

Redis(redis-server /usr/local/redis6/redis6/redis.conf)

NoSQL (not only sql)

在现代的计算系统上每天网络上都会产生庞大的数据量。

这些数据有很大一部分是由关系数据库管理系统（RDBMS）来处理。1970年 E.F.Codd's提出的关系模型的论文 "A relational model of data for large shared data banks", 这使得数据建模和应用程序编程更加简单。

通过应用实践证明，关系模型是非常适合于客户服务器编程，远远超出预期的利益，今天它是结构化数据存储在网络和商务应用的主导技术。

NoSQL 是一项全新的数据库革命性运动，早期就有人提出，发展至2009年趋势越发高涨。NoSQL的拥护者们提倡运用非关系型的数据存储，相对于铺天盖地的关系型数据库运用，这一概念无疑是一种全新的思维的注入。

什么是NoSQL?

NoSQL，指的是非关系型的数据库。NoSQL有时也称作Not Only SQL的缩写，是对不同于传统的关系型数据库的数据库管理系统的统称。

NoSQL用于超大规模数据的存储。（例如谷歌或Facebook每天为他们的用户收集万亿比特的数据）。这些类型的数据存储不需要固定的模式，无需多余操作就可以横向扩展。

对于NoSQL并没有一个明确的范围和定义，但是他们都普遍存在下面一些共同特征：

易扩展

NoSQL数据库种类繁多，但是一个共同的特点都是去掉关系数据库的关系型特性。数据之间无关系，这样就非常容易扩展。无形之间，在架构的层面上带来了可扩展的能力。

大数据量，高性能

NoSQL数据库都具有非常高的读写性能，尤其在大数据量下，同样表现优秀。这得益于它的无关系性，数据库的结构简单。一般MySQL使用Query Cache。NoSQL的Cache是记录级的，是一种细粒度的Cache，所以NoSQL在这个层面上来说性能就要高很多。

灵活的数据模型

NoSQL无须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。这点在大数据量的Web 2.0时代尤其明显。

高可用

NoSQL在不太影响性能的情况，就可以方便地实现高可用的架构。比如Cassandra、HBase模型，通过复制模型也能实现高可用。

NoSQL的优点/缺点

优点：

- - 高可扩展性
- - 分布式计算
- - 低成本
- - 架构的灵活性，半结构化数据
- - 没有复杂的关系

缺点:

- - 没有标准化
- - 有限的查询功能（到目前为止）
- - 最终一致是不直观的程序

NoSQL 数据库分类

键值(Key-Value)存储数据库

这一类数据库主要会使用到一个[哈希表](#)，这个表中有一个特定的键和一个指针指向特定的数据。

Key/value模型对于IT系统来说的优势在于简单、易部署。但是如果[数据库管理员\(DBA\)](#)只对部分值进行查询或更新的时候，Key/value就显得效率低下了。举例如：Tokyo Cabinet/Tyrant， Redis， Voldemort， Oracle BDB。

列存储数据库

这部分数据库通常是用来应对分布式存储的海量数据。键仍然存在，但是它们的特点是指向了多个列。这些列是由列家族来安排的。如：Cassandra， HBase， Riak。

文档型数据库

文档型数据库的灵感是来自于Lotus Notes办公软件的，而且它同第一种键值存储相类似。该类型的数据模型是版本化的文档，半结构化的文档以特定的格式存储，比如JSON。文档型数据库可以看作是键值数据库的升级版，允许之间嵌套键值，在处理网页等复杂数据时，文档型数据库比传统键值数据库的查询效率更高。如：CouchDB， MongoDB. 国内也有文档型数据库SequoiaDB，已经开源。

图形(Graph)数据库

图形结构的数据库同其他行列以及刚性结构的SQL数据库不同，它是使用灵活的图形模型，并且能够扩展到多个服务器上。NoSQL数据库没有标准的查询语言(SQL)，因此进行数据库查询需要制定数据模型。许多NoSQL数据库都有REST式的数据接口或者查询API。如：Neo4j， InfoGrid， Infinite Graph。

类型	部分代表	特点
列存储	HbaseCassandraHypertable	顾名思义，是按列存储数据的。最大的特点是方便存储结构化和半结构化数据，方便做数据压缩，对针对某一列或者某几列的查询有非常大的IO优势。
文档存储	MongoDBCouchDB	文档存储一般用类似json的格式存储，存储的内容是文档型的。这样也就有机会对某些字段建立索引，实现关系数据库的某些功能。
key-value 存储	Tokyo Cabinet / TyrantBerkeley DBMemcacheDBRedis	可以通过key快速查询到其value。一般来说，存储不管value的格式，照单全收。（Redis包含了其他功能）
图存储	Neo4JFlockDB	图形关系的最佳存储。使用传统关系数据库来解决的话性能低下，而且设计使用不方便。
对象存储	db4oVersant	通过类似面向对象语言的语法操作数据库，通过对象的方式存取数据。
xml 数据库	Berkeley DB XMLBaseX	高效的存储XML数据，并支持XML的内部查询语法，比如XQuery,Xpath。

Redis概述

REmote DIctionary Server(Redis) 即远程字典服务，是一个由 Salvatore Sanfilippo 写的 key-value 存储系统，是跨平台的非关系型数据库，也被人们称为结构化数据库。

Redis 是一个开源的使用 ANSI C 语言编写、遵守 BSD 协议、支持网络、可基于内存、分布式、可选持久性的键值对(Key-Value)存储数据库，并提供多种语言的 API。

Redis 通常被称为数据结构服务器，因为值（value）可以是字符串(String)、哈希(Hash)、列表(list)、集合(sets)和有序集合(sorted sets)等类型。

Redis 是一个开源（BSD许可）的，内存中的数据结构存储系统，它可以用作数据库、缓存和消息中间件。它支持多种类型的数据结构，如 [字符串 \(strings\)](#)，[散列 \(hashes\)](#)，[列表 \(lists\)](#)，[集合 \(sets\)](#)，[有序集合 \(sorted sets\)](#) 与范围查询，[bitmaps](#)，[hyperloglogs](#) 和 [地理空间 \(geospatial\)](#) 索引半径查询。Redis 内置了 [复制 \(replication\)](#)，[LUA脚本 \(Lua scripting\)](#)，[LRU 驱动事件 \(LRU eviction\)](#)，[事务 \(transactions\)](#) 和不同级别的 [磁盘持久化 \(persistence\)](#)，并通过 [Redis哨兵 \(Sentinel\)](#) 和自动 [分区 \(Cluster\)](#) 提供高可用性（high availability）。

特点

- 性能极高 – Redis能读的速度是110000次/s,写的速度是81000次/s。
- 丰富的数据类型 – Redis支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。
- 原子 – Redis的所有操作都是原子性的，意思就是要么成功执行要么失败完全不执行。单个操作是原子性的。多个操作也支持事务，即原子性，通过MULTI和EXEC指令包起来。
- 丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性。

Redis能干嘛

- 1、内存存储、持久化，内存中时断电即失，所以说持久化很重要（rdb、aof）
- 2、效率高，可以用于高速缓存
- 3、发布订阅系统
- 4、地图信息分析
- 5、计时器、计数器（浏览量）
- 6、。。。

Windows安装redis

- 1、github下载压缩包<https://github.com/microsoftarchive/redis/releases/tag/win-3.2.100>
- 2、直接解压
- 3、打开服务端
- 4、打开客户端测试



The screenshot shows a terminal window titled "D:\JavaEnvironment\redis\redis-cli.exe". The user enters the command "127.0.0.1:6379> ping", which returns "PONG". This command is highlighted with a red box and labeled "测试连接" (Test connection). Next, the user enters "127.0.0.1:6379> set name youxin", which returns "OK". This command is also highlighted with a red box and labeled "set". Finally, the user enters "127.0.0.1:6379> get name", which returns "\"youxin\"". This command is highlighted with a red box and labeled "get". The prompt "127.0.0.1:6379>" is visible at the bottom.

命令 `redis-server.exe redis.windows.conf` 解决双击闪退问题

Linux安装redis:

下载redis压缩包:

wget <https://download.redis.io/releases/redis-6.2.6.tar.gz>

解压到对应安装目录:

```
mkdir /usr/local/redis6
```

```
sudo tar -zxvf ~/redis-6.2.6.tar.gz -C /usr/local/redis6
```

编译

cd到/usr/local/redis6目录，输入命令make执行编译命令，接下来控制台会输出各种编译过程中输出的内容。

```
make -version
```

查看make是否安装

如果没有安装make可以使用

```
sudo apt install -y make
```

进入安装目录：安装必要的依赖：

```
sudo apt install -y gcc
```

```
sudo apt install -y tcl
```

注意：如果其中下载失败并提示更新，使用：

```
sudo apt-get update
```

执行命令：

```
youxin@youxin-virtual-machine:/usr/local/redis6/redis-6.2.6$ sudo make MALLOC=libc
```

执行命令：

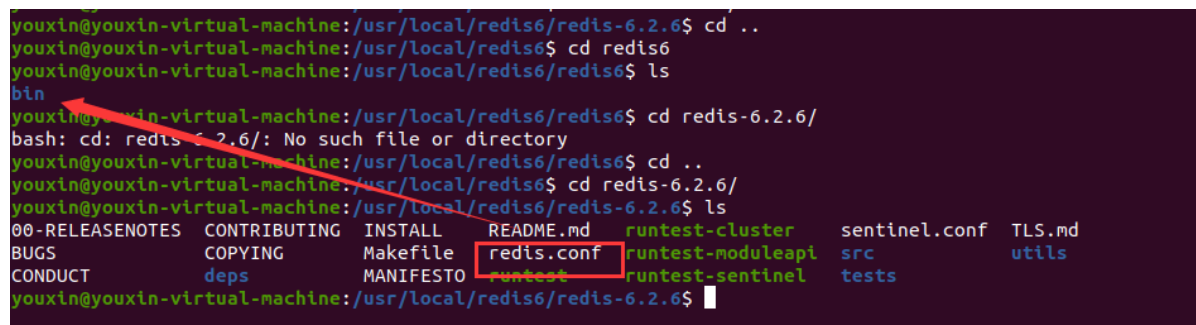
```
make test
```

如果测试全部通过，则证明上一步的make操作准确无误

执行安装命令：

```
sudo make PREFIX=/usr/local/redis6/redis6/ install
```

在redis的根目录下有一个配置文件redis.conf，把它考本到安装的redis目录，也就是上面指定的PREFIX文件夹：



```
youxin@youxin-virtual-machine:/usr/local/redis6/redis-6.2.6$ cd ..
youxin@youxin-virtual-machine:/usr/local/redis6$ cd redis6
youxin@youxin-virtual-machine:/usr/local/redis6/redis6$ ls
bin
youxin@youxin-virtual-machine:/usr/local/redis6/redis6$ cd redis-6.2.6/
bash: cd: redis-6.2.6/: No such file or directory
youxin@youxin-virtual-machine:/usr/local/redis6/redis6$ cd ..
youxin@youxin-virtual-machine:/usr/local/redis6$ cd redis-6.2.6/
youxin@youxin-virtual-machine:/usr/local/redis6/redis-6.2.6$ ls
00-RELEASENOTES  CONTRIBUTING  INSTALL      README.md    runtest-cluster  sentinel.conf  TLS.md
BUGS              COPYING      Makefile     redis.conf   runtest-moduleapi  src            utils
CONDUCT          DEPS         MANIFESTO    runtest      runtest-sentinel  tests
```

```
sudo cp redis.conf ../redis6/
```

配置环境变量：

```
sudo vim /etc/profile
```

在文件最后加入：

```
export REDIS_HOME=/usr/local/redis6/redis6/
```

```
export PATH=$PATH:$REDIS_HOME/bin
```

然后执行：source /etc/profile使之生效

启动redis服务：

```
redis-server /usr/local/redis6/redis6/redis.conf #如果没有指定配置文件，则会走默认的配置文  
件
```

启动redis客户端

```
redis-cli -p 6379 --raw
```

关闭redis服务：

```
shutdown
```

查看redis端口：

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

强行关闭端口：

```
kill -9 PID
```

性能测试

redis-benchmark是一个官方自带的压力测试工具

redis-benchmark测试命令参数：

redis 性能测试工具可选参数如下所示：

序号	选项	描述	默认值
1	-h	指定服务器主机名	127.0.0.1
2	-p	指定服务器端口	6379
3	-s	指定服务器 socket	
4	-c	指定并发连接数	50
5	-n	指定请求数	10000
6	-d	以字节的形式指定 SET/GET 值的数据大小	2
7	-k	1=keep alive 0=reconnect	1
8	-r	SET/GET/INCR 使用随机 key, SADD 使用随机值	
9	-P	通过管道传输 请求	1
10	-q	强制退出 redis。仅显示 query/sec 值	
11	--csv	以 CSV 格式输出	
12	*-l* (L 的小写字母)	生成循环，永久执行测试	
13	-t	仅运行以逗号分隔的测试命令列表。	
14	*-I* (i 的大写字母)	Idle 模式。仅打开 N 个 idle 连接并等待。	

简单测试：

```
redis-benchmark -h localhost -p 6379 -c 100 -n 100000 本地测试，端口6379，100个并发连接，100000个请求总共
```

```

===== GET =====
100000 requests completed in 1.27 seconds  十万个请求再1.27秒处理完
100 parallel clients  100个并发
3 bytes payload  每次传输3个字节
keep alive: 1  1台服务器
host configuration "save": 3600 1 300 100 60 10000
host configuration "appendonly": no
multi-thread: no

Latency by percentile distribution:
0.000% <= 0.127 milliseconds (cumulative count 1)
50.000% <= 0.575 milliseconds (cumulative count 50203)
75.000% <= 0.735 milliseconds (cumulative count 75036)
87.500% <= 0.911 milliseconds (cumulative count 87784)
93.750% <= 1.079 milliseconds (cumulative count 93883)
96.875% <= 1.263 milliseconds (cumulative count 96879)
98.438% <= 1.463 milliseconds (cumulative count 98456)
99.219% <= 1.751 milliseconds (cumulative count 99220)
99.609% <= 2.159 milliseconds (cumulative count 99614)
99.805% <= 2.511 milliseconds (cumulative count 99805)
99.902% <= 2.759 milliseconds (cumulative count 99904)
99.951% <= 2.911 milliseconds (cumulative count 99952)
99.976% <= 3.119 milliseconds (cumulative count 99976)
99.988% <= 3.311 milliseconds (cumulative count 99988)
99.994% <= 3.375 milliseconds (cumulative count 99994)
99.997% <= 3.407 milliseconds (cumulative count 99997)
99.998% <= 3.423 milliseconds (cumulative count 99999)

```

基础知识

redis默认有16个数据库

```

# crash-memcheck-enabled no

# Set the number of databases. The default database is DB 0, you can select
# a different one on a per-connection basis using SELECT <dbid> where
# dbid is a number between 0 and 'databases'-1
databases 16

```

默认使用的是第0个

可以使用select进行切换数据库

```

youxin@youxin-virtual-machine:/usr/local/redis6/redisyouxiyyouxin@yoyouxiyou
youxin@youxin-virtual-machine:/usr/local/redis6/redis6$ redis-cli
127.0.0.1:6379> select 4
OK
127.0.0.1:6379[4]>

```

切换到第四个数据库

dbsize 查看数据库大小

```

127.0.0.1:6379> dbsize
(integer) 4
127.0.0.1:6379>

```

keys * 查看数据库中所有的key

flushall 清除全部数据库

flushdb 清除当前数据库

```
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> dbsize
(integer) 0
127.0.0.1:6379> 
```

注意：redis是单线程的：redis是基于内存操作，CPU并不是Redis的性能瓶颈，redis的瓶颈是根据内存和网络带宽，既然可以使用单线程来实现，就使用单线程了

redis是c语言写的，官方的数据为10W+的QPS，这个不比Memcache差！

为什么redis单线程还这么快：高性能服务器不一定是多线程的，多线程的效率不一定比单线程效率高，redis是将所有数据全部放在内存中的，所以说使用单线程操作效率就是最高的，多线程（CPU会上下文切换：非常耗时），对于内存系统来说，如果没有上下文切换就是效率最高的~多次读写都是在一个CPU上的，在内存情况下，这个就是最佳方案。

五大数据类型

Redis 是一个开源（BSD许可）的，内存中的数据结构存储系统，它可以用作数据库、缓存和消息中间件。它支持多种类型的数据结构，如 [字符串 \(strings\)](#)，[散列 \(hashes\)](#)，[列表 \(lists\)](#)，[集合 \(sets\)](#)，[有序集合 \(sorted sets\)](#) 与范围查询，[bitmaps](#)，[hyperloglogs](#) 和 [地理空间 \(geospatial\)](#) 索引半径查询。Redis 内置了 [复制 \(replication\)](#)，[LUA脚本 \(Lua scripting\)](#)，[LRU 驱动事件 \(LRU eviction\)](#)，[事务 \(transactions\)](#) 和不同级别的 [磁盘持久化 \(persistence\)](#)，并通过 [Redis哨兵 \(Sentinel\)](#) 和自动 [分区 \(Cluster\)](#) 提供高可用性（high availability）。

Redis-Key

exists key 判断是否存在该key

move key 1移除key，1代表移除的数据库

```
127.0.0.1:6379> set name zhangsan
OK
127.0.0.1:6379> get name
"zhangsan"
127.0.0.1:6379> exists name
(integer) 1
127.0.0.1:6379> exists name01
(integer) 0
127.0.0.1:6379> move name 0
(error) ERR source and destination objects are the same
127.0.0.1:6379> move name 1
(integer) 1
127.0.0.1:6379> keys *
(empty array)
```

EXPIRE name 10 设置过期时间为10秒

ttl name 查看剩余时间

```
127.0.0.1:6379> EXPIRE name 10      设置过期时间
(integer) 1
127.0.0.1:6379> keys *
1) "name"
127.0.0.1:6379> keys *      10秒后为空
(empty array)
127.0.0.1:6379> 
```

type key 查看key的类型


```

OK
127.0.0.1:6379> set age 21
OK
127.0.0.1:6379> keys *
1) "name"
2) "age"
127.0.0.1:6379> type name
string
127.0.0.1:6379> type age
string
127.0.0.1:6379>

```

String (字符串类型)

```

#####
127.0.0.1:6379> set key1 "hello" # 设置值
OK
127.0.0.1:6379> get key1 #获取值
"hello"
127.0.0.1:6379> append key1 " world" # 追加值,如果该key不存在则新建一个字符串相当于set
key
(integer) 11
127.0.0.1:6379> get key1
"hello world"
127.0.0.1:6379> strlen key1 # 获取值的长度
(integer) 11
#####
127.0.0.1:6379> set views 0 #设置views初始值为0
OK
127.0.0.1:6379> get views
"0"
127.0.0.1:6379> INCR views # 设置自增1
(integer) 1
127.0.0.1:6379> get views
"1"
127.0.0.1:6379> incr views
(integer) 2
127.0.0.1:6379> get views
"2"
127.0.0.1:6379> decr views # 自减1
(integer) 1
127.0.0.1:6379> DECR views
(integer) 0
127.0.0.1:6379> get views
"0"
# 步长 i+=
127.0.0.1:6379> incrby views 10 #设置步长为10并增加
(integer) 10
127.0.0.1:6379> get views
"10"
#####
# 字符串范围range
127.0.0.1:6379> getrange key1 1 4 #截取字符串1-4
"ello"
127.0.0.1:6379> getrange key1 0 -1 # 截取全部的字符串
"hello world"
# 替换setrange
127.0.0.1:6379> setrange key1 0 nihao #从0位置开始替换

```

```

(integer) 11
127.0.0.1:6379> get key1
"nihao world"
#####
#setex(set with expire) # 设置过期时间
#setnx(set if not exist) # 不存在再设置（在分布式锁中会常常使用）
127.0.0.1:6379> setex key2 30 youxin #设置key2并在30秒后过期
OK
127.0.0.1:6379> ttl key2
(integer) 26
127.0.0.1:6379> ttl key2 #查看剩余时间
(integer) 21
127.0.0.1:6379> setnx key3 cloud # 如果key3不存在就创建，如果存在就创建失败
(integer) 1
127.0.0.1:6379> get key3
"cloud"
127.0.0.1:6379> setnx key3 rain
(integer) 0 # 再次创建失败
127.0.0.1:6379> keys *
1) "key3"
2) "views"
3) "key1"
127.0.0.1:6379> get key3 # 值没有改变
"cloud"
#####
# mset 同时设置多个值
# mget 同时获取多个值
127.0.0.1:6379> mset k1 v1 k2 v2 k3 v3 # 同时设置多个值
OK
127.0.0.1:6379> keys *
1) "k2"
2) "k3"
3) "k1"
127.0.0.1:6379> mget k1 k2 k3 # 同时获取多个值
1) "v1"
2) "v2"
3) "v3"
127.0.0.1:6379> msetnx k3 v3 k5 v5 #如果都不存在就设置多个值， msetnx是一个原子性的操作，要么一起成功，要么一起失败
(integer) 0
127.0.0.1:6379> keys *
1) "k2"
2) "k3"
3) "k1"
#####
# 对象
set user:1 {name:zhangsan,age=20} # 设置一个user:1对象，值为json字符来保存一个对象
# 这里的key是一个巧妙的设计: user:{id}:{field}
127.0.0.1:6379> mset user:1:name zhangsan user:1:age 22
OK
127.0.0.1:6379> mget user:1:name user:1:age
1) "zhangsan"
2) "22"
#####
getset #先get再set
127.0.0.1:6379> getset db mysql #如果不存在，则返回nil
(nil)
127.0.0.1:6379> get db

```

```
"mysql"
127.0.0.1:6379> getset db oracle #如果已经存在，返回原来的值并更新新的值
"mysql"
127.0.0.1:6379> get db
"oracle"
#####
```

String类似的使用场景：value除了是字符串还可以是数字

- 计数器
- 统计多单位数量
- 粉丝数
- 对象缓存存储!

List

在redis可以把list变成栈，队列，阻塞队列

所有的List命令都是以 **L** 开头的

```
#####
127.0.0.1:6379> lpush list1 one # 放置值，并从头部插入
(integer) 1
127.0.0.1:6379> lpush list1 two
(integer) 2
127.0.0.1:6379> lpush list1 three
(integer) 3
127.0.0.1:6379> lrange list1 0 -1 # 遍历所有
1) "three"
2) "two"
3) "one"
127.0.0.1:6379> lrange list1 0 1 # 通过具体的区间得到值
1) "three"
2) "two"
127.0.0.1:6379> rpush list1 four #从队列的尾部插入
(integer) 4
127.0.0.1:6379> lrange list1 0 -1
1) "three"
2) "two"
3) "one"
4) "four"
#####
#LPOP
#RPOP
127.0.0.1:6379> lpop list1 1 #弹出List1的第一个元素
1) "three"
127.0.0.1:6379> rpop list1 #弹出list1的最后一个元素
"four"
127.0.0.1:6379> lrange list1 0 -1
1) "two"
2) "one"
#####
# Lindex
127.0.0.1:6379> lindex list1 0 #获取下标为0的值
"two"
127.0.0.1:6379> lindex list1 1
"one"
```

```

127.0.0.1:6379> lpop list1
"two"
127.0.0.1:6379> lpop list1
"one"
127.0.0.1:6379> lindex list1 0 #当队列为空时返回nil
(nil)
#####
#Llen获取队列长度
127.0.0.1:6379> lpush list2 one
(integer) 1
127.0.0.1:6379> lpush list2 two
(integer) 2
127.0.0.1:6379> lpush list2 three
(integer) 3
127.0.0.1:6379> llen list2 #返回列表长度
(integer) 3
#####
#Lrem 移除指定的值, 精确匹配
127.0.0.1:6379> lpush list2 three
(integer) 4
127.0.0.1:6379> lrem list2 1 one #移除1个one
(integer) 1
127.0.0.1:6379> lrem list2 2 three #移除两个three
(integer) 2
127.0.0.1:6379> lrange list2 0 -1
1) "two"
#####
# trim 修剪: list 截断
127.0.0.1:6379> lpush list3 value1
(integer) 1
127.0.0.1:6379> lpush list3 value2
(integer) 2
127.0.0.1:6379> lpush list3 value3
(integer) 3
127.0.0.1:6379> lpush list3 value4
(integer) 4
127.0.0.1:6379> ltrim list3 1 2 # 截取下标为1到2的值
OK
127.0.0.1:6379> LRANGE list3 0 -1 #输出list3
1) "value3"
2) "value2"
#####
#rpoplpush 移除列表最后一个元素并添加到另一个元素
127.0.0.1:6379> RPOPLPUSH list3 list4
"value2"
127.0.0.1:6379> lrange list3 0 -1 #查看原来的列表
1) "value3"
127.0.0.1:6379> lrange list4 0 -1 #查看新的列表确实存在弹出来的值
1) "value2"
#####
#lset 指定下标替换值, 如果下标不存在则会报错
127.0.0.1:6379> exists list3 #判断是否存在list3
(integer) 1
127.0.0.1:6379> lset list3 0 value4 #替换list3中下标为0的值为value4
OK
127.0.0.1:6379> lrange list3 0 -1
1) "value4"
#####

```

```
# Linsert key after|before pivot value在指定位置之前或之后插入
127.0.0.1:6379> lrange list3 0 -1
1) "value4"
127.0.0.1:6379> LINSERT list3 before value4 value5 #在value4前面插入
(integer) 2
127.0.0.1:6379> lrange list3 0 -1
1) "value5"
2) "value4"
127.0.0.1:6379> LINSERT list3 after value4 value6 #在value4后面插入
(integer) 3
127.0.0.1:6379> lrange list3 0 -1
1) "value5"
2) "value4"
3) "value6"
#####
```

小结:

- 它实际上是一个链表, before node after, left, right都可以插入值
- 如果key不存在, 则创建新的链表
- 如果key存在, 新增内容
- 如果移除了所有值, 空链表, 也代表不存在
- 在两边插入或者改动值, 效率最高! 中间元素改动, 相对来说效率会低一点

消息队列: (LPUSH RPOP) 栈: (LPUSH LPOP)

Set (集合)

set中的值是不能重复的, 且是无序的

```
#####
# sadd添加元素
# smember获取所有元素
# sismember判断是否存在元素
127.0.0.1:6379> sadd set1 hello #set集合中添加元素
(integer) 1
127.0.0.1:6379> sadd set1 " world"
(integer) 1
127.0.0.1:6379> SMEMBERS set1
1) "hello"
2) " world"
127.0.0.1:6379> SISMEMBER set1 "hello" #判断元素是否在set中, 存在返回1, 不存在返回0
(integer) 1
127.0.0.1:6379> sismember set1 "he"
(integer) 0
#####
#scard获取set集合中元素个数
127.0.0.1:6379> scard set1
(integer) 2
127.0.0.1:6379> sadd set1 nihao
(integer) 1
127.0.0.1:6379> scard set1
(integer) 3
#####
#srem移除指定元素
127.0.0.1:6379> srem set1 nihao
(integer) 1
```

```

127.0.0.1:6379> scard set1
(integer) 2
127.0.0.1:6379> SMEMBERS set1
1) "hello"
2) " world"
127.0.0.1:6379> sadd set1 nihao
(integer) 1
127.0.0.1:6379> sadd set1 shijie
(integer) 1
127.0.0.1:6379> srem set1 nihao shijie #可以一次移除多个元素
(integer) 2
127.0.0.1:6379> SMEMBERS set1
1) "hello"
2) " world"
#####
#srandmember 随机获取指定个元素
127.0.0.1:6379> SRANDMEMBER set1
"hello"
127.0.0.1:6379> SRANDMEMBER set1
" world"
127.0.0.1:6379> SRANDMEMBER set1
" world"
127.0.0.1:6379> SRANDMEMBER set1 2
1) "hello"
2) " world"
#####
#spop随机删除Key
127.0.0.1:6379> sadd set1 youxin
(integer) 1
127.0.0.1:6379> spop set1
"youxin"
127.0.0.1:6379> SMEMBERS set1
1) "hello"
2) " world"
#####
#smove将一个指定的值移动到另外一个集合中
127.0.0.1:6379> smove set1 set2 hello
(integer) 1
127.0.0.1:6379> SMEMBERS set1
1) " world"
127.0.0.1:6379> SMEMBERS set2
1) "youxin"
2) "hello"
#####
#sdiff 差集
#sinter 交集
#sunion 并集
127.0.0.1:6379> sadd set3 a
(integer) 1
127.0.0.1:6379> sadd set3 b
(integer) 1
127.0.0.1:6379> sadd set3 c
(integer) 1
127.0.0.1:6379> sadd set4 c
(integer) 1
127.0.0.1:6379> sadd set4 d
(integer) 1
127.0.0.1:6379> sadd set4 f\

```

```

(integer) 1
127.0.0.1:6379> sdiff set3 set4
1) "b"
2) "a"
127.0.0.1:6379> SINTER set3 set4
1) "c"
127.0.0.1:6379> sunion set3 set4
1) "b"
2) "a"
3) "c"
4) "d"
5) "f\\"
应用场景：共同关注等
#####

```

Hash (Map集合)

Map集合, key-map! 这时候的值是一个map集合! 本质和string类型没有太大的区别, 还是一个简单的key-value

set hash field value

```

#####
127.0.0.1:6379> hset hash1 field1 youxin
(integer) 1
127.0.0.1:6379> hget hash1 field1
"youxin"
#####
#hmset设置多个值
#hmget获取多个值
#hgetall获取所有的值
127.0.0.1:6379> hmset hash1 field1 hello field2 world #如果已经存在则覆盖原来的值
OK
127.0.0.1:6379> hmget hash1 field1 field2
1) "hello"
2) "world"
127.0.0.1:6379> HGETALL hash1
1) "field1"
2) "hello"
3) "field2"
4) "world"
#####
#hdel删除某一个值, 对应的value值也删除了
127.0.0.1:6379> hdel hash1 field1
(integer) 1
127.0.0.1:6379> HGETALL hash1
1) "field2"
2) "world"
#####
#hlen 获取当前hash的长度
127.0.0.1:6379> hlen hash1
(integer) 1
127.0.0.1:6379> hset hash1 field3 value3
(integer) 1
127.0.0.1:6379> hlen hash1
(integer) 2
#####

```

```

#hexists判断对应的field和value是否存在
127.0.0.1:6379> HEXISTS hash1 field1
(integer) 0 #0代表不存在
127.0.0.1:6379> HEXISTS hash1 field3
(integer) 1 #1代表存在
#####
#hkeys 只获得所有的field
#hvals 只获得所有的value
127.0.0.1:6379> hkeys hash1
1) "field2"
2) "field3"
127.0.0.1:6379> hvals hash1
1) "world"
2) "value3"
#####
# hincrby 增加相应的步长，步长可以为负数（即减）
127.0.0.1:6379> HINCRBY hash1 field4 10
(integer) 16
127.0.0.1:6379> hget hash1 field4
"16"
127.0.0.1:6379> HINCRBY hash1 field4 -5
(integer) 11
127.0.0.1:6379> hget hash1 field4
"11"
#####
# hsetnx 如果不存在则新增，如果存在则新增失败
127.0.0.1:6379> hsetnx hash1 field5 nihao
(integer) 1
127.0.0.1:6379> hsetnx hash1 field5 nihao
(integer) 0
#####

```

hash应用场景：变更的数据hset user:1 name zhangsan; hset user:1 age 12。尤其是用户信息之类的，经常变动的信息！hash更适合于对象的存储，String更加适合字符串的存储。

Zset（有序集合）

在set的基础上，增加了一个值，set: set k1 v1, zset: zset k1 score1 v1

```

#####
127.0.0.1:6379> zadd zset1 1 one #新增一个值
(integer) 1
127.0.0.1:6379> zadd zset1 2 two 3 four #新增多个值
(integer) 2
127.0.0.1:6379> zrange zset1 0 -1 #正序输出全部，从小到大
1) "one"
2) "two"
3) "four"
127.0.0.1:6379> ZREVRANGE salary 0 -1 #倒序输出全部，从大到小
1) "lisi"
2) "wanger"
#####
#zrangebyscore 排序默认为升序
#zrevrangebyscore 倒序排序
127.0.0.1:6379> zadd salary 8000 zhangsan 9000 lisi 5000 wanger
(integer) 3

```



```

127.0.0.1:6379> ZRANGEBYSCORE salary -inf +inf 按工资排序, 排序范围为负无穷大到正无穷大
1) "wanger"
2) "zhangsan"
3) "lisi"
127.0.0.1:6379> ZRANGEBYSCORE salary -inf +inf withscores #附带成绩
1) "wanger"
2) "5000"
3) "zhangsan"
4) "8000"
5) "lisi"
6) "9000"
127.0.0.1:6379> ZREVRANGEBYSCORE salary +inf -inf withscores limit 0 3#倒序, limit
为分页
1) "lisi"
2) "9000"
3) "zhangsan"
4) "8000"
5) "wanger"
6) "5000"
#####
# zrem 移除元素
127.0.0.1:6379> zrem salary zhangsan
(integer) 1
127.0.0.1:6379> zrange salary 0 -1
1) "wanger"
2) "lisi"
#####
# zcard 获取有序集合元素个数
127.0.0.1:6379> zcard salary
(integer) 2 #返回元素个数
#####
# zcount 判断两个区间的值有几个
127.0.0.1:6379> zcount salary 2000 7000
(integer) 1
127.0.0.1:6379> zcount salary 2000 20000
(integer) 2
#####

```

应用场景: top 榜单等

三种特殊数据类型

bitmap (位图)

位存储: 场景: 统计用户信息, 活跃, 不活跃, 登陆, 未登录, 尤其是只有两个状态的

都是操作二进制为来进行记录, 就只有0和1两个状态!

```

#设置一周的打卡信息, 周一: 1, 打卡, 周二: 1. 打卡, 周三: 1, 打卡, 周四: 0, 未打卡。。。
127.0.0.1:6379> setbit sign 0 1
(integer) 0
127.0.0.1:6379> setbit sign 1 1
(integer) 0
127.0.0.1:6379> setbit sign 2 1
(integer) 0
127.0.0.1:6379> setbit sign 3 0
(integer) 0

```

```

127.0.0.1:6379> setbit sign 4 1
(integer) 0
127.0.0.1:6379> setbit sign 5 0
(integer) 0
127.0.0.1:6379> setbit sign 6 0
(integer) 0
#获取某一天打卡信息
127.0.0.1:6379> getbit sign 1
(integer) 1
127.0.0.1:6379> getbit sign 3
(integer) 0
#统计操作：统计打卡天数
127.0.0.1:6379> BITCOUNT sign
(integer) 4 #返回4天

```

hyperloglog (基数统计)

什么是基数：

A{1,3,4,5,6,8} B{1,3,5,6,8,9} 基数：不重复的元素 = 4, 9并且可以接受误差

Redis Hyperloglogs 基于基数统计的算法，优点：占用的内存是固定的， 2^{64} 个不同的元素，只需要12KB内存，如果要从内存角度比较的话，hyperloglogs首选。

例如：一个人访问一个网站多次，但还是算作一个人。

测试：

```

127.0.0.1:6379> PFADD key1 a b c d e f g g i j k l m n # 添加key1
(integer) 1
127.0.0.1:6379> PFCOUNT key1 # 输出key1不重复的个数
(integer) 13
127.0.0.1:6379> PFADD key2 a b c d e f g , f h # 添加key2
(integer) 1
127.0.0.1:6379> PFCOUNT key2 # 输出key2不重复的个数
(integer) 9
127.0.0.1:6379> PFMERGE key1 key2 # 将key2合并到key1中
OK
127.0.0.1:6379> PFCOUNT key1
(integer) 15
127.0.0.1:6379> PFCOUNT key2
(integer) 9
127.0.0.1:6379> PFMERGE key3 key1 key2 #将key1 和 key2一起合并到不存在的key3中
OK
127.0.0.1:6379> PFCOUNT key3
(integer) 15

```

geospatial (地理位置)

附近的人，打车距离计算等

redis的Geo在redis3.2版本就推出了！这个功能可以推算地理位置的信息，两地之间的距离，方圆半径的人等。

只有六个命令：

相关命令

- **GEOADD**
- **GEODIST**
- **GEOHASH**
- **GEOPOS**
- **GEORADIUS**
- **GEORADIUSBYMEMBER**

geoadd

#添加位置 注意：两级地区是不能直接添加的

参数: **key** 经度 纬度 城市名

```
127.0.0.1:6379> geoadd china:city 106.5 29.5 chongqin  
(integer) 1
```

```
127.0.0.1:6379> geoadd china:city 104.0 30.6 chengdu  
(integer) 1
```

```
127.0.0.1:6379> geoadd china:city 116.40 39.90 beijing 121.47 31.23 shanghai  
(integer) 2
```

geopos 查询具体位置的值

获取指定的城市的经度和纬度

```
127.0.0.1:6379> geopos china:city chengdu
```

```
1) 1) "104.00000184774398804"
```

```
2) "30.599999916504646222"
```

```
127.0.0.1:6379> geopos china:city beijing
```

```
1) 1) "116.39999896287918091"
```

```
2) "39.90000009167092543"
```

```
127.0.0.1:6379> geopos china:city chongqin
```

```
1) 1) "106.49999767541885376"
```

```
2) "29.50000115408581536"
```

geodist 返回两点之间的距离

如果两个位置之间的其中一个不存在，那么命令返回空值。

指定单位的参数 **unit** 必须是以下单位的其中一个：

- **m** 表示单位为米。
- **km** 表示单位为千米。
- **mi** 表示单位为英里。
- **ft** 表示单位为英尺。

```
127.0.0.1:6379> geodist china:city chongqin chengdu
```

```
"269990.9228"
```

```
127.0.0.1:6379> geodist china:city chongqin chengdu km
```

```
"269.9909"
```

```
127.0.0.1:6379> geodist china:city beijing shanghai km
```

```
"1067.3788"
```

`georadius` 以给定的经纬度为中心，返回键包含的位置元素当中，与中心的距离不超过给定最大距离的所有位置元素。

找附近的人：（获得所有附近的人的定位！），一般通过半径来查询

范围可以使用以下其中一个单位：

- **m** 表示单位为米。
- **km** 表示单位为千米。
- **mi** 表示单位为英里。
- **ft** 表示单位为英尺。

在给定以下可选项时，命令会返回额外的信息：

- `WITHDIST`：在返回位置元素的同时，将位置元素与中心之间的距离也一并返回。距离的单位和用户给定的范围单位保持一致。
- `WITHCOORD`：将位置元素的经度和维度也一并返回。
- `WITHHASH`：以 52 位有符号整数的形式，返回位置元素经过原始 geohash 编码的有序集合分値。这个选项主要用于底层应用或者调试，实际中的作用并不大。

命令默认返回未排序的位置元素。通过以下两个参数，用户可以指定被返回位置元素的排序方式：

- `ASC`：根据中心的位置，按照从近到远的方式返回位置元素。
- `DESC`：根据中心的位置，按照从远到近的方式返回位置元素。

在默认情况下，`GEORADIUS` 命令会返回所有匹配的位置元素。虽然用户可以使用 `COUNT <count>` 选项去获取前 N 个匹配元素，但是因为命令在内部可能会需要对所有被匹配的元素进行处理，所以在对一个非常大的区域进行搜索时，即使只使用 `COUNT` 选项去获取少量元素，命令的执行速度也可能会非常慢。但是从另一方面来说，使用 `COUNT` 选项去减少需要返回的元素数量，对于减少带宽来说仍然是非常有用的。

```
127.0.0.1:6379> GEORADIUS china:city 110 30 1000 km # 以110 30为中心，1000km范围内的城市
1) "chengdu"
2) "chongqin"
127.0.0.1:6379> GEORADIUS china:city 110 30 1000 km withcoord count 1 #限定只返回一个，并显示经纬度
1) 1) "chongqin"
   2) 1) "106.49999767541885376"
      2) "29.50000115408581536"
127.0.0.1:6379> GEORADIUS china:city 110 30 1000 km withdist count 1 #并显示直线距离
1) 1) "chongqin"
   2) "342.5131"
```

`georadiusbymember`

这个命令和 `GEORADIUS` 命令一样，都可以找出位于指定范围内的元素，但是 `GEORADIUSBYMEMBER` 的中心点是由给定的位置元素决定的，而不是像 `GEORADIUS` 那样，使用输入的经度和纬度来决定中心点

指定成员的位置被用作查询的中心。

```
127.0.0.1:6379> GEORADIUSBYMEMBER china:city beijing 3000 km #北京3000km半径内的城市
1) "chengdu"
2) "chongqin"
3) "shanghai"
4) "beijing"
```

geohash

返回一个或多个位置元素的 [Geohash](#) 表示。

通常使用表示位置的元素使用不同的技术，使用Geohash位置52点整数编码。由于编码和解码过程中所使用的初始最小和最大坐标不同，编码的编码也不同于标准。此命令返回一个**标准的Geohash**，在[维基百科](#)和[geohash.org](#)网站都有相关描述

该命令将返回11个字符的Geohash字符串，所以没有精度Geohash，损失相比，使用内部52位表示。返回的geohashes具有以下特性：

1. 他们可以缩短从右边的字符。它将失去精度，但仍将指向同一地区。
2. 它可以在 [geohash.org](#) 网站使用，网址 <http://geohash.org/<geohash-string>>。查询例子：<http://geohash.org/sqdr74hyu0>。
3. 与类似的前缀字符串是附近，但相反的是不正确的，这是可能的，用不同的前缀字符串附近。

```
127.0.0.1:6379> GEOHASH china:city beijing chongqin #将二维的经纬度转换为一维的字符串，如果两个字符串越像则越接近
1) "wx4fbxxfke0"
2) "wm5xz6qcvy0"
127.0.0.1:6379> GEOHASH china:city beijing chongqin shanghai
1) "wx4fbxxfke0"
2) "wm5xz6qcvy0"
3) "wtw3sj5zbj0"
```

geospatial的底层就是一个Zset有序集合

```
127.0.0.1:6379> zrange china:city 0 -1 #遍历
1) "chengdu"
2) "chongqin"
3) "shanghai"
4) "beijing"
```

事务

事务(ACID)的本质：一组命令的集合。且一个事务的所有命令都会被序列化，在事务的执行过程中，会按照顺序执行。

一致性，原子性，隔离性，持久性。

redis单条命令保证原子性，但是redis的事务是不保证原子性的。

redis事务没有隔离级别的概念，所有的命令在事务中，并没有直接被执行，只有发起执行命令的时候才会被执行！

redis事务阶段：

- 开启事务 (multi)
- 命令入队 (...)
- 执行事务 (exec)

正常执行事务

```
127.0.0.1:6379> multi #开启事务
OK
#命令入队
127.0.0.1:6379(TX)> set k1 v1
QUEUED
127.0.0.1:6379(TX)> set k2 v2
QUEUED
127.0.0.1:6379(TX)> get k2
QUEUED
127.0.0.1:6379(TX)> set k3 v3
QUEUED
127.0.0.1:6379(TX)> exec #执行事务
1) OK
2) OK
3) "v2"
4) OK
```

放弃事务 discard

```
127.0.0.1:6379> multi #开启事务
OK
127.0.0.1:6379(TX)> set k4 v4
QUEUED
127.0.0.1:6379(TX)> set k5 v5
QUEUED
127.0.0.1:6379(TX)> DISCARD # 放弃事务，事务队列中的命令不会执行
OK
127.0.0.1:6379> get k4
(nil)
```

编译型异常（代码有错，不能编译，所有命令都不会执行）

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set k1 v1
QUEUED
127.0.0.1:6379(TX)> set k2 v2
QUEUED
127.0.0.1:6379(TX)> getset k2 # 错误的命令，直接报错
(error) ERR wrong number of arguments for 'getset' command
127.0.0.1:6379(TX)> set k3 v3
QUEUED
127.0.0.1:6379(TX)> exec #执行事务也是报错的
(error) EXECABORT Transaction discarded because of previous errors.
127.0.0.1:6379> get k3 #所有的命令都不会执行
(nil)
```

运行时异常（语法有错，其他命令是可以正常执行，错误的命令会抛出异常）

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set k1 v1
QUEUED
```

```

127.0.0.1:6379(TX)> INCR k1 #语法错误
QUEUED
127.0.0.1:6379(TX)> set k2 v2
QUEUED
127.0.0.1:6379(TX)> set v3 k3
QUEUED
127.0.0.1:6379(TX)> exec
1) OK
2) (error) ERR value is not an integer or out of range #该条命令会报错，但事务还是执行成功了
3) OK
4) OK
127.0.0.1:6379> get k2 #其他命令都执行成功了
"v2"
127.0.0.1:6379> get v3
"k3"

```

监控：watch

悲观锁：

- 很悲观，认为什么时候都会出问题，无论做什么都会加锁！

乐观锁：

- 很乐观，认为什么时候都不会出现问题，所以不会加锁！更新数据的时候去判断一下，在此期间是否有人修改过数据，
- 获取version
- 更新的时候比较version

Redis监视测试

```

127.0.0.1:6379> set money 100
OK
127.0.0.1:6379> set out 0
OK
127.0.0.1:6379> watch money
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> DECRBY money 30
QUEUED
127.0.0.1:6379(TX)> INCRBY out 30
QUEUED
127.0.0.1:6379(TX)> exec
1) (integer) 70
2) (integer) 30
127.0.0.1:6379> watch money #监视money对象，相当于乐观锁
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> DECRBY money 10
QUEUED
127.0.0.1:6379(TX)> INCRBY out 10
QUEUED
127.0.0.1:6379(TX)> exec # 当上一个事务还没有执行完毕，后一个命令被执行了，由于上一个事务被上了锁，所以上一个事务会执行失败。
(nil)

```

```
#####
在exec之前执行：
127.0.0.1:6379> get money
"70"
127.0.0.1:6379> INCRBY money 100
(integer) 170
```

unwatch 如果发现事务执行失败，就先解锁，然后再次上锁

Jedis

使用java来操作redis

什么是Jedis

Jedis是官方推荐的java连接开发工具，使用java操作Redis中间件。

1、导入对应的依赖：

```
<!-- 导入Jedis的包 -->
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>3.6.0</version>
</dependency>

<!-- 导入fastjson -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.72</version>
</dependency>
```

2、编码测试

- 连接数据库
- 操作命令
- 断开连接

```
public class TestJedis {
    public static void main(String[] args) {
        //1、new jedis对象
        Jedis jedis = new Jedis("192.168.147.129",6379);
        //2、jedis的所有方法就是之前学的命令
        System.out.println(jedis.ping());
        //3、关闭连接
        jedis.close();
    }
}
```

想要远程连接redis需要将受保护模式选项设置为“no”，为了让服务器开始从外部接受连接：

redis-cli中输入：config set protected-mode "no"

输出：


```
testJedis x
"C:\Program Files\Java\jdk1.8.0_131\bin\java.exe" ...
PONG

Process finished with exit code 0
```

测试:

```
System.out.println("清空数据库: "+jedis.flushDB());
System.out.println("判断某个键是否存在: "+jedis.exists("k1"));
System.out.println("新增k1,v1键值对: "+jedis.set("k1","v1"));
System.out.println("新增k2,v2键值对: "+jedis.set("k2","v2"));
System.out.println("系统中所有的键如下: ");
Set<String> keys = jedis.keys("*");
System.out.println(keys);
System.out.println("删除键k2: "+jedis.del("k2"));
System.out.println("判断键k2是否存在: "+jedis.exists("k2"));
System.out.println("查看键 k1 所存储的数据类型: "+jedis.type("k1"));
System.out.println("随机返回 key 空间的一个: "+jedis.randomKey());
System.out.println("重命名 key : "+jedis.rename("k1","newk1"));
System.out.println("取出改后的 newk1 : "+jedis.get("newk1"));
System.out.println("按索引查询: "+jedis.select(0));
System.out.println("删除当前选择数据库中的所有键: "+jedis.flushDB());
System.out.println("返回当前数据库中 key 的数量: "+jedis.dbSize());
System.out.println("删除所有数据库中的所有 key : "+jedis.flushAll());
```

PONG
清空数据库: OK
判断某个键是否存在: false
新增k1, v1键值对: OK
新增k2, v2键值对: OK
系统中所有的键如下:
[k1, k2]
删除键k2: 1
判断键k2是否存在: false
查看键 k1 所存储的数据类型: string
随机返回 key 空间的一个: k1
重命名 key : OK
取出改后的 newk1 : v1
按索引查询: OK
删除当前选择数据库中的所有键: OK
返回当前数据库中 key 的数量: 0
删除所有数据库中的所有 key : OK

测试事务

```
package com.youxin;

import com.alibaba.fastjson.JSONObject;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.Transaction;

/**
 * @author youxin
 * @program redis-study
 * @description 测试事务
 * @date 2021-11-21 17:10
 */
public class TestTX {
    public static void main(String[] args) {
        Jedis jedis = new Jedis("192.168.147.129", 6379);
        JSONObject jsonObject = new JSONObject();
        jsonObject.put("name", "zhangsan");
        jsonObject.put("age", 23);
        jsonObject.put("gender", "man");
        //开启事务
        Transaction multi = jedis.multi();
        String string = jsonObject.toJSONString();
        try {
            multi.set("key1", string);
            multi.exec();
        } catch (Exception e) {
            //如果失败放弃事务
            multi.discard();
            e.printStackTrace();
        } finally {
            //关闭连接
            System.out.println(jedis.get("key1"));
            jedis.close();
        }
    }
}
```

```
"C:\Program Files\Java\jdk1.8.0_131\bin\java.exe" ...
{"gender":"man","name":"zhangsan","age":23}
```

```
Process finished with exit code 0
```

SpringBoot整合Redis

SpringBoot操作数据：spring-data jpa jdbc mongodb redis

SpringData也是和SpringBoot齐名的项目

说明：在SpringBoot2.x之后，原来使用的jedis被替换成了lettuce

jedis：采用的直连，多个线程操作的话，是不安全的，如果想要避免不安全的，使用jedis pool连接池。更像BIO模式。

lettuce：采用netty，实例可以在多个线程中共享，不存在线程不安全的情况。更像NIO模式。

源码分析：

```
@Bean
@ConditionalOnMissingBean(name = "redisTemplate")//可以自定义redisTemplate来替换默认的
@ConditionalOnSingleCandidate(RedisConnectionFactory.class)
public RedisTemplate<Object, Object> redisTemplate(RedisConnectionFactory redisConnectionFactory) {
    //默认的redisTemplate没有过多的设置，redis对象都是需要序列化
    //两个泛型都是Object，Object的类型，使用需要强制转换<String, Object>类型
    RedisTemplate<Object, Object> template = new RedisTemplate<>();
    template.setConnectionFactory(redisConnectionFactory);
    return template;
}

@Bean
@ConditionalOnMissingBean //由于String类型是redis中最常使用的类型，所以单独列出来
@ConditionalOnSingleCandidate(RedisConnectionFactory.class)
public StringRedisTemplate stringRedisTemplate(RedisConnectionFactory redisConnectionFactory) {
    return new StringRedisTemplate(redisConnectionFactory);
}
```

整合测试

1、导入依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

2、配置连接

```
#redis配置
spring:
  redis:
    host: 192.168.147.129
    port: 6379
    database: 0
```

3、测试

```
@SpringBootTest
class Redis02SpringbootApplicationTests {

    @Autowired
    RedisTemplate redisTemplate;

    @Test
```

```

void contextLoads() {
    //redisTemplate 操作不同的数据类型, api指令是一样的
    //    redisTemplate.opsForValue(); //操作字符串String
    //    redisTemplate.opsForList(); //操作List类型
    //    redisTemplate.opsForHash();
    //    redisTemplate.opsForSet();
    //    redisTemplate.opsForZSet();
    //除了基本的操作, 常用的方法可以直接通过redisTemplate操作, 比如事务, 和基本的CRUD
    //获取redis连接对象
    /*RedisConnection connection =
redisTemplate.getConnectionFactory().getConnection();
    connection.flushDb();
    connection.flushAll();*/
    redisTemplate.opsForValue().set("mykey1", "hello world");
    Object mykey1 = redisTemplate.opsForValue().get("mykey1");
    System.out.println(mykey1);
}

@Test
public void test01() throws JsonProcessingException {
    Person user = new Person(1, "张三");
    //开发一般使用json来传递对象
    ObjectMapper objectMapper = new ObjectMapper();
    //转换为json字符串
    String jsonUser = objectMapper.writeValueAsString(user);
    redisTemplate.opsForValue().set("user", jsonUser);
    System.out.println(redisTemplate.opsForValue().get("user"));
}
}

```

```

if (defaultSerializer == null) {
    //默认的序列化方式是jdk序列化
    defaultSerializer = new JdkSerializationRedisSerializer(
        classLoader != null ? classLoader : this.getClass().getClassLoader());
}

```

自定义RedisTemplate

```

package com.youxin.config;

import com.fasterxml.jackson.annotation.JsonAutoDetect;
import com.fasterxml.jackson.annotation.PropertyAccessor;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.boot.autoconfigure.condition.ConditionalOnSingleCandidate;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.serializer.Jackson2JsonRedisSerializer;
import org.springframework.data.redis.serializer.StringRedisSerializer;

```

```

/**
 * @author youxin
 * @program redis-study
 * @description
 * @date 2021-11-22 17:02
 */
@Configuration
public class RedisConfig {

    //自定义的RedisTemplate
    @Bean
    @SuppressWarnings("all")
    public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory
redisConnectionFactory) {
        RedisTemplate<String, Object> template = new RedisTemplate<String,
Object>();
        template.setConnectionFactory(redisConnectionFactory);
        // 序列化配置
        Jackson2JsonRedisSerializer jackson2JsonRedisSerializer = new
Jackson2JsonRedisSerializer(Object.class);
        ObjectMapper om = new ObjectMapper();
        om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
        om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
        jackson2JsonRedisSerializer.setObjectMapper(om);
        StringRedisSerializer stringRedisSerializer = new
StringRedisSerializer();
        // key采用String的序列化方式
        template.setKeySerializer(stringRedisSerializer);
        //hash的key也采用String的序列化方式
        template.setHashKeySerializer(stringRedisSerializer);
        //value采用jackson序列化
        template.setValueSerializer(jackson2JsonRedisSerializer);

        template.setHashValueSerializer(jackson2JsonRedisSerializer);
        template.afterPropertiesSet();
        return template;
    }
}

```

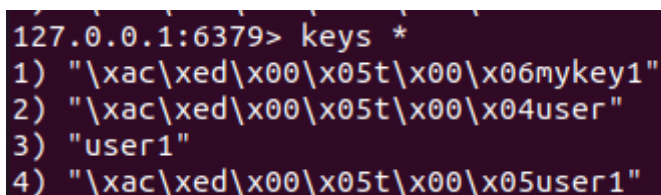
再次执行test02:

```

@Test
public void test02() {
    Person user1 = new Person(2, "李四");
    redisTemplate.opsForValue().set("user1",user1);
    System.out.println(redisTemplate.opsForValue().get("user1"));
}

```

再查看redis中所有key:



```

127.0.0.1:6379> keys *
1) "\xac\xed\x00\x05t\x00\x06mykey1"
2) "\xac\xed\x00\x05t\x00\x04user"
3) "user1"
4) "\xac\xed\x00\x05t\x00\x05user1"

```

可以看到返回的结果是一样的，但是在redis中被序列化了。

但发现在get对象的时候还是中文没有转义：

```
127.0.0.1:6379> get user1
"[\com.youxin.pojo.Person\",{\id\":2,\"name\":\"\xe6\xe9\xe5\x9b\x9b\"}]"
```

这时可以在打开redis-cli时用： `redis-cli --raw`

再次获取不会有问题：

```
youxin@youxin-virtual-machine:~$ redis-cli --raw
127.0.0.1:6379> get user1
["com.youxin.pojo.Person",{"id":2,"name":"李四"}]
```

redis.conf

单位换算

```
#
# 1k => 1000 bytes
# 1kb => 1024 bytes
# 1m => 1000000 bytes
# 1mb => 1024*1024 bytes
# 1g => 1000000000 bytes
# 1gb => 1024*1024*1024 bytes
#
# units are case insensitive so 1GB 1Gb 1gB are all the same.
```

includes可以包含其他配置文件

```
#
# If instead you are interested in using includes to override configur
# options, it is better to use include as the last line.
#
# include /path/to/local.conf
# include /path/to/other.conf
```

网络 NETWORK

```
bind 127.0.0.1 -::1 #绑定端口为127.0.0.1
protected-mode yes #保护模式，默认为开启
port 6379 #端口默认为6379

tcp-backlog 511 #发量大并且客户端速度缓慢的时候，建议修改值大于511。
```

GENERAL 通用设置

```
daemonize yes #后台守护模式。默认为关闭，一般设置为开启
pidfile /var/run/redis_6379.pid #如果以后台的方式运行，就需要指定一个pid文件

loglevel notice # 日志默认为notice级别
#其他级别
# Specify the server verbosity level.
# This can be one of:
# debug (a lot of information, useful for development/testing)
# verbose (many rarely useful info, but not a mess like the debug level)
# notice (moderately verbose, what you want in production probably)
```

```
# warning (only very important / critical messages are logged)

logfile /usr/local/redis6/redis6/log-redis.log #指定日志文件位置

always-show-logo no #是否总是显示logo
```

SNAPSHOTTING 快照

持久化，在规定的时间内，执行了多少次操作，则会持久化到文件.rdb

```
#如果在3600s内，如果至少有一个key进行了修改，就及时对数据库进行持久化操作
# save 3600 1
# save 300 100
# save 60 10000
stop-writes-on-bgsave-error yes #持久化如果出错，是否继续持久化

rdbcompression yes #是否压缩rdb文件，需要消耗一些cpu资源

rdbchecksum yes #保存rdb文件的时候，进行错误的检查校验

dbfilename dump.rdb #rdb文件名默认为dump.rdb

rdb-del-sync-files no # 在没有持久性的情况下删除复制中使用的RDB文件，通常情况下保持默认即可

dir /usr/local/redis6/redis6/redis_dbfiles/ #rdb文件保存的目录，默认是当前目录
```

REPLICATION 主从复制

SECURITY 安全

```
acllog-max-len 128 #ACL日志的最大长度，默认是128M。关于acl，详见：
https://redis.io/topics/acl

# aclfile /etc/redis/users.acl #ACL外部配置文件所在位置

# requirepass foobared #当前redis服务的访问密码，默认是不需要密码
```

CLIENTS 客户端

客户端最大连接数配置默认是10000。

```
# maxclients 10000
```

MEMORY MANAGEMENT 内存管理

指定Redis最大内存限制。达到内存限制时，Redis将尝试删除已到期或即将到期的Key。

```
# maxmemory <bytes>

# volatile-lru -> 对设置了过期时间的keys适用LRU淘汰策略
# allkeys-lru -> 对所有keys适用LRU淘汰策略
# volatile-lfu -> 对设置了过期时间的keys适用LFU淘汰策略
# allkeys-lfu -> 对所有keys适用LFU淘汰策略
# volatile-random -> 对设置了过期时间的keys适用随机淘汰策略
```

```
# allkeys-random -> 对所有keys适用随机淘汰策略
# volatile-ttl -> 淘汰离过期时间最近的keys
# noeviction -> 不淘汰任何key，仅对写入操作返回一个错误
```

内存策略，默认是noeviction

```
# maxmemory-policy noeviction
```

#其他5中方式

```
#volatile-lru: 只对设置了过期时间的key进行LRU（默认值）
```

```
#allkeys-lru : 删除lru算法的key
```

```
#volatile-random: 随机删除即将过期key
```

```
#allkeys-random: 随机删除
```

```
#volatile-ttl : 删除即将过期的
```

```
#noeviction : 永不过期，返回错误
```

LRU,LFU,minimal TTL 算法都不是精准的算法，这里设置抽查的样本数量，默认是5个样本。

```
# maxmemory-samples 5
```

从 Redis 5 开始，默认情况下，replica 节点会忽略 maxmemory 设置（除非在发生 failover 后，此节点被提升为 master 节点）。这意味着只有 master 才会执行过期删除策略，并且 master 在删除键之后会对 replica 发送 DEL 命令。

```
# replica-ignore-maxmemory yes
```

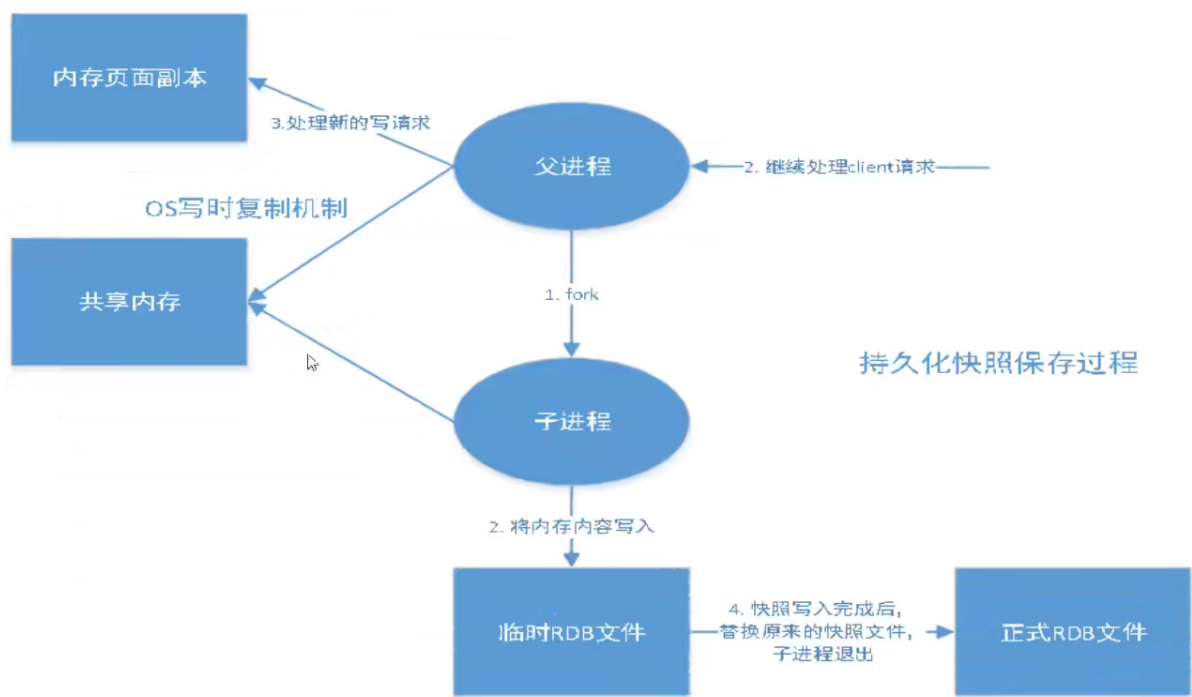
设置过期keys仍然驻留在内存中的比重，默认是为1，表示最多只能有10%的过期key驻留在内存中，该值设置的越小，那么在一个淘汰周期内，消耗的CPU资源也更多，因为需要实时删除更多的过期key。所以该值的配置是需要综合权衡的。

```
# active-expire-effort 1
```

Redis持久化

Redis是内存数据库，如果不将内存中的数据库状态保存到磁盘，那么一旦服务器进程退出，服务器中的数据库状态也会消失，所以Redis提供了持久化功能！

RDB (Redis DataBase)



在指定的时间间隔内将内存中的数据快照写入磁盘，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。

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。

rdb保存的文件是dump.rdb

触发机制

- 1、save的规则满足的情况下, 会自动触发rdb规则
- 2、执行flushall命令, 也会触发rdb规则
- 3、退出redis, 也会产生一个rdb文件

备份就自动生成一个dump.rdb

修改rdb配置策略:

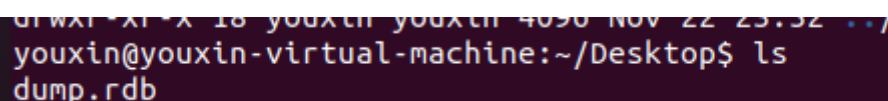
```
# save 3600 1
# save 300 100
# save 60 10000
save 60 3 #60秒内修改3条数据就会save
```

删除原来的sump.rdb : `sudo rm -rf dump.rdb`

60s内保存三条数据:

```
127.0.0.1:6379> set k1 vq
OK
127.0.0.1:6379> set k2 v2
OK
127.0.0.1:6379> set k3 v3
OK
```

在dir默认的目录下生成dump.rdb文件



```
youxin@youxin-virtual-machine:~/Desktop$ ls
dump.rdb
```

如何恢复rdb文件

- 1、只需要将rdb文件放在redis启动目录就可以, redis启动的时候会自动检查dump.rdb恢复其中的数据
- 2、查看需要存在的位置

```
127.0.0.1:6379> config get dir
1) "dir"
2) "/usr/local/redis6/redis6/bin" #如果在这个目录下存在dump.rdb文件, 启动就会自动恢复其中的数据
```

优点:

- 适合大规模的数据恢复
- 如果对数据的完整性要求不高

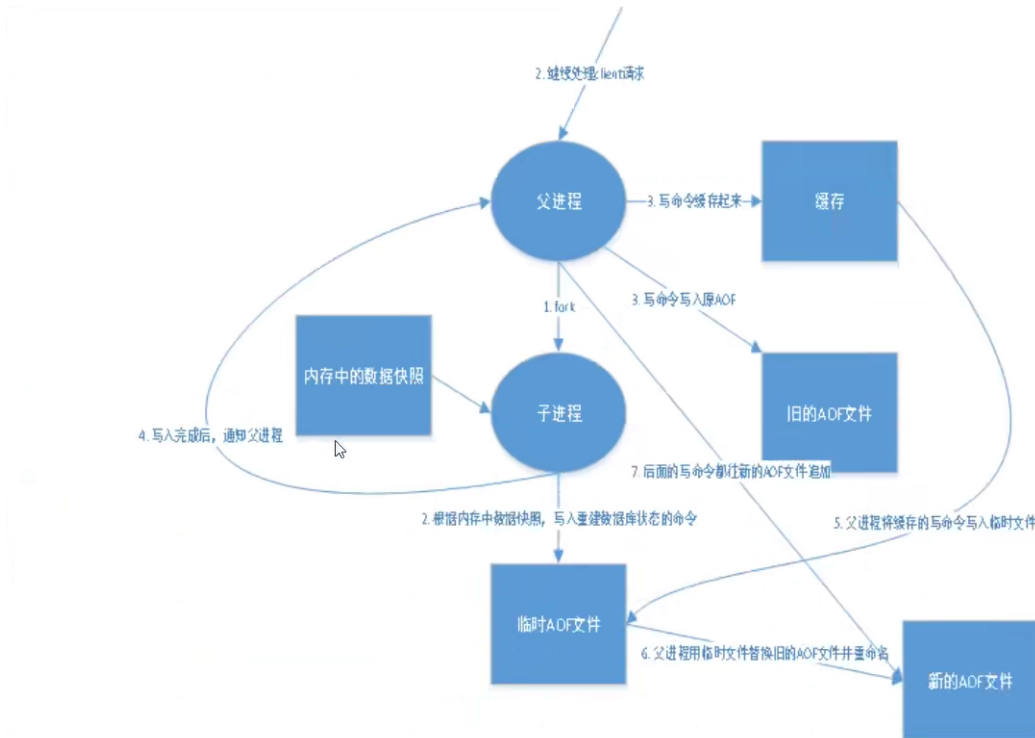
缺点:

- 需要一定的时间间隔进程操作, 如果redis意外宕机了, 最后一次修改的数据就没有了

- fork进程的时候，会占用一定的空间

AOF (Append Only File)

将所有的命令都记录下来，恢复的时候就把这个文件全部都重新执行一遍。



以日志的形式来记录每个写操作，将Redis执行过的所有指令记录下来(读操作不记录)，只许追加文件但不可以改写文件，redis启动之初会读取该文件重新构建数据，换言之，redis重启的话就根据日志文件的内容将写指令从前到后执行一次以完成数据的恢复工作

AOF保存的是appendonly.aof文件

配置，默认是不开启的：

```

##### APPEND ONLY MODE #####

# By default Redis asynchronously dumps the dataset on disk. This mode is
# good enough in many applications, but an issue with the Redis process or
# a power outage may result into a few minutes of writes lost (depending on
# the configured save points).
#
# The Append Only File is an alternative persistence mode that provides
# much better durability. For instance using the default data fsync policy
# (see later in the config file) Redis can lose just one second of writes in a
# dramatic event like a server power outage, or a single write if something
# wrong with the Redis process itself happens, but the operating system is
# still running correctly.
#
# AOF and RDB persistence can be enabled at the same time without problems.
# If the AOF is enabled on startup Redis will load the AOF, that is the file
# with the better durability guarantees.
#
# Please check https://redis.io/topics/persistence for more information.

appendonly no

# The name of the append only file (default: "appendonly.aof")

appendfilename "appendonly.aof"
  
```

只需要将appendonly开启为yes就可以开启aof

重启redis就可以生效:

```
redis-server /usr/local/redis6/redis6/redis.conf
```

```
youxin@youxin-virtual-machine:/usr/local/redis6/redis6$ cd bin
youxin@youxin-virtual-machine:/usr/local/redis6/redis6/bin$ ls
appendonly.aof  redis-benchmark  redis-check-rdb  redis-sentinel
dump.rdb        redis-check-aof  redis-cli        redis-server
```

如果aof文件有错误,这时候redis是启动不起来的,我们需要修复aof配置文件,redis提供了修复工具,redis-check-aof --fix appendonly.aof位置

如果修改appendonly.aof文件内容并删除dump.rdb文件,则启动redis会失败:

```
youxin@youxin-virtual-machine:~$ redis-server /usr/local/redis6/redis6/redis.conf
youxin@youxin-virtual-machine:~$ ps -ef|grep redis
youxin      3304      2160  0 00:52 pts/0    00:00:00 grep --color=auto redis
```

此时修复appendonly.aof文件:

```
redis-check-aof --fix appendon
```

```
youxin@youxin-virtual-machine:/usr/local/redis6/redis6/bin$ redis-check-aof --fix appendonly.aof
0x      6a: Expected \r\n, got: 6571
AOF analyzed: size=116, ok_up_to=81, ok_up_to_line=26, diff=35
This will shrink the AOF from 116 bytes, with 35 bytes, to 81 bytes
Continue? [y/N]: y
Successfully truncated AOF
youxin@youxin-virtual-machine:/usr/local/redis6/redis6/bin$ cat appendonly.aof
*2
$6
SELECT
$1
0
*3
$3
set
$2
k1
$2
v1
*3
$3
set
$2
k2
$2
v2
```

再次启动则启动成功:

```
youxin@youxin-virtual-machine:~$ redis-server /usr/local/redis6/redis6/redis.conf
youxin@youxin-virtual-machine:~$ ps -ef|grep redis
youxin      3657      1604  0 00:57 ?        00:00:00 redis-server *:6379
youxin      3663      2160  0 00:57 pts/0    00:00:00 grep --color=auto redis
```

优点:

- 1、每一次修改都会同步,文件的完整性会更加好
- 2、默认为每秒同步一次,可能会丢失一秒的数据
- 3、如果开启总是同步,则不会丢失数据

缺点:

- 1、相对于数据文件来说，aof总是大于rdb文件，修复速度也比rdb慢
- 2、aof运行效率也比rdb慢，所以redis默认为rdb

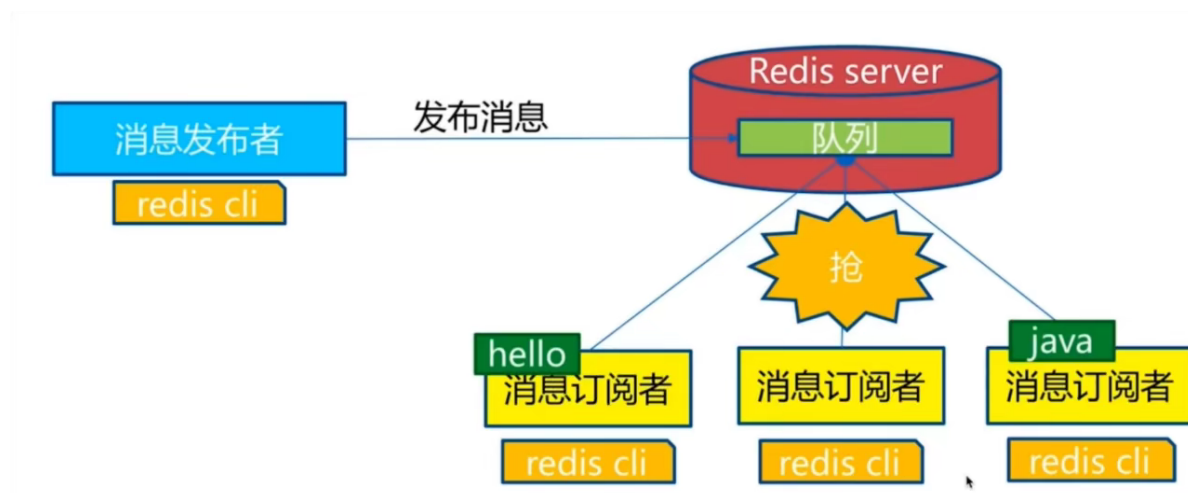
Redis发布订阅

Redis发布订阅 (pub/sub) 是一种消息通信模式：发送者(pub)发送消息，订阅者(sub)接收消息。

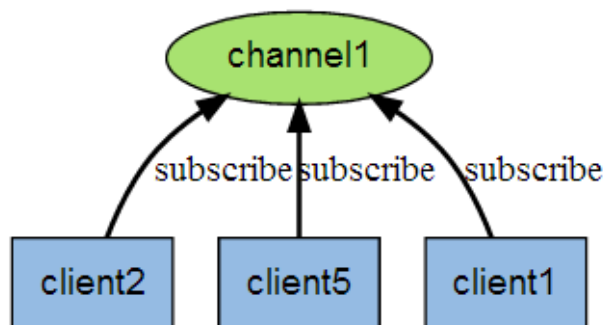
Redis客户端可以订阅任意数量的频道。

角色1：消息发送者 角色2：频道 角色3：消息订阅者

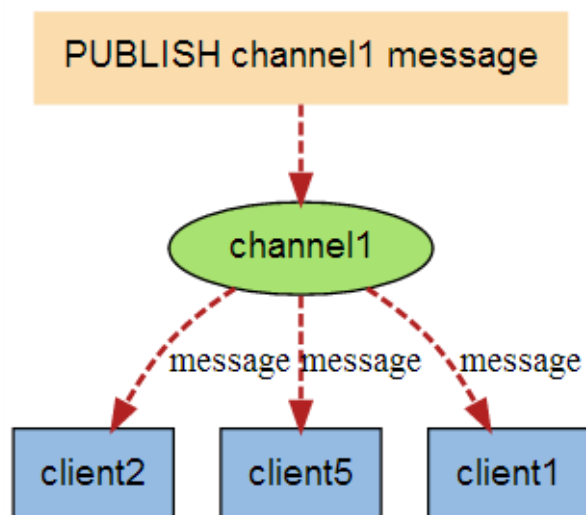
订阅/发布消息图：



下图展示了频道 channel1，以及订阅这个频道的三个客户端 —— client2、client5 和 client1 之间的关系：



当有新消息通过 PUBLISH 命令发送给频道 channel1 时，这个消息就会被发送给订阅它的三个客户端：



https://blog.csdn.net/weixin_44187730

Redis 发布订阅命令

下表列出了 redis 发布订阅常用命令：

序号	命令及描述
1	[PSUBSCRIBE pattern pattern ...] 订阅一个或多个符合给定模式的频道。
2	PUBSUB subcommand [argument [argument ...]] 查看订阅与发布系统状态。
3	PUBLISH channel message 将信息发送到指定的频道。
4	PUNSUBSCRIBE [pattern [pattern ...]] 退订所有给定模式的频道。
5	[SUBSCRIBE channel channel ...] 订阅给定的一个或多个频道的信息。
6	UNSUBSCRIBE [channel [channel ...]] 指退订给定的频道。

测试：

```
#订阅者
127.0.0.1:6379> SUBSCRIBE youxin #订阅对象
subscribe
youxin
1
message
youxin
this is my first test
```

```
#发布者
127.0.0.1:6379> PUBLISH youxin "this is my first test" #发布，发布对象，发布信息
1
```

原理

Redis是使用C实现的，通过分析Redis源码里面的pubsub.c文件，了解发布和订阅机制的底层实现，即此加深对Redis的理解。

Redis通过 PUBLISH、SUBSCRIBE 和 PSUBSCRIBE 等命令实现发布和订阅功能。

通过 SUBSCRIBE 命令订阅某频道后，redis-server 里维护一个字典，字典的键就是一个个channel，而字典的值则是一个链表，链表中保存了所有订阅这个channel的客户端。SUBSCRIBE命令的关键，就是将客户端添加到给定 channel 的订阅链表中。

通过 PUBLISH 命令向订阅者发送消息，redis-server 会使用给定的频道作为键，在它所维护的 channel 字典中查找记录了订阅这个频道的所有客户端的链表，遍历这个链表，将消息发布给所有订阅者。

Pub/Sub 从字面上理解就是发布（Publish）与订阅（Subscribe），在Redis中，你可以设定对某一个key值进行消息发布及消息订阅，当一个key值上进行了消息发布后，所有订阅它的客户端都会收到相应的消息。这一功能最明显的用法就是用作实时消息系统，比如普通的即时聊天，群聊等功能。

Redis主从复制

概念

主从复制，是指将一台Redis服务器的数据，复制到其他的Redis服务器。前者称为主节点(master/leader)，后者称为从节点(slave/follower);数据的复制是单向的，只能由主节点到从节点。Master以写为主，Slave以读为主。

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

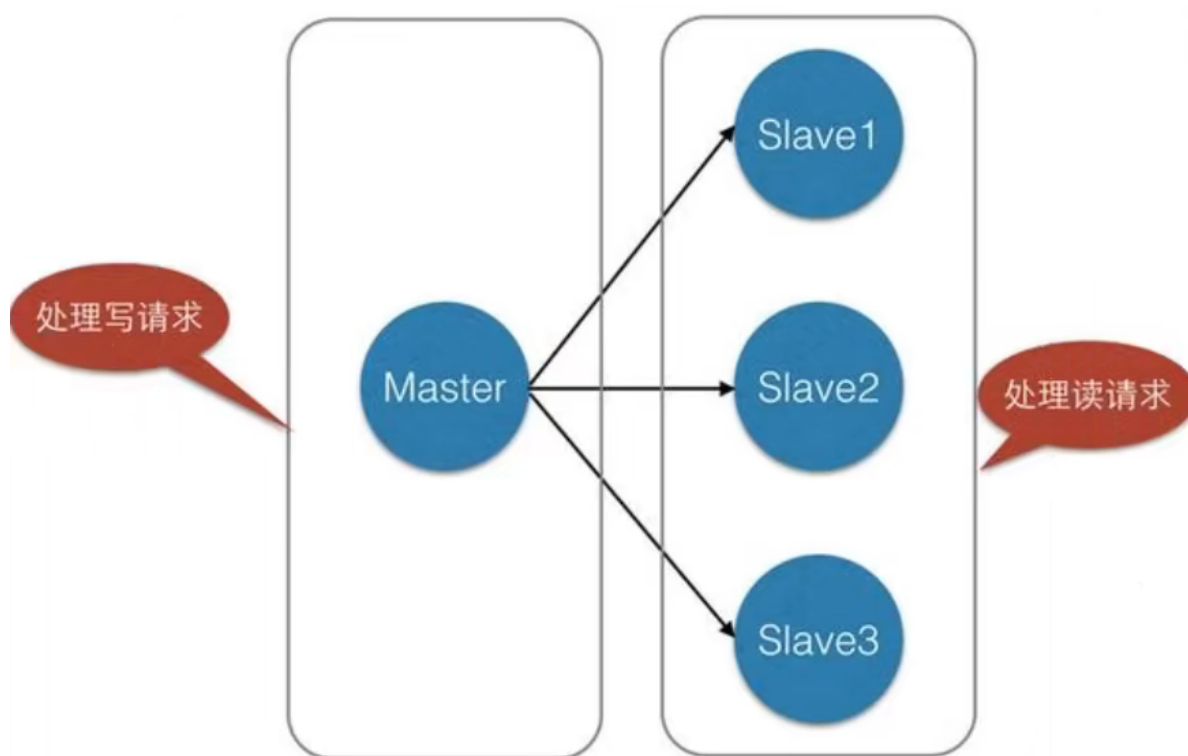
主从复制的作用主要包括：

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

一般来说，要将redis运用于工程项目中，只用一台Redis是万万不能的，原因如下：

- 1、从结构上，单个Redis服务器会发生单点故障，并且一台服务器需要处理所有的请求负载，压力较大；
- 2、从容量上，单个Redis服务器内存容量有限，就算一台服务区容量为256G，也不能将所有内存用作Redis存储内存，一般来说，单个Redis最大使用内存不应该超过20G。

电商网站上的商品，一般都是一次上传，无数次浏览的，也就是“多读少写”



环境配置

只配置从库，不用配置主库！

- 1、复制三份redis.conf
- 2、修改每一份中的配置端口和文件路径等
- 3、启动每一个redis
- 4、查看：

```
youxin@youxin-virtual-machine:/usr/local/redis6/redis6$ ps -ef|grep redis
youxin      7696      1544    0 00:07 ?        00:00:00 redis-server *:6379
youxin      7738      1544    0 00:08 ?        00:00:00 redis-server *:6380
youxin      7762      1544    0 00:08 ?        00:00:00 redis-server *:6381
youxin      7768      2114    0 00:08 pts/0    00:00:00 grep --color=auto redis
```

查看配置库：

```
127.0.0.1:6379> info replication #查看当前库的信息
# Replication
role:master #角色, master
connected_slaves:0 # 从机为0
master_failover_state:no-failover
master_replid:ca8f4258cac7d719392e4cc50624b0b3bde24501
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:0
second_repl_offset:-1
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
```

一主二从

一般情况下只用配置从机就好了

```
# slaveof
slaveof 127.0.0.1 6379
127.0.0.1:6380> SLAVEOF 127.0.0.1 6379
OK
127.0.0.1:6380> info replication
# Replication
role:slave #角色从机
master_host:127.0.0.1 #主机ip
master_port:6379 #主机端口
master_link_status:up
master_last_io_seconds_ago:4
master_sync_in_progress:0
slave_read_repl_offset:14
slave_repl_offset:14
slave_priority:100
slave_read_only:1
replica_announced:1
connected_slaves:0
master_failover_state:no-failover
```

而在真实的工作配置中，是将主从配置配置在配置文件中的，所以在配置文件中：


```
##### REPLICATION #####  
  
# replicaof <masterip> <masterport> #配置主机的ip和端口  
  
# If the master is password protected (using the "requirepass" configuration  
# directive below) it is possible to tell the replica to authenticate before  
# starting the replication synchronization process, otherwise the master will  
# refuse the replica request.  
#  
# masterauth <master-password> #如果主机设置了密码配置主机密码
```

细节

主机可以写，从机不能写，只能读操作！主机中的所有信息和数据，都会自动被从机保存。

主机可以写和读：

```
127.0.0.1:6379> set mykey hello  
OK  
127.0.0.1:6379> keys *  
mykey  
127.0.0.1:6379> get mykey  
hello
```

从机只能读：

```
127.0.0.1:6380> set k2 v2  
READONLY You can't write against a read only replica. #不能够写入  
  
127.0.0.1:6380> get mykey  
hello
```

测试：主机断开连接，从机依然连接到主机，但是没有写操作，这时如果主机重新连接，从机依旧可以直接获取到主机信息！

如果是使用了命令行配置了主从信息，这个时候如果重启了从机，从机就会变成了主机。而如果是在配置文件中配置了主从信息，重启从机后依旧还是从机，还是可以从主机中获取值，即只要重新连接一次主机就会被执行一次全量复制。

复制原理

slave启动成功连接到master后会发送一个sync同步命令

master接到命令，启动后台的存盘进程，同时收集所有接收到的用于修改数据集命令，在后台进程执行完毕之后，master将传送整个数据文件到slave，并完成一次完全同步。

全量复制：slave服务再接收到数据库文件数据后，将其存盘并加载到内存中。

增量复制：master继续将新的所有收集到的修改命令一次传给slave，完成同步。

但是只要是重新连接master，一次完全同步（全量复制）将被自动执行。

除了两个从机都连接一个主机，还可以将一个从机连接一个主机，而另一个从机的主机设置为前一个从机，这样的效果与前面是一样的，也可以完成主从复制

如果主机断开了，能不能重新选一个主机呢

slaveof no one #如果主机断开了连接，可以使自己变成主机，其他的节点就可以手动连接这个主节点，就算原来的主机重新连接后也没有用了，该从机还是作为主机了。

#####断开主机#####

127.0.0.1:6379> SHUTDOWN

#####查看从机配置信息#####

127.0.0.1:6380> info replication

Replication

role:slave

master_host:127.0.0.1

master_port:6379

master_link_status:down

master_last_io_seconds_ago:-1

master_sync_in_progress:0

slave_read_repl_offset:4035

slave_repl_offset:4035

master_link_down_since_seconds:19

slave_priority:100

slave_read_only:1

replica_announced:1

connected_slaves:0

master_failover_state:no-failover

master_replid:e7646763fdca473309caf87775fbc20b537fd110

master_replid2:00

master_repl_offset:4035

second_repl_offset:-1

repl_backlog_active:1

repl_backlog_size:1048576

repl_backlog_first_byte_offset:2523

repl_backlog_histlen:1513

#此时没有主机

#####将6380端口作为主机#####

127.0.0.1:6380> info replication

Replication

role:master

connected_slaves:0

master_failover_state:no-failover

master_replid:44fb1472c499cb87860cd64c34ac4be7eb0473ab

master_replid2:e7646763fdca473309caf87775fbc20b537fd110

master_repl_offset:4035

second_repl_offset:4036

repl_backlog_active:1

repl_backlog_size:1048576

repl_backlog_first_byte_offset:2523

repl_backlog_histlen:1513

#####启动6379端口#####

#此时6380端口还是作为了主机

redis-server /usr/local/redis6/redis6/redis-79.conf

127.0.0.1:6379> info replication

Replication

role:master

connected_slaves:1

slave0:ip=127.0.0.1,port=6381,state=online,offset=14,lag=1

master_failover_state:no-failover

master_replid:96c10b1f28db707d87af81c2c654f77adeae87ab

master_replid2:00

master_repl_offset:14

second_repl_offset:-1

repl_backlog_active:1

```
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:14
```

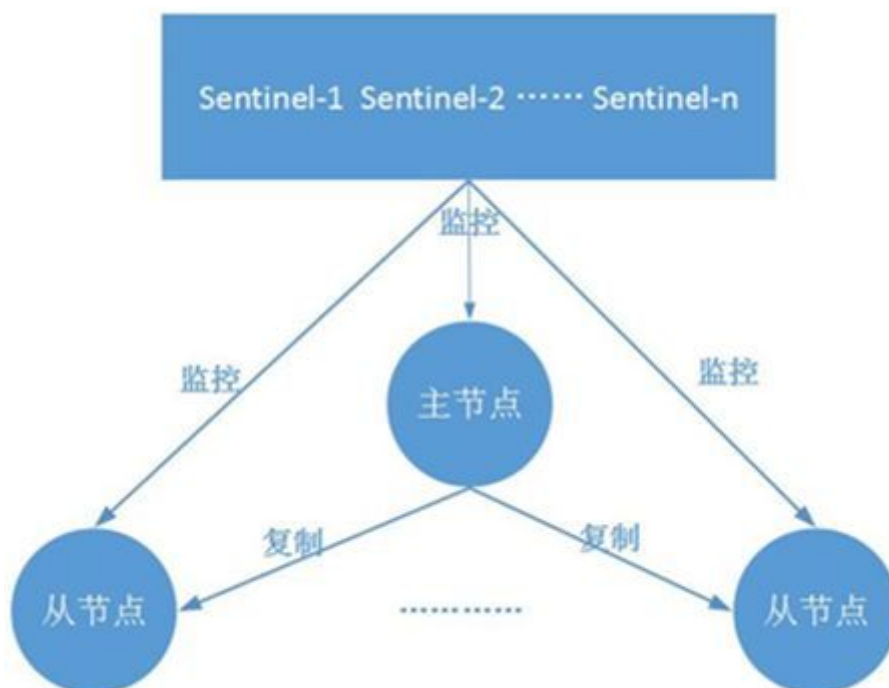
哨兵模式（自动选取主机模式）

概述

主从切换技术的方法是：当主服务器宕机后，需要手动把一台从服务器切换为主服务器，这就需要人工干预，费事费力，还会造成一段时间内服务不可用。这不是一种推荐的方式，更多时候，我们优先考虑哨兵模式。Redis从2.8开始正式提供了Sentinel（哨兵）架构来解决这个问题。

谋朝篡位的自动版，能够后台监控主机是否故障，如果故障了根据投票数自动将从库转换为主库。

哨兵模式是一种特殊的模式，首先Redis提供了哨兵的命令，哨兵是一个独立的进程，作为进程，它会独立运行。其原理是哨兵通过发送命令，等待Redis服务器响应，从而监控运行的多个Redis实例。



哨兵有两个作用：

- 通过发送命令，让redis服务器返回监控其运行状态，包括主服务器和从服务器
- 当哨兵检测到master宕机，会自动将slave切换成master，然后通过发布订阅模式通知其他的从服务器，修改配置文件，让他们切换主机。

当一个哨兵进程对redis服务器进行监控，可能会出现这个问题，为此可以使用多个哨兵监控，各个哨兵之间还会进行监控，这样就形成了多哨兵模式。

假设主服务器宕机，哨兵1先检测到这个结果，系统并不会马上进行failover过程，仅仅是哨兵1主观的认为主服务器不可用，这个现象成为主观下线。当后面的哨兵也检测到主服务器不可用，并且数量达到一定值时，那么哨兵之间就会进行一次投票，投票的结果由一个哨兵发起，进行failover[故障转移]操作。切换成功后，就会通过发布订阅模式，让各个哨兵把自己监控的从服务器实现切换主机，这个过程称为客观下线。

测试

1、配置哨兵配置文件sentinel.conf

```
#sentinel monitor 被监控的名称 host port 1
sentinel monitor mysentinel 127.0.0.1 6379 1 #后面的数字1代表主机如果挂了，slave投票让谁接替主机，票数最多的，就变成主机。
```

2、启动哨兵：

```
sudo ./redis-sentinel /usr/local/redis6/redis6/sentinel.conf
```

```
2725:X 24 Nov 2021 09:08:40.315 # Redis version=6.2.6, bits=64, commit=00000000, modified=0, pid=2725, j
ust started
2725:X 24 Nov 2021 09:08:40.315 # Configuration loaded
2725:X 24 Nov 2021 09:08:40.316 * Increased maximum number of open files to 10032 (it was originally set
to 1024).
2725:X 24 Nov 2021 09:08:40.316 * monotonic clock: POSIX clock_gettime

Redis 6.2.6 (00000000/0) 64 bit

Running in sentinel mode
Port: 26379 端口
PID: 2725

https://redis.io

2725:X 24 Nov 2021 09:08:40.321 # Sentinel ID is ed6efce390bfd9c0bb8b141e116eb8f5042fa0c0 主机和从机信息
2725:X 24 Nov 2021 09:08:40.321 # +monitor master mysentinel 127.0.0.1 6379 quorum 1
2725:X 24 Nov 2021 09:08:40.322 * +slave slave 127.0.0.1:6380 127.0.0.1 6380 @ mysentinel 127.0.0.1 6379
2725:X 24 Nov 2021 09:08:40.323 * +slave slave 127.0.0.1:6381 127.0.0.1 6381 @ mysentinel 127.0.0.1 6379
```

3、断开主机：

```
127.0.0.1:6379> shutdown
#从机信息：
127.0.0.1:6380> info replication
# Replication
role:master #80端口自己升为主机
connected_slaves:1
slave0:ip=127.0.0.1,port=6381,state=online,offset=22558,lag=0
master_failover_state:no-failover
master_replid:cba5caf39b0469a035959af3625fae5b2defea90
master_replid2:9ba967781e0852c3ece92b2a659e8b10f5913bc4
master_repl_offset:22558
second_repl_offset:10958
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:22558
127.0.0.1:6381> info replication
# Replication
role:slave
master_host:127.0.0.1
master_port:6380 #81端口将80端口作为主机
master_link_status:up
master_last_io_seconds_ago:1
master_sync_in_progress:0
slave_read_repl_offset:23936
slave_repl_offset:23936
slave_priority:100
```



```
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:31381
repl_backlog_histlen:3752
```

```
2725:X 24 Nov 2021 09:11:35.789 # +failover-end master mysentinel 127.0.0.1 6379
2725:X 24 Nov 2021 09:11:35.789 # +switch-master mysentinel 127.0.0.1 6379 127.0.0.1 6380
2725:X 24 Nov 2021 09:11:35.790 * +slave slave 127.0.0.1:6381 127.0.0.1 6381 @ mysentinel 127.0.0.1 6380
2725:X 24 Nov 2021 09:11:35.790 * +slave slave 127.0.0.1:6379 127.0.0.1 6379 @ mysentinel 127.0.0.1 6380
2725:X 24 Nov 2021 09:12:05.827 # +sdown slave 127.0.0.1:6379 127.0.0.1 6379 @ mysentinel 127.0.0.1 6380
2725:X 24 Nov 2021 09:16:25.708 # -sdown slave 127.0.0.1:6379 127.0.0.1 6379 @ mysentinel 127.0.0.1 6380
2725:X 24 Nov 2021 09:16:35.633 * +convert-to-slave slave 127.0.0.1:6379 127.0.0.1 6379 @ mysentinel 127
.0.0.1 6380
6379重新连接但是成为了从机
```

优点:

- 1、哨兵集群一般基于主从复制模式，所有的主从配置优点，它全有
- 2、主从可以切换，故障可以转移，系统的可用性就会更好
- 3、哨兵模式就是主从模式的升级，手动到自动，更加健壮！

缺点:

- 1、redis不好在线扩容，集群容量一旦达到上限，在线扩容就十分麻烦！
- 2、实现哨兵模式的配置其实是很麻烦的，里面有很多选择！

哨兵配置文件:

```
# Example sentinel.conf

# 哨兵sentinel实例运行的端口 默认26379
port 26379

# 哨兵sentinel的工作目录
dir /tmp

# 哨兵sentinel监控的redis主节点的 ip port
# master-name 可以自己命名的主节点名字 只能由字母A-z、数字0-9、这三个字符"._-"组成。
# quorum 当这些quorum个数sentinel哨兵认为master主节点失联 那么这时 客观上认为主节点失联了
# sentinel monitor <master-name> <ip> <redis-port> <quorum>
sentinel monitor mymaster 127.0.0.1 6379 2

# 当在Redis实例中开启了requirepass foobared 授权密码 这样所有连接Redis实例的客户端都要提供密码
# 设置哨兵sentinel 连接主从的密码 注意必须为主从设置一样的验证密码
# sentinel auth-pass <master-name> <password>
sentinel auth-pass mymaster MySUPER--secret-0123passw0rd

# 指定多少毫秒之后 主节点没有应答哨兵sentinel 此时 哨兵主观上认为主节点下线 默认30秒
# sentinel down-after-milliseconds <master-name> <milliseconds>
sentinel down-after-milliseconds mymaster 30000

# 这个配置项指定了在发生failover主备切换时最多可以有多少个slave同时对新的master进行 同步，
这个数字越小，完成failover所需的时间就越长，
但是如果这个数字越大，就意味着越 多的slave因为replication而不可用。
可以通过将这个值设为 1 来保证每次只有一个slave 处于不能处理命令请求的状态。
# sentinel parallel-syncs <master-name> <numslaves>
sentinel parallel-syncs mymaster 1
```



```
# 故障转移的超时时间 failover-timeout 可以用在以下这些方面：
#1. 同一个sentinel对同一个master两次failover之间的间隔时间。
#2. 当一个slave从一个错误的master那里同步数据开始计算时间。直到slave被纠正为向正确的master那里同步数据时。
#3.当想要取消一个正在进行的failover所需要的时间。
#4.当进行failover时，配置所有slaves指向新的master所需的最大时间。不过，即使过了这个超时，slaves依然会被正确配置为指向master，但是就不按parallel-syncs所配置的规则来了
# 默认三分钟
# sentinel failover-timeout <master-name> <milliseconds>
sentinel failover-timeout mymaster 180000

# SCRIPTS EXECUTION

#配置当某一事件发生时所需要执行的脚本，可以通过脚本来通知管理员，例如当系统运行不正常时发邮件通知相关人员。
#对于脚本的运行结果有以下规则：
#若脚本执行后返回1，那么该脚本稍后将会被再次执行，重复次数目前默认为10
#若脚本执行后返回2，或者比2更高的一个返回值，脚本将不会重复执行。
#如果脚本在执行过程中由于收到系统中断信号被终止了，则同返回值为1时的行为相同。
#一个脚本的最大执行时间为60s，如果超过这个时间，脚本将会被一个SIGKILL信号终止，之后重新执行。

#通知型脚本:当sentinel有任何警告级别的事件发生时（比如说redis实例的主观失效和客观失效等等），将会去调用这个脚本，
#这时这个脚本应该通过邮件，SMS等方式去通知系统管理员关于系统不正常运行的信息。调用该脚本时，将传给脚本两个参数，
#一个是事件的类型，
#一个是事件的描述。
#如果sentinel.conf配置文件中配置了这个脚本路径，那么必须保证这个脚本存在于这个路径，并且是可执行的，否则sentinel无法正常启动成功。
#通知脚本
# sentinel notification-script <master-name> <script-path>
sentinel notification-script mymaster /var/redis/notify.sh

# 客户端重新配置主节点参数脚本
# 当一个master由于failover而发生改变时，这个脚本将会被调用，通知相关的客户端关于master地址已经发生改变的信息。
# 以下参数将会在调用脚本时传给脚本：
# <master-name> <role> <state> <from-ip> <from-port> <to-ip> <to-port>
# 目前<state>总是“failover”，
# <role>是“leader”或者“observer”中的一个。
# 参数 from-ip, from-port, to-ip, to-port是用来和旧的master和新的master(即旧的slave)通信的
# 这个脚本应该是通用的，能被多次调用，不是针对性的。
# sentinel client-reconfig-script <master-name> <script-path>
sentinel client-reconfig-script mymaster /var/redis/reconfig.sh
```

Redis缓存穿透和雪崩

Redis缓存的使用，极大的提升了应用程序的性能和效率，特别是数据查询方面。但同时，它也带来了一些问题。其中，最要害的问题，就是数据的一致性问题，从严格意义上讲，这个问题无解。如果对数据的一致性要求很高，那么就不能使用缓存。

另外的一些典型问题就是，缓存穿透、缓存雪崩和缓存击穿。目前，业界也都有比较流行的解决方案。

缓存穿透

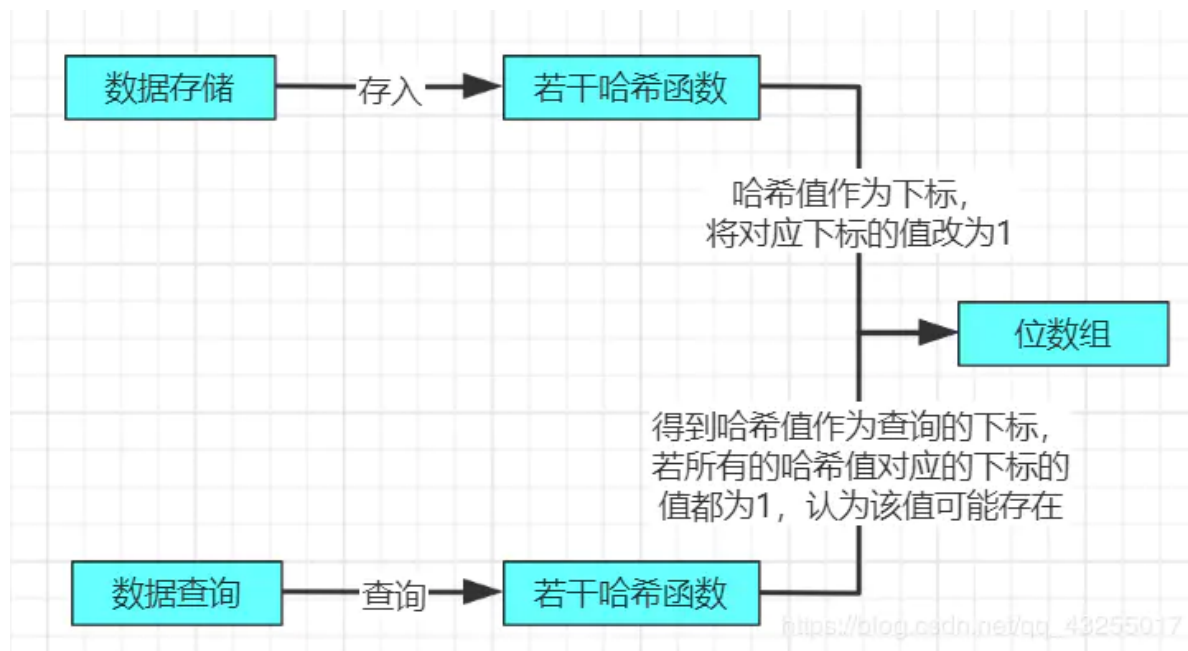
概念

缓存穿透的概念很简单，用户想要查询一个数据，发现redis内存数据库没有，也就是缓存没有命中，于是向持久层数据库查询。发现也没有，于是本次查询失败。当用户很多的时候，缓存都没有命中，于是都去请求了持久层数据库。这会给持久层数据库造成很大的压力，这时候就相当于出现了缓存穿透。

解决方案

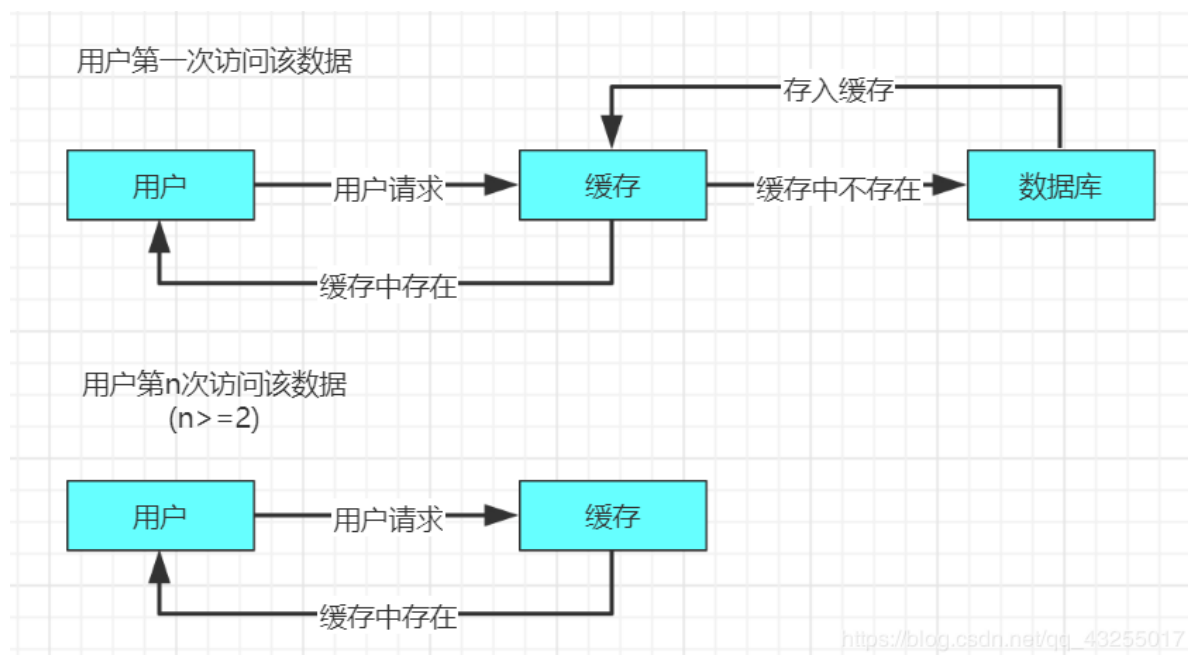
布隆过滤器

布隆过滤器是一种数据结构，对所有可能查询的参数以hash形式存储，在控制层先进行校验，不符合则丢弃，从而避免了对底层存储系统的查询压力；



缓存空对象

当存储层不命中后，即使返回的空对象也将其缓存起来，同时会设置一个过期时间，之后再访问这个数据将会从缓存中获取，保护了后端数据源：



但是这种方法会存在两个问题:

- 1、如果空值能够被缓存起来，这就意味着缓存需要更多的空间存储更多的键，因为这当中可能会有很多的空值的键；
- 2、即使对空值设置了过期时间，还是会存在缓存层和存储层的数据会有一段时间窗口的不一致，这对于需要保持一致性的业务会有影响。

缓存击穿

概述

这里需要注意和缓存击穿的区别，缓存击穿，是指一个key非常热点，在不停的扛着大并发，大并发集中对这一个点进行访问，当这个key在失效的瞬间，持续的大并发就穿破缓存，直接请求数据库，就像在一个屏障上凿开了一个洞。

当某个key在过期的瞬间，有大量的请求并发访问，这类数据一般是热点数据，由于缓存过期，会同时访问数据库来查询最新数据，并且回写缓存，会导致数据库瞬间压力过大。

解决方案

设置热点数据永不过期

从缓存层面来看，没有设置过期时间，所以不会出现热点 key过期后产生的问题。

加互斥锁

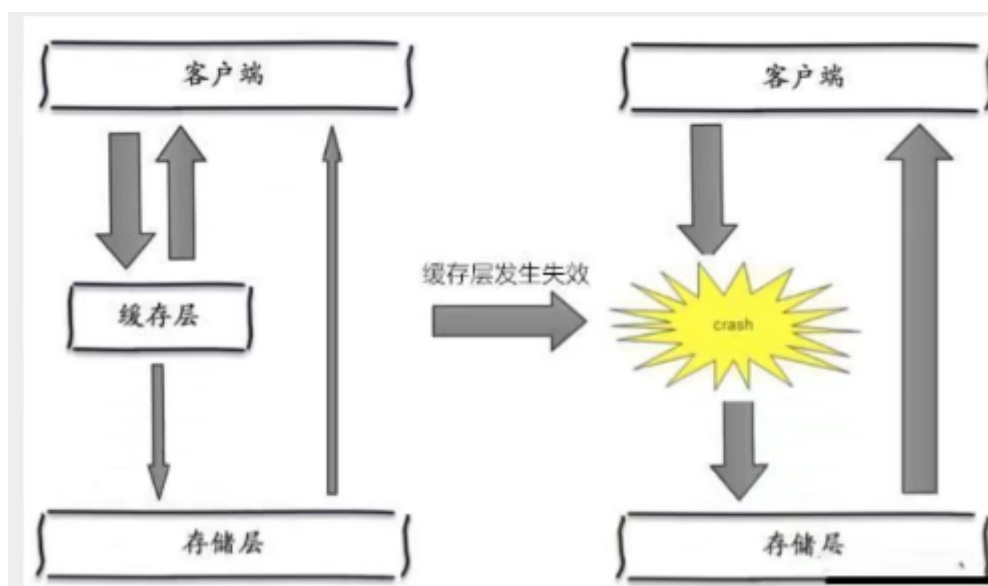
分布式锁：使用分布式锁，保证对于每个key同时只有一个线程去查询后端服务，其他线程没有获得分布式锁的权限，因此只需要等待即可。这种方式将高并发的压力转移到了分布式锁，因此对分布式锁的考验很大。

缓存雪崩

概念

缓存雪崩，是指在某一个时间段，缓存集中过期失效。

产生雪崩的原因之一，比如在写本文的时候，马上就要到双十二零点，很快就会迎来一波抢购，这波商品时间比较集中的放入了缓存，假设缓存一个小时。那么到了凌晨一点钟的时候，这批商品的缓存就都过期了。而对这批商品的访问查询，都落到了数据库上，对于数据库而言，就会产生周期性的压力波峰。于是所有的请求都会达到存储层，存储层的调用量会暴增，造成存储层也会挂掉的情况。



其实集中过期，倒不是非常致命，比较致命的缓存雪崩，是缓存服务器某个节点宕机或断网。因为自然形成的缓存雪崩，一定是在某个时间段集中创建缓存，这个时候，数据库也是可以顶住压力的。无非就是对数据库产生周期性的压力而已。而缓存服务节点的宕机，对数据库服务器造成的压力是不可预知的，很有可能瞬间就把数据库压垮。

解决方案

redis高可用

这个思想的含义是，既然redis有可能挂掉，那我多增设几台redis，这样一台挂掉之后其他的还可以继续工作，其实就是搭建的集群。

限流降级

这个解决方案的思想是，在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。

数据预热

数据加热的含义就是在正式部署之前，我先把可能的数据先预先访问一遍，这样部分可能大量访问的数据就会加载到缓存中。在即将发生大并发访问前手动触发加载缓存不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀。