# Spring与SpringMVC

## Spring细节

### Spring Advice

Advice简单地说，就是一个拦截器（interceptor）拦截一些处理过程。例如，当一个method被执行，Spring AOP能够劫持正在运行的method，在method执行前或者后加入一些额外的功能。

Spring Aop4种Advice

(1)Before advice ——method执行前通知

(2)After returning advice ——method返回一个结果后通知

(3)After throwing advice – method抛出异常后通知

(4)Around advice – 环绕通知，结合了以上三种

结合了以上3种形式的Advice，创建一个实现了接口MethodInterceptor的class，你必须通过methodInvocation.proceed()来调用原来的方法，即通过调用methodInvocation.proceed()来调用CustomerService中的每一个方法，当然也可以不调用原方法.

http://www.cnblogs.com/leiOOlei/p/3557964.html

### Spring PointCut Advisor

Spring AOP Advice例子中，Class（CustomerService）中的全部method都被自动的拦截了。但是大多情况下，你只需要一个方法去拦截一两个method。这样就引入了Pointcut（切入点）的概念，它允许你根据method的名字去拦截指定的method。另外，一个Pointcut必须结合一个Advisor来使用。

在Spring AOP中，有3个常用的概念，Advices、Pointcut、Advisor，解释如下，

Advices：表示一个method执行前或执行后的动作。

Pointcut：表示根据method的名字或者正则表达式去拦截一个method。

Advisor：Advice和Pointcut组成的独立的单元，并且能够传给proxy factory 对象。

你想让DAO层的所有bean都支持AOP，以便写SQL日志，那么你必须手工创建很多的ProxyFactoryBean，这样会直接导致你的xml配置文件内容成几何级的倍增，不利于xml配置维护。

### Spring aop:aspectj-autoproxy

虽然可以通过编程的方式织入切面，但是一般情况下，我们还是使用spring的配置自动完成创建代理织入切面的工作。

通过aop命名空间的<aop:aspectj-autoproxy   
/>声明自动为spring容器中那些配置@aspectJ切面的bean创建代理，织入切面。当然，spring

在内部依旧采用**AnnotationAwareAspectJAutoProxyCreator**进行自动代理的创建工作，但具体实现的细节已经被<aop:aspectj-autoproxy />隐藏起来了

<aop:aspectj-autoproxy />有一个proxy-target-class属性，**默认为false**，表示使用jdk动态代理织入增强，当配为<aop:aspectj-autoproxy  poxy-target-class="true"/>时，表示使用CGLib动态代理技术织入增强。不过即使proxy-target-class设置为false，如果目标类没有声明接口，则spring将自动使用CGLib动态代理。

### [Spring BeanNameAutoProxyCreator 与 ProxyFactoryBean](http://www.cnblogs.com/65702708/archive/2012/08/14/2638043.html)

<http://www.cnblogs.com/65702708/archive/2012/08/14/2638043.html>

http://blog.csdn.net/adoocoke/article/details/8286902

 Spring为我们提供了自动代理机制，让容器为我们自动生成代理，把我们从烦琐的配置工作中解放出来，在内部，Spring 使用BeanPostProcessor自动地完成这项工作。

**1、实现类介绍**

    这些基于BeanPostProcessor的自动代理创建器的实现类，将根据一些规则自动在容器实例化Bean时为匹配的Bean生成代理实例。

    1）基于Bean配置名规则的自动代理创建器：允许为一组特定配置名的Bean自动创建代理实例的代理创建器，实现类为BeanNameAutoProxyCreator；

    2）基于Advisor匹配机制的自动代理创建器：它会对容器中所有的Advisor进行扫描，自动将这些切面应用到匹配的Bean中（即为目标Bean创建代理实例），实现类为DefaultAdvisorAutoProxyCreator；

3）基于Bean中AspjectJ注解标签的自动代理创建器：为包含AspectJ注解的Bean自动创建代理实例，它的实现类是AnnotationAwareAspectJAutoProxyCreator，该类是Spring 2.0的新增类。

**代理 Bean 只有在第一次被应用系统需要的时候才被创建。 我们可以通过在eclipse中设置断点的方式来得知，只有在第一次调用AOP相关方法时，spring才去创建AOP的代理类，代理类创建之后，很自然地下次调用方法就直接进入代理类实例的方法中了。Spring 有两种方式创建代理： Proxy 类创建代理 ( 实现过接口的目标类 ) 和运用 CGLIB 库创建代理 ( 没有实现过任何接口的目标类 ) 。需要注意两点： 1 、对接口创建代理优于对类创建代理，因为这样会产生更加松耦合的系统， 2 、标记为 final 的方法是不能被通知的，因为 Spring 在实现的时候是为目标类产生子类。**

*BeanFactoryPostProcessor*

### BeanFactoryPostProcessor

BeanFactoryPostProcessor可以修改BEAN的配置信息而BeanPostProcessor不能，下面举个例子说明

http://blog.csdn.net/xiao\_jun\_0820/article/details/7242379

### BeanPostProcessor后置处理器

Spring的BeanPostProcessor接口，该接口作用是：如果我们需要在Spring容器完成Bean的实例化，配置和其他的初始化后添加一些自己的逻辑处理，我们就可以定义一个或者多个BeanPostProcessor接口的实现。

有时，你会发现需要立刻在Spring实例化一个bean的前后执行一些附件操作。这些操作可以简单到修改一个bean，也可以复杂到返回一个完全不同的对象。

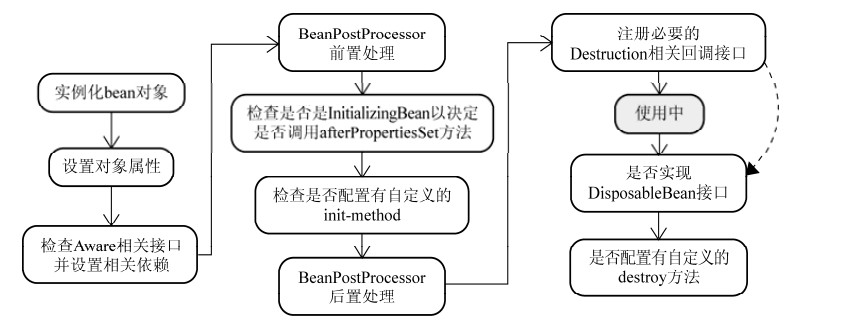
BeanPostProcessor接口包含两个方法：

postProcessBeforeInitialization：在Spring调用任何bean的初始化钩子（例如InitializingBean.afterPropertiesSet或者init方法）之前被调用。

postProcessAfterInitialization：Spring在成功完成嵌入初始化以后调用他。

BeanPostProcessor接口作用是：**如果我们需要在Spring容器完成Bean的实例化、配置和其他的初始化前后添加一些自己的逻辑处理，我们就可以定义一个或者多个BeanPostProcessor接口的实现，然后注册到容器中。**

Spring中Bean的实例化过程图示：



由上图可以看到，Spring中的BeanPostProcessor在实例化过程处于的位置，BeanPostProcessor接口有两个方法需要实现：postProcessBeforeInitialization和postProcessAfterInitialization，

http://uule.iteye.com/blog/2094549

http://blog.csdn.net/partner4java/article/details/6973782

BeanDefinitionRegistry

### BeanDefinitionRegistry

### id和name的区别

id和name都是spring 容器中中bean 的唯一标识符。

id: 一个bean的唯一标识 ， 命名格式必须符合XML ID属性的命名规范

name: 可以用特殊字符，并且一个bean可以用多个名称：

name=“bean1,bean2,bean3” ,用逗号或者分号或者空格隔开。如果没有id，则name的第一个名称默认是id

spring 容器如何处理同名bean？ 同一个spring配置文件中，bean的 id、name是不能够重复的，否则spring容器启动时会报错。

如果一个spring容器从多个配置文件中加载配置信息，则多个配置文件中是允许有同名bean的，并且后面加载的配置文件的中的bean定义会覆盖前面加载的同名bean。

spring 容器如何处理没有指定id、name属性的bean？

如果 一个 <bean> 标签未指定 id、name 属性，则 spring容器会给其一个默认的id，值为其类全名。

如果有多个<bean> 标签未指定 id、name 属性，则spring容器会按照其出现的次序，分别给其指定 id 值为 "类全名#1", "类全名#2"



### <context:component-scan>使用说明

use-default-filters默认值为true

<context:component-scan base-package="tv.huan.weisp.web"/>

Use-default-filter此时为true那么会对base-package包或者子包下的所有的进行java类进行扫描,并把匹配的java类注册成bean。

Use-dafault-filters=”false”的情况下：<context:exclude-filter>指定的不扫描，<context:include-filter>指定的扫描

<http://www.cnblogs.com/youngjoy/p/3817471.html>

### 多个配置文件加载顺序

<http://www.cnblogs.com/GarfieldTom/p/3723915.html>

### Spring Maven项目扫描Jar包存在的问题

Bean的生命周期与InitializingBean和DisposableBean

### Bean的生命周期与InitializingBean和DisposableBean

**(1)initializingBean/init-method**

Spirng的InitializingBean为bean提供了定义初始化方法的方式。InitializingBean是一个接口，它仅仅包含一个方法：afterPropertiesSet()实现org.springframework.beans.factory.InitializingBean接口允许一个bean在它的所有必须属性被BeanFactory设置后，来执行初始化的工作，InitialzingBean仅仅指定了一个方法。

注意通常InitializingBean接口的使用是能够被避免的，（不鼓励使用，因为没有必要把代码通Spring耦合起来） Bean的定义支持**指定一个普通的初始化方法**，在使用XmlBeanFactory的情况下，可以通过指定init-method属性来完成.

<baan id="exampleInitBean" class="example.ExampleBean" init-method="init"/>

public class ExampleBean {

public void init(){

//do some initialization work;

}

}

同下面的是一样的

<bean id="exampleInitBean" class="example.AnotherExampleBean" />

public class AnotherExampleBean implements InitializingBean{

public void afterPropertiesSet(){

//do some initialization work

}

}

但却不把代码耦合于spring中

**(2) DisposableBean/destroy-method**

实现org.springframework.beans.factory.DisposableBean接口的bean允许在容器销毁该bean的时候获得一次回调。DisposableBean接口也只规定了一个方法：

void destroy() throws Exception;

通常，**要避免使用DisposableBean标志接口而且不鼓励使用该接口**，因为这样会将代码与Spring耦合在一起，有一个可选的方案是，在bean定义中指定一个普通的析构方法，然后在XML配置文件中通过指定destroy-method属性来完成。如下面的定义所示：

<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>

public class ExampleBean {

public void cleanup() {

// do some destruction work (like releasing pooled connections)

}

}

...效果与下面完全一样...

<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>

public class AnotherExampleBean implements DisposableBean {

public void destroy() {

// do some destruction work (like releasing pooled connections)

}

}

但是没有将代码与Spring耦合在一起。

**(3)缺省的初始化和析构方法<beans default-init-method="init">**

如果有人没有采用Spring所指定的InitializingBean和DisposableBean回调接口来编写初始化和析构方法回调，会发现自己正在编写的方法，其名称莫过于init()， initialize()，dispose()等等。这种生命周期回调方法的名称最好在一个项目范围内标准化，这样团队中的开发人员就可以使用同样的方法名称，并且确保了某种程度的一致性。Spring容器通过配置可以实现对每个 bean初始化时的查找和销毁时的回调调用。这也就是说，一个应用的开发者可以借助于初始化的回调方法init() 轻松的写一个类（不必想XML配置文件那样为每个bean都配置一个'init-method="init"'属性）。Spring IoC容器在创建bean的时候将调用这个方法 （这和之前描述的标准生命周期回调一致）。为了完全弄清如何使用该特性，让我们看一个例子。出于示范的目的，假设一个项目的编码规范中约定所有的初始化回调方法都被命名为init()而析构回调方法被命名为destroy()。遵循此规则写成的类如下所示：

public class DefaultBlogService implements BlogService {

private BlogDao blogDao;

public void setBlogDao(BlogDao blogDao) {

this.blogDao = blogDao;

}

// this is (unsurprisingly) the initialization callback method

public void init() {

if (this.blogDao == null) {

throw new IllegalStateException("The [blogDao] property must be set.");

}

}

}

<beans default-init-method="init">

<bean id="blogService" class="com.foo.DefaultBlogService">

<property name="blogDao" ref="blogDao" />

</bean>

</beans>

注意在顶级的<beans/>元素中的'default-init-method'属性。这个属性的含义是Spring IoC容器在bean创建和装配的时候会将'init'方法作为实例化回调方法。如果类有这个方法，则会在适当的时候执行。销毁回调方法配置是相同的 (XML配置)，在顶级的<beans/>元素中使用 'default-destroy-method' 属性。使用这个功能可以把你从位每个bean指定初始化和销毁回调的繁杂工作中解救出来。为了一致性，应该强制性的为初始化和销毁回调方法采用一致的命名规则。当已经存在的类的初始化方法的命名规则与惯例有差异的时候，你应该始终使用<bean/>元素中的'init-method'和'destroy-method'属性(在XML配置中)来覆盖默认的方式。最后，请注意Spring容器保证在bean的所有依赖都满足后立即执行配置的初始化回调。这意味着初始化回调在原生bean上调用，这也意味着这个时候任何诸如AOP拦截器之类的将不能被应用。一个目标bean是首先完全创建，然后才应用诸如AOP代理等拦截器链。注意，如果目标bean和代理是分开定义了，你的代码甚至可以绕开代理直接和原生bean通信。因此，在初始化方法上使用拦截器将产生未知的结果，因为这将目标bean和它的代理/拦截器的生命周期绑定并且留下了和初始bean直接通信这样奇怪的方式。

### ApplicationContextAware

<http://blog.csdn.net/kaiwii/article/details/6872642>

<http://www.iteye.com/topic/1122859>

### Spring Aware 相关接口

Spring Aware相关接口是BeanFactoryAware、ApplicaitonContextAware、ResourceLoaderAware、ServletContextAware等等，实现这些 Aware接口的Bean在被初始之后，可以取得一些相对应的资源，例如实现BeanFactoryAware的Bean在初始后，Spring容器将会注入BeanFactory的实例，而实现ApplicationContextAware的Bean，在Bean被初始后，将会**被注入 ApplicationContext的实例**等等。

　Bean取得BeanFactory、ApplicationContextAware的实例目的是什么，一般的目的就是要取得一些档案资源的存取、相 关讯息资源或是那些被注入的实例所提供的机制，例如ApplicationContextAware提供了publishEvent()方法，可以支持基于Observer模式的事件传播机制。

　ApplicationContextAware接口的定义如下：

ApplicationContextAware.java

public interface ApplicationContextAware {

    void setApplicationContext(ApplicationContext context);

}

　我们这边示范如何透过实现ApplicationContextAware注入ApplicationContext来实现事件传播，首先我们的HelloBean如下：

HelloBean.java

package onlyfun.caterpillar;

import org.springframework.context.\*;

public class HelloBean implements ApplicationContextAware {

    private ApplicationContext applicationContext;

    private String helloWord = "Hello!World!";

    public void setApplicationContext(ApplicationContext context) {

        this.applicationContext = context;

    }

    public void setHelloWord(String helloWord) {

        this.helloWord = helloWord;

    }

    public String getHelloWord() {

        applicationContext.publishEvent(

               new PropertyGettedEvent("[" + helloWord + "] is getted"));

        return helloWord;

    }

}

　ApplicationContext会由Spring容器注入，publishEvent()方法需要一个继承ApplicationEvent的对象，我们的PropertyGettedEvent继承了ApplicationEvent，如下：

PropertyGettedEvent.java

package onlyfun.caterpillar;

import org.springframework.context.\*;

public class PropertyGettedEvent extends ApplicationEvent {

    public PropertyGettedEvent(Object source) {

        super(source);

    }

}

　当ApplicationContext执行publishEvent()后，会自动寻找实现ApplicationListener接口的对象并通知其发生对应事件，我们实现了PropertyGettedListener如下：

PrppertyGettedListener.java

package onlyfun.caterpillar;

import org.springframework.context.\*;

public class PropertyGettedListener implements ApplicationListener {

    public void onApplicationEvent(ApplicationEvent event) {

        System.out.println(event.getSource().toString());

    }

}

　Listener必须被实例化，这我们可以在Bean定义档中加以定义：

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE beans PUBLIC "-//SPRING/DTD BEAN/EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

    <bean id="propertyGetterListener" class="onlyfun.caterpillar.PropertyGettedListener"/>

    <bean id="helloBean" class="onlyfun.caterpillar.HelloBean">

        <property name="helloWord"><value>Hello!Justin!</value></property>

    </bean>

</beans>

　我们写一个测试程序来测测事件传播的运行：

Test.java

package onlyfun.caterpillar;

import org.springframework.context.\*;

import org.springframework.context.support.\*;

public class Test {

    public static void main(String[] args) {

        ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");

        HelloBean hello = (HelloBean) context.getBean("helloBean");

        System.out.println(hello.getHelloWord());

    }

}

　执行结果会如下所示：

log4j:WARN No appenders could be found for logger

(org.springframework.beans.factory.xml.XmlBeanDefinitionReader).

log4j:WARN Please initialize the log4j system properly.

org.springframework.context.support.ClassPathXmlApplicationContext:

displayName=[org.springframework.context.support.ClassPathXmlApplicationContext;

hashCode=33219526]; startup date=[Fri Oct 29 10:56:35 CST 2004];

root of ApplicationContext hierarchy

[Hello!Justin!] is getted

Hello!Justin!

　以上是以实现事件传播来看看实现Aware接口取得对应对象后，可以进行的动作，同样的，您也可以实现ResourceLoaderAware接口：

ResourceLoaderAware.java

public interface ResourceLoaderAware {

    void setResourceLoader(ResourceLoader loader);

}

　实现ResourceLoader的Bean就可以取得ResourceLoader的实例，如此就可以使用它的getResource()方法，这对于必须存取档案资源的Bean相当有用。

　基本上，Spring虽然提供了这些Aware相关接口，然而Bean上若实现了这些界面，就算是与Spring发生了依赖，从另一个角度来看，虽然您可以直接在Bean上实现这些接口，但您也可以透过setter来完成依赖注入，例如：

HelloBean.java

package onlyfun.caterpillar;

import org.springframework.context.\*;

public class HelloBean {

    private ApplicationContext applicationContext;

    private String helloWord = "Hello!World!";

    public void setApplicationContext(ApplicationContext context) {

        this.applicationContext = context;

    }

    public void setHelloWord(String helloWord) {

        this.helloWord = helloWord;

    }

    public String getHelloWord() {

        applicationContext.publishEvent(new PropertyGettedEvent("[" + helloWord + "] is getted"));

        return helloWord;

    }

}

　注意这次我们并没有实现ApplicationContextAware，我们在程序中可以自行注入ApplicationContext实例：

ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");

HelloBean hello = (HelloBean) context.getBean("helloBean");

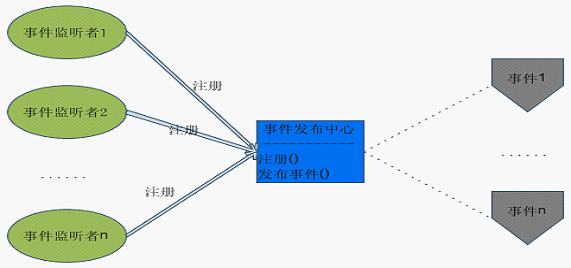
hello.setApplicationContext(context);

System.out.println(hello.getHelloWord());

　就Bean而言，降低了对Spring的依赖，可以比较容易从现有的框架中脱离。

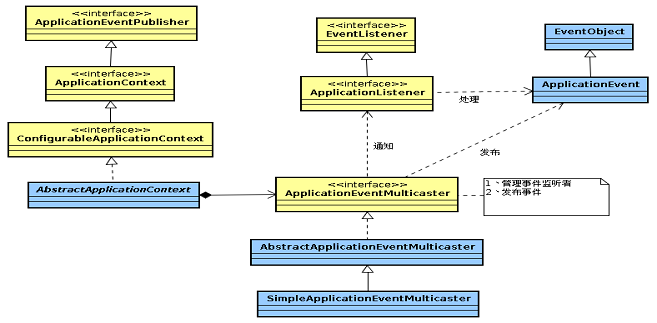
**ApplicationContext容器提供了容器内部事件发布功能，是继承自JavaSE标准自定义事件类而实现的。**

**JavaSE标准自定义事件结构不在此详细描述，一张图很直观的描述清楚：**

****

**EventObject，为JavaSE提供的事件类型基类，任何自定义的事件都继承自该类，例如上图中右侧灰色的各个事件。**[**spring**](http://lib.csdn.net/base/17)**中提供了该接口的子类ApplicationEvent。**

**EventListener为JavaSE提供的事件监听者接口，任何自定义的事件监听者都实现了该接口，如上图左侧的各个事件监听者。Spring中提供了该接口的子类ApplicationListener接口。**

**JavaSE中未提供事件发布者这一角色类，由各个应用程序自行实现事件发布者这一角色。Spring中提供了ApplicationEventPublisher接口作为事件发布者，并且ApplicationContext实现了这个接口，担当起了事件发布者这一角色。但ApplicationContext在具体实现上有所差异，Spring提供了ApplicationEventMulticaster接口，负责管理ApplicationListener和发布ApplicationEvent。ApplicationContext会把相应的事件相关工作委派给ApplicationEventMulticaster接口实现类来做。类图如下所示：**

### Spring AOP-ProxyFactory

http://blog.csdn.net/geloin/article/details/7546381

### Spring FactoryBean

Spring中有两种类型的Bean，一种是普通Bean，另一种是工厂Bean，即FactoryBean，这两种Bean都被容器管理，但工厂Bean跟普通Bean不同，其返回的对象不是指定类的一个实例，其返回的是该FactoryBean的getObject方法所返回的对象。在Spring框架内部，有很多地方有FactoryBean的实现类，它们在很多应用如(Spring的AOP、ORM、事务管理)及与其它**第三框架(EHCache)**集成时都有体现

FactoryBean 是创建 复杂的bean，一般的bean 直接用xml配置即可，如果一个bean的创建过程中涉及到很多其他的bean 和复杂的逻辑，用xml配置比较困难，这时可以考虑用FactoryBean。

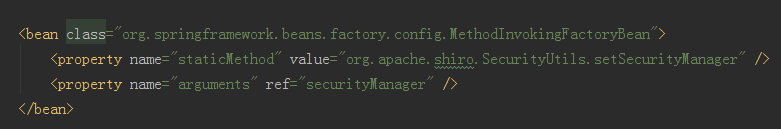
### MethodInvokingFactoryBean

Spring-Beans :

org.springframework.beans.factory.config.MethodInvokingFactoryBean

通过MethodInvokingFactoryBean工厂Bean，可以将指定方法返回值注入成为目标Bean的属性值，MethodInvokingFactoryBean用来获得指定方法的返回值，该方法可以是静态方法 ,也可以是实例方法。   
获得的方法返回值既可以被注入到指定Bean实例的指定属性，也可以直接定义成Bean实例。

Spring提供的MethodInvokingFactoryBean功能很强大，通过这个工厂Bean，我们可以通过Spring配置文件来调用指定的方法，并且获取方法飞返回值。



<http://huangyunbin.iteye.com/blog/1560107>





参考资料：http://www.cnblogs.com/chenying99/archive/2012/09/23/2698878.html

### Spring异常

1. Spring的DAO框架没有抛出与特定技术相关的异常，例如SQLException或HibernateException，抛出的异常都是与特定技术无关的org.springframework.dao.DataAccessException类的子类，避免系统与某种特殊的持久层实现耦合在一起。DataAccessException是RuntimeException，是一个无须检测的异常，不要求代码去处理这类异常，遵循了Spring的一般理念：异常检测会使代码到处是不相关的catch或throws语句，使代码杂乱无章；并且NestedRuntimeException的子类，是可以通过NestedRuntimeException的getCause（）方法获得导致该异常的另一个异常。Spring的异常分类有

|  |  |
| --- | --- |
| **Spring的DAO异常层次** | |
| 异常 | 何时抛出 |
| CleanupFailureDataAccessException | 一项操作成功地执行，但在释放数据库资源时发生异常（例如，关闭一个Connection） |
| DataAccessResourceFailureException | 数据访问资源彻底失败，例如不能连接数据库 |
| DataIntegrityViolationException | Insert或Update数据时违反了完整性，例如违反了惟一性限制 |
| DataRetrievalFailureException | 某些数据不能被检测到，例如不能通过关键字找到一条记录 |
| DeadlockLoserDataAccessException | 当前的操作因为死锁而失败 |
| IncorrectUpdateSemanticsDataAccessException | Update时发生某些没有预料到的情况，例如更改超过预期的记录数。当这个异常被抛出时，执行着的事务不会被回滚 |
| InvalidDataAccessApiusageException | 一个数据访问的JAVA API没有正确使用，例如必须在执行前编译好的查询编译失败了 |
| invalidDataAccessResourceUsageException | 错误使用数据访问资源，例如用错误的SQL语法访问关系型数据库 |
| OptimisticLockingFailureException | 乐观锁的失败。这将由ORM工具或用户的DAO实现抛出 |
| TypemismatchDataAccessException | Java类型和数据类型不匹配，例如试图把String类型插入到数据库的数值型字段中 |
| UncategorizedDataAccessException | 有错误发生，但无法归类到某一更为具体的异常中 |

1. Spring的DAO异常层次是如此的细致缜密，服务对象能够精确地选择需要捕获哪些异常，捕获的异常对用户更有用的信息，哪些异常可以让她继续在调用堆栈中向上传递。

## SpringMVC 细节

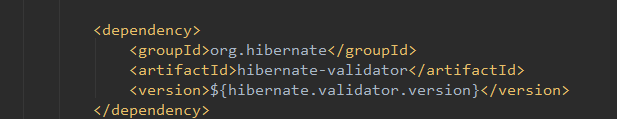
### SpringMVC注解验证国际化

@Valid需要**validation-api-xxxx.jar这个包只是一个协议，还需要加一个实现包，hibernate可以作为实现包hibernate-validator-xxxx.jar**否则会报错

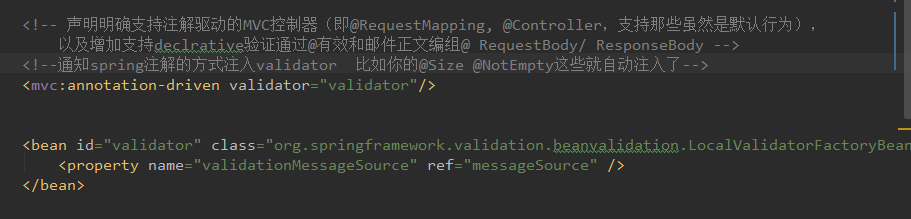
应该是跟错误消息提示的本地化有关系

<http://blog.csdn.net/shuwei003/article/details/7213662>

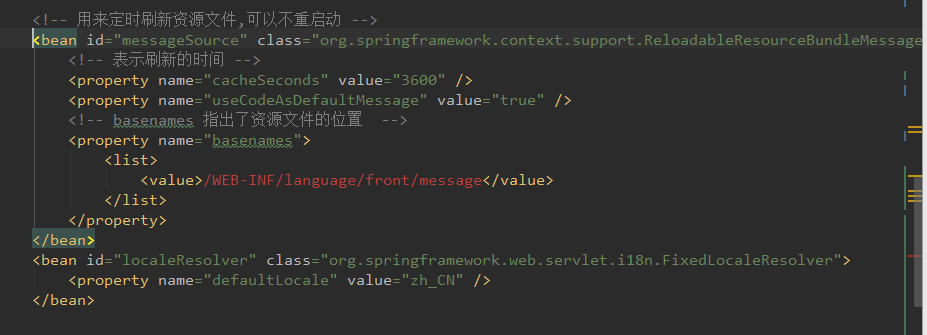
第一步，配置maven依赖环境



第二步，配置spring-mvc.xml,创建validator对象



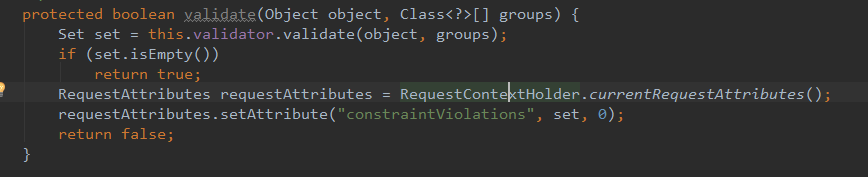
spring-contenxt.xml

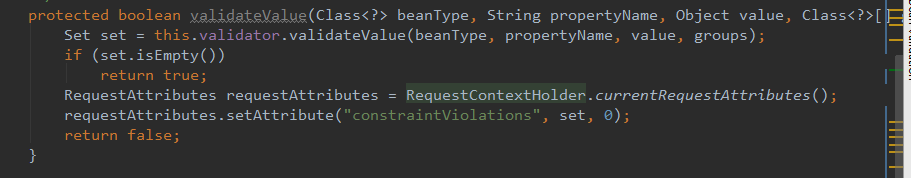


第三步，在BaseController中配置

Validator中有**三个方法**能够被用来校验整个实体对象或者实体对象中的属性.这三个方法都会返回一个Set<ConstraintViolation>对象, 如果整个验证过程没有发现问题的话,那么这个set是空的, 否则, 每个违反约束的地方都会被包装成一个ConstraintViolation的实例然后添加到set当中.

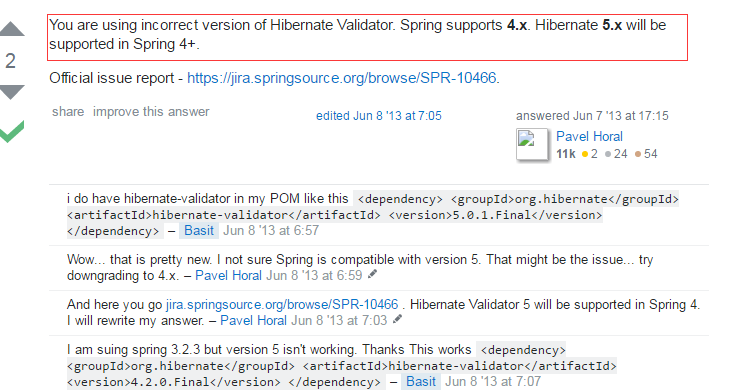
所有的校验方法都**接收零个或多个用来定义此次校验是基于哪个校验组的参数**. 如果没有给出这个参数的话, 那么此次校验将会基于默认的**校验组** (javax.validation.groups.Default)





遇到问题:

Property 'validationMessageSource' threw exception; nested exception is java.lang.NoClassDefFoundError: org/hibernate/validator/resourceloading/ResourceBundleLocator

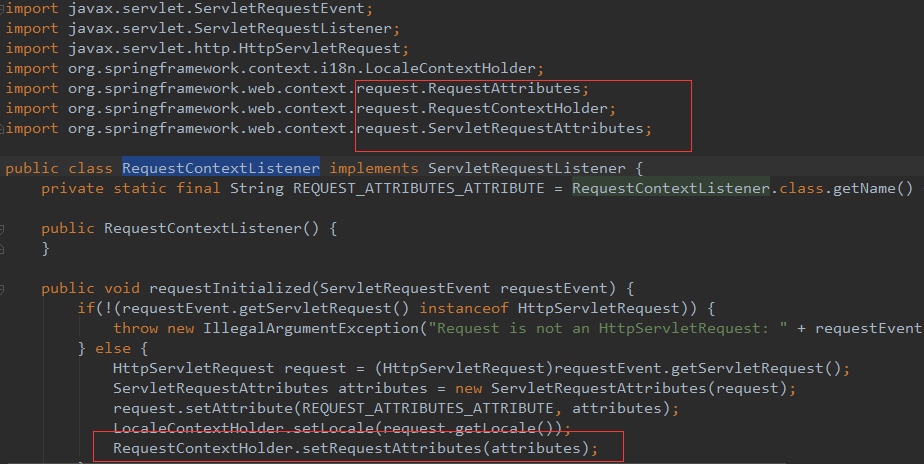


### RequestContextHolder

所在包目录:

import org.springframework.web.context.request.RequestContextHolder;

import org.springframework.web.context.request.ServletRequestAttributes;



如何在Spring中获取HttpRequest对象



### Java.util.EventListener

事件源(被观察者、主题)、事件、监听器(消费者)

### SpringMVC常见注解

#### (1) @RequestBody

作用：

i) 该注解用于读取Request请求的body部分数据，使用系统默认配置的HttpMessageConverter进行解析，然后把相应的数据绑定到要返回的对象上；

ii) 再把HttpMessageConverter返回的对象数据绑定到 controller中方法的参数上。

使用时机：

A) GET、POST方式提时， 根据request header Content-Type的值来判断:

application/x-www-form-urlencoded， 可选（即非必须，因为这种情况的数据@RequestParam, @ModelAttribute也可以处理，当然@RequestBody也能处理）；

multipart/form-data, 不能处理（即使用@RequestBody不能处理这种格式的数据）；

其他格式， 必须（其他格式包括application/json, application/xml等。这些格式的数据，必须使用@RequestBody来处理）；

B) PUT方式提交时， 根据request header Content-Type的值来判断:

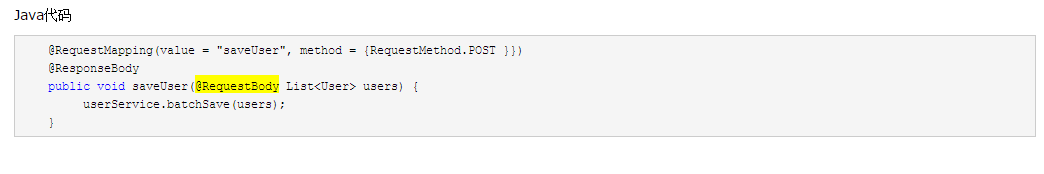
application/x-www-form-urlencoded， 必须；

multipart/form-data, 不能处理；

其他格式， 必须；

说明：request的body部分的数据编码格式由header部分的Content-Type指定；







参考资料: http://www.tuicool.com/articles/MryQfqq

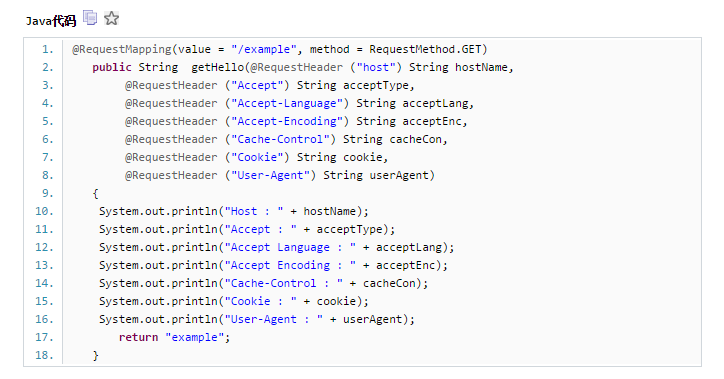
#### (2) @RequestHeader

@RequestHeader 注解，可以把Request请求header部分的值绑定到方法的参数上。

#### (3) @RequestParam

在SpringMVC后台控制层获取参数的方式主要有两种，一种是request.getParameter("name")，另外一种是用注解@RequestParam直接获取。这里主要讲这个注解.



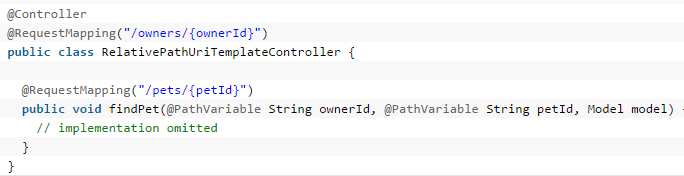


#### (4) @ResponseBody

该注解作用域将Controller的方法的对象，通过适当的HttpMessageConverter转换为指定格式后，写入到Response对象的body数据区。

#### (5) @PathVariable

当使用@RequestMapping URI template 样式映射时， 即 someUrl/{paramId}, 这时的paramId可通过 @Pathvariable注解绑定它传过来的值到方法的参数上。



### HttpMessageConverter

**使用时机**：返回的数据不是html标签的页面，而是其他某种格式的数据时(json、xml)使用。

当用户发送请求后，@Requestbody 注解会读取请求body中的数据，默认的请求转换器HttpMessageConverter通过获取请求头Header中的Content-Type来确认请求头的数据格式，从而来为请求数据适配合适的转换器。

例如contentType:applicatin/json，那么转换器会

MappingJacksonHttpMessageConverter。响应时候的时候同理，@Responsebody注解会启用HttpMessageConverter，通过检测Header中Accept属性来适配的响应的转换器。

HttpMessageConverter接口指定了一个可以把Http request信息和Http response信息进行格式转换的转换器。通常实现HttpMessageConverter接口的转换器有以下几种：

ByteArrayHttpMessageConverter: 负责读取二进制格式的数据和写出二进制格式的数据；

StringHttpMessageConverter： 负责读取字符串格式的数据和写出二进制格式的数据；

ResourceHttpMessageConverter：负责读取资源文件和写出资源文件数据；

FormHttpMessageConverter： 负责读取form提交的数据（能读取的数据格式为 application/x-www-form-urlencoded，不能读取multipart/form-data格式数据）；负责写入application/x-www-from-urlencoded和multipart/form-data格式的数据；

MappingJacksonHttpMessageConverter: 负责读取和写入json格式的数据；

SourceHttpMessageConverter： 负责读取和写入 xml 中javax.xml.transform.Source定义的数据；

Jaxb2RootElementHttpMessageConverter: 负责读取和写入xml 标签格式的数据；

AtomFeedHttpMessageConverter: 负责读取和写入Atom格式的数据；

RssChannelHttpMessageConverter: 负责读取和写入RSS格式的数据；

参考资料：<http://blog.csdn.net/kobejayandy/article/details/12690555>

### Spring MVC异常统一处理(异常信息的国际化,日志记录)

springMVC提供的异常处理主要有两种方式，一种是直接实现自己的HandlerExceptionResolver，另一种是使用注解的方式实现一个专门用于处理异常的Controller——ExceptionHandler。

使用第一种方式可以实现全局异常控制，并且Spring已经提供了一个默认的实现类**SimpleMappingExceptionResolver**；使用第二种方式可以在Controller内部实现更个性化点异常处理方式，灵活性更高。一般来说，项目中只需要采用第一种方式就可以了，每个人都自己定义异常的展现方式，太过个性了，不统一。

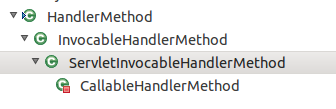
<http://blog.sina.com.cn/s/blog_61e6c583010161kt.html>

SpringMVC异常统一处理(重要参考资料)

http://www.2cto.com/kf/201412/364393.html

### HandlerMethod

HandlerMethod及子类主要用于封装方法调用相关信息,子类还提供调用,参数准备和返回值处理的职责.



分析下各个类的职责吧(顺便做分析目录):

HandlerMethod 封装方法定义相关的信息,如类,方法,参数等.

　　使用场景:HandlerMapping时会使用

InvocableHandlerMethod 添加参数准备,方法调用功能

　　使用场景:执行使用@ModelAttribute注解会使用

ServletInvocableHandlerMethod 添加返回值处理职责,ResponseStatus处理

　　使用场景:执行http相关方法会使用,比如调用处理执行

### 1.2.8 拦截器

参考资料：http://sishuok.com/forum/blogPost/list/5934.html

HandlerInterceptor 接口中定义了三个方法，我们就是通过这三个方法来对用户的请求进行拦截处理的。

(1)preHandle (HttpServletRequest request, HttpServletResponse response, Object handle) 方法，顾名思义，该方法将在请求处理之前进行调用。**SpringMVC 中的Interceptor 是链式的调用的，在一个应用中或者说是在一个请求中可以同时存在多个Interceptor** 。每个Interceptor 的调用会依据它的声明顺序依次执行，而且最先执行的都是Interceptor 中的preHandle 方法，所以可以在这个方法中进行一些前置初始化操作或者是对当前请求的一个预处理，也可以在这个方法中进行一些判断来决定请求是否要继续进行下去。该方法的返回值是布尔值Boolean 类型的，当它返回为false 时，表示请求结束，后续的Interceptor 和Controller 都不会再执行；**当返回值为true 时就会继续调用下一个Interceptor 的preHandle 方法**，如果已经是最后一个Interceptor 的时候就会是调用当前请求的Controller 方法。

(2)postHandle (HttpServletRequest request, HttpServletResponse response, Object handle, ModelAndView modelAndView) 方法，由preHandle 方法的解释我们知道这个方法包括后面要说到的afterCompletion 方法都只能是在当前所属的Interceptor 的preHandle 方法的返回值为true 时才能被调用。postHandle 方法，**顾名思义就是在当前请求进行处理之后，也就是Controller 方法调用之后执行，但是它会在DispatcherServlet 进行视图返回渲染之前被调用**，所以我们可以在这个方法中对Controller 处理之后的ModelAndView 对象进行操作。postHandle 方法被调用的方向跟preHandle 是相反的，也就是**说先声明的Interceptor 的postHandle 方法反而会后执行**，这和Struts2 里面的Interceptor 的执行过程有点类型。Struts2 里面的Interceptor 的执行过程也是链式的，只是在Struts2 里面需要手动调用ActionInvocation 的invoke 方法来触发对下一个Interceptor 或者是Action 的调用，然后每一个Interceptor 中在invoke 方法调用之前的内容都是按照声明顺序执行的，而invoke 方法之后的内容就是反向的。

(3)afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handle, Exception ex) 方法，该方法也是需要当前对应的Interceptor 的preHandle 方法的返回值为true 时才会执行。顾名思义，该方法将在整个请求结束之后，也就是在DispatcherServlet 渲染了对应的视图之后执行。这个方法的主要作用是用于进行资源清理工作的。

下面是一个简单的代码说明：

### 1.2.9RequestContextListener作用

在Spring2.0中除了以前的Singleton和Prototype外又加入了三个新的web作用域，分别为request、session和global session，如果你想让你的容器里的某个bean拥有其中某种新的web作用域，除了在bean级上配置相应的scope属性，还必须在容器级做一个额外的初始化配置。

<!--  
 Spring2.0中除了以前的Singleton和Prototype外又加入了三个新的web作用域，  
 分别为request、session和global session，  
 如果你想让你的容器里的某个bean拥有其中某种新的web作用域，除了在bean级上配置相应的scope属性，  
 还必须在容器级做一个额外的初始化配置  
 -->  
<listener>  
 <listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>  
</listener>

如果你用的是早期版本的web容器（Servlet 2.4以前），那么你要使用一个**javax.servlet.Filte**r的实现。

|  |
| --- |
| 1. <web-app> 2. ... 3. <filter> 4. <filter-name>requestContextFilter</filter-name> 5. <filter-**class**>org.springframework.web.filter.RequestContextFilter</filter-**class**> 6. </filter> 7. <filter-mapping> 8. <filter-name>requestContextFilter</filter-name> 9. <url-pattern>/\*</url-pattern> 10. </filter-mapping> 11. ... 12. </web-app> |

两种方式完成完全一样的功能：基于LocalThread将HTTP request对象绑定到为该请求提供服务的线程上。这使得具有request和session作用域的bean能够在后面的调用链中被访问到。

Request作用域

<bean id="loginAction" class="com.foo.LoginAction" scope="request"/>

针对每次HTTP请求，Spring容器会根据loginAction bean定义创建一个全新的LoginAction bean实例，且该loginAction bean实例仅在当前HTTP request内有效，因此可以根据需要放心的更改所建实例的内部状态，而其他请求中根据loginAction bean定义创建的实例，将不会看到这些特定于某个请求的状态变化。当处理请求结束，request作用域的bean实例将被销毁。

Session作用域

<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>

针对某个HTTP Session，Spring容器会根据userPreferences bean定义创建一个全新的userPreferences bean实例，且该userPreferences bean仅在当前HTTP Session内有效。与request作用域一样，你可以根据需要放心的更改所创建实例的内部状态，而别的HTTP Session中根据userPreferences创建的实例，将不会看到这些特定于某个HTTP Session的状态变化。当HTTP Session最终被废弃的时候，在该HTTP Session作用域内的bean也会被废弃掉。

global session作用域

<bean id="userPreferences" class="com.foo.UserPreferences" scope="globalSession"/>

global session作用域类似于标准的HTTP Session作用域，不过它仅仅在基于portlet的web应用中才有意义。Portlet规范定义了全局Session的概念，它被所有构成某个portlet web应用的各种不同的portlet所共享。在global session作用域中定义的bean被限定于全局portlet Session的生命周期范围内。

请注意，假如你在编写一个标准的基于Servlet的web应用，并且定义了一个或多个具有global session作用域的bean，系统会使用标准的HTTP Session作用域，并且不会引起任何错误

**为什么需要额外的配置RequestContextFilter ???**

也许会有一个疑问，已经通过ContextLoaderListener(或ContextLoaderServlet)将Web容器与Spring容器整合，为什么这里还要用额外的RequestContextListener以支持Bean的另外3个作用域，原因是ContextLoaderListener实现

ServletContextListener监听器接口，而ServletContextListener只负责监听Web容器的启动和关闭的事件。RequestContextFilter实现ServletRequestListener监听器接口，该监听器监听HTTP请求事件，Web服务器接收的每次请求都会通知该监听器。通过配置RequestContextFilter，Spring容器与Web容器结合的更加密切。

**作用域依赖问题 ???**

如果将Web相关作用域的Bean注入到singleton或prototype的Bean中，这种情况下，需要spring AOP.

|  |
| --- |
| 1. <bean name="car" **class**="com.demo.Car" scope="request"> 2. <aop:scoped-proxy/> 3. </bean> 4. <bean id="boss" **class**="com.demo.Boss" > 5. <properrty name="car" ref="car" /> 6. </bean> |

### 1.2.10问题及解决方案

(1) The request sent by the client was syntactically incorrect.

http://stackoverflow.com/questions/20616319/the-request-sent-by-the-client-was-syntactically-incorrect-spring-mvc-jdbc-te

### 1.2.11过滤器

http://www.cnblogs.com/xdp-gacl/p/3948353.html

## CSRF攻击如何防御

JavaScript(Ajax)和Cookie的同源策略：

当用户访问站点时，该站点应该生成一个（密码上很强壮的）伪随机值，并在用户的计算机上将其设为cookie。站点应该要求每个表单都包含该伪随机值（作为表单值和cookie值）。当一个POST请求被发给站点时，只有表单值和cookie值相同时，该请求才会被认为是有效的。

当攻击者以一个用户的名义提交表单时，他只能修改该表单的值。攻击者不能读取任何发自该服务器的数据或者修改cookie值，这是**同源策略**的缘故。这意味着，虽然攻击者可以用表单发送任何他想要的值，但是他却不能修改或者读取存储在该cookie中的值。因为cookie值和表单值必须是相同的，所以除非攻击者能猜出该伪随机值，否则他就无法成功地提交表单。

在所有POST方法提交的数据中提供一个不可预测的参数，比如一个随机数。或者一个根据时间计算的HASH值。并且在Cookie中也同样保存这个参数。把这个参数嵌入标签保存在FORM表单中，当浏览器提交POST请求到服务器端时．从POST数据中取出这个参数并且和Cookie中的值做比较，如果两个值相等则认为请求有效，不相等则拒绝。根据同源策略和Cookie的安全策略，第三方网页是无法取得Cookie中的参数值的．所以它不能构造出相同随机参数的POST请求。

另外，为了保证一个用户同时打开多个表单页面。所有页面都能正常工作，在一次会话的有效期内。只使用同一个随机参数。也就是说，在会话初始化的时候生成一个随机参数，在以后的页面和Cookie中，都使用这个参数。直到会话结束，新的会话开始时，才生成新的参数，否则会只有用户最后一次打开的页面才能正常提交POST请求．多标签或多窗口浏览器会不能正常工作。

<http://blog.csdn.net/ultrani/article/details/9775083>

Cookies相关知识可参考第2章节

ajaxSend方法：



ajaxComplete方法:



普通表单POST方法防御：

普通ajaxPOST方法防御：

## 如何防御XSS攻击

XSS又叫CSS (Cross Site Script) ，跨站脚本攻击。它指的是恶意攻击者往Web页面里插入恶意html代码，当用户浏览该页之时，嵌入其中Web里面的html代码会被执行，从而达到恶意攻击用户的特殊目的。XSS属于被动式的攻击，因为其被动且不好利用，所以许多人常忽略其危害性。所以我们经常只让用户输入纯文本的内容，但这样用户体验就比较差了。

一个更好的解决方法就是使用一个富文本编辑器WYSIWYG如CKEditor 和 TinyMCE。这些可以输出HTML并能够让用户可视化编辑。**虽然他们可以在客户端进行校验，但是这样还不够安全，需要在服务器端进行校验并清除有害的HTML代码**，这样才能确保输入到你网站的HTML是安全的。否则，攻击者能够绕过客户端的Javascript验证，并注入不安全的HMTL直接进入您的网站。

**Jsoup的whitelist清理器能够在服务器端对用户输入的HTML进行过滤，只输出一些安全的标签和属性。**

jsoup提供了一系列的Whitelist基本配置，能够满足大多数要求；但如有必要，也可以进行修改，不过要小心。

这个cleaner非常好用不仅可以避免XSS攻击，还可以限制用户可以输入的标签范围。

OSChina 今天早上刚刚对代码进行了改造，使用 jsoup 替换原有的 [Htmlparser](http://www.oschina.net/p/htmlparser) 来对包括发帖、回帖和评论等内容进行安全过滤。过滤的条件也比原来的要严格很多，主要是为了避免一些跨站点的脚本攻击。

在网站使用input或textarea提供给用户可输入内容的功能，比如发帖子，发文章，发评论等等。这时候需要后端程序对输入内容作安全过滤，比如<script>等可造成安全隐患的标签。java中有个开源包叫Jsoup，本身用来解析html，xml文档的，特点是可以使用类似jquery的选择权语法。最近在解决内容安全过滤的时候，通过google发现Jsoup通过自定义Whitelist（安全标签白名单）提供了这样的功能，非常好用。

参考资料：<http://my.oschina.net/ydsakyclguozi/blog/380861>

<http://my.oschina.net/ydsakyclguozi/blog/380861>

## 如何防御SQL注入

## RSA加密

## 重定向Redirect和Forward

## 开发过程中涉及到的重要类

### 1.8.1 PropertyEditorSupport

通过继承JDK 中的 java.beans.PropertyEditorSupport 类来实现自己的编辑器类 ,该类用于实现将String 类型转换成您需要的数据类型。然后我们只需要在Spring 的容器中对这个编辑器进行有效的“注册”便可以实现Spring 在装配Bean 时自动的将String 类型转换成我们自定义的类型。

如何编辑自己的PropertyEditor,其实需要了解一点java.beans包的知识，在该包中，有一个 java.beans.PropertyEditor的接口，它定义了一套接口方法（12个），即通过这些方法如何将一个String变成内部的一个对 象，这两个方法是比较重要的：   
     a)setValue(Object value) 直接设置一个对象  
     b)setAsText(String text) 通过一个字符串来构造对象，一般在此方法中解析字符串，将构造一个类对象，调用setValue(Object)来完成属性对象设置操作。

该类存在于rt.jar，java.beans.PropertyEditor。“属性编辑器”这个名字可能会让人误以为是一个带用户界面的输入器，其实属性编辑器不一定非得有用户界面，任何实现java.beans.PropertyEditor接口的类都是属性编辑器。属性编辑器的主要功能就是将外部的设置值转换为JVM内部的对应类型，所以属性编辑器其实就是一个类型转换器。

Springmvc PropertyEditor实现绑定时的类型转换(057181595900)

http://blog.csdn.net/superdog007/article/details/25615291

### 1.8.2 HandlerInterceptorAdapter

### 1.8.3 RedirectAttributes

**RedirectAttributes** 是Spring mvc 3.1版本之后出来的一个功能，专门用于重定向之后还能带参数跳转的的工具类  
他有两种带参的方式：

**第一种：**

redirectAttributes.addAttributie("prama",value); 这种方法相当于在重定向链接地址追加传递的参数，例如:

redirectAttributes.addAttributie("prama1",value1);

redirectAttributes.addAttributie("prama2",value2);

return:"redirect：/path/list"

以上重定向的方法等同于 return:"redirect：/path/list？prama1=value1**&**prama2=value2 " ，注意这种方法直接将传递的参数暴露在链接地址上，非常的不安全，慎用。

**第二种：**

redirectAttributes.addFlashAttributie("prama",value); 这种方法是隐藏了参数，链接地址上不直接暴露，但是能且只能在重定向的 **“页面”** 获取prama参数值。其原理就是放到session中，session在跳到页面后马上移除对象。如果是重定向一个controller中是获取不到该prama属性值的。除非在controller中用(@RequestPrama(value = "prama")String prama)注解，采用传参的方式。页面获值例如：

redirectAttributes.addFlashAttributie("prama1",value1);

redirectAttributes.addFlashAttributie("prama2",value2);

return:"redirect：/path/list.jsp"

在以上参数均可在list.jsp页面使用EL表达式获取到参数值${prama\*}

controller获得redirectAttributes重定向的值例如：

redirectAttributes.addFlashAttributie("prama1",value1);

redirectAttributes.addFlashAttributie("prama2",value2);

return:"redirect：/path/list/"

@RequestMapping("list")

public List<Student> list(@RequestPrama(value = "prama1")String prama1,

@RequestPrama(value = "prama2")String prama2,...

){

//TODO

//your code

}

通过在controller中的list方法体中可以获取到参数值。

## Servlet、Request、Response

HttpServletRequest request, HttpServletResponse response

request .getParameter   返回的是String,   用于读取提交的表单中的值

getAttribute 返回的是Object，需进行转换,可用setAttribute设置成任意对象，使用很灵活，可随时用；

### 3.8.1 HttpServletRequest

httpServletRequest.getRequestURI()== getContextPath() + getServletPath() + getPathInfo()

1. request.getServletPath()

URL中调用Servlet的那一部分，不包含附加路径信息和查询字符

1. request.getPathInfo()

这个方法返回请求的实际URL相对于请求的serlvet的url的路径。（个人理解。）

比如，有一个Servlet的映射是这样配置的：

<servlet-mapping>

<servlet-name>TestServlet</servlet-name>

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

</servlet-mapping>

为servlet配置的访问路径是：/servlet/test/\*

我只要访问：

http://localhost:8080/dwr/servlet/test/这里可以是任何东西

就可以访问那个servlet. dwr 是项目的名字

比如，我用这个 URL 来访问它：

http://localhost:8080/dwr/servlet/test/**joejoe1991/a.html**

这个实际的URL，相对于那个servlet 的url （"/servlet/test/\*"）的路径是：/joejoe1991/a.html

所以 request.getPathInfo() 方法返回的就是："/joejoe1991/a.html"

如果你的URL里有查询字符串，getPathInfo() 方法并不返回这些查询字符串。

例如：

http://localhost:8080/dwr/servlet/test/joejoe1991/a.html?name=test

getPathInfo() 返回的仍然是：

"/joejoe1991/a.html" ，而并不包括后面的"?name=test"

我们可以利用这个方法去做类似于多用户博客系统的那种URL。

都是http://www.xxx.com/blog/ 开头

后面跟的是用户名，

比如我要访问joejoe1991的博客：

http://www.xxx.com/blog/joejoe1991

这个joejoe1991并不是一个真实存在的目录。

建一个servlet，配置路径为：/blog/\*

然后在这个servlet里调用request.getPathInfo() 方法。

比如：http://www.xxx.com/blog/jjx

那request.getPathInfo() 方法返回的就是jjx ，表示要访问jjx的博客。

这时再去数据库里查相应的数据就好。

## 重要Web开发插件

### freeMaker模板引擎

### Jcaptcha验证码

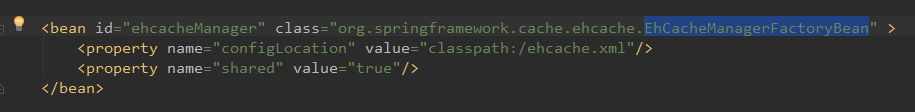
重要参考资料：http://blog.csdn.net/topwqp/article/details/8217794

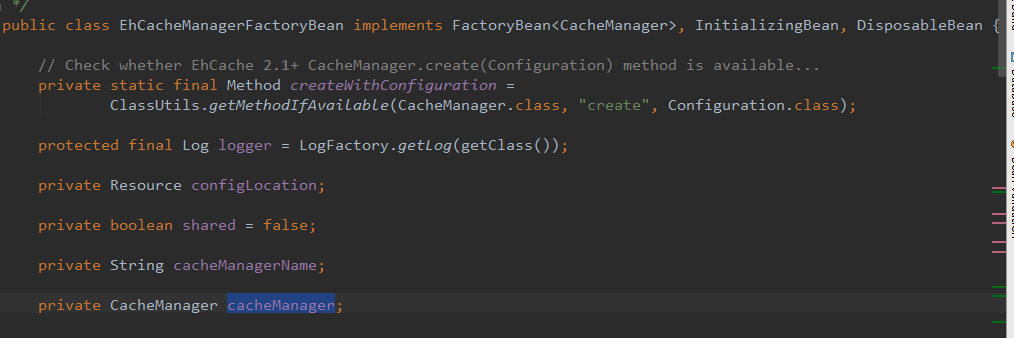
### Shiro身份认证

### Ehcache缓存

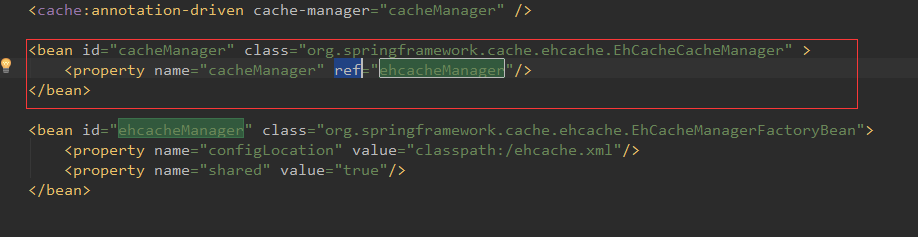
Spring Cache是作用在方法上的，其核心思想是这样的：当我们在调用一个缓存方法时会把**该方法参数和返回结果作为一个键值对存放在缓存中**，等到下次利用同样的参数来调用该方法时将不再执行该方法，而是直接从缓存中获取结果进行返回。

第一步，首先配置**EhCacheManagerFactoryBean**，Ehcache的CacheManager是通过Spring提供的EhCacheManagerFactoryBean来**生成**的，其可以通过指定ehcache的配置文件位置来生成一个Ehcache的CacheManager。若未指定则将按照Ehcache的默认规则取classpath根路径下的ehcache.xml文件，若该文件也不存在，则获取Ehcache对应jar包中的ehcache-failsafe.xml文件作为配置文件。



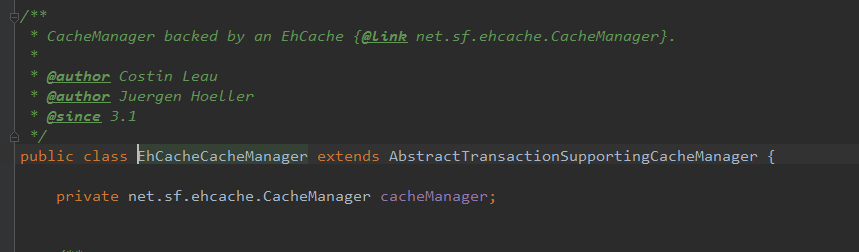


第二步配置**EhCacheCacheManager**，Spring提供的EhCacheCacheManager来生成一个**Spring的CacheManager**，其接收一个Ehcache的CacheManager，因为真正用来存入缓存数据的还是Ehcache。

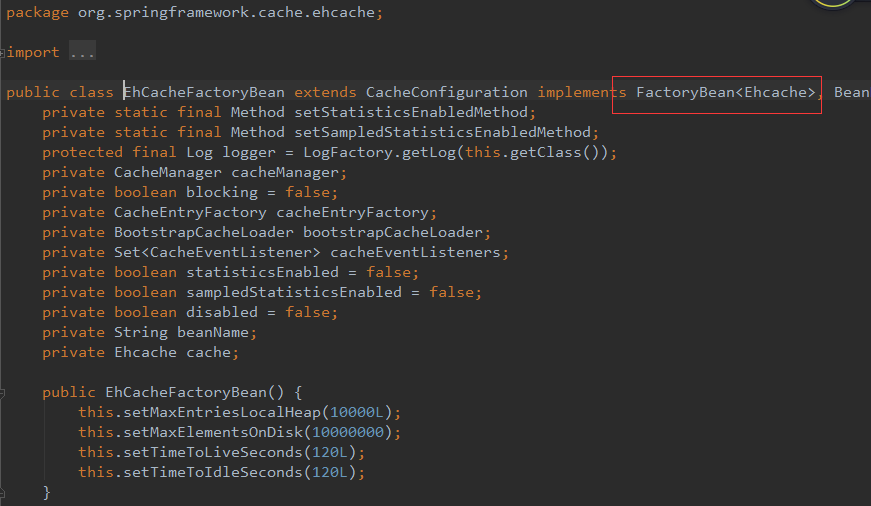


第三部需要增加注解功能，可以增加以下配置，启用缓存注解功能，这个是必须的，否则注解不会生效，另外，该注解一定**要声明在spring主配置文件中才会生效**





Ehcache的工厂Bean，CacheFactoryBean



### SiteMesh模板

### JSP

标签文件模板

http://www.programgo.com/article/54532755488/;jsessionid=BE14097D3B436D6598BC6390C0BD7431

## Filter过滤器

Http Referer是header的一部分，当浏览器向Web服务器发送请求的售后，一般会带上Referer,告诉服务器我是从哪个页面链接过来的。

## Logback

# 2、Cookies与Session相关功能

## 2.1 JavaScript端Cookie处理

### 2.1.1 cookies特点

从JS角度来看，Cookie是一些字符串信息，这些信息放在客户端的计算中，用于客户端计算机与服务器之间传递信息。JS可以通过document.cookie来读取或者设置信息。由于cookie主要用来进行客户机和服务器之间进行通信，因此除了JS以外，**服务端如(Servlet)也可以存储cookie**。

1. cookie 是有大小限制的，每个 cookie 所存放的数据不能超过4kb;
2. cookie 最终都是以文件形式存放在客户端计算机中，所以查看和修改 cookie 都是很方便的，这就是为什么常说 cookie 不能存放重要信息的原因;
3. Cookie是存在有效期的，cookie 最终都是以文件形式存放在客户端计算机中，所以查看和修改 cookie 都是很方便的，这就是为什么常说 cookie 不能存放重要信息的原因
4. document.cookie的结果类型是String，不是数组哦；
5. **Cookie有域和路径概念**。域就是domain的概念，因为浏览器是个注意安全的环境，所以不同的域之间是不能互相访问 cookie 的(当然可以通过特殊设置的达到 cookie 跨域访问)。路径就是routing的概念，一个网页所创建的 cookie 只能被与这个网页在同一目录或子目录下得所有网页访问，而不能被其他目录下得网页访问（这句话有点绕，一会看个例子就好理解了）。
6. 其实创建cookie的方式和定义变量的方式有些相似，都需要使用 cookie 名称和 cookie 值。同个网站可以创建多个 cookie ，而多个 cookie 可以存放在同一个cookie 文件中。

### 2.1.2 cookies基本操作

存取：

读取： 数组；正则



设置有效期：





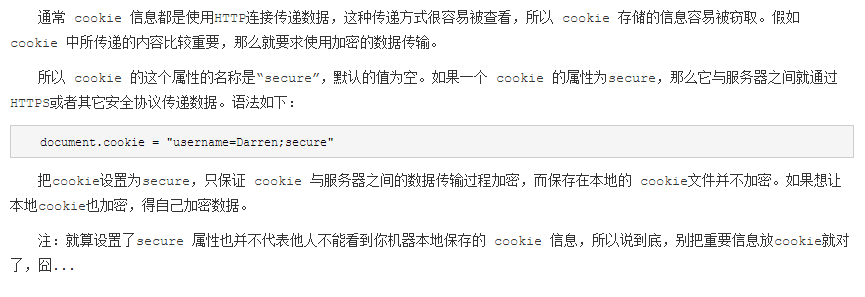
### 2.1.3 cookies路径(Path)



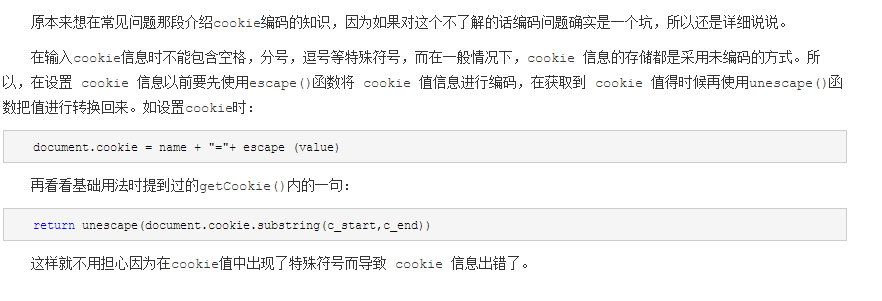
### 2.1.4 cookies域 (domain)



### 2.1.5 cookies安全性



### 2.1.6 cookies编码



## 2.2 服务端Cookie处理

### 2.2.1 Cookie有效期

Cookie的maxAge决定着Cookie的有效期，单位为秒（Second）。Cookie中通过getMaxAge()方法与setMaxAge(int maxAge)方法来读写maxAge属性。

**如果maxAge属性为正数，则表示该Cookie会在maxAge秒之后自动失效。浏览器会将maxAge为正数的Cookie持久化，即写到对应的Cookie**文件中。无论客户关闭了浏览器还是电脑，只要还在maxAge秒之前，登录网站时该Cookie仍然有效。下面代码中的Cookie信息将永远有效。

1. Cookie cookie = **new** Cookie("username", "helloweenvsfei");   
   // 新建Cookie
2. cookie.setMaxAge(Integer.MAX\_VALUE);            //   
   设置生命周期为MAX\_VALUE
3. response.addCookie(cookie);                     //   
   输出到客户端

**如果maxAge为负数，则表示该Cookie仅在本浏览器窗口以及本窗口打开的子窗口内有效，关闭窗口后该Cookie即失效。**maxAge为负数的Cookie，为临时性Cookie，不会被持久化，不会被写到Cookie文件中。Cookie信息保存在浏览器内存中，因此关闭浏览器该Cookie就消失了。Cookie默认的maxAge值为-1。

**如果maxAge为0，则表示删除该Cookie。Cookie机制没有提供删除Cookie的方法，因此通过设置该Cookie即时失效实现删除Cookie的效果。**失效的Cookie会被浏览器从Cookie文件或者内存中删除，例如：

1. Cookie cookie = **new** Cookie("username", "helloweenvsfei");    
   // 新建Cookie
2. cookie.setMaxAge(0);                            //   
   设置生命周期为0，不能为负数
3. response.addCookie(cookie);                     //   
   必须执行这一句

response对象提供的Cookie操作方法只有一个添加操作add(Cookie cookie)。要想修改Cookie只能使用一个同名的Cookie来覆盖原来的Cookie，达到修改的目的。删除时只需要把maxAge修改为0即可。

注意：从客户端读取Cookie时，包括maxAge在内的其他属性都是不可读的，也不会被提交。浏览器提交Cookie时只会提交name与value属性。maxAge属性只被浏览器用来判断Cookie是否过期。

### 2.2.1 URLEncoder和URLDecoder

1、URLEncoder是对字符编码，URLDecoder是对字符进行解码:

(1)大写字母A-Z

(2)小写字母a-z

(3)数字 0-9

(4)标点符号‘.’‘-’‘\*’‘\_’

不会被编码，是安全的。

例子：

text = "abcd";url = "a.jsp?text="+text;

这样没有问题，但是当text = "abcd&edf";

这样url就是"a.jsp?text=abcd&edf"

request.getParameter就获得abcd，剩余的就解析为另外一个参数edf。所以需要对符号"&"编码。

String value1="123&中文";

String enc="UTF-8";

String url="Http://localhost:8080/forum.jsp?id="+URLEncoder.encoder(value1,enc);

如果没有进行URL编码，getParameter("id")的值为 "123";

另外,假设服务器的Http URL encoding 是"UTF-8", 而value1中含有中文。

那么，enc也要是UTF-8。才能正确显示中文。

用于HTML的form中数据编码的类。

这个类包含将字符串转换为application/x-www-form-urlencoded MIME 格式的静态方法.

如果想了解HTML的编码细则，请参考HTML规范。

编码规则如下：

字符"a"-"z"，"A"-"Z"，"0"-"9"，"."，"-"，"\*"，和"\_" 都不被编码，维持原值，

空格" "被转换为加号"+"。

所有其他的字符都被认为是不安全的，首先都根据指定的编码scheme被转换为1个或者多个字节。[凭什么认为其他的字符都是不安全的？看来这些规范的制订者中没有中国人呀！]

然后每个字节都被表示成"%xy"格式的由3个字符组成的字符串，xy是字节的2位16进制的表达（xy is the two-digit hexadecimal representation of the byte），推荐的编码scheme为UTF-8,然而，出于兼容性的考虑，如果没有制定编码的scheme，那么将使用当前操作系统的编码的scheme。

如：如果编码scheme是UTF-8，

"The string ü@foo-bar"将被转换为"The+string+ü@foo-bar" 。

因为载UTF-8中字符ü被编码成2个字节C3 (十六进制) 和BC (十六进制)， 字符@被编码成一个字节40 (十六进制)。

起始于:JDK1.0

这个类共有2个重载方法：

public static String encode(String s, String enc) throws UnsupportedEncodingException。起始于：JDK1.4

和即将被废弃的方法：public static String encode(String s)。（因为这个方法的编码的字符集依赖于程序运行的系统的默认的字符集）。

第一个方法的作用是：根据指定的encode scheme 将一个字符串翻译成application/x-www-form-urlencoded格式。

注意: W3C推荐UTF-8。

参数:

s - 将要被翻译的字符串。

enc - 编码用的character。

返回:翻译后的字符串。

抛出异常: UnsupportedEncodingException - 如果不支持制定的编码

起始于:1.4

另请参考:URLDecoder.decode(java.lang.String, java.lang.String)

类URLDecoder的作用和URLEncoder的作用相反，方法类似，这里就不再赘述了。

如果你想知道你的字符串被编码后的值是什么样，你可以打开www.baidu.com，然后输入你要编码后的数值，然后提交，你可以在地址栏看到你被编码后的字符串

## 2.3 服务端Session

### 2.3.1 获取session

HttpRequest对象有两种形式的getSession的方法调用：

一个是getSession()

一个是getSession(boolen isNew)

这样，前者会检测当前时候是否有session存在，如果不存在则创建一个，如果存在就返回当前的。

getSession()相当于getSession(true);

参数为true时，若存在会话，则返回该会话，否则新建一个会话；

参数为false时，如存在会话，则返回该会话，否则返回NULL；

### 2.3.2 保存session Id的几种方式

保存session id的几种方式

A．保存session id的方式可以采用cookie，这样在交互过程中浏览器可以自动的按照规则把这个标识发送给服务器。

B．**由于cookie可以被人为的禁止**，必须有其它的机制以便在cookie被禁止时仍然能够把session id传递回服务器，经常采用的一种技术叫做**URL重写**，就是把session id附加在URL路径的后面，附加的方式也有两种，一种是作为URL路径的附加信息，另一种是作为查询字符串附加在URL后面。网络在整个交互过程中始终保持状态，就必须在每个客户端可能请求的路径后面都包含这个session id。

C．另一种技术叫做**表单隐藏字段**。就是服务器会自动修改表单，添加一个隐藏字段，以便在表单提交时能够把session id传递回服务器。

### 2.3.3 session 如何删除?

session在下列情况下被删除：

A．程序调用HttpSession.**invalidate**()

B．距离上一次收到客户端发送的session id时间间隔超过了session的最大有效时间

C．服务器进程被停止

关闭浏览器不会导致session被删除，迫使服务器为session设置了一个失效时间，当距离客户上一次使用session的时间超过了这个失效时间时，服务器就可以认为客户端已经停止了活动，才会把session删除以节省存储空间。

MaxInactiveInterval：最大失效时间。

## 2.4 WebSocket协议

WebSocket protocol 是HTML5一种新的协议。它实现了浏览器与服务器全双工通信(full-duplex)。一开始的握手需要借助HTTP请求完成。

很多网站为了实现即时通讯，所用的技术都是轮询(polling)。轮询是在特定的的时间间隔（如每1秒），由浏览器对服务器发出HTTP request，然后由服务器返回最新的数据给客户端的浏览器。这种传统的HTTP request 的模式带来很明显的缺点 – 浏览器需要不断的向服务器发出请求，然而**HTTP request 的header是非常长的，里面包含的有用数据可能只是一个很小的值，这样会占用很多的带宽**。

而比较新的技术去做轮询的效果是Comet – 用了AJAX。但这种技术虽然可达到全双工通信，但依然需要发出请求。

在 WebSocket API，浏览器和服务器只需要做一个握手的动作，然后，**浏览器和服务器之间就形成了一条快速通道，两者之间就直接可以数据互相传送**。在WebSocket 协议中，为我们实现即时服务带来了两大好处：

(1). Header

互相沟通的Header是很小的-大概只有 2 Bytes

(2)Server Push

服务器的推送，服务器不再被动的接收到浏览器的request之后才返回数据，而是在有新数据时就主动推送给浏览器。

在实现websocket连线过程中，需要通过浏览器发出websocket连线请求，然后服务器发出回应，这个过程通常称为“握手” (handshaking)。

PS1：握手协议在后期的版本中，会标明版本编号，下面的例子属于早期的协定之一，对于新版的 chrome 和 Firefox 皆不适用。

PS2：后期的版本大多属于功能上的扩充，例如使用第7版的握手协议同样也适用于第8版的握手协议。

WebSocket API是下一代客户端-服务器的异步通信方法。该通信取代了单个的TCP套接字，使用ws或wss协议，可用于任意的客户端和服务器程序。WebSocket目前由W3C进行标准化。WebSocket已经受到Firefox 4、Chrome 4、Opera 10.70以及Safari 5等浏览器的支持。  
  
WebSocket API最伟大之处在于服务器和客户端可以在给定的时间范围内的任意时刻，相互推送信息。WebSocket并不限于以Ajax(或XHR)方式通信，因为Ajax技术需要客户端发起请求，而WebSocket服务器和客户端可以彼此相互推送信息；XHR受到域的限制，而WebSocket允许跨域通信。  
  
Ajax技术很聪明的一点是没有设计要使用的方式。WebSocket为指定目标创建，用于双向推送消息。

# Mybatis

## 2.1 Mybatis Generator

## 2.1 Mybatis Interceptor

 拦截器已经是各个开源软件必不可少的功能。 在讨论各种问题的时候也经常听说这个对象被拦截了等等。那么在JAVA的世界里， 是怎么实现拦截器的功能的呢 ?  要了解这些， 必须先从代理类（Proxy）说起，但是我们在这里不打算从这里介绍，我们直接上Mybatis的测试代码。

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** PluginTest {
3. @Test
4. **public** **void** mapPluginShouldInterceptGet() {
5. Map map = **new** HashMap();
6. map = (Map) **new** AlwaysMapPlugin().plugin(map);
7. assertEquals("Always", map.get("Anything"));
8. }
10. @Test
11. **public** **void** shouldNotInterceptToString() {
12. Map map = **new** HashMap();
13. map = (Map) **new** AlwaysMapPlugin().plugin(map);
14. assertFalse("Always".equals(map.toString()));
15. }
17. @Intercepts({
18. @Signature(type = Map.**class**, method = "get", args = {Object.**class**})})
19. **public** **static** **class** AlwaysMapPlugin **implements** Interceptor {
20. **public** Object intercept(Invocation invocation) **throws** Throwable {
21. **return** "Always";
22. }
24. **public** Object plugin(Object target) {
25. **return** Plugin.wrap(target, **this**);
26. }
28. **public** **void** setProperties(Properties properties) {
29. }
30. }
32. }

MyBatis 直接抽象了一个plugin的概念，中文意思就是“插头”吧，很形象。我要对一个对象拦截，我就要把这个插头插入到目标对象中：

**Java代码  [收藏代码](javascript:void())**

1. map = (Map) **new** AlwaysMapPlugin().plugin(map);

这里生成了一个AlwaysMapPlugin，并调用plugin方法，把自己插入到Map的一个实例对象中；我们先不管plugin方法里面到底做了什么，我们只要知道它还是给我返回了一个Map的实例。

**Java代码  [收藏代码](javascript:void())**

1. assertEquals("Always", map.get("Anything"));

接着我们就调用了Map实例的get方法，我们纳闷了，我们并没有给Map实例中添加任何数据，但是却能得到"Always”。

很神奇吧...

要解开这个神奇，还是要回到AlwaysMapPlugin().plugin(Object object)的方法上来。

**Java代码  [收藏代码](javascript:void())**

1. **public** Object plugin(Object target) {
2. **return** Plugin.wrap(target, **this**);
3. }

直接委托给了Plugin类的一个静态方法：

**Java代码  [收藏代码](javascript:void())**

1. **public** **static** Object wrap(Object target, Interceptor interceptor) {
2. Map<Class<?>, Set<Method>> signatureMap = getSignatureMap(interceptor);
3. Class<?> type = target.getClass();
4. Class<?>[] interfaces = getAllInterfaces(type, signatureMap);
5. **if** (interfaces.length > 0) {
6. **return** Proxy.newProxyInstance(
7. type.getClassLoader(),
8. interfaces,
9. **new** Plugin(target, interceptor, signatureMap));
10. }
11. **return** target;
12. }

我们先抛开乱七八糟的逻辑，我们看到在满足一定的条件的情况下，会返回一个Proxy.newProxyInstance类的一个实例。也就是说AlwaysMapPlugin().plugin(Object object)的方法返回的实例，是有可能和原来的实例不是同一个，而是原来实例的一个代理类。

**Java代码  [收藏代码](javascript:void())**

1. **public** **static** Object newProxyInstance(ClassLoader loader,
2. Class<?>[] interfaces,
3. InvocationHandler h)

 这个是Proxy.newProxyInstance方法的签名，不明白的同志自己去了解下。

我们已经得到了目标对象的代理类，可是我们只想对目标对象的某几个方法进行拦截， 是怎么做到的呢？

这次我们把目光转向我们的“插头”吧。

**Java代码  [收藏代码](javascript:void())**

1. @Intercepts({
2. @Signature(type = Map.**class**, method = "get", args = {Object.**class**})})
3. **public** **static** **class** AlwaysMapPlugin **implements** Interceptor {
4. **public** Object intercept(Invocation invocation) **throws** Throwable {
5. **return** "Always";
6. }
8. **public** Object plugin(Object target) {
9. **return** Plugin.wrap(target, **this**);
10. }
12. **public** **void** setProperties(Properties properties) {
13. }
14. }

这里通过注解告诉我们，会对Map.get(Object obj)的方法进行拦截。而且在拦截之后，直接返回了Always, 这就是最终的原因了。

罗里吧嗦一大堆，我们直接看MyBatis里面抽象的几个相关类吧：

1.  Plugin  类：也就是我们说的“插板”类， 继承了InvocationHandler 接口，主要的职责是将目标对象、拦截器组装成一个新的代理类。 简单的代码如下：

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** Plugin **implements** InvocationHandler {
3. **private** Object target; // 目标对象
4. **private** Interceptor interceptor; //拦截器对象
5. **private** Map<Class<?>, Set<Method>> signatureMap; //目标对象的方法签名
7. **private** Plugin(Object target, Interceptor interceptor, Map<Class<?>, Set<Method>> signatureMap) {
8. **this**.target = target;
9. **this**.interceptor = interceptor;
10. **this**.signatureMap = signatureMap;
11. }
13. **public** **static** Object wrap(Object target, Interceptor interceptor) {
14. Map<Class<?>, Set<Method>> signatureMap = getSignatureMap(interceptor);
15. Class<?> type = target.getClass();
16. Class<?>[] interfaces = getAllInterfaces(type, signatureMap);
17. **if** (interfaces.length > 0) {
18. **return** Proxy.newProxyInstance(  // 返回一个目标对象的代理类
19. type.getClassLoader(),
20. interfaces,
21. **new** Plugin(target, interceptor, signatureMap));
22. }
23. **return** target;
24. }
26. **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {
27. **try** {
28. Set<Method> methods = signatureMap.get(method.getDeclaringClass());
29. **if** (methods != **null** && methods.contains(method)) {
30. **return** interceptor.intercept(**new** Invocation(target, method, args));//对目标对象中的某个方法进行拦截，将相关的实现给到了拦截器的实现类
31. }
32. **return** method.invoke(target, args);
33. } **catch** (Exception e) {
34. **throw** ExceptionUtil.unwrapThrowable(e);
35. }
36. }

 2. Interceptor 拦截器的接口：

**Java代码  [收藏代码](javascript:void())**

1. **public** **interface** Interceptor {
3. Object intercept(Invocation invocation) **throws** Throwable; // 对方法进行拦截的抽象方法
5. Object plugin(Object target); //把拦截器插入到目标对象的方法
7. **void** setProperties(Properties properties);
9. }

 3. Invocation 真正对目标类方法的拦截的实现， 这里没有什么说的。 值得一提的是， 如果我们想实现类似spring的拦截器，比如说前置通知、后置通知、环绕通知等，应该是可以在这里做文章的。

**Java代码  [收藏代码](javascript:void())**

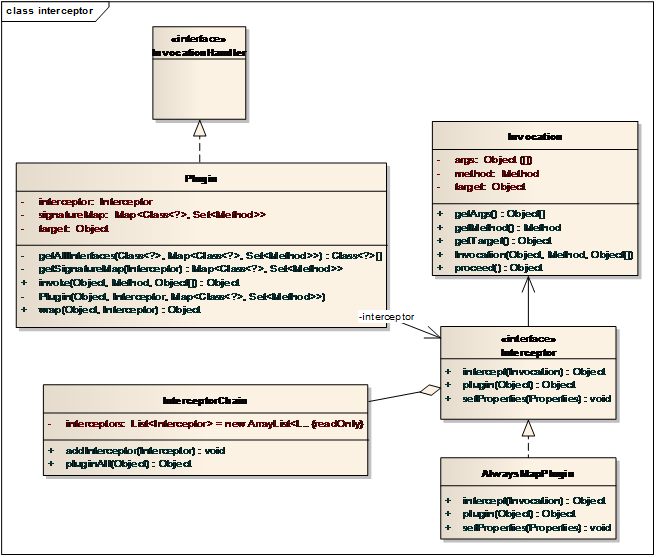
1. **public** **class** Invocation {
3. **private** Object target;
4. **private** Method method;
5. **private** Object[] args;
7. **public** Invocation(Object target, Method method, Object[] args) {
8. **this**.target = target;
9. **this**.method = method;
10. **this**.args = args;
11. }
13. **public** Object getTarget() {
14. **return** target;
15. }
17. **public** Method getMethod() {
18. **return** method;
19. }
21. **public** Object[] getArgs() {
22. **return** args;
23. }
25. **public** Object proceed() **throws** InvocationTargetException, IllegalAccessException {
26. **return** method.invoke(target, args);
27. }
29. }

 4. 搞定。

总结：其实拦截器还是很简单的， 只需要熟识了Proxy类、InvocationHandler接口， 基本都能写出来，只是写的好与坏，是不是低耦合、高内聚的代码。类的命名是不是让别人一看就懂，这些都决定着一个人水平的高低；

拦截器使用的场景： 比如说 权限、日志记录、缓存等等；

先分享到这里.....



## 2.1 Transaction

PROPAGATION\_REQUIRED--支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。

PROPAGATION\_SUPPORTS--支持当前事务，如果当前没有事务，就以非事务方式执行。

PROPAGATION\_MANDATORY--支持当前事务，如果当前没有事务，就抛出异常。

PROPAGATION\_REQUIRES\_NEW--新建事务，如果当前存在事务，把当前事务挂起。

PROPAGATION\_NOT\_SUPPORTED--以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

PROPAGATION\_NEVER--以非事务方式执行，如果当前存在事务，则抛出异常。

PROPAGATION\_NESTED--如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则进行与PROPAGATION\_REQUIRED类似的操作。

· PROPAGATION\_REQUIRED--支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。

· PROPAGATION\_SUPPORTS--支持当前事务，如果当前没有事务，就以非事务方式执行。

· PROPAGATION\_MANDATORY--支持当前事务，如果当前没有事务，就抛出异常。

· PROPAGATION\_REQUIRES\_NEW--新建事务，如果当前存在事务，把当前事务挂起。

· PROPAGATION\_NOT\_SUPPORTED--以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

第七个（**PROPAGATION\_NESTED**）是Spring所提供的一个特殊变量。 它要求事务管理器或者使用JDBC 3.0 Savepoint API提供嵌套事务行为（如Spring的DataSourceTransactionManager）

# 3、Zookeeper

## 3.1 ZK使用

### 3.1.1 ZkClient

默认zookeeper无权限认证功能, 修改些代码实现认证  
zookeeper包括四个内置的scheme，包括world,auth,digest,ip，默认为world，权限相当于任何人可操作。我们现在要做的是加入一种认证方式，使用digest这种username:password方式比较适合我们。

zkCli.sh –server 10.77.20.23:2181

addauth digest password /myconf

ls /myconf

ls2 /myconf

### 3.1.2 Curator

<http://www.big-mouth.cn/blog/65.html>

https://github.com/jamesmorgan/ReloadablePropertiesAnnotation

# 4、Maven项目管理

Compile

Provided

1. 创建web项目

mvn archetype:generate -DgroupId=com.shaobing -DartifactId=struts2-cve-2010-1870 –D archetypeArtifactId=maven-archetype-webapp -DinteractivMode=false -DarchetypeCatalog=intrenal

# 5、Dubbo

# 5、nginx防御方案

## 5.1 限制IP的请求次数以及并发次数

# 6、tomcat

**On Linux:**java.io.tmpdir: [/tmp]

vm运行时会生成一个目录hsperfdata\_$USER($USER是启动java进程的用户),在linux中默认是/tmp,目录下会有些pid文件,存放jvm进程信息,而jmap,jstack等工具会读取/tmp/hsperfdata\_$USER下的pid文件获取连接信息.  
知道这个原因后,我马上检查了/tmp/hsperfdata\_root目录,果然没有pid对应的文件,但在$TOMCAT\_HOME目录中的temp目录中有对应的文件.