SpringBoot 核心模块原理剖析

前言

最近微服务很火，SpringBoot 则以其轻量级、内嵌 Web 容器、一键启动、方便调试等特点被越来越多的微服务实践者所采用。然而知其然还要知其所以然，本文就来讲解 SpringBoot 中的三大核心模块的实现原理。

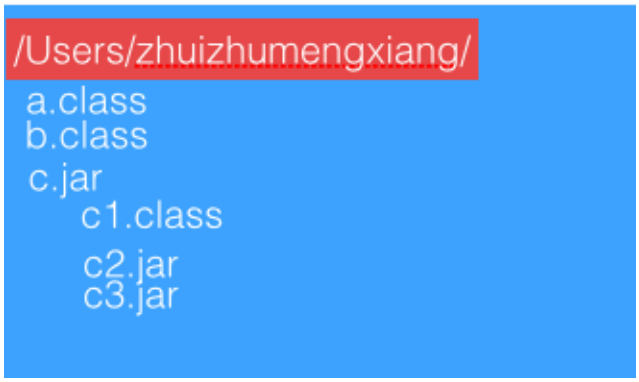
这三大核心模块为：

- **spring-boot-load 模块：**正常情况下一个类加载器只能找到加载路径的 jar 包里当前目录或者文件类里面的 *.class 文件，SpringBoot 允许我们使用 java -jar archive.jar 运行包含嵌套依赖 jar 的 jar 或者 war 文件。
- **spring-boot-autoconfigure 模块：**Spring的出现给我们管理 Bean 的依赖注入提供了便捷，但是当我们需要使用通过 pom 引入的 jar 里面的一个 Bean 时候，还是需要手动在 XML 配置文件里面配置。Springboot 则可以依据 classpath 里面的依赖内容自动配置 Bean 到 Spring 容器。
- **spring-boot 模块：**提供了一些特性用来支持 SpringBoot 中其它模块，本文会讲解到该模块都提供了哪些功能以及实现原理。

spring-boot-loader 模块

Java 原生类加载器局限性及改进思路

Java 中每种 ClassLoader 都会去自己规定的路径下查找字节码文件并加载到内存（可以参考《Java 类加载器揭秘》这场 Chat）。这里需要补充的是 Classloader 只能加载扫描路径当前目录或者当前目录文件夹下的 .class 文件，或当前目录文件夹下 jar 文件里面的.class文件。如果这个 jar 里面又嵌套了其他 jar 包文件，那么这些嵌套 jar 里面的 *.class 文件是不会被 ClassLoader 加载的。



如上图，假设类加载器 cl 扫描字节码文件路径 /Users/zhuizhumengxiang，那么 cl 可以加载到 a.class、b.class 和 c.jar 里面的 c1.class 文件，但是加载不到 c2.jar 和 c3.jar 里面的 .class 文件，因为 c2.jar 和 c3.jar 是嵌套 jar。

为了能够加载嵌套 jar 里面的资源，之前的做法都是把嵌套 jar 里面的 class 文件和应用的 class 文件打包为一个 jar，这样就不存在嵌套 jar 了，但是这样做就不能很清晰的知道哪些是应用自己的，哪些是应用依赖的，另外多个嵌套 jar 里面的 class 文件可能内容不一样但是文件名却一样时候又会引发新的问题。

spring-boot-loader 模块则允许我们使用 java -jar archive.jar 方式运行包含嵌套依赖 jar 的 jar 或者 war 文件，它提供了三种类启动器（JarLauncher、WarLauncher 和 PropertiesLauncher），这些类启动器的目的都是为了能够加载嵌套在 jar 里面的资源（比如 class 文件、配置文件等）。[Jar|War]Launcher 固定去查找当前 jar 的 lib 目录里面的嵌套 jar 文件里面的资源。本文则只介绍 jar 文件。

那么我们可以先思考下，如果让我们自己做一个可以加载嵌套 jar 里面的资源的工具模块，我们会怎么做呢？

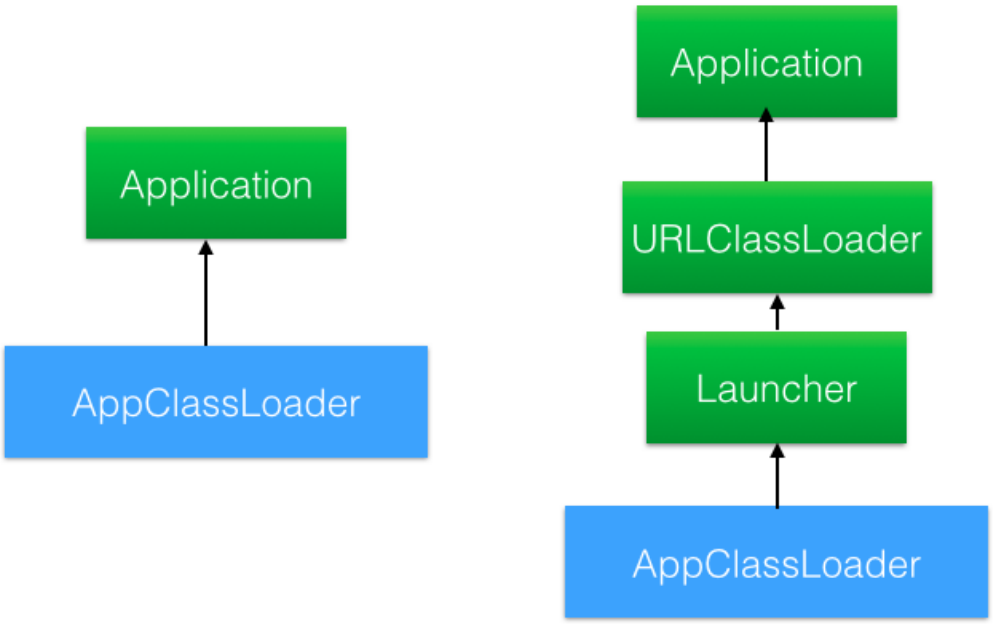
我们知道 Java 中的 AppClassLoader 和 ExtClassLoader 都是继承自 URLClassLoader 并通过构造函数传递自定义的扫描路径，那么我们是不是也可以继承 URLClassLoader，然后把嵌套 jar 里面的多个 jar 的路径作为一个 URLClassLoader 的扫描路径呢？这样该 URLClassLoader 就可以找到嵌套 jar 里面的资源了。URLClassLoader 的构造函数会传递一个 URL[] urls 作为该加载器的类扫描路径，那么针对上图中嵌套的 jar，我们可以创建一个 URLClassLoader，它的 urls 路径内容为：

```
/Users/zhuizhumengxiang/
/Users/zhuizhumengxiang/c.jar/c2.jar
/Users/zhuizhumengxiang/c.jar/c3.jar
```

- 根据第一个路径 URLClassLoader 加载器可以查找到 a.class、b.class 和 c.jar 里面的 c1.class 文件。
- 根据第二个路径可以加载到 c2.jar 里面的 .class 文件。
- 根据第三个路径可以加载到 c3.jar 里面的 .class 文件。

这是一个可以解决嵌套 jar 的思路，但是还有一个问题需要解决，就是默认情况下我们启动 main 函数所在的类时候用的类加载器是 AppClassLoader，而它的加载路径是 classpath。那么我们自定义的 URLClassLoader 什么时候使用呢？

为了使用这个自定义 URLClassLoader，可以想办法让我们自定义的 URLClassLoader 来加载我们的 main 函数，但是一个逃离不了的现实是当使用 Java 命令启动 main 函数所在类时候使用的总是 AppClassLoader，那么现在只有在中间加一层来解决这个问题。具体来说是使用 Java 命令启动时候启动一个中间类的 main 函数，这个中间类里面自定义 URLClassLoader，然后使用自定义 URLClassLoader 来加载我们真正的 main 函数。



如上图 Application 假设为含有 main 函数的类，之前是直接使用 AppClassLoader 进行加载，那么现在我们先使用 APPClassLoader 加载 Launcher 类，该类内部在创建一个 URLClassLoader 用来加载我们的 Application 类。下面具体介绍下 spring-boot-loader 是如何解决嵌套 jar 问题的。

spring-boot-loader 模块提供的 jar 目录结构

为了解决嵌套 jar 问题，Springboot 中 jar 文件格式规定如下。

```
archive.jar
 |
 +-META-INF（1）
 |  +-MANIFEST.MF
 +-org（2）
 |  +-springframework
 |     +-boot
 |        +-loader
 |           +-<spring boot loader classes>
 +-com（3）
 |  +-mycompany
 |     + project
 |        +-YouClasses.class
 +-lib（4）
    +-dependency1.jar
    +-dependency2.jar
```

- 结构（1）是 jar 文件中 MANIFEST.MF 文件的存放处。
- 结构（2）是 Spring-boot-loader 本身需要的 class 放置处。
- 结构（3）是应用本身的文件资源放置处。
- 结构（4）是应用依赖的 jar 固定放置处，即 lib 目录。

那么 spring-boot 是如何去创建这个结构并且按照这个结构加载资源呢？

首先在打包时候会使用 spring-boot-maven-plugin 插件重写打成的 jar 文件，会设置 META-INF/MANIFEST.MF 中的 Main-Class: org.springframework.boot.loader.JarLauncher、 Start-Class: com.mycompany.project.MyApplication，
并拷贝 spring-boot-loader 包里面的 class 文件到结构（2），应用依赖的 jar 拷贝到（4），应用本身的类拷贝到（3）。

接着，运行 java -jar archive.jar ，Launcher 会加载 JarLauncher 类并执行其中的 main 函数，JarLauncher 主要关心构造一个合适的 URLClassLoader 加载器用来调用我们应用程序（MyApplication）的 main 方法。

SpringBoot 的这种格式可以明确地让我们知道应用本身包含哪些类，应用依赖了哪些类。

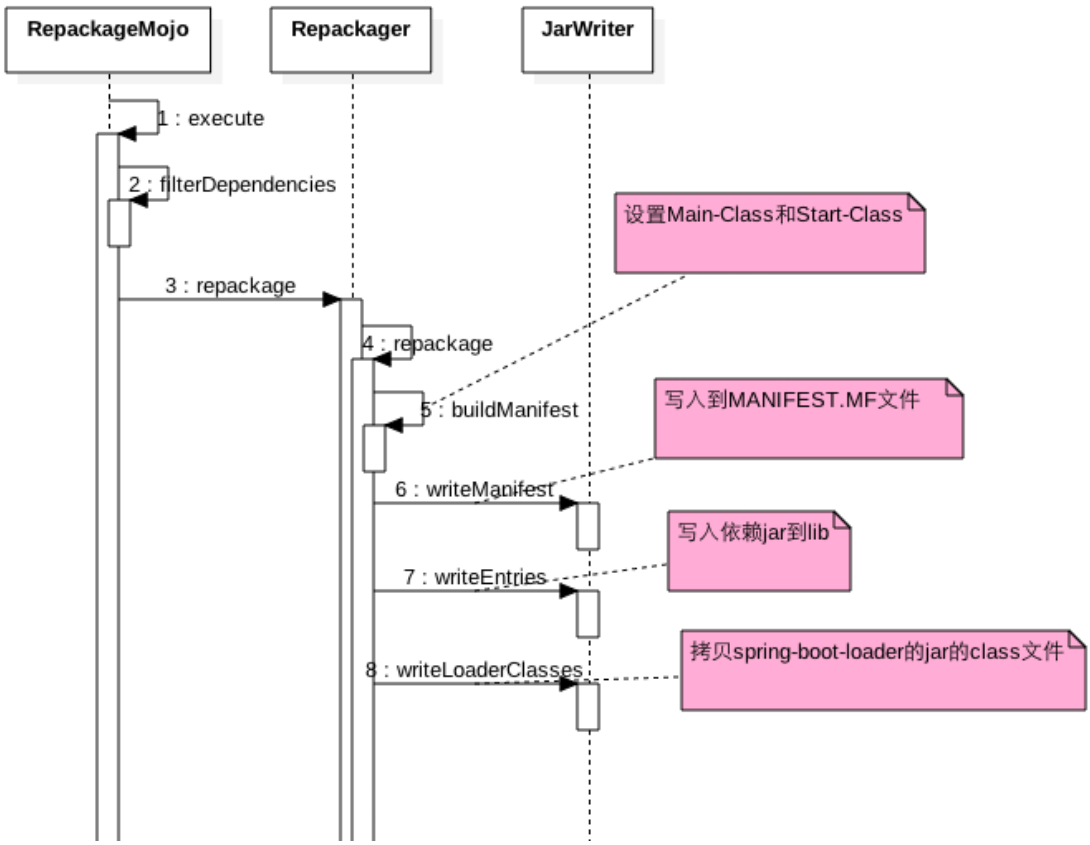
spring-boot-maven-plugin 插件打包流程分析

SpringBoot 应用打包时候需要引入如下 Maven 插件才会生成上面介绍的结构的 jar。

```
<plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>1.5.9.RELEASE</version>
      <executions>
        <execution>
          <goals>
            <goal>repackage</goal>
          </goals>
        </execution>
      </executions>
 </plugin>
```

本文使用 Springboot 版本为 1.3.5.RELEASE，Maven 插件版本为 1.5.9.RELEASE。

当我们执行 mvn clean package 进行打包生成 jar 文件后，spring-boot-maven-plugin 插件会对 jar 文件进行重写，具体重写步骤，请见下面的时序图。



- 步骤（1）是 Maven 插件执行的入口类。
- 步骤（2）设置是否从 jar 本节里面排除掉 spring-boot-devtools 的 jar 包，默认是不排除。这个可以在引入插件的地方配置，如下：

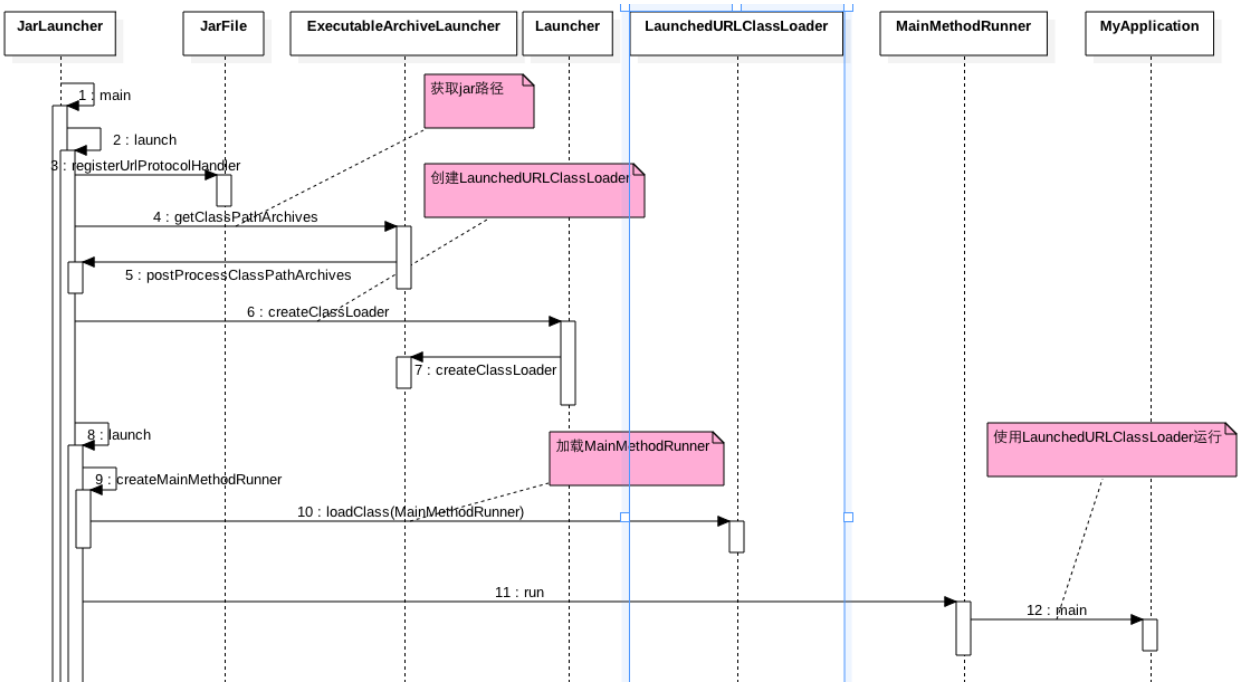
```
<configuration>
    <excludeDevtools>true</excludeDevtools>
</configuration>
```

- 步骤（5）（6）是主要环节，就是设置 MANIFEST.MF，Main-Class: org.springframework.boot.loader.JarLauncher 、Start-Class: com.mycompany.project.MyApplication，
  并写入到文件，注意这里 MyApplication 代表了 SpringBoot 里面启动整体应用的包含 main 函数的那个类，也就是加了 @SpringBootApplication 注解的那个类。
- 步骤（7）写入应用依赖的 jar 包到 lib 目录。
- 步骤（8）拷贝 spring-boot-load 包里面的 class 文件到 jar 包的结构（2）处。

注：这里读者可以先思考下为何要拷贝本来应该放入到 lib 里 spring-boot-loader.jar 里面的 class 到结构（2）？

JarLauncher 执行流程分析

为了解决嵌套 jar 资源加载问题，上节讲解了 Boot 提供的专用 Maven 插件用来修改 jar 包的 Main-Class: org.springframework.boot.loader.JarLauncher 、Start-Class: com.mycompany.project.MyApplication，修改后的结果是当我们执行 java -jar archive.jar 时候会启动 JarLauncher 的 main 函数，而不是我们 SpringBoot 应用里 MyApplication 的 main 函数，下面看看 JarLauncher 的具体执行时序图。



- 当执行 java -jar archive.jar 的时候会使用 AppClassLoader 加载 JarLauncher 类，并执行步骤（1）——执行 JarLauncher 的 main 函数。
- 步骤（4）查找 archive.jar 里面所有的嵌套 jar 后生成一个 `List<Archive>` 列表，每个 Archive 保存了一个嵌套 jar 的信息，对应 jar 包 Archive 的子类是 JarFileArchive。
- 步骤（5）在生成的 `List<Archive>` 列表的第一个位置插入 archive.jar 本身构造成的 Archive 对象。
- 步骤（6）转换步骤（5）生成的 List 列表为 URL[] urls，然后作为参数创建 LaunchedURLClassLoader 类加载器，LaunchedURLClassLoader 继承了 URLClassLoader 并重写了一些方法。
- 步骤（9）（10）（11）（12）使用 LaunchedURLClassLoader 加载并实例化 MyApplication 的一个对象，并通过反射调用 main 函数，这时候 SpringBoot 才正式开始初始化 SpringBoot 的环境，并创建容器。

看完这个流程，再分析下上节说的为何要专门拷贝 lib 目录下 spring-boot-loader.jar 里的类到 springboot jar 的结构（2）。从流程图可知首先使用 Appclassloader 加载 LaunchedURLClassLoader 类，而 LaunchedURLClassLoader 是属于 spring-boot-loader.jar 包里面的，而 Appclassloader 是普通的加载器不能加载嵌套的 jar 里面的文件，所以如果把 spring-boot-loader.jar 放到 lib 目录下，Appclassloader 将找不到 LaunchedURLClassLoader。所以在打包时候
拷贝本来应该放入到 lib 里的 spring-boot-loader.jar 里的 class 到结构（2）。

spring-boot-autoconfigure 模块

Spring 的出现给我们管理 Bean 的依赖注入提供了便捷，但是当我们需要通过 pom 引入 jar 里的一个 Bean，还是需要手动在 XML 配置文件里面配置。SpringBoot 则可以依据 classpath 里面的依赖内容自动配置 Bean 到 IOC 容器，Auto-configuration 会尝试推断哪些 Beans 是用户

可能会需要的。比如如果 HSQLDB 包在当前 classpath 下，并且用户并没有配置其他数据库链接，这时候 Auto-configuration 功能会自动注入一个基于内存的数据库连接到应用的 IOC 容器。再比如当我们在 pom 引入了 Tomcat 的 start 后，如果当前还没 Web 容器被注入到应用 IOC，那么 SpringBoot 就会为我们自动创建并启动一个内嵌 Tomcat 容器来服务。但是要开启这个自动配置功能需要添加 @EnableAutoConfiguration 注解。

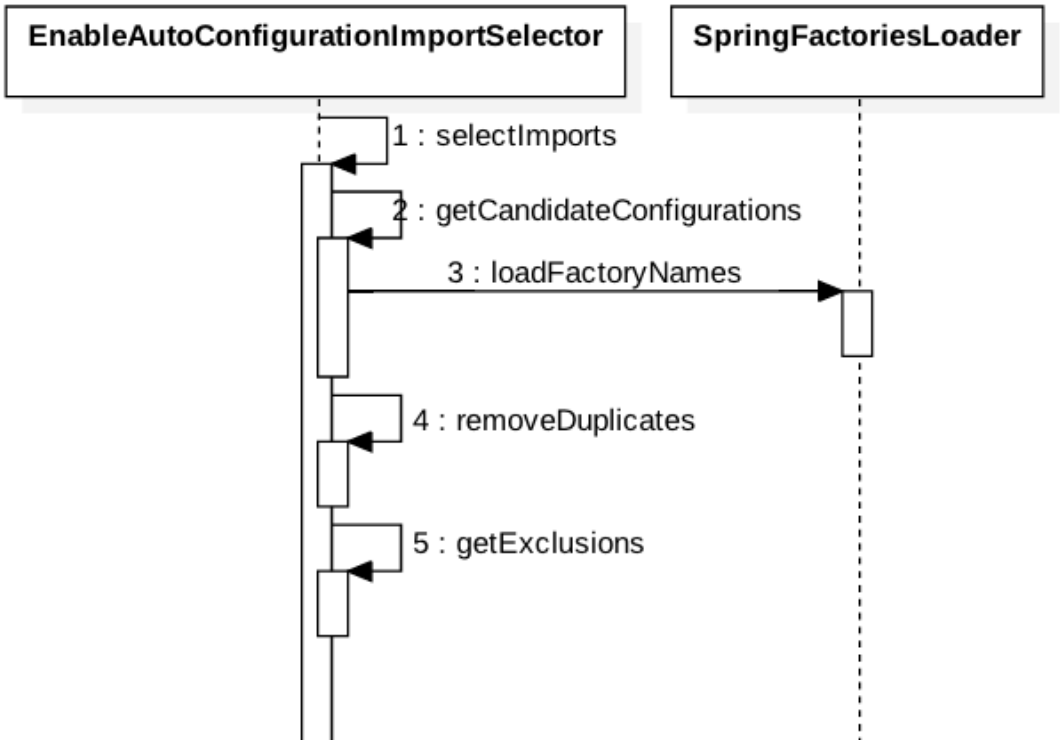
Auto-configuration 原理

Auto-configuration 通过在 class 上标注 @Configuration 注解，并且使用 @Configuration 的时候一般带有一定的约束，比如同时还在 class 上标注了 @ConditionalOnClass（标示 classpath 下是否存在某些类）和 @ConditionalOnMissingBean（标示当前 IOC 容器里面是不是存在某些 Bean）注解。这保证了 classpath 下存在一些相关的类，并且需要的 bean 还没有被注入到 IOC 时候标注 @Configuration 注解的 class 才会被自动注入到 IOC 容器。

要使用 Auto-configuration 的功能首先需要添加 @EnableAutoConfiguration 注解，下面就从这个入手看看实现原理是怎么样的。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
...
```

EnableAutoConfiguration 注解主要作用是通 @Import 注入 EnableAutoConfigurationImportSelector 实例，后者是实现 AutoConfiguration 功能的核心类，下面就来看看它的功能时序图：



- EnableAutoConfigurationImportSelector 类实现了 DeferredImportSelector 接口的 String[] selectImports(AnnotationMetadata metadata) 方法，当 Spring 框架解析 Import 注解时候会调用该方法。
    - 代码（2）（3）作用是扫描 classpath 下含有 Meta-INF/spring.factories 文件的 jar，并解析文件中名字为 org.springframework.boot.autoconfigure.EnableAutoConfiguration 的配置项，如下图所示。

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration,\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.MessageSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.PropertyPlaceholderAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\
org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration,\
```

- 步骤（2）会返回名字为 org.springframework.boot.autoconfigure.EnableAutoConfiguration 的配置项的值的一个列表并且对列表内容进行去重处理。正常情况下，去重后的列表里面的类都会被自动注入到 Spring IOC 容器，但是你也可以选择不自动注入哪些功能，比如如果你不想开启自动创建数据源和事务管理的功能，你可以加入下面的代码：

```
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class,
DataSourceTransactionManagerAutoConfiguration.class})
```

或者通过下面方式：

```
@EnableAutoConfiguration(excludeName=
{"org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfigur
ation"```,
"org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionMa
nagerAutoConfiguration"})
```

- 代码（5）就是读取 EnableAutoConfiguration 注解里面的 excludeName 和 exclude 配置项，然后从去重后的列名列表里面剔除。这样在自动注入时候就不会对排除掉的自动配置功能进行注入了。

现在回顾下，EnableAutoConfiguration 注解的功能是扫描 classpath 下的 jar 里 Meta-INF/spring.factories 文件中名字为 org.springframework.boot.autoconfigure.EnableAutoConfiguration 的配置项的值，这些配置项的值是一个列表，每个元素就是一个自动配置功能，比如 org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration 是 Web 容器自动注入的功能类（比如这个类可以自动扫描 Tomcat 的 start 并创建一个 Tomcat 容器），org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration 是自动注入数据源的功能类，可见每种功能都是使用一个 AutoConfiguration 类来完成自动装配的。下

面一个小结来具体讲解，EmbeddedServletContainerAutoConfiguration 也就是 Web 容器功能类是如何实现自动注入 Web 容器。

Web 容器的自动装配

上节介绍了注解 EnableAutoConfiguration 注入了 EmbeddedServletContainerAutoConfiguration 到 IOC 容器，那么本节就来看下 EmbeddedServletContainerAutoConfiguration 是如何自动创建 Web 容器的。

EmbeddedServletContainerAutoConfiguration 的部分核心代码如下：

```
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
@Configuration
@ConditionalOnWebApplication
@Import(EmbeddedServletContainerCustomizerBeanPostProcessorRegistrar.class)
public class EmbeddedServletContainerAutoConfiguration {

    /**
     * Nested configuration for if Tomcat is being used.
     */
    @Configuration
    @ConditionalOnClass({ Servlet.class, Tomcat.class })
    @ConditionalOnMissingBean(value = EmbeddedServletContainerFactory.class, search = SearchStrategy.CURRENT)
    public static class EmbeddedTomcat {

        @Bean
        public TomcatEmbeddedServletContainerFactory tomcatEmbeddedServletContainerFactory() {
            return new TomcatEmbeddedServletContainerFactory();
        }

    }

    /**
     * Nested configuration if Jetty is being used.
     */
    @Configuration
    @ConditionalOnClass({ Servlet.class, Server.class, Loader.class,
            WebAppContext.class })
    @ConditionalOnMissingBean(value = EmbeddedServletContainerFactory.class, search = SearchStrategy.CURRENT)
    public static class EmbeddedJetty {

        @Bean
        public JettyEmbeddedServletContainerFactory jettyEmbeddedServletContainerFactory() {
            return new JettyEmbeddedServletContainerFactory();
        }

    }
```

- EmbeddedServletContainerAutoConfiguration 类是 Web 容器自动注入的 Auto-configuration 类。
- @ConditionalOnWebApplication 说明当前是 Web 环境上下文时候才注入本类到 IOC。
- 对 Tomcat 容器来说，它的核心代码里面需要 Servlet.class、Tomcat.class 这两个类，所以@ConditionalOnClass({ Servlet.class, Tomcat.class }) 是指如果当前 classpath 的 jar 里面含有 Servlet.class、Tomcat.class 这两个类，才会进入下一个条件的判断，@ConditionalOnMissingBean(value = EmbeddedServletContainerFactory.class, search = SearchStrategy.CURRENT) 说明当前 IOC 容器里面是否没有 EmbeddedServletContainerFactory 的实例，如果两个条件都满足则会创建 TomcatEmbeddedServletContainerFactory 实例到 IOC 容器。那么在 SpringBoot 启动时候会使用 TomcatEmbeddedServletContainerFactory 的实例创建一个 Tomcat 容器。
- 对 Jetty 容器来说，它的核心代码里面需要 Servlet.class、Server.class、Loader.class、WebAppContext.class，所以 @ConditionalOnClass({ Servlet.class, Server.class, Loader.class,WebAppContext.class }) 是看当前 classpath 的 jar 里面是否含有这些类，这些类存在进入下一个条件看当前 IOC 容器里面是否没有

EmbeddedServletContainerFactory 的实例。如果两个条件都满足则会创建 JettyEmbeddedServletContainerFactory 的实例到 IOC 容器，那么在 SpringBoot 启动时候会使用 JettyEmbeddedServletContainerFactory 的实例创建一个 Jetty 容器。

当应用引入 spring-boot-starter-web 时候默认引入的是 Tomcat 的 start，所以会发现 classpath 下存在 Servlet.class 、Tomcat.class 这两个类，并且 IOC 里面没有 EmbeddedServletContainerFactory 的实例，因此会创建 JettyEmbeddedServletContainerFactory 的实例 TomcatEmbeddedServletContainerFactory 到 IOC，最终会创建一个 Tomcat 容器。如果你需要使用 Jetty 则需要在引用 spring-boot-starter-web 的时候排除掉 Tomcat 的 start，然后在引入 Jetty 的 start 即可。

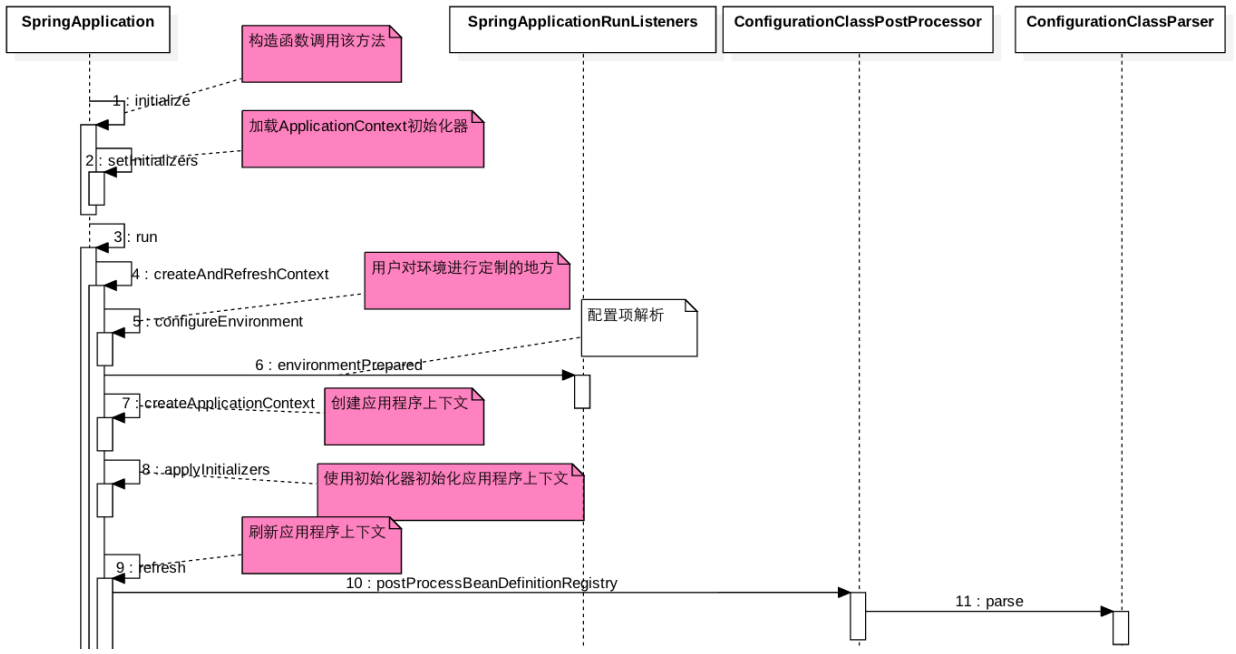
spring-boot 模块

spring-boot 模块提供了一些特性用来支持 SpringBoot 中其他模块，这些特性包含如下：

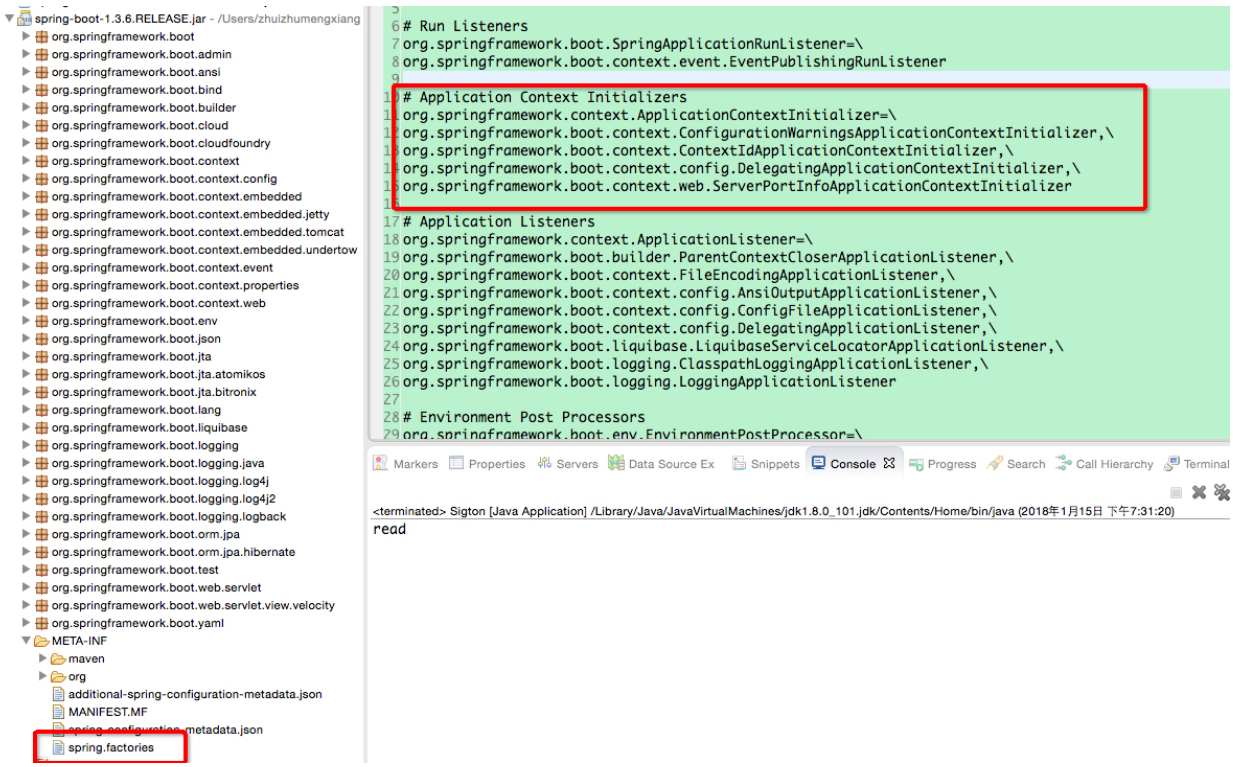
- SpringApplication 类提供了静态方法以便于写一个独立了 Spring 应用程序，该类的主要职责是 create 和 refresh 一个合适的 Spring 应用程序上下文（ApplicationContext）。
- 一流的外部配置的支持（application.properties）。
- 提供了便捷的应用程序上下文（ApplicationContext）的初始化器，以便在 ApplicationContext 使用前对其进行用户定制。
- 给 Web 应用提供了一个可选的 Web 容器（目前有 Tomcat 或 Jetty）。

SpringBoot 的启动

上面，我们讲解到为了能够加载嵌套 jar，SpringBoot 提供的 JarLauncher 的执行流程，本节我们接着上面的启动过程继续往下讲，也就是讲解 SpringBoot 的真正 main 函数启动流程，由于里面实际调用了 SpringApplication 的方法，所以我们只看 SpringApplication 的执行流程，下面先看下执行时序图：



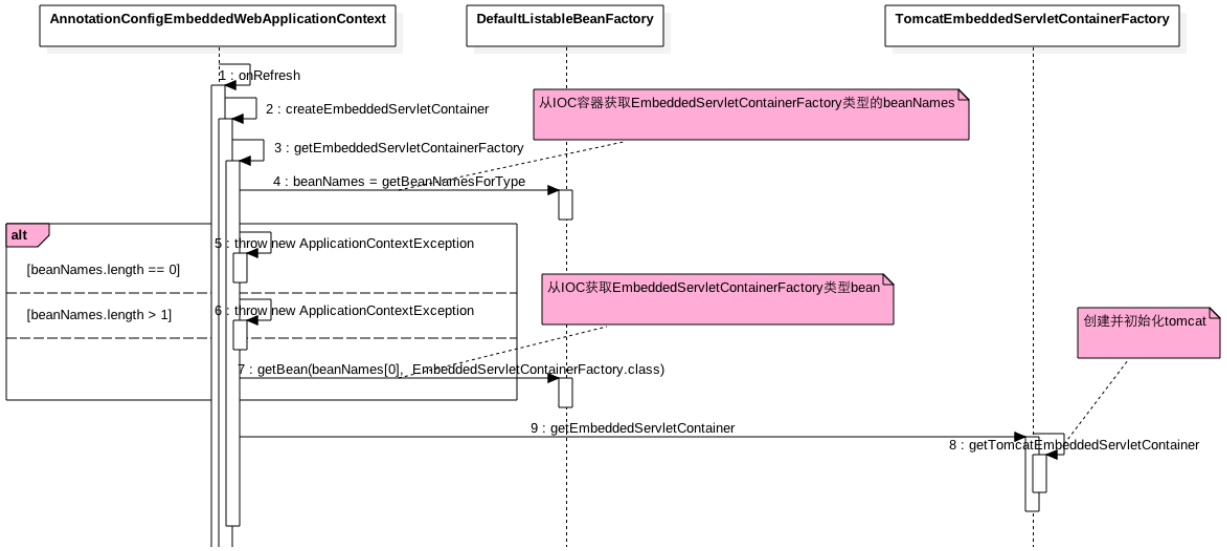
- SpringApplication 的构造函数里面会调用 initialize 方法在 classpath 的 jar 包里面查找 META-INF/spring.factories，如果找到则看里面是否有配置 ApplicationContextInitializer 类型的 Bean，有则加载到 SpringApplication 的变量 initializers 里面存放，比如 spring-boot.jar 里面。



- createAndRefreshContext 做了这几件事情。第一设置环境，加载 application.properties 等配置文件；第二根据 classpath 的 jar 里面是否有 ConfigurableWebEnvironment 判断当前是否需要创建 Web 应用程序上下文还是创建一个非 Web 应用程序上下文；第三使用前面加载的应用程序初始化器对创建的应用程序上下文进行初始化；第四刷新应用程序上下文解析 Bean 定义到应用程序上下文里面的 IOC 容器，在刷新过程的 invokeBeanFactoryPostProcessors 过程中(10)(11)会去解析类上面标注的 @import 注解，然后就会调用所有的 ImportSelector的selectImports 方法，这也是第二部分时序图开始执行的地方。

Web 容器的创建

第二部分我们讲了通过自动配置把 TomcatEmbeddedServletContainerFactory 或者 JettyEmbeddedServletContainerFactory 的实例注入了 IOC 容器，下面我们就来讲解如何根据这其中之一来创建具体的 Web 容器。Web 容器的创建是在容器刷新过程的 onRefresh 阶段进行的（这个阶段是在刷新过程的 invokeBeanFactoryPostProcessors 阶段的后面），下面看下时序图：



- 在应用程序上下文的 refresh() 流程中的 onReresh（ ）方法中创建了 Web 容器。
- getBeanNamesForType 获取了 IOC 容器中的 EmbeddedServletContainerFactory 类型的 Bean 的 name 集合，如果 name 集合为空或者多个则抛出异常。还记得 Web 容器工厂是通过自动配置注入到 IOC 的吧，并且 TomcatEmbeddedServletContainerFactory 或者 JettyEmbeddedServletContainerFactory 都是实现了 EmbeddedServletContainerFactory 接口。
- 如果 IOC 里面只有一个 Web 容器工厂 Bean 则获取该 Bean 的实例，然后调用该 Bean 的 getEmbeddedServletContainer 获取 Web 容器，这里假设 Web 容器工厂为 Tomcat ，则创建 Tomcat 容器并进行初始化。

总结

spring-boot-load 模块通过打包时候重新设置启动类和组织 jar 结构，运行时设置自定义加载器来实现嵌套 jar 资源加载。

springboot 的 spring-boot-autoconfigure 模块通过灵活的 Auto-configuration 注解使 SpringBoot 中的功能实现模块化。spring-boot-autoconfigure 思路类似 SPI(Service Provider Interface)，都是不同的实现类实现了定义的接口，加载时候去查找 classpath 下的实现类，不同在于前者使用 autoconfigure 实现，后者使用的是 ServiceLoader。

Spring-boot 模块为基础模块提供了基础服务，例如装载了其它模块可能使用的配置项，应用程序上下文在使用前的用户定制，以及 Web 容器的创建。