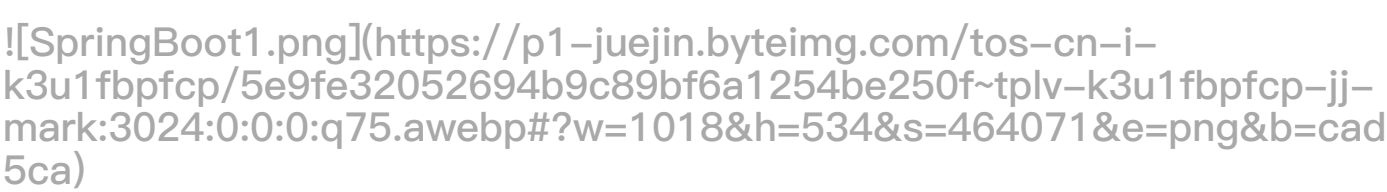
【Spring Boot 源码学习】初识SpringApplication

《Spring Boot 源码学习系列》



引言

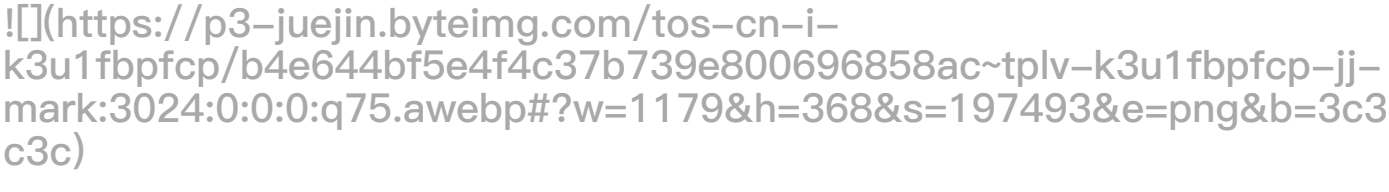
往期的博文，**Huazie** 围绕 **Spring Boot** 的核心功能，带大家从总整体上了解 **Spring Boot** 自动配置的原理以及自动配置核心组件的运作过程。这些内容大家需要重点，只有了解这些基础的组件和功能，我们在后续集成其他三方类库的 **Starters** 时，才能够更加清晰地了解它们都运用了自动配置的哪些功能。

在学习上述 **Spring Boot** 核心功能的过程中，相信大家可能都会尝试启动自己新建的 **Spring Boot** 的项目，并 **Debug** 看看具体的执行过程。本篇开始就将从 **Spring Boot** 的启动类 `SpringApplication` 上入手，带领大家了解 **Spring Boot** 启动过程中所涉及到的源码和知识点。

主要内容

1. Spring Boot 应用程序的启动

在 《【Spring Boot 源码学习】@SpringBootApplication 注解》这篇博文中，我们新建了一个基于 **Spring Boot** 的测试项目。



如上图中的 `DemoApplication` 就是我们这里 **Spring Boot** 项目的入口类。

同时，我们可以看到 `DemoApplication` 的 `main` 方法中，直接调用了 `SpringApplication` 的静态方法 `run`，用于启动整个 **Spring Boot** 项目。

先来看看 `run` 方法的源码：

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

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

阅读上述 `run` 方法，我们可以看到实际上是 `new` 了一个 `SpringApplication` 对象【其构造参数 `primarySources` 为加载的主要资源类，通常就是 `SpringBoot` 的入口类】，并调用其 `run` 方法【其参数 `args` 为传递给应用程序的参数信息】启动，然后返回一个应用上下文对象 `ConfigurableApplicationContext` 。

通过观察这个内部的 `run` 方法实现，我们也可以在自己的 **Spring Boot** 启动入口类中，像如下这样去写：

```
@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication springApplication = new
SpringApplication(DemoApplication.class);
        // 这里可以调用 SpringApplication 提供的 setXX 或 addXX 方法来定
```

制化设置
        springApplication.run(args);
    }

}

```

2. SpringApplication 的实例化
-----------------------

上面已经看到我们在实例化 `SpringApplication` 了，废话不多说，直接翻看其源码【**Spring Boot 2.7.9**】：

```
    public SpringApplication(Class<?>... primarySources) {
        this(null, primarySources);
    }

    @SuppressWarnings({ "unchecked", "rawtypes" })
    public SpringApplication(ResourceLoader resourceLoader, Class<?>...
primarySources) {
        this.resourceLoader = resourceLoader;
        Assert.notNull(primarySources, "PrimarySources must not be null");
        this.primarySources = new
LinkedHashSet<>(Arrays.asList(primarySources));
        // 推断web应用类型
        this.webApplicationType =
WebApplicationType.deduceFromClasspath();
        // 加载并初始化 BootstrapRegistryInitializer及其实现类
        this.bootstrapRegistryInitializers = new ArrayList<>(
getSpringFactoriesInstances(BootstrapRegistryInitializer.class));   //
加载并初始化 ApplicationContextInitializer及其实现类
        setInitializers((Collection)
getSpringFactoriesInstances(ApplicationContextInitializer.class));
        // 加载并初始化ApplicationListener及其实现类
        setListeners((Collection)
getSpringFactoriesInstances(ApplicationListener.class));
        // 推断入口类
        this.mainApplicationClass = deduceMainApplicationClass();
    }

```

由上可知，`SpringApplication` 提供了两个构造方法，而其核心的逻辑都在第
```

二个构造方法中实现。

2.1 构造方法参数

我们从上述源码可知，`SpringApplication` 的第二个构造方法有两个参数，分别是：

* `ResourceLoader resourceLoader` ： `ResourceLoader` 为资源加载的接口，它用于在**Spring Boot** 启动时打印对应的 **banner** 信息，默认采用的就是 `DefaultResourceLoader`。实操过程中，如果未按照 **Spring Boot** 的“约定”将 **banner** 的内容放置于 `classpath` 下，或者文件名不是 `banner.*` 格式，默认资源加载器是无法加载到对应的 **banner** 信息的，此时则可通过 `ResourceLoader` 来指定需要加载的文件路径【这个后面我们专门来实操一下，敬请期待】。
* `Class<?>... primarySources` ： 主要的 **bean** 来源，该参数为可变参数，默认我们会传入 **Spring Boot** 的入口类【即 `main` 方法所在的类】，如上面我们的 `DemoApplication` 。如果作为项目的引导类，该类需要满足一个条件，就是被注解 `@EnableAutoConfiguration` 或其组合注解标注。在前面的《【Spring Boot 源码学习】@SpringBootApplication 注解》博文中，我们已经知道 `@SpringBootApplication` 注解中包含了 `@EnableAutoConfiguration` 注解，因此被 `@SpringBootApplication` 注解标注的类也可作为参数传入。当然，`primarySources` 也可传入其他普通类，但只有传入被`@EnableAutoConfiguration` 标注的类才能够开启 **Spring Boot** 的自动配置。

有些朋友，可能对 `primarySources` 这个可变参数的描述有点疑惑，下面我们就用实例来演示以其他引导类为入口类进行 **Spring Boot** 项目启动：

* 首先，我们在入口类 `DemoApplication` 的同级目录创建一个 `SecondApplication`类，使用 `@SpringBootApplication` 进行注解。

```
@SpringBootApplication
public class SecondApplication {
}
```

* 然后，将 `DemoApplication` 修改成如下：

```
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(SecondApplication.class, args);
    }
}

```
* 最后，我们来运行 `DemoApplication` 的 `main` 方法。

从上图可以看出，我们的应用依然能正常启动，并完成自动配置。因此，决定 **Spring Boot** 启动的入口类并不是一定是 `main` 方法所在类，而是直接或间接被 `@EnableAutoConfiguration` 标注的类。

翻看 `SpringApplication` 的源码，我们在其中还能看到它提供了追加 `primarySources` 的方法，如下所示：

```
public void addPrimarySources(Collection<Class<?>> additionalPrimarySources) {
    this.primarySources.addAll(additionalPrimarySources);
}

```
如果采用 1 中最后的方式启动 **Spring Boot** ，我们就可以调用 `addPrimarySources` 方法来追加额外的 `primarySources`。

我们继续回到 `SpringApplication` 的构造方法里，可以看到如下的代码：

```
this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));

```
上述这里将 `primarySources` 参数转换为 `LinkedHashSet` 集合，并赋值给 `SpringApplication` 的私有成员变量 `Set<Class<?>> primarySources`。

> **知识点：** `LinkedHashSet` 是 **Java** 集合框架中的类，它继承自 `HashSet`，因此具有哈希表的查找性能。这是一个同时使用链表和哈希表特性的数据结构，其中链表用于维护元素的插入顺序。也即是说，当你向 `LinkedHashSet` 添加元素时，元素将按照添加的顺序被存储，并且能够被遍历输出。
> 此外，`LinkedHashSet` 还确保了 **元素的唯一性，即重复的元素在集合中只会存在一份**。
> 如果需要频繁遍历集合，那么 `LinkedHashSet` 可能会比 `HashSet` 效率更高，因为其通过维护一个双向链表来记录元素的添加顺序，从而支持按照插入顺序排序的迭代。但需要注意的是，`LinkedHashSet` 是非线程安全的，如果有多个线程同时访问该集合容器，可能会引发并发问题。

2.2 Web 应用类型推断

我们继续往下翻看源码，这里调用了 `WebApplicationType` 的 `deduceFromClasspath` 方法来进行 **Web** 应用类型的推断。

```
this.webApplicationType = WebApplicationType.deduceFromClasspath();
```

我们继续翻看 `WebApplicationType` 的源码：

```
public enum WebApplicationType {
    // 非Web应用类型
    NONE,
    // 基于Servlet的Web应用类型
    SERVLET,
    // 基于reactive的Web应用类型
    REACTIVE;

    private static final String[] SERVLET_INDICATOR_CLASSES = {
”javax.servlet.Servlet”,
”org.springframework.web.context.ConfigurableWebApplicationContext”
};

    private static final String WEBMVC_INDICATOR_CLASS =
”org.springframework.web.servlet.DispatcherServlet”;
```

```
    private static final String WEBFLUX_INDICATOR_CLASS =
"org.springframework.web.reactive.DispatcherHandler";

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

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

`WebApplicationType` 是一个定义了可能的Web应用类型的枚举类，该枚举类中包含了三块逻辑：

* **枚举类型** ： 非 **Web** 应用、基于 **Servlet** 的 **Web** 应用和基于 **reactive** 的 **Web** 应用。
* **用于下面推断的常量**
* **推断类型的方法 deduceFromClasspath** ：
+ 当 `DispatcherHandler` 存在，并且 `DispatcherServlet` 和 `ServletContainer` 都不存在，则返回类型为 `WebApplicationType.REACTIVE`。
+ 当 `Servlet` 或 `ConfigurableWebApplicationContext` 任何一个不存在时，则说明当前用为非 **Web** 应用，返回 `WebApplicationType.NONE`。
+ 当应用不为 **reactive Web** 应用，并且 `Servlet` 和 `ConfigurableWebApplicationContext` 都存在的情况下，则返回 `WebApplicationType.SERVLET`。

在上述的 `deduceFromClasspath` 方法中，我们可以看到，在判断的过程中使用到了 `ClassUtils` 的 `isPresent` 方法。该工具类方法就是通过反射创建指定的类，根据在创建过程中是否抛出异常来判断该类是否存在。

k3u1fbpfcp/413e2fff697241d3ab26a5832c94dbd7~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=900&h=352&s=210896&e=png&b=303134)

2.3 加载 BootstrapRegistryInitializer

```
this.bootstrapRegistryInitializers = new ArrayList<>(
        getSpringFactoriesInstances(BootstrapRegistryInitializer.class));
```

上述逻辑用于加载并初始化 `BootstrapRegistryInitializer` 及其相关的类。

`BootstrapRegistryInitializer` 是 **Spring Cloud Config** 的组件之一，它的作用是在应用程序启动时初始化 **Spring Cloud Config** 客户端。

在 **Spring Cloud Config** 中，客户端通过向配置中心（**Config Server**）发送请求来获取应用程序的配置信息。而 `BootstrapRegistryInitializer` 就是负责将配置中心的相关信息注册到 **Spring** 容器中的。

由于篇幅有限，有关 `BootstrapRegistryInitializer` 更详细的内容，笔者后续专门讲解。

2.4 加载 ApplicationContextInitializer

```
setInitializers((Collection)
getSpringFactoriesInstances(ApplicationContextInitializer.class));
```

上述代码用于加载并初始化 `ApplicationContextInitializer` 及其相关的类。

`ApplicationContextInitializer` 是 **Spring** 框架中的一个接口，它的主要作用是在**Spring**容器刷新之前初始化 `ConfigurableApplicationContext`。这个接口的实现类可以被视为回调函数

，它们的 `onApplicationEvent` 方法会在**Spring** 容器启动时被自动调用，从而允许开发人员在容器刷新之前执行一些自定义的操作。

例如，我们可能需要在这个时刻加载一些配置信息，或者对某些 **bean** 进行预处理等。通过实现 `ApplicationContextInitializer` 接口并重写其 `onApplicationEvent` 方法，就可以完成这些定制化的需求。

由于篇幅有限，有关 `ApplicationContextInitializer` 更详细的内容，笔者后续专门讲解。

2.5 加载 ApplicationListener

```
setListeners((Collection)
getSpringFactoriesInstances(ApplicationListener.class));
```

上述代码用于加载并初始化 `ApplicationListener` 及其相关的类。

`ApplicationListener` 是 **Spring** 框架提供的一个事件监听机制，它是 **Spring** 应用内部的事件驱动机制，通常被用于监控应用内部的运行状况。其实现的原理是 **观察者设计模式**，该设计模式的初衷是为了实现系统业务逻辑之间的解耦，从而提升系统的可扩展性和可维护性。

我们可以通过自定义一个类来实现 `ApplicationListener` 接口，然后在这个类中定义需要监听的事件处理方法。当被监听的事件发生时，**Spring** 会自动调用这个方法来处理事件。例如，在一个 **Spring Boot** 项目中，我们可能想要在容器启动时执行一些特定的操作，如加载配置等，就可以通过实现 `ApplicationListener` 接口来完成。

由于篇幅有限，有关 `ApplicationListener` 更详细的内容，笔者后续专门讲解。

2.6 推断应用入口类

最后一步，调用 `SpringApplication` 的 `deduceMainApplicationClass` 方法来进行入口类的推断：

```
private Class<?> deduceMainApplicationClass() {
    try {
        StackTraceElement[] stackTrace = new
RuntimeException().getStackTrace();
        for (StackTraceElement stackTraceElement : stackTrace) {
            if (”main”.equals(stackTraceElement.getMethodName())) {
                return Class.forName(stackTraceElement.getClassName());
            }
        }
    } catch (ClassNotFoundException ex) {
        // 这里捕获异常，并继续执行后续逻辑
    }
    return null;
}
```

上述代码的思路就是：

* 首先，创建一个运行时异常，并获得其堆栈数组。
* 接着，遍历数组，判断类的方法中是否包含 `main` 方法。第一个被匹配的类会通过`Class.forName` 方法创建对象，并将其被返回。
* 最后，将上述创建的 `Class` 对象赋值给 `SpringApplication` 的成员变量 `mainApplicationClass`。

总结
==

本篇 **Huazie** 带大家初步了解了 **SpringApplication** 的实例化过程，当然由于篇幅受限，还有些内容暂时无法详解，Huazie 将在后续的博文中继续深入分析。

只有了解 **Spring Boot** 在启动时都做了些什么，我们才能在后续的实践的过程中更好地理解其运行机制，以便遇到问题能更快地定位和排查，使我们应用能够更容易、更方便地接入 **Spring Boot** 。

原文链接: https://juejin.cn/post/7374808507708686386