

二

24 有哪些方法可以在运行时动态生成一个Java类？-极客时间

在开始今天的学习前，我建议你先复习一下【专栏第 6 讲】有关动态代理的内容。作为 Java 基础模块中的内容，考虑到不同基础的同学以及一个循序渐进的学习过程，我当时并没有在源码层面介绍动态代理的实现技术，仅进行了相应的技术比较。但是，有了【上一讲】的类加载的学习基础后，我想是时候该进行深入分析了。

今天我要问你的问题是，**有哪些方法可以在运行时动态生成一个 Java 类？**

典型回答

我们可以从常见的 Java 类来源分析，通常的开发过程是，开发者编写 Java 代码，调用 javac 编译成 class 文件，然后通过类加载机制载入 JVM，就成为应用运行时可以使用的 Java 类了。

从上面过程得到启发，其中一个直接的方式是从源码入手，可以利用 Java 程序生成一段源码，然后保存到文件等，下面就只需要解决编译问题了。

有一种笨办法，直接用 ProcessBuilder 之类启动 javac 进程，并指定上面生成的文件作为输入，进行编译。最后，再利用类加载器，在运行时加载即可。

前面的方法，本质上还是在当前程序进程之外编译的，那么还有没有不那么 low 的办法呢？

你可以考虑使用 Java Compiler API，这是 JDK 提供的标准 API，里面提供了与 javac 对等的编译器功能，具体请参考[java.compiler](#)相关文档。

进一步思考，我们一直围绕 Java 源码编译成为 JVM 可以理解的字节码，换句话说，只要是符合 JVM 规范的字节码，不管它是如何生成的，是不是都可以被 JVM 加载呢？我们能不能直接生成相应的字节码，然后交给类加载器去加载呢？

当然也可以，不过直接去写字节码难度太大，通常我们可以利用 Java 字节码操纵工具和类

库来实现，比如在【专栏第 6 讲】中提到的ASM、Javassist、cglib 等。

考点分析

虽然曾经被视为黑魔法，但在当前复杂多变的开发环境中，在运行时动态生成逻辑并不是什么罕见的场景。重新审视我们谈到的动态代理，本质上不就是在特定的时机，去修改已有类型实现，或者创建新的类型。

明白了基本思路后，我还是围绕类加载机制进行展开，面试过程中面试官很可能从技术原理或实践的角度考察：

- 字节码和类加载到底是怎么无缝进行转换的？发生在整个类加载过程的哪一步？
- 如何利用字节码操纵技术，实现基本的动态代理逻辑？除了动态代理，字节码操纵技术还有那些应用
- 场景？

知识扩展

首先，我们来理解一下，类从字节码到 Class 对象的转换，在类加载过程中，这一步是通过下面的方法提供的功能，或者 defineClass 的其他本地对等实现。

```
protected final Class<?> defineClass(String name, byte[] b, int off, int len,
                                     ProtectionDomain protectionDomain)
protected final Class<?> defineClass(String name, java.nio.ByteBuffer b,
                                     ProtectionDomain protectionDomain)
```

我这里只选取了最基础的两个典型的 defineClass 实现，Java 重载了几个不同的方法。

可以看出，只要能够生成出规范的字节码，不管是作为 byte 数组的形式，还是放到 ByteBuffer 里，都可以平滑地完成字节码到 Java 对象的转换过程。

JDK 提供的 defineClass 方法，最终都是本地代码实现的。

```
static native Class<?> defineClass1(ClassLoader loader, String name, byte[] b, int
                                     ProtectionDomain pd, String source);

static native Class<?> defineClass2(ClassLoader loader, String name, java.nio.ByteB
                                     int off, int len, ProtectionDomain pd,
                                     String source);
```

更进一步，我们来看看 JDK dynamic proxy 的实现代码。你会发现，对应逻辑是实现在 ProxyBuilder 这个静态内部类中，ProxyGenerator 生成字节码，并以 byte 数组的形式保存，然后通过调用 Unsafe 提供的 defineClass 入口。

```
byte[] proxyClassFile = ProxyGenerator.generateProxyClass(
    proxyName, interfaces.toArray(EMPTY_CLASS_ARRAY), accessFlags);
try {
    Class<?> pc = UNSAFE.defineClass(proxyName, proxyClassFile,
                                    0, proxyClassFile.length,
                                    loader, null);
    reverseProxyCache.sub(pc).putIfAbsent(loader, Boolean.TRUE);
    return pc;
} catch (ClassFormatError e) {
    // 如果出现ClassFormatError，很可能是输入参数有问题，比如，ProxyGenerator有bug
}
```

前面理顺了二进制的字节码信息到 Class 对象的转换过程，似乎我们还没有分析如何生成自己需要的字节码，接下来一起来看看相关的字节码操纵逻辑。

JDK 内部动态代理的逻辑，可以参考[java.lang.reflect.ProxyGenerator](#)的内部实现。我觉得可以认为这是种另类的字节码操纵技术，其利用了DataOutputStream提供的能力，配合 hard-coded 的各种 JVM 指令实现方法，生成所需的字节码数组。你可以参考下面的示例代码。

```
private void codeLocalLoadStore(int lvar, int opcode, int opcode_0,
                                DataOutputStream out)
    throws IOException
{
    assert lvar >= 0 && lvar <= 0xFFFF;
    // 根据变量数值，以不同格式，dump操作码
    if (lvar <= 3) {
        out.writeByte(opcode_0 + lvar);
    } else if (lvar <= 0xFF) {
        out.writeByte(opcode);
        out.writeByte(lvar & 0xFF);
    } else {
        // 使用宽指令修饰符，如果变量索引不能用无符号byte
        out.writeByte(opc_wide);
        out.writeByte(opcode);
        out.writeShort(lvar & 0xFFFF);
    }
}
```

这种实现方式的好处是没有太多依赖关系，简单实用，但是前提是你需要懂各种JVM 指令，知道怎么处理那些偏移地址等，实际门槛非常高，所以并不适合大多数的普通开发场景。

幸好，Java 社区专家提供了各种从底层到更高抽象水平的字节码操作类库，我们不需要什么都自己从头做。JDK 内部就集成了 ASM 类库，虽然并未作为公共 API 暴露出来，但是它广泛应用在，如[java.lang.instrumentation](#) API 底层实现，或者[Lambda Call Site](#)生成的内部逻辑中，这些代码的实现我就不在这里展开了，如果你确实有兴趣或有需要，可以参考类似 [LamdaForm](#) 的字节码生成逻辑：[java.lang.invoke.InvokerBytecodeGenerator](#)。

从相对实用的角度思考一下，实现一个简单的动态代理，都要做什么？如何使用字节码操纵技术，走通这个过程呢？

对于一个普通的 Java 动态代理，其实现过程可以简化成为：

- 提供一个基础的接口，作为被调用类型（com.mycorp.HelloImpl）和代理类之间的统一入口，如 com.mycorp.Hello。
- 实现[InvocationHandler](#)，对代理对象方法的调用，会被分派到其 invoke 方法来真正实现动作。
- 通过 Proxy 类，调用其 newProxyInstance 方法，生成一个实现了相应基础接口的代理类实例，可以看下面的方法签名。

```
public static Object newProxyInstance(ClassLoader loader,
                                     Class<?>[] interfaces,
                                     InvocationHandler h)
```

我们分析一下，动态代码生成是具体发生在什么阶段呢？

不错，就是在 newProxyInstance 生成代理类实例的时候。我选取了 JDK 自己采用的 ASM 作为示例，一起来看看用 ASM 实现的简要过程，请参考下面的示例代码片段。

第一步，生成对应的类，其实和我们去写 Java 代码很类似，只不过改为用 ASM 方法和指定参数，代替了我们书写的源码。

```
ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);

cw.visit(V1_8,                               // 指定Java版本
        ACC_PUBLIC,                          // 说明是public类型
        "com/mycorp/HelloProxy",            // 指定包和类的名称
        null,                                // 签名，null表示不是泛型
        "java/lang/Object",                 // 指定父类
        new String[]{ "com/mycorp/Hello" }); // 指定需要实现的接口
```

更进一步，我们可以按照需要为代理对象实例，生成需要的方法和逻辑。

```
MethodVisitor mv = cw.visitMethod(
```

```
ACC_PUBLIC,                // 声明公共方法
"sayHello",                // 方法名称
"()Ljava/lang/Object;",    // 描述符
null,                      // 签名, null表示不是泛型
null);                     // 可能抛出的异常, 如果有, 则指定字符串数组

mv.visitCode();
// 省略代码逻辑实现细节
cw.visitEnd();              // 结束类字节码生成
```

上面的代码虽然有些晦涩, 但总体还是能多少理解其用意, 不同的 visitX 方法提供了创建类型, 创建各种方法等逻辑。ASM API, 广泛的使用了[Visitor](#)模式, 如果你熟悉这个模式, 就会知道它所针对的场景是将算法和对象结构解耦, 非常适合字节码操纵的场合, 因为我们大部分情况都是依赖于特定结构修改或者添加新的方法、变量或者类型等。

按照前面的分析, 字节码操作最后大都应该都是生成 byte 数组, ClassWriter 提供了一个简便的方法。

```
cw.toByteArray();
```

然后, 就可以进入我们熟知的类加载过程了, 我就不再赘述了, 如果你对 ASM 的具体用法感兴趣, 可以参考这个[教程](#)。

最后一个问题, 字节码操纵技术, 除了动态代理, 还可以应用在什么地方?

这个技术似乎离我们日常开发遥远, 但其实已经深入到各个方面, 也许很多你现在正在使用的框架、工具就应用该技术, 下面是我能想到的几个常见领域。

- 各种 Mock 框架
- ORM 框架
- IOC 容器
- 部分 Profiler 工具, 或者运行时诊断工具等
- 生成形式化代码的工具

甚至可以认为, 字节码操纵技术是工具和基础框架必不可少的部分, 大大减少了开发者的负担。

今天我们探讨了更加深入的类加载和字节码操作方面技术。为了理解底层的原理, 我选取的例子是比较偏底层的、能力全面的类库, 如果实际项目中需要进行基础的字节码操作, 可以考虑使用更加高层次视角的类库, 例如[Byte Buddy](#)等。

一课一练

关于今天我们讨论的题目你做到心中有数了吗？试想，假如我们有这样一个需求，需要添加某个功能，例如对某类型资源如网络通信的消耗进行统计，重点是，不开启时必须是**零开销，而不是低开销**，可以利用我们今天谈到的或者相关的技术实现吗？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。

[上一页](#)

[下一页](#)