**Javassist入门手册**

**Author : Shigeru Chiba**

**Translator : 吕承纲**

## 1. 读写字节码

Javassist是一个Java字节码操作类库， Java字节码被保存在一个被称为class文件的二进制文件中， 每个类文件都包含一个Java类或接口。

Javassist.CtClass是类文件的抽象代表。一个CtClass（编译时类）对象负责处理一个类文件 。下面是个简单的例子：

ClassPool pool = ClassPool.getDefault();

CtClass cc = pool.get("test.Rectangle");

cc.setSuperclass(pool.get("test.Point"));

cc.writeFile();

程序首先获取一个ClassPool对象，此对象通过Javassist控制字节码的修改。ClassPool对象是代表类文件的CtClass对 象的容器。它读取类文件来构建CtClass对象，并且记录对象结构，以便于后面的访问。要修改一个类的定义，用于必须首先通过ClassPool的get()方法来得到代表这个类的CtClass对象。如上所述，我们从ClassPool对象中获取代表类test.Rectangle的CtClass对象，并赋值给变量cc。getDefault()方法用于搜索默认的系统路径并返回ClassPool对象。

从实现的角度看，ClassPool就是CtClass对象的哈希表，以类名称作为键值。ClassPool的get()方法通过指定的键值来搜寻CtClass对象。

通过ClassPool获取到的CtClass对象可被修改（后面将展示如何修改CtClass）。在上面的例子中，类test.Rectangle的父类被修改为test.Point。这个变化将会通过CtClass的writeFile()方法调用最终实现。

writeFile()方法将CtClass对象转化为类文件并写入磁盘中。另外，Javassist还提供了一个直接获取和修改字节码的方法 toBytecode()：

byte[] b = cc.toBytecode();

你也可以直接加载CtClass：

Class clazz = cc.toClass();

toClass()方法会要求类加载器的当前线程来加载代表CtClass的类文件，并返回一个代表加载类的java.lang.Class对象。更多详情，请见本章的下面说明。

#### 定义一个新类

要定义一个新类，请使用ClssPool的 makeClass()方法。

ClassPool pool = ClassPool.getDefault();

CtClass cc = pool.makeClass("Point");

上面的代码定义了一个没有任何成员的Point类。Point的成员方法可以通过CtNewMethod的工厂方法创建出来并通过CtClass的addMethod()方法添加到Point类中。

makeClass()方法不能创建一个新的接口，创建接口要使用ClassPool的makeInterface()方法。接口方法可以通过CtNewMethod的abstractMethod()方法创建。请注意接口方法是抽象的。

冻结类

如果一个CtClass对象通过 writeFile(), toClass(), 或toBytecode() 方法被转换为类文件，Javassist就冻结了此对象。对此CtClass对象的后续修改都是不允许的。这是为了警告那些试图修改已经被加载的类文件的开发者，因为JVM不允许再次加载同一个类。

一个冻结的CtClass对象可以被解冻，这样类定义的修改就被允许。例如：

CtClasss cc = ...;

:

cc.writeFile();

cc.defrost();

cc.setSuperclass(...); // OK since the class is not frozen.

执行 defrost()方法后，CtClass对象就可再次被修改。

如果 ClassPool.doPruning()方法设置为true，Javassist可以优化调整一个被冻结的CtClass对象的数据结构。优化调整指的是为了减少内存使用，去除对象内的一些不必要的属性（比如attribute\_info，方法体中的Code\_attribute）。因此，当一个CtClass对象被优化调整后，一个方法的字节码除了方法名，方法签名和注解外都是不可访问的。优化后的CtClass对象不能被再次解冻。ClassPool.doPruning()方法默认值为false。

对一个CtClass对象上执行 stopPruning()方法，可防止其优化调整：

CtClasss cc = ...;

cc.stopPruning(true);

:

cc.writeFile(); // convert to a class file.

// cc is not pruned.

CtClass对象cc不会被优化。这样，在调用writeFile()后还可以被解冻。

注意：你可能想在调试时暂时不优化并冻结对象，以便于将一个改变了的类文件写入磁盘中。debugWriteFile()方法可以方便的达到这个目的。它会先停止优化调整，写入一个class文件，再解冻这个对象，并且再次打开优化开关（如果开始时是打开优化开关的）。

#### 类搜索路径

静态方法 ClassPool.getDefault() 返回的缺省ClassPool会搜索和当前JVM相同的搜索的路径。如果 程序是运行在譬如JBoss和Tomcat之类的web应用服务器上，ClassPool对象可能就找不到用户自己的类，这是由于web应用服务器除了使 用系统类加载器之外，还使用其他多个类加载器。在这种情况下，额外的类路径就需要注册到ClassPool中。假定pool是对ClassPool对象的 引用：

pool.insertClassPath(new ClassClassPath(this.getClass()));

上面的语句将this对象对应的类路径注册进来。除了使用 this.getClass()，你还可以使用任何Class对象作为参数。用于类对象的类加载路径就这样被注册进来了。

你也可以用目录名称作为类搜索路径。比如，下面的代码将 /usr/local/javalib 目录加到了搜索路径中：

ClassPool pool = ClassPool.getDefault();

pool.insertClassPath("/usr/local/javalib");

搜索路径不仅可以是目录，还可以是URL：

ClassPool pool = ClassPool.getDefault();

ClassPath cp = new URLClassPath("www.javassist.org", 80, "/java/", "org.javassist.");

pool.insertClassPath(cp);

上面的代码将 "http://www.javassist.org:80/java/" 加入类搜索路径中。这个URL只能搜索 org.javassist包下的类。比如，要加载org.javassist.test.Main 这个类，class文件可以这样获取：

http://www.javassist.org:80/java/org/javassist/test/Main.class

另外，你还可以通过直接通过字节码构造的方式来获取CtClass对象。要这么做，请使用ByteArrayClassPath()方法。例如：

ClassPool cp = ClassPool.getDefault();

byte[] b = a byte array;

String name = class name;

cp.insertClassPath(new ByteArrayClassPath(name, b));

CtClass cc = cp.get(name);

代表类文件的CtClass对象是通过 b 构造的。当get()方法被调用时，ClassPool通过给定的ByteArrayClassPath来读取类文件，这种方式和通过名称获取CtClass对象是相同的。

如果你不知道类的全限定名，你可以使用ClassPool的 makeClass()方法：

ClassPool cp = ClassPool.getDefault();

InputStream ins = an input stream for reading a class file;

CtClass cc = cp.makeClass(ins);

makeClass() 方法从给定的输入流返回CtClass对象。你可以使用makeClass()方法来快速的将类文件加入到ClassPool对象中。这对于大的jar包搜索来说是一种性能优化。因为 ClassPool是按需读取class文件，这样会造成对jar包内每个文件的重复搜索。makeClass()能起到优化搜索的作用，因为通过makeClass()方法构造的CtClass对象会保持在ClassPool对象中，而与之对应的类文件不会被读取。

用户也可以扩展类搜索路径。你可以定义一个新的ClassPath接口实现类，并将其实例通过insertClassPath() 方法放入ClassPool中。这样允许非标准资源加入到搜索路径中。

## 2. ClassPool

举个例子，假定一个新的getter()方法被加入到类Point对应的CtClass对象中。之后，程序需要编译含有getter()方法的Point源码，并将编译代码加入到另一个类Line。如果代表Point的CtClass丢失的话，编译器就不能编译getter()方法，因为原始的类定义并不包含getter()方法。因此，要正确的编译一个方法调用，ClassPool一定需要包含执行期间所有的CtClass对象。

#### 避免内存溢出

当CtClass对象很大时（这种情况很少发生，因为Javassist会通过各种方式减少内存消耗），对应的ClassPool就会消耗大量内存。要避免这种情况发生，你可以显式的删除不必要的CtClass对象。当你调用CtClass对象的 detach()方法时，CtClass对象就会从ClassPool中删除掉。例如：

CtClass cc = ... ;

cc.writeFile();

cc.detach();

当 detach()方法调用后，你不能再调用 CtClass对象的任何方法。不过，你可以通过ClassPool的get()方法获取一个新的实例。当你调用get()方法时，ClassPool会再次读取class文件并创建一个新的CtClass对象。

另一种方式是用新的ClassPool替代旧的ClassPool。如果旧的ClassPool被垃圾回收，ClassPool中的CtClass对象也同样会被回收掉。创建新的ClassPool代码片段如下：

ClassPool cp = new ClassPool(true);

// if needed, append an extra search path by appendClassPath()

这种方式创建的ClassPool和通过 ClassPool.getDefault() 获取的ClassPool行为一致。ClassPool.getDefault()只是个方便使用的单例模式。上述代码会创建一个新的ClassPool对象。getDefault()方法获取的ClassPool并没有特别之处，只是方便使用而已。

new ClassPool(true) 是个方便的构造器，它会将系统搜索路径加入到ClassPool对象中。上述构造方法和下面代码作用一样：

ClassPool cp = new ClassPool();

cp.appendSystemPath(); // or append another path by appendClassPath()

#### 层叠 ClassPool

如果程序是运行在web应用服务器上，就会有可能创建多个ClassPool实例；每个ClassPool对应一个ClassLoader。程序应该通过ClassPool的构造器而不是 getDefault()方法来创建ClassPool对象。

就像 java.lang.ClassLoader，ClassPool之间也存在层叠关系。比如：

ClassPool parent = ClassPool.getDefault();

ClassPool child = new ClassPool(parent);

child.insertClassPath("./classes");

当 child.get()调用时，子ClassPool会首先委派给父ClassPool。当父ClassPool没找到这个类文件时，子ClassPool才会在 ./classes 目录下寻找此类文件。

当设置 child.childFirstLookup为 true时，子ClassPool就会先于父ClassPool来寻找此类文件。比如：

ClassPool parent = ClassPool.getDefault();

ClassPool child = new ClassPool(parent);

child.appendSystemPath(); // the same class path as the default one.

child.childFirstLookup = true; // changes the behavior of the child.

#### 通过改变类名来定义新类

一个新类可被定义为已有类的拷贝。程序如下：

ClassPool pool = ClassPool.getDefault();

CtClass cc = pool.get("Point");

cc.setName("Pair");

上面程序首先获取 Point的CtClass对象。之后这个CtClass对象通过setName()方法调用被赋予新名称Pair。从此，CtClass对象所代表的类类名就从Point变更为 Pair。类定义的其他部分则保持不变。

CtClass的 setName()方法修改了ClassPool对象的映射记录。从实现的角度看，ClassPool对象是CtClass对象的哈希表。setName()方法改变了CtClass对象在此哈希表中的key关联。key从原先的类名变更为新的类名。

因此，当ClassPool对象的 get("Point")方法再次调用，不会将CtClass对象返回给cc变量。ClassPool对象会再次读取 Point.class 文件并重新构造一个新的 Point CtClass对象，而之前关联Point的CtClass对象已经不存在了。如下：

ClassPool pool = ClassPool.getDefault();

CtClass cc = pool.get("Point");

CtClass cc1 = pool.get("Point"); // cc1 is identical to cc.

cc.setName("Pair");

CtClass cc2 = pool.get("Pair"); // cc2 is identical to cc.

CtClass cc3 = pool.get("Point"); // cc3 is not identical to cc.

cc1和cc2指向同一个CtClass对象cc，而cc3却不是。注意：当cc.setName("Pair")执行后，cc和cc1所表示的CtClass对象就指向了Pair类。

ClassPool对象中类和CtClass对象是一种一一映射关系。Javassist不允许两个不同的CtClass对象指向同一个类，除非是两个独立的ClassPool。这是和程序转换保持一致的一个重要特性。

要生成和通过 ClassPool.getDefault()获取到的默认ClassPool的拷贝，请执行如下代码：

ClassPool cp = new ClassPool(true);

如果你有两个ClassPool对象，那么在每个ClassPool中都可以获取到相同类文件的不同CtClass对象。你可以修改这些CtClass对象来生成类的不同版本。

#### 通过重命名冻结类来定义新的类

当一个CtClass对象通过 writeFile()或toBytecode() 方法变成class文件时，Javassist不允许对CtClass对象的后续修改。因此，当代表Point类的CtClass对象被转换为class文件后，你不能通过执行setName()方法来将Point修改为Pair。下面的代码是错误的：

ClassPool pool = ClassPool.getDefault();

CtClass cc = pool.get("Point");

cc.writeFile();

cc.setName("Pair"); // wrong since writeFile() has been called.

要规避这个限制，你可以执行 ClassPool的getAndRename() 方法。比如：

ClassPool pool = ClassPool.getDefault();

CtClass cc = pool.get("Point");

cc.writeFile();

CtClass cc2 = pool.getAndRename("Point", "Pair");

当getAndRename()方法执行时，ClassPool读取Point.class文件并生成新的CtClass对象。并且，在记录到哈希表之前，它将CtClass对象名称从Point修改为Pair。因此，在writeFile()或toBytecode()方法执行后， getAndRename()方法是可以被调用的。

## 3. 类加载器

如果要修改的类能提前知道，那么修改类最方便的途径就是：

* 1. 通过调用ClassPool.get()方法获取CtClass对象
* 2. 修改
* 3. 对CtClass对象通过调用 writeFile()或toBytecode()方法写入到class文件中。

如果一个类并不是在加载时就能确定是否需要修改，那么我们就需要用到类加载器。Javassist可以在加载时使用类加载器，这样字节码就可以修改了。开发者可以使用自定义的类加载器，也可以使用Javassist中的类加载器。

### 3.1 CtClass.toClass( ) 方法

CtClass提供了一个方便的 toClass()方法来从线程上下文中加载代表这个CtClass对象的类。要调用这个方法，调用者需要拥有恰当的权限，否则就会抛出SecurityException异常。

下面代码展示如何使用 toClass()方法：

public class Hello {

public void say() {

System.out.println("Hello");

}

}

public class Test {

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

ClassPool cp = ClassPool.getDefault();

CtClass cc = cp.get("Hello");

CtMethod m = cc.getDeclaredMethod("say");

m.insertBefore("{ System.out.println(\"Hello.say():\"); }");

Class c = cc.toClass();

Hello h = (Hello)c.newInstance();

h.say();

}

}

Test.main()在Hello的say()方法前插入了 println()调用。然后，构造了一个修改了的Hello类并调用say() 方法。

注意上面的代码，在执行 toClass()方法前，类Hello没有被加载。如果不是这样，JVM在 toClass()方法请求加载修改了的Hello前会加载最初的Hello类，这样加载修改了的Hello类就会失败（抛出LinkageError异常）。比如，如果Test的main()是这样的：

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

Hello orig = new Hello();

ClassPool cp = ClassPool.getDefault();

CtClass cc = cp.get("Hello");

:

}

第一行中原始的Hello类就加载了，之后再调用 toClass()方法就会抛异常，这是由于类加载器不能同时加载两个不同版本的Hello类。

如果程序运行在JBoss和Tomcat之类的Web应用服务器上，toClass() 方法使用上下文类加载器可能会有错。在这种情况下，可能会抛出ClassCastException异常。要避免这种异常，在调用toClass()方法时需要显式的指定类加载器。比如，如果是个会话bean对象，下面的代码：

CtClass cc = ...;

Class c = cc.toClass(bean.getClass().getClassLoader());

就会正常工作。你必须在 toClass()方法中指明类加载器。

toClass()只是提供了一种方便的手段。如果你需要更强大的功能，需要实现你自己的类加载器。

### 3.2 Java中的类加载器

在Java中，多个类加载器可以共存，并且每个类加载器有自己的名词空间。不同的类加载器可以加载具有相同类名称的不同类文件，加载出来的类是不同的。这种特性允许我们在同一个JVM中运行多个程序，即便这些程序包含具有相同名称但不同实现的类。

注意：JVM不允许动态加载类。当一个类被加载后，不能在运行时再加载修改后的类。因此，当JVM加载一个类后，不能再对这个类做修改。除非使用JDPA（Java平台调试体系结构）才有条件的支持类的重新加载。见 [3.6节](file:///D:\installed\javassist-3.8.0\tutorial\Javassist%E5%85%A5%E9%97%A8%E6%89%8B%E5%86%8C.html#hotswap).

如果相同的类文件被两个不同的类加载器加载，JVM会生成两个名称和定义相同的不同类。这两个类是不等同的，一个类的实例是不能赋值给另一个类的。这两个类之间的转换操作会抛出 *ClassCastException* 异常。

例如，下面的代码会抛出异常：

MyClassLoader myLoader = new MyClassLoader();

Class clazz = myLoader.loadClass("Box");

Object obj = clazz.newInstance();

Box b = (Box)obj; // this always throws ClassCastException.

类 Box 被两个类加载器加载。假定CL这个类加载器通过代码的方式加载类。代码的方式指的是通过 MyClassLoader，Class， Object，和Box，当然 CL也加载了这些类。而obj是myLoader类加载器加载的另一个Box类。由于变量b和变量obj是Box类不同对象，因此在最后一行代码转换时会抛出ClassCastException异常。

多个类加载器形成一个树状的结构。除了bootstrap类加载器外，每个类加载器都有一个父类加载器。由于一个类的加载可被委派给他的父 类加载器，因此类可被你所未指定的类加载器加载。也就是说，我们所指定的加载C的类加载器和实际加载C的类加载器不同。比如，我们可以称前者为C的初始加 载器，而后者为C的实际加载器。

更进一步的，如果指定的类加载器CL（初始加载器）将加载C的工作委派给父加载器PL，那么，CL就不会再去加载C中其他类的引用。CL也不会是这些类的初始加载器，PL才是这些类的初始加载器。类C中所引用的其他类的加载是由C的实际加载器加载的。

要理解上面的行为，请看下面的代码：

public class Point { // loaded by PL

private int x, y;

public int getX() { return x; }

:

}

public class Box { // the initiator is L but the real loader is PL

private Point upperLeft, size;

public int getBaseX() { return upperLeft.x; }

:

}

public class Window { // loaded by a class loader L

private Box box;

public int getBaseX() { return box.getBaseX(); }

}

假定类Window是由类加载器L加载。Window的初始和实际类加载器都是L。因为类Window的定义中有对Box的引用，因此JVM会要求 L加载Box。这里，我们假定L将此任务委派给其父加载器PL。这样，Box的初始加载器为L，但实际加载器为PL。在这种情况下，Point的初始加载 器就是PL，而不是L，因为Point的初始加载器和Box的实际加载器需要一致。这样，L就不会要求去加载Point。

下面，我们考虑下上述代码的稍许改动。

public class Point {

private int x, y;

public int getX() { return x; }

:

}

public class Box { // the initiator is L but the real loader is PL

private Point upperLeft, size;

public Point getSize() { return size; }

:

}

public class Window { // loaded by a class loader L

private Box box;

public boolean widthIs(int w) {

Point p = box.getSize();

return w == p.getX();

}

}

现在，Window的定义中含有对Point的引用。这种情况下，L会将Point的加载委派给PL（两个不同的类加载器不能加载同一个类）。

如果L没有将Point的加载委派给PL，widthIs() 方法就会抛出ClassCastException异常。由于Box的实际加载器是PL，那么Box中的Point引用也是PL加载的。这种情况下，getSize()方法返回的Point对象是由PL加载的，而widthIs()方法中变量p的类型是由L加载的，JVM会认为这是两个不同的类型，因此会抛出类型不匹配异常。

这种特性看起来很别扭，但是很有必要的。如果下面的代码：

Point p = box.getSize();

没有抛出异常，那么Window的开发者就会破坏Point对象的封装性。比如，如果成员x是PL加载的Point的私有域，那么，类Window就可以通过如下的定义由类加载器L直接访问Point中的x：

public class Point {

public int x, y; // not private

public int getX() { return x; }

:

}

关于Java类加载器的更多细节，请阅读：

Sheng Liang and Gilad Bracha, "Dynamic Class Loading in the Java Virtual Machine",   
*ACM OOPSLA'98*, pp.36-44, 1998.

### 3.3 使用 javassist.Loader

Javasssit提供了javassist.Loader类加载器。 这个类加载器使用javassist.ClassPool对象来读取类文件。

比如，javassist.Loader能用来加载被Javassist修改了的类。

import javassist.\*;

import test.Rectangle;

public class Main {

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

ClassPool pool = ClassPool.getDefault();

Loader cl = new Loader(pool);

CtClass ct = pool.get("test.Rectangle");

ct.setSuperclass(pool.get("test.Point"));

Class c = cl.loadClass("test.Rectangle");

Object rect = c.newInstance();

:

}

}

上面的代码修改了类 test.Rectangle。其父类被设置为 test.Point。当程序再次加载修改后的类时，会创建出类test.Rectangle的新的对象。

你可以通过给javassist.Loader 加一个监听事件满足在加载时修改一个类。这个事件在类加载时被触发。事件监听类需要实现如下的接口：

public interface Translator {

public void start(ClassPool pool)

throws NotFoundException, CannotCompileException;

public void onLoad(ClassPool pool, String classname)

throws NotFoundException, CannotCompileException;

}

start()方法在监听器通过 javassist.Loader的addTranslator()方法加入监听器的时候被调用。onLoad() 方法在class. onLoad()之前被调用，这样就可以修改一个加载的类了。

比如，下面的监听器在类加载时将类访问权限修改为public。

public class MyTranslator implements Translator {

void start(ClassPool pool)

throws NotFoundException, CannotCompileException {}

void onLoad(ClassPool pool, String classname)

throws NotFoundException, CannotCompileException

{

CtClass cc = pool.get(classname);

cc.setModifiers(Modifier.PUBLIC);

}

}

注意 onLoad()方法不需要调用 toBytecode()或writeFile()方法，因为javassist.Loader会调用这些方法来获取类文件。

要运行带有MyTranslator的MyApp对象，代码如下：

import javassist.\*;

public class Main2 {

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

Translator t = new MyTranslator();

ClassPool pool = ClassPool.getDefault();

Loader cl = new Loader();

cl.addTranslator(pool, t);

cl.run("MyApp", args);

}

}

要运行这个程序，请：

% java Main2 *arg1* *arg2*...

类MyApp和其他类都会被MyTranslator转换。

请注意类MyApp不能访问Main2，MyTranslator，和ClassPool，因为它们是由不同的类加载器加载的。MyApp是由javassist.Loader加载，而Main2是由默认的Java类加载器加载。

javassist.Loader和java.lang.ClassLoader搜索类的方式不一致。ClassLoader会先委派父类加载器进行加载，只有在父类加载器不能加载时才自己加载。而javassist.Loader 在委派给父类加载器加载前会自己加载，只有在如下的情况下才会委派：

* 不能通过ClassPool对象的get()方法加载，或
* 类已经通过 delegateLoadingOf()方法明确的指明由父类加载器加载。

这种搜索方式允许Javassist在加载时修改类。如果由于某些原因，加载不到修改的类时，它会委派给父类加载器。当一个类由父类加载器加载后， 这个类的其他实例也会由父类加载器加载，并且不能再修改。回想前面说到的类C是由C的实际类加载器加载的。如果你的程序加载一个修改类失败了，请确认是否 所有的类都是由javassist.Loader加载的。

### 3.4编写类加载器

一个使用Javasssit的简单类加载器如下：

import javassist.\*;

public class SampleLoader extends ClassLoader {

/\* Call MyApp.main().

\*/

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

SampleLoader s = new SampleLoader();

Class c = s.loadClass("MyApp");

c.getDeclaredMethod("main", new Class[] { String[].class })

.invoke(null, new Object[] { args });

}

private ClassPool pool;

public SampleLoader() throws NotFoundException {

pool = new ClassPool();

pool.insertClassPath("./class"); // MyApp.class must be there.

}

/\* Finds a specified class.

\* The bytecode for that class can be modified.

\*/

protected Class findClass(String name) throws ClassNotFoundException {

try {

CtClass cc = pool.get(name);

// modify the CtClass object here

byte[] b = cc.toBytecode();

return defineClass(name, b, 0, b.length);

} catch (NotFoundException e) {

throw new ClassNotFoundException();

} catch (IOException e) {

throw new ClassNotFoundException();

} catch (CannotCompileException e) {

throw new ClassNotFoundException();

}

}

}

类MyApp是个应用程序。要执行这个程序，请将class文件放入./class目录下，并确保此目录不在类搜索路径中。否则，MyApp.class 将会由系统默认类加载器，也就是SampleLoader的父类加载器加载。./class目录需要在构造函数中通过insertClassPath()加入。你也可以选择一个你喜欢的其他目录名称。之后请如下执行：

% java SampleLoader

类加载器会加载 MyApp (./class/MyApp.class)并执行 MyApp.main()方法。

这是使用Javassist最简单的方式。不过，如果你要写一个复杂的类加载器，你需要了解更多的Java类加载机制。比如，上面的代码将 MyApp和SampleLoader放入两个独立的名称空间中，这是由于它们是由不同的类加载器加载的。因此，类MyApp不能直接访问类 SampleLoader。

### 3.5 修改系统类

诸如象java.lang.String 之类的系统类只能由系统类加载器加载。因此，上面所说的SampleLoader或javassist.Loader不能在加载时修改系统类。

如果你的程序需要这么做，这些系统类只能被静态修改。比如，下面的代码将一个新字段hiddenValue加入到java.lang.String中：

ClassPool pool = ClassPool.getDefault();

CtClass cc = pool.get("java.lang.String");

cc.addField(new CtField(CtClass.intType, "hiddenValue", cc));

cc.writeFile(".");

上面的代码生成了一个新的文件"./java/lang/String.class"。

要运行String修改了的MyApp程序，请执行：

% java -Xbootclasspath/p:. MyApp *arg1* *arg2*...

假定MyApp如下定义：

public class MyApp {

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

System.out.println(String.class.getField("hiddenValue").getName());

}

}

如果修改了的String正确加载，MyApp就会打印出hiddenValue。

注意：应用出于某种目的，使用这种技巧覆盖了*rt.jar* 中的系统类，则不能正确部署，因为这违背了*Java 2 运行环境字节码许可*。

### 3.6 运行时重载类

如果JVM启动并开启了JPDA（Java平台调试体系结构），一个类就可以动态的重载。在JVM加载这个类后，原有的类就被卸载了而新的类被加 载。也就是说，类可以在运行时动态的修改。但是，新的类需要兼容旧的类。JVM不允许两个版本的结构变更。它们必须拥有相同的成员和方法。

Javassist提供了个好用的类方便运行时类的重载。更多信息，请阅读API文档中的javassist.tools.HotSwapper。

## 4. 反射和自定义

CtClass提供了反射方法。Javassist中的反射兼容Java 反射API。CtClass提供了getName()，getSuperclass()，getMethods()等等方法，还提供了类修改方法，允许新增成员，构造器和方法。构造一个方法体也是可以的。

方法由CtMethod对象表示。CtMethod中提供了一些方法修改的方法。如果一个方法继承自父类方法，那么代表此方法的CtMethod对象也指向父类。一个CtMethod对象关联方法的声明。

比如，如果类Point中有个 move()方法，并且子类ColorPoint中没有覆盖此方法，那么这两个类中的 move() 方法由同一个CtMethod对象表示。如果CtMethod对象修改了，两个方法都会被修改。如果你只想改变ColorPoint中的 move()方法，你必须先在代表ColorPoint的CtMethod对象上拷贝Point的move()方法。CtMethod对象的拷贝可以通过CtNewMethod.copy()。

Javassist不允许删除方法或成员，但可以重命名。因此，当一个方法没用的时候，可以通过执行setName()和setModifiers()来重命名和修改为私有方法。

Javassist不允许对已有方法添加额外入参。要这么做，可以通过定义一个新的方法实现。比如，如果你想给下面方法额外加入int型的参数：

void move(int newX, int newY) { x = newX; y = newY; }

你可以在Point中加入下述方法：

void move(int newX, int newY, int newZ) {

// do what you want with newZ.

move(newX, newY);

}

Javassist还支持直接class文件操作的低级API函数。比如，CtClass中的getClassFile() 方法返回代表class文件的ClassFile对象。CtMethod中的getMethodInfo()返回类文件中代表method\_info 结构的MethodInfo对象。低级API函数使用Java虚拟机规范中的词汇。使用者需要了解class文件和字节码。更多信息，请参见[javassist.bytecode package](file:///D:\installed\javassist-3.8.0\tutorial\tutorial3.html#intro) 。

运行时修改类如果需要使用以$开头特殊标识符，就需要依赖 javassist.runtime包。这些特殊的标示符（$开头的）将会在下面说明。而不需要这些特殊标示符来修改类则不需要 javassist.runtime 包。更多信息，请查看javassist.runtime包。

### 4.1 在方法的开头/结束插入源代码

CtMethod和CtConstructor提供了insertBefore(), insertAfter(), 和 addCatch()方法，用于在一个已有方法体中插入代码。Javassist有一个简单的Java编译器用于处理这些源码。编译器接受源码并编译为字节码，并被内联到方法体内。

根据行号来插入代码也是可以的（如果class文件中包含行号）。CtMethod和CtConstrctor中的insertAt()方法根据源码和源文件的指定行号，编译源码并插入指定行号处。

insertBefore(), insertAfter(), addCatch(), 和 insertAt() 方法接受代表程序段的字符串。程序段可以是if，while或以分号结束的表达式。代码段以大括号封装。下面的每一行都是个有效的代码段：

System.out.println("Hello");

{ System.out.println("Hello"); }

if (i < 0) { i = -i; }

代码段可引用成员和方法。如果方法编译时加入-g 选项（class文件中包含本地局部变量），代码段还可以引用方法参数。否则，只能通过$0, $1, $2, ...这些特殊的变量来访问方法参数。即便在代码段中定义局部变量也是不能在代码段的其他地方访问的。不过，如果方法编译加入了-g 选项，insertAt()方法运行代码段可以在指定的行号访问局部变量。

传入 insertBefore(), insertAfter(), addCatch(), 和 insertAt() 方法的字符串对象由Javassist中的编译器编译。由于编译器支持语言扩展，一些以$开头的标示符就有特定的含义

|  |  |
| --- | --- |
| $0, $1, $2, ... &nbsp &nbsp | 本体和实参 |
| $args | 参数数组。类型为 Object[] |
| $$ | 所有的实参 比如， m($$)等价于 m($1,$2,...) |
| &nbsp |
| $cflow(...) | cflow 变量 |
| $r | 返回类型。用于类型转换. |
| $w | 包装类型。用于类型转换 |
| $\_ | 返回值 |
| $sig | 代表参数类型的java.lang.Class对象 |
| $type | 代表返回类型的java.lang.Class对象 |
| $class | 代表当前修改类的java.lang.Class对象 |

#### $0, $1, $2, ...

方法参数可通过$1, $2, ...方式访问。$1代表第一个参数，$2代表第二个参数，以此类推。参数类型和方法参数一致。$0 代表方法本体。如果方法是静态的（static），则$0不可使用。

变量使用如下。假定有个类Point：

class Point {

int x, y;

void move(int dx, int dy) { x += dx; y += dy; }

}

要打印move()方法中的dx和dy的值，执行如下代码：

ClassPool pool = ClassPool.getDefault();

CtClass cc = pool.get("Point");

CtMethod m = cc.getDeclaredMethod("move");

m.insertBefore("{ System.out.println($1); System.out.println($2); }");

cc.writeFile();

请注意insertBefore()方法中源码用大括号括起来了，这是由于 insertBefore()只接受单条语句或括起来的语句块。

修改后的Point定义如下：

class Point {

int x, y;

void move(int dx, int dy) {

{ System.out.println(dx); System.out.println(dy); }

x += dx; y += dy;

}

}

$1和$2 分别代表dx和dy。

$1, $2, $3 ...是可修改的。如果这些变量赋予了新值，变量所对应的参数也就更新了。

#### $args

变量 $args 代表整个参数数组。变量类型为Object数组类型。如果一个参数类型为原生类型，比如int，则会被转换为包装类型java.lang.Integer并存储在$args中。这样，除非 $1是原生类型，否则$args[0]和$1类型一致。请注意$args[0]不是$0：$0代表的是方法本体。

如果 Object数组被赋予$args，则数组中的每个成员会赋值给每个参数。如果参数类型为原生类型，则对应的就是包装类型。在赋值给参数时包装类型会转变为原生类型。

#### $$

变量$$ 是所有以逗号分隔参数的缩写。比如，如果 move()方法的参数个数有三个，那么

move($$)

等同于：

move($1, $2, $3)

如果move()方法没有任何参数，那么move($$)等同于 move()。

$$ 可被另一个方法使用。如果你这么写：

exMove($$, context)

那么等同于：

exMove($1, $2, $3, context)

Note that $$ enables generic notation of method call with respect to the number of parameters. It is typically used with $proceed shown later.

注意：$$开启了方法调用的参数数量的泛型标记。其配合$proceed的典型使用后面说明。

#### $cflow

$cflow表示“流程控制”。这个只读变量返回一个方法调用的回归深度。

假定下面的方法代表CtMethod对象cm：

int fact(int n) {

if (n <= 1)

return n;

else

return n \* fact(n - 1);

}

要使用$cflow，先声明方法fact()监控调用 $cflow：

CtMethod cm = ...;

cm.useCflow("fact");

useCflow()方法参数是声明$cflow变量的标识符。任何Java的有效命名都可作为标识符。标识符也可以包含.(点号)，比如，"my.Test.fact"就是个有效的标识符。

这样，$cflow(fact)代表cm对应方法的调用深度。当方法第一次调用时，$cflow(fact)的值为0，当方法再次调用时，值就变成1。比如：

cm.insertBefore("if ($cflow(fact) == 0)"

+ " System.out.println(\"fact \" + $1);");

$cflow(fact)每次调用前都会进行检查，因此当fact()递归调用时，不会打印参数的值。

$cflow的值是和当前线程中与指定方法关联的堆栈深度。$cflow在其他方法中也可以访问。

#### $r

$r代表方法的返回类型，被用于类型转换。其典型用法如下：

Object result = ... ;

$\_ = ($r)result;

如果返回类型是个原生类型，那么 ($r)遵循特定的语法规则。如果类型转换的操作类型是个原生类型，($r)会作为普通的转换类型返回。如果操作类型是个包装类型，($r)会作为包装类型返回。比如，如果返回类型是int，那么($r)就会从java.lang.Integer转换为int。

如果返回类型为void，那么($r) 不会转换类型，它什么都不做。因此，如果某个操作是对void方法的调用，则($r)返回null值。比如，如果foo() 是个void方法，其返回类型为void，那么

$\_ = ($r)foo();

是个有效的语句。

转换操作符 ($r)在return语句中也很有用。即便返回类型是void，下面的return语句依然有效：

return ($r)result;

如下，return语句等同于一个没有返回值的return语句：

return;

#### $w

$w 是个包装类型，被用于类型转换。($w)将一个原生类型转换为包装类型。代码如下：

Integer i = ($w)5;

被选择的包装类型依赖 ($w)之后的表达式。如果表达式的类型为double，那么包装类型为java.lang.Double。

如果($w)之后的表达式不是个原生类型，那么($w) 什么也不做。

#### $\_

CtMethod和CtConstructor中的insertAfter()方法用于在方法结尾插入编译代码。在insertAfter()方法语句中，不仅上面说到的 $0, $1, ... 有效， $\_ 也是有效的。

变量$\_代表方法返回值。变量的类型就是方法返回类型。如果返回类型为void，则$\_ 类型是Object，$\_ 返回值为null。

insertAfter() 插入的编译代码既可以在方法正常返回前执行，也可以在方法抛出异常时执行。要在方法抛出异常时执行，insertAfter()第二个参数asFinally需要设置为true。

如果异常抛出时，通过insertAfter()插入的编译代码作为 finally语句执行，则$\_ 的值为0或是null值。当编译代码执行结束后，原先抛出的异常会再次抛给调用者。注意这时$\_ 的值会被抛弃，不会返回为调用者。

#### $sig

$sig值是 java.lang.Class对象数组，代表顺序声明的参数类型。

#### $type

$type值是 java.lang.Class对象数组，代表返回值类型。如果是构造函数，则为Void.class。

#### $class

$class 值是 java.lang.Class对象，代表修改方法所对应的类。$class是$0的类型。

#### addCatch()

addCatch() 方法在方法体中插入一段代码，这样当方法抛出异常并返还给调用者时这段代码会被执行。在插入的代码中，异常使用特殊变量 $e表示。

例如：

CtMethod m = ...;

CtClass etype = ClassPool.getDefault().get("java.io.IOException");

m.addCatch("{ System.out.println($e); throw $e; }", etype);

翻译过来就是：

try {

the original method body

}

catch (java.io.IOException e) {

System.out.println(e);

throw e;

}

注意插入的代码一定要以 throw或return 语句结束。

### 4.2 修改方法体

CtMethod和CtConstructor提供 setBody()方法用于替换整改方法体。Javassist会编译给定的源码为字节码并替换原来的方法体。如果给定的源码是null，则替换的方法体就只有一条return语句，并且返回0或null（返回类型为void的情况）

setBody()方法的给定源码中，以 $开头的标示符具有特殊的含义

|  |  |
| --- | --- |
| $0, $1, $2, ... | 本体和实参 |
| $args | 参数数组。类型为 Object[] |
| $$ | 所有的实参 |
| $cflow(...) | cflow 变量 |
| $r | 返回类型。用于类型转换 |
| $w | 包装类型。用于类型转换 |
| $sig | 代表参数类型的java.lang.Class对象 |
| $type | 代表返回类型的java.lang.Class对象 |
| $class | 代表当前修改类的java.lang.Class对象 |
|  |

注意$\_ 变量在此不可用。

#### 使用表达式替换源码

Javassist允许使用javassist.expr.ExprEditor修改方法体中的某个表达式。开发者通过定义ExprEditor的子类来说明如何修改。

要运行一个 ExprEditor对象，请调用CtMethod或CtClass中的 instrument()方法。比如，

CtMethod cm = ... ;

cm.instrument(

new ExprEditor() {

public void edit(MethodCall m)

throws CannotCompileException

{

if (m.getClassName().equals("Point")

&& m.getMethodName().equals("move"))

m.replace("{ $1 = 0; $\_ = $proceed($$); }");

}

});

搜索cm对应的方法体并将所有对类Point的move() 调用替换为：

{ $1 = 0; $\_ = $proceed($$); }

这样，move()方法的第一个参数始终为0。注意替换体不仅可以是一个表达式也可以是一个语句块。

instrument()搜索整个方法体。一旦其找到诸如方法调用，成员访问或对象创建之类的表达式，那么相应的edit()方法就会调用给定的ExprEditor对象。edit()方法的参数是找到表达式的对象。edit()方法在整个对象中检查和替换表达式。

在edit()中调用参数的 replace() 方法用于替换表达式为给定的语句。如果给定的语句块是个空语句块，也就是说，如果执行了replace("{}")，那么方法体中的逻辑语句都被删除了。如果你想在方法体之前/之后插入语句，可使用下述代码：

{ before-statements;

$\_ = $proceed($$);

after-statements; }

上面的表达式可以是方法调用，成员访问或对象创建。 如果表达式是只读的，你还可以这么写：

$\_ = $proceed();

或者，表达式是只写的

$proceed($$);

如果instrument()搜索的方法编译时加入了-g选项，那么replace()方法中替换的源码还可以使用局部变量。

#### javassist.expr.MethodCall

MethodCall对象代表方法调用。MethodCall中的replace()语句用于在方法调用时的语句替换。传入insertBefore()方法的以$开头的标示符具有特殊的含义。

|  |  |
| --- | --- |
| $0 | 方法调用的目标对象，并不等同于this对象。  this指的是调用方对象。当方法是静态方法时，$0 的值为0 |
|  |
|  |
| $1, $2, ... | 方法调用参数 |
| $\_ | 方法调用返回值 |
| $r | 方法调用返回类型 |
| $class | 代表当前修改类的java.lang.Class对象 |
| $sig | 代表参数类型的java.lang.Class对象 |
| $type | 代表返回类型的java.lang.Class对象 |
| $proceed | 代表最初方法调用的名称 |

这里方法调用指的是MethodCall对象。

其他标示符比如$w, $args和$$ 也可以使用。

除非方法调用的返回类型为void，否则需要返回代表返回值的$\_和$\_的类型。如果返回类型为void，则$\_类型为Object，返回值忽略。

$proceed是个特殊语法而不是一个字符串。后面需要跟带括弧的参数列表。

#### javassist.expr.ConstructorCall

ConstructorCall对象代表构造函数中的构造器调用，比如 this()和 super。ConstructorCall中的replace()方法将构造器调用替换为语句或语句块。传入insertBefore()方法的以$开头的标示符具有特殊的含义。

|  |  |
| --- | --- |
| $0 | 目标对象的构造器调用。等同于this |
| $1, $2, ... | 构造函数中的参数 |
| $class | 代表当前修改类的java.lang.Class对象 |
| $sig | 代表构造器参数的java.lang.Class对象数组 |
| $proceed | 代表最初构造器的名称 |

这里构造器调用指的是ConstructorCall对象。

其他标示符比如$w, $args和$$ 也可以使用。

因为在构造函数中要么调用父类的构造器，要么调用类中其他的构造器，因此在替换代码中需要包含一个构造器调用，通常使用 $proceed()方法。

$proceed是个特殊语法而不是一个字符串。后面需要跟带括弧的参数列表。

#### javassist.expr.FieldAccess

FieldAccess对象代表成员访问。表达式中的成员访问参数会传入ExprEditor中的edit()方法。FieldAccess中的replace()方法会将成员访问替换为插入的代码。

以 $ 开头的标识符具有特殊的含义：

|  |  |
| --- | --- |
| $0 | 访问的成员变量，等同于this。 如果成员是静态变量，$0为null。 |
|  |
|  |
| $1 | 如果成员是可写的，则值保存在$1中。否则，$1没有意义 |  |
|  |
| $\_ | 如果成员是可读的，则值保存在$\_中。否则，$\_被丢弃 |  |
|  |
| $r | 如果成员是可读的，则返回类型保存在$r中。否则，$r为void |  |
|  |
| $class | 代表当前类的java.lang.Class对象 |  |
| $type | 代表成员类型的java.lang.Class对象 |  |
| $proceed | 代表最初访问成员的名称 |  |

其他标示符比如$w, $args和$$ 也可以使用。

如果成员是可读的，这返回值需放入$\_中，$\_的类型为访问成员类型。

#### javassist.expr.NewExpr

NewExpr对象代表新对象的创建（不包括数组创建）。ExprEditor的edit()方法接受创建出来的对象。NewExpr中的replace()方法将创建对象替换为插入代码。

以 $ 开头的标识符具有特殊的含义：

|  |  |
| --- | --- |
| $0 | null. |
| $1, $2, ... | 构造器中的参数 |
| $\_ | 创建对象的返回值。 一个新创建对象必须保存在这个值中 |
| &nbsp |
| $r | 创建对象类型 |
| $sig | 构造函数参数java.lang.Class对象数组. |
| $type | 创建对象java.lang.Class对象 |
| $proceed | 最初创建对象的名称 . |

其他标示符比如$w, $args和$$ 也可以使用。

#### javassist.expr.NewArray

NewArray对象代表数组创建。ExprEditor的edit()方法接受创建出来的数组对象。NewArray中的replace()方法将创建的数组对象替换为插入代码。

以 $ 开头的标识符具有特殊的含义：

|  |  |
| --- | --- |
| $0 | null. |
| $1, $2, ... | 数组每一维的大小 |
| $\_ | 创建数组的返回值  新创建的数组需保存在此变量中 |
| &nbsp |
| $r | 创建数组类型 |
| $type | 创建数组的java.lang.Class对象 |
| $proceed | 最初创建数组的名称 |

其他标示符比如$w, $args和$$ 也可以使用

例如，如果数组如下创建，

String[][] s = new String[3][4];

那么$1和$2的值为3和4，$3不能访问。

如果数组如下创建

String[][] s = new String[3][];

那么$1的值为3，$2不能访问。

#### javassist.expr.Instanceof

Instanceof对象代表instance表达式。ExprEditor的edit()方法接受从instanceof表达式中发现的对象。Instanceof中的replace()方法将此表达式替换为插入代码。

以 $ 开头的标识符具有特殊的含义：

|  |  |
| --- | --- |
| $0 | null. |
| $1 | instanceof表达式中左边的值 |
| $\_ | 表达式的返回值。类型为 boolean. |
| $r | instanceof表达式中右边的值 |
| $type | 代表instanceof表达式中右边类型的java.lang.Class对象 |
| $proceed | 最初instanceof表达式的名称。 如果参数是表达式右边类型的实例，则返回参数对象（Object类型）和true， 否则返回false |
|  |
|  |
|  |

The other identifiers such as $w, $args and $$ are also available.

其他标示符比如$w, $args和$$ 也可以使用

#### javassist.expr.Cast

Cast对象代表显式的类型转换。ExprEditor的edit()方法接受转换类型。Cast中的replace()方法将此表达式替换为插入代码。

以 $ 开头的标识符具有特殊的含义：

|  |  |
| --- | --- |
| $0 | null. |
| $1 | 显式转换的类型值 |
| $\_ | 转换表达式的返回值。  $\_类型和显式转换类型一致。也就是说，和括号中的类型一致。 |
|  |
| $r | 显式转换后的类型，即括号中的类型 |  |
| $type | java.lang.Class 对象，和$r代表的类型一致 |  |
| $proceed | 最初类型转换的方法名称。 在显式类型转换结束后返回java.lang.Object |  |
|  |
|  |

其他标示符比如$w, $args和$$ 也可以使用

#### javassist.expr.Handler

Handler对象代表try-catch语句中的catch部分。ExprEditor的edit()方法接受catch从句。Handler中的replace()方法将此catch部分替换为插入代码。

以 $ 开头的标识符具有特殊的含义：

|  |  |
| --- | --- |
| $1 | catch从句的异常对象 |
| $r | catch从句的异常类型，用于类型转换 |
| $w | 包装类型，用于类型转换 |
| $type | java.lang.Class 对象，代表catch从句的异常类型. |
|  |

如果一个新的异常对象赋予了$1，此值也会作为原始catch从句中的异常捕获。

### 4.3 添加新方法或成员

#### 添加方法

Javassist允许开发者添加新的方法和构造函数。CtNewMethod和CtNewConstructor提供了一些静态工厂方法，比如make()方法来创建 CtMethod或CtConstructor。

举例：

CtClass point = ClassPool.getDefault().get("Point");

CtMethod m = CtNewMethod.make(

"public int xmove(int dx) { x += dx; }",

point);

point.addMethod(m);

x为Point的int型成员变量，上面代码在Point中添加了public方法 xmove()。

make()方法中的脚本可以使用除了$\_之外，所有以$开头的特殊标识符。如果目标对象和目标方法名传入make()方法，也可以使用 $proceed标示符。例如：

CtClass point = ClassPool.getDefault().get("Point");

CtMethod m = CtNewMethod.make(

"public int ymove(int dy) { $proceed(0, dy); }",

point, "this", "move");

程序创建的 ymove()方法定义如下：

public int ymove(int dy) { this.move(0, dy); }

请注意$proceed指代的是this.move。

Javassist还提供另一种创建方法的方式。你可以先创建一个抽象方法，再实现方法体：

CtClass cc = ... ;

CtMethod m = new CtMethod(CtClass.intType, "move",

new CtClass[] { CtClass.intType }, cc);

cc.addMethod(m);

m.setBody("{ x += $1; }");

cc.setModifiers(cc.getModifiers() & ~Modifier.ABSTRACT);

因为当抽象方法加入到类中后，此类就变为抽象类，因此，在调用setBody()方法后必须显式的将此类变为非抽象类。

#### 相互调用方法

对于一个方法中调用另一个还没有在类中实现的方法的方式，Javassist是不支持的（Javassist不支持方法递归调用）。可以通过如下的技巧实现类中方法的互调。假定你想在cc所代表的类中添加方法 m()和n()：

CtClass cc = ... ;

CtMethod m = CtNewMethod.make("public abstract int m(int i);", cc);

CtMethod n = CtNewMethod.make("public abstract int n(int i);", cc);

cc.addMethod(m);

cc.addMethod(n);

m.setBody("{ return ($1 <= 0) ? 1 : (n($1 - 1) \* $1); }");

n.setBody("{ return m($1); }");

cc.setModifiers(cc.getModifiers() & ~Modifier.ABSTRACT);

先创建这个类的两个抽象方法，这样即便两个方法之间存在相互调用， 再实现这两个抽象方法的方法体也是可以的。最后，一定要将类修改为非抽象类，因为当抽象方法加入类后，此类就变为抽象类了。

#### 添加成员

Javassist允许开发者创建新的成员变量。

CtClass point = ClassPool.getDefault().get("Point");

CtField f = new CtField(CtClass.intType, "z", point);

point.addField(f);

上述代码将成员z添加到类Point中。

如果新加的成员需要初始化，则代码可以如下修改：

CtClass point = ClassPool.getDefault().get("Point");

CtField f = new CtField(CtClass.intType, "z", point);

point.addField(f, "0"); // initial value is 0.

addField()方法将第二个参数作为初始化值。参数可以是任意Java表达式，只要此表达式返回类型和参数类型一致即可。请注意表达式不能以分号 (;).结束。

更进一步，上述代码可以这么写：

CtClass point = ClassPool.getDefault().get("Point");

CtField f = CtField.make("public int z = 0;", point);

point.addField(f);

#### 删除成员

可以用CtClass的removeField()和removeMethod()方法来删除一个成员或方法，删除构造函数可以用CtConstructor中的removeConstructor()。

### 4.4 注解

CtClass, CtMethod, CtField和CtConstructor均提供getAnnotations()方法来方便的获取注解。方法返回注解类型对象。

例如：

public @interface Author {

String name();

int year();

}

注解也可以这么使用：

@Author(name="Chiba", year=2005)

public class Point {

int x, y;

}

注解值可通过 getAnnotations()获取，返回注解类型对象数组。

CtClass cc = ClassPool.getDefault().get("Point");

Object[] all = cc.getAnnotations();

Author a = (Author)all[0];

String name = a.name();

int year = a.year();

System.out.println("name: " + name + ", year: " + year);

上面代码最后打印：

name: Chiba, year: 2005

因为Point只有@Author 这一个注解，因此数组的长度为1，all[0]即为Author注解对象。注解值通过@Author对象的name()和year()方法获取

要使用getAnnotations()方法，注解类型（比如上面的Author）必须在类搜索路径中。注解是必须能通过ClassPool对象访问的。如果注解类型不在类搜索路径中，Javassist就获取不到注解成员的默认值。

### 4.5 运行时支持类

大多数情况下，使用Javassist修改类不需要启动Javassist，但也有些字节码生成需要Javassist运行时的支持类，这里类都在javassist.runtime包中。javassist.runtime包是唯一支持Javassist运行时修改类文件的包。其他Javassist中的类都不会再运行时修改类文件。

### 4.6 导入

源码中的所有类都必须是全限定名（包含包名），只有java.lang包是个例外。比如，Javassist编译器会把Object当做java.lang.Object。

要告诉编译器在解析类名时搜索包名，请使用ClassPool的importPackage() 方法。比如：

ClassPool pool = ClassPool.getDefault();

pool.importPackage("java.awt");

CtClass cc = pool.makeClass("Test");

CtField f = CtField.make("public Point p;", cc);

cc.addField(f);

上述第二行代码指导编译器导入 java.awt包，这样，第三行代码才不会抛出异常。编译器会知道Point代表的是java.awt.Point。

注意ClassPool中的importPackage()方法不会影响get()方法。只有编译器才考虑导入包，get()方法的参数必须是全限定名。

### 4.7 限制

在目前的实现下，Javassisct的编译器有些和语言规范不相符的使用限制。表现为：

* 不支持J2SE 5.0中的新语法（包括枚举和泛型）。注解只在Javassist的低级API中支持。请参见 javassist.bytecode.annotation包。
* 除了一维数组，数组的初始化和逗号表达式都不支持。
* 内部类和匿名类不支持。
* continue和break 语句不支持。
* 编译器不能正确实现Java方法中的调度算法，这是由于编译器不能明确这些方法是否方法名相同但参数列表不同。

例如，

class A {}

class B extends A {}

class C extends B {}

class X {

void foo(A a) { .. }

void foo(B b) { .. }

}

x代表X的实例，当编译x.foo(new C())，编译器可能调用 foo(A)，也可能调用foo((B)new C())。

* 建议使用井号（# ）分隔类名和静态方法名或成员变量名。比如，常规的方式，

javassist.CtClass.intType.getName()

调用getName()方法获取javassist.CtClass的静态成员intType。在Javassist中，如下的方式更为推荐：

javassist.CtClass#intType.getName()

这样编译器能更快的解析表达式。

## 5. 字节码操作API

Javassist还提供了直接修改类文件的低级API。要使用这些API，你需要了解Java字节码和类文件格式，因为这些API允许你进行各种各样的类文件修改。

### 5.1 获取ClassFile对象

javassist.bytecode.ClassFile对象代表类文件。通过CtClass的getClassFile() 方法可以获取这个对象。

你也可以直接从类文件构造一个javassist.bytecode.ClassFile对象。比如，

BufferedInputStream fin

= new BufferedInputStream(new FileInputStream("Point.class"));

ClassFile cf = new ClassFile(new DataInputStream(fin));

上述代码从Point.class文件构造出ClassFile对象。

ClassFile对象也可以通过write()方法，用一个给定的输出流写入到类文件中。

### 5.2 添加和删除成员

ClassFile提供了addField()和addMethod() 方法用于添加成员和方法（字节码操作中构造函数被认为是一个方法）。另外，还提供了addAttribute()方法用于添加属性。

请注意FieldInfo, MethodInfo和AttributeInfo 对象都和ConstPool（常量池）对象有关联。因此，ClassFile对象以及其中的FieldInfo，MethodInfo等等对象必须共享同 一个ConstPool对象。也就是说，不同ClassFile对象中的FieldInfo，MethodInfo等等对象是不共享的。

要删除ClassFile对象中的一个成员或方法，必须首先获取这个类中的成员或方法集合。这可以通过getFields()和getMethods()来获取。通过调用集合的remove()方法来删除成员或方法。属性的删除也类似，通过调用FieldInfo或MethodInfo中的getAttributes()方法获取属性列表并删除之。

### 5.3 遍历方法体

CodeIterator能非常方便的用来遍历方法体的执行。要获取这个对象，代码如下：

ClassFile cf = ... ;

MethodInfo minfo = cf.getMethod("move"); // we assume move is not overloaded.

CodeAttribute ca = minfo.getCodeAttribute();

CodeIterator i = ca.iterator();

CodeIterator对象使你可以知道一个方法从头至尾的执行情况。下面是CodeIterator声明的方法：

* void begin()  
  移动到第一天指令
* void move(int index)  
  移动指令到给定的索引
* boolean hasNext()  
  如果还有后续指令，返回true
* int next()  
  获取下一条指令（注意并不返回下一条指令的操作码）
* int byteAt(int index)  
  返回索引处的无符号8位值
* int u16bitAt(int index)  
  返回索引处的无符号16位值
* int write(byte[] code, int index)  
  将字节数组写入索引处
* void insert(int index, byte[] code)  
  在索引处插入字节数组。位移量等等参数会自动调整。

下面代码打印一个方法体中的所有指令调用：

CodeIterator ci = ... ;

while (ci.hasNext()) {

int index = ci.next();

int op = ci.byteAt(index);

System.out.println(Mnemonic.OPCODE[op]);

}

### 5.4 生成字节码序列

Bytecode对象代表字节码序列，这是个不断增长的字节码数组。下面是个例子：

ConstPool cp = ...; // constant pool table

Bytecode b = new Bytecode(cp, 1, 0);

b.addIconst(3);

b.addReturn(CtClass.intType);

CodeAttribute ca = b.toCodeAttribute();

这会生成代表如下序列的代码：

iconst\_3

ireturn

你也可以通过Bytecode的get()方法获取字节码序列。获取的字节码数组可被插入到其他代码中。

Bytecode除了提供很多方法来给序列加入指定的指令，还提供了添加8位操作码的addOpcode() 方法以及添加索引的addIndex()方法。每个操作码的8位值都定义在Opcode接口中。

除非流程控制没有分支，否则用于添加特别指令的addOpcode()和其他方法都会自动维护堆栈的深度。这个值可通过Bytecode对象的getMaxStack()方法获取。这也反射到通过Bytecode对象构造出的CodeAttribute对象。要重新计算一个方法体的最大堆栈深度，请使用CodeAttribute的computeMaxStack()方法。

### 5.5 注解 (Meta 标签)

保存在类文件中的注解其实是运行时不可见（或可见）的注解属性。这些属性可通过ClassFile, MethodInfo, 或 FieldInfo 的getAttribute(AnnotationsAttribute.invisibleTag)方法获取。更多详情，请参见javassist.bytecode.annotation包和 javassist.bytecode.AnnotationsAttribute类。

Javassist还允许你通过高级API访问注解。你可以通过CtClass或CtBehavior的 getAnnotations() 方法访问。

## 6. 泛型

Javassist中的低级API完全支持Java5中的泛型，但高级API（比如CtClass）却并不直接支持。不过，这对于字节码转换不是个严重问题。

Java泛型实现使用的是擦除技术。编译后，所有参数类型都会被丢弃掉。举例说明，假定你的代码定义了一个泛型Vector<String>：

Vector<String> v = new Vector<String&gt();

:

String s = v.get(0);

编译后的字节码等同于如下代码：

Vector v = new Vector();

:

String s = (String)v.get(0);

因此当你要写个字节码转换器，只需要丢去掉所有的参数泛型。比如，如果你有这么一个类：

public class Wrapper<T> {

T value;

public Wrapper(T t) { value = t; }

}

当你想给 Wrapper<T>加个接口Getter<T> ：

public interface Getter<T> {

T get();

}

那么实际加给Wrapper的接口和方法就这么简单：

public Object get() { return value; }