

Metaprogramming

Oliver Kaiser

Source Talk Tage 2009
Crossgate Technologies AG

30. September 2009

Überblick

- 1 Metaprogramming allgemein
- 2 Java APIs
- 3 Beispiele

Ganz kurz

Wikipedia

Metaprogramming is the writing of computer programs that write or manipulate other programs (or themselves) as their data, or that do part of the work at compile time that would otherwise be done at runtime.

- Compiler (gcc, javac, yacc)
- Macros (cpp, m4, TeX, Lisp)
- Selbst-modifizierender Code

Kategorien

Begriffe

Ein Metaprogramm operiert auf einem Objektprogramm.
Metasprache beschreibt Objektsprache.

Programme sind Daten!

Unterteilung in:

- Erzeugen bzw. manipulieren eines Objektprogrammes (*generator*)
- Analyse eines Objektprogrammes (*analyzer*)

Merkmale

- compile-time oder runtime
- homogen oder heterogen (Metasprache = Objektsprache)
- Repräsentation: String, Graphen (AST, parse tree),
Algebraisch (Haskell, Standard ML) oder Quasi-Quote (Lisp)
- Objektprogramm ist wiederum ein Metaprogramm:
multi-staged
- Typisiert (Typen-fehler im Objektprogramm werden zur
Meta-prog Compile-zeit erkannt; type-correct MP \Rightarrow
type-correct OP)

Vorhandene Bordmittel

Aspekten der Sprache sind Typen zugeordnet (Klasse, Methode, Feld).

- Reflection API (insbesondere `Proxy`) ([java.lang.reflect](#))

Manipulation von Klassendefinitionen.

- Instrumentation API ([java.lang.instrument](#))
- Debugger API (hotswap mit `VirtualMachine.redefineClasses`)

Nützliche APIs

- Compiler API ([javax.tools.JavaCompiler](#))
- Eigene [ClassLoader](#)
- [Annotationen](#)

Reflection aka. introspection

Beschreibung von Java-Ausdrücken in Java (homogen).

Ausgehend von einer `java.lang.Class`-Instanz können die deklarierte Felder bzw. Methoden ermittelt sowie Methoden (einschliesslich Konstruktoren) aufgerufen werden.

Analoge Typen existieren z.B. in der Bytecode-Bibliotheken [BCEL](#) und [Javassist](#).

Der Begriff *Intercession* bezeichnet die Fähigkeit einer reflektiven Sprache das Laufzeitverhalten eines Programmes zu beeinflussen.

Die Möglichkeiten in Java sind recht beschränkt:

`Method.invoke` sowie Proxy-Klassen.

Reflection performance

Typ	setup	10^5 calls	10^6 calls	10^8 calls
Reflection	-	37 ms	104 ms	5208 ms
Javassist	68 ms	11 ms	14 ms	345 ms
BCEL	98 ms	10 ms	14 ms	347 ms

Java 1.6.0_10, 64-bit Server VM

Quelle der Programme:

<http://www.ibm.com/developerworks/java/library/j-dyn0610/>

Dynamische Proxy Klassen

Erzeugt zur Laufzeit eine Proxy-Klassendefinition. Instanzen dieser Klasse implementieren eine Menge von Interfaces; Methodenaufrufe auf Proxy-Instanzen werden per Reflection zur eigentlichen Implementation weitergeleitet.

Anwendungen

- Decorator: Logging, ACL
(Beispiel: TracingHandler)
- Verhindern von Casts die 'unsicherem' Wissen des Programmieres beruhen; äquivalent zu einem Typ-einschränkenden Adapter.
- Lazy instantiation (z.B. Tapestry5 IoC)

Beispiel: casts verhindern

```
interface Foo { ... }  
class Bar {  
    void baz(Foo foo) {  
        FooImpl impl = (FooImpl)foo;  
        impl.somethingNotInInterface();  
    }  
}
```

ClassCastException bei Verwendung einer Proxy-Instanz.

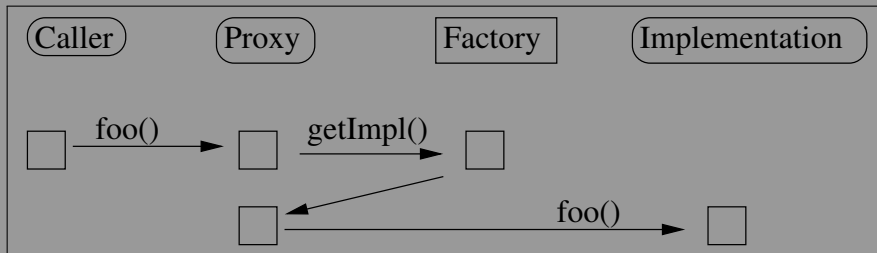
Beispiel: Lazy instantiation I

```
interface SomeService { ... }  
class SomeClass {  
    @InjectService("SomeService");  
    private SomeService proxy;  
  
    void foo() {  
        proxy.bar();  
    }  
}
```

Die Implementation von `SomeService` muss erst beim ersten Methodenaufruf instanziiert werden, nicht zur Konstruktor-Zeit von `SomeClass`.

Tatsächlich basieren t5 Proxies nicht auf Reflektion; entsprechende Klassen werden zur Laufzeit mit `javassist` erzeugt.

Beispiel: Lazy instantiation II



Vorteile:

- Performance falls SomeService 'teuer' ist und nur in einigen Methoden von SomeClass verwendet wird.
- Zyklische Abhängigkeiten von Services einfacher zu verhindern

Compiler API & Pluggable Annotation Processing

Aufruf des Compilers aus einem Java Programm bzw.
Beeinflussung von *javac*; es ergeben sich viele Möglichkeiten:

- *eval* Funktion zur Laufzeit [openjdk.dev.java.net](https://openjdk.dev/java.net)
- Code-Analyse (coding style, erzeugen von Dokumentation)
- Manipulation des AST

Als Einführung erscheint **The Hacker's Guide to Javac** empfehlenswert.

ACL & Logging mit Javassist

Der Code

```
@RequireFeature (Features.bpe_edit)
@Auditing
public long updateOrSaveBpe (BPE bpe) { // ... }
```

Ausgabe im Log

```
... Denying method getBpeCountList \
  for session XXX; lacks feature modules.lascaux.bpe
... updateOrSaveBpe bpe: BPE \
  [id: -1, comment: Lagernummer]
```

Allgemein: Einfügen von Codefragmenten am Anfang bzw. Ende von Methodenrumpfen.

Sinnvolle Anwendung?

```
public long updateOrSaveBpe(BPE bpe) {  
    checkAllowed(METHOD_NAME, FEATURE_NAME);  
    // eigentlicher Rumpf  
    auditMethod(METHOD_NAME);  
    return xxx;  
}
```

- Zugriff auf den Methodennamen sowie Namen & Typen der Argumente; zur Compile-zeit bestimmt (keine Laufzeit-kosten, kein manuelles Anpassen bei *refactoring*)
- Keine Veränderung der Klassensignatur, unkritisch für aufrufenden Code
- Möglichkeit der Mehrfachnutzung (hier: Erstellen der Dokumentation; welche Benutzerrechte sind für den Aufruf einer Funktion notwendig)

Laufzeit-magie

Austauschen von Klassen in einer laufenden JVM ist z.B. mit einem *agent* möglich.

Beispiel: hotpatching

Einschränkungen:

- Klassensignatur darf sich nicht ändern
- Verhalten von existierenden Instanzen?
- Klassen entladen: alle Referenzen & Classloader müssen *garbage-collected* werden

Methodensignatur im aufrufendem Code

Gegeben seien zwei Versionen einer API sowie ein binäres Programm, das gegen die alte Version kompiliert wurde.

```
public class API {  
    public void foo(String s) { ... }  
}
```

```
public class API {  
    public Object foo(String s) { ... }  
}
```

Siehe: [Quelle](#)

ASM MethodAdapter

```
MethodVisitor mv;  
  
void visitMethodInsn(int opcode, String owner,  
    String name, String desc) {  
    if ("api/API".equals(owner) &&  
        "foo".equals(name) &&  
        "(Ljava/lang/String;)V".equals(desc)) {  
        mv.visitMethodInsn(opcode, owner, name,  
            "(Ljava/lang/String;)Ljava/lang/Object;");  
        mv.visitInsn(OpCodes.POP);  
    } else {  
        mv.visitMethodInsn(opcode, owner, name, desc);  
    }  
}
```

Multiple-inheritance

Erzeugen des Effektes durch Delegation zu einem enthaltenen Feld;
entsprechende Modifikationen im Bytecode sind denkbar.

Beispiel: fake multiple inheritance

Problem: Codevervollständigung und Highlighting in IDEs.

Denkbarer Ansatz

- Interfaces für die jeweiligen Klassen; kombiniertes Interface
- im aufrufenden Code das kombinierte Interface zur Variablendeklaration verwenden
- Bytecode: MI Klasse um Interfaces erweitern
- Instanziierung im aufrufenden Code durch *dependency injection*

Geht das nicht einfacher?

<http://code.google.com/p/java-mixins/>

```
@MixinType(implementation = FooImpl.class)
public interface Foo {
    public void fooMethod();
}

@MixinBase
public abstract class Bar implements Foo { ... }
public static void main(String[] args) {
    Bar bar = MixinSupport.getSingleton()
        .create(Bar.class);
    bar.fooMethod();
}
```

Eine technisch noch eindrucksvollere Lösung ist: projectlombok.org.