

**SpringBoot**

# 初始化SpringApplication

SpringApplication—>deduceWebApplicationType

deduceWebApplicationType 判定是否为web工程、reactive工程，普通java工程

SpringApplication—> setInitializers

setInitializers 初始化系统级启动器

getSpringFactoriesInstances 获取实例

SpringFactoriesLoader.*loadFactoryNames*

loadSpringFactories

getOrDefault

createSpringFactoriesInstances

使用标准函数获取对应类型的启动器（后面所有的启动类的东西包括监听器 ，全是这么获取的，所以2.0将获取方式加了缓存）：org.springframework.context.ApplicationContextInitializer

使用classLoader扫描jar中或者sourceFolder中的**META-INF/spring.factories**文件，如：

spring-boot-2.0.4.RELEASE.jar!/META-INF/spring.factories

spring-boot-autoconfigure-2.0.4.RELEASE.jar!/META-INF/spring.factories

spring-beans-5.0.8.RELEASE.jar!/META-INF/spring.factories

mybatis-spring-boot-autoconfigure-1.1.1.jar!/META-INF/spring.factories 本例使用了mybatis，这里也有



添加初始化Class（org.springframework.context.ApplicationContextInitializer）

默认如下六种（"org.springframework.boot.context.ConfigurationWarningsApplicationContextInitializer"

"org.springframework.boot.context.ContextIdApplicationContextInitializer"

"org.springframework.boot.context.config.DelegatingApplicationContextInitializer"

"org.springframework.boot.context.embedded.ServerPortInfoApplicationContextInitializer"

"org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer"

"org.springframework.boot.autoconfigure.logging.AutoConfigurationReportLoggingInitializer"）

实例化所有初始化类

3、setListeners

添加监听器（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"

"org.springframework.boot.autoconfigure.BackgroundPreinitializer"）

# RUN

## 获取启动监听器

Run🡪 getRunListeners（SpringApplicationRunListeners）new实例化

其中参数runListenerList，获取方式同上，type= **org.springframework.boot.SpringApplicationRunListene**r）

EventPublishingRunListener为**SpringApplicationRunListene**r接口的实现，在反射获取对象的过程中，初始化广播器SimpleApplicationEventMulticaster，并为广播器添加监听器，即上面的那10个

该启动器（EventPublishingRunListener）包含的事件为：

ApplicationStartingEvent

ApplicationEnvironmentPreparedEvent

ApplicationPreparedEvent

ContextClosedEvent

ApplicationFailedEvent

## （启动监听器）启动

getRunListeners()🡪starting() 启动监听器

EventPublishingRunListener.starting()

启动监听器SpringApplicationRunListeners

初始化事件—系统启动（ApplicationStartingEvent）

根据事件类型过滤监听器，每一个监听器都有自己支持（support）的事件类型。getApplicationListeners(event, type)

启动事件类型：

0 = {LoggingApplicationListener@2060}

1 = {BackgroundPreinitializer@2061}

2 = {DelegatingApplicationListener@2062}

3 = {LiquibaseServiceLocatorApplicationListener@2063}

1、LoggingApplicationListener

日志监听器

2、BackgroundPreinitializer

当启动事件为ApplicationEnvironmentPreparedEvent的时候

多线程--初始化一部分组件主要是messageConvertor等等，当我们有这个类，他就会帮我们引进来，不像过去，我们要手动配置。比如**JsonGenerator**

，**XmlMapper**等等最终也就是这些MappingJackson2XmlHttpMessageConverter，SourceHttpMessageConverter，MappingJackson2HttpMessageConverter

等等

runSafely(**new** ConversionServiceInitializer());  
runSafely(**new** ValidationInitializer());  
runSafely(**new** MessageConverterInitializer());  
runSafely(**new** MBeanFactoryInitializer());  
runSafely(**new** JacksonInitializer());  
runSafely(**new** CharsetInitializer());

3、DelegatingApplicationListener

这个初始化什么都不做，需要广播事件的时候才触发，比如自定义启动事件，自定义一个事件继承ApplicationListener，support启动事件，就会在这个监听器中广播，但是只能定义ApplicationEnvironmentPreparedEvent事件

4、LiquibaseServiceLocatorApplicationListener

使用*liquibase*才需要启动

事件广播完毕，我们要关注的大概主要有，后台启动初始化，自定义事件，还有日志。

ApplicationArguments applicationArguments = **new** DefaultApplicationArguments(  
 args);

初始化系统参数，主要是启动时我们在启动命令里加的最大参数

## 准备环境参数

Run🡪 prepareEnvironment 准备环境参数（启动监听器（准备事件））

将在该函数中调用EventPublishingRunListener.environmentPrepared

初始化事件—配置环境准备（ApplicationEnvironmentPreparedEvent）

监听器SpringApplicationRunListeners

初始化事件—准备环境（参数等）（ApplicationStartingEvent）

### 初始化StandardServletEnvironment

prepareEnvironment—>getOrCreateEnvironment

初始化StandardServletEnvironment，如果不是web，使用StandardEnvironment

StandardServletEnvironment调用继承自次顶级父类AbstractEnvironment的构造函数

AbstractEnvironment🡪customizePropertySources 构造函数调用当前类的customizePropertySources函数

StubPropertySource 初始化servlet环境参数

servletConfigInitParams

servletContextInitParams

如果有jndi操作，添加**jndiProperties**

注意：这里添加propertySource，并不会读取参数，只是添加源

最后调用父类StandardEnvironment（标准环境）的customizePropertySources

MapPropertySource **systemProperties** getSystemProperties

SystemEnvironmentPropertySource **systemEnvironment** getSystemEnvironment

**systemProperties java（相关参数）虚拟机规范、实现版本、java版本、分隔符等等**

**systemEnvironment 系统参数，就是Path等等乱七八糟的都有**

### 配置环境信息

prepareEnvironment🡪configureEnvironment

1、configureEnvironment🡪configurePropertySources

如果有命令行参数，则启动的时候会将启动行参数放到最前

SimpleCommandLinePropertySource

启动行参数使用--server.port=8080

2、configureEnvironment🡪configureProfiles

AbstractEnvironment🡪 doGetActiveProfiles

获取激活配置文件类型dev,test,prod，一般情况下，我们的配置文件放在properties或者yml文件里，所以这里的其实是空的

### 准备环境（参数整理等）

将在该函数中调用EventPublishingRunListener.environmentPrepared

初始化事件—配置环境准备（ApplicationEnvironmentPreparedEvent）

和初始化一样，使用广播器广播事件。

获取环境准备事件的监听器getApplicationListeners(event, type)

0 = {ConfigFileApplicationListener@2193}

1 = {AnsiOutputApplicationListener@2194}

2 = {LoggingApplicationListener@2195}

3 = {ClasspathLoggingApplicationListener@2196}

4 = {BackgroundPreinitializer@2197}

5 = {DelegatingApplicationListener@2198}

6 = {FileEncodingApplicationListener@2199}

1. ConfigFileApplicationListener

使用1中的方法获取postProcessor SpringFactoriesLoader.*loadFactories*

Class=org.springframework.boot.env.EnvironmentPostProcessor

0 = {SystemEnvironmentPropertySourceEnvironmentPostProcessor@2312}

1 = {SpringApplicationJsonEnvironmentPostProcessor@2313}

2 = {CloudFoundryVcapEnvironmentPostProcessor@2314}

3 = {ConfigFileApplicationListener@2193}

依次调用postProcessEnvironment

1. 将系统变量写入 MutablePropertySources.propertySourceList
2. 获取原始环境变量或者系统参数中的spring.application.json参数，获取顺序为0 = {SimpleCommandLinePropertySource@2193} "SimpleCommandLinePropertySource {name='commandLineArgs'}"

1 = {PropertySource$StubPropertySource@2194} "StubPropertySource {name='servletConfigInitParams'}"

2 = {PropertySource$StubPropertySource@2195} "StubPropertySource {name='servletContextInitParams'}"

3 = {MapPropertySource@2196} "MapPropertySource {name='systemProperties'}"

4 = {SystemEnvironmentPropertySourceEnvironmentPostProcessor$OriginAwareSystemEnvironmentPropertySource@2197} "OriginAwareSystemEnvironmentPropertySource {name='systemEnvironment'}"

只取第一个，即，commandLineArgs优先级最高

Json参数总是加载systemProperties或者jndi参数之前，如果没有系统参数，则加在第一个

1. 这个是云用的 vcap.application.; vcap.services.如果有命令行参数加在命令行之后，否则永远是第一个
2. 初始化配置文件—application.yml;properties

postProcessEnvironment🡪addPropertySources

默认添加RandomValuePropertySource(添加随机数的东西，用处暂时找不到，my.secret=${random.value})

addPropertySources初始化系统参数🡪**new** Loader(environment, resourceLoader).load()

构造函数初始化Loader，使用1中的方法：

Class=org.springframework.boot.env.PropertySourceLoader

0 = {PropertiesPropertySourceLoader@3344}

1 = {YamlPropertySourceLoader@3345}

load()🡪initializeActiveProfiles 获取启动文件

loader🡪load();

加载配置文件

ConfigFileApplicationListener.load()

spring.config.additional-location 2.0中加入其他位置的配置文件。

如果spring.config.location中配置了，忽略additional

2.0中除了这里的改动外，大量运用了lambda表达式，可以学习下，确实看着很飘

spring.config.name 配置文件的名称，默认application

加载位置：file:./config/ file:./ classpath:/config/ classpath:/

按顺序加载properties，xml，yml，yaml

加载文件的顺序变成

classpath:/ yaml

classpath:/config/ yaml

file:./ yaml

file:./config/ yaml

……

classpath:/ properties

classpath:/config/ properties

file:./ properties

file:./config/ properties

也就是，配置文件里，file:./config/ properties的参数优先级最高，如果其中文在参数，将由它覆盖之前所有

最终，将配置文件参数解析器放到总和的最后

表示，将来解析的时候由后向前

顺序为

applicationConfigurationProperties

random 只负责随机数

systemEnvironment 环境变量 java -Dserver.port=9000

systemProperties 系统变量server.port=10000

spring.application.json 系统变量中用spring.application.json配置的json参数

servletContextInitParams servletContext启动参数

servletConfigInitParams servlet启动参数

环境变量最大，他是最早加载的，其次是java启动参数（-D-Dserver.port=9023）

使用servletContextXXXX配置的是最大的，即：EmbeddedServletContainerCustomizer，使用该类自定义的部分配置是最大，超越所有配置文件。

先加入无主配置，全局参数，没有在dev prod test里的

加入有主参数后，会将上面参数覆盖

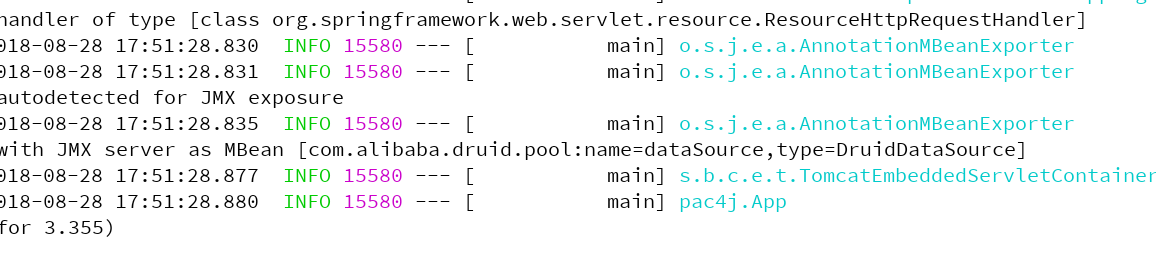
Load()🡪addLoadedPropertySources

将读出的配置文件加入到参数链中，规则也是，加在最后，如果有默认配置文件，加入到默认之前

1. AnsiOutputApplicationListener

终端彩色输出，没太大具体意义

**spring.output.ansi.enabled=true**

大概就这么个玩意没太大意义

**console-available**

这个是System.console，有些平台可能支持这个，具体没试。

1. LoggingApplicationListener

日志类，不说了

1. ClasspathLoggingApplicationListener

同上

1. BackgroundPreinitializer

不支持准备事件

1. DelegatingApplicationListener

context.listener.classes

可以配置自定义的启动器，在准备环境参数阶段，处理参数。

如果配置了的话，在这里直接广播事件

1. FileEncodingApplicationListener

没啥大用，比对fileEncoding的

System.*getProperty*(**"file.encoding"**);

environment.getProperty(**"spring.mandatory-file-encoding"**);

### 绑定参数

prepareEnvironment🡪bindToSpringApplication

将配置信息绑定到application中

## 配置忽略的bean

Run🡪 configureIgnoreBeanInfo

有些bean可能不需要实例化，将配置信息放置在system里**spring.beaninfo.ignore**

## 启动logo

Run🡪 printBanner

自定义启动logo，用处不大

这个文件，放在resources下，就可以了<http://patorjk.com/software/taag/#p=display&v=0&f=Graffiti&t=ASIAINFO>，这个网站生成



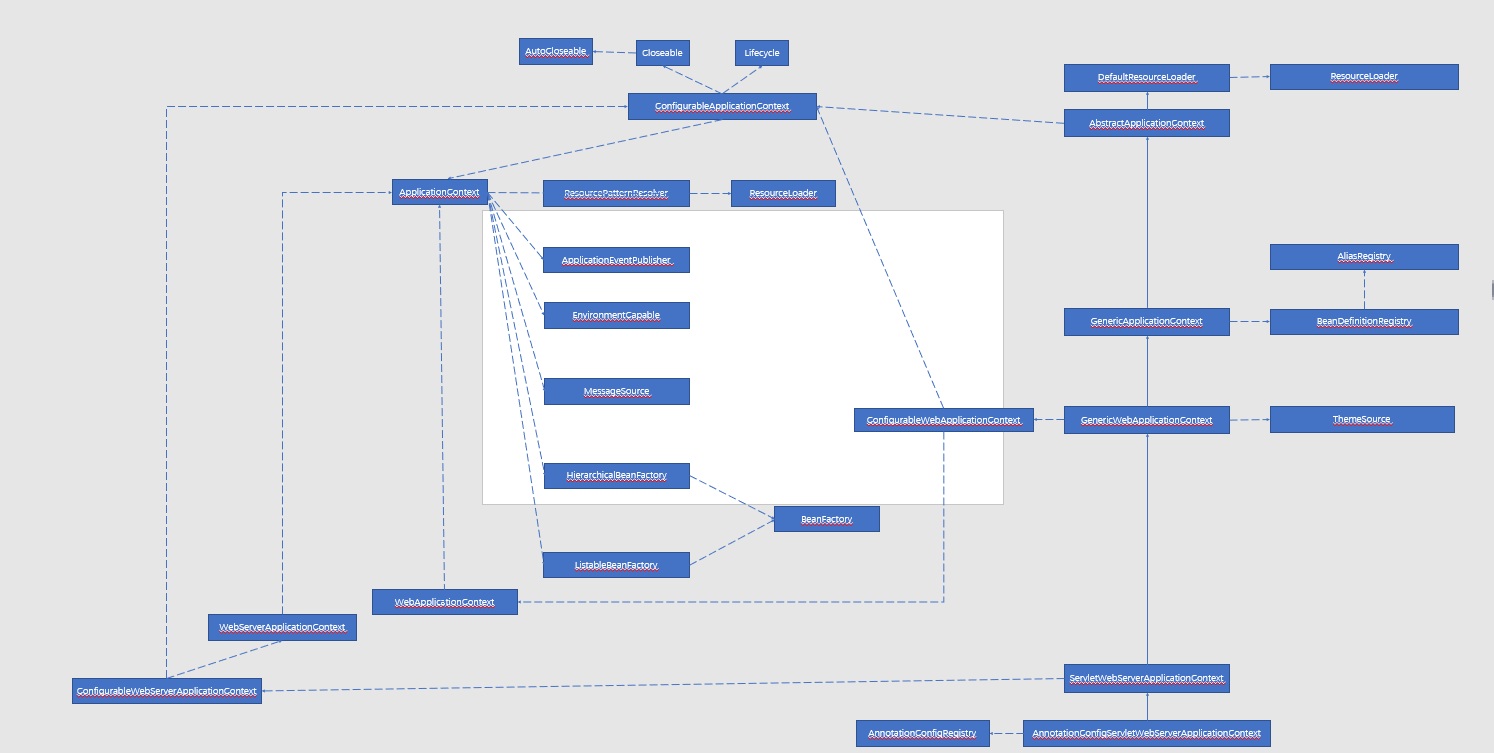
## 初始化ApplicationContext

Run🡪createApplicationContext

实例化

org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContext

spring默认是使用注解的，即便我们引入xml配置文件，也是使用configuration注解做二次解析



（右键打开可放大）

AnnotationConfigServletWebServerApplicationContext 继承示意图

*XmlServletWebServerApplicationContext*

注解和XML方式基本配置一样，只不过XML中实例化XmlBeanDefinitionReader

注解实例化AnnotatedBeanDefinitionReader、ClassPathBeanDefinitionScanner

1. AnnotationConfigServletWebServerApplicationContext

默认的web服务上下文（注解形式的）实现独特接口（AnnotationConfigRegistry，扫描包、注册bean）

1. ServletWebServerApplicationContext

该类会注册、启动一个webserver，在没有特殊声明的情况下，servlet都将注册为小写的bean name，所有的名称为*dispatcherServlet*的servlet，都将注册到/，这个注册将会覆盖（*ServletContextInitializer*，*ServletRegistrationBean*，*FilterRegistrationBean*），这也就解释了2.0后，EmbeddedServletContainerCustomizer被去掉，在这里加入的配置信息，如端口号等，不会覆盖配置文件中的。这个类是可以单独使用，启动服务的，比如在application中setApplicationContextClass，但一般情况下我们不这么用，一般用XML或者注解（Annotation）。

1. GenericWebApplicationContext

*GenericApplicationContext的下级实现（Web实现）*

实现了*ConfigurableWebApplicationContext*接口，希望实现一种编程式的服务启动，替代了原来的web.xml方式。用在这（*WebApplicationInitializer*），如果需要手动添加bean配置（读取配置文件），使用*AbstractRefreshableWebApplicationContext（手动实现loadBeanDefinitions）*—在springboot中几乎没有用（springboot使用@ImportResource将实现XMLBeanDifinitionReader，主动读取配置文件）。

1. GenericApplicationContext

主要的ApplicationContext实现，内置（实例化一个接口工厂DefaultListableBeanFactory），实现了BeanDifinitionRegistry，方便系统在任何时候注册、读取BeanDifinition信息。使用方式如：

*\* GenericApplicationContext ctx = new GenericApplicationContext();  
\* XmlBeanDefinitionReader xmlReader = new XmlBeanDefinitionReader(ctx);  
\* xmlReader.loadBeanDefinitions(new ClassPathResource("applicationContext.xml"));  
\* PropertiesBeanDefinitionReader propReader = new PropertiesBeanDefinitionReader(ctx);  
\* propReader.loadBeanDefinitions(new ClassPathResource("otherBeans.properties"));  
\* ctx.refresh();  
\*  
\* MyBean myBean = (MyBean) ctx.getBean("myBean");*

1. AbstractApplicationContext

上下文的基础定义，定义了refresh函数，里面的很多抽象函数都在后面实现类中实现。定义了spring上下文的规范。

1. DefaultResourceLoader

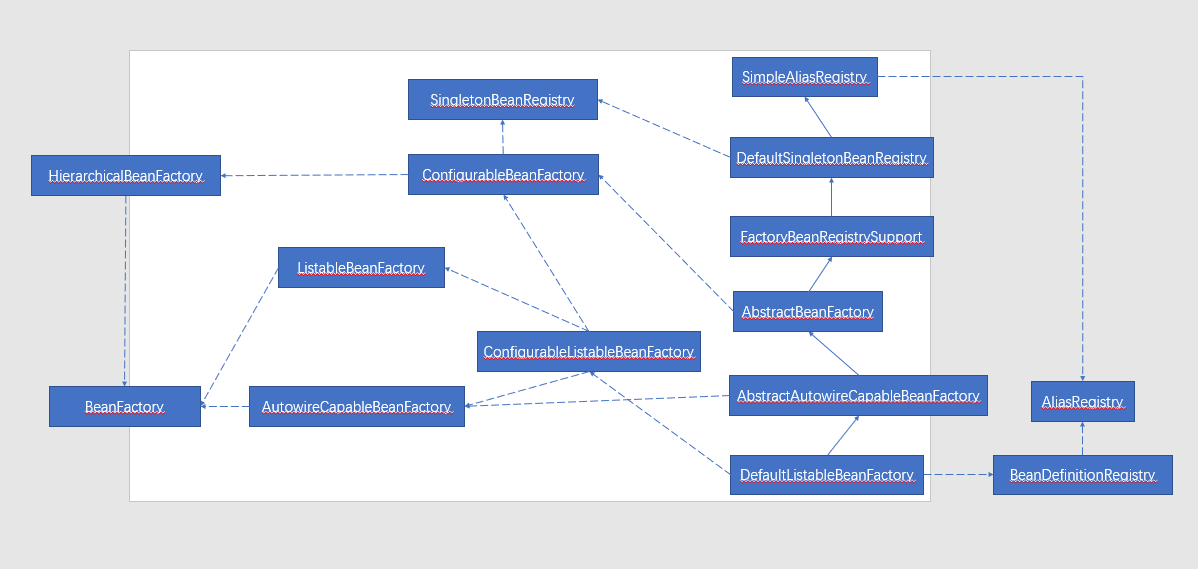
资源读取类

### 创建BeanFactory

父类构造函数分析：

1. GenericApplicationContext

初始化**beanFactory**=**new** DefaultListableBeanFactory();



org.springframework.beans.factory.support.DefaultListableBeanFactory

继承示意图



AbstractAutowireCapableBeanFactory构造函数

添加忽略类（BeanNameAware，BeanFactoryAware，BeanClassLoaderAware）

在依赖注入的时候，忽略这些接口实例。

### 初始化（Bean读取类）

初始化AnnotatedBeanDefinitionReader

所有注解实例都通过该类读取、注册

属性：

1. BeanDefinitionRegistry **registry**

注册中心类，当注解方式实现时，将使用AnnatationXXXX，

1. **beanNameGenerator** = **new** AnnotationBeanNameGenerator()

生成beanName

1. **scopeMetadataResolver** = **new** AnnotationScopeMetadataResolver()

设置默认的作用域代理模式，默认是没有的No

Bean有四个作用域

* 单例(Singleton)：在整个应用中，只创建 Bean 的一个实例。
* 原型(Prototype)：每次注入或者通过 Spring 应用上下文获取的时候，都会创建一个新的 Bean 实例。
* 会话(Session): 在 Web 应用中，为每个会话创建一个 Bean 实例。
* 请求(Request)：在 Web 应用中，为每个请求创建一个 Bean 实例

在非单例模式下，所有的请求都必须使用代理默认来实现，不管是cglib（类）还是jdk（接口），原理是，当我们创建prototype等的时候，spring会创建一个代理，使用代理的模式去调用。原则上这个代理还是单例，但对于调用者来讲，每次都是不同的。

1. **conditionEvaluator** = **new** ConditionEvaluator

用来解释注解，比如某些条件DependsOn或者ConditionalOnXXX

实例化ConditionContextImpl，里面保存着上下文的参数等信息。

条件判断（**conditionEvaluator🡪** shouldSkip）

ConfigurationPhase参数，判断该metaData是Configuration里配置的bean信息，还是普通配置的bean信息，是不同的。

MultiValueMap<String, Object> attributes = metadata.getAllAnnotationAttributes(Conditional.**class**.getName(), **true**);  
Object values = (attributes != **null** ? attributes.get(**"value"**) : **null**);

获取所有Conditional注解，Spring这里获取注解的方式非常复杂，有兴趣可以自行查看

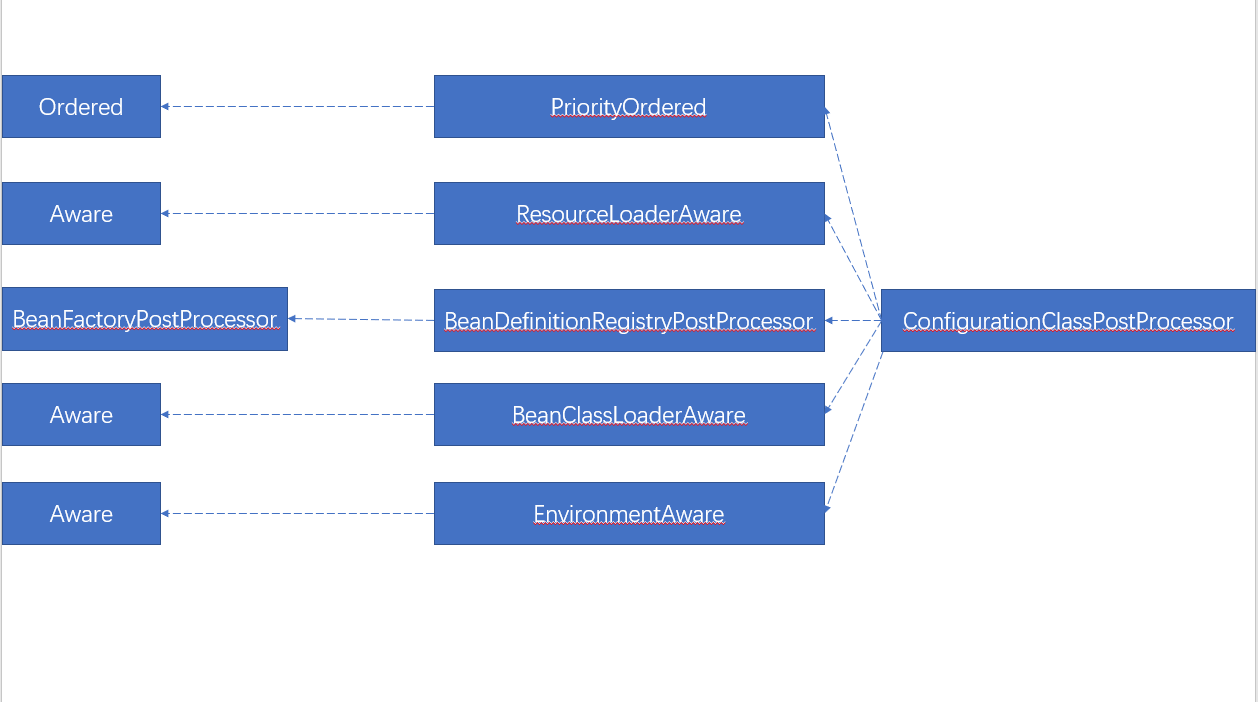
调用注解!condition.matches(**this**.**context**, metadata)

#### 注册PostProcessor

AnnotatedBeanDefinitionReader🡪AnnotationConfigUtils.*registerAnnotationConfigProcessors*

1. **ConfigurationClassPostProcessor（org.springframework.context.annotation.internalConfigurationAnnotationProcessor）**

用于处理*@Configuration*中的Bean，优先级最高，才能在所有的beanPostProcessor执行之前先执行这个



BeanDefinitionRegistryPostProcessor

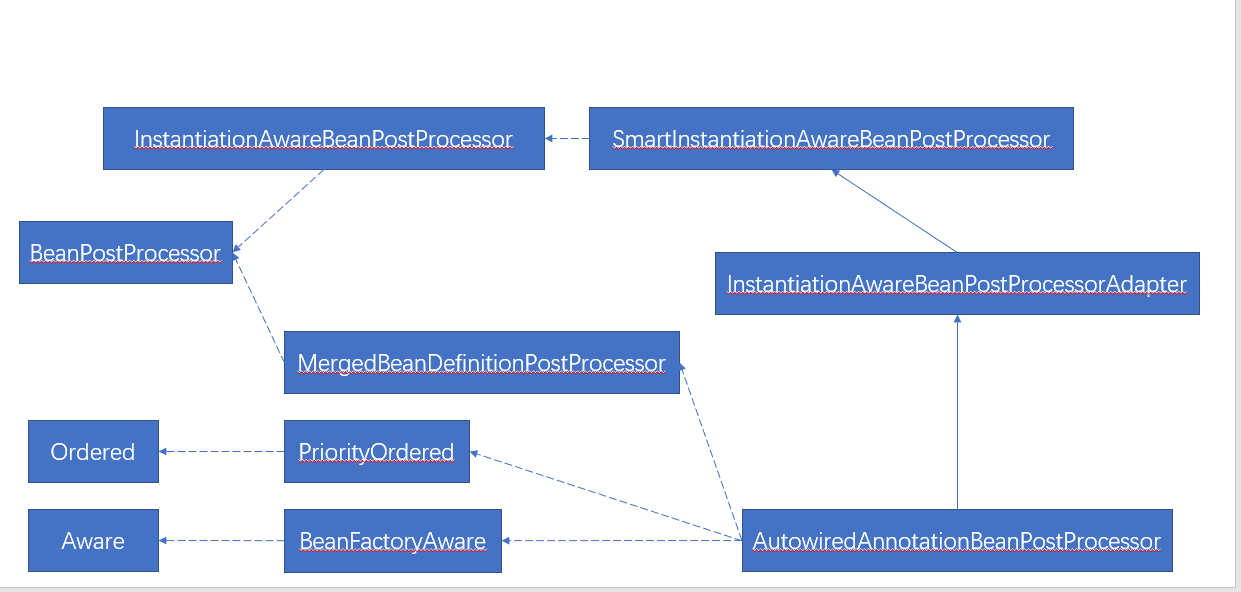
在BeanDifinition已经加载完，并且BeanFactoryPostProcessor执行完之后，但bean开始实例化之前执行，最后一次修改BeanDifinition。

BeanFactoryPostProcessor 实例化所有bean之前，修改BeanDifinition。

1. **AutowiredAnnotationBeanPostProcessor （org.springframework.context.annotation.internalAutowiredAnnotationProcessor）**

处理注解（也会处理XML文件）Bean配置，注入其中的属性，如：*@Autowired*，*@Value*，*@Inject*，只有一个构造函数的情况下使用构造函数，如果存在多个构造函数，则使用多个构造函数中能满足的最大的条件（如：有无参，一个参数，两个参数，会取所有实例（BeanFactory中），假设只能满足一个参数，则使用一个参数的构造函数实例化），默认使用（*context:annotation-config" and "context:component-scan"*）开启这个PostProcessor，注解配置会先进入配置，后置的xml配置，将覆盖注解配置。

postProcessPropertyValues函数实现了InstantiationAwareBeanPostProcessor接口，将属性注入到bean中



MergedBeanDefinitionPostProcessor

合并beanDifinition后的回调。

BeanPostProcessor

Bean实例化之后、用户自定义初始化之前的回调顺序依次为：

BeanPostProcessor🡪 @PostConstruct🡪 InitializingBean🡪init-method

bean post processor

post Construct

after propertiesSet

init method

InstantiationAwareBeanPostProcessorAdapter

SmartInstantiationAwareBeanPostProcessor和InstantiationAwareBeanPostProcessor的默认实现，实际上都是内部用的，如果用户要使用Adapter，继承、实现它，比如我们使用的AutowiredAnnotationBeanPostProcessor

1. **RequiredAnnotationBeanPostProcessor（org.springframework.context.annotation.internalRequiredAnnotationProcessor）**

继承示意图和2中一样，同样实现了InstantiationAwareBeanPostProcessor接口的postProcessPropertyValues，校验required。

1. **CommonAnnotationBeanPostProcessor（org.springframework.context.annotation.internalCommonAnnotationProcessor）**

主要是对于JSR-250的支持，判断也是判断有没有javax.annotation.Resource这个注解，这个通过继承InitDestroyAnnotationBeanPostProcessor支持了—support以下注解：*PostConstruct*，*PreDestroy*。

支撑（*WebServiceRef*，*Resource*）等注解

1. **org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor**

对JPA注解的支撑,没用到

1. **EventListenerMethodProcessor （org.springframework.context.event.internalEventListenerProcessor）**

在bean实例化后，处理一些事情，这里是判断代理模式等等，怀疑所有aop相关，或者bean中method有注解的，都在这里处理了。这个很重要

1. **org.springframework.context.event.internalEventListenerFactory**

供*EventListener*使用的，详细的后面能遇到



### 初始化（Bean扫描类）

初始化ClassPathBeanDefinitionScanner，扫描Bean的助理类，只存在于注解模式下。如果使用XML初始化ApplicationContext，则不初始化该类

ClassPathBeanDefinitionScanner🡪 registerDefaultFilters

注册filter，将所选filter放到**includeFilters**列表中。

主要支持以下注解（*@Component*，*@Repository*，*@Service*，*@Controller*，*ManagedBean*，*Named*）

至此：ApplicationContext初始化完毕。

## 准备上下文ApplicationContext

Run🡪prepareContext

准备上下文信息

### 调用初始化器

prepareContext🡪applyInitializers

1. DelegatingApplicationContextInitializer

其他初始化器的代理，如用户自定义，则在此调用，后续其他初始化器可能会覆盖（customized 初始化器—配置context.initializer.classes，用逗号间隔）

1. ContextIdApplicationContextInitializer

初始化系统名称**spring.application.name**，如果没有这个属性将使用**application**，注意这个只是系统名称，不是contextPath等。将系统名称作为key，系统名称bean（ContextId）作为value，放入beanFactory中。

可以尝试使用1中的初始化器修改系统名称，在后面就会覆盖掉。

1. ConfigurationWarningsApplicationContextInitializer

手动添加一个beanFactoryPostProcessor（ConfigurationWarningsPostProcessor），检查扫描的包（ComponentScan），默认添加**org.springframework**，**org** 并且只有这两个

1. ServerPortInfoApplicationContextInitializer

添加监听器，默认实现ApplicationListener<WebServerInitializedEvent>，监听webServer初始化事件。

1. SharedMetadataReaderFactoryContextInitializer

添加一个BeanFactoryPostProcessor（CachingMetadataReaderFactoryPostProcessor，同样实现了BeanDefinitionRegistryPostProcessor）在springboot和*CachingMetadataReaderFactory*之间搭桥，保证两边配置一致。

1. ConditionEvaluationReportLoggingListener

日志相关的，略过。知道有这么个东西就行了。多个application之前也不共享日志，以后可能有日志相关能用到

### 将启动行参数绑定到beanFactory中，注册一个bean，实例就是参数实例，name= springApplicationArguments

### 注册一个bean 实例就是之前的banner，name=springBootBanner

### 加载Bean定义

prepareContext🡪load

#### 创建加载BeanDifinition的bean（BeanDefinitionLoader）

load🡪createBeanDefinitionLoader

创建加载BeanDifinition的bean

1. 实例化AnnotatedBeanDefinitionReader

由于之前实例化一次了，所以这里基本都是跳过的操作。

1. 实例化XmlBeanDefinitionReader

没有太多要说明的，除了resourceLoader是从applicationContext（实现了resourceLoader接口）中获取

也就是，不管使用哪一种ApplicationContext，都是两种Bean配置方式同时加载的

扫描的时候主函数，我们写主方法的那个，是不扫描的。（存疑）

#### 加载前处理

如果用户自己实例化**beanNameGenerator**，或者**resourceLoader**，这里注入加载器中。一般不需要改名或者改位置。

#### 加载BeanDefinition

BeanDifinitionLoader🡪load()

加载的时候可以有多个主函数的源，这个以后再说，应该理解错了

这里加载的BeanDefinition只是将主函数参数信息加载进去，没有其他的。

AnnotationUtils.*findAnnotation(class,annotationType)经典的获取注解的方式。*

先判断注解是否在当前class上，如果不是的话，获取该类所有的注解，循环，处理每一个注解（不处理java标准注解），添加到set中，如果已经存在的情况下，不处理。递归查找，直到获取到类型符合的。

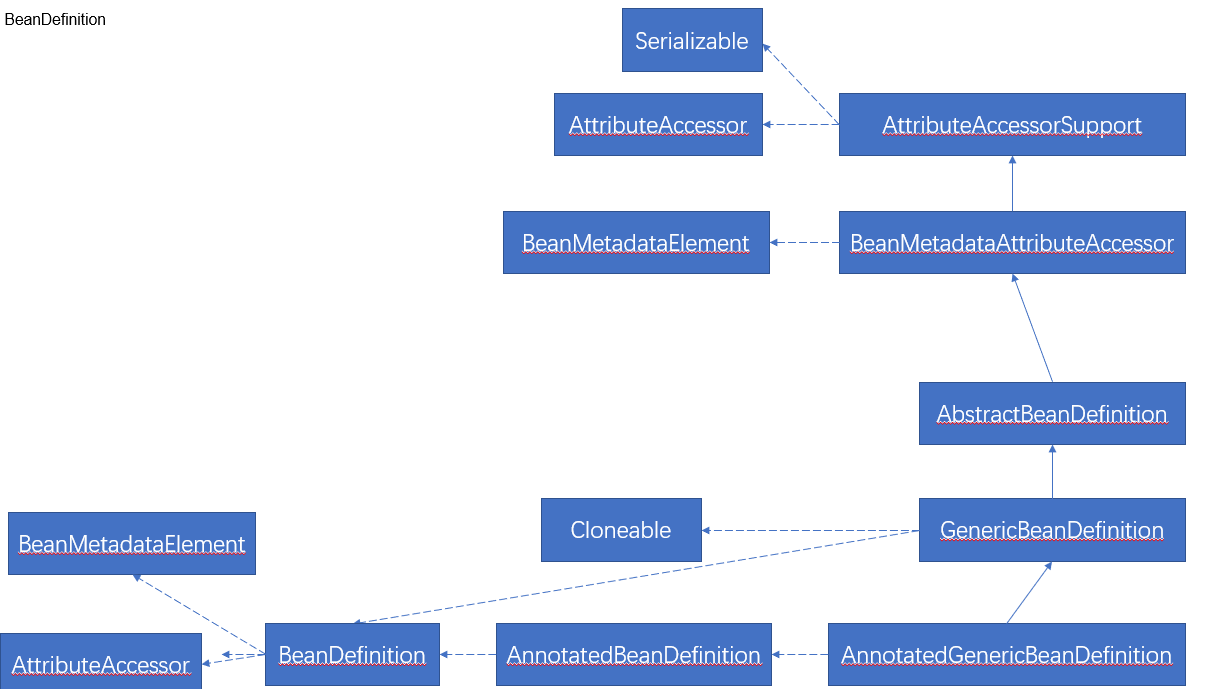
如果class上未找到相应注解，获取接口上的注解，同理获取。

再没有找到的话，获取父类，直到找到Object。

如果当前主函数类（App.java）是一个被Component注解的类，将该类添加到注册中心。

AnnotatedBeanDefinitionReader—> doRegisterBean

**new** AnnotatedGenericBeanDefinition(annotatedClass) 注解方式初始化BeanDifinition



Spring使用StandardClassMetadata保存Class信息，包括是接口，是类，实现接口类型等。

Spring使用StandardAnnotationMetadata保存注解Class信息，多一个注解信息。

使用conditional注解的函数在这里跳过

**if** (**this**.**conditionEvaluator**.shouldSkip(abd.getMetadata())) {

ScopeMetadata scopeMetadata

获取当前bean的代理模式，默认是default，可以指定

初始化BeanName

设置一些Common注解，processCommonDefinitionAnnotations

如：Lazy，DependsOn，Role，Description

qualifiers

**new** BeanDefinitionHolder(abd, beanName);

初始化BeanDefinitionHolder，真正注册的时候是它。

AnnotationConfigUtils.*applyScopedProxyMode*

设置代理模式，默认没有代理模式。

BeanDefinitionReaderUtils.*registerBeanDefinition*

注册

BeanDefinitionRegistry🡪 registerBeanDefinition

调用GenericApplicationContext.**beanFactory.**registerBeanDefinition

最终调用（真实调用）DefaultListableBeanFactory.registerBeanDefinition

很简单，查看存在不存在BeanDifinition，是否已经创建实例，如果都没有，插入到注册中心。

### 上下文准备完毕事件

prepareContext🡪listeners.contextLoaded

事件和之前的处理方式一样，EventPublishingRunListener里的广播器去广播

#### ConfigFileApplicationListener

onApplicationPreparedEvent

添加PropertySourceOrderingPostProcessor，当有新PropopertySource加入的时候，重新排序所有的PropertySource

#### LoggingApplicationListener

如果实例中不存在**springBootLoggingSystem，**添加实例，**loggingSystem**

#### BackgroundPreinitializer

不支持当前事件

#### DelegatingApplicationListener

不支持当前事件

## 刷新上下文

Run🡪 refreshContext

AbstractApplicationContext🡪refresh

调用刷新上下文函数，

### 准备刷新

refresh🡪prepareRefresh

判断环境类型（我们一般都是StandardServletEnvironment）

StandardServletEnvironment🡪 initPropertySources

初始化系统参数（所有的参数，包括命令行，环境变量，application.yml.properties等）

WebApplicationContextUtils.*initServletPropertySources*(getPropertySources(), servletContext, servletConfig);

因为现在servletContext和servletConfig都还没有生成，所以这里几乎不处理。

当使用标准war包启动（非boot方式）时，可能这里都要加进去。

sources.replace(name, **new** ServletContextPropertySource(name, servletContext));

sources.replace(name, **new** ServletConfigPropertySource(name, servletConfig));

### 获取、刷新Bean工厂

refresh🡪obtainFreshBeanFactory

基本什么都不做，就是重新设置一下ID

### 准备Bean工厂

refresh🡪 prepareBeanFactory

1、 添加BeanPostProcessor

1） ApplicationContextAwareProcessor

2） ApplicationListenerDetector

2、 添加不参与依赖注入的接口

1）EnvironmentAware

2）EmbeddedValueResolverAware

3）ResourceLoaderAware

4）ApplicationEventPublisherAware

5）MessageSourceAware

6）ApplicationContextAware

7）BeanFactoryAware

8）BeanNameAware

9）BeanClassLoaderAware

当然也可以加入我们的，不过没必要

1. 添加类型模式的Bean

BeanFactory.**class**, beanFactory)  
ResourceLoader.**class**, **this**);  
ApplicationEventPublisher.**class)**ApplicationContext.**class**, **this**)

### 调用BeanFactoryPostProcessors处理

refresh🡪invokeBeanFactoryPostProcessors

PostProcessorRegistrationDelegate.*invokeBeanFactoryPostProcessors*

顺序调用BeanDefinitionRegistryPostProcessor的postProcessBeanDefinitionRegistry函数

先处理已经加入到AbstractApplicationContext中的PostProcessor

2 = {ConfigFileApplicationListener$PropertySourceOrderingPostProcessor@4000}

1 = {SharedMetadataReaderFactoryContextInitializer$CachingMetadataReaderFactoryPostProcessor@3804}

0 = {ConfigurationWarningsApplicationContextInitializer$ConfigurationWarningsPostProcessor@3707}

* 1. 根据类型（一个实现BeanDefinitionRegistryPostProcessor接口，一个实现BeanFactoryPostProcessor接口）添加到两个List中registryProcessors，regularPostProcessors，获取BeanFactory中已经注册的实现BeanDefinitionRegistryPostProcessor接口的对象，使用PriorityOrdered接口过滤，添加到当前处理列表currentRegistryProcessors和已处理列表processedBeans中，*invokeBeanDefinitionRegistryPostProcessors*，调用postProcessor的postProcessBeanDefinitionRegistry函数。
  2. 同上，这次使用Ordered过滤，继续做以上操作。
  3. 最后没有实现Ordered和PriorityOrdered接口的PostProcessor。
  4. registryProcessors列表中的实例都实现了BeanDefinitionRegistryPostProcessor接口，肯定也实现了BeanFactoryPostProcessor。先调用registryProcessors列表中的postProcessBeanFactory函数。在调用最初初始化regularPostProcessors时加入的实例。

BeanFactoryPostProcessor处理方式和BeanDefinitionRegistryPostProcessor处理方式类似，先去BeanFactory中寻找BeanFactoryPostProcessor实例，然后按顺序执行。

默认的BeanFactoryPostProcessor和BeanDefinitionRegistryPostProcessor都没做太多事，留给扩展接口了。

总结：

处理顺序为，先添加系统默认的BeanDefinitionRegistryPostProcessor（不处理），然后处理实现PriorityOrdered接口的，可动态有新增实现Ordered接口和什么都没有的，然后处理实现Ordered接口的，最后处理没有实现序列接口的。

BeanFactoryPostProcessor类似

重点说下：

ConfigurationClassPostProcessor

1. **sourceExtractor**
2. **metadataReaderFactory**

实例化一个metadataReaderFactory，这个东西是用来生成metadataReader的

metadataReader是一个classReader，ASM编程的一部分，详细请参阅spring asm

classReader 读取class文件

classWriter 写入class

ClassVisitor，methodVistor，等等

ConfigurationClassPostProcessor🡪processConfigBeanDefinitions

1. 获取@Configuration注解的class，注意，很多注解继承了@Configuration
2. 生成ConfigurationClassParser，参数不说了，在构造函数中还构建了ComponentScanAnnotationParser对象
3. 调用parser的parse函数

这地方因为刚开始初始化，所有BeanDefinition都是RootBeanDefinition，走processConfigurationClass的Class参数实现。

parse🡪processConfigurationClass处理Configuration类中的子类。

当sourceClass不为空的时候循环执行，也就是递归向下获取。有可能某个configuration里面有configuration，可以无限向内。

doProcessConfigurationClass，根据生成一个完整的ConfigurationClass 对象。

doProcessConfigurationClass

1. 处理PropertySources注解—processPropertySource

获取encoding，factory等属性，可以自定义PropertySourceFactory，默认为org.springframework.core.io.support.DefaultPropertySourceFactory

this.environment.resolveRequiredPlaceholders(location)，路径中的参数，可以写在主配置文件中，如classpath:/${my.placeholder.default}/a.properties

使用DefaultResourceLoader加载文件，没什么特别的。初始化ResourcePropertySource，也没什么特别的。

addPropertySource，将新配置文件加入到配置文件的最低级，也就是最后

1. 扫描ConfigurationClass中的ComponentScans注解，如果有数据的话，同理，递归扫描，继续调用componentScanParser，记住，最大的Componentscan就是App.java下的，他也一定会扫到。
2. processImports

处理所有的imports，可以引用一个class，注册进来

系统默认的比较重要的import有：

1. ImportSelector

普通import，获取后，立即执行，

DeferredImportSelector，获取后，存储，将来在configuration全部初始化后，再执行。

1. ImportBeanDefinitionRegistrar

非常重要的一个接口类，大多数插件都是用这种方式来注册自己的Bean

1. 普通的Bean
2. importResource，引入XML文件
3. retrieveBeanMethodMetadata

获取beanMethod，就是我们@Bean注解的函数

1. processDeferredImportSelectors

spring将importSelector分成defer和普通两种，普通import将会立即将资源（class定义）导入，defer将会在所有的configuration执行，这样做的做的好处是，我们的defer实现，可以使用conditional，不然的话，如果客户已定义了一部分东西，和系统默认import冲突很尴尬。

最终调用MergedAnnotationAttributesProcessor的postProcess函数

总结：

一切都始于扫描，处理@Configuration注解的class，同一个class里，理论上可以有无限层这个注解。是用parser解析class，因为有无限层次，所以这里使用了大量的递归函数。基础处理函数doProcessConfigurationClass将先处理PropertySource注解。之后处理ComponentScan注解，然后是import注解，注意import注解里有个特殊规则，实现了DeferredImportSelector接口的import，暂时不做处理，在configuration都处理完执行，普通importselector立即执行，调用selectImports函数，获取需要import的classes，之后立即import当前classes，当然，里面也可以很复杂的实现许多其他的注解，然后递归到最终的元注解。系统默认（是用SpringBootApplication注解）了两个import，一个引入了AutoConfigurationImportSelector，所有的自动配置，都是用该类引入。AutoConfigurationPackages$Registrar，同上。

是用AutoConfigurationImportSelector的selectImports函数获取自动配置信息。是用AutoConfigurationMetadataLoader类的loadMetadata函数从**META-INF/spring-autoconfigure-metadata.properties**文件里获取自动配置信息

使用默认filter，AutoConfigurationImportFilter获取可加载对象。默认只有一个，我们可以自己添加（使用SpringFactoriesLoader.loadFactories加载的，详细添加filter的方式，参考最初），filter过滤的方式是当前class的ConditionalOnClass属性。Filter过滤之后，触发自动加载完成事件，默认情况下，没什么好做的fireAutoConfigurationImportEvents，调用onAutoConfigurationImportEvent函数。默认不做太多事。只是，这里可以做。

最终，返回一大堆配置类，都是最终的import，不会再有deferred，有也不会再执行了，然后挨个processImports

### 注册BeanPostProcessor

refresh🡪registerBeanPostProcessors

没什么好说的，就是注册而已

### 注册特殊类—比如WebServer

refresh🡪onRefresh

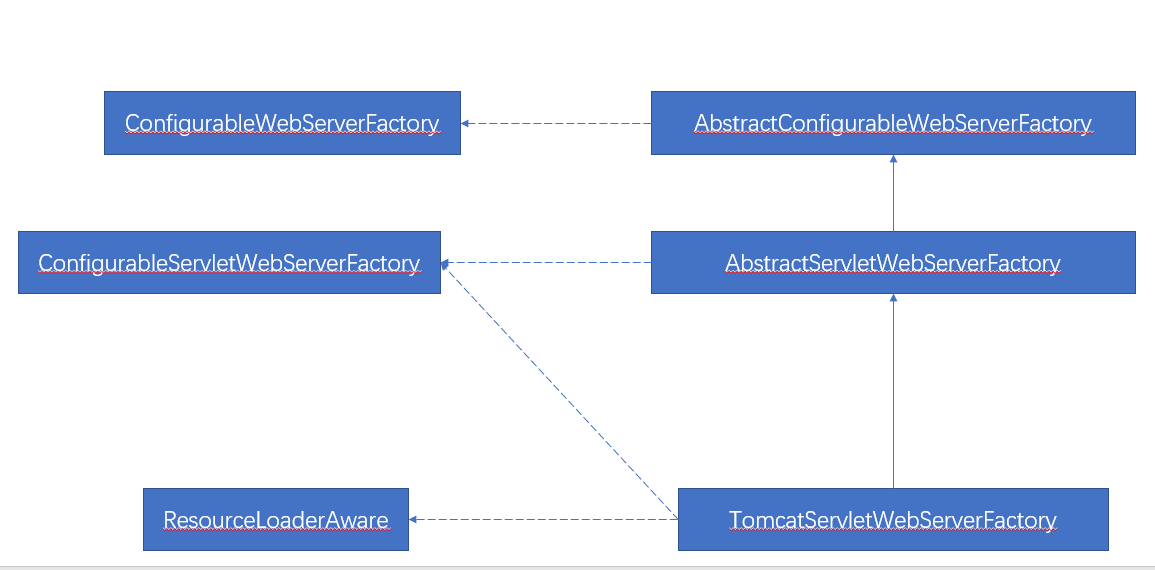
#### 获取web上下文—servletContext

onRefresh🡪getServletContext

初始状态下webserver和servlet都为空

获取webserverFactory（getWebServerFactory），在

#### 创建WebServer



TomcatServletWebServerFactory示意图

ConfigurableWebServerFactory，定义了Tomcat的共有定义。如：setHttp2，setSsl，setSslStoreProvider

ServletWebServerFactory，定义Web工厂接口函数。

我们可以自己实现这个接口，设定一些东西，完成一些事情，只要定义一个bean，继承自这个类就可以。

getWebServer🡪 prepareContext

初始化TomcatEmbeddedContext，设置tomcatContext启动参数，详细启动，请参照springBoot EmbeddedServer章节。

### 实例化Bean

run🡪finishBeanFactoryInitialization，只是按定义，初始化，并填入参数。过程比较复杂，有兴趣可以看看。

### 初始化完成

run🡪 finishRefresh

1. clearResourceCaches

删除资源缓存

1. initLifecycleProcessor

bean的声明环节准备开始。

1. 开启bean
2. publishEvent

启动完成事件。

怀疑部分定时器的东西，这里会启动，没证实。