# Spring3.x知识点

## 简介

Spring是分层的Java SE/EE应用一站式的轻量级开源框架，以IoC（Inverse of Control：控制反转）和AOP（Aspect Oriented Programming:面向切面编程）为内核，提供了展现层Spring MVC和持久层SpringJDBC以及业务层事务管理等众多的企业级应用技术。此外，Spring整合了开源世界里众多著名的第三方框架和库类，逐渐称为使用最多的JaveEE企业应用开源框架。

## Spring体系结构

Spring框架由1400多个类组成,整个框架按其所属功能可以划分成5个主要模块,如下图:



## Spring中的IoC

### 3.1 IoC简介

**IoC**指某一接口具体实现类的选择控制权从调用类中移除，转交给第三方决定。由于IoC确实不够开门见山，因此有人提出DI（依赖注入：Dependency Injection）的概念来代替IoC，即让调用类对某一接口实现类的依赖关系有第三方（容器或协作类）注入，以移除调用类对某一接口实现类的依赖。IoC是Spring容器的内核，AOP、声明式事务等功能都以它为基础。

IoC从注入方法来看，主要划分三种类型：**构造函数注入**、**属性注入**和**接口注入**。由于通过接口注入需要额外声明一个接口，增加了类的数目，而且它的效果和属性注入并无本质区别，所以我们不提倡采用这种方式。**Spring支持构造函数注入和属性注入。**

Spring通过一个配置文件描述Bean即Bean之间的依赖关系，利用Java语言的反射功能实例化Bean并建立Bean之间的依赖关系。Spring的IoC容器在完成这些底层工作的基础上，还提供了Bean实例缓存、生命周期管理、Bean实例代理、事件发布等高级服务。

### 3.2 BeanFactory介绍

Bean工厂（com.springframework.beans.factory.**BeanFactory**）是Spring框架最核心的接口，它提供了高级IoC的配置机制。BeanFactory中最主要的方法就是getBean(String beanName)，该方法从容器中返回特定名称的Bean。BeanFactory使管理不同类型的Java对象成为可能。

BeanFacctory的功能通过其他的接口得到不断的扩展：**ListableBeanFactory**接口中定义了访问容器中Bean基本信息的若干方法，如查看Bean的配置名、查看容器中是否包括某一Bean等方法；**HierarchicalBeanFactory**接口是父子级联IoC容器的接口，子容器可以通过该接口方法访问父容器；**ConfigurableBeanFactory**接口增强了IoC容器的可定制性，它定义了设置类装载器、熟悉编辑器、容器初始化后置处理器等方法；**AutowireCapableBeanFactory**接口定义了将容器中的Bean按某种规则（如按名字匹配、按类型匹配等）进行自动装配的方法。**SingletonBeanRegistry**接口定义了允许在运行期间向容器注册单例Bean的方法；**BeanDefinitionRegistry**接口提供了向容器手工注册BeanDefinition对象的方法，而Spring配置文件中的每一个<bean>节点元素在Spring容器中都是通过一个BeanDefinition对象表示的。

Spring为BeanFacotory提供了多种实现，最常用的就是**XmlBeanFacotory**。

### 3.3 ApplicationContext介绍

应用上下文（com.springframework.context.**ApplicationContext**，也是接口）建立在BeanFactory基础之上，提供了更多的面向应用的功能，它提供了国际化支持和框架事件体系，更易于创建实际应用。我们一般称BeanFactory为IoC容器，而称ApplicationContext为应用上下文。

BeanFactory是Spring框架的基础设施，面向Spring本身；ApplicationContext面向使用Srping框架的开发组，几乎所有的应用场合我们都直接使用ApplicationContext而非底层的BeanFactory。ApplicationContext继承了HierarchicalBeanFactory和ListableBeanFactory接口，在此基础上，还通过多个其他的接口扩展了BeanFactory的功能。

ApplicationContext的主要实现类是**ClassPathXmlApplicationContext** 和**FileSystemXmlApplicationContext**，前者默认从类路径加载配置文件，后者默认从文件系统中装载配置文件。在获取了ApplicationContext实例后，就可以像BeanFactory一样调用getBean(beanName)返回Bean了。但ApplicationContext的初始化和BeanFactory有一个重大的区别：**BeanFactory在初始化容器时，并未实例化Bean，直到第一次访问某个Bean时才实例目标Bean；而ApplicationContext则在初始化应用上下文时就实例化所有单实例的Bean。因此ApplicationContext的初始化时间会比BeanFactory稍长一些，不过稍后的调用则没有“第一次惩罚”的问题。**

**WebApplicationContext**是专门为Web应用准备的，它允许从相对于Web根目录的路径中装载配置文件完成初始化工作。从WebApplicationContext中可以获得ServletContext的引用，整个Web应用上下文对象将作为属性放置到ServletContext中，以便Web应用环境可以访问Spring应用上下文。

WebApplicationContext的初始化方式和BeanFactory、ApplicationContext有所区别，因为WebApplicationContext需要ServletContext实例，也就是说它必须在拥有Web容器的前提下才能完成启动的工作。有过Web开发经验的读者都知道可以在

### 3.3 Bean的生命周期和作用域

BeanFactory中Bean的生命周期大概有如下几个阶段：a.在配置文件中定义Bean。一个配置文件中可以定义多个Bean。b.初始化Bean。可以通过在配置文件中通过init-method属性来指定初始化方法或实现InitializingBean接口。c.调用Bean。可以使用BeanWrapper、BeanFactory和ApplicationContext三种方法调用Bean。d.当Spring容器销毁时，Bean也会销毁，可以通过配置destory-method属性或实现DisposebleBean接口来做销毁之前的处理。

Bean作用域：singleton（在Spring IoC容器中仅仅存在一个Bean实例，Bean以单实例方式存在）；prototype（每次从容器中调用getBean()时，都返回一个新的实例）；request（每次HTTP请求都会创建一个新的Bean）；session（同一个HTTP Session共享一个Bean，不同的HTTP Session使用不同的Bean）globalSession（同一个全局Session共享一个Bean）。

一般情况下，无状态或者状态不可变的类适合使用单例模式，不过Spring对此实现了超越。在传统开发中，由于Dao类只有Connection这个非线程安全的变量往往没有采用单例模式。而在Spring环境下，对于所有的Dao类都可以采用单例模式，因为Spring利用AOP和ThreadLocal的功能，对非线程安全的变量（或称状态）进行了特殊的处理，使这些非线程安全的类变成了线程安全的类。

**通用的异常体系及模板类是Spring整合各种五花八门持久化技术的不二法门。**

**统一的异常体系**是整合不同的持久化实现技术的关键，Spring提供了自己的一套和实现技术无关的，面向DAO层语义的异常体系,并通过转换器将不同持久化技术的异常转换成Spring的异常。这些异常都继承于**org.springframework.dao.DataAccessException**这个抽象类，而DataAccessException本身又继承于org.springframework.core.

NestedRuntimeException，NestedRuntimeException异常以嵌套的方式封装了源异常。

Spring为各种支持的持久化技术都提供了简化操作的模板和回调，在回调中编写具体的数据操作逻辑，使用模板执行数据操作，在Spring中，这是典型的数据操作模式。常用的模板类如下: org.springframework.jdbc.core.JdbcTemplate（为JDBC提供）； org.springframework.orm.hibernate3.HibernateTemplate（为Hibernate3.0提供）；org.springframework.orm.ibatis.SqlMapClientTemplate（为iBatis提供）；org.springframework.orm.jpa.JpaTemplate（为JPA提供）等。如果我们直接使用模板类,一般都需要在DAO中定义一个模板对象并提供数据资源，Spring为每一个持久化技术都提供了支持类，支持类中已经为我们完成了这样的功能，我们只需要扩展这些支持类就可以直接编写实际的数据访问逻辑。常用的支持类如下：org.springframework.jdbc.core. JdbcDaoSupport（为JDBC提供）； org.springframework.orm.hibernate3. HibernateDaoSupport（为Hibernate3.0提供）；org.springframework.orm.ibatis.SqlMapClientDaoSupport（为iBatis提供）；org.springframework.orm.jpa. JpaDaoSupport（为JPA提供）；这些类都继承于org.springframework.dao.support.DaoSupport。所有支持类都是abstract的，其目的是希望被继承使用，而非直接使用。

1. 事务的4个特性：原子性、一致性、隔离性和持久性。在这些事务特性中，数据一致性是最终目标，其他的特性都是为达到这个目标的措施、要求或手段。数据库管理系统一般采用重执行日志保证原子性、一致性和持久性，重执行日志记录了数据库变化的每一个动作，数据库在一个事务中执行一部分操作后发生错误退出，数据库即可以根据重执行日志撤销已经执行的操作。对于已经提交的事务，即使数据库奔溃，在重启数据库时也能够根据日志对尚未持久化的数据进行相应的重执行操作。和Java程序采用对象锁机制惊醒线程同步类似，数据库管理系统采用数据库锁机制保证事务的隔离性。当多个事务试图对相同的数据进行操作时，只有持有锁的事务才能操作数据，知道前一个事务完成后，后面的事务才有机会对数据进行操作。Oracle还使用了数据版本的机制，在回滚段为数据的每一个变化都保存一个版本，使数据的更改不影响数据的读取。
2. 多个客户端并发访问数据库，数据库中相同的数据可能同时被多个事务访问，如果没采取必要的隔离措施，就会导致并发问题，破坏数据的完整性。这些问题可以归纳结为5类，3类数据读问题（脏读、不可重复读和幻想读）以及2类数据更新问题（第一类丢失更新和第二类丢失更新）。脏读：A事务读取到B事务尚未提交的更改数据，并在这个数据的基础上操作。如果恰巧B事务回滚，那么A事务读到的数据根本是不被承认的。幻想读：A事务读取B事务提交的新增数据，这时A事务将出现幻想读的问题。幻想读和不可重复读是两个容易混淆的概念，前者是指读到了其他已经提交的事务的新增数据，而后者是指读到了已经提交事务的更改数据（更改或删除）。为了防止不可重复读，只需要对操作的数据添加行级锁，为了防止幻想读，则往往需要添加表级锁（Oracle使用多版本数据的方式实现）。第一类丢失更新：A事务撤销时（即将数据还原到A事务之前），把已经提交的B事务（比A后开始，在A撤销之前提交）的更新数据覆盖了。第二类丢失更新：A事务覆盖B事务（比A先开始，比A先提交）已经提交的数据，造成B事务所做的操作丢失。
3. 虽然数据库为用户提供了锁的DML操作方式，但直接使用锁管理是非常麻烦的，因此数据库为用户提供了自动锁机制。只要用户指定会话的事务隔离级别，数据库就会分析事务中的SQL语句，然后自动为事务操作的数据资源添加上合适的锁。此外数据库还会维护这些锁，当一个资源上的锁数目太多时，自动进行锁升级以提高系统的运行性能。事务的隔离级别和数据的并发性是对立的，此增彼涨。一般来说，使用READ UNCOMMITED隔离级别的数据库拥有最高的并发性和吞吐量，而使用SERIALIZABLE隔离级别的数据库并发性最低。
4. 像Spring DAO为不同的持久化实现提供了模板类一样，Spring事务管理继承了这一风格，也提供了事务模板类TransactionTemplate（线程安全）。通过TransactionTemplate并配合使用事务回调TransactionCallback指定具体的是酒后操作就可以通过编程的方式实现事务管理，而无需关注资源获取、服用、释放、事务同步和异常处理的操作。Spring支持声明式事务，允许通过声明方式，在IoC配置中指定事务的边界和事务属性，Spring自动在指定的事务边界上应用事务属性。
5. Spring事务管理的抽象层主要包括三个接口：PlatformTransactionManager、TransactionDefinition和TransactionStatus，它们都位于org.springframework.transaction包中。TransactionDefinition用于描述事务的隔离级别、超时时间、是否为只读师傅和事务的传播规则等控制事务具体行为的事务属性。这些属性可以通过XML配置或注解描述提供，也可以手工编程方式设置。PlatformTransactionManager根据TransactionDefinition提供事务属性配置信息，创建事务，并用TransactionStatus描述这个激活事务的状态。
6. 在TransactionDefinition接口中，定义了和java.sql.Connection接口中同名的4个隔离级别：TRANSACTION\_READ\_UNCOMMITTED、TRANSACTION\_READ\_COMMITTED、TRANSACTION\_REPEATABLE\_READ、TRANSACTION\_SERIALIZABLE（或TRANSACTION\_NONE）。
7. Spring将事务管理委托给底层具体的持久化实现框架完成。因此Spring为不同的持久化框架提供了PlatformTransactionManager接口的实现类：org.springframework.orm.jpa.JpaTransactionManager（为JPA提供）；org.springframework.orm.hibernate3.HibernateTransactionManager（为Hibernate3.0提供）；org.springframework.jdbc.datasource. DataSourceTransactionManager（为Spring JDBC或iBatis等基于数据源的持久化技术时提供）；org.springframework.transaction.jta.JtaTransactionManager（为具有多个数据源的全局事务提供）；这些事务管理其都是对特定事务实现框架的代理。
8. Spring在TransactionDefinition中定义了7中类型的事务传播行为，它们规定了事务方法和事务方法发生嵌套调用时事务如何进行传播：**PROPAGATION\_****REQUIRED**（如果当前没有事务，就新建一个事务，如果已经存在一个事务中，则加入到这个事务中。这是最常见的选择）；**PROPAGATION\_SUPPORTS**（支持当前事务，如果当前没有事务，就以非事务方式执行）；**PROPAGATION\_MANDATORY**（使用当前事务，如果没有当前事务，就抛出异常）；**PROPAGATION\_REQUIRES\_NEW**（新建事务，如果当前存在事务，把当前事务挂起）；**PROPAGATION\_NOT\_SUPPORTED**（以非事务方式执行操作，如果当前存在事务，就把当前事务挂起）；**PROPAGATION\_NEVER**（以非事务方式执行，如果当前存在事务，则抛出异常）；**PROPAGATION\_NESTED**（如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION\_REQUIRED类似的操作）。
9. Spring的声明式事务管理是通过Spring AOP实现的，通过事务的声明性信息，Spring负责将事务管理增强逻辑动态植入到业务方法相应的连接点中。这些逻辑包括获取线程绑定资源、开始事务、提交/回滚事务、进行异常转换和处理等工作。

## 从源码解析Spring容器是如何启动的

大家在J2EE项目中使用Spring框架时，一定很熟悉web.xml文件中的如下代码段：

<web-app>

<display-name>mzz-spring</display-name>

<context-param>

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

<param-value>classpath:config/applicationContext.xml</param-value>

</context-param>

<listener>

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

</listener>

<!-- SpringMVC -->

<servlet>

<servlet-name>springmvc</servlet-name>

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

<init-param>

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

<param-value>classpath:config/springmvc-servlet.xml</param-value>

</init-param>

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

</servlet>

<servlet-mapping>

<servlet-name>springmvc</servlet-name>

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

</servlet-mapping>

</web-app>

可以看出，在J2EE项目中，Spring容器的启动和org.springframework.web.context.**ContextLoaderListener**、org.springframework.web.servlet.**DispatcherServlet**这两个类有很大关系，下面，我们来详细解析Spring容器在启动时这两个类做了哪些事情。

创建一个上下文加载监听器ContextLoaderListener将会用包含contextClass和contextConfigLocation这两个servlet上下文参数的ServletContext对象来创建一个WebApplicationContext对象。但ContextLoaderListener类本身只负责Spring容器的启动和销毁，请看ContextLoaderListener中的全部代码：

**public** ContextLoaderListener() {

}

**public** ContextLoaderListener(WebApplicationContext context) {

**super**(context);

}

/\*\*

\* Initialize the root web application context.

\*/

@Override

**public** **void** contextInitialized(ServletContextEvent event) {

initWebApplicationContext(event.getServletContext());

}

/\*\*

\* Close the root web application context.

\*/

@Override

**public** **void** contextDestroyed(ServletContextEvent event) {

closeWebApplicationContext(event.getServletContext());

ContextCleanupListener.*cleanupAttributes*(event.getServletContext());

}

实际的启动应用程序上下文的初始化工作由其父类org.springframework.web.context.**ContextLoader**来完成，前面说的contextClass和contextConfigLocation这两个servlet上下文参数也是由ContextLoader来使用的，ContextLoader将用在web.xml中配置的contextClass来定义生成的上下文对象的类型，如果不配置，则默认用org.springframework.web.context.support.**XmlWebApplicationContext**，任何应用上下文的实现都必须实现org.springframework.web.context.**ConfigurableWebApplicationContext**接口，该接口定义如下：

/\*\*

\* Prefix for ApplicationContext ids that refer to context path and/or servlet name.

\*/

String *APPLICATION\_CONTEXT\_ID\_PREFIX* = WebApplicationContext.**class**.getName() + ":";

/\*\*

\* Name of the ServletConfig environment bean in the factory.

\* **@see** javax.servlet.ServletConfig

\*/

String *SERVLET\_CONFIG\_BEAN\_NAME* = "servletConfig";

/\*\*

\* Set the ServletContext for this web application context.

\* <p>Does not cause an initialization of the context: refresh needs to be

\* called after the setting of all configuration properties.

\* **@see** #refresh()

\*/

**void** setServletContext(ServletContext servletContext);

/\*\*

\* Set the ServletConfig for this web application context.

\* Only called for a WebApplicationContext that belongs to a specific Servlet.

\* **@see** #refresh()

\*/

**void** setServletConfig(ServletConfig servletConfig);

/\*\*

\* Return the ServletConfig for this web application context, if any.

\*/

ServletConfig getServletConfig();

/\*\*

\* Set the namespace for this web application context,

\* to be used for building a default context config location.

\* The root web application context does not have a namespace.

\*/

**void** setNamespace(String namespace);

/\*\*

\* Return the namespace for this web application context, if any.

\*/

String getNamespace();

/\*\*

\* Set the config locations for this web application context in init-param style,

\* i.e. with distinct locations separated by commas, semicolons or whitespace.

\* <p>If not set, the implementation is supposed to use a default for the

\* given namespace or the root web application context, as appropriate.

\*/

**void** setConfigLocation(String configLocation);

/\*\*

\* Set the config locations for this web application context.

\* <p>If not set, the implementation is supposed to use a default for the

\* given namespace or the root web application context, as appropriate.

\*/

**void** setConfigLocations(String... configLocations);

/\*\*

\* Return the config locations for this web application context,

\* or {@code null} if none specified.

\*/

String[] getConfigLocations();

另一个参数contextConfigLocation可以定义由任意逗号和空格来分割的多个文件路径，Ant风格（Ant-style）的路径模式也是支持的。如果这个值没有正确定义或没有定义，则对于XmlWebApplicationContext来说，会取默认路径/WEB-INF/applicationContext.xml。**注意，如果定义了多个文件，则后面定义的文件的bean定义会覆盖前面文件的bean定义。**可以利用通过一个额外的XML文件，来故意忽略某些bean定义。

现在我们先看看ContextLoader做了些什么，ContextLoader直接继承于Object，而且没有实现任何接口，看ContextLoader中有哪些重要的成员属性：

/\*\*

\* Map from (thread context) ClassLoader to corresponding 'current' WebApplicationContext.

\*/

**private** **static** **final** Map<ClassLoader, WebApplicationContext> *currentContextPerThread* =

**new** ConcurrentHashMap<ClassLoader, WebApplicationContext>(1);

/\*\*

\* The 'current' WebApplicationContext, if the ContextLoader class is

\* deployed in the web app ClassLoader itself.

\*/

**private** **static** **volatile** WebApplicationContext *currentContext*;

/\*\*

\* The root WebApplicationContext instance that this loader manages.

\*/

**private** WebApplicationContext context;

**private** static final Properties defaultStrategies;

有人会问，不是要一个context就行了吗？干嘛还要静态的currentContext和线程相关的currentContextPerThread，这个以后再补充。

ContextLoader类中有一段静态代码段，用于在构造ContextLoader对象前从properties文件来加载策略（strategy），请看：

**static** {

// Load default strategy implementations from properties file.

// This is currently strictly internal and not meant to be customized

// by application developers.

**try** {

ClassPathResource resource = **new** ClassPathResource(*DEFAULT\_STRATEGIES\_PATH*, ContextLoader.**class**);

*defaultStrategies* = PropertiesLoaderUtils.*loadProperties*(resource);

}

**catch** (IOException ex) {

**throw** **new** IllegalStateException("Could not load 'ContextLoader.properties': " + ex.getMessage());

}

}

这里的properties不是我们项目中的文件，而是Spring包中默认的文件ContextLoader.properties，所以你可以知道DEFAULT\_STRATEGIES\_PATH的值为ContextLoader.properties，ContextLoader.properties文件在spring-web的org.springframework.web.context包中，对，和ContextLoader在同一个路径下，它里面就一行内容：org.springframework.web.context.WebApplicationContext=org.springframework.web.context.support.XmlWebApplicationContext，这行内容是当你没有在web.xml中配置contextClass参数时，将采取默认策略，创建XmlWebApplicationContext作为应用上下文对象。

由于ContextLoaderListener实现了javax.servlet.ServletContextListener接口，所以Web容器在创建完ContextLoaderListener后会调用其contextInitialized方法：

/\*\*

\* Initialize the root web application context.

\*/

@Override

**public** **void** contextInitialized(ServletContextEvent event) {

initWebApplicationContext(event.getServletContext());

}

可见，它是直接去调用父类ContextLoader中的initWebApplicationContext方法，initWebApplicationContext将用Web容器传过来的ServletContext来构建WebApplicationContext对象。我们来看下这个方法的实现：

**public** WebApplicationContext initWebApplicationContext(ServletContext servletContext) {

**if** (servletContext.getAttribute(WebApplicationContext.*ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE*) != **null**) {

**throw** **new** IllegalStateException(

"Cannot initialize context because there is already a root application context present - " +

"check whether you have multiple ContextLoader\* definitions in your web.xml!");

}

Log logger = LogFactory.*getLog*(ContextLoader.**class**);

servletContext.log("Initializing Spring root WebApplicationContext");

**if** (logger.isInfoEnabled()) {

logger.info("Root WebApplicationContext: initialization started");

}

**long** startTime = System.*currentTimeMillis*();

**try** {

// Store context in local instance variable, to guarantee that

// it is available on ServletContext shutdown.

**if** (**this**.context == **null**) {

**this**.context = createWebApplicationContext(servletContext);

}

**if** (**this**.context **instanceof** ConfigurableWebApplicationContext) {

ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext) **this**.context;

**if** (!cwac.isActive()) {

// The context has not yet been refreshed -> provide services such as

// setting the parent context, setting the application context id, etc

**if** (cwac.getParent() == **null**) {

// The context instance was injected without an explicit parent ->

// determine parent for root web application context, if any.

ApplicationContext parent = loadParentContext(servletContext);

cwac.setParent(parent);

}

configureAndRefreshWebApplicationContext(cwac, servletContext);

}

}

servletContext.setAttribute(WebApplicationContext.*ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE*, **this**.context);

ClassLoader ccl = Thread.*currentThread*().getContextClassLoader();

**if** (ccl == ContextLoader.**class**.getClassLoader()) {

*currentContext* = **this**.context;

}

**else** **if** (ccl != **null**) {

*currentContextPerThread*.put(ccl, **this**.context);

}

**if** (logger.isDebugEnabled()) {

logger.debug("Published root WebApplicationContext as ServletContext attribute with name [" +

WebApplicationContext.*ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE* + "]");

}

**if** (logger.isInfoEnabled()) {

**long** elapsedTime = System.*currentTimeMillis*() - startTime;

logger.info("Root WebApplicationContext: initialization completed in " + elapsedTime + " ms");

}

**return** **this**.context;

}

**catch** (RuntimeException ex) {

logger.error("Context initialization failed", ex);

servletContext.setAttribute(WebApplicationContext.*ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE*, ex);

**throw** ex;

}

**catch** (Error err) {

logger.error("Context initialization failed", err);

servletContext.setAttribute(WebApplicationContext.*ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE*, err);

**throw** err;

}

}

initWebApplicationContext方法首先就去检查传入的ServletContext对象中有没有设置WebApplicationContext.*ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE*属性，因为从ServletContext对象中构建WebApplicationContext对象成功，就会将该WebApplicationContext对象绑定到ServletContext对象的WebApplicationContext.*ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE*属性中，这样做是因为Spring不希望你在web.xml中配置多个ContextLoader\*。接下来调用的createWebApplicationContext就是去创建一个WebApplicationContext对象，像前面说的，如果你没有给ServletContext中配置contextClass参数，它就会采用在ContextLoader.properties文件中读取到的策略，创建一个XmlWebApplicationContext对象来返回。接着，将看该WebApplicationContext对象是否是激活的，因为是新创建的，所以当前WebApplicationContext容器肯定没被激活，如果没激活，接下来将尝试调用loadParentContext方法将去查找该WebApplicationContext容器的父容器，因为该WebApplicationContext容器是第一个创建的容器，所以不存在父容器，所以loadParentContext方法将返回为空。由于该容器没被激活，所以接着则是initWebApplicationContext方法中的重头戏——bean定义加载，这项光荣的任务将由里面的configureAndRefreshWebApplicationContext方法来完成，bean加载是个比较复杂的过程，这里不细说，**configureAndRefreshWebApplicationContext在bean加载之前会给WebApplicationContext对象设置一个id，该id值为WebApplicationContext.class.getName() + ":" + ServletContext.getContextPath()，如org.springframework.web.context.WebApplicationContext:/mzz-spring。**bean加载完成后，将给该ServletContext对象设置前面说到的WebApplicationContext.*ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE*属性，把刚初始化完的WebApplicationContext容器对象绑定到该ServletContext对象中。接着，如你看到的，如果加载ContextLoader类的类加载器就是当前线程的类加载器，则把context（也就是刚创建的WebApplicationContext）赋值给ClassLoader的静态成员变量currentContext中。一般情况下，是相等的，一般都是Web容器自己实现的类加载器。最后，返回给ContextLoaderListener其context，完成！

这样，ContextLoaderListener对象就被注册到了Web容器的ServletContext的监听器中，其实ContextLoaderListener在Spring框架中最重要的功能就是把加载过bean定义的WebApplicationContext上下文容器对象绑定到了ServletContext中。

接着，我们来看看调度Servlet DispatcherServlet的构建过程和作用，你如果使用SpringMVC或者Spring中与RPC相关的功能，你就需要在web.xml中配置。DispatcherServlet类是org.springframework.web.servlet. **FrameworkServlet**的子类，咱们先看看FrameworkServlet的类签名：

**public** **abstract** **class** FrameworkServlet **extends** HttpServletBean **implements** ApplicationContextAware

由此，你可以看出，如果DispatcherServlet想要在Web容器启动应用程序的时候有所“动作”，它只能借助1.静态代码块，2.默认构造函数，3.HttpServlet中的init方法以上三种方式来参与，先看看DispatcherServlet中的静态代码块：

**static** {

// Load default strategy implementations from properties file.

// This is currently strictly internal and not meant to be customized

// by application developers.

**try** {

ClassPathResource resource = **new** ClassPathResource(DEFAULT\_STRATEGIES\_PATH, DispatcherServlet.**class**);

defaultStrategies = PropertiesLoaderUtils.loadProperties(resource);

}

**catch** (IOException ex) {

**throw** **new** IllegalStateException("Could not load 'DispatcherServlet.properties': " + ex.getMessage());

}

}

是不是很眼熟？对，和ContextLoader中的静态代码块几乎一样，那你已经明白它要做的是什么了，继续看默认构造函数：

**public** DispatcherServlet() {

**super**();

}

那再看看FrameworkServlet中的构造函数：

**public** FrameworkServlet() {

}

一样什么都没做，所以，剩下的，我们只要关心最后一个入口——init方法！

从上面可以看到FrameworkServlet是org.springframework.web.servlet. **HttpServletBean**的子类，HttpServletBean只是对HttpServlet做了简单的包装，我们看看其init方法：

/\*\*

\* Map config parameters onto bean properties of this servlet, and

\* invoke subclass initialization.

\* **@throws** ServletException if bean properties are invalid (or required

\* properties are missing), or if subclass initialization fails.

\*/

@Override

**public** **final** **void** init() **throws** ServletException {

**if** (logger.isDebugEnabled()) {

logger.debug("Initializing servlet '" + getServletName() + "'");

}

// Set bean properties from init parameters.

**try** {

PropertyValues pvs = **new** ServletConfigPropertyValues(getServletConfig(), **this**.requiredProperties);

BeanWrapper bw = PropertyAccessorFactory.forBeanPropertyAccess(**this**);

ResourceLoader resourceLoader = **new** ServletContextResourceLoader(getServletContext());

bw.registerCustomEditor(Resource.**class**, **new** ResourceEditor(resourceLoader, getEnvironment()));

initBeanWrapper(bw);

bw.setPropertyValues(pvs, **true**);

}

**catch** (BeansException ex) {

logger.error("Failed to set bean properties on servlet '" + getServletName() + "'", ex);

**throw** ex;

}

// Let subclasses do whatever initialization they like.

**initServletBean();**

**if** (logger.isDebugEnabled()) {

logger.debug("Servlet '" + getServletName() + "' configured successfully");

}

}

该方法中最终要的就是initServletBean()方法的调用，该方法在是希望HttpServletBean的子类去覆盖的，所以在HttpServletBean中它是空实现。我们来看看FrameworkServlet中的initServletBean方法：

/\*\*

\* Overridden method of {@link HttpServletBean}, invoked after any bean properties

\* have been set. Creates this servlet's WebApplicationContext.

\*/

@Override

**protected** **final** **void** initServletBean() **throws** ServletException {

getServletContext().log("Initializing Spring FrameworkServlet '" + getServletName() + "'");

**if** (**this**.logger.isInfoEnabled()) {

**this**.logger.info("FrameworkServlet '" + getServletName() + "': initialization started");

}

**long** startTime = System.currentTimeMillis();

**try** {

**this.webApplicationContext = initWebApplicationContext();**

**initFrameworkServlet();**

}

**catch** (ServletException ex) {

**this**.logger.error("Context initialization failed", ex);

**throw** ex;

}

**catch** (RuntimeException ex) {

**this**.logger.error("Context initialization failed", ex);

**throw** ex;

}

**if** (**this**.logger.isInfoEnabled()) {

**long** elapsedTime = System.currentTimeMillis() - startTime;

**this**.logger.info("FrameworkServlet '" + getServletName() + "': initialization completed in " +

elapsedTime + " ms");

}

}

里面最重要的就是initWebApplicationContext和initFrameworkServlet方法的调用，initWebApplicationContext方法主要功能是创建WebApplicationContext对象（因为上面ContextLoaderListener已经创建了一个根应用程序上下文对象，所以该对象是根应用程序上下文对象的子对象），并加载bean定义，源码如下：

/\*\*

\* Initialize and publish the WebApplicationContext for this servlet.

\* <p>Delegates to {@link #createWebApplicationContext} for actual creation

\* of the context. Can be overridden in subclasses.

\* **@return** the WebApplicationContext instance

\* **@see** #FrameworkServlet(WebApplicationContext)

\* **@see** #setContextClass

\* **@see** #setContextConfigLocation

\*/

**protected** WebApplicationContext initWebApplicationContext() {

WebApplicationContext rootContext =

WebApplicationContextUtils.getWebApplicationContext(getServletContext());

WebApplicationContext wac = **null**;

**if** (**this**.webApplicationContext != **null**) {

// A context instance was injected at construction time -> use it

wac = **this**.webApplicationContext;

**if** (wac **instanceof** ConfigurableWebApplicationContext) {

ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext) wac;

**if** (!cwac.isActive()) {

// The context has not yet been refreshed -> provide services such as

// setting the parent context, setting the application context id, etc

**if** (cwac.getParent() == **null**) {

// The context instance was injected without an explicit parent -> set

// the root application context (if any; may be null) as the parent

cwac.setParent(rootContext);

}

configureAndRefreshWebApplicationContext(cwac);

}

}

}

**if** (wac == **null**) {

// No context instance was injected at construction time -> see if one

// has been registered in the servlet context. If one exists, it is assumed

// that the parent context (if any) has already been set and that the

// user has performed any initialization such as setting the context id

wac = findWebApplicationContext();

}

**if** (wac == **null**) {

// No context instance is defined for this servlet -> create a local one

**wac = createWebApplicationContext(rootContext);**

}

**if** (!**this**.refreshEventReceived) {

// Either the context is not a ConfigurableApplicationContext with refresh

// support or the context injected at construction time had already been

// refreshed -> trigger initial onRefresh manually here.

onRefresh(wac);

}

**if** (**this**.publishContext) {

// Publish the context as a servlet context attribute.

String attrName = getServletContextAttributeName();

getServletContext().setAttribute(attrName, wac);

**if** (**this**.logger.isDebugEnabled()) {

**this**.logger.debug("Published WebApplicationContext of servlet '" + getServletName() +

"' as ServletContext attribute with name [" + attrName + "]");

}

}

**return** wac;

}

