Афоризм

Сиюминутные порывы нередко штопаешь всю жизнь.

Последние статьи

Поддержка проекта

Если Вам сайт понравился и помог, то будем признательны за Ваш «посильный» вклад в его поддержку и развитие

Загрузка классов, ClassLoader

В статье речь пойдет о классе **java.lang.ClassLoader**. Это самый необычный модуль не только в мире Java, но и среди большинства компилируемых языков. ClassLoader - самый низкоуровневый, «глубинный» механизм Java, позволяющий вмешиваться практически в ядро Java-машины, причем оставаясь в рамках программирования на Java.

Структуру организации загрузки java-кода при помощи **ClassLoader** и пример его использования можно увидеть на странице pluggable решений.

ClassLoader обеспечивает загрузку классов Java. Если говорить точнее, обеспечивают загрузку его наследники, конкретные загрузчики классов – сам ClassLoader абстрактен. Каждый раз, когда загружается какой-либо .class-файл, например, вследствие обращения к конструктору или статическому методу соответствующего класса – на самом деле это действие выполняет один из наследников класса ClassLoader.

Существует стандартный вариант реализации **ClassLoader** – так называемый системный загрузчик классов. Этот загрузчик используется по умолчанию при запуске приложений Java командой:

```
java Имя_главного_класса
```

Системный загрузчик классов реализует стандартный алгоритм загрузки из каталогов и JAR-файлов, перечисленных в переменной **CLASSPATH** (переменной среды либо параметре «-cp» утилиты «java»), а также из JAR-файлов, содержащих стандартные системные классы вроде **java.lang.String** и входящих в любой комплект поставки Java.

Одна из замечательных особенностей языка Java заключается в том, что можно реализовать свой собственный загрузчик классов – наследник **ClassLoader** – и использовать его вместо системного.

Наиболее популярный пример применения этой техники – Java-апплеты. Классы Java-апплетов, а также все классы, которыми они пользуются, автоматически загружаются с веб-сервера благодаря специальному загрузчику классов, реализованному «внутри» браузера.

Реализуя наследников **ClassLoader**, можно полностью контролировать процесс загрузки абсолютно всех Java-классов. Можно загружать их из любого источника, к примеру, из собственной системы каталогов, не отраженной в **CLASSPATH**, из базы данных или из Internet. Можно предоставить загрузку стандартных библиотечных классов системному загрузчику, но при этом протоколировать факт обращения к ним. При желании можно даже сконструировать байт-код класса в памяти и после этого работать с ним, как с нормальным классом, загруженным из «добропорядочного» .class-файла. Среди компилируемых языков подобные возможности встречаются разве что в ассемблере.

Единственное, что нельзя сделать с помощью **ClassLoader**'а – это создать новый класс, не располагая его байт-кодом. Для создания нового класса необходимо получить корректный байт-код класса (образ в памяти обычного .class-файла) в виде массива byte[]. Затем его нужно передать специальному стандартному методу ClassLoader.defineClass, который «превратит» его в готовый класс – объект типа Class.

В статье рассмотрен весь этот механизм и с его помощью показано решение практической задачи – динамическая подгрузка изменившихся версий .class-файлов без перезапуска главной Java-программы.

Технология загрузки Java классов

Основной способ работы с классом **ClassLoader** – это реализация наследников от него. Прежде чем переходить к рассмотрению этой техники, мы немного поговорим о том, как Java использует загрузчики классов.

Как уже было отмечено, в системе всегда существует по крайней мере один готовый наследник **ClassLoader** – системный загрузчик. Его всегда можно получить с помощью вызова **ClassLoader.getSystemClassLoader()** – статического метода класса ClassLoader, объявленного следующим образом:

```
public static ClassLoader getSystemClassLoader()
```

Когда запускается приложение Java с помощью стандартной команды:

```
java Имя_главного_класса
```

виртуальная машина **Java** первым делом создает системный загрузчик, загружает с его помощью .class-файла вашего главного класса и вызывает статический метод вашего класса, соответствующий объявлению:

```
public static void main(String[] argv)
```

(или же сообщает об ошибке, не обнаружив такого метода).

Java – язык с отложенной загрузкой кода. Первоначально загружается только один класс – тот, который передан в качестве параметра утилите «java». Как только код этого класса обращается к какому-то другому классу (любым способом: вызовом конструктора, обращением к статическому методу или полю), загружается другой класс. По мере выполнения кода, загружаются всё новые и новые классы. Ни один класс не загружается до тех пор, пока в нем не возникнет реальная потребность. Такое поведение заложено в стандартный системный загрузчик.

Главный класс приложения всегда загружается системным загрузчиком. А какие загрузчики будут использоваться для загрузки всех прочих классов?

В Java поддерживается понятие «текущего» загрузчика классов. Текущий загрузчик – это тот загрузчик классов (экземпляр некоторого наследника **ClassLoader**), который загрузил класс, код которого исполняется в данный момент. Каждый класс «помнит» загрузивший его загрузчик. Загрузчик, загрузивший некоторый класс, всегда можно узнать, вызвав метод **getClassLoader** у объекта типа Class, соответствующего данному классу :

```
public ClassLoader getClassLoader()
```

Например, если мы находимся внутри некоторого метода класса TestClass, то вызов TestClass.class.getClassLoader() вернет ссылку на загрузчик, загрузивший этот класс, т.е. загрузивший тот самый байт-код, который выполняет данный вызов.

Когда возникает необходимость загрузить другой класс вследствие обращения к его конструктору, статическому методу или полю, виртуальная Java-машина автоматически обращается к текущему загрузчику классов, о котором «помнит» текущий исполняемый класс. При этом другой класс также «запоминает» этот загрузчик в качестве текущего. Иначе говоря, текущий загрузчик, загрузивший данный класс, по умолчанию наследуется всеми классами, прямо или косвенно вызываемыми из данного.

Так как главный класс приложения обычно загружается системным загрузчиком, то он же

используется и для загрузки всех остальных классов, необходимых приложению. В случае Java-апплета браузер загружает главный класс апплета своим собственным загрузчиком (умеющим читать классы с веб-сервера); в результате тот же самый загрузчик используется для загрузки всех вспомогательных классов апплета.

На самом деле наследование текущего загрузчика – лишь поведение по умолчанию. Загрузчик классов можно написать и так, что он не будет наследоваться для некоторых классов. В тот момент, когда к загрузчику приходит запрос «выдать класс по заданному имени», он может передать этот запрос какому-нибудь другому загрузчику. Тогда данный класс и другие классы, вызываемые из него, будут загружаться новым загрузчиком. Например, специальный загрузчик, реализуемый браузером для загрузки классов апплетов, вполне может «передать свои полномочия» системному загрузчику, когда дело касается стандартного системного класса вроде **java.lang.String**.

Стандартный способ загрузить некоторый класс загрузчиком, отличным от текущего – специальная версия статического метода **Class.forName**:

```
public static Class forName(String name,
                            boolean initialize,
                            ClassLoader loader);
```

В качестве name передается полное имя класса (с указанием пакета), в качестве loader – требуемый загрузчик. Не столь очевидный (и не столь важный) параметр **initialize** управляет инициализацией класса, т.е. установкой значений всех static-полей класса и исполнением кода в секциях:

```
static {

  ...

}
```

Если значение **initialize** равно true, то инициализация происходит немедленно, в противном случае – откладывается до первого обращения к любому конструктору, статическому методу или полю этого класса.

Более простая форма метода **Class.forName**

```
public static Class forName(String className);
```

всегда использует текущий загрузчик классов. На самом деле, вызов

```
Class.forName(name)
```

эквивалентен вызову

```
Class.forName(name, true,
              Текущий_класс.class.getClassLoader());
```

где Текущий_класс – имя класса, внутри которого содержится данный вызов.

Загрузив класс, можно создать его экземпляр или вызвать статический метод средствами отражений. Дальше этот класс может обычными средствами языка Java обращаться к другим классам – для них будет вызван тот же самый загрузчик loader (либо какие-то другие загрузчики, если реализация loader в какой-то момент «решит» передать управление другому загрузчику). Простейший пример:

```
Class class_= Class.forName("Имя_класса", true, нестандартный_загрузчик);

class_.newInstance(); // создается экземпляр класса
```

Методы класса ClassLoader

Полный список методов можно найти в документации к **ClassLoader**'у? В данной статье рассмотрены наиболее важные из них.

Один из методов **ClassLoader** был уже рассмотрен. Это статический метод, возвращающий ссылку на стандартный системный загрузчик.

```
public static ClassLoader getSystemClassLoader()
```

Среди прочих методов самый «бросающийся в глаза»:

```
public Class loadClass(String name)
```

Этот метод загружает класс с заданным именем. На самом деле его реализация сводится к вызову другого protected-метода:

```
protected synchronized Class loadClass(String name,
                                       boolean resolve);
```

Как можно догадаться, переопределение этого protected-метода – один из основных способов для реализации собственного загрузчика классов.

Не понятно, почему метод **loadClass(String name)** объявлен как public. Ведь уже существует стандартный способ загрузки класса по имени, с помощью произвольного загрузчика – вызов

```
Class.forName("Имя_класса",true,loader);
```

Классы Class и ClassLoader расположены в общем пакете – так что метод **loadClass(String name)** вполне мог бы быть protected. Это не помешало бы методу **Class.forName** его вызвать. Может быть, раз уж loadClass – public-метод, то вместо **Class.forName(«Имя_класса», true, loader)** можно пользоваться прямым обращением loader.loadClass("Имя_класса")?

Судя по всему, следует все же всегда использовать вызов **Class.forName**. Хотя это совершенно не очевидно из документации. Дальше в статье показано, что метод Class.forName выполняет с классом некоторые дополнительные действия, в частности, кэширует его, обеспечивая, в отличие от прямого вызова **loadClass**, стабильную работу даже при недостаточно аккуратной реализации загрузчика loader.

Имеется также группа public методов, предназначенных для загрузки ресурсов:

```
public URL getResource(String name);

public InputStream getResourceAsStream(String name);

public final Enumeration getResources(String name);

public static URL getSystemResource(String name);

static InputStream getSystemResourceAsStream(String name);
```

```
static Enumeration getSystemResources(String name);
```

Методы Class.getResource и Class.getResourceAsStream обращаются к соответствующим методам текущего загрузчика, загрузившего данный класс. Главное отличие методов работы с ресурсами класса **ClassLoader** – абсолютные пути. Путь к ресурсу отсчитывается не от каталога, содержащего данный class-файл (как в случае Class.getResource и Class.getResourceAsStream), а от одного из каталогов, указанных в переменной **CLASSPATH**.

Следует обратить внимание, что названия методов **getSystemResource, getSystemResourceAsStream, getSystemResources** не означают, что загружаются какие-то особые «системные» ресурсы. Слово **«System»** в этих названиях говорит о том, что для загрузки ресурсов будет в любом случае использоваться стандартный системный загрузчик.

Любая реализация **ClassLoader** должна обеспечивать работоспособность перечисленных методов.

Перезагрузка классов «на лету»

Обычно в книгах по языку Java реализацию **ClassLoader** рассматривают на примере загрузки .class-файлов из какого-либо нестандартного источника, например, каталога, не указанного в переменной **CLASSPATH**. Такой пример достаточно прост, но, не очень интересен. В большинстве ситуаций поиск .class-файлов в путях, перечисленных в CLASSPATH, – вполне нормальное решение. Загрузка же из принципиально иных источников вряд ли будет полезна вне контекста куда более сложной задачи, включающей такие вопросы, как политика безопасности или кэширование загруженных файлов на локальном диске.

Попробуем решить другую задачу. Предположим, разрабатывается большая программа на Java. По каким-либо причинам эту программу нежелательно часто перезагружать: останавливать и запускать снова. Например, это может быть сложная серверная программа, каждую секунду обслуживающая многих клиентов, для которой даже сравнительно кратковременная неработоспособность является критичной. Или просто программа настолько «тяжелая», что полный ее перезапуск требует несколько минут, и часто перезапускать ее крайне неудобно. В то же время программа постоянно развивается. Создаются новые независимые блоки, переписываются и отлаживаются старые. Возможно, программа «умеет» исполнять сторонние классы, разработанные пользователями. В таких условиях возникает естественное желание, чтобы подключение новых классов или новых версий старых классов не требовало полной остановки и перезапуска программы.

Что касается действительно новых классов – тут проблем нет. Можно в любой момент скомпилировать новый класс с новым уникальным именем – даже когда программа уже запущена. Если ваша программа после этого выполнит вызов Class.forName(name) с этим именем (например, в результате автоматического сканирования каталогов поиска CLASSPATH в поисках новых .class-файлов), то этот класс будет успешно подключен, и программа сможет им пользоваться.

Что касается версий .class-файлов – тут все значительно хуже. Однажды обратившись к некоторому классу, стандартный системный загрузчик запомнит его в своем внутреннем кэше и будет всегда использовать именно его. Никакие последующие перекомпиляции .class-файла и даже физическое удаление этого файла не отразятся на работе этого класса. Кроме полного перезапуска программы (т.е. всей виртуальной машины Java), нельзя заставить системный загрузчик «забыть» тот байт-код класса, который он однажды загрузил.

Итак, основная задача – написать загрузчик классов, аналогичный стандартному системному, который, в отличие от него, умел бы «забывать» загруженные ранее версии классов и загружать .class-файлы заново.

Заодно будет решена и более простая традиционная задача – загрузка .class-файлов из собственного списка каталогов поиска. Это ничего не будет стоить, т.к. какая разница – загружать файлы непременно из каталогов **CLASSPATH** или из каких-либо других.

Необходимо учесть, что некоторые (или все) каталоги CLASSPATH могут попасть в наш список.

Для простоты не будем поддерживать в этом загрузчике работу с JAR-архивами. Все-таки JAR предназначен для упаковки достаточно стабильных версий программных модулей, которые вряд ли стоит обновлять настолько часто, что ради этого нежелательно выполнять перезапуск основной Java-программы. В частности, загрузку стандартных библиотечных классов (пакеты **java.lang** и подобные), которые обычно находятся в JAR-файле, мы возложим на системный загрузчик.

Назовем новый загрузчик DynamicClass-Overloader – «динамический перезагрузчик классов».

Методы загрузчика ClassLoader'a

Итак, необходимо создать наследника абстрактного класса **ClassLoader**, который умел бы загружать классы из некоторого заданного набора каталогов поиска также, как это делает системный загрузчик для каталогов, перечисленных в переменной **CLASSPATH**. В отличие от системного загрузчика наш вариант ClassLoader должен уметь «забывать» о загруженных ранее классах. Для этого необходимо реализовать следующие методы ClassLoader'a:

```
protected synchronized Class loadClass(String name,
                                       boolean resolve)
                           throws ClassNotFoundException;
protected Class findClass(String name)
                           throws ClassNotFoundException;
protected java.net.URL findResource(String name);
protected java.util.Enumeration findResources(String name)
                           throws IOException;
```

Абстрактный класс **ClassLoader** в действительности предоставляет реализацию для первого метода, loadClass, основанную на двух других protected-методах – **findLoadedClass** и **findClass**. Метод findLoadedClass объявлен как final – его переопределять не нужно. Эта реализация проверяет, не был ли загружен класс раньше вызовом «findLoadedClass(name)». Если нет – делает попытку загрузить класс стандартным образом, и если эта попытка терпит неудачу – обращается к методу findClass.

Для решения более традиционной задачи – обеспечения загрузки классов из нестандартного источника – такая реализация вполне подходит. В этом случае было бы достаточно реализовать метод findClass. Но нам необходимо загружать классы из вполне стандартного источника (хотя и нестандартным образом): из набора каталогов, который может соответствовать стандартной переменной **CLASSPATH**. Значит не нужно первым делом вызывать стандартный загрузчик, полагаясь на то, что для наших имен классов он потерпит неудачу, и loadClass обратится к нашему методу findClass. Нужно реализовать свою версию loadClass, действующую наоборот: вначале пытающуюся загрузить .class-файл самостоятельно и лишь в случае неудачи (скажем, для классов из пакета java.lang, которые обычно упакованы в JAR-архив) обращающуюся к системному загрузчику.

Методы **findResource** и **findResources**, подобно **findClass**, обеспечивают работоспособность public-методов загрузки ресурсов **getResource**, **getResourceAsStream** и **getResources**. Для ресурсов задачу динамической перезагрузки без рестарта программы решать не нужно – они и так всегда загружаются динамически. Поэтому, в отличие от ситуации с findClass, вполне достаточно переопределить методы findResource и findResources. Сделать это необходимо, так как, возможно, придется использовать наш загрузчик с каталогами, неизвестными системному загрузчику, т.е. отличными от каталогов **CLASSPATH**.

Не будем отвлекаться на реализацию метода findResources. Его использование представляется чересчур экзотичным. Реализуем только findResource – это почти ничего не стоит, и на этом методе основаны все типичные приемы работы с ресурсами (через методы Class.getResource и Class.getResourceAsStream).

Согласно документации **loadClass** должен просто найти или загрузить указанный класс с помощью **findClass**, после чего, если параметр resolve содержит true, вызвать для полученного класса protected-метод resolveClass. Что этот метод в точности делает – для нас в данный момент неважно.

Метод **findClass** должен загрузить байт-код указанного класса (в нашем случае это просто чтение файла), после чего выполнить для полученного массива байтов специальный «магический» метод **defineClass**:

```
protected final Class defineClass(String name, byte[] b,
                                  int off, int len)
                           throws ClassFormatError;
```

Это как раз то самое место, где цепочка байтов – образ .class-файла (фрагмент массива b длины len по смещению off) – «чудесным образом» превращается в готовый к использованию класс. Метод **defineClass**, как и следовало ожидать, реализован в native-коде. Именно он помещает байт-код класса в недра виртуальной машины, где он приобретает вид, пригодный для непосредственного исполнения на конкретной аппаратной платформе, в частности, компилируется в машинный код процессора для более быстрого исполнения (так называемая технология **Just-In-Time**, сокращенно JIT-компиляция).

Наконец, метод **findResource** должен просто найти файл, соответствующий данному ресурсу – по тем же правилам, по которым отыскивается файл класса в методе **findClass** – и вернуть ссылку на него в виде URL.

Системный загрузчик классов не просто загружает файлы классов с диска, но еще и запоминает их во внутреннем кэше – так что последующие обращения к **loadClass** для того же имени класса просто выдают готовый объект Class из кэша. Кэширование, вообще говоря, представляется разумной идеей: зачем каждый раз заново обращаться к диску? Мы будем хранить кэш в нестатическом private-поле типа **java.util.HashMap** нашего класса DynamicClassOverloader. Таким образом, каждый новый экземпляр нашего загрузчика будет создаваться с новым кэшем, и для «забывания» загруженных ранее классов будет достаточно просто создать новый экземпляр загрузчика.

Пробный вариант:

```
import java.io.*;

public class DynamicClassOverloader extends ClassLoader
{
    private Map classesHash = new java.util.HashMap();
    public final String[] classPath;
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    public DynamicClassOverloader(String[] classPath)
    {
        // Набор путей поиска - аналог переменной CLASSPATH
        this.classPath= classPath;
    }
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    protected synchronized Class loadClass(String name,
                                           boolean resolve)
                              throws ClassNotFoundException
    {
        Class result= findClass(name);
        if (resolve)
            resolveClass(result);
        return result;
    }
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
protected Class findClass(String name)
                          throws ClassNotFoundException
{
    Class result= (Class)classesHash.get(name);
    if (result!=null) {
        /*
         * System.out.println("% Class " + name +
         *                        " found in cache");
        */
        return result;
    }

    File f= findFile(name.replace('.','/'),".class");
    // Класс mypackage.MyClass следует искать файле
    // mypackage/MyClass.class
    /*
     * System.out.println("% Class " + name +
     *                    (f==null?"":" found in "+f));
    */
    if (f==null) {
         // Обращаемся к системному загрузчику в случае
         // неудачи. findSystemClass - это метод
         // абстрактного класса ClassLoader с объявлением
         // protected Class findSystemClass(String name)
         // (т.е. предназначенный для использования в
         // наследниках и не подлежащий переопределению).
         // Он выполняет поиск и загрузку класса по
         // алгоритму системного загрузчика. Без вызова
         // findSystemClass(name) нам пришлось бы
         // самостоятельно позаботиться о загрузке всех
         // стандартных библиотечных классов типа
         // java.lang.String, что потребовало бы
         // реализовать работу с JAR-архивами
         // (стандартные библиотеки почти всегда
         // упакованы в JAR)
         return findSystemClass(name);
    }

    try {
        byte[] classBytes= loadFileAsBytes(f);
        result= defineClass(name, classBytes, 0,
                                  classBytes.length);
    } catch (IOException e) {
        throw new ClassNotFoundException(
               "Cannot load class " + name + ": " + e);
    } catch (ClassFormatError e) {
        throw new ClassNotFoundException(
            "Format of class file incorrect for class "
            + name + " : " + e);
    }
    classesHash.put(name,result);
    return result;
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
protected java.net.URL findResource(String name)
{
    File f= findFile(name, "");
    if (f==null)
        return null;
    try {
        return f.toURL();
    } catch(java.net.MalformedURLException e) {
```

```
            return null;
        }
    }
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    /**
     * Поиск файла с именем name и, возможно, расширением
     * extension в каталогах поиска, заданных параметром
     * конструктора classPath. Имена подкаталогов в name
     * разделяются символом '/' - даже если в операционной
     * системе принят другой разделитель для подкаталогов.
     * (Именно в таком виде получает свой параметр метод
     * findResource.)
     */
    private File findFile(String name, String extension)
    {
        File fl
        for (int k=0; k <classPath.length; k++) {
            f = new File((new File(classPath[k])).getPath()
                                    + File.separatorChar
                                    + name.replace('/',
                                      File.separatorChar)
                                    + extension);
            if (f.exists())
                return f;
        }
        return null;
    }
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    public static byte[] loadFileAsBytes(File file)
                                          throws IOException
    {
        byte[] result = new byte[(int)file.length()];
        FileInputStream f = new FileInputStream(file);
        try {
            f.read(result,0,result.length);
        } finally {
            try {
               f.close();
            } catch (Exception e) {
               // Игнорируем исключения, возникшие при
               // вызове close. Они крайне маловероятны и не
               // очень важны - файл уже прочитан. Но если
               // они все же возникнут, то они не должны
               // замаскировать действительно важные ошибки,
               // возникшие при вызове read.
            };
        }
        return result;
    }
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
}
```

Для проверки создаем тестовый класс TestModule.java, который будем загружать нашим загрузчиком:

```
public class TestModule
{
    public String toString()
    {
        return "TestModule, version 1!";
    }
```

```
}
```

Создаем тест Test.java, который будет загружать этот класс:

```
import java.io.*;

public class Test
{
    public static void main(String[] argv) throws Exception
    {
        ClassLoader loader;
        for (;;) {
            loader= new DynamicClassOverloader(
                                    new String[] {"."});
            // текущий каталог "." будет единственным
            // каталогом поиска
            Class clazz= Class.forName("TestModule", true,
                                    loader);
            Object object= clazz.newInstance();
            System.out.println(object);
            new BufferedReader(
                new InputStreamReader(System.in)).readLine();
        }
    }
}
```

Размещаем все эти файлы в один каталог, компилируем и запускаем Test:

```
java Test
```

В каждой итерации бесконечного цикла создается экземпляр нашего загрузчика loader, с его помощью загружается класс TestModule, создается его экземпляр и распечатывается, при этом, как обычно, неявно используется метод toString(), реализованный в TestModule. Затем ожидается нажатие на клавиатуре клавиши ENTER (либо Ctrl-C для выхода из программы).

Пока наш тест ждет нажатия ENTER, перейдем в другое окно (ОС Windows) или консоль (ОС Unix), чуть-чуть исправим класс TestModule: изменим результат toString() на «TestModule, version 2!» и перекомпилируем его. После чего вернемся в наш тест и нажмем ENTER.

Сработало! В следующей итерации цикла мы видим результат работы свежей версии класса TestModule.class – будет напечатана новая строка «TestModule, version 2!».

Мы добились успеха, не выходя из программы, модифицировали class-файл, и новая версия класса была успешно загружена.

Класс TestModule можно заменить на любой другой сложный класс, конструктор которого инициирует сколь угодно сложную цепочку действий. Все классы, которые в процессе этого будут задействованы, будут точно так же динамически перезагружаться.

Первое тестирование

Тест работает, но возникает законный вопрос – ну и что? Да, мы можем в определенный момент запустить некий класс, заданный своим именем, после чего он будет выполнять какие-то действия и, в конце концов, вернет строку «object.toString()». Но это в общем-то ничем принципиально не отличается от запуска новой java-программы стандартным вызовом

```
java Имя_класса
```

Вспомним постановку задачи. У нас есть большая, очень большая Java-программа, полный перезапуск которой занимает длительное время и крайне нежелателен. Мы хотим иметь возможность в какой-то момент быстро перезагрузить некоторые ФРАГМЕНТЫ программы, чтобы все классы, относящиеся к этим фрагментам, после этого момента заново считывались с диска. Вероятнее всего, эти классы должны активно взаимодействовать друг с другом и со стационарной, неперезагружаемой частью программы. Например, они могут реализовывать различные интерфейсы, которыми пользуется стационарная часть программы, их экземпляры могут сохраняться в различных переменных в стационарной части и т. д.

В терминах нашего загрузчика классов это означает, что мы должны уметь взаимодействовать с классами, загруженными вызовом

```
Class.forName("Имя_класса", true, loader)
```

Мы должны работать с их экземплярами, вызывать методы, обращаться к полям, перехватывать исключения, причем по возможности стандартными способами языка Java. Аналогично классы, загруженные разными экземплярами загрузчика, например, отвечающие за слабо связанные фрагменты большой программы, которые нужно перезагружать в разные моменты – должны уметь взаимодействовать друг с другом.

Казалось бы, с этим нет никаких проблем. В тесте мы получили вызовом **newInstance()** переменную типа Object. Но если мы знаем, что ее тип – TestModule, мы можем спокойно привести ее к этому типу и работать дальше обычным образом:

```
...

Class clazz= Class.forName("TestModule",true,loader);

TestModule testModule = (TestModule)clazz.newInstance();
```

работаем с полями testModule, вызываем методы и т.д.

Если бы здесь был обычный вызов **Class.forName(«TestModule»)**, все было бы нормально. Это был бы простейший вариант классической схемы построения расширяемых систем. В качестве аргумента forName мог бы выступать любой наследник TestModule (или класс, реализующий интерфейс TestModule), реализация которого неизвестна и не обязательно доступна в момент компиляции системы. Об этом способе работы с отражениями рассказывалось в первой части статьи.

Но с нашим необычным загрузчиком классов нас ждет неприятная неожиданность. При попытке приведения типа будет возбуждено исключение **ClassCastException** – ошибка приведения типа!

Новые свойства языка – «динамические» и «истинно-статические» классы

Здесь мы столкнулись с проявлением достаточно общей проблемы. Чтобы понять ее природу, нужно заново внимательно рассмотреть понятие класса в языке Java.

В мире объектно-ориентированного программирования, в частности в Java, мы привыкли к тому, что существует два уровня иерархии сущностей – класс и экземпляр. Класс с заданным именем в системе всегда один – он однозначно идентифицируется своим полным именем. Экземпляров же у класса может быть много. Так, в Java поля с квалификатором **static** принадлежат целому классу, каждое такое поле существует в системе в единственном экземпляре. В отличие от этого для обычного (нестатического) поля отдельная его версия присутствует в каждом экземпляре класса.

Создав наш загрузчик **DynamicClassOverloader**, всегда загружающий свежие версии class-файлов, мы принципиально изменили ситуацию. Теперь есть три уровня иерархии: сам класс, версия класса – тот байт-код, который был загружен конкретной версией DynamicClassOverloader (возможно, меняющийся в процессе работы программы) и экземпляры конкретной версии класса.

На самом деле виртуальная машина Java «считает» классом как раз то, что для нас является версией. Хотя это и неочевидно из документации, но класс в Java однозначно идентифицируется не только полным именем, но еще и экземпляром загрузчика, загрузившего этот класс. Каждый экземпляр загрузчика классов порождает собственное пространство имен, внутри которого классы действительно однозначно идентифицируются полными именами, но классы в разных пространствах имен, загруженные разными загрузчиками, вполне могут иметь идентичные имена.

На самом деле, с точки зрения Java, класс TestModule, возникающий в результате прямого вызова «TestModule testModule= ...» в тексте файла Test.java, и класс TestModule, полученный вызовом:

```
Class.forName("TestModule",true,loader)
```

– это два совершенно разных класса. Первый был загружен системным загрузчиком (вместе с самым главным классом Test), а второй – одним из экземпляров нашего загрузчика **DynamicClassOverloader**. По классическим законам объектно-ориентированного программирования приведение типов между ними невозможно.

Более того, каждая итерация нашего цикла, оказывается, порождала новую версию класса TestModule, не связанную с предыдущими. В этом можно убедиться следующим образом. Модифицируем класс TestModule:

```
public class TestModule
{
    private static int counter = 0;
    public String toString() {
        return "TestModule, version 1! " + (counter++);
    }
}
```

В нормальных условиях каждое новое обращение к методу toString() такого класса привело бы к получению строки с новым, увеличенным на 1 значением счетчика counter. Если бы в тесте Test.java был обычный вызов **Class.forName**(«TestModule»), мы бы это и увидели: на каждой итерации цикла распечатывались бы разные значения счетчика. А с нашим загрузчиком мы все время будем видеть нулевое значение. Каждое пересоздание экземпляра загрузчика **DynamicClassOverloader** приводит к появлению нового пространства имен, в котором инициализируется совершенно новая версия класса TestModule, ничего не знающая о предыдущих версиях и о содержащихся в них статических переменных.

Фактически, мы придали языку Java новое свойство – «динамичность» классов. Теперь, обращаясь прямо или косвенно к любому классу, придется думать о том, какая именно версия этого класса будет использована и можно ли ее использовать совместно с тем классом, который к ней обращается.

Свойство, что и говорить, крайне неудобное. Как же теперь работать с «динамическими» классами? Ведь все версии классов, которыми пользуется наш первый класс, загруженный **DynamicClassOverloader**, и которые тем самым тоже загружены нашим загрузчиком, – все эти версии неизвестны в пространстве имен стационарной части – в нашем случае в главном классе Test.

Решение этой проблемы – заблокировать свойство «динамичности» для некоторых классов,

т.е. потребовать, чтобы такие классы наш загрузчик загружал стандартным способом – через вызов «findSystemClass(name)». Назовем такие классы «истинно-статическими» – «true-static». Такие «true-static»-классы можно будет свободно совместно использовать в стационарной части программы и всех версиях «динамических» классов. Для «true-static»-классов всегда будет существовать только одна версия, как и предполагается в обычном языке Java, и не будет проблем несоответствия типа. Можно, например, сделать «true-static» все ключевые интерфейсы, основные типы данных, которые должны получать и возвращать «динамические» классы, базовые типы исключений, подлежащие перехвату и единообразной обработке, и т. п.

В сущности, уже в реализованной нами версии загрузчика существовали «true-static»-классы – это библиотечные классы из пакетов типа **java.lang**, которые мы не пытались грузить самостоятельно. Скажем, такими были стандартные типы Object и String. Именно поэтому в первоначальном варианте теста мы смогли получить от созданного экземпляра динамического класса TestModule строку String – результат метода toString().

Можно придумать много соглашений, по которым загрузчик должен опознавать «true-static»-классы. Например, можно проверить существование некоторого ключевого static-поля или проверить, не реализован ли в классе некоторый специальный пустой интерфейс. Мы ограничимся наиболее простым (хотя и не всегда удобным) вариантом: будем проверять, не содержит ли имя класса name цепочки символов «true-static» без учета регистра символов.

Итак, начинаем модифицировать наш загрузчик **DynamicClassOverloader** и добавляем в методе findClass сразу перед вызовом:

```
File f= findFile(name.replace(".","/"),".class");
```

дополнительную проверку имени name. Вот начало исходного текста нового метода **findClass**:

```
protected Class findClass(String name)
                          throws ClassNotFoundException
{
    Class result= (Class)classesHash.get(name);
    if (result!=null) {
      // System.out.println("% Class " + name + "
      //                             found in cache");
      return result;
    }

    if (name.toLowerCase().indexOf("truestatic")!=-1)
        return findSystemClass(name);

   File f= findFile(name.replace('.','/'),".class");
  ...
```

Попробуем этим воспользоваться. Создаем «true-static»-класс TrueStaticModule.java:

```
public class TrueStaticModule
{
    protected static int counter= 0;
    public int getCounter()
        {
        return counter;
    }
}
```

В нем есть public-метод getCounter(), которым мы собираемся пользоваться в стационарной части программы. Наследуем от него «динамический» класс DynamicModule.java:

```
public class DynamicModule extends TrueStaticModule
{
    public String toString()
    {
        return "DynamicModule, version 1! "+(counter++);
    }
}
```

Наконец, переписываем тест Test.java – «стационарную часть» программы:

```
import java.io.*;

public class Test
{
    public static void main(String[] argv) throws Exception
    {
        ClassLoader loader;
        for (;;) {
            loader= new DynamicClassOverloader(
                            new String[] {"."});
            // текущий каталог "." будет единственным
            // каталогом поиска
            Class clazz= Class.forName("DynamicModule",
                                       true, loader);
            TrueStaticModule tsm;
            tsm=(TrueStaticModule) clazz.newInstance();

            System.out.println(tsm.getCounter());
            System.out.println(tsm);

            new BufferedReader(
               new InputStreamReader (System.in)).readLine();
        }
     }
}
```

Компилируем все эти файлы и запускаем:

```
java Test
```

Все должно работать нормально. Как и раньше мы можем прямо в процессе работы теста модифицировать и перекомпилировать класс DynamicModule, и это изменение будет учтено. Но для «общения» со стационарной частью программы и для хранения счетчика обращений к методу **toString()** теперь используется «true-static»-класс TrueStaticModule. Поэтому мы не получаем исключения **ClassCastException**, а счетчик counter корректно увеличивается на протяжении всей работы теста.

Поставленную задачу можно считать в принципе решенной.

Чтобы использование нашего загрузчика стало действительно удобным, стоило бы еще реализовать специальный сервисный «true-static»-класс с методом forName, аналогичным стандартному forName. Только в отличие от стандартного, наш forName обращался бы к нашему загрузчику, экземпляр которого, внутреннее private-поле, создавался бы при первом обращении к **forName**. Параметры конструктора для нашего загрузчика можно было бы настраивать с помощью специальных полей сервисного класса. Кроме того, в нашем сервисном классе был бы специальный метод invalidate, обнуляющий private-поле с нашим загрузчиком и вынуждающий метод forName при следующем вызове заново создать загрузчик.

Метод **invalidate** можно было бы вызывать в Java-программе всякий раз, когда требуется перезагрузить с диска новые версии всех «динамических» классов. Написание подобного сервисного класса – достаточно понятная задача, и мы не будем на ней останавливаться.

Правила работы с «динамическими» классами

При практическом использовании описанного выше загрузчика классов программирование в языке Java заметно усложняется. Нужно помнить о возможной «динамичности» классов: каждый экземпляр загрузчика порождает отдельную версию каждого такого класса в собственном пространстве имен. Нужно заботиться о том, чтобы некоторые классы были «true-static». Все это требует четкого понимания описанных выше механизмов и достаточной аккуратности.

Ниже сформулировано несколько правил, которыми следует руководствоваться при программировании в таком, изменившем свое поведение, языке Java.

Будем называть динамической частью Java-программы ту систему «динамических» классов, которая загружается некоторым экземпляром нашего загрузчика DynamicClassOverloader, и стационарной частью – основную систему классов, которая загружает динамическую часть, используя экземпляры DynamicClassOverloader. В программе может быть и несколько динамических частей, никак не связанных друг с другом, одновременно загруженных несколькими экземплярами DynamicClass-Overloader. В стационарную часть входят, в частности, все «true-static»-классы.

ПРАВИЛО А. Стационарную часть программы – в частности, все «true-static»-классы – следует разрабатывать таким образом, чтобы никак не использовать информацию о структуре «динамических» классов: имена классов, имена их членов, типы параметров у методов. Иными словами, нельзя прямо ссылаться на конкретные имена «динамических» классов и обращаться к ним средствами языка Java. Единственным способом взаимодействия стационарной части программы с «динамическими» классами должна быть их загрузка вызовом

```
Class.forName("TestModule",true,loader)
```

и использование системы «true-static»-классов (или интерфейсов), известных и стационарной, и динамической частям. Например, можно обращаться (через приведение типа) к «true-static»-интерфейсам, которые реализуют «динамические» классы, получать результаты методов этих интерфейсов в виде «true-static»-классов, перехватывать «true-static»-исключения и т. д.

Сформулированное правило вполне логично и «выдержано в духе» объектно-ориентированного программирования. «Динамические» классы для того и сделаны «динамическими», чтобы их можно было разрабатывать и компилировать уже после того, как стационарная часть программы скомпилирована и запущена. Поэтому стационарная часть и не должна ничего «знать» об этих классах кроме того, что они, возможно, реализуют какие-то заранее известные «true-static»-интерфейсы или, скажем, как-то работают со static-полями заранее известных «true-static»-классов.

В первом нашем тесте, вызвавшем ошибку приведения типа, мы нарушили это правило. При приведении типа мы непосредственно сослались из стационарной части программы на имя «динамического» класса TestModule. В правильном решении, которое мы привели позже, мы преобразовывали полученный объект неизвестного нам «динамического» типа к типу «true-static»-класса TrueStaticModule – предка «динамического» класса.

Заметим, сформулированное правило не является категорическим. Его вполне можно нарушать, т.е. прямо ссылаться из стационарной части на «динамические» классы. Просто следует помнить, что классы, которые будут при этом задействованы, и классы с теми же именами, к которым будет обращаться динамическая часть программы, – это совершенно

разные классы, лежащие в разных пространствах имен.

Например, и стационарная, и динамическая части программы могут активно пользоваться некоторыми сервисными библиотеками, и эти библиотеки нет никакой необходимости делать «true-static». Они вполне могут быть «динамическими». Если мы изменим реализацию некоторого метода в таком классе-библиотеке и перекомпилируем этот класс, то стационарная часть этого «не заметит» и будет продолжать работать со своей старой версией библиотеки, а динамическая часть, после очередного пересоздания экземпляра загрузчика, воспользуется новой версией.

Из общего правила А можно выделить несколько более простых частных правил.

ПРАВИЛО А1. Не следует ссылаться из стационарной части программы (в частности, «true-static»-класса) на какие-либо поля, конструкторы или методы «динамического» класса. Точнее, следует иметь в виду, что такая ссылка означает обращение к версии «динамического» класса, загруженной системным загрузчиком по умолчанию, а никак не к версии, загруженной нашим загрузчиком DynamicClassOverloader.

ПРАВИЛО А2. Все аргументы и результат любого метода в «true-static»-классе, используемого для связи между стационарной и динамической частями программы, например, метода «true-static»-интерфейса, который реализуют конкретные «динамические» классы, – должны иметь либо примитивный тип, либо тип «true-static»-класс. То же самое относится к public-полям, используемым с аналогичными целями. (К «true-static»-классам мы относим также все стандартные системные классы, вроде **java.lang.String** или **java.io.File**, которые наш загрузчик не пытается грузить самостоятельно.)

ПРАВИЛО А3. Если «динамические» классы возбуждают какие-либо исключения, которые, возможно, потребуется перехватить оператором

```
catch (Тип_исключения e)
```

в стационарной части программы, то перехватываемый класс Тип_исключения должен быть «true-static».

ПРАВИЛО В. Нужно учитывать, что каждый экземпляр загрузчика порождает независимое пространство имен. Любая ссылка на «динамический» класс: на его конструкторы, методы, статические или обычные поля действует только в пределах текущего пространства имен и не относится к версиям того же класса, загруженным другими экземплярами загрузчика.

Частные следствия из этого правила:

ПРАВИЛО В1. Не следует думать, что статические поля «динамического» класса существуют в системе в единственном экземпляре. В каждом пространстве имен существует отдельная версия класса со своим набором статических полей.

Например, есть такая практика управления поведением Java-класса. Объявляется статическое public-поле, скажем,

```
public static boolean debugMode= false;
```

влияющее на работу некоторых методов класса. Обычно используется значение этого поля по умолчанию. Но при желании главный класс Java-приложения на этапе общей инициализации системы и загрузки конфигурационных файлов может записать в это поле другое значение.

С «динамическими» классами такой прием «не проходит». Главный класс в стационарной части может повлиять на значение static-поля только для одной версии «динамического» класса, загруженной системным загрузчиком. Все последующие версии, загруженные экземплярами DynamicClassOverloader, получат умолчательное, заново инициализированное значение этого static-поля.

Если «динамический» класс действительно нуждается в наборе истинно глобальных полей, разделяемых всеми своими версиями, то самое естественное решение – определить внутри этого класса локальный «true-static»-класс. Например:

```
public class Мой_динамический_класс
{
    public static class TrueStaticSection
    {
        public static boolean debugMode= false;
        // другие глобальные переменные
    }
    ...
}
```

ПРАВИЛО B2. Если «динамический» класс нуждается в доступе к некоторым полям, методам, конструкторам или локальным классам некоторого «true-static»-класса, то все эти члены «true-static»-класса обязаны быть public, даже если «динамический» и «true-static»-класс лежат в одном пакете или внутри одного java-файла. Исключение: если «динамический» класс наследует «true-static», то он, как обычно, имеет доступ к protected-членам «true-static»-класса.

Например, если в «динамический» класс вложен локальный «true-static»-класс, то «динамический» класс, вопреки обычной практике, не может пользоваться private-полями или полями с дружественным доступом вложенного класса.

Дело в том, что с точки зрения виртуальной машины Java-классы, загруженные разными загрузчиками и поэтому лежащие в разных пространствах имен, всегда имеют друг с другом столь же слабую связь, как и классы, лежащие в разных пакетах. «Дружественный» доступ или доступ к private-членам вложенного класса между разными пространствами имен «не работает».

Будьте внимательны: подобная ошибка (разумеется) не отслеживается компилятором Java и обнаруживается в виде системного исключения уже при выполнении программы.

Есть еще одно специфическое правило, не вытекающее из описанных выше общих принципов.

ПРАВИЛО C. Если некоторые методы класса являются native – эти методы реализованы в отдельном машинно-зависимом модуле, загружаемом методом **System.loadLibrary** в секции статической инициализации класса, то такой класс обязан быть «true-static».

Это – внутреннее свойство современной версии Java-машины, по крайней мере, для платформы Windows. Java-машина не допускает повторной загрузки внешнего модуля вызовом **loadLibrary** с тем же самым именем – такая попытка вызывает исключение. Для «динамических» классов инициализация происходит многократно в каждой версии класса. Если в «динамическом» классе попытаться в соответствии с документацией обратиться к **System.loadLibrary** в секции статической инициализации:

```
static {
  System.loadLibrary ("Имя_внешнего_модуля");
}
```

то уже вторая загрузка такого класса нашим загрузчиком возбудит внутреннее исключение с сообщением: «данный внешний модуль уже загружен другим загрузчиком классов».

Не обесценивают ли описанные сложности нового удобства – возможности «на лету» перекомпилировать и перезагружать классы? Я считаю, что нет – при условии грамотного проектирования системы в целом. На самом деле, описанные выше правила касаются только разработки сравнительно небольшого блока: стандартизованного интерфейса между

стационарной и динамической частью. При разработке остальных блоков стационарной части, не работающих с «динамическими» классами, все описанные нюансы несущественны: стационарная часть загружается системным загрузчиком по обычным правилам.

При разработке динамических частей программы, расширяющих ее функциональность, возможно, сторонними разработчиками, все эти правила, за исключением очень простого правила С, обычно также можно спокойно игнорировать. Все правила, за исключением С, относятся к доступу к «динамическим» классам из стационарной части и к работе с различными пространствами имен, но никак не затрагивают разработку системы «динамических» классов в пределах одного пространства имен. Поэтому все новые классы, разрабатываемые для расширения динамической части программы при уже выработанном протоколе общения со стационарной частью, почти всегда можно объявлять «динамическими» и спокойно разрабатывать по обычным правилам языка Java.

Особенности кэширования: различие loadClass и forName

Здесь хотелось бы ненадолго вернуться назад к нашей реализации DynamicClassOverloader. В приведенной выше реализации есть две закомментированные строки с вызовом System.out.println, позволяющим увидеть момент загрузки класса и извлечение его из кэша.

Если их раскомментировать, можно увидеть удивительную вещь. Когда я воспользовался таким загрузчиком для исполнения большого проекта на Java, включающим тысячи различных классов и реализующим интенсивные вычисления, я обнаружил, что проверка наличия класса в кэше

```
Class result = (Class)classesHash.get(name);

if (result!=null) {
...
}
```

не срабатывает никогда!

Я пробовал повторно обращаться к загруженным классам всеми возможными способами: прямым обращением к классу, через **Class.forName** с различными наборами параметров, путем наследования, перехвата исключений и т. д. Во всех случаях, кроме прямого обращения к методу нашего загрузчика **loadClass**, до исполнения наших методов loadClass и **findClass** дело просто не доходило. Очевидно, Java-машина реализует дополнительное кэширование загруженных классов. Пока мы явно не подаем указание использовать другой экземпляр загрузчика с помощью соответствующего аргумента метода **Class.forName**. Все классы, уже загруженные один раз нашим загрузчиком, автоматически извлекаются из какого-то внутреннего кэша. Фактически наше кэширование, реализованное в DynamicClassOverloader, оказалось ненужным – классы и так прекрасно кэшируются.

Я не знаю, насколько можно полагаться на эту особенность Java-машины. Документация к классу **ClassLoader** рекомендует использовать вызов protected-метода **findLoadedClass** для выяснения, был ли уже загружен данный класс. Но в моих тестах мне не удавалось этим воспользоваться – этот метод всегда возвращал null.

Я все же считаю реализацию собственного кэша «на всякий случай» целесообразным, тем более что это требует всего нескольких строк кода. Кроме того, собственный кэш необходим, если при использовании нашего загрузчика классов предполагается пользоваться вместо классического вызова

```
Class.forName ("Имя_класса",true,loader)
```

альтернативным вариантом:

```
loader.loadClass ("Имя_класса")
```

Здесь метод **loadClass** вызывается напрямую – и здесь уже никто кроме нас не позаботится о кэшировании однажды загруженных классов. (В отличие от этого вызов **Class.forName** всегда обращается к внутреннему кэшу, и, если для данного загрузчика loader класс name уже однажды загружался, он будет найден и возвращен в результате **forName** без обращения к **loadClass** и **findClass**.)

Хуже того, если мы не будем реализовывать кэширование в загрузчике, т.е. будем в любом случае читать класс с диска и вызывать для него метод **defineClass**. В этом случае два последовательных вызова

```
loader.loadClass ("Имя_класса")

loader.loadClass ("Имя_класса")
```

для одинакового класса приведут к низкоуровневому исключению LinkageError! Виртуальная машина не разрешает в рамках одного и того же экземпляра загрузчика дважды определять один и тот же класс, т.е. обращаться к методу defineClass. С вызовом

```
Class.forName ("Имя_класса",true,loader)
```

такую проблему пронаблюдать не удастся, даже если мы не будем реализовывать кэш.

Описанное поведение, как мне кажется, является веским аргументом, чтобы по возможности никогда, за исключением тестов, не использовать прямой вызов

```
loader.loadClass("Имя_класса")
```

отдавая предпочтение вызову

```
Class.forName("Имя_класса",true,loader)
```

или

```
Class.forName("Имя_класса",false,loader)
```

У внимательного читателя при изучении нашего загрузчика DynamicClassOverloader мог возникнуть вопрос. Если наша цель – научить загрузчик «забывать» старые версии классов, почему мы попросту не реализовали в классе DynamicClassOverloader метод invalidate:

```
public void invalidate()
{
    classesHash.clear();
}
```

очищающий наш кэш **classesHash**? Почему вместо этого мы пошли по более сложному пути: созданию новых экземпляров загрузчика?

Теперь мы можем ответить на этот вопрос. При использовании вызова

```
Class.forName("Имя_класса",true,loader)
```

метод **invalidate**, очищающий **classesHash**, просто не дал бы никакого эффекта. Виртуальная машина все равно извлекала бы класс из внутреннего кэша, пока мы не сменили бы экземпляр loader используемого загрузчика. А при использовании прямого вызова

```
loader.loadClass("Имя_класса")
```

метод invalidate, вместо ожидаемого «забывания» старых классов, привел бы к низкоуровневому исключению LinkageError.

Применения DynamicClassOverloader

Приведем несколько примеров ситуаций, где можно применить созданный нами загрузчик DynamicClassOverloader помимо решения поставленной нами главной задачи – загрузки измененных версий классов «на лету» без остановки основной программы.

Прежде всего напрашивается простейшее применение: профайлинг загрузки классов. С помощью параметра «-verbose:class» стандартной утилиты «java» можно проследить, какие классы загружаются в процессе работы. Но собственный загрузчик может сделать намного больше. Очень легко дополнить приведенный выше исходный код DynamicClassOverloader сбором любой статистики о загружаемых файлах: сколько классов различных пакетов загружается, сколько времени расходуется на загрузку классов, какие классы загружаются обычно в первую очередь и т. д.

DynamicClassOverloader можно использовать для загрузки классов из нестандартного каталога, не указанного в переменной CLASSPATH. Это требуется не так уж часто, но в некоторых ситуациях без этого сложно обойтись. Например, предположим, что вы пишете систему обработки документов, к которым могут прилагаться специальные Java-классы: как апплеты в веб-страницах или скрипты в документах Microsoft Word. Пользователь может работать с любым числом таких документов, расположенных где угодно в пределах своей файловой системы. Очевидно, для загрузки и исполнения таких классов, сопутствующих документу, стандартный набор путей из CLASSPATH окажется недостаточным.

Наконец, отдельные пространства имен, порождаемые экземплярами нашего загрузчика, из неудобного недостатка могут превратиться в ключевое достоинство. Обычно разработчики Java-классов избегают конфликтов имен своих классов благодаря стандартной системе именования пакетов, когда в имя пакета включается серия вложенных доменов Internet, соответствующих уникальному веб-сайту разработчика. Но если Java применяется в упрощенном виде – скажем, для реализации небольших скриптов конечными пользователями продукта – такая схема именования может оказаться чересчур обременительной. (Те же апплеты редко размещают в пакете. Часто пакет вообще не указывается, т.е. используется корневой package.)

Представленный загрузчик позволяет надежно изолировать друг от друга подобные простые классы, не нуждающиеся во взаимодействии друг с другом и разработанные, возможно, разными разработчиками. Если два класса загружены разными экземплярами DynamicClassOverloader, они могут спокойно иметь идентичное имя. Они «живут» в разных пространствах имен и не могут ничего «знать» друг о друге.

Заключение

Реализовав собственный загрузчик классов, мы познакомились с миром отражений Java и много узнали о «внутренней кухне» виртуальной Java-машины. Конечно, это далеко не все.

Здесь не представлено исчерпывающее руководство, а только предложена краткая экскурсия, позволяющая познакомиться с основными понятиями, технологиями и проблемами мира отражений Java и, надеюсь, позволяющую почувствовать изящество и мощь этого чудесного языка. С помощью всего лишь около сотни строк исходного текста Java был создан инструмент, позволивший существенно «подправить» поведение Java-машины и изменивший

свойства самого языка Java.

Первоисточник статьи http://samag.ru/archive/article/68

Наверх