12_精选面试题 Spring IoC 容器初始化过程

1、开篇

上周介绍了互联网教育系统的项目环境，在正式开始编码之前我们需要补充一些 Spring IOC 和 Spring AOP 的知识。包括 Spring IOC 依赖查询方式，Spring IOC 依赖注入方式，Spring IOC 依赖来源，JDK 动态代理和 CGLib 动态代理以及一些常用的语法特性。在开启元气满满的周课程之前，先来电开胃甜点，来聊聊 Spring IoC 容器初始化过程的相关知识。今天的内容包含如下几个部分：

- IoC 是如何工作的？
- Resource 定位
- 载入 BeanDefinition
- 将 BeanDefiniton 注册到容器

2、IoC 是如何工作的？

如图 1 所示，通过 ApplicationContext 创建 Spring 容器，该容器会读取配置文件"/beans.xml",并统一管理由该文件中定义好的 bean 实例对象，如果要获取某个 bean 实例，使用 getBean 方法就行了。假设将 User 配置在 beans.xml 文件中，之后不需使用 new User()的方式创建实例，而是通过 ApplicationContext 容器来获取 User 的实例。

```
ApplicationContext appContext = new ClassPathXmlApplicationContext("/beans.xml");
User p = (User)appContext.getBean("user");
```

图 1 通过 spring 容器创建实例

下面就来刨析创建 IoC 容器经历的几个阶段：Resource 定位、载入 BeanDefinition、将 BeanDefiniton 注册到容器。

3、Resource 定位

Resource 是 Spring 中用于封装 I/O 操作的接口。在创建 Spring 容器时，会去访问 XML 配置文件，还可以通过文件类型、二进制流、URL 等方式访问资源。这些都可以理解为 Resource，其体系结构如图 2 所示：

- FileSystemResource：以文件绝对路径进行资源访问。
- ClassPathResourcee：以类路径的方式访问资源。
- ServletContextResource：web 应用根目录的方式访问资源。
- UrlResource：访问网络资源的实现类。
- ByteArrayResource: 访问字节数组资源的实现类。

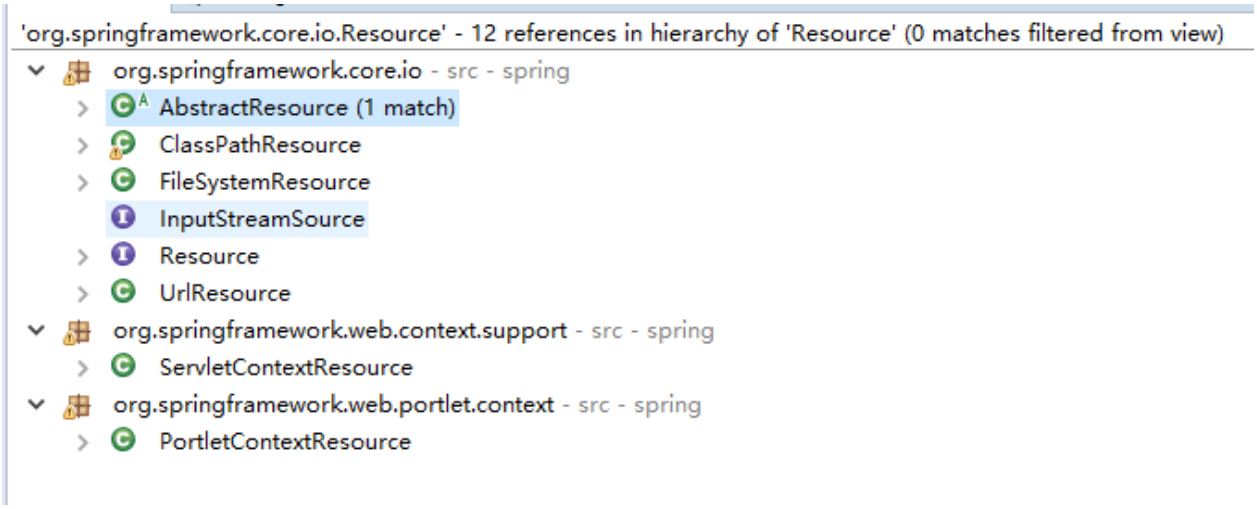

图 2 Resouce 资源访问类型

那么这些类型在 Spring 中是如何访问的呢？Spring 提供了 ResourceLoader 接口用于实现不同的 Resource 加载策略，该接口的实例对象中可以获取一个 resource 对象。如图 3 所示，在 ResourceLoader 接口中只定义了两个方法：

```
Resource getResource(String location); //通过提供的资源location参数获取Resource实例
ClassLoader getClassLoader(); // 获取ClassLoader,通过ClassLoader可将资源载入JVM
```

图 3 ResourceLoader 接口中的两个方法

注：ApplicationContext 的所有实现类都实现 RecourceLoader 接口，因此可以直接调用 getResource（参数）获取 Resoure 对象。不同的 ApplicatonContext 实现类使用 getResource 方法取得的资源类型不同，例如：FileSystemXmlApplicationContext.getResource 获取的就是 FileSystemResource 实例；ClassPathXmlApplicationContext.getResource 获取的就是 ClassPathResource 实例；

XmlWebApplicationContext.getResource 获取的就是 ServletContextResource 实例，另外像不需要通过 xml 直接使用注解@Configuation 方式加载资源的 AnnotationConfigApplicationContext 等等。

在资源定位过程完成以后，就为资源文件中的 bean 的载入创造了 I/O 操作的条件，如何读取资源中的数据将会在下一步介绍的 BeanDefinition 的载入过程中描述。

4、载入 BeanDefinition

BeanDefinition 是一个数据结构，BeanDefinition 是根据 resource 对象中的 bean 来生成的。bean 会在 Spring IoC 容器内部以 BeanDefintion 的形式存在，IoC 容器对 bean 的管理和依赖注入的实现是通过操作 BeanDefinition 来完成的。BeanDefinition 就是 Bean 在 IoC 容器中的存在形式。

由于 Spring 的配置文件主要是 XML 格式，一般而言会使用到 AbstractXmlApplicationContext 类进行文件的读取，如图 4 所示，该类定义了一个名为 loadBeanDefinitions(DefaultListableBeanFactory beanFactory) 的方法用于获取 BeanDefinition。

方法体内会 new 一个 BeanDefinitionReader 对象，然后将生成的实例传入 loadBeanDefintions 方法。

```
// 该方法属于AbstractXmlApplicationContect类
protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws BeansException, IOExceptic
    XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFactory);
    beanDefinitionReader.setEnvironment(this.getEnvironment());
    beanDefinitionReader.setResourceLoader(this);
    beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));
    this.initBeanDefinitionReader(beanDefinitionReader);
    // 用于获取BeanDefinition
    this.loadBeanDefinitions(beanDefinitionReader);
}
```

图 4 AbstractXmlApplicationContext

接下来以 XmlBeanDefinitionReader 对象载入 BeanDefinition 为例，如图 5 所示，调用 loadBeanDefinitions 方法传入对象，分别加载 configResources（定位到的

resource 资源位置）和 configLocation（本地配置文件的位置）。也就是将用户定义的资源以及容器本身需要的资源全部加载到 reader 中。

```
// 该方法属于AbstractXmlApplicationContext类
protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws BeansException, IOException {
    Resource[] configResources = getConfigResources();//获取所有定位到的resource资源位置（用户定义）
    if (configResources != null) {
        reader.loadBeanDefinitions(configResources);//载入resources
    }
    String[] configLocations = getConfigLocations();//获取所有本地配置文件的位置（容器自身）
    if (configLocations != null) {
        reader.loadBeanDefinitions(configLocations);//载入resources
    }
}
```

图 5 loadBeanDefinitions 方法

顺着看 reader 中的 loadBeanDefinitions 方法，该方法 override 了 AbstractBeanDefinitionReader 类，父接口的 BeanDefinitionReader。方法体中，将所有资源全部加在，并且交给 AbstractBeanDefinitionReader 的实现子类处理这些 resource。

```
// 该方法属于AbstractBeanDefinitionReader类，父接口BeanDefinitionReader
@Override
public int loadBeanDefinitions(Resource... resources) throws BeanDefinitionStoreException {
    Assert.notNull(resources, "Resource array must not be null");
    int counter = 0;
    for (Resource resource : resources) {
        // 将所有资源全部加载，交给AbstractBeanDefinitionReader的实现子类处理这些resource
        counter += loadBeanDefinitions(resource);
    }
    return counter;
}
```

图 6 reader 中的 loadBeanDefinitions

如图 7 所示，BeanDefinitionReader 接口定义了 int loadBeanDefinitions（Resource resource）方法。

```
int loadBeanDefinitions(Resource resource) throws BeanDefinitionStoreException;

int loadBeanDefinitions(Resource... resources) throws BeanDefinitionStoreException;
```

图 7 BeanDefinitionReader 接口定义的方法。

此时回到 XmlBeanDefinitionReader 上来，它主要针对 XML 方式的 Bean 进行读取，XmlBeanDefinitionReader 主要是实现了 AbstractBeanDefinitionReader 抽象类，而该类继承与 BeanDefinitionReader，主要实现的方法也是来自于 BeanDefinitionReader 的 loadBeanDefintions（Resource）方法。

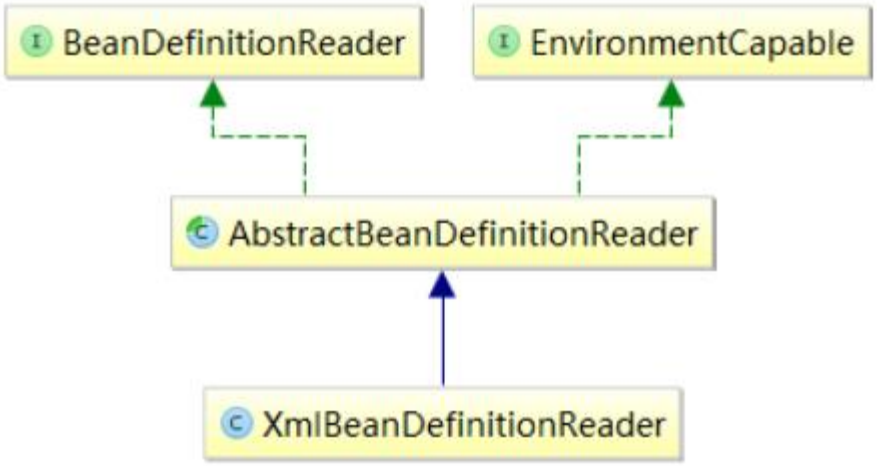

图 8 XmlBeanDefinitionReader 继承关系图

如图 9 所示，读取 Bean 之后就是加载 Bean 的过程了，XmlBeanDefinitionReader 中的 doLoadBeanDefinitions 方法主要来处理加载 Bean 的工作。首先对资源进行验证，然后从资源对象中加载 Document 对象，使用了 documentLoader 中的 loadDocument 方法，然后跟上 registerBeanDefinitions 对文档对应的 resource 进行注册，也就是将 XML 文件中的 Bean 转换成容器中的 BeanDefinition。

```
protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)
        throws BeanDefinitionStoreException {
    try {
        // 获取指定资源的验证模式
        int validationMode = getValidationModeForResource(resource);

        // 从资源对象中加载DocumentL对象，大致过程为：将resource资源文件的内容读入到document中
        // DocumentLoader在容器读取XML文件过程中有着举足轻重的作用!
        // XmlBeanDefinitionReader实例化时会创建一个DefaultDocumentLoader型的私有属性,继而调用loadDocument方法
        // inputSource--要加载的文档的输入源
        Document doc = this.documentLoader.loadDocument(
                inputSource, this.entityResolver, this.errorHandler, validationMode, this.namespaceAware);

        // 将document文件的bean封装成BeanDefinition，并注册到容器
        return registerBeanDefinitions(doc, resource);
    }
    catch ...(略)
}
```

图 9 doLoadBeanDefinitions 方法

如图 10 所示，接下来就是 registerBeanDefinitions 方法了，它主要对 Spring Bean 语义进行转化，变成 BeanDefintion 类型。首先获取 DefaultBeanDefinitionDocumentReader 实例，然后获取容器中的 bean 数量，通过 documentReader 中的 registerBeanDefinitions 方法进行注册和转化工作。

```
/**
 * 属于XmlBeanDefinitionReader类
 * Register the bean definitions contained in the given DOM document.
 * @param doc the DOM document
 * @param resource
 * @return the number of bean definitions found
 * @throws BeanDefinitionStoreException
 */
public int registerBeanDefinitions(Document doc, Resource resource) throws BeanDefinitionStoreException {
    // 获取到DefaultBeanDefinitionDocumentReader实例
    BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();
    // 获取容器中bean的数量
    int countBefore = getRegistry().getBeanDefinitionCount();
    documentReader.registerBeanDefinitions(doc, createReaderContext(resource));
    return getRegistry().getBeanDefinitionCount() - countBefore;
}
```

图 10 registerBeanDefintions

顺着上面的思路继续往下，在 DefaultBeanDefinitionDocumentReader 中的 registerBeanDefinitions 方法如图 11 所示，其获取 document 的根结点然后顺势访问所有的子节点。同时把处理 BeanDefinition 的过程委托给 BeanDefinitionParserDelegate 对象来完成。

```
// DefaultBeanDefinitionDocumentReader implements BeanDefinitionDocumentReader
public void registerBeanDefinitions(Document doc, XmlReaderContext readerContext) {
    this.readerContext = readerContext;

    logger.debug("Loading bean definitions");
    // 获取doc的root节点，通过该节点能够访问所有的子节点
    Element root = doc.getDocumentElement();
    // 处理beanDefinition的过程委托给BeanDefinitionParserDelegate实例对象来完成
    BeanDefinitionParserDelegate delegate = createHelper(readerContext, root);

    // Default implementation is empty.
    // Subclasses can override this method to convert custom elements into standard Spring bean definitions
    preProcessXml(root);
    // 核心方法，代理
    parseBeanDefinitions(root, delegate);
    postProcessXml(root);
}
```

图 11 DefaultBeanDefinitionDocumentReader 中的 registerBeanDefinitions 方法

BeanDefinitionParserDelegate 类主要负责 BeanDefinition 的解析，这里涉及到 JDK 和 CGLIB 动态代理的知识，这里留一个悬念我们后面的章节会深入介绍。BeanDefinitionParserDelegate 代理类会完成对符合 Spring Bean 语义规则的处理，比如<bean></bean>、<import></import>、<alias><alias/>等的检测。如图 12 所示，就是 BeanDefinitionParserDelegate 代理类中的 parseBeanDefinitions 方法，用来对 XML 文件中的节点进行解析。通过遍历 import 标签节点调用 importBeanDefinitionResource 方法对其进行处理，然后接着便利 bean 节点调用 processBeanDefinition 对其处理。

```
protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {
    if (delegate.isDefaultNamespace(root)) {
        NodeList nl = root.getChildNodes();
        // 遍历所有节点，做对应解析工作
        // 如遍历到<import>标签节点就调用importBeanDefinitionResource(ele)方法对应处理
        // 遍历到<bean>标签就调用processBeanDefinition(ele,delegate)方法对应处理
        for (int i = 0; i < nl.getLength(); i++) {
            Node node = nl.item(i);
            if (node instanceof Element) {
                Element ele = (Element) node;
                if (delegate.isDefaultNamespace(ele)) {
                    parseDefaultElement(ele, delegate);
                }
                else {
                    //对应用户自定义节点处理方法
                    delegate.parseCustomElement(ele);
                }
            }
        }
    }
    else {
        delegate.parseCustomElement(root);
    }
}
```

图 12 parseBeanDefinitions 方法

如图 13 再看 parseBeanDefinitions 方法中调用的 parseDefaultElement 方法，顾名思义它是对节点元素进行处理的。从方法体的语句可以看出它对 import 标签、alias 标签、bean 标签进行了处理。每类标签对应不同的 BeanDefinition 的处理方法。

```
private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate) {
    // 解析<import>标签
    if (delegate.nodeNameEquals(ele, IMPORT_ELEMENT)) {
        importBeanDefinitionResource(ele);
    }
    // 解析<alias>标签
    else if (delegate.nodeNameEquals(ele, ALIAS_ELEMENT)) {
        processAliasRegistration(ele);
    }
    // 解析<bean>标签,最常用，过程最复杂
    else if (delegate.nodeNameEquals(ele, BEAN_ELEMENT)) {
        processBeanDefinition(ele, delegate);
    }
    // 解析<beans>标签
    else if (delegate.nodeNameEquals(ele, NESTED_BEANS_ELEMENT)) {
        // recurse
        doRegisterBeanDefinitions(ele);
    }
}
```

图 12 parseDefaultElement 方法

在 parseDefaultElement 调用的众多方法中，我们选取 processBeanDefinition 方法给大家讲解，如图 13 所示，该方法是用来处理 Bean 的。首先通过 delegate 的 parseBeanDefinitionElement 方法传入节点信息，获取该 Bean 对应的 name 和 alias。然后通过 BeanDefinitionReaderUtils 中的 registerBeanDefinition 方法对其进行容器注册，也就是将 Bean 实例注册到容器中进行管理。最后，发送注册事件。

```
protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate delegate) {
    // 该对象持有beanDefinition的name和alias，可以使用该对象完成beanDefinition向容器的注册
    BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
    if (bdHolder != null) {
        bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
        try {
            // 注册最终被修饰的bean实例，下文注册beanDefinition到容器会讲解该方法
            BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder, getReaderContext().getRegist
        }
        catch (BeanDefinitionStoreException ex) {
            getReaderContext().error("Failed to register bean definition with name '" +
                    bdHolder.getBeanName() + "'", ele, ex);
        }
        // Send registration event.
        getReaderContext().fireComponentRegistered(new BeanComponentDefinition(bdHolder));
    }
}
```

图 13 processBeanDefinition 方法

至此完成了 BeanDefinition 的加载工作。

5、将 BeanDefiniton 注册到容器

在加载了 Bean 之后，就需要将其注册到容器中尽心管理。如图 14 所示，Bean 会被解析成 BeanDefinition 并与 BeanName、Alias 一同封装到 BeanDefinitionHolder 类中， 之后 beanFactory.registerBeanDefinition(beanName, bdHolder.getBeanDefinition())，注册到 DefaultListableBeanFactory.beanDefinitionMap 中。如果客户端需要获取 Bean 对象，Spring 容器会根据注册的 BeanDefinition 信息进行实例化。

```
public static void registerBeanDefinition(
        BeanDefinitionHolder bdHolder, BeanDefinitionRegistry beanFactory) throws BeansException {

    // Register bean definition under primary name.
    String beanName = bdHolder.getBeanName();        // 注册beanDefinition!!!
    beanFactory.registerBeanDefinition(beanName, bdHolder.getBeanDefinition());

    // 如果有别名的话也注册进去，Register aliases for bean name, if any.
    String[] aliases = bdHolder.getAliases();
    if (aliases != null) {
        for (int i = 0; i < aliases.length; i++) {
            beanFactory.registerAlias(beanName, aliases[i]);
        }
    }
}
```

图 14 registerBeanDefinition

DefaultListableBeanFactory 实现了上面调用 BeanDefinitionRegistry 接口的 registerBeanDefinition(beanName, bdHolder.getBeanDefinition())方法。如图 15 所示，这一部分的主要逻辑是向 DefaultListableBeanFactory 对象的 beanDefinitionMap 中存放 beanDefinition，也就是说 beanDefinition 都放在 beanDefinitionMap 中进行管理。当初始化容器进行 bean 初始化时，在 bean 的生命周期分析里必然会在这个 beanDefinitionMap 中获取 beanDefition 实例。

```
public void registerBeanDefinition(String beanName, BeanDefinition beanDefinition)
        throws BeanDefinitionStoreException {
    if (beanDefinition instanceof AbstractBeanDefinition) {
        try {
            ((AbstractBeanDefinition) beanDefinition).validate();
        }
        catch (BeanDefinitionValidationException ex) {
            throw new BeanDefinitionStoreException(beanDefinition.getResourceDescription(), beanName,
                    "Validation of bean definition failed", ex);
        }
    }
    // beanDefinitionMap是个ConcurrentHashMap类型数据，用于存放beanDefinition,它的key值是beanName
    Object oldBeanDefinition = this.beanDefinitionMap.get(beanName);
    if (oldBeanDefinition != null) {
        if (!this.allowBeanDefinitionOverriding) {
            throw new BeanDefinitionStoreException(beanDefinition.getResourceDescription(), beanName,
                    "Cannot register bean definition [" + beanDefinition + "] for bean '" + beanName +
                    "': there's already [" + oldBeanDefinition + "] bound");
        }
        else {
            if (logger.isInfoEnabled()) {
                logger.info("Overriding bean definition for bean '" + beanName +
                        "': replacing [" + oldBeanDefinition + "] with [" + beanDefinition + "]");
            }
        }
    }
    else {
        this.beanDefinitionNames.add(beanName);
    }
    // 将获取到的BeanDefinition放入Map中，容器操作使用bean时通过这个HashMap找到具体的BeanDefinition
    this.beanDefinitionMap.put(beanName, beanDefinition);
    removeSingleton(beanName);
}
```

图 15 registerBeanDefinition 方法

6、总结

本节课介绍了 Spring IOC 容器初始化的过程，包括 Resource 定位：通过文件路径、类路径、web 路径等方式获取 Bean 信息；载入 BeanDefinition：介绍的是如何将 Bean 载入到 IoC 中形成 BeanDefinition 的整个过程；将 BeanDefinition 注册到容器。下节课，看看 Spring IOC 依赖查找的方式有哪些？下期见，拜拜。