链滴

用了这么多年的 SpringBoot 你知道什么是 SpringBoot 的 Web 类型推断吗?

作者：zhuSilence

原文链接：https://ld246.com/article/1672057398427

来源网站：链滴

用了这么多年的 SpringBoot 那么你知道什么是 SpringBoot 的 web 类型推断吗?

估计很多小伙伴都不知道，毕竟平时开发做项目的时候做的都是普通的 web 项目并不需要什么特别了解，不过抱着学习的心态，阿粉今天带大家看一下什么是 SpringBoot 的 web 类型推断。

SpringBoot 的 web 类型有哪些

既然是web 类型推断，那我们肯定要知道 SpringBoot 支持哪些类型，然后才能分析是怎样进行类型断的。

根据官方的介绍 SpringBoot 的 web 类型有三种，分别是，NONE、SERVLET 和 REACTIVE，定义枚举 WebApplicationType 中，这三种类型分别代表了三种含义：

1. NONE：不是一个 web 应用，不需要启动内置的 web 服务器；
2. SERVLET：基于 servlet 的 web 应用，需要启动一个内置的 servlet 服务器；
3. REACTIVE：一个 reactive 的 web 应用，需要启动一个内置的 reactive 服务器；

```
public enum WebApplicationType {
    NONE,
    SERVLET,
    REACTIVE;
}
```

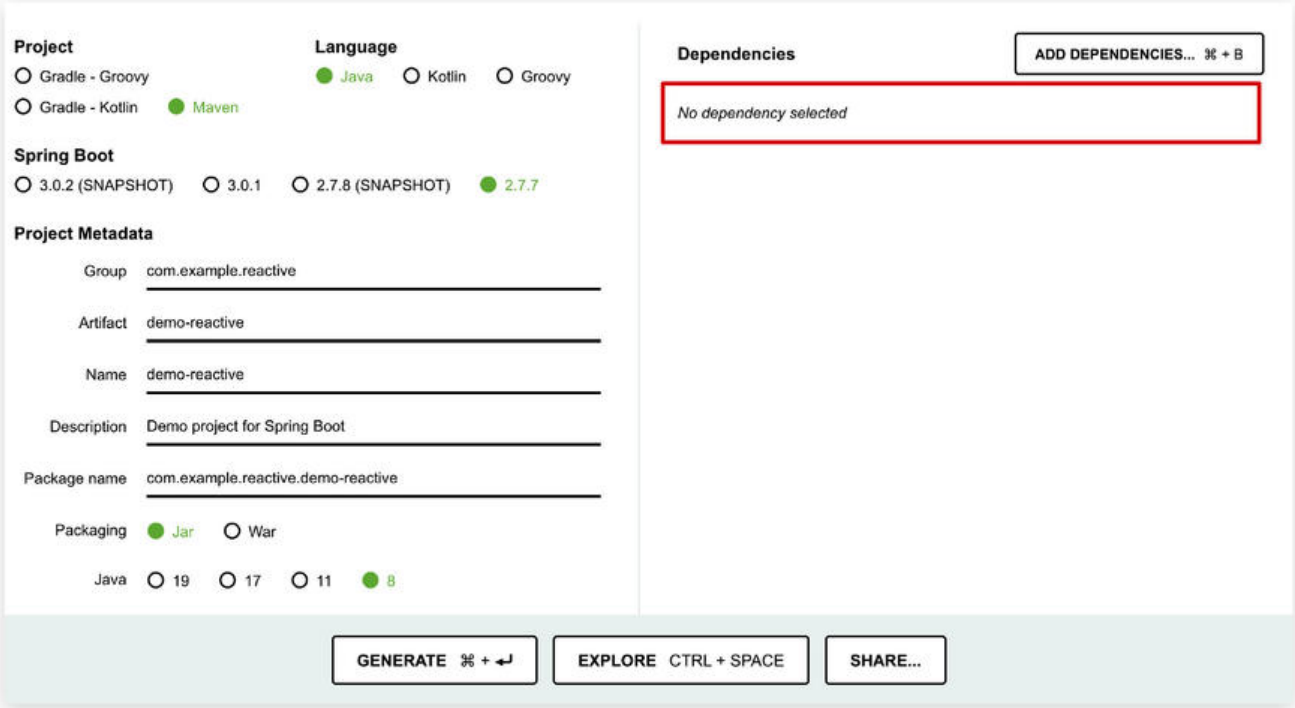
web 类型推断

上面提到了 SpringBoot 的三种 web 类型，接下来我们先通过代码验证一下，然后再分析一下 Sprin Boot 是如何进行类型推断的。

首先我们通过在 https://start.spring.io/ 快速的构建三种类型的项目，三种类型的项目配置除了依赖

一样之外，其他都一样，如下所示

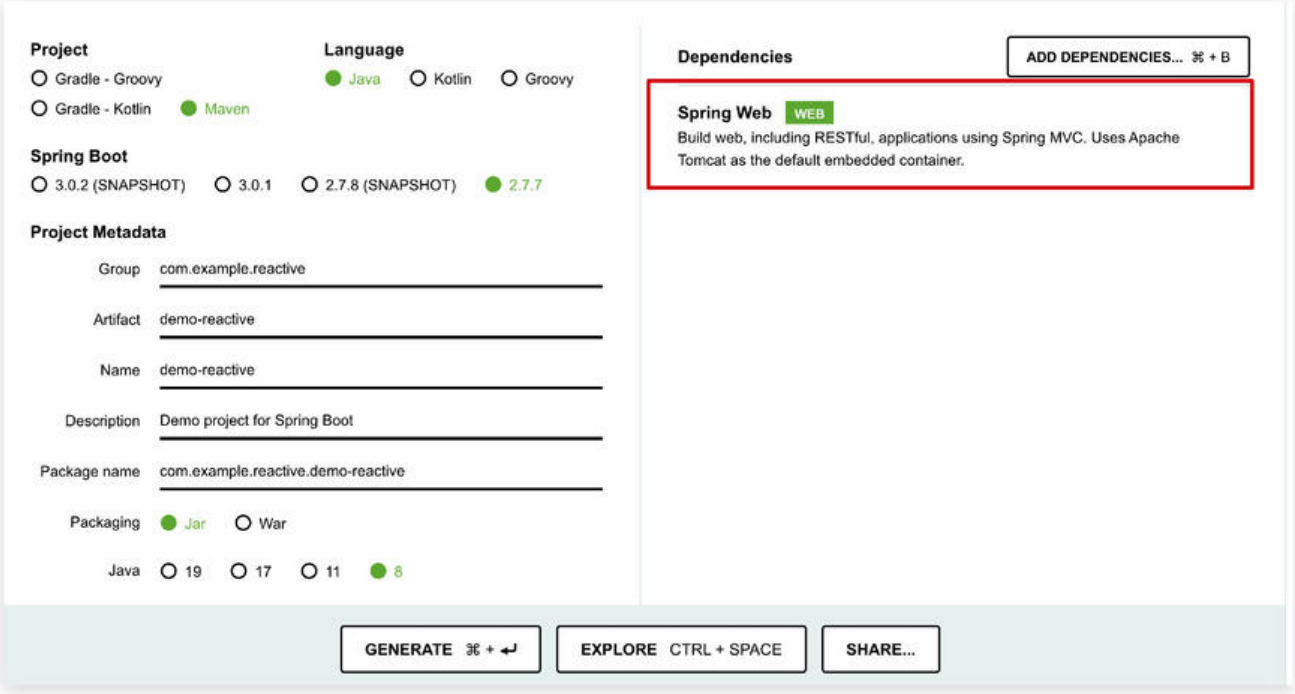
None web



下载后的项目文件 pom 中对应的依赖为

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
</dependency>
```

Servlet web

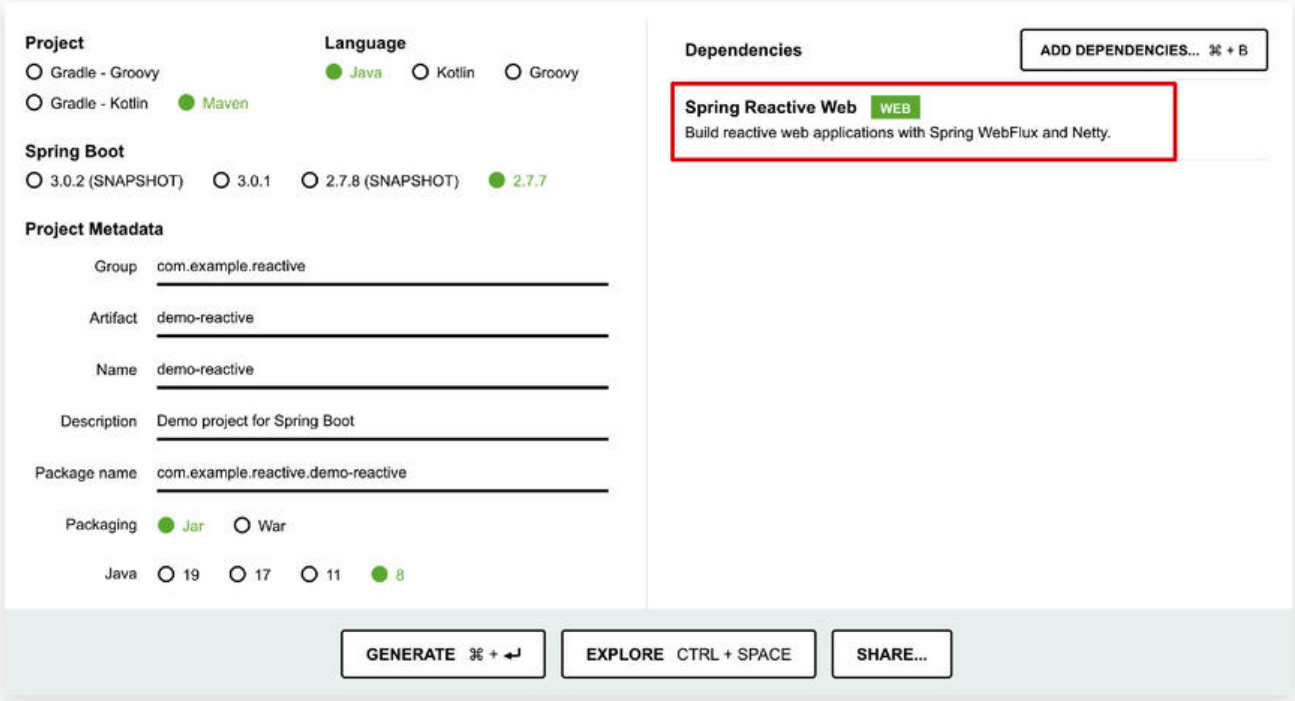


原文链接：用了这么多年的 SpringBoot 你知道什么是 SpringBoot 的 Web 类型推断吗?

下载后的项目文件 pom 中对应的依赖为

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Reactive web

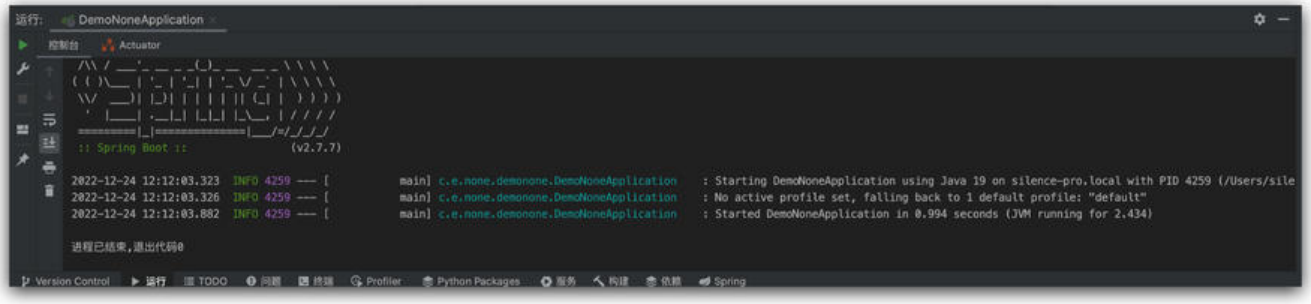


下载后的项目文件 pom 中对应的依赖为

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

接下来我们依次启动三个项目看看有什么区别，

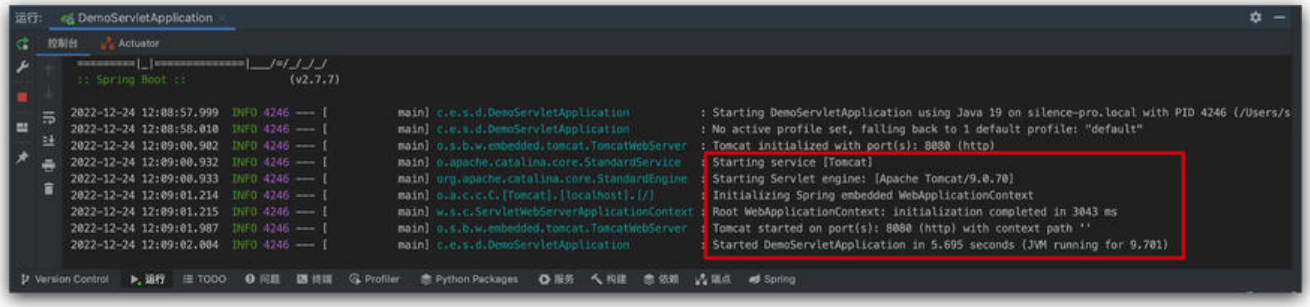
启动 None web



通过启动日志我们可以看到，在 None web 类型下，应用启动运行后就自动关闭了，并没有启动内的 web 服务器，也没有监听任何端口。接下来我们看看其他两种类型 web 的启动日志都是怎么样的。

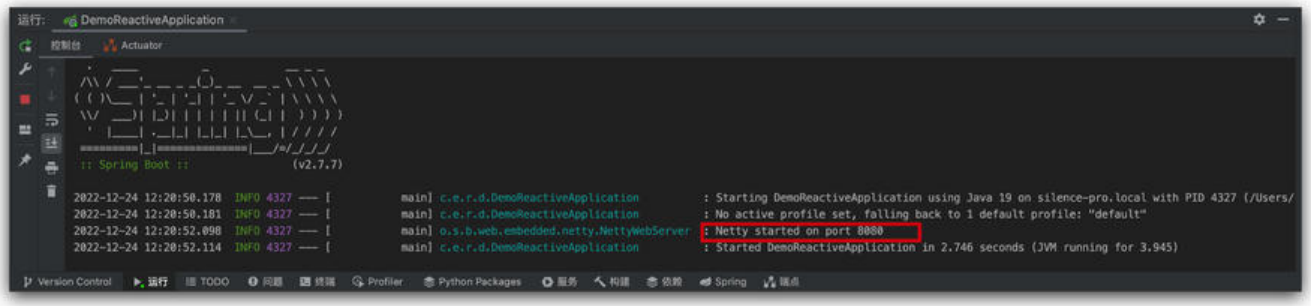
原文链接：用了这么多年的 SpringBoot 你知道什么是 SpringBoot 的 Web 类型推断吗?

启动 Servlet web



通过启动日志我们可以看到这里启动了内置的 Tomcat Servlet 服务器，监听了 8080 端口，应用程并不会像 None 类型一样，启动后就自动关闭。

启动 Reactive web



通过启动日志我们可以看到，这里启动了内置的 Netty 服务器，并监听在 8080 端口上（如果启动失记得把上面 servlet web 关闭，不然端口会冲突）。

三种类型的服务我们都成功启动了，那么接下来的问题就是 SpringBoot 是如何判断出该使用哪种类的呢?

这三个服务我们只有依赖不一样，很明显肯定和依赖有关系，接下来我们就来研究一下 SpringBoot 如何实现的。

SpringBoot Web 类型推断原理

我们在 main 方法中点击 run 方法，

```
public static ConfigurableApplicationContext run(Class<?>[] primarySources, String[] args) {
    return new SpringApplication(primarySources).run(args);
}
```

在构造函数中我们可以看到其中有这么一行 this.webApplicationType = WebApplicationType.deduceFromClasspath();根据属性名称我们可以推断，web 类型就是根据 WebApplicationType.deduceFomClasspath(); 这个静态方法来判断的。接下来我们看下这个方法的细节。

```java
/unchecked, rawtypes/
public SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources) {
    this.resourceLoader = resourceLoader;
    Assert.notNull(primarySources, message: "PrimarySources must not be null");
    this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));
    this.webApplicationType = WebApplicationType.deduceFromClasspath();
    this.bootstrapRegistryInitializers = new ArrayList<>(
            getSpringFactoriesInstances(BootstrapRegistryInitializer.class));
    setInitializers((Collection) getSpringFactoriesInstances(ApplicationContextInitializer.class));
    setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));
    this.mainApplicationClass = deduceMainApplicationClass();
}
```

```java
1 个用法
private static final String[] SERVLET_INDICATOR_CLASSES = { "javax.servlet.Servlet",
        "org.springframework.web.context.ConfigurableWebApplicationContext" };

1 个用法
private static final String WEBMVC_INDICATOR_CLASS = "org.springframework.web.servlet.DispatcherServlet";

1 个用法
private static final String WEBFLUX_INDICATOR_CLASS = "org.springframework.web.reactive.DispatcherHandler";

1 个用法
private static final String JERSEY_INDICATOR_CLASS = "org.glassfish.jersey.servlet.ServletContainer";

1 个用法
static WebApplicationType deduceFromClasspath() {
    if (ClassUtils.isPresent(WEBFLUX_INDICATOR_CLASS, classLoader: null) && !ClassUtils.isPresent(WEBMVC_INDICATOR_CLASS,
            && !ClassUtils.isPresent(JERSEY_INDICATOR_CLASS, classLoader: null)) {
        return WebApplicationType.REACTIVE;
    }
    for (String className : SERVLET_INDICATOR_CLASSES) {
        if (!ClassUtils.isPresent(className, classLoader: null)) {
            return WebApplicationType.NONE;
        }
    }
    return WebApplicationType.SERVLET;
}
```

如上图所示，可以看到 SpringBoot 底层是通过 ClassUtils.isPresent() 方法来判断对应的 web 类型是否存在来判断 web 类型的。

在前类路径下面如果当 org.springframework.web.reactive.DispatcherHandler 存在而且 org.springframework.web.servlet.DispatcherServlet 和 org.glassfish.jersey.servlet.ServletContainer 都不在的时候说明当前应用 web 类型为 Reactive。

当 javax.servlet.Servlet 和 org.springframework.web.context.ConfigurableWebApplicationContext 任何一个不存在的时候，就说明当前应用是 None 类型非 web 应用。否则当前应用就为 Servlet 型。

而我们再看这个 ClassUtils.isPresent() 方法，可以发现底层是通过 className 在类路径上加载对应类，如果存在则返回 true，如果不存在则返回 false。

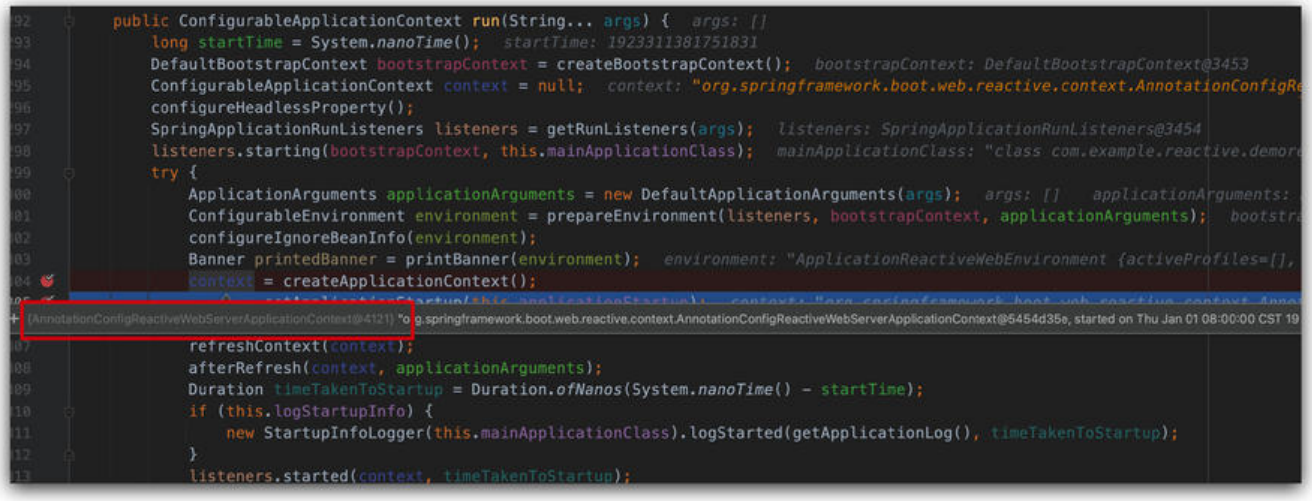
```
public static boolean isPresent(String className, @Nullable ClassLoader classLoader) {
    try {
        forName(className, classLoader);
        return true;
    }
    catch (IllegalAccessError err) {
        throw new IllegalStateException("Readability mismatch in inheritance hierarchy of class [" +
                className + "]: " + err.getMessage(), err);
    }
    catch (Throwable ex) {
        // Typically ClassNotFoundException or NoClassDefFoundError...
        return false;
    }
}
```

因此这也解释了为什么我们在 pom 文件中只要加入对应的依赖就可以直接得到相应的 web 类型了因为当我们在 pom 中加入相应的依赖过后，类路径里面就存在了前面判断的对应的类，再通过 ClassUtils.isPresent() 就判断出来当前应用属于那种 web 类型了。

内置服务器是如何创建的

知道了 SpringBoot 是如何进行 web 类型推断的，那么接下来一个问题就是 SpringBoot 是如何根据 web 类型进行相应内置 web 服务器的启动的呢？这里我们以 Reactive web 为例进行调试追踪。

首先我们在 SpringApplication 的 run 方法 createApplicationContext() 下一行打断点，可以发现建成功的 context 类型为 AnnotationConfigReactiveWebServerApplicationContext 很明显在这步的时候就已经根据类型推断得到了当前的应用 web 类型为 Reactive，并且根据 web 类型创建出对应的 ApplicationContext。



紧接着我们进入 org.springframework.boot.SpringApplication#refreshContext 方法，最后我们以进入到 org.springframework.boot.web.reactive.context.ReactiveWebServerApplicationContext#refresh 方法中，因为 AnnotationConfigReactiveWebServerApplicationContext 继承了 ReactiveWebServerApplicationContext。

```
ReactiveWebServerApplicationContext,
AnnotationConfigApplicationContext
作者:  Phillip Webb

10 个用法
public class AnnotationConfigReactiveWebServerApplicationContext extends ReactiveWebServerApplicationContext
        implements AnnotationConfigRegistry {

    6 个用法
    private final AnnotatedBeanDefinitionReader reader;

    7 个用法
    private final ClassPathBeanDefinitionScanner scanner;
```

继续通过引用关系，我们可以找到 org.springframework.boot.web.reactive.context.ReactiveWebServerApplicationContext#onRefresh 方法，而在这个方法里面我们就会发现了如下代码，此处就创建一个 webServer。

```
@Override
protected void onRefresh() {
    super.onRefresh();
    try {
        createWebServer();
    }
    catch (Throwable ex) {
        throw new ApplicationContextException("Unable to start reactive web server", ex);
    }
}
```

具体创建的方法在 WebServerManager 里面，跟着继续往下找我们可以找到 createHttpServer() 法，在 createHttpServer() 方法中就创建了 HttpServer 并且绑定了默认的端口 8080。具体过程，下几张接入所示，感兴趣的可以自行跟踪 debug，至此一个 Reactive 内置服务器就创建成功了，同的 Servlet 服务器也是类似的。

```
private void createWebServer() {
    WebServerManager serverManager = this.serverManager;
    if (serverManager == null) {
        StartupStep createWebServer = this.getApplicationStartup().start( name: "spring.boot.webserver.create");
        String webServerFactoryBeanName = getWebServerFactoryBeanName();
        ReactiveWebServerFactory webServerFactory = getWebServerFactory(webServerFactoryBeanName);
        createWebServer.tag("factory", webServerFactory.getClass().toString());
        boolean lazyInit = getBeanFactory().getBeanDefinition(webServerFactoryBeanName).isLazyInit();
        this.serverManager = new WebServerManager( applicationContext: this, webServerFactory, this::getHttpHandler, lazyInit);
        getBeanFactory().registerSingleton( s: "webServerGracefulShutdown",
                new WebServerGracefulShutdownLifecycle(this.serverManager.getWebServer()));
        getBeanFactory().registerSingleton( s: "webServerStartStop",
                new WebServerStartStopLifecycle(this.serverManager));
        createWebServer.end();
    }
    initPropertySources();
}
```

```
@Override
public WebServer getWebServer(HttpHandler httpHandler) {
    HttpServer httpServer = createHttpServer();
    ReactorHttpHandlerAdapter handlerAdapter = new ReactorHttpHandlerAdapter(httpHandler);
    NettyWebServer webServer = createNettyWebServer(httpServer, handlerAdapter, this.lifecycleTimeout,
            getShutdown());
    webServer.setRouteProviders(this.routeProviders);
    return webServer;
}
```

原文链接：用了这么多年的 SpringBoot 你知道什么是 SpringBoot 的 Web 类型推断吗?

```
1 个用法
private HttpServer createHttpServer() {
    HttpServer server = HttpServer.create();
    if (this.resourceFactory != null) {
        LoopResources resources = this.resourceFactory.getLoopResources();
        Assert.notNull(resources,  message: "No LoopResources: is ReactorResourceFactory not initialize
        server = server.runOn(resources).bindAddress(this::getListenAddress);
    }
    else {
        server = server.bindAddress(this::getListenAddress);
    }
    if (getSsl() != null && getSsl().isEnabled()) {
        server = customizeSslConfiguration(server);
    }
    if (getCompression() != null && getCompression().getEnabled()) {
        CompressionCustomizer compressionCustomizer = new CompressionCustomizer(getCompression());
        server = compressionCustomizer.apply(server);
    }
    server = server.protocol(listProtocols()).forwarded(this.useForwardHeaders);
    return applyCustomizers(server);
}
```

```
2 个用法
private InetSocketAddress getListenAddress() {
    if (getAddress() != null) {
        return new InetSocketAddress(getAddress().getHostAddress(), getPort());
    }
    return new InetSocketAddress(getPort());
}
```

总结

Spring 的出现给 Java 程序员带来了春天，而 SpringBoot 框架的出现又极大的加速了程序员的开发率，然而很多时候我们在使用她的便利的同时会缺少对于底层系统实现的把握，希望这篇文章弄帮助家对 SpringBoot 产生更多的理解。

原文链接：用了这么多年的 SpringBoot 你知道什么是 SpringBoot 的 Web 类型推断吗?