Классификация и функции загрузчиков классов

**Введение**

Программы на Java компилируются (*командой “****javac ClassName.java****” в командной строке)* в байт-код, который, как правило, хранится в файлах с расширением “.class”. Байт-код потом передается **ClassLoader-ом** Виртуальной машине (Java Virtual Machine - JVM). JVM считывает команды, представленные в class-файлах в виде байт-кода и транслирует их в виде инструкций процессору.

    Все классы в Java загружаются с помощью загрузчиков классов.  Вначале работы программы создается 3 основных загрузчика классов:

1. базовый загрузчик (bootstrap)
2. загрузчик расширений (extention)
3. системный загрузчик (system/application)

Помимо основных загрузчиков классов, существует возможность создания пользовательских загрузчиков классов. О них мы поговорим позже.

Загрузчики классов являются иерархическими. Загрузчик, который загружает основные системные классы, называется **базовым** (Bootstrap или Primordial) загрузчиком классов. Именно он загружает внутренние классы JDK  и пакеты java.\* (rt.jar и i18n.jar) . Важно заметить, что базовый загрузчик является «Изначальным или Корневым» и частью JVM, вследствие чего его нельзя создать внутри кода программы.

Тогда зачем же нужны остальные загрузчики, если базовый загрузчик и так неплохо выполняет свою работу? Зачем понадобилось разбивать процедуру загрузки классов на несколько этапов? Чтобы ответить на этот вопрос, нужно рассмотреть остальные загрузчики классов и их взаимодействие.

Итак,  **загрузчик расширений** – загружает различные пакеты расширений, которые располагаются в директории <JAVA\_HOME>/lib/ext или другой директории, описанной в системном параметре java.ext.dirs. Это позволяет обновлять и добавлять новые расширения без необходимости модифицировать настройки используемых приложений. Загрузчик расширений реализован классом sun.misc.Launcher$ExtClassLoader.

И, наконец, **системный загрузчик** – загружает классы, пути к которым указаны в переменной окружения CLASSPATH или пути, которые указаны в командном рядке после ключей  –classpath или  –cp. Системный загрузчик реализован классом sun.misc.Launcher$AppClassLoader.

**Итак:**

Когда происходит вызов загрузки какого-либо класса:

1. Происходит поиск этого класса в кэше уже загруженных классов текущего загрузчика. Если желаемый класс еще не загружался ранее, по принципу делегирования управление передается родительскому загрузчику, который находится по иерархии на уровень выше.
2. Родительский загрузчик также пытается найти желаемый класс у себя в кэше. Если класс уже был загружен и загрузчик знает о его местонахождении, то будет возвращен объект Class этого класса. Если нет, поиск будет продолжаться до тех пор, пока не дойдет до базового загрузчика.
3. Если и в базовом загрузчике нет информации об искомом классе (т.е. он еще не был загружен), будет выполнен поиск байт-кода этого класса по расположению классов, о котором знает данный загрузчик, и, если загрузить класс не удастся, управление вернется обратно загрузчику-потомку.
4. Загрузчик-потомок будет пытаться выполнить загрузку из известных ему источников.

Как уже упоминалось выше, расположение классов для базового загрузчика это библиотека rt.jar, для загрузчика расширений – каталог с расширениями jre/lib/ext, для системного – CLASSPATH, для пользовательского это может быть что-то свое.

Когда байт-код класса найден, происходит загрузка класса в JVM и получение экземпляра типа Class.

**Принцип работы загрузчиков классов**

    Каждый загрузчик классов (кроме Bootstrap) имеет родительский загрузчик, и в большинстве случаев он запрашивает родительского загрузчика загрузить указанный класс, перед тем как попробовать загрузить его самостоятельно.

    Существует так же **явный  способ инициировать загрузку** требуемого класса. Явное инициирование выполняться с помощью методов ClassLoader.loadClass() или Class.forName(). Например явное инициирование используется при загрузке JDBC драйверов: Class.forName("oracle.jdbc.driver.OracleDriver");

Иерархия загрузчиков классов выглядит следующим образом:

|  |
| --- |
| 1 Bootstrap |
| 2 Extensions |
| 3 Application / Пользовательский (если существует) |
| 4 Следующий Пользовательский (если существует) |

Что бы получить загрузчик класса в коде, нужно воспользоваться методом [getClassLoader()](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Object.html" \l "getClassLoader%28%29), например:

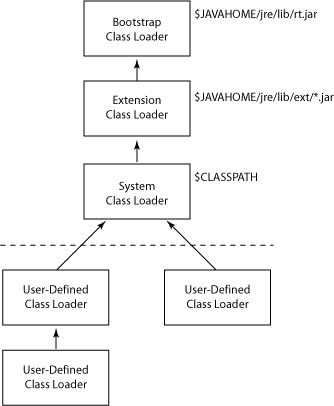
**public** **class** ClassLoadersTest {  
  
    **public** **static** void main(**String**[] args){  
  
        **Integer** i = 23;  
  
        **System**.out.println(i.getClass().getClassLoader());  
  
    }  
  
}

    Вызов i.getClass().getClassLoader() вернет null, что свидетельствует о том, что класс был загружен именно базовым загрузчиком.

    Давайте рассмотрим процесс загрузки классов более детально. Допустим, у нас есть некий класс, который мы будем загружать (например Integer). Процесс загрузки будет следующим:

1. **Cистемный загрузчик** (sun.misc.Launcher$AppClassLoader) проверит, не загружался ли данный класс ранее. Если он уже загружался, то возвращается данный класс из кэша. Если нет,  системный загрузчик делегирует поиск класса родительскому классу-загрузчику.
2. **Загрузчик расширений** (sun.misc.Launcher$ExtClassLoader) выполняет такую же процедуру
3. Наконец, **базовый загрузчик**(bootstrap), загружает класс Integer самостоятельно, поскольку у него нет родительского класса.

    Таким образом, процесс загрузки имеет одно **важное свойство, а именно делегирование** (рисунок 1). Это позволяет загружать классы тем загрузчиком, который находится ближе всего к базовому в иерархии делегирования. Как следствие поиск классов будет происходить в источниках в порядке их доверия: сначала в библиотеке core API,  потом в папке расширений, потом в локальных файлах classpath.



URLClassLoader()

Plugin

Class Loader

    Другими словами, если бы вы скачали стороннюю библиотеку с классом Integer и указали ее в переменной пути, то загрузился бы все равно оригинальный Integer(целое число).

**Еще одно важное свойство -**каждый загрузчик имеет свое пространство имен для создаваемых классов. Т.е. если классы одинаковы и находятся в одном пакете, но загружаются разными загрузчиками - они считаются разными. Таким образом, можно например, создать два объекта синглтона, если постараться :)

**Пользовательские загрузчики классов**

    В Java существует возможность создания собственных загрузчиков классов. Это может быть полезно, когда нет возможности или нежелательно перечислять все используемые библиотеки при старте программы в CLASSPATH. Например, в программе должна быть возможность динамической загрузки плагинов. Или возможностей стандартного загрузчика недостаточно для загрузки нужных классов.

    Собственные загрузчики классов используют все серверы приложений и web-контейнеры, что и понятно – приложения, разворачиваемые на сервере приложений, должны загружаться динамически, в противном случае перечисление в переменной CLASSPATH всех библиотек, используемых приложениями, становится задачей нетривиальной.

    За создание пользовательских загрузчиков классов отвечает класс  ClassLoader. Для того, что бы создать собственный загрузчик классов, необходимо унаследоваться от класса [ClassLoader](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ClassLoader.html) .

   Процесс создания собственного загрузчика хорошо описан в статье: <http://samolisov.blogspot.com/2008/01/java.html>

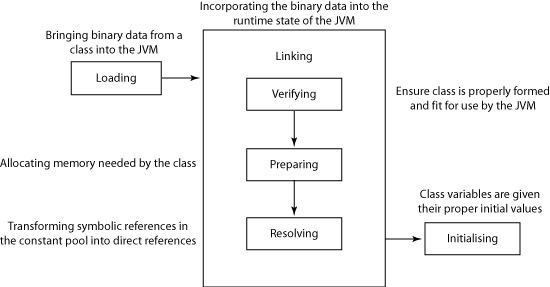
**Процесс загрузки класса более детально**

Процесс загрузки класса состоит из трех частей:

1.     Loading

2.     Linking

3.     Initialization



**Loading** – на этой фазе происходит поиск и физическая загрузка файла класса в определенном источнике (в зависимости от загрузчика). Этот процесс определяет базовое представление класса в памяти. На этом этапе такие понятия как методы, поля и т.д. пока не известны.

**Linking** – процесс, который может быть разбит на 3 части:

1. Bytecode verification – происходит несколько проверок байт-кода  на соответствие ряду зачастую нетривиальных требований определенных в спецификации JVM (http://java.sun.com/docs/books/vmspec/).
2. Class preparation – на этом этапе происходит подготовки структуры данных, отображающей поля, методы и реализованные интерфейсы, которые определены в классе + выделение памяти.
3. Resolving – разрешение все классов, которые ссылаются на текущий класс.

**Initialization** – происходит выполнение статических инициализаторов определенных в классе. Таким образом, статические поля инициализируются стандартными значениями.

    Детально о фазах загрузки:

Тип загрузки, связывания и инициализации

Виртуальная машина Java делает типы доступными для работающей программы в процессе *загрузки*, *компоновки* и *инициализации*.

Загрузка - это процесс переноса двоичной формы для типа в виртуальную машину Java.

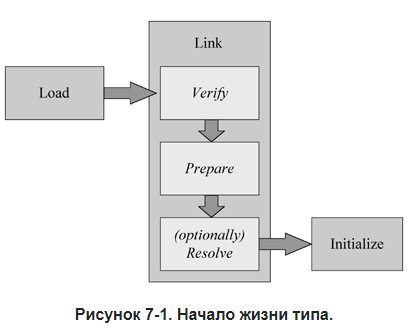
Связывание - это процесс включения данных двоичного типа в состояние выполнения виртуальной машины. Связывание делится на три подэтапа: *проверка* , *подготовка* и *разрешение.*

- *проверка* гарантирует, что тип правильно сформирован и пригоден для использования виртуальной машиной Java.

- *подготовка* включает выделение памяти, необходимой типу, например, памяти для любых переменных класса.

- *разрешение* - это процесс преобразования символьных ссылок в пуле Констант в прямые ссылки. Реализации могут задерживать этап разрешения до тех пор, пока каждая символическая ссылка фактически не будет использована запущенной программой. После завершения проверки, подготовки и (необязательно) разрешения тип готов к инициализации.

Инициализация - переменные класса получают правильные начальные значения. Смотрите рисунок 7-1 для графического изображения этого процесса.

  
  
**Рисунок 7-1. Начало жизни типа.**

Как видно из рисунка 7-1, процессы (1) загрузки, (2) связывания и (3) инициализации должны происходить в этом порядке. Единственным исключением из этого обязательного порядка является третья фаза связывания, разрешения, которая может по желанию иметь место после инициализации.

Спецификация виртуальной машины Java предоставляет реализациям гибкость во времени загрузки класса и интерфейса и связывания, но строго определяет время инициализации. Все реализации должны инициализировать каждый класс или интерфейс при первом *активном использовании* (следующие шесть ситуаций квалифицируются как активные использования):

* Создается новый экземпляр класса (в байт-кодах - выполнение new инструкции. Альтернативно, посредством неявного создания, отражения, клонирования или десериализации.)
* Вызов статического метода, объявленного классом (в байт-кодах, выполнение invokestaticинструкции)
* Использование или присвоение статического поля, объявленного классом или интерфейсом, за исключением статических полей, которые finalинициализируются константным выражением времени компиляции (в байтовых кодах, выполнение инструкции getstaticили putstatic)
* Вызов определенных отражающих методов в Java API, таких как методы в классе Class или в классах в java.lang.reflect пакете
* Инициализация подкласса класса (Инициализация класса требует предварительной инициализации его суперкласса.)
* Обозначение класса в качестве исходного класса (с main()<методом) при запуске виртуальной машины Java

Все другие виды использования типа, кроме шести, перечисленных здесь, являются *пассивными*, которые не вызывают инициализацию типа. Несколько примеров, иллюстрирующих разницу между активным и пассивным использованием, приведены ниже в этой главе.

Как упоминалось в предыдущем списке, инициализация класса требует предварительной инициализации его суперкласса. Применяемое рекурсивно, это правило означает, что все суперклассы класса должны быть инициализированы до инициализации класса. Однако это не относится к интерфейсам. Интерфейс инициализируется только потому, что используется непостоянное поле, объявленное интерфейсом, а не потому, что необходимо инициализировать подинтерфейс или класс, реализующий интерфейс. Таким образом, инициализация класса требует предварительной инициализации всех его суперклассов, но не его суперинтерфейсов. Инициализация интерфейса не требует инициализации его суперинтерфейсов.

Правило «инициализировать при первом активном использовании» управляет механизмом, который загружает, связывает и инициализирует классы. При первом активном использовании тип должен быть инициализирован. Однако, прежде чем он может быть инициализирован, он должен быть связан. И прежде чем он может быть связан, он должен быть загружен. По их выбору реализации могут загружать и связывать типы рано. Им не нужно ждать первого активного использования типа для загрузки и связывания типа. Однако, если тип не был загружен и связан до его первого активного использования, он должен быть загружен и связан в это время, чтобы его можно было инициализировать.

**Исключения**

При работе загрузчиков классов наиболее часто встречаются следующие исключительные ситуации:

1.  **ClassNotFoundException**, бросается, когда приложение пытается загрузиться класс  по его названию (String) с помощью таких средств:

* forName метод в классе Class.
* findSystemClass метод в классе ClassLoader.
* loadClass метод в классе ClassLoader.

Но класса с таким именем  не существует.

2.  **NoClassDefFoundError**, бросается в таких случаях:

* Когда архив, директория, или другой источник необходимых классов не был добавлен в источники текущего загрузчика классов или его предка.
* Загрузчик-предок не был установлен корректно.

    Иногда проблемы, связанные с загрузкой класса, проявляются не только на этапе загрузки, но и на этапе использования класса.

**Выгрузка классов**

    В большинстве случаев, жизненный цикл класса в виртуальной машине схож с жизненным циклом объекта. JVM загружает, связывает и инициализирует классы, позволяя программе пользоваться ими, и выгружает, когда в приложении они более не используется. Важно заметить, что выгрузка классов не работает в том случае, если класс был загружен Bootstrap загрузчиком.

    Загруженные классы, несмотря на то, что являются полноценными Java-объектами, хранятся в особой системной области памяти, называемой permament generation (сокращенно, PermGen) и управляемой сборщиком мусора.

    Выгрузка классов является важной частью механизма работы JVM, поскольку Java программы могут динамически расширяться во время работы, загружая пользовательские классы и, тем самым, занимать много места в оперативной памяти. Держать классы в памяти, которые больше не будут использоваться, нет никакого смысла.

Конкретная политика выгрузки классов во многом зависит от реализации виртуальной машины JVM.

**Заключение**

    Таким образом, мы с вами немного приблизились к пониманию процесса загрузки классов в JVM. Были рассмотрены типы загрузчиков, их иерархия, фазы загрузки класса и исключительные ситуации, которые могут возникнуть в процессе.