**动态代理**

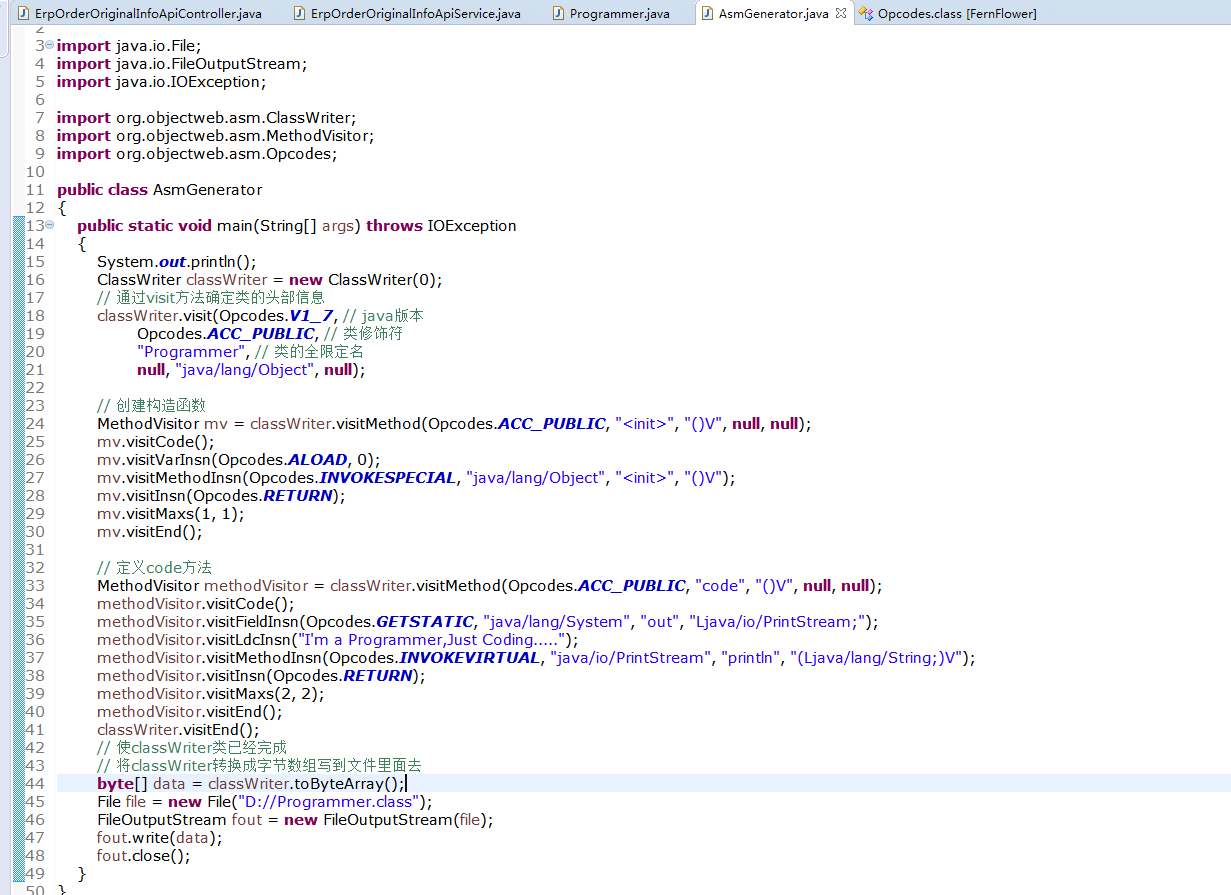
## 一、字节码生成器IMG_256

**1 Java编译器编译好Java文件之后，产生.class 文件在磁盘中。**这种class文件是二进制文件，内容是只有JVM虚拟机能够识别的机器码。JVM虚拟机读取字节码文件，取出二进制数据，加载到内存中，解析.class 文件内的信息，生成对应的 Class对象

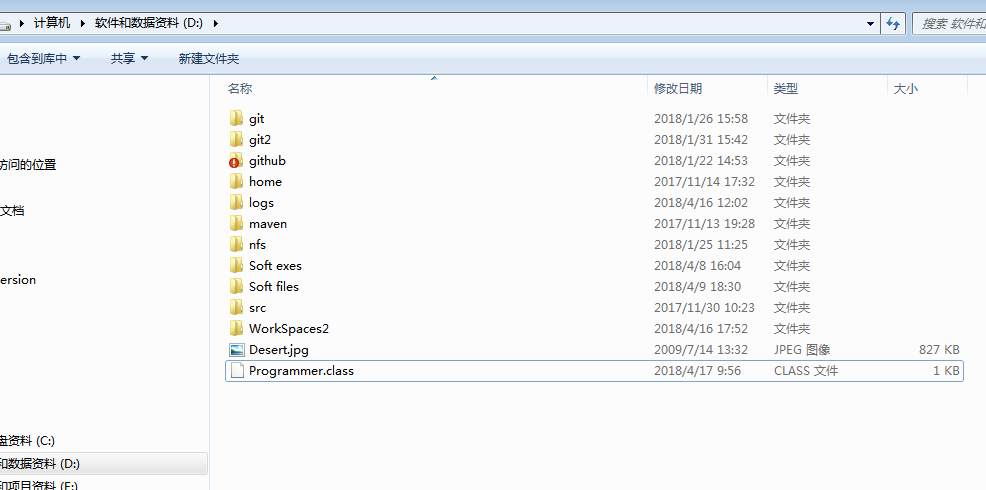
**2 ASM字节码操控框架**

 ASM 是一个 Java 字节码操控框架。它能够以**二进制形式修改已有类或者动态生成类**。ASM 可以直接产生二进制 class 文件，也可以在类被加载入 Java 虚拟机之前动态改变类行为。ASM 从类文件中读入信息后，能够改变类行为，分析类信息，甚至能够根据用户要求生成新类。

**asm代码：**



**运行效果：**



**3 Javassist**

Javassist是一个开源的分析、编辑和创建Java字节码的类库。是由东京工业大学的数学和计算机科学系的 Shigeru Chiba （千叶 滋）所创建的。它已加入了开放源代码JBoss 应用服务器项目,通过使用Javassist对字节码操作为JBoss实现动态AOP框架。javassist是[jboss](http://baike.baidu.com/view/309533.htm" \t "https://blog.csdn.net/skiof007/article/details/_blank)的一个子项目，其主要的优点，在于简单，而且快速。直接使用java编码的形式，而**不需要了解[虚拟机](http://baike.baidu.com/view/1132.htm" \t "https://blog.csdn.net/skiof007/article/details/_blank)指令，就能动态改变类的结构，或者动态生成类**。

**public** **static** **void** main(String[] args) **throws** IOException, CannotCompileException

{

ClassPool pool = ClassPool.*getDefault*();

// 创建Programmer类

CtClass cc = pool.makeClass("com.samples.Programmer");

// 定义code方法

CtMethod method = CtNewMethod.*make*("public void code(){}", cc);

// 插入方法代码

method.insertBefore("System.out.println(\"I'm a Programmer,Just Coding.....\");");

cc.addMethod(method);

// 保存生成的字节码

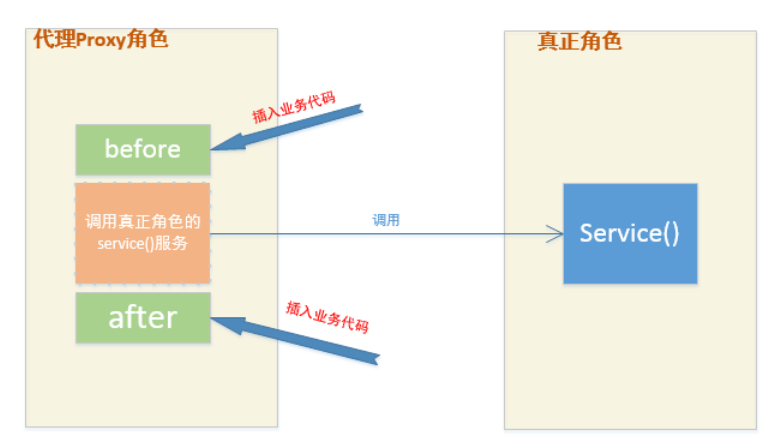
cc.writeFile("d://temp");

}

## 二、动态代理

**1 程序在运行期而不是编译器，生成被代理对象的代理对象**，并且被代理对象并不需要和代理对象实现共同的接口。基于此，我们可以利用代理对象，提供一种以控制对被代理对象的访问。

**如果用asm或者javassist动态生成代理类，含有太多的业务代码。我们使用上述创建Proxy代理类的方式的初衷是减少系统代码的冗杂度，但是上述做法却增加了在动态创建代理类过程中的复杂度：手动地创建了太多的业务代码，并且封装性也不够，完全不具有可拓展性和通用性。如果某个代理类的一些业务逻辑非常复杂，上述的动态创建代理的方式是非常不可取的！**



**2 在这种模式之中：代理Proxy 和RealSubject 应该实现相同的功能**，这一点相当重要。（我这里说的功能，可以理解为某个类的public方法）。在面向对象的编程之中，如果我们想要约定Proxy 和RealSubject可以实现相同的功能，有两种方式：

1. **共同实现接口，**一个比较直观的方式，就是定义一个功能接口，然后让Proxy 和RealSubject来实现这个接口。
2. **子类继承，**还有比较隐晦的方式，就是通过继承。因为如果Proxy 继承自RealSubject，这样Proxy则拥有了RealSubject的功能，Proxy还可以通过重写RealSubject中的方法，来实现多态。

**3 其中JDK中提供的创建动态代理的机制，是以a 这种思路设计的，而cglib 则是以b思路设计的。**

**4 JDK 的InvocationHandler**

package dynamic.proxy;

import java.lang.reflect.InvocationHandler;

import java.lang.reflect.Method;

import java.lang.reflect.Proxy;

public class MyInvocationHandler implements InvocationHandler

{

private Object target;

public MyInvocationHandler(Object target)

{

super();

this.target = target;

}

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

System.out.println("------------------before------------------");

Object result = method.invoke(target, args);

System.out.println("-------------------after------------------");

return result;

}

public Object getProxy() {

return Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(),target.getClass().getInterfaces(), this);

}

}

package dynamic.proxy;

public interface UserService {

public abstract void add();

}

package dynamic.proxy;

public class UserServiceImpl implements UserService {

public void add() {

System.out.println("--------------------add---------------");

}

}

package dynamic.proxy;

import org.junit.Test;

public class ProxyTest {

@Test

public void testProxy() throws Throwable {

UserService userService = new UserServiceImpl();

MyInvocationHandler invocationHandler = new MyInvocationHandler(userService);

UserService proxy = (UserService) invocationHandler.getProxy();

proxy.add();

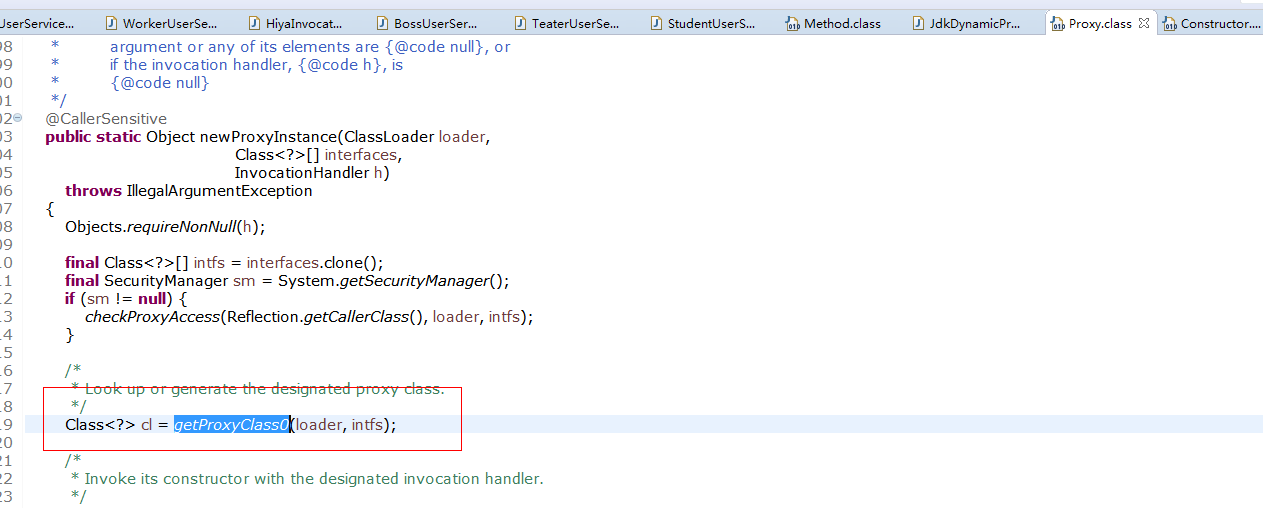
}

}

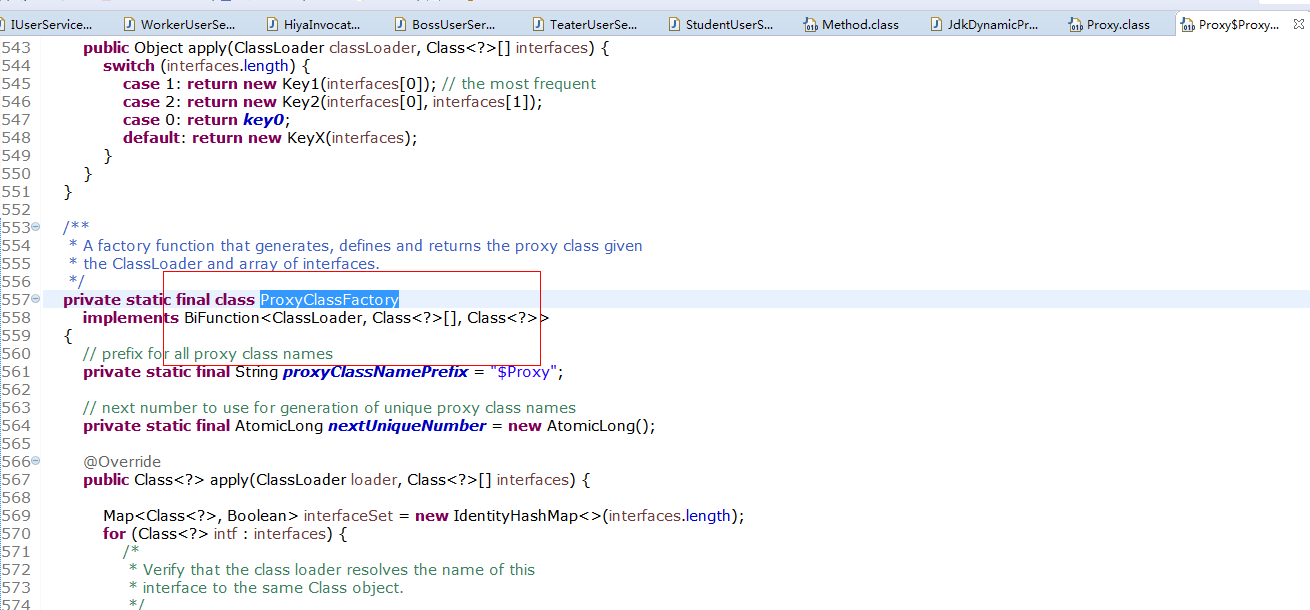
用起来是比较简单，但是如果能知道它背后做了些什么手脚，那就更好不过了。首先来看一下JDK是怎样生成代理对象的。既然生成代理对象是用的Proxy类的静态方newProxyInstance，那么我们就去它的源码里看一下它到底都做了些什么

**5 源码解析**

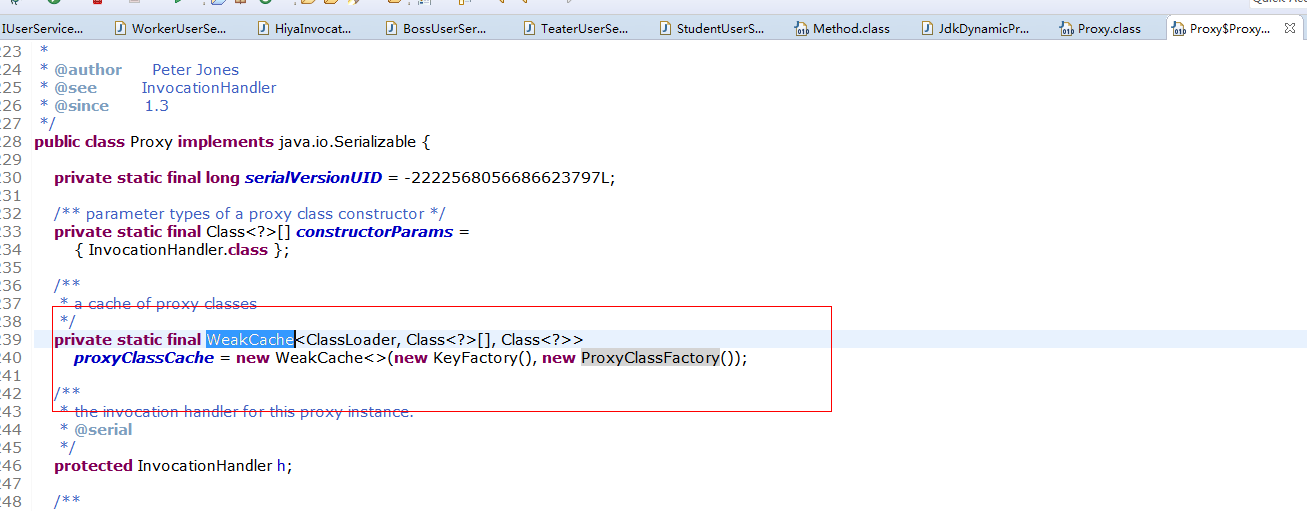
**Proxy类的生成代理实例的方法**



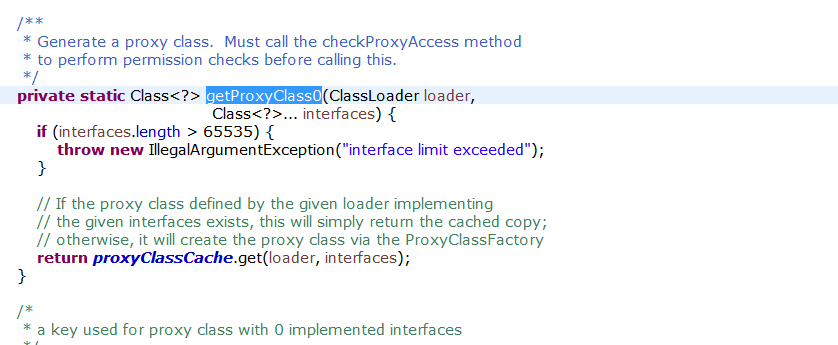
ProxyClassFactory工厂类是Proxy的内部静态类

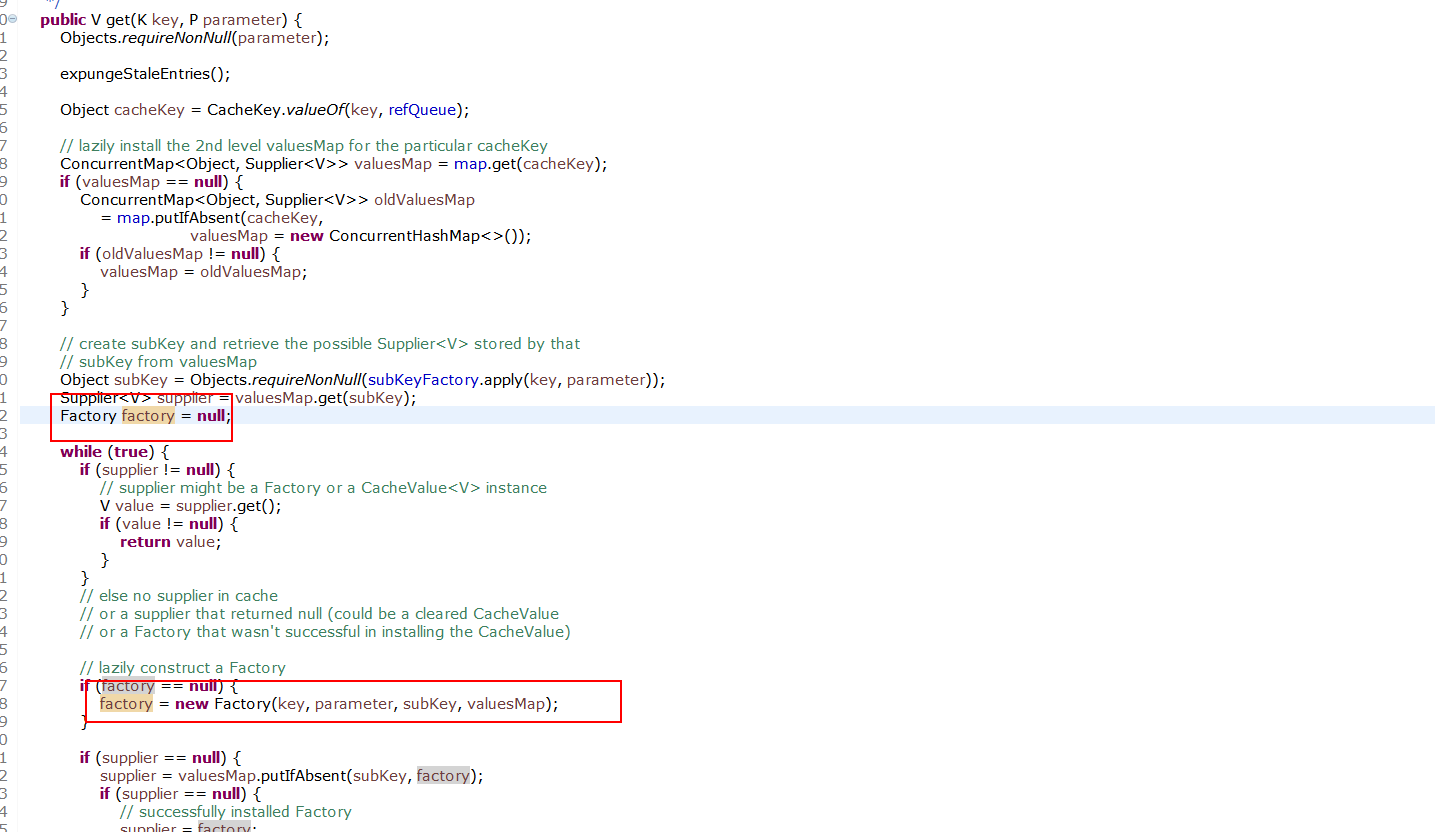


WeakCache缓存类的value是实现foctory的实现类。

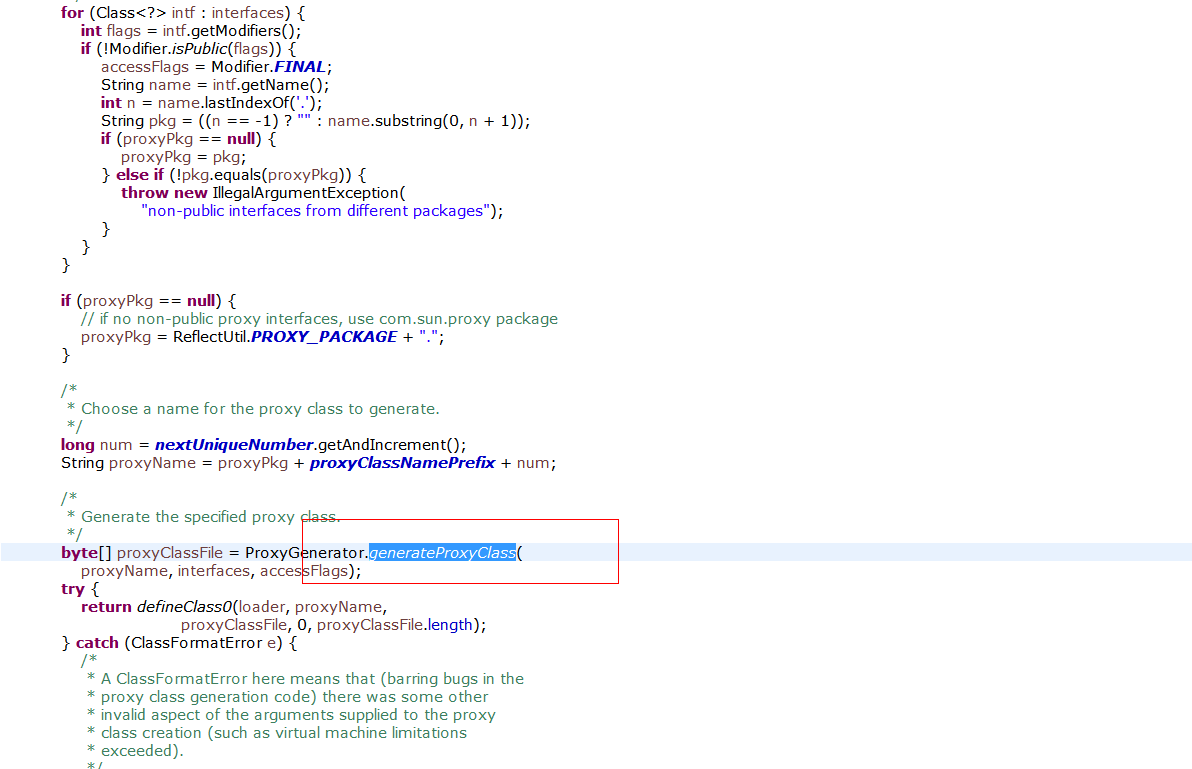


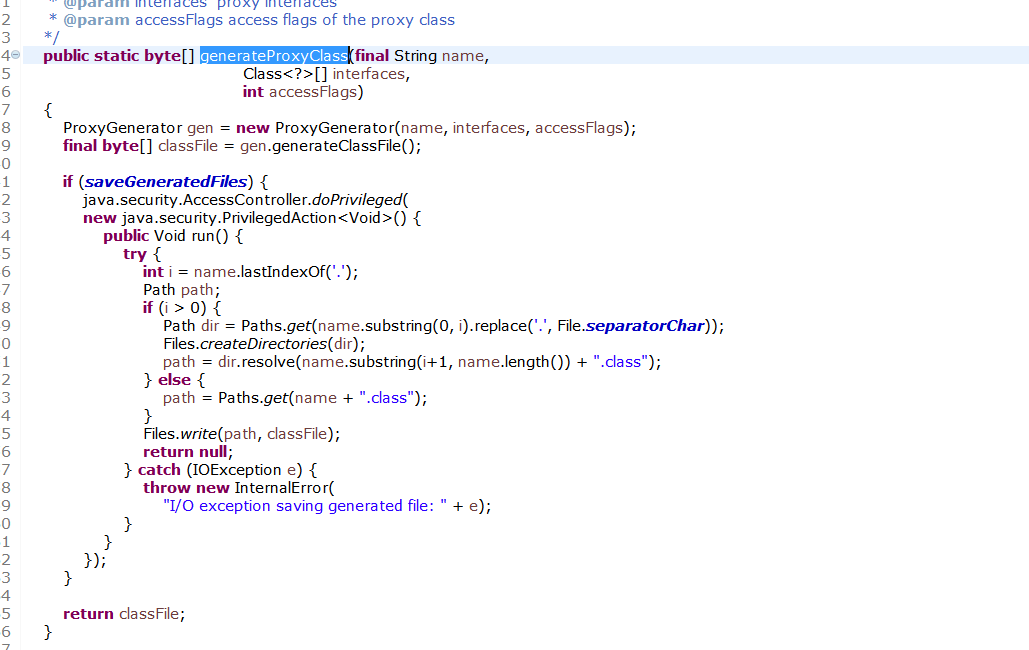
所以说直接从缓存获取，如果缓存中为null，则根据factory创建





ProxyGenerator.generateProxyClass利用IO流把class类生成到硬盘



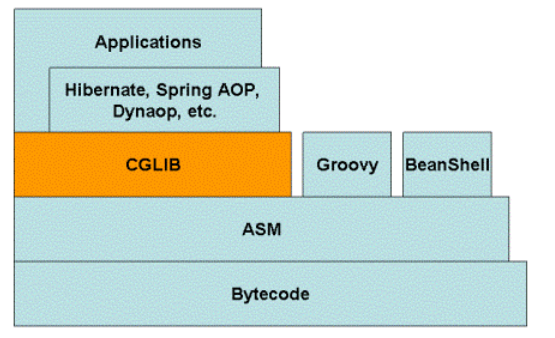


**6 Cglib 代理**

CGLIB代理主要通过对字节码的操作，为对象引入间接级别，以控制对象的访问。我们知道Java中有一个动态代理也是做这个事情的，那我们为什么不直接使用Java动态代理，而要使用CGLIB呢？答案是CGLIB相比于JDK动态代理更加强大，JDK动态代理虽然简单易用，但是其有一个致命缺陷是，只能对接口进行代理。如果要代理的类为一个普通类、没有接口，那么Java动态代理就没法使用了。

**Enhancer既能够代理普通的class，也能够代理接口。CGLIB底层使用了ASM（一个短小精悍的字节码操作框架）来操作字节码生成新的类。**

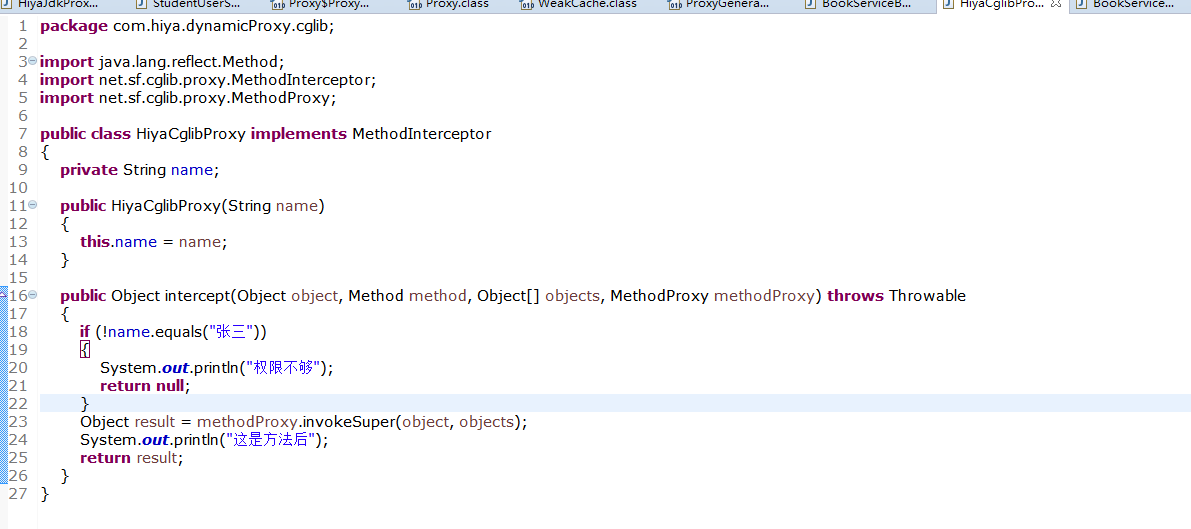
Enhancer对**非final**方法test()、toString()、hashCode()进行了拦截，没有对getClass进行拦截。

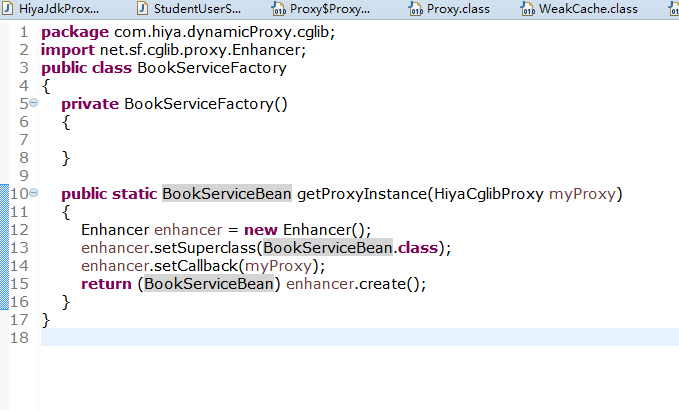


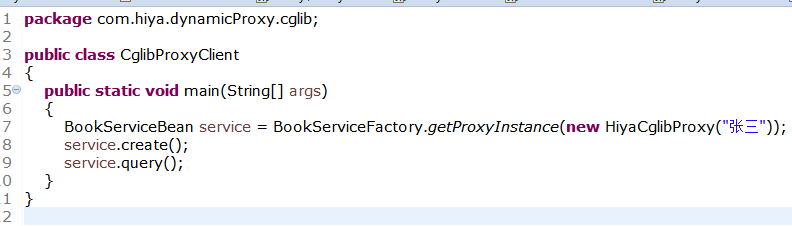
**7 CGLIB和Java动态代理的区别**

1 Java动态代理只能够对接口进行代理，不能对普通的类进行代理（因为所有生成的代理类的父类为Proxy，Java类继承机制不允许多重继承）；CGLIB能够代理普通类；

2 Java动态代理使用Java原生的反射API进行操作，在生成类上比较高效；CGLIB使用ASM框架直接对字节码进行操作，在类的执行过程中比较高效







## 三、Mybatis的动态代理解析

**1 首先有一个MapperProxy implements InvocationHandler**

持有 sqlSession，mapperInterface 对象

利用jdk产生代理对象

**public** **class** MapperProxy<T> **implements** InvocationHandler, Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = -6424540398559729838L;

**private** **final** SqlSession sqlSession;

**private** **final** Class<T> mapperInterface;

**private** **final** Map<Method, MapperMethod> methodCache;

**public** MapperProxy(SqlSession sqlSession, Class<T> mapperInterface, Map<Method, MapperMethod> methodCache) {

**this**.sqlSession = sqlSession;

**this**.mapperInterface = mapperInterface;

**this**.methodCache = methodCache;

}

**public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {

**if** (Object.**class**.equals(method.getDeclaringClass())) {

**try** {

**return** method.invoke(**this**, args);

} **catch** (Throwable t) {

**throw** ExceptionUtil.*unwrapThrowable*(t);

}

}

**final** MapperMethod mapperMethod = cachedMapperMethod(method);

**return** mapperMethod.execute(sqlSession, args);

}

**private** MapperMethod cachedMapperMethod(Method method) {

MapperMethod mapperMethod = methodCache.get(method);

**if** (mapperMethod == **null**) {

mapperMethod = **new** MapperMethod(mapperInterface, method, sqlSession.getConfiguration());

methodCache.put(method, mapperMethod);

}

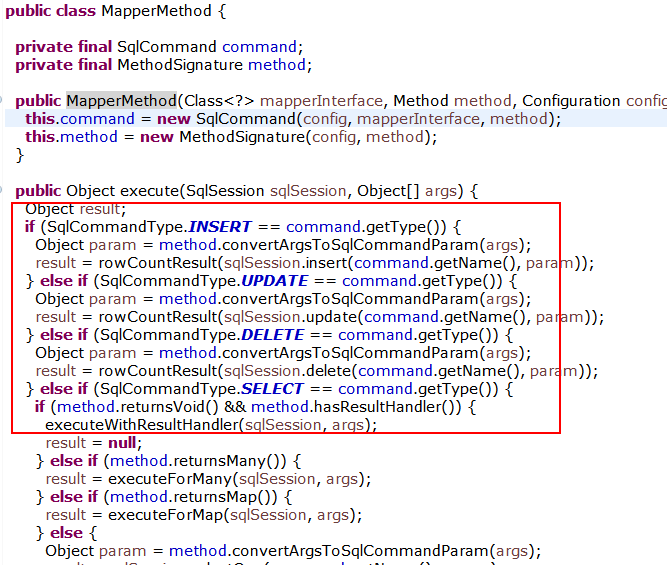
**return** mapperMethod;

}

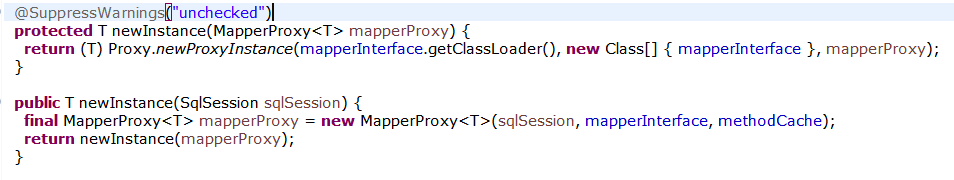
}

**2 MapperMethod是SQL执行的引擎核心类**

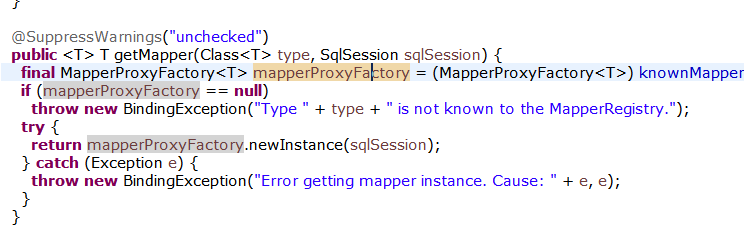
**传入sqlSession执行sql语句，根据增删改查类型执行对应的逻辑**



**3 MapperProxyFactory工厂类专门负责创建代理实例**



**4 MapperRegistry是客户端获取代理类**



**5 启动web服务器**

**初始化扫描器，扫描basePackage下所有以@MyBatisDao注解的接口**

MapperScannerConfigurer的Scanner类负责搜索basePackage类下所有的MapperClass并将其注册至spring的beanfinitionHolder中，其注册的classBean为MapperFactoryBean。

<bean id=*"mapperScannerConfigurer"* class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>

<property name=*"sqlSessionFactoryBeanName"* value=*"sqlSessionFactory"* />

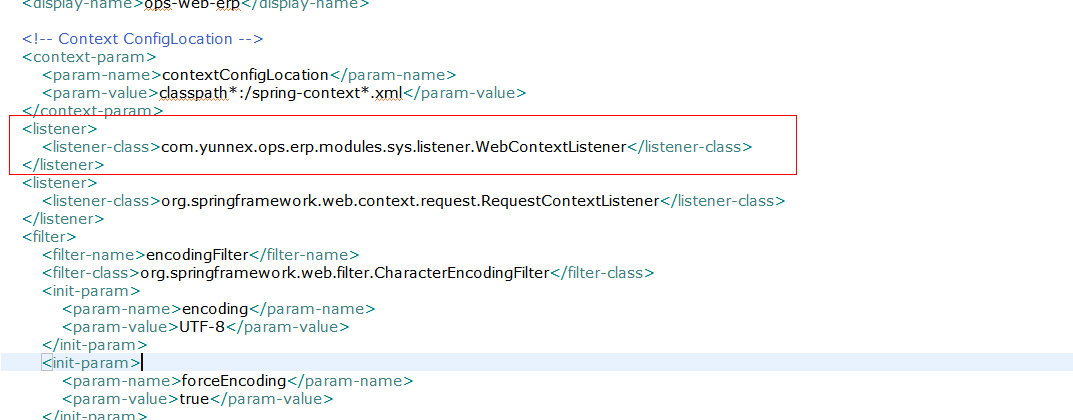
<property name=*"basePackage"* value=*"com.yunnex.ops.erp"* />

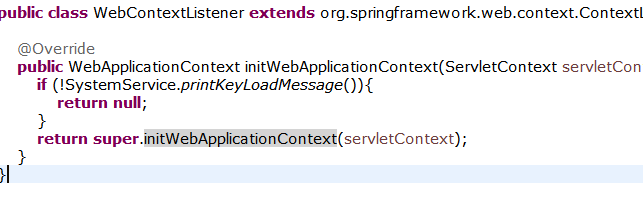
<property name=*"annotationClass"*

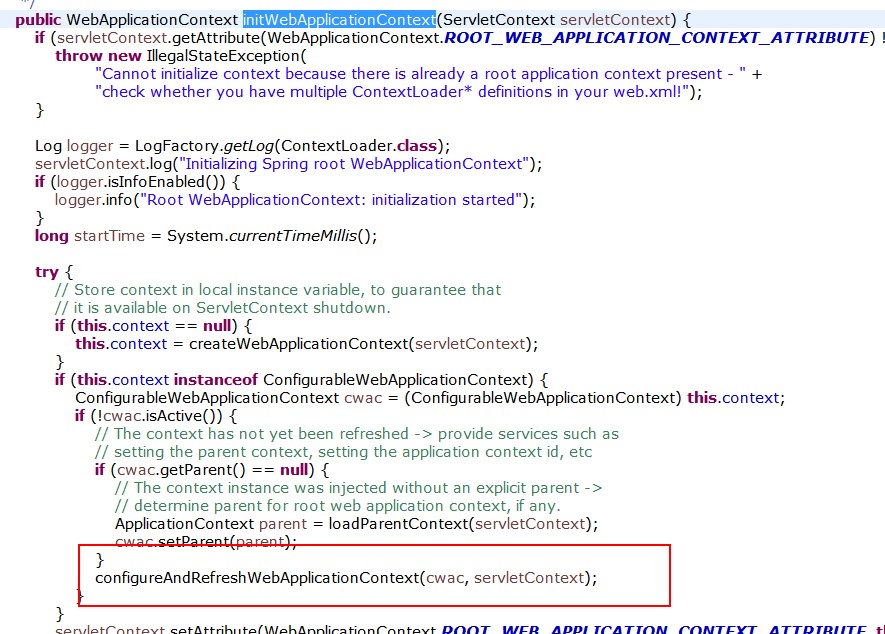
value=*"com.yunnex.ops.erp.common.persistence.annotation.MyBatisDao"* />

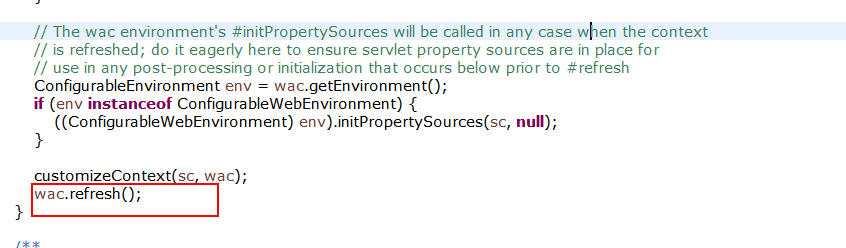
</bean>

**6 Web.xml配置监听器**

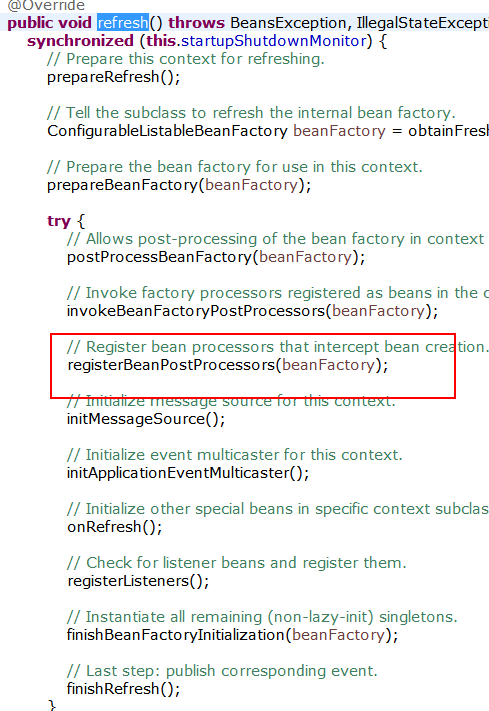






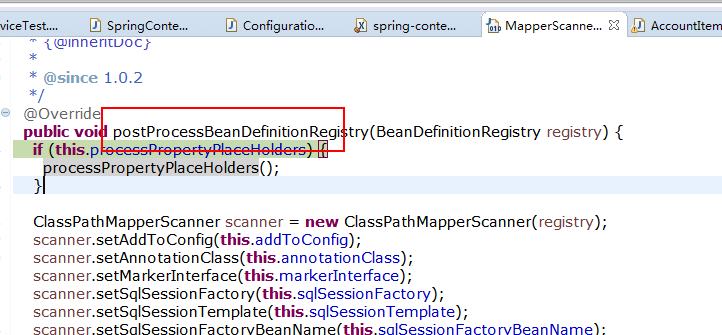


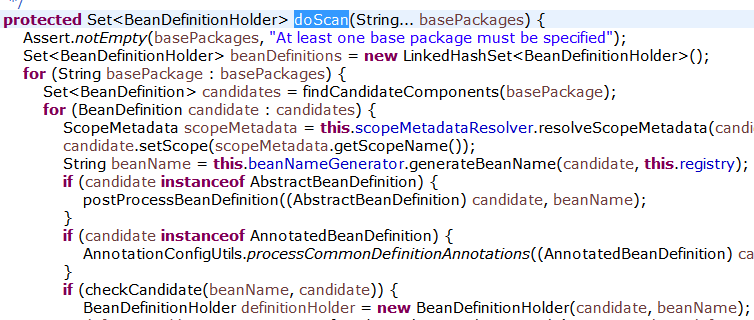
**7 到了AbstractApplicationContext的refresh方法**



**这里会调用前面说的MapperScannerConfigurer.postProcessBeanDefinitionRegistry 方法**

**8 执行 postProcessBeanDefinitionRegistry方法**





**9 调用MapperRegistry的**

**public** **void** addMappers(String packageName) {

addMappers(packageName, Object.**class**);

}

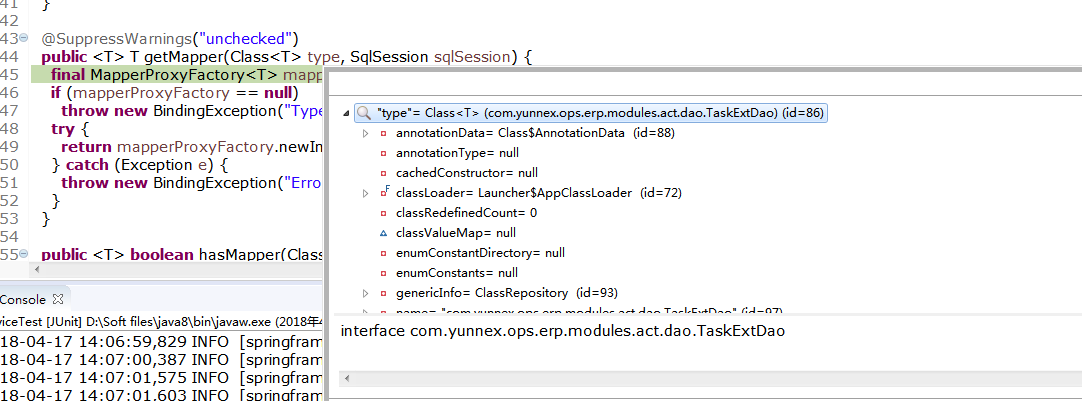
**会把接口和对应的代理类放入到**

**final** Map<Class<?>, MapperProxyFactory<?>> knownMappers

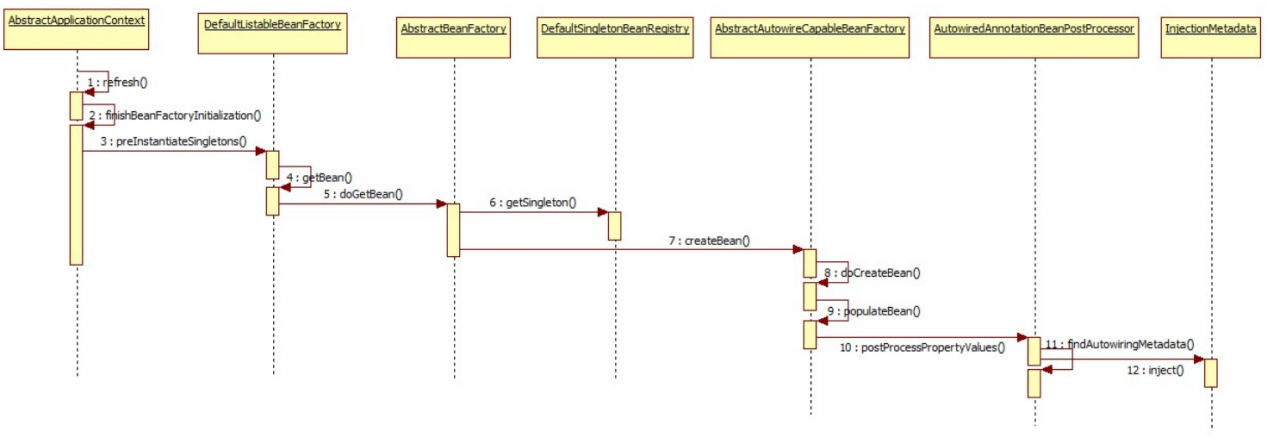
**10 单元测试获取bean,底层调用MapperRegistry.getMapper 方法**

TaskExtDao taskExtDao = SpringContextHolder.*getBean*(TaskExtDao.**class**);

**return** *applicationContext*.getBean(requiredType);



**11 Spring的Bean 注入**



**12 Spring事务的动态代理解析**

**Spring的事务管理机制实现的原理，就是通过这样一个动态代理对所有需要事务管理的Bean进行加载，并根据配置在invoke方法中对当前调用的 方法名进行判定，并在method.invoke方法前后为其加上合适的事务管理代码，这样就实现了Spring式的事务管理。Spring中的AOP实 现更为复杂和灵活，不过基本原理是一致的。**

<tx:annotation-driven transaction-manager="txManager"/>

 该配置方式默认就是JDK动态代理方式

<tx:annotation-driven transaction-manager="txManager" proxy-target-class="true"/>

 该配置方式是基于CGLIB类代理