26_源码分析：Spring AOP CGLIB 动态代理实现原理

1、开篇

上节课针对 Spring AOP JDK 的主题，分析对应的源代码。从 invoke 实现 AOP 中具体的逻辑和 getProxy 获取产生的代理类两个方面进行了分析。本节课依旧是源码分析，分析的对象是另一种 Spring AOP 代理方式： CGLIB 动态代理。内容包括：

- 通过一个例子回顾 CGLIB 的应用
- CGLIB 中 Class 对象创建过程

2、通过一个例子回顾 CGLIB 的应用

尽管在之前的章节就介绍过 CGLIB 的实现方式，这里还是通过一个小例子和大家一起回顾一下。如图 1，所示这里定义了 RealService 作为需要增强的 Class，其方法是 realMethod 作为需要被增强的方法。

```
public class RealService {

    public void realMethod() {
        System.out.println("realMethod execute");
    }
}
```

如图 1 RealService

自定义一个 MyServiceIntercepter 类实现了 MethodInterceptor，主要是 override 了 intercept 方法，在对应的拦截器中做了如下操作：在原有的方法的前面和后面分别加入了”before Execute” 和 “after Execute”的输出。

```java
public class MyServiceInterceptor implements MethodInterceptor {

    public static void main(String[] args) {

        //设置代理类生成目录
        System.setProperty(DebuggingClassWriter.DEBUG_LOCATION_PROPERTY, "D:\\proxy");
        Enhancer enhancer = new Enhancer();
        //设置超类，因为cglib基于父类 生成代理子类
        enhancer.setSuperclass(RealService.class);
        //设置回调，也就是我们的拦截处理
        enhancer.setCallback(new MyServiceInterceptor());

        //创建代理类
        RealService realService = (RealService) enhancer.create();
        //代用代理类的方法
        realService.realMethod();
    }

    @Override
    public Object intercept(Object obj, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {
        System.out.println("before execute");
        Object result=methodProxy.invokeSuper(obj, objects);
        System.out.println("after execute");
        return result;
    }
}
```

图 2 MyServiceInterceptor

3、CGLIB 中 Class 对象创建过程

从上面这个例子可以知道，CGLIB 是通过字节码增强处理框架 ASM，来生成字节码并装载到 JVM。和 JDK 代理基于接口实现方式不同的是，CGLIB 没有局限于接口，采用的是生成子类的方式。这个子类本质上就是一个 Class 对象，换句话说原来是执行原有的 Class，CGLIB 会通过字节码增强的方式，在字节码的层面生成一个子类去集成需要增强的类，在子类中加入需要增强的方法，让这个子类代替原有的类，完成增强的操作。

我们将 CGLIB 生成 Class 对象分为三个步骤：

1. 生成指定类的 Class 对象字节数组。
2. 将 Class 对象字节数组转换为 Class 对象。
3. 通过 Class.forName 方法将 Class 对象装载到 JVM。

（1）生成指定类的 Class 对象字节数组

如图 3 所示，在创建 Enhancer 对象会调用 create 方法从而生成超类的子类。深入到 create 方法内部可以看到，先通过 getClassLoader 获取当前类加载器，通过

AbstractClassGenerator 中的 ClassLoaderData 方法传入加载器，获取加载类的内容 data。并且将其存放到缓存中，接下来利用 data 中的 get 方法获取字节码信息并且返回。

```
protected Object create(Object key) {
    try {
      //获取当前类加载器，应用类加载器
        ClassLoader loader = this.getClassLoader();
        Map<ClassLoader, AbstractClassGenerator.ClassLoaderData> cache = CACHE;
        AbstractClassGenerator.ClassLoaderData data = (AbstractClassGenerator.ClassLoaderData)cache.get(loader);
        if (data == null) {
            Class var5 = AbstractClassGenerator.class;
            synchronized(AbstractClassGenerator.class) {
                cache = CACHE;
                data = (AbstractClassGenerator.ClassLoaderData)cache.get(loader);
                if (data == null) {
                    Map<ClassLoader, AbstractClassGenerator.ClassLoaderData> newCache = new WeakHashMap(cache);
                    //创建AbstractClassGenerator
                    data = new AbstractClassGenerator.ClassLoaderData(loader);
                    newCache.put(loader, data);
                    CACHE = newCache;
                }
            }
        }
        this.key = key;
        //调用 get方法获取字节码，如果没有字节码，则会创建字节码
        Object obj = data.get(this, this.getUseCache());
        return obj instanceof Class ? this.firstInstance((Class)obj) : this.nextInstance(obj);
    } catch (RuntimeException var9) {
        throw var9;
    } catch (Error var10) {
        throw var10;
    } catch (Exception var11) {
        throw new CodeGenerationException(var11);
    }
}
```

图 3 create

顺着 data 中 get 方法查看代码，get 中会判断是否开启缓存，如果没有使用缓存就调用 generate 方法；否则直接从缓存中获取对象。这里的缓存可以通过 enhancer.setUseCache 方法设置，默认为 true。

```
public Object get(AbstractClassGenerator gen, boolean useCache) {
    //判断是否开启缓存，可直接设置：enhancer.setUseCache(false);默认为true
    if (!useCache) {
        return gen.generate(this);
    } else {
        Object cachedValue = this.generatedClasses.get(gen);
        return gen.unwrapCachedValue(cachedValue);
 }
}
```

图 4 get

如图 5 所示，继续跟进到 generate 方法中传入了 data 作为参数，生成对应的代理类名称，然后通过类加载器和类名称尝试加载类，如果之前没有生成过对应的字节码那么创建字节码。接着通过生成字节码的策略生成字节码，当前对象即为 Enhancer 对象，字节数组的形式。

```
protected Class generate(AbstractClassGenerator.ClassLoaderData data) {
    Object save = CURRENT.get();
    CURRENT.set(this);
    Class var8;
    try {
        ClassLoader classLoader = data.getClassLoader();
        if (classLoader == null) {
            throw new IllegalStateException("ClassLoader is null while trying to define class
        }
        String className;
        //生成代理类名称
        synchronized(classLoader) {
            className = this.generateClassName(data.getUniqueNamePredicate());
            data.reserveName(className);
            this.setClassName(className);
        }
        Class gen;
        //这里通过应用类加载器和类名称尝试加载，如果加载不到，才开始创建字节码
        if (this.attemptLoad) {
                gen = classLoader.loadClass(this.getClassName());
                Class var25 = gen;
                return var25;
        }
        //通过生成策略创建字节码，当前对象即为Enhancer对象，字节数组形式
        byte[] b = this.strategy.generate(this);
        className = ClassNameReader.getClassName(new ClassReader(b));
        ProtectionDomain protectionDomain = this.getProtectionDomain();
        synchronized(classLoader) {
            //将字节码加载到JVM内存，同时会触发代理对象初始化
            if (protectionDomain == null) {
                gen = ReflectUtils.defineClass(className, b, classLoader);
            } else {
                gen = ReflectUtils.defineClass(className, b, classLoader, protectionDomain);
            }
        }
        var8 = gen;
    } finally {
        CURRENT.set(save);
    }
    return var8;
}
```

图 5 generate

在通过图 6 看看 getClassName 方法中是如何生成代理类名称的，红框的部分就是生成字节码文件的规则：真实类路径 + 来源(EnhancerByCGLIB) + key 的 hash 值的 16 进制 ：

```
public String getClassName(String prefix, String source, Object key, Predicate names) {
    if (prefix == null) {
        prefix = "org.springframework.cglib.empty.Object";
    } else if (prefix.startsWith("java")) {
        prefix = "$" + prefix;
    }
    //拼接类路径
    String base = prefix + "$$" + source.substring(source.lastIndexOf(46) + 1) + this.getTag() + "$$" + Integer.toHexString(STRESS_HASH_CODE ? 0 : key.hashCode());
    String attempt = base;
    return attempt;
}
```

图 6 getClassName

生成文件的结果就好像如下这样：

com.example.cglib.RealService$$EnhancerByCGLIB$$a9ba5c5e

具体生成字节码的方式就是通过 asm 的工具类 DefaultGeneratorStrategy 来生成，另外提供了一个 DebuggingClassWriter 来写入到指定目录，默认的目录为空，所以生成的代理类只存在于内存中。

（2）将 Class 对象字节数组转换为 Class 对象

通过上面的源代码分析已经生成了 Class 对象字节数组，接下来就需要将这个字节码的数组转换成 Class 对象。跟进 ReflectUtils.defineClass 去继续来看，如图 7 所示，方法 defineClass 传入了 byte[] b，这个 b 的数组中就存放的是字节数组。通过 new object 将字节数组赋值给 args，然后通过 DEFINE_CLASS.invoke 方法将字节数组转化为 Class 对象。

```
public static Class defineClass(String className, byte[] b, ClassLoader loader, ProtectionDomain protectionDomain) throws Exception {
    Object[] args;
    Class c;
    //获取字节码类型
    if (DEFINE_CLASS != null) {
    // 其中b 为cglib生成的字节数组
        args = new Object[]{className, b, new Integer(0), new Integer(b.length), protectionDomain};
        c = (Class)DEFINE_CLASS.invoke(loader, args);
    } else {
        if (DEFINE_CLASS_UNSAFE == null) {
            throw new CodeGenerationException(THROWABLE);
        }
        args = new Object[]{className, b, new Integer(0), new Integer(b.length), loader, protectionDomain};
        c = (Class)DEFINE_CLASS_UNSAFE.invoke(UNSAFE, args);
    }
    //正式加载到jvm内存
    Class.forName(className, true, loader);
    return c;
}
```

图 7 defineClass

通过 Class.forName 方法将 Class 对象装载到 JVM

依旧是图 7 的代码，在 defineClass 的最后调用了 Class.forName 方法传入 className 以及加载器，将 Class 对象装载到 JVM 中以供使用。以后这个 Class 对象就代替了源 Class 对象，完成 AOP 增强的操作。

4、总结

本节课聚焦在 Spring AOP CGLIB 的动态代理的源码分析，首先通过一个例子回顾 CGLIB 的应用是如何实现的，然后通过源码分析将 CGLIB 中 Class 对象创建过程做了讲解，包括生成指定类的 Class 对象字节数组，将 Class 对象字节数组转换为 Class 对象，通过 Class.forName 方法将 Class 对象装载到 JVM。

下节课介绍 Spring AOP 和 AspectJ AOP 存的区别。下期见，拜拜。