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
2019-02-04
Spring

【死磕 Spring】—— IoC 之 Spring 统一资源加载策略

本文主要基于 Spring 5.0.6.RELEASE

摘要: 原创出处 http://cmsblogs.com/?p=2656 「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。

在学 Java SE 的时候，我们学习了一个标准类 java.net.URL，该类在 Java SE 中的定位为统一资源定位器（Uniform Resource Locator），但是我们知道它的实现基本只限于网络形式发布的资源的查找和定位。然而，实际上资源的定义比较广泛，除了网络形式的资源，还有以二进制形式存在的、以文件形式存在的、以字节流形式存在的等等。而且它可以存在于任何场所，比如网络、文件系统、应用程序中。所以 java.net.URL 的局限性迫使 Spring 必须实现自己的资源加载策略，该资源加载策略需要满足如下要求：

1. 职能划分清楚。资源的定义和资源的加载应该要有一个清晰的界限；
2. 统一的抽象。统一的资源定义和资源加载策略。资源加载后要返回统一的抽象给客户端，客户端要对资源进行怎样的处理，应该由抽象资源接口来界定。

1. 统一资源：Resource

org.springframework.core.io.Resource 为 Spring 框架所有资源的抽象和访问接口，它继承 org.springframework.core.io.InputStreamSource接口。作为所有资源的统一抽象，Resource 定义了一些通用的方法，由子类 AbstractResource 提供统一的默认实现。定义如下：

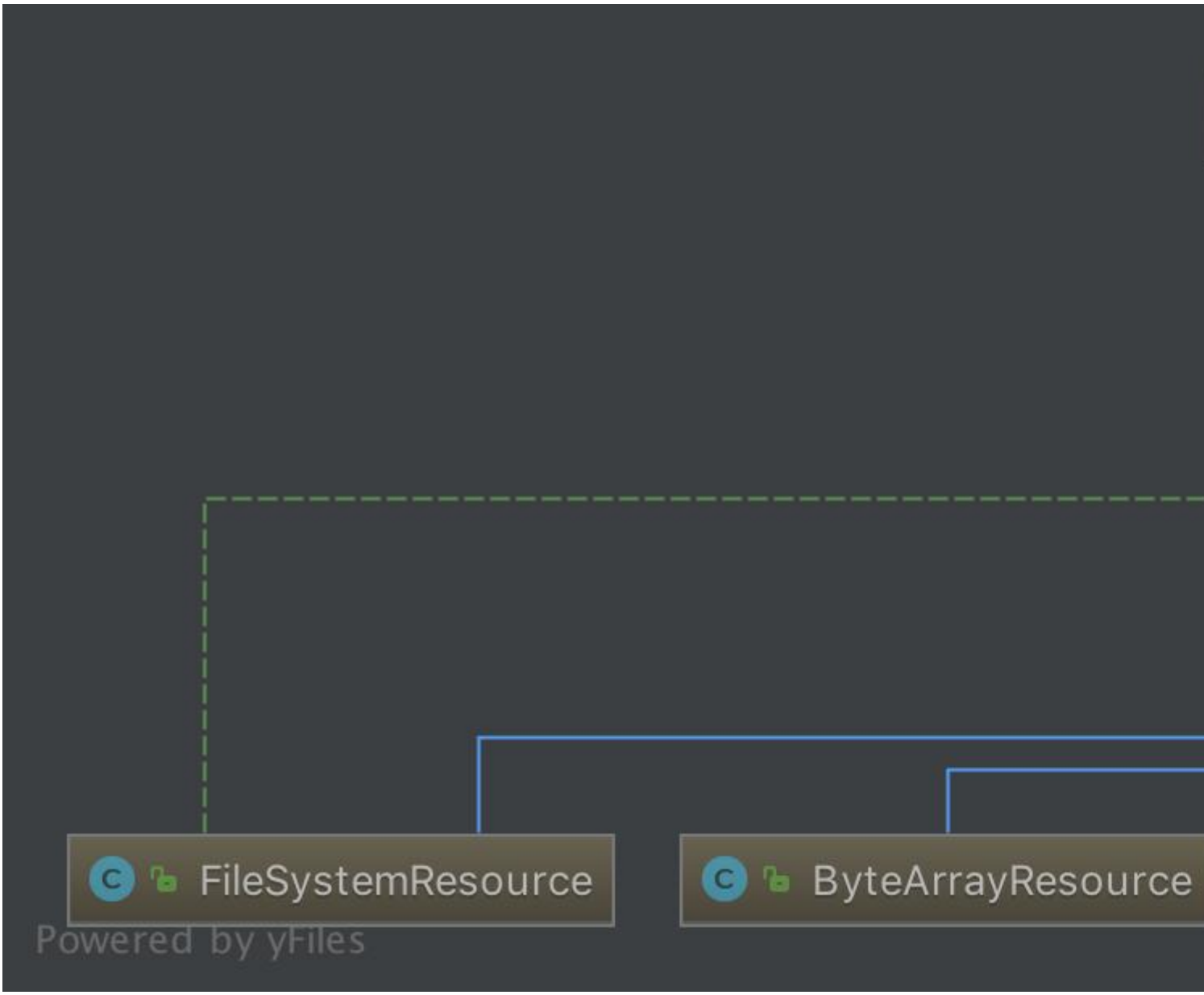
```
public interface Resource extends InputStreamSource {

 /**
     * 资源是否存在
     */
 boolean exists();

 /**
     * 资源是否可读
     */
```

```
default boolean isReadable() {
    return true;
   }

/**
    * 资源所代表的句柄是否被一个 stream 打开了
    */
default boolean isOpen() {
    return false;
   }

/**
    * 是否为 File
    */
default boolean isFile() {
    return false;
   }

/**
    * 返回资源的 URL 的句柄
    */
URL getURL() throws IOException;

/**
    * 返回资源的 URI 的句柄
    */
URI getURI() throws IOException;

/**
    * 返回资源的 File 的句柄
    */
File getFile() throws IOException;

/**
    * 返回 ReadableByteChannel
    */
default ReadableByteChannel readableChannel() throws IOException {
    return java.nio.channels.Channels.newChannel(getInputStream());
   }

/**
    * 资源内容的长度
    */
long contentLength() throws IOException;

/**
    * 资源最后的修改时间
    */
long lastModified() throws IOException;

/**
    * 根据资源的相对路径创建新资源
    */
Resource createRelative(String relativePath) throws IOException;

/**
    * 资源的文件名
    */
@Nullable
String getFilename();
```

```
    /**
     * 资源的描述
     */
    String getDescription();

}
```

1.1 子类结构

类结构图如下：



从上图可以看到，Resource 根据资源的不同类型提供不同的具体实现，如下：

- FileSystemResource ： 对 java.io.File 类型资源的封装，只要是跟 File 打交道的，基本上与 FileSystemResource 也可以打交道。支持文件和 URL 的形式，实现 WritableResource 接口，且从 Spring Framework 5.0 开始，FileSystemResource 使用 NIO2 API进行读/写交互。
- ByteArrayResource ： 对字节数组提供的数据的封装。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一个相应的 ByteArrayInputStream。
UrlResource ： 对 java.net.URL类型资源的封装。内部委派 URL 进行具体的资源操作。
ClassPathResource ： class path 类型资源的实现。使用给定的 ClassLoader 或者给定的 Class 来加载资源。
InputStreamResource ： 将给定的 InputStream 作为一种资源的 Resource 的实现类。

1.2 AbstractResource

org.springframework.core.io.AbstractResource ，为 Resource 接口的默认抽象实现。它实现了 Resource 接口的大部分的公共实现，作为 Resource 接口中的重中之重，其定义如下：

```
public abstract class AbstractResource implements Resource {

 /**
     * 判断文件是否存在，若判断过程产生异常（因为会调用SecurityManager来判断），就关闭对应的流
     */
 @Override
 public boolean exists() {
     try {
          // 基于 File 进行判断
         return getFile().exists();
        }
     catch (IOException ex) {
         // Fall back to stream existence: can we open the stream?
         // 基于 InputStream 进行判断
         try {
                InputStream is = getInputStream();
                is.close();
            return true;
           } catch (Throwable isEx) {
            return false;
           }
        }
    }

 /**
     * 直接返回true，表示可读
     */
 @Override
 public boolean isReadable() {
     return true;
    }

 /**
     * 直接返回 false，表示未被打开
     */
 @Override
 public boolean isOpen() {
     return false;
    }

 /**
     * 直接返回false，表示不为 File
     */
 @Override
 public boolean isFile() {
     return false;
    }
```

```
/**
    * 抛出 FileNotFoundException 异常，交给子类实现
    */
@Override
public URL getURL() throws IOException {
    throw new FileNotFoundException(getDescription() + " cannot be resolved to URL");

   }

/**
    * 基于 getURL() 返回的 URL 构建 URI
    */
@Override
public URI getURI() throws IOException {
       URL url = getURL();
    try {
        return ResourceUtils.toURI(url);
       } catch (URISyntaxException ex) {
        throw new NestedIOException("Invalid URI [" + url + "]", ex);
       }
   }

/**
    * 抛出 FileNotFoundException 异常，交给子类实现
    */
@Override
public File getFile() throws IOException {
    throw new FileNotFoundException(getDescription() + " cannot be resolved to absolute file path");
   }

/**
    * 根据 getInputStream() 的返回结果构建 ReadableByteChannel
    */
@Override
public ReadableByteChannel readableChannel() throws IOException {
    return Channels.newChannel(getInputStream());
   }

/**
    * 获取资源的长度
    *
    * 这个资源内容长度实际就是资源的字节长度，通过全部读取一遍来判断
    */
@Override
public long contentLength() throws IOException {
       InputStream is = getInputStream();
    try {
        long size = 0;
        byte[] buf = new byte[255]; // 每次最多读取 255 字节
        int read;
        while ((read = is.read(buf)) != -1) {
               size += read;
           }
        return size;
       } finally {
        try {
               is.close();
           } catch (IOException ex) {
           }
```

```
        }
    }

    /**
     * 返回资源最后的修改时间
     */
    @Override
    public long lastModified() throws IOException {
        long lastModified = getFileForLastModifiedCheck().lastModified();
        if (lastModified == 0L) {
            throw new FileNotFoundException(getDescription() +
                    " cannot be resolved in the file system for resolving its last-modified timestamp");
        }
        return lastModified;
    }

    protected File getFileForLastModifiedCheck() throws IOException {
        return getFile();
    }

    /**
     * 抛出 FileNotFoundException 异常，交给子类实现
     */
    @Override
    public Resource createRelative(String relativePath) throws IOException {
        throw new FileNotFoundException("Cannot create a relative resource for " + getDescription());
    }

    /**
     * 获取资源名称，默认返回 null ，交给子类实现
     */
    @Override
    @Nullable
    public String getFilename() {
        return null;
    }

    /**
     * 返回资源的描述
     */
    @Override
    public String toString() {
        return getDescription();
    }

    @Override
    public boolean equals(Object obj) {
        return (obj == this ||
                (obj instanceof Resource && ((Resource) obj).getDescription().equals(getDescription())));
    }

    @Override
    public int hashCode() {
        return getDescription().hashCode();
    }

}
```

如果我们想要实现自定义的 Resource ，记住不要实现 Resource 接口，而应该继承 AbstractResource 抽象类，然后根据当前的具体资源特性覆盖相应的方法即可。

1.3 其他子类

来自芳芳

Resource 的子类，例如 FileSystemResource、ByteArrayResource 等等的代码非常简单。感兴趣的胖友，自己去研究。

2. 统一资源定位：ResourceLoader

一开始就说了 Spring 将资源的定义和资源的加载区分开了，Resource 定义了统一的资源，那资源的加载则由 ResourceLoader 来统一定义。

org.springframework.core.io.ResourceLoader 为 Spring 资源加载的统一抽象，具体的资源加载则由相应的实现类来完成，所以我们可以将 ResourceLoader 称作为统一资源定位器。其定义如下：

FROM 《Spring 源码深度解析》P16 页

ResourceLoader，定义资源加载器，主要应用于根据给定的资源文件地址，返回对应的 Resource 。

```
public interface ResourceLoader {

    String CLASSPATH_URL_PREFIX = ResourceUtils.CLASSPATH_URL_PREFIX; // CLASSPATH URL 前缀。默认为："classpath:"

 Resource getResource(String location);

 ClassLoader getClassLoader();

}
```

#getResource(String location) 方法，根据所提供资源的路径 location 返回 Resource 实例，但是它不确保该 Resource 一定存在，需要调用 Resource#exist() 方法来判断。

- 该方法支持以下模式的资源加载：
  - URL位置资源，如 "file:C:/test.dat" 。
  - ClassPath位置资源，如 "classpath:test.dat 。
  - 相对路径资源，如 "WEB-INF/test.dat" ，此时返回的Resource 实例，根据实现不同而不同。
- 该方法的主要实现是在其子类 DefaultResourceLoader 中实现，具体过程我们在分析 DefaultResourceLoader 时做详细说明。

#getClassLoader() 方法，返回 ClassLoader 实例，对于想要获取 ResourceLoader 使用的 ClassLoader 用户来说，可以直接调用该方法来获取。在分析 Resource 时，提到了一个类 ClassPathResource ，这个类是可以根据指定的 ClassLoader 来加载资源的。

2.1 子类结构

作为 Spring 统一的资源加载器，它提供了统一的抽象，具体的实现则由相应的子类来负责实现，其类的类结构图如下：

2.1 DefaultResourceLoader

与 AbstractResource 相似，org.springframework.core.io.DefaultResourceLoader 是 ResourceLoader 的默认实现。

2.1.1 构造函数

它接收 ClassLoader 作为构造函数的参数，或者使用不带参数的构造函数。

> 在使用不带参数的构造函数时，使用的 ClassLoader 为默认的 ClassLoader（一般 Thread.currentThread()#getContextClassLoader() ）。
> 在使用带参数的构造函数时，可以通过 ClassUtils#getDefaultClassLoader()获取。

代码如下：

```
@Nullable
private ClassLoader classLoader;

public DefaultResourceLoader() { // 无参构造函数
 this.classLoader = ClassUtils.getDefaultClassLoader();
}

public DefaultResourceLoader(@Nullable ClassLoader classLoader) { // 带 ClassLoader 参数的构造函数
 this.classLoader = classLoader;
}

public void setClassLoader(@Nullable ClassLoader classLoader) {
 this.classLoader = classLoader;
}

@Override
@Nullable
public ClassLoader getClassLoader() {
 return (this.classLoader != null ? this.classLoader : ClassUtils.getDefaultClassLoader());
}
```

> 另外，也可以调用 #setClassLoader() 方法进行后续设置。

2.1.2 getResource 方法

ResourceLoader 中最核心的方法为 #getResource(String location) ，它根据提供的 location 返回相应的 Resource 。而 DefaultResourceLoader 对该方法提供了核心实现（因为，它的两个子类都没有提供覆盖该方法，所以可以断定 ResourceLoader 的资源加载策略就封装在 DefaultResourceLoader 中），代码如下：

```
// DefaultResourceLoader.java

@Override
public Resource getResource(String location) {
    Assert.notNull(location, "Location must not be null");

    // 首先，通过 ProtocolResolver 来加载资源
    for (ProtocolResolver protocolResolver : this.protocolResolvers) {
        Resource resource = protocolResolver.resolve(location, this);
```

```
            if (resource != null) {
                return resource;
            }
        }
        // 其次，以 / 开头，返回 ClassPathContextResource 类型的资源
        if (location.startsWith("/")) {
            return getResourceByPath(location);
        // 再次，以 classpath: 开头，返回 ClassPathResource 类型的资源
        } else if (location.startsWith(CLASSPATH_URL_PREFIX)) {
            return new ClassPathResource(location.substring(CLASSPATH_URL_PREFIX.length()), getClassLoader());
        // 然后，根据是否为文件 URL ，是则返回 FileUrlResource 类型的资源，否则返回 UrlResource 类型的资源
        } else {
            try {
                // Try to parse the location as a URL...
                URL url = new URL(location);
                return (ResourceUtils.isFileURL(url) ? new FileUrlResource(url) : new UrlResource(url));
            } catch (MalformedURLException ex) {
                // 最后，返回 ClassPathContextResource 类型的资源
                // No URL -> resolve as resource path.
                return getResourceByPath(location);
            }
        }
    }
```

首先，通过 ProtocolResolver 来加载资源，成功返回 Resource 。

其次，若 location 以 "/" 开头，则调用 #getResourceByPath() 方法，构造 ClassPathContextResource 类型资源并返回。代码如下：

```
    protected Resource getResourceByPath(String path) {
     return new ClassPathContextResource(path, getClassLoader());
    }
```

再次，若 location 以 "classpath:" 开头，则构造 ClassPathResource 类型资源并返回。在构造该资源时，通过 #getClassLoader() 获取当前的 ClassLoader。
然后，构造 URL ，尝试通过它进行资源定位，若没有抛出 MalformedURLException 异常，则判断是否为 FileURL ，如果是则构造 FileUrlResource 类型的资源，否则构造 UrlResource 类型的资源。
最后，若在加载过程中抛出 MalformedURLException 异常，则委派 #getResourceByPath() 方法，实现资源定位加载。　　实际上，和【其次】相同落。

2.1.3 ProtocolResolver

org.springframework.core.io.ProtocolResolver ，用户自定义协议资源解决策略，作为 DefaultResourceLoader 的 SPI：它允许用户自定义资源加载协议，而不需要继承 ResourceLoader 的子类。
在介绍 Resource 时，提到如果要实现自定义 Resource，我们只需要继承 AbstractResource 即可，但是有了 ProtocolResolver 后，我们不需要直接继承 DefaultResourceLoader，改为实现 ProtocolResolver 接口也可以实现自定义的 ResourceLoader。

ProtocolResolver 接口，仅有一个方法 Resource resolve(String location, ResourceLoader resourceLoader) 。代码如下：

```
/**
 * 使用指定的 ResourceLoader ，解析指定的 location 。
 * 若成功，则返回对应的 Resource 。
 *
 * Resolve the given location against the given resource loader
 * if this implementation's protocol matches.
 * @param location the user-specified resource location 资源路径
 * @param resourceLoader the associated resource loader 指定的加载器 ResourceLoader
 * @return a corresponding {@code Resource} handle if the given location
 * matches this resolver's protocol, or {@code null} otherwise 返回为相应的 Resource
 */
@Nullable
Resource resolve(String location, ResourceLoader resourceLoader);
```

在 Spring 中你会发现该接口并没有实现类，它需要用户自定义，自定义的 Resolver 如何加入 Spring 体系呢？调用 DefaultResourceLoader#addProtocolResolver(ProtocolResolver) 方法即可。代码如下：

```
/**
 * ProtocolResolver 集合
 */
private final Set<ProtocolResolver> protocolResolvers = new LinkedHashSet<>(4);

public void addProtocolResolver(ProtocolResolver resolver) {
    Assert.notNull(resolver, "ProtocolResolver must not be null");
 this.protocolResolvers.add(resolver);
}
```

2.1.4 示例

下面示例是演示 DefaultResourceLoader 加载资源的具体策略，代码如下（该示例参考《Spring 揭秘》 P89）：

```
ResourceLoader resourceLoader = new DefaultResourceLoader();

Resource fileResource1 = resourceLoader.getResource("D:/Users/chenming673/Documents/spark.txt");
System.out.println("fileResource1 is FileSystemResource:" + (fileResource1 instanceof FileSystemResource));

Resource fileResource2 = resourceLoader.getResource("/Users/chenming673/Documents/spark.txt");
System.out.println("fileResource2 is ClassPathResource:" + (fileResource2 instanceof ClassPathResource));

Resource urlResource1 = resourceLoader.getResource("file:/Users/chenming673/Documents/spark.txt");
System.out.println("urlResource1 is UrlResource:" + (urlResource1 instanceof UrlResource));

Resource urlResource2 = resourceLoader.getResource("http://www.baidu.com");
System.out.println("urlResource1 is urlResource:" + (urlResource2 instanceof  UrlResource));
```

运行结果：

```
fileResource1 is FileSystemResource:false
fileResource2 is ClassPathResource:true
urlResource1 is UrlResource:true
urlResource1 is urlResource:true
```

其实对于 fileResource1 ，我们更加希望是 FileSystemResource 资源类型。但是，事与愿违，它是 ClassPathResource 类型。为什么呢？在 DefaultResourceLoader#getResource() 方法的资源加载策略中，我们知道 "D:/Users/chenming673/Documents/spark.txt" 地址，其实在该方法中没有相应的资源类型，那么它就会在抛出 MalformedURLException 异常时，通过 DefaultResourceLoader#getResourceByPath(...) 方法，构造一个 ClassPathResource 类型的资源。而 urlResource1 和 urlResource2 ，指定有协议前缀的资源路径，则通过 URL 就可以定义，所以返回的都是 UrlResource 类型。

2.2 FileSystemResourceLoader

从上面的示例，我们看到，其实 DefaultResourceLoader 对#getResourceByPath(String) 方法处理其实不是很恰当，这个时候我们可以使用 org.springframework.core.io.FileSystemResourceLoader 。它继承 DefaultResourceLoader ，且覆写了 #getResourceByPath(String) 方法，使之从文件系统加载资源并以 FileSystemResource 类型返回，这样我们就可以得到想要的资源类型。代码如下：

```
@Override
protected Resource getResourceByPath(String path) {
 // 截取首 /
 if (path.startsWith("/")) {
        path = path.substring(1);
    }
 // 创建 FileSystemContextResource 类型的资源
 return new FileSystemContextResource(path);
}
```

2.2.1 FileSystemContextResource

FileSystemContextResource ，为 FileSystemResourceLoader 的内部类，它继承 FileSystemResource 类，实现 ContextResource 接口。代码如下：

```
/**
 * FileSystemResource that explicitly expresses a context-relative path
 * through implementing the ContextResource interface.
 */
private static class FileSystemContextResource extends FileSystemResource implements ContextResource {

 public FileSystemContextResource(String path) {
     super(path);
    }

 @Override
 public String getPathWithinContext() {
     return getPath();
    }
}
```

在构造器中，也是调用 FileSystemResource 的构造函数来构造 FileSystemResource 的。为什么要有 FileSystemContextResource 类的原因是，实现 ContextResource 接口，并实现对应的 #getPathWithinContext() 接口方法。

2.2.2 示例

在回过头看 「2.1.4 示例」 ，如果将 DefaultResourceLoader 改为

FileSystemResourceLoader ，则 fileResource1 则为 FileSystemResource 类型的资源。

2.3 ClassRelativeResourceLoader

org.springframework.core.io.ClassRelativeResourceLoader ，是 DefaultResourceLoader 的另一个子类的实现。和 FileSystemResourceLoader 类似，在实现代码的结构上类似，也是覆写 #getResourceByPath(String path) 方法，并返回其对应的 ClassRelativeContextResource 的资源类型。

感兴趣的胖友，可以看看 《Spring5：就这一次，搞定资源加载器之 ClassRelativeResourceLoader》 文章。

> ClassRelativeResourceLoader 扩展的功能是，可以根据给定的class 所在包或者所在包的子包下加载资源。

2.4 ResourcePatternResolver

ResourceLoader 的 Resource getResource(String location) 方法，每次只能根据 location 返回一个 Resource 。当需要加载多个资源时，我们除了多次调用 #getResource(String location) 方法外，别无他法。org.springframework.core.io.support.ResourcePatternResolver 是 ResourceLoader 的扩展，它支持根据指定的资源路径匹配模式每次返回多个 Resource 实例，其定义如下：

```
public interface ResourcePatternResolver extends ResourceLoader {

    String CLASSPATH_ALL_URL_PREFIX = "classpath*:";

    Resource[] getResources(String locationPattern) throws IOException;

}
```

> ResourcePatternResolver 在 ResourceLoader 的基础上增加了 #getResources(String locationPattern) 方法，以支持根据路径匹配模式返回多个 Resource 实例。
> 同时，也新增了一种新的协议前缀 "classpath*:"，该协议前缀由其子类负责实现。

2.5 PathMatchingResourcePatternResolver

org.springframework.core.io.support.PathMatchingResourcePatternResolver ，为 ResourcePatternResolver 最常用的子类，它除了支持 ResourceLoader 和 ResourcePatternResolver 新增的 "classpath*:" 前缀外，还支持 Ant 风格的路径匹配模式（类似于 "**/*.xml"）。

2.5.1 构造函数

PathMatchingResourcePatternResolver 提供了三个构造函数，如下：

```
/**
 * 内置的 ResourceLoader 资源定位器
 */
private final ResourceLoader resourceLoader;
/**
 * Ant 路径匹配器
 */
private PathMatcher pathMatcher = new AntPathMatcher();
```

```
public PathMatchingResourcePatternResolver() {
 this.resourceLoader = new DefaultResourceLoader();
}

public PathMatchingResourcePatternResolver(ResourceLoader resourceLoader) {
    Assert.notNull(resourceLoader, "ResourceLoader must not be null");
 this.resourceLoader = resourceLoader;
}

public PathMatchingResourcePatternResolver(@Nullable ClassLoader classLoader) {
 this.resourceLoader = new DefaultResourceLoader(classLoader);
}
```

PathMatchingResourcePatternResolver 在实例化的时候，可以指定一个 ResourceLoader，如果不指定的话，它会在内部构造一个 DefaultResourceLoader 。pathMatcher 属性，默认为 AntPathMatcher 对象，用于支持 Ant 类型的路径匹配。

2.5.2 getResource

```
@Override
public Resource getResource(String location) {
 return getResourceLoader().getResource(location);
}

public ResourceLoader getResourceLoader() {
 return this.resourceLoader;
}
```

该方法，直接委托给相应的 ResourceLoader 来实现。所以，如果我们在实例化的 PathMatchingResourcePatternResolver 的时候，如果未指定 ResourceLoader 参数的情况下，那么在加载资源时，其实就是 DefaultResourceLoader 的过程。

其实在下面介绍的 Resource[] getResources(String locationPattern) 方法也相同，只不过返回的资源是多个而已。

2.5.3 getResources

```
@Override
public Resource[] getResources(String locationPattern) throws IOException {
    Assert.notNull(locationPattern, "Location pattern must not be null");
    // 以 "classpath*:" 开头
    if (locationPattern.startsWith(CLASSPATH_ALL_URL_PREFIX)) {
        // 路径包含通配符
        // a class path resource (multiple resources for same name possible)
        if (getPathMatcher().isPattern(locationPattern.substring(CLASSPATH_ALL_URL_PREFIX.length()))) {
            // a class path resource pattern
            return findPathMatchingResources(locationPattern);
        // 路径不包含通配符
        } else {
            // all class path resources with the given name
            return findAllClassPathResources(locationPattern.substring(CLASSPATH_ALL_URL_PREFIX.length()));
        }
    // 不以 "classpath*:" 开头
```

```
        } else {
            // Generally only look for a pattern after a prefix here, // 通常只在这里的前缀后面查找模式
            // and on Tomcat only after the "*/" separator for its "war:" protocol. 而在 Tomcat 上只有在 “*/ ”分隔符之后
            int prefixEnd = (locationPattern.startsWith("war:") ? locationPattern.indexOf("*/") + 1 :
                    locationPattern.indexOf(':') + 1);
            // 路径包含通配符
            if (getPathMatcher().isPattern(locationPattern.substring(prefixEnd))) {
                // a file pattern
                return findPathMatchingResources(locationPattern);
            // 路径不包含通配符
            } else {
                // a single resource with the given name
                return new Resource[] {getResourceLoader().getResource(locationPattern)};
            }
        }
    }
```

处理逻辑如下图：

非 "classpath*:" 开头，且路径不包含通配符，直接委托给相应的 ResourceLoader 来实现。其他情况，调用 #findAllClassPathResources(...)、或 #findPathMatchingResources(...) 方法，返回多个 Resource 。下面，我们来详细分析。

2.5.4 findAllClassPathResources

当 locationPattern 以 "classpath*:" 开头但是不包含通配符，则调用 #findAllClassPathResources(...) 方法加载资源。该方法返回 classes 路径下和所有 jar 包中的所有相匹配的资源。

```
protected Resource[] findAllClassPathResources(String location) throws IOException {
    String path = location;
 // 去除首个 /
 if (path.startsWith("/")) {
        path = path.substring(1);
    }
 // 真正执行加载所有 classpath 资源
    Set<Resource> result = doFindAllClassPathResources(path);
 if (logger.isTraceEnabled()) {
        logger.trace("Resolved classpath location [" + location + "] to resources " + result);
    }
 // 转换成 Resource 数组返回
 return result.toArray(new Resource[0]);
}
```

真正执行加载的是在 #doFindAllClassPathResources(...) 方法，代码如下：

```
protected Set<Resource> doFindAllClassPathResources(String path) throws IOException {
    Set<Resource> result = new LinkedHashSet<>(16);
    ClassLoader cl = getClassLoader();
 // <1> 根据 ClassLoader 加载路径下的所有资源
    Enumeration<URL> resourceUrls = (cl != null ? cl.getResources(path) : ClassLoader.getSystemResources(path));
 // <2>
 while (resourceUrls.hasMoreElements()) {
        URL url = resourceUrls.nextElement();
     // 将 URL 转换成 UrlResource
```

```
            result.add(convertClassLoaderURL(url));
        }
    // <3> 加载路径下得所有 jar 包
    if ("".equals(path)) {
        // The above result is likely to be incomplete, i.e. only containing file system references.
        // We need to have pointers to each of the jar files on the classpath as well...
            addAllClassLoaderJarRoots(cl, result);
        }
    return result;
}
```

<1> 处，根据 ClassLoader 加载路径下的所有资源。在加载资源过程时，如果在构造 PathMatchingResourcePatternResolver 实例的时候如果传入了 ClassLoader，则调用该 ClassLoader 的 #getResources() 方法，否则调用 ClassLoader#getSystemResources(path) 方法。另外，ClassLoader#getResources() 方法，代码如下：

```
// java.lang.ClassLoader.java
public Enumeration<URL> getResources(String name) throws IOException {
    @SuppressWarnings("unchecked")
    Enumeration<URL>[] tmp = (Enumeration<URL>[]) new Enumeration<?>[2];
    if (parent != null) {
        tmp[0] = parent.getResources(name);
    } else {
        tmp[0] = getBootstrapResources(name);
    }
    tmp[1] = findResources(name);

    return new CompoundEnumeration<>(tmp);
}
```

- 看到这里是不是就已经一目了然了？如果当前父类加载器不为 null ，则通过父类向上迭代获取资源，否则调用 #getBootstrapResources() 。这里是不是特别熟悉，(^▽^)。

<2> 处，遍历 URL 集合，调用 #convertClassLoaderURL(URL url) 方法，将 URL 转换成 UrlResource 对象。代码如下：

```
protected Resource convertClassLoaderURL(URL url) {
 return new UrlResource(url);
}
```

<3> 处，若 path 为空（“”）时，则调用 #addAllClassLoaderJarRoots(...)方法。该方法主要是加载路径下得所有 jar 包，方法较长也没有什么实际意义就不贴出来了。感兴趣的胖友，自己可以去看看。　　当然，可能代码也比较长哈。

通过上面的分析，我们知道 #findAllClassPathResources(...) 方法，其实就是利用 ClassLoader 来加载指定路径下的资源，不论它是在 class 路径下还是在 jar 包中。如果我们传入的路径为空或者 /，则会调用 #addAllClassLoaderJarRoots(...) 方法，加载所有的 jar 包。

2.5.5 findPathMatchingResources

当 locationPattern 中包含了通配符，则调用该方法进行资源加载。代码如下：

```
protected Resource[] findPathMatchingResources(String locationPattern) throws IOException {
    // 确定根路径、子路径
    String rootDirPath = determineRootDir(locationPattern);
    String subPattern = locationPattern.substring(rootDirPath.length());
    // 获取根据路径下的资源
    Resource[] rootDirResources = getResources(rootDirPath);
    // 遍历，迭代
    Set<Resource> result = new LinkedHashSet<>(16);
    for (Resource rootDirResource : rootDirResources) {
        rootDirResource = resolveRootDirResource(rootDirResource);
        URL rootDirUrl = rootDirResource.getURL();
        // bundle 资源类型
        if (equinoxResolveMethod != null && rootDirUrl.getProtocol().startsWith("bundle")) {
            URL resolvedUrl = (URL) ReflectionUtils.invokeMethod(equinoxResolveMethod, null, rootDirUrl);
            if (resolvedUrl != null) {
                rootDirUrl = resolvedUrl;
            }
            rootDirResource = new UrlResource(rootDirUrl);
        }
        // vfs 资源类型
        if (rootDirUrl.getProtocol().startsWith(ResourceUtils.URL_PROTOCOL_VFS)) {
            result.addAll(VfsResourceMatchingDelegate.findMatchingResources(rootDirUrl, subPattern, getPathMatcher()))
        // jar 资源类型
        } else if (ResourceUtils.isJarURL(rootDirUrl) || isJarResource(rootDirResource)) {
            result.addAll(doFindPathMatchingJarResources(rootDirResource, rootDirUrl, subPattern));
        // 其它资源类型
        } else {
            result.addAll(doFindPathMatchingFileResources(rootDirResource, subPattern));
        }
    }
    if (logger.isTraceEnabled()) {
        logger.trace("Resolved location pattern [" + locationPattern + "] to resources " + result);
    }
    // 转换成 Resource 数组返回
    return result.toArray(new Resource[0]);
}
```

方法有点儿长，但是思路还是很清晰的，主要分两步：

1. 确定目录，获取该目录下得所有资源。
2. 在所获得的所有资源后，进行迭代匹配获取我们想要的资源。

在这个方法里面，我们要关注两个方法，一个是 `#determineRootDir(String location)` 方法，一个是 `#doFindPathMatchingXXXResources(...)` 等方法。

2.5.5.1 determineRootDir

`determineRootDir(String location)` 方法，主要是用于确定根路径。代码如下：

```
/**
 * Determine the root directory for the given location.
 * <p>Used for determining the starting point for file matching,
 * resolving the root directory location to a {@code java.io.File}
 * and passing it into {@code retrieveMatchingFiles}, with the
 * remainder of the location as pattern.
```

```
     * <p>Will return "/WEB-INF/" for the pattern "/WEB-INF/*.xml",
     * for example.
     * @param location the location to check
     * @return the part of the location that denotes the root directory
     * @see #retrieveMatchingFiles
     */
    protected String determineRootDir(String location) {
     // 找到冒号的后一位
     int prefixEnd = location.indexOf(':') + 1;
     // 根目录结束位置
     int rootDirEnd = location.length();
     // 在从冒号开始到最后的字符串中，循环判断是否包含通配符，如果包含，则截断最后一个由”/”分割的部分。
     // 例如：在我们路径中，就是最后的ap?-context.xml这一段。再循环判断剩下的部分，直到剩下的路径中都不包含通配符。
     while (rootDirEnd > prefixEnd && getPathMatcher().isPattern(location.substring(prefixEnd, rootDirEnd))) {
            rootDirEnd = location.lastIndexOf('/', rootDirEnd - 2) + 1;
        }
     // 如果查找完成后，rootDirEnd = 0 了，则将之前赋值的 prefixEnd 的值赋给 rootDirEnd ，也就是冒号的后一位
     if (rootDirEnd == 0) {
            rootDirEnd = prefixEnd;
        }
     // 截取根目录
     return location.substring(0, rootDirEnd);
    }
```

方法比较绕，效果如下示例：

原路径	确定根路径
classpath*:test/cc*/spring-*.xml	classpath*:test/
classpath*:test/aa/spring-*.xml	classpath*:test/aa/

2.5.5.2 doFindPathMatchingXXXResources

来自芳芳

`#doFindPathMatchingXXXResources(...)` 方法，是个泛指，一共对应三个方法：

- `#doFindPathMatchingJarResources(rootDirResource, rootDirUrl, subPatter)` 方法
- `#doFindPathMatchingFileResources(rootDirResource, subPattern)` 方法
- `VfsResourceMatchingDelegate#findMatchingResources(rootDirUrl, subPattern, pathMatcher)` 方法

因为本文重在分析 Spring 统一资源加载策略的整体流程。相对来说，上面几个方法的代码量会比较多。所以本文不再追溯，感兴趣的胖友，推荐阅读如下文章：

- 《Spring源码情操陶冶-PathMatchingResourcePatternResolver路径资源匹配溶解器》，主要针对 `#doFindPathMatchingJarResources(rootDirResource, rootDirUrl, subPatter)` 方法。
- 《深入 Spring IoC 源码之 ResourceLoader》，主要针对 `#doFindPathMatchingFileResources(rootDirResource, subPattern)` 方法。
- 《Spring 源码学习 —— 含有通配符路径解析（上）》 貌似没有下

3. 小结

至此 Spring 整个资源记载过程已经分析完毕。下面简要总结下：

Spring 提供了 Resource 和 ResourceLoader 来统一抽象整个资源及其定位。使得资源与资源的定位有了一个更加清晰的界限，并且提供了合适的 Default 类，使得自定义实现更加方便和清晰。
AbstractResource 为 Resource 的默认抽象实现，它对 Resource 接口做了一个统一的实现，子类继承该类后只需要覆盖相应的方法即可，同时对于自定义的 Resource 我们也是继承该类。
DefaultResourceLoader 同样也是 ResourceLoader 的默认实现，在自定 ResourceLoader 的时候我们除了可以继承该类外还可以实现 ProtocolResolver 接口来实现自定资源加载协议。
DefaultResourceLoader 每次只能返回单一的资源，所以 Spring 针对这个提供了另外一个接口 ResourcePatternResolver ，该接口提供了根据指定的 locationPattern 返回多个资源的策略。其子类 PathMatchingResourcePatternResolver 是一个集大成者的 ResourceLoader ，因为它即实现了 Resource getResource(String location) 方法，也实现了 Resource[] getResources(String locationPattern) 方法。

另外，如果胖友认真的看了本文的包结构，我们可以发现，Resource 和 ResourceLoader 核心是在 ，spring-core 项目中。

如果想要调试本小节的相关内容，可以直接使用 Resource 和 ResourceLoader 相关的 API ，进行操作调试。

文章目录