Spring Boot 启动流程

Springboot启动主类

注解SpringBootApplication作用

```java
@SpringBootApplication
public class SpringBootAnalysisApplication {

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

```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
        @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
        @Filter(type = FilterType.CUSTOM,
                classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
```

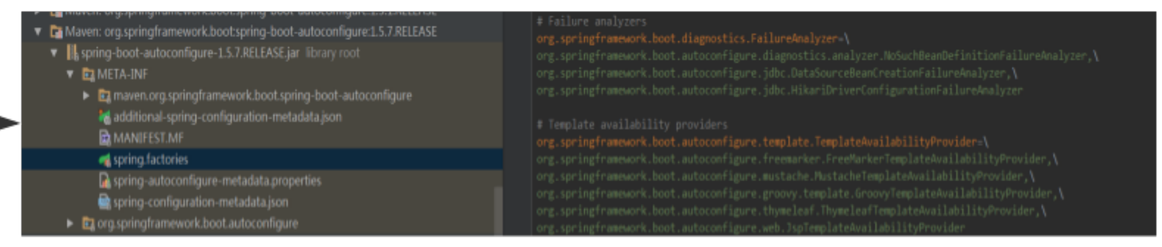
SpringBootApplication这个注解上的注解：

@Inherited这个注解的意思是这个注解所在的类的子类可以继承这个注解。
@SpringBootConfiguration相当用使用@Configuration这个注解(使用这个注解的类相当于beans)。
@ComponentScan扫描包路径。
@EnableAutoConfiguration这个注解的意思是开启自动配置，作用就是借助@Import的支持，收集和注册特定场景相关的bean定义。

@EnableAutoConfiguration开启自动配置，作用就是借助@Import的支持，收集和注册特定场景相关的bean定义。
关键的要属@Import(EnableAutoConfigurationImportSelector.class)
借助EnableAutoConfigurationImportSelector，@EnableAutoConfiguration可以帮助SpringBoot应用将所有符合条件的@Configuration配置都加载到当前SpringBoot创建并使用的IoC容器。

SpringFactoriesLoader其主要功能就是从指定的配置文件META-INF/spring.factories加载配置

将其中org.springframework.boot.autoconfigure.EnableIutoConfiguration对应的配置项通过反射（Java Refletion）实例化为对应的标注了@Configuration的JavaConfig形式的IoC容器配置类，然后汇总为一个并加载到IoC容器。



SpringApplication初始化的时候主要做主要做三件事：

1:根据classpath下是否存在(ConfigurableWebApplicationContext)判断是否要启动一个web applicationContext。

2:SpringFactoriesInstances加载classpath下所有可用的ApplicationContextInitializer

3:SpringFactoriesInstances加载classpath下所有可用的ApplicationListener

```java
    public SpringApplication(Object... sources) {
        //调用initialize方法进行一些初始化的动作。
        initialize(sources);
    }

    private void initialize(Object[] sources) {
        if (sources != null && sources.length > 0) {
            //如果传入的sources有值的话，将Object[]对象转换为List，这里的sources是
            //private final Set<Object> sources = new LinkedHashSet<Object>();
            //这一个Set集合。
            this.sources.addAll(Arrays.asList(sources));
        }
        //判断是否是web环境 1)
        this.webEnvironment = deduceWebEnvironment();
        //加载ApplicationContextInitializer类型的对象 2)
        setInitializers((Collection) getSpringFactoriesInstances(
                ApplicationContextInitializer.class));
        //加载ApplicationListener类型的对象 3)
        setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));
        //寻找启动主类 4)
        this.mainApplicationClass = deduceMainApplicationClass();
    }
```

```java
    private boolean deduceWebEnvironment() {
    //private static final String[] WEB_ENVIRONMENT_CLASSES = { "javax.servlet.Servlet",
    //    "org.springframework.web.context.ConfigurableWebApplicationContext" };
    //WEB_ENVIRONMENT_CLASSES是一个数组内容如上
        for (String className : WEB_ENVIRONMENT_CLASSES) {
            //如果加载不到任何一个类就返回false
            if (!ClassUtils.isPresent(className, null)) {
                return false;
            }
        }
        return true;
    }
```

进入到SpringApplication的run方法

第一个参数source的值是：SpringBootAnalysisApplication.class，在重载的run方法中将传入的SpringBootAnalysisApplication.class封装成了数组，也就是说我们可以调用重载的run方法，传入一个Object[]
第二个参数是：一个可变参数，是我们传入的启动参数。

注意点：
调用run方法的时候会返回一个Spring上下文 ConfigurableApplicationContext的实例

```java
/**
 * Static helper that can be used to run a {@link SpringApplication} from the
 * specified source using default settings.
 * @param primarySource the primary source to load
 * @param args the application arguments (usually passed from a Java main method)
 * @return the running {@link ApplicationContext}
 */
public static ConfigurableApplicationContext run(Class<?> primarySource,
        String... args) {
    return run(new Class<?>[] { primarySource }, args);
}
```

首先遍历初始化过程中加载的SpringApplicationRunListeners，然后调用starting(),开始监听springApplication的启动。

加载SpringBoot配置环境(ConfigurableEnvironment)，如果是通过web容器发布，会加载StandardEnvironment。将配置环境(Environment)加入到监听器对象中(SpringApplicationRunListeners)。

banner属性的设置

ConfigurableApplicationContext(应用配置上下文)创建，根据webEnvironment是否是web环境创建默认的contextClass ,AnnotationConfigEmbeddedWebApplicationContext(通过扫描所有注解类来加载bean)和ConfigurableWebApplicationContext),最后通过BeanUtils实例化上下文对象，并返回。

prepareContext()方法将listeners、environment、applicationArguments、banner等重要组件与上下文对象关联。

refreshContext(context),bean的实例化完成IoC容器可用的最后一道工序。

```java
public ConfigurableApplicationContext run(String... args) {
    StopWatch stopWatch = new StopWatch();
    stopWatch.start();
    ConfigurableApplicationContext context = null;
    Collection<SpringBootExceptionReporter> exceptionReporters = new ArrayList<>();
    configureHeadlessProperty();
    SpringApplicationRunListeners listeners = getRunListeners(args);
    listeners.starting();
    try {
        ApplicationArguments applicationArguments = new DefaultApplicationArguments(
                args);
        ConfigurableEnvironment environment = prepareEnvironment(listeners,
                applicationArguments);
        configureIgnoreBeanInfo(environment);
        Banner printedBanner = printBanner(environment);
        context = createApplicationContext();
        exceptionReporters = getSpringFactoriesInstances(
                SpringBootExceptionReporter.class,
                new Class[] { ConfigurableApplicationContext.class }, context);
        prepareContext(context, environment, listeners, applicationArguments,
                printedBanner);
        refreshContext(context);
        afterRefresh(context, applicationArguments);
        stopWatch.stop();
        if (this.logStartupInfo) {
            new StartupInfoLogger(this.mainApplicationClass)
                    .logStarted(getApplicationLog(), stopWatch);
        }
        listeners.started(context);
        callRunners(context, applicationArguments);
    }
    catch (Throwable ex) {
        handleRunFailure(context, ex, exceptionReporters, listeners);
        throw new IllegalStateException(ex);
    }

    try {
        listeners.running(context);
    }
    catch (Throwable ex) {
        handleRunFailure(context, ex, exceptionReporters, null);
        throw new IllegalStateException(ex);
    }
    return context;
}
```

```java
private Banner printBanner(ConfigurableEnvironment environment) {
    if (this.bannerMode == Banner.Mode.OFF) {
        return null;
    }
    ResourceLoader resourceLoader = this.resourceLoader != null ? this.resourceLoader
            : new DefaultResourceLoader(getClassLoader());
    SpringApplicationBannerPrinter bannerPrinter = new SpringApplicationBannerPrinter(
            resourceLoader, this.banner);
    if (this.bannerMode == Mode.LOG) {
        return bannerPrinter.print(environment, this.mainApplicationClass, logger);
    }
    return bannerPrinter.print(environment, this.mainApplicationClass, System.out);
}
```

```java
protected ConfigurableApplicationContext createApplicationContext() {
    Class<?> contextClass = this.applicationContextClass;
    if (contextClass == null) {
        try {
            contextClass = Class.forName(this.webEnvironment
                    ? DEFAULT_WEB_CONTEXT_CLASS : DEFAULT_CONTEXT_CLASS);
        }
        catch (ClassNotFoundException ex) {
            throw new IllegalStateException(
                    "unable create a default ApplicationContext, "
                            + "please specify an ApplicationContextClass",
                    ex);
        }
    }
    return (ConfigurableApplicationContext) BeanUtils.instantiate(contextClass);
}
```

```java
@Override
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        // Prepare this context for refreshing.
        prepareRefresh();

        // Tell the subclass to refresh the internal bean factory.
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

        // Prepare the bean factory for use in this context.
        prepareBeanFactory(beanFactory);

        try {
            // Allows post-processing of the bean factory in context subclasses.
            postProcessBeanFactory(beanFactory);

            // Invoke factory processors registered as beans in the context.
            invokeBeanFactoryPostProcessors(beanFactory);

            // Register bean processors that intercept bean creation.
            registerBeanPostProcessors(beanFactory);

            // Initialize message source for this context.
            initMessageSource();

            // Initialize event multicaster for this context.
            initApplicationEventMulticaster();

            // Initialize other special beans in specific context subclasses.
            onRefresh();

            // Check for listener beans and register them.
            registerListeners();
```

```java
private void prepareContext(ConfigurableApplicationContext context,
        ConfigurableEnvironment environment, SpringApplicationRunListeners listeners,
        ApplicationArguments applicationArguments, Banner printedBanner) {
    context.setEnvironment(environment);
    postProcessApplicationContext(context);
    applyInitializers(context);
    listeners.contextPrepared(context);
    if (this.logStartupInfo) {
        logStartupInfo(context.getParent() == null);
        logStartupProfileInfo(context);
    }

    // Add boot specific singleton beans
    context.getBeanFactory().registerSingleton("springApplicationArguments",
            applicationArguments);
    if (printedBanner != null) {
        context.getBeanFactory().registerSingleton("springBootBanner", printedBanner);
    }

    // Load the sources
    Set<Object> sources = getSources();
    Assert.notEmpty(sources, "Sources must not be empty");
    load(context, sources.toArray(new Object[sources.size()]));
    listeners.contextLoaded(context);
}
```