面向对象源码分析 —— Javassist

Part0 前言

Javasstst 简介

Javassist 是一个高效的 Java 字节码操作库，专为动态类生成和修改而设计。它通过简洁的 API，使开发者能够在运行时创建新类、修改现有类、插入监控代码和动态重定义类，而无需深入了解 JVM 字节码结构。Javassist 特别适用于 AOP场景，例如在方法执行前后插入日志或性能监控代码。此外，它还支持生成动态代理类，用于实现懒加载和安全检查等功能。凭借其高效性和灵活性，Javassist 成为动态编程、性能优化和框架开发中的重要工具。

Javassist 的各模块

- `javassist`：核心模块，提供对字节码的操作和生成
- `javassist.bytecode`：提供接口允许程序直接修改字节码
- `javassist.compiler`：将源代码编译为字节码
- `javassist.expr`：提供对方法表达式的支持

我选择分析核心模块，具体为 `ClassPool` 和 `CtClass` 及其相关模块，负责管理和修改字节码。

Part1 主要功能分析与建模

主要功能分析

`ClassPool` 负责加载类并缓存其字节码，以便后续操作，使用示例如下：

```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.makeClass("List");
```

`CtClass` 能够添加、删除和修改类的方法，使用示例如下：

```
CtClass ctClass = pool.get("com.example.MyClass");

// 添加新方法
CtMethod newMethod = CtNewMethod.make("public void newMethod() {
System.out.println(\"Hello, World!\"); }", ctClass);
ctClass.addMethod(newMethod);

// 添加新字段
CtField newField = new CtField(CtClass.intType, "newField", ctClass);
ctClass.addField(newField);

// 获取要删除的方法
CtMethod methodToDelete = ctClass.getDeclaredMethod("methodName");
// 删除该方法
ctClass.removeMethod(methodToDelete);

// 获取要修改的方法
CtMethod methodToModify = ctClass.getDeclaredMethod("methodName");
// 修改方法体
```

```
methodToModify.setBody("{ System.out.println(\"This is the new method body.\");
}");

// 插入字节码
methodToModify.insertBefore("{ System.out.println(\"Before method execution\");
}");
methodToModify.insertAfter("{ System.out.println(\"After method execution\");
}");
```

需求建模

根据需求和使用示例可以建立以下模型：

> 需求模型
>
> 【用例名称】
>
> 操作一个 Java 类的字节码。
>
> 【场景】
>
> - Who：调用者，`ClassPool` 实例，`CtClass` 实例
> - Where：内存
> - When：编译时、运行时
>
> 【用例描述】
>
> 1. 调用者创建一个 `CtClass` 实例
>   1. 调用者定义类的结构
>   2. 调用者选择方法和字段的属性
> 2. 调用者删除或修改类的方法
> 3. 调用者调用类的构造函数和方法
> 4. 类在运行时进行字节码的操作
>
> 【用例价值】
>
> 动态生成和修改类和方法的字节码
>
> 【约束和限制】
>
> 修改后的字节码必须符合 Java 虚拟机规范，否则会引发运行时错误。

寻找其中的动词和名词：

> 动词：创建、定义、添加、选择、删除、修改、调用、动态加载
>
> 名词：`CtClass` 实例，`ClassPool` 实例，类结构，成员（方法、字段），方法，字段，构造函数，动态加载类

可以抽象出类和方法：

> 【类】：`ClassPool`
>
> - 【方法】：创建，添加，获取
> - 【属性】：类路径
>
> 【类】：`CtClass`

> - 【方法】: 定义，修改，生成字节码
> - 【属性】: 类名，方法列表，字段列表
>
> 【类】: `CtMethod`
>
> - 【方法】: 添加，改，删除
> - 【属性】: 方法名，返回类型，参数类型
>
> 【类】: `CtField`
>
> - 【方法】: 添加，修改，删除
> - 【属性】: 字段名，类型

小结

对 `ClassPool` 和 `CtClass` 的阅读让我了解到了 Javassist 的强大功能，其优秀的集成使得程序可以高效地进行字节码的生成和操作，实现动态类生成和修改，满足运行时的需求。

Part2 核心流程设计分析

组件设计

Javassist 的核心流程由各个组件共同参与组成，并最终完成对字节码的操作。查阅核心代码和文件目录后发现以下几个类提供核心操作，它们的功能如下：

- `CtField.java`：它可以用来创建新的字段或者修改现有字段的属性，如类型等，并可以将字段添加到 `CtClass` 中。
- `CtMethod.java`：它允许使用者创建新的方法或者修改现有方法的返回类型、访问修饰符以及方法体内容。
- `CtNewMethod.java`：它提供了一些静态方法来快速创建新的方法，如生成 `getter` 和 `setter` 方法，或者根据给定的源代码创建新的方法。
- `CtConstructor.java`：它可以用来创建新的构造函数或者修改现有构造函数的参数和方法体。

获取类文件

Javassist 需要从字节码的层面编辑类文件，并且这个类可能正在运行，即存在内存中，因此需要将类抽象为一个对象，这一过程由 `ClassFile` 类提供。而在核心模块中，类文件可能来自输入流，也可能由 `CtClass` 提供。分析后发现 `ClassFile` 类与 `ConstInfo`、`FieldInfo`、`MethodInfo`、`AttributeInfo` 相互存在依赖关系。在输入流提供类文件时，`ClassFile` 创建抽象的类文件对象并初始化生成各信息供上面四个类使用；反之，若 `CtClass` 提供类文件，则由这四个类的 `get` 方法来构建抽象对象。

```
ClassPool pool = ClassPool.getDefault();
```

编辑类信息

该过程通过 `CtField`、`CtNewMethod`、`CtConstructor` 完成，先由 `CtField` 添加新字段：

```
CtField param = new CtField(pool.get("java.lang.String"), "name", c);
c.addField(param, CtField.Initializer.constant("con"));
```

然后由 `CtNewMethod` 创建新方法 `getter` 和 `setter`：

```
c.addMethod(CtNewMethod.setter("setName", param));
c.addMethod(CtNewMethod.getter("getName", param));
```

再由 `CtConstructor` 添加新的构造函数：

```
CtConstructor cons = new CtConstructor(new CtClass[]
{pool.get("java.lang.String")}, c);
c.addConstructor(cons);
```

修改方法体

该过程通过 `CtMethod` 完成，使用 `CtMethod` 获取后直接进行修改：

```
CtMethod method = c.getDeclaredMethod("myMethod");
method.setBody("{ System.out.println(\"Modified Method\"); }");
```

保存修改

将修改后的类写入文件系统：

```
c.writeFile();
```

小结

Javassist 的核心流程通过 `ClassPool`、`CtClass`、`CtField`、`CtMethod`、`ClassFile` 等若干个关键组件实现，它们分别提供不同的功能又相互依存，共同构成了 Javassist 动态字节码操作的基础架构。

Part3 高级设计意图分析

FactoryMethod 工厂方法模式

> 定义一个用于创建对象的接口，让子类决定实例化哪一个类。FactoryMethod 使一个类的实例化延迟到其子类。

Javassist 在设计中遵循了工厂方法模式。在对字节码进行操作的过程中，每个字节码的类都有其不同的属性，这也导致在进行操作时需要由子类决定是否进行对应操作。例如在创建 CtClass 对象时，需要检查创建的 CtClass 对象是否被冻结，若是被冻结则不会创建类而是直接返回。

```
    public synchronized CtClass makeClass(String classname, CtClass superclass)
        throws RuntimeException
    {
        checkNotFrozen(classname); // 检查是否被冻结
        CtClass clazz = new CtNewClass(classname, this, false, superclass);
        cacheCtClass(classname, clazz, true);
        return clazz;
    }
```

同时，可以注意到上面这段代码中有对 cache 的引用，这里就要提到 Javassist 的一个精妙的设计——ClassPool 中的 Cache，代码如下：

```
protected CtClass getCached(String classname) {
        return (CtClass)classes.get(classname);
    }

    protected void cacheCtClass(String classname, CtClass c, boolean dynamic) {
        classes.put(classname, c);
    }

    protected CtClass removeCached(String classname) {
        return (CtClass)classes.remove(classname);
    }
```

ClassPool 的 Cache 在核心代码中被广泛应用，主要在创建、修改 CtClass 中被使用，用于快速调用和回收 CtClass 以提升整体运行效率。

Strategy 策略模式

> 针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。策略模式使得算法可以在不影响到客户端的情况下发生变化。

Javassist 在设计中遵循了策略模式。这一点在 CtMethod 等模块中有较好体现。CtMethod 主要用于修改对象的方法，在修改时提供了诸多修改方法，比如最为简单的 `setBody`：

```
    public void setBody(String src,
                        String delegateObj, String delegateMethod)
        throws CannotCompileException
    {
        CtClass cc = declaringClass;
        cc.checkModify(); // 检查是否能够修改
        try {
            Javac jv = new Javac(cc);
            if (delegateMethod != null)
                jv.recordProceed(delegateObj, delegateMethod);

            Bytecode b = jv.compileBody(this, src); // 调用 compileBody 生成字节码
            methodInfo.setCodeAttribute(b.toCodeAttribute());
            methodInfo.setAccessFlags(methodInfo.getAccessFlags()
                                      & ~AccessFlag.ABSTRACT);
            methodInfo.rebuildStackMapIf6(cc.getClassPool(), cc.getClassFile2());
            declaringClass.rebuildClassFile();
        }
        catch (CompileError e) {
            throw new CannotCompileException(e);
        } catch (BadBytecode e) {
            throw new CannotCompileException(e);
        }
    }
```

通过 `setBody` 可以直接通过传入字段修改方法的内容，这一方法简单粗暴并且效率较高，但是难以应对一些特殊情况，例如用户在修改时只想在前后加入参数，而要修改的方法对于用户来说可能是黑箱状态，不知道具体代码和形式，此时使用 `setBody` 不但效率有限而且难以达成效果，因此 Javassist 提供了 `insertBefore` 和 `insertAfter` 两个方法来对字段的前后进行插入操作，从而使其能够应对各种使用情况，代码如下：

```
    private void insertBefore(String src, boolean rebuild)
        throws CannotCompileException
    {
        CtClass cc = declaringClass;
        cc.checkModify(); // 检查能否修改
        CodeAttribute ca = methodInfo.getCodeAttribute(); // 获取对应属性
        if (ca == null)
            throw new CannotCompileException("no method body");

        CodeIterator iterator = ca.iterator();
        Javac jv = new Javac(cc);
        try {
            int nvars = jv.recordParams(getParameterTypes(),
                                        Modifier.isStatic(getModifiers()));
            jv.recordParamNames(ca, nvars);
            jv.recordLocalVariables(ca, 0);
            jv.recordReturnType(getReturnType0(), false);
            jv.compileStmnt(src);
            Bytecode b = jv.getBytecode(); // 获取字节码
            int stack = b.getMaxStack();
            int locals = b.getMaxLocals();

            if (stack > ca.getMaxStack())
                ca.setMaxStack(stack);

            if (locals > ca.getMaxLocals()) // 检查是否溢出
                ca.setMaxLocals(locals);

            int pos = iterator.insertEx(b.get());
            iterator.insert(b.getExceptionTable(), pos); // 插入字节码
            if (rebuild)
                methodInfo.rebuildStackMapIf6(cc.getClassPool(),
cc.getClassFile2()); // 重建栈
        }
        catch (NotFoundException e) {
            throw new CannotCompileException(e);
        }
        catch (CompileError e) {
            throw new CannotCompileException(e);
        }
        catch (BadBytecode e) {
            throw new CannotCompileException(e);
        }
    }
```

通过对于 `setBody` 和 `insertBefore` 等函数的灵活调用，Javassist 的策略模式顺利体现，展示了在运行时动态创建和切换策略的灵活性。这种方法非常适用于需要在应用程序运行期间动态调整算法或逻辑的多种情况。

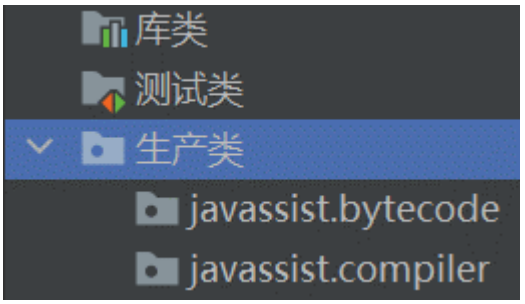
Proxy 代理模式

> 为其他对象提供一种代理以控制对这个对象的访问。

Javassist 在设计中遵循了代理模式，特别是动态代理机制。Javassist 作为 JBoss 框架的子项目，当一个类需要处理复杂的事务操作时，我们不希望将所有接口完全暴露给用户，而是使用一个代理来替代真实对象的访问，从而在不修改原代码的情况下增强功能。静态代理在编译时构建代理类，与原目标耦合度较高，而 Javassist 实现的动态代理则具有更高的灵活性。在类加载到 JVM 虚拟机时，Javassist 可以在 ClassFile 对象的基础上动态地织入一些业务逻辑，代理对真实对象（委托类）的访问。这意味着 ClassFile 的代码不再需要直接处理这些事务性逻辑，开发者可以从繁琐的事务工作中解脱出来，确保客户端和业务提供方的松耦合。此外，通过在类的生命周期中横向插入字节码，Javassist 也实现了热加载和热插装，这体现了 JBoss 框架的 AOP 特性。

依赖分析

通过 IntelliJ IDEA 的依赖分析工具对 Javassist 项目进行循环依赖分析，测试发现整个项目依赖管理良好，唯一的循环依赖发生在 `javassist.bytecode` 和 `javassist.compiler` 中，可能由于这两个模块各自的功能独立，对整体运行的影响较小。



Part4 总结

通过深入分析 Javassist 的源码，我们可以看到它是一个非常强大的字节码操作库。它允许开发者在运行时动态地创建、修改类和方法，从而实现动态代理、AOP 以及其他高级编程技术。总结来说，Javassist 通过合理的模块划分和设计模式应用，实现了高效的字节码操作，为动态编程和性能优化提供了有力支持。在未来的开发中，我们可以借鉴其设计思想和实现方法，不断提升我们自己的代码质量和开发效率。

完结撒花~