# Redis

## 概述

### 在JavaWeb中的应用

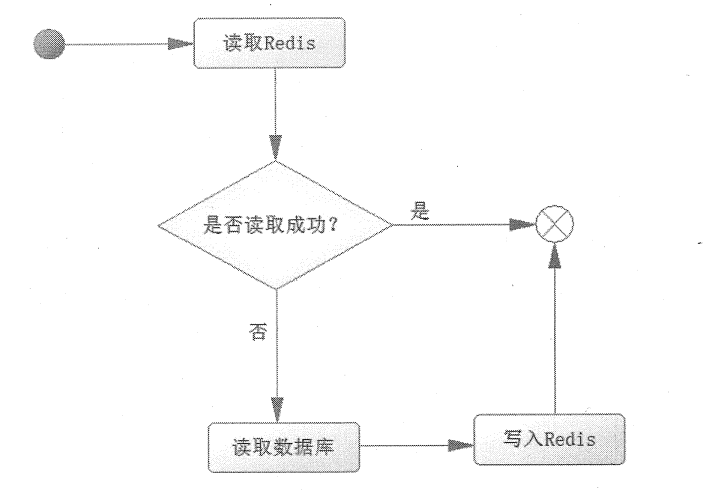
#### 缓存使用

使用redis作为缓存，需要从三个角度思考：

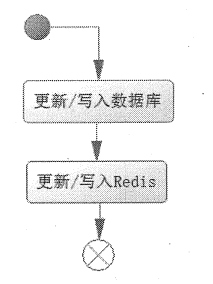
* 业务数据是否常用，命中率高吗？
* 读操作多还是写操作多？写操作多就没必要用redis；
* 业务数据大小如何？如果过大，缓存压力会很大，则是否有必要？

redis作为缓存的读取逻辑：

第一次读取失败，此时读取数据库，并且写入redis；第二次直接读取redis。

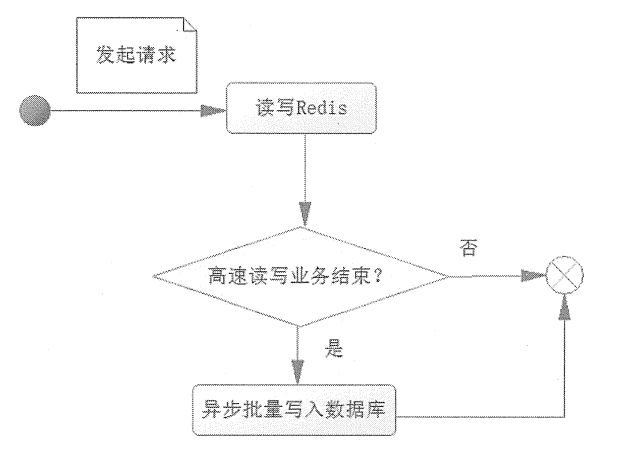


redis写的逻辑：



#### 高速读写场合使用

场景例如：抢红包，秒杀商品等。



### redis的基本安装和使用

**Windows环境下**

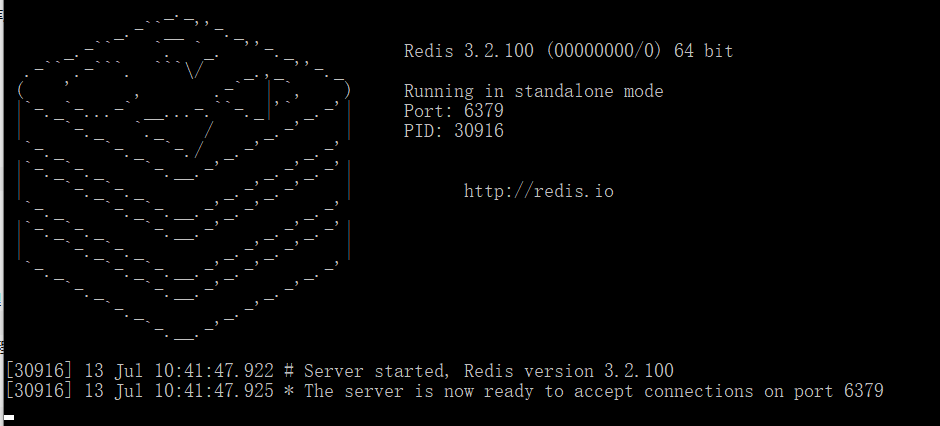
下载地址：

<https://github.com/microsoftarchive/redis/releases/tag/win-3.2.100>

在解压后的目录下新建文件startup.cmd帮助启动redis

文件中录入：意思是通过redis-server的命令读取redis-windows.conf配件内容启动。

redis-server redis.windows.conf



### redis中的JavaAPI

#### Java中使用redis

##### 1）配置maven依赖

|  |
| --- |
| <**dependency**>  <**groupId**>redis.clients</**groupId**>  <**artifactId**>jedis</**artifactId**>  <**version**>2.9.0</**version**> </**dependency**> |

##### 2）测试代码

|  |
| --- |
| @Test **public void** testJedisPool() {  JedisPoolConfig poolCfg = **new** JedisPoolConfig();*//配置对象* poolCfg.setMaxIdle(50);*//最大空闲数* poolCfg.setMaxTotal(100);*//最大连接数* poolCfg.setMaxWaitMillis(50000);*//最大等待毫秒数* JedisPool jp = **new** JedisPool(poolCfg, **"localhost"**);*//通过配置对象创建连接池* Jedis jedis = jp.getResource();*//从连接池获取链接* **int** i = 0;  **try** {  **long** start = System.*currentTimeMillis*();  **while** (**true**) {  **long** end = System.*currentTimeMillis*();  **if**(end - start >= 1000) {  **break**;  }  i++;  jedis.set(**"test"** + i, i + **""**);  }  } **finally** {  jedis.close();  }  System.***out***.println(**"redis每秒操作："** + i + **"次"**); } |

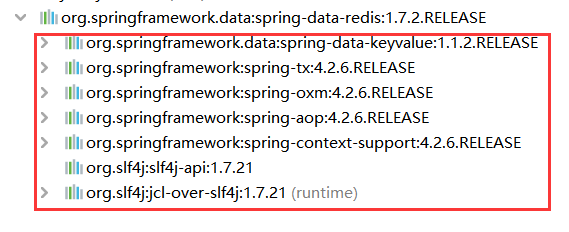
#### Spring中使用redis

由于java中使用的是对象，而redis存储的是字符串数据，因此需要一个转换器将数据转换成java对象。Spring提供了这样的转换类。

1）导入依赖：

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.data</**groupId**>  <**artifactId**>spring-data-redis</**artifactId**>  <**version**>1.7.2.RELEASE</**version**> </**dependency**> |

Maven会自动将相关的Spring框架包导入：

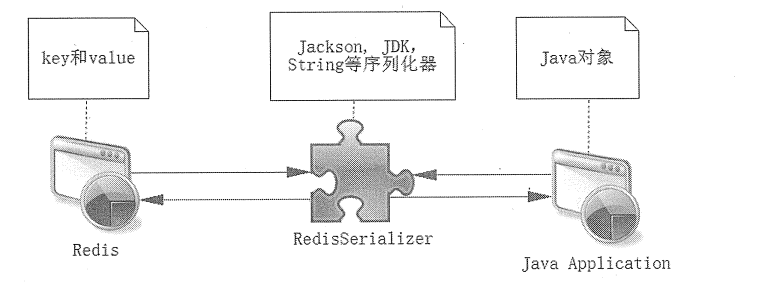


1. 配置RedisConnectionFactory

|  |
| --- |
| <**bean id="poolConfig" class="redis.clients.jedis.JedisPoolConfig"**>  <**property name="maxIdle" value="50"**/>*<!--最大空闲数-->* <**property name="maxWaitMillis" value="3000"**/>*<!--最大等待毫秒数-->* <**property name="maxTotal" value="100"**/>*<!--最大链接-->* </**bean**> <**bean id="connectionFactory" class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory"**>  <**property name="hostName" value="localhost"**/>  <**property name="port" value="6379"**/>  <**property name="poolConfig" ref="poolConfig"**/> </**bean**> |

3)配置序列化器：

Spring序列化器



|  |
| --- |
| *<!--value序列化器-->* <**bean id="jdkSerializationRedisSerializer" class="org.springframework.data.redis.serializer.JdkSerializationRedisSerializer"**/> *<!--key序列化器-->* <**bean id="stringRedisSerializer" class="org.springframework.data.redis.serializer.StringRedisSerializer"**/> <**bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate"**>  <**property name="connectionFactory" ref="connectionFactory"**/>  <**property name="keySerializer" ref="stringRedisSerializer"**/>  <**property name="valueSerializer" ref="jdkSerializationRedisSerializer"** /> </**bean**> |

测试代码：

|  |
| --- |
| @Test **public void** testRedisTemplate() {  //这里的ac是Spring容器  RedisTemplate rt = **ac**.getBean(RedisTemplate.**class**);  Role role1 = **new** Role();  role1.setId(1l);  role1.setRoleName(**"经理"**);  role1.setNote(**"note"**);  rt.opsForValue().set(**"role\_1"**, role1);  Role role2 = (Role)rt.opsForValue().get(**"role\_1"**);  System.***out***.println(role2); } |

另一种方式：

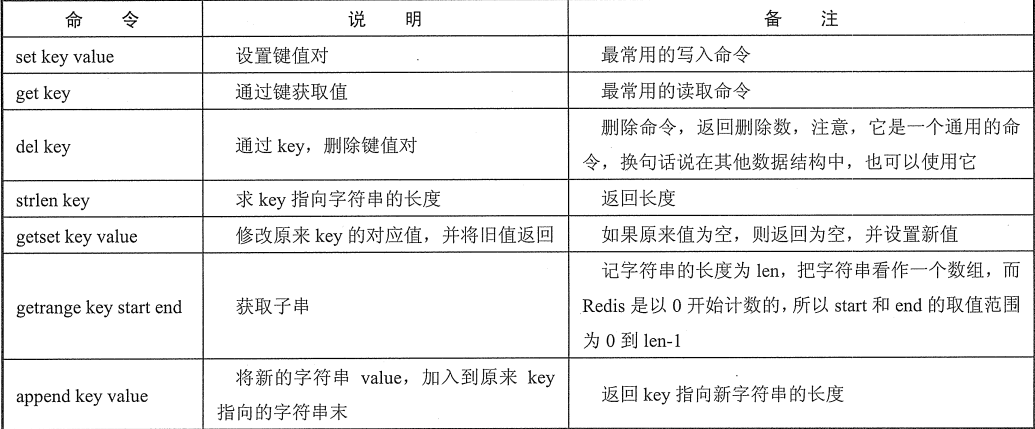
|  |
| --- |
| @Test **public void** testSeesionCallback() {  RedisTemplate rt = **ac**.getBean(RedisTemplate.**class**);  Role role1 = **new** Role();  role1.setId(1l);  role1.setRoleName(**"经理"**);  role1.setNote(**"note"**);  rt.boundValueOps(**"role1"**).set(role1);  Role role2 = (Role)rt.boundValueOps(**"role1"**).get();  System.***out***.println(role2); } |

### redis的6种数据类型

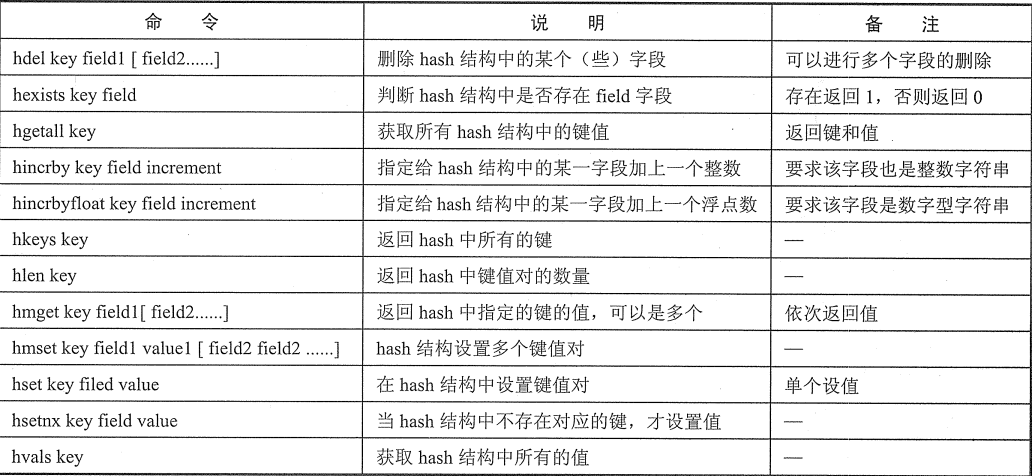
|  |  |  |
| --- | --- | --- |
| 数据类型 | 数据类型存储的值 | 说明 |
| STRING（字符串） | 字符串，整数，浮点数 |  |
| LIST（列表） | 链表，每个节点是个字符串 |  |
| SET（集合） | 收集器，无序且唯一的字符串元素 |  |
| HASH（哈希散列表） | 键值对应的无序列表 |  |
| ZSET（有序集合） | 包含字符串，整数，浮点数，分值。元素的排序根据分值确定 |  |
| HyperLogLog（基数） | 计算重复的值，以确定存储的数量。 |  |

## redis数据结构常用命令

### 字符串



### 哈希



### 链表

### 集合

### 有序集合

### 基数

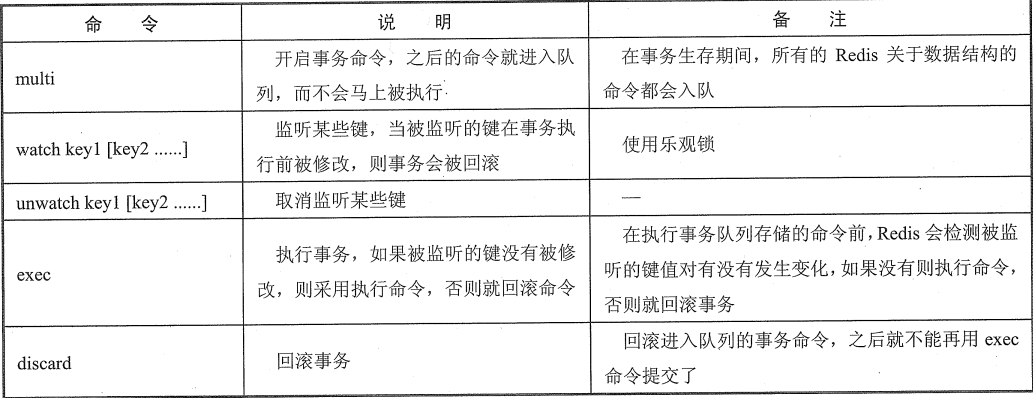
## Redis的一些常用技术

Redis是存在事务的，使用MULTI-EXEC的命令组合。它提供了隔离性和原子性的操作。

Redis使用事务会经历三个过程：

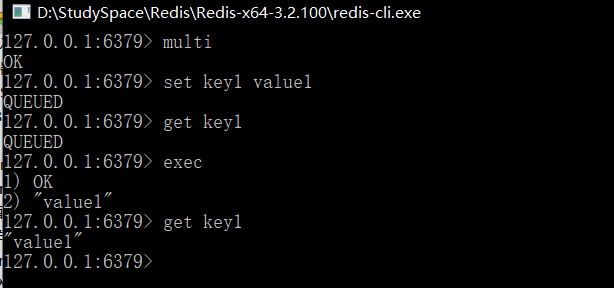
* 开启事务
* 命令进入队列
* 执行事务

**Redis事务命令**



### Redis的基础事务

multi命令到exec命令之间的redis命令会采取进入队列的形式，直到exec命令的出现，才会一次性发送队列中的命令执行，**执行时其它客户端不能再插入任何命令了**。这就是Redis的事务机制。



spring使用代码：

|  |
| --- |
| @Test **public void** testMulti() {  SessionCallback callback = **new** SessionCallback() {  @Override  **public** Object execute(RedisOperations ops) **throws** DataAccessException {  ops.multi();*//开启事务* ops.boundValueOps(**"key1"**).set(**"value1"**);  System.***out***.println(ops.boundValueOps(**"key1"**).get());*//处于消息队列中未执行，所以输出为空* List list = ops.exec();*//list保存所有命令的执行结果* String value = (String)**redisTemplate**.opsForValue().get(**"key1"**);  **return** value;  }  } ;  String value = (String)**redisTemplate**.execute(callback);  System.***out***.println(value); } |

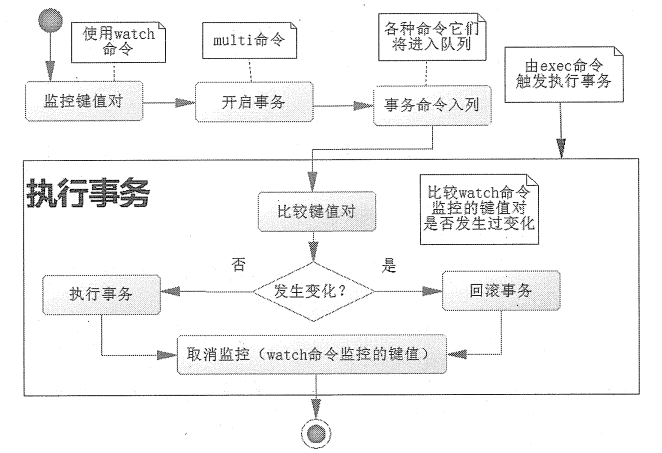
### Redis的事务回滚

redis事务中的命令出现问题会分为两类：

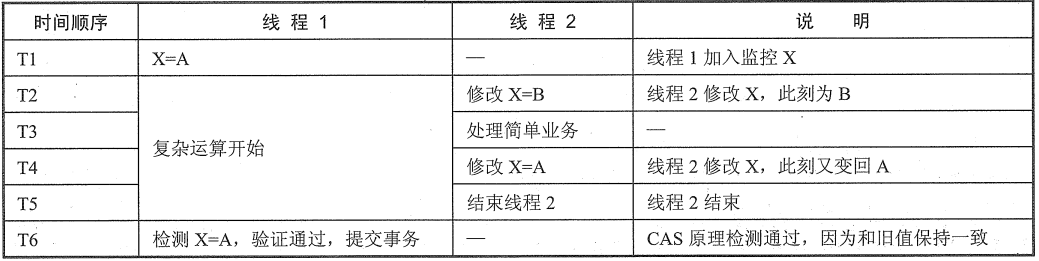
1. 某行命令操作的数据格式错误，则改行命令执行失败，不影响其它命令执行
2. 命令格式错误，则整个事务都会被回滚，事务中没有命令会被执行。

### watch命令监控事务

在multi命令前执行watch监控某些键值对，之后开启事务，命令进入队列，执行exec时先会比对watch命令监控的键值对是否发生变化。如果没有变化则执行命令提交事务，负责则事务回滚，不会执行任何命令。最后不论是否回滚，都会取消之前的watch命令。



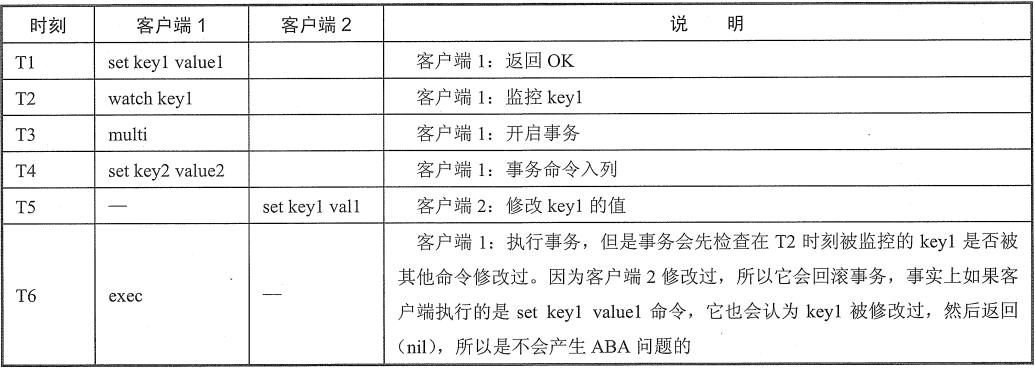
这里用了CAS乐观锁的原理，即每次线程执行任务前先拷贝多线程共享数据作为副本，执行时查看该数据有无修改过，若修改过则当前线程为了保持数据已执行则放弃执行。但是这个原理设计有个缺陷会导致ABA问题：



因此记录旧值去比较是不够的，还需要探测version字段，每操作一次version = version+1，排除ABA问题。

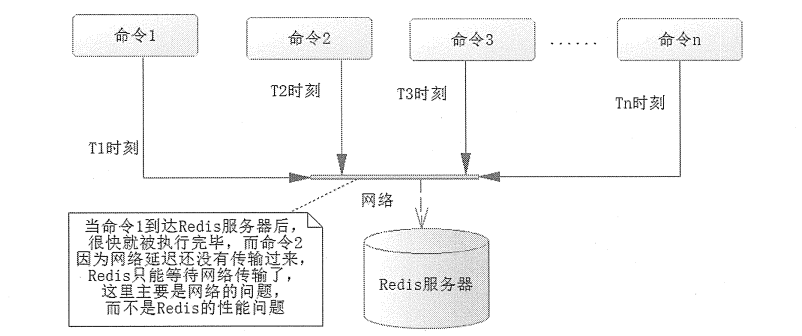
Redis不会通过乐观锁不会阻塞其它事务执行，通过watch保持数据一致性在多线程环境中并发执行。

示例：



### 流水线

使用multi和exec会检测对应的锁和序列化命令，造成了系统开销。有时候希望没有任何附加条件下使用队列批量执行命令，从而提高性能，这就是Redis流水线技术。



redis流水线技术可以解决这个问题，它是一种协议。

测试性能代码：

JavaAPI:

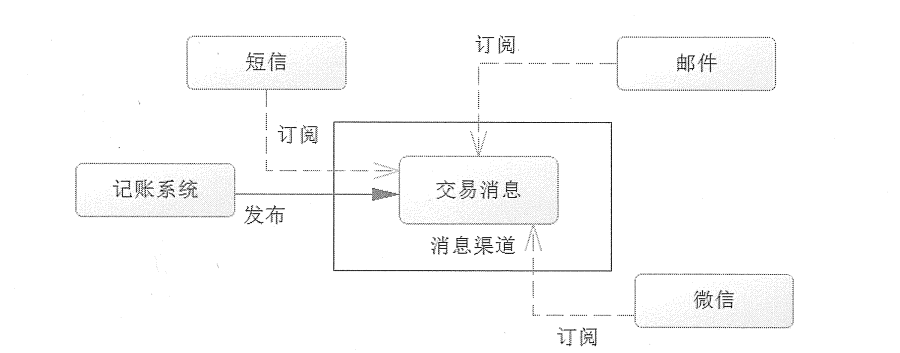
|  |
| --- |
| @Test **public void** testPipelined() {  JedisPoolConfig poolCfg = **new** JedisPoolConfig();*//配置对象* poolCfg.setMaxIdle(50);*//最大空闲数* poolCfg.setMaxTotal(100);*//最大连接数* poolCfg.setMaxWaitMillis(50000);*//最大等待毫秒数* JedisPool jp = **new** JedisPool(poolCfg, **"localhost"**);*//通过配置对象创建连接池* Jedis jedis = jp.getResource();*//从连接池获取链接* Pipeline pipeline = jedis.pipelined();*//开启流水线* **long** start = System.*currentTimeMillis*();  **for**(**int** i = 0; i < 100000; i++) {  **int** j = i + 1;  pipeline.set(**"key"** + j, **"value"** + j);  pipeline.get(**"key"** + j);  }  List list = pipeline.syncAndReturnAll();  **long** end = System.*currentTimeMillis*();  System.***out***.println(**"耗时："** + (end - start) + **"毫秒"**); } |

Spring:

|  |
| --- |
| @Test **public void** testPipelined() {  SessionCallback callback = **new** SessionCallback() {  @Override  **public** Object execute(RedisOperations ops) **throws** DataAccessException {  **for**(**int** i = 0; i < 100000; i++) {  ops.boundValueOps(**"i"**).set(i);  ops.boundValueOps(**"i"**).get();  }  **return null**;  }  };  **long** start = System.*currentTimeMillis*();  List list = **redisTemplate**.executePipelined(callback);  **long** end = System.*currentTimeMillis*();  System.***out***.println(end - start); } |

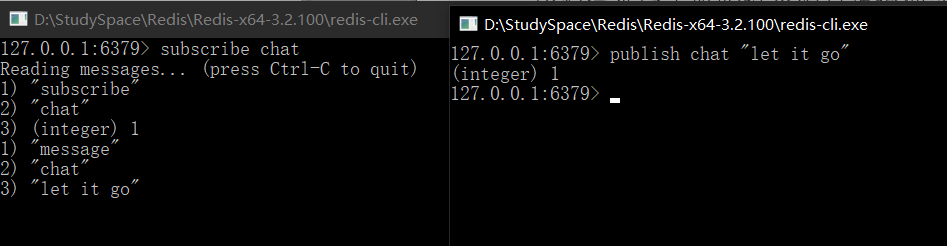
使用redistemplate要比javaAPI要慢一点。

### 发布订阅



一旦记账系统发布了交易消息到消息渠道，则所有监听该消息渠道的系统就能拿到这个消息。

在redis可以通过subscribe和publish命令订阅和发布消息：



### Redis超时命令



如果一个key超时了，Redis不会回收它的存储空间，只是标识它超时了。因为当一个键很大时，回收会耗费时间，会导致服务停顿。Redis有两种方式回收超时键值对：

* 定时回收。在某个确定的时间触发代码，回收超时的键值对。
* 惰性回收。当一个超时的键被再次get命令访问时，触发redis从内存中清除。

### 使用Lua语言

在redis2.6以上版本中，可以通过Lua语言操作redis。执行lua语言是原子性的。

* 可以直接输入一些lua语言的程序代码
* 将lua语言编写成文件

## Redis配置

### Redis基础配置文件

放置在安装目录下，如果windows下则是redis.windows.conf；如果是linux则是redis.conf

### Redis备份（持久化）

### Redis内存回收策略

Redis会因为内存不足导致错误，也会因为长时间回收导致系统长期停顿。

在Redis配置文件中，当内存达到规定的最大值时，允许配置6种策略中的一种进行淘汰键值。