

**SpringBoot**

# 初始化顺序

## ApplicationStartingEvent

## ApplicationEnvironmentPreparedEvent

## ApplicationContextInitializer

## ApplicationPreparedEvent

# 初始化

## 初始化器

初始化事件

初始化时，使用classLoader扫描jar中或者sourceFolder中的**META-INF/spring.factories**

*# Application Context Initializers***org.springframework.context.ApplicationContextInitializer**=**\  
org.springframework.boot.context.ConfigurationWarningsApplicationContextInitializer,\  
org.springframework.boot.context.ContextIdApplicationContextInitializer,\  
org.springframework.boot.context.config.DelegatingApplicationContextInitializer,\  
org.springframework.boot.web.context.ServerPortInfoApplicationContextInitializer**

*\* <p>Typically used within web applications that require some programmatic initialization  
\* of the application context. For example, registering property sources or activating  
\* profiles against the {****@linkplain*** *ConfigurableApplicationContext#getEnvironment()  
\* context's environment}. See {****@code*** *ContextLoader} and {****@code*** *FrameworkServlet} support  
\* for declaring a "contextInitializerClasses" context-param and init-param, respectively*

主要用来实现一些编程式的初始化事件。可以用来注册属性，激活profile等等

ApplicationContextInitializer

在prepareContext中调用，在startingEvent、environmentPrepared事件之后，在contextPrepared事件之前触发。

## 初始化事件SpringApplicationRunListener

Spring定义了一整套事件规则。

继承自Java的EventObject🡪ApplicationEvent🡪SpringApplicationEvent🡪…

可以自己设置RunListener，传播自己定义的事件，在事件中处理特殊事件。

一般情况下，springboot不推荐使用这种方式初始化，一般自己定义一个二级listener就可以了

1. ApplicationStartingEvent

启动事件，第一个事件

1. 获取Runlistener之后，调用starting事件，第一个事件开始。
2. 准备好environment之后，准备上下文prepareContext

在上下文准备的过程中调用初始化事件。

初始化事件，使用SimpleApplicationEventMulticaster，传播事件，到下一级listener中

### ApplicationStartingEvent

启动开始事件，一般不做什么，也没什么要做的

### environmentPrepared

环境准备完毕事件。所有配置文件已加载完成。

初始化的时候先加载一部分配置文件，如：

1. servletConfigInitParams
2. servletContextInitParams
3. systemProperties
4. systemEnvironment

其他的配置文件application.yml.xml.properties等，是在ConfigFileApplicationListener中读取的配置。

springcloud就使用**BootstrapApplicationListener**获取bootstrapContext，读取bootstrap.yml配置文件**spring.cloud.config**获取springcloud配置中心等配置信息下面几个，主要涉及配置文件的bean是**PropertySourceBootstrapConfiguration** ，继承自ApplicationContextInitializer<ConfigurableApplicationContext>，将注册的BeanDefinition（PropertySourceBootstrapConfiguration和EnvironmentDecryptApplicationInitializer）注入到application中去addInitializers，在下一次（二级）preparedContext事件的时候，PropertySourceBootstrapConfiguration会刷新environment。

不管是springcloud配置中心，apollo，nacos等，所有的，都是在这个事件中读取的。Spring会合并远端配置文件，到本地。

**注意**，springcloud有个小bug，如果使用配置文件作为logging的载体，如logback.xml等，当触发springcloud获取远端配置文件时，会先启动日志配置，造成logback.xml中读取不到application.yml中的配置，需要在bootstrap.yml中配置。

**org.springframework.cloud.bootstrap.BootstrapConfiguration**=**\  
org.springframework.cloud.bootstrap.config.PropertySourceBootstrapConfiguration,\  
org.springframework.cloud.bootstrap.encrypt.EncryptionBootstrapConfiguration,\  
org.springframework.cloud.autoconfigure.ConfigurationPropertiesRebinderAutoConfiguration,\  
org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration**

sources：ApplicationContext启动的起点时会配置源，springcloud在启动的时候，使用上述配置，当然，如果使用其他的，比如rebbion等，也会配置这些。

Springboot的一切都是以某个点为源头，读取配置也好，等等的。

Springcloud的parentContext配置了以上这些Bean，不会污染我们的起点。使用context上下文切换的方式来隔离。

顺便：

如nacos获取配置信息的方法，

配置一个NacosConfigBootstrapConfiguration，注册到spring.factories中去org.springframework.cloud.bootstrap.BootstrapConfiguration，在BootstrapApplicationListener启动的时候，把注册信息注入到parentContext中，注册一个NacosConfigProperties（配置文件，读取bootstrap.yml中的），重点是注册了一个NacosPropertySourceLocator🡪PropertySourceLocator，获取配置文件。当然，我们也可以自己实现一个从远端获取配置文件的方式，从文件，从redis，数据库，都可以。

### contextPrepared

默认情况下，啥也不干，当然我们自己也可以做些事情。

准备好之后，开始初始化（注册我们的bean等）。

### contextLoaded

在初始化完成之后，注册applicationContext到listener中。做的事情也不太多，主要是注册了logging的bean。将context注册到RestartListener中。ConfigfileApplicationListener中将default配置文件放到最后。

### started

基本上什么都不干，我们自己可以实现一些东西，不过很少

系统启动后，调用

ApplicationRunner

CommandLineRunner

比如定时器启动啊，什么的，都在这里启动。一般也不需要处理。

### running

一般也什么都不做，没有需要做的事情。启动结束。

### failed

## BeanFactory

### postProcessBeanFactory

修改当前BeanFactory，不过一般很少自己设定BeanFactory，只是spring内部使用。

在ServletWebServerApplicationContext中，使用该方法添加BeanPostProcessor，在BeanPostProcessor事件中将servletContext和servletConfig到bean（ServletContextAware）中

如果想改的话，也自己实现一个

参照：MyAnnotationConfigServletWebServerApplicationContext

### BeanDefinitionRegistryPostProcessor

在标准的BeanFactoryPostProcessor执行之前，先执行这个。同样，可以再下面这些地方添加。

调用postProcessBeanDefinitionRegistry

**在标准的springboot启动时，因为事件错乱了，导致事件和想象的规则不一样。以下特殊说明**

**标准的（EventPublishingRunListener）没有使用contextPrepared事件。**

**在starting事件中处理ApplicationStartingEvent**

**在environmentPrepared事件中处理ApplicationEnvironmentPreparedEvent**

**忽略contextPrepared事件**

**在contextLoaded事件中处理ApplicationPreparedEvent。**

**在started事件中处理ApplicationStartedEvent**

**在running事件中处理ApplicationReadyEvent**

在springboot标准实现中（如果不使用自定义全套事件、RunListener），只能使用ContextLoaded（ApplicationPreparedEvent）或者ApplicationContextInitlizer<ConfigurableApplicationContext>

### BeanFactoryPostProcessor

初始化BeanFactoryPostProcessor，在context生成之后，可以添加BeanFactoryPostProcessor到context中。

在以下地方ApplicationContextInitializer<ConfigurableApplicationContext>、ContextPrepared、ContextLoaded中添加BeanFactoryPostProcessor。

在ContextPrepared、ContextLoaded中使用事件传播的方式在listener中添加。

主要是ConfigFileApplicationListener中添加PropertySourceOrderingPostProcessor。

PropertySourceOrderingPostProcessor，主要是将default配置文件更改到最后。

先执行用户自定义的BeanFactoryPostProcessor，然后再执行在beanFactory中注册的bean，注册到其中的bean会根据Priority判定执行顺序。在自己定义的BeanFactoryPostProcessor中可以再定义BeanFactoryPostProcessor到registry中，会和默认实现的ConfigurationClassPostProcessor一起调用，Config的bean，最好使用static关键字。

#### 默认实现

1. ConfigurationClassPostProcessor

org.springframework.context.annotation.ConfigurationClassPostProcessor

在postProcessBeanDefinitionRegistry中读取源Class，挨个配置所有的Configuration。该类的优先级是最低的，所以永远在最后执行，在ConfigurationClassPostProcessor处理过程中，也可以再生成BeanFactoryPostProcessor，然后继续执行一次。这次之后就不会再有BeanFactoryPostProcessor的执行了。

1. AutowiredAnnotationBeanPostProcessor

org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor

1. RequiredAnnotationBeanPostProcessor

org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor

1. CommonAnnotationBeanPostProcessor

org.springframework.context.annotation.CommonAnnotationBeanPostProcessor

1. EventListenerMethodProcessor

org.springframework.context.event.EventListenerMethodProcessor

1. DefaultEventListenerFactory

org.springframework.context.event.DefaultEventListenerFactory

这个不是PostProcessor

BeanDefinitionRegistryPostProcessor

参数：Registry，可以注册BeanDefinition到注册中心。

BeanFactoryPostProcessor

参数：BeanFactory，可以查询或者更改BeanDefinition

postProcessBeanDefinitionRegistry

执行顺序：

1. 用户自定义的BeanDefinitionRegistryPostProcessor
2. 系统默认配置的ConfigurationClassPostProcessor，还有1中添加到BeanFactory中的BeanDefinitionRegistryPostProcessor，先执行实现了PriorityOrdered接口的。
3. 用户在Configuration中配置的和实现了Ordered接口的BeanDefinitionRegistryPostProcessor
4. 所有在BeanFactory中注册的BeanDefinitionRegistryPostProcessor

总结：

在用户自定义中（事件中）可以添加PriorityOrdered，Ordered的BeanDefinitionRegistryPostProcessor

先执行PriorityOrdered，在执行完ConfigurationClassPostProcessor之后，Configuration注解被解析。用户的Configuration中可以配置Ordered或无序BeanDefinitionRegistryPostProcessor

再执行Ordered，Ordered中可以添加无序的BeanDefinitionRegistryPostProcessor

最后执行无序的BeanDefinitionRegistryPostProcessor。

postProcessBeanFactory

按照上面的执行顺序，再执行一把这个。在添加BeanDefinitionRegistryPostProcessor的时候，同理，也可以添加BeanFactoryPostProcessor。

上述先处理applicationContext中添加的BeanFactoryPostProcessor（使用addBeanFactoryPostProcessor），后面执行添加到BeanFactory中的（使用registry注册的BeanDefinition）

在ConfigurationClassPostProcessor执行的时候，读取了很多很多配置，包括springboot的自动配置一系列的东西，有时间补充，其中使用了很多BeanFactoryPostProcessor。

在执行以上配置型BeanFactoryPostProcessor的时候，顺序和前面的BeanDefinitionRegistryPostProcessor一样，先执行PriorityOrdered，再执行Ordered，再执行无序。

主要的配置，所有的AutoConfiguration全在上述执行的。

## InitBeanDefinition

使用ConfigurationClassPostProcessor初始化SpringBean容器

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。

1. 基础处理函数doProcessConfigurationClass将先处理PropertySource注解。之后处理ComponentScan注解
2. 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
3. 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。
4. 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。使用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

1. MemberClass

当前类的成员Class

1. PropertySources

当前类是否配置了自定义的配置文件

1. ComponentScans

Class中是否含有该配置，如果有的话，递归继续扫描。

1. ImportSelector

立即执行

1. Add ImportResource

将当前class中import的xml配置文件添加到class中，但并没有解析，后面才会解析

1. Add BeanMethod

Config中的Bean，添加进去（使用@Bean注解的）

1. Add Interface

Java8使用的lambada表达式，没有实现，在这里加入。

1. DeferredImportSelector

AutoConfigurationImportSelector，系统auto的主要selector，是个deferred

所有的配置，循环上面，再来一次。加载所有的BeanDefinition，包含所有的渠道

将所有的配置AutoConfiguration都加载进去之后，再次二次加载，Configuration中可能使用的XML配置和Registrar配置，现在都是使用ImportBeanDefinitionRegistrar。3.4是循环调用的，所有的AutoConfiguration都会判定一次。

1. ImportResources

Import XML的文件配置

1. ImportBeanDefinitionRegistrar读取配置

ImportBeanDefinitionRegistrar可以获取到当前class中的注解配置。

当我们要自定义注解，同时根据注解，来注册BeanDefinition的时候，我们可以使用ImportBeanDefinitionRegistrar。

Springboot自己的很多功能都是使用该接口实现。

比如：代理Proxy，Mongo，DataSource等等的配置都是如此。

在解析完ConfigurationClass之后，ConfigurationClassBeanDefinitionReader将会根据ConfigurationClass去注册ConfigurationClass中的Bean（loadBeanDefinitions）

**TrackedConditionEvaluator（有兴趣继续看）**

**注册顺序，依次往下**

1. registerBeanDefinitionForImportedConfigurationClass

先注册使用Import注解的Class

1. loadBeanDefinitionsForBeanMethod

Configuration中的@Bean注解

1. loadBeanDefinitionsFromImportedResources

XML文件配置信息

1. loadBeanDefinitionsFromRegistrars

ImportBeanDefinitionRegistrar接口实现。

不同于BeanDefinitionRegistryPostProcessor，这个接口实现是最后一次添加BeanDefinition，所有的AutoConfiguration大概都在这里做。因为ConfigurationClassPostProcessor永远是最后一个执行的BeanDefinitionRegistryPostProcessor，所以我们基本很难在后面改动spring自动配置的AutoConfiguration，不过一般情况下，我们配置了，spring配置都会跳过。这里是最后一个执行的，ImportBeanDefinitionRegistrar又是在注册Bean的最后一步，使用这种方式写组件，也能保证我们不会改spring自己的AutoConfiguration，保持稳定性。当然，如果需要的话，我们也可以自己实现一个ImportBeanDefinitionRegistrar，然后用里面的Registry去删除或修改之前的配置。

## 添加BeanPostProcessor

按照规则（如下）添加BeanPostProcessor到BeanFactory中

先执行PriorityOrdered实现，再执行Ordered实现，最后Non实现。最后执行MergedBeanDefinitionPostProcessor实现。和BeanFactoryPostProcessor不同的是，BeanPostProcessor在调用过程中是不能再次添加BeanPostProcessor。

## ApplicationEventMulticaster

Spring事件传播，和初始化的事件一样，我们也随时可以自己添加事件到其中。

## LifeCycle

Spring管理的Bean，有个生命周期管理，在服务器启动后（WebServer），启动Bean

lifecycleProcessor，LifeCycleProcessor的名字，注册到BeanFactory中，如果有的话，会使用你设置的。一般情况下，没必要自己搞，默认的会使用DefaultLifeCycleProcessor，这个就没问题。

系统默认的LifeCycleProcessor，默认不会启动LifeCycle，如果想要自动启动该Bean，使用SmartLifeCycle，并且，设置isAutoStartup=true。

可以使用LifeCycle启动一些东西。比如EndPoint什么的。

待填充

## 初始化Bean

在WebServer启动之前，已经初始化了很多Bean。这里不赘述，在讨论WebServer的章节。

preInstantiateSingletons

1. 当使用@DependOn注解时，会先获取其中的Bean

### InstantiationAwareBeanPostProcessor

#### postProcessBeforeInstantiation

初始化Bean前的操作。

在这里，我们可以自己去实现一个Bean，替代spring帮忙初始化。

如果我们自己初始化的话，自动调用BeanPostProcessor的postProcessAfterInitialization。

### Supplier

同样，我们也可以使用Supplier去自定义一个bean

### MergedBeanDefinitionPostProcessor

系统内部PostProcessor

比如：

1. CommonAnnotationBeanPostProcessor

初始化initMethods和destroyMethods

1. AutowiredAnnotationBeanPostProcessor

autoWired注解在这里加入InjectionMetadata，这个InjectionMetadata后面会讲到，同样，我们也可以使用自己拼装的InjectionMetadata

### InstantiationAwareBeanPostProcessor

CommonAnnotationBeanPostProcessor

这是个非常重要的class，里面的postProcessPropertyValues是几乎所有的InjectionMetadata处理的地方，比如，我们可以自己定义一些属性（使用我们自己定义的注解生成的bean），在这里注入到spring管理的bean中。

#### postProcessAfterInstantiation

实例化后的处理函数，我们可以自己处理一些东西，系统默认的什么都没干，这里应该是预留的

#### postProcessProperties

在这个函数中可以处理一些数据变更，比如，我们注入到Bean中的userName等于123，我们可以根据某些规则，在这里改成456。Autowire都是在这里执行的。

在这个之后，可以使用typeConverter更改一些东西，不过很少有人用

**Bean实例化完毕，进入初始化阶段**

先使用编程方式，将BeanClassLoaderAware，BeanNameAware，BeanFactoryAware注入到实例中

#### applyPropertyValues

BeanDefinition中的属性，不管是基本变量还是引用变量，都可以在这里添加

BeanDefinitionValueResolver🡪resolveValueIfNecessary

大部分情况下都是用基本单位的值和引用

BeanDefinitionBuilder.addXXXXX

Builder中客户自定义处理还可以用applyCustomizers

1. RuntimeBeanReference
2. RuntimeBeanNameReference
3. BeanDefinitionHolder
4. BeanDefinition
5. ManagedArray
6. ManagedList
7. ManagedSet
8. ManagedProperties
9. TypedStringValue

上面用的最多的是基本单位和引用1，其他的在条件不允许，比如说还没有对象，只有BD的地方等等，才需要使用，默认使用BDBuilder创建BD的时候，只能添加基本单位和引用1

如果需要添加其他的引用的话，可是用applyCustomizers，

b.getBeanDefinition().getPropertyValues().addPropertyValue()

#### postProcessPropertyValues

同上。新版本中该类不建议再使用，建议换成postProcessProperties

### BeanPostProcessor

#### postProcessBeforeInitialization

1. ApplicationContextAwareProcessor

默认的，主要是用来插入各种Aware

1. EnvironmentAware

环境参数

1. EmbeddedValueResolverAware

valueResourcelver

1. ResourceLoaderAware

resourceLoader，各种文件（class文件、properties文件等等）

1. ApplicationEventPublisherAware

发布事件，比如可以发布一个事件，给其他listener监听，我们可以自己定义一套事件、listener，留给他们处理，等等，不过一般很少用，毕竟这里的bean一般是单例的，我们一般直接调用函数就行了，没必要弄成可配置的listener去处理业务。如果写某些插件的时候，可以这么使用。

1. MessageSourceAware

消息式的，后面有介绍

1. ApplicationContextAware

不解释。

1. ServletContextAwareProcessor

设置servletContext还有servletConfig。

1. ConfigurationClassPostProcessor🡪ImportAwareBeanPostProcessor

将注解引入到Configuration中去，可以参考示例中的BeanPostProcessorConfiguration，可以不加入到扫描包中，也可以不加入，不会影响使用。

1. ConfigurationPropertiesBindingPostProcessor

将Properties绑定到Bean中，没什么好讲的，关于binder，后面有机会再聊，这个是比较大的点，springboot改了版本之后，又改过。

1. WebServerFactoryCustomizerBeanPostProcessor

Webserver的初始化处理。调用WebServerFactoryCustomizer，客户自定义的初始化处理。

1. ErrorPageRegistrarBeanPostProcessor

Webserver初始化errorPage。用处不大。

1. InitDestroyAnnotationBeanPostProcessor

添加处理@PostContruct和@PreDestroy，将注解的函数添加到BeanDefinition中。

#### init-method

1. InitializingBean
2. Init-method注解

#### postProcessAfterInitialization

1. AbstractAdvisingBeanPostProcessor

Proxy相关的处理，对于代理Bean来说，这个比较重要，后面重点讲。

其他的基本都没怎么处理，我们也可以自己写个，然后处理一下初始化信息什么的，不过有很多方式，也就没人在这里初始化了。

### SmartInstantiationAwareBeanPostProcessor

#### predictBeanType

这个和顺序无关，它可能在任何时候执行。

判定该Bean的Class，比如某个Bean明明是AClass，在这里我们可以改成BClass，然后在根据Class类型获取Bean的时候，可以把AClass当成BClass返回去（不过需要注意的是，这个改了，注入的时候可是会报错，而且也调用不了，所以一般，我们不用动这个）。主要是在Proxy模块大量使用了，我们可以更改一个Bean的类型为另一个，不过一般都不用改，除了Proxy。

#### determineCandidateConstructors

优先级最高的是BeanDefinition中已经明确指定的构造函数，当存在多个构造函数的时候，使用postProcessor去判定当前Class使用哪一个构造函数。一般我们也很少用。

如果也没有获取到构造函数。会使用无参构造函数。

系统默认用的最多的就是这个AutowiredAnnotationBeanPostProcessor

如果有构造函数，会自动去获取构造函数的参数，注入。

Lookup

#### getEarlyBeanReference

处理循环引用的时候，特殊处理，基本也都是内部使用的，强烈不推荐研发使用，同样也是Proxy用的，后续有详细介绍。

## InstantiationAwareBeanPostProcessorAdapter

一般所有的处理BeanPostProcessor就继承这个就可以了，这个全都继承了。

# AutoConfiguration

后续有时间添加，重要的就是Proxy等几个spring组件。

## 公共组件

### EnableConfigurationProperties

1. EnableConfigurationPropertiesImportSelector

ImportSelector在processImport时调用，将ConfigurationPropertiesBindingPostProcessorRegistrar和ConfigurationPropertiesBeanRegistrar写入加入到带解析队列。

1. ConfigurationPropertiesBindingPostProcessorRegistrar

添加两个BeadDefinition

1. ConfigurationBeanFactoryMetadata
2. ConfigurationPropertiesBindingPostProcessor
3. ConfigurationPropertiesBeanRegistrar

继承自ImportBeanDefinitionRegistrar，在processImport的时候，先判定ImportSelector，然后再添加ImportBeanDefinitionRegistrar，所以我们很多个组件类的东西，都是先使用ImportSelector注册进去，然后再调用ImportBeanDefinitionRegistrar，有关ImportBeanDefinitionRegistrar的问题，可以参照之前的。

这个Registrar的逻辑还是很简单的，读取EnableConfigurationProperties注解中的Class

读取当前Class的ConfigurationProperties配置，如果没有的话，prefix直接使用空，然后将这个配置Bean加入到Registry中。

## ServletWebServerFactoryAutoConfiguration

很简单，什么都没做，就是判定了一下使用什么Server，默认是使用Tomcat的

## WebMvcAutoConfiguration

后面再说

## AopAutoConfiguration

参照spring aop

## JDBC相关AutoConfiguration

下面是Springboot定义的jdbc相关自定义配置

### DataSourceAutoConfiguration

Import

1. DataSourcePoolMetadataProvidersConfiguration

连接池的wrapper，分别为：HikariDataSource，Dbcp2PoolDataSource，tomcatDataSource，springboot2.0之后，默认使用了hikariDataSource，不过这里使用了druid的连接池。Hikari号称是java界最好的连接池。有机会可以试试。

1. DataSourceInitializationConfiguration
2. DataSourceInitializerInvoker

初始化类，在afterPropertiesSet时，初始化schema-\*.sql，在DataSourceSchemaCreatedEvent事件中，初始化data-\*.sql

DataSourceInitializer

使用**spring.datasource.schema，**使用这个配置schema的sql

使用**spring.datasource.data**，使用这个配置data的sql

没有太大用，一般不在这里初始化。

1. DataSourceInitializationConfiguration.Registrar

将DataSourceInitializerPostProcessor，加入到Registry中。

也没干什么，就是判定一下DataSource的bean，强制初始化DataSourceInitializerInvoker

EmbeddedDatabaseCondition

内部实现的DataSource，一般不用

PooledDataSourceCondition

连接池配置，默认使用Hikri，当然，我们也可以自己配成其他的。

数据库的配置，到这里就结束了，也没啥。

### JdbcTemplateAutoConfiguration

没什么可说的，就是配置JdbcTemplate，还配置了NamedParameterJdbcTemplate，这个可以添加一些参数处理，本身来讲，意义不是特别大。

### JdbcRepositoriesAutoConfiguration

1. JdbcRepositoriesAutoConfigureRegistrar

继承AbstractRepositoryConfigurationSourceSupport，该类的注释是：

Base {@link ImportBeanDefinitionRegistrar} used to auto-configure Spring Data Repositories，也就是说，所有的springdata的自动配置都从这里开始。

初始化RepositoryConfigurationDelegate，并调用registerRepositoriesIn，构造函数需要三个参数，特别的参数是：RepositoryConfigurationSource，

这里使用注解实现AnnotationRepositoryConfigurationSource，构造函数需要五个构造函数，特别的构造函数是：AnnotationMetadata，Class<? extends Annotation> annotation。

初始化AnnotationMetadata，这里使用的是StandardAnnotationMetadata，当我们使用一个要创造一个AnnotationMetadata的时候，只需要将当前类的Class传入构造函数即可。

Annotation参数是：EnableJdbcRepositories。

这种实现的方式我们也可以用在我们自己的实现中去。

当我们可以根据不同的注解去做不同的事情的时候，我们可以创建一个abstract类，在abstract类中去获取实现类中的成员类（该类被目标注解@住），并告知处理函数，我们需要处理的注解类型。这里的目标注解类并没有参数，实际上，我们自己实现的可以有不同的参数。

这是一种编写代码的方式，解决一定的多态问题。

后面更多的Repositories能看到更多的实现，更好理解。

注解@ EnableJdbcRepositories

Import（JdbcRepositoriesRegistrar）

1. JdbcRepositoriesRegistrar

这里不会做什么，前面已经调用过一次了。

调用registerRepositoriesIn的时候，需要特别的参数RepositoryConfigurationExtension

这个参数主要用来解析扫描Repository

这个接口有三个标准实现，主要是Jdbc和Jpa继承RepositoryConfigurationExtensionSupport

完成了扫描工作。

getRepositoryConfigurations()函数完成扫描工作。

使用前面初始化的AnnotationRepositoryConfigurationSource的getCandidates获取所有扫描到的bean类，真实的扫描在getCandidates中进行，有独立的scanner（RepositoryComponentProvider）。

注意，默认的时候，scanner是不会扫描内部接口的。

默认的时候，Spring添加includeFilters：

Repository接口实现（不需要注解，直接实现）

RepositoryDefinition注解实现（注解在接口上）

excludeFilters：NoRepositoryBean注解的bean。

扫描结束之后，开始构建BeanDefinition，构建完成之后，和BeanPostProcessor类似，我们也可以使用自己的Extension，然后实现extension.postProcess。

该BeanDefinition使用的factoryBean的方式实现，

factoryClass使用getRepositoryFactoryBeanClassName获取。这里使用的是JdbcRepositoryFactoryBean

这里继承了TransactionalRepositoryFactoryBeanSupport，主要用于实现事务控制。

因为JdbcRepositoryFactoryBean是注册在Registry中的，实现了BeanFactoryAware，在调用setBeanFactory的时候，初始化了TransactionalRepositoryProxyPostProcessor，

PersistenceExceptionTranslationRepositoryProxyPostProcessor。

注意，上面的PostProcessor并不是spring标准的postProcessor，是springdata实现的

RepositoryProxyPostProcessor，只有一个函数postprocess，在其中添加事务拦截器，在做proxy实例的时候，将该拦截器带入其中（对于aop来说，一个拦截器，只是一个advice而已）。

下面的postprocessor只是添加一个PersistenceExceptionTranslationInterceptor，用来处理异常事件。

PersistenceExceptionTranslator，主要是该类实现，去拦截相应的Exception，也就是，理论上，我们如果使用自定义的Exception的时候，可以特殊处理一下，比如，我们可以在回滚的同时，做些别的事情。比如，当我们加入到一个分布式事务中时，我们可以使用这个，特殊处理，回调主事务，告诉他，我们已经失败，请求回滚。

在createRepositoryFactory的时候，将上述postProcessor加入到Factory实例中。

获取相应的proxyTarget

添加SurroundingTransactionDetectorMethodInterceptor

DefaultMethodInvokingMethodInterceptor

主要是用于获取需要调用的函数。如果是static，abstract等等，需要去lookup，不过没太大需要。

QueryExecutorMethodInterceptor

jdbcRepository将SQL保存起来，实例化成一个RepositoryQuery实例（JdbcRepositoryQuery），在这里统一处理。注意，这里虽然叫query，其实Modifying也在这里处理。也可以update

ImplementationMethodExecutionInterceptor

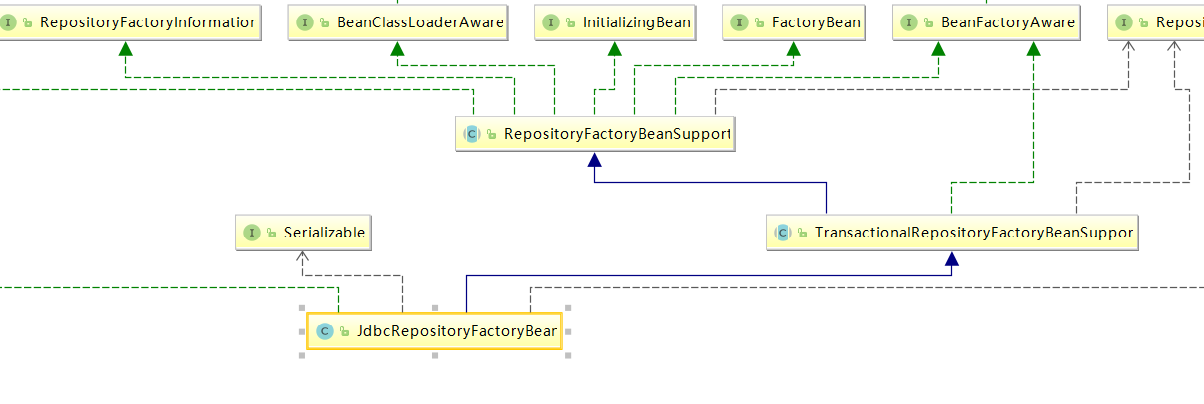
RepositoryComposition，应该是定义的其他查询。

到这里，获取BeanDefinition就结束了，下面主要介绍下FactoryBean实例化Bean，当然这里还是使用这个例子，不出意外的ibatis也是这么实现的，后面详细介绍。

JdbcRepository的proxy和例子中的实现时不一样的，例子中的是直接调用某一个class的某一个函数。而JdbcRepository使用的是拦截器，在相应的环节，使用拦截器，拦截处理，并没有真正的使用某一个target去调用什么method。实际上默认指定的target，只有一些默认实现，比如save，findById，findAll等等。一般情况下，我们的sql在拦截器阶段就直接拦截了，当然，如果使用这些函数名命名我们自己的Repository接口的话，不需要写sql，他可以执行，不过意义不大。

**FactoryBean**

这里使用的是JdbcRepositoryFactoryBean



主要是继承自TransactionalRepositoryFactoryBeanSupport，再继承自RepositoryFactoryBeanSupport

当我们要使用FactoryBean做Bean的时候，将该类型的Bean指向FactoryBean接口实现即可。

例如：

我们可以去解析某种类型的自定义注解，然后使用FactoryBean实现去获取某种Proxy，Proxy去调用我们想要的函数，等等。

### JpaRepositoriesAutoConfiguration

整体思路和JdbcRepositoriesAutoConfiguration类似。AbstractRepositoryConfigurationSourceSupport

同样继承。使用了Hibernate一系列配置，不看了，没太大用。

### JndiDataSourceAutoConfiguration

使用JndiDataSourceLookup获取DataSource，现在基本没人用了。

### XADataSourceAutoConfiguration

现在几乎不会用XA了，本质上就是多加一个配置而已。

### DataSourceTransactionManagerAutoConfiguration

只是造了一个DataSourceTransactionManager，主要不在这里，在DataSourceTransactionManager

AbstractPlatformTransactionManager。

## Transaction相关