# Spring概述

Spring是一个开源框架，它由Rod Johnson创建。它是为了解决企业应用开发的复杂性而创建的。Spring使用基本的JavaBean来完成以前只可能由EJB完成的事情。然而，Spring的用途不仅限于服务器端的开发。从简单性、可测试性和松耦合的角度而言，任何Java应用都可以从Spring中受益。

Spring是一个轻量级的控制反转(IoC)和面向切面(AOP)的容器框架。

**轻量**——从大小与开销两方面而言Spring都是轻量的。完整的Spring框架可以在一个大小只有1MB多的JAR文件里发布。并且Spring所需的处理开销也是微不足道的。此外，Spring是非侵入式的：典型地，Spring应用中的对象不依赖于Spring的特定类。

**控制反转**——Spring通过一种称作控制反转（IoC）的技术促进了松耦合。当应用了IoC，一个对象依赖的其它对象会通过被动的方式传递进来，而不是这个对象自己创建或者查找依赖对象。你可以认为IoC与JNDI相反——不是对象从容器中查找依赖，而是容器在对象初始化时不等对象请求就主动将依赖传递给它。

**面向切面**——Spring提供了面向切面编程的丰富支持，允许通过分离应用的业务逻辑与系统级服务（例如审计（auditing）和事务（transaction）管理）进行内聚性的开发。应用对象只实现它们应该做的——完成业务逻辑——仅此而已。它们并不负责（甚至是意识）其它的系统级关注点，例如日志或事务支持。

**容器**——Spring包含并管理应用对象的配置和生命周期，在这个意义上它是一种容器，你可以配置你的每个bean如何被创建——基于一个可配置原型（prototype），你的bean可以创建一个单独的实例或者每次需要时都生成一个新的实例——以及它们是如何相互关联的。然而，Spring不应该被混同于传统的重量级的EJB容器，它们经常是庞大与笨重的，难以使用。

**框架**——Spring可以将简单的组件配置、组合成为复杂的应用。在Spring中，应用对象被声明式地组合，典型地是在一个XML文件里。Spring也提供了很多基础功能（事务管理、持久化框架集成等等），将应用逻辑的开发留给了你。

所有Spring的这些特征使你能够编写更干净、更可管理、并且更易于测试的代码。它们也为Spring中的各种模块提供了基础支持。

**Spring的初衷：**

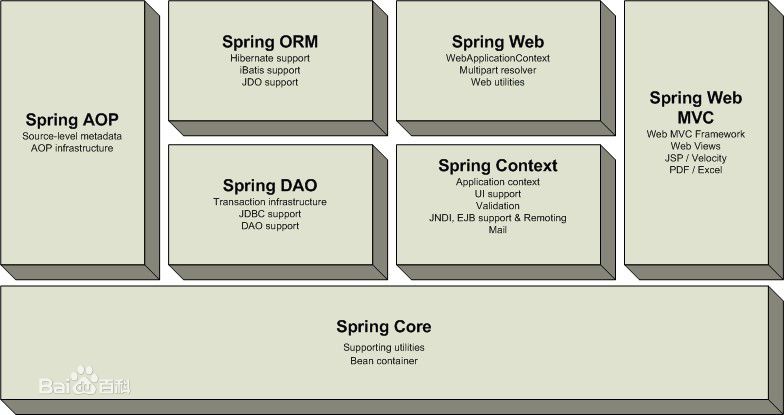
1. J2EE开始应该更加简单
2. 使用接口而不是使用类，是更好的编程习惯。Spring将使用接口的复杂度几乎降低到了零。
3. 为JavaBean提供了一个更好的应用配置框架。
4. 更多地强调面向对象的设计，而不是现行的技术如J2EE。
5. 尽量减少不必要的异常捕捉。
6. 使应用程序更加容易测试。

**Spring的目标：**

1. 可以令人方便愉快的使用Spring。
2. 应用程序代码并不依赖于Spring APIs。
3. Spring不和现有的解决方案竞争，而是致力于将它们融合在一起。

**Spring的基本组成：**

Spring框架由七个定义明确的模块组成



如果作为一个整体，这些模块为你提供了开发企业应用所需的一切。但你不必将应用完全基于Spring框架。你可以自由地挑选适合你的应用的模块而忽略其余的模块。

就像你所看到的，所有的Spring模块都是在核心容器之上构建的。容器定义了Bean是如何创建、配置和管理的——更多的Spring细节。当你配置你的应用时，你会潜在 地使用这些类。但是作为一名开发者，你最可能对影响容器所提供的服务的其它模块感兴趣。这些模块将会为你提供用于构建应用服务的框架，例如AOP和持久性。

1. 核心容器

这是Spring框架最基础的部分，它提供了依赖注入（DependencyInjection）特征来实现容器对Bean的管理。这里最基本的概念是BeanFactory，它是任何Spring 应用的核心。BeanFactory是工厂模式的一个实现，它使用IoC将应用配置和依赖说明从实际的应用代码中分离出来。

1. 应用上下文（Context）模块

核心模块的BeanFactory使Spring成为一个容器，而上下文模块使它成为一个框架。这个模块扩展了BeanFactory的概念，增加了对国际化（I18N）消息、事件传 播以及验证的支持。

另外，这个模块提供了许多企业服务，例如电子邮件、JNDI访问、EJB集成、远程以及时序调度（scheduling）服务。也包括了对模版框架例如Velocity和FreeMarker 集成的支持。

1. Spring的AOP模块

Spring在它的AOP模块中提供了对面向切面编程的丰富支持。这个模块是在Spring应用中实现切面编程的基础。为了确保Spring与其它AOP框架的互用性，Spring 的AOP支持基于AOP联盟定义的API。AOP联盟是一个开源项目，它的目标是通过定义一组共同的接口和组件来促进AOP的使用以及不同的AOP实现之间的互用性。 通过访问他们的站点，你可以找到关于AOP联盟的更多内容。

Spring的AOP模块也将元数据编程引入了Spring。使用Spring的元数据支持，你可以为你的源代码增加注释，指示Spring在何处以及如何应用切面函数。

1. JDBC抽象和DAO模块

使用JDBC经常导致大量的重复代码，取得连接、创建语句、处理结果集，然后关闭连接。Spring的JDBC和DAO模块抽取了这些重复代码，因此你可以保持你的数 据库访问代码干净简洁，并且可以防止因关闭数据库资源失败而引起的问题。

这个模块还在几种数据库服务器给出的错误消息之上建立了一个有意义的异常层。使你不用再试图破译神秘的私有的SQL错误消息！

另外，这个模块还使用了Spring的AOP模块为Spring应用中的对象提供了事务管理服务。

1. 对象/关系映射集成模块

对那些更喜欢使用对象/关系映射工具而不是直接使用JDBC的人，Spring提供了ORM模块。Spring并不试图实现它自己的ORM解决方案，而是为几种流行的ORM 框架提供了集成方案，包括Hibernate、JDO和iBATIS SQL映射。Spring的事务管理支持这些ORM框架中的每一个也包括JDBC。

1. Spring的Web模块

Web上下文模块建立于应用上下文模块之上，提供了一个适合于Web应用的上下文。另外，这个模块还提供了一些面向服务支持。例如：实现文件上传的multipart 请求，它也提供了Spring和其它Web框架的集成，比如Struts、WebWork。

1. Spring的MVC框架

Spring为构建Web应用提供了一个功能全面的MVC框架。虽然Spring可以很容易地与其它MVC框架集成，例如Struts，但Spring的MVC框架使用IoC对控制 逻辑和业务对象提供了完全的分离。

它也允许你声明性地将请求参数绑定到你的业务对象中，此外，Spring的MVC框架还可以利用Spring的任何其它服务，例如国际化信息与验证。

1. Spring框架Web页面乱码问题

在做java Web 项目时，乱码问题时常都会出现，解决方法也不尽相同，有简单也有复杂的；如果加入了Spring框架之后就不一样了，可以采用Spring框架自带的过 滤器CharacterEncodingFilter，这样可以大大减轻了我们的工作量，即简单方便又容易理解

# Spring IoC容器

## IoC基础

### IoC的基本概念

IoC是随着近年来轻量级容器（Lightweight Container)的兴起而逐渐被很多人提起的一个名词，它的全称为Inversion of Control，中文通常翻译为“控制反转”， 它还有一个别名叫做依赖注入（DependencyInjection）。它不是一种编程语言，而是一种编程方式。也是对面向对象的一种补充完善。它实现了解耦各业务对象 间依赖关系的对象绑定方式。

为什么需要IoC，IoC的具体意义是什么？

在以往的Java开发中，如果在一个类中的属性依赖于另一个类，那么我们会在该类中定义一个依赖类的引用。当我们需要使用依赖类中的方法或数据的时候，必须 显示 的通过new关键字实例化依赖类的对象，才能去使用依赖类所提供的服务。回头想想，这是否真的必要？我们最终所要做的，其实就是直接调用依赖对象所提 供的某项服务而已。只要用到这个依赖对象的时候，它能够准备就绪，我们完全可以不管这个对象是如何创建的。

IoC就是为了帮助我们避免之前的“大费周折”，而提供了更加轻松简洁的方式。它的反转，就是反转在让你从原来的通过new关键字实例化对象后才能使用对象的 服务，转变为直接使用对象所提供的服务，无需自己实例化对象，实例化对象的工作直接交由IoC框架来完成。

通常情况下，被注入对象会直接依赖于被依赖对象。但是在IoC的场景中，二者之间通过IoC Service Provider来打交道，所有的被注入对象和依赖对象现在由 IoC Service Provider统一管理。被注入对象需要什么，直接通过IoC Service Provider来获取，后者就会把相应的被依赖对象注入到被注入对象中，从而达到 IoC Service Provide为被注入对象服务的目的。IoC Service Provider在这里就是通常的IoC容器所充当的角色。从被注入对象的角度看，与之前直接寻求依赖 对象相比，依赖对象的取得方式发生了反转，控制也从被注入对象转到了IoC Service Provider里。

### 注入方式

作为被注入对象，要想让IoC Service Provider为其提供服务，并将所需要的被依赖对象送过来，也需要通过某种方式通知IoC Service Provider。其中有三种 不同 的依赖注入的方式，即构造方法注入、setter方法注入以及接口注入。

#### 构造方法注入

构造方法注入，就是被注入对象可以通过在其构造方法中声明依赖对象的参数列表，让外部（IoC容器）知道它需要哪些依赖对象。IoC Service Provider会 检查 被注入对象的构造方法，取得它所需要的依赖对象列表，进而为其注入相应的对象。同一个对象不可能被构造两次的，因此被注入对象的构造乃至其整个 生命走起应该是由IoC Service Provider来管理的。

构造方法注入的方式比较直观，对象被构造完成后，即进入就绪状态，可以马上使用。

#### setter方法注入

对于JavaBean对象来说，通常会通过setXxx()和getXxx()方法来访问对应属性。这些setXxx()方法统称为setter方法，getXxx()方法则称为getter方 法。通过setter方法，可以更改相应的对象属性，通过getter方法，可以获取相应的属性值。所以，当前对象只要为其依赖对象对应的属性添加setter方法， 就可以通过setter方法将相应的依赖对象设置到被注入对象中。

setter方法注入虽不像构造方法注入那样，让对象构造完成后即可使用，但相对来说更宽松一些，可以在对象构造完成后再注入，随意性较强。

#### 接口注入

相对于前两种注入方式来说，接口注入没那么简单明了。被注入对象如果想要IoC Service Provider为其注入依赖对象，就必须实现某个接口。这接口提供一 个方 法，用来为其注入依赖对象。IoC Service Provider最终通过这些接口来了解应该为被注入对象注入什么依赖对象。

#### 三种注入方式的比较

1. 接口注入：从注入方式的使用上来说，接口注入是现在不甚提倡的一种方式，基本处于“退役状态”。因为它强制被注入对象实现不必要的接口，带有侵入性。而构造方法注入和setter方法注入则不需要如此。
2. 构造方法注入：这种注入方式的优点就是，对象在构造完成之后，即已进入就绪状态，可以马上使用。缺点就是，当依赖对象比较多的时候，构造方法的参数列表会比较长。而通过反射构造对象的时候，对相同类型的参数的处理会比较困难，维护和使用上也比较麻烦。而且在Java中，构造方法无法被继承，无法设置默认值。对于非必须的依赖处理，可能需要引入多个构造方法，而参数数量的变动可能造成维护上的不便。
3. setter方法注入：因为方法可以命名，所以setter方法注入在描述性上要比构造方法注入好一些。另外，setter方法可以被继承，允许设置默认值，而且有良好的IDE支持。缺点当然就是对象无法在构造完成后马上进入就绪状态。

综上所述，构造方法注入和setter方法注入因为其侵入性较弱，且易于理解和使用，所以是现在使用最多的注入方式；而接口注入因为侵入性较强，近年来已 经不 流行了。

## IoC Service Provider

虽然业务对象可以通过IoC方式声明相应的依赖，但是最终仍然需要通过某种角色或者服务将这些相互依赖的对象绑定到一起，而IoC Service Provider就是对应IoC 场景中的一个角色。

IoC Service Provider在这里是是一个抽象出来的概念，它可以指代任何将IoC场景中的业务对象绑定到一起的实现方式。它可以是一段代码，也可以是一组相关的类， 甚至可以是比较通用的IoC框架或者IoC容器实现。Spring的IoC容器就是一个提供依赖注入服务的IoC Service Provider。

### IoC Service Provider的职责

IoC Service Provider的职责相对来说比较简单，主要有两个：业务对象的构建管理和业务对象间的依赖绑定。

1. **业务对象的构建管理**

在IoC场景中，业务对象无需关系所依赖的对象如何构建如何取得，但这部分工作始终需要有人来做。所以，IoC Service Provider需要将被依赖对象的构建逻辑从需要依赖的对象哪里剥离出来，以免这部分逻辑污染业务对象的实现。

1. **业务对象的依赖绑定**

对于IoC Service Provider来说，这个职责是最艰巨也是最重要的，这是它的最终使命所在。如果不能完成这个职责，那么也就不存在依赖注入了。IoC Service Provider通过结合之前构建和管理的所有业务对象，以及各个业务对象间可以识别的依赖关系，将这些对象所依赖的对象注入绑定，从而保证每个业务对象在使用的时候，可以处于就绪状态。

### IoC Service Provider如何管理对象间的依赖关系

前面说过，被注入对象可以通过多种方式通知IoC Service Provider为其注入相应依赖。但是收到通知的IoC Service Provider是否就一定能完全领会被注入对 象的意图，并及时有效地为其提供想要的依赖呢？

在当前流行的IoC Service Provider产品中，使用的注册对象关系信息的方式主要有以下几种

#### 直接编码方式

当前大部分的IoC容器都应该支持直接编码方式，比如：PicoContainer、Spring、Avalon等。在容器启动之前，我们就可以通过程序编码的方式将被注入 对象和依赖对象注册到容器中，并明确他们相互之间的依赖注入关系。

IoContainer container = ...;

container.register(FXNewsProvider.**class**, **new** FXNewsProvider());

container.register(IFXNewsListener.**class**, **new** DowJonesNewsListener());

...

FXNewsProvider newsProvider = (FXNewsProvider)container.get(FXNewsProvider.**class**);

newProvider.getAndPersistNews();

通过为相应的类指定对应的具体实例，可以告知IoC容器，当我们要这种类型的对象实例时，请将容器中注册的对应的那个具体实例返回给我们

如果是接口注入，那么伪代码看起来要多些。不过道理上市一样的，只不过除了注册相应对象，还要将“注入标志接口”与相应的依赖对象绑定一下，才能让 容器最终直到是一个什么样的对应关系。

IoContainer container = ...;

container.register(FXNewsProvider.**class**, **new** FXNewsProvider());

container.register(IFXNewsListener.**class**, **new** DowJonesNewsListener());

...

container.bind(IFXNewsListenerCallable.**class**, container.get(IFXNewsListener.**class**));

...

FXNewsProvider newsProvider = (FXNewsProvider)container.get(FXNewsProvider.**class**);

newProvider.getAndPersistNews();

通过bind方法将“被注入对象”（由IFXNewsListenerCallable接口添加标志）所依赖的对象，绑定为容器中注册过的IFXNewsListener类型的对象实例。 容器在返回FXNewsProvider对象实例之前，会根据这个绑定信息，将IFXNewsListener注册到容器中的对象实例注入到“被注入对象”——FXNewsProvider 中，并最终返回已经组装完毕的FXNewsProvider对象。

所以，通过程序编码让最终的IoC Service Provider（也就是各个IoC框架或者容器实现）得以知晓服务的“奥义”，应该是管理依赖绑定关系的最基本方式。

#### 配置文件方式

这是一种较为普遍的依赖注入关系管理方式。像普通文本文件、properties文件、XML文件等，都是可以成为管理依赖注入关系的载体。不过最为常见的还是 通过XML文件来管理对象注册和对象间依赖关系。

<bean id=*"newsProvider"* class=*"..FXNewsProvider"*>

<property name=*"newsListener"*>

<ref bean=*"djNewsListener"*/>

</property>

<property name=*"newPersistener"*>

<ref bean=*"djNewsPersister"*/>

</property>

</bean>

<bean id=*"djNewsListener"*

class=*"..impl.DowJonesNewsListener"*>

</bean>

<bean id=*"djNewsPersister"*

class=*"..impl.DowJonesNewsPersister"*>

</bean>

最后，我们可以通过“newProvider”这个名字，从容器中取得已经组装好的FXNewsProvider并直接使用

...

container.readConfigurationFiles(...);

FXNewsProvider newsProvider = (FXNewsProvider)container.getBean("newsProvider");

newsProvider.getAndPersistNews();

#### 元数据方法

这种方式的代表是现实Google Guice，这是Bob Lee在Java5的注解和Generic的基础上开发的一套IoC框架。我们可以直接在类中使用元数据信息来标 注各个对象之间的依赖关系，然后由Guice框架根据这些注解所提供的信息将这些对象组装后，交给需要依赖的对象使用。

**public** **class** FXNewsProvider{

**private** IFXNewsListener newsListener;

**private** IFXNewsPersister newPersistener;

@Inject

**public** FXNewsProvider(IFXNewsListener listener,IFXNewsPersister persister){

**this**.newsListener = listener;

**this**.newPersistener = persister;

}

...

}

通过@Inject，我们指明需要IoC Service Provider通过构造方法注入方式，为FXNewsProvider注入其所依赖的对象。至于余下的依赖相关信息，在Guice 中是由相应的Module来提供的

**public** **class** NewsBindingModule **extends** AbstractModule{

@Override

**protected** **void** configure() {

bind(IFXNewsListener.**class**).to(DowJonesNewsListener.**class**).in(Scopes.SINGLETON);

bind(IFXNewsPersister.**class**).to(DowJonesNewsPersister.**class**).in(Scopes.SINGLETON);

}

}

通过Module指定进一步的依赖注入相关信息之后，我们就可以直接从Guice那里取得最终已经注入完毕，并直接可用的对象了

Injector injector = Guice.createInjector(**new** NewsBindingModule());

FXNewsProvider newsProvider = injector.getInstance(FXNewsProvider.**class**);

newsProvider.getAndPersistNews();

当然，注解最终也要通过代码处理来确定最终的注入关系，从这点儿来说，注解方式可以算作编码方式的一种特殊情况。

## Spring IcO容器

Spring IoC容器是一个提供IoC支持的轻量级容器，除了基本的IoC支持，它作为轻量级容器还提供了IoC之外的支持。如在Spring IoC容器之上，Spring还提供 了相应的AOP框架支持、企业级服务集成等服务。



Spring提供了两种容器类型：BeanFactory和ApplicationContext。

1. BeanFactory：基础类型IoC容器，提供完整的IoC服务支持。如果没有特殊指定，默认采用延迟初始化策略（lazy-load）。只有当需要依赖的对象需要访问容器中的某个受管理对象的时候，才对该受管理对象进行初始化以及依赖注入操作。所以相对来说，容器启动初期速度较快，所需要的资源有限。对于资源有限且功能要求不是很严格的场景，BeanFactory是比较合适的IoC容器选择。
2. ApplicationContext：ApplicationContext在BeanFactory的基础上构建，是相对比较高级的容器实现，除了拥有BeanFactory的所有支持外还提供了其他高级特性，比如事件发布、国际化信息支持等。ApplicationContext所管理的对象，在该类型容器启动之后，默认全部初始化绑定完成。所以相对于BeanFactory来说，要求更多的系统资源，同时因为启动时就会完成所有初始化，容器启动事件较BeanFactory长一些。

### BeanFactory类型

对于应用程序开发来说，不管是否引入BeanFactory之类的轻量级容器，应用的设计和开发流程实际上没有太大变化。针对系统和业务逻辑，该如何设计和实现当前系统不受是否引入轻量级容器的影响。唯一不同的就是对象之间依赖关系的解决方式改变了。

在不使用BeanFactory的情况下，我们通常会直接在应用程序的入口类的main方法中自己实例化相应的对象并调用其中的方法。

FXNewsProvider newsProvider = **new** FXNewsProvider();

newsProvider.getAndPersistNews();

使用BeanFactory后，我们只需要把依赖注入的关系提交给BeanFactory，BeanFactory会将我们所需要的对象准备好，我们直接拿来使用就可以了。

<beans>

<bean id=*"djNewsProvider"* class=*"..FXNewsProvider"*>

<constructor-arg index=*"0"*>

<ref bean=*"djNewsListener"*/>

</constructor-arg>

<constructor-arg index=*"1"*>

<ref bean=*"djNewsPersister"*/>

</constructor-arg>

</bean>

...

</beans>

当我们在需要使用依赖对象所提供的服务，只需要调用其提供的方法就可以获取到所需要的依赖对象。

BeanFactory container = **new** XmlBeanFactory(**new** ClassPathResource("配置文件路径"));

FXNewsProvider newsProvider = (FXNewsProvider)container.getBean("djNewsProvider");

newsProvider.getAndPersistNews();

或者

ApplicationContext container = **new** ClassPathXmlApplicationContext("配置文件路径");

FXNewsProvider newsProvider = (FXNewsProvider)container.getBean("djNewsProvider");

newsProvider.getAndPersistNews();

亦或

ApplicationContext container = **new** FileSystemXmlApplicationContext("配置文件路径");

FXNewsProvider newsProvider = (FXNewsProvider)container.getBean("djNewsProvider");

newsProvider.getAndPersistNews();

注：其中配置文件路径为spring的配置文件。getBean()方法中的参数为Spring配置文件中<bean>标签的id属性值。

#### BeanFactory中对象注册与依赖绑定方式

BeanFactory作为一个IoC Service Provider，为了能够明确管理各个业务对象以及业务对象之间的依赖绑定关系，同样需要某种途径来记录和管理这些信息。 BeanFactory支持三种方式来管理这些信息。

1. **直接编码方式**

其实，把编码方式单独提出来称为一种方式并不十分恰当。因为不管什么方式，最终都需要编码才能“落实”所有信息并付诸使用。不过，通过这些代码，起码可以让我们更加清楚BeanFactory在底层是如何工作的。

通过编码方式使用BeanFactory实现相关类的注册及绑定

**public** **static** **void** main(String[] args){

DefaultListableBeanFactory beanRegistry = **new** DefaultListableBeanFactory();

BeanFactory container = (BeanFactory)bindViaCode(beanRegistry);

FXNewsProvider newsProvider = (FXNewsProvider)container.getBean("djNewsProvider");

newsProvider.getAndPersistNews();

}

**public** **static** BeanFactory bindViaCode(BeanDefinitionRegistry registry){

AbstractBeanDefinition newsProvider = **new** RootBeanDefinition(FXNewsProvider.**class**,**true**);

AbstractBeanDefinition newsListener = **new** RootBeanDefinition(DowJonesNewsListener.**class**,**true**);

AbstractBeanDefinition newsPersister = **new** RootBeanDefinition(DowJonesNewsPersister.**class**,**true**);

// 将bean定义注册到容器中

registry.registerBeanDefinition("djNewsProvider", newsProvider);

registry.registerBeanDefinition("djListener", newsListener);

registry.registerBeanDefinition("djPersister", newsPersister);

// 指定依赖关系

// 1. 可以通过构造方法注入方式

ConstructorArgumentValues argValues = **new** ConstructorArgumentValues();

argValues.addIndexedArgumentValue(0, newsListener);

argValues.addIndexedArgumentValue(1, newsPersister);

newsProvider.setConstructorArgumentValues(argValues);

// 2. 或者通过setter方法注入方式

MutablePropertyValues propertyValues = **new** MutablePropertyValues();

propertyValues.addPropertyValue(**new** ropertyValue("newsListener",newsListener));

propertyValues.addPropertyValue(**new** PropertyValue("newPersistener",newsPersister));

newsProvider.setPropertyValues(propertyValues);

// 绑定完成

**return** (BeanFactory)registry;

}

BeanFactory只是一个接口，我们最终需要一个该接口的实现来进行实际的Bean管理。DefaultListableBeanFactory就是这么一个比较通用的BeanFactory实现类。DefaultListableBeanFactory除了间接地实现了BeanFactory接口，还实现了BeanDefinitionRegistry接口，该接口才是BeanFactory的实现中担当Bean注册管理的角色。基本上，BeanFactory接口只定义如何访问容器内管理Bean的方法，这个BeanFactory的具体实现类负责具体Bean的注册以及管理工作。BeanDefinitionRegistry接口定义抽象了Bean的注册逻辑。通常情况下，具体的BeanFactory实现类会实现这个接口来管理Bean的注册。

1. **外部配置文件方式**

Spring的IoC容器支持两种配置文件格式：properties文件格式和XML文件格式。采用外部配置文件时，Spring IoC容器有一个统一的处理方式。通常情况下，需要根据不同的外部配置文件格式，给出相应的BeanDefinitionReader实现类，由BeanDefinitionReader的相应实现类负责将相应的配置文件内容读取并映射到BeanDefinition，然后将映射后的BeanDefinition注册到一个BeanDefinitionRegistry，之后，BeanDefinitionRegistry即完成Bean的注册和加载。

1. Properties配置格式的加载

Spring提供了org.springframework.beans.factory.support.PropertiesBeanDefinitionReader类用于Properties格式配置文件的加载。所以我们不用自己实现BeanDefinitionReader，只要根据该类的读取规则，提供相应的配置文件即可。

djNewsProvider.(class)=..FXNewsProvider

# ----------通过构造方法注入的时候-------------

djNewsProvider.$0(ref)=djListener

djNewsProvider.$1(ref)=djPersister

# ----------通过setter方法注入的时候---------

# djNewsProvider.newsListener(ref)=djListener

# djNewsProvider.newPersistener(ref)=djPersister

djListener.(class)=..impl.DowJonesNewsListener

djPersister.(class)=..impl.DowJonesNewsPersister

1. XML配置格式的加载

XML配置格式是Spring支持最完整，功能最强大的表达式。一方面这得益于XML良好的语意表达能力；另一方面就是Spring框架从开始就自始至终保持XML配置加载的统一性。Spring2.x之前，XML配置文件采用DTD实现文档的格式约束。2.x之后，引用了XSD的约束方式。

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

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

<beans>

<bean id=*"djNewsProvider"* class=*"..FXNewsProvider"*>

<constructor-arg index=*"0"*>

<ref bean=*"djNewsListener"*/>

</constructor-arg>

<constructor-arg index=*"1"*>

<ref bean=*"djNewsPersister"*/>

</constructor-arg>

</bean>

<bean id=*"djNewsListener"* class=*"..impl.DowJonesNewsListener"*></bean>

<bean id=*"djNewsPersister"* class=*"..impl.DowJonesNewsPersister"*></bean>

</beans>

Spring同样为XML格式的配置文件提供现成的实现类，即XmlBeanDefinitionReader。XmlBeanDefinitionReader负责读取XML配置文件并解析，之后将解析后的文件内容映射到BeanDefinition，并加载到相应的BeanDefinitionRegistry中（这里是DefaultListableBeanFactory）。

1. **注解方式**

在Spring2.5版本之后，Spring正式支持基于注解的依赖传入方式。注解是Java5之后才引入的，所以要使用注解的方式，则应用程序只能使用Spring2.5 以上版本以及Java5以上版本。如果要通过注解标注的方式实现注入所需要的依赖，可以使用@Autowired以及@Component注解。其中@Autowired注 解用于告知Spring容器需要为当前对象注入那些依赖对象。而@Component注解则是配合Spring2.5中新的classpath-scanning功能使用的。现在我们 只要向Spring的配置文件中增加一个“触发器”，使用@Autowired和@Component标注的类就能获得依赖对象的注入。

@Component

**public** **class** FXNewsProvider{

@Autowired

**private** IFXNewsListener newsListener;

@Autowired

**private** IFXNewsPersister newPersistener;

**public** FXNewsProvider(IFXNewsListener newsListner,IFXNewsPersister newsPersister){

**this**.newsListener = newsListner;

**this**.newPersistener = newsPersister;

}

...

}

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

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

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

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

xmlns: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.xsd*

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

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

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

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

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

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

<!-- 定于需要Spring扫描的组件 -->

<context:component-scan base-package=*"com.uc.\*"*/>

</beans>

<context:component-scan />会到指定的包下面扫描标注有@Component的类，如果找到，则将它们添加到容器中进行管理，并根据它们所标注的 @Autowired为这些类注入符合条件的依赖对象。

完成以上工作之后，我们就可以像通常那样加载配置并执行当前应用程序了。

**public** **static** **void** main(String[] args){

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("配置文件路径");

FXNewsProvider newsProvider = (FXNewsProvider)ctx.getBean("FXNewsProvider");

newsProvider.getAndPersistNews();

}

其中getBean中的参数为类的类名。

#### 依赖注入部分XML配置详解

在Spring2.0版之前，XML配置文件格式遵循Spring提供的DTD规定，也就是说所有Spring容器加载的XML配置文件的头部，都需要以下形式的DOCTYPE声明

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

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

<beans>

...

</beans>

从Spring2.0版本之后，Spring在继续保持向前兼容的前提下，既可以继续使用DTD方式的DOCTYPE进行配置文件格式的限定，又引入了基于XML Schema的文档声明。所以Spring2.0之后，同样可以使用基于XSD的文档声明

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

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

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

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

xmlns: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.xsd*

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

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

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

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

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

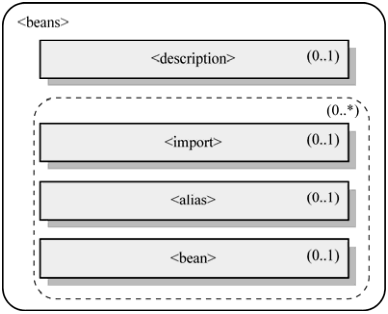
*http://www.springframework.org/schema/aop/spring-aop.xsd"*>

</beans>

不过，不管使用哪一种形式的文档声明，实际上限定的元素基本上是相同的。所有注册到容器的业务对象，在Spring中称之为Bean。所以每一个对象在XML中的映射也自然地对应一个叫做<bean>元素。

1. **<beans>**

<beans>是XML配置文件中最顶层的元素，它下面可以包含0或者1个<description>和多个<bean>以及<import>或者<alias>



**属性**

1. default-lazy-init：其值可以指定为true或者false，默认值为false。用来标志是否对所有的<bean>进行延迟初始化
2. default-autowire：可以取值为no、byName、byType、constructor以及autodetect。默认值为no，如果使用自动绑定的话，用来标志全体bean使用哪一种默认绑定方式。
3. default-dependency-check：可以取值none、objects、simple以及all，默认值为none，即不做依赖检查。
4. default-init-method：如果所管辖的<bean>按照某种规定，都有同样名称的初始化方法的话，可以在这里统一指定这个初始化方法名，而不用在每一个<bean>上都重复单独指定。
5. default-destroy-method：与default-init-method相对应，如果所有管辖的bean有按照某种规则使用了相同名称的对象销毁方法，可以通过这个属性统一指定。
6. **<description>、<import>和<alias>**
7. <description>

可以通过<description>在配置的文件中指定一些描述性的信息。通常情况下，该元素是省略的。

1. <import>

通常情况下，可以根据模块功能或者层次关系，将配置信息分门别类地放到多个配置文件中。在想加载主要配置文件，并将主要配置文件所依赖的配置文件同时加载时，可以在这个主要的配置文件中通过<import>元素对其所依赖的配置文件进行引入。例如：<import resource=”b.xml” />

1. <alias>

可以通过<alias>为某些<bean>起一些别名，通常情况下是为了减少输入。比如假设有个<bean>，它的名称为dataSourceForMasterDatabase，可以为其添加一个<alias>，像这样<alias name=” dataSourceForMasterDatabase” alias=”masterDataSource” />。以后通过dataSourceForMasterDatabase或者masterDataSource来引用这个<bean>都可以。

1. **<bean>**

每个业务对象作为个体，在Spring的XML配置文件中是与<bean>元素一一对应的。最基础的对象配置形式如下：

<bean id=*"djNewsListener"* class=*"..impl.DowJonesNewsListener"*>

</bean>

**属性：**

1. id：通常每个注册到容器的对象都需要一个唯一标志来与其他对象区分开来。通过id属性来指定当前注册对象的beanName是什么。实际上，并非任何情况下都需要指定每个<bean>的id，有些情况下，id可以省略，比如内部<bean>以及不需要根据beanName明确依赖关系的场合等。

除了可以使用id来指定<bean>在容器中的标志，还可以使用name属性来指定<bean>的别名（alias）。

<bean id=*"djNewsListener"* name=*"/news/djNewsListener,dowJonesNewsListener"* class=*"..impl.DowJonesNewsListener"*>

</bean>

与id属性相比，name属性的灵活之处在于，name属性可以使用id不能使用的一些字符，比如/。而且还可以通过都好、空格或者冒号分割指定多个name。name的作用跟使用<alias>为id指定多个别名基本相同：

<alias name=*"djNewsListener"* alias=*"/news/djNewsListener"*/>

<alias name=*"djNewsListener"* alias=*"dowJonesNewsListener"*/>

1. class

每个注册到容器的对象都需要通过<bean>元素的class属性指定其类型，否则，容器可不知道这些对象到底是什么。在大部分情况下，该属性是必须的。仅少数情况下不需要指定，如抽象配置模板的情况。该属性的值为包.类名

1. depends-on

通常情况下，我们可以使用类似<ref>元素或者ref属性明确指定所依赖的对象。让容器在实例化该对象之前先去实例化所依赖的对象。如果有这样一种场景，系统中有一个日志记录的类，都需要在这些类使用之前首先初始化log4j，那么就会非显示地依赖日志类。如果某一个对象需要使用log4j，那么就必须在bean定义中使用depends-on来要求容器在初始化自身实力之前首先爱你实例化日志类，以保证日志系统的可用。

<bean id=*"classAInstance"* class=*"...ClassA"* depends-on=*"configSetup"*/>

<bean id=*"configSetup"* class=*"SystemConfigurationSetup"*/>

如果bean有多个类似的非显示依赖关系，那么可以在depends-on属性中以逗号分隔各个beanName。

<bean id=*"classAInstance"* class=*"...ClassA"* depends-on=*"configSetup,configSetup2,..."*/>

<bean id=*"configSetup"* class=*"SystemConfigurationSetup"*/>

<bean id=*"configSetup2"* class=*"SystemConfigurationSetup2"*/>

1. autowire

除了可以通过配置明确指定bean之间的依赖关系，Spring还提供了根据bean定义的某些特点将相互依赖的某些bean直接自动绑定的功能。通过<bean>的autowire属性，可以指定当前bean定义采用某种类型的自动绑定模式。这样就无需手动明确指定该bean定义相关的依赖关系。

Spring提供了5种自动绑定模式：no、byName、byType、constructor和autodetect。

1. no

容器默认的自动绑定模式，也就是不采用任何形式的自动绑定，完全依赖手工明确配置各个bean之间的依赖关系。

<bean id=*"beanName"* class=*"..."*/>

或者

<bean id=*"beanName"* class=*"..."* autowire=*"no"*/>

1. byName

按照类中声明的实例变量的名称，与XML配置文件中声明的bean定义的beanName的值进行匹配，相匹配的bean定义将被自动绑定到当前实例变量上。这种方式对类定义和配置的bean定义有一定的限制。

假定有如下的类：

**public** **class** Foo{

**private** Bar emphasisAttribute;

...

// 相应的setter方法定义

}

**public** **class** Bar{

...

}

那么应该使用如下的代码自定绑定定义

<bean id=*"fooBean"* class=*"...Foo"* autowire=*"byName"*></bean>

<bean id=*"emphasisAttribute"* class=*"...Bar"*></bean>

需要注意两点：第一，我们并没有明确指定fooBean的依赖关系，而仅指定了它的autowire属性为byName；第二，第二个bean定义的id为emphasisAttribute，于Foo类中的实例变量名称相同。

1. byType

如果指定当前bean定义的autowire模式为byType，那么容器会根据当前bean定义类型，分析其相应的依赖对象类型，然后到容器所管理的所有bean定义中寻找与依赖对象类型相同的bean定义，然后将找到的符合条件的bean自动绑定到当前bean定义。

对于 byName 模式中的实例类 Foo 来说，容器会在其所管理的所有bean定义中寻找类型为Bar的bean定义。如果找到，则将找到的bean绑定到 Foo 的bean定义；如果没有找到，则不做设置。但如果找到多个，容器会告诉你它解决不了“该选用哪一个”的问题，你只好自己查找原因，并自己修正该问题。所以， byType 只能保证，在容器中只存在一个符合条件的依赖对象的时候才会发挥最大的作用，如果容器中存在多个相同类型的bean定义，那么，不好意思，采用手动明确配置吧！

指定byType类型的autowire模式与byName没什么差别，只是autowire的值换成byType而已。

1. constructor

byName和byType类型的自动绑定模式是针对property的自动绑定，而constructor类型则是针对构造方法参数的类型而进行的自动绑定，它同样是byType类型的绑定模式。不过，constructor是匹配构造方法的参数类型，而不是实例属性的类型。与byType模式类似，如果找到不止一个符合条件的bean定义，那么，容器会返回错误。使用上也与byType没有太大差别，只不过是应用到需要使用构造方法注入的bean定义之上

**public** **class** Foo{

**private** Bar bar;

**public** Foo(Bar arg){

**this**.bar = arg;

}

...

}

相应配置为

<bean id=*"foo"* class=*"...Foo"* autowire=*"constructor"*/>

<bean id=*"bar"* class=*"...Bar"*></bean>

1. autodetect

这种模式是byType和constructor模式的结合体，如果对象拥有默认无参数的构造方法，容器会优先考虑 byType 的自动绑定模式。否则，会使用constructor模式。当然，如果通过构造方法注入绑定后还有其他属性没有绑定，容器也会使用 byType 对剩余的对象属性进行自动绑定。

1. dependency-check

我们可以使用每个 <bean> 的 dependency-check 属性对其所依赖的对象进行最终检查，该功能主要与自动绑定结合使用，可以保证当自动绑定完成后，最终确认每个对象所依赖的对象是否按照所预期的那样被注入。当然，并不是说不可以与平常的明确绑定方式一起使用。

该功能可以帮我们检查每个对象某种类型的所有依赖是否全部已经注入完成，不过可能无法细化到具体的类型检查。但某些时候，使用setter方法注入就是为了拥有某种可以设置也可以不设置的灵活性，所以，这种依赖检查并非十分有用，尤其是在手动明确绑定依赖关系的情况下。

Spring容器提供了4种类型的依赖检查：

1. none 。不做依赖检查。将 dependency-check 指定为 none 跟不指定这个属性等效，所以，还是不要多敲那几个字符了吧。默认情况下，容器以此为默认值。
2. simple 。如果将 dependency-check 的值指定为 simple ，那么容器会对简单属性类型以及相关的collection进行依赖检查，对象引用类型的依赖除外。
3. object 。只对对象引用类型依赖进行检查。
4. all 。将 simple 和 object 相结合，也就是说会对简单属性类型以及相应的collection和所有对象引用类型的依赖进行检查。
5. lazy-init

延迟初始化（ lazy-init ）这个特性的作用，主要是可以针对 ApplicationContext 容器的bean初始化行为施以更多控制。与 BeanFactory 不同， ApplicationContext 在容器启动的时候，就会马上对所有的“singleton的bean定义”进行实例化操作。通常这种默认行为是好的，因为如果系统有问题的话，可以在第一时间发现这些问题，但有时，我们不想某些bean定义在容器启动后就直接实例化，可能出于容器启动时间的考虑，也可能出于其他原因的考虑。总之，我们想改变某个或者某些bean定义在 ApplicationContext 容器中的默认实例化时机。这时，就可以通过 <bean> 的 lazy-init 属性来控制这种初始化行为

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

<bean id=*"not-lazy-init-bean"* class=*"..."*/>

这样， ApplicationContext 容器在启动的时候，只会默认实例化 not-lazy-init-bean 而不会实

例化 lazy-init-bean 。

当然，仅指定 lazy-init-bean 的 lazy-init 为 true ，并不意味着容器就一定会延迟初始化该bean

的实例。如果某个非延迟初始化的bean定义依赖于 lazy-init-bean ，那么毫无疑问，按照依赖决计

的顺序，容器还是会首先实例化 lazy-init-bean ，然后再实例化后者

1. parent

该属性用于指定当前bean与其他bean之间的继承关系。

**class** SpecificFXNewsProvider **extends** FXNewsProvider{

**private** IFXNewsListener newsListener;

**private** IFXNewsPersister newPersistener;

...

}

配置信息

<bean id=*"superNewsProvider"* class=*"..FXNewsProvider"*>

<property name=*"newsListener"*>

<ref bean=*"djNewsListener"*/>

</property>

<property name=*"newPersistener"*>

<ref bean=*"djNewsPersister"*/>

</property>

</bean>

<bean id=*"subNewsProvider"* parent=*"superNewsProvider"* class=*"..SpecificFXNewsProvider"*>

<property name=*"newsListener"*>

<ref bean=*"specificNewsListener"*/>

</property>

</bean>

我们在声明 subNewsProvider 的时候，使用了 parent 属性，将其值指定为 superNewsProvider ，这样就继承了 superNewsProvider 定义的默认值，只需要将特定的属性进行更改，而不要全部又重新定义一遍。

parent 属性还可以与 abstract 属性结合使用，达到将相应bean定义模板化的目的。

<bean id=*"newsProviderTemplate"* abstract=*"true"*>

<property name=*"newPersistener"*>

<ref bean=*"djNewsPersister"*/>

</property>

</bean>

<bean id=*"superNewsProvider"* parent=*"newsProviderTemplate"* class=*"..FXNewsProvider"*>

<property name=*"newsListener"*>

<ref bean=*"djNewsListener"*/>

</property>

</bean>

<bean id=*"subNewsProvider"* parent=*"newsProviderTemplate"* class=*"..SpecificFXNewsProvider"*>

<property name=*"newsListener"*>

<ref bean=*"specificNewsListener"*/>

</property>

</bean>

newsProviderTemplate 的 bean 定义通过 abstract 属性声明为 true ，说明这个bean定义不需要实例化。实际上，这就是之前提到的可以不指定class属性的少数场景之一（当然，同时指定 class 和abstract="true" 也是可以的）。该bean定义只是一个配置模板，不对应任何对象。 superNewsProvider 和 subNewsProvider 通过 parent 指向这个模板定义，就拥有了该模板定义的所有属性配置。当多个bean定义拥有多个相同默认属性配置的时候，你会发现这种方式可以带来很大的便利。

另外，既然这里提到 abstract ，对它就多说几句。容器在初始化对象实例的时候，不会关注将abstract 属性声明为 true 的bean定义。如果你不想容器在初始化的时候实例化某些对象，那么可以将其 abstract 属性赋值 true ，以避免容器将其实例化。对于 ApplicationContext 容器尤其如此，因为默认情况下， ApplicationContext 会在容器启动的时候就对其管理的所有bean进行实例化，只有标

志为 abstract 的bean除外。

1. singleton和scope

BeanFactory除了拥有IoC Service Provider的职责，还拥有其他一些职责，其中就包括对象的生命周期管理。singleton属性就是用于bean对象在容器中的生命周期，被称为scope。scope用来声明容器中的对象所应该处的限定场景或者说该对象的存活时间，即容器在对象进入其相应的scope之前，生成并装配这些对象，在该对象不再处于这些scope的限定之后，容器通常会销毁这些对象。

Spring容器提供了5种scope类型，即singleton、prototype、request、session和global session类型。不过这三种类型有所限制，只能在Web应用中使用。也就是说，只有在支持Web应用的 ApplicationContext 中使用这三个scope才是合理的。

我们可以通过使用 <bean> 的singleton或者scope属性来指定相应对象的scope，其中，scope属性只能在 SD格式的文档声明中使用

DTD:

<bean id=*"mockObject1"* class=*"...MockBusinessObject"* singleton=*"false"*/>

XSD:

<bean id=*"mockObject2"* class=*"...MockBusinessObject"* scope=*"prototype"*/>

1. singleton

配置中的bean定义可以看作是一个模板，容器会根据这个模板来构造对象。但是要根据这个模板构造多少对象实例，又该让这些构造完的对象实例存活多久，则由容器根据bean定义的scope语意来决定。标记为拥有singleton scope的对象定义，在Spring的IoC容器中只存在一个实例，所有对该对象的引用将共享这个实例。该实例从容器启动，并因为第一次被请求而初始化之后，将一直存活到容器退出，也就是说，它与IoC容器“几乎”拥有相同的“寿命”。

通常情况下，如果你不指定bean的scope，singleton便是容器默认的scope，所以，下面三种配置形式实际上达成的是同样的效果：

<!-- DTD or XSD -->

<bean id=*"mockObject1"* class=*"...MockBusinessObject"*/>

<!-- DTD -->

<bean id=*"mockObject1"* class=*"...MockBusinessObject"* singleton=*"true"*/>

<!-- XSD -->

<bean id=*"mockObject1"* class=*"...MockBusinessObject"* scope=*"singleton"*/>

1. prototype

针对声明为拥有prototype scope的bean定义，容器在接到该类型对象的请求的时候，会每次都重新生成一个新的对象实例给请求方。虽然这种类型的对象的实例化以及属性设置等工作都是由容器负责的，但是只要准备完毕，并且对象实例返回给请求方之后，容器就不再拥有当前返回对象的引用，请求方需要自己负责当前返回对象的后继生命周期的管理工作，包括该对象的销毁。也就是说，容器每次返回给请求方一个新的对象实例之后，就任由这个对象实例“自生自灭”了。

<!-- DTD -->

<bean id=*"mockObject1"* class=*"...MockBusinessObject"* singleton=*"false"*/>

<!-- XSD -->

<bean id=*"mockObject1"* class=*"...MockBusinessObject"* scope=*"prototype"*/>

1. request、session和global session

这三个scope类型是Spirng 2.0之后新增加的，它们不像之前的singleton和prototype那么“通用”，因为它们只适用于Web应用程序，通常是与 XmlWebApplicationContext 共同使用。

request

Spring容器，即 XmlWebApplicationContext 会为每个HTTP请求创建一个全新的 RequestProcessor 对象供当前请求使用，当请求结束后，该对象实例的生命周期即告结束。当同时有10个HTTP请求进来的时候，容器会分别针对这10个请求返回10个全新的 RequestProcessor 对象实例，且它们之间互不干扰。从不是很严格的意义上说，request可以看作prototype的一种特例，除了场景更加具体之外，语意上差不多。

<bean id=*"requestProcessor"* class=*"...RequestProcessor"* scope=*"request"*/>

session

Spring容器会为每个独立的session创建属于它们自己的全新的 UserPreferences 对象实例。与request相比，除了拥有session scope的bean的实例具有比request scope的bean可能更长的存活时间，其他方面真是没什么差别。

<bean id=*"requestProcessor"* class=*"...RequestProcessor"* scope=*"session"*/>

global session

global session只有应用在基于portlet的Web应用程序中才有意义，它映射到portlet的global范围的session。如果在普通的基于servlet的Web应用中使用了这个类型的scope，容器会将其作为普通的session类型的scope对待。

<bean id=*"requestProcessor"* class=*"...RequestProcessor"* scope=*"globalSession"*/>

1. 自定义scope类型

在Spring 2.0之后的版本中，容器提供了对scope的扩展点，可以根据自己的需要或者应用的场景，来添加自定义的scope类型。需要说明的是，默认的singleton和prototype是硬编码到代码中的，而request、session和global session，包括自定义scope类型，则属于可扩展的scope行列，它们都实现了 rg.springframework.beans.factory.config.Scope接口，该接口定义如下：

**public** **interface** Scope {

Object get(String name, ObjectFactory objectFactory);

Object remove(String name);

**void** registerDestructionCallback(String name, Runnable callback);

String getConversationId();

}

要实现自己的scope类型，首先需要给出一个 Scope 接口的实现类，接口定义中的4个方法并非都是必须的，但 get 和 remove 方法必须实现。

例如：

**public** **class** ThreadScope **implements** Scope {

**private** **final** ThreadLocal threadScope = **new** ThreadLocal() {

**protected** Object initialValue() {

**return** **new** HashMap();

}

};

**public** Object get(String name, ObjectFactory objectFactory) {

Map scope = (Map) threadScope.get();

Object object = scope.get(name);

**if**(object==**null**) {

object = objectFactory.getObject();

scope.put(name, object);

}

**return** object;

}

**public** Object remove(String name) {

Map scope = (Map) threadScope.get();

**return** scope.remove(name);

}

**public** **void** registerDestructionCallback(String name, Runnable callback) {

}

...

}

有了 Scope 的实现类之后，我们需要把这个 Scope 注册到容器中，才能供相应的bean定义使用。通常情况下，我们可以使用 ConfigurableBeanFactory 的以下方法注册自定义 scope ：

**void** registerScope(String scopeName, Scope scope);

其中，参数 scopeName 就是使用的bean定义可以指定的名称，比如Spring框架默认提供的自定义scope类型request或者session。参数 scope 即我们提供的 Scope 实现类实例。

对于以上的 ThreadScope ，如果容器为 BeanFactory 类型（当然，更应该实现 ConfigurableBeanFactory ），我们可以通过如下方式来注册该 Scope ：

Scope threadScope = **new** ThreadScope();

beanFactory.registerScope("thread",threadScope);

之后，我们就可以在需要的bean定义中直接通过“ thread ”名称来指定该bean定义对应的scope为以上注册的 ThreadScope 了。

<bean id=*"beanName"* class=*"..."* scope=*"thread"*/>

除了直接编码调用 ConfigurableBeanFactory 的 registerScope 来注册 scope ，Spring还提供了一个专门用于统一注册自定义scope的 BeanFactoryPostProcessor 实现即 org.springframework.beans.factory.config.CustomScopeConfigurer 。对于 ApplicationContext 来说，因为它可以自动识别并加载 BeanFactoryPostProcessor ，所以我们就可以直接在配置文件中，通过这个 CustomScopeConfigurer 注册来 ThreadScope

<bean class=*"org.springframework.beans.factory.config.CustomScopeConfigurer"*>

<property name=*"scopes"*>

<map>

<entry key=*"thread"* value=*"com.foo.ThreadScope"*/>

</map>

</property>

</bean>

在以上工作全部完成之后，我们就可以在自己的bean定义中使用这个新增加到容器的自定义scope“thread”了，如下代码演示了通常情况下“thread”自定义scope的使用：

<bean id=*"beanName"* class=*"..."* scope=*"thread"*>

<aop:scoped-proxy/>

</bean>

1. factory-method

在强调”面向接口编程“的同时，有一点需要注意：虽然对象可以通过声明接口来避免对特定接口实现类的过度耦合，但总归要一种方式将声明依赖接口的对象与接口实现类关联起来。否则只依赖一个不做任何事情的接口是没有任何用处。

factory-method属性主要用于实现bean对接口对象的依赖，通过以工厂方法模式提供一个工厂类来实例化具体的接口实现类，这样主体对象只需要依赖工厂类，具体使用的实现类有变更的话，只是变更工厂类，而主体对象不需要做任何变动。

**public** **class** Foo{

**private** BarInterface barInterface;

**public** Foo(){

// barInterface = BarInterfaceFactory.getInstance();

// 或者

// barInterface = new BarInterfaceFactory().getInstance();

}

...

}

针对使用工厂方法模式实例化对象的方式，Spring的IoC容器同样提供了对应的集成支持。我们所要做的，只是将工厂类所返回的具体的接口实现类注入给主体对象（这里是 Foo ）。

1. 静态工厂方法

假设某个第三方库发布了BarInterface，为了向使用该接口的对象屏蔽以后肯呢个对BarInterface实现类的变动，同时还提供了一个静态的工厂方法实现类StaticBarInterfaceFactory

**public** **class** StaticBarInterfaceFactory{

**public** **static** BarInterface getInstance(){

**return** **new** BarInterfaceImpl();

}

}

为了将该静态工厂方法类返回的实例注入Foo，我们使用以下方式进行配置（通过setter方法注入方式为Foo注入BarInterface的实例）

<bean id=*"foo"* class=*"...Foo"*>

<property name=*"barInterface"*>

<ref bean=*"bar"*/>

</property>

</bean>

<bean id=*"bar"* class=*"...StaticBarInterfaceFactory"* factory-method=*"getInstance"*/>

class 指定静态方法工厂类， factory-method 指定工厂方法名称，然后，容器调用该静态方法工厂类的指定工厂方法（ getInstance ），并返回方法调用后的结果，即 BarInterfaceImpl 的实例。也就是说，为 foo 注入的 bar 实际上是 BarInterfaceImpl 的实例，即方法调用后的结果，而不是静态工厂方法类（ StaticBarInterfaceFactory ）。我们可以实现自己的静态工厂方法类返回任意类型的对象实例，但工厂方法类的类型与工厂方法返回的类型没有必然的相同关系。

某些时候，有的工厂类的工厂方法可能需要参数来返回相应实例，而不一定非要像我们的getInstance() 这样没有任何参数。对于这种情况，可以通过 <constructor-arg> 来指定工厂方法需要的参数，比如现在 StaticBarInterfaceFactory 需要其他依赖来返回某个 BarInterface 的实现

**public** **class** StaticBarInterfaceFactory{

**public** **static** BarInterface getInstance(Foobar foobar){

**return** **new** BarInterfaceImpl(foobar);

}

}

为了让包含方法参数的工厂方法能够预期返回相应的实现类实例，我们可以通过 <constructor-arg> 为工厂方法传入相应参数。

<bean id=*"foo"* class=*"...Foo"*>

<property name=*"barInterface"*>

<ref bean=*"bar"*/>

</property>

</bean>

<bean id=*"bar"* class=*"...StaticBarInterfaceFactory"* factory-method=*"getInstance"*>

<constructor-arg>

<ref bean=*"foobar"*/>

</constructor-arg>

</bean>

<bean id=*"foobar"* class=*"...FooBar"*/>

唯一需要注意的就是，针对静态工厂方法实现类的bean定义，使用 <constructor-arg> 传入的是工厂方法的参数，而不是静态工厂方法实现类的构造方法的参数。（况且，静态工厂方法实现类也没有提供显式的构造方法。）

1. 非静态工厂方法

既然可以将静态工厂方法实现类的工厂方法调用结果作为bean注册到容器中，我们同样可以针对基于工厂类实例的工厂方法调用结果应用相同的功能，只不过，表达方式可能需要稍微变一下。

现在为 BarInterface 提供非静态的工厂方法实现类，该类定义如下代码所示：

**public** **class** NonStaticBarInterfaceFactory{

**public** BarInterface getInstance(){

**return** **new** BarInterfaceImpl();

}

...

}

配置信息

<bean id=*"foo"* class=*"...Foo"*>

<property name=*"barInterface"*>

<ref bean=*"bar"*/>

</property>

</bean>

<bean id=*"barFactory"* class=*"...NonStaticBarInterfaceFactory"*/>

<bean id=*"bar"* factory-bean=*"barFactory"* factory-method=*"getInstance"*/>

NonStaticBarInterfaceFactory 是作为正常的bean注册到容器的，而 bar 的定义则与静态工厂方法的定义有些不同。现在使用 factory-bean 属性来指定工厂方法所在的工厂类实例，而不是通过class 属性来指定工厂方法所在类的类型。指定工厂方法名则相同，都是通过 factory-method 属性进行的。

如果非静态工厂方法调用时也需要提供参数的话，处理方式是与静态的工厂方法相似的，都可以通过 <constructor-arg> 来指定方法调用参数。

1. FactoryBean

FactoryBean 是Spring容器提供的一种可以扩展容器对象实例化逻辑的接口。 FactoryBean ，其主语是Bean，定语为Factory，也就是说，它本身与其他注册到容器的对象一样，只是一个Bean而已，只不过，这种类型的Bean本身就是生产对象的工厂（Factory）。

当某些对象的实例化过程过于烦琐，通过XML配置过于复杂，使我们宁愿使用Java代码来完成这个实例化过程的时候，或者，某些第三方库不能直接注册到Spring容器的时候，就可以实现 org.springframework.beans.factory.FactoryBean 接口，给出自己的对象实例化逻辑代码。当然，不使用 FactoryBean ，而像通常那样实现自定义的工厂方法类也是可以的。

要实现并使用自己的 FactoryBean 其实很简单， org.springframework.beans.factory.FactoryBean 只定义了三个方法:

**public** **interface** FactoryBean {

Object getObject() **throws** Exception;

Class getObjectType();

**boolean** isSingleton();

}

getObject() 方法会返回该 FactoryBean “生产”的对象实例，我们需要实现该方法以给出自己的对象实例化逻辑； getObjectType() 方法仅返回 getObject() 方法所返回的对象的类型，如果预先无法确定，则返回 null ； isSingleton() 方法返回结果用于表明，工厂方法（ getObject() ）所“生产”的对象是否要以singleton形式存在于容器中。如果以singleton形式存在，则返回 true ，否则返回 false ；

**public** **class** NextDayDateFactoryBean **implements** FactoryBean {

**public** Object getObject() **throws** Exception {

**return** **new** DateTime().plusDays(1);

}

**public** Class getObjectType() {

**return** DateTime.**class**;

}

**public** **boolean** isSingleton() {

**return** **false**;

}

}

要使用 NextDayDateFactoryBean ，只需要如下这样将其注册到容器即可：

<bean id=*"nextDayDateDisplayer"* class=*"...NextDayDateDisplayer"*>

<property name=*"dateOfNextDay"*>

<ref bean=*"nextDayDate"*/>

</property>

</bean>

<bean id=*"nextDayDate"* class=*"...NextDayDateFactoryBean"*></bean>

1. **<constructor-arg>**

该元素用于指定当前bean使用构造方法的方式注入容器中。该元素必须包含在<bean>元素内。

<bean id=*"djNewsProvider"* class=*"..FXNewsProvider"*>

<constructor-arg>

<ref bean=*"djNewsListener"*/>

</constructor-arg>

<constructor-arg>

<ref bean=*"djNewsPersister"*/>

</constructor-arg>

</bean>

其中<ref>元素用于指定依赖的对象，新版本中也可将<constructor-arg>与<ref>一起简写

<bean id=*"djNewsProvider"* class=*"..FXNewsProvider"*>

<constructor-arg ref=*"djNewsListener"*/>

<constructor-arg ref=*"djNewsPersister"*/>

</bean>

**属性：**

1. type：如果一个类定义了两个构造方法，只是构造方法的参数列表类型不一样。如果没有指定type属性，则会默认调用第一个构造方法。type属性就是用于指定调用构造方法注入传入的参数的数据类型，从而确定调用那种构造方法实现注入。

**public** **class** MockBusinessObject {

**private** String dependency1;

**private** **int** dependency2;

**public** MockBusinessObject(String dependency){

**this**.dependency1 = dependency;

}

**public** MockBusinessObject(**int** dependency){

**this**.dependency2 = dependency;

}

...

@Override

**public** String toString() {

**return** **new** ToStringBuilder(**this**).append("dependency1", dependency1).append("dependency2", dependency2).toString();

}

}

该类声明了两个构造方法。分别都只是传入一个参数，且参数类型不同。这时，我们可以进行配置

<bean id=*"mockBO"* class=*"..MockBusinessObject"*>

<constructor-arg>

<value>111111</value>

</constructor-arg>

</bean>

如果从BeanFactory取得对象并调用toString()查看的话，我们发现Spring调用的是第一个构造方法。

如果我们需要调用第二个构造方法可以使用type属性，通过制定构造方法的参数类型来解决这一问题。

<bean id=*"mockBO"* class=*"..MockBusinessObject"*>

<constructor-arg type=*"int"*>

<value>111111</value>

</constructor-arg>

</bean>

1. index

当某个业务对象的构造方法同时传入了多个类型相同的参数时，Spring又该如何将配置中的信息与实际对象的参数一一对应。好在，如果配置项信息和对象参数可以按照顺序初步对应的话。但是如果不按照顺序定义，那么有如何一一对应呢？可以使用index属性来指定参数的顺序，第一个参数的对应index值为0，以此类推。

**public** **class** MockBusinessObject {

**private** String dependency1;

**private** String dependency2;

**public** MockBusinessObject(String dependency1,String dependency2){

**this**.dependency1 = dependency1;

**this**.dependency2 = dependency2;

}

...

@Override

**public** String toString() {

**return** **new** ToStringBuilder(**this**).append("dependency1", dependency1).append("dependency2", dependency2).toString();

}

}

<bean id=*"mockBO"* class=*"..MockBusinessObject"*>

<constructor-arg index=*"1"* value=*"11111"*/>

<constructor-arg index=*"0"* value=*"22222"*/>

</bean>

1. ref

该属性用于指定对应bean对象所依赖的其他对象。其值为被依赖对象的id值。该属性不是必须的，如果没指定该属性，可以使用所对应的<ref>元素来实现相同的功能。<ref>必须包含在<constructor-arg>内部。要么使用ref属性，要么使用<ref>元素。

1. value

该属性用于指定对应bean通过构造方式注入时需要给构造方法传递的参数列表。也可以使用<value>元素实现相同的效果。其元素也必须包含在<constructor-arg>内部。

1. **<property>**

该元素用于指定当前bean使用setter方法的方式注入进容器中。该元素必须包含在<bean>元素中。

<bean id=*"djNewsProvider"* class=*"..FXNewsProvider"*>

<property name=*"newsListener"*>

<ref bean=*"djNewsListener"*/>

</property>

<property name=*"newPersistener"*>

<ref bean=*"djNewsPersister"*/>

</property>

</bean>

**属性：**

1. name

该属性用于指定该<property>将会注入的对象所对应的实例的属性名称，值必须要和属性名称一致。

1. value

该属性用于指定对应bean通过setter方法的方式注入时需要给构造方法传递的参数列表。也可以使用<value>元素实现相同的效果。其元素也必须包含 在<property>内部。

1. ref

该属性用于指定对应bean对象所依赖的其他对象。其值为被依赖对象的id值。该属性不是必须的，如果没指定该属性，可以使用所对应的<ref>元素来 实现相同的功能。<ref>必须包含在<property>内部。要么使用ref属性，要么使用<ref>元素。

之前我们看到，可以通过在<property>和<constructor-arg>这两个元素内部嵌套<value>或者<ref>，来指定将为当前对象注入的简单数据类型或者某个对象的引用。不过，为了能够指定多种注入类型，Spring还提供了其他的元素供我们使用，这包括bean、ref、idref、value、null、list、set、map、props。

1. **<value>**

可以通过value为主体对象注入简单的数据类型，不但可以指定String类型的数据，而且可以指定其他Java语言中的原始类型以及它们的包装器（wrapper） 类型，比如int、Integer等。容器在注入的时候，会做适当的转换工作

<bean id=*"djNewsProvider"* class=*"..FXNewsProvider"*>

<constructor-arg>

<value>111111</value>

</constructor-arg>

<property name=*"attributeName"*>

<value>222222</value>

</property>

</bean>

也可以使用如下的简化形式（不过这里的value是以上一层元素的属性身份出现）

<bean id=*"djNewsProvider"* class=*"..FXNewsProvider"*>

<constructor-arg value=*"111111"*/>

<property name=*"attributeName"* value=*"222222"*/>

</bean>

需要说明的是，<value>是最“底层”的元素，它内部不能再嵌套使用其他元素了。

1. **<ref>**

使用ref来引用容器中其他的对象实例，可以通过ref的local、parent和bean属性来指定引用的对象的beanName是什么。

<bean id=*"djNewsProvider"* class=*"..FXNewsProvider"*>

<constructor-arg>

<ref local=*"djNewsPersister"*/>

</constructor-arg>

或者

<constructor-arg>

<ref parent=*"djNewsPersister"*/>

</constructor-arg>

或者

<constructor-arg>

<ref bean=*"djNewsPersister"*/>

</constructor-arg>

</bean>

local、parent和bean的区别在于：

1. local只能指定与当前配置的对象在同一个配置文件的对象定义的名称（可以获得XML解析器的id约束验证支持）；
2. parent则只能指定位于当前容器的父容器中定义的对象引用；
3. bean则基本上通吃，所以，通常情况下，直接使用bean来指定对象引用就可以了。
4. **<idref>**

如果要为当前对象注入所依赖的对象的名称，而不是引用，那么通常情况下，可以使用<value>来达到这个目的，使用如下形式：

<bean id=*"djNewsProvider"* class=*"..FXNewsProvider"*>

<property name=*"newsListenerBeanName"*>

<value>djNewsListener</value>

</property>

</bean>

但这种场合下，使用idref才是最为合适的。因为使用idref，容器在解析配置的时候就可以帮 你检查这个beanName到底是否存在，而不用等到运行时才发 现这个beanName对应的对象实例不存在。毕竟，输错名字的问题很常见。

<bean id=*"djNewsProvider"* class=*"..FXNewsProvider"*>

<property name=*"newsListenerBeanName"*>

<idref bean=*"djNewsListener"*/>

</property>

</bean>

这段配置跟上面使用<value>达到了相同的目的，不过更加保险。如果愿意，也可以通过local而不是bean来指定最终值，不过，bean比较大众化哦。

1. **内部<bean>**

使用<ref>可以引用容器中独立定义的对象定义。但有时，可能我们所依赖的对象只有当前一个对象引用，或者某个对象定义我们不想其他对象通过<ref>引 用到它。这时，我们可以使用内嵌的<bean>，将这个私有的对象定义仅局限在当前对象。

<bean id=*"djNewsProvider"* class=*"..FXNewsProvider"*>

<constructor-arg index=*"0"*>

<bean class=*"..impl.DowJonesNewsListener"*></bean>

</constructor-arg>

<constructor-arg index=*"1"*>

<ref bean=*"djNewsPersister"*/>

</constructor-arg>

</bean>

这样，该对象实例就只有当前的djNewsProvider可以使用，其他对象无法取得该对象的引用。

内部<bean>的配置只是在位置上有所差异，但配置项上与其他的<bean>是没有任何差别的。

1. **<list>**

<list>对应注入对象类型为java.util.List及其子类或者数组类型的依赖对象。通过<list>可以有序地为当前对象注入以collection形式声明的依赖。

**public** **class** MockDemoObject{

**private** List param1;

**private** String[] param2;

...

// 相应的setter和getter方法

...

}

配置类似于

<property name=*"param1"*>

<list>

<value> something</value>

<ref bean=*"someBeanName"*/>

<bean class=*"..."*/>

</list>

</property>

<property name=*"param2"*>

<list>

<value>stringValue1</value>

<value>stringValue2</value>

</list>

</property>

注意：<list>元素内部可以嵌套其他元素，并且可以像param1所展示的那样夹杂配置。但是从好的编程实践来说，这样的处理并不恰当。

1. **<set>**

如果说<list>可以帮你有序地注入一系列依赖的话，那么<set>就是无序的，而且对于set来说，元素的顺序本来就是无关紧要的。<set>对应注入java.util.Set或者其子类的依赖对象。

**public** **class** MockDemoObject{

**private** Set valueSet;

// 必要的setter和getter方法

...

}

配置类似于

<property name=*"valueSet"*>

<set>

<value> something</value>

<ref bean=*"someBeanName"*/>

<bean class=*"..."*/>

<list>

...

</list>

</set>

</property>

1. **<map>**

与列表（list）使用数字下标来标识元素不同，映射map可以通过指定的键（key）来获取相应的值。<map>元素用于注入java.util.Map或者其子类类型的依赖对。

**public** **class** MockDemoObject{

**private** Map mapping;

// 必要的setter和getter方法

...

}

配置类似于

<property name=*"mapping"*>

<map>

<entry key=*"strValueKey"*>

<value>something</value>

</entry>

<entry>

<key>objectKey</key>

<ref bean=*"someObject"*/>

</entry>

<entry key-ref=*"lstKey"*>

<list>

...

</list>

</entry>

...

</map>

</property>

对于<map>来说，它可以内嵌任意多个<entry>，每一个<entry>都需要为其指定一个键和一个值，就跟真正的java.util.Map所要求的一样。

指定entry的键，可以使用<entry>的属性----key或key-ref来指定，也可以是用<entry>的内嵌元素<key>来制定。

指定entry对应的值。<entry>内部可以使用的元素，除了<key>是用来指定键的，其他元素可以任意使用，来指定entry对应的值。除了前面提到的那些元素，还可以使用<props>。如果对应的值只是简单的原始类型或者单一的对象引用，也可以直接使用<entry>的value或者value-ref这两个属性来指定。

1. **<props>**

<props>是简化后了的<map>，或者说是特殊化的map，该元素对应配置类型为java.util.Properties的对象依赖。因为Properties只能指定String类型的键和值。

**public** **class** MockDemoObject{

**private** Properties emailAddrs;

// 必要的setter和getter方法

...

}

配置类似于

<property name=*"valueSet"*>

<props>

<prop key=*"author"*>fujohnwang@gmail.com</prop>

<prop key=*"support"*>support@spring21.cn</prop>

...

</props>

</property>

每个<props>可以嵌套多个<prop>，每个<prop>可以通过其key属性来指定键，在<prop>内部之间指定其所对应的值。

1. **<null/>**

<null/>元素只是一个空元素，使用的场景并不多。对于String类型来说，如果通过value以这样的方式指定注入，即<value></value>得到的结果是””，而不是null。所以，如果需要为这个String对应的值注入null，可以使用<null/>。当然这并不仅限于String类型。

### ApplicationContext类型

ApplicationContext除了拥有BeanFactory支持的所有功能之外，还进一步扩展了基本容器的功能，包括BeanFactoryPostProcessor 、 BeanPostProcessor 以及其他特殊类型bean的自动识别、容器启动后bean实例的自动初始化、国际化的信息支持、容器内事件发布等。

ApplicationContext类型容器提供了以下几个常用的实现：

1. org.springframework.context.support.FileSystemXmlApplicationContext 。在默认情况下，从文件系统加载bean定义以及相关资源的 ApplicationContext 实现。
2. org.springframework.context.support.ClassPathXmlApplicationContext 。在默认情况下，从Classpath加载bean定义以及相关资源的 ApplicationContext 实现。
3. org.springframework.web.context.support.XmlWebApplicationContext 。Spring提供的用于Web应用程序的 ApplicationContext 实现。

#### 统一资源加载策略

要搞清楚Spring为什么提供这么一个功能，还是从Java SE提供的标准类 java.net.URL 说起比较好。URL全名是Uniform Resource Locator（统一资源定位器），但多少有些名不副实的味道。首先，说是统一资源定位，但基本实现却只限于网络形式发布的资源的查找和定位工作，基本上只提供了基于HTTP、FTP、File等协议（sun.net.www.protocol包下所支持的协议）的资源定位功能。虽然也提供了扩展的接口，但从一开始，其自身的“定位”就已经趋于狭隘了。实际上，资源这个词的范围比较广义，资源可以任何形式存在，如以二进制对象形式存在、以字节流形式存在、以文件形式存在等；而且，资源也可以存在于任何场所，如存在于文件系统、存在于Java应用的Classpath中，甚至存在于URL可以定位的地方。

其次，从某些程度上来说，该类的功能职责划分不清，资源的查找和资源的表示没有一个清晰的界限。当前情况是，资源查找后返回的形式多种多样，没有一个统一的抽象。理想情况下，资源查找完成后，返回给客户端的应该是一个统一的资源抽象接口，客户端要对资源进行什么样的处理，应该由资源抽象接口来界定，而不应该成为资源的定位者和查找者同时要关心的事情。所以，在这个前提下Spring提出了一套基于 org.springframework.core.io.Resource 和org.springframework.core.io.ResourceLoader 接口的资源抽象和加载策略。

##### Resource接口

Spring框架内部使用 org.springframework.core.io.Resource 接口作为所有资源的抽象和访问接口。

BeanFactory beanFactory = **new** XmlBeanFactory(**new** ClassPathResource("..."));

...

其中 ClassPathResource 就是 Resource 的一个特定类型的实现，代表的是位于Classpath中的资源。

Resource 接口可以根据资源的不同类型，或者资源所处的不同场合，给出相应的具体实现。Spring框架在这个理念的基础上，提供了一些实现类（可以在 org.springframework.core.io 包下找到这些实现类）。

1. ByteArrayResource 。将字节（byte）数组提供的数据作为一种资源进行封装，如果通过InputStream形式访问该类型的资源，该实现会根据字节数组的数据，构造相应的 ByteArrayInputStream 并返回。
2. ClassPathResource 。该实现从Java应用程序的ClassPath中加载具体资源并进行封装，可以使用指定的类加载器（ClassLoader）或者给定的类进行资源加载。
3. FileSystemResource 。对 java.io.File 类型的封装，所以，我们可以以文件或者URL的形式对该类型资源进行访问，只要能跟 File 打的交道，基本上跟 FileSystemResource 也可以。
4. UrlResource 。通过 java.net.URL 进行的具体资源查找定位的实现类，内部委派URL进行具体的资源操作。
5. InputStreamResource 。将给定的 InputStream 视为一种资源的 Resource 实现类，较为少用。可能的情况下，以 ByteArrayResource 以及其他形式资源实现代之。

如果以上这些资源实现还不能满足要求，那么我们还可以根据相应场景给出自己的实现，只需实现 org.springframework.core.io.Resource 接口就是了。

**public** **interface** Resource **extends** InputStreamSource {

**boolean** exists();

**boolean** isOpen();

URL getURL() **throws** IOException;

File getFile() **throws** IOException;

Resource createRelative(String relativePath) **throws** IOException;

String getFilename();

String getDescription();

}

**public** **interface** InputStreamSource {

InputStream getInputStream() **throws** IOException;

}

该接口定义了7个方法，可以帮助我们查询资源状态、访问资源内容，甚至根据当前资源创建新的相对资源。不过，要真想实现自定义的 Resource ，倒是真没必要直接实现该接口，我们可以继承org.springframework.core.io.AbstractResource 抽象类，然后根据当前具体资源特征，覆盖相应的方法就可以了。

##### ResourceLoader接口

资源是有了，但如何去查找和定位这些资源，则应该是 ResourceLoader 的职责所在了。 org.springframework.core.io.ResourceLoader 接口是资源查找定位策略的统一抽象，具体的资源查找定位策略则由相应的 ResourceLoader 实现类给出。

**public** **interface** ResourceLoader {

String ***CLASSPATH\_URL\_PREFIX*** = ResourceUtils.CLASSPATH\_URL\_PREFIX;

Resource getResource(String location);

ClassLoader getClassLoader();

}

其中最主要的就是 Resource getResource(String location); 方法，通过它，我们就可以根据指定的资源位置，定位到具体的资源实例。

1. **DefaultResourceLoader**

Org.springframework.core.io.DefaultResourceLoader类为ResourceLoader接口的实现类。该类默认的资源查找处理逻辑如下：

1. 首先检查资源路径是否以classpath：前缀打头，如果是，则尝试构造ClassPathResource类型资源并返回
2. 否则，尝试通过URL，根据资源路径来定位资源，如果没有抛出MalformedURLException，有则会构造UrlResource类型的资源并返回。如果还是无法根据资源路径定位指定的资源，则委派getResourceByPath(String)方法来定位，DefaultResourceLoader的getResourceByPath(String)方法默认实现逻辑是，构造ClassPathResource类型的资源并返回。

ResourceLoader resourceLoader = **new** DefaultResourceLoader();

Resource fakeFileResource = resourceLoader.getResource("D:/spring21site/README");

assertTrue(fakeFileResource **instanceof** ClassPathResource);

assertFalse(fakeFileResource.exists());

Resource urlResource1 = resourceLoader.getResource("file:D:/spring21site/README");

assertTrue(urlResource1 **instanceof** UrlResource);

Resource urlResource2 = resourceLoader.getResource("http://www.spring21.cn");

assertTrue(urlResource2 **instanceof** UrlResource);

**try**{

fakeFileResource.getFile();

fail("no such file with path["+fakeFileResource.getFilename()+"] exists in classpath");

}

**catch**(FileNotFoundException e){

//

}

**try**{

urlResource1.getFile();

}

**catch**(FileNotFoundException e){

fail();

}

尤其注意fakeFileResource资源的类型，并不是我们所预期的FileSystemResource类型，而ClassPathResource类型，这是由 DefaultResourceLoader的资源查找逻辑所决定的。如果最终没有找到符合条件的相应资源，getResourceByPath(String)方法就会构造一个实际上并不存在的资源并返回。而指定有协议前缀的资源路径，则通过URL能够定位，所以，返回的都是UrlResource类型。

1. **FileSystemResourceLoader**

为了避免DefaultResourceLoader在最后getResourceByPath(String)方法上的不恰当处理，我们可以使用 org.springframework.core.io.FileSystemResourceLoader类 ，它继承自 DefaultResourceLoader ，但覆写了 getResourceByPath(String) 方法，使之从文件系统加载资源并以FileSystemResource 类型返回。这样，我们就可以取得预想的资源类型。

**public** **void** testResourceTypesWithFileSystemResourceLoader(){

ResourceLoader resourceLoader = **new** FileSystemResourceLoader();

Resource fileResource = resourceLoader.getResource("D:/spring21site/README");

assertTrue(fileResource **instanceof** FileSystemResource);

assertTrue(fileResource.exists());

Resource urlResource = resourceLoader.getResource("file:D:/spring21site/README");

assertTrue(urlResource **instanceof** UrlResource);

}

FileSystemResourceLoader 在 ResourceLoader 家族中的兄弟 FileSystemXmlApplicationContext ，也是覆写了 getResourceByPath(String) 方法的逻辑，以改变 DefaultResourceLoader 的默认资源加载行为，最终从文件系统中加载并返回 FileSystemResource 类型的资源。

1. **ResourcePatternResolver接口**

org.springframework.core.io.support.ResourcePatternResolver接口是ResourceLoader的扩展，ResourceLoader每次只能根据资源路径返回确定的单个Resource实例，而 ResourcePatternResolver则可以根据指定的资源路径匹配模式，每次返回多个 Resource 实例。

**public** **interface** ResourcePatternResolver **extends** ResourceLoader {

String ***CLASSPATH\_ALL\_URL\_PREFIX*** = "classpath\*:";

Resource[] getResources(String locationPattern) **throws** IOException;

}

ResourcePatternResolver 在继承 ResourceLoader 原有定义的基础上，又引入了 Resource[] getResources(String) 方法定义，以支持根据路径匹配模式返回多个 Resources 的功能。它同时还引入了一种新的协议前缀 classpath\*: ，针对这一点的支持，将由相应的子类实现给出。

ResourcePatternResolver 最常用的一个实现类是 org.springframework.core.io.support.PathMatchingResourcePatternResolver ，该实现类支持 ResourceLoader 级别的资源加载，支持基于Ant风格的路径匹配模式（类似于 \*\* / \* .suffix之类的路径形式），支持 ResourcePatternResolver 新增加的classpath \* :前缀等，基本上集所有技能于一身。

在构造PathMatchingResourcePatternResolver实例的时候，可以指定一个 ResourceLoader，如果不指定的话，则 PathMatchingResourcePatternResolver 内部会默认构造一个 DefaultResourceLoader 实例。 PathMatchingResourcePatternResolver内部会将匹配后确定的资源路径，委派给它的 ResourceLoader 来查找和定位资源。这样，如果不指定任何ResourceLoader的话， PathMatchingResourcePatternResolver在加载资源的行为上会与DefaultResourceLoader基本相同，只存在返回的Resource数量上的差异。

如下代码表明了二者在资源加载行为上的一致性：

ResourcePatternResolver ***resourceResolver*** = **new** PathMatchingResourcePatternResolver();

Resource ***fileResource*** = ***resourceResolver***.getResource("D:/spring21site/README");

assertTrue(fileResource **instanceof** ClassPathResource);

assertFalse(fileResource.exists());

...

不过，可以通过传入其他类型的 ResourceLoader 来替换 PathMatchingResourcePatternResolver内部默认使用的 DefaultResourceLoader ，从而改变其默认行为。

比如，可以如代码清单5-4所示，使用 FileSystemResourceLoader 替换默认的 DefaultResourceLoader ，从而使得 PathMatchingResourcePatternResolver 的行为跟使用 FileSystemResourceLoader 一样。

**public** **void** testResourceTypesWithPathMatchingResourcePatternResolver(){

ResourcePatternResolver resourceResolver = **new** PathMatchingResourcePatternResolver();

Resource fileResource = resourceResolver.getResource("D:/spring21site/README");

assertTrue(fileResource **instanceof** ClassPathResource);

assertFalse(fileResource.exists());

resourceResolver = **new** PathMatchingResourcePatternResolver(**new**

FileSystemResourceLoader());

fileResource = resourceResolver.getResource("D:/spring21site/README");

assertTrue(fileResource **instanceof** FileSystemResource);

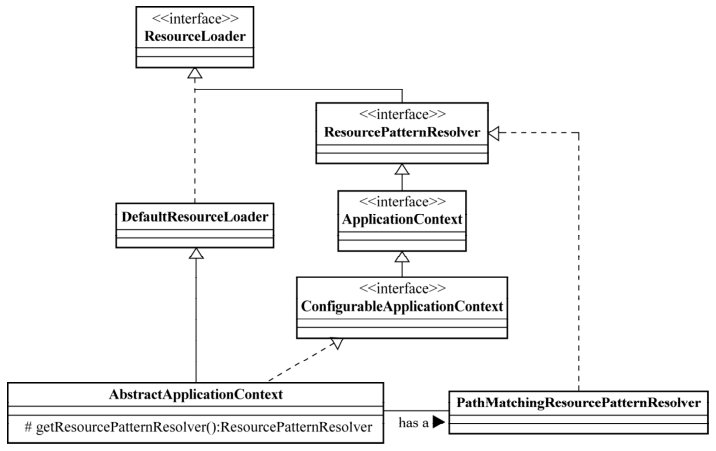
assertTrue(fileResource.exists());

}

##### ApplicationContext

ApplicationContext(统一资源加载策略)继承了ResourcePatternResolver，当然也就间接实现ResourceLoader接口。所以，任何的ApplicationContext 实现都可以看作是一个ResourceLoader甚至ResourcePatternResolver。而这就是 ApplicationContext 支持Spring内统一资源加载策略的真相。

通常，所有的ApplicationContext实现类会直接或者间接地继承 org.springframework.context.support.AbstractApplicationContext，从这个类上，我们就可以看到ApplicationContext与ResourceLoader之间的所有关系。AbstractApplicationContext继承了DefaultResourceLoader，那么，它的getResource(String) 当然就直接用DefaultResourceLoader的了。剩下需要它“效劳”的，就是 ResourcePatternResolver的Resource[]getResources (String) ，当然， AbstractApplicationContext 也不负众望，当即拿下。 AbstractApplicationContext 类的内部声明有一个resourcePatternResolver，类型是ResourcePatternResolver，对应的实例类型PathMatchingResourcePatternResolver。之前我们说过 PathMatchingResourcePatternResolver 构造的时候会接受一个 ResourceLoader ，而 AbstractApplicationContext 本身又继承自DefaultResourceLoader ，当然就直接把自身给“贡献”了。这样，整个 ApplicationContext 的实现类就完全可以支持 ResourceLoader 或者 ResourcePatternResolver 接口，你能说 ApplicationContext 不支持Spring的统一资源加载吗？说白了，ApplicationContext的实现类在作为 ResourceLoader 或者 ResourcePatternResolver 时候的行为，完全就是委派给了 PathMatchingResourcePatternResolver 和 DefaultResourceLoader 来做。



1. **扮演ResourceLoader的角色**

既然 ApplicationContext 可以作为 ResourceLoader 或者 ResourcePatternResolver 来使用，那么，很显然，我们可以通过 ApplicationContext 来加载任何Spring支持的 Resource 类型。

ResourceLoader resourceLoader = **new** ClassPathXmlApplicationContext("配置文件路径");

// 或者

// ResourceLoader resourceLoader = new FileSystemXmlApplicationContext("配置文件路径");

Resource fileResource = resourceLoader.getResource("D:/spring21site/README");

assertTrue(fileResource **instanceof** ClassPathResource);

assertFalse(fileResource.exists());

Resource urlResource2 = resourceLoader.getResource("http://www.spring21.cn");

assertTrue(urlResource2 **instanceof** UrlResource);

1. **ResourceLoader 类型的注入**

在大部分情况下，如果某个bean需要依赖于 ResourceLoader 来查找定位资源，我们可以为其注入容器中声明的某个具体的 ResourceLoader 实现，该bean也无需实现任何接口，直接通过构造方法注入或者setter方法注入规则声明依赖即可，这样处理是比较合理的。

**public** **class** FooBar {

**private** ResourceLoader resourceLoader;

**public** **void** foo(String location){

System.***out***.println(getResourceLoader().getResource(location).getClass());

}

**public** ResourceLoader getResourceLoader() {

**return** resourceLoader;

}

**public** **void** setResourceLoader(ResourceLoader resourceLoader) {

**this**.resourceLoader = resourceLoader;

}

}

以上代码中定义了一个ResourceLoader类型的引用，我们可以直接将当前的ApplicationContext容器作为ResourceLoader注入进类中。因为ApplicationContext容器本身就是一个ResourceLoader的实例。ResourceLoaderAware 和 ApplicationContextAware 接口正好可以帮助我们将ApplicationContext注入到自定义类中。

**public** **class** FooBar **implements** ResourceLoaderAware{

**private** ResourceLoader resourceLoader;

**public** **void** foo(String location){

System.***out***.println(getResourceLoader().getResource(location).getClass());

}

**public** ResourceLoader getResourceLoader() {

**return** resourceLoader;

}

**public** **void** setResourceLoader(ResourceLoader resourceLoader) {

**this**.resourceLoader = resourceLoader;

}

}

**public** **class** FooBar **implements** ApplicationContextAware{

**private** ResourceLoader resourceLoader;

**public** **void** foo(String location){

System.out.println(getResourceLoader().getResource(location).getClass());

}

**public** ResourceLoader getResourceLoader() {

**return** resourceLoader;

}

**public** **void** setApplicationContext(ApplicationContext ctx)**throws** BeansException {

**this**.resourceLoader = ctx;

}

}

剩下的就是直接将一个 FooBar 配置到bean定义文件即可，如下所示：

<bean id=*"fooBar"* class=*"...FooBar"*></bean>

现在，容器启动的时候，就会自动将当前 ApplicationContext 容器本身注入到 FooBar 中，因为 ApplicationContext 类型容器可以自动识别 Aware 接口。

当然，如果应用场景仅使用 ResourceLoader 类型即可满足需求，那么，还是使用 ResourceLoaderAware 比较合适， ApplicationContextAware 相对来说过于宽泛了些（当然，使用也未尝不可）。

1. **Resource 类型的注入**

容器可以将bean定义文件中的字符串形式表达的信息，正确地转换成具体对象定义的依赖类型。对于那些Spring容器提供的默认的PropertyEditors无法识别的对象类型我们可以提供自定义的PropertyEditor实现并注册到容器中，以供容器作类型转换的时候使用。默认情况下BeanFactory容器不会为org.springframework.core.io.Resource类型提供相应的PropertyEditor到BeanFactory容器。不过对于ApplicationContet来说，我们无需这么做，因为ApplicationContext容器可以正确识别Resource类型并转换后注入相关对象。

假设有一个 XMailer 类，它依赖于一个模板来提供邮件发送的内容，我们声明模板为 Resource 类型，那么，最终的 XMailer 定义也就如下代码：

**public** **class** XMailer {

**private** Resource template;

**public** **void** sendMail(Map mailCtx){

// String mailContext = merge(getTemplate().getInputStream(),mailCtx);

//...

}

**public** Resource getTemplate() {

**return** template;

}

**public** **void** setTemplate(Resource template) {

**this**.template = template;

}

}

该类定义与平常的bean定义没有什么差别，我们直接在配置文件中以 String 形式指定 template所在位置， ApplicatonContext 就可以正确地转换类型并注入依赖，配置内容如下：

<bean id=*"mailer"* class=*"...XMailer"*>

<property name=*"template"* value=*"..resources.default\_template.vm"*/>

...

</bean>

ApplicationContext 启动伊始，会通过一个 org.springframework.beans.support.ResourceEditorRegistrar 来注册 Spring 提供的针对 Resource 类型的 PropertyEditor 实现到容器中，这个PropertyEditor 叫做 org.springframework.core.io.ResourceEditor 。这样， ApplicationContext 就可以正确地识别 Resource 类型的依赖了。

#### 国际化信息支持

#### 容器内部事件发布

## Spring IcO容器扩展

#### Spring2.5基于注解的依赖注入

Spring 2.5提供的基于注解的依赖注入功能延续了Spring框架内在IoC容器设计与实现上的一致性。除了依赖关系的“表达”方式上的不同，底层的实现机制基本上保持一致。

##### 基于注解的自动绑定（@Autowired）

**从自动绑定（ autowire ）到 @Autowired**

在使用依赖注入绑定FXNews相关实现类时，为了减少配置量，我们可以采用Spring的IoC容器提供的自动绑定功能

<beans default-autowire=*"byType"*>

<bean id=*"newsProvider"* class=*"..FXNewsProvider"* autowire=*"byType"*/>

<bean id=*"djNewsListener"* class=*"..DowJonesNewsListener"*/>

<bean id=*"djNewsPersister"* class=*"..DowJonesNewsPersister"*/>

</beans>

可以通过 default-autowire 来指定默认的自动绑定方式，也可以通过每个bean定义上的autowire 来指定每个bean定义各自的自动绑定方式，它们都是触发容器对相应对象给予依赖注入的标志。而将自动绑定的标志用注解来表示时，也就得到了基于注解的依赖注入，或者更确切地称为基于注解的自动绑定。

@Autowired 是基于注解的依赖注入的核心注解，它的存在可以让容器知道需要为当前类注入哪些依赖。比如可以使用 @Autowired 对 FXNewsProvider 类进行标注，以表明要为 FXNewsProvider 注入的依赖。

public class FXNewsProvider{

private IFXNewsListener newsListener;

private IFXNewsPersister newPersistener;

@Autowired

public FXNewsProvider(IFXNewsListener newsListner,IFXNewsPersister newsPersister){

this.newsListener = newsListner;

this.newPersistener = newsPersister;

}

...

}

@Autowired 可以标注于类定义的多个位置，包括如下几个。

1. **属性定义**

不管它们声明的访问限制符是private、protected还是public，只要标注了@Autowired，它们所需要的依赖注入需求就都能够被满足

**public** **class** FXNewsProvider{

@Autowired

**private** IFXNewsListener newsListener;

@Autowired

**private** IFXNewsPersister newPersistener;

...

}

1. **构造方法定义**

标注于类的构造方法之上的 @Autowired ，相当于抢夺了原有自动绑定功能中“constructor”方式的权利，它将根据构造方法参数类型，来决定将什么样的依赖对象注入给当前对象。

1. **方法定义**

@Autowired 不仅可以标注于传统的setter方法之上，而且还可以标注于任意名称的方法定义之上，只要该方法定义了需要被注入的参数。

public class FXNewsProvider{

private IFXNewsListener newsListener;

private IFXNewsPersister newPersistener;

@Autowired

public FXNewsProvider(IFXNewsListener newsListner,IFXNewsPersister newsPersister){

this.newsListener = newsListner;

this.newPersistener = newsPersister;

}

...

}

现在，虽然可以随意地在类定义的各种合适的地方标注@Autowired，希望这些被@Autowired标注的依赖能够被注入，但是，仅将@Autowired标注于类定义中并不能Spring的IoC容器聪明到自己去查看这些注解，然后注入符合条件的依赖对象。容器需要某种方式来了解，哪些对象标注了@Autowired ，哪些对象可以作为可供选择的依赖对象来注入给需要的对象。

使用自动绑定的时候，我们将所有对象相关的bean定义追加到了容器的配置文件中，然后使用default-autowire 或者 autowire 告知容器，依照这两种属性指定的绑定方式，将容器中各个对象绑定到一起。在使用 @Autowired 之后， default-autowire 或者 autowire 的职责就转给了 @Autowired ，所以，现在，容器的配置文件中就只剩下了一个个孤伶伶的bean定义

<beans>

<bean id=*"newsProvider"* class=*"..FXNewsProvider"*/>

<bean id=*"djNewsListener"* class=*"..DowJonesNewsListener"*/>

<bean id=*"djNewsPersister"* class=*"..DowJonesNewsPersister"*/>

</beans>

为了给容器中定义的每个bean定义对应的实例注入依赖，可以遍历它们，然后通过反射，检查每个bean定义对应的类上各种可能位置上的 @Autowired 。如果存在的话，就可以从当前容器管理的对象中获取符合条件的对象，设置给 @Autowired 所标注的属性域、构造方法或者方法定义。

Object[] beans = ...;

**for**(Object bean:beans){

**if**(autowiredExistsOnField(bean)){

Field f = getQulifiedField(bean));

setAccessiableIfNecessary(f);

f.set(getBeanByTypeFromContainer());

}

**if**(autowiredExistsOnConstructor(bean)){

...

}

**if**(autowiredExistsOnMethod(bean)){

...

}

}

Spring提供的org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor类用于实现在实例化bean定义的过程中，来检查当前对象是否有@Autowired标注的以来需要注入。

将FXNews相关类定义使用 @Autowired 标注之后，只要在IoC容器的配置文件中追加 AutowiredAnnotationBeanPostProcessor 就可以让整个应用开始运作了

<beans>

<bean class=*"org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"*/>

<bean id=*"newsProvider"* class=*"..FXNewsProvider"*/>

<bean id=*"djNewsListener"* class=*"..DowJonesNewsListener"*/>

<bean id=*"djNewsPersister"* class=*"..DowJonesNewsPersister"*/>

</beans>

##### @Qualifier

@Autowired 是按照类型进行匹配，如果当前 @Autowired 标注的依赖在容器中能够同时找到两个或者多个同一类型的对象实例，IoC容器则不能给出正确的实例对象。@Qualifier注解对依赖注入的条件做进一步限定。@Qualifier 实际上是 byName 自动绑定的注解版，既然IoC容器无法自己从多个同一类型的实例中选取我们真正想要的那个，那么我们就使用 @Qualifier 直接点名要哪个好了。

假设 FXNewsProvider 使用的 IFXNewsListener 有两个实现，一个是 DowJonesNewsListener ，一个是 ReutersNewsListener ，二者相关配置如下：

<beans>

<bean class=*"org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"*/>

<bean id=*"newsProvider"* class=*"..FXNewsProvider"*/>

<bean id=*"djNewsListener"* class=*"..DowJonesNewsListener"*/>

<bean id=*"reutersNewsListner"* class=*"..ReutersNewsListener"*/>

<bean id=*"djNewsPersister"* class=*"..DowJonesNewsPersister"*/>

</beans>

如果我们想让 FXNewsProvider 使用 ReutersNewsListener ，那么就可以在 FXNewsProvider 的类定义中使用 @Qualifier 指定这一选择结果

**public** **class** FXNewsProvider{

@Autowired

@Qualifier("reutersNewsListner")

**private** IFXNewsListener newsListener;

@Autowired

**private** IFXNewsPersister newPersistener;

...

}

以上我们使用的是标注于属性域的 @Autowired 进行依赖注入。如果使用 @Autowired 来标注构造方法或者方法定义的话，同样可以使用 @Qualifier 标注方法参数来达到限定注入实例的目的。

**public** **class** FXNewsProvider{

...

@Autowired

**public** **void** setUp(@Qualifier("reutersNewsListner") IFXNewsListener newsListener,IFXNewsPersister newPersistener){

**this**.newsListener = newsListener;

**this**.newPersistener = newPersistener;

}

...

}

##### @Resource

@Resource 与 @Autowired 不同，它遵循的是 byName 自动绑定形式的行为准则，也就是说，IoC容器将根据 @Resource 所指定的名称，到容器中查找 beanName 与之对应的实例，然后将查找到的对象实例注入给 @Resource 所标注的对象。

**public** **class** FXNewsProvider{

@Resource(name="djNewsListener")

**private** IFXNewsListener newsListener;

@Resource(name="djNewsPersister")

**private** IFXNewsPersister newPersistener;

...

}

如果@Resource标注于属性域或者方法之上的话，相应的容器将负责把指定的资源注入给当前对象，所以，除了像我们这样直接在属性域上标注@Resource，还可以在构造方法或者普通方法定义上标注@Resource，这与@Autowired能够存在的地方大致相同。

##### @PostConstruct和@PreDestroy

@PostConstruct和@PreDestroy不是服务于依赖注入的，它们主要用于标注对象生命周期管理相关方法，这Sprin的InitializingBean和DisposableBean 接口，以及配置项中的init-method和destroy-method 起到类似的作用。

**public** **class** LifecycleEnabledClass{

@PostConstruct

**public** **void** setUp(){

...

}

@PreDestroy

**public** **void** destroy(){

...

}

}

如果想某个方法在对象实例化之后被调用，以做某些准备工作，或者想在对象销毁之前调用某个方法清理某些资源，那么就可以像我们这样，使用 @PostConstruct和@PreDestroy来标注这些方法。

与@Autowired需要AutowiredAnnotationBeanPostProcessor为它IoC容器牵线搭桥一样，@PostConstruc和@PreDestro也需要使用org.springframework.context.annotation.CommonAnnotationBeanPostProcessor类来实现相同的功能。只有将 CommonAnnotationBeanPostProcessor 添加到容器，JSR250的相关注解才能发挥作用。

<beans>

<bean class=*"org.springframework.context.annotation.CommonAnnotationBeanPostProcessor"*/>

<bean id=*"newsProvider"* class=*"..FXNewsProvider"*/>

<bean id=*"djNewsListener"* class=*"..DowJonesNewsListener"*/>

<bean id=*"djNewsPersister"* class=*"..DowJonesNewsPersister"*/>

</beans>

不管是@Autowired还是@Resource都需要添加相应的BeanPostProcessor到容器，那么我们就可以在基于XSD的配置文件中使用一个 <context:annotation-config>配置搞定以上所有的BeanPostProcessor配置。

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

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

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

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

xsi:schemaLocation=*"*

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

*http://www.springframework.org/schema/beans/spring-beans-2.5.xsd*

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

*http://www.springframework.org/schema/context/spring-context-2.5.xsd"*>

<context:annotation-config/>

<bean id=*"newsProvider"* class=*"..FXNewsProvider"*/>

<!--其他bean定义-->

...

</beans>

<context:annotation-config>不 但 帮 我 们 把AutowiredAnnotationBeanPostProcessor和CommonAnnotationBeanPostProcessor注册到容器，同时还会把PersistenceAnnotationBeanPostProcessor和RequiredAnnotationBeanPostProcessor一并进行注册，可谓一举四得啊！

##### classpath-scanning

前面谈到使用注解的方式将依赖关系注入到IoC容器中，但是在定义bean对象的时候还是需要将其配置到配置文件中去。classpath-scanning就是为了解决定义bean对象的而诞生。

classpath-scanning用于从指定的包开始进行扫描，当扫描到某个类标注了相应的注解之后，就会提取该类的相关新，构建对应的BeanDefinition，然后把构建完的BeanDefinition注册到容器。

classpath-scanning功能的触发是由<context:component-scan>决定的，所以需要在XSD形式配置文件中添加该项配置之后，classpath-scanning功能立即开启。

<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-2.5.xsd*

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

*http://www.springframework.org/schema/context/spring-context-2.5.xsd"*>

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

</beans>

现在<context:component-scan>将遍历扫描 org.spring路径下的所有类型定义，寻找标注了相应注解的类，并添加到IoC容器。

<context:component-scan>默认扫描的注解类型是@Component。不过，在@Component语义基础上细化后的@Repository、@Service和 @Controller也同样可以获得<context:component-scan>的青睐。@Component的语义更广 、更宽泛，而 @Repository、@Service和@Controller 的语义则更具体。

@Service：用于定义服务层的类。@Repository：用于定义持久层的类。Controller：用于定义视图层的类。

@Component

**public** **class** FXNewsProvider{

@Autowired

**private** IFXNewsListener newsListener;

@Autowired

**private** IFXNewsPersister newPersistener;

...

}

@Component("djNewsListener")

**public** **class** DowJonesNewsListener **implements** IFXNewsListener{

...

}

<context:component-scan>在扫描相关类定义并将它们添加到容器的时候，会使用一种默认的命名规则，来生成那些添加到容器的bean定义的名称（beanName）。比如DowJonesNewsPersister通过默认命名规则将获得dowJonesNewsPersister作为bean定义名称。如果想改变这一默认行为，就可以像以上DowJonesNewsListener所对应的@Component那样，指定一个自定义的名称

<context:component-scan> 的扫描行为可以进一步定制，默认情况下它只关心 @Component 、@Repository 、 @Service 和 @Controller 四个注解，但我们可以丰富这一范围，或者对默认的扫描结果进行过滤以排除某些类，<context:component-scan>的嵌套配置项可以帮我们达到这一目的。

<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-2.5.xsd*

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

*http://www.springframework.org/ schema/context/spring-context-2.5.xsd"*>

<context:component-scan base-package=*"org.spring"*>

<context:include-filter type=*"annotation"* expression=*"cn.spring1.annotation.FXService"*/>

<context:exclude-filter type=*"aspectj"* expression=*".."*/>

</context:component-scan>

</beans>

include-filter和exclude-filter可以使用的type类型有annotation、assignable、regex和aspectj 四种。

## Spring IoC实例

# Spring AOP

在系统开发中，通过面向对象编程（OOP）的方式能很好的将我们的业务需求进行模块化，并实现整套的系统功能需求。但是在这些义务需求之外，如果需要在对系统中的每个方法在执行之前和执行之后需要进行日志记录或者安全检查。对于这样的需求，如果系统小则可以在每个方法之前添加相应的代码即可。但是对于系统业务的增加，相应的方法会成倍的增加，之前的方法则显得不合理且也会增加后期的维护难度。在这种需求之下则有了AOP的概念

## AOP概述

AOP全程为Aspect Oriented Programming，中文为面向方面编程（又称面向切面编程）。可以说是OOP（Object Oriented Programming，面向对象编程）的补充和完善。OOP引入封装、继承、多态等概念来建立一种对象层次结构，用于模拟公共行为的一个集合。不过OOP允许开发者定义纵向的关系，但并不适合定义横向的关系，例如日志功能。日志代码往往横向地散布在所有对象层次中，而与它对应的对象的核心功能毫无关系对于其他类型的代码，如安全性、异常处理和透明的持续性也都是如此，这种散布在各处的无关的代码被称为横切（cross cutting），在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。

AOP技术恰恰相反，它利用一种称为"横切"的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其命名为"Aspect"，即切面。所谓"切面"，简单说就是那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块之间的耦合度，并有利于未来的可操作性和可维护性。

使用"横切"技术，AOP把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处基本相似，比如权限认证、日志、事物。AOP的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。

## AOP核心概念

### 横切关注点

对哪些方法进行拦截，拦截后怎么处理，这些关注点称之为横切关注点

### 连接点（Joinpoint）

在系统运行之前，AOP的功能模块都需要织入到OOP的功能模块中。所以要进行这种织入过程，需要知道在系统的那些执行点上进行织入操作，这些将要在其之上进行织入操作的系统执行点被称为Joinpoint。

**常见的Joinpoint类型：**

1. 方法调用

当某个方法被调用的时候所处的程序执行点。

1. 方法调用执行

该Joinpoint类型代表的时某个方法内部执行开始时点，应该与方法调用类型的Joinpoint进行区分。方法调用是在调用对象上的执行点，而方法执行则是在被调用到的方法逻辑执行的时点，对于同一个对象，方法调用要先于方法执行。

1. 构造方法调用

程序执行过程中对某个对象调用其构造方法进行初始化的时点。

1. 构造方法执行

构造方法执行和构造方法调用之间的关系类似于方法执行和方法调用之间的关系，指的是某个对象构造方法内部执行的开始时点。

1. 字段设置

对象的某个熟悉通过setter方法被设置或者直接被设置的时点。该Joinpoint的本质是对象的熟悉被设置，而通过setter方法设置还是直接设置触发的时点是相同的。

1. 字段获取

相对于字段设置型的Joinpoint，字段获取型的Joinpoint，对应的是某个对象相应的属性被访问的时点。可以通过getter方法访问，也可以直接访问。

1. 异常处理执行

该类型的Joinpoint对应程序执行过程中，在某些类型异常抛出后，对应的异常处理逻辑执行的时点

1. 类初始化

类初始化的Joinpoint，指的是类中某些静态类型或者静态代码块的初始化时点。

基本上，程序执行过程中你认为必要的执行时点都可以作为Joinpoint。

### 切入点（Pointcut）

Pointcut概念代表的时Joinpoint的表述方式。将横切逻辑织入当前系统的过程中，需要参照Pointcut规定的Joinpoint信息，才可以知道应该往系统的那些Joinpoint上织入横切逻辑。

### 通知（Advice）

Advice是单一横切关注点逻辑的载体，它代表将会织入到Joinpoint的横切逻辑。如果将Aspect比作OOP中的Class，那么Advice就相当于Class中的Method。

按照Advice在Joinpoint位置执行时机的差异或者完成功能的不同，Advice可以分成多种具体形式：

1. **Before Advice**

Before Advice是在Joinpoint指定位置之前执行的Advice类型。通常它不会中断程序执行流程，但如果必要，可以通过在Before Advice中抛出异常的方式来中断当前程序流程。如果当前Before Advice将被织入到方法执行类型的Joinpoint，那么这个Before Advice就会先于方法执行而执行。

通常可以使用Before Advice做一些系统的初始化工作，比如设置系统初始化值，获取必要系统资源等。

1. **After Advice**

After Advice就是在相应连接点之后执行的Advice类型，但该类型的Advice还可以细分为以下三种

1. After returning Advice

只有当前Joinpoint处执行流程正常完成后，After returning Advice才会执行。比如方法执行正常返回而没有抛出异常。

1. After throwing Advice

After throwing Advice又称Throws Advice，只有在当前Joinpoint执行过程中抛出异常的情况下才会执行。比如某个方法执行类型的Joinpoint抛出某异常而没有正常返回。

1. After Advice

该类型的Advice不管Joinpoint处执行流程是否正常执行完毕还是抛出异常都会执行。

1. **Around Advice**

Around Advice对附加其上的Joinpoint进行“包裹”，可以在Joinpoint之前和之后都指定相应的逻辑，甚至于中断或者忽略Joinpoint处原来程序流程的执行。

既然Around Advice可以在Joinpoint之前和之后都能执行相应的逻辑，那么它自然可以完成Before Advice和After Advice的功能。不过，通常情况下，还是应该根据场景选用更为具体的Advice类型。

1. **Introduction**

Introduction不是根据横切逻辑在Joinpoint处的执行时机来区分的，而是根据它可以完成的功能而区别于其他Advice类型。Introduction可以为原有的对象添加新的特性或者行为，

### 切面（Aspect）

Aspect是对系统中的横切关注点逻辑进行模块化封装的AOP概念实体。通常情况下Aspect可以包含多个Pointcut以及相关Advice定义。

Spring AOP最初没有“完全”确切的实体对应真正的Aspect的概念。在2.0发布后，因为集成了AspectJ，所以可以通过使用@AspectJ的注解并结合普通的POJO来声明Aspect。

### 织入（Weave）

织入的过程就是“飞架”AOP和OOP的那座桥，只有经过织入过程后，以Aspect模块化的横切关注点才会集成到OOP的现存系统中。而完成织入过程的那个“人”就称之为织入器。

AspectJ有专门的编译器来完成织入操作，即ajc，所以ajc就是AspectJ完成织入的织入器；Spring AOP使用一组类来完成最终的织入操作，ProxyFactory类则是Spring AOP中最通用的织入器。

### 目标对象

符合Pointcut所指定的条件，将在织入过程中被织入横切逻辑的对象，称为目标对象。

## Java平台上的AOP实现机制

### 动态代理

JDK1.3之后，引入了动态代理机制，可以在运行期间，为相应的接口动态生成对应的代理对象。所以我们可以将横切专注点逻辑封装到动态代理的InvocationHandler中，然后在系统运行期间，根据横切关注点需要织入的模块位置，将横切逻辑织入到相应的代理类中。

这种方式实现的唯一缺点或者说优点就是，所有需要织入横切关注点逻辑的模块类都得实现相应的接口，因为动态代理机制只针对接口有效。毕竟动态代理是在运行期间使用反射，相对于编译后的静态类的执行，性能可能稍逊一些。

Spring AOP默认情况下采用这种机制实现AOP。

### 动态字节码增强

Java虚拟机加载的class文件都是符合一定规范的，所以只要交给Java虚拟机运行的文件符合Java class规范，程序的运行就没有问题。通常的class文件都是从Java源代码文件使用Javac编译器编译而成的，，但只要符合Java class规范，我们也可以使用ASM或者CGLIB等Java工具库，在程序运行期间，动态构建字节码的class文件。

在这样的前提下，可以为需要织入横切逻辑的模块类在运行期间，通过动态字节码增强技术，为这些系统模块类生产相应的子类，而将横切逻辑加到这些子类中，让应用程序在执行期间使用的是这些动态生产的子类，从而达到将横切逻辑织入系统的目的。

使用动态字节码增强技术，即使模块类没有实现相应的接口，依然可以对其进行扩展，而不用像动态代码那样受限于接口。不过这种实现机制依然存在不足，如果需要扩展的类以及类中的实例方法等声明为final的话，则无法对其进行子类的扩展。

Spring AOP在无法采用动态代理的时候会使用CGLIB库的动态字节码增强支持来实现AOP的功能扩展。

### Java代码生成

事务属于跨越整个系统的一种横切关注点，所以EJB容器提供的声明性事务支持，属于一种AOP功能模块实现。但早期EJB容器在实现这一功能的时候，大多会采用Java代码生成技术，这就是我们不需要提供CMP的接口实现类的原因，也是EJB容器提供商大多提供部署接口或者专有部署工具的原因

EJB容器根据部署描述符文件提供的织入信息，会为相应的功能模块类根据描述符所提供的信息生成对应的Java代码，然后通过部署工具或者部署接口编译Java代码生成相应的Java类。之后部署到EJB容器的功能模块类就可以正常工作了。

### 自定义类加载器

所有的Java程序的class都要通过相应的类加载器加载到Java虚拟机之后才可以运行。默认的类加载器会读取class字节码文件，然后按照class字节码规范，解析并加载这些class文件到虚拟机运行。如果能够在这个class文件加载到虚拟机运行期间，将横切逻辑织入到class文件的话，是不是就完成了AOP和OOP的融合。

可以通过自定义类加载器的方式完成横切逻辑到系统的织入，自义定类加载器通过读取外部文件规定的织入规则和必要信息，在加载class文件期间就可以将横切逻辑添加到系统模块类的现有逻辑中，然后将改动后的class交给Java虚拟机运行。

通过类加载器，基本可以对大部分类以及相应的实例进行织入，功能与之前的几种方式相比强大很多。不过这种方式最大的问题就是类加载器本身的使用。某些应用服务器会控制整个的类加载体系，所以这样的场景下使用可能会造成一定的问题。

AOL扩展

AOL扩展是最强大，也是最难掌握的一种方式。在这种方式中，AOP的各种概念在AOL中大都有一一对应的实体。我们可以使用扩展过的AOL实现任何AOP概念实体甚至OOP概念实体。所有的AOP概念在AOL中得到了最完美的表达。

采用扩展的AOL，在AOP概念的表述上颇具实力，使得AOP涉及的所有横切关注点逻辑在进行织入之前，可以自由自在地存活在自己的“国度”中。而且具有强类型检查，基本可以对横切关注点要切入的系统运行时点有更全面的控制。而像“编译到静态类可以提升系统运行性能”，“Java虚拟机可以像加载平常类那样，加载已经织入相应逻辑的AO组件所在的类文件并运行”等特点。

不过该方式强大的代价就是，需要重新学习一门扩展了旧有的语言的AOL或者全新的AOL。

## Spring AOP

Spring AOP是Spring核心框架的重要组成部分，通常认为它与Spring的IoC容器以及Spring框架对其他JavaEE服务的集成共同组成了Spring框架的“质量三角”。Spring AOP采用Java作为AOP的实现语言（AOL），也对AspectJ提供了很好的集成。Spring AOP始终坚持最初的设想，使用动态代理机制构建基于Java语言的简单而强大的AOP框架。

### Spring AOP的实现机制

Spring AOP采用动态代理机制和字节码生成技术实现。动态代理机制和字节码生成都是在运行期间为目标对象生成一个代理对象，而降横切逻辑织入到这个代理对象中，系统最终使用的是织入了横切逻辑的代理对象，而不是真正的目标对象。

#### 代理模式

在代理模式中，通常涉及4中角色

1. ISubject：该接口是对被访问者或者被访问资源的抽象。
2. SubjectImpl：被访问者或者被访问资源的具体实现类。
3. SubjectProxy：被访问者或者被访问资源的代理实现类，该类持有一个ISubject接口的具体实例。这里要对SubjectImpl进行代理，那么SubjectProxy现在持有的就是SubjectImpl的实例。
4. Client：代表访问者的抽象角色，Client将会访问ISubject类型的对象或者资源。这里Client将会请求具体的SubjectImpl实例，但Client无法直接请求其真正要访问的资源SubjetImpl，而是必须要通过ISubject资源的访问代理类SubjectProxy进行。

SubjectImpl和SubjectProxy都实现了相同的接口ISubject，而SubjectProxy内部持有SubjectImpl的引用。当Client通过request()请求服务的时候，SubjectProxy将转发该请求给SubjectImpl。不过，SubjectProxy的作用不只局限于请求的转发，更多时候是对请求添加更多访问限制。

代理对象SubjectProxy就像是SubjectImpl的影子，只不过这个影子通常拥有更多的功能。如果SubjectImpl是系统中的Joinpoint所在的对象，即目标对象，那么就可以为这个目标对象创建一个代理对象，然后将横切逻辑添加到这个代理对象中。当系统使用这个代理对象运行的时候，原有逻辑的实现和横切逻辑就完全融合到一个系统中。

Spring AOP本质上就是采用这种代理机制实现的，但是具体实现细节上有所不同。假设要对系统中所有的request()方法进行拦截，在每天午夜0点到次日6点之间，request调用不被接受，那么我们应该为SubjectImpl提供一个ServiceControlSubjectProxy，以添加这种横切逻辑。

**public** **class** ServiceControlSubjectProxy **implements** ISubject{

**private** **static** **final** Log ***logger*** = LogFactory.getLog(ServiceControlSubjectProxy.**class**);

**private** ISubject subject;

**public** LogSubjectProxy(ISubject s){

**this**.subject = s;

}

**public** String request(){

TimeOfDay startTime = **new** TimeOfDay(0,0,0);

TimeOfDay endTime = **new** TimeOfDay(5,59,59);

TimeOfDay currentTime = **new** TimeOfDay();

**if**(currentTime.isAfter(startTime) && currentTime.isBefore(endTime)){

**return** **null**;

}

String originalResult = subject.request();

**return** "Proxy:" + originalResult;

}

.....

}

之后，我们使用该ServiceControlSubjectProxy代替SubjectImpl使用。

ISubject target = **new** SubjectImpl();

ISubject finalSubject = **new** ServiceControlSubjectProxy(target);

finalSubject.request();

//访问控制相关逻辑已经包含在请求的处理逻辑中

但是，系统中可不一定就ISubject的实现类有request()方法，IRequestable接口以及相应实现类可能也有request()方法，它们也是需要横切的关注点。

**public** **interface** IRequestable{

**void** request();

}

//实现类

**public** **class** RequestableImpl **implements** IRequestable{

**public** **void** request(){

System.***out***.println("request processed in RequestableImpl");

}

}

为了能够为IRequestable相应实现类也织入以上的横切逻辑，我们又得提供对应的代理对象

**public** **class** ServiceControlSubjectProxy **implements** IRequestable{

**private** **static** **final** Log ***logger*** = LogFactory.getLog(ServiceControlSubjectProxy.**class**);

**private** IRequestable requestable;

**public** LogSubjectProxy(IRequestable requestable){

**this**.requestable = requestable;

}

**Public void** request(){

TimeOfDay startTime = **new** TimeOfDay(0,0,0);

TimeOfDay endTime = **new** TimeOfDay(5,59,59);

TimeOfDay currentTime = **new** TimeOfDay();

**if**(currentTime.isAfter(startTime) && currentTime.isBefore(endTime)){

**return**;

}

requestable.request();

}

.....

}

并且将该代理对象而不是目标对象绑定到系统中。

IRequestable target = **new** RequestableImpl();

IRequestable proxy = **new** ServiceControlSubjectProxy(target);

proxy.request();

//访问控制相关逻辑已经包含在请求的处理逻辑中

虽然Joinpoint相同，但是对应的目标对象类型是不同的。针对不同的目标对象类型，我们需要为其单独实现一个代理对象。而实际上，这些代理对象所要添加的横切逻辑是一样的。当系统中存在成百上千的符合Pointcut匹配条件的目标对象时，我们就要为这些目标对象创建成百上千代理对象。这种为对应的目标对象创建静态代理的方法，原理上是可行的，但是具体应用上存在问题。

#### 动态代理

JDK1.3之后引入了动态代理的机制。使用该机制我们可以为指定的接口在系统运行期间动态地生成代理对象。动态代理机制的实现主要由一个类和一个接口组成，即java.lang.reflect.Proxy类和java.lang.reflect.InvocationHandler接口。

如果使用动态代理来实现之前的例子，虽然要为ISubject和IRequestable两种类型提供代理对象，但因为代理对象中要添加的横切逻辑是一样的，所以我们只需实现一个InvocationHandler就可以了。

**public** **class** RequestCtrlInvocationHandler **implements** InvocationHandler{

**private** **static** **final** Log ***logger*** = LogFactory.getLog(ServiceControlSubjectProxy.**class**);

**private** Object target;

**public** **void** RequestCtrlInvocationHandler(Object target){

**this**.target = target;

}

@Override

**public** Object invoke(Object proxy, Method method, Object[] arg2) **throws** Throwable {

**if**(method.getName().equals("request")){

TimeOfDay startTime = **new** TimeOfDay(0,0,0);

TimeOfDay endTime = **new** TimeOfDay(5,59,59);

TimeOfDay currentTime = **new** TimeOfDay();

**if**(currentTime.isAfter(startTime) && currentTime.isBefore(endTime)){

***logger***.warn("service is not available new.");

**return** **null**;

}

**return** method.invoke(target, args);

}

**return** **null**;

}

}

然后，我们就可以使用Proxy类，根据RequestCtrlInvocationHandler的逻辑，为ISubject和IRequestable两种类型生成相应的代理对象实例。

ISubject subject = (ISubject)Proxy.*newProxyInstance*(

ProxyRunner.**class**.getClassLoader(), **new** Class[]{ISubject.**class**},

**new** RequestCtrlInvocationHandler(**new** SubjectImple()));

subject.request();

IRequestable requestable = (IRequestable)Proxy.newProxyInstance(

ProxyRunner.**class**.getClassLoader(), **new** Class[]{IRequestable.**class**},

**new** RequestCtrlInvocationHandler(**new** RequestableImpl()));

requestable.request();

即使还有更多的目标对象类型，只要它们依然织入的横切逻辑相同，用RequestCtrlInvocationHandler一个类并通过Proxy为他们生成相应的动态代理实例就可以满足要求。当Proxy动态生成的代理对象上相应的接口方法被调用时，对应的InvocationHandler就会拦截相应的方法调用，并进行相应处理。

动态代理虽好，但不能满足所有的需求，因为动态代理机制只能对实现了相应的Interface的类使用，如果某个类没有实现任何的Interface，就无法使用动态代理机制为其生成相应的代理对象。

默认情况下，如果Spring AOP发现目标对象实现了相应的Interface，则采用动态代理机制。如果目标对象没有实现任何Interface，Spring AOP会尝试使用一个称为CGLIB的开源的动态字节码生成类库，为目标对象生成动态的代理对象实例。

#### 动态字节码生成

使用动态字节码生成技术扩展对象行为的原理是，可以对目标对象进行继承扩展，为其生成相应的子类，而子类可以通过覆写来扩展父类的行为，只能将横切逻辑的实现放到子类中，然后让系统使用扩展后的目标对象的子类，就可以达到与代理模式相同的效果。

但是使用继承的方式来扩展对象定义，也不能像静态代理模式那样，为每个不同类型的目标对象都单独创建相应的扩展子类。我们可以借助于CGLIB这样的动态字节码生成库，在系统运行期间动态为目标对象生成相应的扩展子类。

**public** **class** Requestable{

**public** **void** request(){

System.***out***.println("rg in Requestable without implement any interface);

}

}

CGLIB可以对象实现了某种接口的类或者没有实现任何接口的类进行扩展。

要对象Requestable类进行扩展，首先需要实现一个net.sf.cglib.proxy.Callback。不过更多的时候我们会直接使用net.sf.cglib.proxy.MethodInterceptor接口（MethodInterceptor接口扩展了Callback接口）

**public** **class** RequestCtrlCallback **implements** MethodInterceptor {

**private** **static** **final** Log ***logger*** = LogFactory.getLog(RequestCtrlCallback.**class**);

**public** Object intercept(Object object,Method method,Object[] args,MethodProxy proxy)**throws** Throwable{

TimeOfDay startTime = **new** TimeOfDay(0,0,0);

TimeOfDay endTime = **new** TimeOfDay(5,59,59);

TimeOfDay currentTime = **new** TimeOfDay();

**if**(currentTime.isAfter(startTime) && currentTime.isBefore(endTime)){

***logger***.warn("service is not available now.");

**return** **null**;

}

**return** proxy.invokeSuper(object, args);

}

**return** **null**;

}

这样RequestCtrlCallback就实现了对象request()方法请求进行访问控制的逻辑。现在我们要通过CGLIB的Enhancer为目标对象动态生成一个子类，并将RequestCtrlCallback中的横切逻辑附加到该子类中。

Enhancer enhancer = **new** Enhancer();

enhancer.setSuperclass(Requestable.**class**);

enhancer.setCallback(**new** RequestCtrlCallback());

Requestable proxy = (Requestable)enhancer.create();

proxy.request();

通过为enhancer指定需要生成的子类对应的父类，以及Callback实现，enhancer最终为我们生成了需要的代理对象实例。使用CGLIB对类进行扩展的唯一限制就是无法对final方法进行覆写。

### Spring AOP的具体实现

#### Joinpoint

AOP的Joinpoint可以有许多种类，如构造方法调用、字段的设置及获取、方法调用、方法执行等。但是Spring AOP中，仅支持方法级别的Joinpoint（只支持方法执行类型的Joinpoint）。

#### Pointcut

Spring中以接口定义org.springframework.aop.Pointcut作为其AOP框架中所有Pointcut的最顶层抽象，该接口定义了两个方法用来帮助捕捉系统中的相应Joinpoint，并提供了一个TruePointcut类型实例。如果Pointcut类型为TruePointcut，默认会对系统中的所有对象，以及对象上所有被支持的Joinpoint进行匹配。org.springframework.aop.Pointcut接口定义如下：

**public** **interface** Pointcut{

ClassFilter getClassFilter();

MethodMatcher getMethodMatcher();

Pointcut ***TRUE*** = TruePointcut.INSTANCE;

}

ClassFilter和MethodMatcher分别用于匹配将被执行织入操作的对象以及相应的方法。之所以将类型匹配和方法匹配分开定义，是因为可以重用不同级别的匹配定义，并且可以在不同的级别或者相同的级别上进行组合操作，或者强制让某个子类只覆写相应的方法定义等。

ClassFilter接口的作用是对Joinpoint所处的对象进行Class级别的类型匹配：

**public** **interface** ClassFilter{

**boolean** matches(Class clazz);

ClassFilter ***TRUE*** = TrueClassFilter.INSTANCE;

}

当织入的目标对象的Class类型与Pointcut所规定的类型相符时，matches方法将会返回true。意味着不会对这个类型的目标对象进行织入操作。

比如，如果仅希望对系统中Foo类型的类执行织入，则可以如下这样定义ClassFilter:

**public** **class** FooClassFilter{

**public** boolea matches(Class clazz){

**return** Foo.**class**.equals(clazz);

}

}

如果对所捕获的Joinpoint无所谓，那么Pointcut中使用的ClassFilter可以直接使用“ClassFilter ***TRUE*** = TrueClassFilter.INSTANCE;”当Pointcut中返回ClassFilter类型为该类型实例时，Pointcut的匹配将会针对系统中所有的目标类以及它们的实例进行。

对于ClassFilter的简单定义，MethodMatcher则要复杂得多。MethodMatcher接口代码如下：

**public** **interface** MethodMatcher{

**boolean** matches(Method method, Class targetClass);

**boolean** isRuntime();

**boolean** matches(Method method, Class targetClass, Object[] args);

MethodMatcher ***TRUE*** = TrueMethodMatcher.INSTANCE;

}

MethodMatcher通过重载，定义了两个matches方法，而这两个方法的分界线就是isRuntime()方法。在对对象具体方法进行拦截的时候，可以忽略每次方法执行的时候调用者传入的参数，也可以每次都检查这些方法调用参数，以强化拦截条件。

假设对以下方法进行拦截：

**public** **boolean** login(String username,String password);

如果只想在login方法之前插入计数功能，那么login方法的参数对于Joinpoint捕捉就是可以忽略。如果想在用户登录的时候对某个用户做单独处理，那么这个方法的参数就是在匹配Joinpoint的时候必须要考虑的。

当isRuntime()方法返回false，表示不会考虑具体的Joinpoint的方法参数，当前的MethodMatcher为StaticMethodMatcher的时候，只有boolean matches(Method method, Class targetClass);方法将被执行。

当isRuntime()方法返回true，表明该MethodMatcher将会每次都对方法调用的参数进行匹配检查，这种类型的MethodMatcher称为DynamicMethodMatcher。

在MethodMatcher类型的基础上，Pointcut可以分为两类，即StaticMethodMatcher和DynamicMethodMatcher。因为StaticMethodMatcher具有明显的性能优势，所以Spring为其提供了更多支持。

1. **常见的Pointcut**
2. **NameMatchMethodPointcut**

这是最简单的Pointcut实现，属于StaticMethodMatcherPointcut的子类，可以根据自身指定的一组方法名称与Joinpoint处的方法的方法名称进行匹配。

NameMatchMethodPointcut pointcut = **new** NameMatchMethodPointcut()

pointcut.setMappedName("matches");

//或者传入多个方法名

pointcut.setMappedNames(**new** String[]{"matches","isRuntime"});

但是，NameMatchMethodPointcut无法对重载的方法名进行匹配，因为它仅对方法名进行匹配，不会考虑参数相关信息，而且也没有提供可以指定参数匹配的路径。

NameMatchMethodPointcut除了可以指定方法名，也对指定的Joinpoint进行匹配，还可以使用“\*”通配符，实现简单的模糊匹配。

pointcut.setMappedNames(**new** String[]{"matches","isRuntime","mat\*es"});

如果基于“\*”通配符的NameMatchMethodPointcut依然无法满足对多个特定Joinpoint的匹配需要，那么可以使用正则表达式。

1. **JdkRegexpMethodPointcut和Per15RegexpMethodPointcut**

StaticMethodMatcherPointcut的子类中有一个专门提供基于正则表达式的实现分支，以抽象类AbstractRegexpMethodPointcut为统帅。与NameMatchMethodPointcut相似，AbstractRegexpMethodPointcut声明了pattern和patterns属性，可以指定一个或者多个正则表达式的匹配模式。其下设JdkRegexpMethodPointcut和Per15RegexpMethodPointcut两种具体实现。

JdkRegexpMethodPointcut的实现基于JDK1.4之后引入的JDK标准正则表达式。

JdkRegexpMethodPointcut pointcut = **new** JdkRegexpMethodPointcut()

pointcut.setPattern(".\*match.\*");

//或者传入多个方法名

pointcut.setPatterns(**new** String[]{".\*match.\*",".\*matches"});

注意，使用正则表达式来匹配相应的Joinpoint所处的方法时，正则表达式的匹配模式必须以匹配整个方法签名的形式指定，而不能像NameMatchMethodPointcut那样仅给出匹配的方法名称。

**package** cn.spring21.sample;

**public** vlass Bar{

**public** **void** doSth(){

}

}

使用正则表达式 .\*doSth.\*则会匹配Bar的doSth方法，即相当于cn.spring21.sample.doSth。但如果Pointcut使用 doSth.\* 作为匹配的正则表达式模式，就无法捕捉到Bar的doSth方法的执行。当然也可以通过正则表达式中的转义，更确切地指定对doSth方法的匹配：cn\.spring21\.sample\.Bar\.doSth。

如果当前应用无法使用JDK1.4或者更高版本，或者喜欢per15风格的正则表达式，那么可以使用Per15RegexpMethodPointcut。该pointcut实现使用JakartaORO提供的正则表达式支持，所以在使用之前需先将JakartaORO的jar包加入到引用程序的classpath中。除了正则表达式的语句上可能有少许差异，Per15RegexpMethodPointcut的使用和需要注意的问题与JdkRegexpMethodPointcut几乎相同。

可以通过pattern或者patterns对象属性指定一个或者多个正则表达式的匹配模式。

指定的正则表达式匹配模式应该覆盖匹配整个方法签名，而不是只指定到方法名称部分。

1. **AnnotationMatchingPointcut**

AnnotationMatchingPointcut根据目标对象中是否存在指定类型的注解来匹配Joinpoint，要使用该类型的Pointcut首先需要声明相应的注解。（基于JDK1.5及更高版本）。

@Retention(RetentionPolicy.***RUNTIME***)

@Target(ElementType.***TYPE***)

**public** **@interface** ClassLevelAnnotation{

}

@Retention(RetentionPolicy.***RUNTIME***)

@Target(ElementType.***METHOD***)

**public** **@interface** MethodLevelAnnotation{

}

注解定义中的@Target指定该注解可以标注的类型。对于我们声明的两个注解来说就是，ClassLevelAnnotation用于类层次，而MethodLevelAnnotation只能用于方法层次。

@ClassLevelAnnotation

**public** **class** GenericTargetObject{

@MethodLevelAnnotation

**public** **void** gMethod1(){

System.***out***.println("gMethod1");

}

**public** **void** gMethod2(){

System.***out***.println("gMethod2");

}

}

针对上述代码中的GenericTargetObject类型，不同的AnnotationMatchingPointcut定义会产生不同的匹配行为

**AnnotationMatchingPointcut pointcut = new AnnotationMatchingPointcut(ClassLevelAnnotation.class)**

AnnotationMatchingPointcut仅指定类级别的注解，GenericTargetObject类中所有方法执行时，将全部匹配，不管该方法指定了注解还是没有指定。也可以通过AnnotationMatchingPointcut的静态方法来构建类级别的注解对应的AnnotationMatchingPointcut实例：

AnnotationMatchingPointcut pointcut = AnnotationMatchingPointcut.forClassAnnotation(ClassLevelAnnotation.**class**);

**AnnotationMatchingPointcut pointcut = AnnotationMatchingPointcut.forMethodAnnotation(MethodLevelAnnotation.class)**

如果指定该方法级别的注解而忽略类级别的注解，则该Pointcut仅匹配特定的注解了指定注解的方法定义，而忽略其他方法。对于GenericTargetObject或者其他类中任何方法来说，只要其方法标注了@MethodLevelAnnotation，则这些方法都会被匹配并织入相应的横切逻辑。

**AnnotationMatchingPointcut pointcut = new AnnotationMatchingPointcut(ClassLevelAnnotation.class, MethodLevelAnnotation.class)**

通过同时限定类级别的注解和方法级别的注解，可以进一步缩小匹配范围。只有标注了@ClassLevelAnnotation的类定义中同时标注了@MethodLevelAnnotation的方法才会被匹配。对于GenericTargetObject类来说，只有gMethod1()方法才会被拦截，而gMethod2()则不会。

1. **ComposablePointcut**

ComposablePointcut是Spring AOP提供的可以进行Pointcut逻辑运算的Pointcut实现。它可以进行Pointcut之间的“并”以及“交”运算。

ComposablePointcut pointcut1 = **new** ComposablePointcut(classFilter1,methodMatcher1);

ComposablePointcut pointcut2 = **new** ComposablePointcut(classFilter2,methodMatcher2);

ComposablePointcut unitedPoincut = pointcut1.union(pointcut2);

ComposablePointcut intersectionPointcut = pointcut1.intersection(unitedPoincut);

assertEquals(pointcut1,intersectionPointcut)

Pointcut定义根ClassFilter和MethodMatcher划分为两部分，一部分是为了重用这些定义，另一部分是为了相互组合。而通过ComposablePointcut，可以很明白地看出这两个目的。

ComposablePointcut pointcut3 = pointcut2.union(classFilter1).intersection(methodMatcher1);

我们在pointcut1和pointcut3中复用了classFilter1和methodMatcher1以及pointcut2的定义，同时还进行了Pointcut同ClassFilter以及MethodMatcher之间的逻辑组合运算。

如果只想进行Pointcut与Pointcut之间的逻辑组合运算，Spring AOP提供了org.spring.framework.aop.support.Pointcuts工具类。

Pointcut pointcut1 = ...;

Pointcut pointcut2 = ...;

Pointcut unitedPoincut = Pointcuts.union(pointcut1,pointcut2);

Pointcut intersectionPointcut = Pointcuts.intersection(pointcut1,pointcut2);

1. **ControlFlowPointcut**

ControlFlowPointcut匹配程序的调用流程，不是对某个方法执行所在的Joinpoint处的单一特征进行匹配。

假设所拦截的目标对象的方法声明如下：

**public** **class** TargetObject{

**public** **void** method1(){

...

}

}

//调用类

**public** **class** TargetCaller{

**private** TargetObject target;

**public** **void** callMethod(){

target.method1();

}

**public** **void** setTarget(TargetObject target){

**this**.target = target;

}

}

通过ControlFlowPointcut，我们可以指定只有当TargetObject的method1()方法在TargetCaller类所声明的方法中被调用的时候，才对method1()方法进行拦截，其他地方调用method1()方法，不对method1()进行拦截。

ControlFlowPointcut pointcut = **new** ControlFlowPointcut(TargetCaller.**class**);

Advice advice = ...;

TargetObject target = **new** TargetObject();

TargetObject targetObjectToUse = weaver.weave(advice).to(target).accordingto(pointcut);

//advice的逻辑在这里将不会被触发执行

targetObjectToUse.method1();

//advice的逻辑在这里将被触发执行

//因为TargetCaller的callMethod()将调用method1()

//正像ControlFlowPointcut(TargetCaller.class)所指定的那样

TargetCaller caller = **new** TargetCaller();

caller.setTarget(targetObjectToUse);

caller.callMethod();

当织入器按照Pointcut的规定，将Advice织入到目标对象之后，从任何其他地方调用method1()是不会触发Advice所包含的横切逻辑的执行。只有ControlFlowPointcut规定的类内部调用目标对象的method1，才会触发Adivce中横切逻辑的执行。

如果在ControlFlowPointcut的构造方法中单独指定Class类型的参数，那么ControlFlowPointcut将尝试匹配指定的Class中声明的所有方法，跟目标对象的Joinpoint处的方法流程组合。所以，如果只是想完成“TargetCaller.callMethod()调用TargetObject.method1()”这样的流程匹配，而忽略TargetCaller中其他方法与TargetObject中方法的Control Flow匹配，可以同时在构造ControlFlowPointcut的时候，传入第二个参数，即调用方法的名称：

ControlFlowPointcut pointcut = **new** ControlFlowPointcut(TargetCaller.**class**,"callMethod");

因为ControlFlowPointcut类型的Pointcut需要在运行期间检查程序的调用栈，而且每次方法调用都需要检查，所以性能比较差。

1. **自定义Pointcut**

要自定义Pointcut，只需要继承Spring AOP提供的相应的抽象父类，然后实现或者覆写相应方法逻辑即可。前面说过，Spring AOP的Pointcut类型可以划分为StaticMethodMatcherPointcut和DynamicMethodMatcherPointcut两类。要实现自定义Pointcut，通常在这两个抽象类的基础上实现相应子类即可。

1. **自定义StaticMethodMatcherPointcut**

StaticMethodMatcherPointcut根据自身语义，为其子类提供了如下几个方面的默认实现。

1. 默认所有StaticMethodMatcherPointcut的子类的ClassFilter均为ClassFilter.TRUE，即忽略类的类型匹配。如果具体子类需要对目标对象的类型做进一步限制，可以通过public void setClassFilter(ClassFilter classFilter)方法设置相应的ClassFilter实现。
2. 因为是StaticMethodMatcherPointcut，所以其MethodMatcher的isRuntime方法返回false，同时三个餐宿的matches方法抛出UnsupportedOperationException异常，以表示该方法不应该被调用到。

最终，我们需要做的就是实现两个参数的matches方法了。

例如，我们想要提供一个Pointcut实现，捕捉系统里数据访问层的数据访问对象中的查询方法所在的Joinpoint，那么可以实现一个StaticMethodMatcherPointcut如下：

**public** **class** QueryMethodPointcut **extends** StaticMethodMatcherPointcut{

**public** **boolean** matches(Method method, Class clazz){

**return** method.getName().startsWith("get") && clazz.getPackage().getName().startsWith("...dao");

}

}

1. **自定义DynamicMethodMatcherPointcut**

DynamicMethodMatcherPointcut也为其子类提供了部分便利。

1. getClassFilter()方法返回ClassFilter.TRUE，如果需要对特定的目标对象类型进行限定，子类只要覆写这个方法即可。
2. 对应的MethodMatcher的isRuntime总是返回true，同时StaticMethodMatcherPointcut提供了两个参数的matches方法的实现，默认直接返回true。

要实现自定义DynamicMethodMatcherPointcut，通常情况下只需要实现三个参数的matches方法逻辑即可。

**public** **class** PKeySpecificQueryMethodPointcut **extends** DynamicMethodMatcherPointcut{

**public** **boolean** matches(Method method, Class clazz, Object[] args){

**if**(method.getName().startsWith("get") && clazz.getPackage().getName().startsWith("...dao")){

**if**(!ArrayUtils.isEmpty(args)){

**return** StringUtils.equals("12345", args[0].toString());

}

}

**return** **false**;

}

}

也可以覆写两个参数的matches方法，这样不用每次都得到三个参数的matches方法执行的时候才检查所有的条件。

#### Spring AOP中的Advice

Advice实现了将被织入到Pointcut规定的Joinpoint处的横切逻辑。在Spring中，Advice按照其自身实例能否在目标对象类的所有实例中共享这一标准，可以划分为两大类，即per-class类型的Advice和per-instance类型的Advice。

1. **per-class类型的Advice**

per-class类型的Advice是指，该类型的Advice的实例可以在目标对象类的所有实例之间共享。这种类型的Advice通常只是提供方法拦截的功能，不会为目标对象类保存任何状态或者添加新的特性。

1. **Before Advice**

Before Advice所实现的横切逻辑在相应的Joinpoint之前执行，在Before Advice执行完成之后，程序执行流程将从Joinpoint处继续执行，所以Before Advice通常不会打断程序的执行流程。但是如果有必要，也可以通过抛出相应的异常的形式中断程序流程。

要在Spring中实现Before Advice，我们通常只需要实现org.springframework.aop.MethodBeforeAdvice接口即可。该接口定义如下：

**public** **interface** MethodBeforeAdvice **extends** BeforeAdvice{

**void** before(Method method, Object[] args, Object target)**throws** Throwable;

}

我们可以使用Before Advice进行整个系统的某些资源初始化或者其他一些准备性工作。当然其应用场景并非仅限于此。

例如，系统需要在文件系统的指定位置生成一些数据文件，创建之前，我们需要首先检查这些指定位置是否粗在，不存在需要创建他们。为了避免不必要的代码散落，我们可以为系统中相应目标提供一个Before Advice，对文件系统的指定路径进行统一的检查或者初始化。

**public** **interface** ResourceSetupBeforeAdvice **implements** MethodBeforeAdvice{

**private** Resource ***resource***;

**public** ResourceSetupBeforeAdvice(Resource resource){

**this**.resource = resource;

}

**public** **void** before(Method method, Object[] args,Object target) **throws** Throwable{

**if**(!resource.exists()){

FileUtils.forceMkdir(resource.getFile());

}

}

}

1. **Throws Advice**

Spring中以接口定义org.springframework.aop.ThrowsAdvice对应通常AOP概念中的AfterThrowingAdvice。虽然该接口没有定义任何方法，但是在实现相应的ThrowsAdvice的时候，我们定义的方法需要遵循如下规则：

void afterThrowing([Method, args, target], ThrowableSubclass);

其中，[]中的三个参数可以省略。我们可以根据将要拦截的Throwable的不同类型，在同一个ThrowsAdvice中实现多个afterThrowing方法。框架将会使用Java反射机制来调用这些方法。

**public** **class** ExceptionBarrierThrowsAdvice **implements** ThrowsAdvice{

**public** **void** afterThrowing(Throwable t) {

//普通异常处理逻辑

}

**public** **void** afterThrowing(RuntimeException e) {

//运行时异常处理逻辑

}

**public** **void** afterThrowing(Method m,Object[] agrs,Object target,ApplicationException e) {

//处理应用程序发生的异常

}

...

}

ThrowsAdvice通常用于对系统中特定的异常情况进行监控，以统一的方式对所发生的异常进行处理。

例如：我们可以对系统中的运行时异常进行监控，一旦捕捉到 异常，需要以某种方式通知系统监控人员。

**public** **class** ExceptionBarrierThrowsAdvice **implements** ThrowsAdvice{

**private** JavaMailSender mailSender;

**private** String[] receiptions;

**public** **void** afterThrowing(Method m,Object[] agrs,Object target,RuntimeException e) {

**final** String exceptionMessage = ExceptionUtils.getFullStackTrace(e);

getMailSender().send(**new** MimeMessagePreparator() {

**public** **void** prepare(MimeMessage message) **throws** Exception{

MimeMessageHelper helper = **new** MimeMessageHelper(message);

helper.setSubject("...");

helper.setTo(getReceiptions());

helper.setText(exceptionMessage);

}

});

}

**public** JavaMailSender getMaileSender() {

**return** mailSender;

}

**public** **void** setMailSender(JavaMailSender mailSender) {

**this**.mailSender = mailSender;

}

**public** String[] getReceiptions() {

**return** receiptions;

}

**public** **void** setReceiptions(String[] receiptions) {

**this**.receiptions = receiptions;

}

}

1. **AfterReturningAdvice**

通过Spring中的AfterReturningAdvice，我们可以访问当前Joinpoint的方法返回值、方法、方法参数以及所在目标对象。org.springframework.aop.AfterReturningAdvice接口定义了Spring的AfterReturningAdvice。代码如下：

**public** **interface** AfterReturningAdvice **extends** AfterAdvice{

**void** afterReturning(Object returnValue, Method method, Object args, Object target) **throws** Throwable;

}

只有方法正常返回的情况下，AfterReturningAdvice才会执行。虽然AfterReturningAdvice可以访问到方法的返回值，但不可以更改返回值。

**public** **class** TaskExecutionAfterReturningAdvice **implements** AfterReturningAdvice{

**private** SqlMapClientTemplate sqlMapClientTemplate;

**public** **void** afterReturning(Object returnValue, Method m, Object[] args, Object target) **throws** Throwable {

Class clazz = target.getClass();

getSqlMapClientTemplate().insert("BATCH.insertTaskStatus", clazz.getName());

}

**public** SqlMapClientTemplate getSqlMapClientTemplate() {

**return** sqlMapClientTemplate;

}

**public** **void** setSqlMapClientTemplate(SqlMapClientTemplate sqlMapClientTemplate) {

**this**.sqlMapClientTemplate = sqlMapClientTemplate;

}

}

1. **Around Advice**

Spring AOP没有提供After(Finally)Advice，使得我们没有一个合适的Advice类型来承载类似于系统资源清理之类的横切逻辑。Spring AOP的AfterReturningAdvice不能更改Joinpoint所在方法的返回值，但是Spring AOP提供了Around Advice。

Spring中没有直接定义对应的Around Advice的实现接口，而是直接采用AOP Alliance的标准接口，即org.aopalliance.intercept.MethodInterceptor。接口定义如下：

**public** **interface** MethodInterceptor **extends** Interceptor{

Object invoke(MethodInvocation invocation) **throws** Throwable;

}

MethodInterceptor作为Around Advice可以说是应用场景很广，之前所说的几种Advice能完成的事情，对于MethodInterceptor都可以实现。

例如：实现简单的检测系统某些方法的执行性能为例来实现MethodInterceptor。

**public** **class** PerformanceMethodInterceptor **implements** MethodInterceptor{

**private** **final** Log logger = LogFactory.getLog(**this**.getClass());

**public** Object invoke(MethodInvocation invocation) **throws** Throwable{

StopWatch watch = **new** StopWatch();

**try** {

watch.start();

**return** invocation.proceed();

}**finally** {

watch.stop();

**if**(logger.isInfoEnabled()) {

logger.info(watch.toString());

}

}

}

}

通过MethodInterceptor的invoke方法的MethodInvocation参数，可以控制对Joinpoint的拦截行为通过调用MethodInvocation的proceed()方法，可以让程序执行继续沿着调用链传播。如果我们在哪一个的MethodInterceptor中没有调用proceed()，那么程序的执行将会在当前的MethodInterceptor处“短路”，Joinpoint上的调用链将被中断，同一Joinpoint上的其他MethodInterceptor的逻辑以及Joinpoint处的方法逻辑不会被执行。

1. **per-instance类型的Advice**

与per-class类型的Advice不同，per-instance类型的Advice不会在目标类所有对象实例之间共享，而是会为不同的实例对象保存他们各自的状态以及相关逻辑。

**Introduction**

Introduction是Spring AOP中唯一的一种per-instance类型的Advice。它可以在不改变目标类定义的情况下，为目标类添加新的属性以及行为。

Spring中，为目标对象添加新的属性和行为必须声明相应的接口以及相应的实现。这样再通过特定的连接器将新的接口定义以及实现类中的逻辑附加到目标对象之上。之后目标对象就拥有了新的状态和行为。这个特定的拦截器就是org.springframework.aop.IntroductionInterceptor。定义如下：

**public** **interface** IntroductionInterceptor **extends** MethodInterceptor,DynamicIntroductionAdvice{

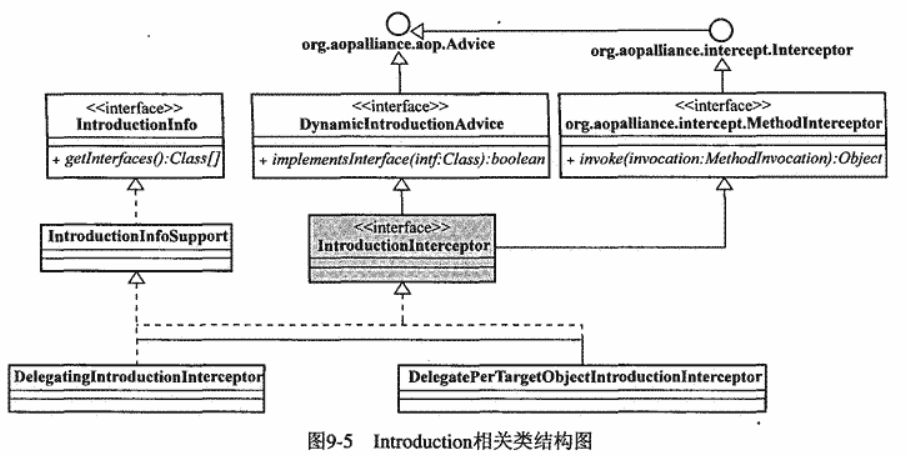
}

**public** **interface** DynamicIntroductionAdvice **extends** Advice{

**boolean** implementsInterface(Class intf);

}

IntroductionInterceptor继承了MethodInterceptor以及DynamicIntroductionAdvice。通过DynamicIntroductionAdvice，我们可以界定当前的IntroductionInterceptor为哪些接口类提供相应的拦截功能。通过MethodInterceptor，IntroductionInterceptor就可以处理新添加的接口上的方法调用了。另外，通常情况下，对于IntroductionInterceptor来说，如果是新增加的接口上的方法调用，不必去调用MethodInterceptor的proceed()方法。



在IntroductionInterceptor的层次继承关系中Introduction类型的Advice拥有两个不同的分支，分别以DynamicIntroductionAdvice为首的动态分支和IntroductionInfo为首的静态可配置分支。从上述代码中可以看出DynamicIntroductionAdvice可以在运行时再去判断当前Introduction可应用到的目标接口类型，而不用预先就设定。而IntroductionInfo类型则完全相反。代码如下：

**public** **interface** IntroductionInfo{

Class[] getInterfaces();

}

实现类必须返回预定的目标接口类型，这样在对IntroductionInfo类型的Introduction进行织入的时候，实际上就不需要指定目标接口类型了，因为它自身就带有这些必要的信息。

要对目标对象进行拦截并添加Introduction逻辑，可以直接扩展IntroductionInterceptor，然后在子类的invoke方法中实现所有的拦截逻辑。不过除非特殊情况下需要直接扩展IntroductionInterceptor，大多数时候，直接使用Spring提供的两个现成的实现类就可以了。

1. **DelegatingIntroductionInterceptor**

DelegatingIntroductionInterceptor不会自己实现将要添加到目标对象上的新的逻辑行为，而是委派(delegate)给其他实现类。

例如：声明一个IDeveloper接口及其相关类：

**public** **interface** IDeveloper{

**void** developSoftware();

}

//实现类

**public** **class** Developer **implements** IDeveloper{

**public** **void** developSoftware() {

System.***out***.println("I am happy with programming.");

}

}

使用DelegatingIntroductionInterceptor为Developer添加新的状态或者行为，可以按照如下步骤进行：

1. 为新的状态和行为定义接口。我们要为Developer添加测试人员的职能，首先需要将需要的只能以接口定义的形式声明。

**public** **interface** ITester{

**boolean** isBusyAsTester();

**void** testSoftware();

}

1. 给出新街口的实现类。接口实现类给出将要添加到目标对象的具体逻辑。当目标对象将要行使新的职能的时候，会通过该实现类寻求帮忙。

**public** **class** Tester **implements** ITester{

**private** **boolean** busyAsTester;

**public** **void** testSoftware() {

System.***out***.println("I will ensure the quality.");

}

**public** **boolean** isBusyAsTester() {

**return** busyAsTester;

}

**public** **void** setBusyAsTester(**boolean** busyAsTester) {

**this**.busyAsTester = busyAsTester;

}

}

我们可以在接口实现类中添加相应的属性甚至辅助方法，就跟实现通常的业务对象一样。

1. 通过DelegatingIntroductionInterceptor进行Introduction的拦截。有了新增加职能的接口定义以及相应的实现类，使用DelegatingIntroductionInterceptor就可以把具体的Introduction拦截委托给具体的是实现类来完。

ITester delegate = **new** Tester();

DelegatingIntroductionInterceptor interceptor = **new** DelegatingIntroductionInterceptor(delegate);

//进行织入

ITester tester = (ITester)weaver.weave(developer).with(interceptor).getProxy();

tester.testSoftware();

1. 虽然DelegatingIntroductionInterceptor是Introduction类型Advice的一个实例，但是却没有实现Introduction作为per-instance型的Advice。实际上DelegatingIntroductionInterceptor会使用它所持有的一个“delegate”接口时狐狸，供同一目标类的所有实例共享使用。
2. 所以如果需要使用严格意义上的Introduction类型Advice的效果，可以使用DelegatePerTargetObjectIntroductionInterceptor。
3. **DelegatePerTargetObjectIntroductionInterceptor**

DelegatePerTargetObjectIntroductionInterceptor会在内部持有一个目标对象与相应Introduction逻辑实现类之间的映射关系。当每个目标对象上的新定义的接口方法被调用的时候，DelegatePerTargetObjectIntroductionInterceptor会拦截这些调用，然后以目标对象实例作为键，到它持有的哪个映射关系中取得对应当前目标对象实例的Introduction实例类实例。如果根据当前目标对象实例没有找到对应的Introduction实例类实例，DelegatePerTargetObjectIntroductionInterceptor将会为其创建一个新的，然后添加到映射关系中。

DelegatePerTargetObjectIntroductionInterceptor与DelegatingIntroductionInterceptor唯一的差别是不需要自己构造delegate接口实例，而只需告知DelegatePerTargetObjectIntroductionInterceptor相应的delegate接口类型和对应实现类的类型。代码如下：

DelegatePerTargetObjectIntroductionInterceptor ***interceptor*** =

nwe DelegatePerTargetObjectIntroductionInterceptor(DelegareImpl.**class**,IDelegate.**class**);

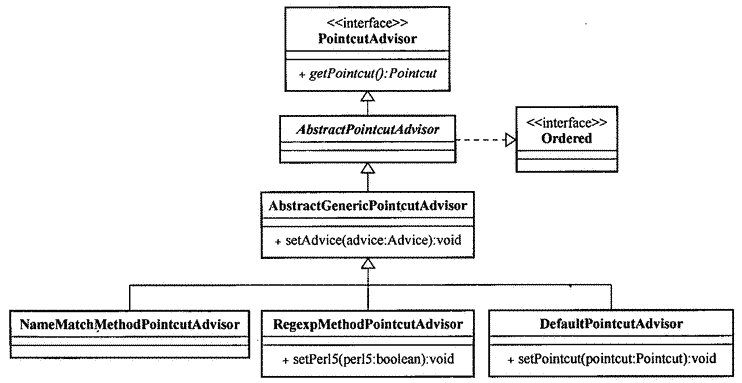
#### Spring AOP中的Aspect

在Spring中Advisor代表了Aspect，但是与正常的Aspect不同，Advisor通常只持有一个Pointcut和一个Advice。而理论上Aspect定义中可以有多个Pointcut和Advice。

将Advisor简单划分可以分为org.springframework.aop.PointcutAdvisor和org.springframework.aop.IntroductionAdvisor分支。

1. **PointcutAdvisor分支**

实际上org.springframework.aop.PointcutAdvisor才是真正的定义一个Pointcut和一个Advice的Advisor，大部分的Advisor实现全部都是PointcutAdvisor的子类或者子接口来实现。



1. **DefaultPointcutAdvisor**

DefaultPointcutAdvisor是最通用的PointcutAdvisot实现。除了不能为其指定Introduction类型的Advice之外，剩下的任何类型的Pointcut、任何类型的Advice都可以通过DefaultPointcutAdvisor来使用。我们可以在构造DefaultPointcutAdvisor的时候就明确指定属于当前DefaultPointcutAdvisor实例的Pointcut和Advice，也可以在DefaultPointcutAdvisor市里构造完成后，再通过setPointcut以及setAdivce方法设置相应的Pointcut和Adivce属性。

Pointcut pointcut = ...; //任何Pointcut类型

Advice advice = ...; //除了Introduction类型外的人格Advice类型

DefaultPointcutAdvisor ***advisor*** = **new** DefaultPointcutAdvisor(***pointcut***,***advice***);

//或者

DefaultPointcutAdvisor advisor = **new** DefaultPointcutAdvisor(***advice***);

advisor.setPointcut(pointcut);

//或者

DefaultPointcutAdvisor advisor = **new** DefaultPointcutAdvisor();

advisor.setPointcut(pointcut);

advisor.setAdvice(advice);

虽然这样可以实现，但是一般情况下不会这么使用，而是将bean通过IoC容器来管理，公共IoC容器来注册和使用Spring AOP的各种概念实体。

<bean id=*"pointcut"* class=*"..."*>

....

</bean>

<bean id=*"advice"* class=*"..."*>

....

</bean>

<bean id=*"advisor"* class=*"org.springframework.aop.support.DefaultPointcutAdvisor"*>

<property name=*"pointcut"* ref=*"pointcut"* />

<property name=*"advice"* ref=*"advice"* />

</bean>

1. **NameMatchMethodPointcutAdvisor**

NameMatchMethodPointcutAdvisor是细化后的DefaultPointcutAdvisor，它限定了自身可以使用的Pointcut类型为NameMatchMethodPointcut，并且外部不可以更改。不过对于使用的Advice来说，除了Introduction，其他人格类型的Advice都可以使用。

NameMatchMethodPointcutAdvisor内部持有一个NameMatchMethodPointcut类型的Pointcut实例。当通过NameMatchMethodPointcutAdvisor公开的setMappedName和setMappendNames方法设置将被拦截的方法名称的时候，实际上是在操作NameMatchMethodPointcutAdvisor所持有的这个NameMatchMethodPointcut实例。

NameMatchMethodPointcutAdvisor的使用通过编码方法或者通过IoC容器都可以实现。

编码方式：

Advice advice = ...; //除了Introduction类型外的任何Advice类型

NameMatchMethodPointcutAdvisor ***advisor*** = **new** NameMatchMethodPointcutAdvisor(***advice***);

advisor.setMappendName("methodName2Intercept");

//或者

NameMatchMethodPointcutAdvisor advisor = **new** NameMatchMethodPointcutAdvisor(***advice***);

advisor.setMappendNames(**new** String[]{"method1","method2"});

通过IoC容器的方式：

<bean id=*"advice"* class=*"..."*>

....

</bean>

<bean id=*"advisor"* class=*"org.springframework.aop.support.NameMatchMethodPointcutAdvisor"*>

<property name=*"advice"* ref=*"advice"* />

<property name=*"mappedNames"*>

<list>

<value>method1</value>

<value>method2</value>

</list>

</property>

</bean>

1. **RegexpMethodPointcutAdvisor**

RegexpMethodPointcutAdvisor也限定了自身可以使用的Pointcut的类型，只能通过正则表达式为其设置相应的Pointcut。

RegexpMethodPointcutAdvisor自身内部持有一个AbstractRegexpMethodPointcut的实例。AbstractRegexpMethodPointcut有两个实现类：Per15RegexpMethodPointcut和JdkRegexpMethodPointcut。默认，RegexpMethodPointcutAdvisor会使用JdkRegexpMethodPointcut。如果要强制使用Per15RegexpMethodPointcut，可以通过setPer15(boolean)实现。

RegexpMethodPointcutAdvisor提供了许多构造方法，我们可以在构造时指定Pointcut的正则表达式匹配模式以及相应的Advice，也可以构造完成后再指定。在使用上与前两个并没太大区别。

<bean id=*"advice"* class=*"..."*>

....

</bean>

<bean id=*"advisor"* class=*"org.springframework.aop.support.RegexpMethodPointcutAdvisor"*>

<property name=*"advice"* ref=*"advice"* />

<property name=*"pattern"*>

<value>cn\.spring21\..\*\.methodNamePattern</value>

</property>

<property name=*"per15"*>

<value>false</value>

</property>

</bean>

1. **DefaultBeanFactoryPointcutAdvisor**

DefaultBeanFactoryPointcutAdvisor是使用比较少的一个Advisor实现，因为自身绑定到了BeanFactory，所以要使用DefaultBeanFactoryPointcutAdvisor必须将应用绑定到Spring的IoC容器了。

DefaultBeanFactoryPointcutAdvisor的作用是，可以通过容器中的Advice注册的beanName来关联对应的Advice。只有当对应的Pointcut匹配成功之后，才去实例化对应的Advice，减少了容器启动初始化Advisor和Advice之间的耦合性。

使用DefaultBeanFactoryPointcutAdvisor，通常需要在容器的配置文件中进行如下配置：

<bean id=*"advice"* class=*"..."*>

....

</bean>

<bean id=*"pointcut"* class=*"org.springframework.aop.support.NameMatchMethodPointcut"*>

<property name=*"mappedName"* value=*"doSth"* />

</bean>

<bean id=*"advisor"* class=*"org.springframework.aop.support.DefaultBeanFactoryPointcutAdvisor"*>

<property name=*"pointcut"* ref=*"pointcut"* />

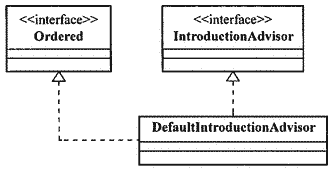
<property name=*"adviceBeanName"* ref=*"advice"* />

</bean>

1. **IntroductionAdvisor分支**

IntroductionAdvisor与PointcutAdvisor的区别就是IntroductionAdvisor只能应用于类级别的拦截，只能使用Introduction型的Advice。

IntroductionAdvisor的类层次比较简单，只有一个默认实现DefaultIntroductionAdvisor。



<bean id=*"introductionInterceptor"* class=*"org.springframework.aop.support.DelegatingIntroductionInterceptor"*>

<constructor-arg>

<bean class=*"...DelegateImpl"*>

</bean>

</constructor-arg>

</bean>

<bean id=*"introductionAdvisor"* class=*"org.springframework.aop.support.DefaultIntroductionAdvisor"*>

<constructor-arg ref=*"introductionInterceptor"* />

<constructor-arg>

<value>...IDelegateInterface</value>

</constructor-arg>

</bean>

我们也可以指定Advice以及一个IntroductionInfo对象类来构造DefaultIntroductionAdvisor，因为IntroductionInfo可以提供必要的目标接口类型。

<bean id=*"introductionInterceptor"* class=*"org.springframework.aop.support.DelegatingIntroductionInterceptor"*>

<constructor-arg>

<bean class=*"...DelegateImpl"*>

</bean>

</constructor-arg>

</bean>

<bean id=*"introductionAdvisor"* class=*"org.springframework.aop.support.DefaultIntroductionAdvisor"*>

<constructor-arg index=*"0"* ref=*"introductionInterceptor"* />

<constructor-arg index=*"1"* ref=*"introductionInterceptor"* />

</bean>

1. **Ordered的作用**

系统中只存在单一个横切关注单的情况很少，大多数时候，都会有多个横切关注点需要处理，那么系统实现中就会有多个Advisor存在。当其中的某些Advisor的Pointcut匹配了同一个Joinpoint的时候，就会在这同一个Joinpoint处执行多个Advice的横切逻辑。如果这些Advisor所关联的Advice之间没有很强的优先级依赖关系，那么谁先执行，谁后执行都不会造成任何影响。而一旦这几个需要在同一Joinpoint处执行的Advice逻辑存在优先顺序依赖的话，就需要我们来干预了，否则系统的行为就会偏离我们的预想。

假设有两个Advisor，一个进行权限检查，当检查到当前调用没有权限的时候，抛出相应异常，称为PermissionAuthAdvisor；另一个Advisor使用一个ThrowsAdvice对系统中的所有需要检测的异常进行拦截，称为ExceptionBarrierAdvisor。如果以如下形式声明这两个Advisor，就不会有问题：

<bean id=*"exceptionBarrierAdvisor"* class=*"...ExceptionBarrierAdvisor"*>

...

</bean>

<bean id=*"pemissionAuthAdvisor"* class=*"...PemissionAuthAdvisor"*>

...

</bean>

即使PermissionAuthAdvisor的Advice抛出异常，我们的ExceptionBarrierAdvisor也可以捕获该异常并进行系统内的统一处理。而如果我们像如下这样颠倒它们的声明顺序，则就有问题了：

<bean id=*"pemissionAuthAdvisor"* class=*"...PemissionAuthAdvisor"*>

...

</bean>

<bean id=*"exceptionBarrierAdvisor"* class=*"...ExceptionBarrierAdvisor"*>

...

</bean>

在PermissionAuthAdvisor的Advice抛出异常之后，ExceptionBarrierAdvisor并没有起作用。

Spring在处理同一Joinpoint处的多个Advisor的时候，实际上会按照指定的顺序和优先级来执行它们，顺序号决定优先级，顺序号越小，优先级越高，我们可以从0或1开始指定，因为小于0的顺序号原则上 由Spring AOP框架内部使用。默认情况下，如果我们不明确指定各个Advisor的执行顺序，那么Spring会按照它们声明的顺序来应用它们。

在Spring框架中，我们可以通过让相应的Advisor以及其他顺序紧要的bean实现org.springframework.core.Ordered接口来明确指定相应顺序号。不过各个Advisor实现类，其实已经实现了Ordered接口，我们无需自己实现这个接口，唯一要做的是直接在配置的时候指定顺序号。

<bean id=*"pemissionAuthAdvisor"* class=*"...PemissionAuthAdvisor"*>

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

...

</bean>

<bean id=*"exceptionBarrierAdvisor"* class=*"...ExceptionBarrierAdvisor"*>

<property name=*"order"* value=*"0"*/>

...

</bean>

#### Spring AOP的织入

要进行织入，AspectJ采用ajc编译器作为它的织入器；JBoss AOP使用自定义的ClassLoader作为它的织入器；而在Spring AOP中，使用类org.springframework.aop.framework.ProxyFactory作为织入器。

1. **使用ProxyFactory**

ProxyFactory并非Spring AOP中唯一可用的注入器，而是最基本的一个注入器实现。使用ProxyFactory来进行横切逻辑的织入很简单。Spring AOP是基于代理模式的AOP实现，织入过程完成后，会返回织入了横切逻辑的目标对象的代理对象。ProxyFactory就会返回哪个织入完成的代理对象。

ProxyFactory weaver = **new** ProxyFactory(yourTargetObject);

//或者

//ProxyFactory weaver = new ProxyFactory(yourTargetObject);

//weaver.setTarget(task);

Advisor advisor = ...;

weaver.addAdvisor(advisor);

Object proxyObject = weaver.getProxy();

//现在可以使用proxyObject了

在不同的应用场景下，我们可以指定更多的ProxyFactory的控制实现，以便让ProxyFactory帮我们生成必要的代理对象。Spring AOP在使用代理模式实现AOP的过程中采用了动态代理和CGLIB两种机制，分别对实现了某些接口的目标类和没有实现任何接口的目标类进行代理。所以在使用ProxyFactory对目标类进行代理的时候，会通过ProxyFactory的某些行为控制属性对这两种情况进行区分。

1. **基于接口的代理**

假设定义了一个ITask接口和一个实现类

**public** **interface** ITask{

**void** execute(TaskExecutionContext ctx);

}

**public** **class** MockTask **implements** ITask{

**public** **void** execute(TaskExecutionContext ctx){

System.***out***.println("task executend");

}

}

为了要拦截的目标类，还得有织入到Joinpoint处的横切逻辑，也就是要用到某个Advice实例。

**public** **class** PerformanceMethodInterceptor **implements** MethodInterceptor{

**private** **final** Log logger = LogFactory.getLog(**this**.getClass());

**public** Object invoke(MethodInvocation invocation) **throws** Throwable{

StopWatch watch = **new** StopWatch();

**try** {

watch.start();

**return** invocation.proceed();

}**finally** {

watch.stop();

**if**(logger.isInfoEnabled()) {

logger.info(watch.toString());

}

}

}

}

MockTask实现了ITask接口，要对象这种实现了某些接口的目标类进行代理，可以为ProxyFactory明确指定代理的接口类型。

MockTask task = **new** MockTask();

ProxyFactory weaver = **new** ProxyFactory(task);

weaver.setInterfaces(**new** Class[]{ITask.**class**});

NameMatchMethodPointcutAdvisor advisor = **new** NameMatchMethodPointcutAdvisor();

advisor.setMappedName("execute");

advisor.setAdvice(**new** PerformanceMethodInterceptor());

weaver.addAdvisor(advisor);

ITask proxyObject = (ITask)weaver.getProxy();

proxyObject.execute(**null**);

通过setInterfaces()方法可以明确告知ProxyFactory，我们要对ITask接口类型进行代理。另外我们通过NameMatchMethodPointcutAdvisor来指定Pointcut和相应的Advice。

如果没有其他行为属性的干预，我们还可以不使用setInterfaces()方法明确指定具体的接口类型。默认情况下ProxyFactory只要检测到目标类实现了相应的接口，也会对目标类进行基于接口的代理。

MockTask task = **new** MockTask();

ProxyFactory weaver = **new** ProxyFactory(task);

NameMatchMethodPointcutAdvisor advisor = **new** NameMatchMethodPointcutAdvisor();

advisor.setMappedName("execute");

advisor.setAdvice(**new** PerformanceMethodInterceptor());

weaver.addAdvisor(advisor);

ITask proxyObject = (ITask)weaver.getProxy();

proxyObject.execute(**null**);

如果目标类实现了至少一个接口，不管我们有没有通过setInterfaces()方法来指定要对特定的接口类型进行代理，只要不将ProxyFactory的optimize和proxyTargetClass两个属性的值设置为true，那么ProxyFactory都会按照面向接口进行代理。

1. **基于类的代理**

如果目标类没有实现任何接口，那么ProxyFactory会对目标类进行基于类的代理，即使用CGLIB。

**public** **class** Executable{

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

System.***out***.println("Executable without any Interfaces");

}

}

定义了一个Executable类，使用Executable作为目标对象类，ProxyFactory就会对其进行基于类的代理。

ProxyFactory weaver = **new** ProxyFactory(**new** Executable());

NameMatchMethodPointcutAdvisor advisor = **new** NameMatchMethodPointcutAdvisor();

advisor.setMappedName("execute");

advisor.setAdvice(**new** PerformanceMethodInterceptor());

weaver.addAdvisor(advisor);

Executable proxyObject = (Executable)weaver.getProxy();

proxyObject.execute();

System.***out***.println(proxyObject.getClass());

如果目标对象实现了至少一个接口，我们也可以通过proxyTargetClass属性强制ProxyFactory采用基于类的代理。

ProxyFactory weaver = **new** ProxyFactory(**new** MockTask());

weaver.setProxyTargetClass(**true**);

NameMatchMethodPointcutAdvisor advisor = **new** NameMatchMethodPointcutAdvisor();

advisor.setMappedName("execute");

advisor.setAdvice(**new** PerformanceMethodInterceptor());

weaver.addAdvisor(advisor);

MockTask proxyObject = (MockTask)weaver.getProxy();

proxyObject.execute(**null**);

System.***out***.println(proxyObject.getClass());

除此之外，如果将ProxyFactory的optimize属性设置为true，ProxyFactory也会采用基于类的代理机制。

1. **Introduction的织入**

之所以将Introduction的织入单独列出是因为Introduction型Advice比较特殊：

1. Introduction可以为已经存在的对象类型添加新的行为，只能应用于对象级别的拦截，而不是通常Advice的方法级别的拦截，所以进行Introduction的织入过程中，不需要指定Pointcut，而只需要指定目标接口类型。
2. Spring的Introduction支持只能通过接口定义为当前对象添加新的行为，所以我们需要在织入的时机指定新织入的接口类型。

鉴于以上两点，使用ProxyFactory进行Introduction的织入代码实例如下：

ProxyFactory weaver = **new** ProxyFactory(**new** Developer());

weaver.setInterfaces(**new** Class[]{IDeveloper.**class**,ITester.**class**});

TesterFeatureIntroductionInterceptor advice = **new** TesterFeatureIntroductionInterceptor();

//DefaultIntroductionAdvisor advisor = new DefaultIntroductionAdvisor(advice,advice);

//weaver.addAdvisor(advisor);

Object proxy = weaver.getProxy();

((ITester)proxy).testSoftware();

((IDeveloper)proxy).developSoftware();

如果不使用Advisor而直接为ProxyFactory指定Advice的话，ProxyFactory会在自身内部构建相应的Advisor来使用。因为TesterFeatureIntroductionInterceptor是IntroductionInfo的子类，所以ProxyFactory内部会创建一个默认的DefaultIntroductionAdvisor实例。

对Introduction进行织入，与基于接口的代理形式有点像，但有少许差异。对Introduction进行织入，新添加的接口类型必须要通过setInterfaces()方法指定，而原来的目标对象，是采用基于接口的代理形式还是采用基于类的代理形式，完全可以自由选择。

基于类的代理形对Introduction进行织入：

ProxyFactory weaver = **new** ProxyFactory(**new** Developer());

weaver.setProxyTargetClass(**true**);

weaver.setInterfaces(**new** Class[]{ITester.**class**});

TesterFeatureIntroductionInterceptor advice = **new** TesterFeatureIntroductionInterceptor();

weaver.addAdvice(advice);

//DefaultIntroductionAdvisor advisor = new DefaultIntroductionAdvisor(advice,advice);

//weaver.addAdvisor(advisor);

Object proxy = weaver.getProxy();

((ITester)proxy).testSoftware();

((IDeveloper)proxy).developSoftware();

1. **ProxyFactory的本质**

要了解ProxyFactory，我们得先从它的根说起，即org.springframework.aop.framework.AopProxy，接口定义如下：

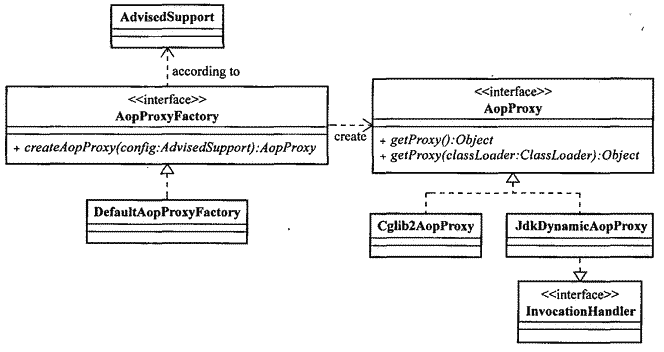
**public** **interface** AopProxy{

Object getProxy();

Object getProxy(ClassLoader classLoader);

}

Spring AOP框架内使用AopProxy对使用的不同的代理实现机制进行了适度的抽象，针对不同的代理实现机制提供相应的AopProxy子类实现。目前Spring AOP框架内提供了针对JDK的动态代理和CGLIB两种机制的AopProxy实现。



当前AopProxy有Cglib2AopProxy和JdkDynamicAopProxy两种实现。因为动态代理需要通过InvocationHandler提供调用拦截，所以JdkDynamicAopProxy同时实现了InvocationHandler接口。不同AopProxy实现的实例化过程采用工厂模式（抽象工厂模式）进行封装。即通过org.springframework.aop.framework.AopProxyFactory进行。AopProxyFactory接口定义如下：

**public** **interface** AopProxyFactory{

AopProxy createAopProxy(AdvisedSuppor config) **throws** AopConfigException;

}

AopProxyFactory根据传入的AdvisedSuppor实例提供的相关信息，来决定生成什么类型的AopProxy。不过具体工作会转交给AopProxyFactory的具体实现类org.springframework.aop.framework.DefaultAopProxyFactory。DefaultAopProxyFactory的实现逻辑很简单，如以下伪代码：

**if**(config.isOptimize() || config.isProxyTargetClass() || hasNoUserSuppliedProxyInterfaces(config)){

//创建Cglib2AopProxy实例，并返回；

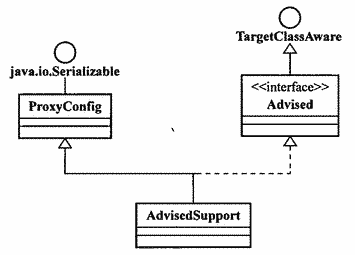
}**else**{

//创建JdkDynamicAopProxy实例，并返回;

}

AopProxyFactory需要根据createAopProxy方法传入的AdvisedSuppor实例信息来构建相应的AopProxy。

AdvisedSuppor其实就是一个生成代理对象所需要的信息的载体，该类相关的类层次如下：



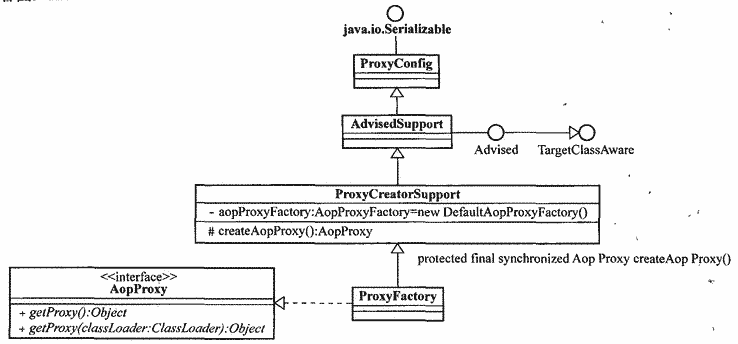
AdvisedSuppor所承载的信息可以划分为两类，一类以org.springframework.aop.framework.ProxyConfig为首记载生成代理对象的控制信息；一类以org.springframework.aop.framework.Advised为首记载生成代理对象所需要的必要信息，如相关目标类、Advice、Advisor等。

ProxyConfig其实就是一个普通的JavaBean，它定义了5个boolean类型的属性，分别控制在生产代理对象的时候应该采用那些行为措施：

1. proxyTargetClass：如果proxyTargetClass属性设置为true，则ProxyFactory将会使用CGLIB对目标对象进行代理。默认值为false
2. optimize：该属性主要用于告知代理对象是否需要采取进一步的优化措施，如代理对象生成之后，即使为其添加或者移除了相应的Advice，代理对象也可以忽略这种变动。另外，该属性为true时，ProxyFactory将会使用CGLIB对目标对象进行代理。默认情况为false。
3. opaque：该属性用于控制生成的代理对象是否可以强制转型为Advised，默认值为false，表示任何生成的代理对象都可以强制转型为Advised，可以通过Advised查询代理对象的一些状态。
4. exposeProxy：设置exposeProxy可以让Spring AOP框架在生成代理对象时，将当前代理对象绑定到ThreadLocal。如果目标对象需要访问当前代理对象，可以通过AopContext.currentProxy()取得。默认值为false。
5. frozen：如果属性设置为true，那么一旦针对代理对象生成的各项信息配置完成，则不允许更改。比如ProxyFactory设置完毕，且fronzen为true，则不能对象Advice进行任何变动，这样可以优化代理对生成的性能。默认为false。

要生成代理对象，只有ProxyConfig提供的控制信息还不够，还需要生成代理对象的一些具体信息，比如：要针对那些目标类生产代理对象，要为代理对象加入那中横切逻辑等。这些信息可以通过org.springframework.aop.framework.Advised设置或者查询。默认情况下，Spring AOP框架返回的代理对象都可以强制转型为Advised，以查询代理对象的相关信息。也可以使用Advised接口访问相应代理对象所持有的Advisor，进行添加Advisor。移除Advisor等相关动作。即使代理对象已经生成了也可以对其进行这些操作。

AopProxyFactory集AopProxy和AdvisedSuppor于一身，所以可以通过AopProxyFactory设置生成代理对象所需要的相关信息，也可以通过AopProxyFactory取得最终生成的代理对象。



为了重用相关逻辑，Spring AOP框架在实现的时候，将一些公用的逻辑抽取到了org.springframework.aop.framework.ProxyCreatorSupport中，它自身就继承了AdvisedSuppor，所以生成代理对象的必要信息从其自身就可以获取。为了简化子类生成不同类型AopProxy的工作ProxyCreatorSupport内部持有一个AopProxyFactory实例，默认采用的是DefaultAopProxyFactory。

1. **容器中的织入器ProxyFactoryBean**

虽然使用ProxyFactory可以让我们能够独立与Spring的IoC容器之外来使用Spring的AOP支持，但是将Spring AOP与Spring的IoC容器支持相结合，通过结合Spring的IoC容器，可以在容器中对Pointcut和Advice等进行管理，即使它们依赖于其他业务对象，也可以很容器地注入其中。

在IoC容器中，使用org.springframework.aop.framework.ProxyFactoryBean作为织入器，它的使用与ProxyFactory无太大差别。

1. **ProxyFactoryBean的本质**

ProxyFactoryBean本质上是一个用来生产Proxy的FactoryBean。如果容器中某个对象依赖于ProxyFactoryBean，那么它将会使用到ProxyFactoryBean的getObject()方法所返回的代理对象。

要让ProxyFactoryBean的getObject()方法返回相应目标对象类的代理对象很简单。因为ProxyFactoryBean继承了与ProxyFactory共有的父类ProxyCreatorSupport，而ProxyCreatorSupport基本上已经把要做的事情（如设置目标对象、配置其他部件、生成对应的AopProxy等）全部完成了。我们只需要在ProxyFactoryBean的getObject()方法中通过父类的createAopProxy()取得相应的AopProxy，然后“return AopProxy.getProxy()”即可。

getObject方法的定义代码如下：

**public** Object getObject()**throws** BeansException{

initializeAdvisorChain();

**if**(isSingleton()){

**return** getSingletonInstance();

}**else**{

**if**(**this**.targetName == **null**){

logger.warn("Using non-singleton proxies with singleton targets is often undesirable Enable prototype proxies by setting the 'targetName' property");

}

**return** newPrototypeInstance();

}

}

FactoryBean定义中要求标明返回的对象是以singleton的scope返回，还是以prototype的scope返回。所以得针对这两种情况分别返回不同的代理对象，以满足FactoryBean的isSingleton()方法的语义

如果将ProxyFactoryBean的singleton属性设置为true，则ProxyFactoryBean在第一次生成代理对象之后，会通过内部实例变量singletonInstance（Object类型）缓存生成的代理对象。之后所有的请求将会发那会这一缓存实例，从而满足singleton的语义。如果将ProxyFactoryBean的singleton属性设置为false，那么ProxyFactoryBean每次都会重新检测各项设置，并为当前调用准备一套新的环境，然后再根据最新的环境数据，返回一个新的代理对象。

1. **ProxyFactoryBean的使用**

ProxyFactoryBean继承了父类的属性外还定义了自有的属性。

1. proxyInterfaces：如果采用基于接口的代理方式，需要通过该属性来配置相应的接口类型。也可以通过配置元素<list>来指定一个或者多个接口类型。
2. interceptorNames：通过该属性可以指定多个将要织入到目标对象的Advice、拦截器以及Advisor。也可以通过配置元素<list>添加需要的拦截器名称。
3. singleton：通过该属性来指定每次getObject调用是返回同一个代理对象，还是返回一个新的。

# Spring访问数据库

# Spring事务

## Spring事务的架构

Spring的事务框架将开发过程中事务管理相关的关注点进行适当的分离，并对这些关注点进行合理的抽象，最终打造了一套使用方便却功能强大的事务管理“利器”。通过Spring的事务框架，我们可以按照同一的编程模式来进行事务编程，却不用关心所使用的数据访问技术以及具体要访问什么类型的事务资源。并且，Spring的事务框架与Spring提供的数据访问支持可以紧密结合。最主要的是结合Spring AOP框架，Spring的事务框架为我们带来了原来只有CMT才有的声明式事务管理，无需绑定到任何的引用服务器上。

Spring的事务框架爱涉及理念的基本原则是：让事务管理的关注点与数据访问关注点相分离。

1. 当在业务层使用事务的抽象API进行事务界定的时候，不需要关心事务将要加诸于上的事务资源是什么，对不同的事务资源的管理将由相应的框架实现类来操心。
2. 当在数据访问层对可能参与事务的数据资源进行访问的时候，只需要使用相应的数据访问API进行数据访问，不需要关系当前的事务资源如何参与事务或者是否需要参与事务。这同样将由事务框架类来打理。

Spring的事务抽象包括3个主要接口，即PlatformTransactionManager、TransactionDefinition以及TransactionStatus。PlatformTransactionManager负责界定事务边界，TransactionDefinition负责定义事务相关属性，包括隔离级别、传播行为等。PlatformTransactionManager将参照TransactionDefinition的属性定义来开启相关事务。事务开启之后到事务结束期间的事务状态由TransactionStatus负责，我们可以通过TransactionStatus对事务进行有限的控制。

### TransactionDefinition

Org.springframework.transaction.TransactionDefinition主要定义了以下属性：

1. 事务的隔离（Isolation）级别。
2. 事务的传播行为（Propagation Bchavior）
3. 事务的超时时间（Timeout）
4. 是否为只读（ReadOnly）事务

TransactionDefinition内定义了如下5个常量用于标志可供选择的隔离级别。

1. ISOLATION\_DEFAULT：如果指定隔离级别为ISOLATION\_DEFAULT，则表示使用数据库默认的隔离级别。通常情况下是Read Committed。
2. ISOLATION\_READ\_UNCOMMITTED：对应Read Uncommitted隔离级别，无法避免脏读，可以重复读和幻读。
3. ISOLATION\_READ\_COMMITTED：对应Read Committed隔离级别，可以避免脏读，但无法避免不可重复读和幻读。
4. ISOLATION\_REPEATABLE\_PEAD：对应Repeatable read隔离级别，可以避免脏读和不可重复读，但不能避免幻读。
5. ISOLATION\_SERIALIZABLE：对应Serializable隔离级别，可以避免所有的脏读，不可重复读以及幻读，但并发性效率最低。

事务的传播行为表示整个事务处理过程所跨越的业务对象，将以什么样的行为参与事务。针对事务的传播行为，TransactionDefinition提供了以下几种选择，除了PROPAGATION\_NESTED是Spring特有的外，其他传播行为的语义与CMT基本相同。

1. PROPAGATION\_REQUIRED：如果当前存在一个事务，则加入当前事务。如果不存在任何事务，则创建一个新的事务。总之，要至少保证在一个事务中运行。通常作为默认的事务传播行为。
2. PROPAGATION\_SUPPORTS：如果当前存在一个事务，则加入当前事务。如果当前不存在事务，则直接执行。对于一些查询方法来说，PROPAGATION\_SUPPORTS通常是比较合适的传播行为选择。如果当前方法直接执行，那么不需要事务的支持。如果当前方法被其他方法调用，而其他方法启动了一个事务，使用PROPAGATION\_SUPPORTS可以保证当前方法能够加入当前事务，并洞察当前对数据资源所做的更新。

### PlatformTransactionManager

org.springframework.transaction.PlatformTransactionManager是Spring事务抽象框架的核心接口。它的主要作用是为应用程序提供事务界定的统一方式。

**public** **interface** PlatformTransactionManager{

TransactionStatus getTransaction(TransactionDefinition definition)**throws** TransactionException;

**void** commit(TransactionStatus status)**throws** TransactionException;

**void** rollback(TransactionStatus status)**throws** TransactionException;

}

Spring的事务框架针对不同的数据访问方式以及全局事务场景提供了相应的PlatformTransactionManager实现类。

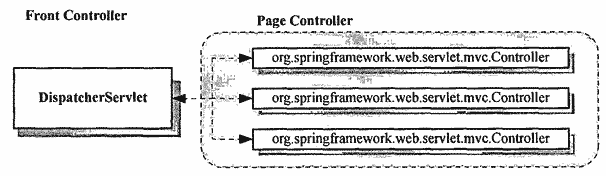
## 使用Spring事务管理

## Spring事务管理扩展

# Spring Web

# Spring MVC

Spring MVC框架的处理控制器的实现策略，与其他的请求驱动的Web框架在总体思路上是相似的，通过引入Front Controller和Page Controller的概念来分离流程控制逻辑与具体的Web请求处理逻辑。Org.springframework.web.servlet.DispatcherServlet就是Spring MVC框架中的Front Controller，它负责接收并处理所有的Web请求，然后将具体的处理逻辑委派给下一级控制器去实现，即org.springframework.web.servlet.mvc.Controller。Controller则对应Page Controller的角色定义。



## DispatcherServlet

### HandlerMapping

DispatcherServlet是整个框架的Front Controller，将它注册到web.xml时，就注定了它要服务于规定的一组Web请求，而不是单独的一个Web请求。注册DispatcherServlet到web.xml配置文件的代码如下：

<servlet>

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

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

<load-on-startup>2</load-on-startup>

</servlet>

<servlet-mapping>

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

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

</servlet-mapping>

对于Web请求与具体的处理类之间的映射匹配，具体的处理方式可能多种多样，完全可以随着应用程序，或者每个具体的Web请求的不同而发生变化：

1. 最常见的就是“掐头去尾”的处理方式，将Web请求的URL路径去除前面的上下文路径和最后的扩展名，取最终剩下的路径信息作为匹配的结果。例如：http://www.nosuchname.com/yourapp/resource.html以resource作为匹配结果。
2. 以Web请求的URL中存在的某个参数的值作为匹配的标准。比如：当发现请求的路径中存在controller这个参数的话，其值就会被作为匹配结果用来调用具体的处理类。

http://www.nosuchname.com/yourapp/resource.htm?controller=yourConroller匹配的结果是yourController

1. 以cookie或者session中的某些信息作为匹配标准。比如：针对某个客户的Web请求，全部转发给一个处理类进行处理。

Spring MVC为了能够灵活地处理映射的匹配，引入了org.springframework.web.servlet.HandlerMapping来专门管理Web请求到具体的处理类之间的映射关系。在Web请求到达DispatcherServlet之后，Dispatcher将寻求具体的HandlerMapping实例，以获取对应当前Web请求的具体处理类，即org.springframework.web.servlet.mvc.Controller

### Controller

Controller本身实现了对应某些具体Web请求的处理逻辑，在通过HandlerMapping查找到当前Web请求对应的Controller的具体实例之后DispatcherServlet即可获得HandlerMapping所返回的结果，并调用org.springframework.web.servlet.mvc.Controller的处理方法来处理当前的Web请求。

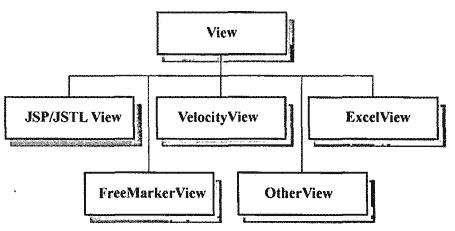
org.springframework.web.servlet.mvc.Controller的处理方法执行完毕之后，将返回一个org.springframework.web.servlet.ModelAndView实例。ModelAndView包含两部分信息：

1. 视图的逻辑名称（或者具体的视图实例）。DispatcherServlet就可以着手视图的渲染工作。
2. 模型数据。视图渲染过程中需要将这些模型数据并入视图的显示中。

### ViewResolver和View

视图技术现在有很多种，Spring提供了一套基于ViewResolver和View接口的Web视图处理抽象层，以屏蔽Web框架在使用不同的Web视图技术时的差异性。

Spring MVC通过引入org.springframework.web.servlet.View接口定义，来统一地抽象视图的生成策略。之后DispatcherServlet只需要根据Spring Controller处理完毕后通过ModelAndView返回的逻辑视图名称查找到具体的View实现，然后委派该具体的View实例类来根据模型数据，输出具体的视图内容即可。Spring的View抽象策略示意图如下：



现在视图模板与模型数据的合并逻辑，以及合并后的视图结构的输出逻辑，全部封装到了相应的View实现类中。DispatcherServlet只需要根据ModelAndView返回的信息，选择具体的View实现类做最终的具体工作即可。不过，DispatcherServlet还需要依赖于某一个org.springframework.web.servlet.ViewResolver来帮它处理逻辑视图名与具体的View实例之间的映射对应关系。ViewResolver将根据ModelAndView中的逻辑视图名查找相应的View实现类，然后将查找的结果返回给DispatcherServlet。DispatcherServlet最终会将ModelAndView中的模型数据提交给返回的View来处理最终的视图渲染工作。

## 第一个Spring MVC应用

一个基于Spring MVC框架的Web应用程序，依然是一个遵循Servlet规范、目录结构和相应的部署描述文件的Web应用程序。Spring MVC框架的Web应用程序在原有的Web应该程序中需添加两个（或者多个）配置文件。并且需要导入相应的jar包。

### 框架部署

首先需要在web.xml配置文件中添加如下代码，将框架注册到Web应用程序中。

<!-- 加载spring的配置文件applicationContext.xml -->

<listener>

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

</listener>

<context-param>

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

<param-value>classpath:com/uc/recruitment/profile/applicationContext.xml</param-value>

</context-param>

<servlet><!-- 配置spring mvc 的配置文件springMvc-servlet.xml-->

<servlet-name>springMvc</servlet-name>

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

</servlet>

<servlet-mapping>

<servlet-name>springMvc</servlet-name>

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

</servlet-mapping>

1. **ContextLoaderListener**

在web.xml中通过<listener>元素添加了一个ServletContextListener的定义，即：org.springframework.web.context.ContextLoaderListener。ContextLoaderListener的职责就是将整个的Web应用程序加载顶层的WebApplicationContext。该顶层WebApplicationContext主要用于提供应用所使用的中间层服务。我们所使用的数据源（DataSource）定义、数据访问对象（DAO）定义、服务对象（Services）定义等，都在改WebApplicationContext中注册。

ContextLoaderListener加载的WebApplicationContext的默认配置文件路径为/WEB/INF/applicationContext.xml。所以我们需要在/WEB/INF/下新建一个名为applicationContext.xml的配置文件。该配置文件符合通用的Spring IoC容器配置文件格式。在实际开发中，允许存在多个配置文件，当存在多个配置文件的时候，可以在web.xml中指定名称为contextConfigLocation的配置参数来实现多个配置文件的整合。

<listener>

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

</listener>

<context-param>

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

<param-value>/WEB-INF/applicationContext.xml,/WEB-INF/applicationContext-module1.xml</param-value>

</context-param>

可以在<param-value>中通过逗号或者空格来分割多个配置文件路径。

ContextLoaderListener或者ContextLoaderServlet加载相应路径下的容器配置文件，并在构建完成相应的顶层WebApplicationContext后，将该顶层WebApplicationContext绑定到ServletContext。如果想获取绑定到ServletContext的WebApplicationContext，可以通过org.springframework.web.context.support.WebApplicationContextUtils类。这样就不需要知道顶层WebApplicationContext绑定到ServletContext的时候使用的key是什么。

WebApplicationContext ***wac*** = WebApplicationContextUtils.getWebApplicationContext(getServletContext());

//或者

WebApplicationContext wac = WebApplicationContextUtils.getRequiredWebApplicationContext(getServletContext());

wac.getBean("serviceBeanName");

了解这一点的主要目的在于，绑定到ServletContext的顶层WebApplicationContext并非只能在基于Spring MVC的Web应用程序中使用。通过ContextLoaderListener或者ContextLoaderServlet将其绑定到ServletContext，任何类型的Web应用程序，只要能够获取ServletContext的引用，就能获取并使用该WebApplicationContext。换句话说，即使当前使用的Web开发框架没有类似于Spring提供的基于IoC容器的中间层业务对象管理支持，通过在web.xml中注册ContextLoaderListener或者ContextLoaderServlet也可以获取这种支持。

1. **DispatcherServlet**

DispatcherServlet是基于Spring MVC框架Web应用程序的Front Controller，它将负责几乎所有对应当前Web应用程序的Web请求的处理。DispatcherServlet使用了一个外部化的配置文件，用来配置Spring MVC框架在处理Web请求过程中所涉及的各个组件，包括HandlerMapping定义、Controller定义、ViewResolver定义等。该外部化的配置文件存在的默认路径也是/WEB-INF/，名称徐哟参照web.xml中定义的DispatcherServlet的<servlet-name>来决定。比如当前定义的DispatcherServlet对应的<servlet-name>为controller，那么默认的配置文件及对应/WEB-INF/controller-servlet.xml。也就是说DispatcherServlet对应的默认配置文件名称将在<servlet-name>的值得基础上后缀-servlet.xml。

<servlet><!-- 配置spring mvc 的配置文件springMvc-servlet.xml-->

<servlet-name>springMvc</servlet-name>

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

</servlet>

<servlet-mapping>

<servlet-name>springMvc</servlet-name>

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

</servlet-mapping>

与ContextLoaderListener一样，基于Spring MVC框架的配置文件在一个引用程序中可以允许有多个，只需要在web.xml配置文件中添加DispatcherServlet的contextConfigLocation初始化参数就可以了。

<servlet>

<servlet-name>controller</servlet-name>

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

<init-param>

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

<param-value>/WEB-INF/controller-servlet.xml,/WEB-INF/module1-servlet.xml,...</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>controller</servlet-name>

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

</servlet-mapping>

我们可以用逗号或者空格来分隔以指定多个配置文件。

### 实现业务逻辑

整体业务流程为：从浏览器中点击链接Web请求将被发送到DispatcherServlet进行处理。DispatcherServlet将寻求相应的HandlerMapping对Web请求进行分析，然后调用匹配结果对应的Controller实例，Controller处理完毕将视图名称和模型数据一同返回，然后DispatcherServlet则借助于相应的ViewResolver，根据返回的视图名称选择相应的视图并显示。

1. 配置web.xml文件

首先我们创建好一个web项目后，先导入所需要的jar包。然后将org.springframework.web.servlet.DispatcherServlet和org.springframework.web.context.ContextLoaderListener通过<servlet>和<listener>元素添加到web.xml配置文件中。另外我们还可以添加相应的Filter以及ServletContextListener以处理字符编码和Log4j初始化等配置内容。

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

<web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns=*"http://xmlns.jcp.org/xml/ns/javaee"* xsi:schemaLocation=*"http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"* id=*"WebApp\_ID"* version=*"3.1"*>

<display-name>simplef</display-name>

<filter>

<filter-name>encodingFilter</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

<init-param>

<param-name>encoding</param-name>

<param-value>UTF-8</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>encodingFilter</filter-name>

<servlet-name>controller</servlet-name>

</filter-mapping>

<listener>

<listener-class>org.springframework.web.util.Log4jConfigListener</listener-class>

</listener>

<listener>

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

</listener>

<servlet>

<servlet-name>controller</servlet-name>

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

<load-on-startup>2</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>controller</servlet-name>

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

</servlet-mapping>

</web-app>

1. 添加spring mvc对应的配置文件

配置完web.xml配置文件后还需要在/WEB-INF/下添加applicationContext.xml和\*\*\*-servlet.xml两个配置文件。可以从spring下载文件包中的sample目录下任何一个web应用程序中复制过来，只不过需要根据命名规则修改相应的文件名称。当然，复制过来的文件内容最好清空，只保留在当前应用中能够通用的配置内容。

1. 开发业务逻辑
2. 定义实体类

**public** **class** TTMRate **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = 2641189625594925925L;

**private** TradeDate frontDate;

**private** String currencyPair;

**private** BigDecimal value;

**public** TradeDate getFrontDate() {

**return** frontDate;

}

**public** **void** setFrontDate(TradeDate frontDate) {

**this**.frontDate = frontDate;

}

**public** String getCurrencyPair() {

**return** currencyPair;

}

**public** **void** setCurrencyPair(String currencyPair) {

**this**.currencyPair = currencyPair;

}

**public** BigDecimal getValue() {

**return** value;

}

**public** **void** setValue(BigDecimal value) {

**this**.value = value;

}

}

为了能够使各种客户端能够获取内容，我们需要规定相应的接口并公开给相应的客户端使用。

**public** **interface** ITTMRateService{

List<TTMRate> getTTMRatesToday();

}

定义接口的实现类

**public** **class** MockTTMRateService **implements** ITTMRateService{

List<TTMRate> getTTMRatesToday(){

//...

}

}

1. 配置applicationContext.xml

整个web应用的中间层服务支出，默认都是通过/WEB-INF/applicationContext.xml注册。所以我们需要将ITTMRateService的实现添加到web应用程序顶层容器的配置文件中。

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

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

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

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

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.xsd*

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

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

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

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

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

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

<bean id=*"ttmRateService"* class=*"cn.spring21.simplefx.service.MockTTMRateService"*></bean>

</beans>

真实环境下，我们可能还需要向该容器配置文件中添加数据源等一系列依赖对象。

1. 提供web请求入口

<a href=*"<c:url value='ttmRateList.do'/>"*>点击</a>

当用户点击该链接之后，浏览器将以http://host:prot/simplefx/ttmRateList.dao的形式向服务器发起web请求。我们在web.xml中将所有以.do结尾的请求模式都映射给了DispatcherServlet来处理。接下来的事情就由DispatcherServlet来接手。

1. 添加HandlerMapping

DispatcherServlet在接收到web请求之后，将寻求相应的HandlerMapping进行web请求到具体的Controller实现的匹配。所以我们需要为DispatcherServlet提供一个HandlerMapping的实现。这里我们暂且使用org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping。它将根据URL与Controller的bean定义的名称进行匹配。现在我们将它添加到DispatcherServlet特定的WebApplicationContext中，即/WEB-INF/controller-servlet.xml。

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

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

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

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

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

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

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

*http://www.springframework.org/schema/security/spring-security-4.2.xsd"*>

<bean id=*"handlerMapping"* class=*"org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"*></bean>

</beans>

Spring mvc默认也是使用BeanNameUrlHandlerMapping进行URL到具体Controller的匹配。总之，我们使用BeanNameUrlHandlerMapping进行URL到具体Controller的匹配之后，BeanNameUrlHandlerMapping将根据http://host:port/simplefx/ttmRateList.do这一URL信息，在当前容器内寻找名称为/ttmRateList.do的Controller定义。

1. 实现对应的Controller

针对当前的web请求，我们需要实现一个Controller来处理它。通常情况下，我们会扩展spring mvc的org.springframework.web.servlet.mvc.AbstractController来实现具体的Contorller。

**public** **class** TTMRateListController **extends** AbstractController{

**private** ITTMRateService ttmRateService;

**private** String viewName;

@Override

**protected** ModelAndView handlerRequestInternal(HttpServletRequest request,HttpServletResponse response) **throws** Exception{

List<TTMRate> ttmRateList = getTtmRateService().getTTMRatesTodat();

ModelAndView mav = **new** ModelAndView(getViewName());

mav.addObject("ttmRates", ttmRateList);

**return** mav;

}

**public** ITTMRateService getTtmRateService() {

**return** ttmRateService;

}

**public** **void** setTtmRateService(ITTMRateService ttmRateService) {

**this**.ttmRateService = ttmRateService;

}

**public** String getViewName() {

**return** viewName;

}

**public** **void** setViewName(String viewName) {

**this**.viewName = viewName;

}

}

只是实现了TTMRateListController，DispatcherServlet并不会知道它的存在，也不会知道如何引用他，所以我们还得将TTMRateListController添加到DispatcherServlet特定的WebApplicationContext中（/WEB-INF/controller-servlet.xml）。

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

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

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

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

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

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

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

*http://www.springframework.org/schema/security/spring-security-4.2.xsd"*>

<bean id=*"handlerMapping"* class=*"org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"*></bean>

<bean name=*"/ttmRateList.do"* class=*".../TTMRateListController"*>

<property name=*"ttmRateService"* ref=*"ttmRateService"*></property>

<property name=*"viewName"* value=*"ttmRateList"*></property>

</bean>

</beans>

这里需要注意的是，TTMRateListController的ttmRateService注入的依赖来自org.springframework.web.context.ContextLoaderListener加载的顶层WebApplicationContext。

1. 添加ViewResolver

TTMRateListController通过ModelAndView返回一个逻辑视图名ttmRateList，而DispatcherServlet要根据这个逻辑视图名查找相应的视图实现，就需要一个ViewResolver来帮它。

spring mvc也为ViewResolver提供了多种实现，我们当前决定使用比较普及的JST/JSTL作为视图技术，所以将org.springframework.web.servlet.view.InternalResourceViewResolver添加到DispatcherServlet特定的WebApplicationContext中。

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

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

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

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

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

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

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

*http://www.springframework.org/schema/security/spring-security-4.2.xsd"*>

<bean id=*"handlerMapping"* class=*"org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"*></bean>

<bean name=*"/ttmRateList.do"* class=*".../TTMRateListController"*>

<property name=*"ttmRateService"* ref=*"ttmRateService"*></property>

<property name=*"viewName"* value=*"ttmRateList"*></property>

</bean>

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

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

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

</bean>

</beans>

我们为InternalResourceViewResolver知道你故意了prefix和suffix两个属性的自定义值。这样当我们的逻辑视图名（viewName）为ttmRateList的时候，InternalResourceViewResolver将去寻找[prefix]+viewName+[suffix]名称的视图模板文件，也就是/WEB-INF/jsp/ttmRateList.jsp。

1. 创建相应视图

现在我们需要创建视图模板文件/WEB-INF/jsp/ttmRateList.jsp

<%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"* pageEncoding=*" UTF-8"*%>

<%@ taglib prefix=*"c"* uri=*"http://java.sun.com/jsp/jstl/core"* %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset= UTF-8"*>

<title>Insert title here</title>

</head>

<body>

<table>

<c:forEach items=*"*${ttmRates}*"* var=*"ttmRate"*>

<tr>

<td><div align=*"center"*>${ttmRate.currencyPair}</div></td>

<td><div align=*"center"*>${ttmRate.value}</div></td>

</tr>

</c:forEach>

</table>

</body>

</html>

## spring mvc详解

### HandlerMapping

HandlerMapping帮助DispatcherServlet进行web请求的URL到具体处理类的匹配。在spring mvc中并不止局限于使用org.springframework.web.servlet.mvc.Controller作为DispatcherServlet的次级控制器来进行具体的web请求的处理。实际上在我们也可以使用其他类型的次级控制器，包括spring mvc提供的除了Controller之外的次级控制器类型，或者第三方web开发框架中的Page Controller组件，所有这些次级控制器类型在spring mvc中都称为Handler。HandlerMapping要处理的也就是web请求到相应Handler之间的映射关系。

HandlerMapping定义很简单，因为大部分实际工作都是由相应的子类来完成。

**public** **interface** HandlerMapping{

String ***PATH\_WITHIN\_HANDLER\_MAPPING\_ATTRIBUTE*** = HandlerMapping.**class**.getName() + ".pathWithinHandlerMapping";

HandlerExecutionChain getHandler(HttpServletRequest request) **throws** Exception;

}

其他的HandlerMapping

spring mvc默认提供了多个HandlerMapping的实现

1. BeanNameUrlHandlerMapping：该实现类是根据URL与Controller的bean定义的名称进行匹配。
2. SimpleUrlHandlerMapping：较之BeanNameUrlHandlerMapping，该实现类进一步解除了请求URL与Handler的beanName之间的耦合，并且支持更灵活的映射表达式。
3. ControllerClassNameHandlerMapping：
4. DefaultAnnotationHandlerMapping：spring2.5之后的spring mvc引入了基于注解的配置方式。
5. **BeanNameUrlHandlerMapping**

BeanNameUrlHandlerMapping实现类是根据URL与Controller的bean定义的名称进行匹配。使用BeanNameUrlHandlerMapping进行web请求到具体Handler的映射管理，需要我们保证视图模板中的请求路径，必须与容器中对应的Handler的beanName一致。

<a href=”requestPath”>Text</a> <bean name=”/requestPath” class=””></bean>

Controller

request

HandlerMapping

1. **SimpleUrlHandlerMapping**

SimpleUrlHandlerMapping实现类对web请求到具体Handler的映射管理比BeanNameUrlHandlerMapping更加的灵活，SimpleUrlHandlerMapping会将请求中的URL与配置文件中所定义的属性值进行匹配，然后再去请求所对应的Controller。

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

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

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

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

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

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

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

*http://www.springframework.org/schema/security/spring-security-4.2.xsd"*>

<bean id=*"handlerMapping"* class=*"org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"*>

<property name=*"mapping"*>

<props>

<prop key=*"ttmRateList.do"*>ttmRateListController</prop>

…

</props>

</property>

</bean>

<bean name=*"ttmRateListController"* class=*".../TTMRateListController"*>

<property name=*"ttmRateService"* ref=*"ttmRateService"*></property>

<property name=*"viewName"* value=*"ttmRateList"*></property>

</bean>

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

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

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

</bean>

</beans>

现在TTMRateListController可以其任何名字，视图中的链接也可以独立提供，只需要通过SimpleUrlHandlerMapping明确指定一下二者的对应关系。SimpleUrlHandlerMapping不对二者做任何限制。

<a href=”requestPath1”>Text</a> <bean name=”/requestPath2” class=””></bean>

request

Controller

HandlerMapping

<a href=”requestPath2”>Text</a> <bean name=”/requestPath1” class=””></bean>

Controller

request

RequestPath1<---->controller2

RequestPath2<---->controller1

使用SimpleUrlHandlerMapping

<bean id=*"handlerMapping"* class=*"org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"*>

<property name=*"mappings"*>

<value>

ttmRateList.do=ttmRateListController

/\*\*/\*list.do=genericListingController

/user\*.do=userManagerController

/module/\*.do=ModuleInOneController

</value>

</property>

</bean>

### HandlerMapping执行顺序

在基于spring mvc的web应用程序中，我们可以为DispatcherServlet提供多个HandlerMapping提供其使用。DispatcherServlet在选用HandlerMapping的过程中，将根据我们所指定的一系列HandlerMapping的优先级进行排序，然后优先使用优先级在前的HandlerMapping。如果当前的HandlerMapping能够返回可用的Handler，DispatcherServlet则使用当前返回的Handler进行web请求的处理，而不再继续询问其他的HandlerMapping。否则DispatcherServlet将继续按照各个HandlerMapping的优先级进行询问，直到获取一个可用的Handler为止。

HandlerMapping的优先级规定遵循spring框架内一贯的Ordered接口所规定的语义。Spring mvc中可用的HandlerMapping实现全都实现了Ordered接口。假设我们优先使用SimpleUrlHandlerMapping进行Handler的映射管理，其次使用BeanNameUrlHandlerMapping，那么可以在DispatcherServlet特定的WebApplicationContext中增加如下代码

<bean id=*"handlerMapping"* class=*"org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"*>

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

<property name=*"mappings"*>

<value>

...

</value>

</property>

</bean>

<bean id=*"defaultHandlerMapping"* class=*"org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"*></bean>

如果不为HandlerMapping明确指定order，那么默认值为Integer.MAX\_VALUE。对应最低优先级。所有拥有order值为1的SimpleUrlHandlerMapping较之BeanNameUrlHandlerMapping。

### Controller

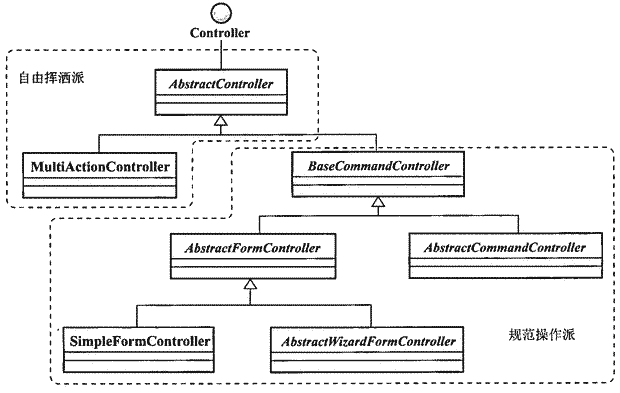
Controller是spring mvc框架支持的用于处理具体web请求的handler类型之一。要实现一个具体的Controller，我们可以直接实现Controller接口：

**public** **interface** Controller{

ModelAndView handleRequest(HttpServletRequest request,HttpServletResponse response) **throws** Exception;

}

但更多时候，我们可能会寻求使用spring mvc提供的更细粒度的Controller框架类。spring mvc提供了一套完善的Controller实现体系，对web处理过程中的所有关注点，如：请求参数的抽取、请求编码的设定、国际化信息的处理、Session数据的管理等底层的细节进行了比较完整的实现。



#### AbstractController

AbstractController是整个Controller继承层次的起源，该类通过模板方法模式帮我们解决了如下几个通用关注点：

1. 管理当前Controller所支持的请求方法类型（GET/POST）
2. 管理页面的缓存设置，即是否允许浏览器缓存当前页面
3. 管理执行流程在绘画上的同步

而我们所要做的，只不过是在AbstractController所公开的handleRequestInternal(request,response)模板方法中实现具体web请求处理过程中的其他逻辑。

虽然直接继承AbstractController实现web处理逻辑没有太多限制，但往往需要关注太多细节，比如参数抽取、数据验证之类。所以对于简单的web请求，可以通过这种方法来处理，如果涉及表单相关的web请求，最好使用BaseCommandController一族的相应子类进行扩展。

#### MultiActionController

对于一组逻辑上相近的web请求来说，比如针对同一对象的CRUD（Create-Read-Update-Delete）操作，或者针对同一对象甚至多个对象的一组查询操作，我们可以将这些web请求交给MultiActionController来统一处理，而不用分别为每个web请求单独实现一个继承AbstractController的处理类。

MultiActionController继承了AbstractController，所以也就拥有了AbstractController所处理的那些通用关注点的能力。除此之外MultiActionController还提供了以下功能：

1. 请求参数到Command对象的绑定。这看起来多少有点多管BaseCommandController闲事的意思。因为正规上说，BaseCommandController才是提供数据绑定功能的基类。不过，BaseCommandController提供的数据绑定和验证更多地服务于其子类的流程控制，而MultiActionController提供的数据绑定能力则比较独立。
2. 通过Validator的数据验证。这个功能以MultiActionController的数据绑定是一起的，一旦启用了Command对象的数据绑定，那么绑定过程中MultiActionController会自动调用我们注册的一系列Validator进行数据验证。当然即使不用数据绑定功能，也同样可以自行决定是否调用相应的Validator。
3. 细化的异常处理方法。可以定义一系列的异常处理方法，用于处理web请求处理过程中所抛出的特定类型的异常。

为了在MultiActionController中处理多个web请求，我们需要定义多个web请求方法，分别对应每个web请求的处理。这些web请求处理方法可以定义在MultiActionController的子类中，也可以定义在某一个将来可以指定给MultiActionController的委派对象内，但web请求处理方法的签名必须符合一定的要求。

(ModelAndView | Map | **void**)methodName(HttpServletRequest request,HttpServletResponse response[,(HttpSession session | Objext command)]);

Web请求处理方法的名称可以去任何有意义的名字，但前两个方法参数是必须的，第三个参数是可选的，可以使HttpSession类型也可以是Object类型。如果是Object类型，则表明对应的是Command对象，那么MultiActionController就会帮我们绑定数据并执行数据验证了。方法的返回值有三种类型：

1. 返回ModelAndView表示正常的web处理方法，后继的ViewResolver和View处理流程，依照之前的DispatcherServlet的流程进行。
2. 返回Map表明只返回了模型数据，而没有返回逻辑视图名。这时，将寻求默认的视图名。这个工作由org.springframework.web.servlet.RequestToViewNameTranslator负责。它能按照某种规则提供一个默认的逻辑视图名。
3. 返回void则表明既没有返回模型数据，也没有返回逻辑视图名。这时我们认为当前web请求处理方法自行处理掉了视图的渲染和输出。
4. 另外spring2.5中可以返回String，代表逻辑视图名，没有相关的模型数据。

**public** **class** GenericCRUDMultiActionController **extends** MultiActionController{

**public** Map list(HttpServletRequest request, HttpServletResponse response){

//...

}

**public** ModelAndView delete(HttpServletRequest request, HttpServletResponse response){

//...

}

**public** ModelAndView update(HttpServletRequest request, HttpServletResponse response, Object command){

//...

}

//...

}

剩下我们需要做的就是，在每个方法内实现当前Controller要完成的功能。

**MethodNameResolver**

Org.springframework.web.servlet.mvc.multiaction.MethodNameResolver的主要工作，就是帮助MultiActionController决定当前web请求应该交给哪个方法来处理。

**public** **interface** MethodNameResolver{

String getHandlerMethodName(HttpServletRequest request) **throws** NoSuchRequestHandlingMethodException;

}

有了MethodNameResolver这一助手，MultiActionController现在只需要询问一声当前需要调用哪个方法即可。

**protected** ModelAndView handleRequestInternal(HttpServletRequest request, HttpServletResponse response) **throws** Exception{

**try**{

String methodName = **this**.methodNameResolver.getHandlerMethodName(request);

**return** invokeNamedMethod(methodName, request, response);

}**catch**(NoSuchRequestHandlingMethodException e){

**return** handleNoSuchRequestHandlingMethod(e, request, response);

}

}

MethodNameResolver为MultiActionController提供了灵活的web请求到对应处理方法的映射策略，包括根据web请求的URL进行映射，或者根据某个参数值进行映射等。在考虑提供自定义的MethodNameResolver之前我们先了解spring mvc提供了哪几个可用的MethodNameResolver实现。

MethodNameResolver本身作为一个策略接口定义，在spring mvc框架内默认提供了如下三种策略实现：

1. **InternalPathMethodNameResolver**

如果没有为MultiActionController明确指定任何MethodNameResolver，那么InternalPathMethodNameResolver将作为默认的MethodNameResolver实现，以进行web请求与具体处理方法之间的映射解析。InternalPathMethodNameResolver将提取URL最后一个（/）之后的部分并去除扩展名，作为要返回的方法名称。

/simplefx/macontroller/listing.do

/simplefx/macontroller/create.do

/simplefx/macontroller/update.do

/simplefx/maconttroller/delete.do

如果将/simplefx/maconttroller/\*通过HandlerMapping指定映射到某个MultiActionController实现来处理，那么对应以上4组URL的web请求将分别由这个MultiActionController的listing、create、update以及delete4个方法来处理。

当然我们可以通过设定InternalPathMethodNameResolver的prefix和suffix来有限的修饰映射行为。

<bean id=*"internalPathMethodNameResolver"* class=*"org.springframework.web.servlet.mvc.multiaction.InternalPathMethodNameResolver"*>

<property name=*"prefix"* value=*"rate\_"* />

</bean>

当使用如上的InternalPathMethodNameResolver定义来映射刚才4组URL的时候，对应的处理方法名应该变成rate\_listing、rate\_create、rate\_update以及rate\_delete。

1. **PropertiesMethodNameResolver**

PropertiesMethodNameResolver与InternalPathMethodNameResolver的唯一不同在于，它们都是基于请求的URL进行映射，但它比InternalPathMethodNameResolver灵活。如果从HandlerMapping与MethodNameResolver都处理映射这一点来看InternalPathMethodNameResolver相当于BeanNameUrlHandlerMapping，而PropertiesMethodNameResolver相当于SimpleUrlHandlerMapping。

通过PropertiesMethodNameResolver可以指定完全匹配的映射关系，或者使用ANT形式的路径匹配模式所表达的映射关系

<bean id=*"propsMethodNameResolver"* class=*"org.springframework.web.servlet.mvc.multiaction.PropertiesMethodNameResolver"*>

<property name=*"mappings"*>

<value>

/listing.do=listingRates

/update.do=updateRate

/create\*.do=addNewRate

</value>

</property>

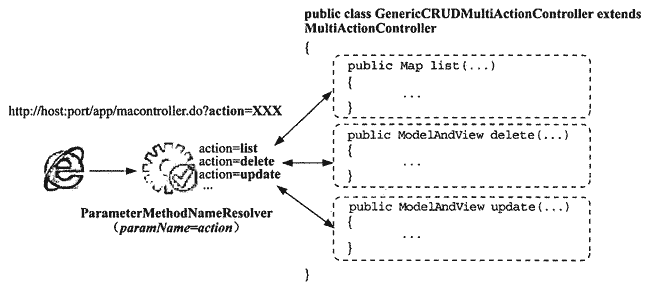
</bean>

MultiActionController中定义的方法不允许重载，所以映射中只需要指定方法名即可，不需要任何参数信息。

1. **ParameterMethodNameResolver**

ParameterMethodNameResolver允许我们根据请求中的某个参数的值作为映射的方法名，也允许我们使用请求中一组参数映射处理方法名称。

1. 根据请求中某个参数的值作为映射后的方法名。在web请求提交之后，我们可以附带一个参数，专门指定由MultiActionController的哪个方法来处理当前请求。



ParameterMethodNameResolver默认检测的参数名称为action，我们也可以通过ParameterMethodNameResolver的setParamName(String)方法来更改默认的参数名称。如下所示，指定paramName为methodName

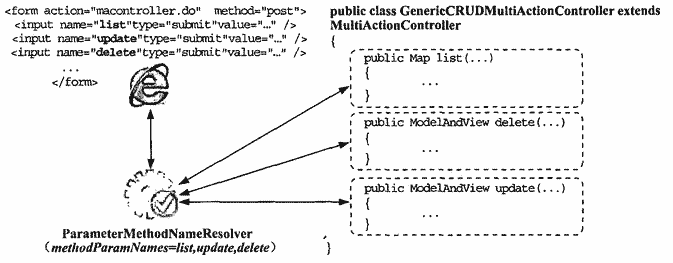
<bean class=*"org.springframework.web.servlet.mvc.multiaction.ParameterMethodNameResolver"*>

<property name=*"paramName"* value=*"methodName"* />

</bean>

HTTP GET形式发送的URL看起来就类似于http://host:port/simplefx/macontroller?methodName=list。

1. 根据请求中的一组参数作为映射后的方法名。在某个页面中存在多种行为选择的时候，可以让每一种行为对应一个参数，这样在web请求提交之后，ParameterMethodNameResolver可以根据提交后的行为参数来调用相应的处理方法。



通过methodParamNames属性为ParameterMethodNameResolver指定一组要检测的参数名。ParameterMethodNameResolver将以指定的一组参数名作为基准，对web请求中的参数进行检测。如果发现寻在其中某个参数，则将当前web请求映射到与参数相同名称的处理方法。通常会为ParameterMethodNameResolver的defaultMethodName属性指定一个默认处理方法。

<bean class=*"org.springframework.web.servlet.mvc.multiaction.ParameterMethodNameResolver"*>

<property name=*"methodParamNames"* value=*"list,update,deleye"* />

<property name=*"defaultMethodName"* value=*"list"* />

</bean>

如果ParameterMethodNameResolver的methodParamNames和paramName两个属性都设置了的话。那就意味着两个策略同时使用。这时以methodParamNames属性为代表的映射策略将被优先考虑。

#### SimpleFormController

AbstractFormController继承了BaseCommandController的自动数据绑定和通过Validator的数据验证功能。在此基础之上发展了一套模板化的form处理流程。从数据的封装，验证，再到处理流程的模板化，整个规范化体系即告建立完成。而SimpleFormController则继承了这些功能。SimpleFormController专门面向单一表单的处理。

1. **数据绑定**

在web应用程序中使用数据绑定的主要好处就是我们再也不用自己通过request.getParameter(String)方法便利获取每个请求参数，然后根据需要转型为自己需要的类型了。spring mvc提供的数据绑定功能帮助我们自动提取HttpServletRequest中的相应参数，然后转型为需要的对象类型。我们唯一需要做的就是为数据绑定提供一个目标对象，这个目标对象在spring中称为Command对象，伺候的web处理逻辑直接同数据绑定的Command对象打交道即可。

对于BaseCommandController及其子类来说，我们可以通过它们的commandClass属性设置数据绑定的目标Command对象类型。

<bean id=*"commandController"* class=*"..AnySubClassOfBaseCommandController"*>

<property name=*"commandClass"* value=*"..Command"* />

</bean>

或者直接在子类的构造方法中直接设定

**public** **class** BindingDemoController **extends** SimpleFormController{

**public** BindingDemoController(){

setCommandClass(Command.**class**);

//进行其他必要设置

}

...

}

1. **数据验证**

spring框架提供的数据验证支持并不只是局限于spring mvc内部使用，从数据验证类所在的包名就能看出来，即org.springframework.validation。我们可以在独立运行的应用程序中使用spring的数据验证功能。

spring数据验证核心类为org.springframework.validation.Validator和org.springframework.validation.Errors，Validator负责实现具体的验证逻辑，而Errors负责承载验证过程中出现的错误信息，二者之间的纽带则是Validator接口定义的主要验证方法validate(target, errors)。

**public** **interface** Validator{

**boolean** supports(Class clazz);

**void** validate(Object target, Errors errors);

}

Validator具体实现类可以在执行验证逻辑的过程中，随时将验证中的错误信息添加到通过方法参数传入的Errors对象内。这样验证逻辑执行完成之后，我们就可通过Errors检索验证结果。

**一个简单的数据验证实现类**

**public** **class** PhoneNumberValidator **implements** Validator{

**public** **boolean** supports(Class clazz){

**return** ClassUtils.*isAssignable*(clazz, PhoneNumber.**class**);

}

**public** **void** validate(Object target, Errors errors){

PhoneNumber phoneNumber = (PhoneNumber)target;

**if**(phoneNumber == **null**){

errors.reject("errorCodes.phoneNumber.is.null");

}

**if**(!StringUtils.isNumeric(phoneNumber.getAreaCode())){

errors.rejectValue("areaCode", "areaCode.not.numeric", "areaCode cann't be empty!");

}

**if**(!StringUtils.isNumeric(phoneNumber.getNumber())){

errors.rejectValue("number", "phoneNumber.not.numeric");

}

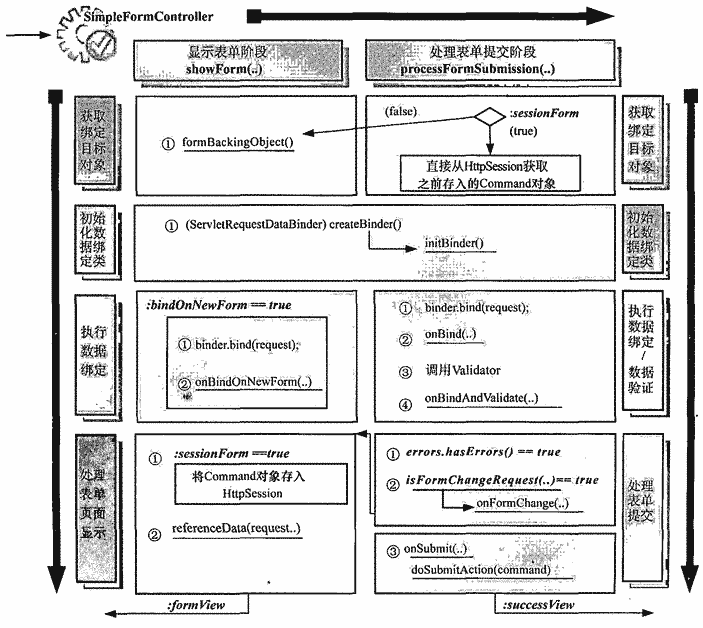
}

}

一个Validator实现类，首先应该通过supports(Class)方法界定自身负责的验证范围，然后才是具体的验证逻辑。具体的验证逻辑通常针对两种数据实体，一种就是被验证对象本身，另一种就是被验证对象的相应属性。如果被验证对象本身都不通过验证，这种错误我们称之为Global Error。这时我们使用Errors的reject(String,..)这组方法，向Errors中添加相应的错误信息。如果被验证对象的某个属性域不能够通过验证，那么我们称这种错误为Field Error。这时，我们要使用Errors的rejectValue(String,String,…)这组方法向Errors中添加相应的错误信息。

1. **深入表单处理流程**

AbstractFormController以模板方法模式从顶层界定了主体上的流程处理逻辑，而处理流程中某些特定的动作则留给其子类SimpleFormController（以及AbstractWizardFormController）实现。但以模板方法模式实现的整个流程控制，并非真的像“模板”所听起来的那样死板，我们可以通过覆写其中的某些方法以添加自定义的行为逻辑，提现了整个流程的可扩展性和灵活性。



我们将AbstractFormController的流程逻辑整合到SimpleFormController的最终流程逻辑中，从而可以看到一个完整的流程实现逻辑（AbstractFormController只是提供了表单处理流程的主体框架）。

在web请求到达SimpleFormController之后，SimpleFormController将首先通过方法isFormSubmission(request)判断当前请求是否为表单提交请求。

**protected** **boolean** isFormSubmission(HttpServletRequest request){

**return** "POST".equals(request.getMethod());

}

只要以POST形式发送的web请求，SimpleFormController将认为当前web请求即为表单提交。我们可以通过覆写该方法改变默认的判断标准。

整个表单处理流程将以isFormSubmission(request)的判定结果为基准，划分为“表单显示阶段”和“处理表单提交阶段”两个逻辑处理区段。如果isFormSubmission(request)返回false，通常表示初次web请求，这时我们需要为用户显示相应的交互表单，这也就是“表单显示阶段”将要做的事情。否则将认为用户已经在表单中编辑完数据，需要处理SimpleFormController将启用“处理表单提交阶段”流程处理逻辑。

**表单显示阶段流程分析**

1. 创建或者获取表单对应的Backing Object。实际上它就是对应绑定表单数据的Command对象。此时formBackingObject()方法将默认通过反射实例化我们为SimpleFormController指定的Command对象实例。

但是某些时候formBackingObject()的默认行为不能符合当前场景需求。比如，如果我们要更新一个实体的信息，那么在为用户显示表单的时候，就应该将要更新的实体的数据加载到表单中，以便用户在原有数据的基础上进行更改。这时，我们就需要覆写formBackingObject()方法，改为自己管理Form Backing Object的初始化。

1. 初始化DataBinder。因为BaseCommandController的所有子类都采用将请求参数绑定到Command对象的处理方式，所以我们就得在执行请求参数到Command对象的数据绑定之前，初始化一个可用的DataBinder实例。

有了一个可用的ServletRequestDataBinder之后，我们可以对其进行定制，比如添加自定义的PropertyEditor以支持某些特殊数据类型的数据绑定，或者排除某些不想绑定的请求参数，这些定制行为可以通过覆写initBinder()方法引入。

**protected** **void** initBinder(HttpServletRequest request, ServletRequestDataBinder binder) **throws** Exception{

PropertyEditor propertyEditor = ...;

binder.registerCustomEditor(SpecialClass.**class**, propertyEditor);

binder.setDisallowedFields(**new** String[]{"parameter1","parameter2"});

}

# Spring ORM