Springboot是如何启动的？

·SpringApplication 初始化

1. 审查ApplicationContext类型
2. 加载ApplicationContextInitializer
3. 加载ApplicationListener

小结：完成初始化后，可以知道如下信息

ApplicationContext是Web或是其他类型

SpringApplication中有一些ApplicationContextInitializer实现类

SpringApplication中有一些ApplicationListener实现类

·Environment初始化

1. 解析命令行参数
2. 创建Environment
3. 配置Environment
4. 配置SpringApplication

小结：环境准备阶段大致工作

根据WebApplicationType枚举创建环境

设置ConversionService用于属性转换

将命令行参数args添加到环境

将外部设置的Profiles添加到环境

绑定SpringApplication属性

发送环境Prepared事件

·ApplicationContext初始化

1. 创建ApplicationContext
2. 设置ApplicationContext
3. 刷新ApplicationContext

·程序运行入口

文件：spring.factories讲解

一个properties文件，在classPath:/META-INF/目录下，每个jar包都有此文件。

String的工具类SpringFactoriesLoader加载，解析此文件

内容：

Key是完整接口名

Value是以逗号间隔完整类型列表，且是key的实现类

**审查ApplicationContext类型**

通过枚举WebApplicationType的静态方法判断，WebApplicationType的三个枚举值：NONE（不是web程序）、SERVLET（基于Servlet的Web程序）、REACTIVE（基于Reactive的web程序）

WebApplicationType枚举中的成员变量存在着完整的class类型。

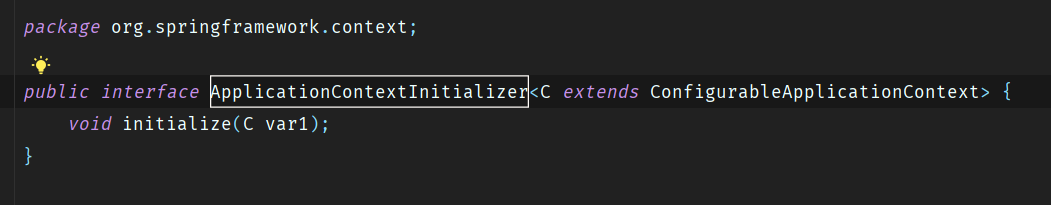
例如：通过pom.xml文件引入spring-boot-starter-web，那classPath下就会由ConfigurableWebApplicationContext和javax.servlet.Servlet类，这样就决定了 程序的ApplicationContext的类型是Servlet

**加载ApplicationContextInitializer**

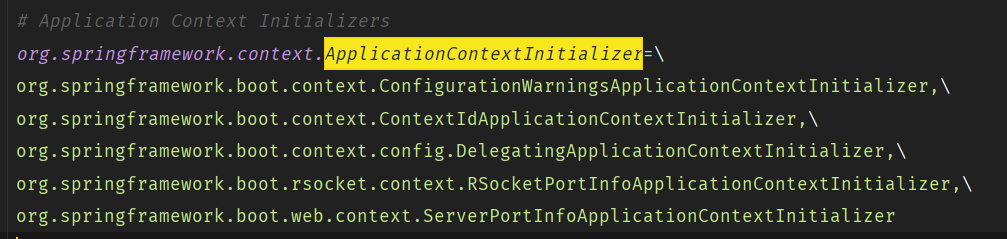
在刷新context之前，做一些额外的工作：添加PropertySource、设置ContextId等，只有一个方法。ApplicationContextInitializer是在spring容器刷新之前执行的一个回调函数。

作用：向springboot容器中注册属性；比如注册一些属性资源额激活文件配置到Environment的环境变量中。

支持Ordered接口和@Order注解

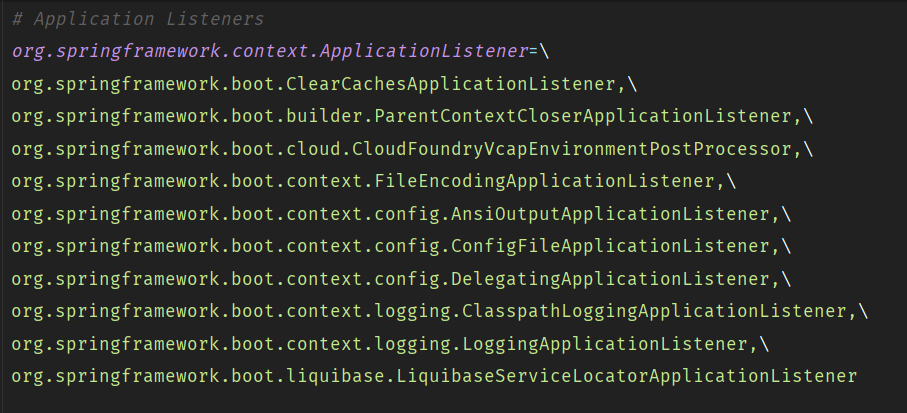


Spring通过SpringFactoriesLoader加载spring.factories的配置，读取key为ApplicationContextInitializer的value（实现类），一系列需要初始化的类



**加载ApplicationListener**

ApplicationListener 用于监听ApplicationEvent事件，它的初始加载流程与ApplicationContextInitializer类似，在spring.factories中找到相应的key



**解析命令行参数**

由main方法的args参数传递

**创建Environment**

在SpringApplication类中



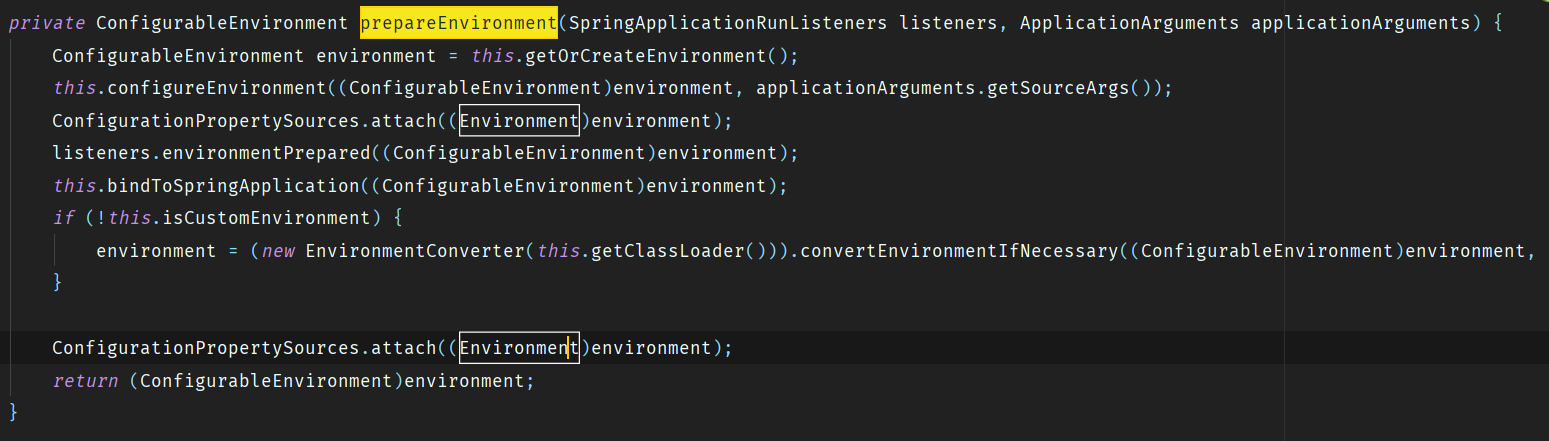
**准备Environment**

环境大致由Profile和PropertyResolver组成：

Profile：是BeanDefinition的逻辑分组，定义Bean时可以指定Profile使SpringBoot在运行时会根据Bean的Profile决定会否注册Bean

PropertyResolver：专门解析属性，SpringBoot会在启动时加载配置文件、系统变量等属性

调用SpringApplication中的prepareEnvironment方法



大致完成：

创建环境、配置环境、设置SpringApplication的属性

**配置Environment**

通过方法configureEnvironment 和 configurePropertySources

设置ConversionService：用于属性转换

将命令行参数添加到环境中

添加额外的ActiveProfiles

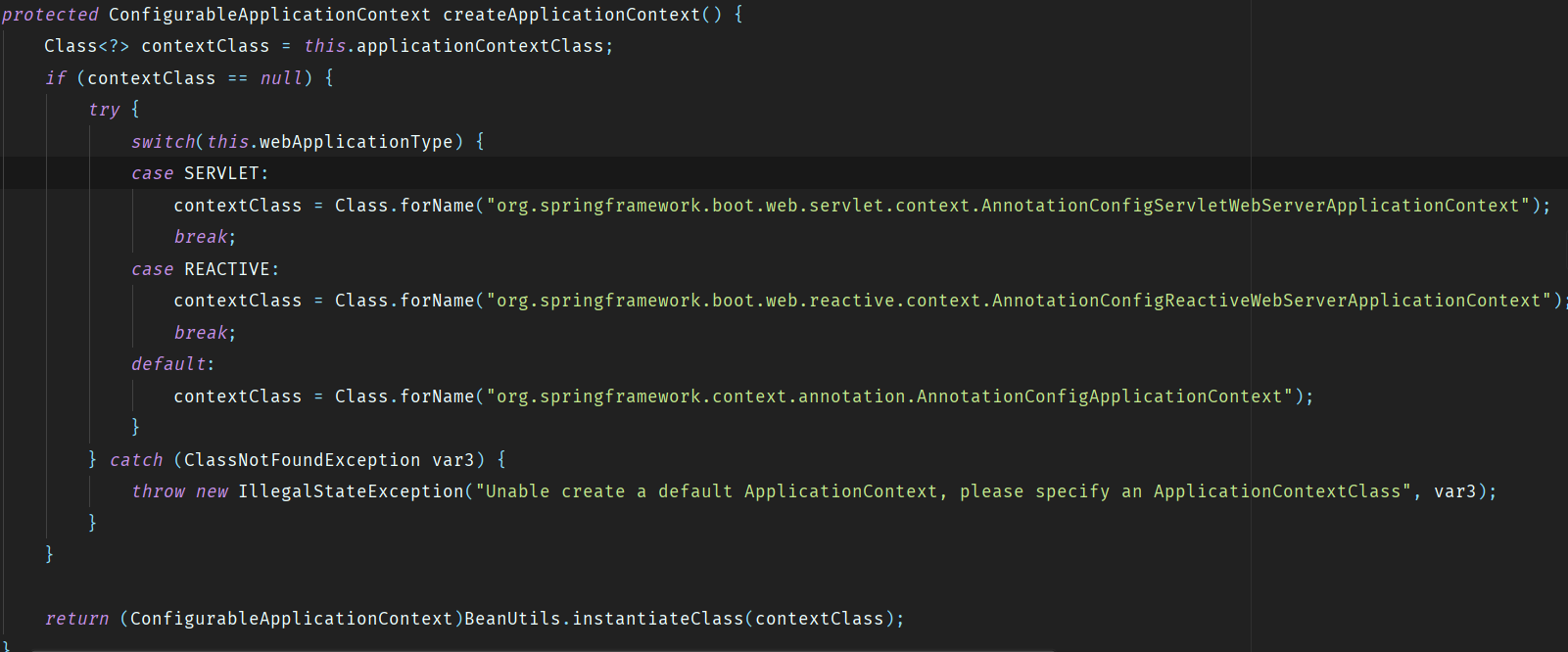
**SpringApplication设置属性**

将已有的属性连接到SpringApplication实例，Springboot会将前缀为spring.main的属性绑定到SpringApplication实例中

**创建ApplicationContext**

前面的工作基本是为了准备**ApplicationContext**而做的，**ApplicationContext**提供了加载Bean、加载资源、发送事件等功能；

根据WebApplicationType来创建不同的ApplicationContext



**准备ApplicationContext**

创建完成后需要初始化，设置环境、应用ApplicationContextInitailizer、注册Source类等信息，归纳如下：

·为ApplicationContext设置环境（之前创建的环境）

·基础设置、操作设置 BeanNameGenerator、ResourceLoader、ConversionService

·执行ApplicationContextInitializer的initialize方法

·注册命令行参数（SpringApplicationArguments）

·注册banner（SpringBootBanner）

·注册sources（由@Configuration注解的类），原理是通过解析注解元数据，创建BeanDefination然后将它注册到ApplicationContext中

**刷新ApplicationContext**

准备刷新

初始化BeanFactory

执行BeanFactoryPostProcessor

注册BeanPostProcessor

初始化MessageSource

初始化事件广播

注册ApplicationListeners

**IOC**

依赖注入，控制反转，IOC容器，用来盛放对象。

BeanDefinitionReader：读取bean的信息。包含读取xml，properties，groovyBean不同读取方式。

BeanDefinition：bean的定义信息。

BeanFactory，bean工厂，根接口。可以生产对象，可以访问当前IOC容器。

反射：利用反射生成对象。

Constructor ctor = Class.getConstructor(); //获取类构造器

Object obj = ctor.newInstanse(); //生成对象 进入IOC容器

Springboot的面试题目

1，bean的作用域有哪些？

IOC容器可以管理bean对象实例，通过xml或注解的方式告诉IOC容器哪些bean需要管理，bean存在生命周期。主要有1，2两种。

1. singleton 单例。意味着整个spring容器注会存在一个Bean的实例
2. prototype 原型意味着每次从IOC容器中获取指定Bean，会返回新的实例对象。

在Spring的web应用里面，增加 会话维度声明周期选项：

1. request 针对每一次http请求都会创建一个新的bean
2. session 以session会话为维度，同一个session下共享同一个bean实例；不同session产生不同的bean实例
3. globalSession 针对全局session维度共享同一个bean的实例。

2，dubbo如果动态感知服务下线？

1. dubbo默认采用zookeeper来实现服务注册和发现。Zookeeper会保存服务提供方的协议地址，dubbo服务消费端会从zookeeper server上查询目标服务地址列表，完成服务发现注册。
2. 通过zookeeper通过心跳检测dubbo服务运行状态。可能会就将服务down掉的机器下线。
3. 消费端通过watch机制，在zookeeper server 的provide节点建立监听，当其子节点发生变化，zookeeper发送事件通知dubbo client，dubbo client会将本地缓存的服务端地址剔除。后续就不会把请求发送到失败的节点上。

3，Spring的事务传播机制有哪些？

7种事务传播行为：

1. required（默认） 当前存在事务，就加入此事务中（合并成一个事务）；不存在则开启一个事务。
2. required\_new 不管是否存在事务，都会开启一个事务执行，新老事务之间相互独立。（当前有事务则先挂起）
3. supports 支持。有事务就加入，没有就不使用事务继续执行。
4. mandatory 要求上下文中必须存在事务，否则会抛出异常。可以防止代码遗漏添加事务。
5. not\_supports 上下文存在事务则先挂起，执行完当前逻辑，结束后恢复事务。
6. never 上下文有事务就抛出异常。
7. nested 嵌套级别事务，如果上下文中存在事务，则创建当前事务的子事务，子事务不能单独提交，即嵌套事务执行，与父事务一起提交。如果不存在事务则新建事务。

子事务的底层实现：B方法执行前会在A所在事务中创建一个savepoint，B异常后回滚到此savepoint。

如果B异常，B一定回滚，B的异常随后会抛给A，如果A catch住了这个异常，A不会回滚，否则A也会回滚。这种情况和REQUIRES\_NEW一样。

如果A发生异常，则A、B都会回滚。