14 сентября 2010 в 21:21

Загрузка классов в Java. Практика

Java*

Мотивация

Зачастую, архитектура сложных систем подразумевает использование механизма динамической загрузки кода. Это бывает необходимо, когда заранее не известно какой именно код будет исполняться в рантайме. Например, всем известная игра для программистов Robocode использует собственный загрузчик классов, для загрузки пользовательских танков в игру. Можно рассматривать отдельный танк как модуль, разрабатываемый изолированно от приложения по заданному интерфейсу. Похожая ситуация рассматривается в статье, только на максимально упрощенном уровне.

Кроме того, можно привести еще несколько очевидных примеров использования механизма динамической загрузки кода. Допустим байт-код классов хранится в БД. Очевидно, что для загрузки таких классов нужен специальный загрузчик, в обязанности которого будет входить еще и выборка кода классов из БД.

Возможно, классы требуется загружать по сети/через интернет. Для таких целей нужен загрузчик, способный получать байт-код по одному из сетевых протоколов. Можно также выделить, существующий в Java Class Library URLClassLoader, который способен загружать классы по указанному пути в URL.

Подготовка

Реализуемое в рамках статьи приложение будет представлять собой каркас движка для динамической загрузки кода в JRE и его исполнения. Каждый модуль будет представлять собой один Java класс, реализующий интерфейс Module. Общий для всех модулей интерфейс необходим для их инвокации. Здесь, важно понимать, что существует еще один способ исполнения динамического кода — Java Reflection API. Однако, для большей наглядности и простоты будет использоваться модель с общим интерфейсом.

При реализации пользовательских загрузчиков важно помнить следующее:
1) любой загрузчик должен явно или неявно расширять класс java.lang.ClassLoader;
2) любой загрузчик должен поддерживать модель делегирования загрузки, образуя иерархию;
3) в классе java.lang.ClassLoader уже реализован метод непосредственной загрузки — defineClass(...), который байт-код преобразует в java.lang.Class, осуществляя его валидацию;
4) механизм рекурентного поиска также реализован в классе java.lang.ClassLoader и заботиться об это не нужно;
5) для корректной реализации загрузчика достаточно лишь переопределить метод findClass() класса java.lang.ClassLoader.

Рассмотрим детально поведение загрузчика классов при вызове метода loadClass() для объяснения последнего пункта вышеуказанного списка.

Реализация по-умолчанию подразумевает следующую последовательность действий:
1) вызов findLoadedClass() для поиска загружаемого класса в кеше;
2) если класса в кеше не оказалось, происходит вызов getParent().loadClass() для делегирования права загрузки родительскому загрузчику;
3) если иерархия родительских загрузчиков не смогла загрузить класс, происходит вызов findClass() для непосредственной загрузки класса.

Поэтому для правильной реализации загрузчиков рекомендуется придерживаться указанного сценария — переопределения метода findClass().

Реализация

Определим интерфейс модулей. Пусть модуль сначала загружается (load), потом исполняется (run), возвращая результат и затем уже выгружается (unload). Данный код представляет собой API для разработки модулей. Его можно скомпилировать отдельно и упаковать в *.jar для поставки отдельно от основного приложения.

```
public interface Module {

  public static final int EXIT_SUCCESS = 0;
  public static final int EXIT_FAILURE = 1;

  public void load();
  public int run();
  public void unload();

}

* This source code was highlighted with Source Code Highlighter.
```

Рассмотрим реализацию загрузчика модулей. Данный загрузчик загружает код классов из определенной директории, путь к которой указан в переменной pathtobin.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;

public class ModuleLoader extends ClassLoader {

  /**
   * Путь до директории с модулями.
   */
  private String pathtobin;

  public ModuleLoader(String pathtobin, ClassLoader parent) {
    super(parent);
    this.pathtobin = pathtobin;
  }

  @Override
  public Class<?> findClass(String className) throws ClassNotFoundException {
    try {
      /**
       * Получем байт-код из файла и загружаем класс в рантайм
       */
      byte b[] = fetchClassFromFS(pathtobin + className + ".class");
      return defineClass(className, b, 0, b.length);
    } catch (FileNotFoundException ex) {
      return super.findClass(className);
    } catch (IOException ex) {
      return super.findClass(className);
    }

  }

  /**
   * Взято из www.java-tips.org/java-se-tips/java.io/reading-a-file-into-a-byte-array.html
   */
  private byte[] fetchClassFromFS(String path) throws FileNotFoundException, IOException {
    InputStream is = new FileInputStream(new File(path));

    // Get the size of the file
    long length = new File(path).length();
```

```
    if (length > Integer.MAX_VALUE) {
      // File is too large
    }

    // Create the byte array to hold the data
    byte[] bytes = new byte[(int)length];

    // Read in the bytes
    int offset = 0;
    int numRead = 0;
    while (offset < bytes.length
        && (numRead=is.read(bytes, offset, bytes.length-offset)) >= 0) {
      offset += numRead;
    }

    // Ensure all the bytes have been read in
    if (offset < bytes.length) {
      throw new IOException("Could not completely read file "+path);
    }

    // Close the input stream and return bytes
    is.close();
    return bytes;

  }
}

* This source code was highlighted with Source Code Highlighter.
```

Теперь рассмотрим реализацию движка загрузки модулей. Директория с модулями (файлами .class) указывается в качестве параметра приложению.

```
import java.io.File;

public class ModuleEngine {

  public static void main(String args[]) {
    String modulePath = args[0];
    /**
     * Создаем загрузчик модулей.
     */
    ModuleLoader loader = new ModuleLoader(modulePath, ClassLoader.getSystemClassLoader());

    /**
     * Получаем список доступных модулей.
     */
    File dir = new File(modulePath);
    String[] modules = dir.list();

    /**
     * Загружаем и исполняем каждый модуль.
     */
    for (String module: modules) {
      try {
        String moduleName = module.split(".class")[0];
        Class clazz = loader.loadClass(moduleName);
```

```
        Module execute = (Module) clazz.newInstance();

        execute.load();
        execute.run();
        execute.unload();

      } catch (ClassNotFoundException e) {
        e.printStackTrace();
      } catch (InstantiationException e) {
        e.printStackTrace();
      } catch (IllegalAccessException e) {
        e.printStackTrace();
      }
    }


  }

}

* This source code was highlighted with Source Code Highlighter.
```

Реализуем простейший модуль, который просто печатает на стандартный вывод информацию о стадиях своего исполнения. Это можно сделать в отдельном приложении добавив к CLASSPATH путь до скомпилированного .jar файла с классом Module (API).

```
public class ModulePrinter implements Module {

  @Override
  public void load() {
    System.out.println("Module " + this.getClass() + " loading ...");
  }

  @Override
  public int run() {
    System.out.println("Module " + this.getClass() + " running ...");
    return Module.EXIT_SUCCESS;
  }

  @Override
  public void unload() {
    System.out.println("Module " + this.getClass() + " inloading ...");
  }
}

* This source code was highlighted with Source Code Highlighter.
```

Скомпилировав данный код, результат в виде одного class файла можно скопировать в отдельную директорию, путь к которой необходимо указать в качестве параметра основного приложения.

Немного иронии

Динамическая загрузка кода, отличный и легальный способ предать обязанности по расширению системы пользователю ;)

java, classloading