Example16

Table of contents

Aspect-Oriented Programming

Reloadable (Class) Aspect

Nel momento della realizzazione di un sistema software, dobbiamo porci il problema di supportare l'*aggiornamento automatico*: permettere al sistema di aggiornare le sue parti scaricando dalla rete evitando di spegnere o arrestare il sistema.
Uno dei motivi per cui JAVA  e' stato realizzato, e' proprio per adempiere a questo compito. La JVM ha la capacita':

- di realizzare software indipendente da hardware;
- di realizzare sistemi capaci di scaricare codice e attivarlo nel modo piu' semplice possibile, proteggendo l'hardware da eventuali malfunzionamenti.

Una classe puo' essere caricata, consentendo la sua eliminazione ogni qual'ora una nuova versione di questa sia presente. Quelle che rendiamo ricaricabili sono le classi. Le classi che preciseremo come *hot swappable* saranno in una cartella e tutto quello che non vi e' dentro, non e' caricabile.

1. creiamo un descrittore di classe, ovvero un oggetto contenuto al suo interno, tutte le informazioni descrittive della classe e l'accesso ai metodi della classe;
2. rendiamo attivo il descrittore, prendendo il codice eseguibile e se serve, compilarlo in codice nativo quando si determina che il suo uso e' frequente e che quindi ne vale la pena (risoluzione della classe).

ClassLoader

ClassLoader in JAVA8 documentation

Il metodo pubblico `loadClass(String name)`, prende il nome completo di una classe e ritorna il suo descrittore di classe: e' il metodo utilizzato dalla JVM quando un metodo ha bisogno di una classe.

Il metodo protetto `loadClass(String name, boolean resolve)` invece ha lo stesso funzionamento, con la differenza di un `boolean` che specifica se *risolvere* la classe indicata o meno:

1. invoca `findLoadedClass(String)` per controllare se la classe e' gia' stata caricata;
2. se non trovato il descrittore di classe associato al nome (classe non caricata), viene usato `loadClass()` sul parent class loader;
3. invoca `findClass(String)` se i class loader in catena, non riescono a trovare la classe specificata.

Per questione di sicurezza e completezza, noi ridefiniamo entrambi i metodi.

`it.unipr.informatica.aspects`

`ReloadableHandler.java`

Da fornire all'utente che utilizza il nostro aspetto, interfaccia che una volta costruita, ha 2 metodi:

- `loadClass()` che carica la classe, la rende pronta per essere utilizzata ma non istanzia oggetti, se la classe e' cambiata verra' caricata la nuova versione;
- `newInstace()` usa il descrittore ottenuto da `loadClass()` per istanziare.

```
public interface ReloadableHandler<T> {
        public T newInstance() throws ClassNotFoundException, IllegalAccessException,
InstantiationException;

        public Class<T> loadClass() throws ClassNotFoundException;
}
```

`ReloadableAspect.java`

Al metodo `newHandler()` passiamo 3 cose:

- `Class<T> reloadableInterface`
  un descrittore di classe da utilizzare per descrivere il tipo di ritorno delle `newInstance()`;

- `String className`
  il nome della classe da caricare effettivamente;

- `String[] classPath`
  array di stringhe che ci dice i percorsi da utilizzare per cercare la classe, un class path in cui andare a cercare le classi ricaricabili.

Gli argomenti vengono controllati e ritornato poi `InnerReloadableHandler`.

```
        // ...
        public static <T> ReloadableHandler<T> newHandler
                (Class<T> reloadableInterface, String className, String[] classPath) {
                if (reloadableInterface == null)
                        throw new IllegalArgumentException
                                ("reloadableInterface == null");

                if (className == null || className.length() == 0)
                        throw new IllegalArgumentException
                                "className == null || className.length() == 0");

                if (classPath == null || classPath.length == 0)
                        throw new IllegalArgumentException
                                ("classpath == null || classpath.length == 0");

                return new InnerReloadableHandler<T>(className, classPath);
        }
        // ...
```

Implementazione locale di `ReloadableHandler`.

- `newInstance()` si limita a chiamare `loadClass()` e `newInstance()` sulla classe caricata;
- `loadClass()` chiama il secondo piu' in basso, metodo privato, che effettivamente verifica se il file esiste, se e' stato modificato ecc.

Lo stato del `ReloadableHandler` e' costituito da:

- il nome della classe restituitoci come argomento;
- l'array di nome di cartelle, il nostro class path;
- il `parentClassLoader` nel caso a un certo punto sia necessario chiedere se caricare o meno il class loader della classe aspetto;
- una mappa che associa a ogni classe ricaricabile, un class loader, cosi' tutte le volte che c'e' bisogno ci caricare una classe verifichiamo se il class loader c'e' o meno.

```
        // ...
        private static final class InnerReloadableHandler<T>
        implements ReloadableHandler<T> {
                private Map<String, InnerClassLoader> classLoaders;
                private ClassLoader parentClassLoader;
                private String[] classPath;
                private String className;

                private InnerReloadableHandler(String className, String[] classPath) {
                        this.className = className;
                        this.classPath = classPath;
                        this.parentClassLoader = getClass().getClassLoader();
                        this.classLoaders = new HashMap<>();
                }

                @Override
                public T newInstance() throws ClassNotFoundException,
                llegalAccessException, InstantiationException {
                        Class<T> clazz = loadClass();
                        return clazz.newInstance();
                }

                @Override
                public Class<T> loadClass() throws ClassNotFoundException {
                        @SuppressWarnings("unchecked")
                        Class<T> clazz = (Class<T>) loadClass(className, false);
                        return clazz;
                }
                // ...
```

`loadClass()` in fondo ha 2 argomenti:

- `String name`
  il nome della classe che vogliamo caricare, che non e' detto sia sempre la classe specificata all'inizio, potrebbe essere una sua dipendenza;
  verifichiamo se la classe specificata e' una che si riesce a trovare nei percorsi con `getClassFile()` e se non lo e' abbiamo 2 opzioni:
- `boolean delegate`
  - e' stato passato il secondo argomento `delegate` a `false`, che significa di non usare il `parentClassLoader` alla `loadClass()`, perche' magari noi vogliamo lanciare un'eccezione anziche' caricare a prescindere
  - e' stato passato `delegate` a `true`, se non trovo il file, posso caricarlo con il `parentClassLoader` e quindi chiedo allo stesso di farlo con `loadClass()`

Se abbiamo associato un class loader a una classe, questo avra' caricato a un certo punto il file della classe e quando l'ha fatto, si e' segnato la data di ultima modifica.
Siccome noi abbiamo trovato il file, vuol dire che possiamo con il metodo `lastModified()` confrontare le date e:

- se la data e' successiva alla data di ultima modifica del class loader, vuol dire che il file e' stato modificato dopo essere stato caricato → gettiamo il class loader caricato e ne costruiamo uno nuovo;
- se le date sono uguali, il file non e' stato modificato e quindi usiamo il class loader per caricarlo.

```
// ...
private Class<?> loadClass(String name, boolean delegate)
throws ClassNotFoundException {
        File file = getClassFile(name);

        if (file == null) {
                if (!delegate)
                        throw new ClassNotFoundException(name);

                return parentClassLoader.loadClass(name);
        }

        InnerClassLoader classLoader = classLoaders.get(name);
        long lastModified = file.lastModified();

        if (classLoader == null) {
                classLoader = new InnerClassLoader
                        (this, name, file, lastModified);
                classLoaders.put(name, classLoader);
        } else if (className.equals(name) &&
                lastModified > classLoader.getLastModified()) {
                classLoaders.clear();
                classLoader = new InnerClassLoader
                        (this, name, file, lastModified);
                classLoaders.put(name, classLoader);
        }

        return classLoader.loadClass(name);
}
// ...
```

Se la `loadClass()` di un class loader associato a una classe, viene invocato con relazione 1-a-1, allora questa verra' caricata; se invocato su un altra classe, motivo ciclo caricamento delle dipendenze per esempio, allora chiamiamo `loadClass()` sull'handler.

Nel caricamento effettivo della classe, prendiamo il file, lo leggiamo, lo mettiamo in memoria come array di byte e chiamiamo su quest'ultimo `defineClass()` che prende e costruisce il descrittore. Se non vogliamo risolvere il descrittore, perche' ci basta che la classe sia solo definita perche' l'argomento e' `false`, semplicemente non chiamiamo `resolveClass()`.

```
// ...
@Override
protected Class<?> loadClass(String name, boolean resolve)
throws ClassNotFoundException {
        if (name == null || name.length() == 0)
                throw new IllegalArgumentException
                        ("name == null || name.length() == 0");

        if (!className.equals(name))
                return handler.loadClass(name, true);
```

```
                        if (clazz == null)
                                return clazz;

                        synchronized (this) {
                                if (!classFile.exists() || !classFile.isFile() ||
                                !classFile.canRead())
                                        throw new ClassNotFoundException(className);

                                try (InputStream inputStream = new FileInputStream(classFile);
                                                BufferedInputStream bufferedInputStream =
                                                        new BufferedInputStream(inputStream);
                                                ByteArrayOutputStream outputStream =
                                                        new ByteArrayOutputStream()) {
                                        System.out.println
                                                ("Loading class " + className + " from " +
classFile);
                                        byte[] buffer = new byte[BUFFER_SIZE];
                                        int read = bufferedInputStream.read(buffer);

                                        while (read >= 0) {
                                                outputStream.write(buffer, 0, read);
                                                read = bufferedInputStream.read(buffer);
                                        }

                                        byte[] bytecode = outputStream.toByteArray();
                                        clazz = defineClass
                                                (className, bytecode, 0, bytecode.length);

                                        if (resolve)
                                                resolveClass(clazz);

                                        System.out.println
                                                ("Loaded class " + className + " from " +
classFile);


                                        return clazz;
                                } catch (Throwable throwable) {
                                        throw new ClassNotFoundException
                                                (className, throwable.getCause());
                                }
                        }
                }
        }
}
```

Example16

`Example16.java`

Definiamo un handler per la classe specificata, non come letterale ma come stringa.
Quello che uscira' una volta caricata la classe, sara' un `Runnable`.
Il terzo argomento e' l'elenco dei percorsi in cui cercare le classi.

Costruiamo nuova istanza, lanciamo thread e ogni 5 secondi costruiamo oggetti.
Modificheremo la classe nel mentre, per verificare che il cambiamento prenda effetto sul controllo della classe modificata.

```
public class Example16 {
        private void go() {
                ReloadableHandler<Runnable> handler =
                ReloadableAspect.newHandler(Runnable.class,
```

```
                        "it.unipr.informatica.examples.SimpleReloadableWorker",
                        new String[] { "bin" });

                for (int i = 0; i < 10; ++i) {
                        try {
                                Runnable worker = handler.newInstance();
                                new Thread(worker).start();
                                Thread.sleep(5000);
                        } catch (Throwable throwable) {
                                throwable.printStackTrace();
                        }
                }
        }

        public static void main(String[] args) {
                new Example16().go();
        }
}
```

`ReloadableWorker.java`

Ci dice con un intero, la versione.

```
public interface ReloadableWorker extends Runnable {
        public int getVersion();
}
```

`SimpleReloadableWorker.java`

Implementazione.
Ha riferimento a delegato, ovvero riferimento che impone dipendenza tra classe `SimpleReloadableWorker` e la classe dell'attributo `WorkerDelegate`.

```
public class SimpleReloadableWorker implements ReloadableWorker {
        private WorkerDelegate delegate = new WorkerDelegate();
        // dato intero da cambiare durante l'esecuzione
        private int version = 1;

        @Override
        public void run() {
                delegate.work(this);
        }

        @Override
        public int getVersion() {
                return version;
        }
}
```

`WorkerDelegate.java`

La `run()` chiama `work()` passando `this`: verranno stampate le versioni, un'attesa casuale viene fatta e stampate una volta ancora le versioni.

```
public class WorkerDelegate {
        public void work(ReloadableWorker worker) {
                try {
                        String name = Thread.currentThread().getName();

                        System.out.println
```

```
                                ("Worker " + name + " in version "
                                        + worker.getVersion() + " started");

                        Thread.sleep((int) (5000 + 3000 * Math.random()));

                        System.out.println
                                ("Worker " + name + " in version "
                                        + worker.getVersion() + " terminated");
                } catch (Throwable throwable) {
                        throwable.printStackTrace();
                }
        }
}
```

Salvando il file `SimpleReloadableWorker.java` con dato intero `version` da `1` a `2` durante l'esecuzione.