

**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。

## EmbeddedWebServerFactoryCustomizerAutoConfiguration

TomcatWebServerFactoryCustomizer

JettyWebServerFactoryCustomizer

UndertowWebServerFactoryCustomizer

NettyWebServerFactoryCustomizer

其他三个用的不多，主要用的是Tomcat

这里举例使用Jetty

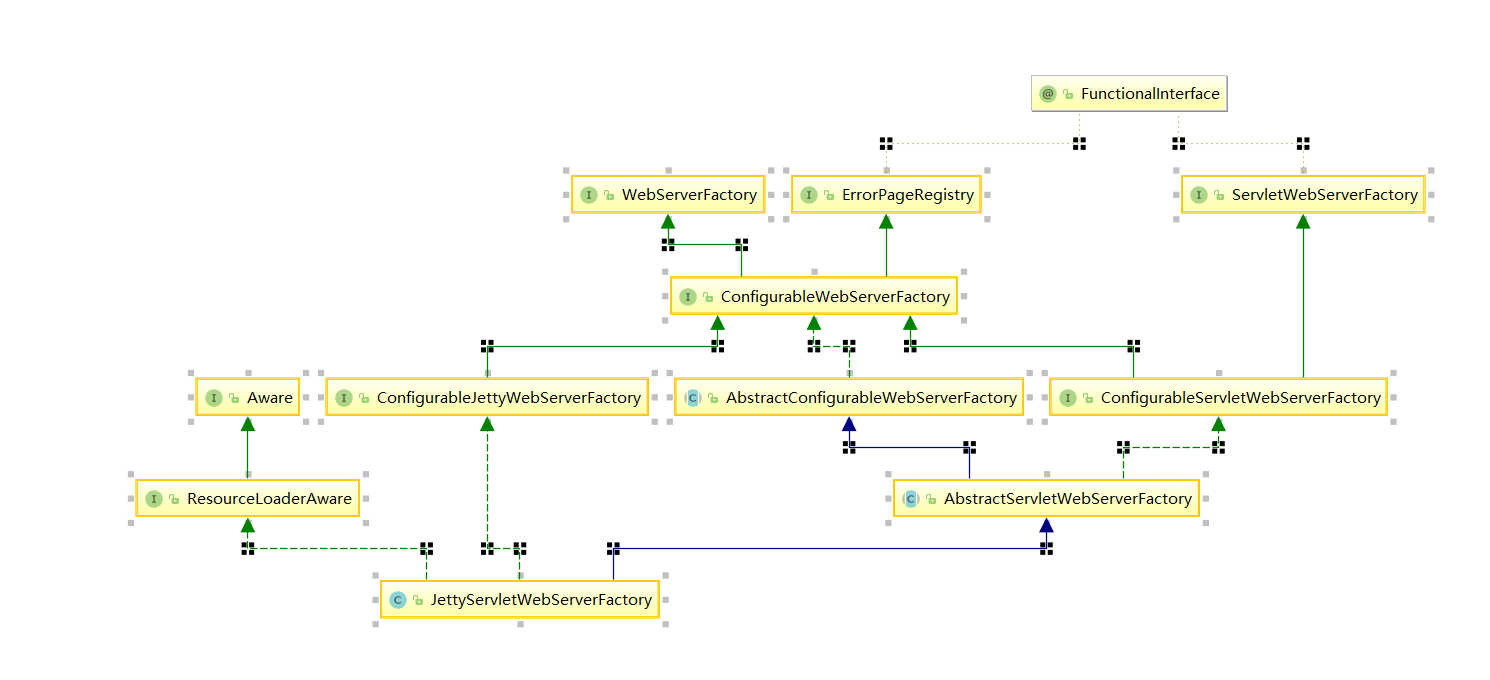
## ServletWebServerFactoryAutoConfiguration

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

注册WebServerFactoryCustomizerBeanPostProcessor到Registry中，这个PostProcessor主要是在postProcessBeforeInitialization时（已经实例化当前类，但是还没有初始化）），判定当前类是不是WebServerFactory，获取BeanFactory中所有的WebServerFactoryCustomizer，调用customize函数，初始化WebServerFactory

后续会有更多解析，关于JettyServletWebServerFactory，TomcatServletWebServerFactory

### JettyServletWebServerFactory



全是继承自AbstractServletWebServerFactory

所有的实现类中都没有做特殊操作（初始化之后），等待getServer()函数调用。

在createServer中创建Server，初始化Connector，将Connector装载给Server。

configureWebAppContext 配置WebAppContext。

设置ContextPath。

addDefaultServlet添加默认的Servlet，（org.eclipse.jetty.servlet.DefaultServlet）设定到根目录（/）

mergeInitializers 添加ServletContextInitializer（使用lambda表达式，生成一个），在onstartup的函数里，调用servletContext的setInitParameter函数，将initParameters设置进去。

添加SessionConfiguringInitializer，设置sessioncookie之类的，不过现在基本不用了。

getWebAppContextConfigurations添加配置信息（Configuration），

先添加ServletContextInitializerConfiguration，将初始化类ServletContextInitializer注入进去。

设置ErrorPage，MimeType等

addHandlerWrappers设置Handler

调用ServerCustomizer的customize函数，初始化当前server

在最终调用结束之后，初始化JettyWebServer，并在构造函数中调用initialize函数

调用server的start函数，开启containerlifecycle（调用start函数）启用当前server的start，还有connector的start，

之前将一部分LifeCycle的bean注册到Context中，在这次的server start事件中，连续调用bean的start函数。

比较重要的，设定的ServletContextInitilizer都是在其中一个bean

ServletContextInitializerConfiguration.Initializer中触发的。

**概述启动：**

再刷新Context的时候，使用ServletWebServerFactory创建WebServer，

## WebServer相关

默认在spring-boot-starter-web是引入tomcat的

### EmbeddedWebServerFactoryCustomizerAutoConfiguration

* 1. JettyWebServerFactoryCustomizer

设置jetty相关信息，主要是Connector还有thread等，和tomcat有一点点不同，他使用addServerCustomizers来初始化maxHttpHeaderSize，本质上区别不大

* 1. NettyWebServerFactoryCustomizer

NettyServer的配置要简单一点，只配置一下httpHeader等。

* 1. UndertowWebServerFactoryCustomizer

没用过，用处不大，设置起来和tomcat差不多。

* 1. TomcatWebServerFactoryCustomizer

初始化器，几乎都一样，设置Connector或者engine的地方略过了，这属于web服务器的机制问题了。

### ServletWebServerFactoryAutoConfiguration

使用import引入ServletWebServerFactoryAutoConfiguration.BeanPostProcessorsRegistrar，注册WebServerFactoryCustomizerBeanPostProcessor到Registry中，这个PostProcessor主要是在postProcessBeforeInitialization时（已经实例化当前类，但是还没有初始化）），判定当前类是不是WebServerFactory，获取BeanFactory中所有的WebServerFactoryCustomizer，调用customize函数，初始化WebServerFactory，初始化的内容为**server.**配置。在customize的时候将host，port，contextpath，compression（里面配置mimetypes，excludeuseragent等），contextparameters等。

不管是Tomcat还是Jetty，最终都是继承自AbstractServletWebServerFactory，本质上只是配置一些host，port，ssl，contextpath等信息，并初始化ServletContextInitializer。

1. TomcatServletWebServerFactory
   * 1. 初始化tomcat
     2. 初始化Connector

默认使用Protocol（**org.apache.coyote.http11.Http11NioProtocol**）

* + 1. 将新生成的Connector添加到Tomcat中

Tomcat中有一个真正的Server，是初始化（new）来的，StandardServer。Tomcat将连接管理当做一个服务，默认添加一个名字为Tomcat的服务。获取Tomcat的默认服务（services的第1个），向其中添加Connector，Connector设置服务等于刚刚获取的，如果当前服务出去启动状态，立即启动它。Tomcat使用事件监听的方式，当我们添加一个Connector后，可以主动探知服务的变更，然后执行监听程序

* + 1. customizeConnector初始化Connector

1. customizeProtocol设置protocol的address，其他什么都没干，设置URIEncoding，默认UTF-8，也可以自行设定，一般我们不会设定它的。customizeSsl设置ssl信息，没什么好讲的，就是配置信息里的。当使用ssl的时候，会特殊提升当前的protocol到upgradeProtocol，
2. 在TomcatWebServerFactoryCustomizer做特别初始化的时候，添加customizer，设置maxConnections，acceptCount，connectionTimeout等，因为connector在这时还没生成，故需要传递一下这些serverproperties，一般，我们简单粗暴的使用getBean的方式，不过，显然，spring并没有这么做（spring并没有将webserver，connector纳入到管理的bean的体系中去）。当然，我们也可以在某个时间点（使用继承TomcatWebServerFactory的方式，或者自己实现一个WebServerFactoryCustomizer的方式）去主动添加一个TomcatConnectorCustomizer。不过一般没必要这么干
   * 1. 将connector设置给Tomcat
     2. 设定autodeploy

**注意：**

**Tomcat的主要思想就是服务（StandService）🡪容器（Engine）🡪Host🡪Context**

**其中一般情况下我们不能也不需要去新增一个StandService和Engine，Host。我们发布服务的时候，使用多个Context即可完成多个服务的注册。和普通Tomcat一样，我们也可以配置一些realm信息，不过用处不大，很少有人看这个了。Tomcat中还有两个组件Pipeline和Valve，Pipeline类似于Netty中的Pipeline，Valve类似于ChannelHandler，只不过不区分in，out。可能还是有点模糊，类似于FilterChain和Filter的关系。在每一个组件Engine，Host，Context中都存在Pipeline和Valve。**

**初始化Engine，我们会将默认Pipeline（StandardPipeline）赋值给Engine—在ContainerBase中，并默认创建一个StandardEngineValve，设置为basic，保证basic永远最后一个执行，在添加或者修改Engine的时候，我们会将oldEngine中的Service设置为空，之前是互相引用的，避免被垃圾回收器回收。如果服务已经在启动状态，需要将oldEngine给stop掉。同时，启动newEngine。并重启MapperListener。**

**Engine添加Host就简单多了。Tomcat中的所有组件，Engine，Host，Context都继承自ContainerBase，其中维护了一个HashMap（children），保存着下级。**

* + 1. configureEngine

在6中设置autoDeploy过程中，生成了默认的Engine和Host，这里要配置Engine，主要就是添加Factory中的Valve到Engine中，我们可以自定义一个FactoryCustomizer，添加一个Valve试试。默认情况下，我们永远无法在默认的EngineValve之后添加Valve

* + 1. prepareContext

初始化TomcatEmbeddedContext，继承自StandardContext，同样继承自ContainerBase（默认初始化一个Valve--StandardContextValve）。

添加LifeCycleListener（FixContextListener）

// 待续，FixContextListener

configureTldSkipPatterns添加扫描Filter（过滤某些jar包，比如使用springboot的时候，一部分jar包，自动过滤）

addDefaultServlet添加默认Servlet，**org.apache.catalina.servlets.DefaultServlet**

将Servlet添加到Container中。

添加LifeCycleListener（StaticResourceConfigurer）

// 静态文件遍历

mergeInitializers添加初始化器

匿名初始化器（将initParameters设置到ServletContext中）

SessionConfiguringInitializer（）

ServletWebServerApplicationContext中的匿名初始化器（从BeanFactory中获取以下类型的Bean—ServletContextInitializer,顺序调用onStartup，完成初始化。）

将context加入到host中。

configureContext配置Context，主要有添加mime类型（有默认的基本类型），tomcat支持persistSession，不过一般不配置。

使用tomcatContextCustomizers个性化配置context

这里主要有这么几个ContextCustomizer

在TomcatWebSocketServletWebServerCustomizer（在WebServerFactoryCustomizerBeanPostProcessor中添加的—在WebServerFactory实例化之后初始化之前执行postProcessBeforeInitialization）中添加的匿名类，在这里为Context添加一个listener（WsContextListener）

在TomcatServletWebServerFactoryCustomizer中添加的匿名类，设置Context中的redrectContextRoot，没有太大用。

* + 1. postProcessContext

什么都没干，占位用的

* + 1. 初始化TomcatWebServer（new），构造函数中initialize

1. 添加LifeCycleListener（在tomcat启动的时候，获取server中的所有服务，复制所有Connector到HashMap中，在服务中删除当前Connector）
2. 启动tomcat

调用tomcat的start函数，实际上是调用tomcat中的server（StandardService）的start函数，当前事件是NEW，

调用init函数，调用当前server（StandardService）中的服务（service）的start函数。服务中存在engine，启动engine。Engine没啥好启动的。

启动connector，初始化CoyoteAdapter，将adapter设置到protocolHandler中。

再调用protocolHandler的初始化（init）函数，到这里，初始化完成（init）

第二次事件—STARTING

启动后，初始化（ServletContextInitializer）

StandardContext启动过程中初始化（startInternal）

每一个ServletRegistrationBean都是继承自ServletContextInitializer

将当前的Servlet注册到ServletContext中去。本质上还是一个StandardWrapper。

将当前添加的Servlet和urlpartten组合到一起注册到ServletContext中。

当然所有的Servlet都是这么设置的。

servletRegistrationBean

dispatcherServletRegistration

statViewServletRegistrationBean

webStatFilterRegistrationBean

TomcatWebServer这就初始化完成了，后面，在ServletWebServerApplicationContext中，WebServerFactory的初始化任务（getWebServer）就完成了。后面就是将ServletContext和ServletConfig的参数绑定到PropertySources（Env）中。

在ServletWebServerApplicationContext的finishRefresh中，启动WebServer（start）。

如果Connector存在的话，启动初始化。获取Tomcat绑定的Host，获取host下的所有Context，默认使用的是刚才添加的TomcatEmbeddedContext，启动deferredLoadOnStartup。

获取之前添加的所有Servlet

StandardEngine[Tomcat].StandardHost[localhost].TomcatEmbeddedContext[].StandardWrapper[default]

StandardEngine[Tomcat].StandardHost[localhost].TomcatEmbeddedContext[].StandardWrapper[dispatcherServlet]

StandardEngine[Tomcat].StandardHost[localhost].TomcatEmbeddedContext[].StandardWrapper[statViewServlet]

StandardEngine[Tomcat].StandardHost[localhost].TomcatEmbeddedContext[].StandardWrapper[object]

使用独立线程去初始化Servlet，每一个Servlet在添加的时候都是StandardWrapper去包装一层。

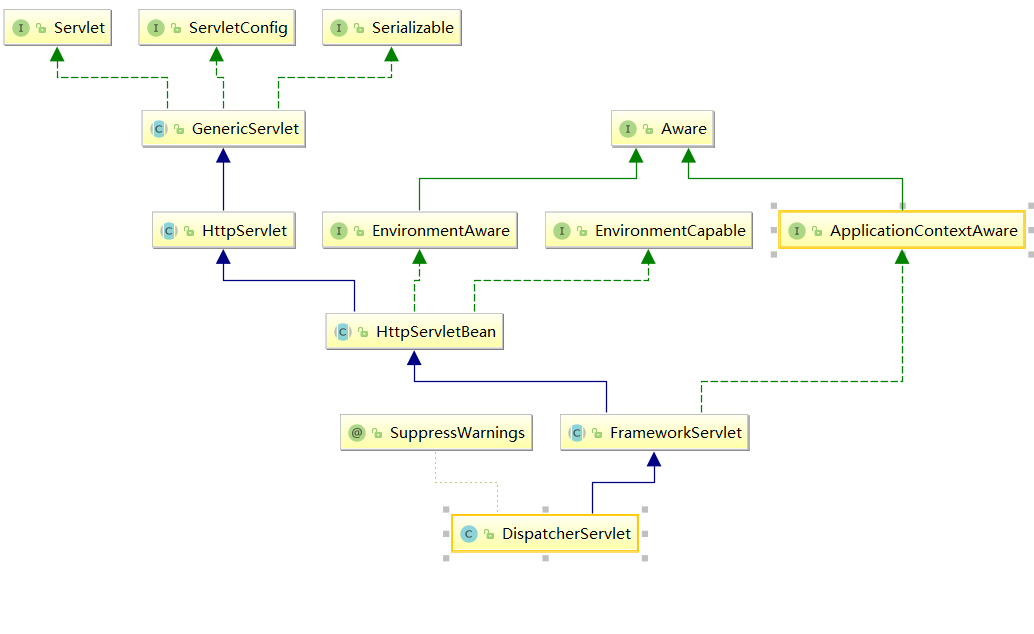
调用Wrapper的load函数，初始化Servlet。

loadServlet（加载Servlet）🡪initServlet（调用init函数，defaultServlet设置了一系列的参数。）这里判定order，order>0的才会在这时加载。

ServletRegistrationBean本身是有初始化的，他会在ServletContextInitializer初始化ServletContext的时候初始化当前Servlet（即StandardContext中的onstartup）。

我们自己创建的Servlet（包括springmvc的），在第一次请求触发的时候，才会初始化。

详细的请求触发就不详细叙述了，大概就是按照Coyote request的结构，先进入Connector，Connector在去找寻服务（我们在将Connector绑定给tomcat的时候，实际上是将Connector绑定给tomcat的service，同时也将这个服务绑定给Connector），使用服务中的Mapper去定位当前的请求，应该指向哪一个Context，指向哪一个Servlet。这里举例，指向了DispatcherServlet，获取到Wrapper之后，如果当前Servlet没有进行初始化，调用初始化函数init



HttpServletBean中的init

如果配置了特定的initParameter，这里配置到ServletConfig中去。

FrameworkServlet🡪initServletBean

调用initWebApplicationContext，获取WebApplicationContext，然后刷新（初始化子类Servlet），onRefresh

1. initMultipartResolver(context);

在MultipartAutoConfiguration中初始化的StandardServletMultipartResolver。将当前Request包装成StandardMultipartHttpServletRequest。

1. initLocaleResolver(context);

不处理这个

1. initThemeResolver(context);

这个也不处理。

1. initHandlerMappings(context);

获取配置好的HandlerMappings

* + - * 1. faviconHandlerMapping

使用SimpleUrlHandlerMapping，urlMapping：**\*\*/favicon.ico**

Handler：ResourceHttpRequestHandler，使用上述四个静态地址去获取favicon.ico

* + - * 1. requestMappingHandlerMapping

RequestMappingHandlerMapping

设置拦截器。

默认拦截器ConversionServiceExposingInterceptor，

ResourceUrlProviderExposingInterceptor

在使用Jpa的时候，Jpa的JpaWebMvcConfiguration会注入

OpenEntityManagerInViewInterceptor

ConversionServiceExposingInterceptor，将ConversionService设置进入Interceptor，在调用interceptor的prehandle的时候，设置进入Request的attibute

如果需要手动添加的话，可以继承WebMvcConfigurer，自行添加，两种interceptor。WebRequestInterceptor🡪 WebRequestInterceptor

HandlerInterceptorAdapter🡪 HandlerInterceptor

setContentNegotiationManager

设置ContentNegotiationManager，content转译管理器。默认情况下处理xml和json，可以使用WebMvcConfigurer去添加ContentTypeStrategy，在build ContentNegotiationManager的时候，先判定strategies，如果他有，其他的就过了。如果没有这个，并且忽略head的话，会添加defaultContentTypeStrategy

setCorsConfigurations

设置Cors，默认情况下是没有的，我们可以自己添加一个

继承自AbstractHandlerMethodMapping，在它初始化完成之后，调用afterPropertiesSet中调用initHandlerMethods继续初始化。processCandidateBean处理可处理的bean。判定当前类是否是Controller，RequestMapping注解的类。

处理当前类中所有的函数，判定是否是使用RequestMapping注解的函数。

将当前获得的url（RequestMapping）和当前类、函数注册到注册中心。

* + - * 1. viewControllerHandlerMapping

SimpleUrlHandlerMapping

详情见ViewControllerWebMvcConfigurer，一种跳转类的配置，不需要弄空的Controller了。不过，不如那个可扩展性强。

* + - * 1. beanNameHandlerMapping

BeanNameUrlHandlerMapping

这个就简单了，把beanName弄到url里，去处理这个请求。

参照这个UserServiceBeanNameController。

* + - * 1. resourceHandlerMapping

SimpleUrlHandlerMapping

在WebMvcSAutoConfiguration中，默认设置了所有的静态文件拦截path是/\*\*，

所以，即便我们不设置，其实他也会走到这里去处理，除非在controller里去处理（比如实例中的b.js），当我们调用/b.js的时候，会先被前面的mapping拦截到。我们可以使用spring.mvc.static-path-pattern去设定path，不过没必要，这个已经基本是最后的拦截mapping了，基本请求都在上面就走完了。所以，这里我们也一般不会默认设置这些。不过，还是注意，不要写出很恶心的请求，比如，在Controller中设定一个RequestMapping为b.js

* + - * 1. defaultServletHandlerMapping

SimpleUrlHandlerMapping

默认情况下没有开启这个，即便开启了，他也只是负责跳转类的。

* + - * 1. welcomePageHandlerMapping

WelcomePageHandlerMapping

放在那四个静态文件夹里就行了，当然，也可以放jsp里。

**总结：HandlerMapping里主要用RequestMappingHandlerMapping，用他来过滤普通请求，包括我们定义的那些，如果没找到的话，继续判断，有可能是跳转的Mapping，还不是的话，有可能是BeanNameHandlerMapping（使用beanname来判定url），再不是的话，就是静态文件，还不是的话，就是后面你的了，主要就是3 4 5 6；**

1. initHandlerAdapters(context);

获取配置好的HandlerAdapter

1. RequestMappingHandlerAdapter

我们普通使用的基本都是这里处理的，我们使用requestMapping注解的函数，将会被HandlerMapping去解析，返回一个Handler（HandlerMethod），之后，就会判定到这个Adapter里。

在afterPropertiesSet里，初始化argumentResolver，包括一部分request，response等的参数，通过这些resolver注入到函数调用过程中去。

请求进来后：

1. 判定当前request的函数（GET,POST等）
2. 使用request response初始化ServletWebRequest
3. 初始化ServletInvocableHandlerMethod，
4. invokeAndHandle

获取需要调用的method，获取method的参数，填充进去，之后，调用method。其中的argumentResolver，也可以自己定义，然后注入函数调用中，不过一般没必要。

在函数调用结束后，如果有返回值。

使用supportsReturnType判定returnValueResolver。

这里使用RequestResponseBodyMethodProcessor，因为使用@ResponseBody注解了返回值。使用canWrite选择messageConverter，最终选择可以处理Map的MessageConverter。最终使用mc的outputStream，发送返回值出去。

1. postHandle

函数调用结束后，调用拦截器（HandlerInterceptor，一般我们都是继承这个

HandlerInterceptorAdapter）。

1. triggerAfterCompletion

最终处理调用结果。Interceptor

请求结束。

1. HttpRequestHandlerAdapter
2. SimpleControllerHandlerAdapter

特殊请求，例如welcome（/）跳转到index的，会进入这里处理。

1. initHandlerExceptionResolvers(context);
2. ExceptionHandlerExceptionResolver

afterPropertiesSet里初始化当前异常

使用@ControllerAdvice注解Class，@ExceptionHandler注解的函数，添加一个新的ExceptionHandlerMethodResolver，里面去解析ExceptionHandler注解，添加handlerMethod到配置缓存里。ExceptionHandler注解里的value就是需要判定的Exception类型，或者在参数里有类型的话，也可以。

1. ResponseStatusExceptionResolver

大概是使用@ResponseStatus注解的异常类，你抛异常后，它会自己去处理返回。

参照UserServiceException，不是特别有用。

1. DefaultHandlerExceptionResolver

所有的默认的异常处理，都是这里处理的，异常类型已经定义

**if** (ex **instanceof** HttpRequestMethodNotSupportedException) {  
 **return** handleHttpRequestMethodNotSupported(  
 (HttpRequestMethodNotSupportedException) ex, request, response, handler);  
}  
**else if** (ex **instanceof** HttpMediaTypeNotSupportedException) {  
 **return** handleHttpMediaTypeNotSupported(  
 (HttpMediaTypeNotSupportedException) ex, request, response, handler);  
}  
**else if** (ex **instanceof** HttpMediaTypeNotAcceptableException) {  
 **return** handleHttpMediaTypeNotAcceptable(  
 (HttpMediaTypeNotAcceptableException) ex, request, response, handler);  
}  
**else if** (ex **instanceof** MissingPathVariableException) {  
 **return** handleMissingPathVariable(  
 (MissingPathVariableException) ex, request, response, handler);  
}  
**else if** (ex **instanceof** MissingServletRequestParameterException) {  
 **return** handleMissingServletRequestParameter(  
 (MissingServletRequestParameterException) ex, request, response, handler);  
}  
**else if** (ex **instanceof** ServletRequestBindingException) {  
 **return** handleServletRequestBindingException(  
 (ServletRequestBindingException) ex, request, response, handler);  
}  
**else if** (ex **instanceof** ConversionNotSupportedException) {  
 **return** handleConversionNotSupported(  
 (ConversionNotSupportedException) ex, request, response, handler);  
}  
**else if** (ex **instanceof** TypeMismatchException) {  
 **return** handleTypeMismatch(  
 (TypeMismatchException) ex, request, response, handler);  
}  
**else if** (ex **instanceof** HttpMessageNotReadableException) {  
 **return** handleHttpMessageNotReadable(  
 (HttpMessageNotReadableException) ex, request, response, handler);  
}  
**else if** (ex **instanceof** HttpMessageNotWritableException) {  
 **return** handleHttpMessageNotWritable(  
 (HttpMessageNotWritableException) ex, request, response, handler);  
}  
**else if** (ex **instanceof** MethodArgumentNotValidException) {  
 **return** handleMethodArgumentNotValidException(  
 (MethodArgumentNotValidException) ex, request, response, handler);  
}  
**else if** (ex **instanceof** MissingServletRequestPartException) {  
 **return** handleMissingServletRequestPartException(  
 (MissingServletRequestPartException) ex, request, response, handler);  
}  
**else if** (ex **instanceof** BindException) {  
 **return** handleBindException((BindException) ex, request, response, handler);  
}  
**else if** (ex **instanceof** NoHandlerFoundException) {  
 **return** handleNoHandlerFoundException(  
 (NoHandlerFoundException) ex, request, response, handler);  
}  
**else if** (ex **instanceof** AsyncRequestTimeoutException) {  
 **return** handleAsyncRequestTimeoutException(  
 (AsyncRequestTimeoutException) ex, request, response, handler);  
}

当然，我们也可以自定义一些异常类型，可以使用继承自AbstractHandlerExceptionResolver的方式，生成Resolver，使用shouldApplyTo判定是否需要这个resolver处理。使用doResolveException，处理异常。一般情况下，我们使用Controller映射的Request，shouldApplyTo的参数Object就是Controller的函数自己，所以，我们可以判断处理某一特定类型的Class。

也可以使用实现HandlerExceptionResolver的方式，使用resolveException处理异常。

1. initRequestToViewNameTranslator(context);

查询beanFactory中的viewNameTranslator（RequestToViewNameTranslator），如果存在就是用beanFactory中的，要么用默认配置的（

org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator）。

系统默认的，没太大用，我们可以自己搞一个。详细可以参照

UserServiceRequestToViewNameTranslatorConfiguration

1. initViewResolvers(context);
2. BeanNameViewResolver

根据viewname判定view，用的比较少

1. ViewResolverComposite
2. InternalResourceViewResolver

获取静态文件地址信息。

1. ContentNegotiatingViewResolver

根据文件类型还有Header信息获取view。

使用contentNegotiationManager，根据MediaType，获取可能的文件地址（一般情况下，不这么用）。

1. initFlashMapManager(context);

FlashMapManager，没看这个，没太大用。

1. JettyServletWebServerFactory

## WebMvc相关

### DispatcherServletAutoConfiguration

在服务器配置（3.2）完成之后，开启DispatcherServlet配置。

后面有详解DispatcherServlet和DispatcherServletRegistration

* 1. DispatcherServletConfiguration

当我们没有配置DispatcherServlet.class或者beanName等于（dispatcherServlet）的bean时触发

使用**spring.http、spring.mvc**完成相关配置。

配置DispatcherServlet，SpringMVC的中央控制器，所有的请求由这里分发。

当需要有MultipartResolver的时候，获取之。

* 1. DispatcherServletRegistrationConfiguration

配置DispatcherServletRegistrationBean信息，在1之后完成注册，所以import了1，防备1还没有解析的情况。

配置完成，主要就是配置了两个Bean

### TaskExecutionAutoConfiguration

创建taskExecutor，本质上，还是一个ThreadPoolExecutor、ExecutorService，用于创建、管理线程池

### ValidationAutoConfiguration

对校验没太大兴趣，略过。

### WebMvcAutoConfiguration

1. OrderedHiddenHttpMethodFilter

**spring.mvc.hiddenmethod.filter.enabled**

根据该属性判定，默认是true

没有太大的用处，在post请求中添加一个\_method参数，判定put delete patch，做一次请求的转换，将当前的post请求转换成为put delete

1. OrderedFormContentFilter

**spring.mvc.formcontent.filter.enabled**

有太大用的东西，上面不是做了一次转译了吗，这里将form发过来的content转换成request的param，一般也用不到。

1. EnableWebMvcConfiguration内部类

继承自DelegatingWebMvcConfiguration，继承自WebMvcConfigurationSupport

1. WebMvcConfigurationSupport
   1. 定义RequestMappingHandlerMapping

1：初始化RequestMappingHandlerMapping

2：设置拦截器（初始化拦截器InterceptorRegistry，）

1. DelegatingWebMvcConfiguration

属性WebMvcConfigurerComposite

@Autowired(required = **false**)  
**public void** setConfigurers(List<WebMvcConfigurer> configurers) {  
 **if** (!CollectionUtils.*isEmpty*(configurers)) {  
 **this**.**configurers**.addWebMvcConfigurers(configurers);  
 }  
}

将WebConfigure信息加入到其中，将来一个一个的设置。默认的使用的有这么几个webconfigure

WebMvcAutoConfigurationAdapter

JpaWebConfiguration

SpringDataWebConfiguration

1. RequestMappingHandlerAdapter

注册该Bean。

1. 设置ContentNegotiationManager

初始化ContentNegotiationConfigurer

设置MediaTypes，默认使用**javax.xml.bind.Binder**作为xml的解析器，使用**com.fasterxml.jackson.databind**作为json的解析器。使用初始化的WebConfigure来初始化configureContentNegotiation。调用每一个WebConfigure的configureXXX函数。

在WebMvcAutoConfigurationAdapter中config的是设置一些ContentNegotiation信息，比如是否强制转化/abc.pdf到application/pdf，我感觉没太大用。还有就是设置mediatype，示例中有，如果不设置的话，应该是all

使用ContentNegotiationManagerFactoryBean去build一个实例。

设置ContentNegotiationStrategy，这个策略是用来判定请求的类型的。

默认添加HeaderContentNegotiationStrategy，使用header中的accept来判定

返回一个ContentNegotiationManager

1. 设置MessageConverter

默认添加了以下的MessageConverter

ByteArrayHttpMessageConverter

StringHttpMessageConverter

ResourceHttpMessageConverter

ResourceRegionHttpMessageConverter

SourceHttpMessageConverter

AllEncompassingFormHttpMessageConverter

Jaxb2RootElementHttpMessageConverter

MappingJackson2HttpMessageConverter

基本覆盖了所有的数据类型，包含了xml，json，string，multipart等，一般情况下，我们不需要自己去设置这个。

1. 设置ConfigurableWebBindingInitializer

WebData的初始化器

设置ConversionService，默认使用DefaultFormattingConversionService

同样，校验略过

1. 设置HandlerMethodArgumentResolver，同样略过。

WebMvcConfigurer

1. 设置HandlerMethodReturnValueHandler

Spring默认使用一下返回结果。

默认使用一下的handler，注意，是有顺序的，一旦上面support，直接就返回了

ModelAndViewMethodReturnValueHandler

ModelMethodProcessor

ViewMethodReturnValueHandler

ResponseBodyEmitterReturnValueHandler

StreamingResponseBodyReturnValueHandler

HttpEntityMethodProcessor

HttpHeadersReturnValueHandler

CallableMethodReturnValueHandler

DeferredResultMethodReturnValueHandler

AsyncTaskMethodReturnValueHandler

ModelAttributeMethodProcessor

RequestResponseBodyMethodProcessor

ViewNameMethodReturnValueHandler

MapMethodProcessor

我们自己设定的handler，优先级最低，最后才会执行。所以，一般情况下，我们自己的是不会执行的，除非我们自己设定一下返回的条件。

1. JsonViewRequestBodyAdvice

XML解析器存在的情况下，设置请求的拦截器，setRequestBodyAdvice，请求来了之后，可以使用这个拦截，防止，被请求解析器获取。

1. JsonViewResponseBodyAdvice

同上，只不过是解析返回的。

1. TaskExecutor

设置这个而已，默认情况下，是使用当前Adapter的。

1. 后面设置callableInterceptor

到这里，初始化完毕了。

1. RequestMappingHandlerMapping

注册该bean

1. setInterceptors

设置拦截器。

默认拦截器ConversionServiceExposingInterceptor，

ResourceUrlProviderExposingInterceptor

在使用Jpa的时候，Jpa的JpaWebMvcConfiguration会注入

OpenEntityManagerInViewInterceptor

ConversionServiceExposingInterceptor，将ConversionService设置进入Interceptor，在调用interceptor的prehandle的时候，设置进入Request的attibute

1. setContentNegotiationManager

设置ContentNegotiationManager

1. setCorsConfigurations

设置Cors，默认情况下是没有的，我们可以自己添加一个

到这里就创建完成了，有时间再添加pathMatcher和pathHelper

1. mvcConversionService

初始化WebConversionService，继承自

DefaultFormattingConversionService

FormattingConversionService

GenericConversionService

和之前的类似，同样是各种WebMvcConfigurer来初始化（addFormatters）。

默认使用SpringDataWebConfiguration中的设置。

DistanceFormatter，这个玩意不知道干嘛的。

PointFormatter，这个也是。

1. GenericConversionService

canConvert：判定当前Object是否可以转换成target

convert：转换

1. FormattingConversionService
2. DefaultFormattingConversionService

构造函数中，添加默认ConversionService

里面有大量不同的Converter，详情参看DefaultConversionService，我们使用的各种对象，基本都有自动转译的。

默认添加的Formatter为NumberFormatter，使用AnnotationPrinterConverter和

AnnotationParserConverter来完成format。

如果**javax.money.MonetaryAmount**存在的情况下，添加Money的各种formatter

目前我没用上

1. WebConversionService
2. mvcResourceUrlProvider

判定资源是否是静态资源

初始化ResourceUrlProvider，并设置pathHelper和pathMatcher

初始化PathMatchConfigurer，默认的pathHelper和pathMatcher都是null

1. AntPathMatcher
2. UrlPathHelper

这两个都没太大用，现在已经不怎么用了。Global级配置。

1. viewControllerHandlerMapping

返回一个默认的SimpleUrlHandlerMapping

1. beanNameHandlerMapping

使用beanName去寻找Controller，其实也没有太大用，一种不同的类型罢了

详情参考UserServiceBeanNameController

1. resourceHandlerMapping

初始化ResourceHandlerRegistry。

添加Resource配置。

WebMvcAutoConfiguration中的WebMvcAutoConfigurationAdapter添加默认配置

默认的静态文件配置在**classpath:/META-INF/resources/webjars/**

**"classpath:/META-INF/resources/"**, **"classpath:/resources/"**,  
**"classpath:/static/"**, **"classpath:/public/"**

返回SimpleUrlHandlerMapping

静态文件指向可以修改**static-path-pattern**: /html/\*\*

然后指到上面的四个文件夹中，需要注意的是，我们使用resources当做最外层文件夹的时候，其实外层还有一个classpath的resources

1. mvcResourceUrlProvider

初始化ResourceUrlProvider

1. defaultServletHandlerMapping

所有的没有找到目标的url指向某一个default，感觉意义不大。和404的没啥区别。

1. httpRequestHandlerAdapter

初始化一个HttpRequestHandlerAdapter

1. simpleControllerHandlerAdapter

初始化SimpleControllerHandlerAdapter

1. handlerExceptionResolver

初始化ExceptionResolver，系统默认的异常处理已经够用，暂时不需要我们再添加，注意，一旦我们添加了，系统默认的就不会添加了。如果需要自行添加，可以这么着来extendHandlerExceptionResolvers

DefaultHandlerExceptionResolver

ResponseStatusExceptionResolver

ExceptionHandlerExceptionResolver

1. mvcViewResolver

默认使用一下四种

beanNameViewResolver

mvcViewResolver

defaultViewResolver

viewResolver—ContentNegotiatingViewResolver

1. WebMvcAutoConfigurationAdapter
2. InternalResourceViewResolver

主要处理一些jsp Servlet等

1. BeanNameViewResolver

按照beanname来解析return view

1. ContentNegotiatingViewResolver

根据content去推断view

1. LocaleResolver

国际化相关，设定locale

1. WelcomePageHandlerMapping

设定开始页，找index.html，获取第一个index.html。顺序参照上面的静态文件

由于我之前设定了static-path-pattern，所以现在找不到index.html了，他只能去找其他provider

比如：FreeMarkerTemplateAvailabilityProvider

MustacheTemplateAvailabilityProvider

GroovyTemplateAvailabilityProvider

ThymeleafTemplateAvailabilityProvider

JspTemplateAvailabilityProvider

这里就不改了，太麻烦，修改static-path-pattern还原成/\*\*就可以了。

1. RequestContextFilter

没太大鸟用，主要是把RequestContext开放给子线程，默认也是不开启的。

1. FaviconConfiguration

就那个小图标的。将那个图标随便放在上面的静态文件夹任何一个位置就能开启了，没太大鸟用。

到这里，所有的配置信息，基本就结束了，后面都是逻辑

### 流程解析

#### DispatcherServlet

本质上还是个Servlet，负责分发请求。

**DispatcherServlet.properties**



配置DispatcherServlet默认使用的

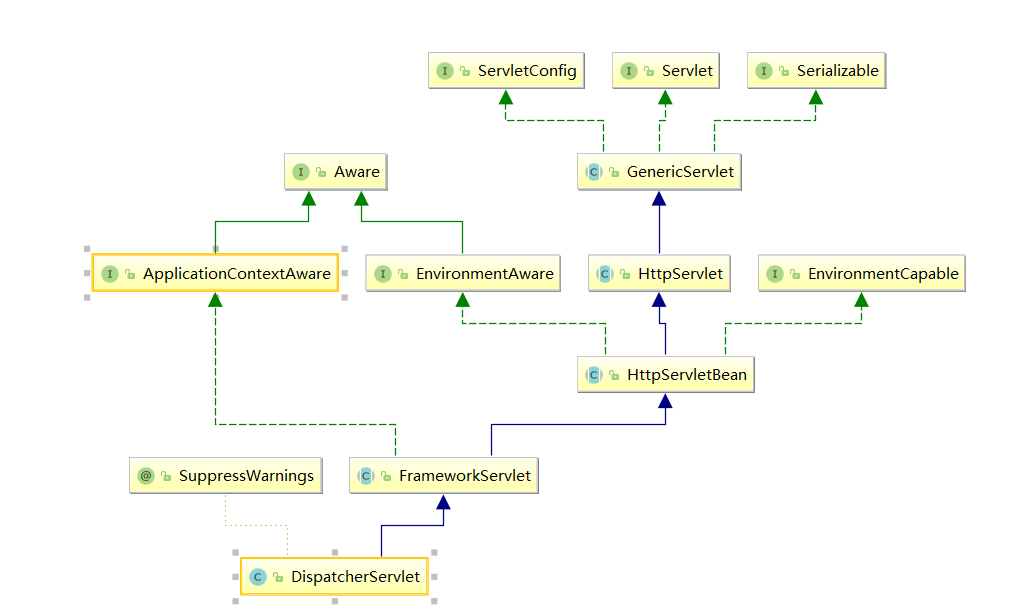
**org.springframework.web.servlet.LocaleResolver**=**org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver  
  
org.springframework.web.servlet.ThemeResolver**=**org.springframework.web.servlet.theme.FixedThemeResolver  
  
org.springframework.web.servlet.HandlerMapping**=**org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,\  
 org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping  
  
org.springframework.web.servlet.HandlerAdapter**=**org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,\  
 org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\  
 org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter  
  
org.springframework.web.servlet.HandlerExceptionResolver**=**org.springframework.web.servlet.mvc.method.annotation.ExceptionHandlerExceptionResolver,\  
 org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver,\  
 org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver  
  
org.springframework.web.servlet.RequestToViewNameTranslator**=**org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator  
  
org.springframework.web.servlet.ViewResolver**=**org.springframework.web.servlet.view.InternalResourceViewResolver  
  
org.springframework.web.servlet.FlashMapManager**=**org.springframework.web.servlet.support.SessionFlashMapManager**

一切的起点都是

**public void** init(ServletConfig config) **throws** ServletException

当容器启动完成之后，会去加载Servlet，调用Servlet的init函数。

详细的容器在什么时间点添加servlet等，在3.2中



调用HttpServletBean中的init函数，启动初始化

## 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

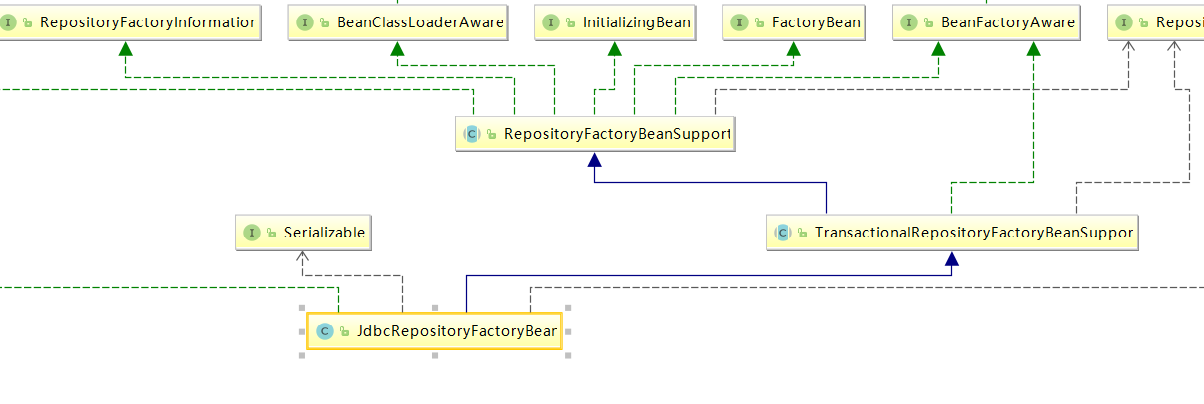
RepositoryComposition，应该是定义的其他查询。

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

EnableTransactionManagement

判定使用哪种代理方式，这里不是重点。

* 1. AbstractPlatformTransactionManager

实现了PlatformTransactionManager，主要有三个函数

* + 1. getTransaction

获取事务。

* + 1. commit

根据事务状态，选择是否提交当前事务，当前事务如果在另一个事务内，则忽略本次提交。如果之前有个事务被挂起了，当提交完当前事务后，唤醒之前的事务。这个函数之后的异常，不会再影响事务本身已经成功（这句话没太大用）

* + 1. rollback

回滚当前事务，如果有外层事务，标记回滚，留给外层处理。

TransactionDefinition接口

事务类型

1. PROPAGATION\_REQUIRED

事务类型，当需要时，开启事务。spring默认事务类型

1. PROPAGATION\_SUPPORTS

支持当前事务，如果当前没有事务，就以非事务方式执行

1. PROPAGATION\_MANDATORY

使用当前的事务，如果当前没有事务，就抛出异常

1. PROPAGATION\_REQUIRES\_NEW

新建事务，如果当前存在事务，把当前事务挂起

1. PROPAGATION\_NOT\_SUPPORTED

以非事务方式执行操作，如果当前存在事务，就把当前事务挂起

1. PROPAGATION\_NEVER

以非事务方式执行，如果当前存在事务，则抛出异常

1. PROPAGATION\_NESTED

如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION\_REQUIRED类 似的操作

以上的事务类型，并不是所有的TransactionManager都支持的。

事务隔离级别和标准的Connection的隔离级别一致。不再过多强调。

1. ISOLATION\_DEFAULT

默认隔离级别。

1. ISOLATION\_READ\_UNCOMMITTED
2. ISOLATION\_READ\_COMMITTED
3. ISOLATION\_REPEATABLE\_READ
4. ISOLATION\_SERIALIZABLE

其他函数只是获取隔离级别和事务类型。

getTransaction()

核心函数，获取事务。

事务对象—JdbcTransactionObjectSupport

继承自SmartTransactionObject（flush和isrollbackonly）发射事务，或者标识当前事务只允许回滚。

主要有以下几个属性

1. ConnectionHolder

保存着连接信息和当前连接信息。属性ConnectionHandle同样保存着连接信息。Spring倾向于支撑jdbc3.0中的savepoint，所以这里也可以添加savepoint。

Savepoint大概是这么个东西，他在事务中创建一个点，当事务回滚的时候，回滚到当前点，之后的事务还是允许提交或者回滚的。Spring的嵌套事务大抵基于此规则完成，在内层事务开启的时候，创建一个savepoint，当内层事务提交的时候，删除该savepoint，当内层事务回滚的时候，回滚到当前savepoint，可以选择回滚或者提交。

1. previousIsolationLevel

当前隔离级别

这里使用的是JpaTransactionObject，继承自JdbcTransactionObjectSupport

初级橙子JdbcTrsactionObjectSupport中的外，该类中有下面重要属性

1. EntityManagerHolder

这个EntityManagerHolder在Jpa的默认实现中使用的是sessionImpl，就是Hibernate的SessionFactory创建的。

事务对象创建完毕之后，开启事务。

获取当前事务对象的Dialect，开启事务，使用Hibernate的Transaction实现开启事务。本质上，和从前使用session.getTransaction().begin()一个意思。返回SessionTransactionData对象。将TransactionData对象加入到TransactionObject中去。获取连接ConnectionHandle，将ConnectionHandle绑定给ConnectionHolder，再将ConnectionHolder绑定给TransactionObject。

至此，获取事务完毕，主要是ConnectionHolder，保存Connection信息，EntityManagerHolder保存对象信息。开启事务本身交给了Dialect来完成，默认使用SessionImpl，最终生成TransactionInfo，绑定给当前线程。

整个操作及其复杂，又曲折。目的是为了增加扩展性，为其他类型的事务管理增加可扩展接口（类），使用ConnectionHandle绑定给ConnectionHolder，再将ConnectionHolder绑定给TransactionObject，使用TransactionData作为对象的保存单元，并且，因为有不同的Dialect，不同的Dialect使用不同的方式开启事务。最终将不同的TransactionStatus绑定给当前线程，完成事务开启。

当然，在开启的时候，要复杂的多，还需要判定当前调用是否支持事务，是否需要新开启一个事务，是否加入当前事务。是否需要挂起等。使用某些唯一的key，将TransactionObject，ConnectionHolder等保存到本地线程。

当我们挂起一个事务的时候，只需要将ConnectionHolder置空，这时由于没有ConnectionHolder，所以，部分操作将被挂起（不再执行），直至正确或异常状态下，恢复当前ConnectionHolder。

本质上，和Hibernate中的session.getTransaction()没有区别，Jpa还是使用的Hibernate的那一套，只不过，官方还有其他实现，注入Jta等。不过由于现在的分布式事务的复杂性，现在这些基本用不上了。感觉Spring官方也暂时搁置这些（事务管理），将重心放到别的地方。

* 1. JpaTransactionManager

上述讲的已经包含了一部分JpaTransactionManager

* 1. DataSourceTransactionManager

这个就比JpaTransactionManager要简单的多，不管是getTransaction还是BeginTransaction都是简单的实现（我感觉也够用了）--因为不存在EntityManager，Session，HibernateTransaction的概念。相对简单的实现，只有ConnectionHolder，开启事务也是简单的conn.setAutoCommit(false)。但是关于事务类型（加入还是创建新的事务，挂起当前事务），都是在父类中完成，故，不影响其他功能，只是没有session级缓存（也就是我们两次查询同一个对象时，真的执行了两次）。

### TransactionAutoConfiguration

#### TransactionManagementConfigurationSelector

AdviceMode.PROXY

默认使用这种方式，当然也可以使用另外一种方式，AspectJ。

1. AutoProxyRegistrar

继承自ImportBeanDefinitionRegistrar，注册相应的Bean

Registry中如果不存在org.springframework.aop.config.internalAutoProxyCreator的话，

注册InfrastructureAdvisorAutoProxyCreator到Registry，顺便将proxyTargetClass写进去。

1. ProxyTransactionManagementConfiguration
2. BeanFactoryTransactionAttributeSourceAdvisor

连接点（设置TransactionAttributeSource，通知advice--TransactionInterceptor）

1. TransactionInterceptor

invoke的时候调用invokeWithinTransaction。

AnnotationTransactionAttributeSource

getTransactionAttribute，判定该类是否需要代理（是否有事务），注解方式。

使用JtaTransactionAnnotationParser和SpringTransactionAnnotationParser判定事务。先判定当前函数Method是否需要添加事务。再判定类级是否需要添加事务。

#### TransactionTemplate

### JtaAutoConfiguration

一般我们不会使用jta的transaction的

### HibernateJpaAutoConfiguration

默认会开启HibernateJpa的配置。当我们关闭这个自动配置后，会使用3.5.7中生成的DataSourceTransactionManager来管理事务。3.5.7的配置级别是相当低的，所以默认使用的是HibernateJpaAutoConfiguration。

#### JpaBaseConfiguration

1. 实例化PlatformTransactionManager，使用的是JpaTransactionManager。
2. 实例化JpaVendorAdapter，spring.jpa的部分配置，会写入到这里，这里就说来话长了，早期ejb没有这么便利，后面ejb3.0之后有关entity的地方大量借用了hibernate的代码（kevin king亲自操刀了3.0的entityBean）所以，后面jpa的标准实现里很多使用了hibernate。

生成EntityManagerFactoryBuilder，将来生成LocalContainerEntityManagerFactoryBean用。

1. 实例化LocalContainerEntityManagerFactoryBean
2. Web环境下，添加opensessioninview到web注册表中。

主要的配置是LocalContainerEntityManagerFactoryBean，后面还有用处。

#### HibernateJpaConfiguration

这里没做太多东西。

## Elasticsearch相关

### ElasticsearchAutoConfiguration

### ElasticsearchDataAutoConfiguration

### ElasticsearchRepositoriesAutoConfiguration

### JestAutoConfiguration

### RestClientAutoConfiguration

## WebSocket相关

### WebSocketReactiveAutoConfiguration

### WebSocketServletAutoConfiguration

### WebSocketMessagingAutoConfiguration

## Oauth2相关

### OAuth2ClientAutoConfiguration

### ReactiveOAuth2ClientAutoConfiguration

### OAuth2ResourceServerAutoConfiguration

### ReactiveOAuth2ResourceServerAutoConfiguration

## Mongo相关

### EmbeddedMongoAutoConfiguration

### MongoAutoConfiguration

### MongoReactiveAutoConfiguration

## Cache相关

一般情况下，不会触发。Spring有一整套cache的操作，不过我们很少用的到。

### CacheAutoConfiguration

## Kafka相关

### KafkaAutoConfiguration

## 定时器（Quartz）相关

### QuartzAutoConfiguration