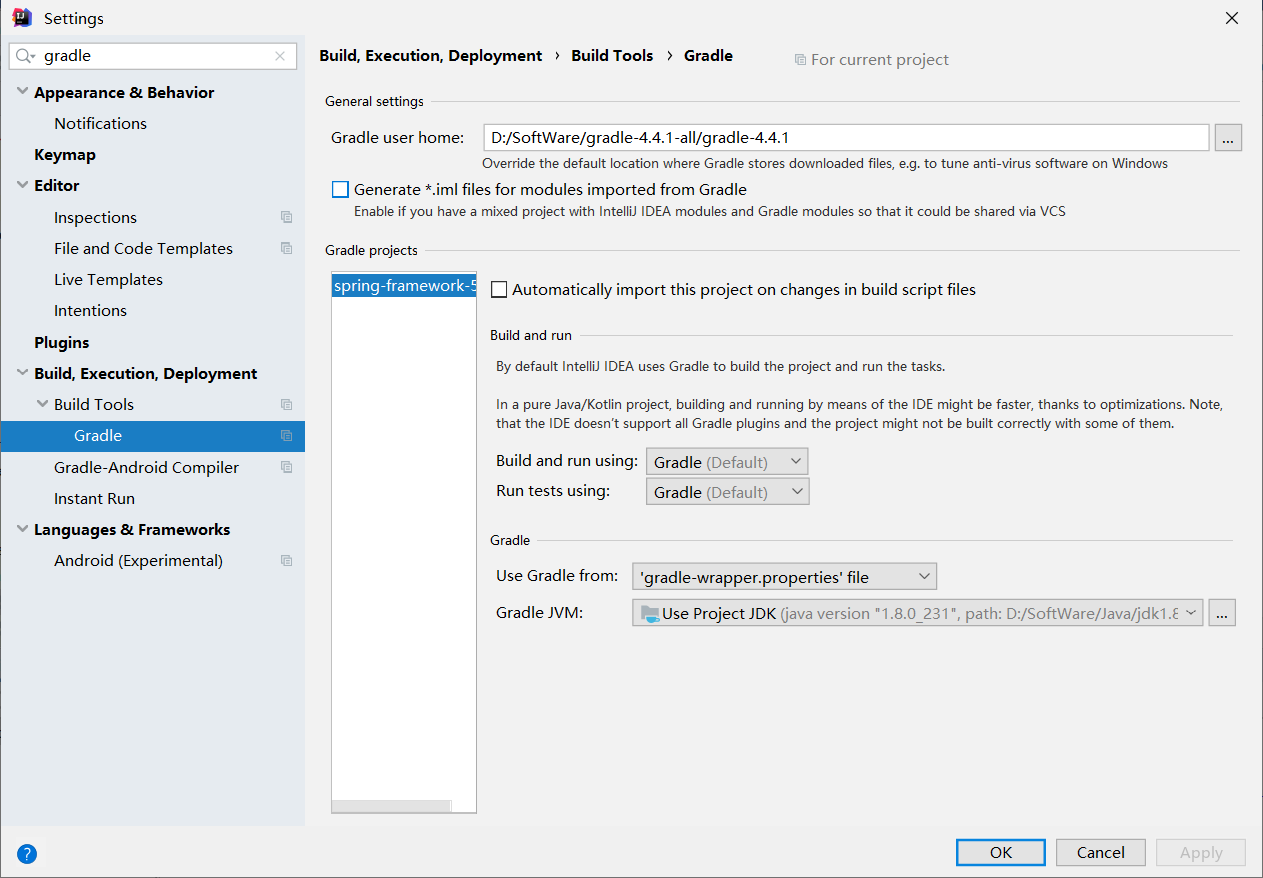
## Spring源码编译

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、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



## 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）autowireMode = 1

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

3、bytype（要提供set方法，和setXXX这里面的XXX无关） 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。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。

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

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

### @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"**);  
}

#### 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() 容器是否运行方法

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

## 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userService bean put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indexService bean put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稽核中添加一条数据，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. createBean实例化Index
37. createBeanInstance获取到Index对象
38. 如果允许循环依赖，addSingletonFactory将index工厂（这里的工厂是通过AOP生成的一个代理对象）放到二级缓存中
39. populateBean属性注入
40. Index对象注入user
41. getBean（user）
42. 第一次调用getSingleton方法
43. 判断是否需要从第三个缓存中拿
44. 拿不到
45. 标识User为正在创建过程中
46. 实例化User
47. 获取到User对象
48. 如果允许循环依赖，将user工厂放到二级缓存中
49. 走User生命周期
50. User对象需要注入Index
51. getBean(Index)
52. 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
53. index对象不等于空 set给user注入
54. 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
55. 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生命周期大概流程

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map validate解析验证。
7. 实例化对象，调用构造方法（new index）
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. 代理aop（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

### 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, RootBeanDefinition mbd, Object bean) {

Object exposedObject = bean;

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

//循环Spring所有内置处理器

for (BeanPostProcessor bp : getBeanPostProcessors()) {

if (bp instanceof SmartInstantiationAwareBeanPostProcessor) {

SmartInstantiationAwareBeanPostProcessor ibp = (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）。

## Spring AOP

Spring官网5.4章节

### 快速启动AOP

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(public com.xiaobi.service (..))")

public void anyPublicOperation(){}

@Before("anyPublicOperation()")

public void doAccessCheck(){}

}

### 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。