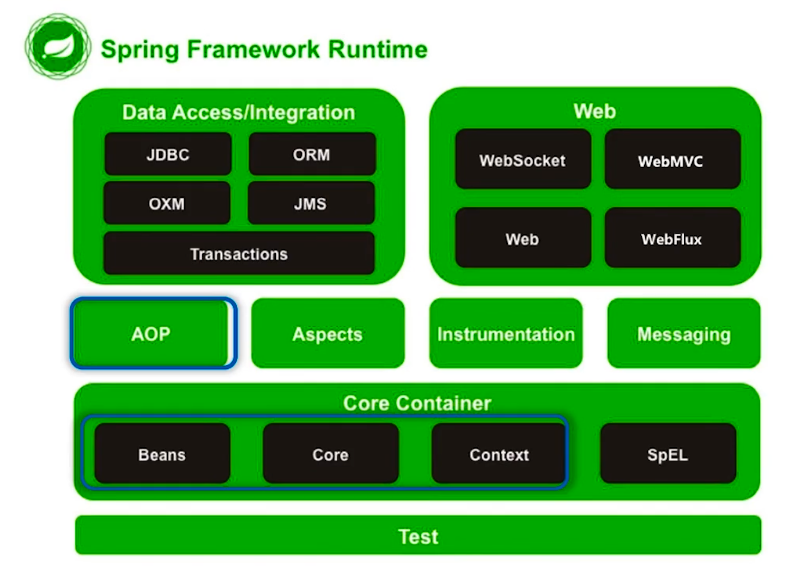
## Spring



重点模块

spring-core:包含框架基本的核心工具类,其它组件要都要使用到这个包里的类,定义并提供资源的访问方式

spring-beans : Spring主要面向Bean编程( BOP)。

Bean的定义

Bean的解析

Bean的创建

spring-context: 为Spring提供运行时环境,保存对象的状态

扩展了BeanFactory

spring-aop:最小化的动态代理实现

JDK态代理

Cglib

只能使用运行时织入,仅支持方法级编织,仅支持方法

执行切入点

## 下载源码

## Spring IOC

### 引入

### 启动

1、创建ClassPathXmlApplicationContext

1、调用父类构造方法,进行相关的对象创建等操作1

2、传递进来的String参数,内部属性和系统属性会把传进来的传输进行替换,${}

2、refresh().prepareRefresh

1、设置系统的启动时间

2、设置容器的标志位。

3、 initPropertySources该方法可以有用户实现,对系统环境变量的获取和设置。

4、验证需要的属性是否都已经放入enviroment对象中。这里会循环变量。应该是验证上一步添加的属性值。

5、准备监听器和事件的集合对象，默认为空的集合。

3、BeanFactory: 加载xmL配置文件的属性值到当前工厂中

1、创建BeanFactory

2、customizeBeanFactory可以重写该方法，自定义beanfactory。

1、是否允许覆盖同名称的不同定义的对象allowBeanDefinitionOverriding

2、循环依赖allowCircularReferences

3、初始化documentReader,并进行XML文件读取及解析。

1、设置系统环境变量Environment

2、设置资源加载器ResourceLoader

3、设置实体解析器ResourceEntityResolve(ResourceEntityResolver.class)，他会加载所需要的dtd文件和xsd文件

(如果没有联网，spring.schemas(spring-bean中的resource)有链接和dtd文件的映射关系), 链接包含了一个标签的信息。

例如: <http://www.springframework.org/schema/beans> 文件中包含beans标签内部的定义和属-性。

4、loadBeanDefinitions

1、创建一个xml的beanDefinitionReader,xml解析器。

2、给reader对象设置环境对象Environment、ResourceLoader、EntityResolver

Environment

ResourceLoader

EntityResolver

3、loadBeanDefinitions

1、获取到当前配置文件的路径，这个文件就是开始进去的配置名称。

2、循环对文件进行处理，返回一个resource对象。

3、循环遍历每个resource对象,读取文件doLoadBeanDefinitions

4、doLoadBeanDefinitions

1、文件格式的校验

2、将resource解析成为document。

3、将document转为beandefinition。

5、doRegisterBeanDefinitions解析 document转为beandefinition

1、创建了bean的解析器。

2、从根结点开始依次解析

命名空间解析:imprort bean等,

自定义解析:

3、例如bean解析(解析的过程中对参数校验)

1、解析其中的name，id, aliase别名,将会对别名进行切割。

2、详细解析

1、解析class属性

2、解析是否有父类。

3、创建当前bean的beanDefination对象，开始设置属性值。

4、将beanDefination放入到beanDefinationMap中。

1、判断是否map中已经存在当前bean。

2、如果允许覆盖，直接覆盖，否则抛出异常。

3、将beanDefination 放入Map<String, BeanDefinition> beanDefinitionMap

Key:beanName value BeanDefinition

5、存放bean的别名 Map<String, String> aliasMap

Key: beanName value: alias

### beanFactory的准备工作

1、设置基本属性

Bean的classloader

Bean的表达式解析器。例如spl表达式。

Bean的属性编辑器(可自定义):对属性值解析，属性值值可能有不同的格式，可以在这里做解析，然后返回对象。在对象的实例化后会调用。

1、自定义实现PropertyEditorSupport。

2、让spring能够识别此编辑其，自定义实现一个属性编辑器的注册器，实现ResourceEditorRegistrar,让spring能够识别到对应的注册器。

Bean的BeanPostProcessor:完成某些对象的aaware对象注入。

### postProcessBeanFactory空方法，拓展方法。

拓展实现: postProcessBeanFactory(beanFactory)

### invokeBeanFactoryPostProcessors(beanFactory) 这里是复数。

作用:实例化并且调用所有已经注册了BeanFactoryPostProcessor，

BeanFactoryPostProcessor的作用:可以对beanfactroy拓展实现，重点修改beandefintions。

BeanFactoryPostProcessor的类型有三种

1、手动添加的beanfactirypostprocessor。

2、实现了beanDefinitionRegistryPostProcessor接口。

3、实现了beanFactoryPostProcesso接口。

ConfigurationClassPostProcessor:构建和验证一个类是否被@configuration修饰，并做相关的解析工作。

1、遍历当前BeanDefinition，判断是否有configuration.class.

2、获取类的注解信息。

3、是否是一个@configuration注解的类，是否是一个@componet注解类，方法是@Bean注解方法。

4、标明是配置类，添加到集合中。如果集合为空，会直接返回。开始解析注解配置类。

5、判断是否还有自定义beanName的生成器。

6、创建解析对象，进行解析工作。

7、

### initMessageSource

为上下文初始化mesasge源，国际化处理。

### initApplicationEventMulticaster

### registerListeners注册监听器

## bean的实例化(finishBeanFactoryInitialization)

beanfactory属性设置

1、为上下文初始化类型转换器:做类型转换,将源类型转为目标类型。

2、beanfactory 注册默认的嵌入值处理器 。主要是做占位符的替换。

3、冻结所bean definiens ，不能修改了。

实例化:preInstantiateSingletons

1、遍历deanfdefinition数组，初始化factoryBean

2、将beanDefinition对象转换成成rootBeanDefinition,直接获取信息。

4、的获取到RootBeanDefinition,如果是有依赖的，先创建依赖的对象。

Beanfactory:通过该接口去是实例化bean对象。在spring的生命周期中会创建两个对象，第一个表示实现了factirybean接口的子类，由spring帮我们创建，回调用getObject方法返回一个对象。会把该对象放到factorybeanObjectcache中。

4、否则getbean-dogetbean()

1、从缓存中查询name的bean是否存在。

2、不存在，

类的加载:

1、获取类的加载器

2、从beandefinition中获取对应bean的限定名。

3、通过反射获取bean的class.

4、look-up overrite 。

Bean的注解扫描，

1、将@initannotatiotype和@destroyAnnotationType修饰的非静态方法,识别出修饰的方法，然后存放到该对象的lifecycleMetaData对象中。该方法会在initmethod阶段执行。例如: @postConstruct和predestroy，然后会将这些信息同步到beanDefinition。

@initannotatiotype:会提前将两种类型放进去。

2、还会处理@resource注解的信息，和上面一样的。

上面的注解都是jdk提供的。

3、@autowired和@value，修饰的属性也会找到，找到对饮的autowired的处理器，然后去解析，把解析的结果放到一个集合中，设置到beandefinines对象中。

## 初始化

### 属性填充(populateBean)

1、postProcessAfterInitialization执行方法,完成属性赋值工作。

1、从RootBeanDefinition. getPropertyValues获取到配置的name和value值。

2、获取注入模型(byname,bytype,construector,no 四种)

Byname:

1、筛选出引用属性值，非简单类型。

2、从缓存中获取对应引用对象，获取不到时，也会去创建该对象。

3、引用对象添加进新的PropertyValues

4、注册propertyname与bean那么的依赖关系。

Map:key对象name value:引用对象的name

Bytype:

1、获取自定义类型转换器，

2、筛选出引用属性值，非简单类型，遍历属性值。

有setter方法

非简单类型

4、注册propertyname与bean那么的依赖关系。

Map:key对象name value:引用对象的name

如果是map，可能会匹配多个值到map中，可能系统中其他属性注入进来。

3、将RootBeanDefinition.的PropertyValues 指向新的PropertyValues。

4、注解注入解析，扫描autowired的后置处理器，将参数传入处理器中，处理器会去容器中查询对应的bean，然后注入到对应的属性。这里会用深拷贝对象。返回的p-v也是深拷贝。

注:遇到找不到对象的时候，会通过递归的形式建造对象，并注入。

5、根据property标签定义的属性值，完成各种属性值的解析和赋值工作。

### 初始化方法1、调用BeanNameAware、BeanClassLoaderAware、BeanFactoryAware方法(如果当前bean对象实现了对应的接口。)

2、BeanPostProcessor

如果Bean实现了BeanPostProcessor接口，Spring就将调用他们的postProcessBeforeInitialization()方法。

3、如果Bean 实现了InitializingBean接口，Spring将调用他们的afterPropertiesSet()方法。类似的，如果bean使用init-method声明了初始化方法，该方法也会被调用。初始化后的最后方法。

Bena创建好后，可以对bean进行修改。

### Bean销毁逻辑

### 循环依赖过程

循环创建对象;

1、先去一级缓存singletonobjects中获取Abean对象。获取不到，下一步。

2、创建A对象，创建A对象放到二级缓存中。

3、将该A的benaName放到三级缓存中

Key:beanname value:

4、去设置属性B，B对象没有，需要创建B对象，B对象创建成功。需要填充属性。

5、将该B的beanname放到三级缓存中

Key:beanname value:

6、去设置B的属性A，创建A，先去缓存中去查找。

一级缓存中获取。

二级缓存中获取。

三级缓存中获取。获取A，获取到的是A的lambd表达式。

7、执行对应对的lambada表达式。获取到A的引用。将A的引用放到二级缓存中。

8、设置B中A属性的值。

9、将B对象放到一级缓存中，移除其他两个缓存中内容。

10、A对象属性B设置成功。删除A对象，在二级和三级缓存中的内容。

注:

二级缓存:在代理情况下,不能解决循环依赖。三级缓存中对象只能创建一份。

1、创建A对象

A:创建对象的时候会创建A对象和A的代理对象。

### 循环依赖

1、生成代理对象之前，要不要生成普通对象？要

2、我们能否确定什么时候贵调用具体的对象？无论是铍铜对象还是代理对象？ 不能。

3、有没有可能，在spring

4、这两个缓存，我优先获取哪一个。

优先获取代理对象，然后才是普通对象，错误的结论。

代理对象中应该包含了普通对象的所有功能，如果一个对象需要被代理了，name此时普通对象就可以不存在了，只要代理对象即可。

也就是说，在后续创建dialing对象之后，就可以代理对象来覆盖普通对象，普通对象就不存在了。

什么时候去调用对象？

我们在调用对象的时候，若果检测到当前对象需要被代理，name直接创建代理对象覆盖即可。

怎么能够在随时需要的时候创建代理对象呢?

传递进去一个生成代理对象的匿名内部类，当需要调用的时候，直接调用匿名内部类lambda生成代理的对象，就是类似于回调机制。

初始化环节

填充属性

执行aware对应的方法

执行beanpostprocessor中的before方法

执行init-method

执行beanpostprocessor中的after方法。

上述步骤执行完成之后是为了获取一个完整的成品对象，但是在初始化前我们能确定对象需要生成代理对象吗。

不能确定，而且我们三级缓存只是一个回调机制，多以能否能把所有的bean所需的创建代理对象的lambada表达式都放在三级缓存中。

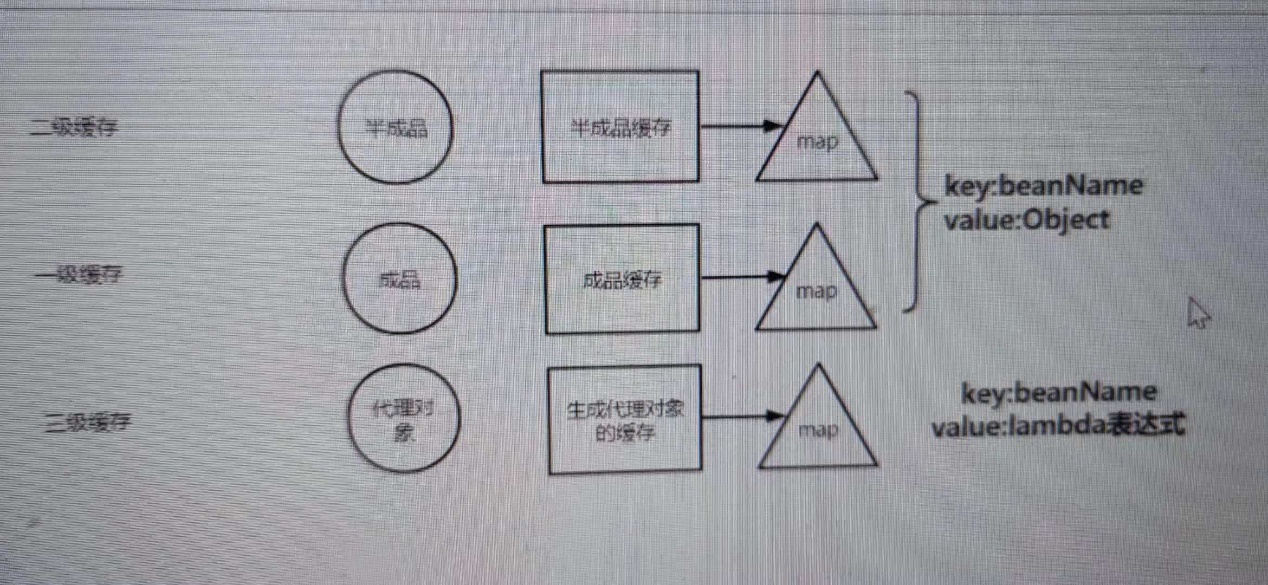
可以将所有的bean对象需要创建代理对象的lambda表达式放在三级缓存中，后续如果我需要调用，直接从三级缓存中执行即可，如果不需要，在生成玩着呢个对象之后可以把三级缓存中的lambda表达式清除掉，(往一级缓存中放成平对象的时候就可以亲清除)

6、我们什么时候哟啊生成具体的代理对象？

A :在进行属性注入的时候，调用该对象生成的时候检测是否需要被代理，如果需要，直接创建代理对象。

B：在整个过程中，没有其他的对象有当前对象的依赖，那么在生成最终的完整对象之前生成代理对象即可。Beanpostprocessor的after中。

### 三级缓存



1、提前暴露的对象:

二级缓存:实例化但是未初始化的。

2、三级缓存在进行对象查找的时候，顺序是什么样的。

1级缓存->二级缓存->三级缓存

3、可坑有三级缓存到一级缓存。

如果单纯了解决循环依赖的问题，name使用二级缓存足够解决问题，三级缓存的意义是为了代理，如果没有代理对象，耳机缓存就够了。

### 解决循环依赖

实话和初始化分开处理的，当完成实例化之后就可以让当前对象引用当前对象，只不过当前对象不是一个完整对象而已，后续需要完成对象的剩余步骤。

直接获取办成平对象的引用地址，保证对选哪个能够被找到，而版成平对象在堆空间是否有设置的属性值，无所谓。

## AOP

### Aop

1、查看配置文件的解析工作，在处理之后的benadefinition中包含什么信息?

2、对aop相关的beandefinition进行实例化操作。

在进行第一个对象创建之前，就必须把要把aop需要的相关对象提前准备好，因为无法预估那些对象需要动态的代理。

3、在那个步骤中可以提前实例化并且生成对应的对象。

BPP(beanpostand)

## Jdk动态代理原理

静态代理的思路，获得目标实例的代理对象一我们需要目标类的实例化对象，二需要一个代理类去接收该对象，获得对象方法，重写方法时在目标对象方法前后加点料（方法增强）。

1、扫描接口中的方法

1、创建set集合

2、获取接口的class对象集合。

3、遍历所有的集合class接口，然后添加到一个代理方法method对象的集合。

4、检测这些方法是否符合规则。

5、生成一个代理类生成器。

6、代理方法method对象的集合，放到代理类生成器中。

7、代理类生成器会生成一个字节码的数据。

8、通过classlodar读取字节码文件，生成代理类对象。

2、生成的代理类对象

1、调用add方法，执行执行代理类的方法，

2、调用父类Proxy. Invoke方法执行。

3、Invoke方法中执行的真正方法是通过反射加载的真正的方法。

|  |
| --- |
| **public final class** $Proxy0 **extends** Proxy **implements** Calculator {  **private static** Method *m3*;   **public** $Proxy0(InvocationHandler var1) **throws** {  **super**(var1);  }   **public final int** add(**int** var1, **int** var2) **throws** {  **try** {  //  **return** (Integer)**super**.**h**.invoke(**this**, *m3*, **new** Object[]{var1, var2});  } **catch** (RuntimeException | Error var4) {  **throw** var4;  } **catch** (Throwable var5) {  **throw new** UndeclaredThrowableException(var5);  }  }    **static** {  **try** {  *m3* = Class.*forName*(**"com.imooc.proxy.jdk.Calculator"**).getMethod(**"add"**, Integer.***TYPE***, Integer.***TYPE***);  } **catch** (NoSuchMethodException var2) {  **throw new** NoSuchMethodError(var2.getMessage());  } **catch** (ClassNotFoundException var3) {  **throw new** NoClassDefFoundError(var3.getMessage());  }  } } |

### Aop

1、AOP的实现，最关键的有两步：

1、得到代理对象

2、利用递归责任链执行前后置通知及目标方法