一文搞懂Java动态代理：为什么Mybatis Mapper不需要实现类?

在学习Java动态代理之前，我想让大家先思考这样几个问题。

* JDK动态代理为什么不能对类进行代理?
* Mybatis Mapper接口为什么不需要实现类?

如果你还不知道上述问题的答案，那么这篇文章一定能消除你心中的疑惑。喜欢“IT果果日记”文章的朋友建议收藏+，方便以后复习查阅。如需转载请注明文章来源及原地址。支持原创，侵权必究。

目录

代理模式

说到Java动态代理，就不得不提代理模式。为什么要使用代理模式呢?

生活中对代理模式的使用无处不在，例如明星经纪人对明星业务的代理；律师对原告官司的代理；4s店对汽车制造商的销售代理等等。这些使用场景告诉我们代理模式的本质是：

代理对象为被代理对象的某种行为提供增强服务。

如何理解“增强”两个字？它的含义其实可以理解为“访问控制”或“业务托管”。所以现在你能告诉我代理模式的作用了吗？在某些不适合直接访问目标对象的情况下，代理对象可以为目标对象提供访问控制和业务托管。

mark:3024:0:0:0:q75.awebp#?w=1289&h=661&s=243453&e=png&b=ffffff)

代理三要素

* 共同的行为。即接口；
* 目标对象。即被代理对象或业务对象 ，目标对象实现了接口；
* 代理对象。即代理对象或增强对象，增强了目标对象的行为。

既然代理模式的作用是访问控制和业务托管，那么将这种模式映射到面向对象的模型里去如何理解?

基于代理模式的特点，我们明白了代理对象如果想要对目标对象的行为增强，首先，它们必须有共同的行为，在代码里我们可以让它们实现同一个接口；其次，在实现目标对象的业务时，代理对象不是自己去实现，而是以某种方式调用目标对象的实现方法，一般常用的方式是将目标对象和代理对象组合在一起，代理对象调用目标对象的方法。

静态代理

实际上代理模式还可以具体划分为静态代理和动态代理。生活中的大多数代理都是类似于静态代理的设计。静态代理有以下几个特点：

特点

* 目标对象固定，在执行前就能确定目标对象
* 代理对象会对目标对象的行为增强
* 每个目标对象都需要一个代理，会造成代理泛滥

代码实现

我们结合代码看下静态代理的实现原理。就拿律师代理为例，首先我们创建一个接口作为原告和律师共同的行为----收集打官司的证据。

```
public interface LawEvidence {
    void collect();
}
```

如果是原告自己为打官司收集证据，我们创建上面接口的实现类作为目标对象。

```
public class LawEvidenceImpl implements LawEvidence {
    @Override
    public void collect() {
        System.out.println("原告收集证据！");
    }
}
```

此时原告找到自己的律师，让律师代理自己收集证据时，我们创建代理类。

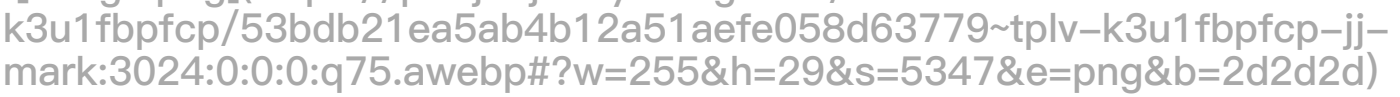
```
public class LawEvidenceProxy implements LawEvidence {
    private LawEvidence lawEvidence;

    public LawEvidenceProxy(LawEvidence lawEvidence) {
        this.lawEvidence = lawEvidence;
    }

    @Override
    public void collect() {
        System.out.print("律师向原告了解案情，并代替" );
        this.lawEvidence.collect();
    }
}
```

新建一个客户端，看下如何使用代理律师给原告收集打官司的证据。

```
public class Client {
    public static void main(String[] args) {
        LawEvidence lawEvidenceProxy = new LawEvidenceProxy(new
LawEvidenceImpl());
        lawEvidenceProxy.collect();
    }
}

```

运行结果如图所示。客户端里我们创建了一个目标对象LawEvidenceImpl，然后封装到代理对象LawEvidenceProxy里，调用代理对象的收集证据方法。

看下运行结果。本来应该显示原告自己收集证据的，但是这里使用了静态代理，所以就变成了“律师向原告了解案情，并代替原告收集证据”。



客户端的代码告诉我们，虽然静态代理LawEvidenceProxy能够对目标对象LawEvidenceImpl增强，但是如果现在我想要再对原告增强一个其他的行为，例如律师代替原告打官司，这个时候就不得不新增一个打官司的代理对象，如果代理的行为越来越多，就会造成代理泛滥。

动态代理

面向对象的代码世界必然比生活中的应用灵活度要更高一些。为了解决静态代理的代理泛滥的问题，我们经常会用到或者看别人用到动态代理模式，例如路由器对光猫访问互联网的代理、Mybatis插件对Mybatis执行器的代理、Mybatis动态Sql，Spring AOP等都是对动态代理的实践。归结起来，动态代理有以下几个特点：

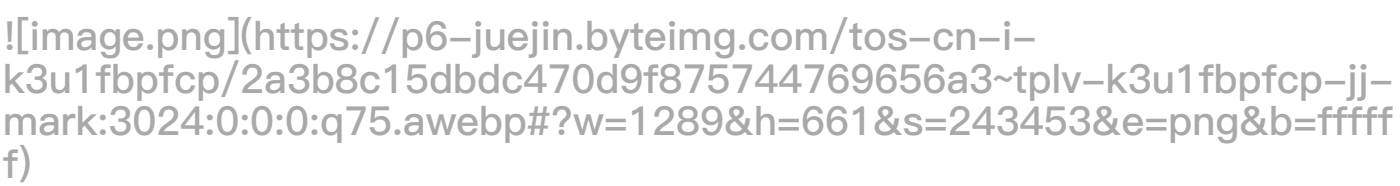
特点

* 目标对象不确定，在执行时动态创建
* 代理对象会对目标对象的行为增强

两者的区别

由动态代理和静态代理的特点，我们能够很轻易的得出一个结论：它们最大不同点是目标对象在执行前是否确定。

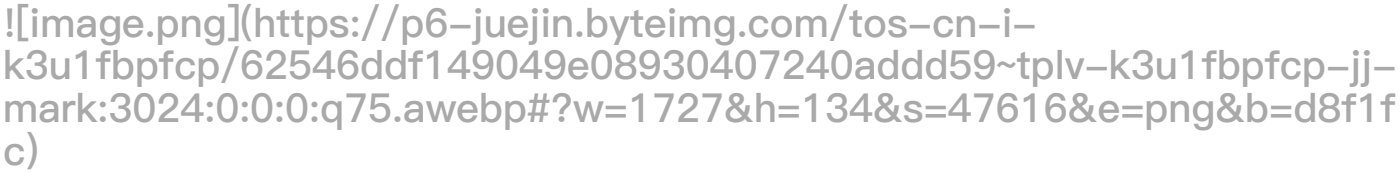
如何理解“目标对象不确定”这句话？我们再来回顾一下静态代理模式的类图。Proxy（代理）包含的属性是realSubject对象，即目标对象，它并不是一个抽象的实体。那我们如果将realSubject换成Subject接口会怎么样呢？这下应该是动态代理模式了吧？但是这样又会有一个问题，Subject接口的doOperation()方法是固定的。所以为了解决这个问题，我们需要在运行时动态构造一个目标对象，并将它封装到一个动态的代理对象里。



doOperation()方法如果想要通用必须至少满足以下三点：

* 目标对象是谁?
* 调用目标对象的方法是什么?
* 调用目标对象的方法参数是什么?

只有知道了这三点，动态代理模式就能像静态代理模式一样调用不同的目标对象方法啦。而这也就是JDK动态代理中InvocationHandler的原理精髓之所在。可以说只要实现了InvocationHandler接口，就能让其自身达到通用目标对象的标准，以达到被通用代理对象使用的目的。



JDK动态代理

动态代理主要有两种实现方式：

* JDK动态代理
* CGLIB动态代理

我们先来看下JDK动态代理的实现。还是使用律师代理的示例作为蓝本，之前原告找律师代理的是收集证据的行为，现在如果想要让律师代理原告打官司，如何实现?

我们再创建一个打官司的接口，将它作为目标对象和代理对象的共同行为。

```
public interface Lawsuit {
    void lawsuit();
}
```

接着创建一个原告自己打官司的实现类作为目标对象。

```
public class LawsuitImpl implements Lawsuit {
    @Override
    public void lawsuit() {
        System.out.println("原告打官司！");
    }
}
```

如果是采用静态代理模式，我们需要依葫芦画瓢给打官司的行为再创建一个代理类。

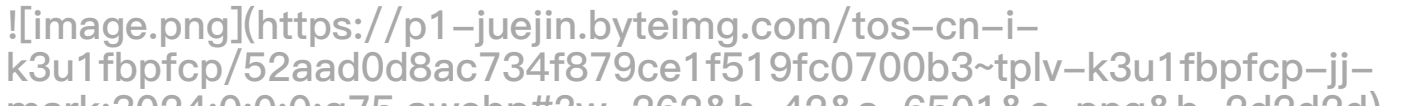
```
public class LawsuitProxy implements Lawsuit {
    private Lawsuit lawsuit;

    public LawsuitProxy(Lawsuit lawsuit) {
        this.lawsuit = lawsuit;
```

```
    }

    @Override
    public void lawsuit() {
        System.out.print("律师向原告了解案情，并代替" );
        this.lawsuit.lawsuit();
    }
}

```

我们可以写一个客户端看看采用静态代理模式，对“收集证据”和“打官司”的行为代理后是什么效果。

```
public class Client {
    public static void main(String[] args) {
        // 收集证据
        LawEvidence lawEvidenceProxy = new LawEvidenceProxy(new LawEvidenceImpl());
        lawEvidenceProxy.collect();
        // 打官司
        LawsuitProxy lawsuitProxy = new LawsuitProxy(new LawsuitImpl());
        lawsuitProxy.lawsuit();
    }
}

```

运行结果如图所示。



这个结果完全符合我们预期。因为我们提供了两个代理给原告服务。但是如果原告还需要找律师代理其他业务，难道又要创建新的代理实现类吗？这样显然会造成代理泛滥。所以这一次我们试试JDK动态代理的实现方式。

```
public class LawHandler implements InvocationHandler {
    private Object target;
```

```
    public LawHandler(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
        System.out.print("律师向原告了解案情，并代替");
        method.invoke(target, args);
        return null;
    }
}
```

这次我们只需要实现一个InvocationHandler接口。在增强的invoke()方法中，让律师代替原告并调用了原告的方法。写个客户端看看执行效果。

```
public class Client {
    public static void main(String[] args) {
        // 收集证据
        LawEvidence lawEvidence = new LawEvidenceImpl();
        InvocationHandler evidenceHandler = new
LawHandler(lawEvidence);
        LawEvidence evidenceProxy = (LawEvidence)
Proxy.newProxyInstance(lawEvidence.getClass().getClassLoader(),
                lawEvidence.getClass().getInterfaces(), evidenceHandler);
        evidenceProxy.collect();
        // 打官司
        Lawsuit lawsuit = new LawsuitImpl();
        InvocationHandler lawsuitHandler = new LawHandler(lawsuit);
        Lawsuit lawsuitProxy = (Lawsuit)
Proxy.newProxyInstance(lawsuit.getClass().getClassLoader(),
                lawsuit.getClass().getInterfaces(), lawsuitHandler);
        lawsuitProxy.lawsuit();
    }
}
```

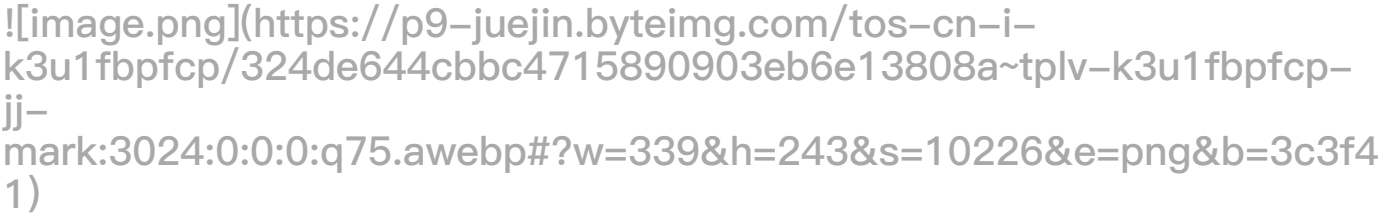
如图所示，和静态代理实现的结果一模一样。

k3u1fbpfcp/2ac29f4e757c449c9f3f7c249670f7e8~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=373&h=60&s=3897&e=png&b=2c2c2c)

观察客户端的代码可以发现，不论是收集证据还是打官司，亦或是后面如果想要再增加其他代理业务，都只需要两个步骤即可实现代理增强。

* 实现InvocationHandler接口。如上面例子中的LawHandler，如果代理增强的逻辑是一样的话，都可以用这个处理器类。
* 使用Proxy#newProxyInstance()生成一个代理类，它会返回代理对象。这个代理对象已经不是从前自己创建的目标对象了，它是被增强过的。例如上例中的evidenceProxy变量就是对lawEvidence变量的增强。

看下代码结构可以看的更加清楚，静态代理模式下有多少代理业务就创建多少代理对象（红框标注）。而JDK动态代理模式下可以共用一个LawHandler处理器（绿框标注），因为它在构造函数里的目标对象参数是抽象的。



反编译

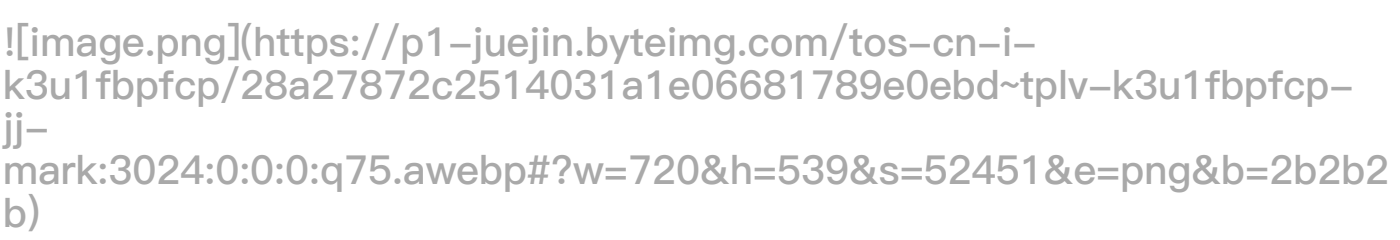
JDK动态代理的实现原理是怎么样的呢？我们可以在客户端main()方法的开头加上一行代码，目的是将Proxy生成的代理类写到本地磁盘里，这样我们就能看到代理类长什么样了。

```
System.getProperties().setProperty("sun.misc.ProxyGenerator.saveGeneratedFiles", "true");
```

如图所示，文件夹里多了两个class文件Proxy0和Proxy0和Proxy0和Proxy1。动态代理对象的名称是有规律的，它们都以$Proxy前缀开头，后面跟着数字。

k3u1fbpfcp/512f3f224d92448abdfa8687c4d11249~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=260&h=103&s=2959&e=png&b=3c3f41)

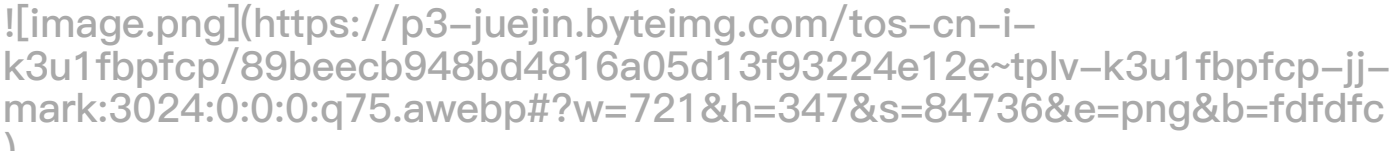
我把Proxy0关键的代码贴出来大家看下就一目了然了，Proxy0关键的代码贴出来大家看下就一目了然了，Proxy0关键的代码贴出来大家看下就一目了然了，Proxy1的原理和$Proxy0类似。



$Proxy0是Proxy的子类，并且实现了LawEvidence接口，这样它既可以是代理对象又可以是目标对象。这就能解释本文一开始就提出的一个问题：JDK动态代理为什么不能对类进行代理？因为在Java语言里不能多继承，所以Proxy#newProxyInstance()生成的对象既然已经默认继承了Proxy类，就不能再继承别的类了。因此这里通过对接口进行代理达到多态的效果。如果实在想要代理对象怎么办呢？后面介绍CGLIB时会提到，CGLIB动态代理支持对类的代理。

再来观察collect()方法，它通过调用InvocationHandler（变量h）的invoke()方法实现。之前提到过，invoke()方法的参数m3就是目标方法，它利用静态代码块在Proxy.class对象构建的时候就初始化了。

下图是JDK动态代理的链路图，从整体上梳理了动态代理的流程。



Mybatis Mapper

上面JDK动态代理的例子实现了被代理接口LawEvidence，但是众所周知，Mabatis动态Sql只需要一个Mapper接口及其对应的XML配置，并不需要实现类。那么Mybatis是如何运用JDK动态代理实现JDBC操作的呢?

要想弄清楚这个问题，我们首先得知道为什么Mybatis Mapper不需要实现类?

这要从Mybatis的职责说起，Mybatis是用来干什么的？Mybatis在Service层与数据库之间起到了桥梁的作用，你也可以理解为Mybatis是Service层访问数据库的代理。Mybatis为Service层访问数据库的行为提供了便捷的接口，便捷到Service层可以完全忽略JDBC的存在。

Mybatis包圆了一切与JDBC的交互：

> 加载驱动是它
>
>
> 建立连接是它
>
>
> 创建Statement是它
>
>
> 构建SQL语句是它
>
>
> 执行SQL是它
>
>
> 返回结果还是它
>
>
> ...
>
>
> 是它是它就是它，它是我们的英雄Mybatis
>

>
> 啧啧啧，不押韵啊

Service层再也不用从零开始一步一步的与JDBC建立联系。这已经不是对访问JDBC的增强了，这完全就是代替Service层把事情都干了，干的任劳任怨，干的漂漂亮亮，不让Service层做一点重复劳动。

从开发者的角度来说，Mybatis Mapper接口也不应该有实现类，如果每个Mapper接口都需要单独创建一个实现类，那么使用Mybatis框架的项目会变得非常的臃总且不够优雅。

Mybatis是如何做到没有实现类就可以完成动态代理的呢?

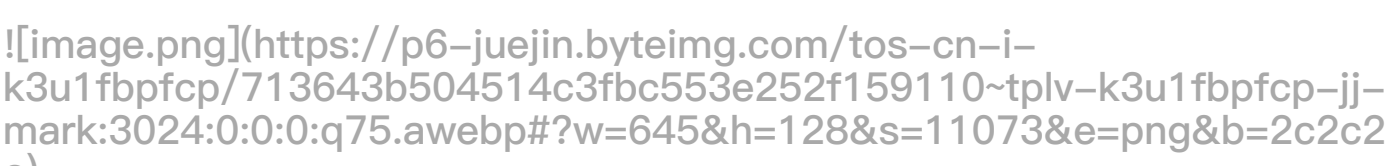
我们可以看看Mybatis源码是怎么写的。先写一个简单的测试代码。

```
@Test
public void main() throws IOException {
    String resource = "mybatis-config.xml";
    InputStream inputStream =
Resources.getResourceAsStream(resource);
    SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
    SqlSession session = sqlSessionFactory.openSession();
    try {
        UserMapper mapper = session.getMapper(UserMapper.class);
        User user = mapper.selectUserById(1);
        System.out.println(user);
    } finally {
        session.close();
    }
}
```

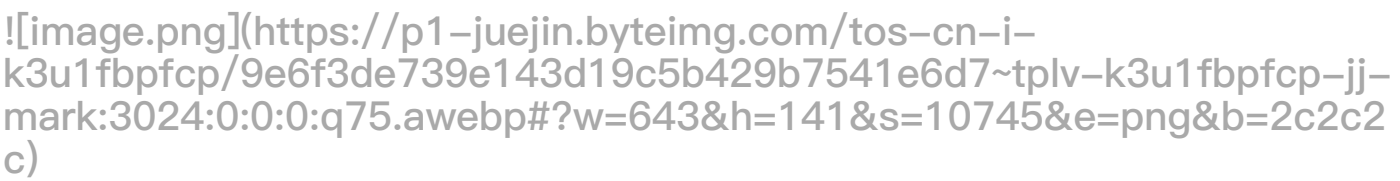
Mybatis执行一次Sql，首先要读取配置文件，通过配置文件构建

SqlSessionFactory，从而得到SqlSession，SqlSession是Mybatis提供的面向用户的Api，它是与数据库交互的会话接口，调用它的getMapper()方法就能得到Mapper接口，并使用Mapper接口执行XML中配置的动态Sql。

如上例中的UserMapper接口，它没有实现类，只定义了selectUserById()方法。前面我们学习了JDK动态代理，很容易想到这里用session#getMapper()方法获取到的UserMapper对象应该是一个动态代理对象，它是对UserMapper这个目标接口的增强。所以我们点进去看个究竟。



Mybatis默认实现了SqlSession接口，可以看到DefaultSqlSession#getMapper()方法里是调用了Configuration#getMapper()方法，代码继续往下走。

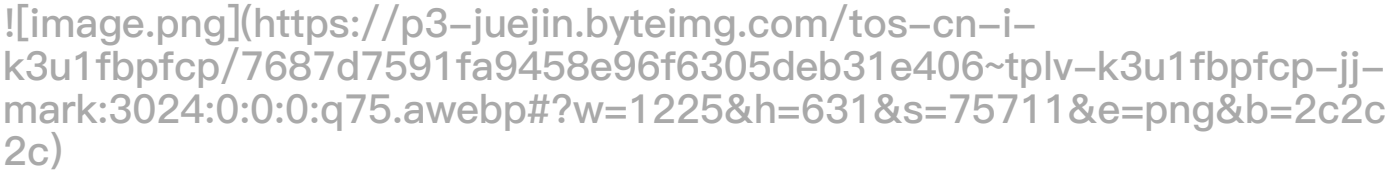


Configuration#getMapper()方法调用的是MapperRegistry#getMapper()方法。MapperRegistry是Configuration里的一个专门用于注册Mapper接口信息的类。MapperRegistry会将Mapper接口的Class对象与MapperProxyFactory对象建立联系，MapperProxyFactory对象可以创建Mapper接口的动态代理对象。看到这里就快要接近真相了，通过Mapper接口的Class对象我们可以从配置中获取到Mybatis的Mapper动态代理对象工厂，从而构建动态代理对象。

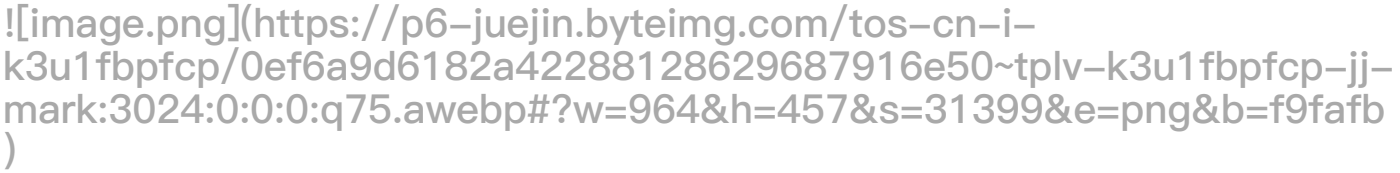
继续往后看，进入MapperRegistry#getMapper()方法后，通过Mapper接口的Class对象查询到其对应的MapperProxyFactory对象，调用MapperProxyFactory#newInstance()方法创建Mapper接口的代理对象。

k3u1fbpfcp/f7d1f3fef3944643931125538bafb3cb~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1019&h=524&s=64710&e=png&b=2b2b2b)

MapperProxyFactory#newInstance()方法主要就干了一件事情，这个代码我们已经很熟悉了，就是利用Proxy#newProxyInstance()方法生成动态代理对象。但是仔细观察你会发现绿色框标注的这部分代码和前文中律师动态代理打官司的代码写法有点不一样。

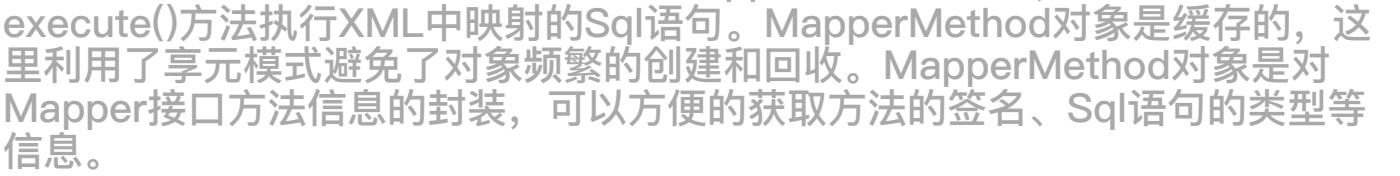


回顾一下律师动态代理打官司的代码，在调用Proxy#newProxyInstance()方法创建动态代理对象时，第二个参数（接口数组）是从Lawsuit实现类的Class数组获取到的。而在MapperProxyFactory中没有实现类，直接new了一个Class数组，数组元素由Mapper接口组成。现在可以得出结论，动态代理有实现类和无实现类的第一个区别是目标接口赋值的方式不一样，前者通过目标接口实现类的getInterfaces()方法获取；后者通过new一个Mapper接口的Class数组赋值。



动态代理有实现类和无实现类的第二个区别在于对InvocationHandler#invoke()方法的调用，前者不仅实现了增强，还通过反射调用了实现类的接口；后者仅仅实现了增强，而没有调用实现类接口。

Proxy#newProxyInstance()方法的第三个参数是传一个InvocationHandler接口，Mybatis使用的是MapperProxy这个实现类。MapperProxy#invoke()方法中绿色框前面的部分不用管，一般不会进入这里，重点看绿色框里的代码。这段代码的意思是根据method创建一个MapperMethod对象，并调用其execute()方法执行XML中映射的Sql语句。MapperMethod对象是缓存的，这里利用了享元模式避免了对象频繁的创建和回收。MapperMethod对象是对Mapper接口方法信息的封装，可以方便的获取方法的签名、Sql语句的类型等信息。



k3u1fbpfcp/685c95e3b19b4a78be113a07f7b7217f~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1282&h=843&s=104868&e=png&b=2b2b2b)

可以看到MapperMethod#execute()方法并没有任何Mapper接口实现类的逻辑。

![image.png](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/86fad445201947fbb8e1e985a4f19451~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=997&h=910&s=110485&e=png&b=2b2b2b)

Mybatis Mapper动态代理的调用时序图如下图，现在看起来是不是变得非常的简单。

![image.png](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/677f0c4d5aea43088984c097d998b321~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1321&h=748&s=492819&e=png&b=fbf5f3)

现在可以解答文章开头的其中一个问题啦，Mybatis Mapper接口为什么不需要实现类？因为执行Sql所需要的所有的JDBC操作都在Mybatis的MapperProxy中实现了，所以不需要实现类。

介绍动态代理就不得不聊一下CGLIB，但是由于篇幅的原因，“IT果果日记”将在另外一篇文章里单独介绍CGLIB的实现及其原理以及CGLIB一个隐藏的很深的坑。感兴趣的朋友可以收藏+，持续“IT果果日记”的动态。