## Spring源码编译

环境准备

IDEA 2019版

JDK1.8以上

Gradle设置成这样



1、下载spring源码https://github.com/spring-projects/spring-framework

2、解压

3、打开idea，file-new-project from existing sources..

4、选择spring源码，用gradle进行编译

5、新建一个Gradle的java项目，build.gradle添加compile(project(":spring-context"))这一行代码

dependencies {

compile(project(":spring-context"))

testCompile group: 'junit', name: 'junit', version: '4.12'

}

6、右键运行org.springframework



7、双击compile.java或者compileText.java



## Spring的自动注入

### 名词解释

Spring 1.0 XML时代

2.5 Annotation注解时代

3.0 java config

AnnotationConifigApplicationContext：注解启动Spring

ClassPathXmlApplicationContext：XML启动Spring

Springboot自动配置 != Spring自动装配（Spring自动注入）

注入模型是基于自动注入（自动注入只针对于XML），注入方式是基于手动注入

### IOC（Inversion of control 控制反转）和DI（Dependecy Injection 依赖注入）的区别

IOC只是DI的另外一种说法，他们两个原则上是一个概念，一个叫控制反转，一个叫依赖注入

### 单例和原型

单例：多次调用只会实例化一次。Singleton

如果单例对象方法想要有原型的效果，看Spring官网1.4.6章节

A类

@Lookup  
**public** XBServiceprototype(){  
**return null**;  
};  
  
**public void** xx(){  
XBService x = prototype();  
System.***out***.println(x.hashCode());  
}

XBService类要带上@Scope(**"prototype"**)

原型：调用几次就会实例化几次。Prototype

@Scope(**"prototype"**)

Public class XBService

### Spring有2种注入方式（注入方式只有手动）

看Spring官网1.4.6章节

1、提供一个构造方法

<**bean id="familyService" class="com.xiaobi.service.FamilyService"**>  
*<!-- 构造器注入 -->  
<!-- 构造器注入和Set注入不能在同时存在 -->*<**constructor-argref="xbBean"**></**constructor-arg**>  
</**bean**>

<**bean id="xbBean" class="com.xiaobi.service.XBService"**></**bean**>

**public** FamilyService(XBServicexbService){  
**this**.**xbService**= xbService;  
System.***out***.println(**"print Family2"**);  
}

2、提供对象的set方法(XML配置里面的<property name="asd">这里的asd是注入对象的set方法（setasd）)

<**bean id="familyService" class="com.xiaobi.service.FamilyService"**>  
*<!-- Set注入 -->*<**property name="XBService"**>  
<**ref bean="xbBean"**></**ref**>  
</**property**>

</**bean**>

<**bean id="xbBean" class="com.xiaobi.service.XBService"**></**bean**>

**public void** setXBService(XBServicexbService) {  
**this**.**xbService**= xbService;  
}

3、这种意义上不能完全是注入方式。他先实现ApplicationContextAware接口，重写createCommand后，在将对象放到Spring中(applicationContext.getBean(“name”,Class对象))

### Spring有4种自动注入模型

XML（default-autowire=”bytype”）修改默认注入模型

1、No（默认）autowireMode = 0

2、byname（要提供set方法，和setXXX这里面的XXX有关，Spring会根据XXX这个去Spring容器找到对应的对象当作传入的参数值）autowireMode = 1

<bean id = “xx” class=””autowire=”byname”></bean>单独修改自动注入模型

3、bytype（必须提供set方法，和setXXX这里面的XXX无关，传入的参数如果在Spring容器中，调用set方法，否则不调用set方法）autowireMode = 2

如果多个子类继承同一个接口，此时如果用bytype会报错，他会提示单个匹配Bean，找到多个（expected single matching bean but found 2:zl1 zl2）

**default-autowire="byType"**

4、constructor（推断构造方法，解析构造方法，通过构造方法找bean，找参数，Spring在容器中找参数最多的构造方法）autowireMode = 3

知识点1：@Autowired这种不能说是自动注入，可以理解为他是手动注入，因为他已经指定了要注入的java对象。他先根据“类型”找，在根据“名称”找到要注入的java对象，这里的“类型”不等于bytype，“名称”不等于byname（参考源码：doCreateBean.populateBean.autowireByType.resolveDependency.doResolveDependency.findAutowireCandidates(beanName, type, descriptor)）。这种注入java对象只针对于注解方式，并且只作用在java对象上面，并且需要依赖的类必须加上@Compoent注解。

XbService1、XbService2继承Xb

XbxbService1：此时注入成功。因为先根据类型Xb找，找到多个，在根据名称xbService1找，找到XbService1对象。如果Class名称不符合规范，比如XBService1，则需要XbXBService1才可以注入成功。

XbxbService3：此时不会报错。因为没有一个名称是xbService3的对象。

知识点2：4种自动注入模型的注入方式：setter注入或者constructor注入

@Autowired的注入方式：field.set反射机制注入，源代码在：AutowiredFieldElement.inject或者AutowiredMethodElement.inject

知识点3：autowireMode是bean definition的一个属性，这个属性是标识是什么类型的自动注入的模型，这个属性可以通过GenericBeanDefinition.getAutowireMode得到对象的注入模型

知识点4：@Autowired方式注入，属性和方法都不能用static修饰

### @Atuowired和@Resource区别

@AutowiredSpring包提供的注解，他是由AutowireAnnotationBeanPostProcessor这个后置处理器解析的，他处理的逻辑是先根据“类型”找，再根据“名称”找。

@ResourceJDK提供的注解，他是有CommonAnnotationBeanPostProcessor这个后置处理器解析的，他处理的逻辑是先根据“名称”找，再根据“类型”找。

### 生命周期回调

#### Bean的生命周期回调

Bean的生命周期回调分为Bean的初始化生命周期回调和Bean的销毁生命周期回调(看Spring官网1.6.1章节Destruction Callbacks)

实现Bean的初始化生命周期回调有3种方法（Spring初始化方法由1个后置处理器和2个方法），看Spring官网1.6.1章节Initialization Callbacks

1. 注解@PostConstruct
2. 实现**implementsInitializingBean**重写**afterPropertiesSet方法**
3. XML配置<**bean init-method="liftAfterInit" id="familyService" class="com.xiaobi.service.FamilyService"**>

考点1：3种方法同时存在的先后顺序是：1 2 3

initializeBean中的applyBeanPostProcessorsBeforeInitialization完成注解@PostConstruct的生命周期回调

initializeBean中的invokeInitMethods完成XML和接口的生命周期回调

applyBeanPostProcessorsBeforeInitialization代码在invokeInitMethods前面。

考点2：多个实现生命周期回调指向同一个方法，只会执行一次

**public** FamilyService(){  
System.***out***.println(**"print init"**);  
}  
  
@PostConstruct  
**public void** liftAfterInit(){  
System.***out***.println(**"初始化之后立马回调"**);  
}

#### Bean的生命周期销毁回调

1、@PreDestory注解

2、继承BeanFactoryPostProcessor

BeanDefinitionbeanDefinition = beanFactory.getBeanDefinition(**"familyService"**);  
beanDefinition.setDestroyMethodName(AbstractBeanDefinition.***INFER\_METHOD***);

当容器关闭(ac.close)的时候，就会调用这个方法，方法二有局限性，他必须要有个close和shutdown的方法

#### 容器的生命周期回调

看Spring官网1.6.1章节Startup and Shutdown Callbacks

1、继承Lifecycle，重写start() 容器启动,stop() 容器停止,isRunning() 容器是否正在运行

2、继承SmartLifecycle（public interface SmartLifecycle extends Lifecycle, Phased），继续重写，重写isAutoStartup()，stop(),getPhase()

isAutoStartup()：显式调用。如果设置为false,也就是通过ac.start()进行启动，true，容器启动自动调用start()方法。

stop(Runnable callback)：如果SmartLifecycle.Stop和Lifecycle.Stop方法同时存在，只调用SmartLifecycle.Stop方法。如果使用ac.stop停止，调用stop方法之后等待30秒停止应用（callback.run()方法可以立即停止），如果是web项目停止，则会调用，因为web项目停止服务器有个钩子概念，服务器会在停止前调用stop方法。

getPhase()：设置优先级。多个类继承SmartLifecycle，启动功能执行的先后顺序，值越小，越先执行，停止的时候是相反的。

容器的生命周期回调在Bean生命周期回调之后

### 提高Spring启动速度

看Spring官网1.10.9章节 Generating an Index of Candidate Components

加入spring-context-indexer包

### 对象放到Spring容器

3种方法

1. 注解@Bean（可以控制对象的产生过程），这种是程序员内部为了引配置而使用的，比如：链接数据库需要Bean对象，此时就不用使用XML配置
2. 继承接口FactoryBean，实现里面的方法，一般是和第三方jar做结合使用，需要注入到Spring容器中，对Spring做二次扩展用到的，这种注入方式可以不用在类上面加@Component，Spring是可以启动起来（@Autowired这种，如果在对象类上没有带@Component，Spring启动就会报错）。

ac.getBean("indexBean")[com.xiaobi.bean.UserBean@6ee52dcd](mailto:com.xiaobi.bean.UserBean@6ee52dcd)返回getObject里面的对象

ac.getBean("&indexBean")[com.xiaobi.bean.IndexBean@4493d195](mailto:com.xiaobi.bean.IndexBean@4493d195)返回自己

1. ac.getBeanFactory().registerSingleton(**"indexBean"**, IndexBean.**class**);注册多个数据源的时候，可以使用这个方法。

这种方法只能把自己注入对象到Spring容器中，意义不是很大，并且有很大的局限性：

1. 在容器初始化完前注入，依赖对象不在容器中，无法完成依赖注入，他只能完成另一个对象依赖indexBean的注入，indexBean依赖其他是无法完成的。
2. 在容器初始化完后注入，依赖对象也不在容器中，需要自己维护关系后注入，才能完成依赖注入，并且另一个对象需要依赖indexBean是依赖不上的，因为他不在Spring容器中。

将对象放到Spring(FactoryBean)和将类(@Component)放到Spring的区别

对象放到Spring：对象产生的过程，程序员是可以控制的。Mybatis就是这样实现

类放到Spring：对象产生的过程，程序员无法控制。

## Spring循环依赖

https://blog.csdn.net/java\_lyvee/article/details/101793774

### 对象和bean的区别

对象：只要类被实例化就可以称为对象，java管理的数据

bean：首先得是一个对象，然后这个对象需要经历一系列的bean生命周期，最后把这个对象put到单例池才能算一个bean，Spring管理的数据

### Spring循环依赖大概流程

循环依赖开始

1、启动容器

2、第一次getBean(indexService)，第一次调用getSingleton方法（这个方法可以简单理解为从单例池里面获取bean，见getSingleton(beanName)方法逻辑）

3、第一次调用getSingleton方法为空，因为indexService不在创建过程中，所以代码继续执行

4、第二次调用getSingleton方法（这2次方法调用同名不同参，逻辑也不同），会调用createBean方法，这个方法会调用doCreateBean方法（doCreateBean方法主要作用是把当前正在创建的类记录到singletonsCurrentlyInCreation这个set集合，完成依赖注入和初始化方法）。此时将IndexService放到set集合中，标识他正在创建过程中，放到singletonFactories（二级缓存），实例化对象，进行依赖注入，在注入的过程中发现需要注入userService

5、第一次getBean(userService)，在经历2次调用getSingleton方法，实例化对象，进行依赖注入，在注入的过程中发现需要注入indexService

6、第二次getBean(indexService)，在第一次调用getSingleton方法后，从singletonFactories（二级缓存）中获取到Bean工厂，因为indexService在第一次getBean(indexService)的第二次调用getSingleton方法后已经标识创建过程中，所以能获取到数据，并返回出去

7、完成第二次的indexService注入。

8、UserService对象的依赖注入完成，但此时UserService还只是一个对象，他不是Bean，IndexService已经是对象也是Bean

9、完成第一次userService注入

10、循环依赖完成。

1. 启动容器
2. 第一次getBean(indexService)
3. 此时Spring会判断单例缓存池（singletonObjects）里面是否存在这个indexService Bean
4. 如果存在直接返回Bean，如果不存在则会判断IndexService对象是否正在创建过程中，如果IndexService对象在创建过程中，则会通过第三个Map中获取indexService对象，否则执行下一步
5. 单例缓存池没有这个indexService Bean也不在创建过程中
6. 再一次调用getSingleton（第一次调用getSingleton和这一次调用getSingleton属于同名不同参数，方法重载）把IndexService对象放到singletonsCurrentlyInCreation这个set集合中（这个集合是为了后面再次需要getBean(indexService)的时候，给IndexService对象做一个正在创建标志）
7. 创建一个IndexService对象
8. 把IndexService这个类的BeanFactory（Spring扫描所有类都会分派到不同的BeanFactory里面）放到另外一个集合中（这个集合是为了后面获取到Bean，Spring就可以通过解析BeanFactory后获取Bean）
9. 创建完IndexService对象之后Spring会进行依赖注入
10. 发现需要注入userService
11. 第一次getBean(userService)
12. Spring会执行UserService从3-9的过程
13. 单例缓存池没有这个userService Bean也不在创建过程中
14. 给一个正在创建标志
15. Spring创建一个UserService对象
16. BeanFactory放到二级缓存集合中
17. 创建完之后Spring会依赖注入，发现需要注入indexService
18. 第二次getBean(indexService)
19. indexService不在单例缓存池，但是他在创建过程中
20. Spring就会从第三个缓存池取出indexService对象注入到userService
21. 此时userService只是一个对象还不是bean
22. 走完userService生命周期，将userServicebeanput到单例池中
23. 此时userService是一个对象也是一个bean
24. 完成userService填充
25. 返回到indexService
26. 此时indexService只是一个对象，还不是bean
27. 走完indexService生命周期，将indexServicebeanput到单例池中
28. 此时indexService是一个对象也是一个bean
29. 完成indexService填充
30. 循环依赖完成
31. getBean(Index)
32. 第一次调用getSingleton方法
33. 判断是否需要从第三个缓存中拿
34. 拿不到
35. 再一次调用getSingleton通过beforeSingletonCreation往singletonsCurrentlyInCreation这个set集合中添加一条数据，标识index为正在创建过程中，第一次调用getSingleton和这一次调用getSingleton属于同名不同参数，方法重载
36. Lambda表达式通过singletonFactory.getObject()去调用createBean方法
37. createBean实例化Index
38. createBeanInstance通过反射去创建Index对象或者是代理对象，并存放到wrappedObject里面
39. instanceWrapper.getWrappedInstance()去获取wrappedObject对象或者代理对象
40. 如果允许循环依赖，addSingletonFactory将index工厂放到二级缓存中，如果原对象需要代理，在这一步就会提前AOP代理
41. index工厂怎么产生？getEarlyBeanReference方法调用后置处理器产生
42. 为什么要存放index工厂？因为index工厂可以存放index代理对象或者原对象。
43. 什么时候需要用代理原对象?比如在使用@lookup注解的时候，就会将index对象改为代理对象
44. populateBean属性注入
45. Index对象注入user
46. getBean（user）
47. 第一次调用getSingleton方法
48. 判断是否需要从第三个缓存中拿
49. 拿不到
50. 标识User为正在创建过程中
51. 实例化User
52. 获取到User对象
53. 如果允许循环依赖，将user工厂放到二级缓存中
54. 走User生命周期
55. User对象需要注入Index
56. getBean(Index)
57. 判断index bean是否为空，并且是创建过程中，从二级缓存中获取index对象
58. index对象不等于空 set给user注入
59. User完成populateBean这个方法，属性注入完成，递归返回到Index对象的populateBean方法里面
60. Index完成populateBean这个方法，属性注入完成

### 第一次调用getSingleton(beanName)方法逻辑

去单例池singletonObjects（一级缓存）中获取，如果获取不到，判断indexService对象是否在创建过程中，如果在创建过程中，会在earlySingletonObjects（三级缓存）中获取到对象，获取不到，会去singletonFactories（二级缓存）中获取对象(此时第二个缓存绝对可以获取到对象)，放到earlySingletonObjects里面（为了避免重复singletonFactories.get(beanName)，浪费性能），并且移除singletonFactories中的数据（为了减少数据，提高gc性能）。如果找到就直接返回。这个方法主要的作用是获取bean对象。

第一次调用：getSingleton(String beanName, booleanallowEarlyReference)

### 第二次调用getSingleton(beanName)方法

这个方法主要作用是：把当前正在创建的类记录到set集合，功能主要作用在createBean里面。

sharedInstance = getSingleton(beanName, () -> {

try {

//完成了目标对象的创建

//如果需要代理，还完成了代理

return createBean(beanName, mbd, args);

}

catch (BeansException ex) {

destroySingleton(beanName);

throw ex;

}

});

getSingleton(String beanName, ObjectFactory<?>singletonFactory)

### 循环依赖的三级Map缓存区别

一级缓存：singletonObjects熟称单例池，这个缓存是用来存放所有单例Bean，不是对象。对象和Bean的区别见”对象和bean的区别”。

二级缓存：singletonFactories，这个缓存是用来存放所有单例对象工厂，这个单例对象工厂的作用是对对象进行二次加工后得到的一个类，Spring可以通过这个对象工厂获取到对象信息。

三级缓存：earlySingletonObjects，这个缓存是用来存放二级缓存通过对象工厂获取到对象的信息。三级缓存和一级缓存的区别在于，三级缓存存放的是对象，一级缓存存放的是Bean。

### Spring生命周期流程

#### 公开课

1. 初始化开始。
2. Scan扫描BeanDefinition。
3. parse解析。
4. 后置处理器调用扩展。
5. 将所有BeanDefinition子类放入map。
6. 遍历mapvalidate解析验证。
7. 实例化对象，调用构造方法（newindex）
8. 属性注入开始，也称依赖注入（生成Spring bean，这只是属性注入其中的一个过程），此时还只是一个目标对象。
9. 提前暴露aop代理（initializeBean中的applyBeanPostProcessorsAfterInitialization这个方法完成代理，具体是由AnnotationAwareAspectAutoProxyCreator来改变对象，变成代理对象）
10. 属性注入结束，注入代理对象
11. 生命周期回调（init生命周期初始化方法）
12. 注入到单例池
13. 初始化结束。

invokeBeanFactoryPostProcessors(beanFactory)完成扫描，2-5步骤

finishBeanFactoryInitialization(beanFactory)Spring完成实例化bean的过程，完成Spring生命周期，6-9步骤

1. new
2. init生命周期初始化方法
3. 注入
4. 代理aop
5. 注入到单例池

#### VIP

1：实例化一个ApplicationContext的对象；

2：调用bean工厂后置处理器完成扫描；

3：循环解析扫描出来的类信息；

4：实例化一个BeanDefinition对象来存储解析出来的信息；

5：把实例化好的beanDefinition对象put到beanDefinitionMap当中缓存起来，以便后面实例化bean；

6：再次调用bean工厂后置处理器；

7：当然spring还会干很多事情，比如国际化，比如注册BeanPostProcessor等等，如果我们只关心如何实例化一个bean的话那么这一步就是spring调用finishBeanFactoryInitialization方法来实例化单例的bean，实例化之前spring要做验证，需要遍历所有扫描出来的类，依次判断这个bean是否Lazy，是否prototype，是否abstract等等；

8：如果验证完成spring在实例化一个bean之前需要推断构造方法，因为spring实例化对象是通过构造方法反射，故而需要知道用哪个构造方法；

9：推断完构造方法之后spring调用构造方法反射实例化一个对象；注意我这里说的是对象、对象、对象；这个时候对象已经实例化出来了，但是并不是一个完整的bean，最简单的体现是这个时候实例化出来的对象属性是没有注入，所以不是一个完整的bean；

10：spring处理合并后的beanDefinition(合并？是spring当中非常重要的一块内容，后面的文章我会分析)；

11：判断是否支持循环依赖，如果支持则提前把一个工厂存入singletonFactories——map；

12：判断是否需要完成属性注入

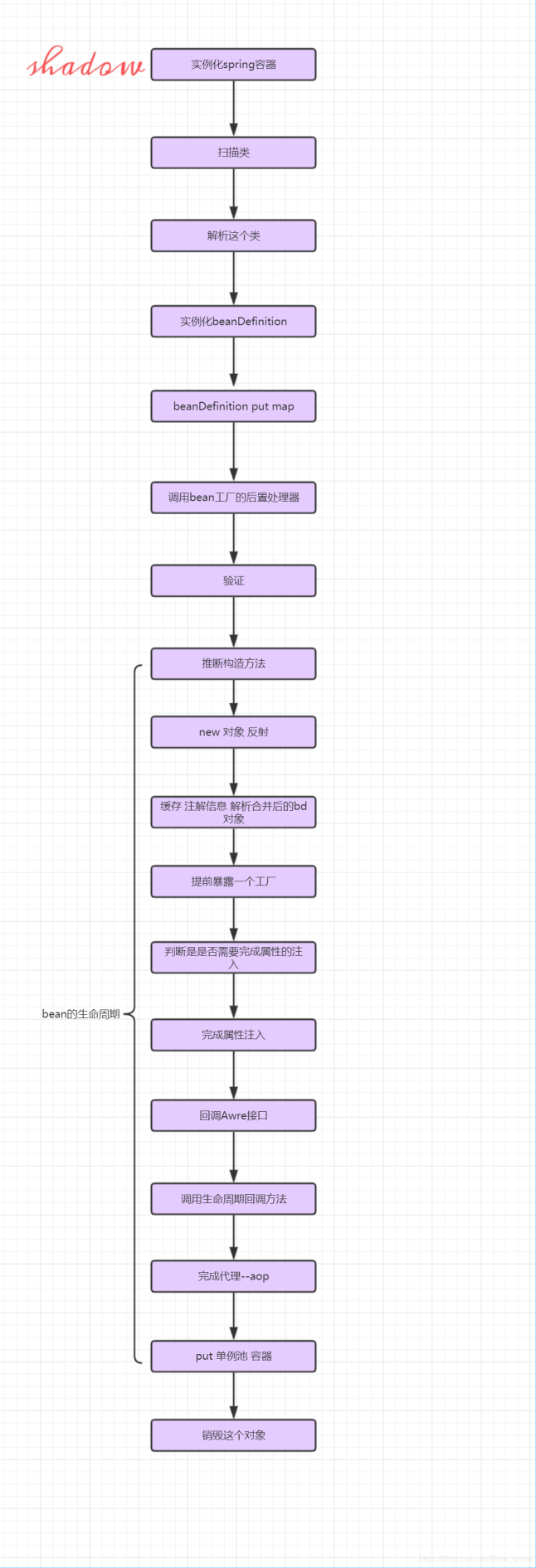
13：如果需要完成属性注入，则开始注入属性

14：判断bean的类型回调Aware接口

15：调用生命周期回调方法

16：如果需要代理则完成代理

17：put到单例池——bean完成——存在spring容器当中



### Spring和类实例化过程区别

类实例化过程：

.java文件被javac编译成.class文件之后，运行main方法，main方法会调用C++代码去启动JVM虚拟机，JVM虚拟机会去磁盘上把刚刚编译好的.class文件加载到JVM的内存当中（方法区），当main方法里面遇到new关键字之后，会根据方法区里面的提供的java模版模版去堆上分配一块内存去存储这个对象

Spring产生Bean过程：

Scan扫描 🡪 parse解析 🡪调用扩展🡪遍历mapvalidate解析验证🡪new

.java文件会被classLoader（类加载器）编译成.class文件，让后将所有的.class文件给Spring管理，此时Spring会循环这些.class文件,他会按照一定的规则用BeanDefinition（BeanDefinition是一个接口，里面存放对象的属性，比如：类的名字，类的上级，是否懒加载，是否是抽象，DependsOn注解内容是什么，Description注解内容是什么）的子类来进行存放这些.class文件内容信息，方便后面Spring的识别调用，然后将这些BeanDefinition的子类存放到BeanDefinitionMap这个Map集合中（实现BeanFactoryPostProcessor.postProcessBeanFactory这个方法，可以修改Map集合中的这些BeanDefinition子类），然后Spring会解析这个Map集合，将符合规则（比如：没有加注解的类，原型类，懒加载类等这些都不会放到里面）的BeanDefinition子类获取他的Bean放到singletonObjects单例池中。

### 关闭Spring循环依赖

AnnotationConfigApplicationContext ac =  
**new** AnnotationConfigApplicationContext();

//关闭循环依赖  
ac.setAllowCircularReferences(**false**);  
ac.register(Appconfig.**class**);  
ac.refresh();

如何说明Spring是默认支持循环依赖：

Spring在初始化Spring bean过程中，当实例化bean之后，会判断当前容器允不允许循环依赖，判断循环依赖是根据allowCircularReferences这个属性来判断的，allowCircularReferences默认为true

循环依赖只支持属性的循环依赖，构造方法不支持

### BeanFactory、BeanDefinitionMap、BeanDefinition、单例池关系

BeanFactory在什么时候实例化：BeanFactory其实就是DefaultListableBeanFactory这个类，他在容器启动调用this()方法，会默认调用他父类GenericApplicationContext的无参构造方法，自动new一个DefaultListableBeanFactory对象，后面的功能都是初始化BeanFactory。

**public** GenericApplicationContext() {  
**this**.**beanFactory**= **new** DefaultListableBeanFactory();  
}

BeanFactory包含BeanDefinitionMap包含BeanDefinition

BeanDefinitionMap将符合规则的Bean注入到单例池

BeanFactory：singletonFactory.getObject()==》getEarlyBeanReference(beanName, mbd, bean)

protected Object getEarlyBeanReference(String beanName, RootBeanDefinitionmbd, Object bean) {

Object exposedObject = bean;

if (!mbd.isSynthetic() &&hasInstantiationAwareBeanPostProcessors()) {

//循环Spring所有内置处理器

for (BeanPostProcessorbp :getBeanPostProcessors()) {

if (bp instanceofSmartInstantiationAwareBeanPostProcessor) {

SmartInstantiationAwareBeanPostProcessoribp = (SmartInstantiationAwareBeanPostProcessor) bp;

exposedObject = ibp.getEarlyBeanReference(exposedObject, beanName);

}

}

}

return exposedObject;

}

### Spring容器

在Spring当中有很多组件，比如：BeanFactory、BeanDefinitionMap、BeanDefinition、三级Map缓存，后置处理器等这些组件组成叫做Spring容器（Spring组件的集合）。

## AOP



https://shimo.im/docs/Nj0bcFUy3SYyYnbI/read

### AOP概念

（1）Aspect：切面，类。

@Aspect  
**class** FamilyAspect{}

（2）Join point：连接点，表达式。可以认为这就是一个连接点。

**execution(\* com.xiaobi.service..\*(..))**

（3）Pointcut：切点，某一类型连接点的集合，通过表达式匹配到类方法的集合，可以认为这就是切点。

@Pointcut(**"execution(\* com.xiaobi.service..\*(..))"**)

（4）Advice：通知，逻辑和执行时机（在逻辑之前还是之后）

@Before(**"anyPublicMethod()"**)  
**public void** doAccessCheck(){  
System.***out***.println(**"AOP：指定包下面，在所有方法执行前调用这个方法"**);  
}

通知有5类通知。

1. Before advice：方法返回之前执行
2. After returning advice：方法返回之后执行
3. After throwing advice：出异常之前执行
4. After (finally) advice：finally之前执行
5. Around advice：环绕通知

（5）weaving：织入，在不同执行时机执行逻辑的过程叫织入。

（6）target：目标对象

（7）proxy：代理对象

### SpringAOP和Aspectj区别

AOP是编程标准

SpringAOP：是Spring实现AOP的手段，他是动态代理，他的代理对象在运行阶段产生的，Spring参考了Aspectj的语法风格去实现动态代理，他有2种开启Aspectj的支持，一种是XML（<aop:aspectj-autoproxy/>），一种是注解（@EnableAspectJAutoProxy）。

Aspectj：也是一个实现AOP的手段，他的代理对象在编译（ctrl+s保存）时产生的。

### 切点的设计方案

Spring官网5.4.3章节

9种设计方案：Execution、within 、this 、target 、args、@target 、@args、@within 、@annotation

Execution：能匹配到方法

Within：只能匹配到类

args：只能匹配参数

this：代理对象

target：目标对象

@within：拦截加了注解的类

切点的统一表达式：execution(modifiers-pattern?ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) throws-pattern?)

modifiers-pattern：方法修饰符public/private/protected，可有可无，没有就都匹配

ret-type-pattern：返回类型，全部就用\*表示

declaring-type-pattern：声明类型，包或者类

name-pattern：方法名字，全部就用\*表示

param-pattern：参数，全部就用点点

throws-pattern：抛出的类型，没有就都匹配

### JDK动态代理和CGLib动态代理

Spring默认使用JDK动态代理技术(Proxy.newProxyInstance)去做AOP代理，他能代理任意的接口。如果业务对象未实现接口（继承类），则使用CGLib动态代理（enhancer.create），Spring可以强制使用CGLib代理。@EnableAspectJAutoProxy(proxyTargetClass = **true**)

JDK动态代理：产生代理时候效率低，执行时快。

CGLib动态代理：产生代理时候效率高，执行时慢。

### Aspectj快速启动AOP

Spring官网5.4章节

拦截指定包下面的所有方法

第一步：[添加@EnableAspectJAutoProxy注解，引入aspectjweaver.jar](mailto:添加@EnableAspectJAutoProxy注解，引入aspectjweaver.jar)包

@Configuration

@EnableAspectJAutoProxy

public class AppConfig {}

第二步：添加@Aspect注解，申明一个切面

第三步：在方法上方添加@Pointcut("execution(\* com.xiaobi.service..(..))"**)**切点

第四步：[通知指向切点方法@Before("anyPublicOperation()")](mailto:通知指向切点方法@Before(%22com.xyz.myapp.SystemArchitecture.dataAccessOperation()%22))

@Compoent

@Aspect

public class NotVeryUsefulAspect {

@Pointcut(**"execution(\* com.xiaobi.service..\*(..))"**)

public void anyPublicOperation(){}

@Before("anyPublicOperation()")

public void doAccessCheck(){}

}

### 循环依赖AOP代理和普通AOP代理

循环依赖AOP代理：在doCreateBean的getEarlyBeanReference里面完成代理对象创建，具体是AbstractAutoProxyCreator.getEarlyBeanReference.wrapIfNecessary. createProxy.getProxy.createAopProxy(DefaultAopProxyFactory.createAopProxy)这个方法完成，方法的调用在第一次getSingleton的singletonObject = singletonFactory.getObject();完成doCreateBean的getEarlyBeanReference的方法调用。

普通AOP代理：doCreateBean.initializeBean的applyBeanPostProcessorsAfterInitialization里面完成，具体是AbstractAutoProxyCreator.postProcessAfterInitialization.wrapIfNecessary完成代理对象创建。

### 冷知识

1. 被AOP作用过的所有对象都已成为代理对象
2. 完成AOP代理是在initializeBean中的applyBeanPostProcessorsAfterInitialization这个方法完成代理，具体是由AnnotationAwareAspectAutoProxyCreator这个后置处理器来改变对象，变成代理对象，但是循环依赖的时候，AOP代理功能会被Spring在添加工厂到singletonFactories二级缓存中的时候提前做了。

## BeanDefinition

自定义注解

Public @interface XB

用来描述SpringBean的一个类





### setAutowireCandidate和setPrimary

FamilyService注入XB这个接口，XBService和XBService1分别实现这个接口，设置默认注入模型为byType。此时不会报错，会注入空值到属性里面。但这种情况可以通过设置setAutowireCandidate和setPrimary来解决。

setAutowireCandidate：设置是否参与自动装配的候选对象。false不参与 true参与。

setPrimary：设置是否是主要参与的候选对象。

比如：

default-autowire="byType"

<bean id="familyService" class="com.xiaobi.service.FamilyService"></bean>  
<bean id="xbService1" primary="true" class="com.xiaobi.service.XBService"></bean>  
<bean id="xbService2" autowire-candidate="false" class="com.xiaobi.service.XBService1"></bean>

### XML的配置都会有一个BeanDefinition与之对应的api

GenericBeanDefinitiongenericBeanDefinition = **new** GenericBeanDefinition();  
genericBeanDefinition.setAutowireMode(2);  
genericBeanDefinition.setBeanClass(XBService1.**class**);  
genericBeanDefinition.setScope(**"singleton"**);  
ac.registerBeanDefinition(**"xb1"**,genericBeanDefinition);

### AttributeAccessor和BeanMetadataElement

AttributeAccessor：主要的功能是为BeanDefinition的属性提供额外存放地方，比如：我想存一个长度。setAttribute

BeanMetadataElement：只有一个接口getSource，获取类的元，也就是class的文件地址。

AbstractBeanDefinition：继承AttributeAccessor和BeanMetadataElement接口。

### ConstructorArgumentValues

将ZL0传给XBService构造方法name1里面，index是构造方法参数的序号，下标从0开始。

<**bean id="xbService" class="com.xiaobi.service.XBService"**>  
<**constructor-argvalue="ZL0" index="0"**></**constructor-arg**>  
<**constructor-argvalue="ZL1" index="1"**></**constructor-arg**>  
<**constructor-argvalue="ZL2" index="2"**></**constructor-arg**>  
</**bean**>

**public** XBService(String name1,String name2,String name3){}

### BeanDefinition子类



(1)RootBeanDefinition：可以是个独立的bd，他只能做最上级的父bd，不能单做子bd

(2)ChildBeanDefinition：不能作为独立的bd，他必须指向一个父bd，不能做最上级的父bd

(3)GenericBeanDefinition：bd通用类,可以是个独立的bd，同时可以做父或子bd，他可以完全代替child，但是他不能完全代替root，因为spring合并最上级的子bd时候需要有个不同于子bd的bd来接收，也就是generic合并，不能用generic接收，Sprign合并的时候（merge）规定只能用root来接收子bd的合并(RootBeanDefinitionmbd = getMergedLocalBeanDefinition(beanName))，所以不能代替root。

(4)AbstractBeanDefinition：bd的模版，用于子类继承的一个公共抽象类

(5)AnnotatedGenericBeanDefinition：解析加了@Configuration

配置类，也可以解析GenericBeanDefinition类，因为他是GenericBeanDefinition子类。

(6)ScannedGenericBeanDefinition：这个bd是为了解析加了注解的类，和AnnotatedGenericBeanDefinition功能大部分相同，他们的区别如下：

1、ScannedGenericBeanDefinition类提供了getFactoryMethodMetadata这个方法，这个方法返回空，因为注解是没有FactoryMethod方法，只有XML才有。

<**bean id="xbService" class="com.xiaobi.service.XBService" factory-method="aspectOf"**>

AnnotatedGenericBeanDefinition这个是扫描配置类，配置类有时候需要扫描XML，所以他可以获得到FactoryMethod（这个方法的作用是必须实例化工厂类（factory-bean）后才能调用工厂方法）指向的方法。

2、AnnotatedGenericBeanDefinition扫描出来的XML没有注解信息，但是他需要将一些信息存放AnnotationMetadata到这里面，但是这个类存放的是注解信息，所以他实例化了一个StandardAnnotationMetadata，变相存放XML信息到AnnotationMetadata里面。

ScannedGenericBeanDefinition因为是扫描注解的bd，所以他可以直接将注解信息存放到AnnotationMetadata里面。

(7)ConfigurationClassBeanDefinition：：解析加了@Bean注解的配置类，也可以解析GenericBeanDefinition类，因为他是GenericBeanDefinition子类，加了@Bean注解的bd他的BeanClassName值为空，因为返回的类型可以指定一个接口，返回的时候返回接口的子类ac.getBeanDefinition("a").getBeanClassName()。

## 后置处理器

扫描、new之前、推断构造方法、new完成、判断是否支持循环依赖、合并、判断属性是否需要注入、属性注入、初始化之前、invoke init method（初始化方法）、aop

这些都用到了后置处理器

### Bean工厂后置处理器

实现顶级BeanFactoryPostProcessor的类都是Bean工程后置处理器，ConfigurationClassPostProcessor这个是常用类。

### Bean后置处理器

BeanPostProcessor：干预Bean初始化和实例化的过程

### 内置后置处理器

#### ApplicationContextAwareProcessor

AbstractApplicationContext.prepareBeanFactory方法里面beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));

#### ConfigurationClassPostProcessor$ImportAwareBeanPostProcessor

PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors循环处理"所有"的BeanDefinitionRegistryPostProcessor.BeanFactoryPostProcessor.postProcessBeanFactory方法，ConfigurationClassPostProcessor后置处理器的方法postProcessBeanFactory

beanFactory.addBeanPostProcessor(new ImportAwareBeanPostProcessor(beanFactory));

#### PostProcessorRegistrationDelegate$BeanPostProcessorChecker

AbstractApplicationContext.registerBeanPostProcessors方法里面的beanFactory.addBeanPostProcessor(new BeanPostProcessorChecker(beanFactory, beanProcessorTargetCount));

#### AnnotationAwareAspectJAutoProxyCreator

解析@EnableAspectJAutoProxy的@Import(AspectJAutoProxyRegistrar.class)注入的一个后置处理器，invokeBeanFactoryPostProcessors.invokeBeanDefinitionRegistryPostProcessors方法里面的ConfigurationClassParser.processImports完成扫描@Import注解，参考mybatis旧对象注入bdMap集合中

#### CommonAnnotationBeanPostProcessor

处理reader的时候生成的，主要是处理@Resource

#### AutowiredAnnotationBeanPostProcessor

处理reader的时候生成的，主要是处理@Atuowired

#### ApplicationListenerDetector

AbstractApplicationContext.prepareBeanFactory方法里面beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(this));

### BeanFactoryPostProcessor和BeanDefinitionRegistryPostProcessor区别

1．BeanFactoryPostProcessor.postProcessBeanFactory：Spring完成扫描之后，实例化之前调用，这个方法可以修改Bean工厂所有api(可以让程序员干预bean工厂的初始化过程)，比如：动态注册类或修改bd。

2．BeanDefinitionRegistryPostProcessor.postProcessBeanDefinitionRegistry：Spring还没有完成扫描之前调用。

问题1：Spring如何保证内置的后置处理器先执行。

Spring扫描后置处理器的时候，会将所有后置处理器扫描出来，如果类继承BeanDefinitionRegistryPostProcessor，则会先执行postProcessBeanDefinitionRegistry这个方法，否则就执行BeanFactoryPostProcessor.postProcessBeanFactory这个方法，这样就可以保证，自己提供的后置处理器在扫描完成之后处理，因为开发很少用到继承BeanDefinitionRegistryPostProcessor来处理业务逻辑。

### Resource和URL

URL：JDK提供扫描并解析class文件的工具

Resource：Spring为了扩展JDK提供的URL

### 添加自定义扫描注解类

AnnotationConfigApplicationContext ac = **new** AnnotationConfigApplicationContext();  
ac.register(Appconfig.**class**);  
ac.scan(**"com.xiaobi.beanDefinition"**);  
ClassPathBeanDefinitionScanner cp = **new** ClassPathBeanDefinitionScanner(ac);  
cp.addIncludeFilter(**new** AnnotationTypeFilter(FamilyScanner.**class**));  
**int** scan = cp.scan(**"com.xiaobi.beanDefinition"**);*//添加了新的注解需要重新扫描*  
System.***out***.println(scan);  
ac.refresh();

### 后置处理器处理顺序

这里使用到策略模式（多态），顺序执行可以简单概括为：

1. 处理BeanDefinitionRegistryPostProcessor

1.1 API提供的后置处理器

1.2 内置的后置处理器

1.3 程序员提供的后置处理器

1. BeanFactoryPostProcessor

2.1 API提供的后置处理器

2.2 内置的后置处理器

2.3 程序员提供的后置处理器

1. 处理通过API提供的后置处理器

ac.addBeanFactoryPostProcessor(**new** XbBeanDefinitionRegisterPostProcessor());

（1）循环处理"通过API提供"的BeanDefinitionRegistryPostProcessor.postProcessBeanDefinitionRegistry方法

（2）循环将“通过API提供”的BeanFactoryPostProcessor放到regularPostProcessors集合中

1. 循环处理"有PriorityOrdered特点"的BeanDefinitionRegistryPostProcessor.postProcessBeanDefinitionRegistry方法
2. 循环处理"有Ordered特点"的BeanDefinitionRegistryPostProcessor.postProcessBeanDefinitionRegistry方法
3. 循环处理"无特点（可以理解为程序员自己提供的后置处理器）"的BeanDefinitionRegistryPostProcessor.postProcessBeanDefinitionRegistry方法
4. 循环处理"所有（因为BeanDefinitionRegistryPostProcessor继承BeanFactoryPostProcessor，所以需要处理父类的方法）"的BeanDefinitionRegistryPostProcessor.BeanFactoryPostProcessor.postProcessBeanFactory方法（这样继承BeanDefinitionRegistryPostProcessor的所有后置处理器功能都处理完，下面就是处理父类的功能）
5. 循环处理regularPostProcessors集合的BeanFactoryPostProcessor.postProcessBeanFactory方法
6. 循环处理"有PriorityOrdered特点"的BeanFactoryPostProcessor.postProcessBeanFactory方法
7. 循环处理"有Ordered特点"的BeanFactoryPostProcessor.postProcessBeanFactory方法
8. 循环处理"无特点（可以理解为程序员自己提供的后置处理器）"的BeanFactoryPostProcessor.postProcessBeanFactory方法

### Mybatis对如何将对象到bdMap中新老版本的区别

新：

1. Spring循环处理"无特点（可以理解为程序员自己提供的后置处理器）"的MapperScannerConfigurer.postProcessBeanDefinitionRegistry方法
2. 通过扫描器（ClassPathMapperScanner.Scan方法），扫描出所有mybatis的mappers
3. 注入到BeanDefinitionMap中

旧：

1. Spring处理内置的BeanDefinitionRegistryPostProcessor.postProcessBeanDefinitionRegistry方法
2. 此时会通过ConfigurationClassPostProcessor.processConfigBeanDefinitions.parse.processConfigurationClass.doProcessConfigurationClass.processImports方法 (扫描Import注解)
3. 处理ImportBeanDefinitionRegistrar.registerBeanDefinitions方法
4. 注册mybatis自己提供的MapperScannerConfigurer
5. 接着执行MapperScannerConfigurer.postProcessBeanDefinitionRegistry
6. 通过扫描器（ClassPathMapperScanner.Scan方法），扫描出所有mybatis的mappers
7. 注入到BeanDefinitionMap中

## CGLib动态代理

### @Configuration作用

配置类不一定要带有@Configuration注解，@Configuration的作用是为了保证配置类能够产生代理对象，因为如果配置类不是代理对象，在配置类里面提供@Bean注解，并且方法有互相调用，会打破Spring单例原则(@Configuration注解的作用是保证@Bean产生的对象只产生一次，方法的互相调用，用到CGLib代理)

@Configuration  
**public class** CGLibAppconfig {  
@Bean  
**public** A a(){  
System.***out***.println(**"init a"**);  
**return new** A();  
 }  
  
@Bean  
**public** B b(){  
System.***out***.println(**"init b"**);  
 a();  
**return new** B();  
 }  
}

加了@Configuration

init a

init b

com.xiaobi.CGLib.CGLibAppconfig$$EnhancerBySpringCGLIB$$94e8b2fc@66133adc

没加@Configuration

init a

init b

init a

com.xiaobi.CGLib.CGLibAppconfig$$EnhancerBySpringCGLIB$$94e8b2fc@66133adc

加了@Configuration实际代理对象是  
public class CGLibAppconfig$$EnhancerBySpringCGLIB$$94e8b2fc@66133adc extend CGLibAppconfigimplementsMethodInterceptor {  
 public A a(){  
System.out.println("init a");  
return new A();  
 }  
 public B b(){  
System.out.println("init b");  
MethodInterceptor.intercept(a方法);//代理对象通过beanFactory.getBean获取a方法返回的Bean  
return new B();  
 }  
}

### CGLib动态代理应用

Enhancer enhancer = **new** Enhancer();  
*//设置原对象*enhancer.setSuperclass(A.**class**);  
enhancer.setCallback(**new** CGLibMethodInterceptor());  
A a = (A)enhancer.create();  
String name = a.cglibMethod(**"你好"**);  
System.***out***.println(name);

**public class** CGLibMethodInterceptor**implements**MethodInterceptor {  
@Override  
*//Object o 父类  
//Method method调用的父类方法  
//Object[] objects 调用父类方法，传进来的值  
//MethodProxymethodProxy代理方法***public** Object intercept(Object o, Method method, Object[] objects, MethodProxymethodProxy) **throws** Throwable {  
System.***out***.println(**"cglib代理对象，执行自己的逻辑"**);  
System.***out***.println(method.toGenericString());  
*//执行父类方法父类没有参数*Object targetObject = methodProxy.invokeSuper(o,objects);  
**return** targetObject;  
 }  
}

### Spring哪里用到代理

1. 加了@Configuration注解的配置类
2. Aop代理

### CGLib代理流程

1. Spring扫描带注解的class
2. 将class转成bd（此时包括加了@Bean注解的方法返回的类）
3. 循环执行后置处理器
4. 执行ConfigurationClassPostProcessor.postProcessBeanFactory.enhanceConfigurationClasses方法
5. 循环判断bd是否是全注解类，如果是put到configBeanDefs
6. 循环configBeanDefs得到所有全注解bd，并获取原对象BeanClass，重新动态代理出新的BeanClass，通过setBeanClass替换原对象BeanClass，此时Appconfig的Bean就是个代理对象，代理继续往下执行
7. 执行到需要实例化对象preInstantiateSingletons这个方法
8. 循环所有beanDefinitionNames，通过getBean(a)得到a的bd

getBean🡪doGetBean🡪getSingleton🡪createBean🡪doCreateBean🡪createBeanInstance

1. createBeanInstance去实例化a的时候发现a是由Appconfig.a()方法产生的，此时Spring就走通过方法去创建Bean（通过invoke动态调用方法），此时的Appconfig已经被代理，在方法a中实际执行的是MethodInterceptor.intercept
2. 判断当前正在创建的方法和当前正在执行的方法是否一致，如果一致则执行Appconfig.a()方法去创建对象，否则通过beanFactory.getBean去获取或创建对象，beanFactory.getBean是单例原则，所以加了@Configuration对象只会创建一次。

## 推断构造方法

doCreateBean.createBeanInstance.determineConstructorsFromBeanPostProcessors推断构造方法在这里开始。

Constructor<?>[] ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanName);  
**if** (ctors != **null** || mbd.getResolvedAutowireMode() == ***AUTOWIRE\_CONSTRUCTOR*** ||  
mbd.hasConstructorArgumentValues() || !ObjectUtils.*isEmpty*(args)) {  
*//找到合理的构造方法***return** autowireConstructor(beanName, mbd, ctors, args);  
}

### 手动装配：

（1）只有1个无参构造方法，ctors返回空，使用无参构造方法实例化对象。

（2）只有1个有参构造方法，ctors返回1(因为他没有其他构造方法，只能用这个构造方法去实例化对象)，使用该构造方法实例化对象。

（3）找到多个有参构造方法、1个无参构造方法，ctors返回空（因为Spring不知道要用那个构造方法去实例化对象），使用无参构造方法实例化对象。

（4）找到多个有参构造方法、0个无参构造方法，报错（因为Spring实例化对象的时候会默认调用无参构造方法，发现没有无参构造方法，所以报错）。

（5）如果程序员自己提供了参数给Spring，ctors返回空，但还进行二次推断构造方法autowireConstructor，因为mbd.hasConstructorArgumentValues()为true，使用构造方法参数和程序员提供参数个数匹配的构造方法，如果这种构造方法有多个则使用排序靠前的构造方法。

genericBeanDefinition.getConstructorArgumentValues().addGenericArgumentValue("com.xiaobi.service.OrderService");

### 自动装配：

通过构造方法进行自动装配（default-autowire="constructor"或者

familyService.setAutowireMode(AbstractBeanDefinition.AUTOWIRE\_CONSTRUCTOR)）：

（1）只有1个无参构造方法，ctors返回空，使用无参构造方法。

（2）只有1个有参构造方法，ctors返回1，使用该构造方法。

（3）找到多个构造方法，不管有没有参数，ctors返回空，使用有效参数（存在Spring容器中）最多的构造方法。

### 半自动装配：

加了@Autowired注解（注意：多个@Autowired注解，不允许存在required为true的构造方法）

（1）只有1个无参构造方法，ctors返回空，使用无参构造方法。

（2）只有1个有参构造方法，ctors返回1，使用该构造方法。

（3）找到多个required为true的构造方法，不管有没有参数，报错(因为提供多个合格的构造方法，Spring不知道使用那一个)。

（4）找到多个required为false，0个required为true的构造方法，不管有没有参数，ctors返回多个，使用有效参数最多的构造方法，如果有效参数相同，则使用排序靠前的构造方法。

（5）找到多个required为false，大于0个required为true的构造方法，不管有没有参数，报错(因为提供多个合格的构造方法，Spring不知道使用那一个)。

### 自动装配+半自动装配

ctors以半自动装配为主

### 手动装配+自动装配

手动给构造方法参数

genericBeanDefinition.getConstructorArgumentValues().addGenericArgumentValue("com.xiaobi.service.OrderService");

设置构造方法自动注入

genericBeanDefinition.setAutowireMode(AbstractBeanDefinition.AUTOWIRE\_CONSTRUCTOR);

假如遇到这种情况：Spring就不会只用参数为1个构造方法，他会使用有效参数最多的构造方法，因为此时手动注入参数值已经没有意义了。

## 合并bd流程

### 第一次合并bd

将所有扫描出来的bd、api注册的bd、Spring内置的bd都进行合并,并将合并后的bd缓存起来（保存到mergedBeanDefinitions里面）,为第二次合并bd提供缓存数据源。

DefaultListableBeanFactory.getBeanNamesForType.doGetBeanNamesForType.getMergedLocalBeanDefinition

### 第二次合并bd

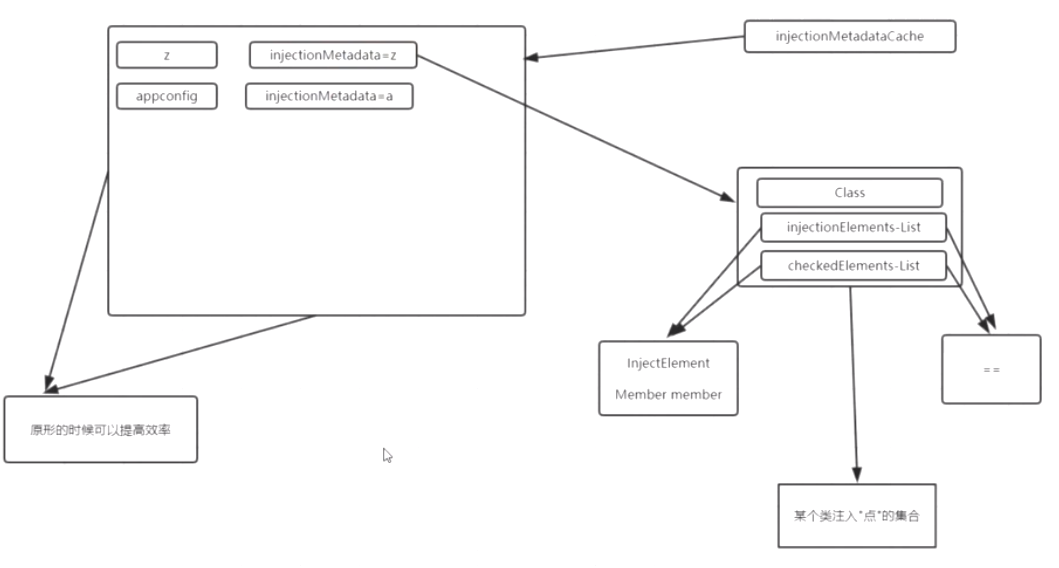
程序员可以通过后置处理器，在初始化之后提供bd，所以需要再次合并bd。

DefaultListableBeanFactory.preInstantiateSingletons.getMergedLocalBeanDefinition

### 处理合并后bd

从合并后的bd里面把需要注入的属性拿出来放到缓存集合中，供populateBean后面获取这些属性进行属性填充，使用的是checkedElements数据(源代码在这里populateBean.AutowiredAnnotationBeanPostProcessor.postProcessProperties.inject)，没有用injectionElements，下面是流程图。

AbstractAutowireCapableBeanFactory.applyMergedBeanDefinitionPostProcessors.postProcessMergedBeanDefinition



### 移除合并后bd

程序员可以通过后置处理器，在初始化之后改变bd，所以在第三次合并bd前需要移除当前bd，让第三次合并bd再次进行合并。

doGetBean.markBeanAsCreated

### 第三次合并bd

这次是合并后的bd是最终数据。

doGetBean.getMergedLocalBeanDefinition