**Pluggable решение с использованием Java**

Плагины (подключаемые модули) используются везде: в средах разработки, в браузерах, в файловых менеджерах и в медиа-плеерах. Сложно найти серьезное приложение, которое не предоставляло бы возможности расширения своих функциональных возможностей путем установки дополнительных плагинов. Даже небольшой текстовый редактор Notepad++ позволяет подключать плагины.

В статье рассматривается вопрос создания **pluggable** решений в Java.

На первый взгляд для расширения приложения нужно совсем немного: описать в *classpath* файлы jar с плагинами, определить интерфейс и файл с описанием плагинов. После запуска приложения обработать файл-описатель и инициализировать плагины.

Такой подход прост в реализации, но имеет ряд существенных недостатков.

1. Плагины должны определяться в classpath приложения, а это не всегда возможно.
2. Если по каким-то причинам, в classpath оказались jar’ы, в которых содержится несколько классов с одинаковыми именами, то поведение приложения станет непредсказуемым.
3. Плагины получают доступ к объектам “ядра” приложения. Это нежелательно: плагины должны жить в своей собственной “песочнице”.
4. Для того чтобы подключить новый плагин необходим перезапуск приложения. Это далеко не всегда допустимо. Если речь идет о разработке некоторой серверной архитектуры, где плагины выполняют роль сервисов, перезагружать сервер ради добавления сервиса неоправданно.

Для того, чтобы создать приложение, лишенное указанных недостатков, необходимо реализовать механизм, который позволит подгружать нужные классы во время исполнения приложения и ограничивать их "зону видимости". В Java таким механизмом являются [ClassLoader’ы](http://java-online.ru/java-classloader.xhtml).

**Загрузка классов ClassLoader**

На классический вопрос: как идентифицировать класс внутри JVM? Обычно отвечают – при помощи полного имени: имя пакетов плюс имя самого класса. Этот ответ не совсем полный/верный. В JVM вполне нормально могут существовать два разных класса с одинаковыми "полными" именами. Класс в JVM однозначно определяется своим полным именем и **ClassLoader**’ом, который его загрузил. Таким образом осуществляется изоляция ядра приложения.

**ClassLoader**’ы в Java имеют иерархическую структуру. Это означает, что loader-наследник "видит" все "свои" классы плюс классы loader’а-родителя. В то время, когда JVM начинает исполнять стартовый метод main, уже существует три class loader’a:

1. Bootstrap ClassLoader – вершина иерархии ClassLoader’ов. Загружает классы ядра Core Java.
2. Extension ClassLoader – загружает библиотеки из lib/ext. Наследуется от Bootstrap.
3. System ClassLoader – загружает классы из classpath. Наследуется от Extension.

Метод main инициализируется в System ClassLoader’е. Что же происходит, когда программа пытается обратиться к какому-нибудь классу, т.е. создать экземпляр, или выполнить Class.forName(…), который содержится в библиотеках из lib/ext?

"Правильные" class loader’ы спрашивают у родительского loader’а есть ли у него данный класс. И только, если родитель у себя этот класс не обнаружит, class loader пытается загрузить класс самостоятельно. То есть, происходит следующее:

1. System Class loader пробует загрузить класс, используя Extension Class loader.
2. Extension Class loader “просит” Bootstrap загрузить класс.
3. Bootstrap не может загрузить класс, ведь ему видны только Core классы. А искомый класс лежит в библиотеке из lib/ext.
4. Extension Class loader пытается загрузить класс. Если ему это удается, то он возвращает класс в System Class Loader.
5. System Class loader не пытается что-либо загружать. Класс уже найден.

Какой класс будет реально загружен, если в lib/ext и в classpath есть классы с одинаковым именем? Конечно же класс из lib/ext – на *classpath* никто и смотреть не будет. То же самое касается и порядка элементов в classpath – класс будет загружен из первого по очереди источника. То есть, если есть a.jar и b.jar и оба содержат класс *com.foo* то при выполнении

java -cp a.jar;b.jar ...

будет использоваться класс из a.jar, а при выполнении

java -cp b.jar;a.jar ...

из b.jar.

Таким образом, **ClassLoader** видит "свои" классы, и классы своих "предков". Ни классы из *ClassLoader*’ов потомков, ни, тем более, классы из "параллельных" *ClassLoader*’ов он видеть не будет. Более того, для JVM – это 'разные' классы.

При попытке привести один класс к другому JVM честно выбрасывает **ClassCastException**. Несмотря на то, что имена классов одинаковые.

**Структура плагинов**

Чтобы изолировать плагины друг от друга достаточно загружать их в отдельных **ClassLoader**’ах. Для этого необходимо определить **public** интерфейс, который должны будут наследовать плагины.

package com.test.plugin;

public interface Plugin

{

public void oke();

}

Данный интерфейс необходим для разработки **plugin**’ов. Упакуем его в отдельный jar c названием plugin-api.jar и будем включать в проекты-плагины. Этого уже достаточно для начала тестирования. В реальном же приложении, естественно, лучше добавить несколько *hook* методов и дать плагину доступ к среде, в которой он исполняется.

Создадим тестовый плагин, который будет только выводить сообщение в консоль.

package com.test;

import com.test.plugin.Plugin;

public class HelloPlugin implements Plugin

{

public void oke()

{

System.out.println("Hello world");

}

}

Плагин создается в отдельном проекте, чтобы основное приложение не "видело" классы плагина раньше режима **Runtime** и размещается в папке plugins основного приложения.

Создадим второй плагин, чтобы убедиться, что плагины не видят друг-друга. Второй класс-плагин будет иметь такое же имя, но текст сообщения будет выводиться другой.

package com.test;

import com.test.plugin.Plugin;

public class HelloPlugin implements Plugin

{

public void oke()

{

System.out.println("That's the second plugin");

}

}

Плагины готовы к "употреблению"; загрузим их в основное приложение. Для этого сформируем массив jar-файлов из папки plugins.

File pluginDir = new File("plugins"); //создаем объект для папки plugins

File[] jars = pluginDir.listFiles(new FileFilter() { //.listFiles() возвращает массив оъектов

классов из папки plugins

public boolean accept(File file) {

return file.isFile() && file.getName().endsWith(".jar"); //проверяет является ли

объект файлом, возвращает имя файла без пути

и которое заканчивается на .jar

}

});

В данном коде файловым фильтром FileFilter выделяем только jar-файлы.

**Загрузчик плагинов URLClassLoader**

Для каждого плагина будем использовать отдельный экземпляр *ClassLoader*’а. Разрабатывать собственный *ClassLoader* не придется, вполне подойдет существующий **URLClassLoader**, который с данной задачей справляется успешно.

Чтобы создать новый экземпляр **URLClassLoader**'а в конструктор этого класса нужно передать массив url’ов (папок и jar-файлов) и, указать объект *ClassLoader*, который *URLClassLoader* будет считать своим родителем. Если родителя явно не передавать, *URLClassLoader* будет пронаследован от текущего ClassLoader'а. А если передать null – то от Bootstrap ClassLoader'a.

Для каждого файла из папки создаем отдельный **URLClassLoader** и получаем объект типа Class по имени.

Class[] pluginClasses = new Class[jars.length];

for (int i = 0; i < jars.length; i++) { // jars – это из кода выше

try {

URL jarURL = jars[i].toURI().toURL();

URLClassLoader classLoader = new URLClassLoader(new URL[]{jarURL});

pluginClasses[i] = classLoader.loadClass("com.test.HelloPlugin");

} catch (MalformedURLException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

Классы плагинов загружаются по известному имени classLoader.loadClass(...). Для запуска плагина на исполнение создаем по объекту из каждого класса и вызываем метод **oke()**:

for (Class clazz : pluginClasses) {

try {

Plugin instance = (Plugin) clazz.newInstance(); //приведение к типу интерфейса

instance.oke(); //вызывается метод интерфейса(поэтому напрямую, а не через invoke())

} catch (InstantiationException e) {

e.printStackTrace();

} catch (IllegalAccessException e) {

e.printStackTrace();

}

}

После старта приложения в консоли должен отобразиться текст:

Hello world

That's the second plugin

Таким образом, плагины не вступили в конфликт между собой; они попросту не обратили друг на друга внимания.

**Дескрипторы**

Обращение к плагину по заранее известному имени несложно. Добавим в плагины файл-дескриптор, в котором можно будет указать, какой файл следует вызвать. Кроме того, для наглядности, наше приложение будет GUI приложением, а для вызовов плагинов будут использоваться кнопки. При нажатии на кнопку будет вызван метод **oke()** соответствующего плагина.

Для каждого плагина создаем файл-дескриптор “plugin.proptrties” типа **properties** с информацией о загружаемом файле и кнопке его вызывающей:

main.class = com.test.HelloPlugin

button.text = Plugin 1

Эту информацию, как и объект **plugin**'а удобно держать в отдельном java классе. В этом же классе будут жить методы по загрузке плагина.

public class PluginInfo

{

private Plugin instance;

private String buttonText;

private JButton associatedButton;

---------------------------------------------------------------------

Конструктор:

---------------------------------------------------------------------

public PluginInfo(File jarFile) throws PluginLoadException //собственное исключение

{

try {

Properties props = getPluginProps(jarFile);

if (props == null)

throw new IllegalArgumentException("No props file found");

String pluginClassName = props.getProperty("main.class"); //получаем путь и

имя файла-плагина из файла “plugin.proptrties”

if (pluginClassName == null || pluginClassName.length() == 0) { //ловим

собственное исключение

throw new PluginLoadException("Missing property main.class");

//реализуем собственное исключение

}

buttonText = props.getProperty("button.text"); //получаем имя кнопки из файла

“plugin.proptrties”

if (buttonText == null || buttonText.length() == 0) {

throw new PluginLoadException("Missing property button.text");

}

URL jarURL = jarFile.toURI().toURL();

URLClassLoader classLoader = new URLClassLoader(new URL[]{jarURL}); //нужно

будет сюда добавить “,getClass().getClassLoader().getParent()” для

получения ClassLoader-а от родителя ClassLoader-а запустившего этот класс

Class pluginClass = classLoader.loadClass(pluginClassName);

instance = (Plugin) pluginClass.newInstance();

} catch (Exception e) {

throw new PluginLoadException(e);

}

}

---------------------------------------------------------------------------

public Plugin **get**PluginInstance()

{

return instance;

}

public String **get**ButtonText()

{

return buttonText;

}

// метод **get**PluginProps() будет искать в нашем jar-файле файл с именем "plugin.properties" и если найдет, то выведет объект класса “Properties” заполненный информацией из найденного файла

private Properties **get**PluginProps(File file) throws IOException {

Properties result = null; //создаем объект класса Properties для последующего хранения

данных

JarFile jarr = new JarFile(file); //создаем объект класса JarFile, с указанием имени

просматриваемого jar-файла

Enumeration entries = jar.entries(); //возвращает объект интерфейса Enumeration,

содержащий имена всех файлов содержащихся в jar-файле

while (entries.hasMoreElements()) { //прокручиваем все имена из списка в объекте “entries”

JarEntry entry = entries.nextElement(); //создаем объект класса JarEntry с очередным

именем файла из объекта “entries”

if (entry.getName().equals("**plugin.properties**")) {

// That's it! Load props

InputStream is = null;

имя файла "plugin.properties"

try {

is = jarr.getInputStream(entry); //создает поток ввода для объекта jarr,

т.е. в “is” помещается информация,

находящаяся в файле "plugin.properties".

result = new Properties();

result.load(is);

} finally {

if (is != null)

is.close();

}

}

}

return result; //выводим, заполненный информацией из файла "plugin.properties", объект класса Properties

}

public void **set**AssociatedButton(JButton associatedButton)

{

this.associatedButton = associatedButton;

}

public JButton **get**AssociatedButton()

{

return associatedButton;

}

}

В главное приложение MainApp добавим отображение простого фрейма, который будет рисовать кнопку для каждого плагина. Кроме этого в родительском классе вынесем всю логику из метода main в метод start и добавим пару полей.

ЯДРО:

private Map<String, PluginInfo> pl;

private JFrame mainFrame;

public MainApp() {}

public void start()

{

Блок описан раньше. Он формирует массив из jar-файлов, находящихся в папке “plugins”

File pluginDir = new File("plugins");

File[] jars = pluginDir.listFiles(new FileFilter() {

public boolean accept(File file) {

return file.isFile() && file.getName().endsWith(".jar");

}

});

pl = new HashMap();

for (File file : jars) {

try {

pl.put(file.getName(), new PluginInfo(file)); //имена файлов + объекты файлов(классов)

} catch (PluginLoadException e) {

e.printStackTrace();

}

}

// Дальше JFrame, JButton это из Swing-а

Настройка окна с “закрывающим” крестиком в правом верхнем углу

mainFrame = new JFrame("Plugin test");

final JFrame frame = mainFrame;

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

frame.setSize(150, 300);

frame.getContentPane().setLayout(new FlowLayout()); //задание способа размещения кнопок в окне

synchronized (pl) {

for (PluginInfo pluginInfo : pl.values()) { //перебираем только объекты классов из “pl”

final PluginInfo plugin = pluginInfo;

final JButton button = new JButton(pluginInfo.getButtonText()); //создает

кнопку с именем

plugin.setAssociatedButton(button); //устанавливаем связь между кнопкой и объектом

класса из “pl”

Вешаем на кнопку слушателя событий с “реагирующим” методом “oke()”

button.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

plugin.getPluginInstance().oke();

}

});

frame.getContentPane().add(button); //размещаем кнопку на окне с крестиком

}

}

frame.setVisible(true); //отображаем окно на экране (так оно невидимо)

}

public static void main(String[] args)

{

new MainApp().start();

}

**Изоляция ядра приложения**

Несмотря на то, что плагины не видят друг-друга они видят классы ядра приложения, т.к. их ClassLoader-ы (*при таком написании как написано выше*) будут наследниками ClassLoader-а ядра. Это может стать проблемой - плагины не должны влиять на среду, где они исполняются, больше дозволенного. Основное приложение (ядро) необходимо изолировать от плагинов.

Техника для изоляции ядра точно такая же, что и для изоляции плагинов - необходимо загрузить ядро в отдельном **ClassLoader**'е. Но при этом необходимо учитывать некоторые нюансы: есть набор классов, которые должны видеть и плагины и ядро: как минимум, к этим классам относится интерфейс **Plugin**. Не забывайте, мы не можем загрузить этот класс отдельно для каждого ClassLoader'а иначе с точки зрения JVM это будут разные классы, и попытка привести один класс к другому будет выкидывать **ClassCastException**. Также необходимо передать в плагины ссылку на те части приложения, которые они реально могут изменить.

Перепишем метод main так, чтобы ядро приложения и плагины запускались в паралельных ClassLoader'ах. Для того, чтобы получить такой эффект реализуем механизм, похожий на механизм загрузки Tomcat.

Для этого в JVM будем загружать не само приложение, а небольшой клacc **Bootstr**, который инициализирует новый ClassLoader (сommonsLoader) - наследник Bootstrap ClassLoader.

Common ClassLoader будет запускать классы общие для ядра и плагинов (такие как plugin-api.jar и все библиотеки) из папки “lib”.

У Common ClassLoader будет наследник - App ClassLoader, который будет содержать классы ядра. Все ClassLoader'ы плагинов также будут наследниками Common.

То есть для того чтобы каждый плагин запускался своим загрузчиком, производным от сommonsLoader-а (т.к. класс PluginInfo запускается тем-же загрузчиком “appLoader”, что и ЯДРО) нужно модифицировать блок кода из PluginInfo следующим образом:

URLClassLoader classLoader = new URLClassLoader(new URL[]{jarURL},

getClass().getClassLoader().getParent());

получим Class Loader производный от Common и ссылок на ядро классы в нем не будет.

Bootstrap тоже будет простым и бесхитростным:

public class Boodstr{

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

File commonsDir = new File("lib"); //создаем объект для папки “lib”

File[] entries = commonsDir.listFiles(); //создаем массив объектов из файлов папки “lib”

URL[] urls = new URL[entries.length]; //создаем массив для URL длиной равной количеству объектов в массиве “entries”

for (int i = 0; i < entries.length; i++) {

Заполняем массив URL-ами

urls[i] = entries[i].toURI().toURL();

}

URLClassLoader commonsLoader = new URLClassLoader(urls, null); //получаем ClassLoader, наследник Bootstrap (т.к. “, null”), которым будут загружаться все файлы из папки “lib”

URL binDirURL = new File("bin").toURI().toURL();

URLClassLoader appLoader = new URLClassLoader(new URL[]{binDirURL}, commonsLoader); //получаем ClassLoader, наследник “commonsLoader”

Class appClass = appLoader.loadClass("com.juriy.plug.MainApp"); //загружаем файл ЯДРА

Object appInstance = appClass.newInstance();

Method m = appClass.getMethod("start");

m.invoke(appInstance);

}

}

Папка lib:

- файл plugin-api.jar (это jar с интерфейсом для плагинов).

- все библиотеки - они видны и плагинам и ядру одновременно.

Обратите внимание, что в последнем блоке использовался reflection API. Только так можно работать с методами класса из другого **ClassLoader**'а (т.к. класс Boodstr будет загружен системным ClassLoader-ом).

Bootstrap

Файлы/классы из папки lib

Loader расширений

commonsLoader

class ЯДРА

System

ClassLoader

class PluginInfo

appLoader

ClassLoader для Плагин1

Class для Плагин1

Class Bootstr:

создает новые

ClassLoader-ы

и запускает метод Start из

ЯДРА

ClassLoader для Плагин2

Class для Плагин2

**Интерфейс PluginContext (так нужно будет сделать для окончательной связки плагинов и приложения)**

Необходимо создать условия для взаимодействия плагинов с приложением. Таким образом можно контролировать, какие именно аспекты приложения доступны плагину.

Окружение плагина, как правило, называют его контекстом. Создадим интерфейс **PluginContext**, через который плагину будет передаваться ссылка на его собственную кнопку и на главный фрейм.

public interface PluginContext {

public JButton getButton();

Нужно создать еще этот интерфейс

public JFrame getFrame();

}

В интерфейс Plugin добавим метод init, который на вход будет принимать PluginContext. Осталось "внедрить" context в плагин. Сделаем это сразу после инициализации кнопки.

plugin.getPluginInstance().init(new PluginContext() {

public JButton getButton() {

return button;

Это добавляется в ЯДРО

}

public JFrame getFrame() {

return frame;

}

});

Теперь из плагина можно обратиться к основному приложению.

public class HelloPlugin implements Plugin

{

private PluginContext pc;

public void oke() {

Так переписываются плагины

System.out.println("That's the second");

pc.getButton().setText("Other text");

}

public void init(PluginContext pc) {

this.pc = pc;

}

}

**Динамическая загрузка и выгрузка плагинов**

Первый шаг, который нужно выполнить для удаления или добавления плагина, является проверка реестра плагинов. Роль реестра выполняет Map plugins. Ниже приводится код для удаления плагина. Добавление плагина абсолютно аналогичен.

public void removePluginByName(String jarName) throws PluginLoadException

{

if (!plugins.containsKey(jarName)) {

throw new PluginLoadException(jarName + " not loaded");

}

PluginInfo pluginInfo = plugins.get(jarName);

mainFrame.remove(pluginInfo.getAssociatedButton());

mainFrame.validate();

mainFrame.repaint();

synchronized (plugins) {

plugins.remove(jarName);

}

}

Скачать исходники примера : [plugins.zip](http://java-online.ru/sources/pluggable.zip) (27.2 Кб)

Данный пример используется для оборачивания исполняемого jar в exe-файл с помощью maven-плагина [launch4j](http://java-online.ru/maven-launch4j.xhtml).

В разделе «Сборка проекта» данный пример используется на странице описания [Многомодульного maven проекта](http://java-online.ru/maven-multimodule.xhtml).