# 商城开发框架

## 一.商城开发架构

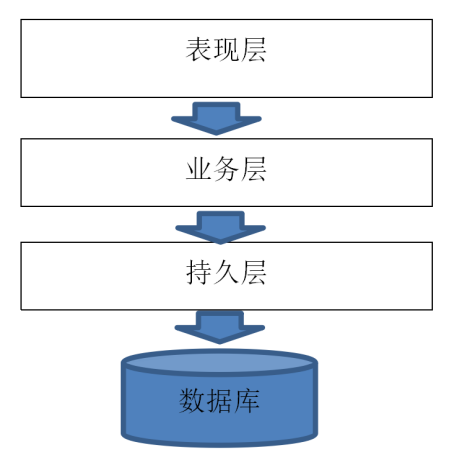
### 1.1 B2C 电商系统架构



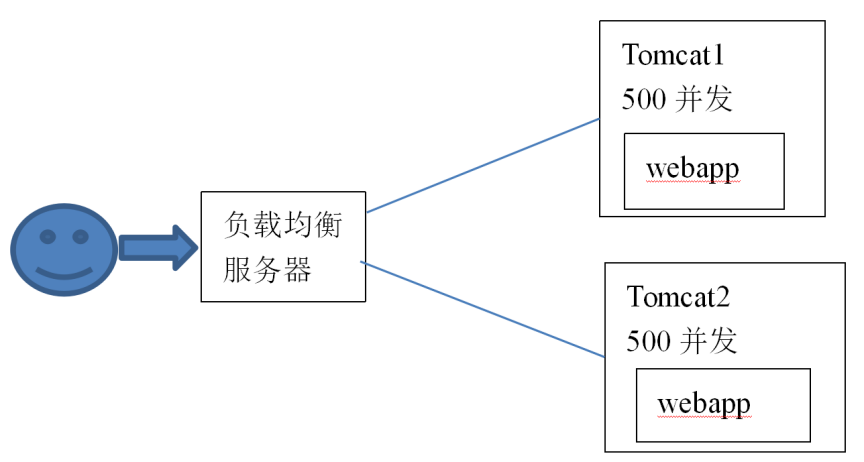
* 后台管理系统：管理商品、订单、类目、商品规格属性、用户管理以及内容发布等功能。
* 前台系统：用户可以在前台系统中进行注册、登录、浏览商品、首页、下单等操作。
* 会员系统：用户可以在该系统中查询已下的订单、收藏的商品、我的优惠券、团购等信息。
* 订单系统：提供下单、查询订单、修改订单状态、定时处理订单。
* 搜索系统：提供商品的搜索功能。
* 单点登录系统：为多个系统之间提供用户登录凭证以及查询登录用户的信息。

### 1.2 架构演进

**传统开发架构和部署架构**



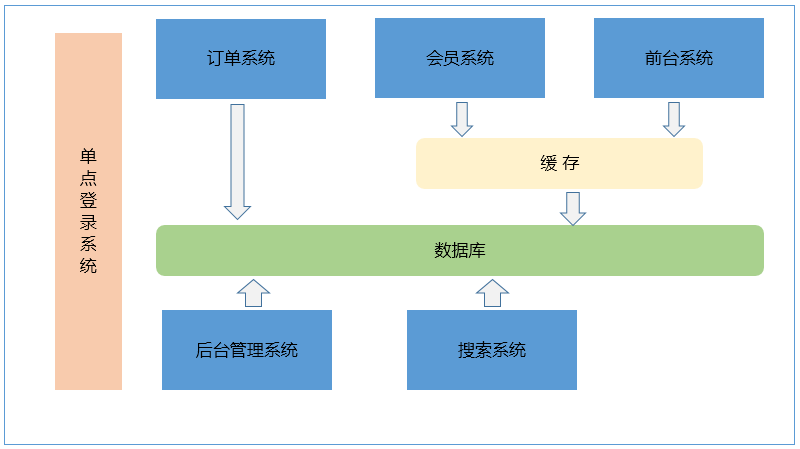
1000并发



需要2台以上服务器做tomcat集群。当tomcat集群中节点数量增加，服务能力先增加后下降。所以集群中节点数量不能太多，一般也就5个左右。

### 1.3 分布式系统架构

需要按照功能点把系统拆分，拆分成独立的功能。单独为某一个节点添加服务器。需要系统之间配合才能完成整个业务逻辑。叫做分布式。



分布式架构: 多个子系统相互协作才能完成业务流程,系统之间需要进行通信!

集群: 同一个工程部署到多个服务器上!

优点:

* 按照功能进行模块拆分,使用接口进行通信,降低了模块之前的耦合!
* 把项目拆分成若干个子项目,便于不同的项目团队维护开发,提供项目的高效,快速
* 便于灵活的进行分布式部署!

缺点:

* 系统之间需要使用远程通信,降低项目功能的性能!

### 1.4 SOA 的架构

SOA: service oriented Architecture ,面向服务的架构,即 把工程拆分成服务层,表现层两个工程! 服务层包含业务逻辑,只需对外提供服务即可.表现层 只需处理和页面的交互,业务逻辑都是调用服务层的服务来实现!



### 1.5 当前开发架构



## 二.Dubbo 中间件

### 2.1 系统间通信

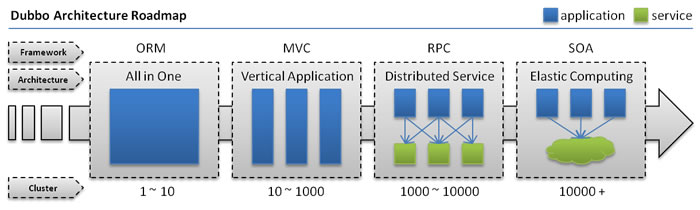
由于淘淘商城是基于soa的架构，表现层和服务层是不同的工程。所以要实现商品列表查询需要两个系统之间进行通信。

如何实现远程通信？

1. Webservice：效率不高基于soap协议。项目中不推荐使用。
2. 使用restful形式的服务：http+json。很多项目中应用。如果服务太多，服务之间调用关系混乱，需要治疗服务。
3. 使用dubbo。使用rpc协议进行远程调用，直接使用socket通信。传输效率高，并且可以统计出系统之间的调用关系、调用次数。

### 2.2 Dubbo 中间件

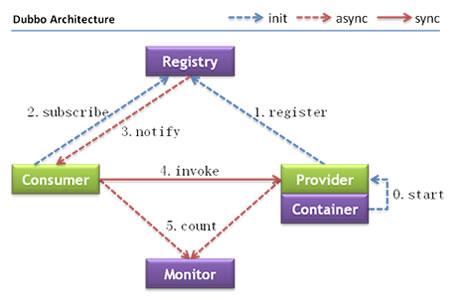
随着互联网的发展，网站应用的规模不断扩大，常规的垂直应用架构已无法应对，分布式服务架构以及流动计算架构势在必行，亟需一个治理系统确保架构有条不紊的演进。



* **单一应用架构**
  + 当网站流量很小时，只需一个应用，将所有功能都部署在一起，以减少部署节点和成本。
  + 此时，用于简化增删改查工作量的 **数据访问框架(ORM)** 是关键。
* **垂直应用架构**
  + 当访问量逐渐增大，单一应用增加机器带来的加速度越来越小，将应用拆成互不相干的几个应用，以提升效率。
  + 此时，用于加速前端页面开发的 **Web框架(MVC)** 是关键。
* **分布式服务架构**
  + 当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求。
  + 此时，用于提高业务复用及整合的 **分布式服务框架(RPC)** 是关键。
* **流动计算架构**
  + 当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。
  + 此时，用于提高机器利用率的 **资源调度和治理中心(SOA)** 是关键。

Dubbo就是资源调度和治理中心的管理工具。Dubbo超时:1秒一次,三次抛异常

### 2.3 Dubbo的架构



节点角色说明：

Provider: 暴露服务的服务提供方。

Consumer: 调用远程服务的服务消费方。

Registry: 服务注册与发现的注册中心。

Monitor: 统计服务的调用次调和调用时间的监控中心。

Container: 服务运行容器。

调用关系说明：

0. 服务容器负责启动，加载，运行服务提供者。

1. 服务提供者在启动时，向注册中心注册自己提供的服务。

2. 服务消费者在启动时，向注册中心订阅自己所需的服务。

3. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。

4. 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。

5. 服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

### 2.4 使用方法

Dubbo采用全Spring配置方式，透明化接入应用，对应用没有任何API侵入，只需用Spring加载Dubbo的配置即可，Dubbo基于Spring的Schema扩展进行加载。

**单一工程中spring的配置**

|  |
| --- |
| <bean id="xxxService" class="com.xxx.XxxServiceImpl" />  <bean id="xxxAction" class="com.xxx.XxxAction">  <property name="xxxService" ref="xxxService" />  </bean> |

**远程服务：**

在本地服务的基础上，只需做简单配置，即可完成远程化：

将上面的local.xml配置拆分成两份，将服务定义部分放在服务提供方remote-provider.xml，将服务引用部分放在服务消费方remote-consumer.xml。

并在提供方增加暴露服务配置<dubbo:service>，在消费方增加引用服务配置<dubbo:reference>。

发布服务：

|  |
| --- |
| <!-- 和本地服务一样实现远程服务 -->  <bean id="xxxService" class="com.xxx.XxxServiceImpl" />  <!-- 增加暴露远程服务配置 -->  <dubbo:service interface="com.xxx.XxxService" ref="xxxService" /> |

调用服务：

|  |
| --- |
| <!-- 增加引用远程服务配置 -->  <dubbo:reference id="xxxService" interface="com.xxx.XxxService" />  <!-- 和本地服务一样使用远程服务 -->  <bean id="xxxAction" class="com.xxx.XxxAction">  <property name="xxxService" ref="xxxService" />  </bean> |

注册中心负责服务地址的注册与查找，相当于目录服务，服务提供者和消费者只在启动时与注册中心交互，注册中心不转发请求，压力较小。使用dubbo-2.3.3以上版本，建议使用zookeeper注册中心。

Zookeeper是Apacahe Hadoop的子项目，是一个树型的目录服务，支持变更推送，适合作为Dubbo服务的注册中心，工业强度较高，可用于生产环境，并推荐使用

**Zookeeper的安装：**

第一步：安装jdk

第二步：解压缩zookeeper压缩包

第三步：将conf文件夹下zoo\_sample.cfg复制一份，改名为zoo.cfg

第四步：修改配置dataDir属性，指定一个真实目录

第五步：

启动zookeeper：bin/zkServer.sh start

关闭zookeeper：bin/zkServer.sh stop

查看zookeeper状态：bin/zkServer.sh status

注意要关闭linux的防火墙。

### 2.5 发布服务

1.工程中添加dubbo依赖的jar包。

|  |
| --- |
| <!-- dubbo相关 -->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>dubbo</artifactId>  <exclusions>  <exclusion>  <groupId>org.springframework</groupId>  <artifactId>spring</artifactId>  </exclusion>  <exclusion>  <groupId>org.jboss.netty</groupId>  <artifactId>netty</artifactId>  </exclusion>  </exclusions>  </dependency>  <dependency>  <groupId>org.apache.zookeeper</groupId>  <artifactId>zookeeper</artifactId>  </dependency>  <dependency>  <groupId>com.github.sgroschupf</groupId>  <artifactId>zkclient</artifactId>  </dependency> |

2、在spring的配置文件中添加dubbo的约束，然后使用dubbo:service发布服务。

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"* xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*  xmlns:dubbo=*"http://code.alibabatech.com/schema/dubbo"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.2.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.2.xsd*  *http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.2.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.2.xsd*  *http://code.alibabatech.com/schema/dubbo http://code.alibabatech.com/schema/dubbo/dubbo.xsd*  *http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-4.2.xsd"*>  <context:component-scan base-package=*"com.taotao.service"*></context:component-scan>  <!-- 使用dubbo发布服务 -->  <!-- 提供方应用信息，用于计算依赖关系 -->  <dubbo:application name=*"xx-manager"* />  <dubbo:registry protocol="zookeeper"  address="192.168.25.154:2181,192.168.25.154:2182,192.168.25.154:2183" />  <!-- 用dubbo协议在20880端口暴露服务 -->  <dubbo:protocol name=*"dubbo"* port=*"20880"* />  <!-- 声明需要暴露的服务接口 -->  <dubbo:service interface=*"com.xx.service.ItemService"* ref=*"itemServiceImpl"* />  </beans> |

### 2.6 引用服务

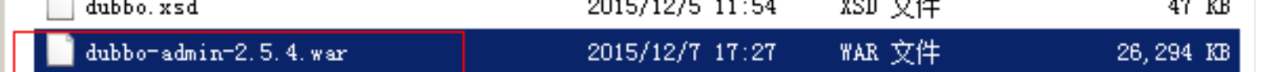
1. 工程中添加dubbo依赖的jar包

|  |
| --- |
| <!-- dubbo相关 -->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>dubbo</artifactId>  <exclusions>  <exclusion>  <groupId>org.springframework</groupId>  <artifactId>spring</artifactId>  </exclusion>  <exclusion>  <groupId>org.jboss.netty</groupId>  <artifactId>netty</artifactId>  </exclusion>  </exclusions>  </dependency>  <dependency>  <groupId>org.apache.zookeeper</groupId>  <artifactId>zookeeper</artifactId>  </dependency>  <dependency>  <groupId>com.github.sgroschupf</groupId>  <artifactId>zkclient</artifactId>  </dependency> |

1. 在springmvc的配置文件中添加服务的引用

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:dubbo=*"http://code.alibabatech.com/schema/dubbo"*  xmlns:mvc=*"http://www.springframework.org/schema/mvc"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.2.xsd*  *http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-4.2.xsd*  *http://code.alibabatech.com/schema/dubbo http://code.alibabatech.com/schema/dubbo/dubbo.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.2.xsd"*>  <context:component-scan base-package=*"com.taotao.controller"* />  <mvc:annotation-driven />  <bean  class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>  <property name=*"prefix"* value=*"/WEB-INF/jsp/"* />  <property name=*"suffix"* value=*".jsp"* />  </bean>    <!-- 引用dubbo服务 -->  <dubbo:application name=*"xx-manager-web"*/>  <dubbo:registry protocol="zookeeper" address="192.168.25.154:2181,192.168.25.154:2182,192.168.25.154:2183"/>  <dubbo:reference interface=*"com.xxx.service.ItemService"* id=*"itemService"* />    </beans> |

### 2.7 Dubbo监控中心



需要安装tomcat，然后部署监控中心即可。

1、部署监控中心：

[root@localhost ~]# cp dubbo-admin-2.5.4.war apache-tomcat-7.0.47/webapps/dubbo-admin.war

1. 启动tomcat
2. 访问http://192.168.25.167:8080/dubbo-admin/

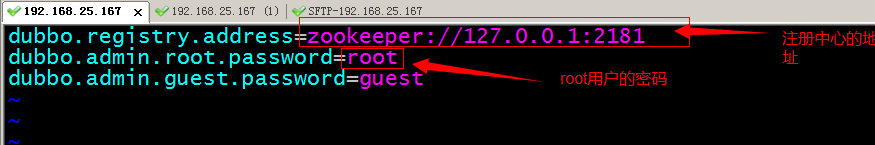
用户名：root

密码：root

如果监控中心和注册中心在同一台服务器上，可以不需要任何配置。

如果不在同一台服务器，需要修改配置文件：

/root/apache-tomcat-7.0.47/webapps/dubbo-admin/WEB-INF/dubbo.properties



## 三.Redis 中间件

### 3.1 Redis的安装

Redis是c语言开发的。

安装redis需要c语言的编译环境。如果没有gcc需要在线安装。yum install gcc-c++

安装步骤：

第一步：redis的源码包上传到linux系统。

第二步：解压缩redis。

第三步：编译。进入redis源码目录。make

第四步：安装。make install PREFIX=/usr/local/redis

PREFIX参数指定redis的安装目录。一般软件安装到/usr目录下

### 3.2连接redis

redis的启动：

前端启动：在redis的安装目录下直接启动redis-server

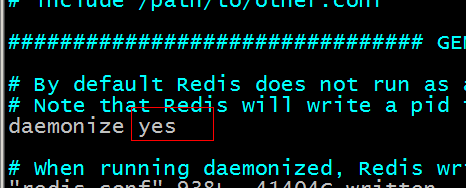
[root@localhost bin]# ./redis-server

后台启动：

把/root/redis-3.0.0/redis.conf复制到/usr/local/redis/bin目录下

[root@localhost redis-3.0.0]# cp redis.conf /usr/local/redis/bin/

修改配置文件：



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

查看redis进程：

[root@localhost bin]# ps aux|grep redis

root 5190 0.1 0.3 33936 1712 ? Ssl 18:23 0:00 ./redis-server \*:6379

root 5196 0.0 0.1 4356 728 pts/0 S+ 18:24 0:00 grep redis

[root@localhost bin]#

Redis-cli

[root@localhost bin]# ./redis-cli

默认连接localhost运行在6379端口的redis服务。

[root@localhost bin]# ./redis-cli -h 192.168.25.153 -p 6379

-h：连接的服务器的地址

-p：服务的端口号

关闭redis：[root@localhost bin]# ./redis-cli shutdown

### 3.3Redis五种数据类型

String：key-value（做缓存）

Redis中所有的数据都是字符串。命令不区分大小写，key是区分大小写的。Redis是单线程的。Redis中不适合保存内容大的数据。

get、set、

incr：加一（生成id）

Decr：减一

Hash：key-fields-values（做缓存）

相当于一个key对于一个map，map中还有key-value

使用hash对key进行归类。

Hset：向hash中添加内容

Hget：从hash中取内容

List：有顺序可重复

192.168.25.153:6379> lpush list1 a b c d

(integer) 4

192.168.25.153:6379> lrange list1 0 -1

1) "d"

2) "c"

3) "b"

4) "a"

192.168.25.153:6379> rpush list1 1 2 3 4

(integer) 8

192.168.25.153:6379> lrange list1 0 -1

1) "d"

2) "c"

3) "b"

4) "a"

5) "1"

6) "2"

7) "3"

8) "4"

192.168.25.153:6379>

192.168.25.153:6379> lpop list1

"d"

192.168.25.153:6379> lrange list1 0 -1

1) "c"

2) "b"

3) "a"

4) "1"

5) "2"

6) "3"

7) "4"

192.168.25.153:6379> rpop list1

"4"

192.168.25.153:6379> lrange list1 0 -1

1) "c"

2) "b"

3) "a"

4) "1"

5) "2"

6) "3"

192.168.25.153:6379>

Set：元素无顺序，不能重复

192.168.25.153:6379> sadd set1 a b c c c d

(integer) 4

192.168.25.153:6379> smembers set1

1) "b"

2) "c"

3) "d"

4) "a"

192.168.25.153:6379> srem set1 a

(integer) 1

192.168.25.153:6379> smembers set1

1) "b"

2) "c"

3) "d"

192.168.25.153:6379>

还有集合运算命令，自学。

SortedSet（zset）：有顺序，不能重复

192.168.25.153:6379> zadd zset1 2 a 5 b 1 c 6 d

(integer) 4

192.168.25.153:6379> zrange zset1 0 -1

1) "c"

2) "a"

3) "b"

4) "d"

192.168.25.153:6379> zrem zset1 a

(integer) 1

192.168.25.153:6379> zrange zset1 0 -1

1) "c"

2) "b"

3) "d"

192.168.25.153:6379> zrevrange zset1 0 -1

1) "d"

2) "b"

3) "c"

192.168.25.153:6379> zrange zset1 0 -1 withscores

1) "c"

2) "1"

3) "b"

4) "5"

5) "d"

6) "6"

192.168.25.153:6379> zrevrange zset1 0 -1 withscores

1) "d"

2) "6"

3) "b"

4) "5"

5) "c"

6) "1"

192.168.25.153:6379>

Key命令

设置key的过期时间。

Expire key second：设置key的过期时间

Ttl key：查看key的有效期

Persist key：清除key的过期时间。Key持久化。

192.168.25.153:6379> expire Hello 100

(integer) 1

192.168.25.153:6379> ttl Hello

(integer) 77

### 3.4Redis的持久化方案

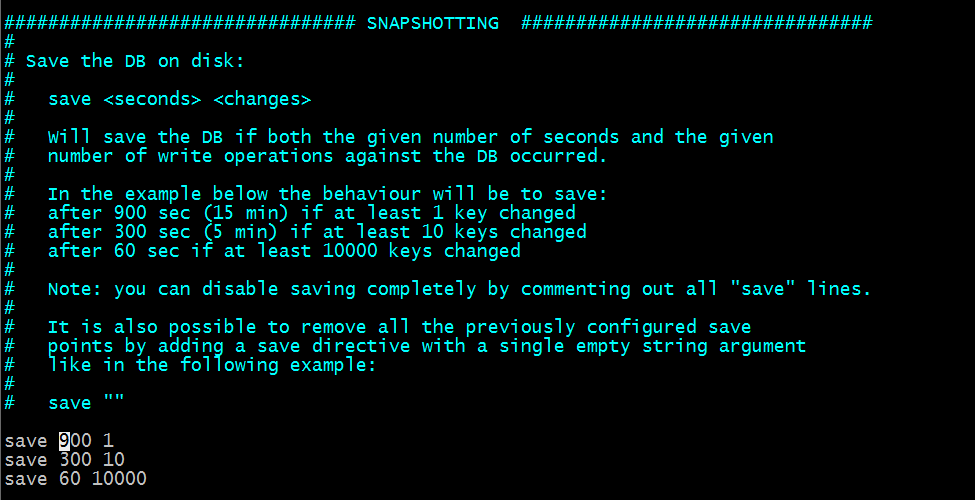
Redis的所有数据都是保存到内存中的。

Rdb：快照形式，定期把内存中当前时刻的数据保存到磁盘。Redis默认支持的持久化方案。

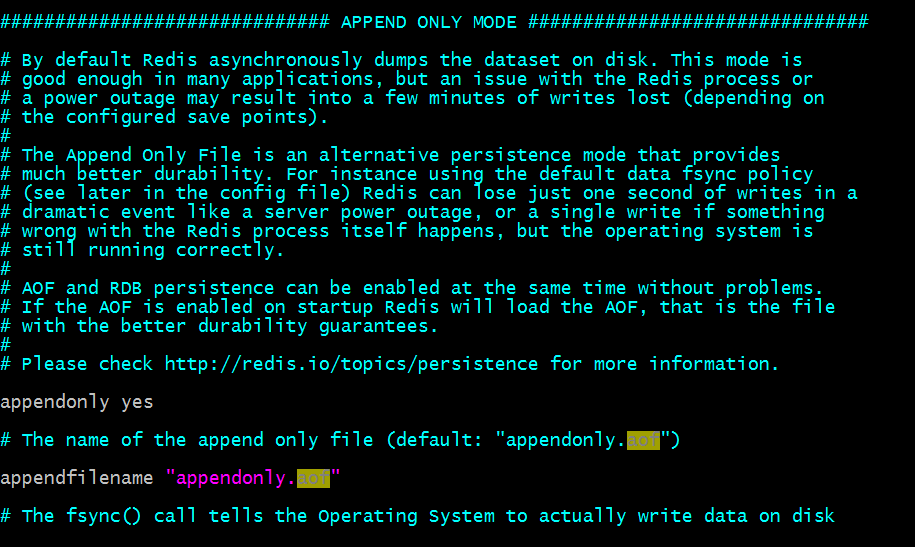
aof形式：append only file。把所有对redis数据库操作的命令，增删改操作的命令。保存到文件中。数据库恢复时把所有的命令执行一遍即可。

在redis.conf配置文件中配置。

Rdb：



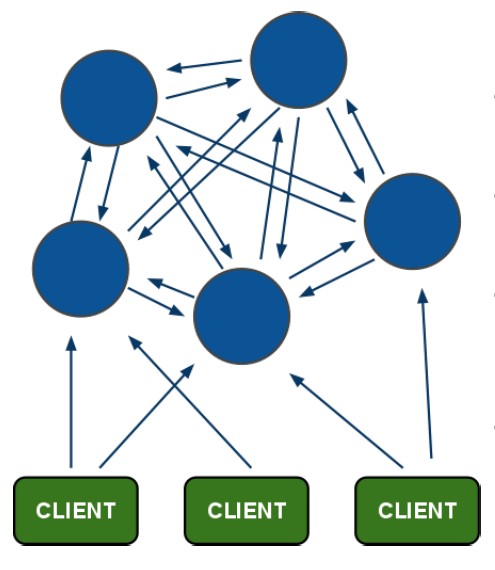
Aof的配置：



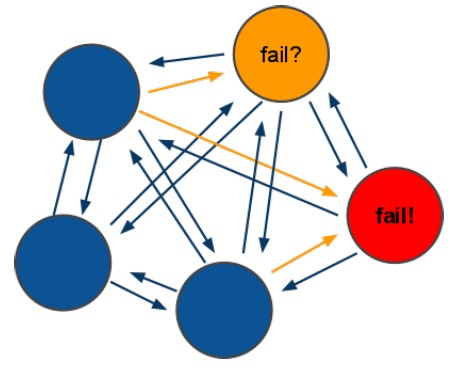
两种持久化方案同时开启使用aof文件来恢复数据库。

### 3.5 Redis集群简介

redis-cluster架构图



redis-cluster投票:容错



架构细节:

(1)所有的redis节点彼此互联(PING-PONG机制),内部使用二进制协议优化传输速度和带宽.

(2)节点的fail是通过集群中超过半数的节点检测失效时才生效.

(3)客户端与redis节点直连,不需要中间proxy层.客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可

(4)redis-cluster把所有的物理节点映射到[0-16383]slot上,cluster 负责维护node<->slot<->value

Redis 集群中内置了 16384 个哈希槽，当需要在 Redis 集群中放置一个 key-value 时，redis 先对 key 使用 crc16 算法算出一个结果，然后把结果对 16384 求余数，这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽，redis 会根据节点数量大致均等的将哈希槽映射到不同的节点

Server1

0-5000

Hello1 1500

Hello 500

Server3

10001-16383

Server2

5001-10000

Hello3 7500

Hello2 11500

### 3.6 Redis集群的搭建

Redis集群中至少应该有三个节点。要保证集群的高可用，需要每个节点有一个备份机。

Redis集群至少需要6台服务器。

搭建伪分布式。可以使用一台虚拟机运行6个redis实例。需要修改redis的端口号7001-7006

集群搭建环境

1、使用ruby脚本搭建集群。需要ruby的运行环境。

安装ruby

yum install ruby

yum install rubygems

1. 安装ruby脚本运行使用的包。

[root@localhost ~]# gem install redis-3.0.0.gem

Successfully installed redis-3.0.0

1 gem installed

Installing ri documentation for redis-3.0.0...

Installing RDoc documentation for redis-3.0.0...

[root@localhost ~]#

[root@localhost ~]# cd redis-3.0.0/src

[root@localhost src]# ll \*.rb

-rwxrwxr-x. 1 root root 48141 Apr 1 2015 redis-trib.rb

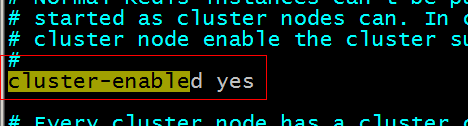
搭建步骤

需要6台redis服务器。搭建伪分布式。

需要6个redis实例。

需要运行在不同的端口7001-7006

第一步：创建6个redis实例，每个实例运行在不同的端口。需要修改redis.conf配置文件。配置文件中还需要把cluster-enabled yes前的注释去掉。



第二步：启动每个redis实例。

第三步：使用ruby脚本搭建集群。

|  |
| --- |
| ./redis-trib.rb create --replicas 1 192.168.25.153:7001 192.168.25.153:7002 192.168.25.153:7003 192.168.25.153:7004 192.168.25.153:7005 192.168.25.153:7006 |

创建关闭集群的脚本：

[root@localhost redis-cluster]# vim shutdow-all.sh

redis01/redis-cli -p 7001 shutdown

redis01/redis-cli -p 7002 shutdown

redis01/redis-cli -p 7003 shutdown

redis01/redis-cli -p 7004 shutdown

redis01/redis-cli -p 7005 shutdown

redis01/redis-cli -p 7006 shutdown

[root@localhost redis-cluster]# chmod u+x shutdow-all.sh

|  |
| --- |
| [root@localhost redis-cluster]# ./redis-trib.rb create --replicas 1 192.168.25.153:7001 192.168.25.153:7002 192.168.25.153:7003 192.168.25.153:7004 192.168.25.153:7005 192.168.25.153:7006  >>> Creating cluster  Connecting to node 192.168.25.153:7001: OK  Connecting to node 192.168.25.153:7002: OK  Connecting to node 192.168.25.153:7003: OK  Connecting to node 192.168.25.153:7004: OK  Connecting to node 192.168.25.153:7005: OK  Connecting to node 192.168.25.153:7006: OK  >>> Performing hash slots allocation on 6 nodes...  Using 3 masters:  192.168.25.153:7001  192.168.25.153:7002  192.168.25.153:7003  Adding replica 192.168.25.153:7004 to 192.168.25.153:7001  Adding replica 192.168.25.153:7005 to 192.168.25.153:7002  Adding replica 192.168.25.153:7006 to 192.168.25.153:7003  M: 2e48ae301e9c32b04a7d4d92e15e98e78de8c1f3 192.168.25.153:7001  slots:0-5460 (5461 slots) master  M: 8cd93a9a943b4ef851af6a03edd699a6061ace01 192.168.25.153:7002  slots:5461-10922 (5462 slots) master  M: 2935007902d83f20b1253d7f43dae32aab9744e6 192.168.25.153:7003  slots:10923-16383 (5461 slots) master  S: 74f9d9706f848471583929fc8bbde3c8e99e211b 192.168.25.153:7004  replicates 2e48ae301e9c32b04a7d4d92e15e98e78de8c1f3  S: 42cc9e25ebb19dda92591364c1df4b3a518b795b 192.168.25.153:7005  replicates 8cd93a9a943b4ef851af6a03edd699a6061ace01  S: 8b1b11d509d29659c2831e7a9f6469c060dfcd39 192.168.25.153:7006  replicates 2935007902d83f20b1253d7f43dae32aab9744e6  Can I set the above configuration? (type 'yes' to accept): yes  >>> Nodes configuration updated  >>> Assign a different config epoch to each node  >>> Sending CLUSTER MEET messages to join the cluster  Waiting for the cluster to join.....  >>> Performing Cluster Check (using node 192.168.25.153:7001)  M: 2e48ae301e9c32b04a7d4d92e15e98e78de8c1f3 192.168.25.153:7001  slots:0-5460 (5461 slots) master  M: 8cd93a9a943b4ef851af6a03edd699a6061ace01 192.168.25.153:7002  slots:5461-10922 (5462 slots) master  M: 2935007902d83f20b1253d7f43dae32aab9744e6 192.168.25.153:7003  slots:10923-16383 (5461 slots) master  M: 74f9d9706f848471583929fc8bbde3c8e99e211b 192.168.25.153:7004  slots: (0 slots) master  replicates 2e48ae301e9c32b04a7d4d92e15e98e78de8c1f3  M: 42cc9e25ebb19dda92591364c1df4b3a518b795b 192.168.25.153:7005  slots: (0 slots) master  replicates 8cd93a9a943b4ef851af6a03edd699a6061ace01  M: 8b1b11d509d29659c2831e7a9f6469c060dfcd39 192.168.25.153:7006  slots: (0 slots) master  replicates 2935007902d83f20b1253d7f43dae32aab9744e6  [OK] All nodes agree about slots configuration.  >>> Check for open slots...  >>> Check slots coverage...  [OK] All 16384 slots covered.  [root@localhost redis-cluster]# |

### 3.7 集群的使用方法

Redis-cli连接集群。

[root@localhost redis-cluster]# redis01/redis-cli -p 7002 -c

-c：代表连接的是redis集群

### 3.8 Jedis

需要把jedis依赖的jar包添加到工程中。Maven工程中需要把jedis的坐标添加到依赖。

推荐添加到服务层。Taotao-content-Service工程中。

#### 3.8.1 连接单机版

第一步：创建一个Jedis对象。需要指定服务端的ip及端口。

第二步：使用Jedis对象操作数据库，每个redis命令对应一个方法。

第三步：打印结果。

第四步：关闭Jedis

|  |
| --- |
| @Test  **public** **void** testJedis() **throws** Exception {  // 第一步：创建一个Jedis对象。需要指定服务端的ip及端口。  Jedis jedis = **new** Jedis("192.168.25.153", 6379);  // 第二步：使用Jedis对象操作数据库，每个redis命令对应一个方法。  String result = jedis.get("hello");  // 第三步：打印结果。  System.***out***.println(result);  // 第四步：关闭Jedis  jedis.close();  } |

#### 3.8.2连接单机版使用连接池

第一步：创建一个JedisPool对象。需要指定服务端的ip及端口。

第二步：从JedisPool中获得Jedis对象。

第三步：使用Jedis操作redis服务器。

第四步：操作完毕后关闭jedis对象，连接池回收资源。

第五步：关闭JedisPool对象。

|  |
| --- |
| @Test  **public** **void** testJedisPool() **throws** Exception {  // 第一步：创建一个JedisPool对象。需要指定服务端的ip及端口。  JedisPool jedisPool = **new** JedisPool("192.168.25.153", 6379);  // 第二步：从JedisPool中获得Jedis对象。  Jedis jedis = jedisPool.getResource();  // 第三步：使用Jedis操作redis服务器。  jedis.set("jedis", "test");  String result = jedis.get("jedis");  System.***out***.println(result);  // 第四步：操作完毕后关闭jedis对象，连接池回收资源。  jedis.close();  // 第五步：关闭JedisPool对象。  jedisPool.close();  } |

#### 3.8.3连接集群版

第一步：使用JedisCluster对象。需要一个Set<HostAndPort>参数。Redis节点的列表。

第二步：直接使用JedisCluster对象操作redis。在系统中单例存在。

第三步：打印结果

第四步：系统关闭前，关闭JedisCluster对象。

|  |
| --- |
| @Test  **public** **void** testJedisCluster() **throws** Exception {  // 第一步：使用JedisCluster对象。需要一个Set<HostAndPort>参数。Redis节点的列表。  Set<HostAndPort> nodes = **new** HashSet<>();  nodes.add(**new** HostAndPort("192.168.25.153", 7001));  nodes.add(**new** HostAndPort("192.168.25.153", 7002));  nodes.add(**new** HostAndPort("192.168.25.153", 7003));  nodes.add(**new** HostAndPort("192.168.25.153", 7004));  nodes.add(**new** HostAndPort("192.168.25.153", 7005));  nodes.add(**new** HostAndPort("192.168.25.153", 7006));  JedisCluster jedisCluster = **new** JedisCluster(nodes);  // 第二步：直接使用JedisCluster对象操作redis。在系统中单例存在。  jedisCluster.set("hello", "100");  String result = jedisCluster.get("hello");  // 第三步：打印结果  System.***out***.println(result);  // 第四步：系统关闭前，关闭JedisCluster对象。  jedisCluster.close();  } |

### 3.9向业务逻辑中添加缓存

#### 3.9.1接口封装

常用的操作redis的方法提取出一个接口，分别对应单机版和集群版创建两个实现类。

接口定义

|  |
| --- |
| **public** **interface** JedisClient {  String set(String key, String value);  String get(String key);  Boolean exists(String key);  Long expire(String key, **int** seconds);  Long ttl(String key);  Long incr(String key);  Long hset(String key, String field, String value);  String hget(String key, String field);  Long hdel(String key, String... field);  } |

#### 3.9.2单机版实现类

|  |
| --- |
| **public** **class** JedisClientPool **implements** JedisClient {    @Autowired  **private** JedisPool jedisPool;  @Override  **public** String set(String key, String value) {  Jedis jedis = jedisPool.getResource();  String result = jedis.set(key, value);  jedis.close();  **return** result;  }  @Override  **public** String get(String key) {  Jedis jedis = jedisPool.getResource();  String result = jedis.get(key);  jedis.close();  **return** result;  }  @Override  **public** Boolean exists(String key) {  Jedis jedis = jedisPool.getResource();  Boolean result = jedis.exists(key);  jedis.close();  **return** result;  }  @Override  **public** Long expire(String key, **int** seconds) {  Jedis jedis = jedisPool.getResource();  Long result = jedis.expire(key, seconds);  jedis.close();  **return** result;  }  @Override  **public** Long ttl(String key) {  Jedis jedis = jedisPool.getResource();  Long result = jedis.ttl(key);  jedis.close();  **return** result;  }  @Override  **public** Long incr(String key) {  Jedis jedis = jedisPool.getResource();  Long result = jedis.incr(key);  jedis.close();  **return** result;  }  @Override  **public** Long hset(String key, String field, String value) {  Jedis jedis = jedisPool.getResource();  Long result = jedis.hset(key, field, value);  jedis.close();  **return** result;  }  @Override  **public** String hget(String key, String field) {  Jedis jedis = jedisPool.getResource();  String result = jedis.hget(key, field);  jedis.close();  **return** result;  }  @Override  **public** Long hdel(String key, String... field) {  Jedis jedis = jedisPool.getResource();  Long result = jedis.hdel(key, field);  jedis.close();  **return** result;  }  } |

配置：applicationContext-redis.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"* xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans4.2.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context4.2.xsd*  *http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop4.2.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx4.2.xsd*  *http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util4.2.xsd"*>  <!-- 配置单机版的连接 -->  <bean id=*"jedisPool"* class=*"redis.clients.jedis.JedisPool"*>  <constructor-arg name=*"host"* value=*"192.168.25.153"*></constructor-arg>  <constructor-arg name=*"port"* value=*"6379"*></constructor-arg>  </bean>  <bean id=*"jedisClientPool"* class=*"com.taotao.jedis.JedisClientPool"*/>    </beans> |

#### 3.9.3集群版实现类

|  |
| --- |
| **package** com.taotao.jedis;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** redis.clients.jedis.JedisCluster;  **public** **class** JedisClientCluster **implements** JedisClient {    @Autowired  **private** JedisCluster jedisCluster;  @Override  **public** String set(String key, String value) {  **return** jedisCluster.set(key, value);  }  @Override  **public** String get(String key) {  **return** jedisCluster.get(key);  }  @Override  **public** Boolean exists(String key) {  **return** jedisCluster.exists(key);  }  @Override  **public** Long expire(String key, **int** seconds) {  **return** jedisCluster.expire(key, seconds);  }  @Override  **public** Long ttl(String key) {  **return** jedisCluster.ttl(key);  }  @Override  **public** Long incr(String key) {  **return** jedisCluster.incr(key);  }  @Override  **public** Long hset(String key, String field, String value) {  **return** jedisCluster.hset(key, field, value);  }  @Override  **public** String hget(String key, String field) {  **return** jedisCluster.hget(key, field);  }  @Override  **public** Long hdel(String key, String... field) {  **return** jedisCluster.hdel(key, field);  }  } |

Spring的配置：

|  |
| --- |
| <!-- 集群版的配置 -->  <bean id=*"jedisCluster"* class=*"redis.clients.jedis.JedisCluster"*>  <constructor-arg>  <set>  <bean class=*"redis.clients.jedis.HostAndPort"*>  <constructor-arg name=*"host"* value=*"192.168.25.153"*></constructor-arg>  <constructor-arg name=*"port"* value=*"7001"*></constructor-arg>  </bean>  <bean class=*"redis.clients.jedis.HostAndPort"*>  <constructor-arg name=*"host"* value=*"192.168.25.153"*></constructor-arg>  <constructor-arg name=*"port"* value=*"7002"*></constructor-arg>  </bean>  <bean class=*"redis.clients.jedis.HostAndPort"*>  <constructor-arg name=*"host"* value=*"192.168.25.153"*></constructor-arg>  <constructor-arg name=*"port"* value=*"7003"*></constructor-arg>  </bean>  <bean class=*"redis.clients.jedis.HostAndPort"*>  <constructor-arg name=*"host"* value=*"192.168.25.153"*></constructor-arg>  <constructor-arg name=*"port"* value=*"7004"*></constructor-arg>  </bean>  <bean class=*"redis.clients.jedis.HostAndPort"*>  <constructor-arg name=*"host"* value=*"192.168.25.153"*></constructor-arg>  <constructor-arg name=*"port"* value=*"7005"*></constructor-arg>  </bean>  <bean class=*"redis.clients.jedis.HostAndPort"*>  <constructor-arg name=*"host"* value=*"192.168.25.153"*></constructor-arg>  <constructor-arg name=*"port"* value=*"7006"*></constructor-arg>  </bean>  </set>  </constructor-arg>  </bean>  <bean id=*"jedisClientCluster"* class=*"com.taotao.jedis.JedisClientCluster"*/> |

注意：单机版和集群版不能共存，使用单机版时注释集群版的配置。使用集群版，把单机版注释。

封装代码测试

|  |
| --- |
| @Test  **public** **void** testJedisClient() **throws** Exception {  //初始化Spring容器  ApplicationContext applicationContext = **new** ClassPathXmlApplicationContext("classpath:spring/applicationContext-\*.xml");  //从容器中获得JedisClient对象  JedisClient jedisClient = applicationContext.getBean(JedisClient.**class**);  jedisClient.set("first", "100");  String result = jedisClient.get("first");  System.***out***.println(result);      } |

### 3.10 使用实例

功能分析

查询内容列表时添加缓存。

1. 查询数据库之前先查询缓存。
2. 查询到结果，直接响应结果。
3. 查询不到，缓存中没有需要查询数据库。
4. 把查询结果添加到缓存中。
5. 返回结果。

向redis中添加缓存：

Key：cid

Value：内容列表。需要把java对象转换成json。

使用hash对key进行归类。

HASH\_KEY:HASH

|--KEY:VALUE

|--KEY:VALUE

|--KEY:VALUE

|--KEY:VALUE

**注意：添加缓存不能影响正常业务逻辑。**

代码实现

|  |
| --- |
| @Override  **public** List<TbContent> getContentList(**long** cid) {  //查询缓存  **try** {  String json = jedisClient.hget(CONTENT\_KEY, cid + "");  //判断json是否为空  **if** (StringUtils.*isNotBlank*(json)) {  //把json转换成list  List<TbContent> list = JsonUtils.*jsonToList*(json, TbContent.**class**);  **return** list;  }  } **catch** (Exception e) {  e.printStackTrace();  }  //根据cid查询内容列表  TbContentExample example = **new** TbContentExample();  //设置查询条件  Criteria criteria = example.createCriteria();  criteria.andCategoryIdEqualTo(cid);  //执行查询  List<TbContent> list = contentMapper.selectByExample(example);  //向缓存中添加数据  **try** {  jedisClient.hset(CONTENT\_KEY, cid + "", JsonUtils.*objectToJson*(list));  } **catch** (Exception e) {  e.printStackTrace();  }  **return** list;  } |

缓存同步

对内容信息做增删改操作后只需要把对应缓存删除即可。

可以根据cid删除。

|  |
| --- |
| @Override  **public** TaotaoResult addContent(TbContent content) {  //补全属性  content.setCreated(**new** Date());  content.setUpdated(**new** Date());  //插入数据  contentMapper.insert(content);  //缓存同步  jedisClient.hdel(CONTENT\_KEY, content.getCategoryId().toString());    **return** TaotaoResult.*ok*();  } |

## 四.*S*olr中间件

### 4.1 单机版solr 服务器

#### 4.1.1 服务搭建

Solr的环境

* Solr是java开发。
* 需要安装jdk。
* 安装环境Linux。
* 需要安装Tomcat。

搭建步骤

第一步：把solr 的压缩包上传到Linux系统

第二步：解压solr。

第三步：安装Tomcat，解压缩即可。

第四步：把solr部署到Tomcat下。

第五步：解压缩war包。启动Tomcat解压。

第六步：把/root/solr-4.10.3/example/lib/ext目录下的所有的jar包，添加到solr工程中。

[root@localhost ext]# pwd

/root/solr-4.10.3/example/lib/ext

[root@localhost ext]# cp \* /usr/local/solr/tomcat/webapps/solr/WEB-INF/lib/

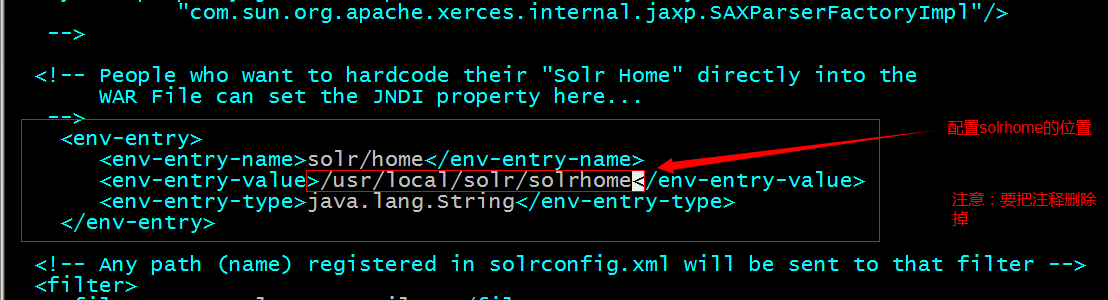
第七步：创建一个solrhome。/example/solr目录就是一个solrhome。复制此目录到/usr/local/solr/solrhome

[root@localhost example]# pwd

/root/solr-4.10.3/example

[root@localhost example]# cp -r solr /usr/local/solr/solrhome

[root@localhost example]#

第八步：关联solr及solrhome。需要修改solr工程的web.xml文件。  


第九步：启动Tomcat

<http://192.168.25.154:8080/solr/>

和windows下的配置完全一样。

配置业务域

schema.xml中定义

1. 商品Id
2. 商品标题
3. 商品卖点
4. 商品价格
5. 商品图片
6. 分类名称
7. 商品描述

创建对应的业务域。需要制定中文分析器。

创建步骤：

第一步：把中文分析器添加到工程中。

1. 把IKAnalyzer2012FF\_u1.jar添加到solr工程的lib目录下
2. 把扩展词典、配置文件放到solr工程的WEB-INF/classes目录下。

第二步：配置一个FieldType，制定使用IKAnalyzer

修改schema.xml文件

修改Solr的schema.xml文件，添加FieldType：

|  |
| --- |
| <fieldType name="text\_ik" class="solr.TextField">  <analyzer class="org.wltea.analyzer.lucene.IKAnalyzer"/>  </fieldType> |

第三步：配置业务域，type制定使用自定义的FieldType。

设置业务系统Field

|  |
| --- |
| <field name="item\_title" type="text\_ik" indexed="true" stored="true"/>  <field name="item\_sell\_point" type="text\_ik" indexed="true" stored="true"/>  <field name="item\_price" type="long" indexed="true" stored="true"/>  <field name="item\_image" type="string" indexed="false" stored="true" />  <field name="item\_category\_name" type="string" indexed="true" stored="true" />  <field name="item\_desc" type="text\_ik" indexed="true" stored="false" />  <field name="item\_keywords" type="text\_ik" indexed="true" stored="false" multiValued="true"/>  <copyField source="item\_title" dest="item\_keywords"/>  <copyField source="item\_sell\_point" dest="item\_keywords"/>  <copyField source="item\_category\_name" dest="item\_keywords"/>  <copyField source="item\_desc" dest="item\_keywords"/> |

第四步：重启tomcat

### 4.2 SlorCloud 集群

#### 4.2.1 什么是SolrCloud

SolrCloud(solr 云)是Solr提供的分布式搜索方案，当你需要大规模，容错，分布式索引和检索能力时使用 SolrCloud。当一个系统的索引数据量少的时候是不需要使用SolrCloud的，当索引量很大，搜索请求并发很高，这时需要使用SolrCloud来满足这些需求。

SolrCloud是基于Solr和Zookeeper的分布式搜索方案，它的主要思想是使用Zookeeper作为集群的配置信息中心。

它有几个特色功能：

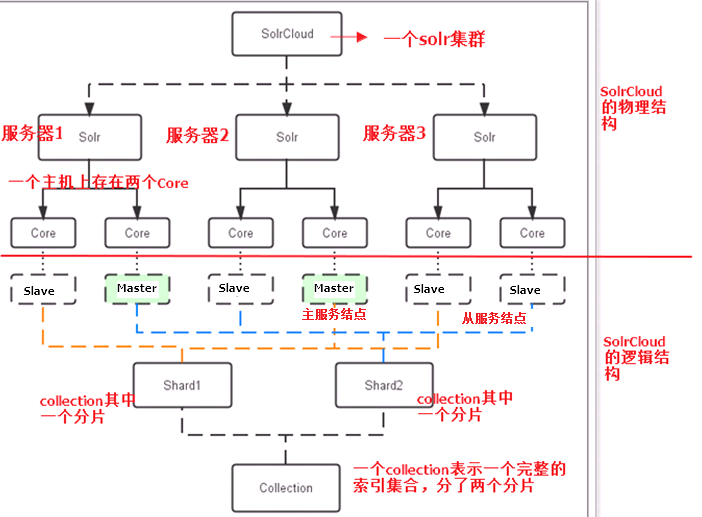
1）集中式的配置信息

2）自动容错

3）近实时搜索

4）查询时自动负载均衡

#### 4.2.2 Solr集群的系统架构



**物理结构**

三个Solr实例（ 每个实例包括两个Core），组成一个SolrCloud。

**逻辑结构**

索引集合包括两个Shard（shard1和shard2），shard1和shard2分别由三个Core组成，其中一个Leader两个Replication，Leader是由zookeeper选举产生，zookeeper控制每个shard上三个Core的索引数据一致，解决高可用问题。

用户发起索引请求分别从shard1和shard2上获取，解决高并发问题。

**collection**

Collection在SolrCloud集群中是一个逻辑意义上的完整的索引结构。它常常被划分为一个或多个Shard（分片），它们使用相同的配置信息。

比如：针对商品信息搜索可以创建一个collection。

collection=shard1+shard2+....+shardX

**Core**

每个Core是Solr中一个独立运行单位，提供 索引和搜索服务。一个shard需要由一个Core或多个Core组成。由于collection由多个shard组成所以collection一般由多个core组成。

**Master或Slave**

Master是master-slave结构中的主结点（通常说主服务器），Slave是master-slave结构中的从结点（通常说从服务器或备服务器）。同一个Shard下master和slave存储的数据是一致的，这是为了达到高可用目的。

**Shard**

Collection的逻辑分片。每个Shard被化成一个或者多个replication，通过选举确定哪个是Leader。

#### 4.2.3需要实现的solr集群架构



Zookeeper作为集群的管理工具。

1. 集群管理：容错、负载均衡。
2. 配置文件的集中管理
3. 集群的入口

需要实现zookeeper 高可用。需要搭建集群。建议是奇数节点。需要三个zookeeper服务器。

搭建solr集群需要7台服务器。

搭建伪分布式：

需要三个zookeeper节点

需要四个tomcat节点。

建议虚拟机的内容1G以上。

**环境准备**

CentOS-6.5-i386-bin-DVD1.iso

jdk-7u72-linux-i586.tar.gz

apache-tomcat-7.0.47.tar.gz

zookeeper-3.4.6.tar.gz

solr-4.10.3.tgz

#### 4.2.4 Zookeeper集群搭建

第一步：需要安装jdk环境。

第二步：把zookeeper的压缩包上传到服务器。

第三步：解压缩。

第四步：把zookeeper复制三份。

[root@localhost ~]# mkdir /usr/local/solr-cloud

[root@localhost ~]# cp -r zookeeper-3.4.6 /usr/local/solr-cloud/zookeeper01

[root@localhost ~]# cp -r zookeeper-3.4.6 /usr/local/solr-cloud/zookeeper02

[root@localhost ~]# cp -r zookeeper-3.4.6 /usr/local/solr-cloud/zookeeper03

第五步：在每个zookeeper目录下创建一个data目录。

第六步：在data目录下创建一个myid文件，文件名就叫做“myid”。内容就是每个实例的id。例如1、2、3

[root@localhost data]# echo 1 >> myid

[root@localhost data]# ll

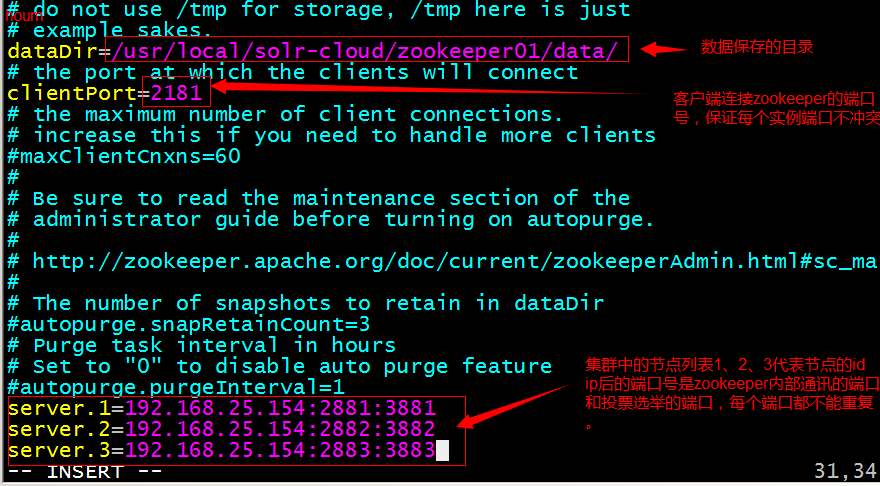
total 4

-rw-r--r--. 1 root root 2 Apr 7 18:23 myid

[root@localhost data]# cat myid

1

第七步：修改配置文件。把conf目录下的zoo\_sample.cfg文件改名为zoo.cfg



|  |
| --- |
| server.1=192.168.25.154:2881:3881  server.2=192.168.25.154:2882:3882  server.3=192.168.25.154:2883:3883 |

第八步：启动每个zookeeper实例。

启动bin/zkServer.sh start

查看zookeeper的状态：

bin/zkServer.sh status

**Solr集群的搭建**

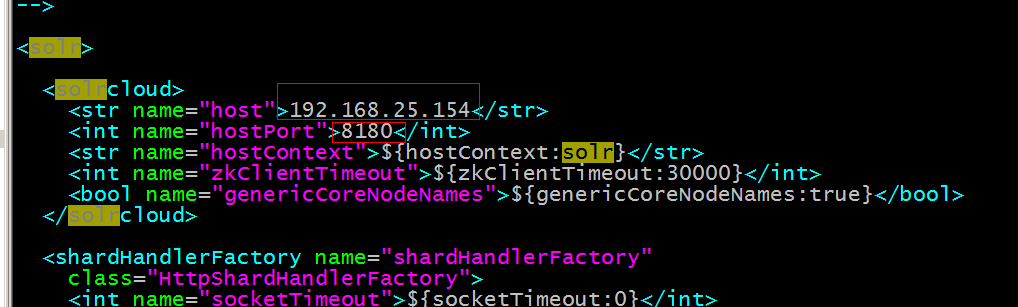
第一步：创建四个tomcat实例。每个tomcat运行在不同的端口。8180、8280、8380、8480

第二步：部署solr的war包。把单机版的solr工程复制到集群中的tomcat中。

第三步：为每个solr实例创建一个对应的solrhome。使用单机版的solrhome复制四份。

第四步：需要修改solr的web.xml文件。把solrhome关联起来。

第五步：配置solrCloud相关的配置。每个solrhome下都有一个solr.xml，把其中的ip及端口号配置好。



第六步：让zookeeper统一管理配置文件。需要把solrhome/collection1/conf目录上传到zookeeper。上传任意solrhome中的配置文件即可。

使用工具上传配置文件：/root/solr-4.10.3/example/scripts/cloud-scripts/zkcli.sh

|  |
| --- |
| ./zkcli.sh -zkhost 192.168.25.154:2181,192.168.25.154:2182,192.168.25.154:2183 -cmd upconfig -confdir /usr/local/solr-cloud/solrhome01/collection1/conf -confname myconf |

查看zookeeper上的配置文件：

使用zookeeper目录下的bin/zkCli.sh命令查看zookeeper上的配置文件：

[root@localhost bin]# ./zkCli.sh

[zk: localhost:2181(CONNECTED) 0] ls /

[configs, zookeeper]

[zk: localhost:2181(CONNECTED) 1] ls /configs

[myconf]

[zk: localhost:2181(CONNECTED) 2] ls /configs/myconf

[admin-extra.menu-top.html, currency.xml, protwords.txt, mapping-FoldToASCII.txt, \_schema\_analysis\_synonyms\_english.json, \_rest\_managed.json, solrconfig.xml, \_schema\_analysis\_stopwords\_english.json, stopwords.txt, lang, spellings.txt, mapping-ISOLatin1Accent.txt, admin-extra.html, xslt, synonyms.txt, scripts.conf, update-script.js, velocity, elevate.xml, admin-extra.menu-bottom.html, clustering, schema.xml]

[zk: localhost:2181(CONNECTED) 3]

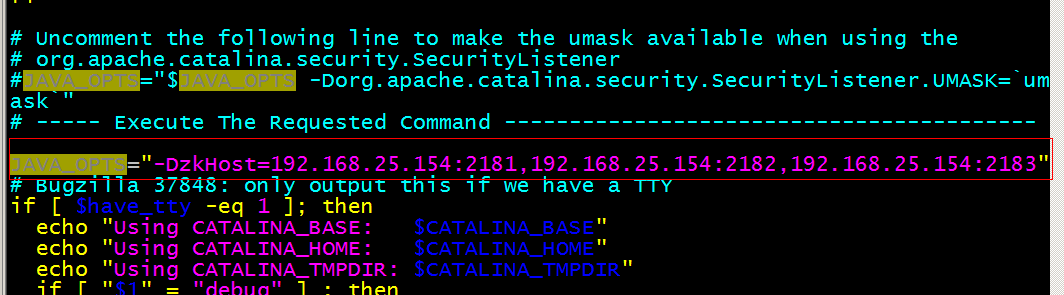
退出：

[zk: localhost:2181(CONNECTED) 3] quit

第七步：修改tomcat/bin目录下的catalina.sh 文件，关联solr和zookeeper。

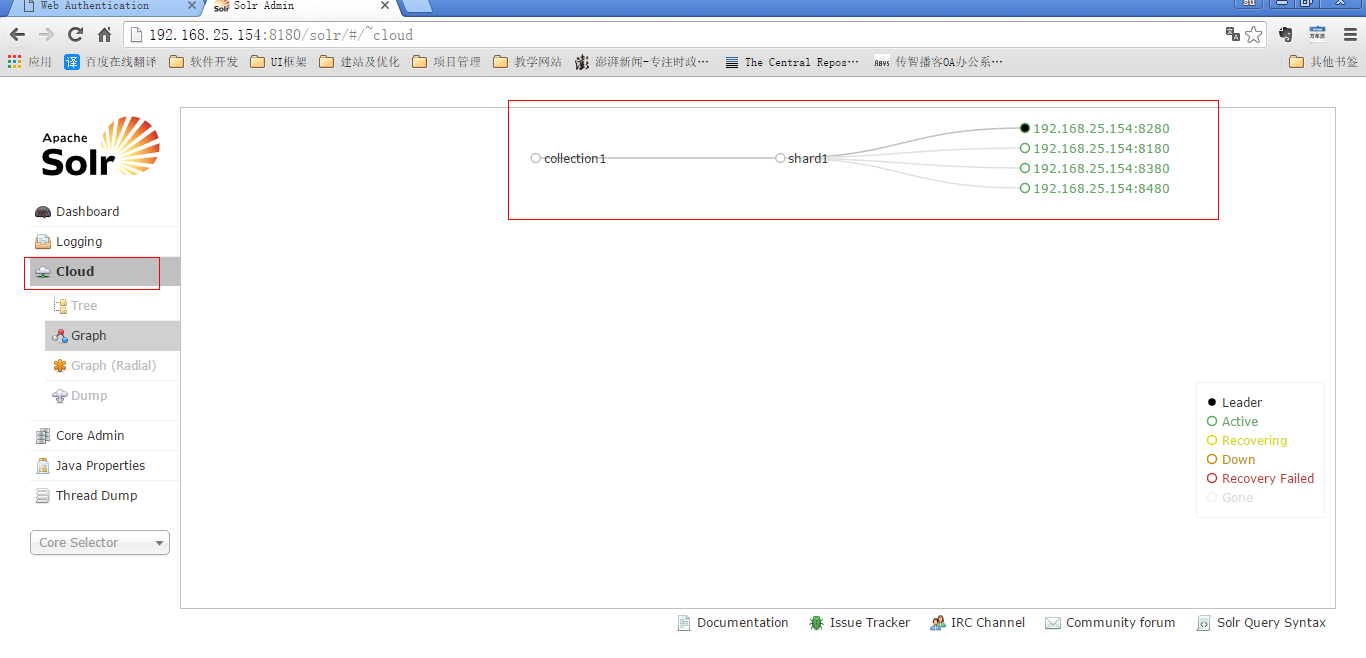
把此配置添加到配置文件中：

JAVA\_OPTS="-DzkHost=192.168.25.154:2181,192.168.25.154:2182,192.168.25.154:2183"



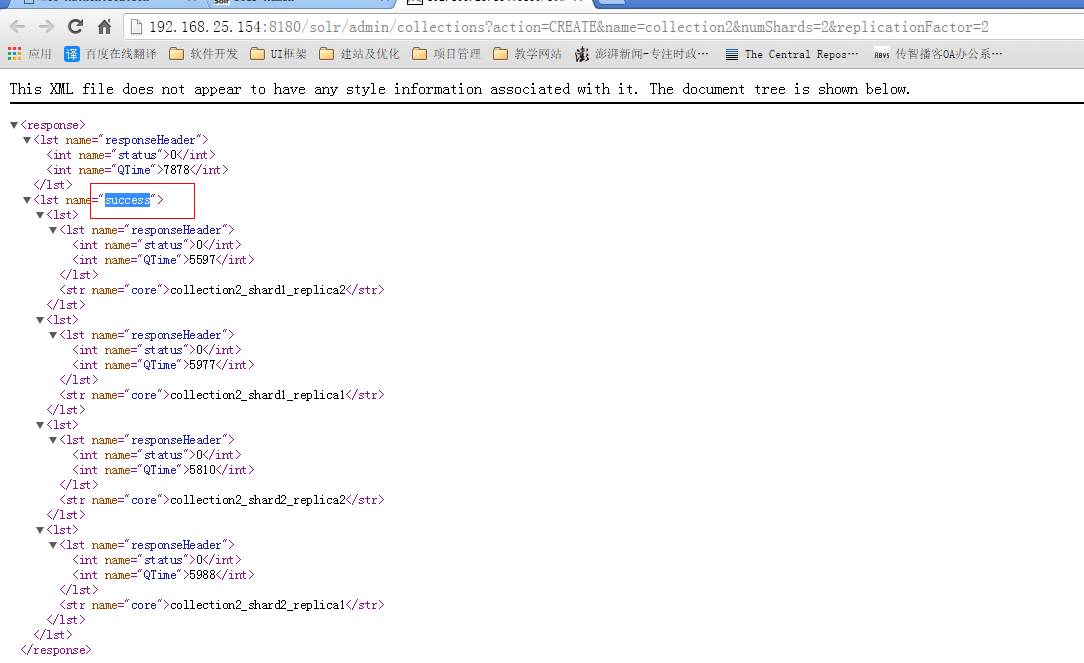
第八步：启动每个tomcat实例。要包装zookeeper集群是启动状态。

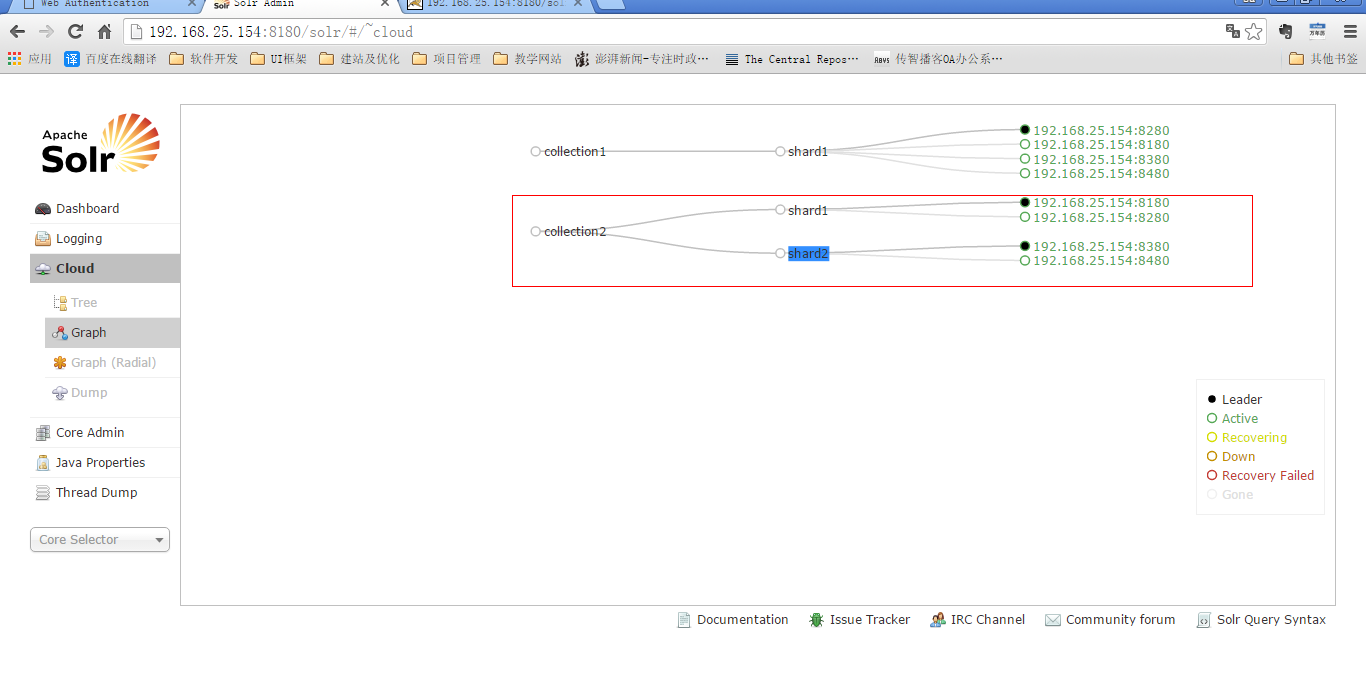
第九步：访问集群



第十步：创建新的Collection进行分片处理。

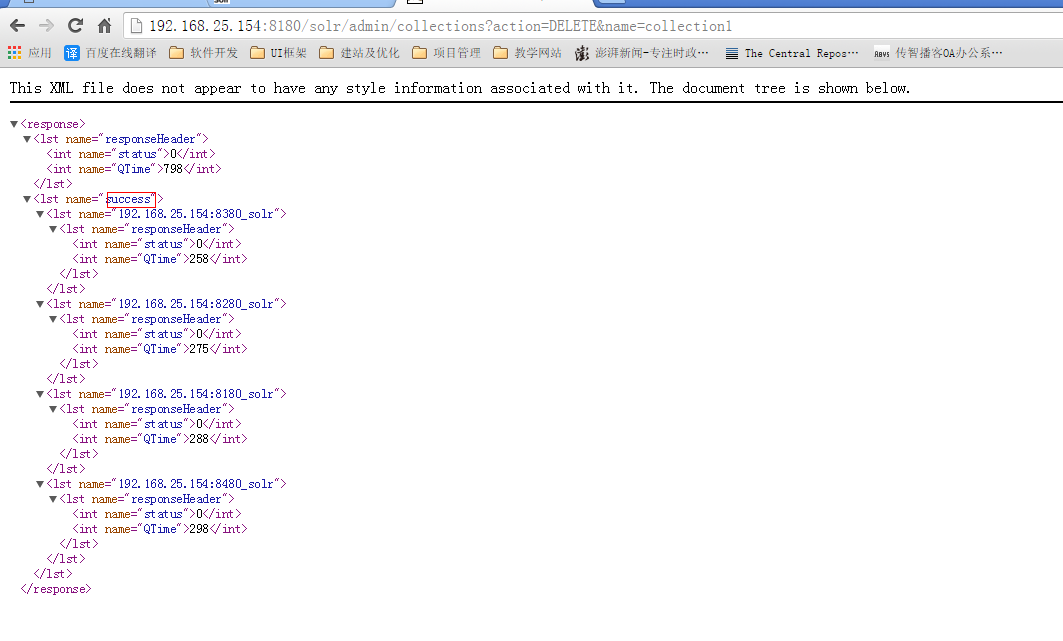
http://192.168.25.154:8180/solr/admin/collections?action=CREATE&name=collection2&numShards=2&replicationFactor=2

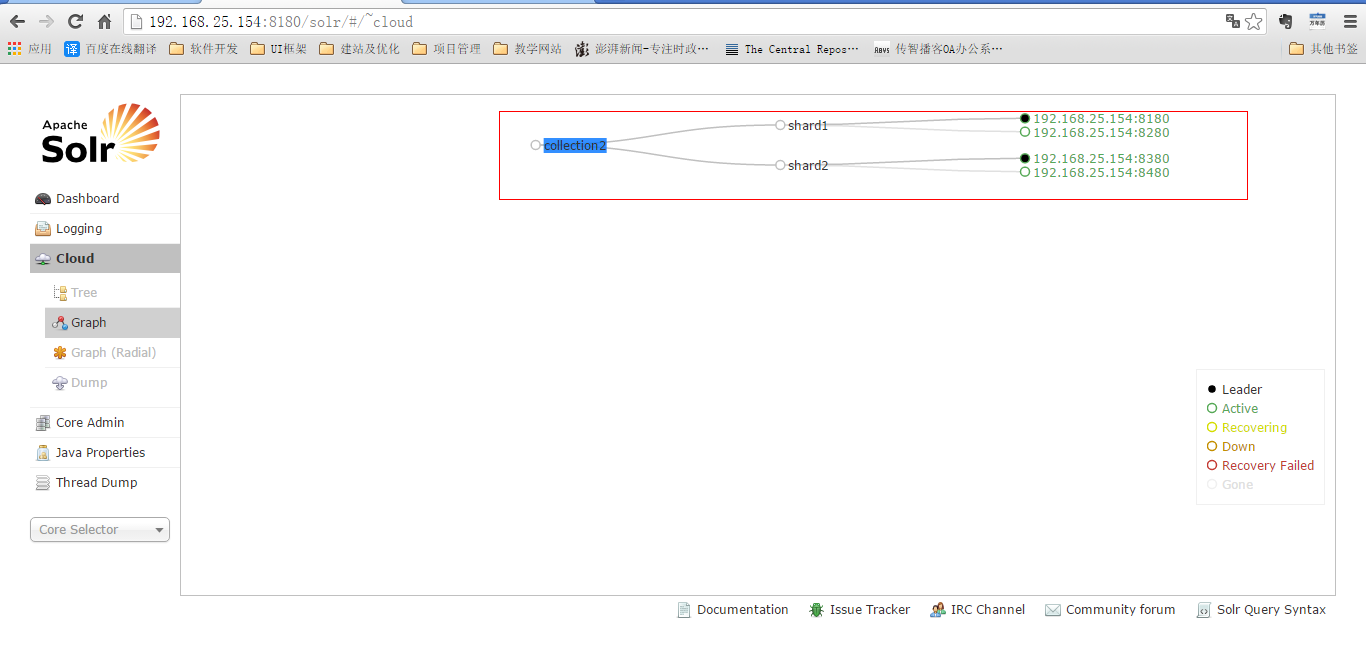




第十一步：删除不用的Collection。

http://192.168.25.154:8180/solr/admin/collections?action=DELETE&name=collection1





#### 4.2.5使用solrJ管理集群

**添加文档**

使用步骤：

第一步：把solrJ相关的jar包添加到工程中。

第二步：创建一个SolrServer对象，需要使用CloudSolrServer子类。构造方法的参数是zookeeper的地址列表。

第三步：需要设置DefaultCollection属性。

第四步：创建一SolrInputDocument对象。

第五步：向文档对象中添加域

第六步：把文档对象写入索引库。

第七步：提交。

|  |
| --- |
| @Test  **public** **void** testSolrCloudAddDocument() **throws** Exception {  // 第一步：把solrJ相关的jar包添加到工程中。  // 第二步：创建一个SolrServer对象，需要使用CloudSolrServer子类。构造方法的参数是zookeeper的地址列表。  //参数是zookeeper的地址列表，使用逗号分隔  CloudSolrServer solrServer = **new** CloudSolrServer("192.168.25.154:2181,192.168.25.154:2182,192.168.25.154:2183");  // 第三步：需要设置DefaultCollection属性。  solrServer.setDefaultCollection("collection2");  // 第四步：创建一SolrInputDocument对象。  SolrInputDocument document = **new** SolrInputDocument();  // 第五步：向文档对象中添加域  document.addField("item\_title", "测试商品");  document.addField("item\_price", "100");  document.addField("id", "test001");  // 第六步：把文档对象写入索引库。  solrServer.add(document);  // 第七步：提交。  solrServer.commit();  } |

**查询文档**

创建一个CloudSolrServer对象，其他处理和单机版一致。

#### 4.2.6把搜索功能切换到集群版

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"* xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans4.2.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context4.2.xsd*  *http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop4.2.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx4.2.xsd*  *http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util4.2.xsd"*>  <!-- 单机版solr服务配置 -->  <!-- <bean id="httpSolrServer" class="org.apache.solr.client.solrj.impl.HttpSolrServer">  <constructor-arg name="baseURL" value="http://192.168.25.154:8080/solr"></constructor-arg>  </bean> -->  <!-- 集群版solr服务 -->  <bean id=*"cloudSolrServer"* class=*"org.apache.solr.client.solrj.impl.CloudSolrServer"*>  <constructor-arg name=*"zkHost"* value=*"192.168.25.154:2181,192.168.25.154:2182,192.168.25.154:2183"*></constructor-arg>  <property name=*"defaultCollection"* value=*"collection2"*></property>  </bean>  </beans> |

## 五MQ中间件

### 5.1 MQ 简介

MQ是一个消息中间件，ActiveMQ、RabbitMQ、kafka

MQ 在本系统中使用场景:

方案一：在taotao-manager中，添加商品的业务逻辑中，添加一个同步索引库的业务逻辑。

缺点：业务逻辑耦合度高，业务拆分不明确

方案二：业务逻辑在taotao-search中实现，调用服务在taotao-manager实现。业务逻辑分开。

缺点：服务之间的耦合度变高。服务的启动有先后顺序。

方案三：使用消息队列。MQ是一个消息中间件。

同步索引库

Consumer

生成静态页面

Consumer

同步缓存

Consumer

添加商品发布消息

Producer

MQ

### 5.2 ActiveMQ

#### 5.2.1什么是ActiveMQ

ActiveMQ 是Apache出品，最流行的，能力强劲的开源消息总线。ActiveMQ 是一个完全支持JMS1.1和J2EE 1.4规范的 JMS Provider实现,尽管JMS规范出台已经是很久的事情了,但是JMS在当今的J2EE应用中间仍然扮演着特殊的地位。

主要特点：

1. 多种语言和协议编写客户端。语言: Java, C, C++, C#, Ruby, Perl, Python, PHP。应用协议: OpenWire,Stomp REST,WS Notification,XMPP,AMQP

2. 完全支持JMS1.1和J2EE 1.4规范 (持久化,XA消息,事务)

3. 对Spring的支持,ActiveMQ可以很容易内嵌到使用Spring的系统里面去,而且也支持Spring2.0的特性

4. 通过了常见J2EE服务器(如 Geronimo,JBoss 4, GlassFish,WebLogic)的测试,其中通过JCA 1.5 resource adaptors的配置,可以让ActiveMQ可以自动的部署到任何兼容J2EE 1.4 商业服务器上

5. 支持多种传送协议:in-VM,TCP,SSL,NIO,UDP,JGroups,JXTA

6. 支持通过JDBC和journal提供高速的消息持久化

7. 从设计上保证了高性能的集群,客户端-服务器,点对点

8. 支持Ajax

9. 支持与Axis的整合

10. 可以很容易得调用内嵌JMS provider,进行测试

#### 5.2.2 ActiveMQ的消息形式

对于消息的传递有两种类型：

**一种是点对点的**，即一个生产者和一个消费者一一对应；

**另一种是发布/订阅模式**，即一个生产者产生消息并进行发送后，可以由多个消费者进行接收。

JMS定义了五种不同的消息正文格式，以及调用的消息类型，允许你发送并接收以一些不同形式的数据，提供现有消息格式的一些级别的兼容性。

　　· StreamMessage -- Java原始值的数据流

　　· MapMessage--一套名称-值对

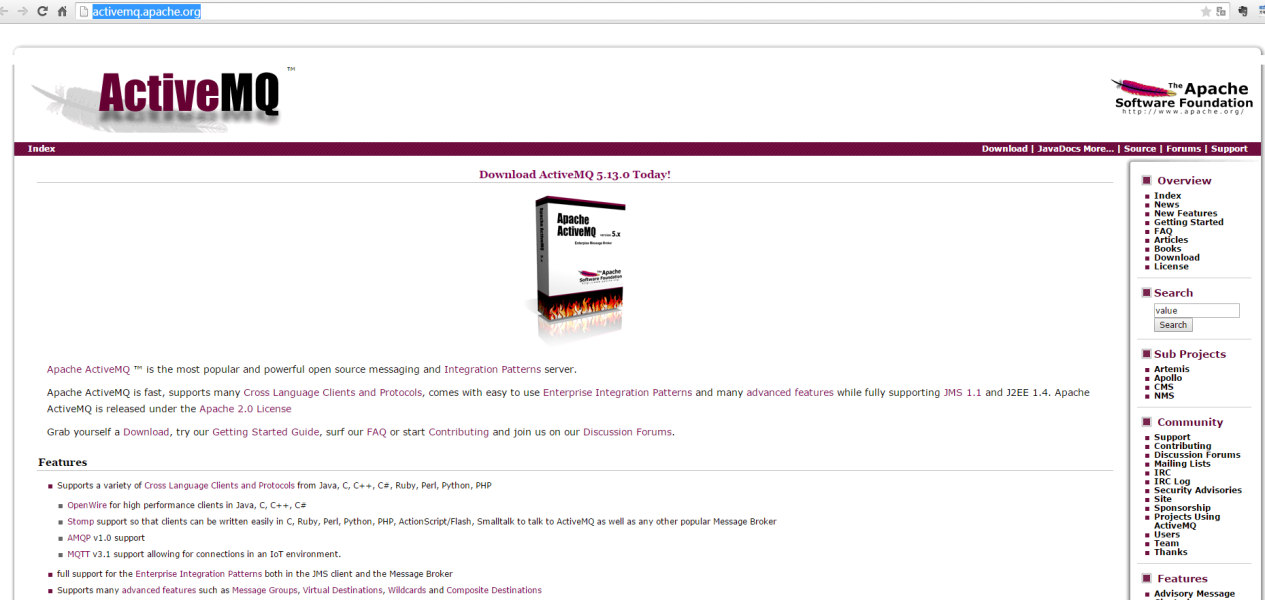
　　· TextMessage--一个字符串对象

　　· ObjectMessage--一个序列化的 Java对象

　　· BytesMessage--一个字节的数据流

#### 5.2.3 ActiveMQ的安装

进入http://activemq.apache.org/下载ActiveMQ



使用的版本是5.12.0

**安装环境：**

1. 需要jdk
2. 安装Linux系统。生产环境都是Linux系统。

**安装步骤**

第一步： 把ActiveMQ 的压缩包上传到Linux系统。

第二步：解压缩。

第三步：启动。

使用bin目录下的activemq命令启动：

[root@localhost bin]# ./activemq start

关闭：

[root@localhost bin]# ./activemq stop

查看状态：

[root@localhost bin]# ./activemq status

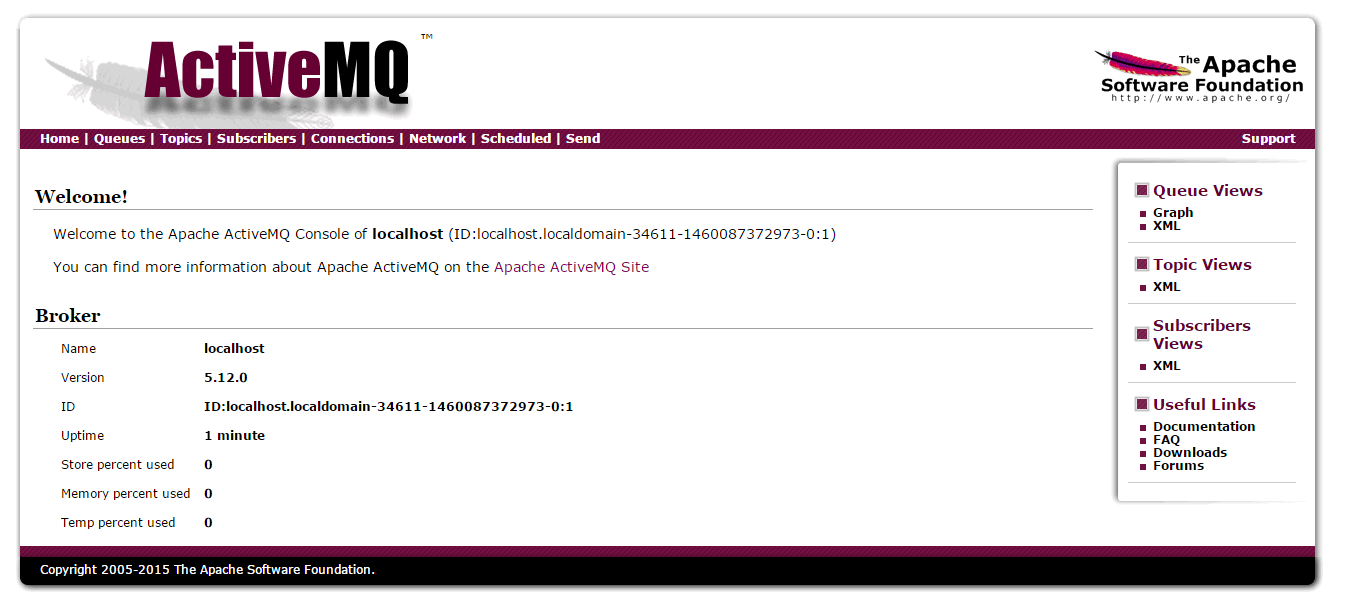
注意：如果ActiveMQ整合spring使用不要使用activemq-all-5.12.0.jar包。建议使用5.11.2

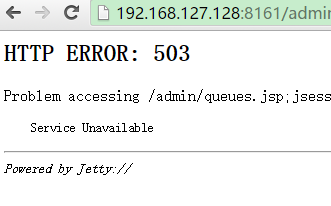
进入管理后台：

<http://192.168.25.168:8161/admin>

用户名：admin

密码：admin





503错误解决:

1、查看机器名

[root@itcast168 bin]# cat /etc/sysconfig/network

NETWORKING=yes

HOSTNAME=itcast168

2、修改host文件

[root@itcast168 bin]# cat /etc/hosts

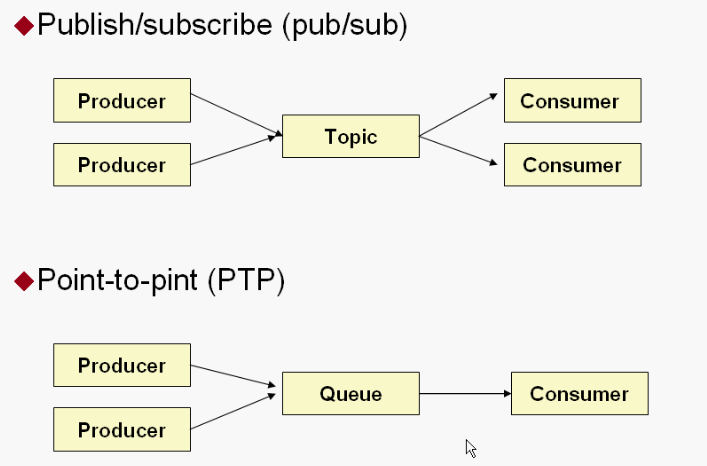
127.0.0.1 localhost localhost.localdomain localhost4 localhost4.localdomain4 itcast168

::1 localhost localhost.localdomain localhost6 localhost6.localdomain6

[root@itcast168 bin]#

3、重启Activemq服务

#### 5.2.4 ActiveMQ的使用方法



#### 5.2.5 Queue

**Producer**

生产者：生产消息，发送端。

把jar包添加到工程中。使用5.11.2版本的jar包。



第一步：创建ConnectionFactory对象，需要指定服务端ip及端口号。

第二步：使用ConnectionFactory对象创建一个Connection对象。

第三步：开启连接，调用Connection对象的start方法。

第四步：使用Connection对象创建一个Session对象。

第五步：使用Session对象创建一个Destination对象（topic、queue），此处创建一个Queue对象。

第六步：使用Session对象创建一个Producer对象。

第七步：创建一个Message对象，创建一个TextMessage对象。

第八步：使用Producer对象发送消息。

第九步：关闭资源。

|  |
| --- |
| @Test  **public** **void** testQueueProducer() **throws** Exception {  // 第一步：创建ConnectionFactory对象，需要指定服务端ip及端口号。  //brokerURL服务器的ip及端口号  ConnectionFactory connectionFactory = **new** ActiveMQConnectionFactory("tcp://192.168.25.168:61616");  // 第二步：使用ConnectionFactory对象创建一个Connection对象。  Connection connection = connectionFactory.createConnection();  // 第三步：开启连接，调用Connection对象的start方法。  connection.start();  // 第四步：使用Connection对象创建一个Session对象。  //第一个参数：是否开启事务。true：开启事务，第二个参数忽略。  //第二个参数：当第一个参数为false时，才有意义。消息的应答模式。1、自动应答2、手动应答。一般是自动应答。  Session session = connection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);  // 第五步：使用Session对象创建一个Destination对象（topic、queue），此处创建一个Queue对象。  //参数：队列的名称。  Queue queue = session.createQueue("test-queue");  // 第六步：使用Session对象创建一个Producer对象。  MessageProducer producer = session.createProducer(queue);  // 第七步：创建一个Message对象，创建一个TextMessage对象。  /\*TextMessage message = new ActiveMQTextMessage();  message.setText("hello activeMq,this is my first test.");\*/  TextMessage textMessage = session.createTextMessage("hello activeMq,this is my first test.");  // 第八步：使用Producer对象发送消息。  producer.send(textMessage);  // 第九步：关闭资源。  producer.close();  session.close();  connection.close();  } |

**Consumer**

消费者：接收消息。

第一步：创建一个ConnectionFactory对象。

第二步：从ConnectionFactory对象中获得一个Connection对象。

第三步：开启连接。调用Connection对象的start方法。

第四步：使用Connection对象创建一个Session对象。

第五步：使用Session对象创建一个Destination对象。和发送端保持一致queue，并且队列的名称一致。

第六步：使用Session对象创建一个Consumer对象。

第七步：接收消息。

第八步：打印消息。

第九步：关闭资源

|  |
| --- |
| @Test  **public** **void** testQueueConsumer() **throws** Exception {  // 第一步：创建一个ConnectionFactory对象。  ConnectionFactory connectionFactory = **new** ActiveMQConnectionFactory("tcp://192.168.25.168:61616");  // 第二步：从ConnectionFactory对象中获得一个Connection对象。  Connection connection = connectionFactory.createConnection();  // 第三步：开启连接。调用Connection对象的start方法。  connection.start();  // 第四步：使用Connection对象创建一个Session对象。  Session session = connection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);  // 第五步：使用Session对象创建一个Destination对象。和发送端保持一致queue，并且队列的名称一致。  Queue queue = session.createQueue("test-queue");  // 第六步：使用Session对象创建一个Consumer对象。  MessageConsumer consumer = session.createConsumer(queue);  // 第七步：接收消息。  consumer.setMessageListener(**new** MessageListener() {    @Override  **public** **void** onMessage(Message message) {  **try** {  TextMessage textMessage = (TextMessage) message;  String text = **null**;  //取消息的内容  text = textMessage.getText();  // 第八步：打印消息。  System.***out***.println(text);  } **catch** (JMSException e) {  e.printStackTrace();  }  }  });  //等待键盘输入  System.***in***.read();  // 第九步：关闭资源  consumer.close();  session.close();  connection.close();  } |

#### 5.2.6 Topic

**Producer**

使用步骤：

第一步：创建ConnectionFactory对象，需要指定服务端ip及端口号。

第二步：使用ConnectionFactory对象创建一个Connection对象。

第三步：开启连接，调用Connection对象的start方法。

第四步：使用Connection对象创建一个Session对象。

第五步：使用Session对象创建一个Destination对象（topic、queue），此处创建一个Topic对象。

第六步：使用Session对象创建一个Producer对象。

第七步：创建一个Message对象，创建一个TextMessage对象。

第八步：使用Producer对象发送消息。

第九步：关闭资源。

|  |
| --- |
| @Test  **public** **void** testTopicProducer() **throws** Exception {  // 第一步：创建ConnectionFactory对象，需要指定服务端ip及端口号。  // brokerURL服务器的ip及端口号  ConnectionFactory connectionFactory = **new** ActiveMQConnectionFactory("tcp://192.168.25.168:61616");  // 第二步：使用ConnectionFactory对象创建一个Connection对象。  Connection connection = connectionFactory.createConnection();  // 第三步：开启连接，调用Connection对象的start方法。  connection.start();  // 第四步：使用Connection对象创建一个Session对象。  // 第一个参数：是否开启事务。true：开启事务，第二个参数忽略。  // 第二个参数：当第一个参数为false时，才有意义。消息的应答模式。1、自动应答2、手动应答。一般是自动应答。  Session session = connection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);  // 第五步：使用Session对象创建一个Destination对象（topic、queue），此处创建一个topic对象。  // 参数：话题的名称。  Topic topic = session.createTopic("test-topic");  // 第六步：使用Session对象创建一个Producer对象。  MessageProducer producer = session.createProducer(topic);  // 第七步：创建一个Message对象，创建一个TextMessage对象。  /\*  \* TextMessage message = new ActiveMQTextMessage(); message.setText(  \* "hello activeMq,this is my first test.");  \*/  TextMessage textMessage = session.createTextMessage("hello activeMq,this is my topic test");  // 第八步：使用Producer对象发送消息。  producer.send(textMessage);  // 第九步：关闭资源。  producer.close();  session.close();  connection.close();  } |

**Consumer**

消费者：接收消息。

第一步：创建一个ConnectionFactory对象。

第二步：从ConnectionFactory对象中获得一个Connection对象。

第三步：开启连接。调用Connection对象的start方法。

第四步：使用Connection对象创建一个Session对象。

第五步：使用Session对象创建一个Destination对象。和发送端保持一致topic，并且话题的名称一致。

第六步：使用Session对象创建一个Consumer对象。

第七步：接收消息。

第八步：打印消息。

第九步：关闭资源

|  |
| --- |
| @Test  **public** **void** testTopicConsumer() **throws** Exception {  // 第一步：创建一个ConnectionFactory对象。  ConnectionFactory connectionFactory = **new** ActiveMQConnectionFactory("tcp://192.168.25.168:61616");  // 第二步：从ConnectionFactory对象中获得一个Connection对象。  Connection connection = connectionFactory.createConnection();  // 第三步：开启连接。调用Connection对象的start方法。  connection.start();  // 第四步：使用Connection对象创建一个Session对象。  Session session = connection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);  // 第五步：使用Session对象创建一个Destination对象。和发送端保持一致topic，并且话题的名称一致。  Topic topic = session.createTopic("test-topic");  // 第六步：使用Session对象创建一个Consumer对象。  MessageConsumer consumer = session.createConsumer(topic);  // 第七步：接收消息。  consumer.setMessageListener(**new** MessageListener() {  @Override  **public** **void** onMessage(Message message) {  **try** {  TextMessage textMessage = (TextMessage) message;  String text = **null**;  // 取消息的内容  text = textMessage.getText();  // 第八步：打印消息。  System.***out***.println(text);  } **catch** (JMSException e) {  e.printStackTrace();  }  }  });  System.***out***.println("topic的消费端03。。。。。");  // 等待键盘输入  System.***in***.read();  // 第九步：关闭资源  consumer.close();  session.close();  connection.close();  } |

#### 5.2.7 Activemq整合spring

**使用方法**

第一步：引用相关的jar包。

|  |
| --- |
| <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-jms</artifactId>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context-support</artifactId>  </dependency> |

第二步：配置Activemq整合spring。配置ConnectionFactory

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"* xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.2.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.2.xsd*  *http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.2.xsd*  *http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.2.xsd*  *http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-4.2.xsd"*>  <!-- 真正可以产生Connection的ConnectionFactory，由对应的 JMS服务厂商提供 -->  <bean id=*"targetConnectionFactory"* class=*"org.apache.activemq.ActiveMQConnectionFactory"*>  <property name=*"brokerURL"* value=*"tcp://192.168.25.168:61616"* />  </bean>  <!-- Spring用于管理真正的ConnectionFactory的ConnectionFactory -->  <bean id=*"connectionFactory"*  class=*"org.springframework.jms.connection.SingleConnectionFactory"*>  <!-- 目标ConnectionFactory对应真实的可以产生JMS Connection的ConnectionFactory -->  <property name=*"targetConnectionFactory"* ref=*"targetConnectionFactory"* />  </bean>  </beans> |

第三步：配置生产者。

使用JMSTemplate对象。发送消息。

第四步：在spring容器中配置Destination。

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"* xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.2.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.2.xsd*  *http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.2.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.2.xsd*  *http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-4.2.xsd"*>  <!-- 真正可以产生Connection的ConnectionFactory，由对应的 JMS服务厂商提供 -->  <bean id=*"targetConnectionFactory"* class=*"org.apache.activemq.ActiveMQConnectionFactory"*>  <property name=*"brokerURL"* value=*"tcp://192.168.25.168:61616"* />  </bean>  <!-- Spring用于管理真正的ConnectionFactory的ConnectionFactory -->  <bean id=*"connectionFactory"*  class=*"org.springframework.jms.connection.SingleConnectionFactory"*>  <!-- 目标ConnectionFactory对应真实的可以产生JMS Connection的ConnectionFactory -->  <property name=*"targetConnectionFactory"* ref=*"targetConnectionFactory"* />  </bean>  <!-- 配置生产者 -->  <!-- Spring提供的JMS工具类，它可以进行消息发送、接收等 -->  <bean id=*"jmsTemplate"* class=*"org.springframework.jms.core.JmsTemplate"*>  <!-- 这个connectionFactory对应的是我们定义的Spring提供的那个ConnectionFactory对象 -->  <property name=*"connectionFactory"* ref=*"connectionFactory"* />  </bean>  <!--这个是队列目的地，点对点的 -->  <bean id=*"queueDestination"* class=*"org.apache.activemq.command.ActiveMQQueue"*>  <constructor-arg>  <value>spring-queue</value>  </constructor-arg>  </bean>  <!--这个是主题目的地，一对多的 -->  <bean id=*"topicDestination"* class=*"org.apache.activemq.command.ActiveMQTopic"*>  <constructor-arg value=*"topic"* />  </bean>  </beans> |

第五步：代码测试

#### 5.2.8 代码测试

发送消息

第一步：初始化一个spring容器

第二步：从容器中获得JMSTemplate对象。

第三步：从容器中获得一个Destination对象

第四步：使用JMSTemplate对象发送消息，需要知道Destination

|  |
| --- |
| @Test  **public** **void** testQueueProducer() **throws** Exception {  // 第一步：初始化一个spring容器  ApplicationContext applicationContext = **new** ClassPathXmlApplicationContext("classpath:spring/applicationContext-activemq.xml");  // 第二步：从容器中获得JMSTemplate对象。  JmsTemplate jmsTemplate = applicationContext.getBean(JmsTemplate.**class**);  // 第三步：从容器中获得一个Destination对象  Queue queue = (Queue) applicationContext.getBean("queueDestination");  // 第四步：使用JMSTemplate对象发送消息，需要知道Destination  jmsTemplate.send(queue, **new** MessageCreator() {    @Override  **public** Message createMessage(Session session) **throws** JMSException {  TextMessage textMessage = session.createTextMessage("spring activemq test");  **return** textMessage;  }  });  } |

**接收消息**

Taotao-search-Service中接收消息。

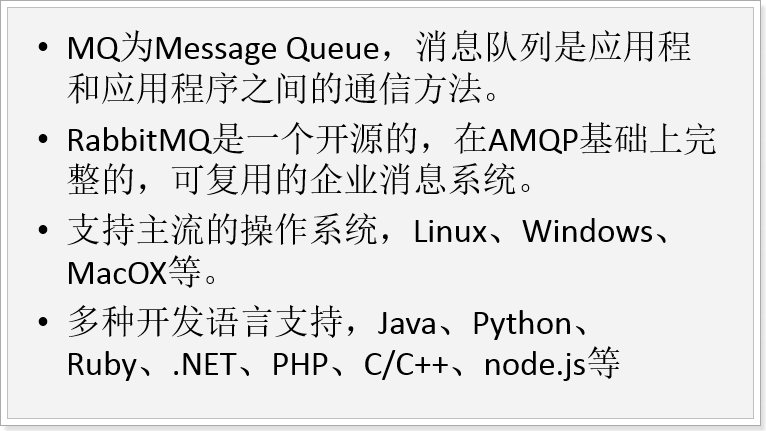
第一步：把Activemq相关的jar包添加到工程中

第二步：创建一个MessageListener的实现类。

|  |
| --- |
| **public** **class** MyMessageListener **implements** MessageListener {  @Override  **public** **void** onMessage(Message message) {    **try** {  TextMessage textMessage = (TextMessage) message;  //取消息内容  String text = textMessage.getText();  System.***out***.println(text);  } **catch** (JMSException e) {  e.printStackTrace();  }  }  } |

### 5.3 RabbitMQ

#### 5.3.1 RabbitMQ的简介



开发语言：Erlang – 面向并发的编程语言。

[MQ](https://baike.baidu.com/item/MQ" \t "https://baike.baidu.com/item/rabbitmq/_blank)是消费-生产者模型的一个典型的代表，一端往[消息队列](https://baike.baidu.com/item/%E6%B6%88%E6%81%AF%E9%98%9F%E5%88%97" \t "https://baike.baidu.com/item/rabbitmq/_blank)中不断写入消息，而另一端则可以读取或者订阅队列中的消息。MQ和[JMS](https://baike.baidu.com/item/JMS" \t "https://baike.baidu.com/item/rabbitmq/_blank)类似，但不同的是JMS是SUN JAVA[消息中间件](https://baike.baidu.com/item/%E6%B6%88%E6%81%AF%E4%B8%AD%E9%97%B4%E4%BB%B6" \t "https://baike.baidu.com/item/rabbitmq/_blank)服务的一个标准和API定义，而MQ则是遵循了AMQP协议的具体实现和产品。

#### 5.3.2 AMQP

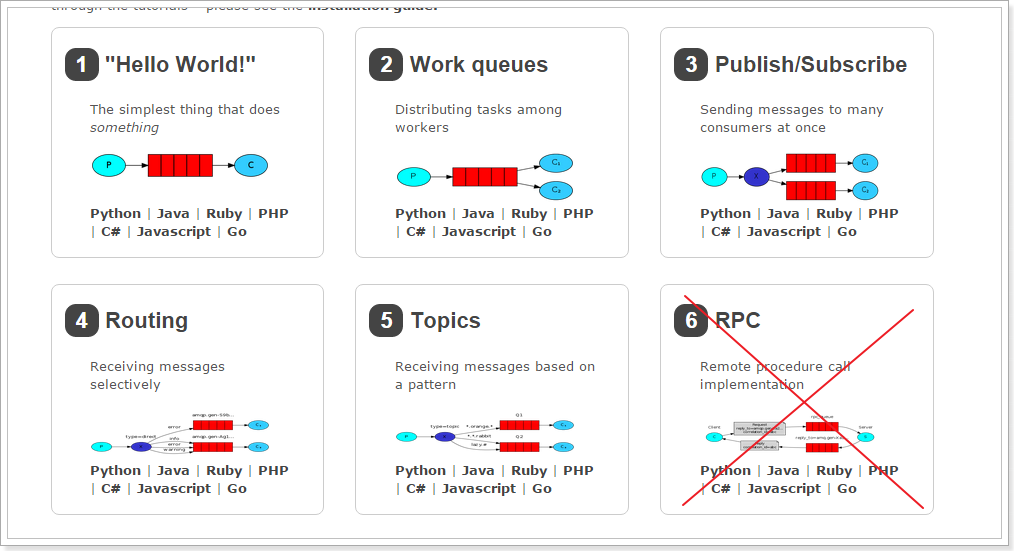
AMQP是消息队列的一个协议。



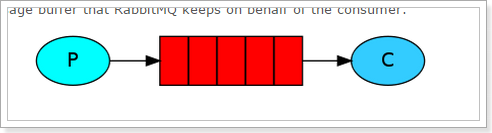
**MQ的其他产品**



#### 5.3.3 RabbitMQ 五种队列



##### 5.3.3.1 简单队列



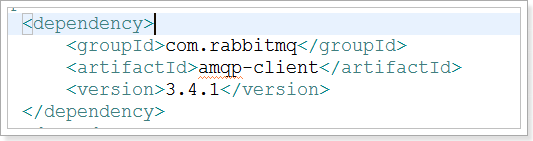
P：消息的生产者

C：消息的消费者

红色：队列

生产者将消息发送到队列，消费者从队列中获取消息。

导入RabbitMQ的客户端依赖



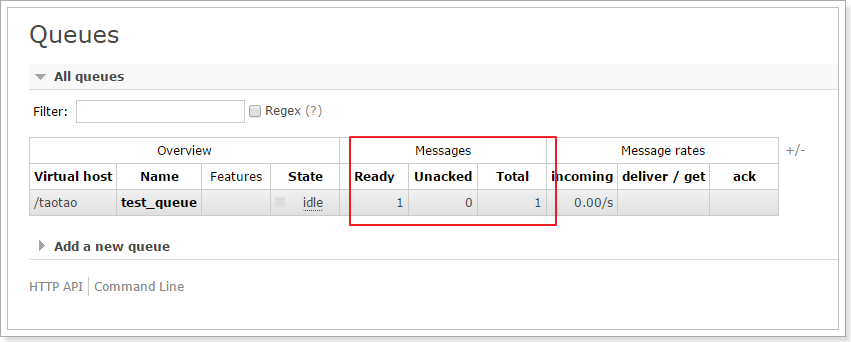
获取MQ的连接



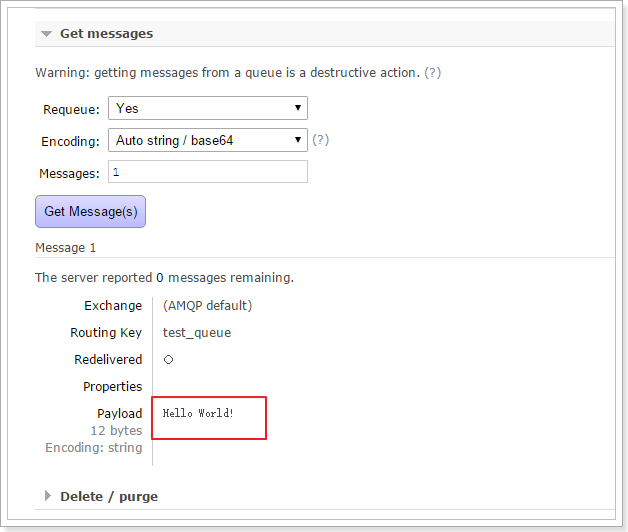
生产者发送消息到队列



管理工具中查看消息



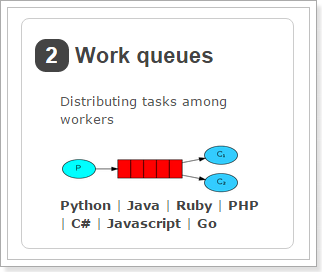
点击上面的队列名称，查询具体的队列中的信息：

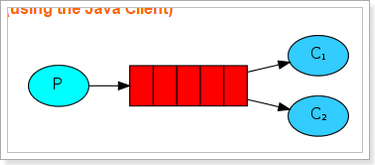


消费者从队列中获取消息



##### 5.3.3.2 Work模式





一个生产者、2个消费者。

一个消息只能被一个消费者获取。

消费者1

**public** **class** Recv {

**private** **final** **static** String ***QUEUE\_NAME*** = "test\_queue\_work";

**public** **static** **void** main(String[] argv) **throws** Exception {

// 获取到连接以及mq通道

Connection connection = ConnectionUtil.*getConnection*();

Channel channel = connection.createChannel();

// 声明队列

channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);

// 同一时刻服务器只会发一条消息给消费者

//channel.basicQos(1);

// 定义队列的消费者

QueueingConsumer consumer = **new** QueueingConsumer(channel);

// 监听队列，手动返回完成

channel.basicConsume(***QUEUE\_NAME***, **false**, consumer);

// 获取消息

**while** (**true**) {

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

String message = **new** String(delivery.getBody());

System.***out***.println(" [x] Received '" + message + "'");

//休眠

Thread.*sleep*(10);

// 返回确认状态

channel.basicAck(delivery.getEnvelope().getDeliveryTag(), **false**);

}

}

}

消费者2

**public** **class** Recv2 {

**private** **final** **static** String ***QUEUE\_NAME*** = "test\_queue\_work";

**public** **static** **void** main(String[] argv) **throws** Exception {

// 获取到连接以及mq通道

Connection connection = ConnectionUtil.*getConnection*();

Channel channel = connection.createChannel();

// 声明队列

channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);

// 同一时刻服务器只会发一条消息给消费者

//channel.basicQos(1);

// 定义队列的消费者

QueueingConsumer consumer = **new** QueueingConsumer(channel);

// 监听队列，手动返回完成状态

channel.basicConsume(***QUEUE\_NAME***, **false**, consumer);

// 获取消息

**while** (**true**) {

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

String message = **new** String(delivery.getBody());

System.***out***.println(" [x] Received '" + message + "'");

// 休眠1秒

Thread.*sleep*(1000);

channel.basicAck(delivery.getEnvelope().getDeliveryTag(), **false**);

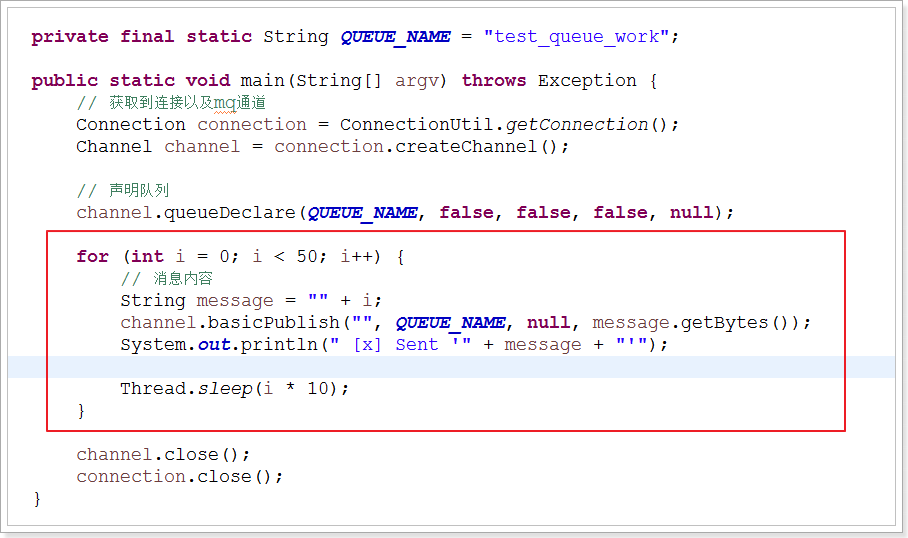
}

}

}

生产者

向队列中发送50条消息。



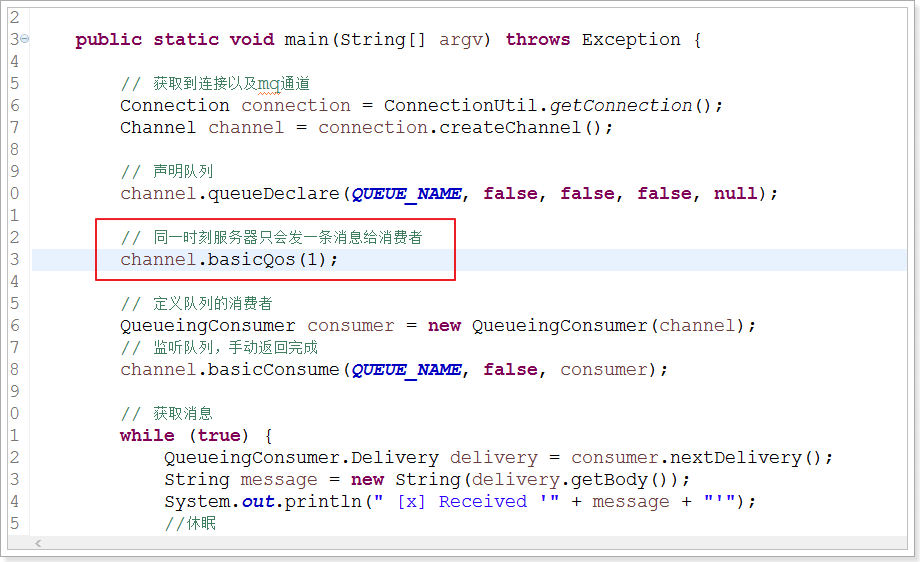
测试

测试结果：

1. 消费者1和消费者2获取到的消息内容是不同的，同一个消息只能被一个消费者获取。
2. 消费者1和消费者2获取到的消息的数量是相同的，一个是奇数一个是偶数。

其实，这样是不合理的，应该是消费者1要比消费者2获取到的消息多才对。

Work模式的“能者多劳”



测试：

消费者1比消费者2获取的消息更多。

消息的确认模式

消费者从队列中获取消息，服务端如何知道消息已经被消费呢？

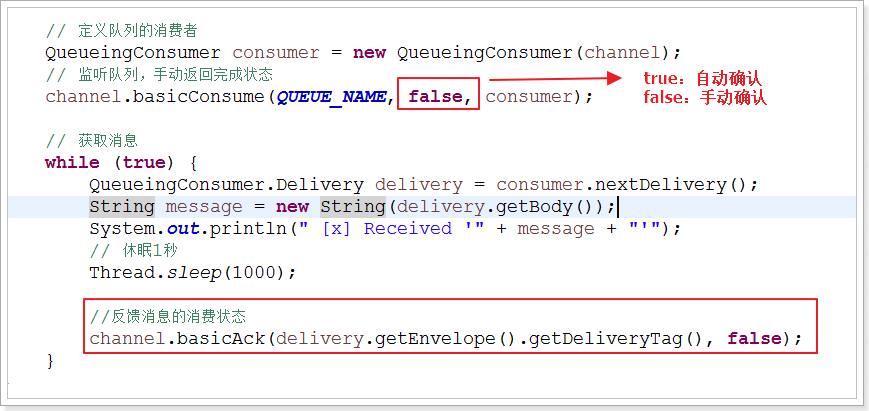
模式1：自动确认

只要消息从队列中获取，无论消费者获取到消息后是否成功消息，都认为是消息已经成功消费。

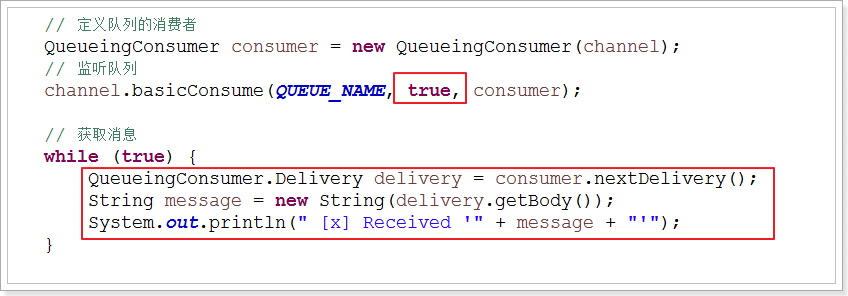
模式2：手动确认

消费者从队列中获取消息后，服务器会将该消息标记为不可用状态，等待消费者的反馈，如果消费者一直没有反馈，那么该消息将一直处于不可用状态。

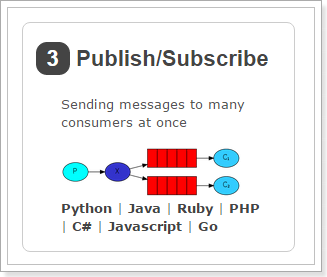
手动模式：



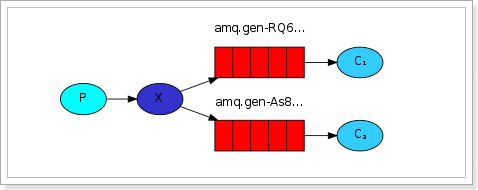
自动模式：



##### 5.3.3.3订阅模式



图示



解读：

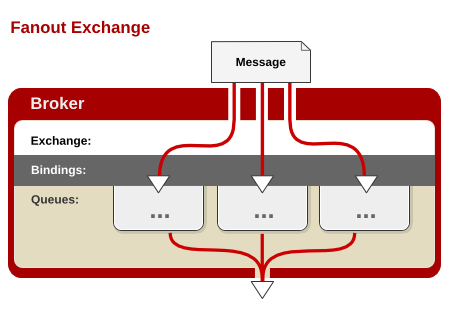
1、1个生产者，多个消费者

2、每一个消费者都有自己的一个队列

3、生产者没有将消息直接发送到队列，而是发送到了交换机

4、每个队列都要绑定到交换机

5、生产者发送的消息，经过交换机，到达队列，实现，一个消息被多个消费者获取的目的



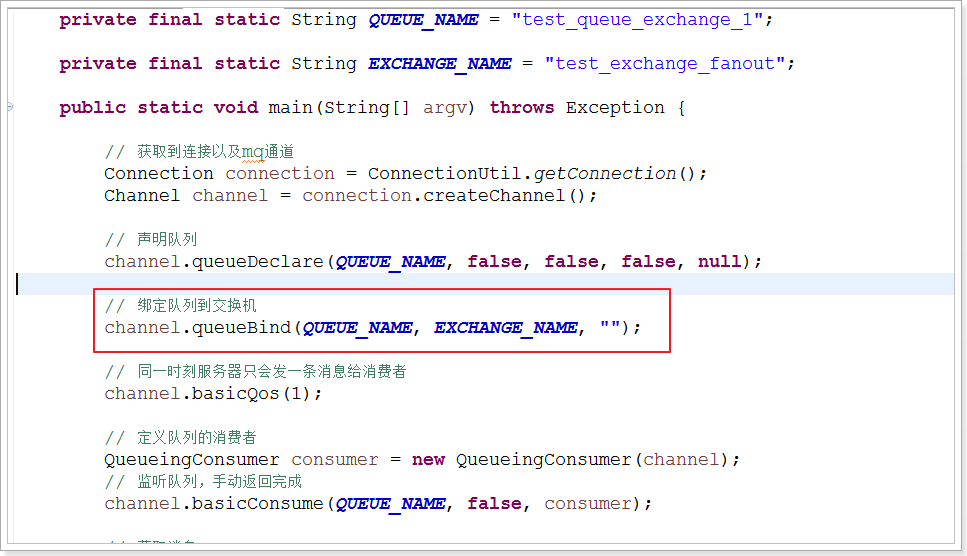
消息的生产者（看作是后台系统）

向交换机中发送消息。

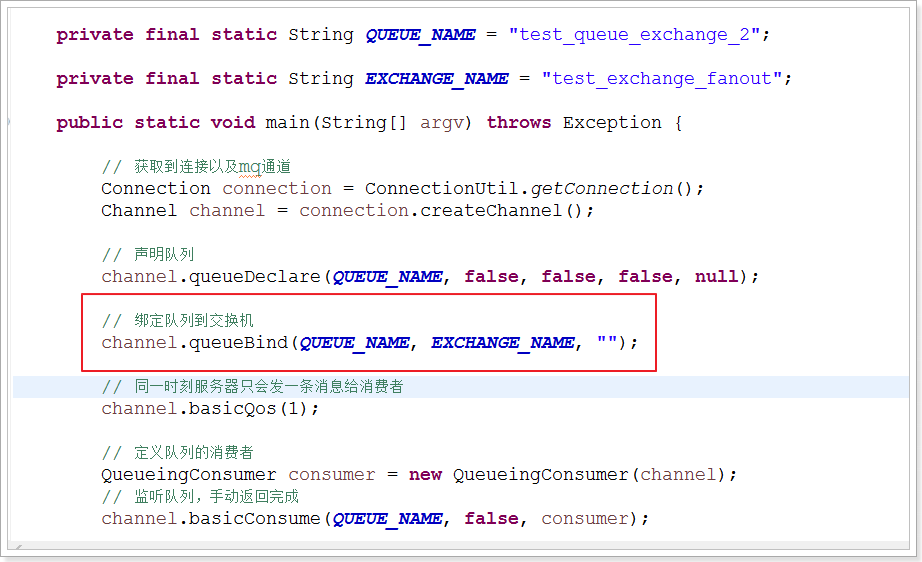


注意：消息发送到没有队列绑定的交换机时，消息将丢失，因为，交换机没有存储消息的能力，消息只能存在在队列中。

消费者1（看作是前台系统）



消费者2（看作是搜索系统）

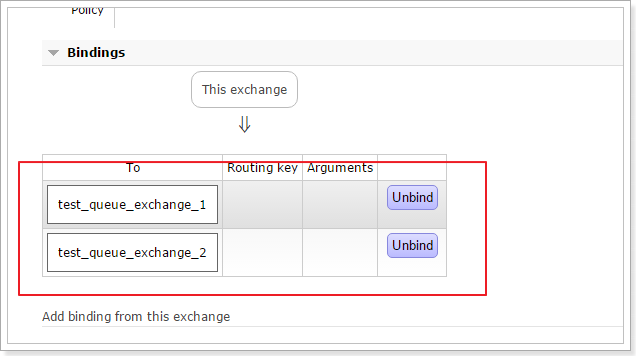


测试

测试结果：

同一个消息被多个消费者获取。

在管理工具中查看队列和交换机的绑定关系：



**使用订阅模式能否实现商品数据的同步？**

答案：可以的。

后台系统就是消息的生产者。

前台系统和搜索系统是消息的消费者。

后台系统将消息发送到交换机中，前台系统和搜索系统都创建自己的队列，然后将队列绑定到交换机，即可实现。

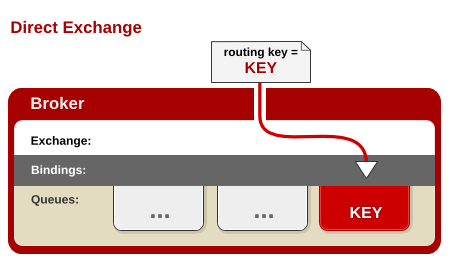
消息，新增商品、修改商品、删除商品。

前台系统：修改商品、删除商品。

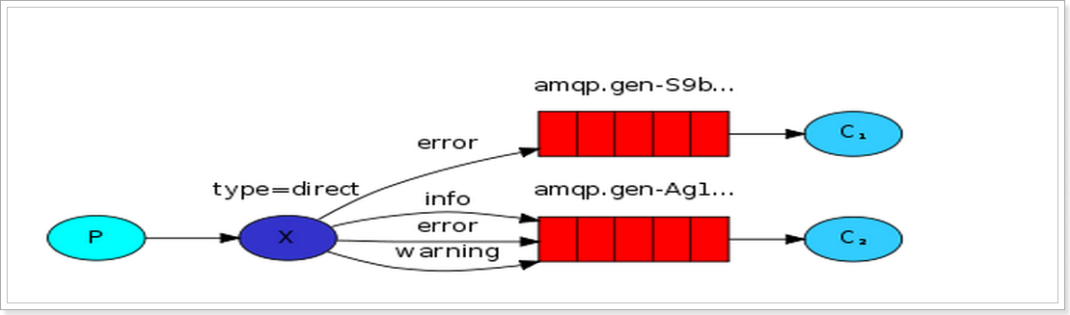
搜索系统：新增商品、修改商品、删除商品。

所以使用订阅模式实现商品数据的同步并不合理。

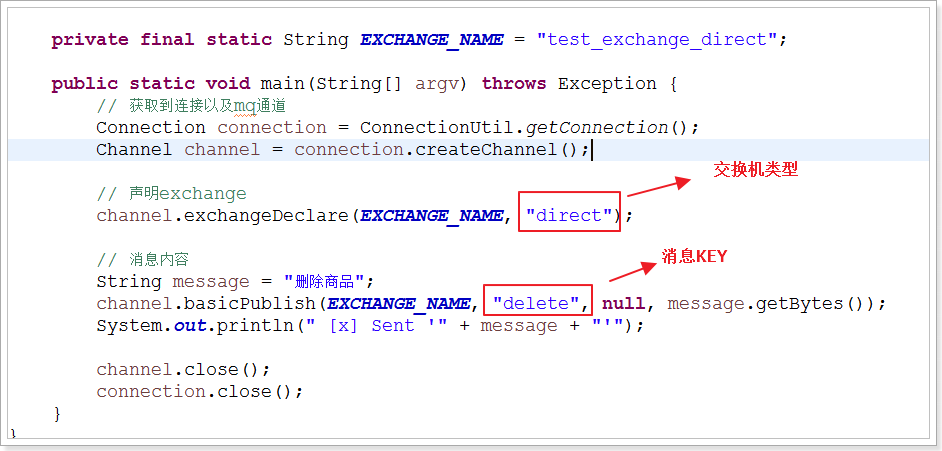
##### 5.3.3.4路由模式



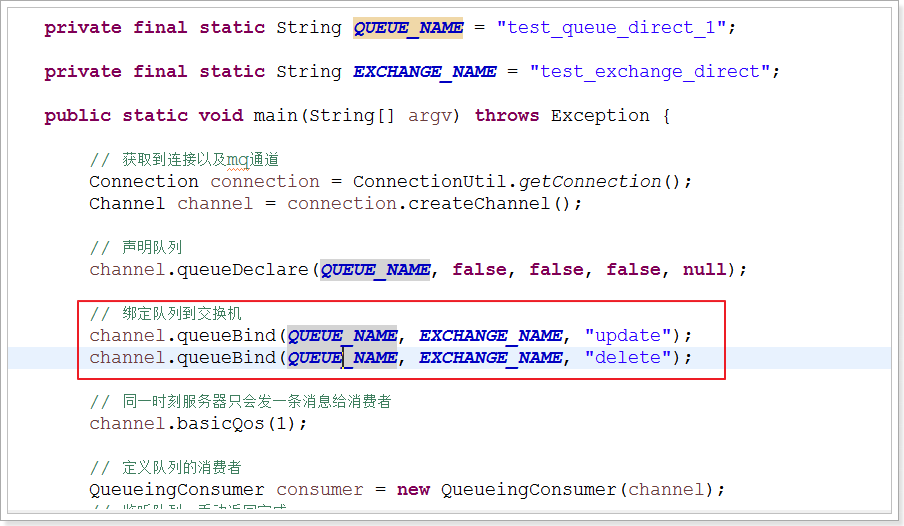
图示



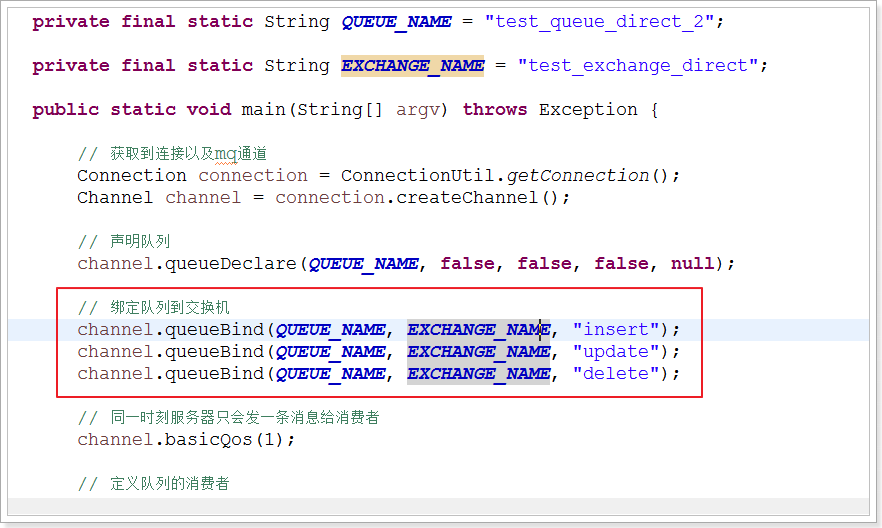
生产者



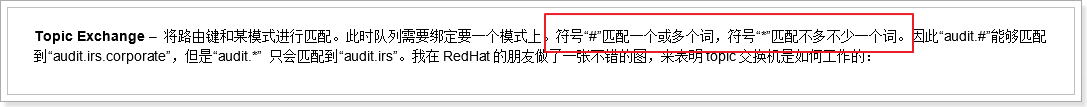
消费者1(前台系统)

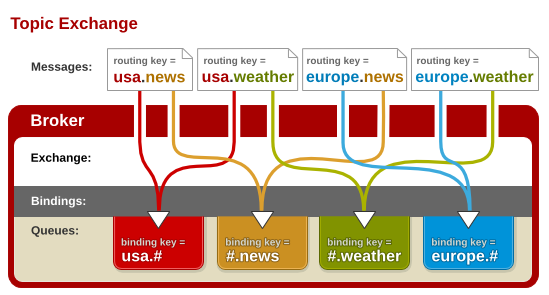


消费2（搜索系统）

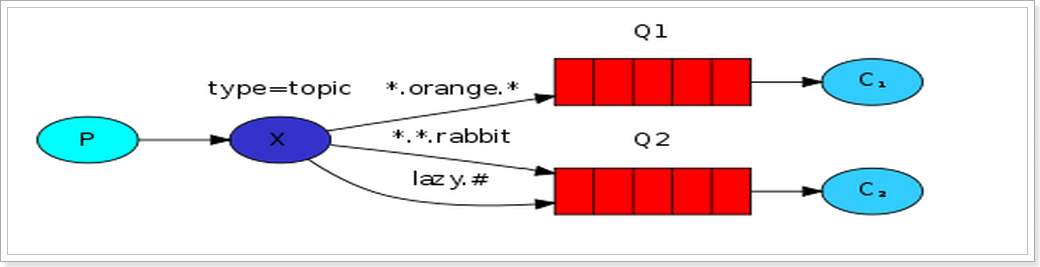


##### 5.3.3.5通配符模式

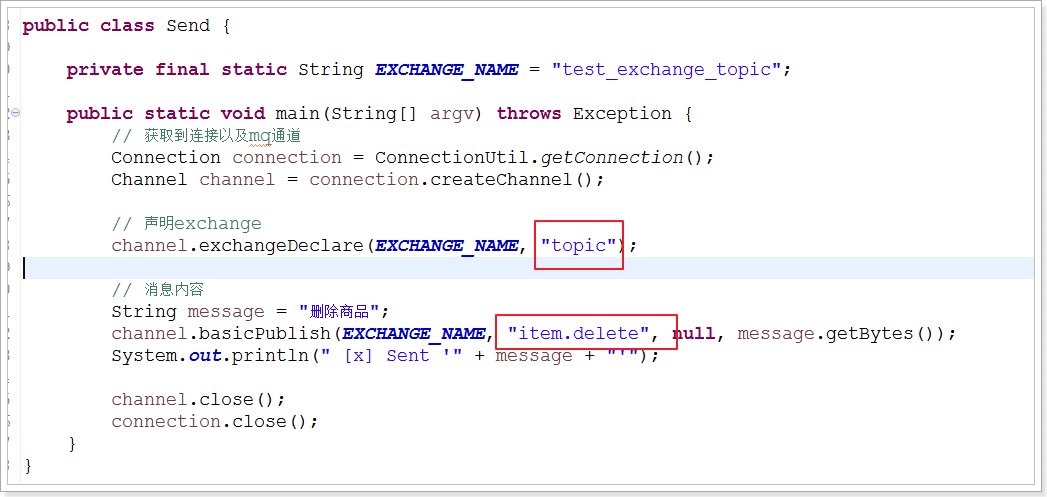




图示



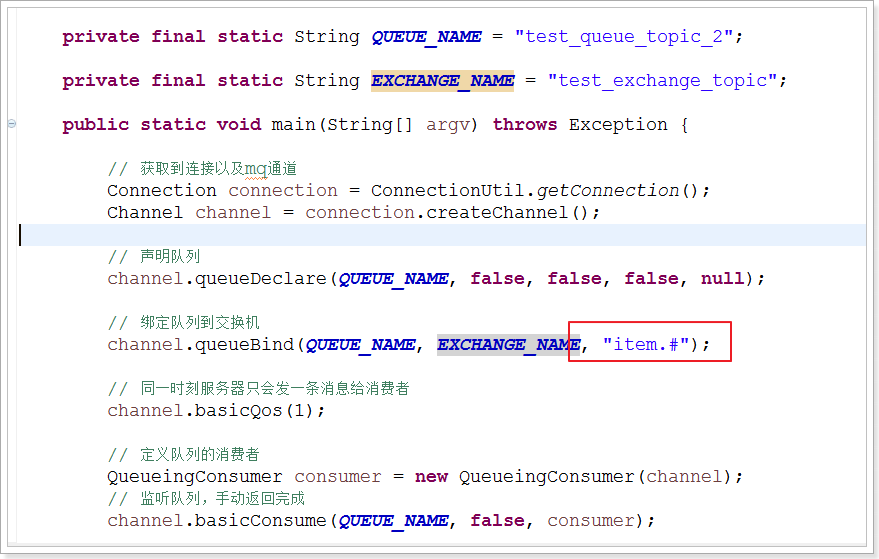
生产者



消费者1（前台系统）



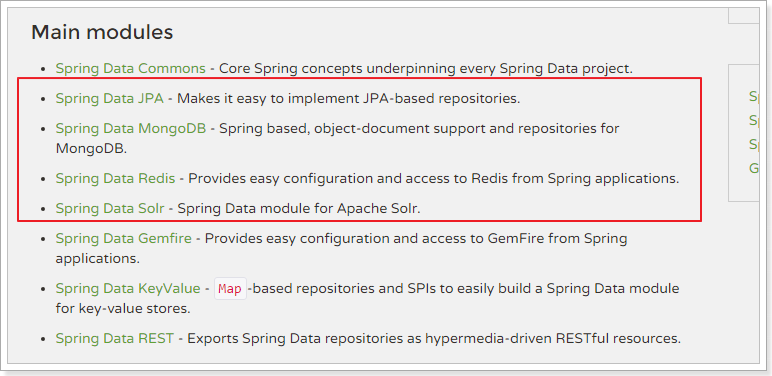
消费者2（搜索系统）



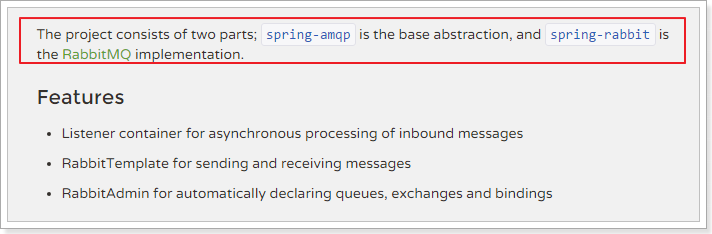
#### 5.3.4 Spring-Rabbit

##### 5.3.4.1 Spring项目

<http://spring.io/projects>



简介



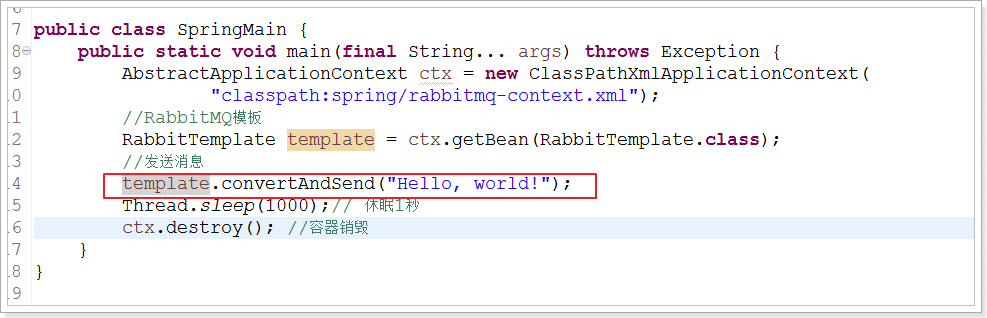


##### 5.3.4.2使用

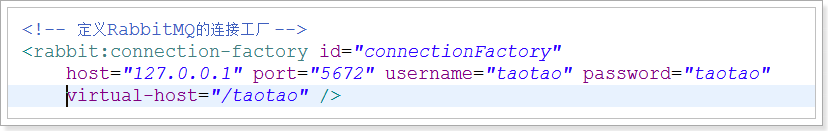
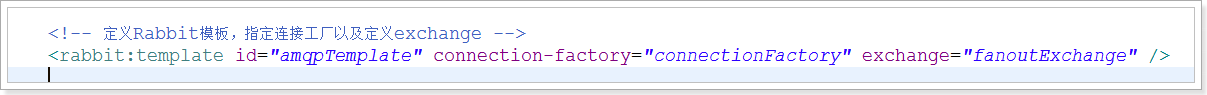
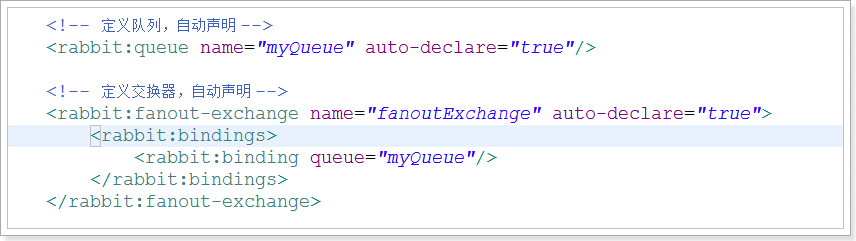
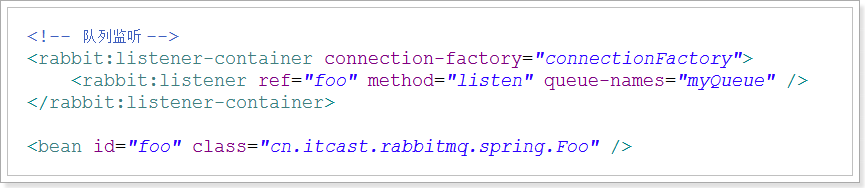
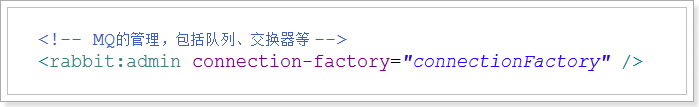
消费者



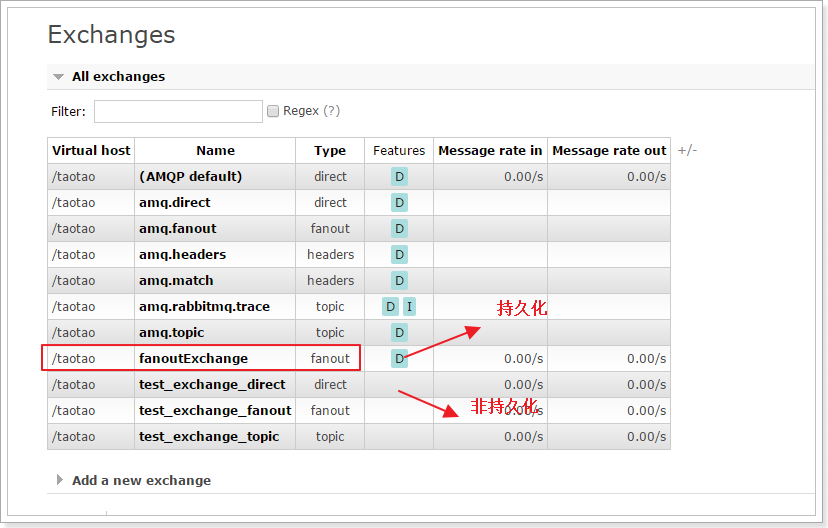
生产者



配置文件

1. 定义连接工厂  
   
2. 定义模板（可以指定交换机或队列）  
   
3. 定义队列、交换机、以及完成队列和交换机的绑定  
   
4. 定义监听  
   
5. 定义管理，用于管理队列、交换机等：  
   

持久化交换机和队列



持久化：将交换机或队列的数据保存到磁盘，服务器宕机或重启之后依然存在。

非持久化：将交换机或队列的数据保存到内存，服务器宕机或重启之后将不存在。

非持久化的性能高于持久化。

如何选择持久化？非持久化？ -- 看需求。

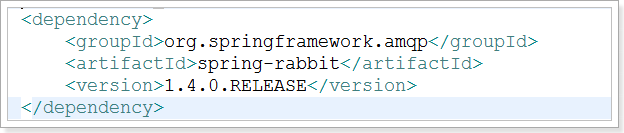
##### 5.3.4.3实现商品数据的同步

1)使用什么队列模式?

使用通配符模式。

2)在后台系统中发送消息到交换机

导入依赖



队列和交换机的绑定关系

实现：

1. 在配置文件中将队列和交换机完成绑定
2. 可以在管理界面中完成绑定
   1. 绑定关系如果发生变化，需要修改配置文件，并且服务需要重启
   2. 管理更加灵活
   3. 更容易对绑定关系的权限管理，流程管理

配置



消息内容

方案：

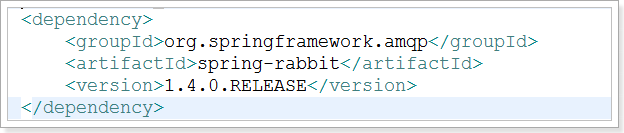
1. 将Item对象做json序列化发送
   1. 数据大
   2. 有些数据其他人是可能用不到的
2. 发送商品的id、操作类型

实现



**前台系统接收消息**

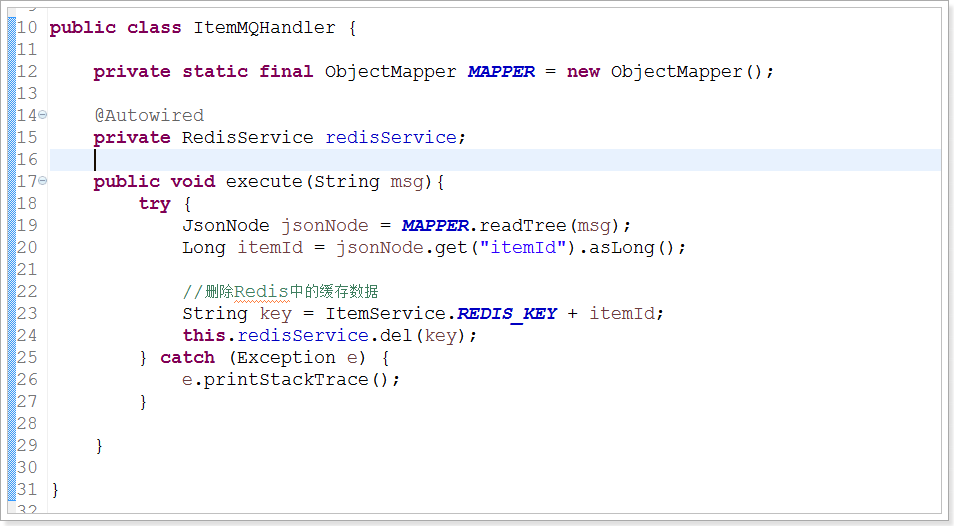
导入依赖



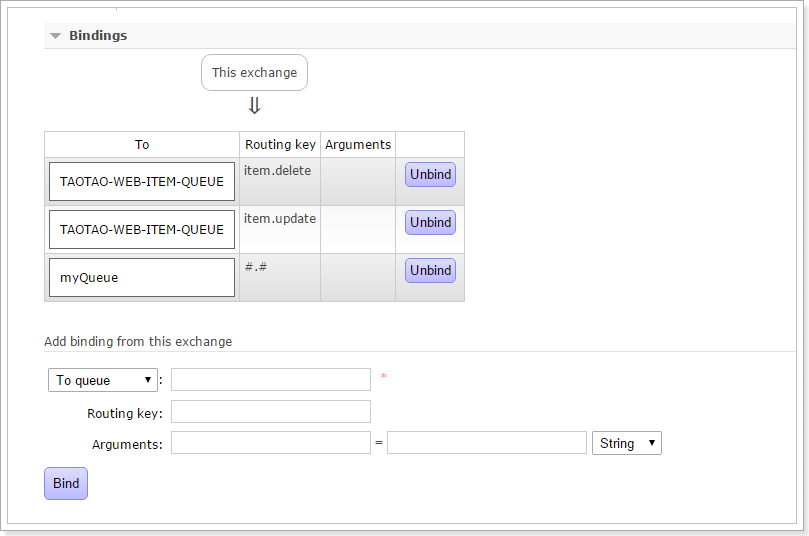
配置



具体处理逻辑



在界面管理工具中完成绑定关系



**搜索系统中接收消息**

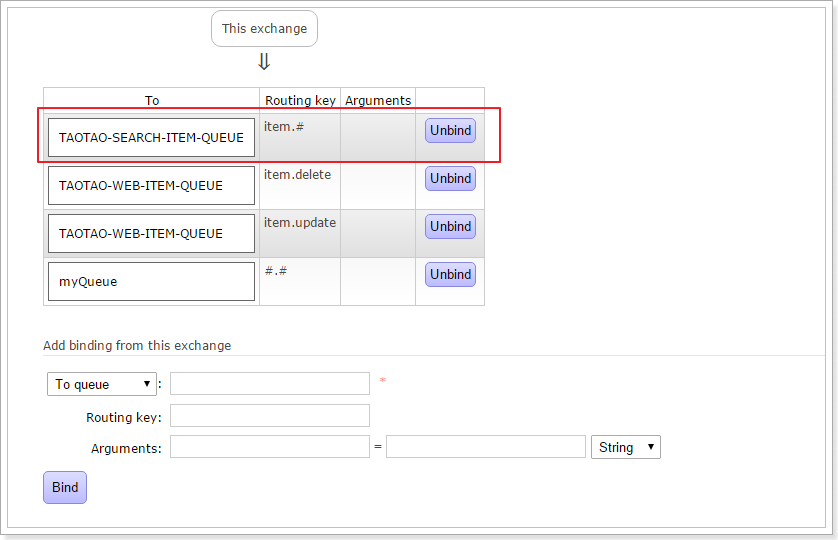
配置



处理业务逻辑



在管理工具中绑定队列和交换机



## 六.页面静态化方案

### 6.1摘要

   大型的网站,比如门户网站,在面对大量的用户访问,高并发请求方面,基本解决都是HTML静态化,图片服务器分离,数据库集群,负载均衡等方案!其中,HTML 静态化大大降低了大量的数据库访问请求,在面对高并发请求时有明显的作用!

   大家知道,效率高,消耗小的就是纯静态化的HTML 页面,所有我尽可能的是我们的网站上的页面采用静态页面来实现!于是便有了freemarker 这样的模板技术出现,大大提高了页面静态化的步骤!

   除了一些门户和消息发布类型的网站,对应交互性要求高的一些网站来说,即可能的静态化也是提高性能的必要手段,比如,将系统的首页,文章,社区帖子进行实时静态化,有更新时再重新静态化也是大量使用的策略,像网易新闻,凤凰新闻等!

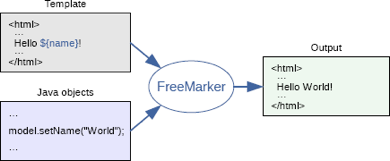
   同时,HTML 静态化也是某些缓存策略使用的手段,对于系统中频繁使用数据查询但是内容更新很小的应用,可以考虑使用Freemarker 将HTML 静态化,比如一些网站的共用设置信息,这些信息基本都是通过后台管理并存储在数据库中,这些信息其实大量的被前台程序调用,每次调用都会查询一次数据库,但是这些信息更新频率有会很小,因此可以考虑将这部分内容进行后台更新的时候静态化,这样避免大量数据库访问请求!

### 6.2 Freemarker 介绍

#### 6.2.1 原理

FreeMarker 是一款 模板引擎： 即一种基于模板和要改变的数据， 并用来生成输出文本(HTML网页，电子邮件，配置文件，源代码等)的通用工具。 它不是面向最终用户的，而是一个Java类库，是一款程序员可以嵌入他们所开发产品的组件。

**其原理:数据模型 + 模板**



说白点,就是可以根据提供的数据和创建好的模板,去自动的创建html静态页面.

放在电商项目中的话,比如说商品详情页面,为了减少数据库的压力,提高用户体验度,我们想到了如果直接把商品详情页面准备好,那么久不需要在去数据库查找了,节约了大量的资源.

freemarker就是起到了这么一个作用,每增加一个商品就自动生成一个静态页面,保存到web层中.

#### 6.2.2基本简单应用

  第一步:创建ftl 模板  hello.ftl --> 内容 为${hello}

  第二步：创建一个Configuration对象，直接new一个对象。构造方法的参数就是freemarker对于的版本号。

  第三步：设置模板文件所在的路径。

  第四步：设置模板文件使用的字符集。一般就是utf-8.

  第五步：加载一个模板，创建一个模板对象。

  第六步：创建一个模板使用的数据集，可以是pojo也可以是map。一般是Map。

  第七步：创建一个Writer对象，一般创建一FileWriter对象，指定生成的文件名。

  第八步：调用模板对象的process方法输出文件。

   第九步：关闭流。

public void genFile() throws Exception {

// 第一步：创建一个Configuration对象，直接new一个对象。构造方法的参数就是freemarker对于的版本号。

Configuration configuration = new Configuration(Configuration.getVersion());

// 第二步：设置模板文件所在的路径。

configuration.setDirectoryForTemplateLoading(new File("D:/workspaces-itcast/term197/taotao-item-web/src/main/webapp/WEB-INF/ftl"));

// 第三步：设置模板文件使用的字符集。一般就是utf-8.

configuration.setDefaultEncoding("utf-8");

// 第四步：加载一个模板，创建一个模板对象。

Template template = configuration.getTemplate("hello.ftl");

// 第五步：创建一个模板使用的数据集，可以是pojo也可以是map。一般是Map。

Map dataModel = new HashMap<>();

//向数据集中添加数据

dataModel.put("hello", "this is my first freemarker test.");

// 第六步：创建一个Writer对象，一般创建一FileWriter对象，指定生成的文件名。

Writer out = new FileWriter(new File("D:/temp/term197/out/hello.html"));

// 第七步：调用模板对象的process方法输出文件。 template.process(dataModel, out);

// 第八步：关闭流。 out.close();

}

#### 6.2.3  基本语法

       6.2.3.1 访问map中的key

               使用 ${key}

   6.2.3.2 访问pojo中的属性

Student对象。学号、姓名、年龄 使用${key.property}



       6.2.3.3 集合中的数据

 <#list studentList as student> ${student.id}/${studnet.name} </#list>

      循环使用格式:  <#list 要循环的数据 as 循环后的数据>   </#list>



   2.3.4 取循环中的下标

          <#list studentList as student> ${student\_index} </#list>

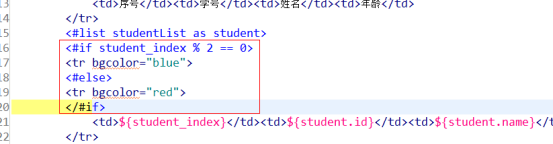


    2.3.5  判断

<#if student\_index % 2 == 0>

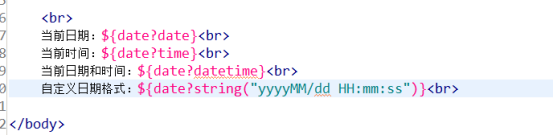
<#else>

</#if>



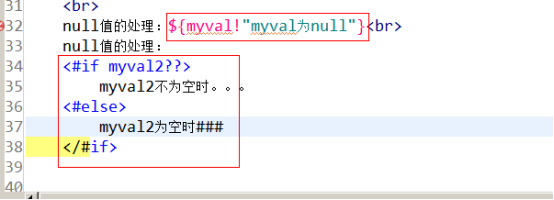
   2.3.6 日期类型格式化

        直接取值:${date}(date是属性名)如果传来的是一个Date型数据会报错 ${date?date} -- > 2016-9-13 ${date?time} --> 17:53:55 ${date?datetime} -->2016-9-13 17:53:55



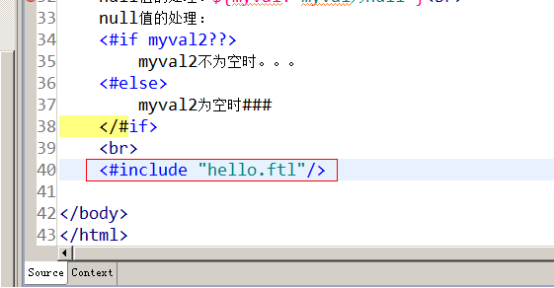
     2.3.7 Null值的处理

      如果直接取一个不存在的值(值为null)时会报异常 ${aaa} 处理: ${aaa!”默认值”}或者${aaa! }代表空字符串



     2.3.8 Include标签

         <#include “模板名称”> (相当于jstl中的包含)



#### 6.2.4 Freemarker整合spring

<!-- 页面静态化 freemarker-->

<dependency>

<groupId>org.freemarker</groupId>

<artifactId>freemarker</artifactId>

<version>2.3.28</version>

</dependency>

<!-- spring 的整合文件 -->

<?xml version="1.0" encoding="UTF-8"?><beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:dubbo="http://code.alibabatech.com/schema/dubbo" xmlns:mvc="http://www.springframework.org/schema/mvc"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-4.2.xsd

http://code.alibabatech.com/schema/dubbo http://code.alibabatech.com/schema/dubbo/dubbo.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">

<bean id="freemarkerConfig"

class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">

<property name="templateLoaderPath" value="/WEB-INF/ftl/" />

<property name="defaultEncoding" value="UTF-8" />

</bean>

</beans>

### 6.3 页面静态化

    在电商项目中,商品详情页面 为了减少数据库的压力,提高用户的体验度,我们可以提前将商品的详情页面准备好,就是在没创建一个商品时就自动生成一个商品的静态页面中!

   具体实现如下架构图:

      ①后台管理系统 添加商品后,想 Activemq 消息队列中添加消息;

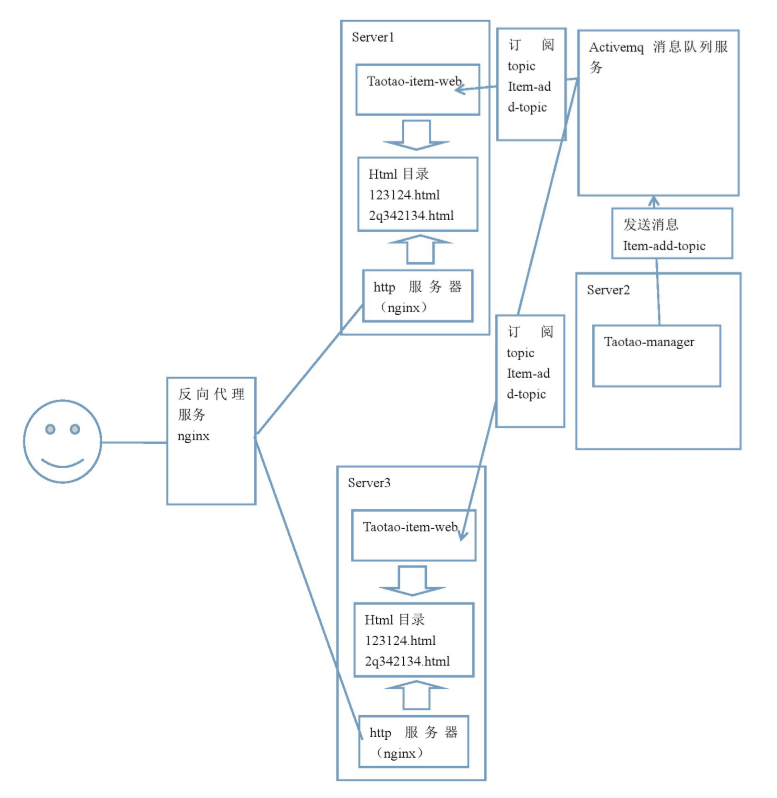
      ② Activemq 消息队列就会把这个消息推送到各个订阅端;

      ③商品详情对应的服务端接收到订阅消息后,根据消息中有效内容,使用Freemarker 技术,自动的为新建的商品创建静态html 页面,然后将html 页面存放到特定的位置;

      ④ 客户端访问该html 页面时,通过nginx ,http 服务访问即可!

      ⑤ 为了提高页面访问高可用,可以添加多态商品详情服务端,然后订阅消息,创建html 页面, 然后通过nginx http服务器,提供html 页面访问服务!

      ⑥ 当有多个 http 服务器提供html 页面访问服务的时候,需要在这些http 服务器前添加一个nginx 反向代理服务器!



## 七.Nginx

### 7.1什么是nginx

Nginx是一款高性能的http 服务器/反向代理服务器及电子邮件（IMAP/POP3）代理服务器。由俄罗斯的程序设计师Igor Sysoev所开发，官方测试nginx能够支支撑5万并发链接，并且cpu、内存等资源消耗却非常低，运行非常稳定。

### 7.2应用场景

1. http服务器。Nginx是一个http服务可以独立提供http服务。可以做网页静态服务器。
2. 虚拟主机。可以实现在一台服务器虚拟出多个网站。例如个人网站使用的虚拟主机。
3. 反向代理，负载均衡。当网站的访问量达到一定程度后，单台服务器不能满足用户的请求时，需要用多台服务器集群可以使用nginx做反向代理。并且多台服务器可以平均分担负载，不会因为某台服务器负载高宕机而某台服务器闲置的情况。

### 7.3nginx安装

下载nginx：

官方网站：

<http://nginx.org/>

#### 7.3.1要求的安装环境

1. 需要安装gcc的环境。yum install gcc-c++
2. 第三方的开发包。

* PCRE

PCRE(Perl Compatible Regular Expressions)是一个Perl库，包括 perl 兼容的正则表达式库。nginx的http模块使用pcre来解析正则表达式，所以需要在linux上安装pcre库。

**yum install -y pcre pcre-devel**

注：pcre-devel是使用pcre开发的一个二次开发库。nginx也需要此库。

* zlib

zlib库提供了很多种压缩和解压缩的方式，nginx使用zlib对http包的内容进行gzip，所以需要在linux上安装zlib库。

**yum install -y zlib zlib-devel**

* openssl

OpenSSL 是一个强大的安全套接字层密码库，囊括主要的密码算法、常用的密钥和证书封装管理功能及SSL协议，并提供丰富的应用程序供测试或其它目的使用。

nginx不仅支持http协议，还支持https（即在ssl协议上传输http），所以需要在linux安装openssl库。

**yum install -y openssl openssl-devel**

#### 7.3.2安装步骤

第一步：把nginx的源码包上传到linux系统

第二步：解压缩

[root@localhost ~]# tar zxf nginx-1.8.0.tar.gz

第三步：使用configure命令创建一makeFile文件。

./configure \

--prefix=/usr/local/nginx \

--pid-path=/var/run/nginx/nginx.pid \

--lock-path=/var/lock/nginx.lock \

--error-log-path=/var/log/nginx/error.log \

--http-log-path=/var/log/nginx/access.log \

--with-http\_gzip\_static\_module \

--http-client-body-temp-path=/var/temp/nginx/client \

--http-proxy-temp-path=/var/temp/nginx/proxy \

--http-fastcgi-temp-path=/var/temp/nginx/fastcgi \

--http-uwsgi-temp-path=/var/temp/nginx/uwsgi \

--http-scgi-temp-path=/var/temp/nginx/scgi

**注意：启动nginx之前，上边将临时文件目录指定为/var/temp/nginx，需要在/var下创建temp及nginx目录**

[root@localhost sbin]# mkdir /var/temp/nginx/client -p

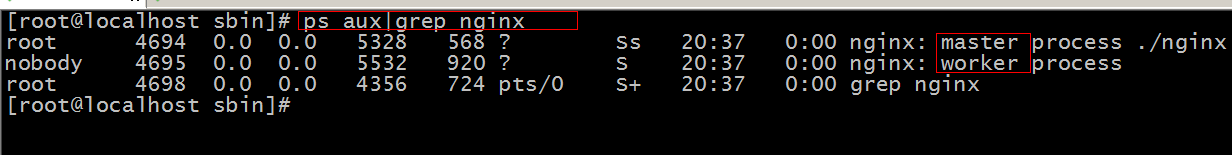
第四步：make

第五步：make install



**启动nginx**

进入sbin目录

[root@localhost sbin]# ./nginx   


关闭nginx：

[root@localhost sbin]# ./nginx -s stop

推荐使用：

[root@localhost sbin]# ./nginx -s quit

重启nginx：

1. 先关闭后启动。
2. 刷新配置文件：

[root@localhost sbin]# ./nginx -s reload

**访问nginx**



默认是80端口。

注意：是否关闭防火墙。

### 7.4配置虚拟主机

就是在一台服务器启动多个网站。

如何区分不同的网站：

1. 域名不同
2. 端口不同

#### 7.4.1通过端口区分不同虚拟机

Nginx的配置文件：

/usr/local/nginx/conf/nginx.conf

|  |
| --- |
| #user nobody;  worker\_processes 1;  #error\_log logs/error.log;  #error\_log logs/error.log notice;  #error\_log logs/error.log info;  #pid logs/nginx.pid;  events {  worker\_connections 1024;  }  http {  include mime.types;  default\_type application/octet-stream;  #log\_format main '$remote\_addr - $remote\_user [$time\_local] "$request" '  # '$status $body\_bytes\_sent "$http\_referer" '  # '"$http\_user\_agent" "$http\_x\_forwarded\_for"';  #access\_log logs/access.log main;  sendfile on;  #tcp\_nopush on;  #keepalive\_timeout 0;  keepalive\_timeout 65;  #gzip on;  一个server节点就是一个虚拟主机  server {  listen 80;  server\_name localhost;  #charset koi8-r;  #access\_log logs/host.access.log main;  Html是nginx安装目录下的html目录(静态页面存放的位置,也可以使用绝对路径)  location / {  root html;  index index.html index.htm;  }  }  } |

可以配置多个server，配置了多个虚拟主机。

添加虚拟主机：

|  |
| --- |
| #user nobody;  worker\_processes 1;  #error\_log logs/error.log;  #error\_log logs/error.log notice;  #error\_log logs/error.log info;  #pid logs/nginx.pid;  events {  worker\_connections 1024;  }  http {  include mime.types;  default\_type application/octet-stream;  #log\_format main '$remote\_addr - $remote\_user [$time\_local] "$request" '  # '$status $body\_bytes\_sent "$http\_referer" '  # '"$http\_user\_agent" "$http\_x\_forwarded\_for"';  #access\_log logs/access.log main;  sendfile on;  #tcp\_nopush on;  #keepalive\_timeout 0;  keepalive\_timeout 65;  #gzip on;  server {  listen 80;  server\_name localhost;(域名)  #charset koi8-r;  #access\_log logs/host.access.log main;  Location / {  root html;  index index.html index.htm;  }  }  server {  listen 81;  server\_name localhost;  #charset koi8-r;  #access\_log logs/host.access.log main;  location / {  root html-81;  index index.html index.htm;  }  }  } |

重新加载配置文件

[root@localhost nginx]# sbin/nginx -s reload

#### 7.4.2通过域名区分虚拟主机

**什么是域名**

域名就是网站。

[www.baidu.com](http://www.baidu.com)

[www.taobao.com](http://www.taobao.com)

[www.jd.com](http://www.jd.com)

Tcp/ip

Dns服务器：把域名解析为ip地址。保存的就是域名和ip的映射关系。

一级域名：

Baidu.com

Taobao.com

Jd.com

二级域名：

[www.baidu.com](http://www.baidu.com)

Image.baidu.com

Item.baidu.com

三级域名：

1. Image.baidu.com

Aaa.image.baidu.com

一个域名对应一个ip地址，一个ip地址可以被多个域名绑定。

**Nginx的配置**

|  |
| --- |
| #user nobody;  worker\_processes 1;  #error\_log logs/error.log;  #error\_log logs/error.log notice;  #error\_log logs/error.log info;  #pid logs/nginx.pid;  events {  worker\_connections 1024;  }  http {  include mime.types;  default\_type application/octet-stream;  #log\_format main '$remote\_addr - $remote\_user [$time\_local] "$request" '  # '$status $body\_bytes\_sent "$http\_referer" '  # '"$http\_user\_agent" "$http\_x\_forwarded\_for"';  #access\_log logs/access.log main;  sendfile on;  #tcp\_nopush on;  #keepalive\_timeout 0;  keepalive\_timeout 65;  #gzip on;  server {  listen 80;  server\_name localhost;  #charset koi8-r;  #access\_log logs/host.access.log main;  location / {  root html;  index index.html index.htm;  }  }  server {  listen 81;  server\_name localhost;  #charset koi8-r;  #access\_log logs/host.access.log main;  location / {  root html-81;  index index.html index.htm;  }  }  server {  listen 80;  server\_name www.xx.com;  #charset koi8-r;  #access\_log logs/host.access.log main;  location / {  root html-taobao;  index index.html index.htm;  }  }  server {  listen 80;  server\_name www.baidu.com;  #charset koi8-r;  #access\_log logs/host.access.log main;  location / {  root html-baidu;  index index.html index.htm;  }  }  } |

域名的配置：

192.168.25.148 www.taobao.com

192.168.25.148 www.baidu.com

### 7.5反向代理

#### 7.5.1什么是反向代理

正向代理

局域网

代理服务器

可以上网

请求转发

Pc3

Pc2

Pc1

Internet

网站

反向代理：

internet

Tomcatn

网站入口

公网ip

反向代理服务器

nginx

Tomcat2

Tomcat1

反向代理服务器决定哪台服务器提供服务。

返回代理服务器不提供服务器。也是请求的转发。

#### 7.5.2Nginx实现反向代理

两个域名指向同一台nginx服务器，用户访问不同的域名显示不同的网页内容。

两个域名是www.sian.com.cn和www.sohu.com

nginx服务器使用虚拟机192.168.101.3

[www.sina.com.cn](http://www.sina.com.cn)

Tomcat:8080

反向代理服务器

Nginx

192.168.25.148

www.sohu.com

Tomcat:8081

第一步：安装两个tomcat，分别运行在8080和8081端口。

第二步：启动两个tomcat。

第三步：反向代理服务器的配置

复制一个server节点,将root改成proxy\_pass

配置upstream

|  |
| --- |
| upstream tomcat1 {  server 192.168.25.148:8080;  }  server {  listen 80;  server\_name www.sina.com.cn;  #charset koi8-r;  #access\_log logs/host.access.log main;  location / {  proxy\_pass http://tomcat1;  index index.html index.htm;  }  }  upstream tomcat2 {  server 192.168.25.148:8081;  }  server {  listen 80;  server\_name www.sohu.com;  #charset koi8-r;  #access\_log logs/host.access.log main;  location / {  proxy\_pass http://tomcat2;  index index.html index.htm;  }  } |

第四步：nginx重新加载配置文件

第五步：配置域名

在hosts文件中添加域名和ip的映射关系

192.168.25.148 www.sina.com.cn

192.168.25.148 www.sohu.com

### 7.5.4负载均衡

如果一个服务由多条服务器提供，需要把负载分配到不同的服务器处理，需要负载均衡。

upstream tomcat2 {

server 192.168.25.148:8081;

server 192.168.25.148:8082;

}

可以根据服务器的实际情况调整服务器权重。权重越高分配的请求越多，权重越低，请求越少。默认是都是1

|  |
| --- |
| upstream tomcat2 {  server 192.168.25.148:8081;  server 192.168.25.148:8082 weight=2;  } |

### 7.5.5Nginx的高可用（了解）

要实现nginx的高可用，需要实现备份机。

什么是负载均衡高可用

nginx作为负载均衡器，所有请求都到了nginx，可见nginx处于非常重点的位置，如果nginx服务器宕机后端web服务将无法提供服务，影响严重。

为了屏蔽负载均衡服务器的宕机，需要建立一个备份机。主服务器和备份机上都运行高可用（High Availability）监控程序，通过传送诸如“I am alive”这样的信息来监控对方的运行状况。当备份机不能在一定的时间内收到这样的信息时，它就接管主服务器的服务IP并继续提供负载均衡服务；当备份管理器又从主管理器收到“I am alive”这样的信息时，它就释放服务IP地址，这样的主服务器就开始再次提供负载均衡服务。

## 八.单点登录 sso

### 8.1 什么是sso系统

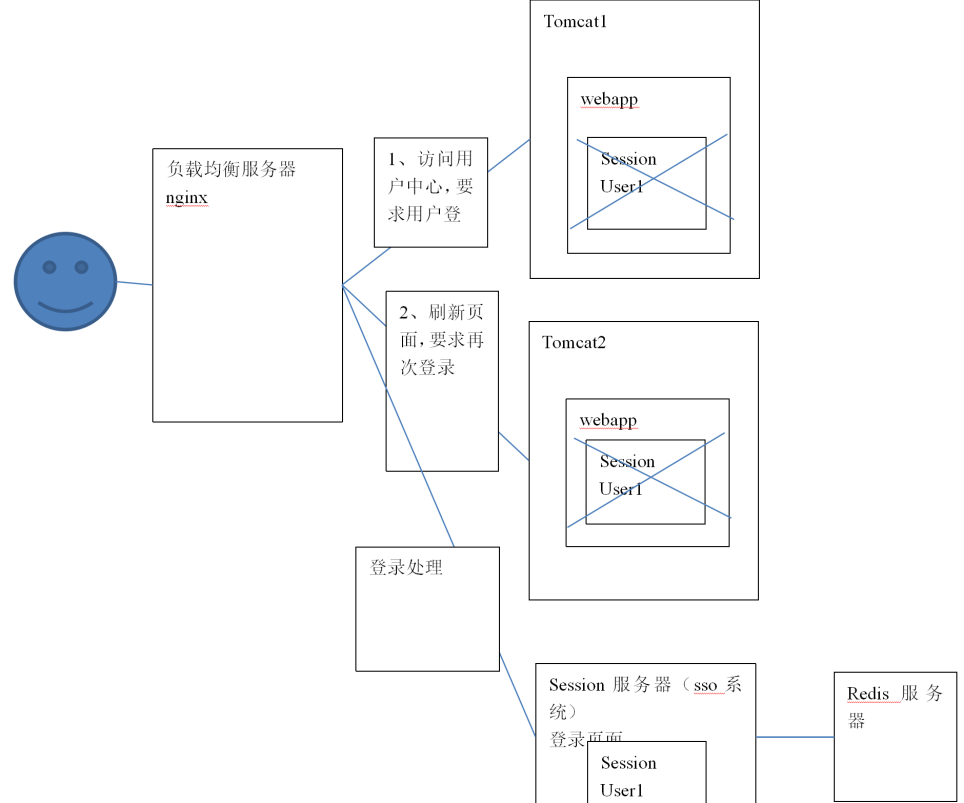
SSO英文全称Single Sign On，单点登录。SSO是在多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统。它包括可以将这次主要的登录映射到其他应用中用于同一个用户的登录的机制。它是目前比较流行的企业业务整合的解决方案之一。

### 8.2 传统的登录实现方式



此方式在只有一个web工程时是没有问题。

### 8.3集群环境下



集群环境下会出现要求用户多次登录的情况。

解决方案：

1. 配置tomcat集群。配置tomcatSession复制。节点数不要超过5个。
2. 可以使用Session服务器，保存Session信息，使每个节点是无状态。需要模拟Session。

单点登录系统是使用redis模拟Session，实现Session的统一管理。



登录的处理流程：

1. 登录页面提交用户名密码。
2. 登录成功后生成token。Token相当于原来的jsessionid，字符串，可以使用uuid。
3. 把用户信息保存到redis。Key就是token，value就是TbUser对象转换成json。
4. 使用String类型保存Session信息。可以使用“前缀:token”为key
5. 设置key的过期时间。模拟Session的过期时间。一般半个小时。
6. 把token写入cookie中。
7. Cookie需要跨域。例如www.taotao.com\sso.taotao.com\order.taotao.com，可以使用工具类。
8. Cookie的有效期。关闭浏览器失效。

登录成功。

## 九.购物车

### 9.1 购物车功能

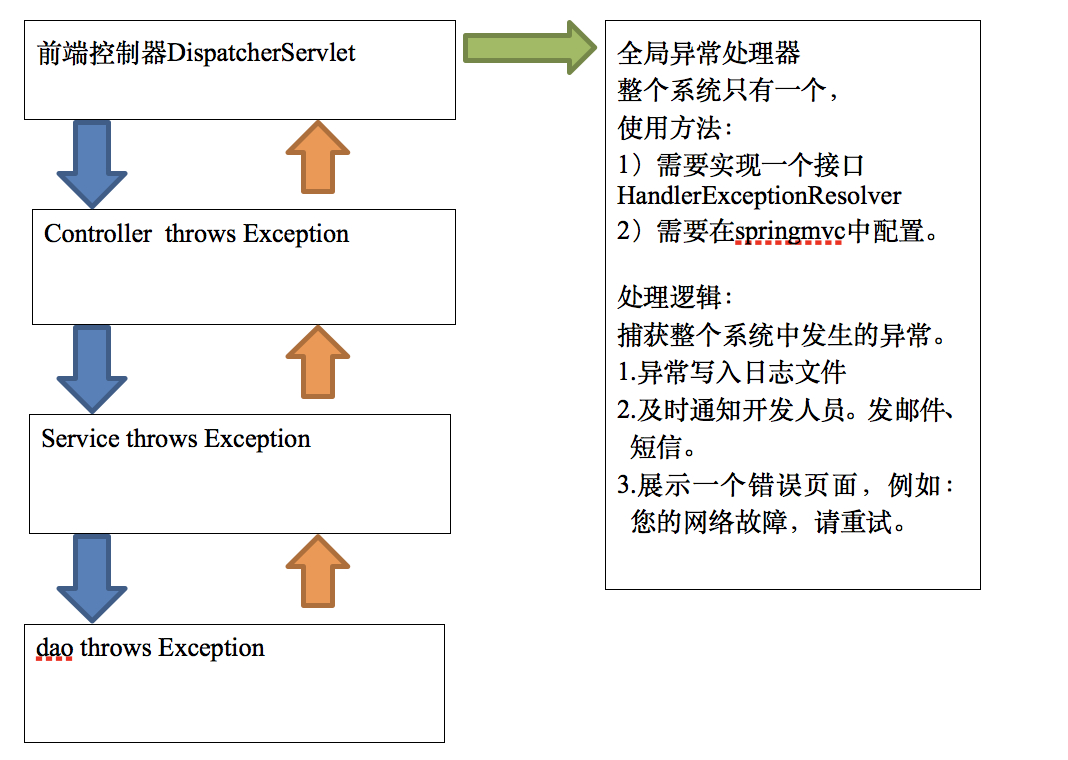
* 用户可以在登录状态下将商品添加到购物车
* 用户可以在未登录状态下将商品添加到购物车
* 用户可以使用购物车一起结算下单
* 用户可以查询自己的购物车
* 用户可以在购物车中可以修改购买商品的数量。
* 用户可以在购物车中删除商品。

### 9.2 添加购物车流程



## 十.全局异常解析

### 10.1核心思想



### 10.2创建全局异常处理器

"public class GlobalExceptionReslover implements HandlerExceptionResolver {

Logger logger = LoggerFactory.getLogger(GlobalExceptionReslover.class);

@Override

public ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response, Object handler,

Exception ex) {

//写日志文件

logger.error("系统发生异常", ex);

//发邮件、发短信

//Jmail：可以查找相关的资料

//需要在购买短信。调用第三方接口即可。

//展示错误页面

ModelAndView modelAndView = new ModelAndView();

modelAndView.addObject("message", "系统发生异常，请稍后重试");

modelAndView.setViewName("error/exception");

return modelAndView;

}

}"

### 10.3.配置

在SpringMvc 的配置文件中

<?xml version="1.0" encoding="UTF-8"?><beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:mvc="http://www.springframework.org/schema/mvc"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-4.3.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd

">

<!-- 开启注解驱动 -->

<mvc:annotation-driven/>

<!--开启默认 Servlet -->

<mvc:default-servlet-handler/>

<!-- 配置扫描Controller -->

<context:component-scan base-package="com.monkey.zhg38.manager.controller"/>

<!-- 配置全局异常解析器 -->

<bean class="com.monkey.zhg38.manager.Exception.GlobalExceptionReslover"/>

<!-- 配置 jsp的视图解析器 -->

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/jsp/"/>

<property name="suffix" value=".jsp"/>

</bean>

</beans

## 十一.跨域

### 11.1什么是跨域

跨域是指一个域下的文件或脚步试图去请求另一个域下的资源! 这里跨域是广义的!

广义的跨域

1.) 资源跳转： A链接、重定向、表单提交

2.) 资源嵌入： <link>、<script>、<img>、<frame>等dom标签，还有样式中background:url()、@font-face()等文件外链

3.) 脚本请求： js发起的ajax请求、dom和js对象的跨域操作等

      其实我们通常所说的跨域是狭义的,是由浏览器同源策略限制的一类请求场景!

### 11.2同源策略

     同源策略/SOP（Same origin policy）是一种约定，由Netscape公司1995年引入浏览器，它是浏览器最核心也最基本的安全功能，如果缺少了同源策略，浏览器很容易受到XSS、CSFR等攻击。

       所谓同源是指"协议+域名+端口"三者相同，即便两个不同的域名指向同一个ip地址，也非同源。

#### 11.2.1同源策略限制以下几种行为

　　　　 1.) Cookie,LocalStore 和 IndexDB 无法读取;

　　　　 2.) DOM 和js 对象无法获取;

　　　　 3.) Ajax 请求不能发送

#### **11.2.2常见跨域场景**

|  |  |  |
| --- | --- | --- |
| URL | 说明 | 是否允许通讯 |
| http://www.domain.com/a.js  http://www.domain.com/b.js  http://www.domain.com/lab/c.js | **同一域名，不同文件或路径** | 是 |
| http://www.domain.com:8000/a.js  http://www.domain.com/b.js | 同一域名，不同端口 | 否 |
| http://www.domain.com/a.js  https://www.domain.com/b.js | 同一域名，不同协议 | 否 |
| http://www.domain.com/a.js  http://192.168.4.12/b.js | 域名和域名对应相同ip | 否 |
| http://www.domain.com/a.js  http://x.domain.com/b.js | 主域相同，子域不同 | 否 |
| http://domain.com/c.js  http://www.domain1.com/a.js  http://www.domain2.com/b.js | 不同域名 | 否 |

### 11.3跨域解决方案

#### 11.3.1 JSONP

   通常情况下,为了减轻web服务器的负载, 我们把js,css,img 等静态资源文件分离到另外一台独立的域名的服务器上,在html 页面中 在通过响应的标签从不同的域名下加载静态资源,这是被浏览器允许的!

   如:

<script type="text/javascript" src="http://g.alicdn.com/dingding/open-develop/1.9.0/dingtalk.js"></script>

    基于此原理: 我们可以通过动态创建script,在请求一个带参网址实现跨域通信.

##### 11.3.1.1前端设置

1.）原生ajax

// 前端设置是否带cookie

xhr.withCredentials = true;

示例代码：

var xhr = new XMLHttpRequest(); // IE8/9需用window.XDomainRequest兼容

// 前端设置是否带cookie

xhr.withCredentials = true;

xhr.open('post', 'http://www.domain2.com:8080/login', true);

xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');

xhr.send('user=admin');

xhr.onreadystatechange = function() {

if (xhr.readyState == 4 && xhr.status == 200) {

alert(xhr.responseText);

}

};

     2)jquery ajax:

  <script>

$.ajax({

url: 'http://www.domain2.com:8080/login',

type: 'get',

dataType: 'jsonp', // 请求方式为jsonp

jsonpCallback: "onBack", // 自定义回调函数名

data: {}

});

// 回调执行函数

function onBack(res) {

alert(JSON.stringify(res));

}

</script>

  3)vue.js：

this.$http.jsonp('http://www.domain2.com:8080/login', {

params: {},

jsonp: 'onBack'

}).then((res) => {

console.log(res);

})

##### 11.3.1.2 服务器后端设置

   1) 后端node.js代码示例：

var querystring = require('querystring');

var http = require('http');

var server = http.createServer();

server.on('request', function(req, res) {

var params = qs.parse(req.url.split('?')[1]);

var fn = params.callback;

// jsonp返回设置

res.writeHead(200, { 'Content-Type': 'text/javascript' });

res.write(fn + '(' + JSON.stringify(params) + ')');

res.end();

});

server.listen('8080');

console.log('Server is running at port 8080...');

   2)java

    单独Controller处理

@RequestMapping(value = "token/{token}",method = RequestMethod.GET,

//指定返回响应数据的content-type

produces = MediaType.APPLICATION\_JSON\_UTF8\_VALUE

)

public @ResponseBody Object getUserByToken(@PathVariable("token") String token,String callback){

TaotaoResult res = this.iTbUserService.getUserByToken(token);

//判断是否为jsonp 请求

if (StringUtils.isNotEmpty(callback)){

return callback+"("+JsonUtil.obj2Json(res)+")";

}

return JsonUtil.obj2Json(res);

}

//或 不过此方法 只适合 spring 4.0 以上

@RequestMapping(value = "token/{token}",method = RequestMethod.GET,

//指定返回响应数据的content-type

produces = MediaType.APPLICATION\_JSON\_UTF8\_VALUE

)

public @ResponseBody Object getUserByToken(@PathVariable("token") String token,String callback){

TaotaoResult res = this.iTbUserService.getUserByToken(token);

//判断是否为jsonp 请求

if (StringUtils.isNotEmpty(callback)){

MappingJacksonValue mappingJacksonValue = new MappingJacksonValue(res);

//设置回调

mappingJacksonValue.setJsonpFunction(callback);

return mappingJacksonValue;

}

return res;

}

   通用的Jsonp 解决方案

  1) 扩展   MappingJackson2HttpMessageConverter

import com.fasterxml.jackson.core.JsonEncoding;  
import com.monkey.zhg38.common.utils.JsonUtil;  
import org.apache.commons.io.IOUtils;  
import org.apache.commons.lang3.StringUtils;  
import org.springframework.http.HttpOutputMessage;  
import org.springframework.http.converter.HttpMessageNotWritableException;  
import org.springframework.http.converter.json.MappingJackson2HttpMessageConverter;  
import org.springframework.web.context.request.RequestContextHolder;  
import org.springframework.web.context.request.ServletRequestAttributes;  
  
import javax.servlet.http.HttpServletRequest;  
import java.io.IOException;  
import java.lang.reflect.Type;

public class CallbackMappingJackson2HttpMessageConverter extends MappingJackson2HttpMessageConverter {

private String callbackName;

@Override

protected void writeInternal(Object object, Type type, HttpOutputMessage outputMessage) throws IOException, HttpMessageNotWritableException {

//从threadLocal中获取当前的reqeust 对象

HttpServletRequest request = ((ServletRequestAttributes) RequestContextHolder.currentRequestAttributes()).getRequest();

String callbackParam = request.getParameter(callbackName);

if (StringUtils.isEmpty(callbackParam)){

super.writeInternal(object, type, outputMessage);

}else{

//支持jsonp

JsonEncoding encoding = getJsonEncoding(outputMessage.getHeaders().getContentType());

try {

String result=callbackParam+"("+JsonUtil.obj2Json(object)+")";

IOUtils.write(result,outputMessage.getBody(),encoding.getJavaName());

} catch (IOException e) {

throw new HttpMessageNotWritableException("Could not write JSON L"+e.getMessage(),e);

}

}

}

public String getCallbackName() {

return callbackName;

}

public void setCallbackName(String callbackName) {

this.callbackName = callbackName;

}

}

   2) 在springmvc 的配置文件自定义扩展

<mvc:annotation-driven>

<!-- 支持 全局 jsonp -->

<mvc:message-converters>

<bean class="com.monkey.zhg38.common.converter.CallbackMappingJackson2HttpMessageConverter">

<!-- 指定请求中的回调函数参数名 -->

<property name="callbackName" value="callback"/>

</bean>

</mvc:message-converters>

</mvc:annotation-driven>

#### 11.3.2 跨域资源共享（CORS）

 普通跨域请求：只服务端设置Access-Control-Allow-Origin即可，前端无须设置。  
带cookie请求：前后端都需要设置字段，另外需注意：所带cookie为跨域请求接口所在域的cookie，而非当前页。  
目前，所有浏览器都支持该功能(IE8+：IE8/9需要使用XDomainRequest对象来支持CORS）)，CORS也已经成为主流的跨域解决方案。

##### 11.3.2.1 前端设置：

1.）原生ajax

// 前端设置是否带cookie

xhr.withCredentials = true;

示例代码：

var xhr = new XMLHttpRequest(); // IE8/9需用window.XDomainRequest兼容

// 前端设置是否带cookie

xhr.withCredentials = true;

xhr.open('post', 'http://www.domain2.com:8080/login', true);

xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');

xhr.send('user=admin');

xhr.onreadystatechange = function() {

if (xhr.readyState == 4 && xhr.status == 200) {

alert(xhr.responseText);

}

};

2.）jQuery ajax

 $.ajax({ ... xhrFields: { withCredentials: true // 前端设置是否带cookie }, crossDomain: true, // 会让请求头中包含跨域的额外信息，但不会含cookie ... });

3.）vue框架在vue-resource封装的ajax组件中加入以下代码：

 Vue.http.options.credentials = true

##### 11.3.2.2服务端设置：

若后端设置成功，前端浏览器控制台则不会出现跨域报错信息，反之，说明没设成功。

1.）Java后台：

/\*

\* 导入包：import javax.servlet.http.HttpServletResponse;

\* 接口参数中定义：HttpServletResponse response

\*/

response.setHeader("Access-Control-Allow-Origin", "http://www.domain1.com"); // 若有端口需写全（协议+域名+端口）

response.setHeader("Access-Control-Allow-Credentials", "true");

##### 11.3.2.3 组件解决方法

首先下载jar包

cors-filter与java-property-utils

<!-- https://mvnrepository.com/artifact/com.thetransactioncompany/cors-filter -->

<dependency>

<groupId>com.thetransactioncompany</groupId>

<artifactId>cors-filter</artifactId>

<version>2.5</version>

</dependency>

<!-- https://mvnrepository.com/artifact/com.thetransactioncompany/java-property-utils -->

<dependency>

<groupId>com.thetransactioncompany</groupId>

<artifactId>java-property-utils</artifactId>

<version>1.10</version>

</dependency>

**修改web.xml**

增加以下代码（最好放在其他filter前边）

<filter>

<filter-name>CORS</filter-name>

<filter-class>com.thetransactioncompany.cors.CORSFilter</filter-class>

<init-param>

<param-name>cors.allowOrigin</param-name>

<param-value>\*</param-value>

</init-param>

<init-param>

<param-name>cors.supportedMethods</param-name>

<param-value>GET, POST, HEAD, PUT, DELETE</param-value>

</init-param>

<init-param>

<param-name>cors.supportedHeaders</param-name>

<param-value>Accept, Origin, X-Requested-With, Content-Type, Last-Modified</param-value>

</init-param>

<init-param>

<param-name>cors.exposedHeaders</param-name>

<param-value>Set-Cookie</param-value>

</init-param>

<init-param>

<param-name>cors.supportsCredentials</param-name>

<param-value>true</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>CORS</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

**jQuery请求示例**  
$.ajax("url", {

type: "POST",

xhrFields: {

withCredentials: true,

useDefaultXhrHeader: false

},

data: {

type: "test"

},

dataType: 'json',

crossDomain: true,

success: function(data, status, xhr) {

console.log(data);

}

});

## 十二.数据库读写分离

请参考 <使用spring 实现读写分离(MySQL 实现主从复制)> 以及<使用MyCat

实现数据读写分离>