



[回到首页](#)

芋道源码 —— 知识星球

我是一段不羁的公告！

记得给芳芳这 3 个项目加油，添加一个 STAR 噢。

<https://github.com/YunaiV/SpringBoot-Labs>

<https://github.com/YunaiV/onemall>

<https://github.com/YunaiV/ruoyi-vue-pro>

2019-02-13

[Spring](#)

【死磕 Spring】—— IoC 之获取 Document 对象

本文主要基于 Spring 5.0.6.RELEASE

摘要：原创出处 <http://cmsblogs.com/?p=2695> 「小明哥」，谢谢！

作为「小明哥」的忠实读者，「老芳芳」略作修改，记录在理解过程中，参考的资料。

在 `XmlBeanDefinitionReader#doLoadDocument(InputStream inputStream, Resource resource)` 方法，中做了两件事情：

- 调用 `#getValidationModeForResource(Resource resource)` 方法，获取指定资源（xml）的验证模式。

- 上篇博客，我们已经详细解析。

- 调用 `DocumentLoader#loadDocument(InputStream inputStream, EntityResolver entityResolver, ErrorHandler errorHandler, int validationMode, boolean namespaceAware)` 方法，获取 XML Document 实例。

- 本篇博客，我们来详细解析。

1. DocumentLoader

获取 Document 的策略，由接口 `org.springframework.beans.factory.xml.DocumentLoader` 定义。代码如下：

FROM 《Spring 源码深度解析》P16 页

定义从资源文件加载到转换为 Document 的功能。

```
public interface DocumentLoader {

    Document loadDocument(
        InputStream inputStream, EntityResolver entityResolver,
        ErrorHandler errorHandler, int validationMode, boolean namespaceAware)
        throws Exception;

}
```

`inputSource` 方法参数，加载 Document 的 Resource 资源。

entityResolver 方法参数，解析文件的解析器。

errorHandler 方法参数，处理加载 Document 对象的过程的错误。

validationMode 方法参数，验证模式。

namespaceAware 方法参数，命名空间支持。如果要提供对 XML 名称空间的支持，则需要值为 true 。

1.1 DefaultDocumentLoader

该方法由 DocumentLoader 的默认实现类 org.springframework.beans.factory.xml.DefaultDocumentLoader 实现。代码如下：

```
/**
 * Load the {@link Document} at the supplied {@link InputSource} using the standard JAXP-configured
 * XML parser.
 */
@Override
public Document loadDocument(InputSource inputSource, EntityResolver entityResolver,
    ErrorHandler errorHandler, int validationMode, boolean namespaceAware) throws Exception {
    // <1> 创建 DocumentBuilderFactory
    DocumentBuilderFactory factory = createDocumentBuilderFactory(validationMode, namespaceAware);
    if (logger.isTraceEnabled()) {
        logger.trace("Using JAXP provider [" + factory.getClass().getName() + "]");
    }
    // <2> 创建 DocumentBuilder
    DocumentBuilder builder = createDocumentBuilder(factory, entityResolver, errorHandler);
    // <3> 解析 XML InputSource 返回 Document 对象
    return builder.parse(inputSource);
}
```

首先，调用 #createDocumentBuilderFactory(...) 方法，创建 javax.xml.parsers.DocumentBuilderFactory 对象。代码如下：

```
/**
 * JAXP attribute used to configure the schema language for validation.
 */
private static final String SCHEMA_LANGUAGE_ATTRIBUTE = "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
/**
 * JAXP attribute value indicating the XSD schema language.
 */
private static final String XSD_SCHEMA_LANGUAGE = "http://www.w3.org/2001/XMLSchema";
protected DocumentBuilderFactory createDocumentBuilderFactory(int validationMode, boolean namespaceAware)
    throws ParserConfigurationException {
    // 创建 DocumentBuilderFactory
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    factory.setNamespaceAware(namespaceAware); // 设置命名空间支持
    if (validationMode != XmlValidationModeDetector.VALIDATION_NONE) {
        factory.setValidating(true); // 开启校验
        // XSD 模式下，设置 factory 的属性
        if (validationMode == XmlValidationModeDetector.VALIDATION_XSD) {
            // Enforce namespace aware for XSD...
            factory.setNamespaceAware(true); // XSD 模式下，强制设置命名空间支持
            // 设置 SCHEMA_LANGUAGE_ATTRIBUTE
            try {
                factory.setAttribute(SCHEMA_LANGUAGE_ATTRIBUTE, XSD_SCHEMA_LANGUAGE);
            } catch (IllegalArgumentException ex) {
```

```

        ParserConfigurationException pcex = new ParserConfigurationException(
            "Unable to validate using XSD: Your JAXP provider [" + factory +
            "] does not support XML Schema. Are you running on Java 1.4 with Apache Crimson? " +
            "Upgrade to Apache Xerces (or Java 1.5) for full XSD support.");
        pcex.initCause(ex);
        throw pcex;
    }
}
}
return factory;
}

```

然后，调用 `#createDocumentBuilder(DocumentBuilderFactory factory, EntityResolver entityResolver, ErrorHandler errorHandler)` 方法，创建 `javax.xml.parsers.DocumentBuilder` 对象。代码如下：

```

protected DocumentBuilder createDocumentBuilder(DocumentBuilderFactory factory,
    @Nullable EntityResolver entityResolver, @Nullable ErrorHandler errorHandler)
    throws ParserConfigurationException {
    // 创建 DocumentBuilder 对象
    DocumentBuilder docBuilder = factory.newDocumentBuilder();
    // <x> 设置 EntityResolver 属性
    if (entityResolver != null) {
        docBuilder.setEntityResolver(entityResolver);
    }
    // 设置 ErrorHandler 属性
    if (errorHandler != null) {
        docBuilder.setErrorHandler(errorHandler);
    }
    return docBuilder;
}

```

- 在 <x> 处，设置 `DocumentBuilder` 的 `EntityResolver` 属性。关于它，在 [\[2. EntityResolver\]](#) 会详细解析。

最后，调用 `DocumentBuilder#parse(InputSource)` 方法，解析 `InputSource`，返回 `Document` 对象。

2. EntityResolver

通过 `DocumentLoader#loadDocument(...)` 方法来获取 `Document` 对象时，有一个方法参数 `entityResolver`。该参数是通过 `XmlBeanDefinitionReader#getEntityResolver()` 方法来获取的。代码如下：

`#getEntityResolver()` 方法，返回指定的解析器，如果没有指定，则构造一个未指定的默认解析器。

```

// XmlBeanDefinitionReader.java

/**
 * EntityResolver 解析器
 */
@Nullable
private EntityResolver entityResolver;

```

```

protected EntityResolver getEntityResolver() {
    if (this.entityResolver == null) {
        // Determine default EntityResolver to use.
        ResourceLoader resourceLoader = getResourceLoader();
        if (resourceLoader != null) {
            this.entityResolver = new ResourceEntityResolver(resourceLoader);
        } else {
            this.entityResolver = new DelegatingEntityResolver(getBeanClassLoader());
        }
    }
    return this.entityResolver;
}

```

如果 ResourceLoader 不为 null，则根据指定的 ResourceLoader 创建一个 ResourceEntityResolver 对象。

如果 ResourceLoader 为 null，则创建一个 DelegatingEntityResolver 对象。该 Resolver 委托给默认的 BeansDtdResolver 和 PluggableSchemaResolver。

2.1 子类

上面的方法，一共涉及四个 EntityResolver 的子类：

org.springframework.beans.factory.xml.BeansDtdResolver：实现 EntityResolver 接口，Spring Bean dtd 解码器，用来从 classpath 或者 jar 文件中加载 dtd。部分代码如下：

```

private static final String DTD_EXTENSION = ".dtd";

private static final String DTD_NAME = "spring-beans";

```

org.springframework.beans.factory.xml.PluggableSchemaResolver，实现 EntityResolver 接口，读取 classpath 下的所有 "META-INF/spring.schemas" 成一个 namespaceURI 与 Schema 文件地址的 map。代码如下：

```

/**
 * The location of the file that defines schema mappings.
 * Can be present in multiple JAR files.
 *
 * 默认 {@link #schemaMappingsLocation} 地址
 */
public static final String DEFAULT_SCHEMA_MAPPINGS_LOCATION = "META-INF/spring.schemas";

@Nullable
private final ClassLoader classLoader;

/**
 * Schema 文件地址
 */
private final String schemaMappingsLocation;

/** Stores the mapping of schema URL -> local schema path. */
@Nullable
private volatile Map<String, String> schemaMappings; // namespaceURI 与 Schema 文件地址的映射集合

```

org.springframework.beans.factory.xml.DelegatingEntityResolver : 实现 EntityResolver 接口, 分别代理 dtd 的 BeansDtdResolver 和 xml schemas 的 PluggableSchemaResolver 。代码如下:

```
/** Suffix for DTD files. */
public static final String DTD_SUFFIX = ".dtd";

/** Suffix for schema definition files. */
public static final String XSD_SUFFIX = ".xsd";

private final EntityResolver dtdResolver;

private final EntityResolver schemaResolver;

// 默认
public DelegatingEntityResolver(@Nullable ClassLoader classLoader) {
    this.dtdResolver = new BeansDtdResolver();
    this.schemaResolver = new PluggableSchemaResolver(classLoader);
}

// 自定义
public DelegatingEntityResolver(EntityResolver dtdResolver, EntityResolver schemaResolver) {
    Assert.notNull(dtdResolver, "'dtdResolver' is required");
    Assert.notNull(schemaResolver, "'schemaResolver' is required");
    this.dtdResolver = dtdResolver;
    this.schemaResolver = schemaResolver;
}
```

org.springframework.beans.factory.xml.ResourceEntityResolver : 继承自 DelegatingEntityResolver 类, 通过 ResourceLoader 来解析实体的引用。代码如下:

```
private final ResourceLoader resourceLoader;

public ResourceEntityResolver(ResourceLoader resourceLoader) {
    super(resourceLoader.getClassLoader());
    this.resourceLoader = resourceLoader;
}
```

2.2 作用

EntityResolver 的作用就是, 通过实现它, 应用可以自定义如何寻找【验证文件】的逻辑。

FROM 《Spring 源码深度解析》

在 loadDocument 方法中涉及一个参数 EntityResolver , 何为EntityResolver? 官网这样解释: 如果 SAX 应用程序需要实现自定义处理外部实体, 则必须实现此接口并使用 setEntityResolver 方法向SAX 驱动器注册一个实例。也就是说, 对于解析一个 XML, SAX 首先读取该 XML 文档上的声明, 根据声明去寻找相应的 DTD 定义, 以便对文档进行一个验证。默认的寻找规则, 即通过网络(实现上就是声明的DTD的URI地址)来下载相应的DTD声明, 并进行认证。下载的过程是一个漫长的过程, 而且当网络中断或不可用时, 这里会报错, 就是因为相应的DTD声明没有被找到的原因。

EntityResolver 的作用是项目本身就可以提供一个如何寻找 DTD 声明的方法, 即由程

序来实现寻找 DTD 声明的过程，比如我们将 DTD 文件放到项目中某处，在实现时直接将此文档读取并返回给 SAX 即可。这样就避免了通过网络来寻找相应的声明。

org.xml.sax.EntityResolver 接口，代码如下：

```
public interface EntityResolver {  
  
    public abstract InputSource resolveEntity (String publicId, String systemId)  
        throws SAXException, IOException;  
  
}
```

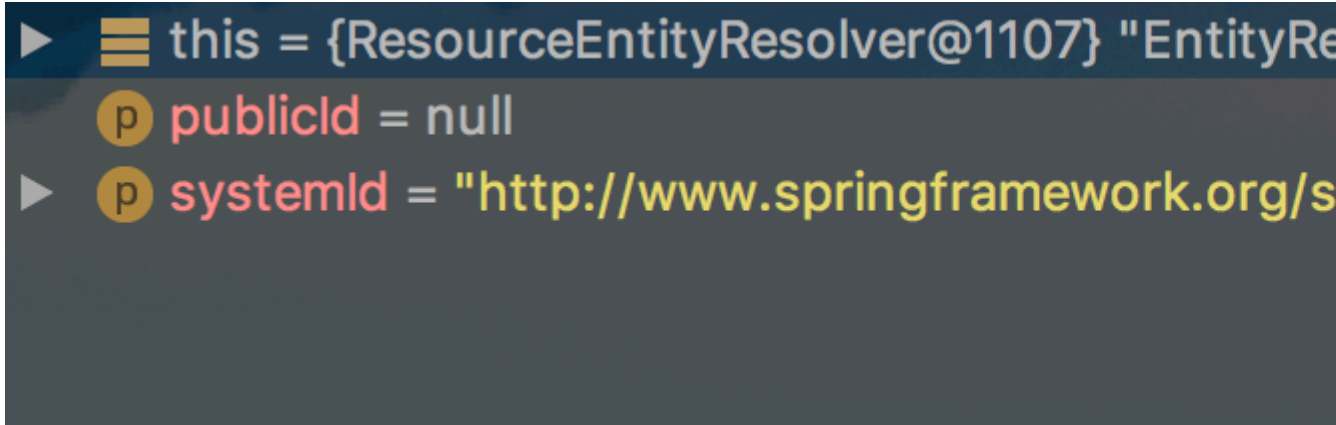
接口方法接收两个参数 publicId 和 systemId，并返回 InputSource 对象。两个参数声明如下：

publicId：被引用的外部实体的公共标识符，如果没有提供，则返回 null。
systemId：被引用的外部实体的系统标识符。

这两个参数的实际内容和具体的验证模式的关系如下：

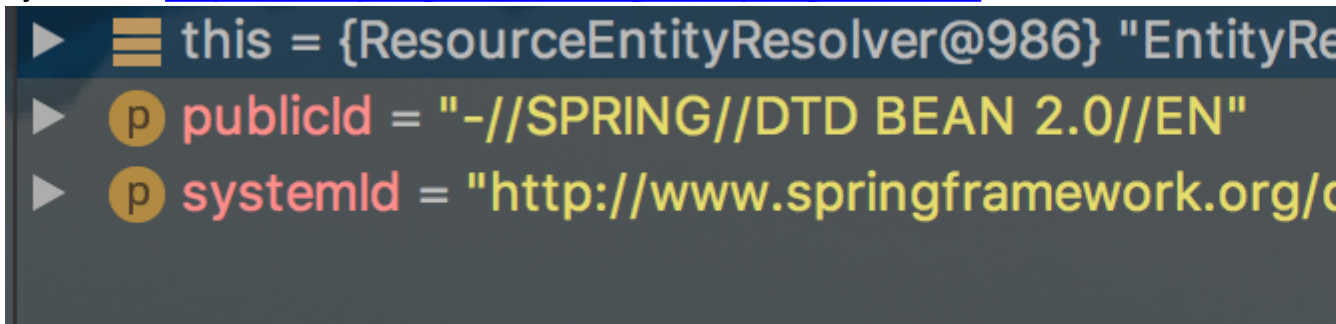
XSD 验证模式

- publicId: null
- systemId: <http://www.springframework.org/schema/beans/spring-beans.xsd>



DTD 验证模式

- publicId: -//SPRING//DTD BEAN 2.0//EN
- systemId: <http://www.springframework.org/dtd/spring-beans.dtd>



2.3 DelegatingEntityResolver

我们知道在 Spring 中使用 DelegatingEntityResolver 为 EntityResolver 的实现类。
#resolveEntity(String publicId, String systemId) 方法，实现如下：

```

@Override
@Nullable
public InputSource resolveEntity(String publicId, @Nullable String systemId) throws SAXException, IOException {
    if (systemId != null) {
        // DTD 模式
        if (systemId.endsWith(DTD_SUFFIX)) {
            return this.dtdResolver.resolveEntity(publicId, systemId);
        }
        // XSD 模式
        } else if (systemId.endsWith(XSD_SUFFIX)) {
            return this.schemaResolver.resolveEntity(publicId, systemId);
        }
    }
    return null;
}

```

如果是 DTD 验证模式，则使用 BeansDtdResolver 来进行解析

如果是 XSD 验证模式，则使用 PluggableSchemaResolver 来进行解析。

2.4 BeansDtdResolver

BeansDtdResolver 的解析过程，代码如下：

```

/**
 * DTD 文件的后缀
 */
private static final String DTD_EXTENSION = ".dtd";
/**
 * Spring Bean DTD 的文件名
 */
private static final String DTD_NAME = "spring-beans";

@Override
@Nullable
public InputSource resolveEntity(String publicId, @Nullable String systemId) throws IOException {
    if (logger.isTraceEnabled()) {
        logger.trace("Trying to resolve XML entity with public ID [" + publicId +
            "] and system ID [" + systemId + "]");
    }
    // 必须以 .dtd 结尾
    if (systemId != null && systemId.endsWith(DTD_EXTENSION)) {
        // 获取最后一个 / 的位置
        int lastPathSeparator = systemId.lastIndexOf('/');
        // 获取 spring-beans 的位置
        int dtdNameStart = systemId.indexOf(DTD_NAME, lastPathSeparator);
        if (dtdNameStart != -1) { // 找到
            String dtdFile = DTD_NAME + DTD_EXTENSION;
            if (logger.isTraceEnabled()) {
                logger.trace("Trying to locate [" + dtdFile + "] in Spring jar on classpath");
            }
            try {
                // 创建 ClassPathResource 对象
                Resource resource = new ClassPathResource(dtdFile, getClass());
                // 创建 InputSource 对象，并设置 publicId、systemId 属性
                InputSource source = new InputSource(resource.getInputStream());
                source.setPublicId(publicId);
                source.setSystemId(systemId);
            }
        }
    }
}

```

```

        if (logger.isTraceEnabled()) {
            logger.trace("Found beans DTD [" + systemId + "] in classpath: " + dtdFile);
        }
        return source;
    }
    catch (IOException ex) {
        if (logger.isDebugEnabled()) {
            logger.debug("Could not resolve beans DTD [" + systemId + "]: not found in classpath", ex);
        }
    }
}

// 使用默认行为，从网络上下载
// Use the default behavior -> download from website or wherever.
return null;
}

```

从上面的代码中，我们可以看到，加载 DTD 类型的 `BeansDtdResolver#resolveEntity(...)` 过程，只是对 `systemId` 进行了简单的校验（从最后一个 / 开始，内容中是否包含 `spring-beans`），然后构造一个 `InputSource` 对象，并设置 `publicId`、`systemId` 属性，然后返回。

2.5 PluggableSchemaResolver

`PluggableSchemaResolver` 的解析过程，代码如下：

```

@Nullable
private final ClassLoader classLoader;

/**
 * Schema 文件地址
 */
private final String schemaMappingsLocation;

/** Stores the mapping of schema URL -> local schema path. */
@Nullable
private volatile Map<String, String> schemaMappings; // namespaceURI 与 Schema 文件地址的映射集合

@Override
@Nullable
public InputSource resolveEntity(String publicId, @Nullable String systemId) throws IOException {
    if (logger.isTraceEnabled()) {
        logger.trace("Trying to resolve XML entity with public id [" + publicId +
            "] and system id [" + systemId + "]");
    }

    if (systemId != null) {
        // 获得 Resource 所在位置
        String resourceLocation = getSchemaMappings().get(systemId);
        if (resourceLocation != null) {
            // 创建 ClassPathResource
            Resource resource = new ClassPathResource(resourceLocation, this.classLoader);
            try {
                // 创建 InputSource 对象，并设置 publicId、systemId 属性
                InputSource source = new InputSource(resource.getInputStream());
                source.setPublicId(publicId);
            }
        }
    }
}

```



```

        source.setSystemId(systemId);
        if (logger.isTraceEnabled()) {
            logger.trace("Found XML schema [" + systemId + "] in classpath: " + resourceLocation);
        }
        return source;
    }
    catch (FileNotFoundException ex) {
        if (logger.isDebugEnabled()) {
            logger.debug("Could not find XML schema [" + systemId + "]: " + resource, ex);
        }
    }
}
}
return null;
}

```

首先调用 `#getSchemaMappings()` 方法，获取一个映射表(systemId 与其在本地的对照关系)。代码如下：

```

private Map<String, String> getSchemaMappings() {
    Map<String, String> schemaMappings = this.schemaMappings;
    // 双重检查锁，实现 schemaMappings 单例
    if (schemaMappings == null) {
        synchronized (this) {
            schemaMappings = this.schemaMappings;
            if (schemaMappings == null) {
                if (logger.isTraceEnabled()) {
                    logger.trace("Loading schema mappings from [" + this.schemaMappingsLocation + "]");
                }
                try {
                    // 以 Properties 的方式，读取 schemaMappingsLocation
                    Properties mappings = PropertiesLoaderUtils.loadAllProperties(this.schemaMappingsLocation,
                        if (logger.isTraceEnabled()) {
                            logger.trace("Loaded schema mappings: " + mappings);
                        }
                    }
                    // 将 mappings 初始化到 schemaMappings 中
                    schemaMappings = new ConcurrentHashMap<>(mappings.size());
                    CollectionUtils.mergePropertiesIntoMap(mappings, schemaMappings);
                    this.schemaMappings = schemaMappings;
                } catch (IOException ex) {
                    throw new IllegalStateException(
                        "Unable to load schema mappings from location [" + this.schemaMappingsLocation + "]");
                }
            }
        }
    }
    return schemaMappings;
}

```

。映射表如下（部分）：

```

} "http://www.springframework.org/schema/context/s
"http://www.springframework.org/schema/cache/spr
} "http://www.springframework.org/schema/beans/sp
} "http://www.springframework.org/schema/context/s
} "http://www.springframework.org/schema/context/s

```

然后，根据传入的 `systemId` 获取该 `systemId` 在本地的路径 `resourceLocation`。

最后，根据 `resourceLocation`，构造 `InputStream` 对象。

2.6 ResourceEntityResolver

`ResourceEntityResolver` 的解析过程，代码如下：

```
private final ResourceLoader resourceLoader;

@Override
@Nullable
public InputStream resolveEntity(String publicId, @Nullable String systemId) throws SAXException, IOException {
    // 调用父类的方法，进行解析
    InputStream source = super.resolveEntity(publicId, systemId);
    // 解析失败，resourceLoader 进行解析
    if (source == null && systemId != null) {
        // 获得 resourcePath，即 Resource 资源地址
        String resourcePath = null;
        try {
            String decodedSystemId = URLDecoder.decode(systemId, "UTF-8"); // 使用 UTF-8，解码 systemId
            String givenUrl = new URL(decodedSystemId).toString(); // 转换成 URL 字符串
            // 解析文件资源的相对路径（相对于系统根路径）
            String systemRootUrl = new File("").toURL().toURL().toString();
            // Try relative to resource base if currently in system root.
            if (givenUrl.startsWith(systemRootUrl)) {
                resourcePath = givenUrl.substring(systemRootUrl.length());
            }
        } catch (Exception ex) {
            // Typically a MalformedURLException or AccessControlException.
            if (logger.isDebugEnabled()) {
                logger.debug("Could not resolve XML entity [" + systemId + "] against system root URL", ex);
            }
            // No URL (or no resolvable URL) -> try relative to resource base.
            resourcePath = systemId;
        }
        if (resourcePath != null) {
            if (logger.isTraceEnabled()) {
                logger.trace("Trying to locate XML entity [" + systemId + "] as resource [" + resourcePath + "]");
            }
            // 获得 Resource 资源
            Resource resource = this.resourceLoader.getResource(resourcePath);
            // 创建 InputStream 对象
            source = new InputStream(resource.getInputStream());
            // 设置 publicId 和 systemId 属性
            source.setPublicId(publicId);
            source.setSystemId(systemId);
            if (logger.isDebugEnabled()) {
                logger.debug("Found XML entity [" + systemId + "]: " + resource);
            }
        }
    }
    return source;
}
```

首先，调用父类的方法，进行解析。

如果失败，使用 `resourceLoader`，尝试读取 `systemId` 对应的 `Resource` 资源。

2.7 自定义 EntityResolver

老芳芳：本小节，为选读内容。

`#getEntityResolver()` 方法返回 `EntityResolver` 对象。那么怎么进行自定义 `EntityResolver` 呢？

If a SAX application needs to implement customized handling for external entities, it must implement this interface and register an instance with the SAX driver using the `setEntityResolver` method.

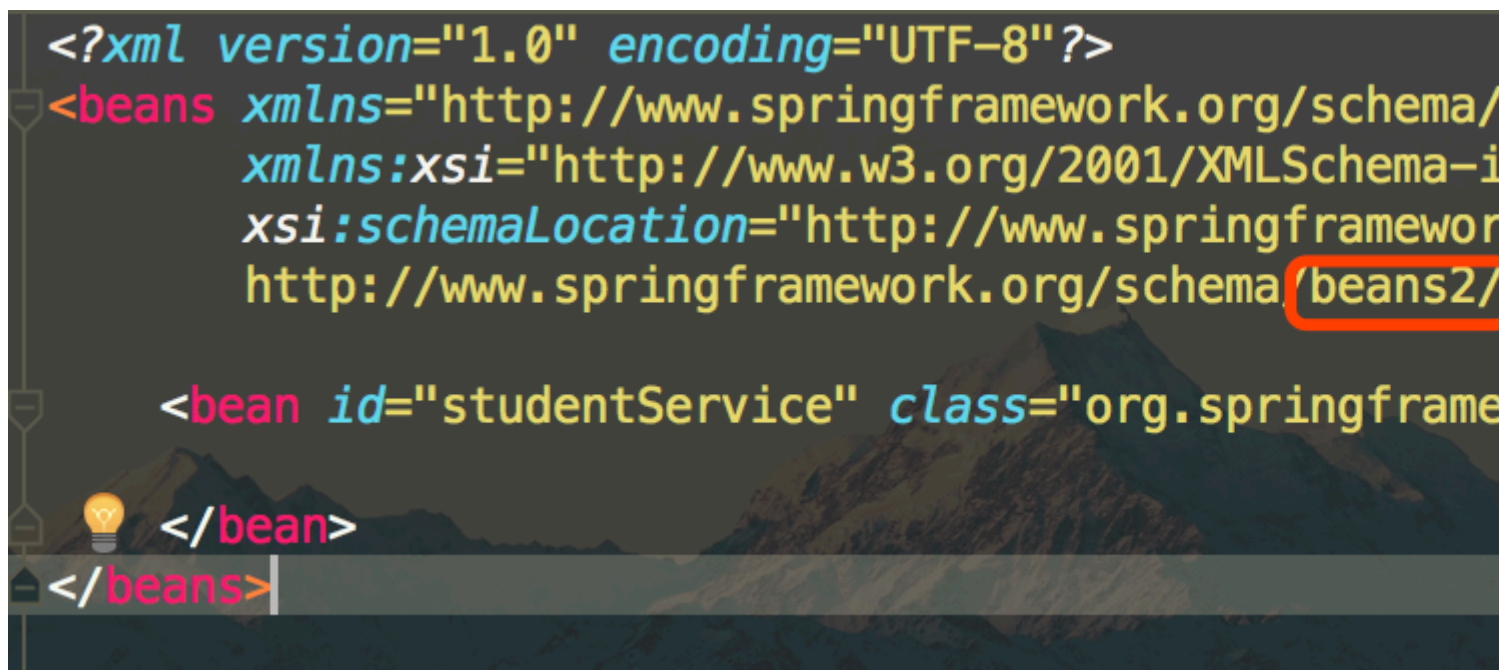
就是说：如果 SAX 应用程序需要实现自定义处理外部实体，则必须实现此接口，并使用 `#setEntityResolver(EntityResolver entityResolver)` 方法，向 SAX 驱动器注册一个 `EntityResolver` 实例。

示例如下：

```
public class MyResolver implements EntityResolver {

    @Override
    public InputSource resolveEntity(String publicId, String systemId) {
        if (systemId.equals("http://www.myhost.com/today")) {
            MyReader reader = new MyReader();
            return new InputSource(reader);
        } else {
            // use the default behaviour
            return null;
        }
    }
}
```

我们首先将 “spring-student.xml” 文件中的 XSD 声明的地址改掉，如下：



```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans2/beans2.xsd"
>

    <bean id="studentService" class="org.springframework

</bean>
</beans>
```

如果我们再次运行，则会报如下错误：

警告： Ignored XML validation warning
org.xml.sax.SAXParseException; lineNumber: 5; c

从上面的错误可以看到，是在进行文档验证时，无法根据声明找到 XSD 验证文件而导致无法进行 XML 文件验证。在 [《【死磕 Spring】—— IoC 之获取验证模型》](#) 中讲到，如果要解析一个 XML 文件，SAX 首先会读取该 XML 文档上的声明，然后根据声明去寻找相应的 DTD 定义，以便对文档进行验证。默认的加载规则是通过网络方式下载验证文件，而在实际生产环境中我们会遇到网络中断或者不可用状态，那么就应用就会因为无法下载验证文件而报错。

666. 彩蛋

是不是看到此处，有点懵逼，不是说好了分享获取 Document 对象，结果内容主要是 EntityResolver 呢？因为，从 XML 中获取 Document 对象，已经有 javax.xml 库进行解析。而 EntityResolver 的重点，是在于如何获取【验证文件】，从而验证用户写的 XML 是否通过验证。

文章目录

1. [1. 1. DocumentLoader](#)
 1. [1.1. 1.1 DefaultDocumentLoader](#)
2. [2. 2. EntityResolver](#)
 1. [2.1. 2.1 子类](#)
 2. [2.2. 2.2 作用](#)
 3. [2.3. 2.3 DelegatingEntityResolver](#)
 4. [2.4. 2.4 BeansDtdResolver](#)
 5. [2.5. 2.5 PluggableSchemaResolver](#)
 6. [2.6. 2.6 ResourceEntityResolver](#)
 7. [2.7. 2.7 自定义 EntityResolver](#)
3. [3. 666. 彩蛋](#)