Javassist 源码阅读

1 什么是 Java 字节码？

当我们编写完一个 .java 文件并使用 javac 编译后，会生成一个 .class 文件，这个文件包含了 Java 字节码（bytecode），可以在任何安装了 JVM（Java 虚拟机）的设备上运行。这一编译过程是与平台无关的，因此无论在什么操作系统上编译，同样的 .java 文件会生成相同的 .class 文件。这也是 Java “一次编译，到处运行”特性的核心所在，极大地提升了代码的可移植性。
在此过程中，JVM 起到了关键作用，它将底层硬件平台进行抽象，为上层 Java 程序提供了统一的虚拟架构。Java 字节码就是运行在这一虚拟架构上的机器指令，通常每条指令为一个字节（即“字节码”）。一个 .class 文件中包含了类运行所需的字节码指令，JVM 会将这些指令转化为目标平台的机器代码，使程序能够在不同硬件架构上执行。为了适应不同的硬件，Java 字节码主要通过栈来运算，从而避免直接依赖硬件的寄存器配置。

2 什么是 Javassist？

Javassist（Java Programming Assistant）是一个开源的 Java 字节码操作库，主要用于在运行时动态生成、编辑和操作 Java 类。它提供了一个简单易用的 API，让开发者能够通过类似 Java 源代码的方式来操作字节码，而无需深入了解 JVM 字节码的复杂细节。这使得 Javassist 特别适合于开发需要在运行时修改类行为的应用程序，例如 AOP（面向切面编程）、代理类生成、代码注入、性能监控、日志记录等场景。
Javassist 的主要功能包括：
动态生成类和方法：可以在运行时创建新类或方法，无需提前在源码中定义。
修改现有类：支持在加载类之前修改已有类的字节码，例如插入新的方法、修改方法体等。
简洁的 API：提供了面向高层的 API，通过类似源代码的方式编写字节码操作，降低了学习成本。
字节码操作：可以直接操作 Java 字节码，使得动态代理、增强代码等操作更加灵活。
Javassist 由 JBOSS 组织维护，通常被集成在 Java 框架。

3 .class 文件如何记录信息

.class 文件是 Java 编译器将 Java 源代码（.java 文件）编译后的输出文件，它包含了 Java 虚拟机（JVM）能够直接执行的字节码。.class 文件记录的信息包括：
魔数：每个.class 文件的前四个字节是魔数 0xCAFEBABE，用于标识文件格式，确保文件是一个有效的 Java 类文件。

版本号：紧跟魔数后面的是两个字节，表示.class 文件的次版本号和主版本号，帮助 JVM 识别编译时的 Java 版本。
常量池：存储类、方法名、字符串字面量等，提供指向各种数据的引用池，方便字节码中使用。常量池是.class 文件中最复杂的部分。
访问标志：用于标志类的访问权限和其他属性，比如是否为 public、abstract 或 final 等。
类信息：包括当前类和父类的名称索引，指向常量池中的对应项。
接口信息：记录该类实现的接口列表，用于支持多态和接口调用。
字段表：包含类的字段信息（如属性的名称、类型和访问权限）。
方法表：包含类的方法信息，包括方法名、参数类型、返回类型和字节码等。每个方法都有自己对应的字节码，这就是 JVM 运行的核心内容。
属性表：记录附加信息，比如 SourceFile 属性表示源代码文件的名称，LineNumberTable 记录代码行号信息等，有助于调试和反编译。
这些信息使.class 文件能够被 JVM 加载、解析并执行，从而实现 Java 程序的跨平台能力。

4 ClassFileWriter 类

以下为 javassist 手册中的例子：

```
package sample;
public class Test {
    public int value;
    public long value2;
    public Test() {
        super();
    }
    public one() {
        return 1;
    }
}
```

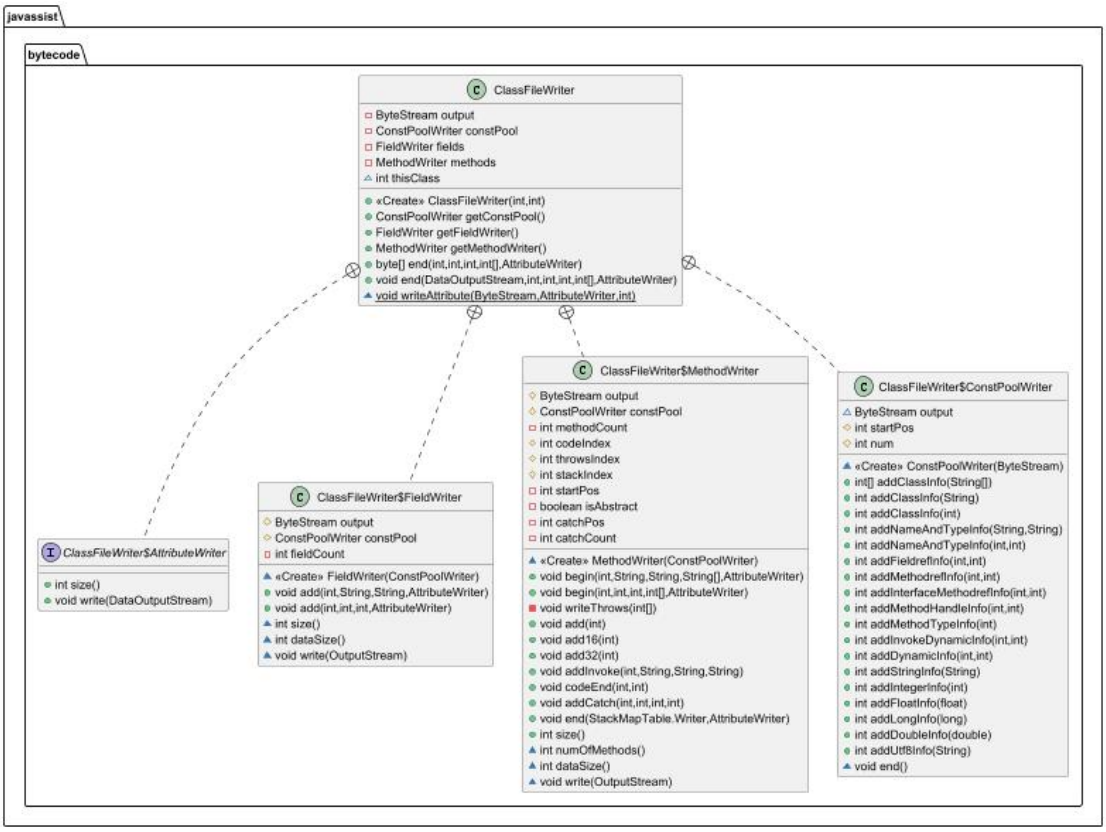
创建以上的类时需要用到以下功能：
1.创建字节码对象，写入头部信息
2.添加常量
3.添加字段/成员变量
4.添加方法
5.更新附加属性
需要以下的类来实现其功能：

类	属性	方法
类文件编写器	.class 的当前类、父类、常量池数据、成员数据、方法数据、输出流	填写初始数据、填写常量池、填写字段表、填写方法表、传递表项至外界
常量编写器	常量输出流、写入的起始位置、已写入的常量个数	Java 规范中 14 种项目类型各自的添加方法、常量池数据写回至输出流
字段编写器	字段输出流、字段的常量池、	添加一个字段、输出字段个

	已写入的字段个数	数、输出字段占用空间大小、写回字段表
方法编写器	方法输出流、方法的常量池、已写入的方法数、写入的起始位置、代码段索引、例外程序索引、栈索引、抽象方法标志位、 catch 子句位置、catch 子句数目	填入方法头、添加字节指令、添加 catch 子句、添加 invoke 调用、汇总 code 信息、输出方法数、输出方法占用空间、 写 throw 信息、写回方法表

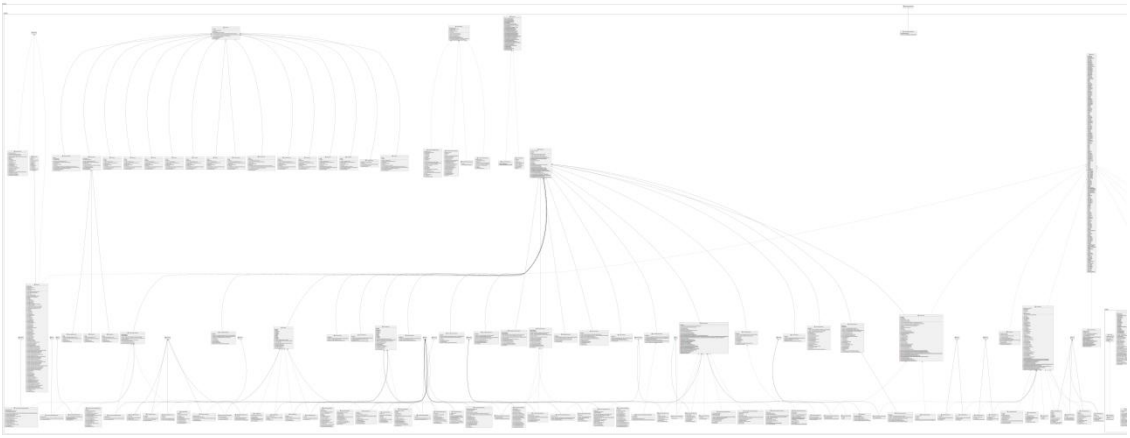
以上功能可通过 ClassFileWriter 类中所提供的 API 实现。

在 ClassFileWriter 中，提供了 ConstPoolWriter，FieldWriter 和 MethodWriter 三个类以实现上述功能。而填写文件头的初始信息可以通过 ClassFileWriter 的构造方法完成。



可以看到，AttributeWriter 接口将更新附加属性这一功能抽象出来，其余三个 Writer 类通过实现该接口获得更新属性的功能。这简化了代码的复杂度，体现了继承和复用的思想。

5 bytecode 模块



可以看出,Bytecode 类是 ByteVector 的子类,而 FloatInfo, Utf8Info 和 StringInfo 等数据类型的信息则继承自 ConstInfo 这个父类，将数据类型的宽度等信息作 为常量存储在.class 文件中，为跨平台的特点提供支持。 ClassFile，即类文件类主要依赖于 ConstInfo, FieldInfo, MethodInfo 以及 AttributeInfo 等类。

6.bytecode 处理类文件的流程

使用 bytecode 模块处理类文件的流程可以被概括为：
获取一个 classfile 对象 — 编辑字段（field）— 编辑方法 — 将 classfile 写回至文件。这里的编辑包括增-删-改-查等操作。在这一过程中的每一步，都可能对常量池和属性表进行修改。其中最主要的类为 ClassFile，FieldInfo，MethodInfo，ConstPool 和 Attribute Info，而子模块 analysis，annotation 和 stackmap 是工具模块，为文件中表项的编辑提供支持。

7.类文件对象的获取

从字节码的层面编辑一个类文件，首先应该将这个类文件抽象成一个对象来进行操作，按照设计需求，我们需要动态修改的类文件可能已经在运行中，也有可能是静态的，还存储于硬盘而未搬到内存。ClassFile 类就提供了这样一层抽象，.class 文件会被抽象为属于 ClassFile 类的对象 classfile。classfile 可能来自输入 DataInputStream，也可能来自编译时类 CtClass，两种情况需要做不同的处理，这通过构造方法的重载来区分。

ClassFile 类和 ConstInfo、FieldInfo、MethodInfo 和 AttributeInfo 类存在相互依赖关系，ClassFile 创建了一个抽象的类文件对象，初始化好版本信息等部分数据，创建好常量池、字段表、方法表和属性表这 4 部分供上述 4 个类后续使用，并将这些类/表的 get 和 set 方法封装成整个文件每部分的 get 和 set 方法。而如果 classfile 对象是从 CtClass 中获得的，则需要调用这 4 个类的 get 方法来构建该对象，所以存在相互依赖关系。这里的方法重载也是多态性的一种体现，通过调用同名方法来完成不同的功能，实现了一定程度的复用。另外，在 ClassFile 中还有一个值得我们关注的细节是成员 fields，methods 和 attributes，它们代表各自表项中的一个条目，都被声明为泛型，利

用了 List 容器，便于表中条目的增删，而这三种条目的组成又有所不同，具有不同的类型。

8. 维护常量池

在上面提到的 5 个类中，都需要取维护常量池，所以在进入 FieldInfo 之前，有必要先分析一下 ConstPool 类与其他类的交互。当我们在 ClassFile 中创建 ConstPool 类的对象 constPool 时，ConstPool 的构造方法就已经将部分常量池的表项初始化好了（这些初始化好的静态信息用 final 关键字声明，以免子类在继承时修改了这些已经写好的信息）。在 ConstPool.java 文件中，ConstPool 类代表了常量池中的若干条目，并为这些条目提供基本的 get 和 set 方法，而抽象类 ConstInfo 代表了其中的某一条目，由于不同的条目具有不同的组成，且具有不同的操作方法，ConstInfo 中只提供一些共同的 write、print 和 getName 等方法，而更多更具体的方法则交给其子类去实现，是面向父类编程的一种体现。
接下来有 16 个类都是 ConstInfo 抽象类的子类，它们提供了对.class 文件中数据类型、成员、方法和接口等信息的描述，具体信息可以参考上文提到的博客。其中有一个子类 MemberrefInfo 也是抽象类，它在 ConstInfo 的基础上，为 MethedrefInfo，FieldrefInfo 和 InterfaceMethodrefInfo 三个子类提供了更具体的拷贝、取 Hash 值等操作，而与表项组成相关的操作则同样交给这三个子类去实现，体现了复用和解耦的特点。bytecode 模块中的继承关系也主要体现在此处。代表表项的各个子类从父类 ConstInfo 处继承了基本的成员和方法，而仅靠 ConstInfo 又还不足以实现对常量池中单条信息完整的操作，它是一个抽象类，具体的任务还需要区分成下面的 16 个子类才能完成。若干条 ConstInfo 信息组合成一个常量池 ConstPool，于是它们之间存在组合关系。初始化好常量池后，它此后可能会由 FieldInfo 和 MethodInfo 等类来更新，可以说维护常量池的操作贯穿了 classfile 对象的整个生命周期。

9. 编辑字段

在编写 JAVA 程序时，在创建一个类之后做的第一件事就是为这个类设置一些成员变量，也就是.class 文件中的字段（field），在字节码的层面，这一任务主要由 FieldInfo 类来完成。FieldInfo 类主要依赖于 AttributeInfo、DuplicateMemberException 和 AccessFlag 这三个类，它们分别起着维护字段对应的属性表、抛出成员重复的异常和为成员设置访问标志的作用。在该类中同样将属性表中的一个条目 attribute 声明为 List 泛型，便于增删。

10. 维护属性表

属性表与常量池类似，都需要在编辑.class 类文件的整个过程中维护。AttributeInfo 类即代表了属性表，它提供了对属性表的通用描述，包括对表项的增删改查等通用操作。值得注意的是，多个表项的 remove 方法前带有关键字 `synchronized`，表示该方法带有同步锁，同一时刻只有一个线程能进到该代码片段内。这是为了保证名称匹配和删除这两个操作的原子性，避免当前线程在比对完属性名称到删除该属性的中间有其他线

程插入进来，先把这一项删除了，当原来的进程被调度回来再去删除这一属性，就会发生错误。设计者在安全性方面的考虑是比较周密的，这其中也体现了面向对象程序设计中并发的特点。
bytecode 模块中有 21 个类都是 AttributeInfo 的子类，它们都代表了属性表中的一个部分，从 AttributeInfo 中继承对表项的基本操作。属性表中有一个重要的属性——Code 属性，对方法表的构建起着至关重要的作用，方法体中的字节指令就由这一属性描述。Code 作为属性表中的一个属性，相应的类 CodeAttribute 也继承自 AttributeInfo 父类。Code 属性中主要包含最大栈深度、最大局部变量数、字节指令和例外表等部分，描述了.class 文件在 JAVA 虚拟机上的运行信息，CodeAttribute 类中也提供了这些成员的 get 和 set 方法。
此外，CodeAttribute 还实现了 Opcode 接口。Opcode 接口中提供了每条 JAVA 字节指令中操作码的二进制编码和这些字节指令对栈深度的改变量，没有声明方法，仅提供一些公共的成员变量，实现了复用。Opcode 接口和 Mnemonic 接口之间存在相关关系，这是项目作者在注解中定义的，Mnemonic 接口以字符串的形式给出了这些字节指令的助记符。将字节指令写入到.class 文件中时，并不需要再将助记符记录进去，如果需要将指令输出，再用 Mnemonic 接口去查询就可以了，不需要占用额外的存储空间，同时也实现了两个功能的解耦。CodeAttribute 类中还有一个 exceptions 字段，代表了 Code 属性中例外表的若干条目，是 ExceptionTable 类的对象，所以此处也存在依赖关系。在 ExceptionTable.java 文件中，ExceptionTableEntry 类代表了例外表中的一个条目，其中包含起始 PC、例外入口和捕捉类型等信息；ExceptionTable 类实现了 Cloneable 接口，以调用 clone()方法来克隆异常信息，它代表了例外表中的若干表项，其中的表项，即成员 entries，同样声明为 List 泛型，其类型为 ExceptionTableEntry，所以多个 ExceptionTableEntry 条目组合成一个 ExceptionTable，二者存在组合关系。

11. 编辑方法

由于构造方法也是由字节码序列组成的，它在字节码的层面和普通的方法并没有太大区别，所以在 bytecode 模块中，除了方法的名称有所区别外，我们会以同样的方式对待它们。
编辑一个 classfile 对象中的方法主要是由 MethodInfo 类完成的，其构造方法会先初始化好方法对应的常量池数据和方法名等内容，然后提供了访问标志和描述符的 get 和 set 方法，所以 MethodInfo 类也依赖于 Descriptor 和 AccessFlag 这两个类。
在声明好方法头之后，就需要在方法体内加入字节指令了。字节指令的添加依赖于 Bytecode 类，一个 Bytecode 类的对象代表了一条 JAVA 字节指令，可以通过 CodeAttribute 提供的方法添加到 Code 属性中。而 Bytecode.java 文件中包含了两个类，一个是 ByteVector 类，它实现了 Cloneable 接口，主要是为了使用 clone()方法来克隆字节指令，它以字节向量的粒度提供了操作字节指令的基本方法。另一个类是 Bytecode 类，正是它达成了生成字节指令序列的目的。Bytecode 类继承自 ByteVector 父类，主要是为了复用 clone()和操作字节向量的基本方法，它也实现了 Cloneable 和 Opcode 接口，在 Opcode 提供的操作码的基础上组装出若干字节指令。

12. 与其它类的交互

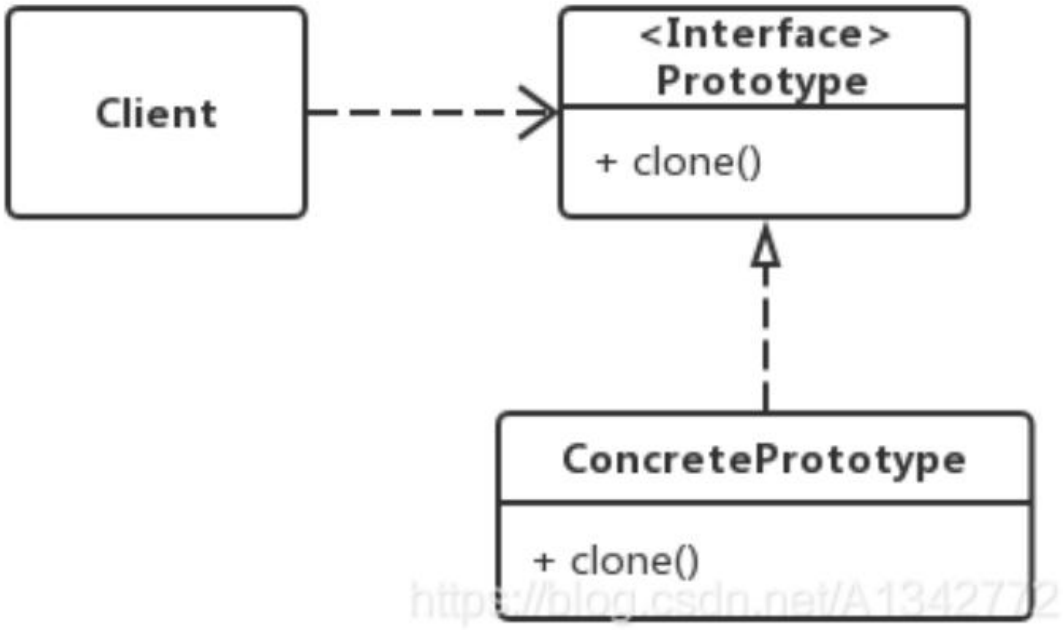
除了 stackmap 包外，bytecode 模块中还包含了另外两个工具包 analysis 和 annotation，前者用于分析字节指令的数据流，可以作为独立的 API，它不在我们的主要流程中。而 annotation 包从字节码的层面去解析注解，并配置相应的属性表。其功能通过 bytecode 模块中的 AnnotationsAttribute 和 AnnotationDefaultAttribute 等四个类封装起来，供模块中的其他类使用。

13. 原型模式

原型模式属于创建者模式中的一种，当我们需要创建一个对象时，如果我们之前已经创建过一个和它一模一样的对象，那么这时我们就可以原封不动地复制一份返回给使用者，降低创建一个对象时的开销，特别是这个对象所属类的构造方法特别复杂时，使用原型模式能够获得较好的性能，可以说是一种创建对象的极佳方式。

bytecode 模块里的原型模式主要体现在 ByteVector、Bytecode 和 ExceptionTable 这三个类中，它们都实现了 Java 的 Cloneable 接口，前两者各自重写的 clone 方法，封装在 Bytecode 类的 clone 方法中。在动态注入字节码时，不免会频繁地遇到重复的字节码，这时就可以通过调用 clone 方法来降低创建一个 Bytecode 对象的开销。

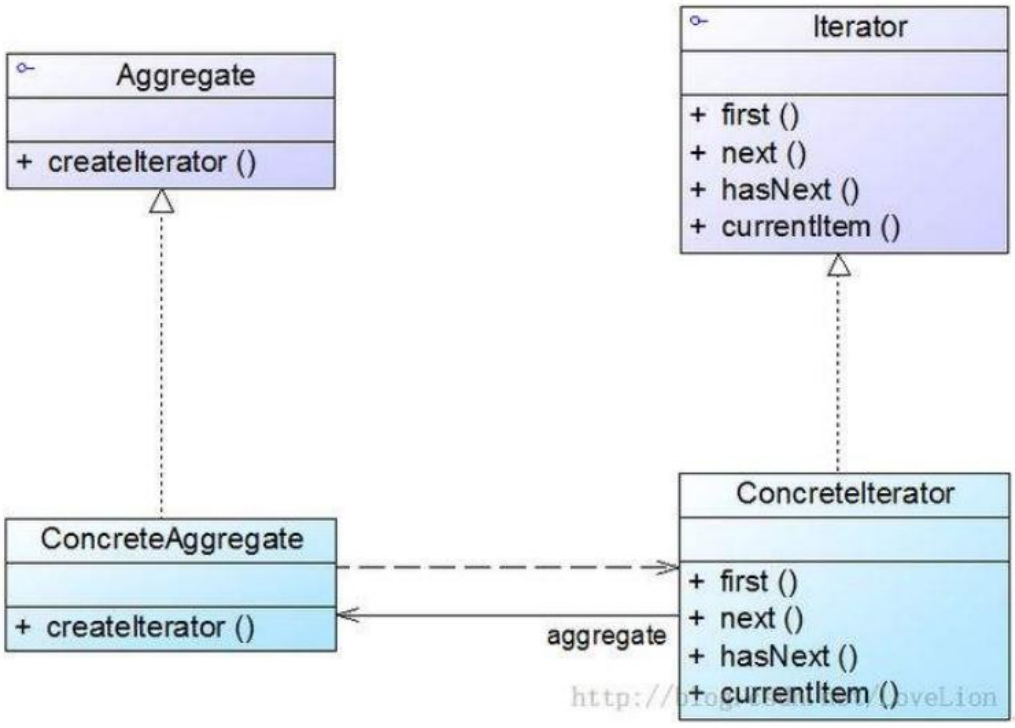
原型模式的结构图如下图所示，图中的 Prototype 为接口，是抽象原型，规定了具体原型需要实现的方法，即 Java 组件中的 Cloneable；ConcretePrototype 是实现这个接口的具体原型，是客户端需要复制的对象，即上文中的 Bytecode，而 Client 是这一原型的实际使用者，即新对象的需求方，也就是 Javassist 的使用者。



14. 迭代器模式

迭代器模式（Iterator Pattern）属于行为模式中的一种，它要求设计者提供一种能够顺序访问一个聚合对象中各个元素的方法，同时又不暴露该对象的内部表示。字节指令作为一个类文件中十分重要的组成部分，在动态注入时我们会对它们进行频繁地访问，在 bytecode 模块中，作者专门设计了一个 CodeIterator 类作为修改 Code 属性，或者说修改一条字节指令的迭代器。除了提供常见的 hasNext()和 next()两个方法以外，还有设计了按下标访问的 byteAt()和按操作码访问的 nextOpcode，以及在中间插入指令的 insert()和在末尾添加指令的 append()等方法，为用户提供了功能齐全的遍历接口。

该设计模式的结构图如下页图 8。图中 Iterator 是抽象迭代器，相当于一个规范，它描述了迭代器应当实现的接口，而 ConcreteIterator 即为此处的 CodeIterator 类，是迭代行为的具体实施者。Aggregate 是抽象的聚合类，描述了遍历对象应当由哪些部分聚合而成（比如一条指令应当由操作码、操作数等域组成，一连串的指令组成了一个指令序列），而 ConcreteAggregate 即为一条一条字节指令聚合成的遍历实体。



在遍历一个聚合体的同时修改这个聚合体可能会产生危险的后果，所以如果需要设计修改的方法，则应当保证迭代器足够健壮。在 CodeIterator 中我们可以看出，大部分代码都与插入指令的 insert()和 append()方法有关，用于维护字节指令的序列和相关的数据结构。作者在该类中并没有设计迭代器常见的 remove()方法，而是选择让用户添加 NOP，将无用的指令覆盖掉作为替代方案。这是因为实现 remove()方法，需要复制一份拷贝供删改，或者使用向聚类注册迭代器的方式，这会极大地增加该类的设计复杂度。

15.抽象工厂模式

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。

前文已经提到过的 ConstPool 体现了该模式。通过创建抽象类 ConstInfo 而不指明具体类，将对象的实例化交给子类，支持不同类型的信息的抽象，即支持拓展。同时交给用户相应的接口，避免用户直接操作子类。

16.其他设计模式

Javassist 遵循代理模式，且运用了其中的动态代理机制。但笔者本人对代理模式的理解不够深入，因此无法给出相应的解析。