

## 16 | Compiler编译：神乎其神的编译你是否有过胆怯？

2023-01-23 何辉 来自北京



天下无鱼

<https://shikey.com/>

《Dubbo源码剖析与实战》

[课程介绍 >](#)



讲述：何辉

时长 13:07 大小 11.98M



你好，我是何辉。首先祝你新年快乐。

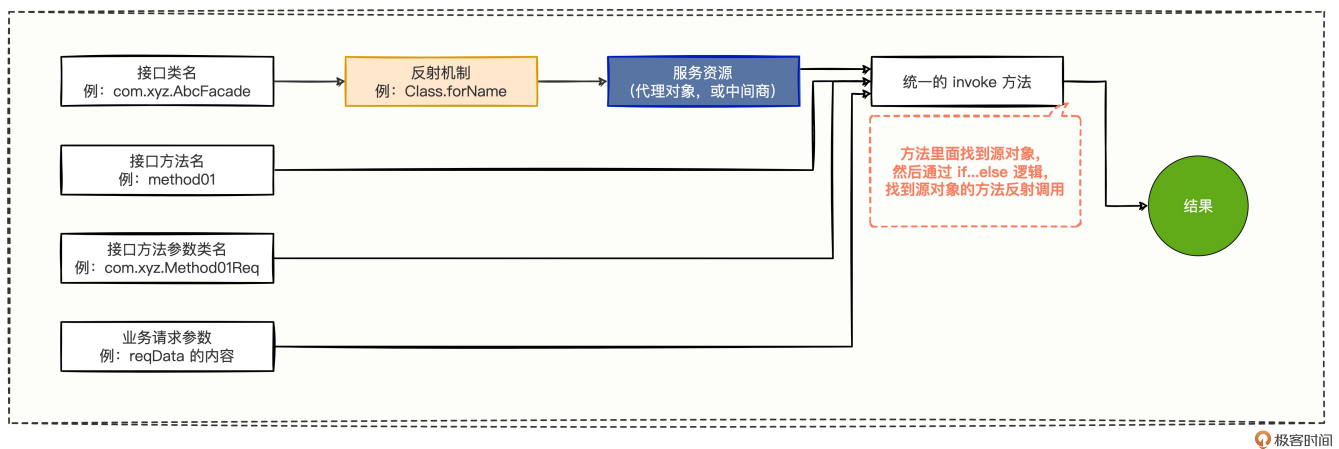
今天是我们深入研究 Dubbo 源码的第五篇，Compiler 编译。

在“[点点直连](#)”中，我们实现了一套万能管控平台来修复数据，其中就有通过市场上的 Groovy 插件编译 Java 源代码来生成类信息。

而上一讲“[Wrapper 机制](#)”中，在实现自定义代理的时候，我们也使用了 JavaCompiler 来编译源代码，只不过编译的时候，借助了磁盘上的 class 文件才得以生成类信息。

掌握了这两种动态编译方式，相信你在动态编译这块已经有了一定的基础，如果你还是觉得有点胆怯，今天我们上点难度，再学习 2 种 Compiler 方式，帮助你在底层框架开发层面拥有更强大的技术支撑。

还是以上一讲的自定义代理为例。这张图你应该还有印象，我们尝试通过添加一层代理的方式，把各种 if...else 的硬编码逻辑转变为动态生成：



在实现自定义代理的过程中，我们采用的是最纯粹的字符串拼接的方式，拼接出了动态的源代码，虽然实用，但是写起来也费劲。

有没有通过 set 或 get 操作就能实现创建类的简单方式，来改造图中的代理实现过程呢？

我们在脑内检索一番，平常都是直接将编写好的代码交给 Javac 编译器去编译的，现在要通过某种工具简单的进行 set 或 get 动态创建一个类，怎么办呢，突然灵光一闪，在上一讲“Wrapper 机制的原理”代码流程中，我们看到了一段 makeClass 的样例代码，难道 Dubbo 已经有了类似的先进操作么？

不管真假，我们先去验证看看，找到了相应的代码：

复制代码

```
1 // org.apache.dubbo.common.bytecode.ClassGenerator#toClass(java.lang.Class<?>,
2 public Class<?> toClass(Class<?> neighborClass, ClassLoader loader, ProtectionD
3     if (mCtc != null) {
4         mCtc.detach();
5     }
6     // 自增长类名尾巴序列号，类似 $Proxy_01.class 这种 JDK 代理名称的 01 数字
7     long id = CLASS_NAME_COUNTER.getAndIncrement();
8     try {
9         // 从 ClassPool 中获取 mSuperClass 类的类型
10        // 我们一般还可以用 mPool 来看看任意类路径对应的 CtClass 类型对象是什么
11        // 比如可以通过 mPool.get("java.lang.String") 看看 String 的 CtClass 类型对
12        // 之所以要这么做，主要是为了迎合这样的API语法而操作的
```

```

13 CtClass ctcs = mSuperClass == null ? null : mPool.get(mSuperClass);
14 if (mClassName == null) {
15     mClassName = (mSuperClass == null || javassist.Modifier.isPublic(ct
16         ? ClassGenerator.class.getName() : mSuperClass + "$sc") + i
17 }
18 // 通过 ClassPool 来创建一个叫 mClassName 名字类
19 mCtc = mPool.makeClass(mClassName);
20 if (mSuperClass != null) {
21     // 然后设置一下 mCtc 这个新建类的父类为 ctcs
22     mCtc.setSuperclass(ctcs);
23 }
24 // 为 mCtc 新建类添加一个实现的接口
25 mCtc.addInterface(mPool.get(DC.class.getName())); // add dynamic class
26 if (mInterfaces != null) {
27     for (String cl : mInterfaces) {
28         mCtc.addInterface(mPool.get(cl));
29     }
30 }
31 // 为 mCtc 新建类添加一些字段
32 if (mFields != null) {
33     for (String code : mFields) {
34         mCtc.addField(CtField.make(code, mCtc));
35     }
36 }
37 // 为 mCtc 新建类添加一些方法
38 if (mMethods != null) {
39     for (String code : mMethods) {
40         if (code.charAt(0) == ':') {
41             mCtc.addMethod(CtNewMethod.copy(getCtMethod(mCopyMethods.ge
42                 code.substring(1, code.indexOf('(')), mCtc, null));
43         } else {
44             mCtc.addMethod(CtNewMethod.make(code, mCtc));
45         }
46     }
47 }
48 // 为 mCtc 新建类添加一些构造方法
49 if (mDefaultConstructor) {
50     mCtc.addConstructor(CtNewConstructor.defaultConstructor(mCtc));
51 }
52 if (mConstructors != null) {
53     for (String code : mConstructors) {
54         if (code.charAt(0) == ':') {
55             mCtc.addConstructor(CtNewConstructor
56                 .copy(getCtConstructor(mCopyConstructors.get(code.s
57         } else {
58             String[] sn = mCtc.getSimpleName().split("\\$+"); // inner
59             mCtc.addConstructor(
60                 CtNewConstructor.make(code.replaceFirst(SIMPLE_NAME
61         }
62     }
63 }
64 // 将 mCtc 新建的类转成 Class 对象

```

```
65     try {
66         return mPool.toClass(mCtc, neighborClass, loader, pd);
67     } catch (Throwable t) {
68         if (!(t instanceof CannotCompileException)) {
69             return mPool.toClass(mCtc, loader, pd);
70         }
71         throw t;
72     }
73 } catch (RuntimeException e) {
74     throw e;
75 } catch (NotFoundException | CannotCompileException e) {
76     throw new RuntimeException(e.getMessage(), e);
77 }
78
```



凭着印象从代码中找到了一些特别有趣的 API，比如 `setSuperclass` 设置父类属性、`addInterface` 添加实现类属性、`addField` 添加字段、`addMethod` 添加方法、`addConstructor` 添加构造方法等等。刚刚还在寻思，有没有一种简单的赋值操作的方式来创建类，结果就发现了意外收获，从名字上看，感觉这些 API 都和类创建有关，好像都能用上。

我们继续深挖，发现**这些 API 都是属于 Javassist 插件中的**，这就难怪了，既然该插件能提供这样的 API 来创建类，如果不将这些类最终编译成为 Class 类信息，未免也有点做事做半截的感觉。

不过这只是我们从源码层面的推测，到底有没有这个效果呢，还是要确认一下。至于如何确认，我给个小小的建议，如果你在源码中发现一些比较感兴趣的插件，可以去官方网站大致了解一下，也许疑惑和推测也就烟消云散了。

我们进入 [🔗 Javassist 的官网](https://jvassit.github.io/)会看到这段英文解释：

Javassist (Java Programming Assistant) makes Java bytecode manipulation simple. It is a class library for editing bytecodes in Java; it enables Java programs to define a new class at runtime and to modify a class file when the JVM loads it. Unlike other similar bytecode editors, Javassist provides two levels of API: source level and bytecode level. If the users use the source-level API, they can edit a class file without knowledge of the specifications of the Java bytecode. The whole API is designed with only the vocabulary of the Java language. You can even specify inserted bytecode in the form of source text; **Javassist compiles it on the fly**. On the other hand, the bytecode-level API allows the users to directly edit a class file as other editors.

大致含义就是，Javassist 让用 Java 编辑字节码变为一件非常简单的事情，众多开发人员可以在不懂字节码规范的情况下，针对字节码文件进行编辑修改，改完之后 Javassist 可以实时编译它。



这下验证了我们刚才对源码的猜想。是不是很兴奋又接触到了新技术，别着急，我们还是先看 [官网关于 API 介绍的教程](#)。

基本了解如何使用之后，上一讲的代码模板，我们可以用 Javassist 实现一遍，代码如下：

复制代码

```
1  //////////////////////////////////////
2  // 采用 Javassist 的 API 来动态创建代码模板
3  //////////////////////////////////////
4  public class JavassistProxyUtils {
5      private static final AtomicInteger INC = new AtomicInteger();
6      public static Object newProxyInstance(Object sourceTarget) throws Exception
7          // ClassPool: Class对象的容器
8          ClassPool pool = ClassPool.getDefault();
9
10         // 通过ClassPool生成一个public类
11         Class<?> targetClazz = sourceTarget.getClass().getInterfaces()[0];
12         String proxyClassName = "$" + targetClazz.getSimpleName() + "CustomInvoke";
13         CtClass ctClass = pool.makeClass(proxyClassName);
14         ctClass.setSuperclass(pool.get("com.hmilyylimh.cloud.compiler.custom.Cu");
15
16         // 添加方法 public Object invokeMethod(Object instance, String mtdName,
17         CtClass returnType = pool.get("java.lang.Object");
18         CtMethod newMethod=new CtMethod(
19             returnType,
20             "invokeMethod",
21             new CtClass[]{ returnType, pool.get("java.lang.String"), pool.g
22             ctClass);
23         newMethod.setModifiers(Modifier.PUBLIC);
24         newMethod.setBody(buildBody(targetClazz).toString());
25         ctClass.addMethod(newMethod);
26
27         // 生成 class 类
28         Class<?> clazz = ctClass.toClass();
29
30         // 将 class 文件写到 target 目录下，方便调试查看
31         String filePath = JavassistProxyUtils.class.getResource("/").getPath()
32             + JavassistProxyUtils.class.getPackage().toString().substring("
33         ctClass.writeFile(filePath);
34
35         // 反射实例化创建对象
36         return clazz.newInstance();
37     }
```

```


38 // 构建方法的内容字符串
39 private static StringBuilder buildBody(Class<?> targetClazz) {
40     StringBuilder sb = new StringBuilder("{\n");
41     for (Method method : targetClazz.getDeclaredMethods()) {
42         String methodName = method.getName();
43         Class<?>[] parameterTypes = method.getParameterTypes();
44         // if ("sayHello".equals(mtdName)) {
45         String ifHead = "if (\\"" + methodName + "\".equals($2)) {\n";
46         // return ((DemoFacade) instance).sayHello(String.valueOf(args[0]))
47         String ifContent = null;
48         // 这里有 bug，姑且就入参就传一个入参对象吧
49         if(parameterTypes.length != 0){
50             ifContent = "return ((" + targetClazz.getName() + ") $1)." + me
51         } else {
52             ifContent = "return ((" + targetClazz.getName() + ") $1)." + me
53         }
54         // }
55         String ifTail = "}\n";
56         sb.append(ifHead).append(ifContent).append(ifTail);
57     }
58     // throw new NoSuchMethodException("Method [" + mtdName + "] not found.
59     String invokeMethodTailContent = "throw new " + org.apache.dubbo.common
60     sb.append(invokeMethodTailContent);
61     return sb;
62 }
63 }

```



可以发现确实比拼接字符串简单多了，而且 API 使用起来也比较清晰明了，完全按照平常的专业术语命名规范，马上就能找到对应的 API，根本不需要花很多准备工作。

改造代码需要注意 3 点。

1. 在获取各种类对应的 CtClass 类型对象时，可以通过从 ClassPool 的 get 方法中传入类路径得到。
2. 在对方法的入参字段名进行逻辑处理时，就得替换成 \$ 占位符，方法中的 this 引用，用 \$0 表示，方法中的第一个参数用 \$1 表示，第二个参数用 \$2 表示，以此类推。
3. 若要重写的父类的方法，是否设置 @Override 属性不太重要，但是千万别为了重写而拿父类的 CtMethod 属性一顿乱改。

用新方案编译源代码后，我们验证一下结果，编写测试验证代码。

 复制代码

```

1 public static void main(String[] args) throws Exception {

```



```
2 // 创建源对象（即被代理对象）
3 DemoFacadeImpl demoFacade = new DemoFacadeImpl();
4 // 生成自定义的代理类
5 CustomInvoker invoker = (CustomInvoker) JavassistProxyUtils.newProxyInstance
6 // 调用代理类的方法
7 invoker.invokeMethod(demoFacade, "sayHello", new Class[]{String.class}, new
8 }
```

如预期所料，正常打印出了结果，没想到一句简短的 `CtClass.toClass()` 方法就帮我们快速编译代码并转成 `Class` 对象信息了，非常简单实用。

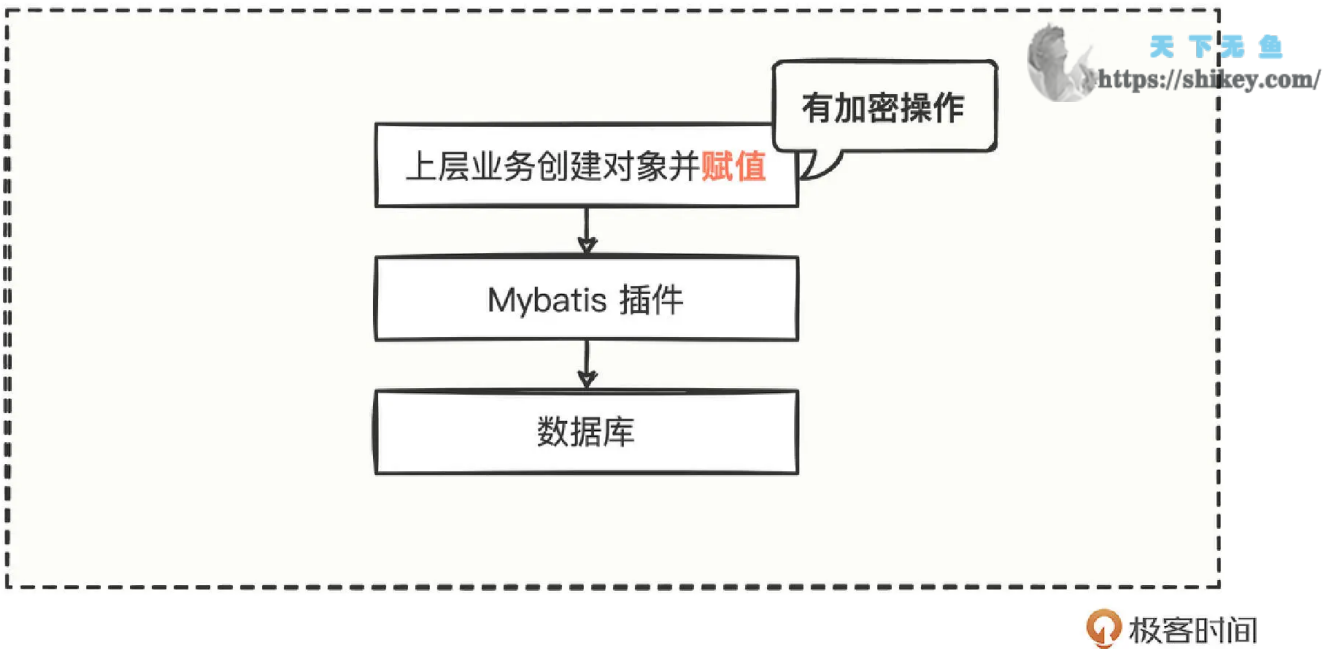
## ASM 编译

现在你有没有觉得信心大增，原来操作字节码这么简单，`Javassist` 简直是神器，有了它的存在，我想你再去理解那些 `Java Agent` 技术知识点，会发现在编译的世界修改字节码已经是小菜一碟了。

不过做技术我们讲究精益求精，既然 `Javassist` 这么好用，为什么公司的大佬还在用 `ASM` 进行操作呢？

其实，`ASM` 是一款侧重于性能的字节码插件，属于一种轻量级的高性能字节码插件，但同时实现的难度系数也会变大。这么讲你也许会好奇了，能有多难？

我们还是举例来看，例子是把敏感字段加密存储到数据库。



复制代码

```
1 public class UserBean {
2     private String name;
3     public UserBean(String name) { this.name = name; }
4     public String getName() { return name; }
5     public void setName(String name) { this.name = name; }
6     @Override
7     public String toString() { return "UserBean{name='" + name + "'}"; }
8 }
```

上层业务有一个对象，创建对象后，需要给对象的 `setName` 方法进行赋值。

如果想要给传入的 `name` 字段进行加密，一般我们会这么做。

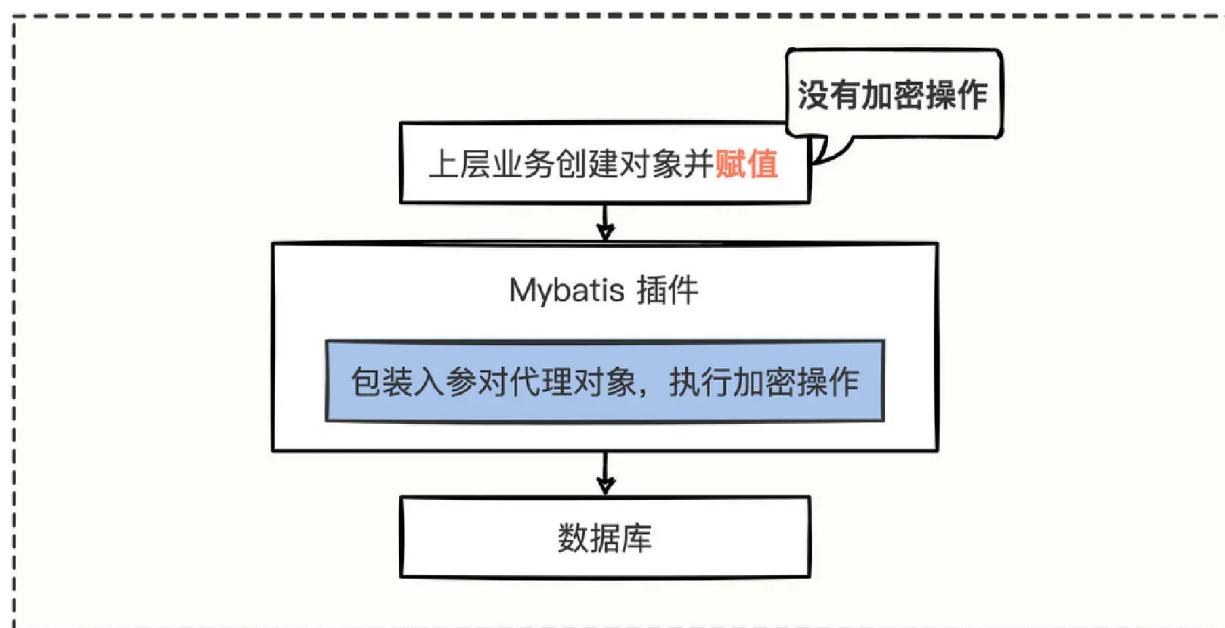
复制代码

```
1 // 先创建一个对象
2 UserBean userBean = new UserBean();
3 // 将即将赋值的 Geek 先加密，然后设置到 userBean 对象中
4 userBean.setName(AESUtils.encrypt("Geek"));
5 // 最后将 userBean 插入到数据库中
6 userDao.insertData(userBean);
```

把传入 `setName` 的值先进行加密处理，然后把加密后的值放到 `userBean` 对象中，在入库时，就能把密文写到数据库了。



但是这样就显得很累赘，今天这个字段需要加密，明天那个字段需要加密，那就没完没了，于是有人就想到了，可以将加密的这段操作内嵌到代理对象中，比如这样：



在上层业务中，该怎么赋值还是继续怎么赋值，不用感知加密的操作，所有加密的逻辑全部内嵌到代理对象中。当然，如果这么做，就得设计一个代码模板，借助自定义代理的经验，想必你也有了设计思路：

 复制代码

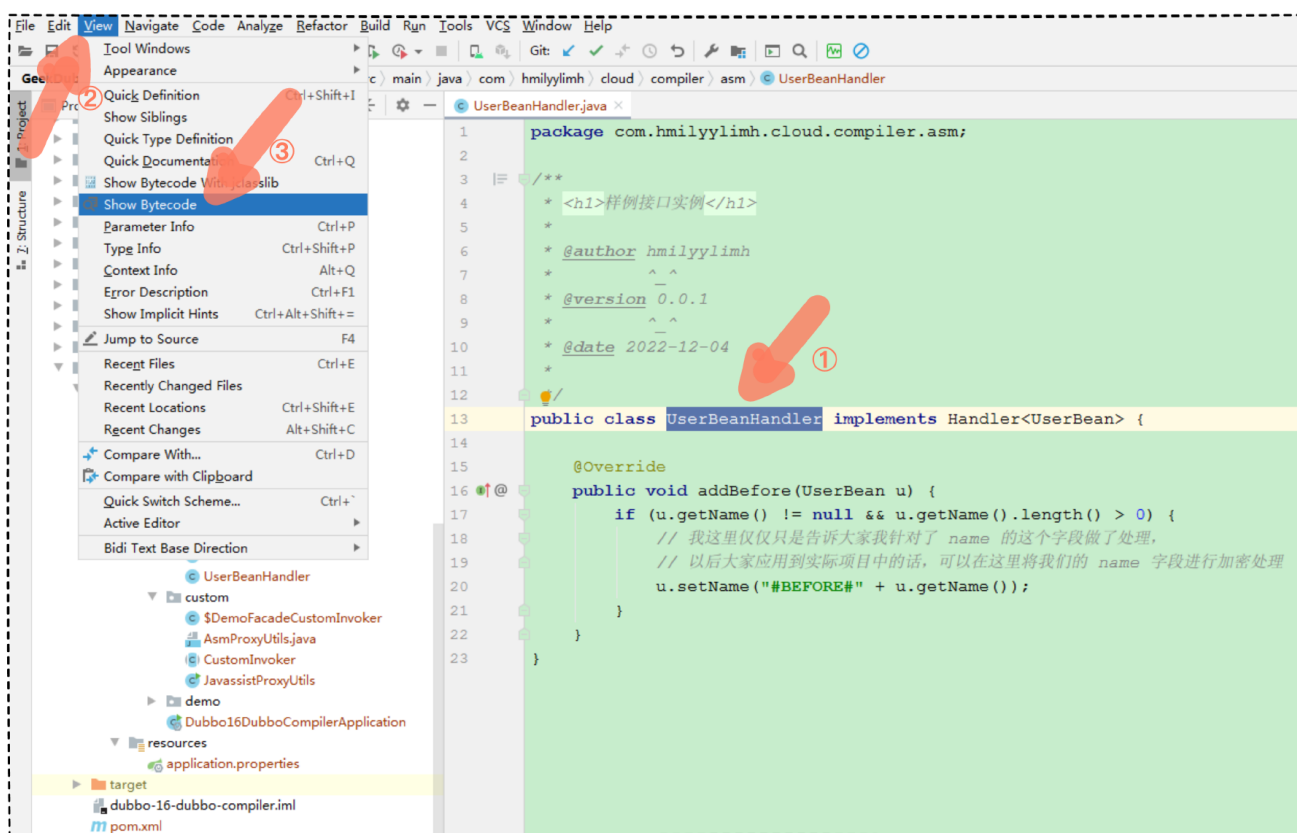
```
1 ///////////////////////////////////////////////////
2 // 代码模板，将 UserBean 变成了 UserBeanHandler 代理对象，并且实现一个自己定义的 Handler
3 ///////////////////////////////////////////////////
4 public class UserBeanHandler implements Handler<UserBean> {
5     @Override
6     public void addBefore(UserBean u) {
7         if (u.getName() != null && u.getName().length() > 0) {
8             // 我这里仅仅是告诉大家我针对了 name 的这个字段做了处理，
9             // 以后大家应用到实际项目中的话，可以在这里将我们的 name 字段进行加密处理
10            u.setName("#BEFORE#" + u.getName());
11        }
12    }
13 }
14
15 ///////////////////////////////////////////////////
16 // 配合代码模板设计出来的一个接口
17 ///////////////////////////////////////////////////
18 public interface Handler<T> {
19     public void addBefore(T t);
20 }
```

代码模板的思路也很简单，主要注意 2 点。

- 设计一个对象的代理类，暴露一个 `addBefore` 方法来将字段进行加密操作。
- 代理类为了迎合具备一个 `addBefore` 方法，就得设计出一个接口，避免 Java 类单继承无法扩展的瓶颈。

代码模板是定义好了，可是操作字节码的时候，去哪里弄到该 `UserBeanHandler` 的字节码呢？

其实 IDEA 工具已经为你预留了一个查看字节码的入口。



选中代码模板后，展开顶部的 View 菜单，选中 Show Bytecode 看到该类对应的字节码。

复制代码

```
1 // class version 50.0 (50)
2 // access flags 0x21
```

```

3 // signature java/lang/Object;Lcom/hmilyylimh/cloud/compiler/asm/Handler<Lcom/
4 // declaration: com/hmilyylimh/cloud/compiler/asm/UserBeanHandler implements cc
5 public class com/hmilyylimh/cloud/compiler/asm/UserBeanHandler extends Ljava/la
6
7 // compiled from: UserBeanHandler.java
8
9 // access flags 0x1
10 public <init>()V
11     ALOAD 0
12     INVOKESPECIAL java/lang/Object.<init> ()V
13     RETURN
14     MAXSTACK = 1
15     MAXLOCALS = 1
16
17 // access flags 0x1
18 public addBefore(Lcom/hmilyylimh/cloud/compiler/asm/UserBean;)V
19     ALOAD 1
20     INVOKEVIRTUAL com/hmilyylimh/cloud/compiler/asm/UserBean.getName ()Ljava/la
21     IFNULL L0
22     ALOAD 1
23     INVOKEVIRTUAL com/hmilyylimh/cloud/compiler/asm/UserBean.getName ()Ljava/la
24     INVOKEVIRTUAL java/lang/String.length ()I
25     IFLE L0
26     ALOAD 1
27     NEW java/lang/StringBuilder
28     DUP
29     INVOKESPECIAL java/lang/StringBuilder.<init> ()V
30     LDC "#BEFORE#"
31     INVOKEVIRTUAL java/lang/StringBuilder.append (Ljava/lang/String;)Ljava/lang
32     ALOAD 1
33     INVOKEVIRTUAL com/hmilyylimh/cloud/compiler/asm/UserBean.getName ()Ljava/la
34     INVOKEVIRTUAL java/lang/StringBuilder.append (Ljava/lang/String;)Ljava/lang
35     INVOKEVIRTUAL java/lang/StringBuilder.toString ()Ljava/lang/String;
36     INVOKEVIRTUAL com/hmilyylimh/cloud/compiler/asm/UserBean.setName (Ljava/lan
37     L0
38     FRAME SAME
39     RETURN
40     MAXSTACK = 3
41     MAXLOCALS = 2
42
43 // access flags 0x1041
44 public synthetic bridge addBefore(Ljava/lang/Object;)V
45     ALOAD 0
46     ALOAD 1
47     CHECKCAST com/hmilyylimh/cloud/compiler/asm/UserBean
48     INVOKEVIRTUAL com/hmilyylimh/cloud/compiler/asm/UserBeanHandler.addBefore (
49     RETURN
50     MAXSTACK = 2
51     MAXLOCALS = 2
52 }

```

看到一大片密密麻麻的字节码指令，想必你已经头都大了，不过别慌，这个问题在 [ASM 的官网指引](#) 中也解答了，我们只需要按部就班把字节码指令翻译成为 Java 代码就可以了。



好吧，既然官网都这么贴心了，那就勉强当一回工具人，我们按照官网的指示，依葫芦画瓢把代码模板翻译出来。

经过一番漫长的翻译之后，我们终于写出了自己看看都觉得头皮发麻的长篇大论的代码，关键位置我都加注释了。

复制代码

```
1  //////////////////////////////////////
2  // ASM 字节码操作的代理工具类
3  //////////////////////////////////////
4  public class AsmProxyUtils implements Opcodes {
5      /**
6       * <h2>创建代理对象。</h2>
7       *
8       * @param originClass: 样例: UserBean.class
9       * @return
10      */
11      public static Object newProxyInstance(Class originClass) throws Exception{
12          String newClzNameSuffix = "Handler";
13          byte[] classBytes = generateByteCode(originClass, newClzNameSuffix);
14
15          // 可以想办法将 classBytes 存储为一个文件
16          String filePath = AsmProxyUtils.class.getResource("/").getPath()
17              + AsmProxyUtils.class.getPackage().toString().substring("packag
18          FileOutputStream fileOutputStream = new FileOutputStream(new File(fileP
19              originClass.getSimpleName() + newClzNameSuffix + ".class"));
20          fileOutputStream.write(classBytes);
21          fileOutputStream.close();
22
23          // 还得把 classBytes 加载到 JVM 内存中去
24          ClassLoader loader = Thread.currentThread().getContextClassLoader();
25          Class<?> loaderClass = Class.forName("java.lang.ClassLoader");
26          Method defineClassMethod = loaderClass.getDeclaredMethod("defineClass",
27              String.class,
28              byte[].class,
29              int.class,
30              int.class);
31          defineClassMethod.setAccessible(true);
32          Object respObject = defineClassMethod.invoke(loader, new Object[]{
33              originClass.getName() + newClzNameSuffix,
34              classBytes,
35              0,
36              classBytes.length
37          });
```

```

38         // 实例化对象
39         return ((Class)respObject).newInstance();
40     }
41 }
42 /**
43  * <h2>生成字节码的核心。</h2><br/>
44  *
45  * <li><h2>注意：接下来的重点就是如何用asm来动态产生一个 UserBeanHandler 类。</h2></li>
46  *
47  * @param originClass: 样例: UserBean.class
48  * @param newClzNameSuffix: 样例: Handler
49  * @return
50  */
51 private static byte[] generateByteCode(Class originClass, String newClzName
52     String newClassSimpleNameAndSuffix = originClass.getSimpleName() + newC
53     /*****
54     // 利用 ASM 编写创建类文件头的相关信息
55     /*****
56     ClassWriter classWriter = new ClassWriter(0);
57     //////////////////////////////////////
58     // class version 50.0 (50)
59     // access flags 0x21
60     // signature Ljava/lang/Object;Lcom/hmilyylimh/cloud/compiler/asm/Handl
61     // declaration: com/hmilyylimh/cloud/compiler/asm/UserBeanHandler imple
62     // public class com/hmilyylimh/cloud/compiler/asm/UserBeanHandler exten
63     //////////////////////////////////////
64     classWriter.visit(
65         V1_6,
66         ACC_PUBLIC + ACC_SUPER,
67         Type.getInternalName(originClass) + newClzNameSuffix,
68         Type.getDescriptor(Object.class)+Type.getDescriptor(Handler.class),
69         Type.getDescriptor(Object.class),
70         new String[]{ Type.getInternalName(Handler.class) }
71     );
72     //////////////////////////////////////
73     // UserBeanHandler.java
74     //////////////////////////////////////
75     classWriter.visitSource(newClassSimpleNameAndSuffix, null);
76     /*****
77     // 创建构造方法
78     /*****
79     //////////////////////////////////////
80     // compiled from: UserBeanHandler.java
81     // access flags 0x1
82     // public <init>()V
83     //////////////////////////////////////
84     MethodVisitor initMethodVisitor = classWriter.visitMethod(
85         ACC_PUBLIC,
86         "<init>",
87         "()V",
88         null,
89         null

```



```
90     );
91     initMethodVisitor.visitCode();
92     //////////////////////////////////////
93     // ALOAD 0
94     // INVOKESPECIAL java/lang/Object.<init> ()V
95     // RETURN
96     //////////////////////////////////////
97     initMethodVisitor.visitVarInsn(ALOAD, 0);
98     initMethodVisitor.visitMethodInsn(INVOKEVIRTUAL,
99         Type.getInternalName(Object.class),
100         "<init>",
101         "()V"
102     );
103     initMethodVisitor.visitInsn(RETURN);
104     //////////////////////////////////////
105     // MAXSTACK = 1
106     // MAXLOCALS = 1
107     //////////////////////////////////////
108     initMethodVisitor.visitMaxs(1, 1);
109     initMethodVisitor.visitEnd();
110
111     /*****
112     // 创建 addBefore 方法
113     /*****
114     //////////////////////////////////////
115     // access flags 0x1
116     // public addBefore(Lcom/hmilyylimh/cloud/compiler/asm/UserBean;)V
117     //////////////////////////////////////
118     MethodVisitor addBeforeMethodVisitor = classWriter.visitMethod(
119         ACC_PUBLIC,
120         "addBefore",
121         "(" + Type.getDescriptor(originClass) + ")V",
122         null,
123         null
124     );
125     addBeforeMethodVisitor.visitCode();
126     //////////////////////////////////////
127     // ALOAD 1
128     // INVOKEVIRTUAL com/hmilyylimh/cloud/compiler/asm/UserBean.getName ()L
129     //////////////////////////////////////
130     addBeforeMethodVisitor.visitVarInsn(ALOAD, 1);
131     addBeforeMethodVisitor.visitMethodInsn(INVOKEVIRTUAL,
132         Type.getInternalName(originClass),
133         "getName",
134         "()" + Type.getDescriptor(String.class));
135     //////////////////////////////////////
136     // IFNULL L0
137     // ALOAD 1
138     // INVOKEVIRTUAL com/hmilyylimh/cloud/compiler/asm/UserBean.getName ()L
139     // INVOKEVIRTUAL java/lang/String.length ()I
140     // IFLE L0
141     //////////////////////////////////////
```

```
Label L0 = new Label();
addBeforeMethodVisitor.visitJumpInsn(IFNULL, L0);
addBeforeMethodVisitor.visitVarInsn(ALOAD, 1);
addBeforeMethodVisitor.visitMethodInsn(INVOKEVIRTUAL,
    Type.getInternalName(originClass),
    "getName",
    "()" + Type.getDescriptor(String.class));
addBeforeMethodVisitor.visitMethodInsn(INVOKEVIRTUAL,
    Type.getInternalName(String.class),
    "length",
    "()I");
addBeforeMethodVisitor.visitJumpInsn(IFLE, L0);
/*****
// 接下来要干的事情就是: u.setName("#BEFORE#" + u.getName());
*****/
////////////////////////////////////
// ALOAD 1
// NEW java/lang/StringBuilder
// DUP
////////////////////////////////////
addBeforeMethodVisitor.visitVarInsn(ALOAD, 1);
addBeforeMethodVisitor.visitTypeInsn(NEW, Type.getInternalName(StringBu
addBeforeMethodVisitor.visitInsn(DUP);
////////////////////////////////////
// INVOKESPECIAL java/lang/StringBuilder.<init> ()V
// LDC "#BEFORE#"
// INVOKEVIRTUAL java/lang/StringBuilder.append (Ljava/lang/String;)Lja
////////////////////////////////////
addBeforeMethodVisitor.visitMethodInsn(INVOKEVIRTUAL,
    Type.getInternalName(StringBuilder.class),
    "<init>",
    "()V");
addBeforeMethodVisitor.visitLdcInsn("#BEFORE#");
addBeforeMethodVisitor.visitMethodInsn(INVOKEVIRTUAL,
    Type.getInternalName(StringBuilder.class),
    "append",
    "(" + Type.getDescriptor(String.class) + ")" + Type.getDescripto
////////////////////////////////////
// ALOAD 1
// INVOKEVIRTUAL com/hmilyylimh/cloud/compiler/asm/UserBean.getName ()L
// INVOKEVIRTUAL java/lang/StringBuilder.append (Ljava/lang/String;)Lja
// INVOKEVIRTUAL java/lang/StringBuilder.toString ()Ljava/lang/String;
// INVOKEVIRTUAL com/hmilyylimh/cloud/compiler/asm/UserBean.setName (Lj
////////////////////////////////////
addBeforeMethodVisitor.visitVarInsn(ALOAD, 1);
addBeforeMethodVisitor.visitMethodInsn(INVOKEVIRTUAL,
    Type.getInternalName(originClass),
    "getName",
    "()" + Type.getDescriptor(String.class));
addBeforeMethodVisitor.visitMethodInsn(INVOKEVIRTUAL,
    Type.getInternalName(StringBuilder.class),
    "append",
```



```
194         "(" + Type.getDescriptor(String.class) + ")" + Type.getDescriptor
195         addBeforeMethodVisitor.visitMethodInsn(INVOKEVIRTUAL,
196             Type.getInternalName(StringBuilder.class),
197             "toString",
198             "()" + Type.getDescriptor(String.class));
199         addBeforeMethodVisitor.visitMethodInsn(INVOKEVIRTUAL,
200             Type.getInternalName(originClass),
201             "setName",
202             "(" + Type.getDescriptor(String.class) + ")V");
203         //////////////////////////////////////
204         // L0
205         // FRAME SAME
206         // RETURN
207         //////////////////////////////////////
208         addBeforeMethodVisitor.visitLabel(L0);
209         addBeforeMethodVisitor.visitFrame(F_SAME, 0, null, 0, null);
210         addBeforeMethodVisitor.visitInsn(RETURN);
211         //////////////////////////////////////
212         // LMAXSTACK = 3
213         // MAXLOCALS = 2
214         //////////////////////////////////////
215         addBeforeMethodVisitor.visitMaxs(3, 2);
216         addBeforeMethodVisitor.visitEnd();
217         /*****
218         // 创建桥接 addBefore 方法
219         *****/
220         //////////////////////////////////////
221         // access flags 0x1041
222         // public synthetic bridge addBefore(Ljava/lang/Object;)V
223         //////////////////////////////////////
224         MethodVisitor bridgeAddBeforeMethodVisitor = classWriter.visitMethod(AC
225             "addBefore",
226             "(" + Type.getDescriptor(Object.class) + ")V",
227             null,
228             null
229         );
230         bridgeAddBeforeMethodVisitor.visitCode();
231         //////////////////////////////////////
232         // ALOAD 0
233         // ALOAD 1
234         //////////////////////////////////////
235         bridgeAddBeforeMethodVisitor.visitVarInsn(ALOAD, 0);
236         bridgeAddBeforeMethodVisitor.visitVarInsn(ALOAD, 1);
237         //////////////////////////////////////
238         // CHECKCAST com/hmilyylimh/cloud/compiler/asm/UserBean
239         // INVOKEVIRTUAL com/hmilyylimh/cloud/compiler/asm/UserBeanHandler.addB
240         // RETURN
241         //////////////////////////////////////
242         bridgeAddBeforeMethodVisitor.visitTypeInsn(CHECKCAST, Type.getInternalN
243         bridgeAddBeforeMethodVisitor.visitMethodInsn(INVOKEVIRTUAL,
244             Type.getInternalName(originClass) + newClzNameSuffix,
245             "addBefore",
```

```

246         "(" + Type.getDescriptor(originClass) + ")V");
247         bridgeAddBeforeMethodVisitor.visitInsn(RETURN);
248         //////////////////////////////////////
249         // MAXSTACK = 2
250         // MAXLOCALS = 2
251         //////////////////////////////////////
252         bridgeAddBeforeMethodVisitor.visitMaxs(2, 2);
253         bridgeAddBeforeMethodVisitor.visitEnd();
254         /*****
255         // 创建结束
256         /*****
257         classWriter.visitEnd();
258
259

```



天下无鱼

<https://shikey.com/>

写的过程有些卡壳，难度系数也不低，我们有 3 个小点要注意。

- 有些字节码指令不知道如何使用 ASM API，比如 INVOKESPECIAL 不知道怎么调用 API，你可以网络检索一下“**MethodVisitor INVOKESPECIAL**”关键字，就能轻松找到与之对应的 API 了。
- 重点关注调用 API 各参数的位置，千万别放错了，否则问题排查起来比较费时间。
- 生成的字节码文件直接保存到文件中，然后利用 `ClassLoader.defineClass` 方法，把字节码交给 JVM 虚拟机直接变成一个 Class 类型实例。

在写的时候，你一定要沉下心慢慢转换，一步都不能错，否则时间浪费了还得不到有效的成果。

写好之后，你一定非常兴奋，我们还是先写个测试代码验证一下：

 复制代码

```

1  public static void main(String[] args) throws Exception {
2      UserBean userBean = new UserBean("Geek");
3      // 从 mybatis 的拦截器里面拿到了准备更新 db 的数据对象，然后创建代理对象
4      Handler handler = (Handler) AsmProxyUtils.newProxyInstance(userBean.getClass()
5      // 关键的一步，在 mybatis 中模拟将入参对象进行加密操作
6      handler.addBefore(userBean);
7      // 这里为了观察效果，先打印一下 userBean 的内容看看
8      System.out.println(userBean);
9
10     // 接下来，假设有执行 db 的操作，那就直接将密文入库了
11
12     // db 操作完成之后，还得将 userBean 的密文变成明文，这里应该还有 addAfter 解密操作
13 }

```

```
1 打印一下加密内容: UserBean{name=''#BEFORE#Geek'}
```

结果如预期所料，把入参的数据成功加密了，我们终于可以喘口气了，不过辛苦是值得的，学到了大量的底层 **ASM** 操控字节码的知识，也见识到了底层功能的强大威力。

## Compiler 编译方式的适用场景

今天我们见识到 **Javassist** 和 **ASM** 的强大威力，之前也用过 **JavaCompiler** 和 **Groovy** 插件，这么多款工具可以编译生成类信息，有哪些适用场景呢？

- **JavaCompiler**: 是 JDK 提供的一个工具包，我们熟知的 **Javac** 编译器其实就是 **JavaCompiler** 的实现，不过 JDK 的版本迭代速度快，变化大，我们升级 JDK 的时候，本来在低版本 JDK 能正常编译的功能，跑到高版本就失效了。
- **Groovy**: 属于第三方插件，功能很多很强大，几乎是开发小白的首选框架，不需要考虑过多 **API** 和字节码指令，会构建源代码字符串，交给 **Groovy** 插件后就能拿到类信息，拿起来就可以直接使用，但同时也是比较重量级的插件。
- **Javassist**: 封装了各种 **API** 来创建类，相对于稍微偏底层的风格，可以动态针对已有类的字节码，调用相关的 **API** 直接增删改查，非常灵活，只要熟练使用 **API** 就可以达到很高的境界。
- **ASM**: 是一个通用的字节码操作的框架，属于非常底层的插件了，操作该插件的技术难度相当高，需要对字节码指令有一定的了解，但它体现出来的性能却是最高的，并且插件本身就是定位为一款轻量级的高性能字节码插件。

有了众多动态编译方式的法宝，从简单到复杂，从重量级到轻量级，你都学会了，相信再遇到一堆神乎其神的 **Compiler** 编译方式，内心也不会胆怯了。

不过工具多了，有同学可能就有选择困难症，这里我也讲下个人的选择标准。

如果需要开发一些底层插件，我倾向使用 **Javassist** 或者 **ASM**。使用 **Javassist** 是因为用 **API** 简单而且方便后人维护，使用 **ASM** 是在一些高度频繁调用的场景出于对性能的极致追求。如

果开发应用系统的业务功能，对性能没有太强的追求，而且便于加载和卸载，我倾向使用 Groovy 这款插件。



## 总结

今天，我们接着上一讲刚学会的自定义代理案例，在不使用 Wrapper 已有机制的能力下，自己尝试使用简单的 API 操作来创建代理类，见识到了 Javassist 的强大之处，有一套非常齐全的 API 来创建代理类并实时编译成类信息。

使用 Javassist 编译的三大基本步骤。

- 首先，设计一个代码模板。
- 然后，使用 Javassist 的相关 API，通过 `ClassPool.makeClass` 得到一个操控类的 `CtClass` 对象，然后针对 `CtClass` 进行 `addField` 添加字段、`addMethod` 添加方法、`addConstructor` 添加构造方法等等。
- 最后，调用 `CtClass.toClass` 方法并编译得到一个类信息，有了类信息，就可以实例化对象处理业务逻辑了。

然后借助把敏感字段加密存储到数据库的案例，我们研究了大佬常用的 ASM 开发底层框架，尝试先将代码模板变成字节码指令，然后按照 ASM 的规范，将字节码指令一个个翻译成为 ASM 对应的方法，最终通过 `ClassLoader.defineClass` 将字节码变成了类信息。

也总结下使用 ASM 编译的四大基本步骤。

- 首先，还是设计一个代码模板。
- 其次，通过 IDEA 的协助得到代码模板的字节码指令内容。
- 然后，使用 Asm 的相关 API 依次将字节码指令翻译为 Asm 对应的语法，比如创建 `ClassWriter` 相当于创建了一个类，继续调用 `ClassWriter.visitMethod` 方法相当于创建了一个方法等等，对于生僻的字节码指令实在找不到对应的官方文档的话，可以通过“**MethodVisitor + 字节码指令**”来快速查找对应的 Asm API。
- 最后，调用 `ClassWriter.toByteArray` 得到字节码的字节数组，传递到 `ClassLoader.defineClass` 交给 JVM 虚拟机得出一个 `Class` 类信息。

总的来说，无论使用哪种方式进行动态编译，不管是出于对性能的极致追求，还是出于对项目具体业务功能的通用处理，只要适合自身业务，且不会带来沉重性能开销，都是一种好方式。只有合适的才是最好的。



## 思考题

留个作业给你，前面讲到 **Javassist** 动态编译时，其实里面也支持直接将编写的源代码编译成类信息，你可以试着从 **Dubbo** 的 `org.apache.dubbo.common.compiler.Compiler` 接口研究一番，看看能不能找到现成的方法来编译源代码？

期待看到你的思考，如果你对今天的内容还有什么困惑，欢迎在留言区提问，我会第一时间回复。我们下一讲见。

## 参考资料

考虑到 **ASM** 比较难，我之前也录制了 [ASM 实操一步步教学视频](#)，你可以学习。

## 15 思考题参考

上一期留了个作业，总结下 `java.lang.Class#getDeclaredMethod` 方法的调用流程，以及排查下存在哪些耗时和占用内存的地方。

既然要研究方法的调用流程，那最简单的方式就是直接去看代码。

复制代码

```
1 // java.lang.Class#getDeclaredMethod
2 public Method getDeclaredMethod(String name, Class<?>... parameterTypes)
3     throws NoSuchMethodException, SecurityException {
4     // 检查方法的权限
5     checkMemberAccess(Member.DECLARED, Reflection.getCallerClass(), true);
6     // 从该类中的所有方法列表中找到一个匹配方法名称和方法参数的方法对象
7     // 并且返回的 method 对象是一个克隆拷贝出来的对象
8     Method method = searchMethods(privateGetDeclaredMethods(false), name, param
9     if (method == null) {
10         throw new NoSuchMethodException(getName() + "." + name + argumentTypesT
11     }
12     return method;
13 }
14
15 // 从缓存或者通过 native 调用从 JVM 中获取该 Class 中声明的方法列表集合
16 private Method[] privateGetDeclaredMethods(boolean publicOnly) {
17     checkInitiated();
```

```
18     Method[] res;
19     // 从缓存中获取该 Class 中声明的方法列表集合
20     ReflectionData<T> rd = reflectionData();
21     if (rd != null) {
22         res = publicOnly ? rd.declaredPublicMethods : rd.declaredMethods;
23         if (res != null) return res;
24     }
25     // No cached value available; request value from VM
26     // 从 JVM 中获取该 Class 中声明的方法列表集合
27     // getDeclaredMethods0 尾巴上是个数字0，一般是 native 方法
28     // 而 native 调用产生的消耗一般可以达到 java 调用的 10 倍以上
29     res = Reflection.filterMethods(this, getDeclaredMethods0(publicOnly));
30     if (rd != null) {
31         if (publicOnly) {
32             rd.declaredPublicMethods = res;
33         } else {
34             rd.declaredMethods = res;
35         }
36     }
37     return res;
38 }
39
40 // 从缓存中获取方法列表对象
41 private ReflectionData<T> reflectionData() {
42     SoftReference<ReflectionData<T>> reflectionData = this.reflectionData;
43     int classRedefinedCount = this.classRedefinedCount;
44     ReflectionData<T> rd;
45     if (useCaches &&
46         reflectionData != null &&
47         (rd = reflectionData.get()) != null &&
48         rd.redefinedCount == classRedefinedCount) {
49         return rd;
50     }
51     // else no SoftReference or cleared SoftReference or stale ReflectionData
52     // -> create and replace new instance
53     return new ReflectionData(reflectionData, classRedefinedCount);
54 }
55
56 // 存方法列表的反射对象类
57 // 从该类中发现含有大量的字段集合、方法集合、构造方法集合等等
58 // 存储太多太多的重要数据，缓存占据的容量大小也是一个问题因素
59 private static class ReflectionData<T> {
60     volatile Field[] declaredFields;
61     volatile Field[] publicFields;
62     volatile Method[] declaredMethods;
63     volatile Method[] publicMethods;
64     volatile Constructor<T>[] declaredConstructors;
65     volatile Constructor<T>[] publicConstructors;
66     // Intermediate results for getFields and getMethods
67     volatile Field[] declaredPublicFields;
68     volatile Method[] declaredPublicMethods;
69     volatile Class<?>[] interfaces;
```

70 // Value of classRedefinedCount when we created this ReflectionData instance

71 final int redefinedCount;

72 ReflectionData(int redefinedCount) {

73 this.redefinedCount = redefinedCount;

74 }

75 }

76 ↓

77 // 从声明的方法列表集合中，检索出一个匹配方法名称和方法参数的 method 对象

78 private static Method searchMethods(Method[] methods,

79 String name,

80 Class<?>[] parameterTypes)

81 {

82 Method res = null;

83 String internedName = name.intern();

84 for (int i = 0; i < methods.length; i++) {

85 Method m = methods[i];

86 if (m.getName() == internedName

87 && arrayContentsEq(parameterTypes, m.getParameterTypes())

88 && (res == null

89 || res.getReturnType().isAssignableFrom(m.getReturnType()))

90 res = m;

91 }

92 // 重点看这里，如果找到了一个匹配的 Method 的话

93 // 则重新 copy 一份新对象返回，且新对象的 root 属性都会指向原来的 Method 对象

94 // 也就意味着大量调用的话，又会产生大量的对象，是不可忽视的一个重要环节

95 return (res == null ? res : getReflectionFactory().copyMethod(res));

96 }



天下无鱼

<https://shikey.com/>

代码看下来，总结起来，获取方法的流程主要有三步。

- 第一步，检查方法权限。
- 第二步，从 Class 对象的所有方法列表中查找匹配的 Method 对象。
- 第三步，返回 Method 对象的克隆对象。

第二步每次都是  $O(n)$  时间复杂度的检索，并且缓存中的 ReflectionData 内容实在是太多了，占据了大量的内存开销。

第三步每次都是返回一个克隆对象，而 Method 对象本身就有很多东西，每次都拷贝出一个新对象，又是吃内存的操作。

一个简简单单的直接调用，换做反射后，即使不谈内存占用，就光凭代码量的行数，就相当于调用了很多字节码指令，虽然字节码指令执行很快，但是蚊子腿也是肉，时间上也有数十倍甚至百倍的耗时增长开销。



有时候，虽然 Java 反射性能会存在一定损耗开销，但并不是告诉我们不能使用 Java 反射，其实，我们应该根据程序设计来按需考量设计。



如果你只是根据某些条件或配置文件来使用 Java 反射创建对象，然后就做其它操作（如数据库查询等），那 Java 反射损耗的那点性能可以忽略不计。

但如果你在连续循环中通过反射不断创建大量对象，这样的操作就得好好斟酌了，看看是不是可以不用反射的机制实现，避免大量耗时和内存的开销。

分享给需要的人，Ta购买本课程，你将得 18 元

生成海报并分享

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 15 | Wrapper机制：Wrapper是怎么降低调用开销的？

下一篇 17 | Adaptive适配：Dubbo的Adaptive特殊在哪里？

## 精选留言 (3)

写留言



王轲

2023-01-25 来自美国

`...用 0表示，方法中的第一个参数用1 表示...` 这里看起来显示有误，应该是`...用 \$0表示，方法中的第一个参数用\$1 表示...`

作者回复: 你好，王轲：是滴，显示有误，非常感谢～

正确的是这样的：

在对方法的入参字段名进行逻辑处理时，就得替换成 \$ 占位符，方法中的 this 引用，用 \$0 表示，方法中的第一个参数用 \$1 表示，第二个参数用 \$2 表示，以此类推。



斯瓦辛武Roy

2023-01-24 来自江苏

老师春节好，请教个问题，再看官方文档的时候<https://cn.dubbo.apache.org/zh/docsv2.7/user/references/telnet/>

用了这个命令`trace XxxService`: 跟踪 1 次服务任意方法的调用情况

我调用自己本地的dubbo服务，但这个命令没有反应，请教一下老师这个是什么原因

作者回复: 你好，斯瓦辛武Roy: 我这边按照你所描述的方式，验证了一下，都是正常的，我这边我说下我的操作步骤，你尝试按照我的步骤试试:

1. 启动提供方成功后，打开命令窗口，输入命令: `telnet 127.0.0.1 28045`
2. 输入完命令后，直接回车，然后你会看到“dubbo>”这样的内容显示。
3. 然后在“dubbo>”后面继续追加 `trace` 命令，比如: `dubbo>trace com.hmilylimh.cloud.facade.demo.DemoFacade`
4. 在第 3 步骤的命令输入完后，紧接着再次回车
5. 运行消费方来调用 `DemoFacade` 接口，然后就能在命令窗口看到 `trace` 命令跟踪的结果了。



熊悟空的凶

2023-01-23 来自北京

老师，新年好

作者回复: 你好，熊悟空的凶: 兔年吉祥，在过年的闹市中还能静心学习，让我想起了我刚工作的那几年，每年过年的晚上都在挑灯奋战努力写代码学习。你还是很棒的，闹市中能静下心来学习你，将来一定是钱兔无量，为你加油喝彩，点赞~

