个数

```
127.0.0.1:6379> scard set01
(integer) 3     # 集合中有3个元素
```

- srem：删除集合中的元素
  - srem key value

```
127.0.0.1:6379> srem set01 2        # 移除set01中的元素2
(integer) 1     # 1表示移除成功
```

- srandmember：从集合中随机获取几个元素
  - srandmember 整数（个数）

```
127.0.0.1:6379> sadd set01 1 2 3 4 5 6 7 8 9
(integer) 9
127.0.0.1:6379> smembers set01
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
6) "6"
7) "7"
8) "8"
9) "9"
127.0.0.1:6379> srandmember set01 3     # 从set01中随机获取3个元素
1) "8"
2) "2"
3) "3"
127.0.0.1:6379> srandmember set01 5     # 从set01中随机获取5个元素
1) "5"
2) "8"
3) "7"
4) "4"
5) "6"
```

- spop：随机出栈（移除）

```
127.0.0.1:6379> smembers set01
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
6) "6"
7) "7"
8) "8"
9) "9"
127.0.0.1:6379> spop set01      # 随机移除一个元素
"8"
127.0.0.1:6379> spop set01      # 随机移除一个元素
"7"
```

- smove：移动元素：将key1某个值赋值给key2

```
127.0.0.1:6379> sadd set01 1 2 3 4 5
(integer) 5
127.0.0.1:6379> sadd set02 x y z
(integer) 3
127.0.0.1:6379> smove set01 set02 3     # 将set01中的元素3移动到set02中
(integer) 1     # 移动成功
```

- 数学集合类
  - 交集：sinter
  - 并集：sunion
  - 差集：sdiff

```
127.0.0.1:6379> sadd set01 1 2 3 4 5
(integer) 5
127.0.0.1:6379> sadd set02 2 a 1 b 3
(integer) 5
127.0.0.1:6379> sinter set01 set02      # set01和set02共同存在的元素
1) "1"
2) "2"
3) "3"
127.0.0.1:6379> sunion set01 set02      # 将set01和set02中所有元素合并起来（排除重复的）
1) "5"
2) "4"
3) "3"
4) "2"
5) "b"
6) "a"
7) "1"
127.0.0.1:6379> sdiff setr01 set02
(empty list or set)
127.0.0.1:6379> sdiff set01 set02       # 在set01中存在，在set02中不存在
```

```
1) "4"
2) "5"
127.0.0.1:6379> sdiff set02 set01       # 在set02中存在，在set01中不存在
1) "b"
2) "a"
```

3.1.4 哈希Hash

类似java里面的Map<String,Object>

KV模式不变，但V是一个键值对

- **hset/hget/hmset/hmget/hgetall/hdel**：添加/得到/多添加/多得到/得到全部/删除属性

```
127.0.0.1:6379> hset user id 1001           # 添加user，值为id=1001
(integer) 1
127.0.0.1:6379> hget user
(error) ERR wrong number of arguments for 'hget' command
127.0.0.1:6379> hget user id                # 查询user，必须指明具体的字段
"1001"
127.0.0.1:6379> hmset student id 101 name tom age 22    # 添加学生student，属性一堆
OK
127.0.0.1:6379> hget student name           # 获取学生名字
"tom"
127.0.0.1:6379> hmget student name age      # 获取学生年龄
1) "tom"
2) "22"
127.0.0.1:6379> hgetall student             # 获取学生全部信息
1) "id"
2) "101"
3) "name"
4) "tom"
5) "age"
6) "22"
127.0.0.1:6379> hdel student age            # 删除学生年龄属性
(integer) 1     # 删除成功
127.0.0.1:6379> hgetall student
1) "id"
2) "101"
3) "name"
4) "tom"
```

- hlen：返回元素的属性个数

```
127.0.0.1:6379> hgetall student
1) "id"
2) "101"
3) "name"
4) "tom"
127.0.0.1:6379> hlen student
(integer) 2     # student属性的数量，id和name，共两个属性
```

- hexists：判断元素是否存在某个属性

```
127.0.0.1:6379> hexists student name        # student中是否存在name属性
(integer) 1     # 存在
127.0.0.1:6379> hexists student age         # student中是否存在age属性
(integer) 0     # 不存在
```

- hkeys/hvals：获得属性的所有key/获得属性的所有value

```
127.0.0.1:6379> hkeys student       # 获取student所有的属性名
1) "id"
2) "name"
127.0.0.1:6379> hvals student       # 获取student所有属性的值（内容）
1) "101"
2) "tom"
```

- hincrby/hincrbyfloat：自增（整数）/自增（小数）

```
127.0.0.1:6379> hmset student id 101 name tom age 22
OK
127.0.0.1:6379> hincrby student age 2           # 自增整数2
(integer) 24
127.0.0.1:6379> hget student age
"24"
127.0.0.1:6379> hmset user id 1001 money 1000
OK
127.0.0.1:6379> hincrbyfloat user money 5.5     # 自增小数5.5
"1005.5"
127.0.0.1:6379> hget user money
"1005.5"
```

- hsetnx：添加的时候，先判断是否存在

```
127.0.0.1:6379> hsetnx student age 18   # 添加时，判断age是否存在
(integer) 0     # 添加失败，因为age已存在
127.0.0.1:6379> hsetnx student sex 男     # 添加时，判断sex是否存在
(integer) 1     # 添加成功，因为sex不存在
127.0.0.1:6379> hgetall student
1) "id"
2) "101"
3) "name"
4) "tom"
5) "age"
6) "24"
7) "sex"
8) "\xe7\x94\xb7"       # 可以添加中文，但是显示为乱码（后期解决）
```

3.1.5 有序集合Zset

真实需求：

充10元可享vip1；

充20元可享vip2；

充30元可享vip3；

以此类推...

- zadd/zrange（ withscores ）：添加/查询

```
127.0.0.1:6379> zadd zset01 10 vip1 20 vip2 30 vip3 40 vip4 50 vip5
(integer) 5
127.0.0.1:6379> zrange zset01 0 -1          # 查询数据
1) "vip1"
2) "vip2"
3) "vip3"
4) "vip4"
5) "vip5"
127.0.0.1:6379> zrange zset01 0 -1 withscores   # 带着分数查询数据
 1) "vip1"
 2) "10"
 3) "vip2"
 4) "20"
 5) "vip3"
 6) "30"
 7) "vip4"
 8) "40"
 9) "vip5"
10) "50"
```

- zrangebyscore：模糊查询
  - (： 不包含
  - limit：跳过几个截取几个

```
127.0.0.1:6379> zrangebyscore zset01 20 40              # 20 <= score <= 40
1) "vip2"
2) "vip3"
3) "vip4"
127.0.0.1:6379> zrangebyscore zset01 20 (40             # 20 <= score < 40
1) "vip2"
2) "vip3"
127.0.0.1:6379> zrangebyscore zset01 (20 (40            # 20 < score < 40
1) "vip3"
127.0.0.1:6379> zrangebyscore zset01 10 40 limit 2 2    # 10 <= score <=
40，共返回四个，跳过前2个，取2个
1) "vip3"
2) "vip4"
127.0.0.1:6379> zrangebyscore zset01 10 40 limit 2 1    # 20 <= score <=
40，共返回四个，跳过前2个，取1个
1) "vip3"
```

- zrem：删除元素

```
127.0.0.1:6379> zrem zset01 vip5        # 移除vip5
(integer) 1
```

- zcard/zcount/zrank/zscore：集合长度/范围内元素个数/得元素下标/通过值得分数

```
127.0.0.1:6379> zcard zset01            # 集合中元素的个数
(integer) 4
127.0.0.1:6379> zcount zset01 20 30     # 分数在20~40之间，共有几个元素
(integer) 2
127.0.0.1:6379> zrank zset01 vip3       # vip3在集合中的下标（从上向下）
(integer) 2
127.0.0.1:6379> zscore zset01 vip2      # 通过元素获得对应的分数
"20"
```

- zrevrank：逆序找下标（从下向上）

```
127.0.0.1:6379> zrevrank zset01 vip3
(integer) 1
```

- zrevrange：逆序查询

```
127.0.0.1:6379> zrange zset01 0 -1      # 顺序查询
1) "vip1"
2) "vip2"
3) "vip3"
4) "vip4"
127.0.0.1:6379> zrevrange zset01 0 -1   # 逆序查询
1) "vip4"
2) "vip3"
3) "vip2"
4) "vip1"
```

- zrevrangebyscore：逆序范围查找

```
127.0.0.1:6379> zrevrangebyscore zset01 30 20   # 逆序查询分数在30~20之间的
（注意，先写大值，再写小值）
1) "vip3"
2) "vip2"
127.0.0.1:6379> zrevrangebyscore zset01 20 30   # 如果小值在前，则结果为null
(empty list or set)
```

3.3 持久化

3.3.1 RDB

Redis **D**ata**B**ase

- 在指定的时间间隔内，将内存中的数据集的快照写入磁盘；
- 默认保存在/usr/local/bin中，文件名dump.rdb;

3.3.1.1 自动备份

- redis是内存数据库，当我们每次用完redis，关闭linux时，按道理来说，内存释放，redis中的数据也会随之消失
- 为什么我们再次启动redis的时候，昨天的数据还在，并没有消失呢？
- 正是因为，每次关机时，redis会自动将数据备份到一个文件中 ：/usr/local/bin/**dump.rdb**
- 接下来我们就来全方位的认识 自动备份机制

1. 默认的自动备份策略不利于我们测试，所以修改redis.conf文件中的自动备份策略

   ```
   vim redis.conf
   /SNAP   # 搜索

   save 900 1      # 900秒内，至少变更1次，才会自动备份
   save 120 10     # 120秒内，至少变更10次，才会自动备份
   save 60 10000   # 60秒内，至少变更10000次，才会自动备份
   ```

   当然如果你只是用Redis的缓存功能，不需要持久化，那么你可以注释掉所有的 save 行来停用保存功能。可以直接一个空字符串来实现停用：save ""

2. 使用shutdown模拟关机 ，关机之前和关机之后，对比dump.rdb文件的更新时间

   注意：当我们使用shutdown命令，redis会自动将数据库备份，所以，dump.rdb文件创建时间更新了

3. 开机启动redis，我们要在120秒内保存10条数据，再查看dump.rdb文件的更新时间（开两个终端窗口，方便查看）

4. 120秒内保存10条数据这一动作触发了备份指令，目前，dump.rdb文件中保存了10条数据，将dump.rdb拷贝一份dump10.rdb，此时两个文件中都保存10条数据

5. 既然有数据已经备份了，那我们就肆无忌惮的将数据全部删除flushall，再次shutdown关机

6. 再次启动redis，发现数据真的消失了，并没有按照我们所想的 将dump.rdb文件中的内容恢复到redis中。为什么？

   > 因为，当我们保存10条以上的数据时，数据备份起来了；
   > 然后删除数据库，备份文件中的数据，也没问题；
   > 但是，问题出在shutdown上，这个命令一旦执行，就会立刻备份，将删除之后的空数据库生成备份文件，将之前装10条数据的备份文件覆盖掉了。所以，就出现了上图的结果。自动恢复失败。
   > 怎么解决这个问题呢？要将备份文件再备份

7. 将dump.rdb文件删除，将dump10.rdb重命名为dump.rdb

8. 启动redis服务，登录redis，数据10条，全部恢复！

3.3.1.2 手动备份

- 之前自动备份，必须更改好多数据，例如上边，我们改变了十多条数据，才会自动备份；
- 现在，我只保存一条数据，就想立刻备份，应该怎么做？
- 每次操作完成，执行命令 save 就会立刻备份

3.3.1.3 与RDB相关的配置

- stop-writes-on-bgsave-error：进水口和出水口，出水口发生故障与否
  - yes：当后台备份时候反生错误，前台停止写入
  - no：不管死活，就是往里怼
- rdbcompression：对于存储到磁盘中的快照，是否启动LZF压缩算法，**一般都会启动，因为这点性能，多买一台电脑，完全搞定N个来回了。**
  - yes：启动
  - no：不启动（不想消耗CPU资源，可关闭）
- rdbchecksum：在存储快照后，是否启动CRC64算法进行数据校验；
  - 开启后，大约增加10%左右的CPU消耗；
  - 如果希望获得最大的性能提升，可以选择关闭；
- dbfilename：快照备份文件名字
- dir：快照备份文件保存的目录，默认为当前目录

优势and劣势

- 优：适合大规模数据恢复，对数据完整性和一致行要求不高；
- 劣：一定间隔备份一次，意外down掉，就失去最后一次快照的所有修改

3.3.2 AOF

Append **O**nly **F**ile

- 以日志的形式记录每个写操作；
- 将redis执行过的写指令全部记录下来（读操作不记录）；
- 只许追加文件，不可以改写文件；
- redis在启动之初会读取该文件从头到尾执行一遍，这样来重新构建数据；

3.3.2.1 开启AOF

1. 为了避免失误，最好将redis.conf总配置文件备份一下，然后再修改内容如下：

```
appendonly  yes
appendfilename  appendonly.aof
```

2. 重新启动redis，以新配置文件启动

```
redis-server /usr/local/redis5.0.4/redis.conf
```

3. 连接redis，加数据，删库，退出
4. 查看当前文件夹多一个aof文件，看看文件中的内容,保存的都是 **写** 操作
   - 文件中最后一句要删除，否则数据恢复不了

    - 编辑这个文件，最后要 :wq! 强制执行
5. 只需要重新连接，数据恢复成功

3.3.2.2 共存？谁优先？

我们查看redis.conf文件，AOF和RDB两种备份策略可以同时开启，那系统会怎样选择？

1. 动手试试，编辑appendonly.aof，胡搞乱码，保存退出
2. 启动redis 失败，所以是AOF优先载入来恢复原始数据！因为AOF比RDB数据保存的完整性更高！
3. 修复AOF文件，杀光不符合redis语法规范的代码

```
reids-check-aof  --fix appendonly.aof
```

3.3.2.3 与AOF相关的配置

- appendonly：开启aof模式
- appendfilename：aof的文件名字，最好别改！
- appendfsync：追写策略
    - always：每次数据变更，就会**立即记录**到磁盘，性能较差，但数据完整性好
    - everysec：默认设置，异步操作，**每秒记录**，如果一秒内宕机，会有数据丢失
    - no：不追写
- no-appendfsync-on-rewrite：重写时是否运用Appendfsync追写策略；用默认no即可，保证数据安全性。
    - AOF采用文件追加的方式，文件会越来越大，为了解决这个问题，增加了重写机制，redis会自动记录上一次AOF文件的大小，当AOF文件大小达到预先设定的大小时，redis就会启动AOF文件进行内容压缩，只保留可以恢复数据的最小指令集合
- auto-aof-rewrite-percentage：如果AOF文件大小已经超过原来的100%，也就是一倍，才重写压缩
- auto-aof-rewrite-min-size：如果AOF文件已经超过了64mb，才重写压缩

3.3.3 总结（如何选择？）

- RDB：只用作后备用途，建议15分钟备份一次就好
- AOF：
    - 在最恶劣的情况下，也只丢失不超过2秒的数据，数据完整性比较高，但代价太大，会带来持续的IO
    - 对硬盘的大小要求也高，默认64mb太小了，企业级最少都是5G以上；
    - 后面要学习的master/slave才是新浪微博的选择！！

3.4 事务

- 可以一次执行多个命令，是一个命令组，一个事务中，所有命令都会序列化（排队），不会被插队；
- 一个队列中，一次性，顺序性，排他性的执行一系列命令

- 三特性
  - 隔离性：所有命令都会按照顺序执行，事务在执行的过程中，不会被其他客户端送来的命令打断
  - 没有隔离级别：队列中的命令没有提交之前都不会被实际的执行，不存在“事务中查询要看到事务里的更新，事务外查询不能看到”这个头疼的问题
  - 不保证原子性：冤有头债有主，如果一个命令失败，但是别的命令可能会执行成功，没有回滚
- 三步走
  - 开启multi
  - 入队queued
  - 执行exec
- 与关系型数据库事务相比，
  - multi：可以理解成关系型事务中的 begin
  - exec ：可以理解成关系型事务中的 commit
  - discard ：可以理解成关系型事务中的 rollback

3.4.1 一起生

开启事务，加入队列，一起执行，并成功

```
127.0.0.1:6379> multi           # 开启事务
OK
127.0.0.1:6379> set k1 v1
QUEUED      # 加入队列
127.0.0.1:6379> set k2 v2
QUEUED      # 加入队列
127.0.0.1:6379> get k2
QUEUED      # 加入队列
127.0.0.1:6379> set k3 v3
QUEUED      # 加入队列
127.0.0.1:6379> exec            # 执行，一起成功！
1) OK
2) OK
3) "v2"
4) OK
```

3.4.2 一起死

放弃之前的操作，恢复到原来的值

```
127.0.0.1:6379> multi           # 开启事务
OK
127.0.0.1:6379> set k1 v1111
QUEUED
127.0.0.1:6379> set k2 v2222
QUEUED
127.0.0.1:6379> discard         # 放弃操作
OK
127.0.0.1:6379> get k1
"v1"        # 还是原来的值
```

3.4.3 一粒老鼠屎坏一锅汤

一句报错，全部取消，恢复到原来的值

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set k4 v4
QUEUED
127.0.0.1:6379> setlalala       # 一句报错
(error) ERR unknown command `setlalala`, with args beginning with:
127.0.0.1:6379> set k5 v5
QUEUED
127.0.0.1:6379> exec            # 队列中命令全部取消
(error) EXECABORT Transaction discarded because of previous errors.
127.0.0.1:6379> keys *          # 还是原来的值
1) "k2"
2) "k3"
3) "k1"
```

3.4.4 冤有头债有主

追究责任，谁的错，找谁去

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> incr k1     # 虽然v1不能++，但是加入队列并没有报错，类似java中的通过编
译
QUEUED
127.0.0.1:6379> set k4 v4
QUEUED
127.0.0.1:6379> set k5 v5
QUEUED
127.0.0.1:6379> exec
1) (error) ERR value is not an integer or out of range      # 真正执行的时候，报错
2) OK       # 成功
3) OK       # 成功
127.0.0.1:6379> keys *
1) "k5"
2) "k1"
3) "k3"
4) "k2"
5) "k4"
```

3.4.5 watch监控

测试：模拟收入与支出

- 正常情况下：

```
127.0.0.1:6379> set in 100      # 收入100元
OK
127.0.0.1:6379> set out 0       # 支出0元
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379> decrby in 20    # 收入-20
QUEUED
127.0.0.1:6379> incrby out 20   # 支出+20
QUEUED
127.0.0.1:6379> exec
1) (integer) 80
2) (integer) 20     # 结果，没问题！
```

- 特殊情况下：

```
127.0.0.1:6379> watch in        # 监控收入in
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379> decrby in 20
QUEUED
127.0.0.1:6379> incrby out 20
QUEUED
127.0.0.1:6379> exec
(nil)       # 在exec之前，我开启了另一个窗口（线程），对监控的in做了修改，所以本次的事务将被打断（失效），类似于“乐观锁”
```

- unwatch：取消watch命令对所有key的操作
  - 一旦执行了exec命令，那么之前加的所有监控自动失效！

3.5 Redis的发布订阅

- 进程间的一种消息通信模式：发送者(pub)发送消息，订阅者(sub)接收消息。例如：微信订阅号
- 订阅一个或多个频道

```
127.0.0.1:6379> subscribe cctv1 cctv5 cctv6     # 1.订阅三个频道
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "cctv1"
3) (integer) 1
1) "subscribe"
2) "cctv5"
3) (integer) 2
1) "subscribe"
2) "cctv6"
3) (integer) 3
1) "message"                                    # 3.cctv5接收到推送过来的信息
2) "cctv5"
3) "NBA"
```

```
127.0.0.1:6379> publish cctv5 NBA            # 2.发送消息给cctv5
(integer) 1
```

3.5 主从复制

- 就是 redis集群的策略
- 配从（库）不配主（库）：小弟可以选择谁是大哥，但大哥没有权利去选择小弟
- 读写分离：主机写，从机读

3.5.1 一主二仆

1. 准备三台服务器，并修改redis.conf

```
bind 0.0.0.0
```

2. 启动三台redis，并查看每台机器的角色，都是master

```
info replication
```

3. 测试开始
    1. 首先，将三个机器全都清空，第一台添加值

    ```
    mset k1 v1 k2 v2
    ```

    2. 其余两台机器，复制（找大哥）

    ```
    slaveof 192.168.204.141 6379
    ```

    3. 第一台再添加值

    ```
    set k3 v3
    ```

    - **思考1**：**slave之前的k1和k2是否能拿到？**
        - 可以获得，只要跟了大哥，之前的数据也会立刻同步
    - **思考2**：**slave之后的k3是否能拿到？**
        - 可以获得，只要跟了大哥，数据会立刻同步
    - **思考3**：**同时添加k4，结果如何？**
        - 主机（141master）可以添加成功，从机（142和143是slave）失败，从机只负责读取数据，无权写入数据，这就是“读写分离”
    - **思考4**：**主机shutdown，从机如何？**
        - 142和143仍然是slave，并显示他们的master已离线
    - **思考5**：**主机重启，从机又如何？**
        - 142和143仍然是slave，并显示他们的master已上线
    - **思考6**：**从机死了，主机如何？从机归来身份是否变化？**
        - 主机没有变化，只是显示少了一个slave

- 主机和从机没有变化，而重启归来的从机自立门户成为了master，不和原来的集群在一起了

3.5.2 血脉相传

- 一个主机理论上可以多个从机，但是这样的话，这个主机会很累
- 我们可以使用java面向对象**继承中的传递性**来解决这个问题，减轻主机的负担
- 形成祖孙三代：

```
127.0.0.1:6379> slaveof 192.168.204.141 6379        # 142跟随141
OK
127.0.0.1:6379> slaveof 192.168.204.142 6379        # 143跟随142
OK
```

3.5.3 谋权篡位

- 1个主机，2个从机，当1个主机挂掉了，只能从2个从机中再次选1个主机
- 国不可一日无君，军不可一日无帅
- 手动选老大
- 模拟测试：1为master，2和3为slave，当1挂掉后，2篡权为master，3跟2

```
slaveof no one                  # 2上执行，没有人能让我臣服，那我就是老大
```

```
slaveof 192.168.204.142 6379    # 3跟随2号
```

- **思考**：**当1再次回归，会怎样？**
  - 2和3已经形成新的集群，和1没有任何的关系了。所以1成为了光杆司令

3.5.4 复制原理

完成上面几个步骤后就完成了从服务器数据初始化的所有操作，从服务器此时可以接收来自用户的读请求

- **全量复制**：Slave初始化阶段，这时Slave需要将Master上的所有数据都复制一份slave接收到数据文件后，存盘，并加载到内存中；（步骤1234）
- **增量复制**：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；（步骤56）
    - 但，只要是重新连接master，一次性（全量复制）同步将自动执行；
- 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。

3.5.5 哨兵模式

- 自动版的谋权篡位！
- 有个哨兵一直在巡逻，突然发现！！！！！老大挂了，小弟们会自动投票，从众小弟中选出新的老大
- Sentinel是Redis的高可用性解决方案：
    - 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代替已下线的主服务器继续处理命令请求
- 模拟测试
    1. 1主，2和3从
    2. 每一台服务器中创建一个配置文件sentinel.conf，名字绝不能错，并编辑sentinel.conf

        ```
        # sentinel monitor 被监控主机名（自定义） ip port 票数
        sentinel monitor redis141 192.168.204.141 6379 1
        ```

    3. 启动服务的顺序：主Redis --> 从Redis --> Sentinel1/2/3

        ```
        redis-sentinel sentinel.conf
        ```

    4. 将1号老大挂掉，后台自动发起激烈的投票，选出新的老大

        ```
        127.0.0.1:6379> shutdown
        not connected> exit
        ```

    5. 查看最后权利的分配
        - 3成为了新的老大，2还是小弟
    6. 如果之前的老大再次归来呢？
        - 1号再次归来，自己成为了master，和3平起平坐
        - 过了几秒之后，被哨兵检测到了1号机的归来，1号你别自己玩了，进入集体吧，但是新的老大已经产生了，你只能作为小弟再次进入集体！

3.5.6 缺点

- 由于所有的写操作都是在master上完成的；

- 然后再同步到slave上，所以两台机器之间通信会有延迟；
- 当系统很繁忙的时候，延迟问题会加重；
- slave机器数量增加，问题也会加重

3.6 总配置redis.conf 详解

```
# Redis 配置文件示例

# 注意单位: 当需要配置内存大小时, 可能需要指定像1k,5GB,4M等常见格式
#
# 1k => 1000 bytes
# 1kb => 1024 bytes
# 1m => 1000000 bytes
# 1mb => 1024*1024 bytes
# 1g => 1000000000 bytes
# 1gb => 1024*1024*1024 bytes
#
# 单位是对大小写不敏感的 1GB 1Gb 1gB 是相同的。

################################## INCLUDES 包含文件相关
####################################

# 可以在这里包含一个或多个其他的配置文件。如果你有一个适用于所有Redis服务器的标准配置模板
# 但也需要一些每个服务器自定义的设置，这个功能将很有用。被包含的配置文件也可以包含其他配置文
件，
# 所以需要谨慎的使用这个功能。
#
# 注意“inclue”选项不能被admin或Redis哨兵的"CONFIG REWRITE"命令重写。
# 因为Redis总是使用最后解析的配置行最为配置指令的值，你最好在这个文件的开头配置includes来
# 避免它在运行时重写配置。
# 如果相反你想用includes的配置覆盖原来的配置，你最好在该文件的最后使用include
#
# include /path/to/local.conf
# include /path/to/other.conf

################################## GENERAL 综合配置
####################################

# 默认Rdis不会作为守护进程运行。如果需要的话配置成'yes'
# 注意配置成守护进程后Redis会将进程号写入文件/var/run/redis.pid
daemonize no

# 当以守护进程方式运行时，默认Redis会把进程ID写到 /var/run/redis.pid。你可以在这里修改路
径。
pidfile /var/run/redis.pid

# 接受连接的特定端口，默认是6379
# 如果端口设置为0，Redis就不会监听TCP套接字。
port 6379

# TCP listen() backlog.
# server在与客户端建立tcp连接的过程中，SYN队列的大小
# 在高并发环境下你需要一个高backlog值来避免慢客户端连接问题。注意Linux内核默默地将这个值减小
# 到/proc/sys/net/core/somaxconn的值，所以需要确认增大somaxconn和tcp_max_syn_backlog
# 两个值来达到想要的效果。
```

```
tcp-backlog 511

# 默认Redis监听服务器上所有可用网络接口的连接。可以用"bind"配置指令跟一个或多个ip地址来实现
# 监听一个或多个网络接口
#
# 示例:
#
# bind 192.168.1.100 10.0.0.1
# bind 127.0.0.1

# 指定用来监听Unix套套接字的路径。没有默认值， 所以在没有指定的情况下Redis不会监听Unix套接
字
#
# unixsocket /tmp/redis.sock
# unixsocketperm 755

# 一个客户端空闲多少秒后关闭连接。(0代表禁用，永不关闭)
timeout 0

# TCP keepalive.
#
# 如果非零，则设置SO_KEEPALIVE选项来向空闲连接的客户端发送ACK，由于以下两个原因这是很有用
的:
#
# 1）能够检测无响应的对端
# 2）让该连接中间的网络设备知道这个连接还存活
#
# 在Linux上，这个指定的值(单位：秒)就是发送ACK的时间间隔。
# 注意：要关闭这个连接需要两倍的这个时间值。
# 在其他内核上这个时间间隔由内核配置决定
#
# 这个选项的一个合理值是60秒
tcp-keepalive 0

# 指定服务器调试等级
# 可能值:
# debug （大量信息，对开发/测试有用）
# verbose （很多精简的有用信息，但是不像debug等级那么多）
# notice （适量的信息，基本上是你生产环境中需要的）
# warning （只有很重要/严重的信息会记录下来）
loglevel notice

# 指明日志文件名。也可以使用"stdout"来强制让Redis把日志信息写到标准输出上。
# 注意:如果Redis以守护进程方式运行，而设置日志显示到标准输出的话，日志会发送到/dev/null
logfile ""

# 要使用系统日志记录器，只要设置 "syslog-enabled" 为 "yes" 就可以了。
# 然后根据需要设置其他一些syslog参数就可以了。
# syslog-enabled no

# 指明syslog身份
# syslog-ident redis

# 指明syslog的设备。必须是user或LOCAL0 ~ LOCAL7之一。
# syslog-facility local0

# 设置数据库个数。默认数据库是 DB 0，
```

```
# 可以通过select <dbid>  (0 <= dbid <= 'databases' - 1 ）来为每个连接使用不同的数据
库。
databases 16

################################ SNAPSHOTTING 快照，持久化操作配置
################################
#
# 把数据库存到磁盘上:
#
#  save <seconds> <changes>
#
#  会在指定秒数和数据变化次数之后把数据库写到磁盘上。
#
#  下面的例子将会进行把数据写入磁盘的操作:
#  900秒（15分钟）之后，且至少1次变更
#  300秒（5分钟）之后，且至少10次变更
#  60秒之后，且至少10000次变更
#
#  注意：你要想不写磁盘的话就把所有 "save" 设置注释掉就行了。
#
#  通过添加一条带空字符串参数的save指令也能移除之前所有配置的save指令
#  像下面的例子：
#  save ""

save 900 1
save 300 10
save 60 10000

# 默认如果开启RDB快照(至少一条save指令)并且最新的后台保存失败，Redis将会停止接受写操作
# 这将使用户知道数据没有正确的持久化到硬盘，否则可能没人注意到并且造成一些灾难。
#
# 如果后台保存进程能重新开始工作，Redis将自动允许写操作
#
# 然而如果你已经部署了适当的Redis服务器和持久化的监控，你可能想关掉这个功能以便于即使是
# 硬盘，权限等出问题了Redis也能够像平时一样正常工作，
stop-writes-on-bgsave-error yes

# 当导出到 .rdb 数据库时是否用LZF压缩字符串对象？
# 默认设置为 "yes"，因为几乎在任何情况下它都是不错的。
# 如果你想节省CPU的话你可以把这个设置为 "no"，但是如果你有可压缩的key和value的话，
# 那数据文件就会更大了。
rdbcompression yes

# 因为版本5的RDB有一个CRC64算法的校验和放在了文件的最后。这将使文件格式更加可靠但在
# 生产和加载RDB文件时，这有一个性能消耗(大约10%)，所以你可以关掉它来获取最好的性能。
#
# 生成的关闭校验的RDB文件有一个0的校验和，它将告诉加载代码跳过检查
rdbchecksum yes

# 持久化数据库的文件名
dbfilename dump.rdb

# 工作目录
#
# 数据库会写到这个目录下，文件名就是上面的 "dbfilename" 的值。
#
# 累加文件也放这里。
#
```

```
# 注意你这里指定的必须是目录，不是文件名。
dir ./

################################ REPLICATION 主从复制的配置
################################
# 主从同步。通过 slaveof 指令来实现Redis实例的备份。
# 注意，这里是本地从远端复制数据。也就是说，本地可以有不同的数据库文件、绑定不同的IP、监听
# 不同的端口。
#
# slaveof <masterip> <masterport>

# 如果master设置了密码保护（通过 "requirepass" 选项来配置），那么slave在开始同步之前必须
# 进行身份验证，否则它的同步请求会被拒绝。
#
# masterauth <master-password>

# 当一个slave失去和master的连接，或者同步正在进行中，slave的行为有两种可能：
#
# 1) 如果 slave-serve-stale-data 设置为 "yes" (默认值)，slave会继续响应客户端请求，
#    可能是正常数据，也可能是还没获得值的空数据。
# 2) 如果 slave-serve-stale-data 设置为 "no"，slave会回复"正在从master同步
#  （SYNC with master in progress）"来处理各种请求，除了 INFO 和 SLAVEOF 命令。
#
slave-serve-stale-data yes

# 你可以配置salve实例是否接受写操作。可写的slave实例可能对存储临时数据比较有用(因为写入
salve
# 的数据在同master同步之后将很容被删除)，但是如果客户端由于配置错误在写入时也可能产生一些问
题。
#
# 从Redis2.6默认所有的slave为只读
#
# 注意:只读的slave不是为了暴露给互联网上不可信的客户端而设计的。它只是一个防止实例误用的保护
层。
# 一个只读的slave支持所有的管理命令比如config,debug等。为了限制你可以用'rename-
command'来
# 隐藏所有的管理和危险命令来增强只读slave的安全性
slave-read-only yes

# slave根据指定的时间间隔向master发送ping请求。
# 时间间隔可以通过 repl_ping_slave_period 来设置。
# 默认10秒。
#
# repl-ping-slave-period 10

# 以下选项设置同步的超时时间
#
# 1）slave在与master SYNC期间有大量数据传输，造成超时
# 2）在slave角度，master超时，包括数据、ping等
# 3）在master角度，slave超时，当master发送REPLCONF ACK pings
#
# 确保这个值大于指定的repl-ping-slave-period，否则在主从间流量不高时每次都会检测到超时
#
# repl-timeout 60

# 是否在slave套接字发送SYNC之后禁用 TCP_NODELAY ？
#
```

```
# 如果你选择“yes”Redis将使用更少的TCP包和带宽来向slaves发送数据。但是这将使数据传输到
slave
# 上有延迟，Linux内核的默认配置会达到40毫秒
#
# 如果你选择了 "no" 数据传输到salve的延迟将会减少但要使用更多的带宽
#
# 默认我们会为低延迟做优化，但高流量情况或主从之间的跳数过多时，把这个选项设置为“yes”
# 是个不错的选择。
repl-disable-tcp-nodelay no

# 设置数据备份的backlog大小。backlog是一个slave在一段时间内断开连接时记录salve数据的缓
冲，
# 所以一个slave在重新连接时，不必要全量的同步，而是一个增量同步就足够了，将在断开连接的这段
# 时间内slave丢失的部分数据传送给它。
#
# 同步的backlog越大，slave能够进行增量同步并且允许断开连接的时间就越长。
#
# backlog只分配一次并且至少需要一个slave连接
#
# repl-backlog-size 1mb

# 当master在一段时间内不再与任何slave连接，backlog将会释放。以下选项配置了从最后一个
# slave断开开始计时多少秒后，backlog缓冲将会释放。
#
# 0表示永不释放backlog
#
# repl-backlog-ttl 3600

# slave的优先级是一个整数展示在Redis的Info输出中。如果master不再正常工作了，哨兵将用它来
# 选择一个slave提升=升为master。
#
# 优先级数字小的salve会优先考虑提升为master，所以例如有三个slave优先级分别为10，100，25，
# 哨兵将挑选优先级最小数字为10的slave。
#
# 0作为一个特殊的优先级，标识这个slave不能作为master，所以一个优先级为0的slave永远不会被
# 哨兵挑选提升为master
#
# 默认优先级为100
slave-priority 100

# 如果master少于N个延时小于等于M秒的已连接slave，就可以停止接收写操作。
#
# N个slave需要是“oneline”状态
#
# 延时是以秒为单位，并且必须小于等于指定值，是从最后一个从slave接收到的ping（通常每秒发送）
# 开始计数。
#
# This option does not GUARANTEES that N replicas will accept the write, but
# will limit the window of exposure for lost writes in case not enough slaves
# are available, to the specified number of seconds.
#
# 例如至少需要3个延时小于等于10秒的slave用下面的指令：
#
# min-slaves-to-write 3
# min-slaves-max-lag 10
#
# 两者之一设置为0将禁用这个功能。
#
```

```
# 默认 min-slaves-to-write 值是0（该功能禁用）并且 min-slaves-max-lag 值是10。

################################## SECURITY 安全相关配置
##################################

# 要求客户端在处理任何命令时都要验证身份和密码。
# 这个功能在有你不信任的其它客户端能够访问redis服务器的环境里非常有用。
#

# 为了向后兼容的话这段应该注释掉。而且大多数人不需要身份验证(例如:它们运行在自己的服务器上)
#
# 警告：因为Redis太快了，所以外面的人可以尝试每秒150k的密码来试图破解密码。这意味着你需要
# 一个高强度的密码，否则破解太容易了。
#
# requirepass foobared

# 命令重命名
#
# 在共享环境下，可以为危险命令改变名字。比如，你可以为 CONFIG 改个其他不太容易猜到的名字，
# 这样内部的工具仍然可以使用，而普通的客户端将不行。
#
# 例如：
#
# rename-command CONFIG b840fc02d524045429941cc15f59e41cb7be6c52
#
# 也可以通过改名为空字符串来完全禁用一个命令
#
# rename-command CONFIG ""
#
# 请注意：改变命令名字被记录到AOF文件或被传送到从服务器可能产生问题。

################################### LIMITS 范围配置
###################################

# 设置最多同时连接的客户端数量。默认这个限制是10000个客户端，然而如果Redis服务器不能配置
# 处理文件的限制数来满足指定的值，那么最大的客户端连接数就被设置成当前文件限制数减32（因
# 为Redis服务器保留了一些文件描述符作为内部使用）
#
# 一旦达到这个限制，Redis会关闭所有新连接并发送错误'max number of clients reached'
#
# maxclients 10000

# 不要用比设置的上限更多的内存。一旦内存使用达到上限，Redis会根据选定的回收策略（参见：
# maxmemmory-policy）删除key
#
# 如果因为删除策略Redis无法删除key，或者策略设置为 "noeviction"，Redis会回复需要更
# 多内存的错误信息给命令。例如，SET,LPUSH等等，但是会继续响应像Get这样的只读命令。
#
# 在使用Redis作为LRU缓存，或者为实例设置了硬性内存限制的时候（使用 "noeviction" 策略）
# 的时候，这个选项通常事很有用的。
#
# 警告：当有多个slave连上达到内存上限的实例时，master为同步slave的输出缓冲区所需
# 内存不计算在使用内存中。这样当驱逐key时，就不会因网络问题 / 重新同步事件触发驱逐key
# 的循环，反过来slaves的输出缓冲区充满了key被驱逐的DEL命令，这将触发删除更多的key，
# 直到这个数据库完全被清空为止
#
# 总之...如果你需要附加多个slave，建议你设置一个稍小maxmemory限制，这样系统就会有空闲
# 的内存作为slave的输出缓存区(但是如果最大内存策略设置为"noeviction"的话就没必要了)
```

```
#
# maxmemory <bytes>

# 最大内存策略：如果达到内存限制了，Redis如何选择删除key。你可以在下面五个行为里选：
#
# volatile-lru -> 根据LRU算法生成的过期时间来删除。
# allkeys-lru -> 根据LRU算法删除任何key。
# volatile-random -> 根据过期设置来随机删除key。
# allkeys->random -> 无差别随机删。
# volatile-ttl -> 根据最近过期时间来删除（辅以TTL）
# noeviction -> 谁也不删，直接在写操作时返回错误。
#
# 注意：对所有策略来说，如果Redis找不到合适的可以删除的key都会在写操作时返回一个错误。
#

#       目前为止涉及的命令：set setnx setex append
#       incr decr rpush lpush rpushx lpushx linsert lset rpoplpush sadd
#       sinter sinterstore sunion sunionstore sdiff sdiffstore zadd zincrby
#       zunionstore zinterstore hset hsetnx hmset hincrby incrby decrby
#       getset mset msetnx exec sort
#

# 默认值如下：
#
# maxmemory-policy volatile-lru

# LRU和最小TTL算法的实现都不是很精确，但是很接近（为了省内存），所以你可以用样本量做检测。
# 例如：默认Redis会检查3个key然后取最旧的那个，你可以通过下面的配置指令来设置样本的个数。
#
# maxmemory-samples 3

############################## APPEND ONLY MODE AOF模式配置
################################

# 默认情况下，Redis是异步的把数据导出到磁盘上。这种模式在很多应用里已经足够好，但Redis进程
# 出问题或断电时可能造成一段时间的写操作丢失(这取决于配置的save指令)。
#
# AOF是一种提供了更可靠的替代持久化模式，例如使用默认的数据写入文件策略（参见后面的配置）
# 在遇到像服务器断电或单写情况下Redis自身进程出问题但操作系统仍正常运行等突发事件时，Redis
# 能只丢失1秒的写操作。
#
# AOF和RDB持久化能同时启动并且不会有问题。
# 如果AOF开启，那么在启动时Redis将加载AOF文件，它更能保证数据的可靠性。
#
# 请查看 http://redis.io/topics/persistence 来获取更多信息.

appendonly no

# 纯累加文件名字（默认："appendonly.aof"）

appendfilename "appendonly.aof"

# fsync() 系统调用告诉操作系统把数据写到磁盘上，而不是等更多的数据进入输出缓冲区。
# 有些操作系统会真的把数据马上刷到磁盘上；有些则会尽快去尝试这么做。
#
# Redis支持三种不同的模式：
#
# no：不要立刻刷，只有在操作系统需要刷的时候再刷。比较快。
```

```
# always：每次写操作都立刻写入到aof文件。慢，但是最安全。
# everysec：每秒写一次。折中方案。
#
# 默认的 "everysec" 通常来说能在速度和数据安全性之间取得比较好的平衡。根据你的理解来
# 决定，如果你能放宽该配置为"no" 来获取更好的性能(但如果你能忍受一些数据丢失，可以考虑使用
# 默认的快照持久化模式)，或者相反，用“always”会比较慢但比everysec要更安全。
#
# 请查看下面的文章来获取更多的细节
# http://antirez.com/post/redis-persistence-demystified.html
#
# 如果不能确定，就用 "everysec"

# appendfsync always
appendfsync everysec
# appendfsync no

# 如果AOF的同步策略设置成 "always" 或者 "everysec"，并且后台的存储进程（后台存储或写入AOF
# 日志）会产生很多磁盘I/O开销。某些Linux的配置下会使Redis因为 fsync()系统调用而阻塞很久。
# 注意，目前对这个情况还没有完美修正，甚至不同线程的 fsync() 会阻塞我们同步的write(2)调用。
#
# 为了缓解这个问题，可以用下面这个选项。它可以在 BGSAVE 或 BGREWRITEAOF 处理时阻止
fsync()。
#
# 这就意味着如果有子进程在进行保存操作，那么Redis就处于"不可同步"的状态。
# 这实际上是说，在最差的情况下可能会丢掉30秒钟的日志数据。（默认Linux设定）
#
# 如果把这个设置成"yes"带来了延迟问题，就保持"no"，这是保存持久数据的最安全的方式。

no-appendfsync-on-rewrite no

# 自动重写AOF文件
# 如果AOF日志文件增大到指定百分比，Redis能够通过 BGREWRITEAOF 自动重写AOF日志文件。
#
# 工作原理：Redis记住上次重写时AOF文件的大小（如果重启后还没有写操作，就直接用启动时的AOF大
小）
#
# 这个基准大小和当前大小做比较。如果当前大小超过指定比例，就会触发重写操作。你还需要指定被重写
# 日志的最小尺寸，这样避免了达到指定百分比但尺寸仍然很小的情况还要重写。
#
# 指定百分比为0会禁用AOF自动重写特性。

auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb

################################ LUA SCRIPTING ###############################
# 设置lua脚本的最大运行时间，单位为毫秒，redis会记个log，然后返回error。当一个脚本超过了最
大时限。只有SCRIPT KILL和SHUTDOWN NOSAVE可以用。第一个可以杀没有调write命令的东西。要是已
经调用了write，只能用第二个命令杀。
lua-time-limit 5000

################################ SLOW LOG ###################################
# 是redis用于记录慢查询执行时间的日志系统。由于slowlog只保存在内存中，因此slowlog的效率很
高，完全不用担心会影响到redis的性能。
# 只有query执行时间大于slowlog-log-slower-than的才会定义成慢查询，才会被slowlog进行记
录。
# 单位是微妙
slowlog-log-slower-than 10000
```

```
# slowlog-max-len表示慢查询最大的条数
slowlog-max-len 128

############################# EVENT NOTIFICATION ##############################
# 这个功能可以让客户端通过订阅给定的频道或者模式，来获知数据库中键的变化，以及数据库中命令的执行情况，所以在默认配置下，该功能处于关闭状态。
# notify-keyspace-events 的参数可以是以下字符的任意组合，它指定了服务器该发送哪些类型的通知：
# K 键空间通知，所有通知以 __keyspace@__ 为前缀
# E 键事件通知，所有通知以 __keyevent@__ 为前缀
# g DEL 、 EXPIRE 、 RENAME 等类型无关的通用命令的通知
# $ 字符串命令的通知
# l 列表命令的通知
# s 集合命令的通知
# h 哈希命令的通知
# z 有序集合命令的通知
# x 过期事件：每当有过期键被删除时发送
# e 驱逐(evict)事件：每当有键因为 maxmemory 政策而被删除时发送
# A 参数 g$lshzxe 的别名
# 输入的参数中至少要有一个 K 或者 E，否则的话，不管其余的参数是什么，都不会有任何 通知被分发。详细使用可以参考http://redis.io/topics/notifications

notify-keyspace-events ""

############################### ADVANCED CONFIG ###############################
# 单位字节:数据量小于等于hash-max-ziplist-entries的用ziplist，大于hash-max-ziplist-entries用hash
hash-max-ziplist-entries 512
# value大小小于等于hash-max-ziplist-value的用ziplist，大于hash-max-ziplist-value用hash。
hash-max-ziplist-value 64

# 数据量小于等于list-max-ziplist-entries用ziplist(压缩列表)，大于list-max-ziplist-entries用list。
list-max-ziplist-entries 512
# value大小小于等于list-max-ziplist-value的用ziplist，大于list-max-ziplist-value用list。
list-max-ziplist-value 64

# 数据量小于等于set-max-intset-entries用iniset，大于set-max-intset-entries用set。
set-max-intset-entries 512

# 数据量小于等于zset-max-ziplist-entries用ziplist，大于zset-max-ziplist-entries用zset。
zset-max-ziplist-entries 128
# value大小小于等于zset-max-ziplist-value用ziplist，大于zset-max-ziplist-value用zset。
zset-max-ziplist-value 64

# 基数统计的算法 HyperLogLog 键只需要花费 12 KB 内存，就可以计算接近 2^64 个不同元素的基数
# 设置HyeperLogLog的字节数限制，这个值通常在0~15000之间，默认为3000，基本不超过16000。value大小小于等于hll-sparse-max-bytes使用稀疏数据结构（sparse），大于hll-sparse-max-bytes使用稠密的数据结构（dense）。一个比16000大的value是几乎没用的，建议的value大概为3000。如果对CPU要求不高，对空间要求较高的，建议设置到10000左右。
hll-sparse-max-bytes 3000
```

```
# 重置hash。 Redis将在每100毫秒时使用1毫秒的CPU时间来对redis的hash表进行重新hash，可以降低内存的使用。当你的使用场景中，有非常严格的实时性需要，不能够接受Redis时不时的对请求有2毫秒的延迟的话，把这项配置为no。如果没有这么严格的实时性要求，可以设置为yes，以便能够尽可能快的释放内存。
activerehashing yes

# 对于Redis服务器的输出（也就是命令的返回值）来说，其大小通常是不可控制的。有可能一个简单的命令，能够产生体积庞大的返回数据。另外也有可能因为执行了太多命令，导致产生返回数据的速率超过了往客户端发送的速率，这是也会导致服务器堆积大量消息，从而导致输出缓冲区越来越大，占用过多内存，甚至导致系统崩溃。
# 用于强制断开出于某些原因而无法以足够快的速度从服务器读取数据的客户端的连接。
#对于normal client，包括monitor。第一个0表示取消hard limit，第二个0和第三个0表示取消soft limit，normal client默认取消限制，因为如果没有寻问，他们是不会接收数据的。
client-output-buffer-limit normal 0 0 0
#对于slave client和MONITER client，如果client-output-buffer一旦超过256mb，又或者超过64mb持续60秒，那么服务器就会立即断开客户端连接。
client-output-buffer-limit slave 256mb 64mb 60
#对于pubsub client，如果client-output-buffer一旦超过32mb，又或者超过8mb持续60秒，那么服务器就会立即断开客户端连接。
client-output-buffer-limit pubsub 32mb 8mb 60

# redis执行任务的频率
hz 10

# aof rewrite过程中,是否采取增量"文件同步"策略,默认为"yes",而且必须为yes.
# rewrite过程中,每32M数据进行一次文件同步,这样可以减少"aof大文件"写入对磁盘的操作次数.
aof-rewrite-incremental-fsync yes
```

- 通常情况下，默认的配置足够你解决问题！
- 没有极特殊的要求，不要乱改配置！

3.7 Jedis

java和redis打交道的API客户端

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>3.1.0</version>
</dependency>
```

3.7.1 连接redis

```
public static void main(String[] args) {
    Jedis jedis = new Jedis("192.168.204.141",6379);
    String pong = jedis.ping();
    System.out.println("pong = " + pong);
}

// 运行前：
// 1.关闭防火墙 systemctl stop firewalld.service
// 2.修改redis.conf [ bind 0.0.0.0 ] 允许任何ip访问，以这个redis.conf启动redis服务（重启redis）
//      redis-server /opt/redis5.0.4/redis.conf
```

3.7.2 常用API

```
private void testString(){
    Jedis jedis = new Jedis("192.168.204.141",6379);

    // string
    jedis.set("k1","v1");
    jedis.set("k2","v2");
    jedis.set("k3","v3");

    Set<String> set = jedis.keys("*");
    Iterator<String> iterator = set.iterator();
    for (set.iterator();iterator.hasNext();){
        String k = iterator.next();
        System.out.println(k+"->"+jedis.get(k));
    }
    Boolean k2Exists = jedis.exists("k2"); // 查看k2是否存在
    System.out.println("k2Exists = " + k2Exists);
    System.out.println( jedis.ttl("k1") );// 查看k1的过期时间

    jedis.mset("k4","v4","k5","v5");
    System.out.println( jedis.mget("k1","k2","k3","k4","k5") );
    System.out.println("------------------------------------------------------------");

}

private void testList(){
    Jedis jedis = new Jedis("192.168.204.141",6379);
    // list
    jedis.lpush("list01", "l1","l2","l3","l4","l5");
    List<String> list01 = jedis.lrange("list01", 0, -1);
    for(String s : list01){
        System.out.println(s);
    }
    System.out.println("------------------------------------------------------------");
}

private void testSet(){
    Jedis jedis = new Jedis("192.168.204.141",6379);
```

```
    // set
    jedis.sadd("order","jd001");
    jedis.sadd("order","jd002");
    jedis.sadd("order","jd003");
    Set<String> order = jedis.smembers("order");
    Iterator<String> order_iterator = order.iterator();
    while(order_iterator.hasNext()){
        String s = order_iterator.next();
        System.out.println(s);
    }
    jedis.srem("order", "jd002");
    System.out.println( jedis.smembers("order").size() );
}

private void testHash(){
    Jedis jedis = new Jedis("192.168.204.141",6379);
    jedis.hset("user1", "username","james");
    System.out.println( jedis.hget("user1", "username") );

    HashMap<String, String> map = new HashMap<String, String>();
    map.put("username", "tom");
    map.put("gender", "boy");
    map.put("address", "beijing");
    map.put("phone", "13590875543");

    jedis.hmset("user2", map);
    List<String> list = jedis.hmget("user2", "username", "phone");
    for(String s: list){
        System.out.println(s);
    }
}

private void testZset(){
    Jedis jedis = new Jedis("192.168.204.141",6379);
    jedis.zadd("zset01", 60d, "zs1");
    jedis.zadd("zset01", 70d, "zs2");
    jedis.zadd("zset01", 80d, "zs3");
    jedis.zadd("zset01", 90d, "zs4");
    Set<String> zset01 = jedis.zrange("zset01", 0, -1);
    Iterator<String> iterator = zset01.iterator();
    while (iterator.hasNext()){
        String s = iterator.next();
        System.out.println(s);
    }
}

public static void main(String[] args) {
    new Test2_API().testZset();
}
```

3.7.3 事务

- 初始化余额和支出

```
set yue 100
set zhichu 0
```

```
public static void main(String[] args) throws  Exception{
    Jedis jedis = new Jedis("192.168.204.141",6379);

    int yue  = Integer.parseInt( jedis.get("yue") );
    int zhichu = 10;

    jedis.watch("yue"); // 监控余额
    Thread.sleep(5000); // 模拟网络延迟

    if(yue < zhichu){
        jedis.unwatch(); //解除监控
        System.out.println("余额不足！");
    }else{
        Transaction transaction = jedis.multi(); // 开启事务
        transaction.decrBy("yue", zhichu);  // 余额减少
        transaction.incrBy("zhichu", zhichu); // 累计消费增加
        transaction.exec();
        System.out.println("余额：" + jedis.get("yue"));
        System.out.println("累计支出：" + jedis.get("zhichu"));
    }
}
```

- 模拟网络延迟：，10秒内，进入linux修改余额为5，这样，余额<支出，就会进入if

3.7.4 JedisPool

- redis的连接池技术
- 详情：https://help.aliyun.com/document_detail/98726.html

```
<dependency>
    <groupId>commons-pool</groupId>
    <artifactId>commons-pool</artifactId>
    <version>1.6</version>
</dependency>
```

- 使用单例模式进行优化

```
package com.lagou;

import jdk.nashorn.internal.scripts.JD;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;
import redis.clients.jedis.JedisPoolConfig;

/**
 * @BelongsProject: lagou-jedis
 * @Author: GuoAn.Sun
 * @CreateTime: 2020-08-05 17:46
 * @Description: 单例模式优化jedis连接池
 */
```

```
public class JedisPoolUtil {

    private JedisPoolUtil(){}

    private volatile static JedisPool jedisPool = null;
    private volatile static Jedis jedis = null;

    // 返回一个连接池
    private static JedisPool getInstance(){
        // 双层检测锁（企业中用的非常频繁）
        if(jedisPool == null){ // 第一层：检测体温
            synchronized (JedisPoolUtil.class){  // 排队进站
                if(jedisPool == null) { //第二层：查看健康码
                    JedisPoolConfig config = new JedisPoolConfig();
                    config.setMaxTotal(1000);  // 资源池中的最大连接数
                    config.setMaxIdle(30);  // 资源池允许的最大空闲连接数
                    config.setMaxWaitMillis(60*1000); // 当资源池连接用尽后，调用者的最大等待时间（单位为毫秒）
                    config.setTestOnBorrow(true); //向资源池借用连接时是否做连接有效性检测(业务量很大时候建议设置为false，减少一次ping的开销)
                    jedisPool = new JedisPool( config, "192.168.204.141",6379 );
                }
            }
        }
        return jedisPool;
    }

    // 返回jedis对象
    public static Jedis getJedis(){
        if(jedis == null){
            jedis = getInstance().getResource();
        }
        return jedis;
    }
}
```

- 测试类

```
/**
 * @BelongsProject: lagou-jedis
 * @Author: GuoAn.Sun
 * @CreateTime: 2020-08-05 17:54
 * @Description: 测试jedis连接池
 */
public class Test_JedisPool {
    public static void main(String[] args) {
        Jedis jedis1 = JedisPoolUtil.getJedis();
        Jedis jedis2 = JedisPoolUtil.getJedis();

        System.out.println(jedis1==jedis2);
    }
}
```

3.8 高并发下的分布式锁

- 经典案例：秒杀，抢购优惠券等

3.8.1 搭建工程并测试单线程

```
<packaging>war</packaging>

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.2.7.RELEASE</version>
    </dependency>
    <!--实现分布式锁的工具类-->
    <dependency>
        <groupId>org.redisson</groupId>
        <artifactId>redisson</artifactId>
        <version>3.6.1</version>
    </dependency>
    <!--spring操作redis的工具类-->
    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-redis</artifactId>
        <version>2.3.2.RELEASE</version>
    </dependency>
    <!--redis客户端-->
    <dependency>
        <groupId>redis.clients</groupId>
        <artifactId>jedis</artifactId>
        <version>3.1.0</version>
    </dependency>
    <!--json解析工具-->
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.9.8</version>
    </dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.tomcat.maven</groupId>
            <artifactId>tomcat7-maven-plugin</artifactId>
            <configuration>
                <port>8001</port>
                <path>/</path>
            </configuration>
            <executions>
                <execution>
                    <!-- 打包完成后,运行服务 -->
                    <phase>package</phase>
                    <goals>
                        <goal>run</goal>
                    </goals>
```

```
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         id="WebApp_ID" version="3.1">

    <servlet>
        <servlet-name>springmvc</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:spring/spring.xml</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="controller"/>
    <bean id="stringRedisTemplate"
class="org.springframework.data.redis.core.StringRedisTemplate">
        <property name="connectionFactory" ref="connectionFactory"></property>
    </bean>
    <bean id="connectionFactory"
class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory">
        <property name="hostName" value="192.168.204.141"></property>
        <property name="port" value="6379"/>
    </bean>

</beans>
```

```
package controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.StringRedisTemplate;
```

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

/**
 * @BelongsProject: lagou-killreids
 * @Author: GuoAn.Sun
 * @CreateTime: 2020-08-06 11:57
 * @Description: 测试秒杀
 */
@Controller
public class TestKill {

    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    @RequestMapping("kill")
    // 只能解决一个tomcat的并发问题：synchronized锁的一个进程下的线程并发，如果分布式环境，多个进程并发，这种方案就失效了！
    public @ResponseBody synchronized String kill() {
        // 1.从redis中获取 手机的库存数量
        int phoneCount = Integer.parseInt(stringRedisTemplate.opsForValue().get("phone"));
        // 2.判断手机的数量是否够秒杀的
        if(phoneCount > 0){
            phoneCount--;
            // 库存减少后，再将库存的值保存回redis
            stringRedisTemplate.opsForValue().set("phone", phoneCount+"");
            System.out.println("库存-1，剩余："+ phoneCount);
        }else{
            System.out.println("库存不足！");
        }
        return "over!";
    }
}
```

3.8.2 高并发测试

1. 启动两次工程，端口号分别8001和8002
2. 使用nginx做负载均衡

```
upstream sga{
    server 192.168.204.1:8001;
    server 192.168.204.1:8002;
}

server {
    listen       80;
    server_name  localhost;

    #charset koi8-r;

    #access_log  logs/host.access.log  main;

    location / {
```

```
        proxy_pass http://sga;
        root   html;
        index  index.html index.htm;
    }
```

```
/usr/local/nginx/sbin/nginx -c /usr/local/nginx/conf/nginx.conf
```

3. 使用 JMeter 模拟1秒内发出100个http请求，会发现同一个商品会被两台服务器同时抢购！

3.8.3 实现分布式锁的思路

1. 因为**redis是单线程**的，所以命令也就具备原子性，使用setnx命令实现锁，保存k-v
    - 如果k不存在，保存（当前线程加锁），执行完成后，删除k表示释放锁
    - 如果k已存在，阻塞线程执行，表示有锁
2. 如果加锁成功，在执行业务代码的过程中出现异常，导致没有删除k（释放锁失败），那么就会造成死锁（后面的所有线程都无法执行）！
    - 设置过期时间，例如10秒后，redis自动删除
3. 高并发下，由于时间段等因素导致服务器压力过大或过小，每个线程执行的时间不同
    - 第一个线程，执行需要13秒，执行到第10秒时，redis自动过期了k（释放锁）
    - 第二个线程，执行需要7秒，加锁，执行第3秒（锁 被释放了，为什么，是被第一个线程的finally主动deleteKey释放掉了）
    - 。。。连锁反应，当前线程刚加的锁，就被其他线程释放掉了，周而复始，导致锁会永久失效
4. 给每个线程加上唯一的标识UUID随机生成，释放的时候判断是否是当前的标识即可
5. 问题又来了，过期时间如果设定？
    - 如果10秒太短不够用怎么办？
    - 设置60秒，太长又浪费时间
    - 可以开启一个定时器线程，当过期时间小于总过期时间的1/3时，增长总过期时间（吃仙丹续命！）

自己实现分布式锁，太难了！

3.8.4 Redisson

- Redis 是最流行的 NoSQL 数据库解决方案之一，而 Java 是世界上最流行（注意，我没有说“最好”）的编程语言之一。
- 虽然两者看起来很自然地在一起“工作”，但是要知道，Redis 其实并没有对 Java 提供原生支持。
- 相反，作为 Java 开发人员，我们若想在程序中集成 Redis，必须使用 Redis 的第三方库。
- 而 Redisson 就是用于在 Java 程序中操作 Redis 的库，它使得我们可以在程序中轻松地使用 Redis。
- Redisson 在 java.util 中常用接口的基础上，为我们提供了一系列具有**分布式特性**的工具类。

```
package controller;

import org.redisson.Redisson;
import org.redisson.api.RLock;
```

```java
import org.redisson.config.Config;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

import java.util.concurrent.TimeUnit;

/**
 * @BelongsProject: lagou-killreids
 * @Author: GuoAn.Sun
 * @CreateTime: 2020-08-06 11:57
 * @Description: 测试秒杀
 */
@Controller
public class TestKill {

    @Autowired
    private Redisson redisson;

    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    @RequestMapping("kill")
    // 只能解决一个tomcat的并发问题：synchronized锁的一个进程下的线程并发，如果分布式环境，多个进程并发，这种方案就失效了！
    public @ResponseBody synchronized String kill() {

        // 定义商品id
        String productKey = "HUAWEI-P40";
        // 通过redisson获取锁
        RLock rLock = redisson.getLock(productKey); // 底层源码就是集成了setnx，过期时间等操作
        // 上锁（过期时间为30秒）
        rLock.lock(30, TimeUnit.SECONDS);

        try{
        // 1.从redis中获取 手机的库存数量
        int phoneCount = Integer.parseInt(stringRedisTemplate.opsForValue().get("phone"));
        // 2.判断手机的数量是否够秒杀的
        if (phoneCount > 0) {
            phoneCount--;
            // 库存减少后，再将库存的值保存回redis
            stringRedisTemplate.opsForValue().set("phone", phoneCount + "");
            System.out.println("库存-1，剩余：" + phoneCount);
        } else {
            System.out.println("库存不足！");
        }
        }catch (Exception e){
            e.printStackTrace();
        }finally {
            // 释放锁
            rLock.unlock();
        }
        return "over!";
```

```
    }


    @Bean
    public Redisson redisson(){
        Config config = new Config();
        // 使用单个redis服务器

 config.useSingleServer().setAddress("redis://192.168.204.141:6379").setDatabase
(0);
        // 使用集群redis
        //
config.useClusterServers().setScanInterval(2000).addNodeAddress("redis://192.168
.204.141:6379","redis://192.168.204.142:6379","redis://192.168.204.143:6379");
        return (Redisson)Redisson.create(config);
    }
}
```

- 实现分布式锁的方案其实有很多，我们之前用过的zookeeper的特点就是**高可靠性**，现在我们用的redis特点就是**高性能**。
- 目前分布式锁，应用最多的仍然是“**Redis**”