前言

我们生成一个spring boot 项目时,会自带一个启动类. 代码如下:

```
@SpringBootApplication
public class SpringBootAnalysisApplication {

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

就是这么简单的代码,构成了spring boot的世界. 那么代码中只有一个@SpringBootApplication 注解 和 调用了SpringApplication#run方法.那么我们先来解析SpringApplication的run方法.

解析

1. 首先调用了org.springframework.boot.SpringApplication#run(Object, String...) 方法.代码如下:

   ```
   public static ConfigurableApplicationContext run(Object source, String... args) {
       return run(new Object[] { source }, args);
   }
   ```

   接着调用如下代码:

   ```
   public static ConfigurableApplicationContext run(Object[] sources, String[] args) {
       return new SpringApplication(sources).run(args);
   }
   ```

   可以发现 首先初始化了SpringApplication,然后调用其实例方法:run.

2. 在 SpringApplication 的构造器中,调用了 initialize 方法.

   ```
   public SpringApplication(Object... sources) {
       initialize(sources);
   }
   ```

3. SpringApplication#initialize方法代码如下:

   ```
   private void initialize(Object[] sources) {
       if (sources != null && sources.length > 0) {
           this.sources.addAll(Arrays.asList(sources));
       }
       this.webEnvironment = deduceWebEnvironment();
       setInitializers((Collection) getSpringFactoriesInstances(
               ApplicationContextInitializer.class));
       setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));
       this.mainApplicationClass = deduceMainApplicationClass();
   }
   ```

   可以看到做了如下5件事:

   1. 如果sources长度大于0的话,加入到SpringApplication的sources中,该sources是一个LinkedHashSet.
   2. 调用deduceWebEnvironment方法判断是否是web环境
   3. 设置initializers.
   4. 设置Listeners.
   5. 设置mainApplicationClass.

4. deduceWebEnvironment代码如下:

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

   private boolean deduceWebEnvironment() {
       for (String className : WEB_ENVIRONMENT_CLASSES) {
           if (!ClassUtils.isPresent(className, null)) {
               return false;
           }
       }
       return true;
   }
   ```

   可以发现会调用ClassUtils类的isPresent方法，检查classpath中是否存在javax.servlet.Servlet类和org.springframework.web.context.ConfigurableWebApplicationContext类,如果存在的话,返回true.否则返回false.

5. 在设置Initializers时首先调用getSpringFactoriesInstances方法加载ApplicationContextInitializer.然后直接赋值给initializers.代码如下:

   ```
   private <T> Collection<? extends T> getSpringFactoriesInstances(Class<T> type) {
       return getSpringFactoriesInstances(type, new Class<?>[] {});
   }
   ```

   转而调用如下代码:

   ```
   private <T> Collection<? extends T> getSpringFactoriesInstances(Class<T> type,
           Class<?>[] parameterTypes, Object... args) {
       ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
       // Use names and ensure unique to protect against duplicates
       // 使用Set保存names来避免重复元素
       Set<String> names = new LinkedHashSet<String>(
               SpringFactoriesLoader.loadFactoryNames(type, classLoader));
       // 根据names来进行实例化
       List<T> instances = createSpringFactoriesInstances(type, parameterTypes,
               classLoader, args, names);
       // 对实例进行排序
       AnnotationAwareOrderComparator.sort(instances);
       return instances;
   }
   ```

   该方法逻辑如下:

   1. 首先获得ClassLoader.
   2. 调用SpringFactoriesLoader#loadFactoryNames进行加载,然后放入到LinkedHashSet进行去重.
   3. 调用createSpringFactoriesInstances进行初始化
   4. 排序

   其中SpringFactoriesLoader#loadFactoryNames代码如下:

```
public static List<String> loadFactoryNames(Class<?> factoryClass, ClassLoader classLoader) {
    String factoryClassName = factoryClass.getName();
    try {
        Enumeration<URL> urls = (classLoader != null ? classLoader.getResources(FACTORIES_RESOURC
E_LOCATION) :
                ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
        List<String> result = new ArrayList<String>();
        while (urls.hasMoreElements()) {
            URL url = urls.nextElement();
            Properties properties = PropertiesLoaderUtils.loadProperties(new UrlResource(url));
            String factoryClassNames = properties.getProperty(factoryClassName);
            result.addAll(Arrays.asList(StringUtils.commaDelimitedListToStringArray(factoryClassN
ames)));
        }
        return result;
    }
    catch (IOException ex) {
        throw new IllegalArgumentException("Unable to load [" + factoryClass.getName() +
                "] factories from location [" + FACTORIES_RESOURCE_LOCATION + "]", ex);
    }
}
```

逻辑如下:

1. 获得factoryClassName,对于当前来说factoryClassName =org.springframework.context.ApplicationContextInitializer.
2. 通过传入的classLoader加载META-INF/spring.factories文件.
3. 通过调用PropertiesLoaderUtils#loadProperties将其转为Properties.
4. 获得factoryClassName对应的值进行返回.

对于当前来说,由于我们只加入了spring-boot-starter-web的依赖,因此会加载如下的配置:

1. 在spring-boot/META-INF/spring.factories中.org.springframework.context.ApplicationContextInitializer值如下:

    ```
    org.springframework.context.ApplicationContextInitializer=\
    org.springframework.boot.context.ConfigurationWarningsApplicationContextInitializer,\
    org.springframework.boot.context.ContextIdApplicationContextInitializer,\
    org.springframework.boot.context.config.DelegatingApplicationContextInitializer,\
    org.springframework.boot.context.embedded.ServerPortInfoApplicationContextInitializer
    ```

2. 在spring-boot-autoconfigure/src/main/resources/META-INF/spring.factories中.org.springframework.context.ApplicationContextInitializer值如下:

    ```
    org.springframework.context.ApplicationContextInitializer=\
    org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer,\
    org.springframework.boot.autoconfigure.logging.AutoConfigurationReportLoggingInitializer
    ```

因此会加载6个.

SpringApplication#createSpringFactoriesInstances方法如下:

```
private <T> List<T> createSpringFactoriesInstances(Class<T> type,
        Class<?>[] parameterTypes, ClassLoader classLoader, Object[] args,
        Set<String> names) {
    List<T> instances = new ArrayList<T>(names.size());
    for (String name : names) {
        try {
            Class<?> instanceClass = ClassUtils.forName(name, classLoader);
            Assert.isAssignable(type, instanceClass);
            Constructor<?> constructor = instanceClass
                    .getDeclaredConstructor(parameterTypes);
            T instance = (T) BeanUtils.instantiateClass(constructor, args);
            instances.add(instance);
        }
        catch (Throwable ex) {
            throw new IllegalArgumentException(
                    "Cannot instantiate " + type + " : " + name, ex);
        }
    }
    return instances;
}
```

逻辑如下:遍历传入的names,也就是之前通过SpringFactoriesLoader加载的类名.通过遍历,依次调用其构造器进行初始化.加入到instances.然后进行返回.

对于当前场景来说:
ConfigurationWarningsApplicationContextInitializer,DelegatingApplicationContextInitializer,ServerPortInfoApplicationContextInitializer
初始化没有做任何事.

ContextIdApplicationContextInitializer在初始化时.会获得spring boot的应用名.搜索路径如下:

1. spring.application.name
2. vcap.application.name
3. spring.config.name
4. 如果都没有配置的话,返回application.

代码如下:

```
private static final String NAME_PATTERN = "${spring.application.name:${vcap.application.name:${s
pring.config.name:application}}}";

public ContextIdApplicationContextInitializer() {
    this(NAME_PATTERN);
}

public ContextIdApplicationContextInitializer(String name) {
    this.name = name;
}
```

6. 设置SpringApplication#setListeners时,还是同样的套路.调用getSpringFactoriesInstances加载META-INF/spring.factories中配置的org.springframework.context.ApplicationListener. 对于当前来说.加载的类如下:

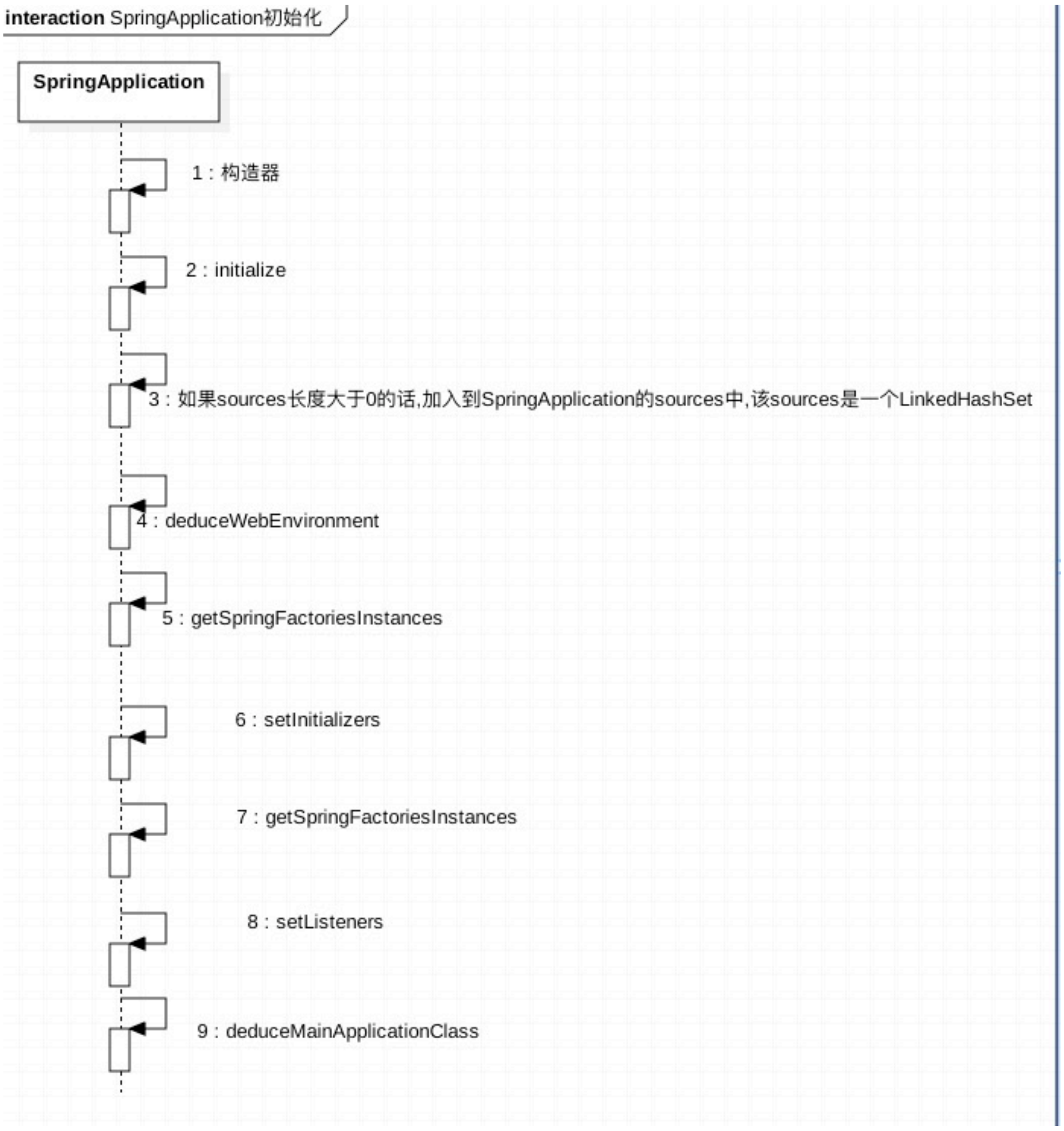
```
org.springframework.context.ApplicationListener=\
org.springframework.boot.ClearCachesApplicationListener,\
org.springframework.boot.builder.ParentContextCloserApplicationListener,\
org.springframework.boot.context.FileEncodingApplicationListener,\
org.springframework.boot.context.config.AnsiOutputApplicationListener,\
org.springframework.boot.context.config.ConfigFileApplicationListener,\
org.springframework.boot.context.config.DelegatingApplicationListener,\
org.springframework.boot.liquibase.LiquibaseServiceLocatorApplicationListener,\
org.springframework.boot.logging.ClasspathLoggingApplicationListener,\
org.springframework.boot.logging.LoggingApplicationListener
```

这些类在构造器中都没有做任何事.

7. 调用SpringApplication#deduceMainApplicationClass方法.获得应用的启动类.该方法通过获取当前方法调用栈，找到main函数的类.代码如下:

```
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) {
        // Swallow and continue
    }
    return null;
}
```

流程图如下:

interaction SpringApplication初始化
SpringApplication
1 : 构造器
2 : initialize
3 : 如果sources长度大于0的话,加入到SpringApplication的sources中,该sources是一个LinkedHashSet
4 : deduceWebEnvironment
5 : getSpringFactoriesInstances
6 : setInitializers
7 : getSpringFactoriesInstances
8 : setListeners
9 : deduceMainApplicationClass