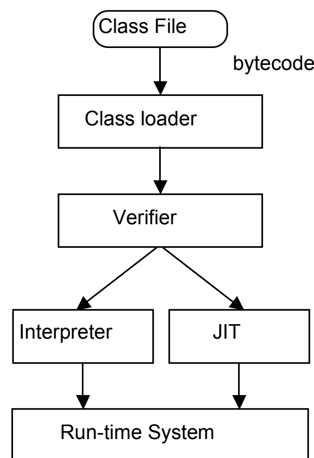
Java Virtual Machine (JVM)

Un file .class è ottenuto compilando il codice sorgente Java. Esso contiene il bytecode, ovvero la sequenza di istruzioni per la JVM, ed anche molte informazioni simboliche. Il bytecode è il linguaggio macchina della JVM.

Una parte del file .class è costituita dal constant pool, che conserva le costanti usate, i nomi dei metodi, la loro signature, i riferimenti a classi e metodi usati da questa classe, i nomi della superclasse e delle interfacce. Può costituire metà della dimensione del file .class.



Ogni JVM ha un bootstrap class loader che carica le classi delle librerie Java. Le classi caricate con questo sono fidate quindi il Verifier evita alcuni dei controlli normalmente effettuati.

Un extension class loader è usato per caricare classi che estendono le API Java standard. Un system class loader è usato per caricare le altre classi (anche le classi delle applicazioni).

JVM

Le applicazioni possono definire propri class loader, come sottoclassi di `java.lang.ClassLoader`

Ogni class loader mantiene una lista delle classi caricate.

Il nuovo class loader permette di rilevare il caricamento di una classe.

Possiamo quindi riscrivere parti del file .class, al momento del caricamento, per modificarne il comportamento.

Il run-time system eseguirà la versione modificata della classe.

Riflessione e Linguaggi

Con behavioural reflection intendiamo che i meccanismi riflessivi sono usati per cambiare il comportamento di un'applicazione (modello metaoggetto). Con structural reflection i meccanismi riflessivi forniti sono usati per cambiare la struttura di un'applicazione, cambiandone le classi per es. aggiungendo metodi, campi, etc. (modello metaclasse). Variandone la struttura, viene variato anche il comportamento dell'applicazione.

Per ottenere trasparenza persino nell'implementazione, un linguaggio riflessivo dovrebbe:

- far rimanere il codice sorgente del livello base invariato anche quando aggiungiamo il metalivello
- nascondere al programmatore del metalivello i dettagli della connessione tra livello base e meta
- fornire sia behavioural che structural reflection.

Per il linguaggio C++ esiste OpenC++

- alcuni costrutti devono essere iniettati nel codice delle classi
- è parecchio difficile da usare, ma permette la ridefinizione di qualsiasi costrutto C++ (si può cambiare la semantica dei costrutti)
- il codice OpenC++ viene trasformato in codice C++ da un precompilatore
- richiede sia l'implementazione di meta-oggetti, sia di un programma che guida il precompilatore nella connessione tra meta-oggetti ed oggetti.

Linguaggi Riflessivi

Per il linguaggio Java esistono:

- OpenJava: ha caratteristiche simili a OpenC++.
- Dalang: supporta il modello metaoggetto; l'intercettazione di metodi avviene tramite un oggetto wrapper, che è inserito nell'applicazione modificandone il codice sorgente con un tool; l'associazione oggetto-metaoggetto è specificata in un file di configurazione.
- MetaXa: supporta il modello metaoggetto; l'intercettazione viene fatta modificando la JVM.
- Kava: è l'evoluzione di Dalang; permette behavioural reflection modificando il bytecode dell'applicazione; l'associazione oggetto-metaoggetto è specificata in un file di configurazione.
- **Javassist**: permette sia structural che behavioural reflection; i metaoggetti sono associati agli oggetti tramite una versione modificata del class loader che cambia parti selezionate delle classi dell'applicazione; le modifiche possono essere registrate su file .class. Solo il bytecode delle classi è necessario per renderle riflessive.

La libreria di classi che costituiscono Javassist è disponibile su:

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

Bisogna settare il CLASSPATH della JVM opportunamente, es.: setenv CLASSPATH .:/usr/bin/javassist2.6/javassist.jar

Javassist - 1

Javassist permette di modificare il bytecode di una classe per aggiungere o modificare campi e metodi.

La classe `javassist.ClassPool` rende le classi caricate dalla JVM dei dati sui quali possono essere fatte operazioni attraverso le API di Javassist.

Le classi caricate sono istanze di `javassist.CtClass`

Su `CtClass` (compile-time class) è possibile invocare dei metodi sia per ispezionare campi e metodi dell'istanza di `CtClass` (come per `Class`) che di aggiungere nuovi campi, metodi e costruttori, e di alterare il nome della classe, della superclasse e delle interfacce. Non fornisce metodi per la cancellazione di campi, metodi e costruttori.

Campi, metodi e costruttori sono rappresentati da istanze di `javassist.CtField`, `javassist.CtMethod` e `javassist.CtConstructor`, rispettivamente.

Javassist consente di alterare il codice di un metodo, per esempio aggiungendo all'inizio o alla fine del nuovo codice, fornito direttamente come codice sorgente Java.

Piccole variazioni sono introdotte nella sintassi Java del codice fornito per rappresentare i parametri dei metodi.

Le aggiunte al codice di una classe vengono effettuate nel momento in cui classe è caricata dalla JVM (a load-time).

Javassist - 2

Un programma che legge una classe, modifica un suo metodo e scrive su disco la classe così modificata.

```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.get("Cerchio");
CtMethod m = cc.getDeclaredMethod("area");
m.insertBefore(" { System.out.println($1); }");
cc.writeFile();
```

Le precedenti linee di codice eseguono i seguenti passi, rispettivamente:

1. Ottiene una istanza di `ClassPool`, così da avere una sorta di contenitore per le classi.
2. Ottiene una istanza di `CtClass` che si riferisce alla classe che vogliamo modificare, di cui passiamo il nome (`Cerchio`), attraverso il metodo `get()` di `ClassPool`
3. Ottiene il riferimento ad un metodo, di cui indichiamo il nome (`area`), con `getDeclaredMethod()` di `CtClass`
4. Inserisce il codice, che passiamo come sorgente Java, attraverso il metodo `insertBefore()`. Sarà Javassist a compilare il codice prima di inserirlo nel metodo.
5. Scrive su disco il file .class appena modificato, con `writefile()`, dove `$1` rappresenta l'argomento (il primo o l'unico) passato al metodo `area()`. Un potenziale secondo argomento sarebbe indicato con `$2`.

Javassist - 3

La classe `CtClass` include i metodi:

`addField(CtField f)`

serve ad inserire un nuovo campo, specificato attraverso una istanza di `CtField`.

`addMethod(CtMethod m)`

serve ad inserire un nuovo metodo, specificato attraverso una istanza di `CtMethod`.

`setName(String name)`

serve a cambiare il nome della classe.

`setSuperclass(CtClass c)`

serve a cambiare la superclasse.

`toClass()`

converte la classe in una istanza di `java.lang.Class`.

`writeFile()`

scrive la classe nella directory corrente.

La creazione di una classe vuota, su cui possono essere aggiunti campi e metodi:

```
CtClass c = new CtClass();
```


Javassist - 4

La classe `CtMethod` include i metodi:

`insertAfter(String s)`

serve ad inserire un frammento di codice, descritto da una stringa, alla fine del corpo del metodo.

`insertBefore(String s)`

serve ad inserire un frammento di codice all'inizio del corpo del metodo.

`setBody(String s)`

serve a definire il corpo del metodo.

`setName(String name)`

serve a cambiare il nome del metodo.

`instrument(ExprEditor ee)`

serve a modificare il corpo di un metodo. Ogni volta che una invocazione o una `new` sono trovati all'interno del metodo, viene invocato `edit()` sull'istanza di `ExprEditor` passata ad `instrument()`.

`edit()` contiene delle istruzioni che tipicamente variano il codice del metodo su cui `instrument()` è invocato.

(Tali modifiche avvengono a load-time, cioè prima di aver mandato in esecuzione il codice della classe.)

Javassist - 5

La classe `CtField` include i metodi:

`getName(), getType() getDeclaringClass()`

`setName(String name)`

serve a cambiare il nome del campo.

`setType(CtClass c)`

serve a cambiare il tipo del campo.

Le precedenti classi consentono di ottenere structural reflection.

Tramite modifiche al bytecode, è possibile realizzare behavioural reflection.

Per ottenere l'intercettazione dei metodi di una classe, occorre:

- aggiungere alla classe un campo che tiene il riferimento al metaoggetto
- rinominare i metodi della classe
- inserire dei metodi con i vecchi nomi che invocano il metaoggetto

Queste operazioni sono già realizzate da Javassist, con la libreria `javassist.reflect`

Metaoggetti con Javassist

Data un'applicazione composta dalle classi `Gente`, `Persona`, `Studente`, associamo alle classi `Persona` e `Studente` un metaoggetto `VerboseMO` che scrive informazioni sulle operazioni fatte dalle classi.

Occorre una classe che dice al class loader di rendere riflessive le classi dell'applicazione.

```
Loader cl = (Loader) Main.class.getClassLoader();

cl.makeReflective("Studente", "VerboseMO",
                  "javassist.reflect.ClassMetaobject");
```

`makeReflective` viene chiamato per ciascuna associazione classe che deve essere resa riflessiva.

La classe `VerboseMO` deve ereditare la classe `javassist.reflect.Metaobject` ed implementare il costruttore, ed i metodi:

- `trapFieldRead(String n)`
- `trapFieldWrite(String n, Object v)`
- `trapMethodcall(int id, Object[] args)`

Il costruttore di `VerboseMO` viene invocato quando l'oggetto associato deve essere creato. I metodi ricevono il controllo dall'applicazione quando viene fatta una lettura di un campo, una scrittura o un'invocazione di metodo, rispettivamente.

Considerazioni

Quando si usa Javassist bisogna stare attenti alla variazione di prestazioni.

- Un primo decadimento delle prestazioni è dovuto alla modifica del file .class per rendere una classe riflessiva.

  Tuttavia, il file .class può essere modificato una sola volta, facendo modifiche permanenti, anzichè a load-time.

- Un secondo decadimento delle prestazioni è dovuto al fatto che ogni invocazione di metodo di una classe dell'applicazione scatena altre 2 invocazioni. Inoltre, l'esecuzione di `invoke()` di Java è più lenta di una invocazione di metodo diretta.

Riepilogo dei vantaggi offerti dalla riflessione:

- Fornisce il modo di comporre vari aspetti di un'applicazione (funzionalità, sincronizzazione, distribuzione, etc.), mantenendo i codici sorgenti separati.
- La separazione tra vari aspetti di un'applicazione rende più semplice lo sviluppo, il riuso e l'evoluzione dei singoli aspetti. Tali aspetti possono essere resi indipendenti gli uni dagli altri.
- Evoluzione di un'applicazione anche quando il codice sorgente non è disponibile.
- Adattamento di classi di un'applicazione all'ambiente di esecuzione nel momento in cui vengono caricate dalla JVM.

Architetture riflessive sviluppate

Esempi di sistemi riflessivi realizzati dall'Universita' di Catania

- Framework per sincronizzare l'accesso ad oggetti di una applicazione. Gli schemi di sincronizzazione sono indipendenti dall'applicazione e possono essere facilmente riusati.
- Framework per la trasformazione di una qualsiasi applicazione centralizzata in una versione distribuita. La gestione di tutti gli aspetti riguardanti il distribuito (allocazione, localizzazione, load-balancing, fault-tolerance, comunicazione tra host) è non visibile all'applicazione e facilmente riusabile.
- Framework per l'assistenza ad un browser web attraverso un sistema multi-agente che: caratterizza il profilo utente, cattura dati dalle pagine web e gestisce i dati sull'host dell'utente.
- Estensione di un editor con la capacità di evidenziare la sintassi del codice Java.
- Framework per il monitoraggio e la misurazione di applicazioni.
- Adattamento a run-time di applicazioni tramite inserimento di nuovo codice conosciuto solo al momento dell'esecuzione.