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。