# Day04

# redis学习

## 01/ nosql介绍

NoSQL：一类新出现的数据库(not only sql)，它的特点：

1. 不支持SQL语法
2. 存储结构跟传统关系型数据库中的那种关系表完全不同，nosql中存储的数据都是KV形式
3. NoSQL的世界中没有一种通用的语言，每种nosql数据库都有自己的api和语法，以及擅长的业务场景
4. NoSQL中的产品种类相当多：
   1. Mongodb 文档型nosql数据库，擅长做CMS系统（内容管理系统）
   2. Redis 内存数据库，**数据结构**服务器，号称瑞士军刀（精巧），只要你有足够的想象力，它可以还给你无限惊喜
   3. Hbase hadoop生态系统中原生的一种nosql数据库，重量级的分布式nosql数据库，用于海量数据的场景
   4. Cassandra hadoop生态系统中原生的一种分布式nosql数据库，后起之秀

NoSQL和SQL数据库的比较：

1、适用场景不同：sql数据库适合用于关系特别复杂的数据查询场景，nosql反之

2、“事务”特性的支持：sql对事务的支持非常完善，而nosql基本不支持事务

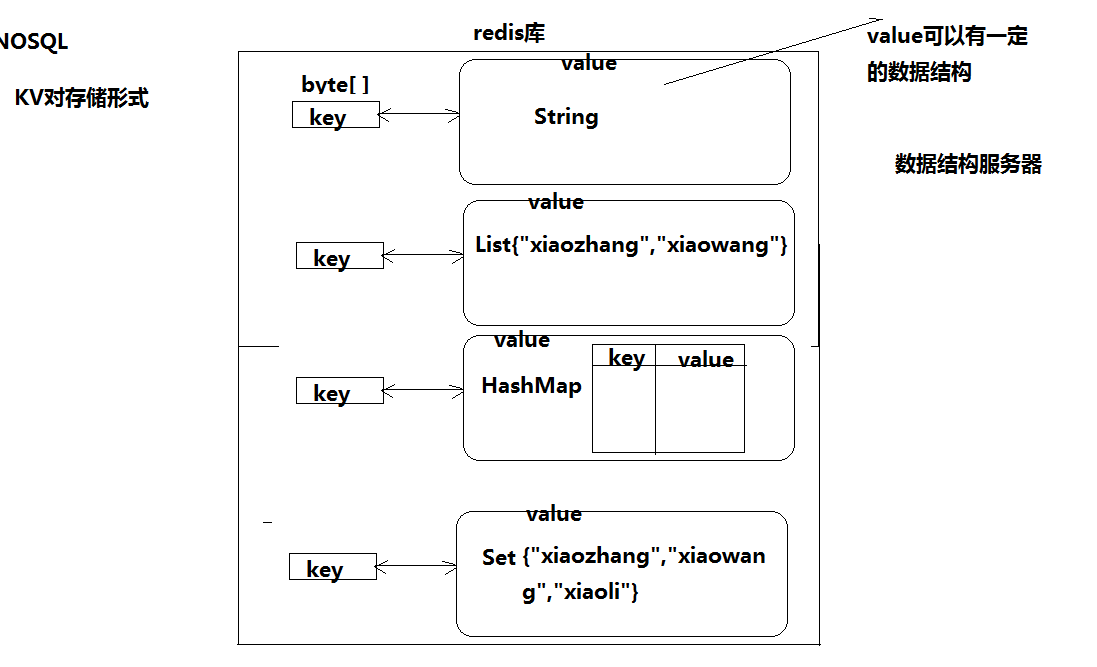
3、两者在不断地取长补短，呈现融合趋势

## 02/ redis介绍

### 2.1 简述

Redis是一个高性能的**kv对**缓存和内存数据库(存的不像mysql那样的表)

Redis的存储结构就是key-value，形式如下：



注： redis中的value内部可以支持各种数据结构类型，比如可以存入一个普通的string，还可以存list，set，hashmap，sortedSet（有序的set）

### 2.2 redis应用场景

A、用来做缓存(ehcache/memcached)——redis的所有数据是放在内存中的（内存数据库）

B、可以在某些特定应用场景下替代传统数据库——比如社交类的应用

C、在一些大型系统中，巧妙地实现一些特定的功能：session共享、购物车

只要你有丰富的想象力，redis可以用在可以给你无限的惊喜…….

### 2.3 redis的特性

1、redis数据访问速度快（数据在内存中）

2、redis有数据持久化机制（持久化机制有两种：1、定期将内存数据dump到磁盘；2、aof(append only file)持久化机制——用记日志的方式记录每一条数据更新操作，一旦出现灾难事件，可以通过日志重放来恢复整个数据库）

3、redis支持集群模式（容量可以线性扩展）

4、redis相比其他缓存工具（ehcach/memcached），有一个鲜明的优势：支持丰富的数据结构

## 03/ 安装redis

### 3.1 获取源码包

1、先去官网（<http://redis.io/download> ）下载一个源码工程（redis官网版本只支持linux/微软开源事业部维护了一个windows版本）

2、把安装包上传到服务器，解压缩

### 3.2 编译源码

1、切换到解压出来的源码工程目录中

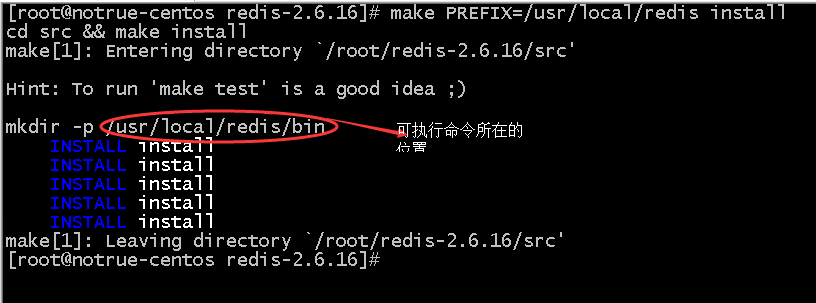
cd redis-2.6.16

2、用make命令来对redis的c语言源码工程进行编译

3、编译完成之后，用make install命令进行安装

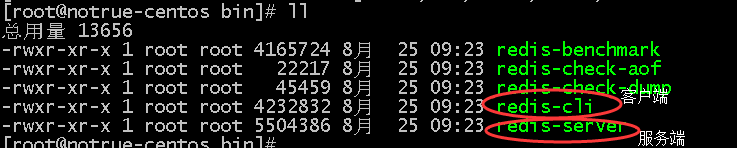
|  |
| --- |
| [root@notrue-centos redis-2.6.16]# make PREFIX=/usr/local/redis install |

安装成功的显示：



### 3.3 启动redis服务

#### 1、进入redis的bin目录



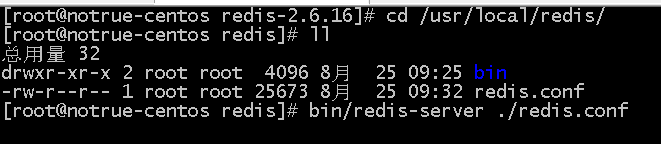
#### 2、准备配置文件

Redis服务在启动的时候可以指定配置文件，那，我们可以从redis的源码目录中拷贝一份配置文件模板到redis的安装目录，修改后使用

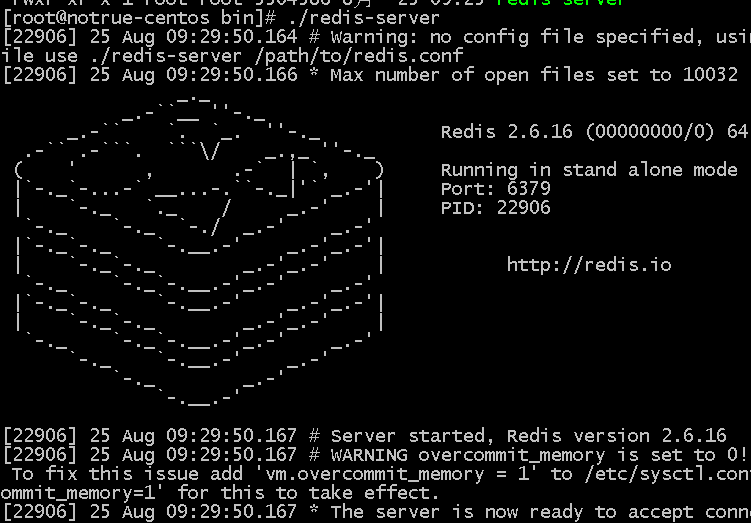
|  |
| --- |
| [root@notrue-centos redis-2.6.16]# cp /root/redis-2.6.16/redis.conf /usr/local/redis/ |

#### 3、启动redis服务

并指定使用的配置文件



#### 4、启动成功的显示



#### 5、启动为后台服务

上述启动方法，会让redis服务进程运行在console前台，最好应该放到后台运行，可将启动命令改为如下方式：

**1/ 方式一**

|  |
| --- |
| [root@notrue-centos redis]# nohup bin/redis-server ./redis.conf 1>/dev/null 2>&1 & |

Nohup：控制台关闭或闲置超时，也不退出

1>/dev/null ： 把程序的“1”——标准输出，重定向到文件/dev/null

2>&1 ： 把程序的“2”——错误输出，重定向到“1”所去的文件

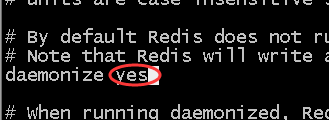
& : 把程序放到后台运行

**2/ 方式二**

修改配置文件，

vi redis.conf

修改其中一个配置



保存文件后再用普通命令启动，也可以启动为后台模式

[root@notrue-centos redis]# bin/redis-server ./redis.conf

## 04/ 客户端连接

1. 用redis自带的命令行客户端

|  |
| --- |
| [root@notrue-centos redis]# bin/redis-cli -h notrue-centos -p 6379  redis notrue-centos:6379> ping  PONG  redis notrue-centos:6379> |

1. 或者用redis的api客户端连接

新建一个maven工程，导入jedis的maven依赖坐标

|  |
| --- |
| <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  <version>2.7.2</version>  <type>jar</type>  <scope>compile</scope>  </dependency> |

然后写一个类用来测试服务器跟客户端的连通性：

|  |
| --- |
| public class RedisClientConnectionTest {  public static void main(String[] args) {  // 构造一个redis的客户端对象  Jedis jedis = new Jedis("pinshutang.zicp.net", 6379);  String ping = jedis.ping();  System.out.println(ping);  }  } |

## 05/ Redis的数据功能

### 5.1 String类型的数据

(常作为缓存使用)

1/插入和读取一条string类型的数据

|  |
| --- |
| redis notrue-centos:6379> set sessionid-0001 "zhangsan"  OK  redis notrue-centos:6379> get sessionid-0001  "zhangsan" |

2/对string类型数据进行增减（前提是这条数据的value可以看成数字）

|  |
| --- |
| DECR key  INCR key  DECRBY key decrement  INCRBY key increment |

3/一次性插入或者获取多条数据

|  |
| --- |
| MGET key1 key2  MSET key1 value1 key2 value2 ….. |

4/在插入一条string类型数据的同时为它指定一个存活期限

|  |
| --- |
| setex bancao 10 weige  ### bancao这条数据就只会存活10秒钟，过期会被redis自动清除 |

应用：将一个自定义的对象比如product存入redis

实现方式一：将对象序列化成byte数组

|  |
| --- |
| /\*\*  \* 将对象缓存到redis的string结构数据中  \* @throws Exception  \*  \*/  @Test  public void testObjectCache() throws Exception{    ProductInfo p = new ProductInfo();    p.setName("苏菲");  p.setDescription("angelababy专用");  p.setCatelog("unknow");  p.setPrice(10.8);  //将对象序列化成字节数组    ByteArrayOutputStream ba = new ByteArrayOutputStream();  ObjectOutputStream oos = new ObjectOutputStream(ba);    //用对象序列化流来将p对象序列化，然后把序列化之后的二进制数据写到ba流中  oos.writeObject(p);    //将ba流转成byte数组  byte[] pBytes = ba.toByteArray();        //将对象序列化之后的byte数组存到redis的string结构数据中  jedis.set("product:01".getBytes(), pBytes);      //根据key从redis中取出对象的byte数据  byte[] pBytesResp = jedis.get("product:01".getBytes());    //将byte数据反序列出对象  ByteArrayInputStream bi = new ByteArrayInputStream(pBytesResp);    ObjectInputStream oi = new ObjectInputStream(bi);    //从对象读取流中读取出p对象  ProductInfo pResp = (ProductInfo) oi.readObject();    System.out.println(pResp);      } |

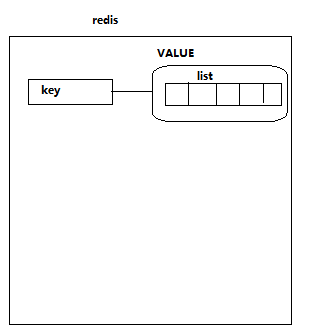
实现方式二：

将对象转成json字符串来存取

|  |
| --- |
| /\*\*  \* 将对象转成json字符串缓存到redis的string结构数据中  \*/  @Test  public void testObjectToJsonCache(){      ProductInfo p = new ProductInfo();    p.setName("ABC");  p.setDescription("刘亦菲专用");  p.setCatelog("夜用型");  p.setPrice(10.8);  //利用gson将对象转成json串  Gson gson = new Gson();  String pJson = gson.toJson(p);    //将json串存入redis  jedis.set("prodcut:02", pJson);      //从redis中取出对象的json串  String pJsonResp = jedis.get("prodcut:02");    //将返回的json解析成对象  ProductInfo pResponse = gson.fromJson(pJsonResp, ProductInfo.class);    //显示对象的属性  System.out.println(pResponse);      } |

### 5.2 List数据结构

#### 5.2.1 List图示



#### 5.2.2 List功能演示

|  |
| --- |
| #从头部（左边）插入数据  redis>LPUSH key value1 value2 value3  #从尾部（右边）插入数据  redis>RPUSH key value1 value2 value3  #读取list中指定范围的values  redis>LRANGE key start end  redis> lrange task-queue 0 -1 读取整个list  #从头部弹出一个元素  LPOP key  #从尾部弹出一个元素  RPOP key  #从一个list的尾部弹出一个元素插入到另一个list  RPOPLPUSH key1 key2 ## 这是一个原子性操作 |

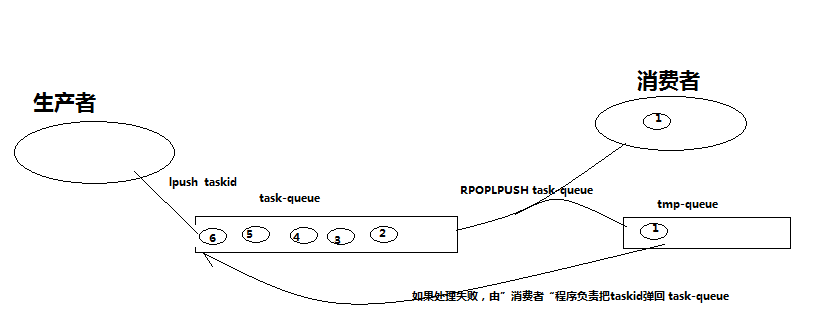
#### 5.2.3 List的应用案例demo

##### 1 需求描述

任务调度系统：

生产者不断产生任务，放入task-queue排队

消费者不断拿出任务来处理，同时放入一个tmp-queue暂存，如果任务处理成功，则清除tmp-queue，否则，将任务弹回task-queue



##### 2 代码实现

**1/ 生产者**

——模拟产生任务

|  |
| --- |
| public class TaskProducer {  // 获取一个redis的客户端连接对象  public static Jedis getRedisConnection(String host, int port) {  Jedis jedis = new Jedis(host, port);  return jedis;  }  public static void main(String[] args) {  Jedis jedis = getRedisConnection("angelababy", 6379);  Random random = new Random();  // 生成任务  while (true) {  try {  // 生成任务的速度有一定的随机性，在1-2秒之间  Thread.sleep(random.nextInt(1000) + 1000);  // 生成一个任务  String taskid = UUID.randomUUID().toString();  // 往任务队列"task-queue"中插入，第一次插入时，"task-queue"还不存在  //但是lpush方法会在redis库中创建一条新的list数据  jedis.lpush("task-queue", taskid);  System.out.println("向任务队列中插入了一个新的任务: " + taskid);  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  }  } |

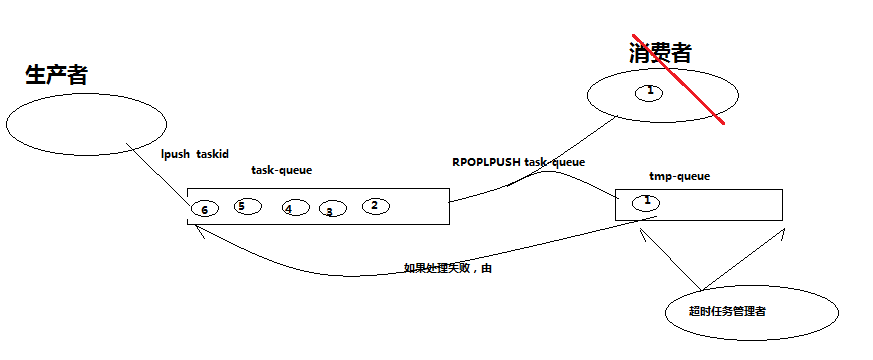
**2/ 消费者**

——模拟处理任务，并且管理暂存队列

|  |
| --- |
| public class TaskConsumer {  public static void main(String[] args) {  Jedis jedis = new Jedis("angelababy", 6379);  Random random = new Random();  while (true) {  try {  // 从task-queue中取一个任务，同时放入"tmp-queue"  String taskid = jedis.rpoplpush("task-queue", "tmp-queue");  // 模拟处理任务  Thread.sleep(1000);  // 模拟有成功又有失败的情况  int nextInt = random.nextInt(13);  if (nextInt % 7 == 0) { // 模拟失败的情况  // 失败的情况下，需要将任务从"tmp-queue"弹回"task-queue"  jedis.rpoplpush("tmp-queue", "task-queue");  System.out.println("-------任务处理失败： " + taskid);  } else { // 模拟成功的情况  // 成功的情况下，将任务从"tmp-queue"清除  jedis.rpop("tmp-queue");  System.out.println("任务处理成功： " + taskid);  }  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  }  } |

上述机制是一个简化版，真实版的任务调度系统会更加复杂，如下所示：

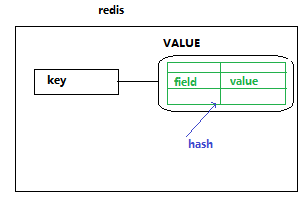
（增加了一个专门用来管理暂存队列的角色，以便就算消费者程序失败退出，那些处理失败的任务依然可以被弹回task-queue）



### 5.3 Hash数据结构

#### 5.3.1 Hash图示

Redis中的Hashes类型可以看成具有String Key和String Value的map容器



#### 5.3.2 Hash功能演示

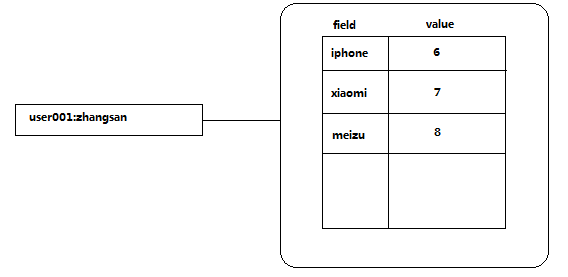
1、往redis库中插入一条hash类型的数据

|  |
| --- |
| redis> HSET key field value |

举例：

|  |
| --- |
| redis 127.0.0.1:6379> hset user001:zhangsan iphone 6  (integer) 1  redis 127.0.0.1:6379> hset user001:zhangsan xiaomi 7  (integer) 1  redis 127.0.0.1:6379> hset user001:zhangsan meizu 8  (integer) 1 |

在redis库中就形成了这样一条数据：



2、从redis库中获取一条hash类型数据的value

* 取出一条hash类型数据中所有field-value对

|  |
| --- |
| redis 127.0.0.1:6379> hgetall user001:zhangsan  1) "iphone"  2) "6"  3) "xiaomi"  4) "7"  5) "meizu"  6) "8" |

* 取出hash数据中所有fields

|  |
| --- |
| redis 127.0.0.1:6379> HKEYS user001:zhangsan  1) "iphone"  2) "xiaomi"  3) "meizu" |

* 取出hash数据中所有的value

|  |
| --- |
| redis 127.0.0.1:6379> hvals user001:zhangsan  1) "6"  2) "7"  3) "8" |

* 取出hash数据中一个指定field的值

|  |
| --- |
| redis 127.0.0.1:6379> hget user001:zhangsan xiaomi  "8" |

* 为hash数据中指定的一个field的值进行增减

|  |
| --- |
| redis 127.0.0.1:6379> HINCRBY user001:zhangsan xiaomi 1  (integer) 8 |

* 从hash数据中删除一个字段field及其值

|  |
| --- |
| redis 127.0.0.1:6379> hgetall user001:zhangsan  1) "iphone"  2) "6"  3) "xiaomi"  4) "7"  5) "meizu"  6) "8"  redis 127.0.0.1:6379> HDEL user001:zhangsan iphone  (integer) 1  redis 127.0.0.1:6379> hgetall user001:zhangsan  1) "xiaomi"  2) "7"  3) "meizu"  4) "8" |

### 5.4 Set数据结构功能

集合的特点：无序、无重复元素

1. 插入一条set数据

|  |
| --- |
| redis 127.0.0.1:6379> sadd frieds:zhangsan bingbing baby fengjie furong ruhua tingting  (integer) 6  redis 127.0.0.1:6379> scard frieds:zhangsan  (integer) 6  redis 127.0.0.1:6379> |

2、获取一条set数据的所有members

|  |
| --- |
| redis 127.0.0.1:6379> smembers frieds:zhangsan  1) "fengjie"  2) "baby"  3) "furong"  4) "bingbing"  5) "tingting"  6) "ruhua" |

3、判断一个成员是否属于某条指定的set数据

|  |
| --- |
| redis 127.0.0.1:6379> sismember frieds:zhangsan liuyifei #如果不是，则返回0  (integer) 0  redis 127.0.0.1:6379> sismember frieds:zhangsan baby #如果是，则返回1  (integer) 1 |

4、求两个set数据的差集

|  |
| --- |
| #求差集  redis 127.0.0.1:6379> sdiff frieds:zhangsan friends:xiaotao  1) "furong"  2) "fengjie"  3) "ruhua"  4) "feifei"  #求差集，并将结果存入到另一个set  redis 127.0.0.1:6379> sdiffstore zhangsan-xiaotao frieds:zhangsan friends:xiaotao  (integer) 4  #查看差集结果  redis 127.0.0.1:6379> smembers zhangsan-xiaotao  1) "furong"  2) "fengjie"  3) "ruhua"  4) "feifei" |

1. 求交集，求并集

|  |
| --- |
| #求交集  redis 127.0.0.1:6379> sinterstore zhangsan:xiaotao frieds:zhangsan friends:xiaotao  (integer) 2  redis 127.0.0.1:6379> smembers zhangsan:xiaotao  1) "bingbing"  2) "baby"  #求并集  redis 127.0.0.1:6379> sunion frieds:zhangsan friends:xiaotao  1) "fengjie"  2) "tangwei"  3) "liuyifei"  4) "bingbing"  5) "ruhua"  6) "feifei"  7) "baby"  8) "songhuiqiao"  9) "furong"  10) "yangmi" |

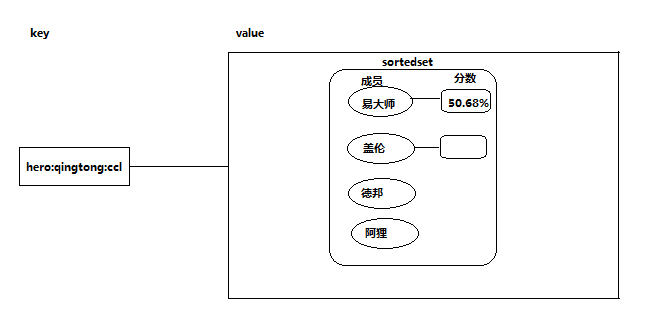
### 5.5 sortedSet（有序集合）数据结构

#### 5.5.1 sortedSet图示

sortedset中存储的成员都有一个附带的分数值

而redis就可以根据分数来对成员进行各种排序（正序、倒序）

1. sortedSet存储内容示意图：



#### 5.5.2 SortedSet功能演示

1、往redis库中插入一条sortedset数据

|  |
| --- |
| redis 127.0.0.1:6379> zadd nansheng:yanzhi:bang 70 liudehua 90 huangbo 100 weixiaobao 250 yangwei 59 xiaotao  (integer) 5 |

1. 从sortedset中查询有序结果

|  |
| --- |
| #正序结果  redis 127.0.0.1:6379> zrange nanshen:yanzhi:bang 0 4  1) "xiaotao"  2) "liudehua"  3) "huangbo"  4) "weixiaobao"  5) "yangwei"  #倒序结果  redis 127.0.0.1:6379> zrevrange nanshen:yanzhi:bang 0 4  1) "yangwei"  2) "weixiaobao"  3) "huangbo"  4) "liudehua"  5) "xiaotao" |

1. 查询某个成员的名次

|  |
| --- |
| #在正序榜中的名次  redis 127.0.0.1:6379> zrank nanshen:yanzhi:bang xiaotao  (integer) 0  #在倒序榜中的名次  redis 127.0.0.1:6379> zrevrank nanshen:yanzhi:bang xiaotao  (integer) 4 |

4、修改成员的分数

|  |
| --- |
| redis 127.0.0.1:6379> zincrby nanshen:yanzhi:bang 300 xiaotao  "359"  redis 127.0.0.1:6379> zrevrank nanshen:yanzhi:bang xiaotao  (integer) 0 |

## 06/ Redis应用案例（1）

案例：

Lol盒子英雄数据排行榜：

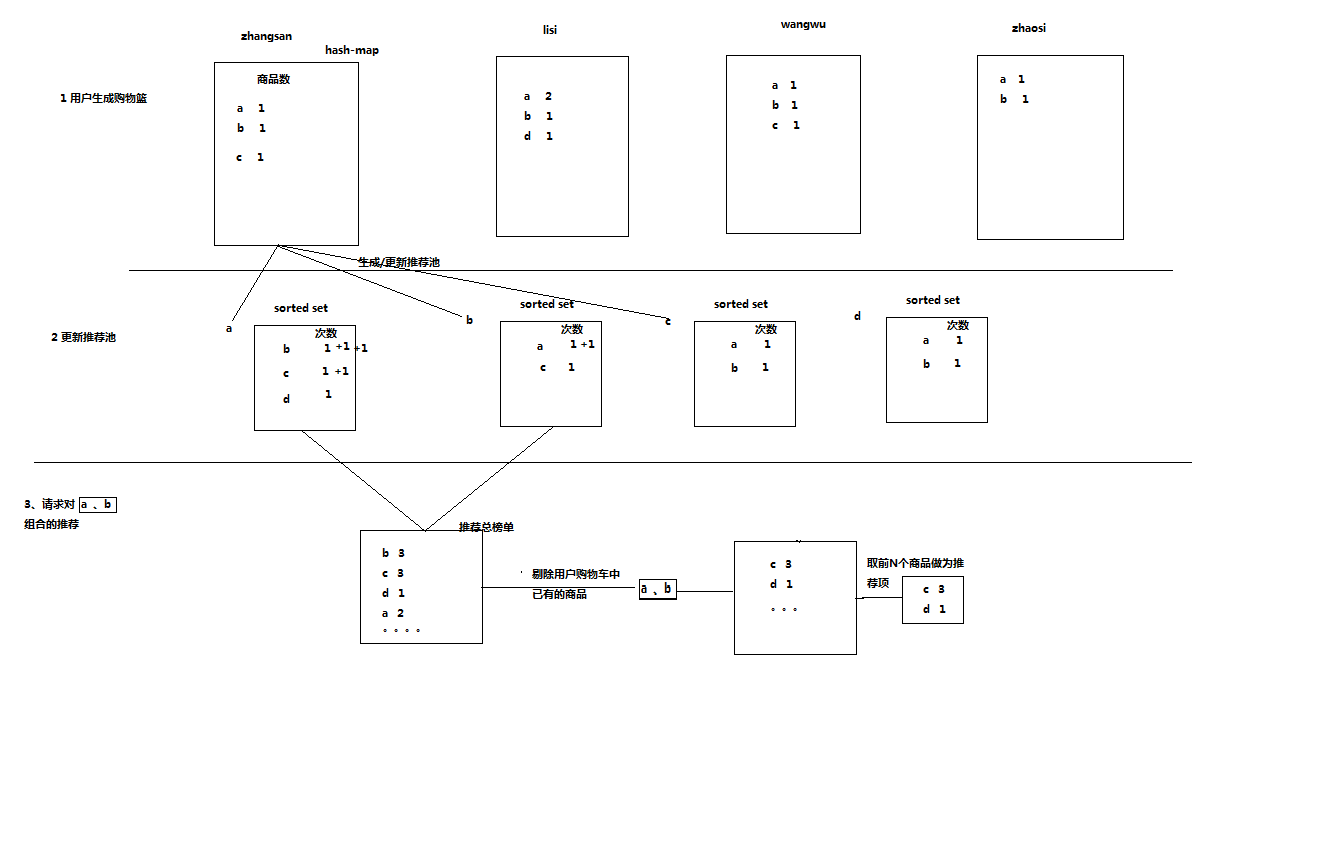
1. 在redis中需要一个榜单所对应的sortedset数据
2. 玩家每选择一个英雄打一场游戏，就对sortedset数据的相应的英雄分数+1
3. Lol盒子上查看榜单时，就调用zrange来看榜单中的排序结果

## 07/ Redis应用案例（2）

实现步骤：

1. 每来一个用户创建购物车，就对购物车中的每一件商品在redis中插入一条以商品名称为key的sortedset，成员为同时出现在购物车中的其他商品，分数为1
2. 每新产生一个购物车，就对车中的商品更新redis中的sortedset，将同时出现的商品的分数增1
3. 推荐时，用户放入一件商品到购物车，则从这件商品对应的sortedset中查询分数最高的同现商品，推荐给该用户

设计思想如下，详见《代码》



# java高级特性增强

今天内容安排：

1、掌握多线程

2、掌握并发包下的队列

3、了解JMS

4、掌握JVM技术

5、掌握反射和动态代理

* **java多线程增强**

## java多线程基本知识

**1.1进程介绍**

不管是我们开发的应用程序，还是我们运行的其他的应用程序，都需要先把程序安装在本地的硬盘上。然后找到这个程序的启动文件，启动程序的时候，其实是电脑把当前的这个程序加载到内存中，在内存中需要给当前的程序分配一段独立的运行空间。这片空间就专门负责当前这个程序的运行。

不同的应用程序运行的过程中都需要在内存中分配自己独立的运行空间，彼此之间不会相互的影响。我们把每个独立应用程序在内存的独立空间称为当前应用程序运行的一个进程。

**进程：它是内存中的一段独立的空间，可以负责当前应用程序的运行。当前这个进程负责调度当前程序中的所有运行细节。**

### 1.2 线程介绍

启动的QQ聊天软件，需要和多个人进行聊天。这时多个人之间是不能相互影响，但是它们都位于当前QQ这个软件运行时所分配的内容的独立空间中。

在一个进程中，每个独立的功能都需要独立的去运行，这时又需要把当前这个进程划分成多个运行区域，每个独立的小区域（小单元）称为一个线程。

**线程：它是位于进程中，负责当前进程中的某个具备独立运行资格的空间。**

进程是负责整个程序的运行，而线程是程序中具体的某个独立功能的运行。**一个进程中至少应该有一个线程。**

**1.3多线程介绍**

现在的操作系统基本都是多用户，多任务的操作系统。每个任务就是一个进程。而在这个进程中就会有线程。

真正可以完成程序运行和功能的实现靠的是进程中的线程。

多线程：在一个进程中，我们同时开启多个线程，让多个线程同时去完成某些任务（功能）。

多线程的目的：提高程序的运行效率。

**1.4 多线程运行的原理**

cpu在线程中做时间片的切换。

其实真正电脑中的程序的运行不是同时在运行的。CPU负责程序的运行，而CPU在运行程序的过程中某个时刻点上，它其实只能运行一个程序。而不是多个程序。而CPU它可以在多个程序之间进行高速的切换。而切换频率和速度太快，导致人的肉看看不到。

每个程序就是进程， 而每个进程中会有多个线程，而CPU是在这些线程之间进行切换。

了解了CPU对一个任务的执行过程，我们就必须知道，多线程可以提高程序的运行效率，但不能无限制的开线程。

**1.5实现线程的两种方式**

1、继承Thread的原理

见代码MyThreadWithExtends

2、声明实现 Runnable 接口的类

见代码MyThreadWithImpliment

3、还可以实现Callable接口

## java同步关键词解释

**2.1 synchronized**

synchronized是用来实现**线程同步**的！！！

加同步格式：

synchronized( 需要一个任意的对象（锁） ){

代码块中放操作共享数据的代码。

}

见代码MySynchronized

* synchronized的缺陷

synchronized是java中的一个关键字，也就是说是Java语言内置的特性。

如果一个代码块被synchronized修饰了，当一个线程获取了对应的锁，并执行该代码块时，其他线程便只能一直等待，等待获取锁的线程释放锁，而这里获取锁的线程释放锁只会有两种情况：

　　1）获取锁的线程执行完了该代码块，然后线程释放对锁的占有；

2）线程执行发生异常，此时JVM会让线程自动释放锁。

例子1：

　　如果这个获取锁的线程由于要等待IO或者其他原因（比如调用sleep方法）被阻塞了，但是又没有释放锁，其他线程便只能干巴巴地等待，试想一下，这多么影响程序执行效率。

　　因此就需要有一种机制可以不让等待的线程一直无期限地等待下去（比如只等待一定的时间或者能够响应中断），通过Lock就可以办到。

例子2：

当有多个线程读写文件时，读操作和写操作会发生冲突现象，写操作和写操作会发生冲突现象，但是读操作和读操作不会发生冲突现象。

　　但是采用synchronized关键字来实现同步的话，就会导致一个问题：

如果多个线程都只是进行读操作，当一个线程在进行读操作时，其他线程只能等待无法进行读操作。

　　因此就需要一种机制来使得多个线程都只是进行读操作时，线程之间不会发生冲突，通过Lock就可以办到。

　　另外，通过Lock可以知道线程有没有成功获取到锁。这个是synchronized无法办到的。

　　总的来说，也就是说Lock提供了比synchronized更多的功能。

### 2.2 lock

* lock和synchronized的区别

　　1）Lock不是Java语言内置的，synchronized是Java语言的关键字，因此是内置特性。Lock是一个类，通过这个类可以实现同步访问；

　　2）Lock和synchronized有一点非常大的不同，采用**synchronized不需要用户去手动释放锁**，当synchronized方法或者synchronized代码块执行完之后，系统会自动让线程释放对锁的占用；而**Lock则必须要用户去手动释放锁**，如果没有主动释放锁，就有可能导致出现死锁现象。

* java.util.concurrent.locks包下常用的类
  + - * + Lock

　　首先要说明的就是Lock，通过查看Lock的源码可知，Lock是一个接口：

|  |
| --- |
| public interface Lock {  void lock();  void lockInterruptibly() throws InterruptedException;  boolean tryLock();  boolean tryLock(long time, TimeUnit unit) throws InterruptedException;  void unlock();  } |

Lock接口中每个方法的使用：

lock()、tryLock()、tryLock(long time, TimeUnit unit)、lockInterruptibly()是用来获取锁的。 unLock()方法是用来释放锁的。

**四个获取锁方法的区别：**

　　lock()方法是平常使用得最多的一个方法，就是用来获取锁。如果锁已被其他线程获取，则进行等待。

由于在前面讲到如果采用Lock，必须主动去释放锁，并且在发生异常时，不会自动释放锁。因此一般来说，使用Lock必须在try{}catch{}块中进行，并且将释放锁的操作放在finally块中进行，以保证锁一定被被释放，防止死锁的发生。

tryLock()方法是有返回值的，它表示用来尝试获取锁，如果获取成功，则返回true，如果获取失败（即锁已被其他线程获取），则返回false，也就说这个方法**无论如何都会立即返回。**在拿不到锁时不会一直在那等待。

　　tryLock(long time, TimeUnit unit)方法和tryLock()方法是类似的，只不过区别在于这个方法在拿不到锁时会等待一定的时间，在时间期限之内如果还拿不到锁，就返回false。如果如果一开始拿到锁或者在等待期间内拿到了锁，则返回true。

　　lockInterruptibly()方法比较特殊，当通过这个方法去获取锁时，如果线程正在等待获取锁，则这个线程能够响应中断，即中断线程的等待状态。也就使说，当两个线程同时通过lock.lockInterruptibly()想获取某个锁时，假若此时线程A获取到了锁，而线程B只有在等待，那么对线程B调用threadB.interrupt()方法能够中断线程B的等待过程。

　　注意，当一个线程获取了锁之后，是不会被interrupt()方法中断的。

　　因此当通过lockInterruptibly()方法获取某个锁时，如果不能获取到，只有进行等待的情况下，是可以响应中断的。

　　而用synchronized修饰的话，当一个线程处于等待某个锁的状态，是无法被中断的，只有一直等待下去。

* + ReentrantLock

直接使用lock接口的话，我们需要实现很多方法，不太方便，ReentrantLock是唯一实现了Lock接口的类，并且ReentrantLock提供了更多的方法，ReentrantLock，意思是“可重入锁”。

以下是ReentrantLock的使用案例：

　　例子1，lock()的正确使用方法

见代码MyLockTest

例子2，tryLock()的使用方法

见代码MyTryLock

例子3，lockInterruptibly()响应中断的使用方法：

见代码MyInterruptibly

* ReadWriteLock (可以区别对待读、写的操作)

　　ReadWriteLock也是一个接口，在它里面只定义了两个方法：

|  |
| --- |
| public interface ReadWriteLock {  /\*\*  \* Returns the lock used for reading.  \*  \* @return the lock used for reading.  \*/  Lock readLock();    /\*\*  \* Returns the lock used for writing.  \*  \* @return the lock used for writing.  \*/  Lock writeLock();  } |

　　一个用来获取读锁，一个用来获取写锁。**也就是说将文件的读写操作分开，分成2个锁来分配给线程，从而使得多个线程可以同时进行读操作。**下面的ReentrantReadWriteLock实现了ReadWriteLock接口。

* ReentrantReadWriteLock

　　ReentrantReadWriteLock里面提供了很多丰富的方法，不过最主要的有两个方法：readLock()和writeLock()用来获取读锁和写锁。

下面通过几个例子来看一下ReentrantReadWriteLock具体用法。

例子1：　　假如有多个线程要同时进行读操作的话，先看一下synchronized达到的效果

见代码MySynchronizedReadWrite

例子2：改成用读写锁的话：

见代码MyReentrantReadWriteLock

注意：

　　不过要注意的是，如果有一个线程已经占用了读锁，则此时其他线程如果要申请写锁，则申请写锁的线程会一直等待释放读锁。

如果有一个线程已经占用了写锁，则此时其他线程如果申请写锁或者读锁，则申请的线程会一直等待释放写锁。

* Lock和synchronized的选择

　　1）Lock是一个接口，而synchronized是Java中的关键字，synchronized是内置的语言实现；

　　2）synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁；

　　3）Lock可以让等待锁的线程响应中断，而synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断；

　　4）通过Lock可以知道有没有成功获取锁，而synchronized却无法办到。

　　5）Lock可以提高多个线程进行读操作的效率。

　　在性能上来说，如果竞争资源不激烈，两者的性能是差不多的，而当竞争资源非常激烈时（即有大量线程同时竞争），此时Lock的性能要远远优于synchronized。所以说，在具体使用时要根据适当情况选择。

## 3.java并发包

### 3.1 java并发包介绍

JDK5.0 以后的版本都引入了高级并发特性，大多数的特性在java.util.concurrent 包中，是专门用于多线程发编程的，充分利用了现代多处理器和多核心系统的功能以编写大规模并发应用程序。主要包含原子量、并发集合、同步器、可重入锁，并对线程池的构造提供

了强力的支持。

**线程池**

* 线程池的5中创建方式：

1. Single Thread Executor : 只有一个线程的线程池，因此所有提交的任务是顺序执行，

代码： Executors.newSingleThreadExecutor()

1. Cached Thread Pool : 线程池里有很多线程需要同时执行，老的可用线程将被新的任务触发重新执行，如果线程超过60秒内没执行，那么将被终止并从池中删除，

代码：Executors.newCachedThreadPool()

1. Fixed Thread Pool : 拥有固定线程数的线程池，如果没有任务执行，那么线程会一直等待，

代码： Executors.newFixedThreadPool(4)

在构造函数中的参数4是线程池的大小，你可以随意设置，也可以和cpu的核数量保持一致，获取cpu的数量int cpuNums = Runtime.getRuntime().availableProcessors();

1. Scheduled Thread Pool : 用来调度即将执行的任务的线程池，

代码：Executors.newScheduledThreadPool()

1. Single Thread Scheduled Pool : 只有一个线程，用来调度执行将来的任务，代码：Executors.newSingleThreadScheduledExecutor()

* 线程池的使用

所谓给线程池提交任务，就是：

1、你将任务（业务处理逻辑）写到一个runnable或者callable的执行方法<run() | call()>

2、将这个runnable对象提交给线程池即可

提交 Runnable ，任务完成后 Future 对象返回 null

见代码：ThreadPoolWithRunable

提交 Callable，该方法返回一个 Future 实例表示任务的状态

见代码：ThreadPoolWithcallable

* 1. **java并发包消息队列及在开源软件中的应用**

消息队列常用于有生产者和消费者两类角色的多线程同步场景

BlockingQueue也是java.util.concurrent下的主要用来控制线程同步的工具。

主要的方法是：put、take一对阻塞存取；add、poll一对非阻塞存取。

插入:

1)add(anObject):把anObject加到BlockingQueue里,即如果BlockingQueue可以容纳,则返回true,否则抛出异常

2)offer(anObject):表示如果可能的话,将anObject加到BlockingQueue里,即如果BlockingQueue可以容纳,则返回true,否则返回false.

3)put(anObject):把anObject加到BlockingQueue里,如果BlockQueue没有空间,则调用此方法的线程被阻塞直到BlockingQueue里面有空间再继续.

读取：

4)poll(time):取走BlockingQueue里排在首位的对象,若不能立即取出,则可以等time参数规定的时间,取不到时返回null

5)take():取走BlockingQueue里排在首位的对象,若BlockingQueue为空,阻断进入等待状态直到Blocking有新的对象被加入为止

其他

int remainingCapacity();返回队列剩余的容量，在队列插入和获取的时候，不要瞎搞，数 据可能不准

boolean remove(Object o); 从队列移除元素，如果存在，即移除一个或者更多，队列改 变了返回true

public boolean contains(Object o); 查看队列是否存在这个元素，存在返回true

int drainTo(Collection<? super E> c); 传入的集合中的元素，如果在队列中存在，那么将 队列中的元素移动到集合中

int drainTo(Collection<? super E> c, int maxElements); 和上面方法的区别在于，制定了移 动的数量

BlockingQueue有四个具体的实现类,常用的两种实现类为：

1、ArrayBlockingQueue：一个由**数组支持的有界阻塞队列**，规定大小的BlockingQueue,其构造函数必须带一个int参数来指明其大小.其所含的对象是以FIFO(先入先出)顺序排序的。

2、LinkedBlockingQueue：大小不定的BlockingQueue,若其构造函数带一个规定大小的参数,生成的BlockingQueue有大小限制,若不带大小参数,所生成的BlockingQueue的大小由Integer.MAX\_VALUE来决定.其所含的对象是以FIFO(先入先出)顺序排序的。

LinkedBlockingQueue **可以指定容量，也可以不指定**，不指定的话，默认最大是Integer.MAX\_VALUE,其中主要用到put和take方法，put方法在队列满的时候会阻塞直到有队列成员被消费，take方法在队列空的时候会阻塞，直到有队列成员被放进来。

LinkedBlockingQueue和ArrayBlockingQueue区别：

LinkedBlockingQueue和ArrayBlockingQueue比较起来,它们背后所用的数据结构不一样,导致LinkedBlockingQueue的数据吞吐量要大于ArrayBlockingQueue,但在线程数量很大时其性能的可预见性低于ArrayBlockingQueue.

生产者消费者的示例代码：

见代码 TestBlockingQueue TestBlockingQueueConsumer TestBlockingQueueProducer

## 4.java JMS技术

**4.1 什么是JMS**

JMS即Java消息服务（Java Message Service）应用程序**接口**是一个Java平台中关于面向消息中间件（MOM）的API，**用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信**。Java消息服务是一个与具体平台无关的API，绝大多数MOM提供商都对JMS提供支持。

**JMS是一种与厂商无关的 API**，用来访问消息收发系统消息。它类似于JDBC(Java Database Connectivity)：这里，JDBC 是可以用来访问许多不同关系数据库的 API，而 JMS 则提供同样与厂商无关的访问方法，以访问消息收发服务。许多厂商都支持 JMS，包括 IBM 的 MQSeries、BEA的 Weblogic JMS service和 Progress 的 SonicMQ，这只是几个例子。 JMS 使您能够通过消息收发服务（有时称为消息中介程序或路由器）从一个 JMS 客户机向另一个 JMS客户机发送消息。**消息是 JMS 中的一种类型对象，由两部分组成：报头和消息主体**。报头由路由信息以及有关该消息的元数据组成。消息主体则携带着应用程序的数据或有效负载。根据有效负载的类型来划分，可以将消息分为几种类型，它们分别携带：简单文本(TextMessage)、可序列化的对象 (ObjectMessage)、属性集合 (MapMessage)、字节流 (BytesMessage)、原始值流 (StreamMessage)，还有无有效负载的消息 (Message)。

**5. JMS规范**

**5.1 专业技术规范**

JMS（Java Messaging Service）**是Java平台上有关面向消息中间件(MOM)的技术规范**，它便于消息系统中的Java应用程序进行消息交换,并且通过提供标准的产生、发送、接收消息的接口简化企业应用的开发，翻译为Java消息服务。

**5.2 体系架构**

JMS由以下元素组成。

**JMS提供者：**连接面向消息中间件的，**JMS接口的一个实现**。提供者可以是Java平台的JMS实现，也可以是非Java平台的面向消息中间件的适配器。

**JMS客户**：生产或消费基于消息的Java的应用程序或对象。

**JMS生产者**：创建并发送消息的JMS客户。

**JMS消费者**：接收消息的JMS客户。

**JMS消息**：包括可以在JMS客户之间传递的数据的对象

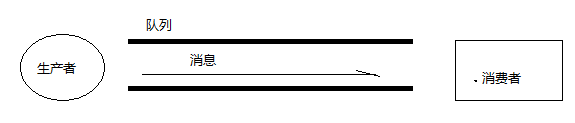
**JMS队列**：一个容纳那些被发送的等待阅读的消息的区域。一旦一个消息被阅读，该消息将被从队列中移走。

**JMS主题**：一种支持发送消息给多个订阅者的机制。

**5.3 Java消息服务应用程序结构支持两种模型**

1. 点对点或**队列模型**

在点对点或队列模型下，一个生产者向一个特定的队列发布消息，一个消费者从该队列中读取消息。这里，生产者知道消费者的队列，并直接将消息发送到消费者的队列。



这种模式被概括为：

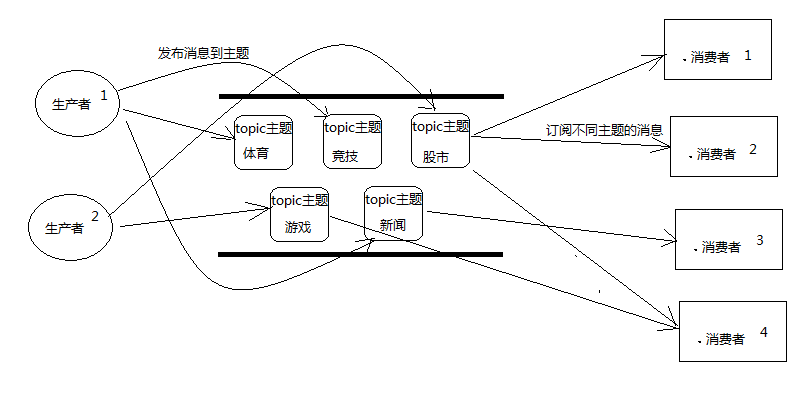
只有一个消费者将获得消息

生产者不需要在接收者消费该消息期间处于运行状态，接收者也同样不需要在消息发送时处于运行状态。

每一个成功处理的消息都由接收者签收

2、**发布者/订阅者模型**

发布者/订阅者模型支持向一个特定的消息主题发布消息。0或多个订阅者可能对接收来自特定消息主题的消息感兴趣。在这种模型下，发布者和订阅者彼此不知道对方。这种模式好比是匿名公告板。

这种模式被概括为：

多个消费者可以获得消息

在发布者和订阅者之间存在时间依赖性。发布者需要建立一个订阅（subscription），以便客户能够订阅。订阅者必须保持持续的活动状态以接收消息，除非订阅者建立了持久的订阅。在那种情况下，在订阅者未连接时发布的消息将在订阅者重新连接时重新发布。

**5.4 常用的JMS实现**

要使用Java消息服务，你必须要有一个JMS提供者，管理会话和队列。既有开源的提供者也有专有的提供者。

* 开源的提供者包括：

**Apache ActiveMQ**

JBoss 社区所研发的 HornetQ

Joram

Coridan的MantaRay

The OpenJMS Group的OpenJMS

* 专有的提供者包括：

BEA的BEA WebLogic Server JMS

TIBCO Software的EMS

GigaSpaces Technologies的GigaSpaces

Softwired 2006的iBus

IONA Technologies的IONA JMS

SeeBeyond的IQManager（2005年8月被Sun Microsystems并购）

webMethods的JMS+ -

my-channels的Nirvana

Sonic Software的SonicMQ

SwiftMQ的SwiftMQ

IBM的WebSphere MQ

**5.5 代码演示**

*注：新版直接解压即可运行*

1.下载ActiveMQ

去官方网站下载：http://activemq.apache.org/

2.运行ActiveMQ

解压缩apache-activemq-5.5.1-bin.zip，

~~修改配置文件activeMQ.xml，将0.0.0.0修改为localhost~~

|  |
| --- |
| <transportConnectors>  <transportConnector name="openwire" uri="tcp://localhost:61616"/>  <transportConnector name="ssl" uri="ssl://localhost:61617"/>  <transportConnector name="stomp" uri="stomp://localhost:61613"/>  <transportConnector uri="http://localhost:8081"/>  <transportConnector uri="udp://localhost:61618"/> |

然后双击apache-activemq-5.5.1\bin\win64\activemq.bat运行ActiveMQ程序。

启动ActiveMQ以后，登陆：http://localhost:8161/admin/，创建一个Queue，命名为FirstQueue。

3.运行代码

package cn.itcast\_03\_mq.queue

package cn.itcast\_03\_mq.topic

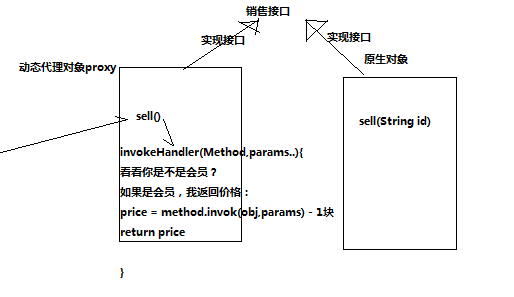
## 6.java反射、动态代理、sockect通信

**6.1反射**

通过反射的方式可以获取class对象中的属性、方法、构造函数等，一下是实例：

|  |
| --- |
| package cn.java.reflect;  import java.lang.reflect.Constructor;  import java.lang.reflect.Field;  import java.lang.reflect.Method;  import java.util.ArrayList;  import java.util.List;  import org.junit.Before;  import org.junit.Test;  public class MyReflect {  public String className = null;  @SuppressWarnings("rawtypes")  public Class personClass = null;  /\*\*  \* 反射Person类  \* @throws Exception  \*/  @Before  public void init() throws Exception {  className = "cn.java.reflect.Person";  personClass = Class.forName(className);  }  /\*\*  \*获取某个class文件对象  \*/  @Test  public void getClassName() throws Exception {  System.out.println(personClass);  }  /\*\*  \*获取某个class文件对象的另一种方式  \*/  @Test  public void getClassName2() throws Exception {  System.out.println(Person.class);  }  /\*\*  \*创建一个class文件表示的真实对象，底层会调用空参数的构造方法  \*/  @Test  public void getNewInstance() throws Exception {  System.out.println(personClass.newInstance());  }  /\*\*  \*获取非私有的构造函数  \*/  @SuppressWarnings({ "rawtypes", "unchecked" })  @Test  public void getPublicConstructor() throws Exception {  Constructor constructor = personClass.getConstructor(Long.class,String.class);  Person person = (Person)constructor.newInstance(100L,"zhangsan");  System.out.println(person.getId());  System.out.println(person.getName());  }  /\*\*  \*获得私有的构造函数  \*/  @SuppressWarnings({ "rawtypes", "unchecked" })  @Test  public void getPrivateConstructor() throws Exception {  Constructor con = personClass.getDeclaredConstructor(String.class);  con.setAccessible(true);//强制取消Java的权限检测  Person person2 = (Person)con.newInstance("zhangsan");  System.out.println(person2.getName());  }  /\*\*  \*获取非私有的成员变量  \*/  @SuppressWarnings({ "rawtypes", "unchecked" })  @Test  public void getNotPrivateField() throws Exception {  Constructor constructor = personClass.getConstructor(Long.class,String.class);  Object obj = constructor.newInstance(100L,"zhangsan");    Field field = personClass.getField("name");  field.set(obj, "lisi");  System.out.println(field.get(obj));  }  /\*\*  \*获取私有的成员变量  \*/  @SuppressWarnings({ "rawtypes", "unchecked" })  @Test  public void getPrivateField() throws Exception {  Constructor constructor = personClass.getConstructor(Long.class);  Object obj = constructor.newInstance(100L);    Field field2 = personClass.getDeclaredField("id");  field2.setAccessible(true);//强制取消Java的权限检测  field2.set(obj,10000L);  System.out.println(field2.get(obj));  }  /\*\*  \*获取非私有的成员函数  \*/  @SuppressWarnings({ "unchecked" })  @Test  public void getNotPrivateMethod() throws Exception {  System.out.println(personClass.getMethod("toString"));    Object obj = personClass.newInstance();//获取空参的构造函数  Object object = personClass.getMethod("toString").invoke(obj);  System.out.println(object);  }  /\*\*  \*获取私有的成员函数  \*/  @SuppressWarnings("unchecked")  @Test  public void getPrivateMethod() throws Exception {  Object obj = personClass.newInstance();//获取空参的构造函数  Method method = personClass.getDeclaredMethod("getSomeThing");  method.setAccessible(true);  Object value = method.invoke(obj);  System.out.println(value);  }  /\*\*  \*  \*/  @Test  public void otherMethod() throws Exception {  //当前加载这个class文件的那个类加载器对象  System.out.println(personClass.getClassLoader());  //获取某个类实现的所有接口  Class[] interfaces = personClass.getInterfaces();  for (Class class1 : interfaces) {  System.out.println(class1);  }  //反射当前这个类的直接父类  System.out.println(personClass.getGenericSuperclass());  /\*\*  \* getResourceAsStream这个方法可以获取到一个输入流，这个输入流会关联到name所表示的那个文件上。  \*/  //path 不以’/'开头时默认是从此类所在的包下取资源，以’/'开头则是从ClassPath根下获取。其只是通过path构造一个绝对路径，最终还是由ClassLoader获取资源。  System.out.println(personClass.getResourceAsStream("/log4j.properties"));  //默认则是从ClassPath根下获取，path不能以’/'开头，最终是由ClassLoader获取资源。  System.out.println(personClass.getResourceAsStream("/log4j.properties"));    //判断当前的Class对象表示是否是数组  System.out.println(personClass.isArray());  System.out.println(new String[3].getClass().isArray());    //判断当前的Class对象表示是否是枚举类  System.out.println(personClass.isEnum());  System.out.println(Class.forName("cn.java.reflect.City").isEnum());    //判断当前的Class对象表示是否是接口  System.out.println(personClass.isInterface());  System.out.println(Class.forName("cn.java.reflect.TestInterface").isInterface());      }  } |

**6.2 动态代理**



在之前的代码调用阶段，我们用action调用service的方法实现业务即可。

由于之前在service中实现的业务可能不能够满足当先客户的要求，需要我们重新修改service中的方法，但是service的方法不只在我们这个模块使用，在其他模块也在调用，其他模块调用的时候，现有的service方法已经能够满足业务需求，所以我们不能只为了我们的业务而修改service，导致其他模块授影响。

那怎么办呢？

可以通过动态代理的方式，扩展我们的service中的方法实现，使得在原油的方法中增加更多的业务，而不是实际修改service中的方法，这种实现技术就叫做动态代理。

动态代理：在不修改原业务的基础上，基于原业务方法，进行重新的扩展，实现新的业务。

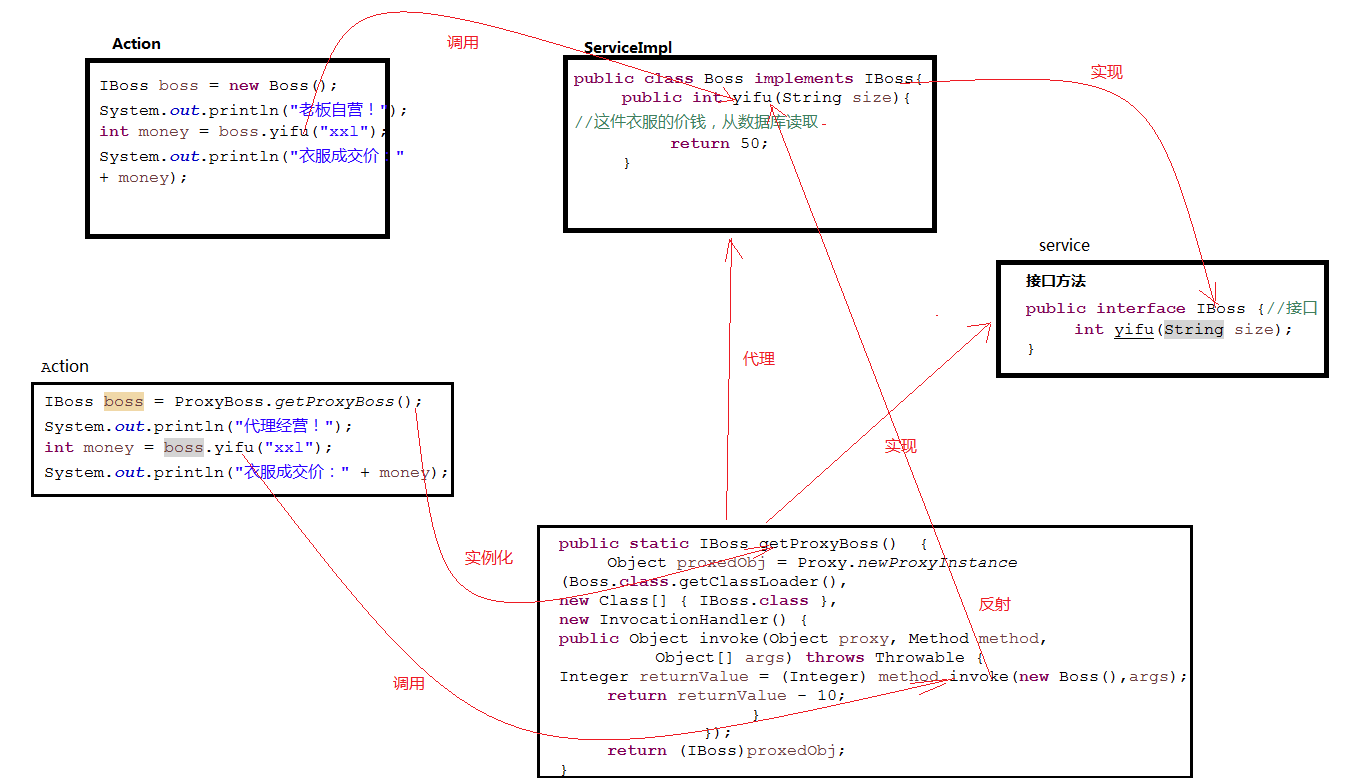
例如下面的例子：

1. 旧业务

买家调用action，购买衣服，衣服在数据库的标价为50元，购买流程就是简单的调用。

1. 新业务

在原先的价格上可以使用优惠券，但是这个功能在以前没有实现过，我们通过代理类，代理了原先的接口方法，在这个方法的基础上，修改了返回值。



代理实现流程：

1. 书写代理类和代理方法，在代理方法中实现代理Proxy.newProxyInstance
2. 代理中需要的参数分别为：被代理的类的类加载器soneObjectclass.getClassLoader()，被代理类的所有实现接口new Class[] { Interface.class }，句柄方法new InvocationHandler()
3. 在句柄方法中复写invoke方法，invoke方法的输入有3个参数Object proxy（代理类对象）, Method method（被代理类的方法）,Object[] args（被代理类方法的传入参数），在这个方法中，我们可以定制化的开发新的业务。
4. 获取代理类，强转成被代理的接口
5. 最后，我们可以像没被代理一样，调用接口的认可方法，方法被调用后，方法名和参数列表将被传入代理类的invoke方法中，进行新业务的逻辑流程。

原业务接口IBoss

|  |
| --- |
| public interface IBoss {//接口  int yifu(String size);  } |

原业务实现类

|  |
| --- |
| public class Boss implements IBoss{  public int yifu(String size){  System.err.println("天猫小强旗舰店，老板给客户发快递----衣服型号："+size);  //这件衣服的价钱，从数据库读取  return 50;  }  public void kuzi(){  System.err.println("天猫小强旗舰店，老板给客户发快递----裤子");  }  } |

原业务调用

|  |
| --- |
| public class SaleAction {  @Test  public void saleByBossSelf() throws Exception {  IBoss boss = new Boss();  System.out.println("老板自营！");  int money = boss.yifu("xxl");  System.out.println("衣服成交价：" + money);  }  } |

代理类

|  |
| --- |
| public static IBoss getProxyBoss(final int discountCoupon) throws Exception {  Object proxedObj = Proxy.newProxyInstance(Boss.class.getClassLoader(),  new Class[] { IBoss.class }, new InvocationHandler() {  public Object invoke(Object proxy, Method method,  Object[] args) throws Throwable {  Integer returnValue = (Integer) method.invoke(new Boss(),  args);// 调用原始对象以后返回的值  return returnValue - discountCoupon;  }  });  return (IBoss)proxedObj;  }  } |

新业务调用

|  |
| --- |
| public class ProxySaleAction {  @Test  public void saleByProxy() throws Exception {  IBoss boss = ProxyBoss.getProxyBoss(20);// 将代理的方法实例化成接口  System.out.println("代理经营！");  int money = boss.yifu("xxl");// 调用接口的方法，实际上调用方式没有变  System.out.println("衣服成交价：" + money);  }  } |

**6.3 socket编程**

注意：此处讲的socket编程模型是jdk1.4之前的机制BIO (同步阻塞IO)

1.4之后引入了NIO （同步非阻塞IO）

1.7之后引入了AIO （NIO 2） （异步非阻塞IO）