芋道源码 —— 知识星球

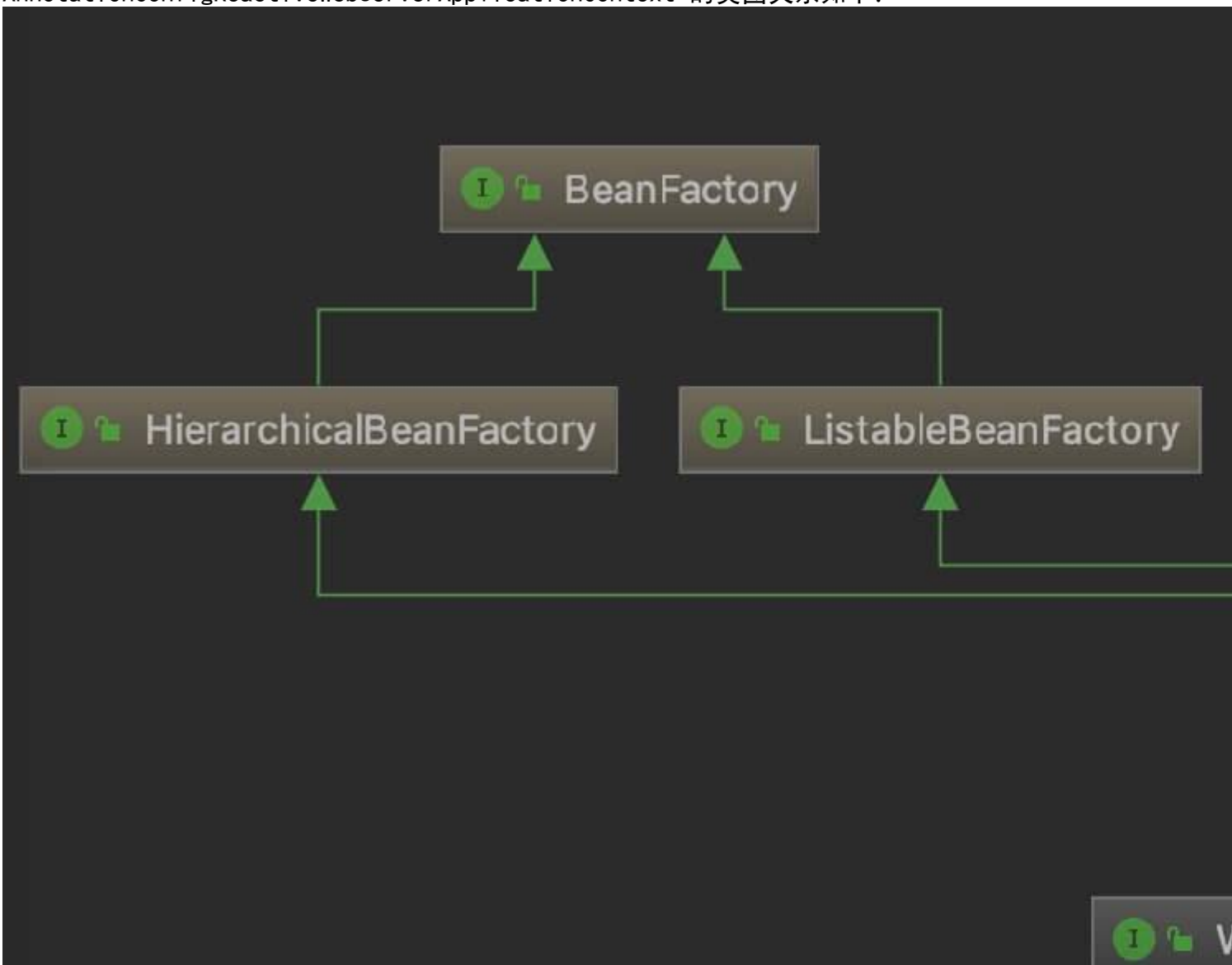
我是一段不羁的公告！
记得给芳芳这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro
2021-01-19
Spring Boot

精尽 Spring Boot 源码分析 —— ReactiveWebServerApplicationContext

1. 概述

本文接 《精尽 Spring Boot 源码分析 —— ServletWebServerApplicationContext》 一文，我们来分享 ReactiveWebServerApplicationContext 类，它提供 Reactive Web 环境的 Spring 容器。

AnnotationConfigReactiveWebServerApplicationContext 的类图关系如下：



相比来说，ReactiveWebServerApplicationContext 比 ServletWebServerApplicationContext 有更多的层级关系。不过没事，我们一点一点来看。

芮芮：Spring Webflux ，我自己目前没太使用过。看这块，就单纯好奇下，哈哈哈。

2. ReactiveWebApplicationContext

org.springframework.boot.web.reactive.context.ReactiveWebApplicationContext ，继承 ApplicationContext 接口，Reactive Web ApplicationContext 接口。代码如下：

```
// ReactiveWebApplicationContext.java

public interface ReactiveWebApplicationContext extends ApplicationContext {
}
```

3. ConfigurableReactiveWebApplicationContext

org.springframework.boot.web.reactive.context.ConfigurableReactiveWebApplicationContext ，继承 ConfigurableApplicationContext、「2. ReactiveWebApplicationContext」 接口，可配置的 ReactiveWebApplicationContext 接口。代码如下：

```
// ReactiveWebApplicationContext.java

public interface ConfigurableReactiveWebApplicationContext extends ConfigurableApplicationContext, ReactiveWebApplica
}
```

4. GenericReactiveWebApplicationContext

org.springframework.boot.web.reactive.context.GenericReactiveWebApplicationContext ，实现 「3. ConfigurableReactiveWebApplicationContext」 接口，继承 GenericApplicationContext 类，通用的 Reactive Web ApplicationContext 实现类。代码如下：

```
// GenericReactiveWebApplicationContext.java

public class GenericReactiveWebApplicationContext extends GenericApplicationContext
        implements ConfigurableReactiveWebApplicationContext {

    public GenericReactiveWebApplicationContext() {
    }

    public GenericReactiveWebApplicationContext(DefaultListableBeanFactory beanFactory) {
        super(beanFactory);
    }

    @Override // 覆写 AbstractApplicationContext 方法
    protected ConfigurableEnvironment createEnvironment() {
```

```
        return new StandardReactiveWebEnvironment(); // <X>
    }

    @Override // 覆写 AbstractApplicationContext 方法
    protected Resource getResourceByPath(String path) {
        // We must be careful not to expose classpath resources
        return new FilteredReactiveWebContextResource(path); // <Y>
    }

}
```

重点在 <X> 和 <Y> 处，覆写了方法，分别返回了 StandardReactiveWebEnvironment 和 FilteredReactiveWebContextResource 对象。不过看了下这两个对象，暂时也没什么特殊的方法。所以，可暂时忽略~

5. ReactiveWebServerApplicationContext

芮芮：正戏开始~

org.springframework.boot.web.reactive.context.ReactiveWebServerApplicationContext ，实现 ConfigurableWebServerApplicationContext 接口，继承 GenericReactiveWebApplicationContext 类，Spring Boot 使用 Reactive Web 服务器的 ApplicationContext 实现类。

5.1 构造方法

```
// ReactiveWebServerApplicationContext.java

/**
 * ServerManager 对象
 */
private volatile ServerManager serverManager;
/**
 * 通过 {@link #setServerNamespace(String)} 注入。
 *
 * 不过貌似，一直没被注入过，可以暂时先无视
 */
private String serverNamespace;

public ReactiveWebServerApplicationContext() {
}

public ReactiveWebServerApplicationContext(DefaultListableBeanFactory beanFactory) {
    super(beanFactory);
}
```

serverManager 属性，ServerManager 对象。详细解析，见 「5.2 ServerManager」 。

5.2 ServerManager

ServerManager 是 ReactiveWebServerApplicationContext 的内部静态类，实现 org.springframework.http.server.reactive.HttpHandler 接口，内含 Server 的管理器。

5.2.1 构造方法

```
// ReactiveWebServerApplicationContext#ServerManager.java

/**
 * WebServer 对象
 */
private final WebServer server;
/**
 * HttpHandler 对象，具体在 {@link #handle(ServerHttpRequest, ServerHttpResponse)} 方法中使用。
 */
private volatile HttpHandler handler;

private ServerManager(ReactiveWebServerFactory factory) {
    this.handler = this::handleUninitialized; // <1> 同下面
//          this.handler = new HttpHandler() {
//                @Override
//                public Mono<Void> handle(ServerHttpRequest request, ServerHttpResponse response) {
//                    return handleUninitialized(request, response);
//                }
//            };
    this.server = factory.getWebServer(this); // <2>
}

private Mono<Void> handleUninitialized(ServerHttpRequest request, ServerHttpResponse response) {
    throw new IllegalStateException("The HttpHandler has not yet been initialized");
}
```

<1> 处，创建的 handler 对象，内部实现调用了 #handleUninitialized(ServerHttpRequest request, ServerHttpResponse response) 方法，会抛出 IllegalStateException 异常。表示，此时该 server 还不可用。因为，server 都还没启动。
<2> 处，调用 ReactiveWebServerFactory#getWebServer(HttpHandler handler) 方法，创建 WebServer 对象。其中，方法参数 handler 传递是 this 自己，因为后面的请求处理的 #handle(ServerHttpRequest request, ServerHttpResponse) 方法，需要调用自己哟。

5.2.2 handle

实现 #handle(ServerHttpRequest request, ServerHttpResponse) 方法，处理请求。代码如下：

```
// ReactiveWebServerApplicationContext#ServerManager.java

@Override
public Mono<Void> handle(ServerHttpRequest request, ServerHttpResponse response) {
 return this.handler.handle(request, response);
}
```

委托给 handler 属性，对应的方法，处理请求。

5.2.3 get

#get(ReactiveWebServerFactory factory) 静态方法，创建一个 ServerManager 对象。代码如下：

```
// ReactiveWebServerApplicationContext#ServerManager.java
```

```
public static ServerManager get(ReactiveWebServerFactory factory) {
 return new ServerManager(factory);
}
```

5.2.4 getWebServer

#getWebServer(ServerManager manager) 静态方法，获得 manager 的 WebServer 对象。代码如下：

```
// ReactiveWebServerApplicationContext#ServerManager.java

public static WebServer getWebServer(ServerManager manager) {
 return (manager != null) ? manager.server : null;
}
```

5.2.5 start

#start(ServerManager manager, Supplier<HttpHandler> handlerSupplier) 静态方法，启动。代码如下：

```
// ReactiveWebServerApplicationContext#ServerManager.java

public static void start(ServerManager manager, Supplier<HttpHandler> handlerSupplier) {
    if (manager != null && manager.server != null) {
        // <1> 赋值 handler
        manager.handler = handlerSupplier.get();
        // <2> 启动 server
        manager.server.start();
    }
}
```

<1> 处，将 handlerSupplier 赋值给 manager.handler 。
<2> 处，调用 WebServer#start() 方法，启动 Server 。

5.2.6 stop

#stop(ServerManager manager) 静态方法，停止。代码如下：

```
// ReactiveWebServerApplicationContext#ServerManager.java

public static void stop(ServerManager manager) {
 if (manager != null && manager.server != null) {
     try {
            manager.server.stop();
        } catch (Exception ex) {
         throw new IllegalStateException(ex);
        }
    }
}
```

调用 ServerManager#stop() 方法，停止 Server 。

到此时，可能胖友有点懵逼。没事，我们下面看看 ReactiveWebServerApplicationContext 怎么使用它。

5.3 refresh

覆写 `#refresh()` 方法，初始化 Spring 容器。代码如下：

```
// ReactiveWebServerApplicationContext.java

@Override
public final void refresh() throws BeansException, IllegalStateException {
    try {
        // 调用父方法
        super.refresh();
    } catch (RuntimeException ex) {
        // <X> 停止 Reactive WebServer
        stopAndReleaseReactiveWebServer();
        throw ex;
    }
}
```

主要是 `<X>` 处，如果发生异常，则调用 `#stopAndReleaseWebServer()` 方法，停止 WebServer。详细解析，见 「5.3.1 stopAndReleaseWebServer」 。

5.3.1 stopAndReleaseWebServer

`#stopAndReleaseWebServer()` 方法，停止 WebServer。代码如下：

```
// ReactiveWebServerApplicationContext.java

private void stopAndReleaseReactiveWebServer() {
    ServerManager serverManager = this.serverManager;
 try {
        ServerManager.stop(serverManager); // <Y>
    } finally {
     this.serverManager = null;
    }
}
```

`<Y>` 处，调用 `ServerManager#stop(serverManager)` 方法，停止 WebServer 。

5.4 onRefresh

覆写 `#onRefresh()` 方法，在容器初始化时，完成 WebServer 的创建（不包括启动）。代码如下：

```
// ReactiveWebServerApplicationContext.java

@Override
protected void onRefresh() {
    // <1> 调用父方法
    super.onRefresh();
    try {
```

```
            // <2> 创建 WebServer
            createWebServer();
        } catch (Throwable ex) {
            throw new ApplicationContextException("Unable to start reactive web server", ex);
        }
    }
```

<1> 处，调用父 `#onRefresh()` 方法，执行父逻辑。这块，暂时不用了解。
<2> 处，调用 `#createWebServer()` 方法，创建 WebServer 对象。详细解析，见 「5.4.1 createWebServer」 。

5.4.1 createWebServer

`#createWebServer()` 方法，创建 WebServer 对象。代码如下：

```
// ReactiveWebServerApplicationContext.java

private void createWebServer() {
 // 获得 ServerManager 对象。
    ServerManager serverManager = this.serverManager;
 // 如果不存在，则进行初始化
 if (serverManager == null) {
     this.serverManager = ServerManager.get(getWebServerFactory()); // <1>
    }
 // <2> 初始化 PropertySource
    initPropertySources();
}
```

<1> 处，调用 `#getWebServerFactory()` 方法，获得 ReactiveWebServerFactory 对象。代码如下：

```
// ReactiveWebServerApplicationContext.java

protected ReactiveWebServerFactory getWebServerFactory() {
    // Use bean names so that we don't consider the hierarchy
    // 获得 ServletWebServerFactory 类型对应的 Bean 的名字们
    String[] beanNames = getBeanFactory().getBeanNamesForType(ReactiveWebServerFactory.class);
    // 如果是 0 个，抛出 ApplicationContextException 异常，因为至少要一个
    if (beanNames.length == 0) {
        throw new ApplicationContextException("Unable to start ReactiveWebApplicationContext due to missing " +
    }
    // 如果是 > 1 个，抛出 ApplicationContextException 异常，因为不知道初始化哪个
    if (beanNames.length > 1) {
        throw new ApplicationContextException("Unable to start ReactiveWebApplicationContext due to multiple "
                + "ReactiveWebServerFactory beans : " + StringUtils.arrayToCommaDelimitedString(beanNam
    }
    // 获得 ReactiveWebServerFactory 类型对应的 Bean 对象
    return getBeanFactory().getBean(beanNames[0], ReactiveWebServerFactory.class);
}
```

- 默认情况下，此处返回的会是 `org.springframework.boot.web.embedded.netty.NettyReactiveWebServerFactory` 对象。
- 在我们引入 `spring-boot-starter-webflux` 依赖时，`org.springframework.boot.autoconfigure.web.reactive.ReactiveWebServerFactoryConfiguration` 在自动配置时，会配置出 NettyReactiveWebServerFactory Bean 对象。因此，此时会获得

NettyReactiveWebServerFactory 对象。

<1> 处，调用 ServerManager#get(ReactiveWebServerFactory) 静态方法，创建 ServerManager 对象。

<2> 处，调用父 #initPropertySources() 方法，初始化 PropertySource 。

5.5 finishRefresh

覆写 #finishRefresh() 方法，在容器初始化完成时，启动 WebServer 。代码如下：

```
// ReactiveWebServerApplicationContext.java

@Override
protected void finishRefresh() {
    // <1> 调用父方法
    super.finishRefresh();
    // <2> 启动 WebServer
    WebServer webServer = startReactiveWebServer();
    // <3> 如果创建 WebServer 成功，发布 ReactiveWebServerInitializedEvent 事件
    if (webServer != null) {
        publishEvent(new ReactiveWebServerInitializedEvent(webServer, this));
    }
}
```

<1> 处，调用 #finishRefresh() 方法，执行父逻辑。这块，暂时不用了解。
<2> 处，调用 #startReactiveWebServer() 方法，启动 WebServer 。详细解析，见 「5.5.1 startReactiveWebServer」 。
<3> 处，如果创建 WebServer 成功，发布 ReactiveWebServerInitializedEvent 事件。

5.5.1 startReactiveWebServer

#startReactiveWebServer() 方法，启动 WebServer 。代码如下：

```
// ReactiveWebServerApplicationContext.java

private WebServer startReactiveWebServer() {
    ServerManager serverManager = this.serverManager;
    // <1> 获得 HttpHandler
    // <2> 启动 WebServer
    ServerManager.start(serverManager, this::getHttpHandler);
    // <3> 获得 WebServer
    return ServerManager.getWebServer(serverManager);
}
```

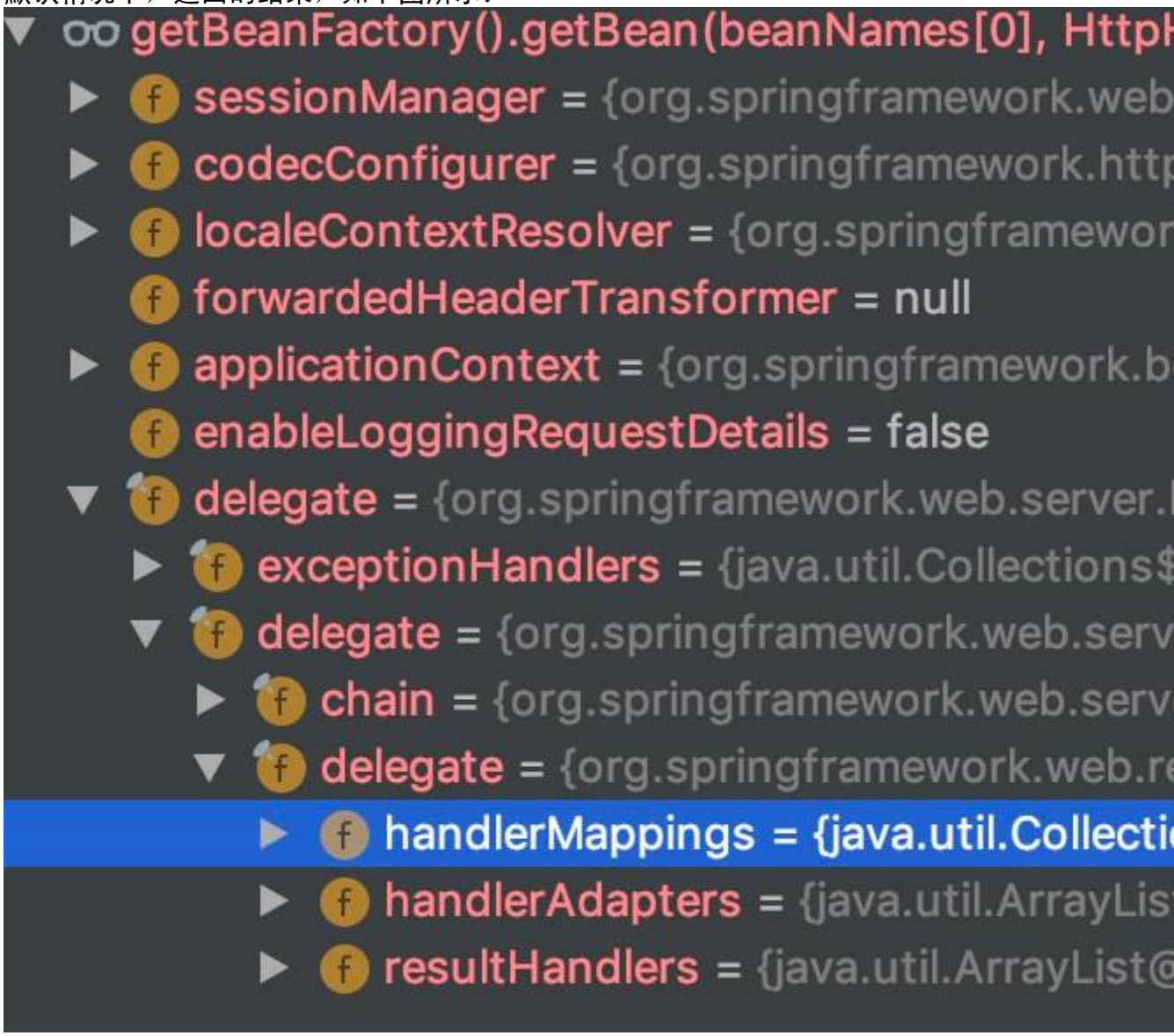
<1> 处，调用 #getHttpHandler() 方法，获得 HttpHandler 对象。代码如下：

```
// ReactiveWebServerApplicationContext.java

protected HttpHandler getHttpHandler() {
    // Use bean names so that we don't consider the hierarchy
    // 获得 HttpHandler 类型对应的 Bean 的名字们
    String[] beanNames = getBeanFactory().getBeanNamesForType(HttpHandler.class);
    // 如果是 0 个，抛出 ApplicationContextException 异常，因为至少要一个
```

```
        if (beanNames.length == 0) {
            throw new ApplicationContextException("Unable to start ReactiveWebApplicationContext due to missing Htt
        }
        // 如果是 > 1 个，抛出 ApplicationContextException 异常，因为不知道初始化哪个
        if (beanNames.length > 1) {
            throw new ApplicationContextException("Unable to start ReactiveWebApplicationContext due to multiple Ht
                    + StringUtils.arrayToCommaDelimitedString(beanNames));
        }
        // 获得 HttpHandler 类型对应的 Bean 对象
        return getBeanFactory().getBean(beanNames[0], HttpHandler.class);
    }
```

- 默认情况下，返回的结果，如下图所示：



- 该 HttpHandler Bean 对象，是在 `org.springframework.boot.autoconfigure.web.reactive.HttpHandlerAutoConfiguration` 配置类上，被初始化出来。

<2> 处，调用 `ServerManager#start(ServerManager manager, Supplier<HttpHandler> handlerSupplier)` 静态方法，启动 WebServer 。

<3> 处，调用 `ServerManager#getWebServer(serverManager)` 静态方法，获得 WebServer 对象。

5.6 onClose

覆写 `#onClose()` 方法，在 Spring 容器被关闭时，关闭 WebServer 。代码如下：

```
// ReactiveWebServerApplicationContext.java

@Override
protected void onClose() {
 // 调用父类方法
 super.onClose();
 // 关闭 WebServer
    stopAndReleaseReactiveWebServer();
}
```

6. AnnotationConfigReactiveWebServerApplicationCon

`org.springframework.boot.web.reactive.context.AnnotationConfigReactiveWebServerApplicationContext` ，继承 ReactiveWebServerApplicationContext 类，实现 AnnotationConfigRegistry 接口，作用和 AnnotationConfigServletWebServerApplicationContext 相同，就不重复赘述了。

如果真要看的胖友，参考 《精尽 Spring Boot 源码分析 —— ServletWebServerApplicationContext》 的 「3. AnnotationConfigServletWebServerApplicationContext」 即可。

666. 彩蛋

水更一篇，无可“狡辩”。嘻嘻。

文章目录

回到首页