Spring MVC

[环境搭建： 2](#_Toc447122895)

[POM.xml 文件 3](#_Toc447122896)

[配置分配器 dispacther（web.xml） 4](#_Toc447122897)

[配置 dispatcher-servlet.xml 前端控制器（分配器）与view视图的值传递。 5](#_Toc447122898)

[获取参数的几种方法： 7](#_Toc447122899)

# 环境搭建：

Maven 插件：eclipse marketplace m2e-intergation-for-eclipse

Properties： maven 映射本地maven包地址

设置：

**1.**build compiler set 1.7

**2.**project facets set java version 1.7;dynamic module 2.5/3.0; futher configuration avaliable,设置src/main/webapp,web-xml路径

3.新建src/test/resources,在properties->java build path/source 设置outputfolder->target/classes(test-classes)，此时包含main/java,main/resources,test/java,test/resources,**文件属性为source folder**

4.libraries包含maven+jre, order&export 要包含maven dependencies

5.depolyment assermbly 删除两个test folders

# POM.xml 文件

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jsp-api</artifactId>

<version>2.0</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.2.2.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

<version>4.2.2.RELEASE</version>

</dependency>

<dependency>

<groupId>taglibs</groupId>

<artifactId>standard</artifactId>

<version>1.1.2</version>

</dependency>

# 配置分配器 dispacther（web.xml）

<!DOCTYPE web-app PUBLIC

"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"

"http://java.sun.com/dtd/web-app\_2\_3.dtd" >

<web-app>

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<init-param>

<param-name>contextConfigLoaction</param-name>

<param-value>/WEB-INF/dispatcher-servlet.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

# 配置 dispatcher-servlet.xml 前端控制器（分配器）与view视图的值传递。

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

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:**aop**=*"http://www.springframework.org/schema/aop"*

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

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

xmlns:**tx**=*"http://www.springframework.org/schema/tx"*

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

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

*http://www.springframework.org/schema/aop/spring-aop.xsd*

*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.xsd*

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

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

*http://www.springframework.org/schema/tx*

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

<mvc:annotation-driven />

<context:component-scan base-package=*"liming.maven.example"*/>

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

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

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

</bean>

</beans>

# Controller.java

**package** liming.maven.example.view;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.ui.Model;

**import** org.springframework.web.bind.annotation.RequestMapping;

@Controller

**public** **class** GeneralController {

@RequestMapping(value="index.do")

**public** String index\_do(Model model){

model.addAttribute("liming", "hello");

System.***out***.println("hello");

**return** "hello";

}

}

# 获取参数的几种方法：

Html中：

<form method=”post” action=”index.do”>

<input type=”text” name=”id”/>

<input type=”text” name=”name”/>

<input type=”submit” value=ok/>

</form>

Controller.java中：

@requestMapping(value=”” method=requestMethod.get)

Public void excute(string id,string name,Model model)

@requestMapping(value=”” method=requestMethod.get)

Public void excute(@ModelAttribute(“bean”),Bean bean){

Bean.getid()

}

@requestMapping(value=”” method=requestMethod.get)

Public void excute(HttpServletRequest req, HttpServletResponse res){

String id=Req.getParameter(“id”);

…

}

**SpringMVC 4.2.2 - Web.xml,Dispatcher-Servlet及ApplicationContext配置笔记**

  Spring MVC 项目中常用的主要配置项Web.XML, Dispatcher-servlet.xm及applicationContext.xml项笔记。若能给需要的朋友提供参考，那胜感荣幸！  
  
**1. Web.XML**

DispatcherServlet项：

SpringMVC **DispatcherServlet**是继承自HttpServlet一个 Servlet类 ，它实现对请求进行拦截。 DispatcherServlet需要在你的应用的Web.xml声明加载。

**[html]** [view plain](http://blog.csdn.net/chendev1/article/details/50294381" \o "view plain) [copy](http://blog.csdn.net/chendev1/article/details/50294381" \o "copy)

1. **<servlet>**
2. **<servlet-name>**dispatcher**</servlet-name>**
3. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
4. **<load-on-startup>**1**</load-on-startup>**
5. **</servlet>**
6. **<servlet-mapping>**
7. **<servlet-name>**dispatcher**</servlet-name>**
8. **<url-pattern>**/**</url-pattern>**
9. **</servlet-mapping>**

默认情况，Spring MVC会在WEB-INF寻找 dispacher-servlet.xml

**servlet-name:**

以上配置中，servlet-name定义的名称为dispatcher, ,这样spring会自动扫描web-inf下名为dispatcher-servlet.xml文件。当然你可以使用其它[servet-name], 如，springmvcdispatcher, 那XML文件需要命名为：springmvcdispatcherr-servlet.xml。 里都是spring自动扫描的，如果你没有提供，将会报一个文件查找不到的异常。

**ContextLoaderListener**：

 启动**ContextLoaderListener**监听器，需要在Web.XML中添加如下配置，它会自动装配ApplicationContext的配置信息。

**[html]** [view plain](http://blog.csdn.net/chendev1/article/details/50294381" \o "view plain) [copy](http://blog.csdn.net/chendev1/article/details/50294381" \o "copy)

1. **<listener>**
2. **<listener-class>**org.springframework.web.context.ContextLoaderListener**</listener-class>**
3. **</listener>**

默认情况下会自动查找 /WEB-INF/applicationContext.xml，所以，需要在WEB-INF目录下创建的applicationContext.xml文件；文件 applicationContext.xml内容可以为空，但不能不创建这个文件， 如果没有applicationContext.xml文件，启动时会报错。

2.**Dispatcher-Servlet.XML**

* **激活@Controller注解模式**

**[html]** [view plain](http://blog.csdn.net/chendev1/article/details/50294381" \o "view plain) [copy](http://blog.csdn.net/chendev1/article/details/50294381" \o "copy)

1. **<mvc:annotation-driven** **/>**

* **设置对类的注解进行扫描，创建Bean及自动依赖注入**

**[html]** [view plain](http://blog.csdn.net/chendev1/article/details/50294381" \o "view plain) [copy](http://blog.csdn.net/chendev1/article/details/50294381" \o "copy)

1. **<context:component-scan** base-package="com.swirebev.mvc.Controller" **/>**

譬如，项目中MVC Controller包为com.kungfo.helloworld.Controller，则Spring会自动扫描包中的所有类。 根据实际项目包名修改即可。 如果有多个需要扫描的包，则可以用逗号分隔。

* **ViewResolver**

此项设置将Controller返回的viewname映射到对应的View,即jsp页面。

如，Controller 返回 viewname为“ Welcome”,则会解析到/Web-INF/jsp路径下的 **Welcome.jsp**.

**[html]** [view plain](http://blog.csdn.net/chendev1/article/details/50294381" \o "view plain) [copy](http://blog.csdn.net/chendev1/article/details/50294381" \o "copy)

1. **<bean** class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter" **/>**
2. **<bean** id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>**
3. **<property** name="prefix"**>**
4. **<value>**/WEB-INF/jsp/**</value>**
5. **</property>**
6. **<property** name="suffix"**>**
7. **<value>**.jsp**</value>**
8. **</property>**
9. **</bean>**

**3.ApplicationContext**

通常是一些bean管理，资源管理（读取资源文件），加入监听器等。

如，**加载资源文件**

**[html]** [view plain](http://blog.csdn.net/chendev1/article/details/50294381" \o "view plain) [copy](http://blog.csdn.net/chendev1/article/details/50294381" \o "copy)

1. <!-- 加载资源文件 其中包含变量信息 -->
2. **<context:property-placeholder** location="classpath:jdbc.properties" **/>**

**数据源bean:**

**[html]** [view plain](http://blog.csdn.net/chendev1/article/details/50294381" \o "view plain) [copy](http://blog.csdn.net/chendev1/article/details/50294381" \o "copy)

1. **<bean** id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method="close"**>**
2. **<property** name="driverClassName" value="${jdbc.driver}" **/>**
3. **<property** name="url" value="${jdbc.url}" **/>**
4. **<property** name="username" value="${jdbc.user}" **/>**
5. **<property** name="password" value="${jdbc.password}" **/>**
6. **<property** name="connectionProperties" value="useUnicode=yes;characterEncoding=utf8;" **/>**
7. **</bean>**

**4.供参考例子：**

   以下例子均已经验证可用。

* **Web.XML**

**[html]** [view plain](http://blog.csdn.net/chendev1/article/details/50294381" \o "view plain) [copy](http://blog.csdn.net/chendev1/article/details/50294381" \o "copy)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app**
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns="http://xmlns.jcp.org/xml/ns/javaee"
5. xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee

http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"

1. id="WebApp\_ID" version="3.1"**>**
2. **<display-name>**FrontOffice**</display-name>**

5. **<servlet>**
6. **<servlet-name>**dispatcher**</servlet-name>**
7. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
8. **<load-on-startup>**1**</load-on-startup>**
9. **</servlet>**
10. **<servlet-mapping>**
11. **<servlet-name>**dispatcher**</servlet-name>**
12. **<url-pattern>**/**</url-pattern>**
13. **</servlet-mapping>**

16. **<listener>**
17. **<listener-class>**org.springframework.web.context.ContextLoaderListener**</listener-class>**
18. **</listener>**
20. **</web-app>**

* **Dispatcher-servlet.xml**

**[html]** [view plain](http://blog.csdn.net/chendev1/article/details/50294381" \o "view plain) [copy](http://blog.csdn.net/chendev1/article/details/50294381" \o "copy)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"
4. xmlns:mvc="http://www.springframework.org/schema/mvc" xmlns:context="http://www.springframework.org/schema/context"
5. xmlns:util="http://www.springframework.org/schema/util" xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
6. http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.2.xsd
7. http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-4.2.xsd
8. http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-4.2.xsd"**>**


12. <!-- 激活注解模式，如@Controller -->
13. **<mvc:annotation-driven** **/>**
14. <!-- 对包中的类的注解进行扫描，创建Bean及自动依赖注入  -->
15. **<context:component-scan** base-package="com.swirebev.mvc.Controller" **/>**
17. **<bean** class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter" **/>**
18. **<bean** id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>**
19. **<property** name="prefix"**>**
20. **<value>**/WEB-INF/views/**</value>**
21. **</property>**
22. **<property** name="suffix"**>**
23. **<value>**.jsp**</value>**
24. **</property>**
25. **</bean>**
26. **</beans>**

* **applicationContext.xml**

**[html]** [view plain](http://blog.csdn.net/chendev1/article/details/50294381" \o "view plain) [copy](http://blog.csdn.net/chendev1/article/details/50294381" \o "copy)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:aop="http://www.springframework.org/schema/aop"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"
5. xsi:schemaLocation="
6. http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
8. http://www.springframework.org/schema/context
9. http://www.springframework.org/schema/context/spring-context-4.2.xsd
10. http://www.springframework.org/schema/aop
11. http://www.springframework.org/schema/aop/spring-aop-4.2.xsd"**>**
13. <!-- 加载资源文件 其中包含变量信息 -->
14. **<context:property-placeholder** location="classpath:jdbc.properties" **/>**
16. <!-- 数据库Datasource -->
17. **<bean** id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method="close"**>**
18. **<property** name="driverClassName" value="${jdbc.driver}" **/>**
19. **<property** name="url" value="${jdbc.url}" **/>**
20. **<property** name="username" value="${jdbc.user}" **/>**
21. **<property** name="password" value="${jdbc.password}" **/>**
22. **<property** name="connectionProperties" value="useUnicode=yes;characterEncoding=utf8;" **/>**
23. **</bean>**


27. **</beans>**

IOC

IOC（DI）：其实这个Spring架构核心的概念没有这么复杂，更不像有些书上描述的那样晦涩。java程序员都知道：java程序中的每个业务逻辑至少需要两个或以上的对象来协作完成，通常，每个对象在使用他的合作对象时，自己均要使用像new object（） 这样的语法来完成合作对象的申请工作。你会发现：对象间的耦合度高了。而IOC的思想是：Spring容器来实现这些相互依赖对象的创建、协调工作。对象只需要关系业务逻辑本身就可以了。从这方面来说，对象如何得到他的协作对象的责任被反转了（IOC、DI）。  
  
这是我对Spring的IOC的体会。DI其实就是IOC的另外一种说法。DI是由Martin Fowler 在2004年初的一篇论文中首次提出的。他总结：控制的什么被反转了？就是：获得依赖对象的方式反转了。  
  
如果对这一核心概念还不理解：这里引用一个叫Bromon的blog上找到的浅显易懂的答案：

IoC与DI

　　首先想说说IoC（Inversion of Control，控制倒转）。这是spring的核心，贯穿始终。所谓IoC，对于spring框架来说，就是由spring来负责控制对象的生命周期和对象间的关系。这是什么意思呢，举个简单的例子，我们是如何找女朋友的？常见的情况是，我们到处去看哪里有长得漂亮身材又好的mm，然后打听她们的兴趣爱好、qq号、电话号、ip号、iq号………，想办法认识她们，投其所好送其所要，然后嘿嘿……这个过程是复杂深奥的，我们必须自己设计和面对每个环节。传统的程序开发也是如此，在一个对象中，如果要使用另外的对象，就必须得到它（自己new一个，或者从JNDI中查询一个），使用完之后还要将对象销毁（比如Connection等），对象始终会和其他的接口或类藕合起来。

　　那么IoC是如何做的呢？有点像通过婚介找女朋友，在我和女朋友之间引入了一个第三者：婚姻介绍所。婚介管理了很多男男女女的资料，我可以向婚介提出一个列表，告诉它我想找个什么样的女朋友，比如长得像李嘉欣，身材像林熙雷，唱歌像周杰伦，速度像卡洛斯，技术像齐达内之类的，然后婚介就会按照我们的要求，提供一个mm，我们只需要去和她谈恋爱、结婚就行了。简单明了，如果婚介给我们的人选不符合要求，我们就会抛出异常。整个过程不再由我自己控制，而是有婚介这样一个类似容器的机构来控制。Spring所倡导的开发方式就是如此，所有的类都会在spring容器中登记，告诉spring你是个什么东西，你需要什么东西，然后spring会在系统运行到适当的时候，把你要的东西主动给你，同时也把你交给其他需要你的东西。所有的类的创建、销毁都由 spring来控制，也就是说控制对象生存周期的不再是引用它的对象，而是spring。对于某个具体的对象而言，以前是它控制其他对象，现在是所有对象都被spring控制，所以这叫控制反转。如果你还不明白的话，我决定放弃。

IoC的一个重点是在系统运行中，动态的向某个对象提供它所需要的其他对象。这一点是通过DI（Dependency Injection，依赖注入）来实现的。比如对象A需要操作数据库，以前我们总是要在A中自己编写代码来获得一个Connection对象，有了 spring我们就只需要告诉spring，A中需要一个Connection，至于这个Connection怎么构造，何时构造，A不需要知道。在系统运行时，spring会在适当的时候制造一个Connection，然后像打针一样，注射到A当中，这样就完成了对各个对象之间关系的控制。A需要依赖 Connection才能正常运行，而这个Connection是由spring注入到A中的，依赖注入的名字就这么来的。那么DI是如何实现的呢？ Java 1.3之后一个重要特征是反射（reflection），它允许程序在运行的时候动态的生成对象、执行对象的方法、改变对象的属性，spring就是通过反射来实现注入的。关于反射的相关资料请查阅java doc。  
　理解了IoC和DI的概念后，一切都将变得简单明了，剩下的工作只是在spring的框架中堆积木而已。  
  
如果还不明白，放弃java吧！

**下面来让大家了解一下Spring到底是怎么运行的。**

****[java]**** [view plain](http://blog.csdn.net/it_man/article/details/4402245" \o "view plain) [copy](http://blog.csdn.net/it_man/article/details/4402245" \o "copy)

1. **public static void main(String[] args) {**
2. **ApplicationContext context = new FileSystemXmlApplicationContext(**
3. "applicationContext.xml"**);**
4. **Animal animal = (Animal) context.getBean(**"animal"**);**
5. **animal.say();**
6. **}**

这段代码你一定很熟悉吧，不过还是让我们分析一下它吧，首先是applicationContext.xml

1. **<bean id=**"animal"**class=**"phz.springframework.test.Cat"**>**
2. **<property name=**"name"**value=**"kitty"**/>**
3. **</bean>**

他有一个类phz.springframework.test.Cat

Java代码

1. **public class Cat implements Animal {**
2. **private String name;**
3. **public void say() {**
4. **System.out.println(**"I am "**+ name +**"!"**);**
5. **}**
6. **public void setName(String name) {**
7. **this.name = name;**
8. **}**
9. **}**

实现了phz.springframework.test.Animal接口

[view plain](http://blog.csdn.net/it_man/article/details/4402245" \o "view plain) [copy](http://blog.csdn.net/it_man/article/details/4402245" \o "copy)

1. **public interface Animal {**
2. **public void say();**
3. **}**

很明显上面的代码输出I am kitty!   
那么到底Spring是如何做到的呢？   
接下来就让我们自己写个Spring 来看看Spring 到底是怎么运行的吧！   
首先，我们定义一个Bean类，这个类用来存放一个Bean拥有的属性

1. /\* Bean Id \*/
2. **private String id;**
3. /\* Bean Class \*/
4. **private String type;**
5. /\* Bean Property \*/
6. **private Map<String, Object> properties = new HashMap<String, Object>();**

一个Bean包括id,type,和Properties。   
  
接下来Spring 就开始加载我们的配置文件了，将我们配置的信息保存在一个HashMap中，HashMap的key就是Bean 的 Id ，HasMap 的value是这个Bean，只有这样我们才能通过context.getBean("animal")这个方法获得Animal这个类。我们都知道Spirng可以注入基本类型，而且可以注入像List，Map这样的类型，接下来就让我们以Map为例看看Spring是怎么保存的吧   
  
Map配置可以像下面的

1. **<bean id=**"test"**class=**"Test"**>**
2. **<property name=**"testMap"**>**
3. **<map>**
4. **<entry key=**"a"**>**
5. **<value>**1**</value>**
6. **</entry>**
7. **<entry key=**"b"**>**
8. **<value>**2**</value>**
9. **</entry>**
10. **</map>**
11. **</property>**
12. **</bean>**

Spring是怎样保存上面的配置呢？，代码如下：

1. **if (beanProperty.element(**"map"**) != null) {**
2. **Map<String, Object> propertiesMap = new HashMap<String, Object>();**
3. **Element propertiesListMap = (Element) beanProperty**
4. **.elements().get(**0**);**
5. **Iterator<?> propertiesIterator = propertiesListMap**
6. **.elements().iterator();**
7. **while (propertiesIterator.hasNext()) {**
8. **Element vet = (Element) propertiesIterator.next();**
9. **if (vet.getName().equals(**"entry"**)) {**
10. **String key = vet.attributeValue(**"key"**);**
11. **Iterator<?> valuesIterator = vet.elements()**
12. **.iterator();**
13. **while (valuesIterator.hasNext()) {**
14. **Element value = (Element) valuesIterator.next();**
15. **if (value.getName().equals(**"value"**)) {**
16. **propertiesMap.put(key, value.getText());**
17. **}**
18. **if (value.getName().equals(**"ref"**)) {**
19. **propertiesMap.put(key, new String[] { value**
20. **.attributeValue(**"bean"**) });**
21. **}**
22. **}**
23. **}**
24. **}**
25. **bean.getProperties().put(name, propertiesMap);**
26. **}**

接下来就进入最核心部分了，让我们看看Spring 到底是怎么依赖注入的吧，其实依赖注入的思想也很简单，它是通过反射机制实现的，在实例化一个类时，它通过反射调用类中set方法将事先保存在HashMap中的类属性注入到类中。让我们看看具体它是怎么做的吧。   
首先实例化一个类，像这样

[copy](http://blog.csdn.net/it_man/article/details/4402245" \o "copy)

1. **public static Object newInstance(String className) {**
2. **Class<?> cls = null;**
3. **Object obj = null;**
4. **try {**
5. **cls = Class.forName(className);**
6. **obj = cls.newInstance();**
7. **} catch (ClassNotFoundException e) {**
8. **throw new RuntimeException(e);**
9. **} catch (InstantiationException e) {**
10. **throw new RuntimeException(e);**
11. **} catch (IllegalAccessException e) {**
12. **throw new RuntimeException(e);**
13. **}**
14. **return obj;**
15. **}**

接着它将这个类的依赖注入进去，像这样

1. **public static void setProperty(Object obj, String name, String value) {**
2. **Class<? extends Object> clazz = obj.getClass();**
3. **try {**
4. **String methodName = returnSetMthodName(name);**
5. **Method[] ms = clazz.getMethods();**
6. **for (Method m : ms) {**
7. **if (m.getName().equals(methodName)) {**
8. **if (m.getParameterTypes().length ==**1**) {**
9. **Class<?> clazzParameterType = m.getParameterTypes()[**0**];**
10. **setFieldValue(clazzParameterType.getName(), value, m,**
11. **obj);**
12. **break;**
13. **}**
14. **}**
15. **}**
16. **} catch (SecurityException e) {**
17. **throw new RuntimeException(e);**
18. **} catch (IllegalArgumentException e) {**
19. **throw new RuntimeException(e);**
20. **} catch (IllegalAccessException e) {**
21. **throw new RuntimeException(e);**
22. **} catch (InvocationTargetException e) {**
23. **throw new RuntimeException(e);**
24. **}**
25. **}**

最后它将这个类的实例返回给我们，我们就可以用了。我们还是以Map为例看看它是怎么做的，我写的代码里面是创建一个HashMap并把该HashMap注入到需要注入的类中，像这样，

[copy](http://blog.csdn.net/it_man/article/details/4402245" \o "copy)

1. **if (value instanceof Map) {**
2. **Iterator<?> entryIterator = ((Map<?, ?>) value).entrySet()**
3. **.iterator();**
4. **Map<String, Object> map = new HashMap<String, Object>();**
5. **while (entryIterator.hasNext()) {**
6. **Entry<?, ?> entryMap = (Entry<?, ?>) entryIterator.next();**
7. **if (entryMap.getValue() instanceof String[]) {**
8. **map.put((String) entryMap.getKey(),**
9. **getBean(((String[]) entryMap.getValue())[**0**]));**
10. **}**
11. **}**
12. **BeanProcesser.setProperty(obj, property, map);**
13. **}**

**AOP**

此前对于AOP的使用仅限于声明式事务，除此之外在实际开发中也没有遇到过与之相关的问题。最近项目中遇到了以下几点需求，仔细思考之后，觉得采用AOP 来解决。一方面是为了以更加灵活的方式来解决问题，另一方面是借此机会深入学习Spring AOP相关的内容。本文是权当本人的自己AOP学习笔记，以下需求不用AOP肯定也能解决，至于是否牵强附会，仁者见仁智者见智。

1. 对部分函数的调用进行日志记录，用于观察特定问题在运行过程中的函数调用情况
2. 监控部分重要函数，若抛出指定的异常，需要以短信或邮件方式通知相关人员
3. 金控部分重要函数的执行时间

    事实上，以上需求没有AOP也能搞定，只是在实现过程中比较郁闷摆了。

1. 需要打印日志的函数分散在各个包中，只能找到所有的函数体，手动添加日志。然而这些日志都是临时的，待问题解决之后应该需要清除打印日志的代码，只能再次手动清除^\_^!
2. 类 似1的情况，需要捕获异常的地方太多，如果手动添加时想到很可能明天又要手动清除，只能再汗。OK，该需求相对比较固定，属于长期监控的范畴，并不需求临 时添加后再清除。然而，客户某天要求，把其中20%的异常改为短信提醒，剩下的80%改用邮件提醒。改之，两天后，客户抱怨短信太多，全部改成邮件提 醒...
3. 该需求通常用于监控某些函数的执行时间，用以判断系统执行慢的瓶颈所在。瓶颈被解决之后，烦恼同情况1

    终于下定决心，采用AOP来解决！代码如下：

    切面类TestAspect

**Java代码  [IMG_256](http://pandonix.iteye.com/blog/336873/javascript:void())**

1. **package** com.spring.aop;
2. /\*\*
3. \* 切面
4. \*
5. \*/
6. **public** **class** TestAspect {
8. **public** **void** doAfter(JoinPoint jp) {
9. System.out.println("log Ending method: "
10. + jp.getTarget().getClass().getName() + "."
11. + jp.getSignature().getName());
12. }
14. **public** Object doAround(ProceedingJoinPoint pjp) **throws** Throwable {
15. **long** time = System.currentTimeMillis();
16. Object retVal = pjp.proceed();
17. time = System.currentTimeMillis() - time;
18. System.out.println("process time: " + time + " ms");
19. **return** retVal;
20. }
22. **public** **void** doBefore(JoinPoint jp) {
23. System.out.println("log Begining method: "
24. + jp.getTarget().getClass().getName() + "."
25. + jp.getSignature().getName());
26. }
28. **public** **void** doThrowing(JoinPoint jp, Throwable ex) {
29. System.out.println("method " + jp.getTarget().getClass().getName()
30. + "." + jp.getSignature().getName() + " throw exception");
31. System.out.println(ex.getMessage());
32. }
34. **private** **void** sendEx(String ex) {
35. //TODO 发送短信或邮件提醒
36. }
37. }

**Java代码  [IMG_257](http://pandonix.iteye.com/blog/336873/javascript:void())**

1. **package** com.spring.service;
2. /\*\*
3. \* 接口A
4. \*/
5. **public** **interface** AService {
7. **public** **void** fooA(String \_msg);
9. **public** **void** barA();
10. }

**Java代码  [IMG_258](http://pandonix.iteye.com/blog/336873/javascript:void())**

1. **package** com.spring.service;
2. /\*\*
3. \*接口A的实现类
4. \*/
5. **public** **class** AServiceImpl **implements** AService {
7. **public** **void** barA() {
8. System.out.println("AServiceImpl.barA()");
9. }
11. **public** **void** fooA(String \_msg) {
12. System.out.println("AServiceImpl.fooA(msg:"+\_msg+")");
13. }
14. }

**Java代码  [IMG_259](http://pandonix.iteye.com/blog/336873/javascript:void())**

1. **package** com.spring.service;
3. /\*\*
4. \*   Service类B
5. \*/
6. **public** **class** BServiceImpl {
8. **public** **void** barB(String \_msg, **int** \_type) {
9. System.out.println("BServiceImpl.barB(msg:"+\_msg+" type:"+\_type+")");
10. **if**(\_type == 1)
11. **throw** **new** IllegalArgumentException("测试异常");
12. }
14. **public** **void** fooB() {
15. System.out.println("BServiceImpl.fooB()");
16. }
18. }

    ApplicationContext

**Java代码  [IMG_260](http://pandonix.iteye.com/blog/336873/javascript:void())**

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:aop="http://www.springframework.org/schema/aop"
5. xsi:schemaLocation="
6. http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
8. http://www.springframework.org/schema/aop
9. http://www.springframework.org/schema/aop/spring-aop-2.5.xsd"
10. **default**-autowire="autodetect">
11. <aop:config>
12. <aop:aspect id="TestAspect" ref="aspectBean">
13. <!--配置com.spring.service包下所有类或接口的所有方法-->
14. <aop:pointcut id="businessService"
15. expression="execution(\* com.spring.service.\*.\*(..))" />
16. <aop:before pointcut-ref="businessService" method="doBefore"/>
17. <aop:after pointcut-ref="businessService" method="doAfter"/>
18. <aop:around pointcut-ref="businessService" method="doAround"/>
19. <aop:after-throwing pointcut-ref="businessService" method="doThrowing" throwing="ex"/>
20. </aop:aspect>
21. </aop:config>
23. <bean id="aspectBean" **class**="com.spring.aop.TestAspect" />
24. <bean id="aService" **class**="com.spring.service.AServiceImpl"></bean>
25. <bean id="bService" **class**="com.spring.service.BServiceImpl"></bean>
27. </beans>

    测试类AOPTest

**Java代码  [IMG_261](http://pandonix.iteye.com/blog/336873/javascript:void())**

1. **public** **class** AOPTest **extends** AbstractDependencyInjectionSpringContextTests {
3. **private** AService aService;
5. **private** BServiceImpl bService;
7. **protected** String[] getConfigLocations() {
8. String[] configs = **new** String[] { "/applicationContext.xml"};
9. **return** configs;
10. }

13. /\*\*
14. \* 测试正常调用
15. \*/
16. **public** **void** testCall()
17. {
18. System.out.println("SpringTest JUnit test");
19. aService.fooA("JUnit test fooA");
20. aService.barA();
21. bService.fooB();
22. bService.barB("JUnit test barB",0);
23. }
25. /\*\*
26. \* 测试After-Throwing
27. \*/
28. **public** **void** testThrow()
29. {
30. **try** {
31. bService.barB("JUnit call barB",1);
32. } **catch** (IllegalArgumentException e) {
34. }
35. }
37. **public** **void** setAService(AService service) {
38. aService = service;
39. }
41. **public** **void** setBService(BServiceImpl service) {
42. bService = service;
43. }
44. }

    运行结果如下：

**Java代码  [IMG_262](http://pandonix.iteye.com/blog/336873/javascript:void())**

1. log Begining method: com.spring.service.AServiceImpl.fooA
2. AServiceImpl.fooA(msg:JUnit test fooA)
3. log Ending method: com.spring.service.AServiceImpl.fooA
4. process time: 0 ms
5. log Begining method: com.spring.service.AServiceImpl.barA
6. AServiceImpl.barA()
7. log Ending method: com.spring.service.AServiceImpl.barA
8. process time: 0 ms
9. log Begining method: com.spring.service.BServiceImpl.fooB
10. BServiceImpl.fooB()
11. log Ending method: com.spring.service.BServiceImpl.fooB
12. process time: 0 ms
13. log Begining method: com.spring.service.BServiceImpl.barB
14. BServiceImpl.barB(msg:JUnit test barB type:0)
15. log Ending method: com.spring.service.BServiceImpl.barB
16. process time: 0 ms
18. log Begining method: com.spring.service.BServiceImpl.barB
19. BServiceImpl.barB(msg:JUnit call barB type:1)
20. log Ending method: com.spring.service.BServiceImpl.barB
21. method com.spring.service.BServiceImpl.barB **throw** exception
22. 测试异常

    《Spring参考手册》中定义了以下几个AOP的重要概念，结合以上代码分析如下：

* 切面（Aspect） ：官方的抽象定义为“一个关注点的模块化，这个关注点可能会横切多个对象”，在本例中，“切面”就是类TestAspect所关注的具体行为，例如，AServiceImpl.barA()的调用就是切面TestAspect所关注的行为之一。“切面”在ApplicationContext中<aop:aspect>来配置。
* 连接点（Joinpoint） ：程序执行过程中的某一行为，例如，AServiceImpl.barA()的调用或者BServiceImpl.barB(String \_msg, int \_type)抛出异常等行为。
* 通知（Advice） ：“切面”对于某个“连接点”所产生的动作，例如，TestAspect中对com.spring.service包下所有类的方法进行日志记录的动作就是一个Advice。其中，一个“切面”可以包含多个“Advice”，例如TestAspect
* 切入点（Pointcut） ：匹配连接点的断言，在AOP中通知和一个切入点表达式关联。例如，TestAspect中的所有通知所关注的连接点，都由切入点表达式execution(\* com.spring.service.\*.\*(..))来决定
* 目标对象（Target Object） ：被一个或者多个切面所通知的对象。例如，AServcieImpl和BServiceImpl，当然在实际运行时，Spring AOP采用代理实现，实际AOP操作的是TargetObject的代理对象。
* AOP代理（AOP Proxy） 在Spring AOP中有两种代理方式，JDK动态代理和CGLIB代理。默认情况下，TargetObject实现了接口时，则采用JDK动态代理，例如，AServiceImpl；反之，采用CGLIB代理，例如，BServiceImpl。强制使用CGLIB代理需要将 <aop:config> 的 proxy-target-class 属性设为true

       通知（Advice）类型

* 前置通知（Before advice） ：在某连接点（JoinPoint）之前执行的通知，但这个通知不能阻止连接点前的执行。ApplicationContext中在<aop:aspect>里面使用<aop:before>元素进行声明。例如，TestAspect中的doBefore方法
* 后通知（After advice） ：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。ApplicationContext中在<aop:aspect>里面使用<aop:after>元素进行声明。例如，TestAspect中的doAfter方法，所以AOPTest中调用BServiceImpl.barB抛出异常时，doAfter方法仍然执行
* 返回后通知（After return advice） ：在某连接点正常完成后执行的通知，不包括抛出异常的情况。ApplicationContext中在<aop:aspect>里面使用<after-returning>元素进行声明。
* 环绕通知（Around advice） ：包围一个连接点的通知，类似Web中Servlet规范中的Filter的doFilter方法。可以在方法的调用前后完成自定义的行为，也可以选择不执行。ApplicationContext中在<aop:aspect>里面使用<aop:around>元素进行声明。例如，TestAspect中的doAround方法。
* 抛出异常后通知（After throwing advice） ： 在方法抛出异常退出时执行的通知。 ApplicationContext中在<aop:aspect>里面使用<aop:after-throwing>元素进行声明。例如，TestAspect中的doThrowing方法。

       切入点表达式

* 通常情况下，表达式中使用”execution“就可以满足大部分的要求。表达式格式如下：

**Java代码  [IMG_263](http://pandonix.iteye.com/blog/336873/javascript:void())**

1. execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) **throws**-pattern?)

modifiers-pattern：方法的操作权限

ret-type-pattern：返回值

declaring-type-pattern：方法所在的包

name-pattern：方法名

parm-pattern：参数名

throws-pattern：异常

其中，除ret-type-pattern和name-pattern之外，其他都是可选的。上例中，execution(\* com.spring.service.\*.\*(..))表示com.spring.service包下，返回值为任意类型；方法名任意；参数不作限制的所有方法。

* 通知参数

可以通过args来绑定参数，这样就可以在通知（Advice）中访问具体参数了。例如，<aop:aspect>配置如下

**Java代码  [IMG_264](http://pandonix.iteye.com/blog/336873/javascript:void())**

1. <aop:config>
2. <aop:aspect id="TestAspect" ref="aspectBean">
3. <aop:pointcut id="businessService"
4. expression="execution(\* com.spring.service.\*.\*(String,..)) and args(msg,..)" />
5. <aop:after pointcut-ref="businessService" method="doAfter"/>
6. </aop:aspect>
7. </aop:config>
8. <bean id =”businessService” class = .../>
9. <bean id =”aspectBean” class = .../>

TestAspect的doAfter方法中就可以访问msg参数，但这样以来AService中的barA()和BServiceImpl中的barB()就不再是连接点，因为execution(\* com.spring.service.\*.\*(String,..))只配置第一个参数为String类型的方法。其中，doAfter方法定义如下：

**Java代码  [IMG_265](http://pandonix.iteye.com/blog/336873/javascript:void())**

1. **public** **void** doAfter(JoinPoint jp,String msg)

* 访问当前的连接点

任何通知（Advice）方法可以将第一个参数定义为 org.aspectj.lang.JoinPoint 类型。JoinPoint 接口提供了一系列有用的方法， 比如 getArgs() （返回方法参数）、getThis() （返回代理对象）、getTarget() （返回目标）、getSignature() （返回正在被通知的方法相关信息）和 toString()（打印出正在被通知的方法的有用信息。

**Advisor 型aop**

和上述aspect型的区别在于，advisor型aop 的通知advice类 继承

前置增强

**MethodBeforeAdvice.before**（Method method，objects[] obj, object target）

后置增强

**AfterReturningAdvice.afterReturning**（Object returnObj ，Method method，objects[] obj, object target）

**MethodInterceptor.invoke**(MethodInvocation invocation)

**ThrowAdvice.afterThrowing**(Method method，objects[] obj, object target,Exception ex)

**码  [IMG_264](http://pandonix.iteye.com/blog/336873/javascript:void())**

1. <aop:config>
2. <aop:pointcut id="businessService"
3. expression="execution(\* com.spring.service.\*.\*(String,..)) and args(msg,..)" />
4. <aop:advicor pointcut-ref="businessService" advice-ref="advicebean"/>
5. </aop:config>

**JDBC**

Jdbctemplate 配置

线程安全

全程可用一个bean

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"  
 destroy-method="close"  
 p:driverClassName="${jdbc.driverClassName}"  
 p:url="${jdbc.url}"  
 p:username="${jdbc.username}"  
 p:password="${jdbc.password}"  
/>

String sqlStr = " SELECT user\_id,user\_name "  
 + " FROM t\_user WHERE user\_name =? ";

//List<User> users = new ArrayList<User>();  
jdbcTemplate.query(sqlStr, new Object[]{userName},  
 new RowCallbackHandler() {  
 public void processRow(ResultSet rs) throws SQLException {

//User user = new User();  
 user.setUserId(rs.getInt("user\_id"));  
 user.setUserName(userName);

//users.add(user);  
 }  
 });