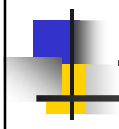


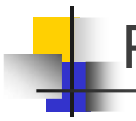
Javassist

Load-Time Structural Reflection in Java

Stefan Neusatz Guilhen

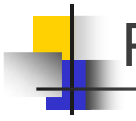
Reflexão em Java

- Java provê uma API para reflexão, que é quase toda voltada para introspecção.
- As possibilidades para se alterar o comportamento de um programa são muito limitadas.
- Diversas extensões foram propostas para lidar com as limitações da API de reflexão.



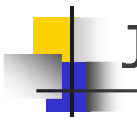
Reflexão comportamental

- A maioria das extensões propostas ativam a reflexão comportamental, que é a habilidade de se interceptar certas operações e modificar se comportamento.
- Apenas permitem alterações de tipos específicos de operações, como invocação de métodos e acesso a campos.
- Se baseiam na técnica de inserção de ganchos (*hook-insertion*) e no modelo de meta-objetos.



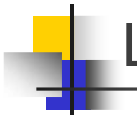
Reflexão estrutural

- Esse tipo de reflexão permite alterar as estruturas de dados de um programa, como classes e métodos.
- Javassist é uma biblioteca que habilita esse tipo de reflexão em java.
- Implementa reflexão estrutural através de mudanças no bytecode em tempo de compilação ou de carga.
- Não permite a reflexão depois que a classe foi carregada na JVM.



Javassist - Objetivos

- O Javassist atende a três objetivos primários:
  - Fornecer uma abstração de código fonte para os programadores. O Javassist foi desenvolvido de modo que os programadores não precisem ter conhecimento dos bytecodes Java.
  - Executar reflexão estrutural da maneira mais eficiente possível.
  - Ajudar os programas a realizar reflexão estrutural de maneira segura com relação aos tipos.



Lendo Bytecodes

- `javassist.CtClass` (compile-time class) é uma representação abstrata de um arquivo `class`.
- `javassist.ClassPool` é um contâiner de objetos `CtClass`. Ele lê arquivos `class` para construir instâncias de `CtClass` e mantém as instâncias construídas até que elas sejam escritas em um arquivo ou um output stream.
- O `ClassPool` mantém uma relação um-para-um entre os arquivos `class` e os respectivos objetos `CtClass`.

Lendo Bytecodes

- Um exemplo inicial:

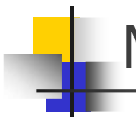
```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.get(“test.Rectangle”);
cc.setSuperclass(pool.get(“test.Point”));
pool.writeFile(“test.Rectangle”);
```

- `ClassPool.getDefault()` é um método de fábrica singleton, fornecido por conveniência.
- O `ClassPool` retornado usa como search path o `classpath` da JVM que está rodando.

Lendo Bytecodes

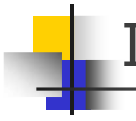
- Estendendo o search path de um `ClassPool`:

```
ClassPool pool = Classpool.getDefault();
pool.insertClassPath(“/usr/local/javalib”);

ClassPath path = new URLClassPath(“www.foo.com”,
       80, “/java”, “com.foo”);
pool.insertClassPath(path);

byte[] b = um array;
String name = nome da classe;
pool.insertClassPath(new ByteArrayClassPath(
       name, b));
```



Modificando uma classe

- A maneira usual de se modificar uma classe em tempo de carga (*load time*) é como segue:
  - Obter um objeto `CtClass` correspondente à classe chamando o método `ClassPool.get()`;
  - Modificar a classe;
  - Chamar `ClassPool.write()` ou `ClassPool.writeFile()`;
- Pode-se também definir um *event-listener* que é notificado quando a classe é carregada na JVM. Um class loader trabalhando com Javassist deve chamar o método `ClassPool.write()` para obter uma classe. O *listener* é notificado quando este método é chamado.

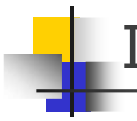


Interface Translator

- Toda classe que for um *event-listener* deve implementá-la.

```
public interface Translator {
   public void start(ClassPool pool) throws
       NotFoundException, CannotCompileException;

   public void onWrite(ClassPool pool, String classname)
      throws NotFoundException, CannotCompileException;
}
```

- Para registrar um *listener* em um `ClassPool`, ele deve ser passado para o construtor do `ClassPool`.
- Apenas um único *listener* pode ser registrado a um `ClassPool`.

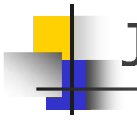


Interface Translator

- Se mais de um *listener* for necessário, múltiplos `ClassPool`s devem ser encadeados. Exemplo:

```
Translator t1 = new MyTranslator();
ClassPool c1 = new ClassPool(t1);
Translator t2 = new MyAnotherTranslator();
ClassPool c2 = new ClassPool(c1, t2);
```

- Se um class loader chamar `write()` em `c2`, a classe especificada será primeiro modificada por `t1` e depois por `t2`.



Javassist Class Loader

- Javassist fornece o class loader `javassist.Loader`, que usa um `ClassPool` para ler as classes.

```
Import javassist.*

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

       ClassPool pool = ClassPool.getDefault();
       Loader cl = new Loader(pool);
       CtClass ct = pool.get("Rectangle");
       ct.setSuperclass(pool.get("Point"));

       Class c = cl.loadClass("Rectangle");
       Object rect = c.newInstance();
    }
}
```

Javassist Class Loader

- Usando os *event-listeners* para modificar as classes de uma aplicação (MyApp):

```
Import javassist.*

public class Main {
    public static void main(String[] args) throws
       Throwable {
       Translator t = new MyTranslator();
       ClassPool pool = new ClassPool(t);
       Loader cl = new Loader(pool);
       cl.run(“MyApp”, args);
    }
}
```

Javassist Class Loader

- `javassist.Loader` busca pelas classes em uma ordem diferente de `java.lang.ClassLoader`:
  - `ClassLoader` primeiro delega a tarefa ao `ClassLoader` ancestral e só carrega a classe se o `ClassLoader` ancestral não conseguir fazê-lo.
  - `javassit.Loader` tenta inicialmente carregar a classe antes de delegar.
- `javassist.Loader` só delega nos seguintes casos:
  - As classes não forem encontradas usando o `ClassPool`;
  - As classes foram definidas para serem carregadas pelo class loader ancestral através do método `delegateLoadingOf`()

Construindo um Class Loader

- Os métodos e construtores dos objetos criados por um class loader podem referenciar outras classes. Para determinar as classes referenciadas, a JVM chama o método `loadClass()` do class loader que originalmente criou a classe.
- Esse método executa as seguintes operações:
  - Chama `findLoadedClass(String)` para checar se a classe já foi carregada.
  - Chama o método `loadClass()` do class loader ancestral.
  - Chama o método `findClass(String)` para encontrar a classe.

Construindo um Class Loader

```
Import javassist.*

public class SimpleLoader extends ClassLoader {
  public static void main(String[] args) throws Throwable{
    SimpleLoader s = new SimpleLoader();
    Class c = s.loadClass("MyApp");
    c.getDeclaredMethod("main", new Class[]
{String[].class}).invoke(null, new Object[] {args});
  }

  private ClassPool pool;

  public SimpleLoader() throws notFoundException {
    pool = ClassPool.getDefault();
    pool.insertClassPath("./class"); //MyApp está aqui
  }
  .
  .
```

Construindo um Class Loader

```
protected Class findClass(String name) throws
ClassNotFoundException {
  try{
     CtClass cc = pool.get(name);
     // CtClass é modificado aqui
     byte[] b = pool.write(name);
     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();
  }
 }
}
```

Introspecção e Customização

- `CtClass` fornece métodos para introspecção, compatíveis com a API de reflexão de Java (`getName()`, `getSuperclass()`, etc).
- Também fornece métodos para modificar a definição de uma classe. Permite a adição de campos, contrutores e métodos.
- Métodos são representados por `CtMethod`, que provê diversos métodos para alterar a definição do mesmo.
- De maneira análoga, os campos são representados por `CtField` e construtores por `CtConstructor`.
- Javassist não permite a remoção de métodos ou campos, nem alterar o número de parâmetros.

Introspecção e Customização

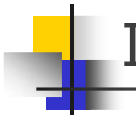
- `CtMethod` e `CtConstructor` fornecem os métodos `insertBefore()`, `insertAfter()` e `addCatch()`. São usados para inserir fragmentos de código no corpo do método ou construtor.
- Usuários podem especificar estes fragmentos usando código-fonte. Javassist provê um compilador Java simplificado para transformar o código nos bytecodes correspondentes.
- Esses métodos recebem uma `String` representando um comando ou um bloco (conjunto de comandos encapsulados por { }).

Introspecção e Customização

- O comando ou o bloco adicionados podem se referenciar aos campos e métodos de classe, mas **não podem** se referenciar a variáveis locais declaradas no corpo do método.
- O compilador incluído com o Javassist fornece algumas extensões interessantes, como o conjunto de identificadores especiais (iniciados por $) que podem ser utilizados pelo bloco ou comando que está sendo incluído.

Identificadores especiais

`$0, $1, $2,...`	Parâmetros.
`$args`	Um array dos parâmetros cujo tipo é `Object`.
`$$`	Abreviação para todos os parâmetros.
`$cflow(...)`	Variável `cflow`.
`$r`	Tipo do resultado, usado em casting.
`$w`	Tipo wrapper, usado em casting.
`$_`	O valor do resultado.
`$sig`	Um array de `Class` representando os tipos dos parâmetros.
`$type`	Um objeto `Class` representando tipo de retorno.
`$class`	Um `Class` representando a classe editada.



Identificadores especiais

- `$0, $1, $2, ...`
  - `$1` representa o primeiro parâmetro fornecido ao método ou construtor. `$2` representa o segundo e assim por diante.
  - `$0` é equivalente ao `this`. Não disponível se o método é estático.

  Exemplo:

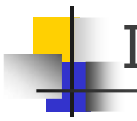
```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.get("Point");
CtMethod cm = cc.getDeclaredMethod("move");
cm.insertBefore("{System.out.println($1);
                  System.out.println($2); }");
cc.writeFile();
```

Identificadores especiais

- `$args`
  - Representa um array de Object contendo todos os parâmetros. Se o tipo de um parâmetro for primitivo, o valor é convertido no wrapper correspondente.
- `$$`
  - É uma abreviação da lista de todos os parâmetros separados por vírgulas. Por exemplo, se o número de parâmetros do método `move()` é três, `$$` é equivalente a `$1, $2, $3`.
  - Se `move()` não recebe parâmetros, então `move($$)` é equivalente a `move()`.
- `$cflow`
  - Retorna a profundidade da chamada recursiva.

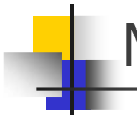
Identificadores especiais

- `$r`
  - Representa o tipo do retorno do método. Deve ser usado como o tipo do cast em uma expressão de casting. Exemplo: `$_ = ($r) foo();`
- `$w`
  - Representa um tipo wrapper. Deve ser usado como o tipo do cast em uma expressão de casting, convertendo um tipo primitivo em um tipo wrapper. Ex: `Integer i = ($w) 5;`
- `$_`
  - Disponível para o método `insertAfter()`. Representa o valor resultante do método.
  - O código adicionado pode ser executado mesmo que uma exceção seja lançada, se o parâmetro `asFinally` for `true`.



Identificadores especiais

- `$sig`
  - O valor de `$sig` é um array de `java.lang.Class` representando os tipos formais dos parâmetros.
- `$type`
  - O valor de `$type` é uma `java.lang.Class` representando o tipo formal de retorno. Disponível apenas no método `inserAfter()`.
- `$class`
  - O valor de `$class` é um `java.lang.Class` que representa a classe na qual o método editado é declarado.



Método addCatch()

- `addCatch` insere um fragmento de código que é executado quando o corpo do método lança uma exceção. O código inserido deve terminar com `throw` ou `return`.

```
CtMethod cm = ...;
CtClass etype = ClassPool.getDefault().get(
       “java.io.IOException”);
cm.addCatch(“{System.out.println($e); throw $e; }”,
       etype);
```

```
try{
    //o corpo original
} catch (IOException e) {
    System.out.println(e); throw e;
}
```

Definindo uma nova classe

- Para definir uma nova classe, `makeClass()` deve ser invocado em um `ClassPool`:

```
ClassPool pool = Classpool.getDefault();
CtClass cc = pool.makeClass(“Point”);
CtClass cc1 = pool.get(“Point”);

cc.setName(“Pair”);
CtClass cc2 = pool.get(“Pair”);
```

Modificando um método

- `CtMethod` e `CtConstructor` fornece um método `setBody()` para substituir o corpo todo do método.
- Os identificadores especiais podem ser usados no código fornecido ao método `setBody()`, com exceção do identificador `$_`.
- Javassist também permite modificar apenas uma expressão contida no corpo do método. A classe `javassist.expr.ExprEditor` faz a substituição da expressão.
- Usuários podem definir uma subclasse de `ExprEditor` para definir como a expressão é modificada.

Usando ExprEditor

```
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($$);}");
           }
       });
```

- O método `instrument()` procura por expressões no corpo do método. Quando ele encontra uma, ele chama o método `edit()` do `ExprEditor`. Depois, o método `replace` substitui a expressão encontrada.

Usando ExprEditor

- `javassist.expr.MethodCall`
  - Representa uma chamada de método.
- `javassist.expr.FieldAccess`
  - Representa um acesso a um campo.
- `javassist.expr.NewExpr`
  - Representa a criação de um objeto.
- `javassist.expr.Instanceof`
  - Representa uma expressão `instanceof`.
- `javassist.expr.Cast`
  - Representa um casting explícito.
- `javassist.expr.Handler`
  - Representa uma cláusula `catch`.

Adicionando métodos

- `CtNewMethod` e `CtNewConstructor` fornecem diversos métodos de fábrica para criar objetos `CtMethod` e `CtConstructor`. Exemplo: `make()`.

```
CtClass point = ClassPool.getDefault().get(“Point”);
CtMethod cm = CtNewMethod.make(
              “public int xmove(int dx) { x += dx;}”
              , point);
point.addMethod(cm);
```

- Pode-se também adicionar um método abstrato à classe e depois fornecer seu corpo.

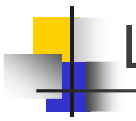
```
CtClass cc = ...;
CtMethod cm = new CtMethod(CtClass.intType, “move”,
       new CtClass[] { CtClass.intType }, cc);
cc.addMethod(cm);
cm.setBody(“{ x += $1; }”);
cc.setModifiers(cc.getModifiers()&~Modifier.ABSTRACT
```

Adicionando campos.

- Pode-se facilmente adicionar um campo a uma classe:

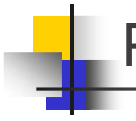
```
CtClass point = ClassPool.getDefault().get(“Point”);
CtField f = new CtField(CtClass.intType,
                              “z”, point);
point.addField(f, “0”);
```

- O segundo parâmetro fornecido à `addField()` é o valor com o qual o campo deve ser inicializado no construtor e é opcional.



Limitações

- A notação `.class` não é suportada;
- `switch`, `syncronized`, `continue` e `break` ainda não são suportados;
- Cláusula `finally` seguindo cláusulas `try` e `catch` não é suportada.
- O compilador do Javassist implementa muito menos checagens do código do que requerido pela especificação da linguagem Java. Resultado: resultados surpreendentes podem aparecer se o programador não for cuidadoso.



Referências

- Página do projeto no JBossGroup:

http://www.jboss.org/index.html?module=html&op=userdisplay&id=developers/projects/javassist

- Página original do projeto, com links para baixar o biblioteca e para o tutorial e a API:

http://www.csg.is.titech.ac.jp/~chiba/javassist/