

课前准备

- 准备redis安装包

课堂主题

Redis单机安装及数据类型分析、消息模式及事务、Java客户端的使用

课堂目标

- Redis是什么？
- Redis的主要应用场景有哪些？
- Redis单机安装要掌握？
- Redis数据类型有哪些？
- Redis的数据类型各自的使用场景及注意事项是什么？
- Redis的消息模式是如何实现的？
- Redis的事务是如何实现的？
- Java的客户端Jedis是如何使用的？

知识要点

课前准备

课堂主题

课堂目标

知识要点

Redis介绍

什么是Redis？

Redis官网

什么是NoSQL？

NoSQL数据库分类

Redis发展历史

Redis应用场景

Redis单机版安装配置

Redis下载

Redis安装环境

Redis安装

Redis启动

前端启动

后端启动（守护进程启动）

后端启动的关闭方式

其他命令说明

Redis客户端

Redis命令行客户端

Java客户端Jedis

Jedis介绍

- 添加依赖
- 单实例连接
- 连接池连接
- 连接redis集群
- Jedis整合spring

Redis数据类型

string类型

命令

- 赋值
- 取值
- 取值并赋值
- 数值增减
 - 注意事项
 - 递增数字
 - 增加指定的整数
 - 递减数值
 - 减少指定的整数
- 仅当不存在时赋值
- 其它命令
 - 向尾部追加值
 - 获取字符串长度
 - 同时设置/获取多个键值

应用场景之自增主键

hash类型

hash类型介绍

命令

- 赋值
 - 设置一个字段值
 - 设置多个字段值
 - 当字段不存在时赋值
- 取值
 - 获取一个字段值
 - 获取多个字段值
 - 获取所有字段值
- 删除字段
- 增加数字
- 其它命令(自学)
 - 判断字段是否存在
 - 只获取字段名或字段值
 - 获取字段数量
 - 获取所有字段

string类型和hash类型的区别

应用之存储商品信息

list类型

ArrayList与LinkedList的区别

list类型介绍

命令

- LPUSH/RPUSH
- LRANGE
- LPOP/RPOP
- LLEN
- 其它命令(自学)
 - LREM

LINDEX

LTRIM

LINSERT

RPOPLPUSH

应用之商品评论列表

set类型

set类型介绍

命令

SADD/SREM

SMEMBERS

SISMEMBER

集合运算命令

SDIFF

SINTER

SUNION

其它命令(自学)

SCARD

SPOP

zset类型 (sortedset)

zset介绍

命令

ZADD

ZRANGE/ZREVRANGE

ZSCORE

ZREM

其它命令(自学)

ZRANGEBYSCORE

ZINCRBY

ZCARD

ZCOUNT

ZREMRANGEBYRANK

ZREMRANGEBYSCORE

ZRANK/ZREVRANK

应用之商品销售排行榜

通用命令

keys

del

exists

expire (重点)

rename

type

Redis消息模式

队列模式

发布订阅模式

Redis事务

Redis事务介绍

事务命令

MULTI

EXEC

DISCARD

WATCH

UNWATCH

事务演示

Redis介绍

什么是Redis？

- Redis 是用C语言开发的一个开源的高性能键值对 (key-value) 内存数据库，它是一种 NoSQL 数据库。
- 它是【单进程单线程】的内存数据库，所以说不存在线程安全问题。
- 它可以支持并发 10W QPS，所以说性能非常优秀。之所以单进程单线程性能还这么好，是因为底层采用了【IO多路复用 (NIO思想)】
- 相比Memcache这种专业缓存技术，它有更优秀的读写性能，及丰富的数据类型。
- 它提供了五种数据类型来存储【值】：字符串类型 (string)、散列类型 (hash)、列表类型 (list)、集合类型 (set)、有序集合类型 (sortedset、zset)

Redis官网

- 官网地址：<http://redis.io/>
- 中文官网地址：<http://www.redis.cn/>
- 下载地址：<http://download.redis.io/releases/>

什么是NoSQL？

- NoSQL，即 Not-Only SQL (不仅仅是 SQL)，泛指非关系型的数据库。
- 什么是关系型数据库？数据结构是一种有行有列的数据库
- NoSQL 数据库是为了解决高并发、高可用、高可扩展、大数据存储问题而产生的数据库解决方案。
- NoSQL 可以作为关系型数据库的良好补充，但是不能替代关系型数据库。

MySQL (关系型数据库) ----> NoSQL ----> NewSQL (TiDB)

NoSQL数据库分类

- 键值(Key-Value)存储数据库

相关产品：Tokyo Cabinet/Tyrant、Redis、Voltdemort、Berkeley DB

典型应用：内容缓存，主要用于处理大量数据的高访问负载。

数据模型：一系列键值对

优势：快速查询

劣势：存储的数据缺少结构化

- 列存储数据库

相关产品：Cassandra, HBase, Riak

典型应用：分布式的文件系统

数据模型：以列簇式存储，将同一列数据存在一起

优势：查找速度快，可扩展性强，更容易进行分布式扩展

劣势：功能相对局限

- 文档型数据库

相关产品：CouchDB、MongoDB

典型应用：Web 应用（与 Key-Value 类似，value 是结构化的）

数据模型：一系列键值对

优势：数据结构要求不严格

劣势：

- 图形(Graph)数据库

相关数据库：Neo4J、InfoGrid、Infinite Graph

典型应用：社交网络

数据模型：图结构

优势：利用图结构相关算法。

劣势：需要对整个图做计算才能得出结果，不容易做分布式的集群方案。

Redis发展历史

2008年，意大利的一家创业公司 Merzia 推出了一款基于 MySQL 的网站实时统计系统 LL00GG，然而没过多久该公司的创始人 Salvatore Sanfilippo 便对 MySQL 的性能感到失望，于是他决定亲自为 LL00GG 量身定做一个数据库，并于2009年开发完成，这个数据库就是 Redis。

不过 Salvatore Sanfilippo 并不满足只将 Redis 用于 LL00GG 这一款产品，而是希望更多的人使用它，于是在同一年 Salvatore Sanfilippo 将 Redis 开源发布，并开始和 Redis 的另一名主要的代码贡献者 Pieter Noordhuis 一起继续着 Redis 的开发，直到今天。

Salvatore Sanfilippo 自己也没有想到，短短的几年时间，Redis 就拥有了庞大的用户群体。Hacker News 在2012年发布了一份数据库的使用情况调查，结果显示有近12%的公司在使用Redis。国内如新浪微博、街旁网、知乎网，国外如 GitHub、Stack Overflow、Flickr 等都是 Redis 的用户。

VMware 公司从2010年开始赞助 Redis 的开发，Salvatore Sanfilippo 和 Pieter Noordhuis 也分别在3月和5月加入 VMware，全职开发 Redis。

Redis应用场景

- 内存数据库（登录信息、购物车信息、用户浏览记录等）
- 缓存服务器（商品数据、广告数据等等）（最多使用）
- 解决分布式集群架构中的 session 分离问题（session 共享）
- 任务队列（秒杀、抢购、12306等等）
- 分布式锁的实现
- 支持发布订阅的消息模式
- 应用排行榜(有序集合)
- 网站访问统计
- 数据过期处理（可以精确到毫秒）

Redis单机版安装配置

Redis下载

- 官网地址：<http://redis.io/>
- 中文官网地址：<http://www.redis.cn/>
- 下载地址：<http://download.redis.io/releases/>

Redis安装环境

Redis 没有官方的 windows 版本，所以建议在 Linux 系统上安装运行，我们使用

CentOS 7（Linux 操作系统的一个系列）作为安装环境。

Windows版本

Redis没有官方的Windows版本，但是微软开源技术团队（Microsoft Open Tech group）开发和维护着这个Win64的版本。更多信息请参考[这里](#)。

Redis安装

第一步：安装 C 语言需要的 GCC 环境

```
1 yum install -y gcc-c++
2 yum install -y wget
```

第二步：下载并解压缩 Redis 源码压缩包

```
1 wget http://download.redis.io/releases/redis-5.0.4.tar.gz
2 tar -zxf redis-5.0.4.tar.gz
```

第三步：编译 Redis 源码，进入 redis-3.2.9 目录，执行编译命令

```
1 cd redis-5.0.4
2 make
```

第四步：安装 Redis，需要通过 PREFIX 指定安装路径

```
1 | make install PREFIX=/kkb/server/redis
```

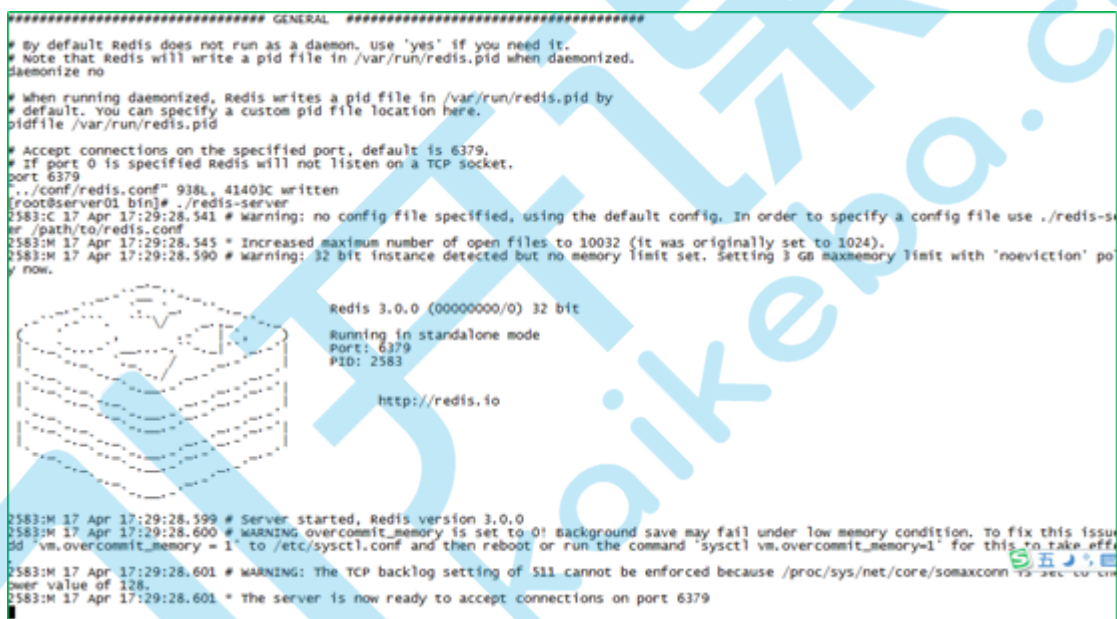
Redis启动

前端启动

- 启动命令：`redis-server`，直接运行 `bin/redis-server` 将以前端模式启动

```
1 | ./redis-server
```

- 关闭命令：`ctrl+c`
- 启动缺点：客户端窗口关闭则 `redis-server` 程序结束，不推荐使用此方法
- 启动图例：



```
##### GENERAL #####
# By default Redis does not run as a daemon. Use 'yes' if you need it.
# Note that Redis will write a pid file in /var/run/redis.pid when daemonized.
daemonize no

# When running daemonized, Redis writes a pid file in /var/run/redis.pid by
# default. You can specify a custom pid file location here.
pidfile /var/run/redis.pid

# Accept connections on the specified port, default is 6379.
# If port 0 is specified Redis will not listen on a TCP socket.
port 6379
./conf/redis.conf 938L, 41403C written
[root@server01 bin]# ./redis-server
2583:C 17 Apr 17:29:28.541 # Warning: no config file specified, using the default config. In order to specify a config file use ./redis-server /path/to/redis.conf
2583:M 17 Apr 17:29:28.545 * Increased maximum number of open files to 10032 (it was originally set to 1024).
2583:M 17 Apr 17:29:28.590 # Warning: 32 bit instance detected but no memory limit set. Setting 3 GB maxmemory limit with 'noeviction' policy now.

Redis 3.0.0 (00000000/0) 32 bit
Running in standalone mode
Port: 6379
PID: 2583
http://redis.io

2583:M 17 Apr 17:29:28.599 # Server started, Redis version 3.0.0
2583:M 17 Apr 17:29:28.600 # WARNING overcommit_memory is set to 0! Background save may fail under low memory condition. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take effect.
2583:M 17 Apr 17:29:28.601 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
2583:M 17 Apr 17:29:28.601 * The server is now ready to accept connections on port 6379
```

后端启动（守护进程启动）

- 第一步：拷贝 `redis-5.0.4/redis.conf` 配置文件到 Redis 安装目录的 `bin` 目录

```
1 | cp /root/redis-5.0.4/redis.conf /kkb/server/redis/bin/
```

- 第二步：修改 `redis.conf`

```
1 | vim redis.conf
```

```
1 # 将`daemonize`由`no`改为`yes`
2 daemonize yes
3
4 # 默认绑定的是回环地址，默认不能被其他机器访问
5 # bind 127.0.0.1
6
7 # 是否开启保护模式，由yes该为no
8 protected-mode no
```

- 第三步：启动服务

```
1 | ./redis-server redis.conf
```

后端启动的关闭方式

```
1 | ./redis-cli shutdown
```

其他命令说明

- `redis-server` : 启动 redis 服务
- `redis-cli` : 进入 redis 命令客户端
- `redis-benchmark` : 性能测试的工具
- `redis-check-aof` : aof 文件进行检查的工具
- `redis-check-dump` : rdb 文件进行检查的工具
- `redis-sentinel` : 启动哨兵监控服务

Redis客户端

Redis命令行客户端



The screenshot shows a terminal window with the following content:

```
-rwxr--r--. 1 root root      18 9月 30 10:00 dump.rdb
-rwxr-xr-x. 1 root root 4171542 9月 30 09:49 redis-benchmark
-rwxr-xr-x. 1 root root   16415 9月 30 09:49 redis-check-aof
-rwxr-xr-x. 1 root root   37647 9月 30 09:49 redis-check-dump
-rwxr-xr-x. 1 root root 4260308 9月 30 09:49 redis-cli
-rw-r--r--. 1 root root   41404 9月 30 09:57 redis.conf
lrwxrwxrwx. 1 root root      12 9月 30 09:49 redis-sentinel -> redis-server
-rwxr-xr-x. 1 root root 5690105 9月 30 09:49 redis-server
[root@localhost ~]# ps aux|grep redis
```

- 命令格式

```
1 | ./redis-cli -h 127.0.0.1 -p 6379
```

- 参数说明

```
1 -h : redis服务器的ip地址
2 -p : redis实例的端口号
```

- 默认方式

如果不指定主机和端口也可以

- 默认主机地址是127.0.0.1
- 默认端口是6379

```
1 | ./redis-cli
```

Java客户端Jedis

Jedis介绍

Redis不仅使用命令来操作，而且可以使用程序客户端操作。现在基本上主流的语言都有客户端支持，比如java、C、C#、C++、php、Node.js、Go等。

在官方网站里列一些Java的客户端，有Jedis、Redisson、Jredis、JDBC-Redis、等其中官方推荐使用Jedis和Redisson。 在企业中用的最多的就是Jedis，下面我们就重点学习下Jedis。

Jedis同样也是托管在github上，地址：<https://github.com/xetorthio/jedis>

添加依赖

```
1  <dependencies>
2      <dependency>
3          <groupId>redis.clients</groupId>
4          <artifactId>jedis</artifactId>
5          <version>2.9.0</version>
6      </dependency>
7
8      <dependency>
9          <groupId>org.springframework</groupId>
10         <artifactId>spring-context</artifactId>
11         <version>5.0.7.RELEASE</version>
12     </dependency>
13
14     <dependency>
15         <groupId>org.springframework</groupId>
16         <artifactId>spring-test</artifactId>
17         <version>5.0.7.RELEASE</version>
18     </dependency>
19
20     <!-- 单元测试JUnit -->
21     <dependency>
22         <groupId>junit</groupId>
23         <artifactId>junit</artifactId>
24         <version>4.12</version>
25     </dependency>
26 </dependencies>
27 <build>
28     <plugins>
```

```

29         <!-- 配置Maven的JDK编译级别 -->
30         <plugin>
31             <groupId>org.apache.maven.plugins</groupId>
32             <artifactId>maven-compiler-plugin</artifactId>
33             <version>3.2</version>
34             <configuration>
35                 <source>1.8</source>
36                 <target>1.8</target>
37                 <encoding>UTF-8</encoding>
38             </configuration>
39         </plugin>
40     </plugins>
41 </build>
42

```

单实例连接

```

1  @Test
2  public void testJedis() {
3      //创建一个Jedis的连接
4      Jedis jedis = new Jedis("127.0.0.1", 6379);
5      //执行redis命令
6      jedis.set("mytest", "hello world, this is jedis client!");
7      //从redis中取值
8      String result = jedis.get("mytest");
9      //打印结果
10     System.out.println(result);
11     //关闭连接
12     jedis.close();
13
14 }
15

```

连接池连接

```

1  @Test
2  public void testJedisPool() {
3      //创建一连接池对象
4      JedisPool jedisPool = new JedisPool("127.0.0.1", 6379);
5      //从连接池中获得连接
6      Jedis jedis = jedisPool.getResource();
7      String result = jedis.get("mytest");
8      System.out.println(result);
9      //关闭连接
10     jedis.close();
11
12     //关闭连接池
13     jedisPool.close();
14 }
15

```

连接redis集群

创建JedisCluster类连接redis集群。

```
1  @Test
2  public void testJedisCluster() throws Exception {
3      //创建一连接, JedisCluster对象,在系统中是单例存在
4      Set<HostAndPort> nodes = new HashSet<>();
5      nodes.add(new HostAndPort("192.168.242.129", 7001));
6      nodes.add(new HostAndPort("192.168.242.129", 7002));
7      nodes.add(new HostAndPort("192.168.242.129", 7003));
8      nodes.add(new HostAndPort("192.168.242.129", 7004));
9      nodes.add(new HostAndPort("192.168.242.129", 7005));
10     nodes.add(new HostAndPort("192.168.242.129", 7006));
11     JedisCluster cluster = new JedisCluster(nodes);
12     //执行JedisCluster对象中的方法,方法和redis一一对应。
13     cluster.set("cluster-test", "my jedis cluster test");
14     String result = cluster.get("cluster-test");
15     System.out.println(result);
16     //程序结束时需要关闭JedisCluster对象
17     cluster.close();
18 }
19
```

Jedis整合spring

配置spring配置文件applicationContext.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://www.springframework.org/schema/beans
5          http://www.springframework.org/schema/beans/spring-beans.xsd">
6
7      <!-- 连接池配置 -->
8      <bean id="jedisPoolConfig"
9          class="redis.clients.jedis.JedisPoolConfig">
10         <!-- 最大连接数 -->
11         <property name="maxTotal" value="30" />
12         <!-- 最大空闲连接数 -->
13         <property name="maxIdle" value="10" />
14         <!-- 每次释放连接的最大数目 -->
15         <property name="numTestsPerEvictionRun" value="1024" />
16         <!-- 释放连接的扫描间隔(毫秒) -->
17         <property name="timeBetweenEvictionRunsMillis" value="30000" />
18         <!-- 连接最小空闲时间 -->
19         <property name="minEvictableIdleTimeMillis" value="1800000" />
20         <!-- 连接空闲多久后释放,当空闲时间>该值 且 空闲连接>最大空闲连接数 时直接释放 -->
21         <property name="softMinEvictableIdleTimeMillis" value="10000" />
22         <!-- 获取连接时的最大等待毫秒数,小于零:阻塞不确定的时间,默认-1 -->
```

```

23     <property name="maxwaitMillis" value="1500" />
24     <!-- 在获取连接的时候检查有效性，默认false -->
25     <property name="testOnBorrow" value="true" />
26     <!-- 在空闲时检查有效性，默认false -->
27     <property name="testWhileIdle" value="true" />
28     <!-- 连接耗尽时是否阻塞，false报异常，ture阻塞直到超时，默认true -->
29     <property name="blockWhenExhausted" value="false" />
30 </bean>
31
32 <!-- redis单机 通过连接池 -->
33 <bean id="jedisPool" class="redis.clients.jedis.JedisPool"
34     destroy-method="close">
35     <constructor-arg name="poolConfig"
36         ref="jedisPoolConfig" />
37     <constructor-arg name="host" value="192.168.10.135" />
38     <constructor-arg name="port" value="6379" />
39 </bean>
40
41 <!-- redis集群 -->
42 <bean id="jedisCluster" class="redis.clients.jedis.JedisCluster">
43     <constructor-arg index="0">
44         <set>
45             <bean class="redis.clients.jedis.HostAndPort">
46                 <constructor-arg index="0" value="192.168.10.135"></constructor-
arg>
47                 <constructor-arg index="1" value="7001"></constructor-arg>
48             </bean>
49             <bean class="redis.clients.jedis.HostAndPort">
50                 <constructor-arg index="0" value="192.168.10.135"></constructor-
arg>
51                 <constructor-arg index="1" value="7002"></constructor-arg>
52             </bean>
53             <bean class="redis.clients.jedis.HostAndPort">
54                 <constructor-arg index="0" value="192.168.10.135"></constructor-
arg>
55                 <constructor-arg index="1" value="7003"></constructor-arg>
56             </bean>
57             <bean class="redis.clients.jedis.HostAndPort">
58                 <constructor-arg index="0" value="192.168.10.135"></constructor-
arg>
59                 <constructor-arg index="1" value="7004"></constructor-arg>
60             </bean>
61             <bean class="redis.clients.jedis.HostAndPort">
62                 <constructor-arg index="0" value="192.168.10.135"></constructor-
arg>
63                 <constructor-arg index="1" value="7005"></constructor-arg>
64             </bean>
65             <bean class="redis.clients.jedis.HostAndPort">
66                 <constructor-arg index="0" value="192.168.10.135"></constructor-
arg>
67                 <constructor-arg index="1" value="7006"></constructor-arg>
68             </bean>
69         </set>

```

```

70         </constructor-arg>
71         <constructor-arg index="1" ref="jedisPoolConfig"></constructor-arg>
72     </bean>
73 </beans>

```

测试代码

```

1  package com.kkb.redis.test;
2
3  import javax.annotation.Resource;
4
5  import org.junit.Test;
6  import org.junit.runner.RunWith;
7  import org.springframework.beans.factory.annotation.Autowired;
8  import org.springframework.test.context.ContextConfiguration;
9  import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
10
11 import redis.clients.jedis.Jedis;
12 import redis.clients.jedis.JedisCluster;
13 import redis.clients.jedis.JedisPool;
14
15 @RunWith(SpringJUnit4ClassRunner.class)
16 @ContextConfiguration(locations = "classpath:application.xml")
17 public class TestJedis2 {
18
19     @Autowired
20     private JedisPool jedisPool;
21
22     @Resource
23     private JedisCluster cluster;
24
25     @Test
26     public void testJedisPool() {
27         // 从连接池中获得连接
28         Jedis jedis = jedisPool.getResource();
29         String result = jedis.get("mytest");
30         System.out.println(result);
31         // 关闭连接
32         jedis.close();
33     }
34
35     @Test
36     public void testJedisCluster() throws Exception {
37         // 执行JedisCluster对象中的方法，方法和redis一一对应。
38         cluster.set("cluster-test", "my jedis cluster test");
39         String result = cluster.get("cluster-test");
40         System.out.println(result);
41     }
42
43 }
44

```

Redis数据类型

官方命令大全网址：<http://www.redis.cn/commands.html>

Redis 中存储数据是通过 key-value 格式存储数据的，其中 value 可以定义五种数据类型：

- String (字符串类型)
- Hash (散列类型)
- List (列表类型)
- Set (集合类型)
- SortedSet (有序集合类型，简称zset)

注意：在 redis 中的命令语句中，命令是忽略大小写的，而 key 是不忽略大小写的。

string类型

命令

赋值

- 语法：

```
1 | SET key value
```

- 示例：

```
1 | 127.0.0.1:6379> set test 123
2 | OK
```

取值

- 语法：

```
1 | GET key
```

```
127.0.0.1:6379> get test "123 "
```

- 示例：

```
1 | 127.0.0.1:6379> get test
2 | "123"
```

取值并赋值

语法：

```
1 | GETSET key value
```

示例：

```
1 | 127.0.0.1:6379> getset s2 222
2 | "111"
3 | 127.0.0.1:6379> get s2
4 | "222"
```

数值增减

注意事项

- 1、当value为整数数据时，才能使用以下命令操作数值的增减。
- 2、数值递增都是【原子】操作。
- 3、redis中的每一个单独的命令都是原子性操作。当多个命令一起执行的时候，就不能保证原子性，不过我们可以使用事务和lua脚本来保证这一点。

非原子性操作示例：

```
1 | int i = 1;
2 | i++;
3 | System.out.println(i)
```

递增数字

- 语法 (increment)：

```
1 | INCR key
```

- 示例：

```
1 | 127.0.0.1:6379> incr num
2 | (integer) 1
3 | 127.0.0.1:6379> incr num
4 | (integer) 2
5 | 127.0.0.1:6379> incr num
6 | (integer) 3
```

增加指定的整数

- 语法：

```
1 | INCRBY key increment
```

- 示例：

```
1 127.0.0.1:6379> incrby num 2
2 (integer) 5
3 127.0.0.1:6379> incrby num 2
4 (integer) 7
5 127.0.0.1:6379> incrby num 2
6 (integer) 9
```

递减数值

- 语法：

```
1 DECR key
```

- 示例：

```
1 127.0.0.1:6379> incr num
2 (integer) 1
3 127.0.0.1:6379> incr num
4 (integer) 2
5 127.0.0.1:6379> incr num
6 (integer) 3
```

减少指定的整数

- 语法：

```
1 DECRBY key decrement
```

- 示例

```
1 127.0.0.1:6379> decr num
2 (integer) 6
3 127.0.0.1:6379> decr num
4 (integer) 5
5 127.0.0.1:6379> decrby num 3
6 (integer) 2
7 127.0.0.1:6379> decrby num 3
8 (integer) -1
9
```

仅当不存在时赋值

使用该命令可以实现【分布式锁】的功能，后续讲解！！

- 语法：

```
1 setnx key value
```


- 示例：

```
1 redis> EXISTS job                # job 不存在
2 (integer) 0
3 redis> SETNX job "programmer"    # job 设置成功
4 (integer) 1
5 redis> SETNX job "code-farmer"   # 尝试覆盖 job , 失败
6 (integer) 0
7 redis> GET job                  # 没有被覆盖
8 "programmer"
```

其它命令

向尾部追加值

APPEND 命令，向键值的末尾追加 value。

如果键不存在则将该键的值设置为 value，即相当于 SET key value。返回值是追加后字符串的总长度。

- 语法：

```
1 APPEND key value
```

- 示例：

```
1 127.0.0.1:6379> set str hello
2 OK
3 127.0.0.1:6379> append str " world!"
4 (integer) 12
5 127.0.0.1:6379> get str
6 "hello world!"
```

获取字符串长度

STRLEN 命令，返回键值的长度，如果键不存在则返回0。

- 语法：

```
1 STRLEN key
```

- 示例：

```
1 127.0.0.1:6379> strlen str
2 (integer) 0
3 127.0.0.1:6379> set str hello
4 OK
5 127.0.0.1:6379> strlen str
6 (integer) 5
```

同时设置/获取多个键值

- 语法：

```
1 MSET key value [key value ...]
2
3 MGET key [key ...]
```

- 示例：

```
1 127.0.0.1:6379> mset k1 v1 k2 v2 k3 v3
2 OK
3 127.0.0.1:6379> get k1
4 "v1"
5 127.0.0.1:6379> mget k1 k3
6 1) "v1"
7 2) "v3"
```

应用场景之自增主键

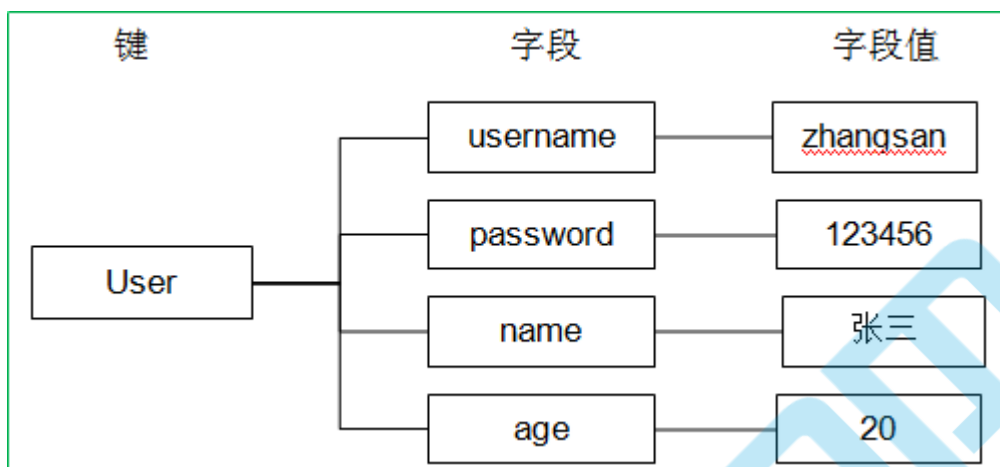
- 需求：商品编号、订单号采用 INCR 命令生成。
- 设计：key 命名要有一定的设计
- 实现：定义商品编号 key：items:id

```
1 192.168.101.3:7003> INCR items:id
2 (integer) 2
3 192.168.101.3:7003> INCR items:id
4 (integer) 3
```

hash类型

hash类型介绍

hash 类型也叫散列类型，它提供了字段和字段值的映射。字段值只能是字符串类型，不支持散列类型、集合类型等其它类型。如下：



命令

赋值

HSET 命令不区分插入和更新操作，当执行插入操作时 HSET 命令返回 1，当执行更新操作时返回 0。

设置一个字段值

- 语法：

```
1 | HSET key field value
```

- 示例：

```
1 | 127.0.0.1:6379> hset user username zhangsan
2 | (integer) 1
```

设置多个字段值

- 语法：

```
1 | HMSET key field value [field value ...]
```

- 示例：

```
1 | 127.0.0.1:6379> hmset user age 20 username lisi
2 | OK
```

当字段不存在时赋值

类似 HSET，区别在于如果字段存在，该命令不执行任何操作

- 语法：

```
1 | HSETNX key field value
```

- 示例：

```
1 | 127.0.0.1:6379> hsetnx user age 30 # 如果user中没有age字段则设置age值为30，否则不做任何操作
2 | (integer) 0
3 |
```

取值

获取一个字段值

- 语法：

```
1 | HGET key field
```

- 示例：

```
1 | 127.0.0.1:6379> hget user username
2 | "zhangsan"
```

获取多个字段值

- 语法：

```
1 | HMGET key field [field ...]
```

- 示例：

```
1 | 127.0.0.1:6379> hmget user age username
2 | 1) "20"
3 | 2) "lisi"
```

获取所有字段值

- 语法：

```
1 | HGETALL key
```

- 示例：

```
1 127.0.0.1:6379> hgetall user
2 1) "age"
3 2) "20"
4 3) "username"
5 4) "lisi"
```

删除字段

可以删除一个或多个字段，返回值是被删除的字段个数

- 语法：

```
1 HDEL key field [field ...]
```

- 示例：

```
1 127.0.0.1:6379> hdel user age
2 (integer) 1
3 127.0.0.1:6379> hdel user age name
4 (integer) 0
5 127.0.0.1:6379> hdel user age username
6 (integer) 1
```

增加数字

- 语法：

```
1 HINCRBY key field increment
```

- 示例：

```
1 127.0.0.1:6379> hincrby user age 2 # 将用户的年龄加2
2 (integer) 22
3 127.0.0.1:6379> hget user age # 获取用户的年龄
4 "22"
```

其它命令(自学)

判断字段是否存在

- 语法：

```
1 HEXISTS key field
```

- 示例：

```
1 127.0.0.1:6379> hexists user age      查看user中是否有age字段
2 (integer) 1
3 127.0.0.1:6379> hexists user name    查看user中是否有name字段
4 (integer) 0
```

只获取字段名或字段值

- 语法：

```
1 HKEYS key
2 HVALS key
```

- 示例：

```
1 127.0.0.1:6379> hmset user age 20 name lisi
2 OK
3 127.0.0.1:6379> hkeys user
4 1) "age"
5 2) "name"
6 127.0.0.1:6379> hvals user
7 1) "20"
8 2) "lisi"
```

获取字段数量

- 语法：

```
1 HLEN key
```

- 示例：

```
1 127.0.0.1:6379> hlen user
2 (integer) 2
```

获取所有字段

获得 hash 的所有信息，包括 key 和 value

- 语法：

```
1 hgetall key
```

string类型和hash类型的区别

hash类型适合存储那些对象数据，特别是对象属性经常发生【增删改】操作的数据。 string类型也可以存储对象数据，将java对象转成json字符串进行存储，这种存储适合【查询】操作。

应用之存储商品信息

- 商品信息字段

```
1 | 【商品id、商品名称、商品描述、商品库存、商品好评】
```

- 定义商品信息的key

```
1 | 商品ID为1001的信息在 Redis中的key为：[items:1001]
```

- 存储商品信息

```
1 | 192.168.101.3:7003> HMSET items:1001 id 3 name apple price 999.9
2 | OK
3 |
```

- 获取商品信息

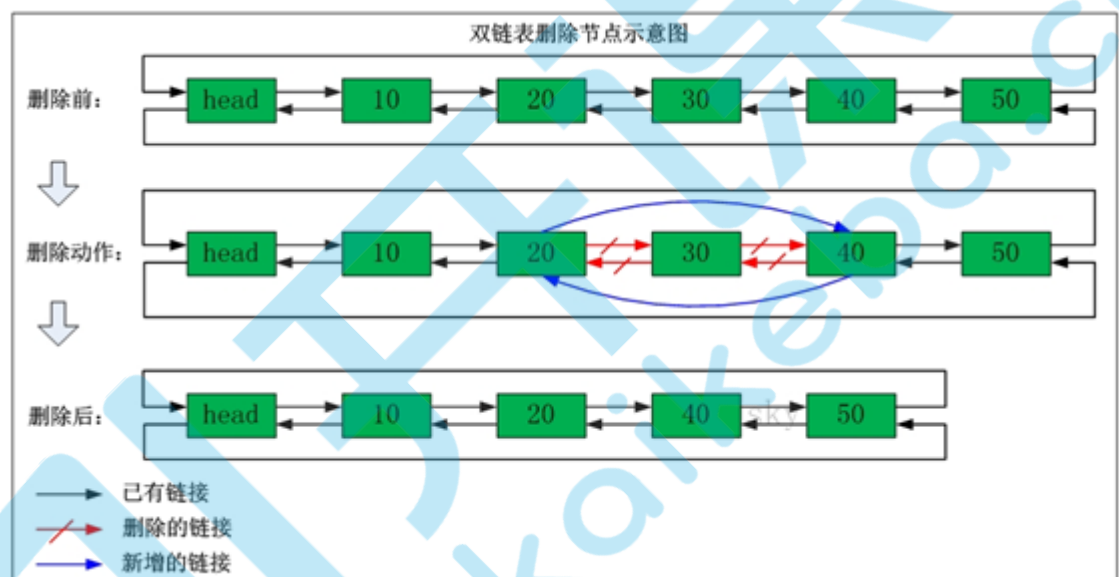
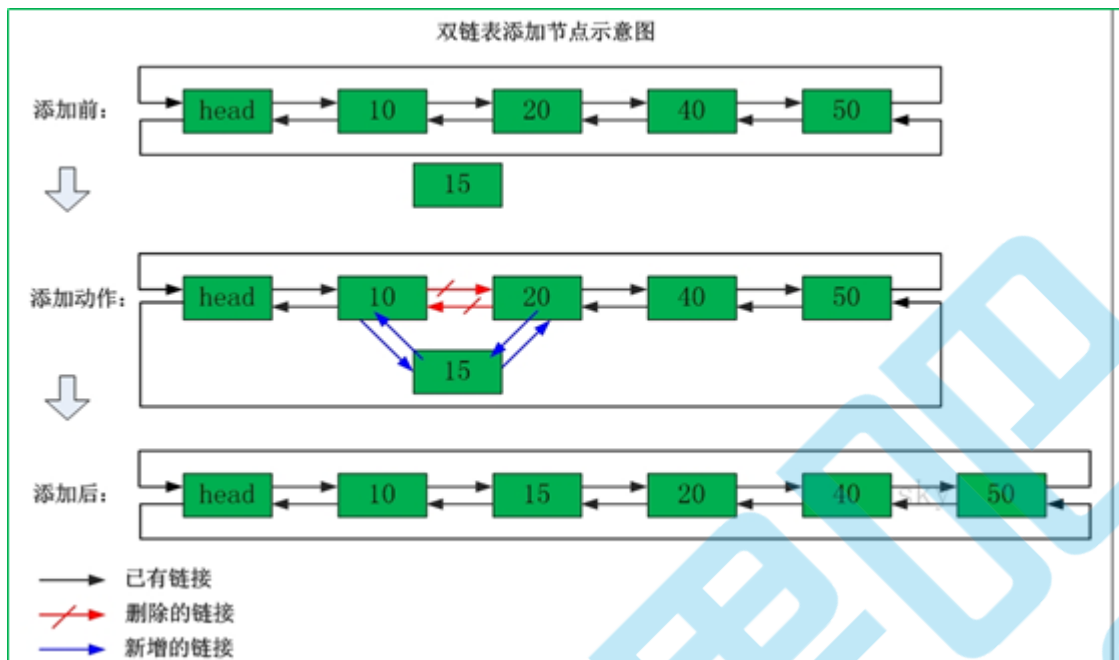
```
1 | 192.168.101.3:7003> HGET items:1001 id
2 | "3"
3 | 192.168.101.3:7003> HGETALL items:1001
4 | 1) "id"
5 | 2) "3"
6 | 3) "name"
7 | 4) "apple"
8 | 5) "price"
9 | 6) "999.9"
10 |
```

list类型

ArrayList与LinkedList的区别

ArrayList 使用数组方式存储数据，所以根据索引查询数据速度快，而新增或者删除元素时需要设计到位移操作，所以比较慢。

LinkedList 使用双向链表方式存储数据，每个元素都记录前后元素的指针，所以插入、删除数据时只是更改前后元素的指针指向即可，速度非常快。然后通过下标查询元素时需要从头开始索引，所以比较慢，但是如果查询前几个元素或后几个元素速度比较快。



list类型介绍

Redis 的列表类型 (`list` 类型) 可以存储一个有序的字符串列表，常用的操作是向列表两端添加元素，或者获得列表的某一个片段。

列表类型内部是使用双向链表 (`double linked list`) 实现的，所以向列表两端添加元素的时间复杂度为 $O(1)$ ，获取越接近两端的元素速度就越快。这意味着即使是一个有几千万个元素的列表，获取头部或尾部的10条记录也是极快的。

命令

LPUSH/RPUSH

- 语法：


```
1 LPUSH key value [value ...]
2 RPUSH key value [value ...]
```

- 示例：

```
1 127.0.0.1:6379> lpush list:1 1 2 3
2 (integer) 3
3 127.0.0.1:6379> rpush list:1 4 5 6
4 (integer) 3
```

LRange

```
1 获取列表中的某一片段。将返回`start`、`stop`之间的所有元素（包含两端的元素），索引从`0`开始。索引可以是负数，如：“`-1`”代表最后边的一个元素。
```

- 语法：

```
1 LRange key start stop
```

- 示例：

```
1 127.0.0.1:6379> lrange list:1 0 2
2 1) "2"
3 2) "1"
4 3) "4"
```

LPOP/RPOP

```
1 从列表两端弹出元素
```

从列表左边弹出一个元素，会分两步完成：

- 第一步是将列表左边的元素从列表中移除
- 第二步是返回被移除的元素值。

语法：

```
1 LPOP key
2 RPOP key
```

示例

```
1 127.0.0.1:6379>lpop list:1
2 "3"
3 127.0.0.1:6379>rpop list:1
4 "6"
```

LLEN

```
1 | 获取列表中元素的个数
```

- 语法：

```
1 | llen key
```

- 示例：

```
1 | 127.0.0.1:6379> llen list:1
2 | (integer) 2
```

其它命令(自学)

LREM

```
1 | 删除列表中指定个数的值
```

LREM 命令会删除列表中前 count 个值为 value 的元素，返回实际删除的元素个数。根据 count 值的不同，该命令的执行方式会有所不同：

```
1 | - 当count>0时， LREM会从列表左边开始删除。
2 | - 当count<0时， LREM会从列表后边开始删除。
3 | - 当count=0时， LREM删除所有值为value的元素。
```

- 语法：

```
1 | LREM key count value
```

LINDEX

```
1 | 获得指定索引的元素值
```

- 语法：

```
1 | LINDEX key index
```

- 示例：

```
1 | 127.0.0.1:6379>lindex l:list 2
2 | "1"
```

Ø 设置指定索引的元素值

语法：LSET key index value

```
127.0.0.1:6379> lset l:list 2 2 OK 127.0.0.1:6379> lrange l:list 0 -1 1) "6" 2) "5" 3) "2" 4) "2"
```

LTRIM

1 | 只保留列表指定片段,指定范围和LRANGE一致

• 语法 :

1 | LTRIM key start stop

• 示例 :

```
1 | 127.0.0.1:6379> lrange l:list 0 -1
2 | 1) "6"
3 | 2) "5"
4 | 3) "0"
5 | 4) "2"
6 | 127.0.0.1:6379> ltrim l:list 0 2
7 | OK
8 | 127.0.0.1:6379> lrange l:list 0 -1
9 | 1) "6"
10 | 2) "5"
11 | 3) "0"
```

LINSERT

- 1 | 向列表中插入元素。
- 2 | 该命令首先会在列表中从左到右查找值为pivot的元素,然后根据第二个参数是BEFORE还是AFTER来决定将value插入到该元素的前面还是后面。

语法 :

1 | LINSERT key BEFORE|AFTER pivot value

示例 :

```
1 127.0.0.1:6379> lrange list 0 -1
2 1) "3"
3 2) "2"
4 3) "1"
5 127.0.0.1:6379> linsert list after 3 4
6 (integer) 4
7 127.0.0.1:6379> lrange list 0 -1
8 1) "3"
9 2) "4"
10 3) "2"
11 4) "1"
12
```

RPOPLPUSH

1 | 将元素从一个列表转移到另一个列表中

- 语法：

1 | RPOPLPUSH source destination

- 示例：

```
1 127.0.0.1:6379> rpoplpush list newlist
2 "1"
3 127.0.0.1:6379> lrange newlist 0 -1
4 1) "1"
5 127.0.0.1:6379> lrange list 0 -1
6 1) "3"
7 2) "4"
8 3) "2"
```

应用之商品评论列表

- 需求：

```
1 | 用户针对某一商品发布评论，一个商品会被不同的用户进行评论，存储商品评论时，要按时间顺序排序。
2
3 | 用户在前端页面查询该商品的评论，需要按照时间顺序降序排序。
```

- 分析：

1 | 使用list存储商品评论信息，KEY是该商品的ID，VALUE是商品评论信息列表

- 实现：

商品编号为 1001 的商品评论 key 【items: comment:1001】

```
1 192.168.101.3:7001> LPUSH items:comment:1001 '{"id":1,"name":"商品不错，很好！！","date":1430295077289}'
```

set类型

set类型介绍

set 类型即集合类型，其中的数据是不重复且没有顺序。

集合类型和列表类型的对比：

	集合类型	列表类型
存储内容	至多 $2^{32}-1$ 个字符串	至多 $2^{32}-1$ 个字符串
有序性	否	是
唯一性	是	否

集合类型的常用操作是向集合中加入或删除元素、判断某个元素是否存在等，由于集合类型的 Redis 内部是使用值为空的散列表实现，所有这些操作的时间复杂度都为 $O(1)$ 。

Redis 还提供了多个集合之间的交集、并集、差集的运算。

命令

SADD/SREM

1 添加元素/删除元素

语法：

```
1 SADD key member [member ...]
2 SREM key member [member ...]
```

示例：

```
1 127.0.0.1:6379> sadd set a b c
2 (integer) 3
3 127.0.0.1:6379> sadd set a
4 (integer) 0
5 127.0.0.1:6379> srem set c d
6 (integer) 1
```

SMEMBERS

1 获得集合中的所有元素

语法：

```
1 | SMEMBERS key
```

示例：

```
1 | 127.0.0.1:6379> smembers set
2 | 1) "b"
3 | 2) "a"
```

SISMEMBER

```
1 | 判断元素是否在集合中
```

语法：

```
1 | SISMEMBER key member
```

示例：

```
1 | 127.0.0.1:6379>sismember set a
2 | (integer) 1
3 | 127.0.0.1:6379>sismember set h
4 | (integer) 0
```

集合运算命令

SDIFF

```
1 | 集合的差集运算 A-B：属于A并且不属于B的元素构成的集合。
```



语法：

```
1 | SDIFF key [key ...]
```

示例：

```
1 127.0.0.1:6379> sadd setA 1 2 3
2 (integer) 3
3 127.0.0.1:6379> sadd setB 2 3 4
4 (integer) 3
5 127.0.0.1:6379> sdiff setA setB
6 1) "1"
7 127.0.0.1:6379> sdiff setB setA
8 1) "4"
```

SINTER

1 集合的交集运算 $A \cap B$: 属于A且属于B的元素构成的集合。



语法：

```
1 SINTER key [key ...]
```

示例：

```
1 127.0.0.1:6379> sinter setA setB
2 1) "2"
3 2) "3"
```

SUNION

1 集合的并集运算 $A \cup B$: 属于A或者属于B的元素构成的集合

语法：

```
1 SUNION key [key ...]
```

示例：

```
1 127.0.0.1:6379> sunion setA setB
2 1) "1"
3 2) "2"
4 3) "3"
5 4) "4"
```

其它命令(自学)

SCARD

1 | 获得集合中元素的个数

语法：

1 | SCARD key

示例：

```
1 127.0.0.1:6379> smembers setA
2 1) "1"
3 2) "2"
4 3) "3"
5 127.0.0.1:6379> scard setA
6 (integer) 3
7
```

SPOP

1 | 从集合中弹出一个元素。
2 | 注意：由于集合是无序的，所有SPOP命令会从集合中随机选择一个元素弹出

语法：

1 | SPOP key

示例：

```
1 127.0.0.1:6379> spop setA
2 "1"
```

zset类型 (sortedset)

zset介绍

在 set 集合类型的基础上，有序集合类型为集合中的每个元素都关联一个分数，这使得我们不仅可以完成插入、删除和判断元素是否存在于集合中，还能够获得分数最高或最低的前N个元素、获取指定分数范围内的元素等与分数有关的操作。

在某些方面有序集合和列表类型有些相似：

- 1 1、二者都是有序的。
- 2
- 3 2、二者都可以获得某一范围的元素。

但是，二者有着很大区别：

- 1 1、列表类型是通过链表实现的，获取靠近两端的数据速度极快，而当元素增多后，访问中间数据的速度会变慢。
- 2
- 3 2、有序集合类型使用散列表实现，所有即使读取位于中间部分的数据也很快。
- 4
- 5 3、列表中不能简单的调整某个元素的位置，但是有序集合可以（通过更改分数实现）
- 6
- 7 4、有序集合要比列表类型更耗内存。

命令

ZADD

- 1 增加元素。
- 2
- 3 向有序集合中加入一个元素和该元素的分数，如果该元素已经存在则会用新的分数替换原有的分数。返回值是新加入到集合中的元素个数，不包含之前已经存在的元素。

语法：

- 1 ZADD key score member [score member ...]

示例：

- 1 127.0.0.1:6379> zadd scoreboard 80 zhangsan 89 lisi 94 wangwu
- 2 (integer) 3
- 3 127.0.0.1:6379> zadd scoreboard 97 lisi
- 4 (integer) 0
- 5

ZRANGE/ZREVRANGE

- 1 获得排名在某个范围的元素列表。
- 2 - ZRANGE：按照元素分数从小到大的顺序返回索引从start到stop之间的所有元素（包含两端的元素）
- 3 - ZREVRANGE：按照元素分数从大到小的顺序返回索引从start到stop之间的所有元素（包含两端的元素）

语法：

- 1 ZRANGE key start stop [WITHSCORES]
- 2 ZREVRANGE key start stop [WITHSCORES]

示例：

```
1 127.0.0.1:6379> zrange scoreboard 0 2
2 1) "zhangsan"
3 2) "wangwu"
4 3) "lisi"
5 127.0.0.1:6379> zrevrange scoreboard 0 2
6 1) " lisi "
7 2) "wangwu"
8 3) " zhangsan "
```

- 如果需要获得元素的分数的可以在命令尾部加上 WITHSCORES 参数

```
1 127.0.0.1:6379> zrange scoreboard 0 1 WITHSCORES
2 1) "zhangsan"
3 2) "80"
4 3) "wangwu"
5 4) "94"
```

ZSCORE

1 获取元素的分数。

语法：

```
1 ZSCORE key member
```

示例：

```
1 127.0.0.1:6379> zscore scoreboard lisi
2 "97"
```

ZREM

```
1 删除元素。
2 移除有序集合key中的一个或多个成员，不存在的成员将被忽略。
3 当key存在但不是有序集类型时，返回一个错误。
```

语法：

```
1 ZREM key member [member ...]
```

示例：

```
1 127.0.0.1:6379> zrem scoreboard lisi
2 (integer) 1
```

其它命令(自学)

ZRANGEBYSCORE

- 1 | 获得指定分数范围的元素。

语法：

- 1 | ZRANGEBYSCORE key min max [WITHSCORES]

示例：

- ```
1 | 127.0.0.1:6379> ZRANGEBYSCORE scoreboard 90 97 WITHSCORES
2 | 1) "wangwu"
3 | 2) "94"
4 | 3) "lisi"
5 | 4) "97"
6 | 127.0.0.1:6379> ZRANGEBYSCORE scoreboard 70 100 limit 1 2
7 | 1) "wangwu"
8 | 2) "lisi"
```

### ZINCRBY

- 1 | 增加某个元素的分数。
- 2 | 返回值是更改后的分数

语法：

- 1 | ZINCRBY key increment member

示例：

- ```
1 | 127.0.0.1:6379> ZINCRBY scoreboard 4 lisi
2 | "101"
```

ZCARD

- 1 | 获得集合中元素的数量。

语法：

- 1 | ZCARD key

示例：

```
1 127.0.0.1:6379> ZCARD scoreboard
2 (integer) 3
```

ZCOUNT

1 获得指定分数范围内的元素个数

语法：

```
1 ZCOUNT key min max
```

示例：

```
1 127.0.0.1:6379> ZCOUNT scoreboard 80 90
2 (integer) 1
```

ZREMRANGEBYRANK

1 按照排名范围删除元素

语法：

```
1 ZREMRANGEBYRANK key start stop
```

示例：

```
1 127.0.0.1:6379> ZREMRANGEBYRANK scoreboard 0 1
2 (integer) 2
3 127.0.0.1:6379> ZRANGE scoreboard 0 -1
4 1) "lisi"
5
```

ZREMRANGEBYSCORE

1 按照分数范围删除元素

语法：

```
1 ZREMRANGEBYSCORE key min max
```

示例：

```
1 127.0.0.1:6379> zadd scoreboard 84 zhangsan
2 (integer) 1
3 127.0.0.1:6379> ZREMRANGEBYSCORE scoreboard 80 100
4 (integer) 1
5
```

ZRANK/ZREVRANK

```
1 获取元素的排名。
2   - ZRANK : 从小到大
3   - ZREVRANK : 从大到小
```

语法：

```
1 ZRANK key member
2 ZREVRANK key member
```

示例：

```
1 127.0.0.1:6379> ZRANK scoreboard lisi
2 (integer) 0
3 127.0.0.1:6379> ZREVRANK scoreboard zhangsan
4 (integer) 1
```

应用之商品销售排行榜

- 需求：

```
1 根据商品销售量对商品进行排行显示
```

- 设计：

```
1 定义商品销售排行榜 (sorted set集合)，key为items:sellsort，分数为商品销售量。
```

写入商品销售量：

- 商品编号 1001 的销量是 9，商品编号 1002 的销量是 10

```
1 192.168.101.3:7007> ZADD items:sellsort 9 1001 10 1002
```

- 商品编号 1001 的销量加 1

```
1 192.168.101.3:7001> ZINCRBY items:sellsort 1 1001
```

- 商品销量前 10 名：

```
1 192.168.101.3:7001> ZREVRANGE items:sellsort 0 9 withscores
```

通用命令

keys

1 | 返回满足给定pattern 的所有key

语法：

1 | keys pattern

示例：

```
1 redis 127.0.0.1:6379> keys mylist*
2 1) "mylist"
3 2) "mylist5"
4 3) "mylist6"
5 4) "mylist7"
6 5) "mylist8"
```

del

语法：

1 | DEL key

示例：

```
1 127.0.0.1:6379> del test
2 (integer) 1
```

exists

1 | 确认一个key 是否存在

语法：

1 | exists key

示例：从结果来看，数据库中不存在 Hongwan 这个key，但是 age 这个key 是存在的

```
1 redis 127.0.0.1:6379> exists Hongwan
2 (integer) 0
3 redis 127.0.0.1:6379> exists age
4 (integer) 1
5 redis 127.0.0.1:6379>
```

expire (重点)

- 1 | Redis在实际使用过程中更多的用作缓存，然而缓存的数据一般都是需要设置生存时间的，即：到期后数据销毁。

语法：

- 1 | EXPIRE key seconds 设置key的生存时间（单位：秒）key在多少秒后会自动删除
- 2 | TTL key 查看key的生存时间
- 3 | PERSIST key 清除生存时间
- 4 | PEXPIRE key milliseconds 生存时间设置单位为：毫秒

示例：

- 1 | 192.168.101.3:7002> set test 1 设置test的值为1
- 2 | OK
- 3 | 192.168.101.3:7002> get test 获取test的值
- 4 | "1"
- 5 | 192.168.101.3:7002> EXPIRE test 5 设置test的生存时间为5秒
- 6 | (integer) 1
- 7 | 192.168.101.3:7002> TTL test 查看test的生存时间还有1秒删除
- 8 | (integer) 1
- 9 | 192.168.101.3:7002> TTL test
- 10 | (integer) -2
- 11 | 192.168.101.3:7002> get test 获取test的值，已经删除
- 12 | (nil)

rename

- 1 | 重命名key

语法：

- 1 | rename oldkey newkey

示例：age 成功的被我们改名为 age_new 了

- 1 | redis 127.0.0.1:6379[1]> keys *
- 2 | 1) "age"
- 3 | redis 127.0.0.1:6379[1]> rename age age_new
- 4 | OK
- 5 | redis 127.0.0.1:6379[1]> keys *
- 6 | 1) "age_new"
- 7 | redis 127.0.0.1:6379[1]>

type

1 | 显示指定key的数据类型

语法：

1 | type key

示例：这个方法可以非常简单的判断出值的类型

```
1 redis 127.0.0.1:6379> type addr
2 string
3 redis 127.0.0.1:6379> type myzset2
4 zset
5 redis 127.0.0.1:6379> type mylist
6 list
7 redis 127.0.0.1:6379>
8
```

Redis消息模式

队列模式

使用list类型的lpush和rpop实现消息队列

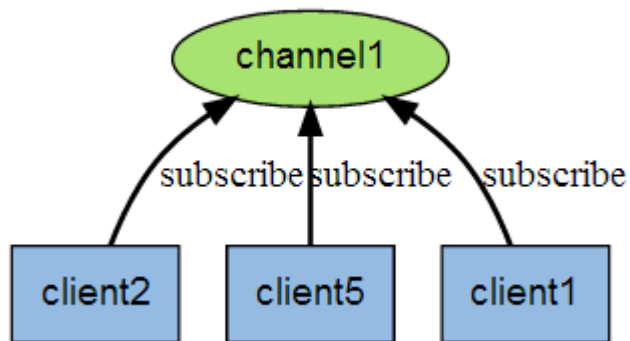


注意事项：

- 消息接收方如果不知道队列中是否有消息，会一直发送rpop命令，如果这样的话，会每一次都建立一次连接，这样显然不好。
- 可以使用brpop命令，它如果从队列中取不出来数据，会一直阻塞，在一定范围内没有取出则返回null、

发布订阅模式

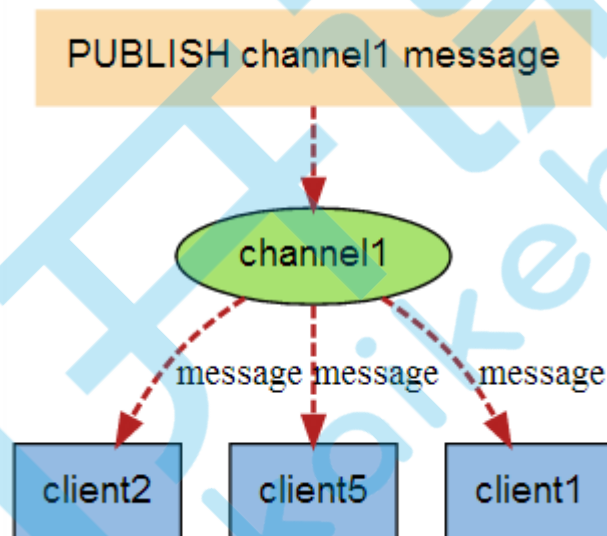
- 订阅消息 (subscribe)



示例：

```
1 | subscribe kkb-channel
```

- 发布消息 (publish)



```
1 | publish kkb-channel "我是灭霸詹"
```

- Redis发布订阅命令

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

Redis事务

Redis事务介绍

- Redis 的事务是通过 `MULTI`、`EXEC`、`DISCARD` 和 `WATCH`、`UNWATCH`这五个命令来完成的。
- Redis 的单个命令都是原子性的，所以这里需要确保事务性的对象是命令集合。
- Redis 将命令集合序列化并确保处于同一事务的命令集合连续且不被打断的执行
- Redis 不支持回滚操作。

事务命令

MULTI

- 用于标记事务块的开始。
- Redis会将后续的命令逐个放入队列中，然后使用`EXEC`命令原子化地执行这个命令序列。

语法：

- `multi`

EXEC

- 在一个事务中执行所有先前放入队列的命令，然后恢复正常的连接状态

语法：

```
1 | exec
```

DISCARD

```
1 | 清除所有先前在一个事务中放入队列的命令，然后恢复正常的连接状态。
```

语法：

```
1 | discard
```

WATCH

```
1 | 当某个[事务需要按条件执行]时，就要使用这个命令将给定的[键设置为受监控]的状态。
```

语法：

```
1 | watch key [key...]
```

注意事项：使用该命令可以实现 Redis 的乐观锁。

UNWATCH

```
1 | 清除所有先前为一个事务监控的键。
```

语法：

```
1 | unwatch
```

事务演示

```
1 | 127.0.0.1:6379> multi
2 | OK
3 | 127.0.0.1:6379> set s1 111
4 | QUEUED
5 | 127.0.0.1:6379> hset set1 name zhangsan
6 | QUEUED
7 | 127.0.0.1:6379> exec
8 | 1) OK
9 | 2) (integer) 1
10 | 127.0.0.1:6379> multi
11 | OK
12 | 127.0.0.1:6379> set s2 222
13 | QUEUED
```

```

14 127.0.0.1:6379> hset set2 age 20
15 QUEUED
16 127.0.0.1:6379> discard
17 OK
18 127.0.0.1:6379> exec
19 (error) ERR EXEC without MULTI
20
21 127.0.0.1:6379> watch s1
22 OK
23 127.0.0.1:6379> multi
24 OK
25 127.0.0.1:6379> set s1 555
26 QUEUED
27 127.0.0.1:6379> exec          # 此时在没有exec之前，通过另一个命令窗口对监控的s1字段进行修改
28 (nil)
29 127.0.0.1:6379> get s1
30 111

```

事务失败处理

- Redis 语法错误（编译期）

```

127.0.0.1:6379> multi
OK
127.0.0.1:6379> sets s1 111
(error) ERR unknown command 'sets'
127.0.0.1:6379> set s1
(error) ERR wrong number of arguments for 'set' command
127.0.0.1:6379> set s4 444
QUEUED
127.0.0.1:6379> exec
(error) EXECABORT Transaction discarded because of previous errors.
127.0.0.1:6379> get s4
(nil)

```

- Redis 运行错误

```

127.0.0.1:6379> multi
OK
127.0.0.1:6379> set s4 444
QUEUED
127.0.0.1:6379> lpush s4 111 222
QUEUED
127.0.0.1:6379> exec
1) OK
2) (error) WRONGTYPE Operation against a key holding the wrong kind of value
127.0.0.1:6379> get s4
"444"
127.0.0.1:6379>

```

- Redis 不支持事务回滚（为什么呢）

- 大多数事务失败是因为语法错误或者类型错误，这两种错误，在开发阶段都是可以预见的
- Redis 为了性能方面就忽略了事务回滚。

扩展点

spring-data-redis

本课总结

布置作业

课后互动问答

下节预告

