Spring IoC容器ApplicationContext

作为Spring提供的较之 BeanFactory更为先进的IoC容器实现， 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。

Spring为基本的BeanFactory类型容器提供了XmlBeanFactory实现。相应地，它也为ApplicationContext类型容器提供了以下几个常用的实现。

- 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。
- 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。
- org.springframework.web.context.support.XmlWebApplicationContext 。Spring提供用于Web应用程序的 ApplicationContext 实现。

1 统一资源加载策略

首先从Java SE提供的标准类java.net.URL说起。URL全名是Unifore Resource LLocator(统一资源定位器)。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。

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。

1.1 Spring中的Resource

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。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：

- **ByteArrayResource** 将字节数组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。
- **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或者给定的类进行资源加载。
- **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。
- **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
- **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接口就可以了。

  Resource接口定义：

```
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。

1.2 ResourceLoader

资源有了，但如何去查找和定位这些资源，应该是ResourceLoader的职责所在。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由相应的ResourceLoader实现类给出。ResourceLoader接口的定义如下：

```
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。

1.2.1 可用的ResourceLoader

- **DefaultResourceLoader**

  ResourceLoader有一个默认的实现类，即org.springframework.core.io.DefaultResourceLoader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，(a)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；(b)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。

  下面来看一下DefaultResourceLoader的行为是如何反应到程序中的：

  ```
  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();
}
```

- **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， 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);
}
```

1.2.2 ResourcePatternResolver——批量查找的ResourceLoader

ResourcePatternResolver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。

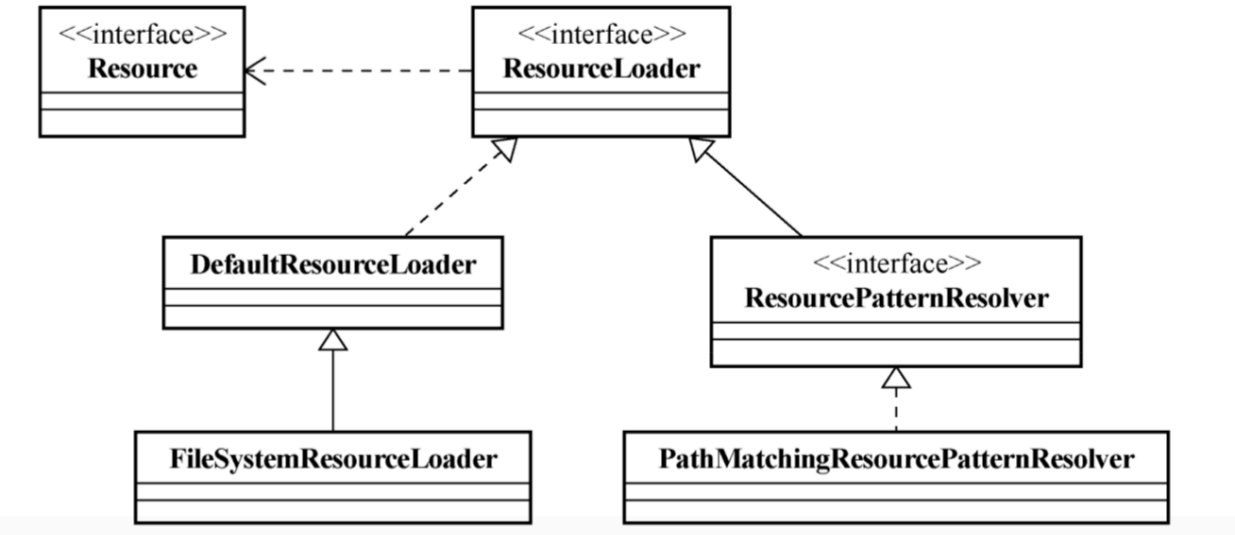
接口 org.springframework.core.io.support.ResourcePatternResolver定义如下：

```
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。

1.2.3 回顾与展望

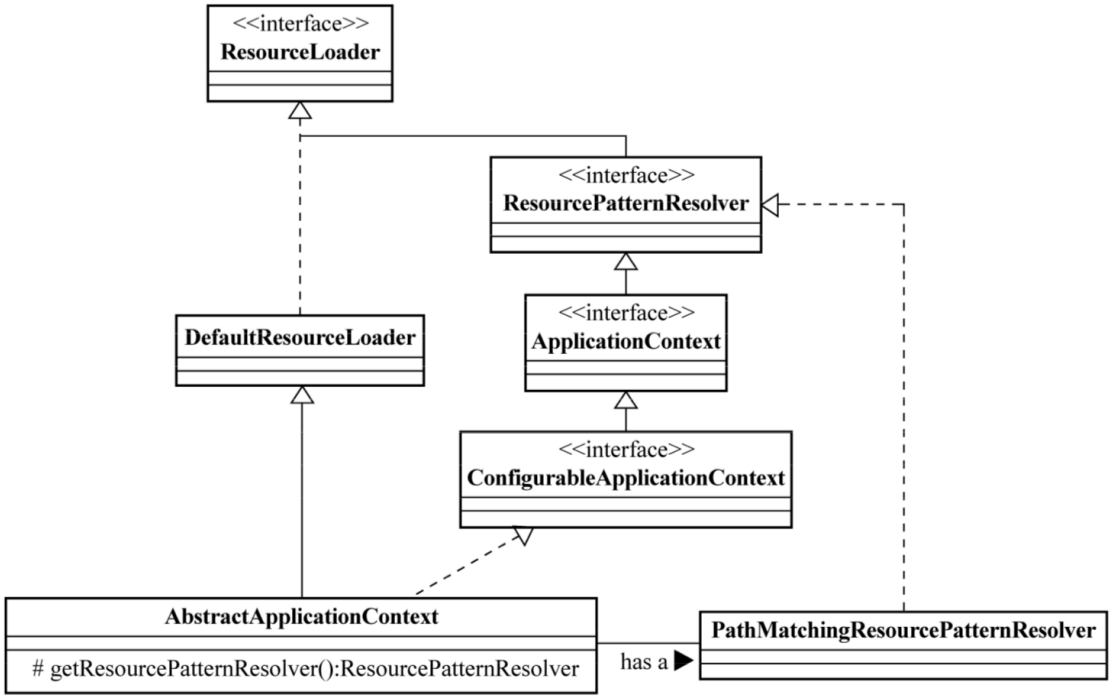
Resource和ResourceLoader类层次图：



1.3 ApplicationContext与ResourceLoader

ApplicationContext继承了ResourcePatternResolver，当然就间接实现了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。

AbstractApplicationContext作为ResourceLoader和ResourcePatternResolver：



下面看看作为ResourceLoader或者ResourcePatternResolver的ApplicationContext，到底因此拥有了何等神通。

- **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);
```

- **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。不过，我们也可以考虑用一下Spring提供的便利：ResourceLoaderAware和ApplicationContextAware接口。

假设一个类出于某种目的需要依赖于ResourceLoader，我们可以直接将当前的ApplicationContext容器作为ResourceLoader注入，ResourceLoaderAware和ApplicationContextAware接口可以帮助我们做到一点，只需让该类实现ResourceLoaderAware或ApplicationContextAware接口，容器启动的时候，就会自动将当前ApplicationContext容器本身注入到该类中，因为ApplicationContext类型容器可以自动识别Aware接口。

- **Resource类型的注入**

  ApplicationContext容器可以正确识别Resource类型并转换后注入相关对象。看一个例子：

```
public class XMailer {
    private Resource template;//声明模板为Resource类型
    public void sendMail(Map mailCtx) {
        // String mailContext =
merge(getTemplate().getInputStream(),mailCtx);
    public Resource getTemplate() {
        return template;
    }
    public void setTemplate(Resource template) {
        this.template = template;
    }
}
```

  配置内容如下：

```
<bean id="mailer" class="...XMailer"> <property name="template"
value="..resources.default_template.vm"/>
...
</bean>
```

  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所在的位置，ApplicationContext就可以正确地转换类型并注入依赖。

- **在特定情况下，ApplicationContext的Resource加载行为**

特定的ApplicationContext容器实现，在作为ResourceLoader加载资源时，会有其特定的行为。 我们下面主要讨论两种类型的ApplicationContext容器，即ClassPathXmlApplicationContext和 FileSystemXmlApplicationContext。

我们知道，对于URL所接受的资源路径来说，通常开始都会有一个协议前缀，比如file:、http:、ftp: 等。既然Spring使用UrlResource对URL定位查找的资源进行了抽象，那么，同样也支持这样类型的资源路径，而且，在这个基础上，Spring还扩展了协议前缀的集合。ResourceLoader中增加了一种新的资源路径协议——classpath:，ResourcePatternResolver又增加了一种——classpath*:。这 样，我们就可以通过这些资源路径协议前缀，明确地告知Spring容器要从classpath中加载资源，如下所示：

```
//代码中使用协议前缀
ResourceLoader resourceLoader = new
FileSystemXmlApplicationContext("classpath:conf/container-
conf.xml");
```

```
<bean id="..." class="...">
    <property name="...">
        <value>classpath:resource/template.vm</value>
    </property>
</bean>
```

classpath*:与classpath:的唯一区别就在于，如果能够在classpath中找到多个指定的资源则返回多个。ClassPathXmlApplicationContext和FileSystemXmlApplicationContext在处理资源加载的。默认行为上有所不同。 当ClassPathXmlApplicationContext 在实例化的时候， 即使没有指明 classpath:或者classpath*:等前缀，它会默认从classpath中加载bean定义配置文件。

未完待续。。。。。。