# 1 Spring IoC

以下内容基于spring 3.2版本

## 1.1 Spring IoC概述

Spring框架对控制反转(Inversion of Control，IoC)[1]理论的实现。

org.springframework.beans及org.springframework.context包 是Spring IoC容器的基础。BeanFactory 提供的高级配置机制，使得管理各种对象成为可能。 ApplicationContext 是BeanFactory的子接口，功能得到了进一步增强，比如更易 与Spring AOP集成、资源处理(国际化处理)、事件传递及各种不同应用层的上下文(context)实现 (如针对web应用的WebApplicationContext)。

简而言之，BeanFactory提供了配制框架及基本功能，而 ApplicationContext 则增加了更多支持企业核心内容的功能。 ApplicationContext完全由BeanFactory扩展而来， 因而BeanFactory所具备的能力和行为也适用于ApplicationContext。

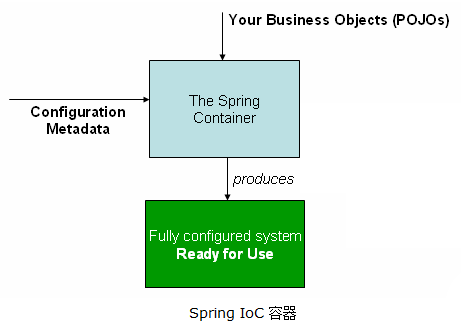
**Bean**

在Spring中，那些组成你应用程序的主体(backbone)及由Spring IoC容器所管理的对象，被称之为bean。 简单地讲，bean就是由Spring容器初始化、装配及管理的对象，除此之外，bean就与应用程序中的其他对象没有什么区别了。 而bean定义以及bean相互间的依赖关系将通过配置元数据来描述。

## 1.2 Spring IoC容器

org.springframework.context.ApplicationContext接口代表了Spring IoC容器，负责实例化、配置和装配Bean。容器通过读取XML或JAVA注解配置的元数据实现实例、装配对象及对象之间的依赖关系。

一般称BeanFactory为IoC容器，而称ApplicationContext为应用上下文(Spring容器)。



### 1.2.1 配置元数据

配置元数据传统上是以直观的XML格式提供，但是基于XML 的元数据并不是唯一的配置元数据的格式，还有以下格式

* 基于注解的配置：Spring 2.5引入基于注解元数据的支持。
* 基于Java的配置：Spring 3.0开始，由Spring JavaConfig项目提供的特性称为Spring Framework的核心。使用Java代码而不是XML文件，可参考@Configuration，@Bean，@Import等注解。

### 1.2.2 BeanFactory和ApplicationContext

beans包提供了以编程的方式管理和操控bean的基本功能，而context包下的ApplicationContext以一种更加面向框架的方式增强了BeanFactory的功能。多数用户可以采用声明的方式来使用ApplicationContext，甚至不用手动创建它，而通过ContextLoader这样的支持类，把它作为J2EE web应用的一部分自动启动。当然，我们仍然可以采用编程的方式创建一个ApplicationContext。

context包的核心是ApplicationContext接口。它由BeanFactory接口派生而来，因而提供了BeanFactory所有的功能。为了以一种更向面向框架的方式工作以及对上下文进行分层和实现继承，context包还提供了以下的功能：

* MessageSource, 提供国际化的消息访问
* 资源访问，如URL和文件
* 事件传播，实现了ApplicationListener接口的bean
* 载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层

**BeanFactory与ApplicationContext比较**

简单的说：除非你有更好的理由，否则尽量使用ApplicationContext。

| **特性** | **BeanFactory** | **ApplicationContext** |
| --- | --- | --- |
| Bean 实例化/装配 | Yes | Yes |
| 自动 BeanPostProcessor 注册 | No | Yes |
| 自动 BeanFactoryPostProcessor 注册 | No | Yes |
| 便捷的 MessageSource 访问( i18n) | No | Yes |
| ApplicationEvent 发送 | No | Yes |

### 1.2.3 ApplicationContext在Web应用实例化

ContextLoader机制有两种方式，ContextLoaderListener 和ContextLoaderServlet，他们功能相同但是listener不能在Servlet2.3容器下使用。Servlet2.4规范中servlet context listeners需要在web应用启动并能处理初始请求时立即运行。(servlet context listener关闭的时候也是相同的)。servlet context listener是初始化Spring ApplicationContext理想的方式。你可能愿意选择ContextLoaderListener，虽然是一样的，但决定权在于你。你可以查看ContextLoaderServlet的Javadoc来获得更详细的信息。

示例：使用ContextLoaderListener注册一个ApplicationContext

context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

*<!-- or use the ContextLoaderServlet instead of the above listener*

<servlet>

<servlet-name>context</servlet-name>

<servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>

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

</servlet>

-->

监听器首先检查contextConfigLocation参数，如果它不存在，它将使用/WEB-INF/applicationContext.xml作为默认值。如果已存在，它将使用分隔符（逗号、冒号或空格）将字符串分解成应用上下文件位置路径。可以支持ant-风格的路径模式，如/WEB-INF/\*Context.xml(WEB-INF文件夹下所有以"Context.xml"结尾的文件)。或者/WEB-INF/\*\*/\*Context.xml(WEB-INF文件夹及子文件夹下的以"Context.xml"结尾的文件)。

ContextLoaderServlet同ContextLoaderListener一样使用'contextConfigLocation'参数。

### 1.2.4利用MessageSource实现国际化

ApplicationContext接口扩展了MessageSource接口，因而提供了消息处理的功能（i18n或者国际化）。与HierarchicalMessageSource一起使用，它还能够处理嵌套的消息，这些是Spring提供的处理消息的基本接口。让我们快速浏览一下它所定义的方法：

* String getMessage(String code, Object[] args, String default, Locale loc)：用来从MessageSource获取消息的基本方法。如果在指定的locale中没有找到消息，则使用默认的消息。args中的参数将使用标准类库中的MessageFormat来作消息中替换值。
* String getMessage(String code, Object[] args, Locale loc)：本质上和上一个方法相同，其区别在：没有指定默认值，如果没找到消息，会抛出一个NoSuchMessageException异常。
* String getMessage(MessageSourceResolvable resolvable, Locale locale)：上面方法中所使用的属性都封装到一个MessageSourceResolvable实现中，而本方法可以指定MessageSourceResolvable实现。

当一个ApplicationContext被加载时，它会自动在context中查找已定义为MessageSource类型的bean。此bean的名称须为messageSource。如果找到，那么所有对上述方法的调用将被委托给该bean。否则ApplicationContext会在其父类中查找是否含有同名的bean。如果有，就把它作为MessageSource。如果它最终没有找到任何的消息源，一个空的StaticMessageSource将会被实例化，使它能够接受上述方法的调用。

Spring目前提供了两个MessageSource的实现:ResourceBundleMessageSource和StaticMessageSource。它们实现了HierarchicalMessageSource以便能够处理嵌套的消息。StaticMessageSource很少被使用，但能以编程的方式向消息源添加消息。ResourceBundleMessageSource会用得更多一些，为此提供了一下示例：

<beans>

*<!-- this MessageSource is being used in a web application -->*

<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">

<property name="basename" value="test-messages"/>

</bean>

*<!-- lets inject the above MessageSource into this POJO -->*

<bean id="example" class="com.foo.Example">

<property name="messages" ref="messageSource"/>

</bean>

</beans>

**public** **class** Example {

**private** MessageSource messages;

**public** **void** setMessages(MessageSource messages) {

**this**.messages = messages;

}

**public** **void** execute() {

String message = **this**.messages.getMessage("argument.required",

**new** Object [] {"userDao"}, "Required", null);

System.out.println(message);

}

}

结果：The userDao argument is required.

### 1.2.5 标准和自定义事件

ApplicationContext中的事件处理是通过ApplicationEvent类和ApplicationListener接口来提供的。如果在上下文中部署一个实现了ApplicationListener接口的bean，那么每当一个ApplicationEvent发布到ApplicationContext时，这个bean就得到通知。实质上，这是标准的Observer设计模式。Spring内置事件：

| **事件** | **解释** |
| --- | --- |
| ContextRefreshedEvent | 当ApplicationContext初始化或刷新时发送的事件。如使用ConfigurableApplicationContext接口的refresh()方法。这里的初始化意味着：所有的bean被装载，后处理bean被检测和激活，singleton被预实例化，以及ApplicationContext已就绪可用。刷新在context关闭会触发多次。选择ApplicationContext可以提供“热”刷新的功能(如：XmlWebApplicationContext 可以但是GenericApplicationContext 则不可以.) |
| ContextStartedEvent | 当ApplicationContext启动时发送的事件,使用ConfigurableApplicationContext接口的start()方法。这里"启动"意味着生命周期 beans将获得一个确实的启动信号。这经常使用在确实停止后重新启动的场合，但也可以用在启动一个没有被配置为自动启动的组件中(如：在完成初始化后还没有启动)。 |
| ContextStoppedEvent | 当使用ConfigurableApplicationContext接口的stop() 方法使ApplicationContext停止时候发送的事件。这里"停止"意味着生命周期 beans将获得一个确实的停止信号. 停止的context可以通过调用start()来重新启动。 |
| ContextClosedEvent | 当使用ConfigurableApplicationContext接口close() 方法使ApplicationContext 关闭时候发送的事件。 这里关闭意味着所有的singleton bean都被销毁。关闭的context不能刷新和重新启动。 |
| RequestHandledEvent | web特性的事件通告所有的bean有一个http request（将在request结束后才会发送）。注意这种事件只兼容于使用SpringDispatcherServlet兼容的web应用。 |

你可以创建和发布自定义的事件，如下实例

**public** **class** BlackListEvent **extends** ApplicationEvent {

**private** **final** String address;

**private** **final** String test;

**public** BlackListEvent(Object source, String address, String test) {

**super**(source);

**this**.address = address;

**this**.test = test;

}

*// accessor and other methods...*

}

注意：Sping的事件机制设计为在同一个Application context的Spring beans之间的通讯。因此，对于很复杂的企业级综合需求

Spring's eventing mechanism is designed for simple communication between Spring beans within the same application context. However, for more sophisticated enterprise integration needs, the separately-maintained Spring Integration project provides complete support for building lightweight, pattern-oriented, event-driven architectures that build upon the well-known Spring programming model.

## 1.3 依赖

典型的企业应用不会只由单一的对象（或Spring的术语bean)组成。毫无疑问，即使最简单的系统也需要多个对象共同来展示给用户一个整体的应用。接下来的的内容除了阐述如何单独定义一系列bean外，还将描述如何让这些bean对象一起协同工作来实现一个完整的真实应用。

### 1.3.1 依赖注入

依赖注入（DI）背后的基本原理是对象之间的依赖关系（即一起工作的其它对象）只会通过以下几种方式来实现：构造器的参数、工厂方法的参数，或给由构造函数或者工厂方法创建的对象设置属性。因此，容器的工作就是创建bean时注入那些依赖关系。相对于由bean自己来控制其实例化、直接在构造器中指定依赖关系或者类似服务定位器（Service Locator）模式这3种自主控制依赖关系注入的方法来说，控制从根本上发生了倒转，这也正是控制反转（Inversion of Control， IoC） 名字的由来。

应用DI原则后，代码将更加清晰。而且当bean自己不再担心对象之间的依赖关系（甚至不知道依赖的定义指定地方和依赖的实际类）之后，实现更高层次的松耦合将易如反掌。DI主要有两种注入方式，即Setter注入和构造器注入

**依赖注入示例**

注意：详细可参考spring-beans schema

<bean id="exampleBean" class="examples.ExampleBean">

*<!-- setter injection using the nested <ref/> element -->*

<property name="beanOne"><ref bean="anotherExampleBean"/></property>

*<!-- setter injection using the neater 'ref' attribute -->*

<property name="beanTwo" ref="yetAnotherBean"/>

<property name="integerProperty" value="1"/>

</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>

<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

**public** **class** ExampleBean {

**private** AnotherBean beanOne;

**private** YetAnotherBean beanTwo;

**private** **int** i;

**public** **void** setBeanOne(AnotherBean beanOne) {

**this**.beanOne = beanOne;

}

**public** **void** setBeanTwo(YetAnotherBean beanTwo) {

**this**.beanTwo = beanTwo;

}

**public** **void** setIntegerProperty(**int** i) {

**this**.i = i;

}

}

**循环依赖**

在采用构造器注入的方式配置bean时，很有可能会产生循环依赖的情况。

比如说，一个类A，需要通过构造器注入类B，而类B又需要通过构造器注入类A。如果为类A和B配置的bean被互相注入的话，那么Spring IoC容器将检测出循环引用，并抛出 BeanCurrentlyInCreationException异常。

对于此问题，一个可能的解决方法就是修改源代码，将某些构造器注入改为setter注入。另一个解决方法就是完全放弃构造器注入，只使用setter注入。换句话说，除了极少数例外，大部分的循环依赖都是可以避免的，不过采用setter注入产生循环依赖的可能性也是存在的。

与通常我们见到的非循环依赖的情况有所不同，在两个bean之间的循环依赖将导致一个bean在被完全初始化的时候被注入到另一个bean中（如同我们常说的先有蛋还是先有鸡的情况）。

**构造器注入还是Setter注入?**

由于大量的构造器参数可能使程序变得笨拙，特别是当某些属性是可选的时候。因此通常情况下，Spring开发团队提倡使用setter注入。而且setter DI在以后的某个时候还可将实例重新配置（或重新注入）（JMX MBean就是一个很好的例子）。

尽管如此，构造器注入还是得到很多纯化论者（也有很好的理由）的青睐。一次性将所有依赖注入的做法意味着，在未完全初始化的状态下，此对象不会返回给客户代码（或被调用），此外对象也不需要再次被重新配置（或重新注入）。

对于注入类型的选择并没硬性的规定。只要能适合你的应用，无论使用何种类型的DI都可以。对于那些没有源代码的第三方类，或者没有提供setter方法的遗留代码，我们则别无选择－－构造器注入将是你唯一的选择。

### 1.3.2 依赖配置详解

JavaBean的PropertyEditor可以用来转换这些字符串值从String到属性或参数的真实类型。

<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

*<!-- results in a setDriverClassName(String) call -->*

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql://localhost:3306/mydb"/>

<property name="username" value="root"/>

<property name="password" value="masterkaoli"/>

</bean>

对于更简洁的XML配置，可使用p-命名空间。如下示例

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

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

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

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

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

<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close"

p:driverClassName="com.mysql.jdbc.Driver"

p:url="jdbc:mysql://localhost:3306/mydb"

p:username="root"

p:password="masterkaoli"/>

</beans>

**idref元素**

idref元素用来将容器内其它bean的id传给<constructor-arg/> 或 <property/>元素，同时提供错误验证功能。

<bean id="theTargetBean" class="..."/>

<bean id="theClientBean" class="...">

<property name="targetName">

<idref bean="theTargetBean" />

</property>

</bean>

上述bean定义片段完全地等同于（在运行时）以下的片段：

<bean id="theTargetBean" class="..." />

<bean id="client" class="...">

<property name="targetName" value="theTargetBean" />

</bean>

第一种形式比第二种更可取的主要原因是：使用idref标记允许容器在部署时验证所被引用的bean是否存在。而第二种方式中，传给client bean的targetName属性值并没有被验证。任何的输入错误仅在client bean实际实例化时才会被发现（可能伴随着致命的错误）。

**引用其它的bean（协作者）**

在<constructor-arg/>或<property/>元素内部还可以使用ref元素。该元素用来将bean中指定属性的值设置为对容器中的另外一个bean的引用。如前所述，该引用bean将被作为依赖注入，而且在注入之前会被初始化（如果是singleton bean则已被容器初始化）。尽管都是对另外一个对象的引用，但是通过id/name指向另外一个对象却有三种不同的形式，不同的形式将决定如何处理作用域及验证。

第一种形式也是最常见的形式是通过使用<ref/>标记指定bean属性的目标bean，通过该标签可以引用同一容器或父容器内的任何bean（无论是否在同一XML文件中）。XML 'bean'元素的值既可以是指定bean的id值也可以是其name值。

<ref bean="someBean"/>

第二种形式是使用ref的local属性指定目标bean，它可以利用XML解析器来验证所引用的bean是否存在同一文件中。local属性值必须是目标bean的id属性值。如果在同一配置文件中没有找到引用的bean，XML解析器将抛出一个例外。如果目标bean是在同一文件内，使用local方式就是最好的选择（为了尽早地发现错误）。

<ref local="someBean"/>

第三种方式是通过使用ref的parent属性来引用当前容器的父容器中的bean。parent属性值既可以是目标bean的id值，也可以是name属性值。而且目标bean必须在当前容器的父容器中。使用parent属性的主要用途是为了用某个与父容器中的bean同名的代理来包装父容器中的一个bean(例如，子上下文中的一个bean定义覆盖了他的父bean)。

*<!—在父上下文中 -->*

<bean id="accountService" class="com.foo.SimpleAccountService">

*<!—在这里插入所需的依赖 -->*

</bean>

*<!—在子（后继）上下文中 -->*

<bean id="accountService" *<-- bean名称和父bean相同 -->*

class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="target">

<ref parent="accountService"/>

*<!—注意我们如何引用父bean -->*

</property>

*<!-- insert other configuration and dependencies as required here -->*

</bean>

**内部bean**

一个bean的<property/>或 <constructor-arg/>元素中使用<bean/>元素定义的bean。内部bean定义不需要有id或name属性，即使指定id 或 name属性值也将会被容器忽略，也会忽略scope属性。也不能作为协作Bean

<bean id="outer" class="...">

*<!-- instead of using a reference to a target bean, simply define the target bean inline -->*

<property name="target">

<bean class="com.example.Person"> *<!-- this is the inner bean -->*

<property name="name" value="Fiona Apple"/>

<property name="age" value="25"/>

</bean>

</property>

</bean>

**集合**

通过<list/>、<set/>、<map/>及<props/>元素可以定义和设置与Java Collection类型对应List、Set、Map及Properties的值。

<bean id="moreComplexObject" class="example.ComplexObject">

*<!-- results in a setAdminEmails(java.util.Properties) call -->*

<property name="adminEmails">

<props>

<prop key="administrator">administrator@example.org</prop>

<prop key="support">support@example.org</prop>

<prop key="development">development@example.org</prop>

</props>

</property>

*<!-- results in a setSomeList(java.util.List) call -->*

<property name="someList">

<list>

<value>a list element followed by a reference</value>

<ref bean="myDataSource" />

</list>

</property>

*<!-- results in a setSomeMap(java.util.Map) call -->*

<property name="someMap">

<map>

<entry key="an entry" value="just some string"/>

<entry key ="a ref" value-ref="myDataSource"/>

</map>

</property>

*<!-- results in a setSomeSet(java.util.Set) call -->*

<property name="someSet">

<set>

<value>just some string</value>

<ref bean="myDataSource" />

</set>

</property>

</bean>

### 1.3.3初始化Bean

ApplicationContext实现的默认行为就是在启动时将所有singleton bean提前进行实例化。如果你不想让一个singleton bean在ApplicationContext初始化时被提前实例化，可以将Bean设置为延迟实例化，延迟实例化是告诉IoC容器在该Bean第一次被请求时再来实例化。

<bean id="lazy" class="com.foo.ExpensiveToCreateBean" **lazy-init="true"**/>

<bean name="not.lazy" class="com.foo.AnotherBean"/>

如果一个bean被设置为延迟初始化，而另一个非延迟初始化的singleton bean依赖于它，那么当ApplicationContext提前实例化singleton bean时，它必须也确保所有上述singleton 依赖bean也被预先初始化，当然也包括设置为延迟实例化的bean。因此，如果Ioc容器在启动的时候创建了那些设置为延迟实例化的bean的实例，你也不要觉得奇怪，因为那些延迟初始化的bean可能在配置的某个地方被注入到了一个非延迟初始化singleton bean里面。

* Bean默认为Singleton Bean，IoC容器启动时进行Bean的实例化
* 如果Singleton Bean为延迟加载，IoC容器在该Bean第一次被请求时实例化
* 对于prototype的Bean，则是在getBean的时候被实例化

### 1.3.4 自动装配（autowire）协作者

Spring IoC容器可以自动装配（autowire）相互协作bean之间的关联关系。如下为自动装配模式。

| **模式** | **说明** |
| --- | --- |
| no | 默认情况，没有自动装配。Bean的引用必须通过ref元素来定义。对于大型的部署，修改默认设置是不推荐的，因为明确地指定协作者会给予更多的控制和清晰。 |
| byName | 根据属性名自动装配。此选项将检查容器并根据名字查找与属性完全一致的bean，并将其与属性自动装配。例如，在bean定义中将autowire设置为by name，而该bean包含*master*属性（同时提供*setMaster(..)*方法），Spring就会查找名为master的bean定义，并用它来装配给master属性。 |
| byType | 如果容器中存在一个与指定属性类型相同的bean，那么将与该属性自动装配。如果存在多个该类型的bean，那么将会抛出异常，并指出不能使用*byType*方式进行自动装配。若没有找到相匹配的bean，则什么事都不发生，属性也不会被设置。如果你不希望这样，那么可以通过设置dependency-check="objects"让Spring抛出异常。 |
| constructor | 与*byType*的方式类似，不同之处在于它应用于构造器参数。如果在容器中没有找到与构造器参数类型一致的bean，那么将会抛出异常。 |

### 1.3.5 不同生命周期Bean的注入

在大部分情况下，容器中的bean都是singleton类型的。如果一个singleton bean要引用另外一个singleton bean，或者一个非singleton bean要引用另外一个非singleton bean时，通常情况下将一个bean定义为另一个bean的property值就可以了。不过对于具有不同生命周期的bean来说这样做就会有问题了，比如在调用一个singleton类型bean A的某个方法时，需要引用另一个非singleton（prototype）类型的bean B，对于bean A来说，容器只会创建一次，这样就没法在需要的时候每次让容器为bean A提供一个新的的bean B实例。

**方法注入**

第一种解决办法就是放弃控制反转。通过实现ApplicationContext接口让bean A能够感知bean 容器，并且在需要的时候通过使用getBean("B")方式向容器请求一个新的bean B实例。看下下面这个例子，其中故意使用了这种方法：

*// a class that uses a stateful Command-style class to perform some processing*

**package** fiona.apple;

*// 导入了Spring APIS*

**import** org.springframework.beans.BeansException;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.ApplicationContextAware;

**public** **class** CommandManager **implements** ApplicationContextAware {

**private** ApplicationContext applicationContext;

**public** Object process(Map commandState) {

*// grab a new instance of the appropriate Command*

Command command = createCommand();

*// set the state on the (hopefully brand new) Command instance*

command.setState(commandState);

**return** command.execute();

}

**protected** Command createCommand() {

*// notice the Spring API dependency!*

**return** **this**.applicationContext.getBean("command", Command.**class**);

}

**public** **void** setApplicationContext(ApplicationContext applicationContext)

**throws** BeansException {

**this**.applicationContext = applicationContext;

}

}

上面的例子显然不是最好的，因为业务代码和Spring Framework产生了耦合。方法注入，作为Spring IoC容器的一种高级特性，可以以一种干净的方法来处理这种情况。

**Lookup方法注入**

第二种方法，Lookup方法注入利用了容器的覆盖受容器管理的bean方法的能力，从而返回指定名字的bean实例。在上述场景中，Lookup方法注入适用于原型bean。Lookup方法注入的内部机制是Spring利用了CGLIB库在运行时生成二进制代码功能，通过动态创建Lookup方法bean的子类而达到复写Lookup方法的目的。

**package** fiona.apple;

*// no more Spring imports!*

**public** **abstract** **class** CommandManager {

**public** Object process(Object commandState) {

*// grab a new instance of the appropriate Command interface*

Command command = createCommand();

*// set the state on the (hopefully brand new) Command instance*

command.setState(commandState);

**return** command.execute();

}

*// okay... but where is the implementation of this method?*

**protected** **abstract** Command createCommand();

}

<public|protected> [abstract] <return-type> theMethodName(*no-arguments*);

如果方法是abstract的，动态生成的子类就实现这个方法。

*<!-- a stateful bean deployed as a prototype (non-singleton) -->*

<bean id="command" class="fiona.apple.AsyncCommand" scope="prototype">

*<!-- inject dependencies here as required -->*

</bean>

*<!-- commandProcessor uses statefulCommandHelper -->*

<bean id="commandManager" class="fiona.apple.CommandManager">

<lookup-method name="createCommand" bean="command"/>

</bean>

注意：要让动态子类起作用，在classpat下必须有CGLIB的jar文件，容器中的子类不能是final类型的，要被覆盖的方法也不能是final类型。方法注入的目标对象不能被序列化。

## 1.4 Bean的作用域

创建一个bean定义，其实质是用该bean定义对应的类来创建真正实例的“配方(recipe)”。把bean定义看成一个配方很有意义，它与class很类似，只根据一张“处方”就可以创建多个实例。

你不仅可以控制注入到对象中的各种依赖和配置值，还可以控制该对象的作用域。这样你可以灵活选择所建对象的作用域，而不必在Java Class级定义作用域。Spring Framework支持五种作用域（其中有三种只能用在基于web的Spring ApplicationContext）。

| **作用域** | **描述** |
| --- | --- |
| [singleton](mk:@MSITStore:E:\learn\开源框架\spring\Spring2.5-中文参考手册.chm::/ch03s04.html#beans-factory-scopes-singleton) | 在每个Spring IoC容器中一个bean定义对应一个对象实例。 |
| [prototype](mk:@MSITStore:E:\learn\开源框架\spring\Spring2.5-中文参考手册.chm::/ch03s04.html#beans-factory-scopes-prototype) | 一个bean定义对应多个对象实例。 |
| [request](mk:@MSITStore:E:\learn\开源框架\spring\Spring2.5-中文参考手册.chm::/ch03s04.html#beans-factory-scopes-request) | 在一次HTTP请求中，一个bean定义对应一个实例；即每次HTTP请求将会有各自的bean实例， 它们依据某个bean定义创建而成。该作用域仅在基于web的Spring ApplicationContext情形下有效。 |
| [session](mk:@MSITStore:E:\learn\开源框架\spring\Spring2.5-中文参考手册.chm::/ch03s04.html#beans-factory-scopes-global-session) | 在一个HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。 |
| [global session](mk:@MSITStore:E:\learn\开源框架\spring\Spring2.5-中文参考手册.chm::/ch03s04.html#beans-factory-scopes-global-session) | 在一个全局的HTTP Session中，一个bean定义对应一个实例。典型情况下，仅在使用portlet context的时候有效。该作用域仅在基于web的Spring ApplicationContext情形下有效。 |

### 1.4.1 Singleton作用域

## 1.5 基于注解的配置

要使注解可用，您必须使用 Java 5 (Tiger)或更新的版本，以使得可以访问源代码层次的注解。这些注解可以被注册为独立 bean 的定义，但它们也可以被隐式地注册，通过基于 XML 的配置方式，如下例（请注意包含 'context' 命名空间）：

注意：注解注入会在XML注入之前执行，因此通过两种方式，那么XML配置会覆盖注解的配置。

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

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

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

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

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

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

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

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

*<context:annotation-config/>*

</beans>

（隐式注册 post-processors 包括了 AutowiredAnnotationBeanPostProcessor，CommonAnnotationBeanPostProcessor，PersistenceAnnotationBeanPostProcessor及 RequiredAnnotationBeanPostProcessor。）

注意：<context:annotation-config/>仅仅查找定义在同一个Application Context的Bean的注解。这就意味着，如果你为DispatcerServlet将<context:annotation-config/>放置在WebApplictionContext中，那么它仅仅检查控制器中的@Autowired bean，其它的bean（如服务层bean）不会检查。

### 1.5.1 @Required

@Required注解应用于bean属性的setter方法中，表明受影响Bean的属性必须在Bean的定义中或者自动装配中填充。如果受影响的Bean属性没有被填充，容器会抛出异常。

**public** **class** SimpleMovieLister {

**private** MovieFinder movieFinder;

*@Autowired*

**public** **void** setMovieFinder(MovieFinder movieFinder) {

**this**.movieFinder = movieFinder;

}

*// ...*

}

### 1.5.2 @Component注解

@Component是所有受Spring管理组件的通用形式； 而@Repository、@Service和 @Controller则是@Component的细化， 用来表示更具体的用例：

@Repositor：持久化层，用它来标记充当储存库(又称 Data Access Object或DAO)

@Service：服务层

@Controller：表现层

也就是说，用@Componen来注解你的组件类，但如果用@Repository、@Service 或@Controller来注解它们，你的类也许能更好地被工具处理，或与切面进行关联。 例如，这些典型化注解可以成为理想的切入点目标。当然，在Spring Framework以后的版本中， @Repository、@Service和 @Controller也许还能携带更多语义。如此一来，如果你正在考虑服务层中是该用 @Component还是@Service， 那@Service显然是更好的选择。同样的，就像前面说的那样， @Repository已经能在持久化层中进行异常转换时被作为标记使用了。

## 1.6 类路径扫描和组件管理

通过扫描类路径的方式来隐式检测候选组件，并在容器中注册对于的Bean定义，代替XML来执行Bean的注册。可以使用注解(eg：@Component)，AspectJ风格的表达式来选择容器中注册的Bean。

### 1.6.1 自动检测类和Bean的注册

Spring可以自动检测“被典型化”(stereotyped)的类，在ApplicationContext中注册对应的BeanDefinition。下面就是自动检测的例子：

*@Service*

**public** **class** SimpleMovieLister {

**private** MovieFinder movieFinder;

*@Autowired*

**public** SimpleMovieLister(MovieFinder movieFinder) {

**this**.movieFinder = movieFinder;

}

}

*@Repository*

**public** **class** JpaMovieFinder **implements** MovieFinder {

*// implementation elided for clarity*

}

要自动检测这些类并注册对应的Bean，你需要包含如下XML元素，其中base-package 元素通常是类的父包。（多个父包可以用逗号隔开）

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

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

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

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

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

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:component-scan base-package="org.example"/>

</beans>

注意：对类路径扫描需要存在类路径下对应的目录。当年使用Ant来构建JAR包时，要保证你没有激活JAR目标中的files-only开关。

当你使用component-scan时，AutowiredAnnotationBeanPostProcessor和CommonAnnotationBeanPostProcessor 两者是隐式包含的。两个组将被自动检测之后就装配在一起了，而不需要在XML中提供Bean的配置元素。

注意：你可以使用annotation-config 属性置为false来关闭AutowiredAnnotationBeanPostProcessor 的 CommonAnnotationBeanPostProcessor 注册。

### 1.6.2 使用过滤器自定义扫描

默认情况下，用@Component、 @Repository、@Service或 @Controller (或本身使用了@Component注解的自定义注解) 注解的类是唯一会被检测到的候选组件。但是可以很方便地通过自定义过滤器来改变并扩展这一行为。 可以用'component-scan'的include-filter或 exclude-filter子元素来进行添加。 每个过滤器元素都要求有'type'和'expression'属性。

| **过滤器类型** | **表达式范例** |
| --- | --- |
| annotation | org.example.SomeAnnotation |
| assignable | org.example.SomeClass |
| regex | org\.example\.Default.\* |
| aspectj | org.example..\*Service+ |

# 2 Spring MVC

以下内容基于spring 3.2版本

## 2.1 Spring MVC简介

Spring的web模块提供了大量独特的功能，包括:

* 清晰的角色划分：控制器（controller）、验证器（validator）、 命令对象（command object）、表单对象（form object）、模型对象（model object）、 Servlet分发器（DispatcherServlet）、 处理器映射（handler mapping）、视图解析器（view resolver）等等。 每一个角色都可以由一个专门的对象来实现。
* 强大而直接的配置方式：将框架类和应用程序类都能作为JavaBean配置，支持跨多个context的引用，例如，在web控制器中对业务对象和验证器（validator）的引用。
* 可适配、非侵入：可以根据不同的应用场景，选择合适的控制器子类 （simple型、command型、form型、wizard型、multi-action型或者自定义），而不是从单一控制器 （比如Action/ActionForm）继承。
* 可重用的业务代码：可以使用现有的业务对象作为命令或表单对象，而不需要去扩展某个特定框架的基类。
* 可定制的绑定（binding） 和验证（validation）：比如将类型不匹配作为应用级的验证错误， 这可以保存错误的值。再比如本地化的日期和数字绑定等等。在其他某些框架中，你只能使用字符串表单对象， 需要手动解析它并转换到业务对象。
* 可定制的handler mapping和view resolution：Spring提供从最简单的URL映射， 到复杂的、专用的定制策略。与某些web MVC框架强制开发人员使用单一特定技术相比，Spring显得更加灵活。
* 灵活的model转换：在Springweb框架中，使用基于Map的 键/值对来达到轻易地与各种视图技术的集成。
* 可定制的本地化和主题（theme）解析：支持在JSP中可选择地使用Spring标签库、支持JSTL、支持Velocity（不需要额外的中间层）等等。
* 简单而强大的JSP标签库（Spring Tag Library）：支持包括诸如数据绑定和主题（theme） 之类的许多功能。它提供在标记方面的最大灵活性。如欲了解详情，请参阅附录附录 D, spring.tld
* JSP表单标签库：在Spring2.0中引入的表单标签库，使得在JSP中编写 表单更加容易。如欲了解标签库详情，请参阅附录附录 E, spring-form.tld
* Spring Bean的生命周期可以被限制在当前的HTTP Request或者HTTP Session。 准确的说，这并非Spring MVC框架本身特性，而应归属于Sping MVC使用的WebApplicationContext容器。 该功能在第 3.4.4 节 “其他作用域”有详细描述。

Spring MVC是当前最优秀的MVC框架，自从Spring 2.5版本发布后，由于支持注解配置，易用性有了大幅度的提高。Spring 3.0更加完善，实现了对Struts 2的超越。现在越来越多的开发团队选择了Spring MVC。

Spring3 MVC的优点：

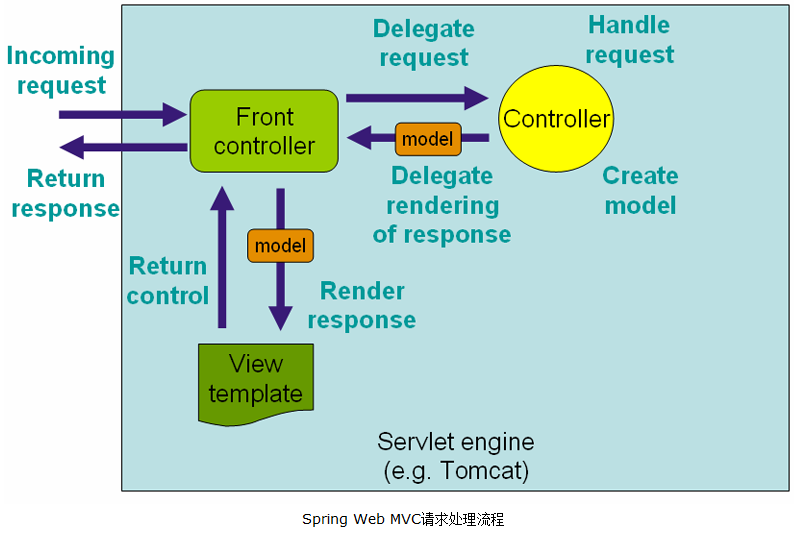
1、Spring3 MVC使用简单，学习成本低。学习难度小于Struts2，Struts2用不上的多余功能太多。呵呵，当然这不是决定因素。

2、Spring3 MVC很容易就可以写出性能优秀的程序，Struts2要处处小心才可以写出性能优秀的程序（指MVC部分）

3、Spring3 MVC的灵活是你无法想像的，Spring框架的扩展性有口皆碑，Spring3 MVC当然也不会落后，不会因使用了MVC框架而感到有任何的限制。

## 2.2 DispatcherServlet

Spring的web MVC框架是一个请求驱动的web框架，其设计围绕一个中心的servlet进行， 它能将请求分发给控制器，并提供其它功能帮助web应用开发。然而，Spring的DispatcherServlet 所做的不仅仅是这些，它和Spring的IoC容器完全集成在一起，从而允许你使用Spring的其它功能。下图展示了Spring Web MVC DispatcherServlet处理请求的流程。 熟悉设计模式的读者可能会发现DispatcherServlet应用了“ Front Controller”模式



DispatcherServlet实际上是一个Servlet （它继承了HttpServlet）。与其它Servlet一样， DispatcherServlet定义在web应用的web.xml文件中。 DispatcherServlet处理的请求必须在同一个web.xml文件里使用url-mapping定义映射

*<servlet>*

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

*<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>*

*<init-param>*

*<param-name>contextConfigLocation</param-name>*

*<param-value>classpath\*:/springMVC.xml</param-value>*

*</init-param>*

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

*</servlet>*

*<servlet-mapping>*

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

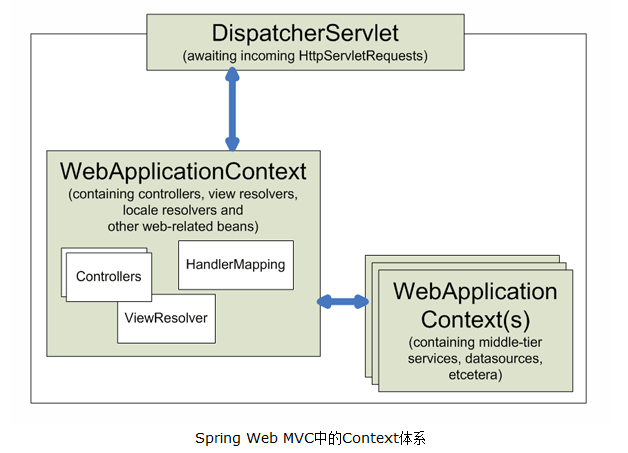
*<url-pattern>/</url-pattern>*

*</servlet-mapping>*

其中<param-value>\*\*.xml</param-value> 支持多种写法  
1、不写,使用默认值:/WEB-INF/<servlet-name>-servlet.xml  
2、<param-value>/WEB-INF/classes/springMVC.xml</param-value>  
3、<param-value>classpath\*:springMVC-mvc.xml</param-value>  
4、多个值用逗号分隔

* 在DispatcherServlet的初始化过程中，框架会在web应用的 WEB-INF文件夹下寻找名为[servlet-name]-servlet.xml 的配置文件，生成文件中定义的bean。这些bean会覆盖在全局范围（global cope）中定义的同名的bean。

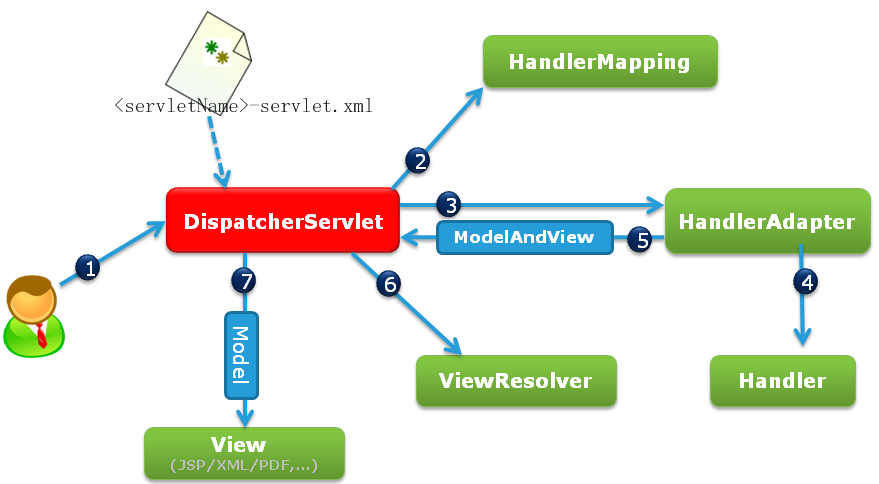
Spring中的 ApplicationContext实例可以被限制在不同的作用域（scope）中。在web MVC框架中，每个 DispatcherServlet有它自己的WebApplicationContext ，这个context继承了根 WebApplicationContext 的所有bean定义。这些继承的bean也可以在每个serlvet自己的所属的域中被覆盖（override），覆盖后的bean 可以被设置成只有这个servlet实例自己才可以使用的属性。



DispatcherServlet配置完成后，当相应的请求到达时，处理就开始了。下面的描述了DispatcherServlet处理请求的全过程：

1. 找到WebApplicationContext并将其绑定到请求的一个属性上， 以便控制器和处理链上的其它处理器能使用WebApplicationContext。 默认的属性名为DispatcherServlet.WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE。
2. 将本地化解析器（localResolver）绑定到请求上，这样使得处理链上的处理器在处理请求（准备数据、显示视图等等） 时能进行本地化处理。若不使用本地化解析器，也不会有任何副作用，因此如果不需要本地化解析，忽略它即可。
3. 将主题解析器绑定到请求上，这样视图可以决定使用哪个主题。如果你不需要主题，可以忽略它，不会有任何影响。
4. 如果上传文件解析器被指定，Spring会检查每个接收到的请求是否存在上传文件，如果存在， 这个请求将被封装成MultipartHttpServletRequest以便被处理链中的其它处理器使用
5. 找到合适的处理器，执行和这个处理器相关的执行链（预处理器，后置处理器，控制器），以便为视图准备模型数据（用于渲染）。
6. 如果模型数据被返回，就使用配置在WebApplicationContext中的视图解析器显示视图， 否则视图不会被显示。有多种原因可以导致返回的数据模型为空，比如预处理器或后处理器可能截取了请求，这可能是出于安全原因， 也可能是请求已经被处理，没有必要再处理一次。

Spring MVC处理流程如下：



Spring MVC流程描述如下：

1. 用户向服务器发送请求，请求被Spring 前端控制Servelt DispatcherServlet捕获；

2. DispatcherServlet对请求URL进行解析，得到请求资源标识符（URI）。然后根据该URI，调用HandlerMapping获得该Handler配置的所有相关的对象（包括Handler对象以及Handler对象对应的拦截器），最后以HandlerExecutionChain对象的形式返回；

3. DispatcherServlet 根据获得的Handler，选择一个合适的HandlerAdapter。（附注：如果成功获得HandlerAdapter后，此时将开始执行拦截器的preHandler(...)方法）

4. 提取Request中的模型数据，填充Handler入参，开始执行Handler（Controller)。 在填充Handler的入参过程中，根据你的配置，Spring将帮你做一些额外的工作：

HttpMessageConveter： 将请求消息（如Json、xml等数据）转换成一个对象，将对象转换为指定的响应信息

数据转换：对请求消息进行数据转换。如String转换成Integer、Double等

数据根式化：对请求消息进行数据格式化。 如将字符串转换成格式化数字或格式化日期等

数据验证： 验证数据的有效性（长度、格式等），验证结果存储到BindingResult或Error中

5. Handler执行完成后，向DispatcherServlet 返回一个ModelAndView对象；

6. 根据返回的ModelAndView，选择一个适合的ViewResolver（必须是已经注册到Spring容器中的ViewResolver)返回给DispatcherServlet ；

7. ViewResolver 结合Model和View，来渲染视图

8. 将渲染结果返回给客户端。

**表 2.1 WebApplicationContext中特殊的bean**

| Bean类型 | **描述** |
| --- | --- |
| 控制器（Controllers） | [控制器](mk:@MSITStore:E:\learn\开源框架\spring\Spring2.5-中文参考手册.chm::/ch13s03.html)实现的是MVC中的C 。 |
| 处理器映射（Handler mapping） | [处理器映射](mk:@MSITStore:E:\learn\开源框架\spring\Spring2.5-中文参考手册.chm::/ch13s04.html)包含预处理器（pre-processor）， 后置处理器（post-processor）和控制器的列表，它们在符合某种条件时才被执行（例如符合控制器指定的URL）。 |
| 视图解析器（View resolvers） | [视图解析器](mk:@MSITStore:E:\learn\开源框架\spring\Spring2.5-中文参考手册.chm::/ch13s05.html)可以将视图名解析为对应的视图。 |
| 本地化解析器（Locale resolver） | [本地化解析器](mk:@MSITStore:E:\learn\开源框架\spring\Spring2.5-中文参考手册.chm::/ch13s06.html)能够解析用户正在使用的本地化设置，以提供国际化视图。 |
| 主题解析器（Theme resolver） | [主题解析器](mk:@MSITStore:E:\learn\开源框架\spring\Spring2.5-中文参考手册.chm::/ch13s07.html)能够解析你的web应用所使用的主题，以提供个性化的布局。 |
| 文件上传解析器（Multipart File resolver） | [文件上传解析器](mk:@MSITStore:E:\learn\开源框架\spring\Spring2.5-中文参考手册.chm::/ch13s08.html)提供HTML表单文件上传功能。 |
| 处理器异常解析器(Handler exception resolver(s)) | [处理器异常解析器](mk:@MSITStore:E:\learn\开源框架\spring\Spring2.5-中文参考手册.chm::/ch13s10.html)可以将异常对应到视图，或者实现更加复杂的异常处理逻辑。 |
| FlashMapManager | 用于传递Request 的attributes到另外一个Request，通常用于重定向 |

**表2.2 DispatcherServlet初始化参数**

| **参数** | **描述** |
| --- | --- |
| contextClass | 实现WebApplicationContext接口的类，当前的servlet用它来创建上下文。如果这个参数没有指定， 默认使用XmlWebApplicationContext。 |
| contextConfigLocation | 传给上下文实例（由contextClass指定）的字符串，用来指定上下文的位置。这个字符串可以被分成多个字符串（使用逗号作为分隔符） 来支持多个上下文（在多上下文的情况下，如果同一个bean被定义两次，后面一个优先）。 |
| namespace | WebApplicationContext命名空间。默认值是[server-name]-servlet。 |

Spring MVC配置样例文件

## 2.3控制器

控制器的概念是MVC设计模式的一部分（MVC中的*C*）。应用程序的行为通常被定义为服务接口， 而控制器使得用户可以访问应用所提供的服务。控制器解析用户输入，并将其转换成合理的模型数据，从而可以进一步由视图展示给用户。 Spring以一种抽象的方式实现了控制器概念，这样可以支持不同类型的控制器。Spring本身包含表单控制器、命令控制器、向导型控制器等多种多样的控制器。

### 2.3.1 基于注解的控制器

Spring 2.5为MVC控制器引入了一种基于注解的编程模型，在其中使用诸如@RequestMapping、@RequestParam、@ModelAttribute，等等。 这种注解支持在Servlet MVC和Portlet MVC中均可使用。通过这种方式实现的控制器不必由特定的基类继承而来，或者实现特定的接口。 更进一步的，它们通常并不直接依赖于Servlet或Portlet API。

**使用@Controller定义一个控制器**

注解@Controller指明一个特定的类承担控制器的职责， 而没有扩展任何控制器基类或者引用Servlet API的必要。当然，如果需要还是可以引用特定Servlet功能。

所注解的控制器bean可以被显式定义，这个过程是在dispatcher的上下文中使用一个标准的Spring bean定义完成的。 然而，@Controller原型也允许自动探测，就像Spring 2.5对探测组件的类以及为它们自动注册bean定义的普遍支持一样。

要实现对这样的所注解的控制器的自动探测，必须要向配置中加入组件扫描的部分。

<!-- 自动扫描的包名 -->

<context:component-scan base-package=*"com.springmvc.web"*/>

**使用@RequestMapping映射请求**

注解@RequestMapping被用于映射如“/editPet.do”这样的URL到一个完整的类或者一个特定的处理方法。 典型的，顶层的注解映射一个特定的请求路径（或者路径模式）到一个表单控制器，另外的方法一级的注解可以缩小这个主要映射的范围，包括对于一个特定的HTTP请求方法（“GET/POST”）或者特定的HTTP请求参数。

*@Controller*

**@RequestMapping("/appointments")**

**public** **class** AppointmentsController {

**private** **final** AppointmentBook appointmentBook;

*@Autowired*

**public** AppointmentsController(AppointmentBook appointmentBook) {

**this**.appointmentBook = appointmentBook;

}

**@RequestMapping(method = RequestMethod.GET)**

**public** Map<String, Appointment> get() {

**return** appointmentBook.getAppointmentsForToday();

}

**@RequestMapping(value="/{day}", method = RequestMethod.GET)**

**public** Map<String, Appointment> getForDay(*@PathVariable* *@DateTimeFormat(iso=ISO.DATE)* Date day, Model model) {

**return** appointmentBook.getAppointmentsForDay(day);

}

**@RequestMapping(value="/new", method = RequestMethod.GET)**

**public** AppointmentForm getNewForm() {

**return** **new** AppointmentForm();

}

**@RequestMapping(method = RequestMethod.POST)**

**public** String add(*@Valid* AppointmentForm appointment, BindingResult result) {

**if** (result.hasErrors()) {

**return** "appointments/new";

}

appointmentBook.addAppointment(appointment);

**return** "redirect:/appointments";

}

}

对于一个传统的multi-action控制器，由于控制器会响应多个URL，URL就通常被直接映射到方法上。 下面是一个使用了@RequestMapping的multi-action控制器的例子

*@Controller*

**public** **class** ClinicController {

**private** **final** Clinic clinic;

*@Autowired*

**public** ClinicController(Clinic clinic) {

**this**.clinic = clinic;

}

**@RequestMapping("/")**

**public** **void** welcomeHandler() {

}

**@RequestMapping("/vets")**

**public** ModelMap vetsHandler() {

**return** **new** ModelMap(**this**.clinic.getVets());

}

}

## 2.4 处理映射器

通过处理器映射，可以将web请求映射到正确的处理器（handler）上。Spring内置了很多处理器映射策略，HandlerMapping的基本功能是将请求传递到HandlerExecutionChain上。

首先，这个HandlerExecutionChain必须包含一个能处理该请求的处理器。 其次，这个链也可以包含一系列可以拦截请求的拦截器。 当收到请求时，DispatcherServlet将请求交给处理器映射，让它检查请求并找到一个适当的HandlerExecutionChain。

然后，DispatcherServlet执行定义在链中的处理器和拦截器（interceptor）。

使用基于注解的Controller，RequestMappingHandlerMapping自动地查找所有包含@RequestMapping注解的@Controller Bean。所有的处理器映射（HandlerMapping）都是AbstractHandlerMapping的子类，以下属性可根据需要自行定制。

* interceptors：在映射中使用的拦截器列表。
* defaultHandler：缺省的处理器。 当没有合适的处理器可以匹配请求时，该处理器就会被使用。
* order： 根据每个映射的order属性值 (由org.springframework.core.Ordered 接口定义)，Spring 将上下文中可用的映射进行排序，然后选用第一个和请求匹配的处理器。
* alwaysUseFullPath：如果这个属性被设成true，Spring 将会使用绝对路径在当前的servlet context中寻找合适的处理器。 这个属性的默认值是false，在这种情况下，Spring会使用当前servlet context中的相对路径。 例如，如果一个servlet在servlet-mapping中用的值是/testing/\*，当alwaysUseFullPath 设成true时， 处理器映射中的URL格式应该使用/testing/viewPage.html，当这个属性设成false，同一个URL应该写成 /viewPage.html。
* urlDecode：这个属性的默认值是true，和2.5版本一样。 如果想比较编码后的路径，可以把这个属性设为false。 不过，需要注意的是，HttpServletRequest总是返回解码后的servlet路径， 与编码后的格式进行比较时可能不会匹配。
* lazyInitHandlers：这个属性允许设置是否延迟singleton处理器的初始化工作（prototype处理器的初始化都是延迟的)。 这个属性的默认值是false。

如下示例中所有的HttpServletRequest请求都会被officeHoursInterceptor 拦截器拦截：

<beans>

<bean id="handlerMapping" class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping">

<property name="interceptors">

<list>

<ref bean="officeHoursInterceptor"/>

</list>

</property>

</bean>

<bean id="officeHoursInterceptor"

class="samples.TimeBasedAccessInterceptor">

<property name="openingTime" value="9"/>

<property name="closingTime" value="18"/>

</bean>

<beans>

**package** samples;

**public** **class** TimeBasedAccessInterceptor **extends** HandlerInterceptorAdapter {

**private** **int** openingTime;

**private** **int** closingTime;

**public** **void** setOpeningTime(**int** openingTime) {

**this**.openingTime = openingTime;

}

**public** **void** setClosingTime(**int** closingTime) {

**this**.closingTime = closingTime;

}

**public** **boolean** preHandle(

HttpServletRequest request,

HttpServletResponse response,

Object handler) **throws** Exception {

Calendar cal = Calendar.getInstance();

**int** hour = cal.get(HOUR\_OF\_DAY);

**if** (openingTime <= hour && hour < closingTime) {

**return** true;

} **else** {

response.sendRedirect("http://host.com/outsideOfficeHours.html");

**return** false;

}

}

}

## 2.5 视图解析

所有web应用的MVC框架都有它们定位视图的方式。 Spring提供了视图解析器供你在浏览器显示模型数据，而不必被束缚在特定的视图技术上。 Spring内置了对JSP，Velocity模版和XSLT视图的支持。

ViewResolver和View是Spring的视图处理方式中特别重要的两个接口。 ViewResolver提供了从视图名称到实际视图的映射。 View处理请求的准备工作，并将该请求提交给某种具体的视图技术。

### 2.5.1视图解析器

Spring Web MVC框架的所有控制器都必须返回一个视图名，明确地（By returning a String、View or ModelAndView）或暗示的（Base on conventions）。 Sprnig中的视图以名字为标识，视图解析器通过名字来解析视图。Spring提供了多种视图解析器。我们将举例加以说明。

| **ViewResolver** | **描述** |
| --- | --- |
| AbstractCachingViewResolver | 抽象视图解析器实现了对视图的缓存。在视图被使用之前，通常需要进行一些准备工作。 从它继承的视图解析器将对要解析的视图进行缓存。 |
| XmlViewResolver | XmlViewResolver实现ViewResolver，支持XML格式的配置文件。 该配置文件必须采用与Spring XML Bean Factory相同的DTD。默认的配置文件是 /WEB-INF/views.xml。 |
| ResourceBundleViewResolver | ResourceBundleViewResolver实现ViewResolver， 在一个ResourceBundle中寻找所需bean的定义。 这个bundle通常定义在一个位于classpath中的属性文件中。默认的属性文件是views.properties。 |
| UrlBasedViewResolver | UrlBasedViewResolver实现ViewResolver， 将视图名直接解析成对应的URL，不需要显式的映射定义。 如果你的视图名和视图资源的名字是一致的，就可使用该解析器，而无需进行映射。 |
| InternalResourceViewResolver | 作为UrlBasedViewResolver的子类， 它支持InternalResourceView（对Servlet和JSP的包装）， 以及其子类JstlView和TilesView。 通过setViewClass方法，可以指定用于该解析器生成视图使用的视图类。 更多信息请参考UrlBasedViewResolver的Javadoc。 |
| VelocityViewResolver / FreeMarkerViewResolver | 作为UrlBasedViewResolver的子类， 它能支持VelocityView（对Velocity模版的包装）和FreeMarkerView以及它们的子类。 |
| ContentNegotiatingViewResolver | 实现ViewResolver接口，根据request请求的file name或Accept header决定视图。 |

举例来说，当使用JSP作为视图层技术时，就可以使用UrlBasedViewResolver。 这个视图解析器会将视图名解析成URL，并将请求传递给RequestDispatcher来显示视图。

<bean id="viewResolver"

class="org.springframework.web.servlet.view.UrlBasedViewResolver">

<property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>

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

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

</bean>

当返回的视图名为test时， 这个视图解析器将请求传递给RequestDispatcher，RequestDispatcher再将请求传递给/WEB-INF/jsp/test.jsp。

### 2.5.2 视图解析链

Spring支持多个视图解析器一起使用。可以把它们当作一个解析链。 这样有很多好处，比如在特定情况下重新定义某些视图。 定义视图解析链很容易，只要在应用上下文中定义多个解析器就可以了。 必要时，也可以通过order属性来声明每个解析器的序列。 要记住的是，某个解析器的order越高, 它在解析链中的位置越靠后。

下面这个例子展示了一个包含两个解析器的解析链。 一个是InternalResourceViewResolver，这个解析器总是被自动的放到链的末端。 另一个是XmlViewResolver，它支持解析Excel视图（而InternalResourceViewResolver不可以）。

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

<property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>

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

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

</bean>

<bean id="excelViewResolver" class="org.springframework.web.servlet.view.XmlViewResolver">

<property name="order" value="1"/>

<property name="location" value="/WEB-INF/views.xml"/>

</bean>

*<!-- in views.xml -->*

<beans>

<bean name="report" class="org.springframework.example.ReportExcelView"/>

</beans>

如果某个解析器没有找到合适的视图，Spring会在上下文中寻找是否配置了其它的解析器。 如果有，它会继续进行解析，否则，Srping会抛出一个ServletException。

### 2.5.3 重定向到另一个视图

一个控制器通常会返回视图名，然后由视图解析器解析到某种视图实现。对于像JSP这样实际上由Servlet/JSP引擎处理的视图， 我们通常使用InternalResourceViewResolver和InternalResourceView。 这种视图实现最终会调用Servlet API的RequestDispatcher.forward(..)方法或RequestDispatcher.include()方法将用户指向最终页面。 对于别的视图技术而言（比如Velocity、XSLT等等），视图本身就会生成返回给用户的内容。

有些时候，在视图显示以前，我们可能需要给用户发一个HTTP redirect重定向指令。 比如，一个控制器成功的处理了一个表单提交（数据以HTTP POST的方式发送），它最终可能委托给另一个控制器来完成剩下的工作。 在这种情况下，如果我们使用内部forward，接手工作的那个控制器将会得到所有以POST方式提交的表单数据， 这可能会引起潜在的混淆，干扰那个控制器的正常工作。 另一个在显示视图之前返回HTTP redirect的原因是这可以防止用户重复提交同一表单。 具体一点讲，浏览器先用POST的方式提交表单,然后它接收到重定向的指令，它继续用GET的方式去下载新的页面。 从浏览器的角度看，这个新的页面不是POST的返回结果，而是GET的。 这样，用户不可能在点击刷新的时候不小心再次提交表单，因为刷新的结果是再次用GET 去下载表单提交后的结果页面，而不是重新提交初始的POST数据。

**RedirectView**

在控制器中强制重定向的方法之一是让控制器创建并返回一个Spring的RedirectView的实例。 在这种情况下，DispatcherServlet不会使用通常的视图解析机制， 既然它已经拿到了一个（重定向）视图，它就让这个视图去完成余下的工作。

RedirectView会调用HttpServletResponse.sendRedirect()方法， 其结果是给用户的浏览器发回一个HTTP redirect。所有的模型属性都被转换成以HTTP请求的访问参数。 这意味着这个模型只能包含可以被简便的转换成string形式的HTTP请求访问参数的对象，比如String或者可以被转换成String的类型。

如果使用RedirectView视图，并且它是由控制器创建的， 重定向的URL最好是用Spring所提供的IoC功能注射到控制器中。 这样这个URL就可以和视图名一起在上下文中被声明，而不是固化在控制器内。

**redirect:前缀**

尽管使用RedirectView帮我们达到了目的，但是如果控制器生成RedirectView的话， 控制器不可避免地要知道某个请求的结果是让用户重定向到另一个页面。这不是最佳的实现，因为这使得系统不同模块之间结合得过于紧密。 其实控制器不应该过问返回结果是如何生成的，通常情况下，它应该只关心注入给它的视图名称。

解决上述问题的方法是依靠redirect:前缀。 如果返回的视图名包含redirect:前缀，UrlBasedViewResolver （以及它的子类） 会知道系统要生成一个HTTP redirect。 视图名其余的部分会被当作重定向URL。

这样做的最终结果跟控制器返回RedirectView是一样的，但现在控制器只需要和逻辑上的视图名打交道。 redirect:/my/response/controller.html这个逻辑视图名中的URL是当前servlet context中的相对路径。 与之相比，redirect:http://myhost.com/some/arbitrary/path.html中的URL是绝对路径。 重要的是，只要这个重定向视图名和其他视图名以相同的方式注入到控制器中，控制器根本不知道重定向是否发生。

**forward:前缀**

类似的，我们也可以使用包含有forward:前缀的视图名。 这些视图名会被UrlBasedViewResolver和它的子类正确解析。 解析的内部实现是生成一个InternalResourceView， 这个视图最终会调用RequestDispatcher.forward()方法，将forward视图名的其余部分作为URL。 所以，当使用InternalResourceViewResolver/InternalResourceView， 并且你所用的视图技术是JSP时，你没有必要使用这个前缀。 但是，当你主要使用其它的视图技术，但仍需要对Servlet/JSP engine处理的页面强制forward时， 这个forward前缀还是很有用的（但就这个问题而言，如果不想使用forward前缀，也可以使用视图解析链）。

和redirect:前缀一样，如果含有forward前缀的视图名和其他视图名一样被注入控制器， 控制器根本不需要知道在处理响应的过程中是否发生任何特殊情况。

### 2.5.4 ContentNegotiatingViewResolver

ContentNegotiatingViewResolver自身不会去决定视图，但会委派其他的视图解析器来选择匹配客户端请求的视图。

## 2.6 Spring MVC常用注解说明

**@Controller**

在类上面定义，表明该类为控制器，返回字符串与redirect:xxx

**@RequestMapping**

在类或方法上面使用此注解，设置URL访问地址。它有两个属性，value指定访问路径，method指定指定请求方式，请求方式在RequestMethod这个类中，全部以常量形式定义，它默认使用GET请求。

**@RequestParam**

指定Request请求参数，在方法参数中定义，相当于传统的request.getParameter()。

**@PathVariable**

获取URL访问路径变量，这是Spring MVC 3.0框架才加入的特性，基于RESTful风格的URL访问路径。

**@ModelAttribute**

全局式的方法，在一组URL访问路径中，每次都会执行，方法返回结果保存在module会话中。

**@Service**

在类上面定义，指定被注解的类是业务逻辑组件，如果不指定具体的Bean ID，则采用默认命名方式，即类名的首字母小写。之前在SSH2中，大象曾对Dao组件使用@Repository，本例只有业务层，所以就只用@Service注解。

**@Autowired**

IoC自动注入功能，替换以前的set写法.

**@Qualifier**

对同一接口类有不同实现指定具体的实现类。

**@ResponseBody**

同样定义在方法上，Ajax调用声明，指定方法返回结果为Ajax回调函数结果。这是Spring MVC 3.0框架中增加的一个新特性。

**@InitBinder**

初始化数据绑定与类型转换，将传入的参数转换为自定义类型，或者对参数进行自定义处理。

## 2.7 配置Spring MVC

**使用Spring MVC默认配置**

在[servlet-name]-servlet.xml 配置文件中加入如下配置信息：

<mvc:annotation-driven />

上述配置等同于注册了RequestMappingHandlerMaping，RequestMappingHandlerAdapter及ExceptionHandlerExceptionResolver ，用于支持Spring MVC注解。如 @RequestMapping , @ExceptionHandle， @NumberFormat，@DateTimeFormat 等注解标签。

通过配置mvc:annotation-driven，使能够支持以下：

1. 通过ConversionService 实例使用Spring3 类型转换，数据绑定。
2. 支持数据格式转换，如@NumberFormat、@DateTimeFormat、@Valid 等
3. 如下的HttpMessageConverters

* ByteArrayHttpMessageConverter converts byte arrays.
* StringHttpMessageConverter converts strings.
* ResourceHttpMessageConverter converts to/from org.springframework.core.io.Resource for all media types.
* SourceHttpMessageConverter converts to/from a javax.xml.transform.Source.
* FormHttpMessageConverter converts form data to/from a MultiValueMap<String, String>.
* Jaxb2RootElementHttpMessageConverter converts Java objects to/from XML — added if JAXB2 is present on the classpath.
* MappingJackson2HttpMessageConverter (or MappingJacksonHttpMessageConverter) converts to/from JSON — added if Jackson 2 (or Jackson) is present on the classpath.
* AtomFeedHttpMessageConverter converts Atom feeds — added if Rome is present on the classpath.
* RssChannelHttpMessageConverter converts RSS feeds — added if Rome is present on the classpath.

**自定义Spring MVC配置**

<mvc:annotation-driven conversion-service="conversionService">

<mvc:message-converters>

<bean class="org.example.MyHttpMessageConverter"/>

<bean class="org.example.MyOtherHttpMessageConverter"/>

</mvc:message-converters>

</mvc:annotation-driven>

<bean id="conversionService" class="org.springframework.format.support.FormattingConversionServiceFactoryBean">

<property name="formatters">

<list>

<bean class="org.example.MyFormatter"/>

<bean class="org.example.MyOtherFormatter"/>

</list>

</property>

</bean>

具体可参考  [Spring MVC XML schema](http://static.springsource.org/schema/mvc/spring-mvc-3.1.xsd)

**配置拦截器**

<mvc:interceptors>

<bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor" />

<mvc:interceptor>

<mapping path="/secure/\*"/>

<bean class="org.example.SecurityInterceptor" />

</mvc:interceptor>

</mvc:interceptors>

# 3 Spring AOP

# 4资源(Resource)

Spring的 Resource 接口是为了提供更强的访问底层资源能力的抽象。

Resource 接口一些比较重要的方法如下：

* getInputStream(): 定位并打开资源，返回读取此资源的一个 InputStream。每次调用预期会返回一个新的 InputStream，由调用者负责关闭这个流。
* exists(): 返回标识这个资源在物理上是否的确存在的 boolean 值。
* isOpen(): 返回标识这个资源是否有已打开流的处理类的 boolean 值。如果为 true，则此InputStream 就不能被多次读取，而且只能被读取一次然后关闭以避免资源泄漏。除了 InputStreamResource，常见的resource实现都会返回 false。
* getDescription(): 返回资源的描述，一般在与此资源相关的错误输出时使用。此描述通常是完整的文件名或实际的URL地址。

Spring自身处理资源请求的多种方法声明中将Resource 抽象作为参数而广泛地使用。 Spring APIs中的一些其它方法(比如许多ApplicationContext的实现构造函数)，使用普通格式的 String 来创建与context相符的Resource，也可以使用特殊的路径String前缀来让调用者指定创建和使用特定的 Resource 实现。

Resource不仅被Spring自身大量地使用，它也非常适合在你自己的代码中独立作为辅助类使用。用户代码甚至可以在不用关心Spring其它部分的情况下访问资源。这样的确会造成代码与Spring之间的耦合，但也仅仅是与很少量的辅助类耦合。这些类可以作为比 URL 更有效的替代，而且与为这个目的而使用其它类库基本相似。

需要注意的是 Resource 抽象并没有改变功能：它尽量使用封装。 比如 UrlResource 封装了URL，然后使用被封装的 URL 来工作。

## 4.1 内置Resource实现

**UrlResource**

UrlResource 封装了java.net.URL，它能够被用来访问任何通过URL可以获得的对象，例如：文件、HTTP对象、FTP对象等。所有的URL都有个标准的 String表示，这些标准前缀可以标识不同的URL类型，包括file:访问文件系统路径，http: 通过HTTP协议访问的资源，ftp: 通过FTP访问的资源等等。

UrlResource 对象可以在Java代码中显式地使用 UrlResource 构造函数来创建。但更多的是通过调用带表示路径的 String 参数的API函数隐式地创建。在后一种情况下，JavaBeans的 PropertyEditor 会最终决定哪种类型的 Resource 被创建。如果这个字符串包含一些众所周知的前缀，比如 classpath:，它就会创建一个对应的已串行化的 Resource。 然而，如果不能分辨出这个前缀，就会假定它是个标准的URL字符串，然后创建UrlResource。

**ClassPathResource**

这个类标识从classpath获得的资源。它会使用线程context的类加载器(class loader)、给定的类加载器或者用来载入资源的给定类。

如果类路径上的资源存在于文件系统里，这个 Resource 的实现会提供类似于java.io.File的功能。而如果资源是存在于还未解开(被servlet引擎或其它的环境解开)的jar包中，这些 Resource 实现会提供类似于java.net.URL 的功能。

ClassPathResource对象可以在Java代码中显式地使用ClassPathResource 构造函数来创建。但更多的是通过调用带表示路径的String参数的API函数隐式地创建。在后一种情况下，JavaBeans的 PropertyEditor 会分辨字符串中 classpath: 前缀，然后相应创建 ClassPathResource。

**FileSystemResource**

这是为处理 java.io.File 而准备的Resource实现。它既可以作为File提供，也可以作为URL。

**ServletContextResource**

这是为 ServletContext 资源提供的 Resource 实现，它负责解析相关web应用根目录中的相对路径。

它始终支持以流和URL的方式访问。 但是只有当web应用包被解开并且资源在文件系统的物理路径上时，才允许以 java.io.File 方式访问。是否解开并且在文件系统中 访问，还是直接从JAR包访问或以其它方式访问如DB(这是可以想象的)，仅取决于Servlet容器。

**InputStreamResource**

这是为给定的 InputStream 而准备的 Resource 实现。它只有在没有其它合适的 Resource 实现时才使用。而且，只要有可能就尽量使用 ByteArrayResource 或者其它基于文件的Resource 实现。

与其它 Resource 实现不同的是，这是个已经打开资源的描述符-因此 isOpen() 函数返回 true。 如果你需要在其它位置保持这个资源的描述符或者多次读取一个流，请不要使用它。

**ByteArrayResource**

这是为给定的byte数组准备的 Resource 实现。 它会为给定的byte数组构造一个 ByteArrayInputStream。

它在从任何给定的byte数组读取内容时很有用，因为不用转换成单一作用的 InputStreamResource。

## 4.2 ResourceLoader接口

所有的application context都实现了 ResourceLoader 接口， 因此它们可以用来获取Resource 实例。

当你调用特定application context的 getResource() 方法， 而且资源路径并没有特定的前缀时，你将获得与该application context相应的 Resource 类型。例如：假定下面的代码片断是基于ClassPathXmlApplicationContext 实例上执行的，这将返回ClassPathResource。

| **前缀** | **例子** | **说明** |
| --- | --- | --- |
| classpath: | classpath:com/myapp/config.xml | 从classpath中加载。 |
| file: | file:/data/config.xml | 作为 URL 从文件系统中加载。[[a](mk:@MSITStore:E:\\learn\\开源框架\\spring\\Spring2.5-中文参考手册.chm::/ch04s04.html" \l "ftn.id470855)] |
| http: | http://myserver/logo.png | 作为 URL 加载。 |
| (none) | /data/config.xml | 根据 ApplicationContext 进行判断。 |

# 5 Spring基础

## 5.1 Spring注解 – Java Annotation

有必要对JDK 5.0新增的注解（Annotation）技术进行简单的学习，因为Spring 支持@AspectJ，而@AspectJ本身就是基于JDK 5.0的注解技术。所以学习JDK 5.0的注解知识有助于我们更好地理解和掌握Spring的AOP技术。

### 5.1.1了解注解

对于Java开发人员来说，在编写代码时，除了源程序以外，我们还会使用Javadoc标签对类、方法或成员变量进行注释，以便使用Javadoc工具生成和源代码配套的Javadoc文档。这些@param、@return等Javadoc标签就是注解标签，它们为第三方工具提供了描述程序代码的注释信息。使用过Xdoclet的朋友，对此将更有感触，像Struts、Hibernate都提供了Xdoclet标签，使用它们可以快速地生成对应程序代码的配置文件。

JDK5.0注解可以看成是Javadoc标签和Xdoclet标签的延伸和发展。在JDK5.0中，我们可以自定义这些标签，并通过Java语言的反射机制中获取类中标注的注解，完成特定的功能。

注解是代码的附属信息，它遵循一个基本原则：注解不能直接干扰程序代码的运行，无论增加或删除注解，代码都能够正常运行。Java语言解释器会忽略这些注解，而由第三方工具负责对注解进行处理。第三方工具可以利用代码中的注解间接控制程序代码的运行，它们通过Java反射机制读取注解的信息，并根据这些信息更改目标程序的逻辑，而这正是Spring AOP对@AspectJ提供支持所采取的方法。

### 5.1.2引用

很多东西的设计都必须遵循最基本的原则，为了防止机器人伤害人类，科幻作家阿西莫夫于1940年提出了“机器人三原则”：第一，机器人不能伤害人类；第二，机器人应遵守人类的命令，与第一条违背的命令除外；第三，机器人应能保护自己，与第一条违背的命令除外。这是给机器人赋予的伦理性纲领，机器人学术界一直将这三条原则作为机器人开发的准则。

一个简单的注解类

通常情况下，第三方工具不但负责处理特定的注解，本身还提供了这些注解的定义，所以我们通常仅需关注如何使用注解就可以了。但定义注解类本身并不困难，Java提供了定义注解的语法。下面，我们马上着手编写一个简单的注解类，如代码清单7-1所示：

代码清单7-1 NeedTest注解类

Java代码

package com.baobaotao.aspectj.anno;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME) //①声明注解的保留期限

@Target(ElementType.METHOD)//②声明可以使用该注解的目标类型

public @interface NeedTest {//③定义注解

boolean value() default true;//④声明注解成员

}

Java新语法规定使用@interface修饰符定义注解类，如③所示，一个注解可以拥有多个成员，成员声明和接口方法声明类似，这里，我们仅定义了一个成员，如④所示。成员的声明有以下几点限制：

**成员以无入参无抛出异常的方式声明**，如boolean value(String str)、boolean value() throws Exception等方式是非法的；

可以通过default为成员指定一个默认值，如String level() default "LOW\_LEVEL"、int high() default 2是合法的，当然也可以不指定默认值；

成员类型是受限的，合法的类型包括原始类型及其封装类、String、Class、enums、注解类型，以及上述类型的数组类型。如ForumService value()、List foo()是非法的。

在①和②处，我们所看到的注解是Java预定义的注解，称为元注解（Meta-Annotation），它们被Java编译器使用，会对注解类的行为产生影响。@Retention(RetentionPolicy. RUNTIME)表示NeedTest这个注解可以在运行期被JVM读取，注解的保留期限类型在java.lang.annotation.Retention类中定义，介绍如下：

SOURCE：注解信息仅保留在目标类代码的源码文件中，但对应的字节码文件将不再保留；

CLASS：注解信息将进入目标类代码的字节码文件中，但类加载器加载字节码文件时不会将注解加载到JVM中，也即运行期不能获取注解信息；

RUNTIME：注解信息在目标类加载到JVM后依然保留，在运行期可以通过反射机制读取类中注解信息。

Target(ElementType.METHOD)表示NeedTest这个注解只能应用到目标类的方法上，注解的应用目标在java.lang.annotation.ElementType类中定义：

TYPE：类、接口、注解类、Enum声明处，相应的注解称为类型注解；

FIELD：类成员变量或常量声明处，相应的注解称为域值注解；

METHOD：方法声明处，相应的注解称为方法注解；

PARAMETER：参数声明处，相应的注解称为参数注解；

CONSTRUCTOR：构造函数声明处，相应的注解称为构造函数注解；

LOCAL\_VARIABLE：局部变量声明处，相应的注解称为局域变量注解；

ANNOTATION\_TYPE：注解类声明处，相应的注解称为注解类注解，ElementType. TYPE包括ElementType.ANNOTATION\_TYPE；

PACKAGE：包声明处，相应的注解称为包注解。

如果注解只有一个成员，则成员名必须取名为value()，在使用时可以忽略成员名和赋值号（=），如@NeedTest(true)。注解类拥有多个成员时，如果仅对value成员进行赋值则也可不使用赋值号，如果同时对多个成员进行赋值，则必须使用赋值号，如DeclareParents (value = "NaiveWaiter", defaultImpl = SmartSeller.class)。注解类可以没有成员，没有成员的注解称为标识注解，解释程序以标识注解存在与否进行相应的处理；此外，所有的注解类都隐式继承于java.lang.annotation.Annotation，但注解不允许显式继承于其他的接口。

我们希望使用NeedTest注解对业务类的方法进行标注，以便测试工具可以根据注解情况激活或关闭对业务类的测试。在编写好NeedTest注解类后，就可以在其他类中使用它了。

### 5.1.3使用注解

我们在ForumService中使用NeedTest注解，标注业务方法是否需要测试，如代码清单7-2所示：

代码清单7-2 ForumService：使用注解

Java代码

package com.baobaotao.aspectj.anno;

public class ForumService {

@NeedTest(value=true) ①

public void deleteForum(int forumId){

System.out.println("删除论坛模块："+forumId);

}

@NeedTest(value=false) ②

public void deleteTopic(int postId){

System.out.println("删除论坛主题："+postId);

}

}

如果注解类和目标类不在同一个包中，需要通过import引用的注解类。在①和②处，我们使用NeedTest分别对deleteForum()和deleteTopic()方法进行标注。在标注注解时，可以通过以下格式对注解成员进行赋值：

引用

@<注解名>(<成员名1>=<成员值1>,<成员名1>=<成员值1>,...)

如果成员是数组类型，可以通过{}进行赋值，如boolean数组的成员可以设置为{true,false,true}。下面是几个注解标注的例子：

示例1，多成员的注解：

Java代码

@AnnoExample(id= 2868724, synopsis = "Enable time-travel",

engineer = "Mr. Peabody",date = "4/1/2007")

示例2，一个成员的注解，成员名为value。可以省略成员名和赋值符号：

Java代码

@Copyright("2011 bookegou.com All Right Reserved")

示例3，无成员的注解：

Java代码

@Override

示例4，成员为字符串数组的注解：

Java代码

@SuppressWarnings(value={"unchecked","fallthrough"})

示例5，成员为注解数组类型的注解：

Java代码

@Reviews({@Review(grade=Review.Grade.EXCELLENT,reviewer="df"),

@Review(grade=Review.Grade.UNSATISFACTORY,reviewer="eg",

comment="This method needs an @Override annotation")})

@Reviews注解拥有一个@Review注解数组类型的成员，@Review注解类型有三个成员，其中reviewer、comment都是String类型，但comment有默认值，grade是枚举类型的成员。

由于NeedTest注解的保留限期是RetentionPolicy.RUNTIME类型，因此当ForumService被加载到JVM时，仍就可通过反射机制访问到ForumService各方法的注解信息。

### 5.1.4访问注解

前面提到过，注解不会直接影响程序的运行，但是第三方程序或工具可以利用代码中的注解完成特殊的任务，间接控制程序的运行。对于RetentionPolicy.RUNTIME保留期限的注解，我们可以通过反射机制访问类中的注解。

在JDK5.0里，Package、Class、Constructor、Method以及Field等反射对象都新增了访问注解信息的方法：<T extends Annotation>T getAnnotation(Class<T> annotationClass)，该方法支持通过泛型直接返回注解对象。

下面，我们就通过反射来访问注解，得出ForumService 类中通过@NeedTest注解所承载的测试需求，如代码清单7-3所示：

代码清单7-3 TestTool：访问代码中的注解

Java代码

package com.baobaotao.aspectj.anno;

import java.lang.reflect.Method;

public class TestTool {

public static void main(String[] args) {

//①得到ForumService对应的Class对象

Class clazz = ForumService.class;

//②得到ForumSerivce对应的Method数组

Method[] methods = clazz.getDeclaredMethods();

System.out.println(methods.length);

for (Method method : methods) {

//③获取方法上所标注的注解对象

NeedTest nt = method.getAnnotation(NeedTest. class);

if (nt != null) {

if (nt.value()) {

System.out.println(method.getName() + "()需要测试");

} else {

System.out.println(method.getName() + "()不需要测试");

}

}

}

}

}

在③处，通过方法的反射对象，我们获取了方法上所标注的NeedTest注解对象，接着就可以访问注解对象的成员，从而得到ForumService类方法的测试需求。运行以上代码，输出以下的信息：

引用

deleteForum()需要测试

deleteTopic()不需要测试

## 5.2 Spring IoC - Java反射知识

引述要学习Spring框架的技术内幕，必须事先掌握一些基本的Java知识，正所谓“登高必自卑，涉远必自迩”。以下几项Java知识和Spring框架息息相关，不可不学（我将通过一个系列分别介绍这些Java基础知识，希望对大家有所帮助。）：

Java语言允许通过程序化的方式间接对Class进行操作，Class文件由类装载器装载后，在JVM中将形成一份描述Class结构的元信息对象，通过该元信息对象可以获知Class的结构信息：如构造函数、属性和方法等。Java允许用户借由这个Class相关的元信息对象间接调用Class对象的功能，这就为使用程序化方式操作Class对象开辟了途径。

简单实例

我们将从一个简单例子开始探访Java反射机制的征程，下面的Car类拥有两个构造函数、两个方法以及三个属性，如代码清单3-9所示：

代码清单3-9 Car

Java代码

package com.baobaotao.reflect;

public class Car {

private String brand;

private String color;

private int maxSpeed;

//①默认构造函数

public Car(){}

//②带参构造函数

public Car(String brand,String color,int maxSpeed){

this.brand = brand;

this.color = color;

this.maxSpeed = maxSpeed;

}

//③未带参的方法

public void introduce() {

System.out.println("brand:"+brand+";color:"+color+";maxSpeed:" +maxSpeed);

}

//省略参数的getter/Setter方法

…

}

一般情况下，我们会使用如下的代码创建Car的实例：

Java代码

Car car = new Car();

car.setBrand("红旗CA72");

或者：

Java代码

Car car = new Car("红旗CA72","黑色");

以上两种方法都采用传统方式的直接调用目标类的方法，下面我们通过Java反射机制以一种更加通用的方式间接地操作目标类：

代码清单3-10 ReflectTest

Java代码

package com.baobaotao. reflect;

import java.lang.reflect.Constructor;

import java.lang.reflect.Field;

import java.lang.reflect.Method;

public class ReflectTest {

public static Car initByDefaultConst() throws Throwable

{

//①通过类装载器获取Car类对象

ClassLoader loader = Thread.currentThread().getContextClassLoader();

Class clazz = loader.loadClass("com.baobaotao.reflect.Car");

//②获取类的默认构造器对象并通过它实例化Car

Constructor cons = clazz.getDeclaredConstructor((Class[])null);

Car car = (Car)cons.newInstance();

//③通过反射方法设置属性

Method setBrand = clazz.getMethod("setBrand",String.class);

setBrand.invoke(car,"红旗CA72");

Method setColor = clazz.getMethod("setColor",String.class);

setColor.invoke(car,"黑色");

Method setMaxSpeed = clazz.getMethod("setMaxSpeed",int.class);

setMaxSpeed.invoke(car,200);

return car;

}

public static void main(String[] args) throws Throwable {

Car car = initByDefaultConst();

car.introduce();

}

}

运行以上程序，在控制台上将打印出以下信息：

引用

brand:红旗CA72;color:黑色;maxSpeed:200

这说明我们完全可以通过编程方式调用Class的各项功能，这和直接通过构造函数和方法调用类功能的效果是一致的，只不过前者是间接调用，后者是直接调用罢了。

在ReflectTest中，使用了几个重要的反射类，分别是ClassLoader、Class、Constructor和Method，通过这些反射类就可以间接调用目标Class的各项功能了。在①处，我们获取当前线程的ClassLoader，然后通过指定的全限定类“com.baobaotao.beans.Car”装载Car类对应的反射实例。在②处，我们通过Car的反射类对象获取Car的构造函数对象cons，通过构造函数对象的newInstrance()方法实例化Car对象，其效果等同于new Car()。在③处，我们又通过Car的反射类对象的getMethod（String methodName,Class paramClass）获取属性的Setter方法对象，第一个参数是目标Class的方法名；第二个参数是方法入参的对象类型。获取方法反射对象后，即可通过invoke（Object obj,Object param）方法调用目标类的方法，该方法的第一个参数是操作的目标类对象实例；第二个参数是目标方法的入参。

在代码清单3 10中，粗体所示部分的信息即是通过反射方法操控目标类的元信息，如果我们将这些信息以一个配置文件的方式提供，就可以使用Java语言的反射功能编写一段通用的代码对类似于Car的类进行实例化及功能调用操作了。

类装载器ClassLoader

类装载器工作机制

类装载器就是寻找类的节码文件并构造出类在JVM内部表示对象的组件。在Java中，类装载器把一个类装入JVM中，要经过以下步骤：

[1.]装载：查找和导入Class文件；

[2.]链接：执行校验、准备和解析步骤，其中解析步骤是可以选择的：

[2.1]校验：检查载入Class文件数据的正确性；

[2.2]准备：给类的静态变量分配存储空间；

[2.3]解析：将符号引用转成直接引用；

[3.]初始化：对类的静态变量、静态代码块执行初始化工作。

类装载工作由ClassLoader及其子类负责，ClassLoader是一个重要的Java运行时系统组件，它负责在运行时查找和装入Class字节码文件。JVM在运行时会产生三个ClassLoader：根装载器、ExtClassLoader（扩展类装载器）和AppClassLoader（系统类装载器）。其中，根装载器不是ClassLoader的子类，它使用C++编写，因此我们在Java中看不到它，根装载器负责装载JRE的核心类库，如JRE目标下的rt.jar、charsets.jar等。ExtClassLoader和AppClassLoader都是ClassLoader的子类。其中ExtClassLoader负责装载JRE扩展目录ext中的JAR类包；AppClassLoader负责装载Classpath路径下的类包。

这三个类装载器之间存在父子层级关系，即根装载器是ExtClassLoader的父装载器，ExtClassLoader是AppClassLoader的父装载器。默认情况下，使用AppClassLoader装载应用程序的类，我们可以做一个实验：

代码清单3-11 ClassLoaderTest

Java代码

public class ClassLoaderTest {

public static void main(String[] args) {

ClassLoader loader = Thread.currentThread().getContextClassLoader();

System.out.println("current loader:"+loader);

System.out.println("parent loader:"+loader.getParent());

System.out.println("grandparent loader:"+loader.getParent(). getParent());

}

}

运行以上代码，在控制台上将打出以下信息：

引用

current loader:sun.misc.Launcher$AppClassLoader@131f71a

parent loader:sun.misc.Launcher$ExtClassLoader@15601ea

//①根装载器在Java中访问不到，所以返回null

grandparent loader:null

通过以上的输出信息，我们知道当前的ClassLoader是AppClassLoader，父ClassLoader是ExtClassLoader，祖父ClassLoader是根类装载器，因为在Java中无法获得它的句柄，所以仅返回null。

JVM装载类时使用“全盘负责委托机制”，“全盘负责”是指当一个ClassLoader装载一个类的时，除非显式地使用另一个ClassLoader，该类所依赖及引用的类也由这个ClassLoader载入；“委托机制”是指先委托父装载器寻找目标类，只有在找不到的情况下才从自己的类路径中查找并装载目标类。这一点是从安全角度考虑的，试想如果有人编写了一个恶意的基础类（如java.lang.String）并装载到JVM中将会引起多么可怕的后果。但是由于有了“全盘负责委托机制”，java.lang.String永远是由根装载器来装载的，这样就避免了上述事件的发生。

ClassLoader重要方法

在Java中，ClassLoader是一个抽象类，位于java.lang包中。下面对该类的一些重要接口方法进行介绍：

Class loadClass(String name)

name参数指定类装载器需要装载类的名字，必须使用全限定类名，如com.baobaotao. beans.Car。该方法有一个重载方法loadClass(String name ,boolean resolve)，resolve参数告诉类装载器是否需要解析该类。在初始化类之前，应考虑进行类解析的工作，但并不是所有的类都需要解析，如果JVM只需要知道该类是否存在或找出该类的超类，那么就不需要进行解析。

Class defineClass(String name, byte[] b, int off, int len)

将类文件的字节数组转换成JVM内部的java.lang.Class对象。字节数组可以从本地文件系统、远程网络获取。name为字节数组对应的全限定类名。

Class findSystemClass(String name)

从本地文件系统载入Class文件，如果本地文件系统不存在该Class文件，将抛出ClassNotFoundException异常。该方法是JVM默认使用的装载机制。

Class findLoadedClass(String name)

调用该方法来查看ClassLoader是否已装入某个类。如果已装入，那么返回java.lang.Class对象，否则返回null。如果强行装载已存在的类，将会抛出链接错误。

ClassLoader getParent()

获取类装载器的父装载器，除根装载器外，所有的类装载器都有且仅有一个父装载器，ExtClassLoader的父装载器是根装载器，因为根装载器非Java编写，所以无法获得，将返回null。

除JVM默认的三个ClassLoader以外，可以编写自己的第三方类装载器，以实现一些特殊的需求。类文件被装载并解析后，在JVM内将拥有一个对应的java.lang.Class类描述对象，该类的实例都拥有指向这个类描述对象的引用，而类描述对象又拥有指向关联ClassLoader的引用，如图3-4所示。

每一个类在JVM中都拥有一个对应的java.lang.Class对象，它提供了类结构信息的描述。数组、枚举、注解以及基本Java类型（如int、double等），甚至void都拥有对应的Class对象。Class没有public的构造方法。Class对象是在装载类时由JVM通过调用类装载器中的defineClass()方法自动构造的。

Java反射机制

Class反射对象描述类语义结构，可以从Class对象中获取构造函数、成员变量、方法类等类元素的反射对象，并以编程的方式通过这些反射对象对目标类对象进行操作。这些反射对象类在java.reflect包中定义，下面是最主要的三个反射类：

 Constructor：类的构造函数反射类，通过Class#getConstructors()方法可以获得类的所有构造函数反射对象数组。在JDK5.0中，还可以通过getConstructor(Class... parameterTypes)获取拥有特定入参的构造函数反射对象。Constructor的一个主要方法是newInstance(Object[] initargs)，通过该方法可以创建一个对象类的实例，相当于new关键字。在JDK5.0中该方法演化为更为灵活的形式：newInstance (Object... initargs)。

 Method：类方法的反射类，通过Class#getDeclaredMethods()方法可以获取类的所有方法反射类对象数组Method[]。在JDK5.0中可以通过getDeclaredMethod(String name, Class... parameterTypes)获取特定签名的方法，name为方法名；Class...为方法入参类型列表。Method最主要的方法是invoke(Object obj, Object[] args)，obj表示操作的目标对象；args为方法入参，代码清单3 10③处演示了这个反射类的使用方法。在JDK 5.0中，该方法的形式调整为invoke(Object obj, Object... args)。此外，Method还有很多用于获取类方法更多信息的方法：

1）Class getReturnType()：获取方法的返回值类型；

2）Class[] getParameterTypes()：获取方法的入参类型数组；

3）Class[] getExceptionTypes()：获取方法的异常类型数组；

4）Annotation[][] getParameterAnnotations()：获取方法的注解信息，JDK 5.0中的新方法；

 Field：类的成员变量的反射类，通过Class#getDeclaredFields()方法可以获取类的成员变量反射对象数组，通过Class#getDeclaredField(String name)则可获取某个特定名称的成员变量反射对象。Field类最主要的方法是set(Object obj, Object value)，obj表示操作的目标对象，通过value为目标对象的成员变量设置值。如果成员变量为基础类型，用户可以使用Field类中提供的带类型名的值设置方法，如setBoolean(Object obj, boolean value)、setInt(Object obj, int value)等。

此外，Java还为包提供了Package反射类，在JDK 5.0中还为注解提供了AnnotatedElement反射类。总之，Java的反射体系保证了可以通过程序化的方式访问目标类中所有的元素，对于private或protected的成员变量和方法，只要JVM的安全机制允许，也可以通过反射进行调用，请看下面的例子：

代码清单3-12 PrivateCarReflect

Java代码

package com.baobaotao.reflect;

public class PrivateCar {

//①private成员变量：使用传统的类实例调用方式，只能在本类中访问

private String color;

//②protected方法：使用传统的类实例调用方式，只能在子类和本包中访问

protected void drive(){

System.out.println("drive private car! the color is:"+color);

}

}

color变量和drive()方法都是私有的，通过类实例变量无法在外部访问私有变量、调用私有方法的，但通过反射机制却可以绕过这个限制：

代码清单3-13 PrivateCarReflect

Java代码

…

public class PrivateCarReflect {

public static void main(String[] args) throws Throwable{

ClassLoader loader = Thread.currentThread().getContextClassLoader();

Class clazz = loader.loadClass("com.baobaotao.reflect.PrivateCar");

PrivateCar pcar = (PrivateCar)clazz.newInstance();

Field colorFld = clazz.getDeclaredField("color");

//①取消Java语言访问检查以访问private变量

colorFld.setAccessible(true);

colorFld.set(pcar,"红色");

Method driveMtd = clazz.getDeclaredMethod("drive",(Class[])null);

//Method driveMtd = clazz.getDeclaredMethod("drive"); JDK5.0下使用

//②取消Java语言访问检查以访问protected方法

driveMtd.setAccessible(true);

driveMtd.invoke(pcar,(Object[])null);

}

}

运行该类，打印出以下信息：

引用

drive private car! the color is:红色

在访问private、protected成员变量和方法时必须通过setAccessible(boolean access)方法取消Java语言检查，否则将抛出IllegalAccessException。如果JVM的安全管理器设置了相应的安全机制，调用该方法将抛出SecurityException。

## 5.3 Spring AOP - Java动态代理

Spring AOP使用动态代理技术在运行期织入增强的代码，为了揭示Spring AOP底层的工作机理，有必要对涉及到的Java知识进行学习。Spring AOP使用了两种代理机制：一种是基于JDK的动态代理；另一种是基于CGLib的动态代理。之所以需要两种代理机制，很大程度上是因为JDK本身只提供接口的代理，而不支持类的代理。

带有横切逻辑的实例

我们通过具体化代码实现上一节所介绍例子的性能监视横切逻辑，并通过动态代理技术对此进行改造。在调用每一个目标类方法时启动方法的性能监视，在目标类方法调用完成时记录方法的花费时间。

代码清单6-2 ForumService：包含性能监视横切代码

Java代码

package com.baobaotao.proxy;

public class ForumServiceImpl implements ForumService {

public void removeTopic(int topicId) {

//①-1开始对该方法进行性能监视

PerformanceMonitor.begin(

"com.baobaotao.proxy.ForumServiceImpl. removeTopic");

System.out.println("模拟删除Topic记录:"+topicId);

try {

Thread.currentThread().sleep(20);

} catch (Exception e) {

throw new RuntimeException(e);

}

//①-2结束对该方法进行性能监视

PerformanceMonitor.end();

}

public void removeForum(int forumId) {

//②-1开始对该方法进行性能监视

PerformanceMonitor.begin(

"com.baobaotao.proxy.ForumServiceImpl. removeForum");

System.out.println("模拟删除Forum记录:"+forumId);

try {

Thread.currentThread().sleep(40);

} catch (Exception e) {

throw new RuntimeException(e);

}

//②-2结束对该方法进行性能监视

PerformanceMonitor.end();

}

}

代码清单6-2中粗体表示的代码就是具有横切逻辑特征的代码，每个Service类和每个业务方法体的前后都执行相同的代码逻辑：方法调用前启动PerformanceMonitor，方法调用后通知PerformanceMonitor结束性能监视并给记录性能监视结果。

PerformanceMonitor是性能监视的实现类，我们给出一个非常简单的实现版本，其代码如代码清单6-3所示：

代码清单6-3 PerformanceMonitor

Java代码

package com.baobaotao.proxy;

public class PerformanceMonitor {

//①通过一个ThreadLocal保存调用线程相关的性能监视信息

private static ThreadLocal<MethodPerformace> performanceRecord =

new ThreadLocal<MethodPerformance>();

//②启动对某一目标方法的性能监视

public static void begin(String method) {

System.out.println("begin monitor...");

MethodPerformance mp = new MethodPerformance(method);

performanceRecord.set(mp);

}

public static void end() {

System.out.println("end monitor...");

MethodPerformance mp = performanceRecord.get();

//③打印出方法性能监视的结果信息。

mp.printPerformance();

}

}

ThreadLocal是将非线程安全类改造为线程安全类的法宝，在9.2节中我们将详细介绍这个Java基础知识。PerformanceMonitor提供了两个方法：通过调用begin(String method)方法开始对某个目标类方法的监视，method为目标类方法的全限定名；而end()方法结束对目标类方法的监视，并给出性能监视的信息。这两个方法必须配套使用。

用于记录性能监视信息的MethodPerformance类的代码如所示：

代码清单6-4 MethodPerformance

Java代码

package com.baobaotao.proxy;

public class MethodPerformance {

private long begin;

private long end;

private String serviceMethod;

public MethodPerformance(String serviceMethod){

this.serviceMethod = serviceMethod;

//①记录目标类方法开始执行点的系统时间

this.begin = System.currentTimeMillis();

}

public void printPerformance(){

//②获取目标类方法执行完成后的系统时间，并进而计算出目标类方法执行时间

end = System.currentTimeMillis();

long elapse = end - begin;

//③报告目标类方法的执行时间

System.out.println(serviceMethod+"花费"+elapse+"毫秒。");

}

}

通过下面的代码测试拥有性能监视能力的ForumServiceImpl业务方法：

Java代码

package com.baobaotao.proxy;

public class TestForumService {

public static void main(String[] args) {

ForumService forumService = new ForumServiceImpl();

forumService .removeForum(10);

forumService .removeTopic(1012);

}

}

我们得到以下输出信息：

引用

begin monitor... ①removeForum(10)方法的性能监视报告

模拟删除Forum记录:10

end monitor...

com.baobaotao.proxy.ForumServiceImpl.removeForum花费47毫秒。

begin monitor... ①removeTopic(1012)方法的性能监视报告

模拟删除Topic记录:1012

end monitor...

com.baobaotao.proxy.ForumServiceImpl.removeTopic花费26毫秒。

正如代码清单6 2实例所示，当某个方法需要进行性能监视，就必须调整方法代码，在方法体前后分别添加上开启性能监视和结束性能监视的代码。这些非业务逻辑的性能监视代码破坏了ForumServiceImpl业务逻辑的纯粹性。我们希望通过代理的方式，将业务类方法中开启和结束性能监视的这些横切代码从业务类中完全移除。并通过JDK动态代理技术或CGLib动态代理技术将横切代码动态织入到目标方法的相应位置。

JDK动态代理

JDK 1.3以后，Java提供了动态代理的技术，允许开发者在运行期创建接口的代理实例。在Sun刚推出动态代理时，还很难想象它有多大的实际用途，现在我们终于发现动态代理是实现AOP的绝好底层技术。

JDK的动态代理主要涉及到java.lang.reflect包中的两个类：Proxy和InvocationHandler。其中InvocationHandler是一个接口，可以通过实现该接口定义横切逻辑，并通过反射机制调用目标类的代码，动态将横切逻辑和业务逻辑编织在一起。

而Proxy利用InvocationHandler动态创建一个符合某一接口的实例，生成目标类的代理对象。这样讲一定很抽象，我们马上着手使用Proxy和InvocationHandler这两个魔法戒对上一节中的性能监视代码进行革新。

首先，我们从业务类ForumServiceImpl中删除性能监视的横切代码，使ForumServiceImpl只负责具体的业务逻辑，如代码清单6-5所示：

代码清单6-5 ForumServiceImpl：移除性能监视横切代码

Java代码

package com.baobaotao.proxy;

public class ForumServiceImpl implements ForumService {

public void removeTopic(int topicId) {

①

System.out.println("模拟删除Topic记录:"+topicId);

try {

Thread.currentThread().sleep(20);

} catch (Exception e) {

throw new RuntimeException(e);

}

①

}

public void removeForum(int forumId) {

②

System.out.println("模拟删除Forum记录:"+forumId);

try {

Thread.currentThread().sleep(40);

} catch (Exception e) {

throw new RuntimeException(e);

}

②

}

}

在代码清单6-5中的①和②处，原来的性能监视代码被移除了，我们只保留了真正的业务逻辑。

从业务类中移除的性能监视横切代码当然不能漂浮在空气中，它还得找到一个安身之所，InvocationHandler就是横切代码的安家乐园，我们将性能监视的代码安置在PerformanceHandler中，如代码清单6-6所示：

Java代码

代码清单6-6 PerformanceHandler

package com.baobaotao.proxy;

import java.lang.reflect.InvocationHandler;

import java.lang.reflect.Method;

public class PerformanceHandler implements InvocationHandler {//①实现InvocationHandler

private Object target;

public PerformanceHandler(Object target){ //②target为目标的业务类

this.target = target;

}

public Object invoke(Object proxy, Method method, Object[] args) ③

throws Throwable {

PerformanceMonitor.begin(target.getClass().getName()+"."+ method. getName());③-1

Object obj = method.invoke(target, args);// ③-2通过反射方法调用业务类的目标方法

PerformanceMonitor.end();③-1

return obj;

}

}

③处invoke()方法中粗体所示部分的代码为性能监视的横切代码，我们发现，横切代码只出现一次，而不是原来那样星洒各处。③-2处的method.invoke()语句通过Java反射机制间接调用目标对象的方法，这样InvocationHandler的invoke()方法就将横切逻辑代码（③-1）和业务类方法的业务逻辑代码（③-2）编织到一起了，所以我们可以将InvocationHandler看成是一个编织器。下面，我们对这段代码做进一步的说明。

首先，我们实现InvocationHandler接口，该接口定义了一个 invoke(Object proxy, Method method, Object[] args)的方法，proxy是最终生成的代理实例，一般不会用到；method是被代理目标实例的某个具体方法，通过它可以发起目标实例方法的反射调用；args是通过被代理实例某一个方法的入参，在方法反射调用时使用。

此外，我们在构造函数里通过target传入希望被代理的目标对象，如②处所示，在InvocationHandler接口方法invoke(Object proxy, Method method, Object[] args)里，将目标实例传给method.invoke()方法，调用目标实例的方法，如③所示。

下面，我们通过Proxy结合PerformanceHandler创建ForumService接口的代理实例，如代码清单6-7所示：

代码清单6-7 TestForumService：创建代理实例

Java代码

package com.baobaotao.proxy;

import java.lang.reflect.Proxy;

public class TestForumService {

public static void main(String[] args) {

//①希望被代理的目标业务类

ForumService target = new ForumServiceImpl();

//②将目标业务类和横切代码编织到一起

PerformanceHandler handler = new PerformanceHandler(target);

//③根据编织了目标业务类逻辑和性能监视横切逻辑的InvocationHandler实例创建代理实例

ForumService proxy = (ForumService) Proxy.newProxyInstance(

target.getClass().getClassLoader(),

target.getClass().getInterfaces(),

handler);

//④调用代理实例

proxy.removeForum(10);

proxy.removeTopic(1012);

}

}

上面的代码完成业务类代码和横切代码的编织工作并生成了代理实例。在②处，我们让PerformanceHandler将性能监视横切逻辑编织到ForumService实例中，然后在③处，通过Proxy的newProxyInstance()静态方法为编织了业务类逻辑和性能监视逻辑的handler创建一个符合ForumService接口的代理实例。该方法的第一个入参为类加载器；第二个入参为创建代理实例所需要实现的一组接口；第三个参数是整合了业务逻辑和横切逻辑的编织器对象。

按照③处的设置方式，这个代理实例实现了目标业务类的所有接口，即Forum ServiceImpl的ForumService接口。这样，我们就可以按照调用ForumService接口实例相同的方式调用代理实例，如④所示。运行以上的代码，输出以下信息：

引用

begin monitor...

模拟删除Forum记录:10

end monitor...

com.baobaotao.proxy.ForumServiceImpl.removeForum花费47毫秒。

begin monitor...

模拟删除Topic记录:1012

end monitor...

com.baobaotao.proxy.ForumServiceImpl.removeTopic花费26毫秒。

我们发现，程序的运行效果和直接在业务类中编写性能监视逻辑的效果一致，但是在这里，原来分散的横切逻辑代码已经被我们抽取到PerformanceHandler中。当其他业务类（如UserService、SystemService等）的业务方法也需要使用性能监视时，我们只要按照代码清单6-7相似的方式，分别为它们创建代理对象就可以了。下面，我们通过时序图描述通过创建代理对象进行业务方法调用的整体逻辑，以进一步认识代理对象的本质，如图6-3所示。

我们在上图中使用虚线的方式对通过Proxy创建的ForumService代理实例加以凸显，ForumService代理实例内部利用PerformaceHandler整合横切逻辑和业务逻辑。调用者调用代理对象的removeForum()和removeTopic()方法时，上图的内部调用时序清晰地告诉我们实际上所发生的一切。

CGLib动态代理

使用JDK创建代理有一个限制，即它只能为接口创建代理实例，这一点我们可从Proxy的接口newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h)的方法签名中就看得很清楚：第二个入参interfaces就是需要代理实例实现的接口列表。虽然面向接口编程的思想被很多大师级人物（包括Rod Johnson）推崇，但在实际开发中，许多开发者也对此深感困惑：难道对一个简单业务表的操作也需要老老实实地创建5个类（领域对象类、Dao接口，Dao实现类，Service接口和Service实现类）吗？难道不能直接通过实现类构建程序吗？对于这个问题，我们很难给出一个孰好孰劣的准确判断，但我们确实发现有很多不使用接口的项目也取得了非常好的效果（包括大家所熟悉的SpringSide开源项目）。

对于没有通过接口定义业务方法的类，如何动态创建代理实例呢？JDK的代理技术显然已经黔驴技穷，CGLib作为一个替代者，填补了这个空缺。

CGLib采用非常底层的字节码技术，可以为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，并顺势织入横切逻辑。下面，我们采用CGLib技术，编写一个可以为任何类创建织入性能监视横切逻辑代理对象的代理创建器，如代码清单 6-8所示：

代码清单6-8 CglibProxy

Java代码

package com.baobaotao.proxy;

import java.lang.reflect.Method;

import net.sf.cglib.proxy.Enhancer;

import net.sf.cglib.proxy.MethodInterceptor;

import net.sf.cglib.proxy.MethodProxy;

public class CglibProxy implements MethodInterceptor {

private Enhancer enhancer = new Enhancer();

public Object getProxy(Class clazz) {

enhancer.setSuperclass(clazz); //① 设置需要创建子类的类

enhancer.setCallback(this);

return enhancer.create(); //②通过字节码技术动态创建子类实例

}

//③拦截父类所有方法的调用

public Object intercept(Object obj, Method method, Object[] args,

MethodProxy proxy) throws Throwable {

PerformanceMonitor.begin(obj.getClass().getName()+"."+method. getName());//③-1

Object result=proxy.invokeSuper(obj, args); ③-2

PerformanceMonitor.end();//③-1通过代理类调用父类中的方法

return result;

}

}

在上面代码中，用户可以通过getProxy(Class clazz)为一个类创建动态代理对象，该代理对象通过扩展clazz创建代理对象。在这个代理对象中，我们织入性能监视的横切逻辑（③-1）。intercept(Object obj, Method method, Object[] args,MethodProxy proxy)是CGLib定义的Interceptor接口的方法，它拦截所有目标类方法的调用，obj表示目标类的实例；method为目标类方法的反射对象；args为方法的动态入参；而proxy为代理类实例。

下面，我们通过CglibProxy为ForumServiceImpl类创建代理对象，并测试代理对象的方法，如代码清单6-9所示：

代码清单6-9 TestForumService：测试Cglib创建的代理类

Java代码

package com.baobaotao.proxy;

import java.lang.reflect.Proxy;

public class TestForumService {

public static void main(String[] args) {

CglibProxy proxy = new CglibProxy();

ForumServiceImpl forumService = ①

(ForumServiceImpl )proxy.getProxy(ForumServiceImpl.class);

forumService.removeForum(10);

forumService.removeTopic(1023);

}

}

在①中，我们通过CglibProxy为ForumServiceImpl动态创建了一个织入性能监视逻辑的代理对象，并调用代理类的业务方法。运行上面的代码，输入以下信息：

引用

begin monitor...

模拟删除Forum记录:10

end monitor...

com.baobaotao.proxy.ForumServiceImpl$$EnhancerByCGLIB$$2a9199c0.removeForum花费47毫秒。

begin monitor...

模拟删除Topic记录:1023

end monitor...

com.baobaotao.proxy.ForumServiceImpl$$EnhancerByCGLIB$$2a9199c0.removeTopic花费16毫秒。

观察以上的输出，除了发现两个业务方法中都织入了性能监控的逻辑外，我们还发现代理类的名字是com.baobaotao.proxy.ForumServiceImpl$$EnhancerByCGLIB$$2a9199c0，这个特殊的类就是CGLib为ForumServiceImpl动态创建的子类。

代理知识小结

Spring AOP的底层就是通过使用JDK动态代理或CGLib动态代理技术为目标Bean织入横切逻辑。在这里，我们对前面两节动态创建代理对象作一个小结。

我们虽然通过PerformanceHandler或CglibProxy实现了性能监视横切逻辑的动态织入，但这种实现方式存在三个明显需要改进的地方：

1）目标类的所有方法都添加了性能监视横切逻辑，而有时，这并不是我们所期望的，我们可能只希望对业务类中的某些特定方法添加横切逻辑；

2）我们通过硬编码的方式指定了织入横切逻辑的织入点，即在目标类业务方法的开始和结束前织入代码；

3）我们手工编写代理实例的创建过程，为不同类创建代理时，需要分别编写相应的创建代码，无法做到通用。

以上三个问题，在AOP中占用重要的地位，因为Spring AOP的主要工作就是围绕以上三点展开：Spring AOP通过Pointcut（切点）指定在哪些类的哪些方法上织入横切逻辑，通过Advice（增强）描述横切逻辑和方法的具体织入点（方法前、方法后、方法的两端等）。此外，Spring通过Advisor（切面）将Pointcut和Advice两者组装起来。有了Advisor的信息，Spring就可以利用JDK或CGLib的动态代理技术采用统一的方式为目标Bean创建织入切面的代理对象了。

JDK动态代理所创建的代理对象，在JDK 1.3下，性能强差人意。虽然在高版本的JDK中，动态代理对象的性能得到了很大的提高，但是有研究表明，CGLib所创建的动态代理对象的性能依旧比JDK的所创建的代理对象的性能高不少（大概10倍）。但CGLib在创建代理对象时所花费的时间却比JDK动态代理多（大概8倍），所以对于singleton的代理对象或者具有实例池的代理，因为无须频繁创建代理对象，所以比较适合用CGLib动态代理技术，反之适合用JDK动态代理技术。值得一提的是，由于CGLib采用动态创建子类的方式生成代理对象，所以不能对目标类中的final方法进行代理。

## 5.4 Spring事务管理 - ThreadLocal

我们知道Spring通过各种模板类降低了开发者使用各种数据持久技术的难度。这些模板类都是线程安全的，也就是说，多个DAO可以复用同一个模板实例而不会发生冲突。我们使用模板类访问底层数据，根据持久化技术的不同，模板类需要绑定数据连接或会话的资源。但这些资源本身是非线程安全的，也就是说它们不能在同一时刻被多个线程共享。虽然模板类通过资源池获取数据连接或会话，但资源池本身解决的是数据连接或会话的缓存问题，并非数据连接或会话的线程安全问题。

按照传统经验，如果某个对象是非线程安全的，在多线程环境下，对对象的访问必须采用synchronized进行线程同步。但模板类并未采用线程同步机制，因为线程同步会降低并发性，影响系统性能。此外，通过代码同步解决线程安全的挑战性很大，可能会增强好几倍的实现难度。那么模板类究竟仰仗何种魔法神功，可以在无须线程同步的情况下就化解线程安全的难题呢？答案就是ThreadLocal！

ThreadLocal在Spring中发挥着重要的作用，在管理request作用域的Bean、事务管理、任务调度、AOP等模块都出现了它们的身影，起着举足轻重的作用。要想了解Spring事务管理的底层技术，ThreadLocal是必须攻克的山头堡垒。

### 5.4.1 ThreadLocal是什么 ？

早在JDK 1.2的版本中就提供java.lang.ThreadLocal，ThreadLocal为解决多线程程序的并发问题提供了一种新的思路。使用这个工具类可以很简洁地编写出优美的多线程程序。

ThreadLocal，顾名思义，它不是一个线程，而是线程的一个本地化对象。当工作于多线程中的对象使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程分配一个独立的变量副本。所以每一个线程都可以独立地改变自己的副本，而不会影响其他线程所对应的副本。从线程的角度看，这个变量就像是线程的本地变量，这也是类名中“Local”所要表达的意思。

线程局部变量并不是Java的新发明，很多语言（如IBM XL、FORTRAN）在语法层面就提供线程局部变量。在Java中没有提供语言级支持，而以一种变通的方法，通过ThreadLocal的类提供支持。所以，在Java中编写线程局部变量的代码相对来说要笨拙一些，这也是为什么线程局部变量没有在Java开发者中得到很好普及的原因。

### 5.4.2 ThreadLocal的接口方法

ThreadLocal类接口很简单，只有4个方法，我们先来了解一下。

**void set(Object value)**

设置当前线程的线程局部变量的值；

**public Object get()**

该方法返回当前线程所对应的线程局部变量；

**public void remove()**

将当前线程局部变量的值删除，目的是为了减少内存的占用，该方法是JDK 5.0新增的方法。需要指出的是，当线程结束后，对应该线程的局部变量将自动被垃圾回收，所以显式调用该方法清除线程的局部变量并不是必须的操作，但它可以加快内存回收的速度；

**protected Object initialValue()**

返回该线程局部变量的初始值，该方法是一个protected的方法，显然是为了让子类覆盖而设计的。这个方法是一个延迟调用方法，在线程第1次调用get()或set(Object)时才执行，并且仅执行1次。ThreadLocal中的默认实现直接返回一个null。

值得一提的是，在JDK5.0中，ThreadLocal已经支持泛型，该类的类名已经变为ThreadLocal<T>。API方法也相应进行了调整，新版本的API方法分别是void set(T value)、T get()以及T initialValue()。

ThreadLocal是如何做到为每一个线程维护变量的副本的呢？其实实现的思路很简单：在ThreadLocal类中有一个Map，用于存储每一个线程的变量副本，Map中元素的键为线程对象，而值对应线程的变量副本。我们自己就可以提供一个简单的实现版本：

代码清单9-3 SimpleThreadLocal

Java代码

public class SimpleThreadLocal {

private Map valueMap = Collections.synchronizedMap(new HashMap());

public void set(Object newValue) {

//①键为线程对象，值为本线程的变量副本

valueMap.put(Thread.currentThread(), newValue);

}

public Object get() {

Thread currentThread = Thread.currentThread();

//②返回本线程对应的变量

Object o = valueMap.get(currentThread);

//③如果在Map中不存在，放到Map中保存起来

if (o == null && !valueMap.containsKey(currentThread)) {

o = initialValue();

valueMap.put(currentThread, o);

}

return o;

}

public void remove() {

valueMap.remove(Thread.currentThread());

}

public Object initialValue() {

return null;

}

}

虽然代码清单9 3中这个ThreadLocal实现版本显得比较幼稚，但它和JDK所提供的ThreadLocal类在实现思路上是非常相近的。

一个TheadLocal实例

下面，我们通过一个具体的实例了解一下ThreadLocal的具体使用方法。

代码清单9-4 SequenceNumber

Java代码

package com.baobaotao.basic;

public class SequenceNumber {

//①通过匿名内部类覆盖ThreadLocal的initialValue()方法，指定初始值

private static ThreadLocal<Integer> seqNum = new ThreadLocal<Integer>(){

public Integer initialValue(){

return 0;

}

};

//②获取下一个序列值

public int getNextNum(){

seqNum.set(seqNum.get()+1);

return seqNum.get();

}

public static void main(String[ ] args)

{

SequenceNumber sn = new SequenceNumber();

//③ 3个线程共享sn，各自产生序列号

TestClient t1 = new TestClient(sn);

TestClient t2 = new TestClient(sn);

TestClient t3 = new TestClient(sn);

t1.start();

t2.start();

t3.start();

}

private static class TestClient extends Thread

{

private SequenceNumber sn;

public TestClient(SequenceNumber sn) {

this.sn = sn;

}

public void run()

{

//④每个线程打出3个序列值

for (int i = 0; i < 3; i++) {

System.out.println("thread["+Thread.currentThread().getName()+

"] sn["+sn.getNextNum()+"]");

}

}

}

}

通常我们通过匿名内部类的方式定义ThreadLocal的子类，提供初始的变量值，如①处所示。TestClient线程产生一组序列号，在③处，我们生成3个TestClient，它们共享同一个SequenceNumber实例。运行以上代码，在控制台上输出以下的结果：

引用

thread[Thread-2] sn[1]

thread[Thread-0] sn[1]

thread[Thread-1] sn[1]

thread[Thread-2] sn[2]

thread[Thread-0] sn[2]

thread[Thread-1] sn[2]

thread[Thread-2] sn[3]

thread[Thread-0] sn[3]

thread[Thread-1] sn[3]

考查输出的结果信息，我们发现每个线程所产生的序号虽然都共享同一个Sequence Number实例，但它们并没有发生相互干扰的情况，而是各自产生独立的序列号，这是因为我们通过ThreadLocal为每一个线程提供了单独的副本。

### 5.4.3与Thread同步机制的比较

ThreadLocal和线程同步机制相比有什么优势呢？ThreadLocal和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。

在同步机制中，通过对象的锁机制保证同一时间只有一个线程访问变量。这时该变量是多个线程共享的，使用同步机制要求程序缜密地分析什么时候对变量进行读写，什么时候需要锁定某个对象，什么时候释放对象锁等繁杂的问题，程序设计和编写难度相对较大。

而ThreadLocal则从另一个角度来解决多线程的并发访问。ThreadLocal为每一个线程提供一个独立的变量副本，从而隔离了多个线程对访问数据的冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的对象封装，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal。

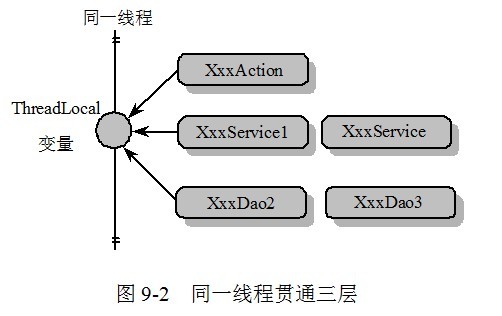
由于ThreadLocal中可以持有任何类型的对象，低版本JDK所提供的get()返回的是Object对象，需要强制类型转换。但JDK 5.0通过泛型很好的解决了这个问题，在一定程度上简化ThreadLocal的使用，代码清单9-2就使用了JDK 5.0新的ThreadLocal<T>版本。

概括起来说，对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式：访问串行化，对象共享化。而ThreadLocal采用了“以空间换时间”的方式：访问并行化，对象独享化。前者仅提供一份变量，让不同的线程排队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响。

### 5.4.4 Spring使用ThreadLocal解决线程安全问题

我们知道在一般情况下，只有无状态的Bean才可以在多线程环境下共享，在Spring中，绝大部分Bean都可以声明为singleton作用域。就是因为Spring对一些Bean（如RequestContextHolder、TransactionSynchronizationManager、LocaleContextHolder等）中非线程安全的“状态性对象”采用ThreadLocal进行封装，让它们也成为线程安全的“状态性对象”，因此有状态的Bean就能够以singleton的方式在多线程中正常工作了。

一般的Web应用划分为展现层、服务层和持久层三个层次，在不同的层中编写对应的逻辑，下层通过接口向上层开放功能调用。在一般情况下，从接收请求到返回响应所经过的所有程序调用都同属于一个线程，如图9-2所示。



这样用户就可以根据需要，将一些非线程安全的变量以ThreadLocal存放，在同一次请求响应的调用线程中，所有对象所访问的同一ThreadLocal变量都是当前线程所绑定的。

下面的实例能够体现Spring对有状态Bean的改造思路：

代码清单9-5 TopicDao：非线程安全

Java代码

public class TopicDao {

//①一个非线程安全的变量

private Connection conn;

public void addTopic(){

//②引用非线程安全变量

Statement stat = conn.createStatement();

…

}

}

由于①处的conn是成员变量，因为addTopic()方法是非线程安全的，必须在使用时创建一个新TopicDao实例（非singleton）。下面使用ThreadLocal对conn这个非线程安全的“状态”进行改造：

代码清单9-6 TopicDao：线程安全

Java代码

import java.sql.Connection;

import java.sql.Statement;

public class TopicDao {

//①使用ThreadLocal保存Connection变量

private static ThreadLocal<Connection> connThreadLocal = new ThreadLocal<Connection>();

public static Connection getConnection(){

//②如果connThreadLocal没有本线程对应的Connection创建一个新的Connection，

//并将其保存到线程本地变量中。

if (connThreadLocal.get() == null) {

Connection conn = ConnectionManager.getConnection();

connThreadLocal.set(conn);

return conn;

}else{

//③直接返回线程本地变量

return connThreadLocal.get();

}

}

public void addTopic() {

//④从ThreadLocal中获取线程对应的

Statement stat = getConnection().createStatement();

}

}

不同的线程在使用TopicDao时，先判断connThreadLocal.get()是否为null，如果为null，则说明当前线程还没有对应的Connection对象，这时创建一个Connection对象并添加到本地线程变量中；如果不为null，则说明当前的线程已经拥有了Connection对象，直接使用就可以了。这样，就保证了不同的线程使用线程相关的Connection，而不会使用其他线程的Connection。因此，这个TopicDao就可以做到singleton共享了。

当然，这个例子本身很粗糙，将Connection的ThreadLocal直接放在Dao只能做到本Dao的多个方法共享Connection时不发生线程安全问题，但无法和其他Dao共用同一个Connection，要做到同一事务多Dao共享同一个Connection，必须在一个共同的外部类使用ThreadLocal保存Connection。但这个实例基本上说明了Spring对有状态类线程安全化的解决思路。在本章后面的内容中，我们将详细说明Spring如何通过ThreadLocal解决事务管理的问题。

## 5.5 Spring事务管理 – 事务基础知识

Spring虽然提供了灵活方便的事务管理功能，但这些功能都是基于底层数据库本身的事务处理机制工作的。要深入了解Spring的事务管理和配置，有必要先对数据库事务的基础知识进行学习。

### 5.5.1何为数据库事务

“一荣俱荣，一损俱损”这句话很能体现事务的思想，很多复杂的事物要分步进行，但它们组成一个整体，要么整体生效，要么整体失效。这种思想反映到数据库上，就是多个SQL语句，要么所有执行成功，要么所有执行失败。

数据库事务有严格的定义，它必须同时满足 4 个特性：原子性（Atomic）、一致性（Consistency）、隔离性（Isolation）和持久性（Durabiliy），简称为ACID。下面是对每个特性的说明。

原子性：表示组成一个事务的多个数据库操作是一个不可分割的原子单元，只有所有的操作执行成功，整个事务才提交，事务中任何一个数据库操作失败，已经执行的任何操作都必须撤销，让数据库返回到初始状态。

一致性：事务操作成功后，数据库所处的状态和它的业务规则是一致的，即数据不会被破坏。如从A账户转账100元到B账户，不管操作成功与否，A和B的存款总额是不变的。

隔离性：在并发数据操作时，不同的事务拥有各自的数据空间，它们的操作不会对对方产生干扰。准确地说，并非要求做到完全无干扰，数据库规定了多种事务隔离级别，不同隔离级别对应不同的干扰程度，隔离级别越高，数据一致性越好，但并发性越弱。

持久性：一旦事务提交成功后，事务中所有的数据操作都必须被持久化到数据库中，即使提交事务后，数据库马上崩溃，在数据库重启时，也必须能保证能够通过某种机制恢复数据。

在这些事务特性中，数据“一致性”是最终目标，其他的特性都是为达到这个目标的措施、要求或手段。

数据库管理系统一般采用重执行日志保证原子性、一致性和持久性，重执行日志记录了数据库变化的每一个动作，数据库在一个事务中执行一部分操作后发生错误退出，数据库即可以根据重执行日志撤销已经执行的操作。此外，对于已经提交的事务，即使数据库崩溃，在重启数据库时也能够根据日志对尚未持久化的数据进行相应的重执行操作。

和Java程序采用对象锁机制进行线程同步类似，数据库管理系统采用数据库锁机制保证事务的隔离性。当多个事务试图对相同的数据进行操作时，只有持有锁的事务才能操作数据，直到前一个事务完成后，后面的事务才有机会对数据进行操作。Oracle数据库还使用了数据版本的机制，在回滚段为数据的每个变化都保存一个版本，使数据的更改不影响数据的读取。

### 5.5.2数据并发的问题

一个数据库可能拥有多个访问客户端，这些客户端都可以并发方式访问数据库。数据库中的相同数据可能同时被多个事务访问，如果没有采取必要的隔离措施，就会导致各种并发问题，破坏数据的完整性。这些问题可以归结为5类，包括3类数据读问题（脏读、不可重复读和幻象读）以及2类数据更新问题（第一类丢失更新和第二类丢失更新）。下面，我们分别通过实例讲解引发问题的场景。

**脏读（dirty read）**

A事务读取B事务尚未提交的更改数据，并在这个数据的基础上操作。如果恰巧B事务回滚，那么A事务读到的数据根本是不被承认的。来看取款事务和转账事务并发时引发的脏读场景：



在这个场景中，B希望取款500元而后又撤销了动作，而A往相同的账户中转账100元，就因为A事务读取了B事务尚未提交的数据，因而造成账户白白丢失了500元。在Oracle数据库中，不会发生脏读的情况。

引用

一个有结巴的人在饮料店柜台前转悠，老板很热情地迎上来说：“喝一瓶？”结巴连忙说：“我…喝…喝…”老板麻利地打开易拉罐递给结巴，结巴终于憋出了他的那句话：“我…喝…喝…喝不起啊！”

**不可重复读（unrepeatable read）**

不可重复读是指A事务读取了B事务已经提交的更改数据。假设A在取款事务的过程中，B往该账户转账100元，A两次读取账户的余额发生不一致：



在同一事务中，T4时间点和T7时间点读取账户存款余额不一样。

**幻象读（phantom read）**

A事务读取B事务提交的新增数据，这时A事务将出现幻象读的问题。幻象读一般发生在计算统计数据的事务中，举一个例子，假设银行系统在同一个事务中，两次统计存款账户的总金额，在两次统计过程中，刚好新增了一个存款账户，并存入100元，这时，两次统计的总金额将不一致：



如果新增数据刚好满足事务的查询条件，这个新数据就进入了事务的视野，因而产生了两个统计不一致的情况。

幻象读和不可重复读是两个容易混淆的概念，前者是指读到了其他已经提交事务的新增数据，而后者是指读到了已经提交事务的更改数据（更改或删除），为了避免这两种情况，采取的对策是不同的，防止读取到更改数据，只需要对操作的数据添加行级锁，阻止操作中的数据发生变化，而防止读取到新增数据，则往往需要添加表级锁——将整个表锁定，防止新增数据（Oracle使用多版本数据的方式实现）。

**第一类丢失更新**

A事务撤销时，把已经提交的B事务的更新数据覆盖了。这种错误可能造成很严重的问题，通过下面的账户取款转账就可以看出来：



A事务在撤销时，“不小心”将B事务已经转入账户的金额给抹去了。

**第二类丢失更新**

A事务覆盖B事务已经提交的数据，造成B事务所做操作丢失：



上面的例子里由于支票转账事务覆盖了取款事务对存款余额所做的更新，导致银行最后损失了100元，相反如果转账事务先提交，那么用户账户将损失100元。

### 5.5.3数据库锁机制

数据并发会引发很多问题，在一些场合下有些问题是允许的，但在另外一些场合下可能却是致命的。数据库通过锁的机制解决并发访问的问题，虽然不同的数据库在实现细节上存在差别，但原理基本上是一样的。

按锁定的对象的不同，一般可以分为表锁定和行锁定，前者对整个表进行锁定，而后者对表中特定行进行锁定。从并发事务锁定的关系上看，可以分为共享锁定和独占锁定。共享锁定会防止独占锁定，但允许其他的共享锁定。而独占锁定既防止其他的独占锁定，也防止其他的共享锁定。为了更改数据，数据库必须在进行更改的行上施加行独占锁定，INSERT、UPDATE、DELETE和SELECT FOR UPDATE语句都会隐式采用必要的行锁定。下面我们介绍一下Oracle数据库常用的5种锁定。

* 行共享锁定：一般通过SELECT FOR UPDATE语句隐式获得行共享锁定，在Oracle中用户也可以通过LOCK TABLE IN ROW SHARE MODE语句显式获得行共享锁定。行共享锁定并不防止对数据行进行更改的操作，但是可以防止其他会话获取独占性数据表锁定。允许进行多个并发的行共享和行独占性锁定，还允许进行数据表的共享或者采用共享行独占锁定。
* 行独占锁定：通过一条INSERT、UPDATE或DELETE语句隐式获取，或者通过一条LOCK TABLE IN ROW EXCLUSIVE MODE语句显式获取。这个锁定可以防止其他会话获取一个共享锁定、共享行独占锁定或独占锁定。
* 表共享锁定：通过LOCK TABLE IN SHARE MODE语句显式获得。这种锁定可以防止其他会话获取行独占锁定（INSERT、UPDATE或DELETE），或者防止其他表共享行独占锁定或表独占锁定，它允许在表中拥有多个行共享和表共享锁定。该锁定可以让会话具有对表事务级一致性访问，因为其他会话在用户提交或者回溯该事务并释放对该表的锁定之前不能更改这个被锁定的表。
* 表共享行独占：通过LOCK TABLE IN SHARE ROW EXCLUSIVE MODE语句显式获得。这种锁定可以防止其他会话获取一个表共享、行独占或者表独占锁定，它允许其他行共享锁定。这种锁定类似于表共享锁定，只是一次只能对一个表放置一个表共享行独占锁定。如果A会话拥有该锁定，则B会话可以执行SELECT FOR UPDATE操作，但如果B会话试图更新选择的行，则需要等待。
* 表独占：通过LOCK TABLE IN EXCLUSIVE MODE显式获得。这个锁定防止其他会话对该表的任何其他锁定。

### 5.5.4事务隔离级别

尽管数据库为用户提供了锁的DML操作方式，但直接使用锁管理是非常麻烦的，因此数据库为用户提供了自动锁机制。只要用户指定会话的事务隔离级别，数据库就会分析事务中的SQL语句，然后自动为事务操作的数据资源添加上适合的锁。此外数据库还会维护这些锁，当一个资源上的锁数目太多时，自动进行锁升级以提高系统的运行性能，而这一过程对用户来说完全是透明的。

ANSI/ISO SQL 92标准定义了4个等级的事务隔离级别，在相同数据环境下，使用相同的输入，执行相同的工作，根据不同的隔离级别，可以导致不同的结果。不同事务隔离级别能够解决的数据并发问题的能力是不同的，如表9-1所示。



事务的隔离级别和数据库并发性是对立的，两者此增彼长。一般来说，使用READ UNCOMMITED隔离级别的数据库拥有最高的并发性和吞吐量，而使用SERIALIZABLE隔离级别的数据库并发性最低。

SQL 92定义READ UNCOMMITED主要是为了提供非阻塞读的能力，Oracle虽然也支持READ UNCOMMITED，但它不支持脏读，因为Oracle使用多版本机制彻底解决了在非阻塞读时读到脏数据的问题并保证读的一致性，所以，Oracle的READ COMMITTED隔离级别就已经满足了SQL 92标准的REPEATABLE READ隔离级别。

SQL 92推荐使用REPEATABLE READ以保证数据的读一致性，不过用户可以根据应用的需要选择适合的隔离等级。

### 5.5.5JDBC对事务支持

并不是所有的数据库都支持事务，即使支持事务的数据库也并非支持所有的事务隔离级别，用户可以通过Connection#getMetaData()方法获取DatabaseMetaData对象，并通过该对象的supportsTransactions()、supportsTransactionIsolationLevel(int level)方法查看底层数据库的事务支持情况。

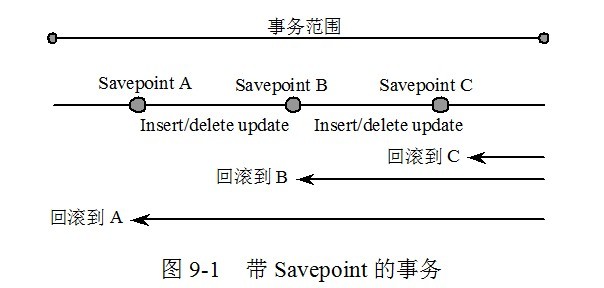
Connection默认情况下是自动提交的，也即每条执行的SQL都对应一个事务，为了能够将多条SQL当成一个事务执行，必须先通过Connection#setAutoCommit(false)阻止Connection自动提交，并可通过Connection#setTransactionIsolation()设置事务的隔离级别，Connection中定义了对应SQL 92标准4个事务隔离级别的常量。通过Connection#commit()提交事务，通过Connection#rollback()回滚事务。下面是典型的JDBC事务数据操作的代码：

代码清单9-1 JDBC事务代码

Java代码

1. Connection conn ;
2. **try**{
3. conn ＝ DriverManager.getConnection();//①获取数据连接
4. conn.setAutoCommit(**false**); //②关闭自动提交的机制
5. conn.setTransactionIsolation(Connection.TRANSACTION\_SERIALIZABLE); //③设置事务隔离级别
7. Statement stmt = conn.createStatement();
9. **int** rows = stmt.executeUpdate( "INSERT INTO t\_topic VALUES(1,’tom’) " );
10. rows = stmt.executeUpdate( "UPDATE t\_user set topic\_nums = topic\_nums +1 "+
11. "WHERE user\_id = 1");
13. conn.commit();//④提交事务
14. }**catch**(Exception e){
15. …
16. conn.rollback();//⑤回滚事务
17. }**finally**{
18. …
19. }

在JDBC 2.0中，事务最终只能有两个操作：提交和回滚。但是，有些应用可能需要对事务进行更多的控制，而不是简单地提交或回滚。JDBC 3.0（JDK 1.4及以后的版本）引入了一个全新的保存点特性，Savepoint 接口允许用户将事务分割为多个阶段，用户可以指定回滚到事务的特定保存点，而并非像JDBC 2.0一样只回滚到开始事务的点，如图9-1所示。



下面的代码使用了保存点的功能，在发生特定问题时，回滚到指定的保存点，而非回滚整个事务，如代码清单9-2所示：

代码清单9-2 使用保存点的事务代码

Java代码

1. …
2. Statement stmt = conn.createStatement();
3. **int** rows = stmt.executeUpdate( "INSERT INTO t\_topic VALUES(1,’tom’)");
5. Savepoint svpt = conn.setSavepoint("savePoint1");//①设置一个保存点
6. rows = stmt.executeUpdate( "UPDATE t\_user set topic\_nums = topic\_nums +1 "+
7. "WHERE user\_id = 1");
8. …
9. //②回滚到①处的savePoint1，①之前的SQL操作，在整个事务提交后依然提交，
10. //但①到②之间的SQL操作被撤销了
11. conn.rollback(svpt);
12. …
13. conn.commit();//③提交事务

并非所有数据库都支持保存点功能，用户可以通过DatabaseMetaData#supportsSavepoints()方法查看是否支持。

## 5.6 Spring MVC - 国际化知识

假设我们正在开发一个支持多国语言的Web应用程序，要求系统能够根据客户端的系统的语言类型返回对应的界面：英文的操作系统返回英文界面，而中文的操作系统则返回中文界面——这便是典型的i18n国际化问题。对于有国际化要求的应用系统，我们不能简单地采用硬编码的方式编写用户界面信息、报错信息等内容，而必须为这些需要国际化的信息进行特殊处理。简单来说，就是为每种语言提供一套相应的资源文件，并以规范化命名的方式保存在特定的目录中，由系统自动根据客户端语言选择适合的资源文件。

### 5.6.1 基础知识

“国际化信息”也称为“本地化信息”，一般需要两个条件才可以确定一个特定类型的本地化信息，它们分别是“语言类型”和“国家/地区的类型”。如中文本地化信息既有中国大陆地区的中文，又有中国台湾、中国香港地区的中文，还有新加坡地区的中文。Java通过java.util.Locale类表示一个本地化对象，它允许通过语言参数和国家/地区参数创建一个确定的本地化对象。

语言参数使用ISO标准语言代码表示，这些代码是由ISO-639标准定义的，每一种语言由两个小写字母表示。在许多网站上都可以找到这些代码的完整列表，下面的网址是提供了标准语言代码的信息：

<http://www.loc.gov/standards/iso639-2/php/English_list.php>。

国家/地区参数也由标准的ISO国家/地区代码表示，这些代码是由ISO-3166标准定义的，每个国家/地区由两个大写字母表示。用户可以从以下网址查看ISO-3166的标准代码：

[http://www.iso.ch/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html。](http://www.iso.ch/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html%E3%80%82)

表5-2给出了一些语言和国家/地区的标准代码：



### 5.6.2 Locale

java.util.Locale是表示语言和国家/地区信息的本地化类，它是创建国际化应用的基础。下面给出几个创建本地化对象的示例：

Java代码

1. //①带有语言和国家/地区信息的本地化对象
2. Locale locale1 = **new** Locale("zh","CN");
4. //②只有语言信息的本地化对象
5. Locale locale2 = **new** Locale("zh");
7. //③等同于Locale("zh","CN")
8. Locale locale3 = Locale.CHINA;
10. //④等同于Locale("zh")
11. Locale locale4 = Locale.CHINESE;
13. //⑤获取本地系统默认的本地化对象
14. Locale locale 5= Locale.getDefault();

用户既可以同时指定语言和国家/地区参数定义一个本地化对象①，也可以仅通过语言参数定义一个泛本地化对象②。Locale类中通过静态常量定义了一些常用的本地化对象，③和④处就直接通过引用常量返回本地化对象。此外，用户还可以获取系统默认的本地化对象，如⑤所示。

注意：在测试时，如果希望改变系统默认的本地化设置，可以在启动JVM时通过命令参数指定：java -Duser.language=en -Duser.region=US MyTest。

### 5.6.3本地化工具类

JDK的java.util包中提供了几个支持本地化的格式化操作工具类：NumberFormat、DateFormat、MessageFormat。下面，我们分别通过实例了解它们的用法：

Java代码

1. Locale locale = **new** Locale("zh", "CN");
2. NumberFormat currFmt = NumberFormat.getCurrencyInstance(locale);
3. **double** amt = 123456.78;
4. System.out.println(currFmt.format(amt));

上面的实例通过NumberFormat按本地化的方式对货币金额进行格式化操作，运行实例，输出以下信息：￥123,456.78

Java代码

1. Locale locale = **new** Locale("en", "US");
2. Date date = **new** Date();
3. DateFormat df = DateFormat.getDateInstance(DateFormat.MEDIUM, locale);
4. System.out.println(df.format(date));

通过DateFormat#getDateInstance(int style,Locale locale)方法按本地化的方式对日期进行格式化操作。该方法第一个入参为时间样式，第二个入参为本地化对象。运行以上代码，输出以下信息：Jan 8, 2007

MessageFormat在NumberFormat和DateFormat的基础上提供了强大的占位符字符串的格式化功能，它支持时间、货币、数字以及对象属性的格式化操作。下面的实例演示了一些常见的格式化功能：

1. //①信息格式化串
2. String pattern1 = "{0}，你好！你于{1}在工商银行存入{2} 元。";
3. String pattern2 = "At {1,time,short} On{1,date,long}，{0} paid {2,number, currency}.";
5. //②用于动态替换占位符的参数
6. Object[] params = {"John", **new** GregorianCalendar().getTime(),1.0E3};
8. //③使用默认本地化对象格式化信息
9. String msg1 = MessageFormat.format(pattern1,params);
11. //④使用指定的本地化对象格式化信息
12. MessageFormat mf = **new** MessageFormat(pattern2,Locale.US);
13. String msg2 = mf.format(params);
14. System.out.println(msg1);
15. System.out.println(msg2);

pattern1是简单形式的格式化信息串，通过{n}占位符指定动态参数的替换位置索引，{0}表示第一个参数，{1}表示第二个参数，以此类推。

pattern2格式化信息串比较复杂一些，除参数位置索引外，还指定了参数的类型和样式。从pattern2中可以看出格式化信息串的语法是很灵活的，一个参数甚至可以出现在两个地方：如 {1,time,short}表示从第二个入参中获取时间部分的值，显示为短样式时间；而{1,date,long}表示从第二个入参中获取日期部分的值，显示为长样式时间。关于MessageFormat更详细的使用方法，请参见JDK的Javadoc。

在②处，定义了用于替换格式化占位符的动态参数，这里，我们使用到了JDK5.0自动装包的语法，否则必须采用封装类表示基本类型的参数值。

在③处，通过MessageFormat的format()方法格式化信息串。它使用了系统默认的本地化对象，由于我们是中文平台，因此默认为Locale.CHINA。而在④处，我们显式指定MessageFormat的本地化对象。

运行上面的代码，输出以下信息：

John，你好！你于07-1-8 下午9:58在工商银行存入1,000元。   
At 9:58 PM OnJanuary 8, 2007，John paid $1,000.00.   
ResourceBoundle

如果应用系统中某些信息需要支持国际化功能，则必须为希望支持的不同本地化类型分别提供对应的资源文件，并以规范的方式进行命名。国际化资源文件的命名规范规定资源名称采用以下的方式进行命名：

<资源名>\_<语言代码>\_<国家/地区代码>.properties

## 5.7 Spring MVC – HTTP报文

学习Web开发不好好学习HTTP报文，将会“打拳不练功，到老一场空”，你花在犯迷糊上的时间比你沉下心来学习HTTP的时间肯定会多很多。

### 5.7.1 HTTP请求报文解剖

HTTP请求报文由3部分组成（请求行+请求头+请求体）：



下面是一个实际的请求报文：



①是请求方法，GET和POST是最常见的HTTP方法，除此以外还包括DELETE、HEAD、OPTIONS、PUT、TRACE。不过，当前的大多数浏览器只支持GET和POST，Spring 3.0提供了一个HiddenHttpMethodFilter，允许你通过“\_method”的表单参数指定这些特殊的HTTP方法（实际上还是通过POST提交表单）。服务端配置了HiddenHttpMethodFilter后，Spring会根据\_method参数指定的值模拟出相应的HTTP方法，这样，就可以使用这些HTTP方法对处理方法进行映射了。   
  
②为请求对应的URL地址，它和报文头的Host属性组成完整的请求URL，③是协议名称及版本号。   
  
④是HTTP的报文头，报文头包含若干个属性，格式为“属性名:属性值”，服务端据此获取客户端的信息。   
  
⑤是报文体，它将一个页面表单中的组件值通过param1=value1&param2=value2的键值对形式编码成一个格式化串，它承载多个请求参数的数据。不但报文体可以传递请求参数，请求URL也可以通过类似于“/chapter15/user.html? param1=value1&param2=value2”的方式传递请求参数。   
  
对照上面的请求报文，我们把它进一步分解，你可以看到一幅更详细的结构图：



参考：HttpWatch是强大的网页数据分析工具，安装后将集成到Internet Explorer工具栏中。它不用代理服务器或一些复杂的网络监控工具，就能抓取请求及响应的完整信息，包括Cookies、消息头、查询参数、响应报文等，是Web应用开发人员的必备工具。

### **5.7.2 HTTP请求报文头属性**

常见的HTTP请求报文头属性

**Accept**

请求报文可通过一个“Accept”报文头属性告诉服务端 客户端接受什么类型的响应。

如下报文头相当于告诉服务端，俺客户端能够接受的响应类型仅为纯文本数据啊，你丫别发其它什么图片啊，视频啊过来，那样我会歇菜的~~~：

Accept:text/plain

Accept属性的值可以为一个或多个MIME类型的值，关于MIME类型，大家请参考：<http://en.wikipedia.org/wiki/MIME_type>

Cookie

这是第一个要说的，客户端的Cookie就是通过这个报文头属性传给服务端的哦！如下所示：

Cookie: $Version=1; Skin=**new**;jsessionid=5F4771183629C9834F8382E23BE13C4C

注意到后台的那个jsessionid=5F4771183629C9834F8382E23BE13C4C没有，服务端是怎么知道客户端的多个请求是属于一个Session的，原来就是通过HTTP请求报文头的Cookie属性的jsessionid的值关联起来的！（当然也可以通过重写URL的方式将会话ID附带在每个URL的后后面哦）。

**Referer**

表示这个请求是从哪个URL过来的，假如你通过google搜索出一个商家的广告页面，你对这个广告页面感兴趣，鼠标一点发送一个请求报文到商家的网站，这个请求报文的Referer报文头属性值就是http://www.google.com。

很多貌似神奇的网页监控软件（如著名的 我要啦），只要在你的网页上放上一段JavaScript，就可以帮你监控流量，全国访问客户的分布情况等报表和图表，其原理就是通过这个Referer及其它一些HTTP报文头工作的。

**Cache-Control**

对缓存进行控制，如一个请求希望响应返回的内容在客户端要被缓存一年，或不希望被缓存就可以通过这个报文头达到目的。

如以下设置，相当于让服务端将对应请求返回的响应内容不要在客户端缓存（当然响应报文也是通过响应报文头通知浏览器客户端的，这个下面再说）：

Cache-Control: no-cache

### 5.7.3 如何访问请求报文头

由于请求报文头是客户端发过来的，服务端当然只能读取了，以下是HttpServletRequest一些用于读取请求报文头的API：

1. //获取请求报文中的属性名称
2. java.util.Enumeration<java.lang.String>   getHeaderNames();
4. //获取指定名称的报文头属性的值
5. java.lang.String getHeader(java.lang.String name) //获取报文头中的Cookie(读取Cookie的报文头属性）
6. Cookie[]   getCookies() ;
8. //获取客户端本地化信息（读取 Accept-Language 的报文头属性）
9. java.util.Locale    getLocale()
11. //获取请求报文体的长度（读取Content-Length的报文头属性）
12. **int** getContentLength();
13. HttpSession getSession()

获取请求所关联的HttpSession,其内部的机理是通过读取请求报文头中Cookie属性的JSESSIONID的值，在服务端的一个会话Map中，根据这个JSESSIONID获取对应的HttpSession的对象。（这样，你就不会觉得HttpSession很神秘了吧，你自己也可以做一个类似的会话管理）

### 5.7.4 HTTP响应报文解剖

HTTP的响应报文也由三部分组成（响应行+响应头+响应体）：



以下是一个实际的HTTP响应报文：



①报文协议及版本；   
②状态码及状态描述；   
③响应报文头，也是由多个属性组成；   
④响应报文体，即我们真正要的“干货”。

**响应状态码**

和请求报文相比，响应报文多了一个“响应状态码”，它以“清晰明确”的语言告诉客户端本次请求的处理结果。

HTTP的响应状态码由5段组成：

1xx 消息，一般是告诉客户端，请求已经收到了，正在处理，别急...

2xx 处理成功，一般表示：请求收悉、我明白你要的、请求已受理、已经处理完成等信息.

3xx 重定向到其它地方。它让客户端再发起一个请求以完成整个处理。

4xx 处理发生错误，责任在客户端，如客户端的请求一个不存在的资源，客户端未被授权，禁止访问等。

5xx 处理发生，责任在服务端，如服务端抛出异常，路由出错，HTTP版本不支持等。

### 5.7.5 HTTP响应报文头属性

**Cache-Control**

响应输出到客户端后，服务端通过该报文头属告诉客户端如何控制响应内容的缓存。

下面，的设置让客户端对响应内容缓存3600秒，也即在3600秒内，如果客户再次访问该资源，直接从客户端的缓存中返回内容给客户，不要再从服务端获取（当然，这个功能是靠客户端实现的，服务端只是通过这个属性提示客户端“应该这么做”，做不做，还是决定于客户端，如果是自己宣称支持HTTP的客户端，则就应该这样实现）。

Cache-Control: max-age=3600

**ETag**

一个代表响应服务端资源（如页面）版本的报文头属性，如果某个服务端资源发生变化了，这个ETag就会相应发生变化。它是Cache-Control的有益补充，可以让客户端“更智能”地处理什么时候要从服务端取资源，什么时候可以直接从缓存中返回响应。

关于ETag的说明，你可以参见：<http://en.wikipedia.org/wiki/HTTP_ETag>。

Spring 3.0还专门为此提供了一个org.springframework.web.filter.ShallowEtagHeaderFilter（实现原理很简单，对JSP输出的内容MD5，这样内容有变化ETag就相应变化了），用于生成响应的ETag，因为这东东确实可以帮助减少请求和响应的交互。

ETag: "737060cd8c284d8af7ad3082f209582d"

**Location**

我们在JSP中让页面Redirect到一个某个A页面中，其实是让客户端再发一个请求到A页面，这个需要Redirect的A页面的URL，其实就是通过响应报文头的Location属性告知客户端的，如下的报文头属性，将使客户端redirect到iteye的首页中：

Location: http://www.iteye.com

**Set-Cookie**

服务端可以设置客户端的Cookie，其原理就是通过这个响应报文头属性实现的：

Set-Cookie: UserID=JohnDoe; Max-Age=3600; Version=1

## 5.8 Spring IoC – 属性编辑器

在Spring配置文件里，我们往往通过字面值为Bean各种类型的属性提供设置值：不管是double类型还是int类型，在配置文件中都对应字符串类型的字面值。BeanWrapper填充Bean属性时如何将这个字面值转换为对应的double或int等内部类型呢？我们可以隐约地感觉到一定有一个转换器在其中起作用，这个转换器就是属性编辑器。

“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。

PropertyEditor是JavaBean规范定义的接口，JavaBean规范中还有其他一些PropertyEditor配置的接口。为了彻底理解属性编辑器，必须对JavaBean中有关属性编辑器的规范进行学习，相信这些知识对学习和掌握Spring中的属性编辑器会大有帮助。

### 5.8.1 JavaBean的编辑器

Sun所制定的JavaBean规范，很大程度上是为IDE准备的——它让IDE能够以可视化的方式设置JavaBean的属性。如果在IDE中开发一个可视化应用程序，我们需要通过属性设置的方式对组成应用的各种组件进行定制，IDE通过属性编辑器让开发人员使用可视化的方式设置组件的属性。

一般的IDE都支持JavaBean规范所定义的属性编辑器，当组件开发商发布一个组件时，它往往将组件对应的属性编辑器捆绑发行，这样开发者就可以在IDE环境下方便地利用属性编辑器对组件进行定制工作。

JavaBean规范通过java.beans.PropertyEditor定义了设置JavaBean属性的方法，通过BeanInfo描述了JavaBean哪些属性是可定制的，此外还描述了可定制属性与PropertyEditor的对应关系。

BeanInfo与JavaBean之间的对应关系，通过两者之间规范的命名确立：对应JavaBean的BeanInfo采用如下的命名规范：<Bean>BeanInfo。如ChartBean对应的BeanInfo为ChartBeanBeanInfo；Car对应的BeanInfo为CarBeanInfo。当JavaBean连同其属性编辑器相同的组件注册到IDE中后，当在开发界面中对JavaBean进行定制时，IDE就会根据JavaBean规范找到对应的BeanInfo，再根据BeanInfo中的描述信息找到JavaBean属性描述（是否开放、使用哪个属性编辑器），进而为JavaBean生成特定开发编辑界面。

JavaBean规范提供了一个管理默认属性编辑器的管理器：PropertyEditorManager，该管理器内保存着一些常见类型的属性编辑器，如果某个JavaBean的常见类型属性没有通过BeanInfo显式指定属性编辑器，IDE将自动使用PropertyEditorManager中注册的对应默认属性编辑器。

由于JavaBean对应的属性编辑器等IDE环境相关的资源和组件需要动态加载，所以在纯Java的IDE中开发基于组件的应用时，总会感觉IDE反应很迟钝，不像Delphi、C++Builder一样灵敏快捷。但在Eclipse开发环境中，设计包括可视化组件的应用时却很快捷，原因是Eclipse没有使用Java的标准用户界面组件库，当然也就没有按照JavaBean的规范开发设计GUI组件了

# 6 Spring 事务管理

Spring框架引人注目的重要因素之一是它全面的事务支持。Spring框架提供了一致的事务管理抽象，这带来了以下好处：

* 为复杂的事务API提供了一致的编程模型，如JTA、JDBC、Hibernate、JPA和JDO
* 支持 声明式事务管理
* 提供比大多数复杂的事务API（诸如JTA）更简单的，更易于使用的 编程式 事务管理API
* 非常好地整合Spring的各种数据访问抽象

## 6.1 Spring事务抽象

Spring事务抽象的关键是事务策略的概念。这个概念由org.springframework.transaction.PlatformTransactionManager接口定义

TransactionDefinition接口指定：

* 事务隔离：当前事务和其它事务的隔离的程度。例如，这个事务能否看到其他事务未提交的写数据？
* 事务传播：通常在一个事务中执行的所有代码都会在这个事务中运行。但是，如果一个事务上下文已经存在，有几个选项可以指定一个事务性方法的执行行为：例如，简单地在现有的事务中继续运行（大多数情况）；或者挂起现有事务，创建一个新的事务。Spring提供EJB CMT中常见的事务传播选项。
* 事务超时: 事务在超时前能运行多久（自动被底层的事务基础设施回滚）。
* 只读状态: 只读事务不修改任何数据。只读事务在某些情况下（例如当使用Hibernate时），是一种非常有用的优化。

使用Spring时，无论你选择编程式还是声明式的事务管理，定义一个正确的 PlatformTransactionManager 实现都是至关重要的。按照Spring的良好风格，这种重要定义都是通过IoC实现的。

**纯粹JDBC环境**

定义一个JDBC DataSource，然后使用Spring的DataSourceTransactionManager，并传入指向DataSource的引用

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="${jdbc.driverClassName}" />

<property name="url" value="${jdbc.url}" />

<property name="username" value="${jdbc.username}" />

<property name="password" value="${jdbc.password}" />

</bean>

PlatformTransactionManager bean的定义如下:

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

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

</bean>

其他环境(如Hibernate，JDO)参考Spring开发手册。

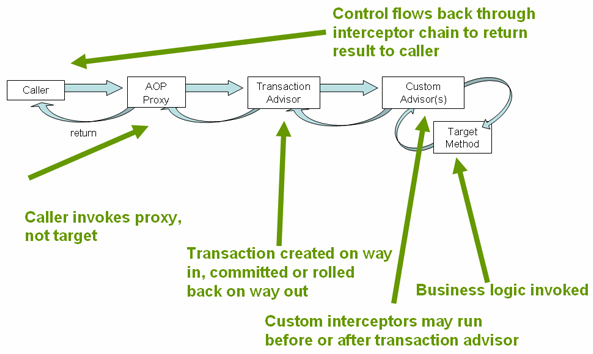
## 6.2 声明式事务管理

大多数Spring用户选择声明式事务管理。这是对应用代码影响最小的选择，因此也最符合 非侵入式 轻量级容器的理念。

Spring的声明式事务管理是通过Spring AOP实现的，因为事务方面的代码与Spring绑定并以一种样板式风格使用，不过尽管如此，你一般并不需要理解AOP概念就可以有效地使用Spirng的声明式事务管理。

### 6.2.1 理解Spring的声明式事务管理实现

Spring的事务管理是通过AOP代理实现的。 其中的事务通知由元数据（目前基于XML或注解）驱动。 代理对象与事务元数据结合产生了一个AOP代理，它使用一个PlatformTransactionManager 实现品配合TransactionInterceptor，在方法调用前后实施事务。概念上来说，在事务代理上调用方法的工作过程看起来像这样：



*示例可参考Spring参考手册*

### 6.2.2 事务回滚

在Spring框架的事务架构里指出当context的事务里的代码抛出 Exception 时事务进行回滚。Spring框架的事务基础架构代码将从调用的堆栈里捕获到任何未处理的 Exception，并将标识事务将回滚。

然而，请注意Spring框架的事务基础架构代码将默认地只在抛出运行时和unchecked exceptions时才标识事务回滚。 也就是说，当抛出一个 RuntimeException 或其子类例的实例时。（Errors 也一样 - 默认地 - 标识事务回滚。）从事务方法中抛出的Checked exceptions将不被标识进行事务回滚。

可以配置哪些 Exception类型将被标识进行事务回滚。 下面的XML配置片断里示范了如何配置一个用于回滚的checked、应用程序特定的 Exception 类型。

<tx:advice id="txAdvice" transaction-manager="txManager">

<tx:attributes>

<tx:method name="get\*" read-only="true" ***rollback-for="NoProductInStockException"***/>

<tx:method name="\*"/>

</tx:attributes>

</tx:advice>

有时候你不想在异常抛出的时候回滚事务，就可以使用“不回滚规则”。 在下面的例子中，我们告诉Spring 框架即使遇到没有经过处理的InstrumentNotFoundException异常，也不要回滚事务。

<tx:advice id="txAdvice">

<tx:attributes>

<tx:method name="updateStock" ***no-rollback-for="InstrumentNotFoundException"***/>

<tx:method name="\*"/>

</tx:attributes>

</tx:advice>

当Spring框架捕获到一个异常后会检查配置回滚规则来决定是不是要回滚事务，这时候会遵循最匹配的规则。 所以在下面这种配置中，除了InstrumentNotFoundException这种类型的异常不会导致事务回滚以外，其他任何类型的异常都会。

<tx:advice id="txAdvice">

<tx:attributes>

<tx:method name="\*" rollback-for="Throwable" no-rollback-for="InstrumentNotFoundException"/>

</tx:attributes>

</tx:advice>

### 6.2.3 <tx:advice/> 的设置

通过 <tx:advice/> 标签来指定不同的事务性设置。默认的 <tx:advice/> 设置如下：

* 事务传播设置 是 REQUIRED
* 隔离级别是DEFAULT
* 事务是 读/写
* 事务超时默认是依赖于事务系统的，或者事务超时没有被支持。
* 任何 RuntimeException 将触发事务回滚，但是任何 checked Exception 将不触发事务回滚

这些默认的设置当然也是可以被改变的。 <tx:advice/> 和 <tx:attributes/> 标签里的 <tx:method/> 各种属性设置总结如下：

| **属性** | **是否需要？** | **默认值** | **描述** |
| --- | --- | --- | --- |
| name | 是 |  | 与事务属性关联的方法名。通配符（\*）可以用来指定一批关联到相同的事务属性的方法。 如：'get\*'、'handle\*'、'on\*Event'等等。 |
| propagation | 不 | REQUIRED | 事务传播行为 |
| isolation | 不 | DEFAULT | 事务隔离级别 |
| timeout | 不 | -1 | 事务超时的时间（以秒为单位） |
| read-only | 不 | false | 事务是否只读？ |
| rollback-for | 不 |  | 将被触发进行回滚的 Exception(s)；以逗号分开。 如：'com.foo.MyBusinessException,ServletException' |
| no-rollback-for | 不 |  | *不* 被触发进行回滚的 Exception(s)；以逗号分开。 如：'com.foo.MyBusinessException,ServletException' |

### 6.2.4 使用 @Transactional

除了基于XML文件的声明式事务配置外，你也可以采用基于注解式的事务配置方法。直接在Java源代码中声明事务语义的做法让事务声明和将受其影响的代码距离更近了，而且一般来说不会有不恰当的耦合的风险，因为，使用事务性的代码几乎总是被部署在事务环境中。

注意：@Transactional 注解及其支持类所提供的功能最低要求使用Java 5（Tiger）。

*// the service class that we want to make transactional*

**@Transactional**

**public** **class** DefaultFooService **implements** FooService {

Foo getFoo(String fooName);

Foo getFoo(String fooName, String barName);

**void** insertFoo(Foo foo);

**void** updateFoo(Foo foo);

}

当上述的POJO定义在Spring IoC容器里时，上述bean实例仅仅通过一 行xml配置就可以使它具有事务性的。如下：

*<!-- from the file 'context.xml' -->*

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

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

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

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

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

xsi:schemaLocation="

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

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

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

http://www.springframework.org/schema/tx/spring-tx-3.0.xsd

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

http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

*<!-- this is the service object that we want to make transactional -->*

<bean id="fooService" class="x.y.service.DefaultFooService"/>

*<!-- enable the configuration of transactional behavior based on annotations -->*

**<tx:annotation-driven transaction-manager="txManager"/>**

*<!-- a PlatformTransactionManager is still required -->*

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

*<!-- (this dependency is defined somewhere else) -->*

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

</bean>

*<!-- other <bean/> definitions here -->*

</beans>

*提示：实际上，如果你用 'transactionManager' 来定义 PlatformTransactionManager bean的名字的话，你就可以忽略 <tx:annotation-driven/> 标签里的 'transaction-manager' 属性。 如果 PlatformTransactionManager bean你要通过其它名称来注入的话，你必须用 'transaction-manager' 属性来指定它，如上所示。*

@Transactional 注解可以被应用于接口定义和接口方法、类定义和类的 public 方法上。 然而，请注意只是使用 @Transactional 注解并不会启用事务行为， 它仅仅 是一种元数据，能够被可以识别 @Transactional 注解和上述的配置适当的具有事务行为的beans所使用。上面的例子中，其实正是 <tx:annotation-driven/>元素的出现 开启 了事务行为。

Spring团队的建议是你只在具体的类上使用 @Transactional 注解， 而不要注解在接口上。你当然可以在接口（或接口方法）上使用 @Transactional 注解， 但是这只有在你使用基于接口的代理时它才会生效。因为注解是 不能继承 的， 这就意味着如果你正在使用基于类的代理时，事务的设置将不能被基于类的代理所识别，而且对象也不会被事务代理所包装 （这是很糟糕的）。 因此，请接受Spring团队的建议，在具体的类（包括该类的方法）上使用 @Transactional 注解。

**方法的可见度和 @Transactional**

在使用代理的时候，@Transactional 注解应该只被应用到 public 可见度的方法上。 如果你在 protected、private 或者 package-visible 的方法上使用 @Transactional 注解，系统也不会报错， 但是这个被注解的方法将不会执行已配置的事务设置。如果你非要注解非公共方法的话，请参考使用AspectJ

**<tx:annotation-driven/> 设置**

| **属性** | **默认值** | **描述** |
| --- | --- | --- |
| transaction-manager | transactionManager | 使用的事务管理器的名字。只有像在上面的例子那样，事务管理器不是 transactionManager的情况下才需要。 |
| mode | proxy | 默认的模式“proxy”会用Spring的AOP框架来代理注解过的bean（就像在前面讨论过的那样， 下面代理的语义只对通过代理传递过来的方法调用起效）。 另一种可行的模式"aspectj"会使用Spring的AspectJ事务切面来编织类（通过修改目标对象的字节码应用到任何方法调用上）。 AspectJ织入需要在classpath中有spring-aspects.jar这个文件，并且启用装载时织入 (或者编译时织入)。 (关于如何设置装载时编织的详情请参见 [第 6.8.4.5 节 “Spring配置”](mk:@MSITStore:E:\learn\开源框架\spring\Spring2.5-中文参考手册.chm::/ch06s08.html#aop-aj-ltw-spring) ) |
| proxy-target-class | false | 只对代理模式有效。决定为那些使用了@Transactional注解的类创建何种事务代理。 如果 "proxy-target-class" 属性被设为 "true"， 那么基于类的代理就会被创建。如果 "proxy-target-class"属性被设为"false" 或者没设，那么基于接口的标准JDK代理就会被创建。(关于不同代理类型的解释请参见 [第 6.6 节 “代理机制”](mk:@MSITStore:E:\learn\开源框架\spring\Spring2.5-中文参考手册.chm::/ch06s06.html) ) |
| order | Ordered.LOWEST\_PRECEDENCE | 定义事务通知的顺序会作用到使用@Transactional注解的bean上。 更多的关于AOP通知顺序的定义可以在章节(参见 [第 6.2.4.7 节 “通知顺序”](mk:@MSITStore:E:\learn\开源框架\spring\Spring2.5-中文参考手册.chm::/ch06s02.html#aop-ataspectj-advice-ordering))。 请注意如果不指定任何顺序将会把决定权交给AOP子系统。 |

**@Transactional 有关的设置**

@Transactional 注解是用来指定接口、类或方法必须拥有事务语义的元数据。 如：“当一个方法开始调用时就开启一个新的只读事务，并停止掉任何现存的事务”。 默认的 @Transactional 设置如下：

* 事务传播设置是 PROPAGATION\_REQUIRED
* 事务隔离级别是 ISOLATION\_DEFAULT
* 事务是 读/写
* 事务超时默认是依赖于事务系统的，或者事务超时没有被支持。
* 任何 RuntimeException 将触发事务回滚，但是任何 checked Exception 将不触发事务回滚

这些默认的设置当然也是可以被改变的。 @Transactional 注解的各种属性设置总结如下：

| **属性** | **类型** | **描述** |
| --- | --- | --- |
| [propagation](mk:@MSITStore:E:\learn\开源框架\spring\Spring2.5-中文参考手册.chm::/ch09s05.html#tx-propagation) | 枚举型：Propagation | 可选的传播性设置 |
| isolation | 枚举型：Isolation | 可选的隔离性级别（默认值：ISOLATION\_DEFAULT） |
| readOnly | 布尔型 | 读写型事务 vs. 只读型事务 |
| timeout | int型（以秒为单位） | 事务超时 |
| rollbackFor | 一组 Class 类的实例，必须是Throwable 的子类 | 一组异常类，遇到时 必须 进行回滚。默认情况下checked exceptions不进行回滚，仅unchecked exceptions（即RuntimeException的子类）才进行事务回滚。 |
| rollbackForClassname | 一组 Class 类的名字，必须是Throwable的子类 | 一组异常类名，遇到时 必须 进行回滚 |
| noRollbackFor | 一组 Class 类的实例，必须是Throwable 的子类 | 一组异常类，遇到时 必须不 回滚。 |
| noRollbackForClassname | 一组 Class 类的名字，必须是Throwable 的子类 | 一组异常类，遇到时 必须不 回滚 |

### 6.2.5 事务传播

在由Spring管理的事务中，请记住物理和逻辑 事务存在的差异，以及传播设置是如何影响到这些差异的。

**required**

当事务传播被设置PROPAGATION\_REQUIRED的时候， 会为每一个被应用到的方法创建一个逻辑事务作用域。 每一个这样的逻辑事务作用域都可以自主地决定rollback-only状态，当这样的逻辑事务作用域被外部的一个逻辑事务作用域所包含的时候， 他们在逻辑上是独立的。当然了，对于正常的 PROPAGATION\_REQUIRED设置来说，他会被映射到相同的物理事务上。 所以一个标记有rollback-only的内部逻辑事务作用域的确会影响到外部的逻辑事务作用域（就像你所预料的那样）。

然而，当内部的事务作用域标记为rollback-only，同时外部的事务作用域并没有决定要回滚， 这样的回滚是意料不到的（静悄悄地由内部事务作用域触发的）： 一个对应的UnexpectedRollbackException 异常会在这个时候被抛出。这是 可以预料到的行为， 只有这样，这个事务的调用者才不会被误导，在事务没有提交的情况下误以为事务已经提交。所以如果内部的事务（外部的调用者并不知情）标记该事务为 rollback-only，而外部的调用者却依旧在不知情的情况下提交后，它需要收到一个 UnexpectedRollbackException 异常来清楚的了解事务并没有提交而是发生了回滚。

**RequiresNew**

PROPAGATION\_REQUIRES\_NEW，与之前相反，为每一个相关的事务作用域使用了一个完全 独立的事务。在这种情况下，物理事务也将是不同的，因此外部事务可以不受内部事务回滚状态的影响独立提交或者回滚。

**Nested**

PROPAGATION\_NESTED 是一个完全不同的设置。它使用了一个单独的物理事务， 这个事务拥有多个可以回滚的保存点。这样部分回滚允许内部事务在它的作用域内触发一个回滚， 并且外部事务能够不受影响的继续。 这通常是对应于JDBC的保存点，所以只会在 JDBC 资源事务管理上起效。

**第二个版本**

以下是事务的7种传播级别：

1） PROPAGATION\_REQUIRED ，默认的spring事务传播级别，使用该级别的特点是，如果上下文中已经存在事务，那么就加入到事务中执行，如果当前上下文中不存在事务，则新建事务执行。所以这个级别通常能满足处理大多数的业务场景。

2）PROPAGATION\_SUPPORTS ，从字面意思就知道，supports，支持，该传播级别的特点是，如果上下文存在事务，则支持事务加入事务，如果没有事务，则使用非事务的方式执行。所以说，并非所有的包在transactionTemplate.execute中的代码都会有事务支持。这个通常是用来处理那些并非原子性的非核心业务逻辑操作。应用场景较少。

3）PROPAGATION\_MANDATORY ， 该级别的事务要求上下文中必须要存在事务，否则就会抛出异常！配置该方式的传播级别是有效的控制上下文调用代码遗漏添加事务控制的保证手段。比如一段代码不能单独被调用执行，但是一旦被调用，就必须有事务包含的情况，就可以使用这个传播级别。

4）PROPAGATION\_REQUIRES\_NEW ，从字面即可知道，new，每次都要一个新事务，该传播级别的特点是，每次都会新建一个事务，并且同时将上下文中的事务挂起，执行当前新建事务完成以后，上下文事务恢复再执行。

这是一个很有用的传播级别，举一个应用场景：现在有一个发送100个红包的操作，在发送之前，要做一些系统的初始化、验证、数据记录操作，然后发送100封红包，然后再记录发送日志，发送日志要求100%的准确，如果日志不准确，那么整个父事务逻辑需要回滚。

怎么处理整个业务需求呢？就是通过这个PROPAGATION\_REQUIRES\_NEW 级别的事务传播控制就可以完成。发送红包的子事务不会直接影响到父事务的提交和回滚。

5）PROPAGATION\_NOT\_SUPPORTED ，这个也可以从字面得知，not supported ，不支持，当前级别的特点就是上下文中存在事务，则挂起事务，执行当前逻辑，结束后恢复上下文的事务。

这个级别有什么好处？可以帮助你将事务极可能的缩小。我们知道一个事务越大，它存在的风险也就越多。所以在处理事务的过程中，要保证尽可能的缩小范围。比如一段代码，是每次逻辑操作都必须调用的，比如循环1000次的某个非核心业务逻辑操作。这样的代码如果包在事务中，势必造成事务太大，导致出现一些难以考虑周全的异常情况。所以这个事务这个级别的传播级别就派上用场了。用当前级别的事务模板抱起来就可以了。

6）PROPAGATION\_NEVER ，该事务更严格，上面一个事务传播级别只是不支持而已，有事务就挂起，而PROPAGATION\_NEVER传播级别要求上下文中不能存在事务，一旦有事务，就抛出runtime异常，强制停止执行！这个级别上辈子跟事务有仇。

7）PROPAGATION\_NESTED ，字面也可知道，nested，嵌套级别事务。该传播级别特征是，如果上下文中存在事务，则嵌套事务执行，如果不存在事务，则新建事务。

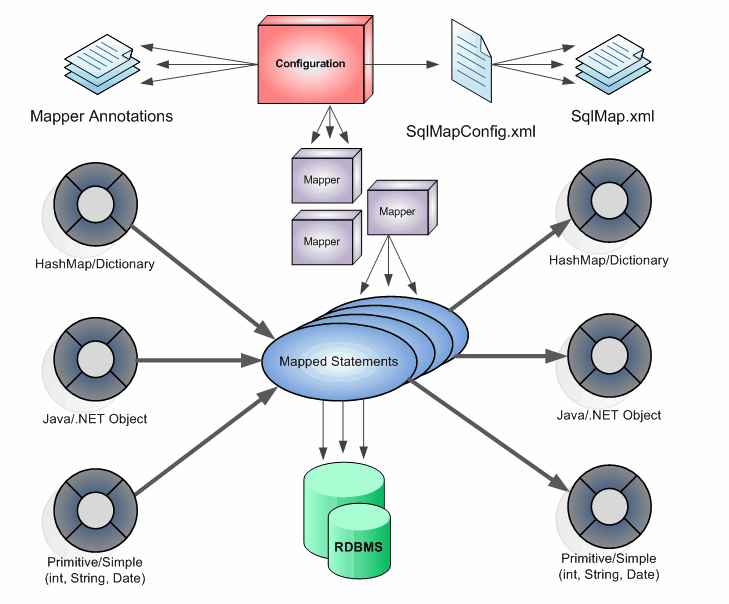
# 7 MyBatis

基于Mybatis3.1.1版本

## 7.1 MyBatis简介

MyBatis是支持普通SQL查询，存储过程和高级映射的优秀持久层框架。MyBatis消除几乎所有的JDBC代码和参数的手工配置以及结果集的检索。MyBatis使用简单的XML或注解用于配置和原始映射，将接口和JAVA的POJO映射成数据库中的记录。

每一个MyBatis的应用程序都以SqlSessionFactory对象的实例为核心。SqlSessionFactory对象的实例可通过SqlSessionFactoryBuilder对象来获得。SqlSessionFactoryBuilder对象可通过XML配置或Configuration类创建SqlSessionFactory实例。



## 7.2作用域和生命周期

**SqlSessionFactoryBuilder**

一旦成功创建SqlSessionFactory后，SqlSessionFactoryBuilder对象就不需要存在了。因此SqlSessionFactoryBuilder实例最佳范围就是本地方法变量。

**SqlSessionFactory**

一旦被创建，SqlSessionFactory在应用程序执行期间都存在。使用SqlSessionFactory的最佳实践就是在一个应用程序运行期间不要多次创建SqlSessionFactory。多次创建是非常糟糕。SqlSessionFactory的作用域是应用级的(Application Context)。可使用单例模式，使用Spring IoC依赖注入来管理单例SqlSessionFactory的生命周期。

**SqlSession**

每个线程都有自己的SqlSession实例，SqlSession实例不能被共享且不是线程安全的。最佳的作用域是请求范围(Request)或方法(Method)。基于HTTP 请求，打开一个SqlSession，然后返回HTTP 响应，就可以关闭它了。关闭SqlSession非常重要，确保使用finally块来关闭。如下示例：

SqlSession session = sqlSessionFactory.openSession();

try {

// do work

} finally {

session.close();

}

注意：绝对不能将SqlSession实例的引用放在类的静态字段或实例字段中。也不能将SqlSession实例的引用放在任何类型的管理范围中(比如Servlet的HttpSession)。

**Mapper实例**

Mappers是创建来绑定映射语句的接口，Mapper实例通过SqlSession获得。Mapper的作用域与创建它的SqlSession是一样。但是Mapper实例最佳的作用域是method，也就是应该在方法内被调用，使用完被销毁。Mapper实例不用显式地被关闭

SqlSession session = sqlSessionFactory.openSession();

try {

BlogMapper =session.getMapper(BlogMapper.class);

} finally {

session.close();

}

## 7.3 XML 映射配置文件

MyBatis的XML配置文件包含了影响MyBatis行为甚深的设置和属性信息。XML文档的高层级结构如下：

configuration配置

 properties属性

 settings设置

 typeAliases 类型命名

 typeHandlers 类型处理器

 objectFactory对象工厂

 plugins插件

 environments环境

 environment环境变量

 transactionManager 事务管理器

 dataSource数据源

 映射器

### 7.3.1 properties

这些是外部化的，可替代的属性，这些属性也可以配置在典型的Java属性配置文件中，或者通过properties元素的子元素来传递。例如：

<properties resource="org/mybatis/example/config.properties">

<property name="username" value="dev\_user"/>

<property name="password" value="F2Fa3!33TYyg"/>

</properties>

其中的属性就可以在整个配置文件中使用，使用可替换的属性来实现动态配置。比如：

<dataSource type="POOLED">

<property name="driver" value="${driver}"/>

<property name="url" value="${url}"/>

<property name="username" value="${username}"/>

<property name="password" value="${password}"/>

</dataSource>

如果在这些地方，属性多于一个的话，MyBatis按照如下的顺序加载它们：

* 在properties元素体内指定的属性首先被读取。
* 从类路径下资源或properties元素的url属性中加载的属性第二被读取，它会

覆盖已经存在的完全一样的属性。

* 作为方法参数传递的属性最后被读取，它也会覆盖任一已经存在的完全一样的

属性，这些属性可能是从properties元素体内和资源/url 属性中加载的。

因此，最高优先级的属性是那些作为方法参数的，然后是资源/url 属性，最后是properties元素中指定的属性。

### 7.3.2 Settings

这些是极其重要的调整，它们会修改MyBatis在运行时的行为方式。下面这个表格描述了设置信息，它们的含义和默认值。



### 7.3.3 typeAliases

类型别名是为Java类型命名一个短的名字。它只和XML配置有关，只用来减少类完全限定名的多余部分。例如：

<typeAliases>

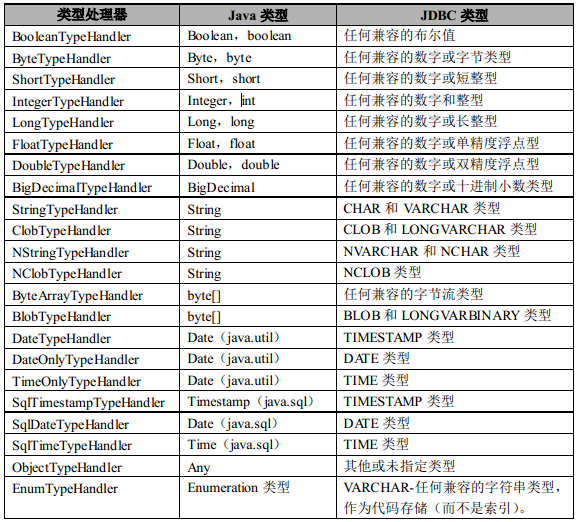
<typeAlias alias="Author" type="domain.blog.Author"/>

</typeAliases>

使用这个配置，“Author”可以任意用来替代“domain.blog.Autho”所使用的地方。

### 7.3.4 typeHandlers

无论是MyBatis在预处理语句中设置一个参数，还是从结果集中取出一个值时，类型处理器被用来将获取的值以合适的方式转换成Java类型。下面这个表格描述了默认的类型处理器。



### 7.3.5 environments

MyBatis可以配置多种环境。这会帮助你将SQL映射应用于多种数据库之中。例如，

你也许为开发要设置不同的配置，测试和生产环境。或者你可能有多种生产级数据库却共享相同的模式，所以你会想对不同数据库使用相同的SQL映射。这种用例是很多的。

**一个很重要的问题要记得：你可以配置多种环境，但你只能为每个SqlSessionFactory实例选择一个。**

所以，如果你想连接两个数据库，你需要创建两个SqlSessionFactory实例，每个数据库对应一个。而如果是三个数据库，你就需要三个实例，以此类推。记忆起来很简单：

 **每个数据库对应一个SqlSessionFactory**

为了明确创建哪种环境，你可以将它作为可选的参数传递给SqlSessionFactoryBuilder。可以接受环境配置的两个方法签名是：

SqlSessionFactory factory = sqlSessionFactoryBuilder.build(reader, environment);

SqlSessionFactory factory = sqlSessionFactoryBuilder.build(reader,

environment,properties);

如果环境被忽略，那么默认环境将会被加载，如下进行：

SqlSessionFactory factory = sqlSessionFactoryBuilder.build(reader);

SqlSessionFactory factory = sqlSessionFactoryBuilder.build(reader,properties);

环境元素定义了如何配置环境。

<environments default="development">

<environment id="development">

<transactionManager type="JDBC">

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

</transactionManager>

<dataSource type="POOLED">

<property name="driver" value="${driver}"/>

<property name="url" value="${url}"/>

<property name="username" value="${username}"/>

<property name="password" value="${password}"/>

</dataSource>

</environment>

</environments>

注意这里的键：

* 默认的环境ID（比如：default=”development”）。
* 每个environment元素定义的环境ID（比如：id=”development”）。
* 事务管理器的配置（比如：type=”JDBC”）。
* 数据源的配置（比如：type=”POOLED”）。

### 7.3.6 transactionManager

在MyBatis中有两种事务管理器类型（也就是type=”[JDBC|MANAGED]”）：

* JDBC – 这个配置直接简单使用了JDBC的提交和回滚设置。它依赖于从数据源得到的连接来管理事务范围。
* MANAGED – 这个配置几乎没做什么。它从来不提交或回滚一个连接。而它会让容器来管理事务的整个生命周期（比如Spring或JEE应用服务器的上下文）。默认情况下它会关闭连接。然而一些容器并不希望这样，因此如果你需要从连接中停止它，将closeConnection属性设置为false。例如：

<transactionManager type="MANAGED">

<property name="closeConnection" value="false"/>

</transactionManager>

这两种事务管理器都不需要任何属性。然而它们都是类型别名，要替换使用它们，你需要放置将你自己的类的完全限定名或类型别名，它们引用了你对TransacFactory接口的实现类。

### 7.3.7 DataSource

DataSource元素使用基本的JDBC数据源接口来配置JDBC连接对象的资源。

* 许多MyBatis的应用程序将会按示例中的例子来配置数据源。然而它并不是必须的。要知道为了方便使用延迟加载，数据源才是必须的。

有三种内建的数据源类型（也就是type=”???”）：

**UNPOOLED**

这个数据源的实现是每次被请求时简单打开和关闭连接。它有一点慢，这是对简单应用程序的一个很好的选择，因为它不需要及时的可用连接。不同的数据库对这个的表现也是不一样的，所以对某些数据库来说配置数据源并不重要，这个配置也是闲置的。

UNPOOLED类型的数据源仅仅用来配置以下4种属性：

* driver – 这是JDBC驱动的Java类的完全限定名（如果你的驱动包含，它也不是数据源类）。
* url – 这是数据库的JDBC URL地址。
* username – 登录数据库的用户名。
* password – 登录数据库的密码。
* defaultTransactionIsolationLevel – 默认的连接事务隔离级别。

作为可选项，你可以传递数据库驱动的属性。要这样做，属性的前缀是以“driver.”开

头的，例如：

* driver.encoding=UTF8

这样就会传递以值“UTF8”来传递属性“encoding ”，它是通过DriverManager.getConnection(url,driverProperties)方法传递给数据库驱动。

**POOLED**

这是JDBC连接对象的数据源连接池的实现，用来避免创建新的连接实例时必要的初始连接和认证时间。这是一种当前Web应用程序用来快速响应请求很流行的方法。

以下属性可以用来配置POOLED数据源：

* poolMaximumActiveConnections – 在任意时间存在的活动（也就是正在使用）连接的数量。默认值：10
* poolMaximumIdleConnections – 任意时间存在的空闲连接数。
* poolMaximumCheck outTime – 在被强制返回之前，池中连接被检查的时间。默认值：20000毫秒（也就是20秒）
* poolTimeToWait – 这是给连接池一个打印日志状态机会的低层次设置，还有重新尝试获得连接，这些情况下往往需要很长时间（为了避免连接池没有配置时静默失败）。默认值：20000毫秒（也就是20秒）
* poolPingQuery – 发送到数据的侦测查询，用来验证连接是否正常工作，并且准备接受请求。默认是“NO PING QUERY SET”，这会引起许多数据库驱动连接由一个错误信息而导致失败。
* poolPingEnabled – 这是开启或禁用侦测查询。如果开启，你必须用一个合法的SQL语句（最好是很快速的）设置poolPingQuery属性。默认值：false。
* poolPingConnectionsNotUsedFor – 这是用来配置poolPingQuery多次时间被用一次。这可以被设置匹配标准的数据库连接超时时间，来避免不必要的侦测。默认值：0（也就是所有连接每一时刻都被侦测-但仅仅当poolPingEnabled为true 时适用）。

**JNDI**

这个数据源的实现是为了使用如Spring或应用服务器这类的容器，容器可以集中或在外部配置数据源，然后放置一个JNDI上下文的引用。这个数据源配置只需要两个属性：

* initial\_context – 这个属性 用来从 初始 上下 文中寻 找环 境（ 也就是

initialContext.lookup（initial——context））。这是个可选属性，如果被忽略，那么

data\_source属性将会直接以initialContext 为背景再次寻找。

* data\_source – 这是引用数据源实例位置的上下文的路径。它会以由initial\_context查询返回的环境为背景来查找，如果initial\_context 没有返回结果时，直接以初始上下文为环境来查找。

和其他数据源配置相似，它也可以通过名为“env.”的前缀直接向初始上下文发送属性。

* env.encoding=UTF8

### 7.3.8 mappers

可以使用相对于类路径的资源引用，或者字符表示，或url引用的完全限定名（包括file:///URLs）。例如：

// Using classpath relative resources

<mappers>

<mapper resource="org/mybatis/builder/AuthorMapper.xml"/>

<mapper resource="org/mybatis/builder/BlogMapper.xml"/>

<mapper resource="org/mybatis/builder/PostMapper.xml"/>

</mappers>

// Using url fully qualified paths

<mappers>

<mapper url="file:///var/sqlmaps/AuthorMapper.xml"/>

<mapper url="file:///var/sqlmaps/BlogMapper.xml"/>

<mapper url="file:///var/sqlmaps/PostMapper.xml"/>

</mappers>

## 7.4 SQL映射的XML文件

MyBatis的构建就是聚焦于SQL的，使其远离于普通的方式。SQL映射文件有如下的几个顶级元素（按照它们应该被定义的顺序）：

 cache - 配置给定命名空间的缓存。

 cache-ref – 从其他命名空间引用缓存配置。

 resultMap – 最复杂，也是最有力量的元素，用来描述如何从数据库结果集中来加

载你的对象。

 parameterMap – 已经被废弃了！老式风格的参数映射。内联参数是首选，这个元

素可能在将来被移除。这里不会记录。

 sql – 可以重用的SQL块，也可以被其他语句引用。

 insert – 映射插入语句

 update – 映射更新语句

 delete – 映射删除语句

 select – 映射查询语句

### 7.4.1 SELECT

查询语句是使用MyBatis时最常用的元素之一。对简单类别的查询元素是非常简单的。比如：

**<select id=”selectPerson” parameterType=”int” resultType=”hashmap”>**

SELECT \* FROM PERSON WHERE ID = #{id}

**</select>**

这条语句叫做selectPerson，以int型（或者Integer型）作为参数，并返回一个以数据库列名作为键值的HashMap。

注意参数注释： #{id}

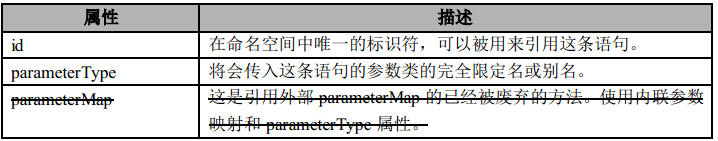
这就告诉MyBatis创建一个预处理语句参数。使用JDBC，这样的一个参数在SQL中会由一个“?”来标识，并被传递到一个新的预处理语句中，就像这样：

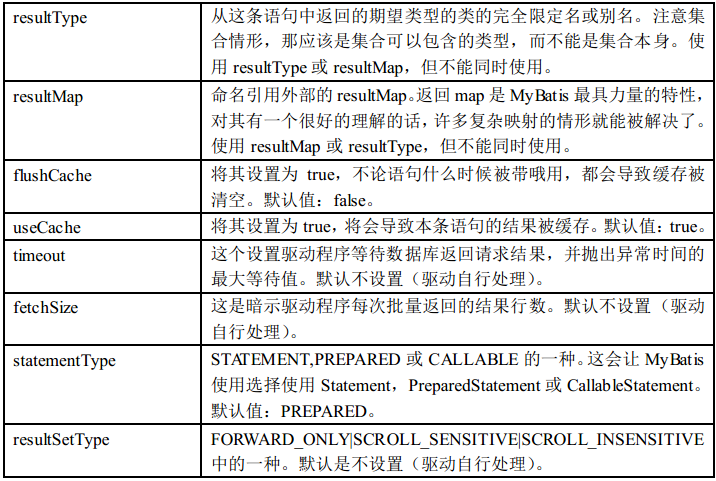
String selectPerson = “SELECT \* FROM PERSON WHERE ID=?”;

PreparedStatement ps = conn.prepareStatement(selectPerson);

ps.setInt(1,id);

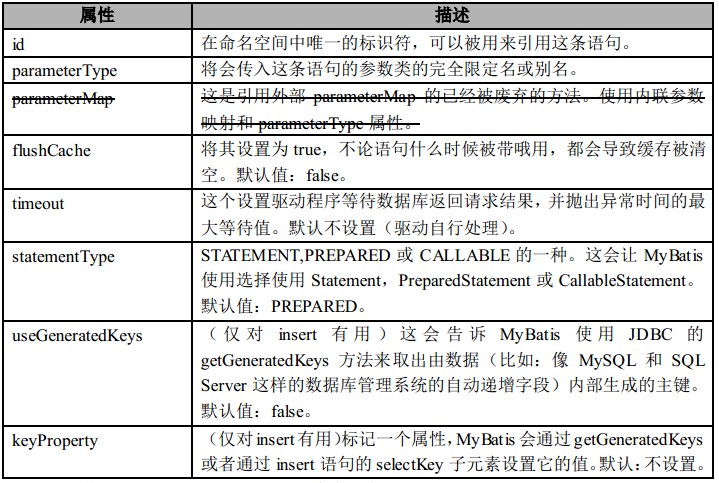
select元素有很多属性允许你配置，来决定每条语句的作用细节。





### 7.4.2 insert，update，delete

数据变更语句insert，update和delete在它们的实现中非常相似：



下面就是insert，update和delete语句的示例：

**<insert i d="insertAuthor" parame terType ="domain.bl og.Author">**

insert into Author (id,username,password,email,bio)

values (#{id},#{username},#{password},#{email},#{bio})

**</insert>**

**<update i d="updateAuthor " par ameter Type="domain.bl og.Author">**

update Author set username = #{username}, password = #{password},

email = #{email}, bio = #{bio} where id = #{id}

**</update>**

**<delete id="deleteAuthor” parameterType="int">**

delete from Author where id = #{id}

**</ delete>**

### 7.4.3 sql

这个元素可以被用来定义可重用的SQL代码段，可以包含在其他语句中。比如：

<sql id=”userColumns”> id,username,password </sql>

这个SQL片段可以被包含在其他语句中，例如：

<select id=”selectUsers” parameterType=”int” resultType=”hashmap”>

select <include refid=”userColumns”/> from some\_table

where id = #{id}

</select>

### 7.4.3 resultMap

resultMap 元素是MyBatis中最重要最强大的元素。resultMap元素有很多子元素和结构。下面是resultMap 元素的概念视图

resultMap

 constructor – 类在实例化时，用来注入结果到构造方法中

 idArg – ID参数；标记结果作为ID 可以帮助提高整体效能

 arg – 注入到构造方法的一个普通结果

 id – 一个ID结果；标记结果作为ID可以帮助提高整体效能

 result – 注入到字段或JavaBean属性的普通结果

 association – 一个复杂的类型关联；许多结果将包成这种类型

 嵌入结果映射 – 结果映射自身的关联，或者参考一个

 collection – 复杂类型的集

 嵌入结果映射 – 结果映射自身的集，或者参考一个

 discriminator – 使用结果值来决定使用哪个结果映射

 case – 基于某些值的结果映射

 嵌入结果映射 – 这种情形结果也映射它本身，因此可以包含很多相

同的元素，或者它可以参照一个外部的结果映射。

如下为resultMap示例：

public class User {

private int id;

private String username;

private String hashedPassword;

//省略get/set方法

}

<resultMap id="userResultMap" type="User">

<id property="id" column="user\_id" />

<result property="username" column="username"/>

<result property="password" column="password"/>

</resultMap>

引用它的语句使用resultMap 属性就行了（注意我们去掉了resultType属性）。比如：

<select id=”selectUsers” parameterType=”int”resultMap=”userResultMap”>

select user\_id, user\_name, hashed\_password

from some\_table

where id = #{id}

</select>

## 7.5动态SQL

MyBatis的一个强大的特性之一通常是它的动态SQL能力。MyBatis采用功能强大的基于OGNL的表达式来消除其他元素。

* if
* choose(when,otherwise)
* trim(where,set)
* foreach

**IF**

在动态SQL中所做的最通用的事情是包含部分where字句的条件。

<select id=”findActiveBlogWithTitleLike”

parameterType=”Blog” resultType=”Blog”>

SELECT \* FROM BLOG WHERE state = 'ACTIVE'

<if test=”title != null”>

AND title like #{title}

</if>

<if test=”author != null and author.name != null”>

AND title like #{author.name}

</if>

</select>

**choose, when, otherwise**

有时我们不想应用所有的条件，相反我们想选择很多情况下的一种。和Java中的switch语句相似，MyBatis提供choose元素。

<select id=”findActiveBlogLike”

parameterType=”Blog” resultType=”Blog”>

SELECT \* FROM BLOG WHERE state = ?ACTIVE?

<choose>

<when test=”title != null”>

AND title like #{title}

</when>

<when test=”author != null and author.name != null”>

AND title like #{author.name}

</when>

<otherwise>

AND featured = 1

</otherwise>

/choose>

</select>

# 8 测试

8.1 单元测试

8.2 Web层测试

8.3 整合测试

8.4 Junit

# 9 参考资料

多图详解Spring框架的设计理念与设计模式http://developer.51cto.com/art/201006/205212.htm