# 项目说明

1. 项目描述： 一个纯后端的购物网站项目，个人独立完成，主要包括用户模块、分类模块、
2. 商品模块、购物车模块、收获地址模块和订单模块。
3. 技术栈： Spring、 Spring MVC、 Mybatis、 Mysql、 Redis、 Maven、 Nginx、 Tomcat、 Git
4. 项目内容：
   1. 实现了注册登录-搜索-购物车-提交订单-支付的电商购物基本流程。
   2. 实现项目从单服务器到 Tomcat 集群的架构演进及迭代开发。
   3. 实现从 Guava Cache 到 Redis 单服务再到 Redis 分布式的迁移。
   4. 使用 Redis+Cookie+Jackson+Filter 实现单点登录。
   5. 使用 Spring Schedule + Redis 实现分布式锁及定时关单。
   6. 使用 SpringMVC 进行全局异常处理和权限统一校验。

# 版本一说明

## 2.1 版本一主要内容

1. 项目描述： 一个纯后端的购物网站项目，个人独立完成，主要包括用户模块、
2. 分类模块、 商品模块、购物车模块、收获地址模块和订单模块。
3. 技术栈： Spring+Spring MVC+MyBatis+Tomcat+Maven+MySql+PageHelper+Nginx.
4. 项目内容：
5. 完成项目环境搭建与初始化(git 版本控制、 各种软件的环境搭建)。
6. 完成数据库表设计，主要包括(购物车表、商品类别表、订单表、支付信息表、
7. 用户表等)。
8. 完成表现层、业务层和持久层项目的分层构建；前台、后台以及订单系统等分
9. 开设计，方便后续扩展和升级。
10. 完成代码编写，并使用 Restlet 完成功能测试。

## 2.2 版本一使用到的技术和使用细节

### 2.2.1 SSM

### 2.2.2 Mysql

#### 2.2.2.1 数据库表设计

1. 用户表mmall\_user

主键：PRIMARY KEY (`id`)

密码：password 存储的是md5加密的密码

UNIQUE KEY：`user\_name\_unique` (`username`) USING BTREE

2. 商品表mmall\_product

主键：PRIMARY KEY (`id`)

2.1商品分类表mmall\_category

主键：PRIMARY KEY (`id`)

3. 订单表mmall\_order

主键：PRIMARY KEY (`id`)

UNIQUE KEY：UNIQUE KEY `order\_no\_index` (`order\_no`) USING BTREE

3.1订单子表item表mmall\_order\_item

主键：PRIMARY KEY (`id`)

KEY：KEY `order\_no\_index` (`order\_no`) USING BTREE

KEY：`order\_no\_user\_id\_index` (`user\_id`,`order\_no`) USING BTREE

4. 购物车表mmall\_cart

主键：PRIMARY KEY (`id`)

KEY：KEY `user\_id\_index` (`user\_id`) USING BTREE

5. 支付信息表mmall\_pay\_info

主键：PRIMARY KEY (`id`)

6. 订单信息表mmall\_shipping

主键：PRIMARY KEY (`id`)

#### 2.2.2.2 Q&A：

1. 主键 key 和 unique key的区别

首先需要说下索引，索引是一种特殊的文件(InnoDB数据表上的索引是表空间的一个组成部分)，它们包含着对数据表里所有记录的引用指针。更通俗的说，[数据库](http://lib.csdn.net/base/mysql" \t "_blank" \o "MySQL知识库)索引好比是一本书前面的目录，能加快数据库的查询速度。

* 普通索引(由关键字KEY或INDEX定义的索引)的唯一任务是加快对数据的访问速度。因此，应该只为那些最经常出现在查询条件(WHERE column = ...)或排序条件(ORDER BY column)中的数据列创建索引。只要有可能，就应该选择一个数据最整齐、最紧凑的数据列(如一个整数类型的数据列)来创建索引。
* 与普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值（注意和主键不同）。如果是组合索引，则列值的组合必须唯一，创建方法和普通索引类似。如果能确定某个数据列将只包含彼此各不相同的值，在为这个数据列创建索引的时候就应该用关键字UNIQUE把它定义为一个唯一索引。这么做的好处：一是简化了MySQL对这个索引的管理工作，这个索引也因此而变得更有效率；二是MySQL会在有新记录插入数据表时，自动检查新记录的这个字段的值是否已经在某个记录的这个字段里出现过了；如果是，MySQL将拒绝插入那条新记录。也就是说，唯一索引可以保证数据记录的唯一性。事实上，在许多场合，人们创建唯一索引的目的往往不是为了提高访问速度，而只是为了避免数据出现重复。
* 在前面已经反复多次强调过：必须为主键字段创建一个索引，这个索引就是所谓的"主索引"。主索引与唯一索引的唯一区别是：前者在定义时使用的关键字是PRIMARY而不是UNIQUE。

2. primary key主键与unique键的区别以及作用？

共同作用是为了约束字段/建立索引/提高查询效率

mysql主键的属性：

* + - 主键具有唯一性：是指一张表里只能有一个主键；
    - 主键作用：主键primary key是为了唯一标识一个字段，使其唯一且不能为NULL，自动生成索引；
    - 隐含定义：如果这些列没有被明确地定义为NOT NULL，MySQL会隐含地定义这些列。
    - 主键其实也是索引，甚至在MySQL的术语里面“键”就等于“索引”，

UNIQUE键的属性：

* + - 唯一性：定义了UNIQUE约束的字段中不能包含重复值，
    - 可为空：在UNIQUE约束的字段上可以包含空值.
    - 扩展：unique就是唯一,当你需要限定你的某个表字段每个值都唯一,没有重复值时使用.比如说,如果你有一个person\_Info表,并且表中有个身份证的column,那么你就可以指定该字段unique.

关系：主键=NOT NULL +UNIQUE键的结合；

NOT NULL和UNIQUE约束最好的结合。如果这些列没有被明确地定义为NOT NULL，MySQL会隐含地定义这些列。

区别：

* + 唯一性约束所在的列允许空值，但是主键约束所在的列不允许空值。
  + 可以把唯一性约束放在一个或者多个列上，这些列或列的组合必须有唯一的。但是，唯一性约束所在的列并不是表的主键列。
  + 唯一性约束强制在指定的列上创建一个唯一性索引。在默认情况下，创建唯一性的非聚簇索引，但是，也可以指定所创建的索引是聚簇索引。
  + 建立主键的目的是让外键来引用.
  + 一个表最多只有一个主键，但可以有很多唯一键

### 2.2.3 Maven

#### 2.2.3.1 坐标与依赖

1. 坐标

为了能够自动化地解析任何一个Java构件, Maven必须将它们唯一标识, 这就是依赖管理的底层基础-坐标.

Maven 坐标为Java构件引入了秩序, 任何一个构件都必须明确定义自己的坐标, 坐标元素包括groupId、artifactId、version、packaging、classfier:

|  |
| --- |
| <groupId>org.apache.commons</groupId>  <artifactId>commons-lang3</artifactId>  <version>3.4</version>  <packaging>jar</packaging> |



2. 依赖

Maven最著名的(也是几乎每个人最先接触到的)就是Maven的依赖管理, 它使得我们不必再到开源项目的官网一个个下载开源组件, 然后再一个个放入classpath. 一个依赖声明可以包含如下元素:

|  |
| --- |
| <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-web</artifactId>  <version>4.2.7.RELEASE</version>  <type>jar</type>  <scope>compile</scope>  <optional>false</optional>  <exclusions>  <exclusion>  <groupId>log4j</groupId>  <artifactId>log4j</artifactId>  </exclusion>  </exclusions>  </dependency> |

* 依赖传递

Maven依赖传递机制会自动加载我们引入的依赖包的依赖, 而不必去手动指定。

* 声明一个或者多个项目依赖, 可以包含的元素有:



* 依赖管理

Maven提供了dependency插件可以对Maven项目依赖查看以及优化, 如$ mvn dependency:tree可以查看当前项目的依赖树, 详见$ mvn dependency:help。

#### 2.2.3.2 maven仓库

Maven 中, 任何一个依赖、插件或项目构建的输出, 都可称为构件, 而Maven仓库就是集中存储这些构件的地方.

Maven仓库可简单分成两类: **本地仓库**与**远程仓库**. 当Maven根据坐标寻找构件时, 它会首先检索本地仓库, 如果本地存在则直接使用, 否则去远程仓库下载.

* 本地仓库: 默认地址为~/.m2/, 一个构件只有在本地仓库存在之后, 才能由Maven项目使用.
* 远程仓库: 远程仓库又可简单分成两类: 中央仓库和私服.
  + 由于原始的本地仓库是空的, Maven必须至少知道一个远程仓库才能在执行命令时下载需要的构件, 中央仓库就是这样一个默认的远程仓库.

#### 2.2.3.3 生命周期与插件

Maven 将所有项目的构建过程统一抽象成一套生命周期: 项目的清理、初始化、编译、测试、打包、集成测试、验证、部署和站点生成 … 几乎所有项目的构建,都能映射到这一组生命周期上. 但生命周期是抽象的(Maven的生命周期本身是不做任何实际工作), 任务执行(如编译源代码)均交由插件完成. 其中每个构建步骤都可以绑定一个或多个插件的目标,而且Maven为大多数构建步骤都编写并绑定了默认插件.当用户有特殊需要的时候, 也可以配置插件定制构建行为, 甚至自己编写插件.

1. 生命周期

Maven 拥有三套相互独立的生命周期: clean、default 和 site, 而每个生命周期包含一些phase阶段, 阶段是有顺序的, 并且后面的阶段依赖于前面的阶段. 而三套生命周期相互之间却并没有前后依赖关系, 即调用site周期内的某个phase阶段并不会对clean产生任何影响.

* clean
  + clean生命周期的目的是清理项目:
  + 执行如$ mvn clean;
* default
  + default生命周期定义了真正构建时所需要执行的所有步骤:
  + 执行如$ mvn clean install;
* site
  + site生命周期的目的是建立和发布项目站点: Maven能够基于POM所包含的信息,自动生成一个友好的站点,方便团队交流和发布项目信息
  + 执行命令如$ mvn clean deploy site-deploy;

### 2.2.4 Nginx

Nginx 是一个高性能的 Web 和反向代理服务器, 它具有有很多非常优越的特性:

* 作为 Web 服务器：相比 Apache，Nginx 使用更少的资源，支持更多的并发连接，体现更高的效率，这点使 Nginx 尤其受到虚拟主机提供商的欢迎。能够支持高达 50,000 个并发连接数的响应，感谢 Nginx 为我们选择了 epoll and kqueue 作为开发模型.
* 作为负载均衡服务器：Nginx 既可以在内部直接支持 Rails 和 PHP，也可以支持作为 HTTP代理服务器 对外进行服务。Nginx 用 C 编写, 不论是系统资源开销还是 CPU 使用效率都比 Perlbal 要好的多。
* 作为邮件代理服务器: Nginx 同时也是一个非常优秀的邮件代理服务器（最早开发这个产品的目的之一也是作为邮件代理服务器），Last.fm 描述了成功并且美妙的使用经验。
* Nginx 安装非常的简单，配置文件 非常简洁（还能够支持perl语法），Bugs非常少的服务器: Nginx 启动特别容易，并且几乎可以做到7\*24不间断运行，即使运行数个月也不需要重新启动。你还能够在 不间断服务的情况下进行软件版本的升级。

### 2.2.5 Tomcat

### 2.2.6 Git

git跟传统的代码管理器（如:svn）不同， 主要区别在于git多了个本地仓库以及缓存区，所以即使无法联网也一样能提交代码。术语解释：

* 工作区间: 即我们创建的工程文件， 在编辑器可直观显示；
* 缓存区: 只能通过git GUI或git shell 窗口显示，提交代码、解决冲突的中转站；
* 本地仓库: 只能在git shell 窗口显示，连接本地代码跟远程代码的枢纽，不能联网时本地代码可先提交至该处；
* 远程仓库: 即保存我们代码的服务器，本文以公共版本控制系统：github为例，登录github账号后可直观显示；

<https://www.cnblogs.com/cb0327/p/5066685.html>

# 单点登陆

当多个tomcat并用Nginx分配服务器时，会出现单点登陆这类问题。

## 3.1 单点登陆

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.jd.com\sso.jd.com\order.jd.com，可以使用工具类。

8、Cookie的有效期。关闭浏览器失效。

9、登录成功。

## 3.2 重置有效期

在以上流程中，有一个问题值得关注：登陆信息有效期的问题。在每次访问的时候重置有效期，可以使用Filter实现。

1. web.xml中配置重置session时间的filter

2.编写sessionExpireFilter类

|  |
| --- |
| <filter>  <filter-name>sessionExpireFilter</filter-name>  <filter-class>com.mmall.controller.common.SessionExpireFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>sessionExpireFilter</filter-name>  <url-pattern>\*.do</url-pattern>  </filter-mapping> |

## 3.3 Q&A

#### 1. Filter和Interceptor的区别

Interceptor主要作用：拦截用户请求，进行处理，比如判断用户登录情况，权限验证，主要针对Action请求进行处理。是通过HandlerInterceptor 实现的。

* Filter是基于函数回调（doFilter()方法）的，而Interceptor则是基于Java反射的（AOP思想）。
* Filter依赖于Servlet容器，而Interceptor不依赖于Servlet容器。
* Filter对几乎所有的请求起作用，而Interceptor只能对action请求起作用。
* Interceptor可以访问Action的上下文，值栈里的对象，而Filter不能。
* 在action的生命周期里，Interceptor可以被多次调用，而Filter只能在容器初始化时调用一次。

#### 2. 单点登陆其他实现方案以及优缺点

#### 3. Cookie

Cookie译为小型文本文件或小甜饼，Web应用程序利用Cookie在客户端缓存服务器端文件。Cookie是以键值对形式存储在客户端主机硬盘中，由服务器端发送给客户端，客户端再下一次访问服务器端时，服务器端可以获取到客户端Cookie缓存文件。

Cookie是由服务器端创建的，然后由服务器端发送给客户端，客户端以键值对形式存储Cookie，并标注Cookie的来源。客户端再次访问服务器端时，存储的Cookie会保存在请求协议中，服务器端可以获取上次存储的缓存文件内容。

Cookie规范：

Http协议提供了有关Cookie的规范，现今市场上出现大量浏览器，一些浏览器对该Cookie规范进行了一些“扩展”，但Cookie缓存文件不会占满硬盘空间。

* Cookie存储的大小上限为4KB。
* 一个服务器最多在客户端浏览器中可以保存20个Cookie。
* 一个浏览器最多可以保存300个Cookie。

值得注意的是，不同浏览器之间不能共享Cookie缓存文件。例如，上次使用IE浏览器访问网站，下次使用火狐浏览器访问该网站时，不能使用IE浏览器保存的Cookie缓存文件。

Cookie的缺点：

* 多人共用一台计算机（例如导致用户名和密码不安全等问题）。
* Cookie被删除时，利用Cookie统计用户数量出现偏差。
* 一人使用多台计算机（网站会将看成多个用户等问题）
* Cookie会被附加在每次Http请求协议中，增加流量。
* Cookie使用明文（未加密）传递的，安全性低。
* Cookie的大小限制在4KB左右，无法存储复杂需求。

ck.setHttpOnly(true); 可以一定程度上防止脚本攻击。

#### 4. 说说Cookie使用的注意事项

* ck.setHttpOnly(true); 可以一定程度上防止脚本攻击。
* 正确使用cookie中的domain和path。
* 根据需要是否ck.setMaxAge(int )，如果设置为-1，代表永久。
* 删除cookie是通过ck.setMaxAge(0)。

#### 5. 说说Redis

Redis是一个开源的使用ANSI C语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value数据库，并提供多种语言的API。

#### 6. 说说Jedis，你是怎么使用的？

Jedis是Redis官方推荐的Java连接开发工具。在项目中使用JedisPool构建了一个RedisPool这个类，并通过RedisPool统一管理jedis的获取与回收，封装了相应的工具类。JedisPool的配置如下：

|  |
| --- |
| JedisPoolConfig -> GenericObjectPoolConfig -> BaseObjectPoolConfig **\*** JedisPool-连接池 **\*** maxTotal-最大连接数,默认8 **\*** maxIdle-在jedispool中最大的idle状态(空闲的)的jedis实例的个数,默认8 **\*** minIdle-在jedispool中最小的idle状态(空闲的)的jedis实例的个数,默认0 **\*** testOnBorrow-在borrow一个jedis实例的时候，是否要进行验证操作，如果赋值true。则得到的jedis实例肯定是可以用的。默认false **\*** testOnReturn-在return一个jedis实例的时候，是否要进行验证操作，如果赋值true。则放回jedispool的jedis实例肯定是可以用的。默认false **\*** redisIp-连接的ip **\*** redisPort-连接的port |

#### 7. Redis只能保存字符串等类型，你是怎么将用户信息存储到Redis里面的？

通过jackson的ObjectMapper封装了一个util类，通过这个工具类实现字符串与java对象的转换。该mapper的主要配置如下：

|  |
| --- |
| static{  objectMapper.setSerializationInclusion(Inclusion.ALWAYS);  objectMapper.configure(SerializationConfig.Feature.WRITE\_DATES\_AS\_TIMESTAMPS,false);  objectMapper.configure(SerializationConfig.Feature.FAIL\_ON\_EMPTY\_BEANS,false);  objectMapper.setDateFormat(new SimpleDateFormat(DateTimeUtil.STANDARD\_FORMAT));  objectMapper.configure(DeserializationConfig.Feature.FAIL\_ON\_UNKNOWN\_PROPERTIES,false);  } |

1. objectMapper.setSerializationInclusion:设置序列化与反序列化包括哪些内容，有以下参数可选
   1. ALWAYS
   2. NON\_NULL
   3. NON\_DEFAULT
   4. NON\_EMPTY
2. objectMapper.configure(SerializationConfig.Feature.WRITE\_DATES\_AS\_TIMESTAMPS,false)：取消默认转换timestamps形式
3. objectMapper.configure(SerializationConfig.Feature.FAIL\_ON\_EMPTY\_BEANS,false)：忽略空Bean转json的错误
4. objectMapper.setDateFormat(new SimpleDateFormat(DateTimeUtil.STANDARD\_FORMAT))：统一日期格式为yyyy-MM-dd HH:mm:ss
5. objectMapper.configure(DeserializationConfig.Feature.FAIL\_ON\_UNKNOWN\_PROPERTIES,false)：忽略 在json字符串中存在，但是在java对象中不存在对应属性的情况。防止错误

#### 8. Redis有分布式部署吗？说说

有的，但是只部署了两个Redis。另外需要注意的是使用的hash算法。本项目使用的是一致性hash算法。

配置几乎一样，这里需要注意的是使用的hash算法。本项目使用的是一致性hash算法。  
```java  
pool = new ShardedJedisPool(config,jedisShardInfoList, Hashing.MURMUR\_HASH, Sharded.DEFAULT\_KEY\_TAG\_PATTERN);  
```  
Hashing.MURMUR\_HASH：使用一致性hash算法

9. Redis和MySql等普通数据库的区别，Redis事务呢？

|  |
| --- |
| 1.mysql和redis的数据库类型  mysql是关系型数据库，主要用于存放持久化数据，将数据存储在硬盘中，读取速度较慢。  redis是NOSQL，即非关系型并且是基于键值对的数据库，也是缓存数据库，即将数据存储在缓存中，缓存的读取速度快，能够大大的提高运行效率，但是保存时间有限  2.mysql的运行机制  mysql作为持久化存储的关系型数据库，相对薄弱的地方在于每次请求访问数据库时，都存在着I/O操作，如果反复频繁的访问数据库。第一：会在反复链接数据库上花费大量时间，从而导致运行效率过慢；第二：反复的访问数据库也会导致数据库的负载过高，那么此时缓存的概念就衍生了出来。  3.缓存  缓存就是数据交换的缓冲区（cache），当浏览器执行请求时，首先会对在缓存中进行查找，如果存在，就获取；否则就访问数据库。  缓存的好处就是读取速度快  4.redis数据库  redis数据库就是一款缓存数据库，用于存储使用频繁的数据，这样减少访问数据库的次数，提高运行效率。  5.redis和mysql的区别总结  （1）类型上  从类型上来说，mysql是关系型数据库，redis是缓存数据库  （2）作用上  mysql用于持久化的存储数据到硬盘，功能强大，但是速度较慢  redis用于存储使用较为频繁的数据到缓存中，读取速度快  （3）需求上  mysql和redis因为需求的不同，一般都是配合使用。 |

|  |
| --- |
| mysql事务具有原子性，隔离性，一致性的特点。  redis提供multi， exec，watch来支持事务。  原子性，一致性：  redis保证在multi，exec之间的语句作为一个整体执行，redis在exec后，是无法回滚的，会出现部分成功，部分失败情况。  隔离性（mysql默认可重复读：事务中多次读取同一数据是一致的）：  redis的事务类似mysql的串行化隔离界别，执行期间不会乱入其他语句。redis在事务使用乐观锁。  redis通过watch来监测数据，在执行exec前，监测的数据被其他人更改会抛出错误，取消执行。而exec执行时，redis保证不会插入其他人语句来实现隔离。（可以预见到此机制如果事务中包裹过多的执行长指令，可能导致长时间阻塞其他人）  mysql的事务默认可重复读级别，串行化隔离级别可以防止幻读，由于redis事务中事务中查询指令无效，故不存在这类问题，mysql事务中可以有其他人的语句乱入，mysql通过加锁方式，保证被锁的资源不会被其他人存取，保证事务全程可重复读。  （比如mysql通过select for update对读取的数据行进行加锁，可以防止事务提交前数据被其他人修改，而对于读，允许其他人的select，但是其他人的select for update 则不允许（这种方式不会长时间挂起其他客户端，但是又会有锁？））  锁方式：  redis使用的是乐观锁方式，这种方式允许exec前修改，这时会触发异常通知。mysql的悲观锁，会锁住资源，其他客户端不能修改数据，直到事务结束释放锁。 |

# 全局异常

## 4.1 Spring异常处理

<https://www.cnblogs.com/junzi2099/p/7840294.html>

Spring 统一异常处理有 3 种方式，分别为：

1. 使用 @ ExceptionHandler 注解
2. 实现 HandlerExceptionResolver 接口
3. 使用 @controlleradvice 注解

### 4.1.1 使用@ ExceptionHandler 注解

使用该注解有一个不好的地方就是：进行异常处理的方法必须与出错的方法在同一个Controller里面。使用如下：

|  |
| --- |
| @Controller  public class GlobalController {  /\*\*  \* 用于处理异常的  \* @return  \*/  @ExceptionHandler({MyException.class})  public String exception(MyException e) {  System.out.println(e.getMessage());  e.printStackTrace();  return "exception";  }  @RequestMapping("test")  public void test() {  throw new MyException("出错了！");  }  } |

### 4.1.2 实现 HandlerExceptionResolver 接口

这种方式可以进行全局的异常控制。例如：

|  |
| --- |
| @Slf4j  @Component  public class ExceptionResolver implements HandlerExceptionResolver{  @Override  public ModelAndView resolveException(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o, Exception e) {  log.error("{} Exception",httpServletRequest.getRequestURI(),e);  ModelAndView modelAndView = new ModelAndView(new MappingJacksonJsonView());  //当使用是jackson2.x的时候使用MappingJackson2JsonView，课程中使用的是1.9。  modelAndView.addObject("status", ResponseCode.ERROR.getCode());  modelAndView.addObject("msg","接口异常,详情请查看服务端日志的异常信息");  modelAndView.addObject("data",e.toString());  return modelAndView;  }  } |

### 4.1.3 使用 @ControllerAdvice+ @ ExceptionHandler 注解

上文说到 @ ExceptionHandler 需要进行异常处理的方法必须与出错的方法在同一个Controller里面。那么当代码加入了 @ControllerAdvice，则不需要必须在同一个 controller 中了。这也是 Spring 3.2 带来的新特性。从名字上可以看出大体意思是控制器增强。 也就是说，@controlleradvice + @ ExceptionHandler 也可以实现全局的异常捕捉。

**请确保此WebExceptionHandle 类能被扫描到并装载进 Spring 容器中。**

|  |
| --- |
| @ControllerAdvice  @ResponseBody  public class WebExceptionHandle {  private static Logger logger = LoggerFactory.getLogger(WebExceptionHandle.class);  /\*\*  \* 400 - Bad Request  \*/  @ResponseStatus(HttpStatus.BAD\_REQUEST)  @ExceptionHandler(HttpMessageNotReadableException.class)  public ServiceResponse handleHttpMessageNotReadableException(HttpMessageNotReadableException e) {  logger.error("参数解析失败", e);  return ServiceResponseHandle.failed("could\_not\_read\_json");  }    /\*\*  \* 405 - Method Not Allowed  \*/  @ResponseStatus(HttpStatus.METHOD\_NOT\_ALLOWED)  @ExceptionHandler(HttpRequestMethodNotSupportedException.class)  public ServiceResponse handleHttpRequestMethodNotSupportedException(HttpRequestMethodNotSupportedException e) {  logger.error("不支持当前请求方法", e);  return ServiceResponseHandle.failed("request\_method\_not\_supported");  }  /\*\*  \* 415 - Unsupported Media Type  \*/  @ResponseStatus(HttpStatus.UNSUPPORTED\_MEDIA\_TYPE)  @ExceptionHandler(HttpMediaTypeNotSupportedException.class)  public ServiceResponse handleHttpMediaTypeNotSupportedException(Exception e) {  logger.error("不支持当前媒体类型", e);  return ServiceResponseHandle.failed("content\_type\_not\_supported");  }  /\*\*  \* 500 - Internal Server Error  \*/  @ResponseStatus(HttpStatus.INTERNAL\_SERVER\_ERROR)  @ExceptionHandler(Exception.class)  public ServiceResponse handleException(Exception e) {  if (e instanceof BusinessException){  return ServiceResponseHandle.failed("BUSINESS\_ERROR", e.getMessage());  }    logger.error("服务运行异常", e);  e.printStackTrace();  return ServiceResponseHandle.failed("server\_error");  }  } |

如果 @ExceptionHandler 注解中未声明要处理的异常类型，则默认为参数列表中的异常类型。所以还可以写成这样：

|  |
| --- |
| @ControllerAdvice  public class GlobalExceptionHandler {  @ExceptionHandler()  @ResponseBody  String handleException(Exception e){  return "Exception Deal! " + e.getMessage();  }  } |

参数对象就是 Controller 层抛出的异常对象！

继承 ResponseEntityExceptionHandler 类来实现针对 Rest 接口 的全局异常捕获，并且可以返回自定义格式：

|  |
| --- |
| @Slf4j  @ControllerAdvice  public class ExceptionHandlerBean extends ResponseEntityExceptionHandler {  /\*\*  \* 数据找不到异常  \* @param ex  \* @param request  \* @return  \* @throws IOException  \*/  @ExceptionHandler({DataNotFoundException.class})  public ResponseEntity<Object> handleDataNotFoundException(RuntimeException ex, WebRequest request) throws IOException {  return getResponseEntity(ex,request,ReturnStatusCode.DataNotFoundException);  }  /\*\*  \* 根据各种异常构建 ResponseEntity 实体. 服务于以上各种异常  \* @param ex  \* @param request  \* @param specificException  \* @return  \*/  private ResponseEntity<Object> getResponseEntity(RuntimeException ex, WebRequest request, ReturnStatusCode specificException) {  ReturnTemplate returnTemplate = new ReturnTemplate();  returnTemplate.setStatusCode(specificException);  returnTemplate.setErrorMsg(ex.getMessage());  return handleExceptionInternal(ex, returnTemplate,  new HttpHeaders(), HttpStatus.OK, request);  }  } |

### 4.1.4 区别

@ControllerAdvice只是标注当前bean是对Controller增强，具体是增强什么，则是配合@ExceptionHandler对异常进行精确处理。HandlerExceptionResolver则可以进行全局的异常控制，因此，所有的controller发生的异常都会进入其实现类，我们需要做通用处理。查看源码，可以发现HandlerExceptionResolver接口的抽象类AbstractHandlerMethodExceptionResolver和其子类ExceptionHandlerExceptionResolver一起组成使用@ExceptionHandler注释的方法进行异常解析的功能。现在基本都使用@ControllerAdvice+@ExceptionHandler注解来实现全局的异常捕获。

# 拦截器

## 5.1 使用Spring MVC拦截器实现权限统一校验

### 5.1.1 配置

1. dispatcher-servlet.xml

|  |
| --- |
| <mvc:interceptors>  <!-- 定义在这里的，所有的都会拦截-->  <mvc:interceptor>  <!--manage/a.do /manage/\*-->  <!--manage/b.do /manage/\*-->  <!--manage/product/save.do /manage/\*\*-->  <!--manage/order/detail.do /manage/\*\*-->  <mvc:mapping path="/manage/\*\*"/>  <mvc:exclude-mapping path="/manage/user/login.do"/>  <bean class="com.mmall.controller.common.interceptor.AuthorityInterceptor" />  </mvc:interceptor>  </mvc:interceptors> |

分析配置中的节点

* + interceptors节点：这个节点是SpringMVC的拦截器集配置节点，在这个节点里面我们可以声明多个interceptor
  + interceptor节点：这个节点是配置拦截路径以及你拦截器实现类的节点
    - mapping节点：符合mapping路径匹配的请求都会经过拦截器
    - exclude-mapping节点：符合exclude-mapping路径匹配的请求都不会经过拦截器
    - bean节点：配置我们自己的实现类，实现类要实现HandlerInterceptor接口

path情况分析

* + 如果要拦截同一个controller下的所有的请求，如/manage/\*\*\*.do，path应该为 /manage/\*
  + 如果要拦截请求路径为manage包下不同controller的所有请求，如 /manage/product/b.do，/manage/order/list.do，path应该为/manage/\*\*
  + 如果想要某个请求不走拦截器，那么可以配置exclude-mapping，path应该为具体的url，准确到方法，如/manage/login.do

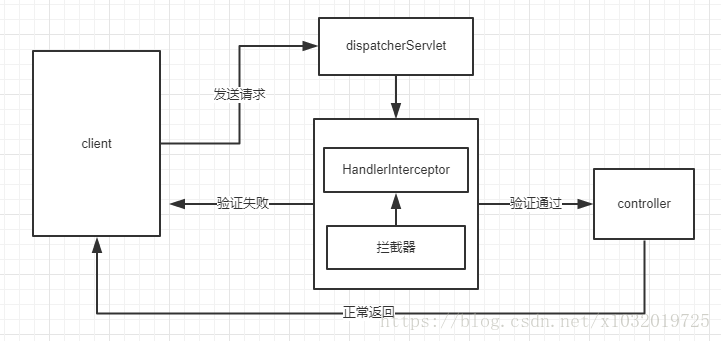
### 5.1.2 使用，coding

|  |
| --- |
| /\*\*  \* Created by dnhua  \*/  @Slf4j  public class AuthorityInterceptor implements HandlerInterceptor{  @Override  public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  return true;  }  @Override  public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {  log.info("postHandle");  }  @Override  public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {  log.info("afterCompletion");  }  } |

spring MVC拦截器请求流程

spring请求流程具体如下：

* 1. 浏览器/客户端发送请求给我们的服务器
  2. 请求来到Spring的dispatchServlet进行请求分发
  3. dispatchServlet检查我们在dispatch-servlet.xml定义好的规则，按照规则判断请求是否走拦截器
  4. 如果判断请求不走拦截器，请求直接分发到对应的controller(这里默认有对应的controller)
  5. 如果判断请求是走拦截器的，就走我们自己实现的拦截器，验证请求，通过就放行，走到controller，反之则提前返回给客户端



### 5.1.3 原理分析-关键源码分析

#### 5.1.3.1 简单回答

#### 5.1.3.2 源码分析

* HandlerInterceptor接口
* HandlerExecutionChain类
* HandlerMapping接口
* DispatcherServlet类

1. HandlerInterceptor接口

|  |
| --- |
| default boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  return true;  }  default void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, @Nullable ModelAndView modelAndView) throws Exception {  }  default void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler,  @Nullable Exception ex) throws Exception {  } |

preHandle是在找到处理handler对象的HandlerMapping之后，HandlerAdapter调度handler之前执行。

postHandle是在HandlerAdapter调度handler之后，DispatcherServlet渲染视图之前执行，可以通过ModelAndView来向视图中添加一些信息等。

afterCompletion是在渲染视图结束后执行，主要可以用来进行事后的资源清理。

2. HandlerExecutionChain类

从函数名中我们可以看出这些方法分别是做什么的，分别是执行interceptorList中所有interceptor的preHandle、postHandle和afterCompletion方法。

先从applyPreHandle()看起，我们发现这个方法就是做的这样一个工作，按照列表中interceptor的顺序来执行它们的preHandle方法，直到有一个返回false。再看一下返回false后这个方法所做的工作，这时会调用triggerAfterCompletion方法，此时this.interceptorIndex指向上一个返回true的interceptor的位置，所以它会按逆序执行所有返回true的interceptor的afterCompletion方法。换言之，也就是对于任意的返回false的interceptor都不会执行afterCompletion方法。而且是中断之前所有的preHandle执行完成之后才会执行afterCompletion方法。

接下来是applyPostHandle()，这个方法较为简单，就是按照逆序执行所有interceptor的postHandle方法。

最后的triggerAfterCompletion()也是一样，就是从最后一次preHandle成功的interceptor处逆序执行afterCompletion。

3. HandlerMapping接口

HandlerExecutionChain是通过HandlerMapping的getHandler方法返回的。

继承该接口的类是来实现请求和handler对象的映射关系的。

|  |
| --- |
| HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception; |

4. DispatcherServlet类

DispatcherServlet类中调用HandlerMapping的getHandler的方法为getHandler（同名）

* getHandler
* doDispatch

# RESTful改造

# 七、定时关单

## 7.1 Spring Schedule实现定时关单

### 7.1.1 Spring Schedule的介绍

在日常项目运行中，我们总会有需求在某一时间段周期性的执行某个动作。比如每天在某个时间段导出报表，或者每隔多久统计一次现在在线的用户量。在springboot中可以有很多方案去帮我们完成定时器的工作，有Java自带的java.util.Timer类，也有强大的调度器Quartz，还有SpringBoot自带的Scheduled，今天主要说说Scheduled。

| **框架名称** | **Cron表达式** | **固定间隔执行** | **固定频率执行** | **任务持久化** | **难易度** |
| --- | --- | --- | --- | --- | --- |
| [TimerTask](https://docs.oracle.com/javase/8/docs/api/java/util/TimerTask.html) | 不支持 | 支持 | 支持 | 不支持 | 一般 |
| [schedule](https://spring.io/guides/gs/scheduling-tasks/) | 支持 | 支持 | 支持 | 不支持 | 简单 |
| [Quartz](https://baike.baidu.com/item/quartz/3643055) | 支持 | 支持 | 支持 | 支持 | 难 |

在实际应用中，如果没有分布式场景(quartz 支持分布式, schedule 不支持(需要自己实现,用分布式锁)，schedule跟spring结合的更好，还是很适用的。

### 7.1.2 Cron表达式

cron一共有7位，但是最后一位是年，可以留空，所以我们可以写6位：

* 第一位，表示秒，取值0-59
* 第二位，表示分，取值0-59
* 第三位，表示小时，取值0-23
* 第四位，日期天/日，取值1-31
* 第五位，日期月份，取值1-12
* 第六位，星期，取值1-7，星期一，星期二...，注：不是第1周，第二周的意思 另外：1表示星期天，2表示星期一。
* 第七位，年份，可以留空，取值1970-2099

cron中，还有一些特殊的符号，含义如下：

|  |
| --- |
| (\*)星号：可以理解为每的意思，每秒，每分，每天，每月，每年...  (?)问号：问号只能出现在日期和星期这两个位置，表示这个位置的值不确定，每天3点执行，所以第六位星期的位置，我们是不需要关注的，就是不确定的值。同时：日期和星期是两个相互排斥的元素，通过问号来表明不指定值。比如，1月10日，比如是星期1，如果在星期的位置是另指定星期二，就前后冲突矛盾了。  (-)减号：表达一个范围，如在小时字段中使用“10-12”，则表示从10到12点，即10,11,12  (,)逗号：表达一个列表值，如在星期字段中使用“1,2,4”，则表示星期一，星期二，星期四  (/)斜杠：如：x/y，x是开始值，y是步长，比如在第一位（秒） 0/15就是，从0秒开始，每15秒，最后就是0，15，30，45，60 另：\*/y，等同于0/y |

一些例子：

|  |
| --- |
| "0 0 12 \* \* ?" 每天中午12点触发  "0 15 10 ? \* \*" 每天上午10:15触发  "0 15 10 \* \* ?" 每天上午10:15触发  "0 15 10 \* \* ? \*" 每天上午10:15触发  "0 15 10 \* \* ? 2005" 2005年的每天上午10:15触发  "0 \* 14 \* \* ?" 在每天下午2点到下午2:59期间的每1分钟触发  "0 0/5 14 \* \* ?" 在每天下午2点到下午2:55期间的每5分钟触发  "0 0/5 14,18 \* \* ?" 在每天下午2点到2:55期间和下午6点到6:55期间的每5分钟触发  "0 0-5 14 \* \* ?" 在每天下午2点到下午2:05期间的每1分钟触发  "0 10,44 14 ? 3 WED" 每年三月的星期三的下午2:10和2:44触发  "0 15 10 ? \* MON-FRI" 周一至周五的上午10:15触发  "0 15 10 15 \* ?" 每月15日上午10:15触发  "0 15 10 L \* ?" 每月最后一日的上午10:15触发  "0 15 10 ? \* 6L" 每月的最后一个星期五上午10:15触发  "0 15 10 ? \* 6L 2002-2005" 2002年至2005年的每月的最后一个星期五上午10:15触发  "0 15 10 ? \* 6#3" 每月的第三个星期五上午10:15触发  每隔5秒执行一次：\*/5 \* \* \* \* ?  每隔1分钟执行一次：0 \*/1 \* \* \* ?  每天23点执行一次：0 0 23 \* \* ?  每天凌晨1点执行一次：0 0 1 \* \* ?  每月1号凌晨1点执行一次：0 0 1 1 \* ?  每月最后一天23点执行一次：0 0 23 L \* ?  每周星期天凌晨1点实行一次：0 0 1 ? \* L |

这里有个在线生成器：[https://tool.lu/crontab/](https://tool.lu/crontab/" \t "_blank" \o "请叫我头头哥)

### 7.1.3 配置和实战

#### 7.1.3.1 配置

applicationContext.xml

|  |
| --- |
| <task:annotation-driven/> |

#### 7.1.3.2 定时关单版本一

|  |
| --- |
| @Component  @Slf4j  public class CloseOrderTask {  @Autowired  private IOrderService iOrderService;    @Scheduled(cron="0 \*/1 \* \* \* ?")//每1分钟(每个1分钟的整数倍)  public void closeOrderTaskV1(){  log.info("关闭订单定时任务启动");  int hour = Integer.parseInt(PropertiesUtil.getProperty("close.order.task.time.hour","2"));  iOrderService.closeOrder(hour);  log.info("关闭订单定时任务结束");  }  } |

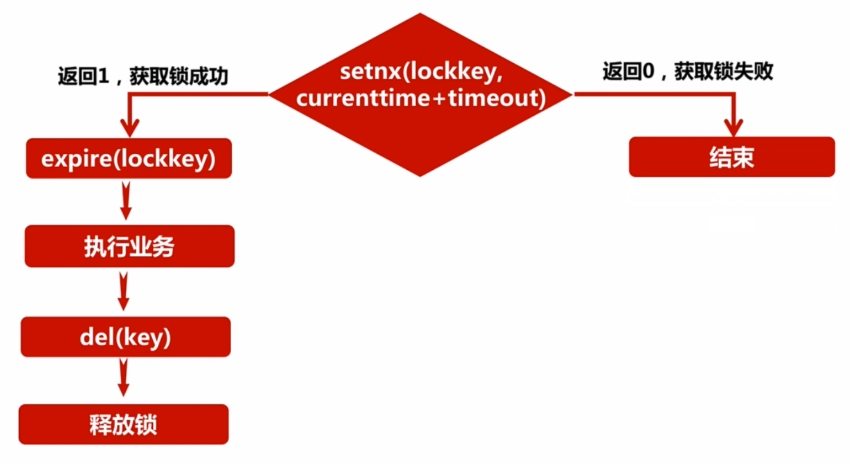
存在的问题：在tomcat集群环境中会被重复执行。

## 7.2 Redis分布式锁-定时关单的升级1

### 7.2.1 Redis分布式锁介绍

相关命令：

* setnx
* getset
* expire
* del



### 7.2.2 实战

|  |
| --- |
| @Scheduled(cron="0 \*/1 \* \* \* ?")  public void closeOrderTaskV2(){  log.info("关闭订单定时任务启动");  long lockTimeout = Long.parseLong(PropertiesUtil.getProperty("lock.timeout","5000"));  Long setnxResult = RedisShardedPoolUtil.setnx(  Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK,String.valueOf  (System.currentTimeMillis()+lockTimeout));  if(setnxResult != null && setnxResult.intValue() == 1){  //如果返回值是1，代表设置成功，获取锁  closeOrder(Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK);  }else{  log.info("没有获得分布式锁:{}", Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK);  }  log.info("关闭订单定时任务结束");  }  private void closeOrder(String lockName){  RedisShardedPoolUtil.expire(lockName,5);//有效期5秒，防止死锁  log.info("获取{},ThreadName:{}", Const.REDIS\_LOCK.  CLOSE\_ORDER\_TASK\_LOCK,Thread.currentThread().getName());  int hour = Integer.parseInt(PropertiesUtil.getProperty("close.order.task.time.hour","2"));  iOrderService.closeOrder(hour);  RedisShardedPoolUtil.del(Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK);  log.info("释放{},ThreadName:{}", Const.REDIS\_LOCK.  CLOSE\_ORDER\_TASK\_LOCK,Thread.currentThread().getName());  log.info("===============================");  } |

问题：如果因为某种原因关闭了服务，未释放锁，会发生死锁。

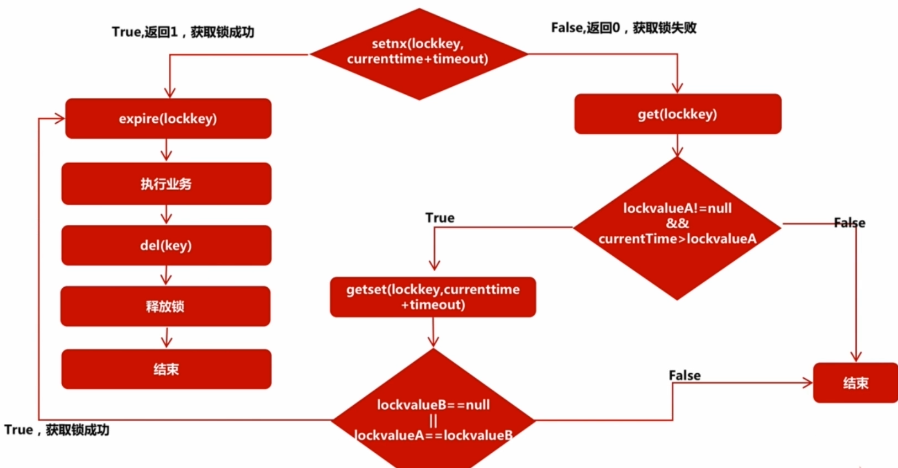
尝试解决，温柔的关闭tomcat，然后加上以下这段代码：

|  |
| --- |
| @PreDestroy  public void delLock(){  RedisShardedPoolUtil.del(Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK);  } |

仍然存在问题，关闭耗时，而且如果是直接kill掉tomcat就无能为力了。

## 7.3 Redis分布式锁-定时关单的升级2

|  |
| --- |
| @Scheduled(cron="0 \*/1 \* \* \* ?")  public void closeOrderTaskV3(){  log.info("关闭订单定时任务启动");  long lockTimeout = Long.parseLong(PropertiesUtil.getProperty("lock.timeout","5000"));  Long setnxResult = RedisShardedPoolUtil.setnx  (Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK,String.valueOf  (System.currentTimeMillis()+lockTimeout));  if(setnxResult != null && setnxResult.intValue() == 1){  closeOrder(Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK);  }else{  //未获取到锁，继续判断，判断时间戳，看是否可以重置并获取到锁  String lockValueStr = RedisShardedPoolUtil  .get(Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK);  if(lockValueStr != null && System.currentTimeMillis() > Long.parseLong(lockValueStr)){  String getSetResult = RedisShardedPoolUtil  .getSet(Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK,String  .valueOf(System.currentTimeMillis()+lockTimeout));  //再次用当前时间戳getset。  //返回给定的key的旧值，->旧值判断，是否可以获取锁  //当key没有旧值时，即key不存在时，返回nil ->获取锁  //这里我们set了一个新的value值，获取旧的值。  if(getSetResult == null || (getSetResult != null && StringUtils.equals(lockValueStr,getSetResult))){  //真正获取到锁  closeOrder(Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK);  }else{  log.info("没有获取到分布式锁:{}", Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK);  }  }else{  log.info("没有获取到分布式锁:{}", Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK);  }  }  log.info("关闭订单定时任务结束");  } |



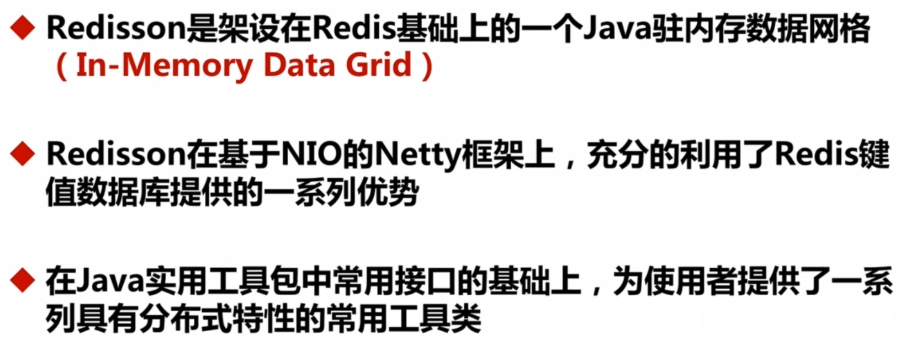
说明：

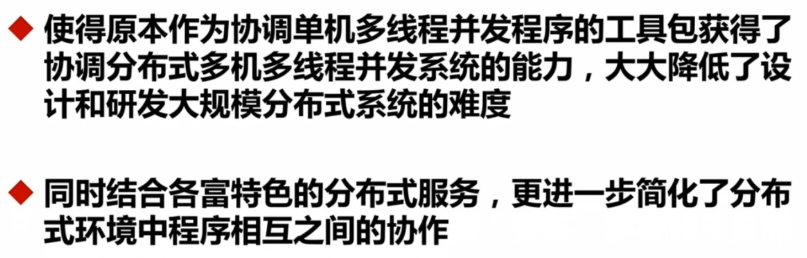
String getSetResult = RedisShardedPoolUtil

.getSet(Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK,String.valueOf(System.currentTimeMillis()+lockTimeout));

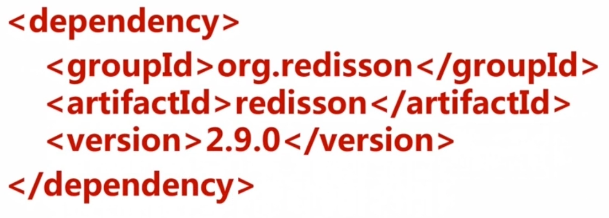
这很重要，把当前时间加上一个时间放进redis用来判断锁的状态。

## 7.4 Redisson框架解决定时关单的问题





* + 1. 添加依赖
    2. 需要添加fasterxml依赖





|  |
| --- |
| @Scheduled(cron="0 \*/1 \* \* \* ?")  public void closeOrderTaskV4(){  RLock lock = redissonManager.getRedisson().getLock(Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK);  boolean getLock = false;  try {  if(getLock = lock.tryLock(0,50, TimeUnit.SECONDS)){  log.info("Redisson获取到分布式锁:{},ThreadName:{}", Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK,Thread.currentThread().getName());  int hour = Integer.parseInt(PropertiesUtil.getProperty("close.order.task.time.hour","2"));  iOrderService.closeOrder(hour);  }else{  log.info("Redisson没有获取到分布式锁:{},ThreadName:{}", Const.REDIS\_LOCK.CLOSE\_ORDER\_TASK\_LOCK,Thread.currentThread().getName());  }  } catch (InterruptedException e) {  log.error("Redisson分布式锁获取异常",e);  } finally {  if(!getLock){  return;  }  lock.unlock();  log.info("Redisson分布式锁释放锁");  }  } |

# 附录零 Web基础

Servlet（Server Applet）是Java Servlet的简称，称为小服务程序或服务连接器，用Java编写的服务器端程序，具有独立于平台和协议的特性，主要功能在于交互式地浏览和生成数据，生成动态Web内容。

狭义的Servlet是指Java语言实现的一个接口，广义的Servlet是指任何实现了这个Servlet接口的类，一般情况下，人们将Servlet理解为后者。Servlet运行于支持Java的应用服务器中。从原理上讲，Servlet可以响应任何类型的请求，但绝大多数情况下Servlet只用来扩展基于HTTP协议的Web服务器。

最早支持Servlet标准的是JavaSoft的Java Web Server，此后，一些其它的基于Java的Web服务器开始支持标准的Servlet。

教程：<https://www.runoob.com/servlet/servlet-tutorial.html>

# 附录一 Spring相关

# 附录二 Spring MVC相关

## 一、Spring MVC DispatcherServlet

doDispatch大致流程：

* 通过请求找到对应的执行链，执行链包含了拦截器和开发者控制器。
* 通过处理器找到对应的适配器。
* 执行拦截器的事前方法，如果返回false，则流程结束，不再处理。
* 通过适配器运行处理器，然后返回模型和视图。
* 如果视图没有名称，则给出默认的视图名称。
* 执行拦截器的事后方法。
* 处理分发请求得到的数据模型和视图的渲染。

视图渲染：

* 国际化的设置
* 判断是否为逻辑视图，如果是就转化它为真实视图，或者直接获取视图即可。
* 最后进入视图的render方法，将模型和视图中的数据模型传递到该方法中去，这样就完成了视图的渲染，展示给用户。

以上内容比较粗略建议阅读DispatcherServlet源码。

## 二、Servlet filter的原理

2.1 基本介绍和使用

### 概念

* 过滤器位于客户端和web应用程序之间，用于检查和修改两者之间流过的请求和响应。
* 在请求到达Servlet/JSP之前，过滤器截获请求。
* 在响应送给客户端之前，过滤器截获响应。
* 多个过滤器形成一个过滤器链，过滤器链中不同过滤器的先后顺序由部署文件web.xml中过滤器映射<filter-mapping>的顺序决定。
* 最先截获客户端请求的过滤器将最后截获Servlet/JSP的响应信息。

### 链式结构

可以为一个Web应用组件部署多个过滤器，这些过滤器组成一个过滤器链，每个过滤器只执行某个特定的操作或者检查。这样请求在到达被访问的目标之前，需要经过这个过滤器链。

### 实现过滤器

在Web应用中使用过滤器需要实现javax.servlet.Filter接口，实现Filter接口中所定义的方法，并在web.xml中部署过滤器。

|  |
| --- |
| public class MyFilter implements Filter {  public void init(FilterConfig fc) {  //过滤器初始化代码  }  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) {  //在这里可以对客户端请求进行检查  //沿过滤器链将请求传递到下一个过滤器。  chain.doFilter(request, response);  //在这里可以对响应进行处理  }  public void destroy( ) {  //过滤器被销毁时执行的代码  }  } |

public void init(FilterConfig config)：

web容器调用本方法，说明过滤器正被加载到web容器中去。容器只有在实例化过滤器时才会调用该方法一次。容器为这个方法传递一个FilterConfig对象，其中包含与Filter相关的配置。

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)：

每当请求和响应经过过滤器链时，容器都要调用一次该方法。需要注意的是过滤器的一个实例可以同时服务于多个请求，特别需要注意线程同步问题，尽量不用或少用实例变量。 在过滤器的doFilter()方法实现中，任何出现在FilterChain的doFilter方法之前地方，request是可用的；在doFilter()方法之后response是可用的。

public void destroy()：

容器调用destroy()方法指出将从服务中删除该过滤器。如果过滤器使用了其他资源，需要在这个方法中释放这些资源。

### 部署过滤器

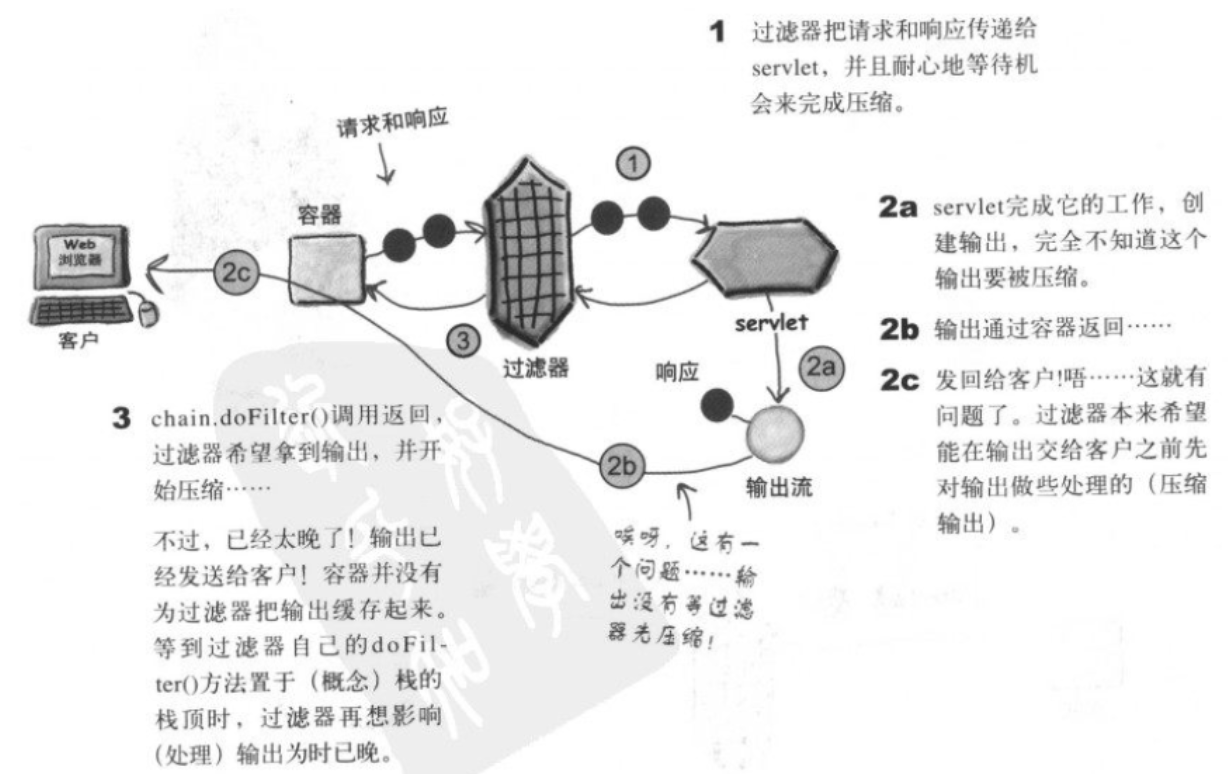
|  |
| --- |
| <filter>  <filter-name>sessionExpireFilter</filter-name>  <filter-class>com.mmall.controller.common.SessionExpireFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>sessionExpireFilter</filter-name>  <url-pattern>\*.do</url-pattern>  <dispatcher>REQUEST</dispatcher>  </filter-mapping> |

在Web应用的WEB-INF目录下，找到web.xml文件，在其中添加如下代码来声明Filter。在2.4版本的servlet规范在部属描述符中新增加了一个<dispatcher>元素，这个元素有四个可能的值：即REQUEST,FORWARD,INCLUDE和ERROR，

可以在一个<filter-mapping>元素中加入任意数目的<dispatcher>，使得filter将会作用于直接从客户端过来的request，通过forward过来的request，通过include过来的request和通过<error-page>过来的request。如果没有指定任何<dispatcher>元素，默认值是REQUEST。

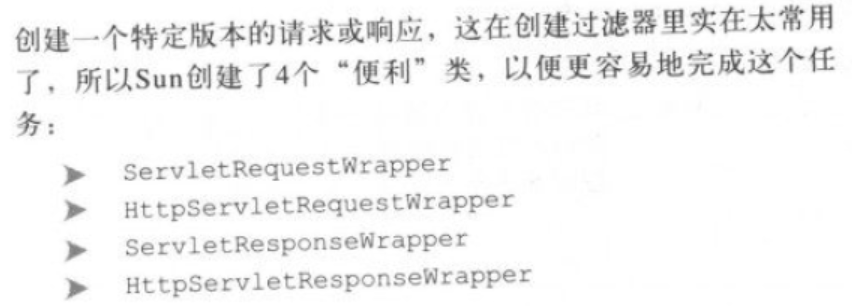
### 响应过滤器

比如要实现输出压缩：



这样不行！输出会从servlet直接返回给客户。但是我们的目标是压缩输出。

先来想想这样一个问题…… servlet 实际上是从响应对象得到输出流或书写器。那么，如果不把实际的相应对象传给servlet，而是由过滤器换入一个定制的相应对象，而且这个定制响应对象有你能控制的一个输出流，这样可以吗？需要建立我们自己的HttpServletResponse 接口定制实现，并把它通过chain.doFilter() 调用传递到servlet。而且这个定制实现还必须包含一个定制输出流，因为这正是我们的目标，在servlet写输出之后并且在输出返回给客户之前，过滤器就能拿到这个输出。



### servlet中使用HttpServletResponseWrapper截获返回的页面内容

要截获页面返回的内容，整体的思路是先把原始返回的页面内容写入到一个字符Writer，然后再组装成字符串并进行分析，最后再返回给客户端。代码如下：

|  |
| --- |
| package cn.telling.Filter;  import java.io.CharArrayWriter;  import java.io.PrintWriter;  import javax.servlet.http.HttpServletResponse;  import javax.servlet.http.HttpServletResponseWrapper;  /\*\*  \* 自定义一个响应结果包装器，将在这里提供一个基于内存的输出器来存储所有  \* 返回给客户端的原始HTML代码。  \* @ClassName: ResponseWrapper TODO  \* @author xingle  \* @date 2015-10-27 上午9:22:14  \*/  public class ResponseWrapper extends HttpServletResponseWrapper {  private PrintWriter cachedWriter;  private CharArrayWriter bufferedWriter;  /\*\*  \* @param response  \*/  public ResponseWrapper(HttpServletResponse response) {  super(response);  // 这个是我们保存返回结果的地方  bufferedWriter = new CharArrayWriter();  // 这个是包装PrintWriter的，让所有结果通过这个PrintWriter写入到bufferedWriter中  cachedWriter = new PrintWriter(bufferedWriter);  }  public PrintWriter getWriter(){  return cachedWriter;  }  /\*\*  \* 获取原始的HTML页面内容。  \* @return  \*/  public String getResult() {  return bufferedWriter.toString();  }  } |

然后再写一个过滤器来截获内容并处理：

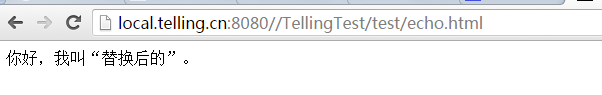
|  |
| --- |
| package cn.telling.Filter;  import java.io.IOException;  import java.io.PrintWriter;  import javax.servlet.Filter;  import javax.servlet.FilterChain;  import javax.servlet.FilterConfig;  import javax.servlet.ServletException;  import javax.servlet.ServletRequest;  import javax.servlet.ServletResponse;  import javax.servlet.http.HttpServletResponse;  /\*\*  \*  \* @ClassName: MyServletFilter TODO  \* @author xingle  \* @date 2015-10-27 上午9:24:34  \*/  public class MyServletFilter implements Filter {  /\*\*  \*  \* @Description: TODO  \* @param filterConfig  \* @throws ServletException  \* @author xingle  \* @data 2015-10-27 上午9:24:47  \*/  @Override  public void init(FilterConfig filterConfig) throws ServletException {  // TODO Auto-generated method stub  }  /\*\*  \*  \* @Description: TODO  \* @param request  \* @param response  \* @param chain  \* @throws IOException  \* @throws ServletException  \* @author xingle  \* @data 2015-10-27 上午9:24:47  \*/  @Override  public void doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) throws IOException, ServletException {  // 使用我们自定义的响应包装器来包装原始的ServletResponse  ResponseWrapper wrapper = new ResponseWrapper((HttpServletResponse) response);  // 这句话非常重要，注意看到第二个参数是我们的包装器而不是response  chain.doFilter(request, wrapper);  // 处理截获的结果并进行处理，比如替换所有的“名称”为“铁木箱子”  String result = wrapper.getResult();  result = result.replace("名称", "替换后的");  // 输出最终的结果  PrintWriter out = response.getWriter();  out.write(result);  out.flush();  out.close();  }  /\*\*  \*  \* @Description: TODO  \* @author xingle  \* @data 2015-10-27 上午9:24:47  \*/  @Override  public void destroy() {  // TODO Auto-generated method stub  }  } |

然后将该servlet配置在web.xml文件中，如下：

|  |
| --- |
| <filter>  <filter-name>MyFilter</filter-name>  <filter-class>cn.telling.Filter.MyServletFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>MyFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

然后我们在web应用根目录下建立一个jsp文件echo.jsp，内容如下：

|  |
| --- |
| <%@page language="java" contentType="text/html; charset=UTF-8"  pageEncoding="UTF-8"%>  <html xmlns="http://www.w3.org/1999/xhtml">  <head>  <head>  <title>页面返回结果过滤测试</title></head>  </head>  <body>  你好，我叫“名称”。  </body>  </html> |

配置完后，部署到tomcat，然后访问应用下的echo.jsp文件，就可以发现返回的内容变成了：

从而也就达到了我们想要的效果了。在文章开头我也提到了说有一个问题，那就是有可能在运行的过程中页面只输出一部分，尤其是在使用多个框架后(比如sitemesh)出现的可能性非常大，在探究了好久之后终于发现原来是响应的ContentLength惹的祸。因为在经过多个过滤器或是框架处理后，很有可能在其他框架中设置了响应的输出内容的长度，导致浏览器只根据得到的长度头来显示部分内容。知道了原因，处理起来就比较方便了，我们在处理结果输出前重置一下ContentLength即可，如下：

|  |
| --- |
| // 重置响应输出的内容长度  response.setContentLength(-1);  // 输出最终的结果  PrintWriter out = response.getWriter();  out.write(result);  out.flush();  out.close(); |

这样处理后就不会再出现只出现部分页面的问题了！