Инструментирование в Java

Теория и примеры



[Валентин Коваленко](https://sites.google.com/site/aboutmale/), 2012

# Содержание

[Содержание](#_87qzw7ti6xa)

[О статье](#_u2p67bk7a2b7)

[Запуск JVM и загрузка классов](#_4aipa55jm035)

[Загрузчики классов](#_qbktor8af88w)

[Первичный загрузчик](#_tusqd9sa49rx)

[Загрузчик расширений](#_vaipns5097lj)

[Системный загрузчик](#_rpatozxygiad)

[Модель делегации загрузки классов](#_yzc9qoeb0bn)

[Трансформация классов](#_l1uoi1jko6jc)

[Java-агент](#_1d2wia8c7atu)

[Интерфейс java.lang.instrument.Instrumentation](#_ktnkxg6kwi21)

[Трансформация класса перед его определением](#_a5xfitki8d7n)

[Трансформация уже определённого класса](#_fcxa8umz8fqw)

[Манипулирование байт-кодом и библиотека Javassist](#_4fjf51lkturl)

[Примеры](#_ejvr0y1jks0r)

[Структура файлов в примерах](#_tw23c9lfx5as)

[Сборка примеров](#_p51o9wd53qb5)

[Запуск примеров](#_2qmxyiv79w1j)

[Пример A - трансформация класса: замена обращения к полю класса](#_49sx40uog12t)

[Пример B - трансформация класса: замена вызова конструктора класса, создание CGLib-прокси](#_6rvu2rubowpz)

[Пример С - трансформация класса: замена возвращаемого значения метода, создание Java-прокси](#_fi4211yyn5cj)

[Пример D - переопределение класса: трансформация уже загруженного класса java.lang.String](#_ssb8zxv8l3o1)

[Лицензионное соглашение с конечным пользователем](#_fm8gdm4lt75)

[Литература](#_9zajd4rr6niy)

# О статье

В статье рассказывается о том что такое инструментирование и трансформация классов в Java, какие механизмы существуют в Java Platform SE для инструментирования, рассматриваются примеры трансформации классов. Все примеры готовы к запуску и вам не потребуется ничего кроме установленной JRE для их запуска.

Статья и примеры рассчитаны на читателя, который знаком с языком Java и представляет что такое JRE, но при этом он может совершенно не иметь представления о том что такое инструментирование. Целью статьи являтся агрегирование основных сведений, чтобы даже полностью незнакомый с инструментированием классов человек, после прочтения и изучения примеров, мог самостоятельно работать в этом направлении.

Всё написанное в статье справедливо для Java Platform SE 5, за исключением явно оговариваемых случаев, когда приводятся факты касающиеся других версий.

Все примеры запускались в JRE 5 (и JRE 9) с Java HotSpot VM. Каждый раз, когда в тексте встречается "JRE\*", подразумевается JRE 5 с Java HotSpot VM, а под "JVM\*" подразумевается Java HotSpot VM входящая в состав JRE 5. Дело в том, что некоторые возможности и факты могут быть специфичны для конкретной реализации JRE, потому в некоторых случаях нужно понимать о какой реализации JRE идёт речь.

Домашняя страница проекта, содержащего статью и примеры: <https://github.com/stIncMale/java-instrumentation>.

# Запуск JVM и загрузка классов

Код любого Java-приложения исполняется в JVM. Существует как минимум два способа запустить JVM и сообщить какой код нужно выполнить:

* указав класс, или пакет, содержащий публичный статический метод main(String[]), например

java some/package/Main

java some.package.Main

java -jar main.jar (в этом случае манифест JAR-файла должен содержать атрибут указывающий класс в котором определён метод main(String[])).

* определив Java-агент, например

java -javaagent:agent.jar (в этом случае манифест JAR-файла должен содержать атрибут указывающий класс агента; подробнее о Java-агенте рассказано в разделе [Java-агент](#odzn55ze3opc), а о том что такое манифест JAR-файла можно прочитать в [[4]](#30dkt13auuyb)).

## Загрузчики классов

В JVM используется механизм ленивой загрузки классов, т.е. классы загружаются только тогда, когда они потребовались. Когда запускается процесс Java, и указывается класс содержащий код для исполнения, то машинный код, составляющий процесс Java, пробует загрузить указанный Java-класс, чтобы в итоге запустить его метод main(...), или метод premain(...) Java-агента. При этом ему потребуется загрузить также связанные классы: родительский класс или интерфейс, классы, использующиеся этим классом и т.д. Загрузка классов в JVM осуществляется при помощи специальных загрузчиков которые, за исключением одного загрузчика, написаны на языке Java и являются наследниками абстрактного класса ClassLoader. Чтобы загрузчик мог выполнить загрузку Java-класса, он сам, и все классы от которых он зависит, предварительно должны быть загружены в JVM. Таким образом ясно, что должен существовать какой-то особый загрузчик, который не нуждается в собственной загрузке через механизм загрузки классов в JVM. Такой загрузчик существует, он написан не на языке Java, является частью JVM и представлен машинным кодом. Этот загрузчик называется первичным загрузчиком (bootstrap class loader). Первичный загрузчик является не единственным стандартным загрузчиком, кроме него есть ещё два стандартных загрузчика: загрузчик расширений (extension class loader) и системный загрузчик (system class loader, или application class loader).

### Первичный загрузчик

Как уже было сказано, это единственный стандартный загрузчик, который реализован полностью не на языке Java. Его основной целью служит первичная загрузка Java-классов загрузчиков, написанных на языке Java, т.к. сами себя они загрузить не могут. JRE\* позволяет узнать откуда первичный загрузчик производит загрузку классов, в нём есть нестандартное системное свойство sun.boot.class.path, определяющее пути для загрузки классов первичным загрузчиком (нестандартное значит что в другой реализации JRE аналогичного свойства может не быть, или оно может называться иначе). Это свойство можно задать используя нестандартный параметр запуска JVM\* -Xbootclasspath. Если выполнить

|  |
| --- |
| System.out.println(System.getProperty("sun.boot.class.path")); |

то получим примерно следующее:

|  |
| --- |
| ${JAVA\_HOME}/jre/lib/rt.jar;  ${JAVA\_HOME}/jre/lib/i18n.jar;  ${JAVA\_HOME}/jre/lib/sunrsasign.jar;  ${JAVA\_HOME}/jre/lib/jsse.jar;  ${JAVA\_HOME}/jre/lib/jce.jar;  ${JAVA\_HOME}/jre/lib/charsets.jar;  ${JAVA\_HOME}/jre/classes |

Видно, что тут присутствует директория ${JAVA\_HOME}/jre/classes, которую вы скорее всего не увидите у себя. Но если её создать и поместить туда свои классы, они будут загружены первичным загрузчиком без каких-либо дополнительных манипуляций с вашей стороны.

### Загрузчик расширений

Кроме первичного загрузчика в JRE есть ещё один загрузчик, целью которого не является загрузка классов приложения - это загрузчик расширений (extension class loader), предназначенный для загрузки классов относящихся к расширениям Java. Механизм расширений в Java предоставляет стандартный способ сделать определённый API видимым для нескольких приложений, т.е. приложениям не нужно "заботиться" о том, откуда грузить классы, являющиеся частью расширения. Классы расширений "видны" приложению так же прозрачно, как классы входящие в Java Platform, в этом смысле классы расширений похожи на классы Java Platform. Системное свойство java.ext.dirs определяет пути для загрузки классов extension загрузчиком. Если выполнить

|  |
| --- |
| System.out.println(System.getProperty("java.ext.dirs")); |

то получим примерно следующее:

|  |
| --- |
| ${JAVA\_HOME}/jre/lib/ext |

Кроме того, начиная с Java Platform SE 6, загрузчик расширений может грузить классы ещё и из специфической для конкретной операционной системы директории. Например для Oracle Solaris это /usr/jdk/packages/lib/ext, а для MS Windows - %WINDIR%\Sun\Java\lib\ext. Значение свойства java.ext.dirs может быть переопределено через параметр запуска JVM -Djava.ext.dirs.

Расширения, о которых сейчас шла речь, называются установленными расширениями (installed extensions). Бывает и другая разновидность расширений - скачиваемые расширения (download extensions). Такие расширения описываются в манифестах JAR-файлов, которые эти расширения используют. Скачиваемые расширения называются расширениями потому, что могут быть скачаны и загружены например извне операционной системы, в которой запущена JVM, без дополнительных усилий со стороны приложения. В отличие от установленных расширений, классы скачиваемых расширений загружаются не загрузчиком расширений, а системным загрузчиком, как и классы приложения, и "видны" только классам того приложения, которое их загрузило. Подробнее о расширениях можно прочитать в [[3]](#ojnfbxj7goic).

### Системный загрузчик

Системный загрузчик служит для загрузки классов приложения и для загрузки скачиваемых расширений. Как и в случае двух описанных выше загрузчиков, при запуске JVM существует возможность задать пути (JAR/ZIP-файлы, или директории содержащие файлы Java-классы), откуда системный загрузчик может грузить классы. Это делается через переменную окружения CLASSPATH, или параметры запуска JVM -classpath/-cp. Эти параметры являются синонимами, а их действие полностью идентично действию переменной окружения CLASSPATH: они переопределяют пути загрузки классов системным загрузчиком. По умолчанию среди таких путей есть только рабочая директория "." процесса java. Тут стоит сделать одно неочевидное замечание: если при запуске JVM использовался агрумент -jar, то переменная окружения CLASSPATH и параметры запуска -classpath/-cp будут проигнорированы. При использовании параметра -jar, пути загрузки классов системным загрузчиком должны, при необходимости, переопределяться через атрибут Class-Path манифеста указанного JAR-файла. В случае использования CLASSPATH, или -classpath/-cp, можно получить заданные через них пути из системного свойства java.class.path:

|  |
| --- |
| System.out.println(System.getProperty("java.class.path")); |

## Модель делегации загрузки классов

Загрузчики классов состоят в иерархических отношениях друг с другом и образуют направленное дерево (не нужно путать эту иерархию с иерархией наследования Java-классов загрузчиков). Корнем дерева является первичный загрузчик, загрузчик расширений является ребёнком первичного загрузчика, а системный загрузчик является ребёнком загрузчика расширений. Любые другие загрузчики обычно являются потомками системного загрузчика, хотя возможно создать загрузчик, который будет ребёнком первичного загрузчика, или загрузчика расширений.

Загрузка любого класса, кроме классов для которых первичный загрузчик является инициирующим (см. [шаг 4](#ha215rww817g) алгоритма делегации загрузки классов), происходит в результате вызова одного из методов loadClass(...) определённых в классе ClassLoader. Алгоритм делегации загрузки классов выглядит следующим образом (см. также [рис. 1](#dqjjpb4o1i63)):

1. загрузчик, от которого первоначально потребовалась загрузка класса, становится текущим загрузчиком
2. проверяется не отмечен ли в JVM текущий загрузчик в качестве инициирующего (см. [шаг 4](#ha215rww817g)) загрузчика требуемого класса
   1. если результат проверки положительный (т.е. требуемый класс уже загружен), то результатом первоначально инициированной загрузки становится загруженный ранее класс
   2. иначе проверяется есть ли у текущего загрузчика родительский загрузчик
      1. если результат проверки положительный, то родительский загрузчик становится текущим и алгоритм переходит к [шагу 2](#gfqjdrpuil3h) (загрузка делегируется родительскому загрузчику)
      2. иначе (т.е. текущий загрузчик не имеет родителя) текущим загрузчиком является первичный загрузчик, и алгоритм переходит к [шагу 3](#64hzd9kbl2tq)
3. текущий загрузчик пытается самостоятельно загрузить класс
   1. проверяется удалось ли текущему загрузчику успешно загрузить класс
      1. если удалось, то алгоритм переходит к [шагу 4](#ha215rww817g), а текущий загрузчик становится определяющим (defining classloader) для загруженного класса
      2. иначе проверяется является ли текущий загрузчик тем загрузчиком, от которого первоначально потребовалось загрузить класс?
         1. если результат проверки положительный, то результатом первоначально инициированной загрузки становится выбрасывание ClassNotFoundException
         2. иначе текущим загрузчиком становится дочерний загрузчик текущего загрузчика, из которого ранее ([шаг 2Bi](#tjuugu64bl5d)) была произведена делегация загрузки, и алгоритм переходит к [шагу 3](#64hzd9kbl2tq)
4. результатом первоначально инициированной загрузки становится загруженный класс, а загрузчик, от которого первоначально потребовалась загрузка класса, отмечается в JVM как инициирующий загрузчик (initiating class loader) загруженного класса

В большинстве случаев загрузчики должны реализовывать модель делегации загрузки классов, но не всегда она является подходящей, и фактически не существует механизма, который бы заставлял любой загрузчик следовать описанному алгоритму. Первичный загрузчик, загрузчик расширений и системный загрузчик реализуют описанную модель делегации загрузки.

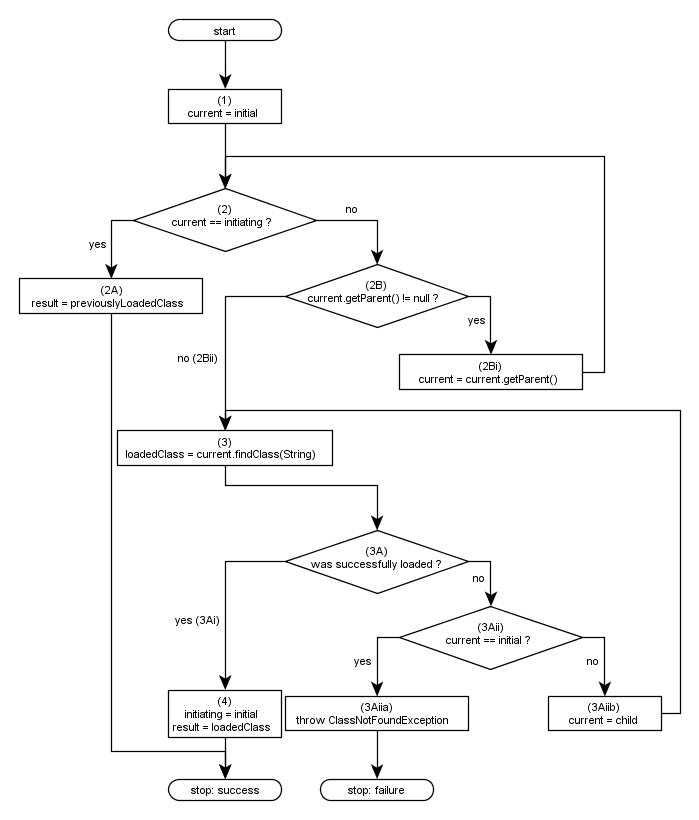


рис. 1, блок-схема алгоритма делегации загрузки классов в Java

В заключение несколько интересных и полезных фактов:

1. Можно сказать, что каждый определяющий загрузчик формирует отдельное пространство имён классов, которые были им определены, так что во время выполнения загруженный класс идентифицируется своим бинарным именем (binary name) и определяющим загрузчиком ([шаг 3Ai](#yalzx1unc1nk) в [алгоритме делегации загрузки](#gi214pqt0qcc)). А раз так, то из одного класс файла можно загрузить несколько разных классов в рамках одной работающей JVM, если использовать разные загрузчики.
2. В JRE\* классы, чьё бинарное имя начинается с java., могут быть загружены только первичным загрузчиком. Например при попытке загрузить пользовательский класс java.hack.Cls, будет брошено исключение: Prohibited package name: java.hack at java.lang.ClassLoader.preDefineClass. Классы с именем начинающимся на javax. не являются частью Java Platform SE и могут быть загружены другими загрузчиками. Это значит, что если написать свой класс, имя которого начинается на javax., то он сможет быть загружен в JVM без каких-либо ограничений.
3. Если загрузка незагруженного класса X вызвана необходимостью загрузки другого класса C (например класс C использует класс X), или загрузка класса X вызвана вызовом метода Class.forName(String) из кода класса C, то для загрузки класса X будет использоваться определяющий загрузчик класса C. Нужно понимать, что в такой ситуации определяющий загрузчик класса С будет инициирующим загрузчиком ([шаг 4](#ha215rww817g) в [алгоритме делегации загрузки](#gi214pqt0qcc)) класса Х, но при этом он не обязательно будет определяющим загрузчиком класса Х.
4. При создании загрузчика конструктором ClassLoader(), его родительским загрузчиком будет системный загрузчик. Если требуется указать другой загрузчик в качестве родительского, можно использовать конструктор ClassLoader(ClassLoader). Чтобы указать первичный загрузчик в качестве родителя, нужно передать null в конструктор загрузчика.
5. Ссылку на системный загрузчик можно получить используя метод ClassLoader.getSystemClassLoader().
6. Ссылку на инициирующий загрузчик ([шаг 4](#ha215rww817g) в [алгоритме делегации загрузки](#gi214pqt0qcc)), можно получить используя метод Class.getClassLoader(). [VAKOTODO do we need a bookmark here? Check all bookmarks]
7. Не нужно путать инициализацию и загрузку класса. Вполне возможна ситуация когда класс был загружен, но так и не был проинициализирован за всё время работы JVM. Подробнее о инициализации классов и интерфейсов можно прочитать в [[8]](#r3bfu79z68bh).

Хочется ещё отметить прекрасные статьи [[1]](#i8nzt2srd140) и [[2]](#sy8muka6qvz6), которые я очень рекомендую прочитать тем, кто хочет лучше понять что такое загрузчики классов.

# Трансформация классов

В Java Platform SE 5 существует возможность инструментирования Java-классов. Инструментирование - это обычно добавление байт-кода в уже существующий байт-код методов, например для сбора информации, или для выполнения каких-то дополнительных действий непосредственно перед, или после действий оригинального метода. Инструментирование может использоваться профайлерами, логгерами, мониторящими агентами и т.д. Однако технически средства, присутствующие в платформе, имеют более широкие возможности: можно не только добавлять новый байт-код, но и менять уже существующий (в некоторых случаях есть ограничения на возможные изменения кода). Любое изменение байт-кода класса в рантайме будем называть трансформацией класса.

Трансформация может быть осуществлена

* перед определением класса в JVM. Ограничений на трансформацию байт-кода не накладывается, но имя класса изменить нельзя.
* в отношении уже загруженного и определённого класса. В этом случае существуют ограничения. Такую трансформацию ещё называют "горячей" заменой классов, или переопределением (hotswap class replacement, или redefinition).

Существует два механизма трансформации байт-кода:

* высокоуровневый механизм, работа с которым возможна с использованием языка Java. Он является частью Java Platform SE и представлен программным интерфейсом Java-пакета java.lang.instrument.
* низкоуровневый механизм, работа с которым возможна с использованием любого низкоуровневого языка, который поддерживает модель вызовов языка С и определения из языков C/C++ (*any native language that supports C language calling conventions and C or C++ definitions* - цитата из спецификации JVM TI, входящей в [[10]](#o9we5svgyhb)). Представлен программным интерфейсом JVM Tools Interface (JVM TI). JVM TI является низкоуровневым API, которое предоставляется Java Platform Debugger Architecture (JPDA). Над JVM TI есть высокоуровневый интерфейс Java Debug Interface (JDI), представленный программными интерфейсами Java-пакетов с именами начинающимися на com.sun.jdi. Отмечу что в спецификации JVM TI сказано, что JVM TI может отсутствовать в некоторых JVM.

JVM TI появился раньше чем пакет java.lang.instrument и имеет значительно более широкие возможности, но когда требуется только функциональность трансформации классов, API представленный в пакете java.lang.instrument является альтернативой, которую могут легко использовать Java программисты.

Перед тем как говорить о трансформации классов при помощи пакета java.lang.instrument, необходимо рассказать о том что такое Java-агент.

## Java-агент

Java-агент - это программа, написанная на языке Java, которая может быть запущена перед основной программой, содержащей метод main(String[]). Класс Java-агента загружается системным загрузчиком и всегда выполняется в той же JVM в которой выполняется основная программа.

В Java Platform SE 5 для запуска Java-агента нужно использовать параметр запуска JVM -javaagent:, указав архив, содержащий Java-класс агента и манифест со специальными атрибутами. Опционально можно указать параметры агента, например:

|  |
| --- |
| java -javaagent:agent.jar=some\_arguments\_as\_string -jar main.jar |

Параметры агента должны быть указаны одной строкой без пустых символов. Задача интерпретации строки параметров лежит на самом агенте. Можно определить несколько агентов, указав несколько -javaagent: аргументов. Кроме того, один и тот же архив агента может использоваться разными агентами, которым например передаются разные аргументы.

Класс агента должен определять метод public static void premain(String agentArgs, Instrumentation inst), аналогичный методу main(String[] args). После старта JVM, методы premain(String, Instrumentation) агентов начинают вызываться последовательно в том порядке, в котором агенты были определены параметрами -javaagent:. В смысле своих возможностей, Java-агент является полноценной Java-программой, и метод premain(String, Instrumentation) может выполнять любые действия, которые может выполнять метод main(String[]).

Манифест ахива агента может содержать следующие специальные атрибуты:

* Premain-Class - обязательный атрибут, определяющий имя класса агента
* Boot-Class-Path - опциональный атрибут, позволяющий добавить новые пути для поиска классов первичным загрузчиком (случай использования этого атрибута представлен в [примере "D"](#19i20kq3nykr))
* Can-Redefine-Classes - опциональный атрибут, задающий возможность переопределения классов, т.е. трансформации уже определённых классов (случай использования этого атрибута представлен в [примере "D"](#19i20kq3nykr))

Отмечу что в Java Platform SE 6 была добавлена возможность запуска Java-агента после того, как метод main(String[]) был вызван. В связи с этим, у класса агента появились новые специальные методы, кроме метода premain(String, Instrumentation), и были добавлены новые специальные атрибуты для манифеста архива агента.

Подробнее о Java-агенте можно прочитать в API спецификации пакета java.lang.instrument в [[9]](#9eio59bdric3).

## Интерфейс java.lang.instrument.Instrumentation

Интерфейс java.lang.instrument.Instrumentation предоставляет API который позволяет осуществлять трансформацию классов. В версиях 5, 6, 7 и 8 Java Platform SE получить объект типа Instrumentation можно только через Java-агент. В Java Platform SE 5 JVM передаёт ссылку на объект типа Instrumentation как аргумент методу premain(String, Instrumentation).

## Трансформация класса перед его определением

Объект Instrumentation можно использовать для трансформации классов при помощи трансформеров классов. Трансформер классов - это объект типа ClassFileTransformer, с методом byte[] transform(ClassLoader, String, Class<?>, ProtectionDomain, byte[]). При первоначальной загрузке класса, JVM передаёт в трансформер байт-код класса, так что трансформер может внести в байт-код свои изменения и возвратить изменённый байт-код.

Чтобы трансформер мог участвовать в трансформации классов, он должен быть зарегистрирован в JVM. Регистрация трансформера осуществляется методом Instrumentation.addTransformer(ClassFileTransformer). После того как трансформер был зарегистрирован, его метод transform(...) вызывается при каждом новом определении класса и при каждом переопределении класса. Запрос на определение класса осуществляется вызовом одного из методов ClassLoader.defineClass(...), а запрос на переопределение класса осуществляется вызовом метода Instrumentation.redefineClasses(ClassDefinition[]), или аналогичного низкоуровневого метода.

Трансформеры вызываются до проверки и применения байт-кода класса. Это значит что если трансформер был зарегистрирован до того как какой-то класс был определён в JVM, он имеет возможность модифицировать байт-код этого класса, в то время как JVM ничего не знает о классе. Единственное ограничение - полное имя класса не может быть изменено.

Стоит ещё обратить внимание на обработку исключительных ситуаций, которые могут произойти при трансформации классов: в случае если метод transform(...) выбрасывает ошибку, или исключение, она будет тихо "проглочена", а трансформация будет продолжена.

Более детально о трансформерах классов можно прочитать в API спецификации интерфейсов Instrumentation и ClassFileTransformer в [[9]](#9eio59bdric3).

## Трансформация уже определённого класса

Уже определённый в JVM класс можно трансформировать путём переопределения класса. Для этого интерфейс Instrumentation предоставляет метод redefineClasses(ClassDefinition[]).

Метод redefineClasses(ClassDefinition[]) может быть неудобен в некоторых случаях. Дело в том, что конструктору ClassDefinition.ClassDefinition(Class<?>, byte[]) требуется байт-код класса, и в случае когда требуется только запустить трансформацию класса, который уже был определён, программисту всё равно придётся каким-то образом получать байт-код класса, хотя JVM может получить его самостоятельно. Поэтому в Java Platform SE 6 появился метод Instrumentation.retransformClasses(Class<?>...), позволяющий сделать то же самое, что redefineClasses(ClassDefinition[]) но без явного указания байт-кода.

В отличие от трансформации класса в процессе его первоначальной загрузки, на трансформацию уже загруженного класса накладываются следующие ограничения: нельзя добавлять/удалять/переименовывать поля/методы класса, изменять сигнатуру методов, изменять отношения наследования классов и интерфейсов. При этом можно изменять тела методов и пул констант класса. Кроме перечисленных ограничений, переопределение классов имеет ещё одно важное отличие от трансформации классов в процессе первоначальной загрузки: переопределение является опциональной возможностью JVM. Это значит что конкретная JVM может или вообще не поддерживать переопределение классов, или поддерживать его только при определённых условиях (например при указании определённых аргументов запуска JVM). Узнать о том, поддерживается ли переопределение классов, можно при помощи метода Instrumentation.isRedefineClassesSupported(). Отмечу что этот метод может вернуть false не только в случае когда имплементация JVM в принципе не поддерживает переопределение классов, но и в случае когда переопределение не было разрешено/включено при запуске JVM (см. [Can-Redefine-Classes](#jyjupnebq1s7)).

## Манипулирование байт-кодом и библиотека Javassist

Трансформация/переопределение класса подразумевают работу напрямую с массивом байт, представляющим байт-код класса, а это непросто. К счастью существуют библиотеки, облегчающие работу с байт-кодом. Такие библиотеки тоже могут предоставлять разный уровень абстракции при работе с байт-кодом, например популярная библиотека ASM является довольно низкоуровневой, а библиотека Javassist позволяет манипулировать байт-кодом описывая изменения строкой, которая содержит привычный любому разработчику Java-код. Т.о. манипулировать байт-кодом при помощи Javassist можно даже не имея представления о структуре байт-кода. По той причине, что Javassist является довольно высокоуровневой библиотекой, она и была выбрана для использования в примерах статьи.

Хорошим руководством для начинающих использовать Javassist является [[5]](#xec354wlzlxr). Дублировать это небольшое, простое и информативное руководство нет смысла, потому предваряя описание примеров, я лишь кратко опишу необходимые базовые сведение о Javassist, которые облегчат понимание примеров. Речь пойдёт только о той части библиотеки, которая касается работы с байт-кодом.

Javassist представляет пользователю Java-классы в виде объектов типа javassist.CtClass. Получать объекты CtClass можно из объектов типа javassist.ClassPool. После того как ClassPool загрузил какой-то CtClass, объект класса продолжает храниться в ClassPool так что при следующих обращениях ClassPool вернёт этот же объект. Объекты типа ClassPool могут быть организованы в иерархическую структуру аналогично загрузчикам классов. Подобная организация может понадобится в лучаях, когда классы с одинаковыми именами могут грузиться разными загрузчиками, например в случае когда на одном сервере приложений запущено несколько приложений, использующих классы с одинаковыми именами (это могут быть классы сторонних библиотек).

Манипулирование байт-кодом класса осуществляется через соответствующий ему объект CtClass. При конвертировании объекта типа CtClass в байт-код методами writeFile(...), toClass(...), или toBytecode(...), этот объект становится "замороженным", так что дальнейшее изменение его байт-кода невозможно. Если же возникает необходимость последующих изменений, класс может быть разморожен методом defrost().

Детальное описание классов Javassist можно посмотреть в [[6]](#4fza2osfu1cs).

# Примеры

Теперь, когда рассказана общая информация о возможностях и способах инструментирования Java-классов, можно перейти к примерам. Код примеров, документацию и всё необходимое для запуска примеров можно скачать [тут](http://code.google.com/p/java-instrumentation/downloads/list).

https://code.google.com/archive/p/java-instrumentation/downloads

Код примеров логически состоит из трёх частей: код для запуска примеров, непосредственно классы примеров и инструментирующий код:

|  |  |  |
| --- | --- | --- |
| Имя пакета | JAR | Описание |
| com.gl.vn.me.ko.sample.instrumentation.**env**  com.gl.vn.me.ko.sample.instrumentation.**env.misc** | main.jar | Классы составляющие окружение необходимое для запуска примеров. Интреса для читателя не представляют. |
| com.gl.vn.me.ko.sample.instrumentation.**example** | example.jar | Классы примеров. Эти классы можно рассматривать как код некоторого приложения, чьё поведение мы намереваемся поменять при помощи инструментирования. |
| com.gl.vn.me.ko.sample.instrumentation.**example.agent**  com.gl.vn.me.ko.sample.instrumentation.**example.proxy**  com.gl.vn.me.ko.sample.instrumentation.**example.transform** | agent-abc.jar  agent-d.jar | Инструментирующий код. Обратите внимание что весь инструментирующий код находится в отдельных архивах, и код приложения об инструментирующем коде ничего не "знает". |
| com.gl.vn.me.ko.sample.instrumentation.**util**  com.gl.vn.me.ko.sample.instrumentation.**util.cglib**  com.gl.vn.me.ko.sample.instrumentation.**util.javassist** | instrumentation-util.jar | Код, который мог бы быть помещён вместе с остальным инструментирующим кодом, но вынесен отдельно, т.к. может оказаться полезен не только в рамках примеров статьи. |

Код классов подробно задокументирован с использованием Javadoc-комментариев, которые нужно рассматривать как часть описания примеров. Тут я лишь кратко опишу основные классы, назначение которых нужно знать для того чтобы понимать примеры (часть com.gl.vn.me.ko.sample.instrumentation. имени классов здесь и далее будет опускаться).

|  |  |
| --- | --- |
| Класс | Краткое описание  (подробнее см. в документации к примерам: [Java Instrumentation Examples. Code Documentation](http://wiki.java-instrumentation.googlecode.com/git/content/docs/index.html)) |
| util.InstrumentationEnvironment | Позволяет сохранить объект типа Instrumentation из класса Java-агента, чтобы иметь доступ к Instrumentation из кода приложения. |
| example.agent.Agent | Является суперклассом для классов Java-агентов. Содержит общие методы, необходимые классам Java-агентов. |
| util.AbstractClassFileTransformer | Является суперклассом для трансформеров байт-кода, выполняет обработку возникающих ошибок и исключительных ситуаций (помним что JVM никаких специальных действий по обработке ошибок и исключительных ситуаций сгенерированных методом ClassFileTransformer.transform(...) не производит).  Получив имя класса, который должен быть трансформирован, AbstractClassFileTransformer находит в ClassPool соответствующий классу объект CtClass и передаёт найденный объект в метод AbstractClassFileTransformer.doTransform(CtClass), в котором происходит изменение байт-кода класса.  Таким образом байт-код класса, который передаёт JVM при вызове метода AbstractClassFileTransformer.transform(...) не используется. Потому если трансформация класса производится несколькими трансформерами, так что каждый следующий должен видеть изменения в байт-коде, сделанные предыдущим трансформером, то AbstractClassFileTransformer должен передавать один и тот же объект CtClass в метод doTransform(CtClass), т.е. все трансформеры должны работать с одним и тем же объектом ClassPool, из которого они получают объекты CtClass. Для решения этой задачи служит класс JavassistEnvironment. |
| util.javassist.JavassistEnvironment | Облегчает работу по созданию и использованию объектов ClassPool. Позволяет работать с одним и тем же объектом ClassPool из разных мест кода. |

## Структура файлов в примерах

|  |  |
| --- | --- |
| Папка | Описание |
| . | Содержит скрипт сборки примеров и генерации Javadoc-документации, который может быть выполнен при помощи инструмента Apache Ant. |
| ./doc | Содержит сгенерированную Javadoc-документацию примеров. |
| ./lib | Содержит сторонние библиотеки, необходимые для компиляции и запуска примеров, а также файлы лицензий к этим библиотекам. |
| ./src | Содержит исходный код примеров и текст файлов манифеста, которые используются при генерации JAR-файлов примеров. |

## Сборка примеров

Сборку примеров можно осуществить при помощи инструмента [Apache Ant](http://www.google.com/url?q=http%3A%2F%2Fant.apache.org%2F&sa=D&sntz=1&usg=AFQjCNFES7LK4aOUN0h6wxhH2k7ycUHGRA) и скрипта ./build.xml. Для этого нужно запустить программу ant из директории в которой находится скрипт build.xml. Возможны несколько вариантов:

|  |  |
| --- | --- |
| Вариант | Описание |
| ant bin | Компиляция и сборка кода. Создаст директорию ./build, если её нет, и поместит в неё собранные JAR-файлы c примерами. |
| ant doc | Генерация Javadoc-документации. Создаст директории ./build и ./build/docs, если их нет, и поместит в ./build/docs сгенерированную документацию. |
| ant all | Равносильно последовательному запуску ant doc, ant bin, а также просто запуску ant из директории в которой находится файл build.xml. |

## Запуск примеров

Запуск примеров осуществляется следующим образом: из директории ./build нужно запустить

|  |
| --- |
| java -jar main.jar -example d |

В этом случае будет запущен пример "D". У параметра -example есть более короткий синоним -ex. При необходимости, можно изменить детализированность информации выводимой при работе примеров, для этого существует параметр -logLevel, или короче -logl. Чтобы запустить пример "A" с уровнем логирования trace, нужно запустить:

|  |
| --- |
| java -jar main.jar -logl trace -ex a |

Возможны следующие уровни логирования: info (используется по умолчанию), debug, trace. При запуске примеров с использованием Java-агента, агенту тоже можно указать один из указанных уровней логирования:

|  |
| --- |
| java -javaagent:agent-d.jar=debug -jar main.jar -logl trace -ex d |

Вот пример вывода сообщения при работе примера:

|  |
| --- |
| >ExampleA run - counter = 0 before incrementation |

читать такое сообщение следует следующим образом:

|  |
| --- |
| >ИмяКласса имяМетода - сообщение |

Обратите внимание, что при запуске примеров не нужно задавать -classpath, указывая сторонние библиотеки из директории ./lib, т.к. все необходимые библиотеки указаны в манифестах JAR-файлов. Способ запуска JVM с параметром -jar является менее гибким по сравнению с явным указанием класса, содержащего метод main(String[]), ведь при этом нельзя переопределить пути загрузки классов системным загрузчиком (см. [выше](#b8o2ao9xjobs)), но в данном случае этот способ упрощает запуск примеров, а гибкость нам не нужна.

Описания примеров рекомендуется читать в том порядке, в каком они приведены. Если вам всё же хочется изменить порядок, всё равно начните с примера "A", т.к. в нём в том числе описано то, что одинаково для всех примеров и потому не будет повторяться. Кроме того, если вы подробно разберётесь с примером "A", то для понимания остальных примеров потребуется совсем мало времени.

## Пример A - трансформация класса: замена обращения к полю класса

Начнём с простого примера, логика которого реализована в классе example.ExampleA. Его метод run() увеличивает на 1 значение счётчика counter с изначальным значением 0. Для увеличения значения используется следующий метод:

|  |
| --- |
| /\*example.ExampleA\*/  private final void increment(final int incrementValue) {  counter += incrementValue;  } |

которому в качестве аргумента передаётся 1.

При запуске примера

|  |
| --- |
| java -jar main.jar -ex a |

он выводит следующие сообщения:

|  |
| --- |
| >ExampleA run - counter = 0 before incrementation  >ExampleA run - counter = 1 after incrementation |

Теперь при помощи трансформера классов мы изменим тело метода increment(int) таким образом, чтобы он инкрементировал значение счётчика на 2 независимо от переданного аргумента. Для этого нам нужно решить две задачи:

* написать трансформер классов, который сможет трансформировать класс при его первоначальной загрузке.
* написать класс Java-агента, который зарегистрирует созданный трансформер.

Трансформером для этого примера является example.transform.ClassFileTransformerExampleA, который расширяет абстрактный класс util.AbstractClassFileTransformer, имплементирующий интерфейс ClassFileTransformer, о котором говорилось в разделе [Трансформация класса в процессе его первоначальной загрузки](#2c2vhjbuwc92). Здесь реализуется абстрактный метод acceptClassForTransformation(ClassLoader, String) так, чтобы трансформер трансформировал только класс example.ExampleA, т.к. именно его метод increment(int) нам нужно трансформировать. Кроме того, реализуется метод doTransform(CtClass), который делает всю работу по трансформации, используя возможности библиотеки Javassist. Чтобы продемонстрировать больше возможностей библиотеки Javassist, трансформация происходит по следующему алгоритму:

* в методе increment(int) ищутся все обращения к любым полям трансформируемого класса.
* проверяется что найденное обращение является записью в поле и что поле имеет тип int, и если всё так, обращение к полю заменяется на инкрементирование поля на 2. Это делается при помощи объекта типа javassist.expr.ExprEditor.

Классом Java-агента для этого примера является example.agent.AgentExampleAbc, который расширяет класс example.agent.Agent и определяет метод premain(String, Instrumentation). Метод premain(...) регистрирует трансформер example.transform.ClassFileTransformerExampleA. Отмечу что этот же класс агента используется для примеров "B" и "C" с точно такой же целью: зарегистрировать их трансформеры классов. Поэтому метод регистрирует сразу три трансформера, хотя фактически в каждом примере используется только свой трансформер, т.к. благодаря методу acceptClassForTransformation(ClassLoader, String), трансформеры пропускают классы, для трансформации которых они не предназначены.

Мы уже видели результат запуска примера без Java-агента, регистрирующего трансформер, а теперь запустим пример с Java-агентом:

|  |
| --- |
| java -javaagent:agent-abc.jar -jar main.jar -ex a |
| >ExampleA run - counter = 0 before incrementation  >ExampleA run - counter = 2 after incrementation |

Как видно, вместо инкрементирования на 1, счётчик был увеличен на 2. Можем посмотреть подробнее что происходит при запуске примера:

|  |
| --- |
| java -javaagent:agent-abc.jar=trace -jar main.jar -logl trace -ex a |
| >AgentExampleAbc premain - Invocation  >Agent initInstrumentationEnvironment -  Instrumentation environment was initialized  >Agent registerClassFileTransformers -  Class transformer 'ClassFileTransformerExampleA' was successfully added to instrumentation  ...  >AgentExampleAbc premain - Invocation finished  >Main main - Invocation  >AbstractClassFileTransformer unsafeTransform -  Transforming class 'com/gl/vn/me/ko/sample/instrumentation/example/ExampleA'  >ClassFileTransformerExampleA doTransform - Transforming method 'increment(...)'  >ClassFileTransformerExampleA$WriteAccessIntExprEditor edit - Modifying access to the field 'counter I'  >AbstractClassFileTransformer unsafeTransform -  Class 'com/gl/vn/me/ko/sample/instrumentation/example/ExampleA' was successfully transformed  >ExampleA run - counter = 0 before incrementation  >ExampleA run - counter = 2 after incrementation  >Main main - Invocation finished |

Видно что сначала был запущен метод premain(...), который зарегистрировал крансформер классов, потом запустился метод main(...), который инициировал загрузку класса example.ExampleA, вызвав метод Class.forName(...) (см. код класса env.Main). Загрузка класса example.ExampleA, инициировала процесс определения класса, который привёл к вызову метода transform(...) у зарегистрированного трансформера классов, как описано в разделе [Трансформация класса в процессе его первоначальной загрузки](#2c2vhjbuwc92), и наконец был вызван метод doTransform(...). После завершения трансформации, была завершена процедура загрузки класса example.ExampleA и был вызван его трансформированный метод run().

## Пример B - трансформация класса: замена вызова конструктора класса, создание CGLib-прокси

Логика примера находится в классе example.ExampleB. Тут создаются два объекта типа BigDecimal, представляющих значение числа π: один из объекта String, а другой из double, и у них вызываются методы pow(int) и negate(). Запуск примера выглядит следующим образом:

|  |
| --- |
| java -jar main.jar -ex b |
| >ExampleB run - pi.pow(2) = 9.87  >ExampleB run - pi.negate() = -3.14 |

Трансформируем метод run() примера так, чтобы вместо использования объектов BigDecimal, он использовал прокси объекты типа BigDecimal.

Учитывая что BigDecimal - это класс, а не интерфейс, нам придётся использовать не Java-прокси, а CGLib-прокси. Дело в том, что Java-прокси подходят для тех случаев, где ими нужно заменить объекты, к которым обращаются через интерфейс. А если обращение к объекту происходит не через интерфейс, то Java-прокси бессильны, за то отлично подходят CGLib-прокси. Генерация прокси классов и прокси объектов не входит в тему статьи, но я решил включить их для повышения информативности примеров. Если вы не знакомы с функционалом CGLib-прокси, посмотрите документацию классов util.cglib.Proxy и util.cglib.MethodInterceptor, а также документацию класса net.sf.cglib.proxy.Enhancer в [[7]](#dm2y6bc575ex). Для создания прокси объектов типа BigDecimal предназначен класс example.proxy.BigDecimalProxyFactoryExampleB. У этого класса есть методы create(String) и create(double), создающие прокси объекты типа BigDecimal. В общем случае нам бы потребовалось реализовать метод create(...) для каждого конструктора класса BigDecimal, но т.к. в примере используются только конструкторы BigDecimal(String) и BigDecimal(double), нам достаточно двух методов create(...) принимающих такие же типы параметров. Возвращаемые методами create(...) прокси объекты устроены таким образом, что при вызове у них любого не final и не private метода класса BigDecimal, вызывается метод example.proxy.MethodInterceptorExampleB.intercept(...). Этот метод мог бы просто вызывать оригинальный метод класса BigDecimal, но он сделан хитрее: если оригинальный метод возвращает объект типа BigDecimal, то у этого объекта "меняется знак" (вызывается метод negate()), и после этого объект возвращается вместо результата оригинального метода. Такая подмена должна будет изменить сообщения, выводимые при работе примера после трансформаци метода run().

Трансформирующий код в этом примере находится в классе трансформера example.transform.ClassFileTransformerExampleB, который как и в случае примера "A", регистрируется в классе example.agent.AgentExampleAbc. Трансформация происходит следующим образом:

* внутри каждого метода определённого в трансформируемом классе example.ExampleB, ищутся вызовы конструкторов класса BigDecimal.
* каждый найденный вызов конструктора BigDecimal заменяется на вызов метода BigDecimalProxyFactoryExampleB.create(...) с теми же аргументами, которые передавались в конструктор. Это делается при помощи объекта javassist.CodeConverter, предварительно настроенного вызовом его метода javassist.CodeConverter.replaceNew(...).

При запуске примера с агентом, получим следующее:

|  |
| --- |
| java -javaagent:agent-abc.jar --add-opens=java.base/java.lang=ALL-UNNAMED -jar main.jar -logl debug -ex b  //the startup parameter --add-opens is required since JRE 9, see [JEP 261: Module System](http://openjdk.java.net/jeps/261) for details |
| >AbstractClassFileTransformer unsafeTransform -  Transforming class 'com/gl/vn/me/ko/sample/instrumentation/example/ExampleB'  >ClassFileTransformerExampleB doTransform -  Substituting constructor calls 'new java.math.BigDecimal(...)' with the method 'com.gl.vn.me.ko.sample.instrumentation.example.proxy.BigDecimalProxyFactoryExampleB.create(...)' invocations  >AbstractClassFileTransformer unsafeTransform -  Class 'com/gl/vn/me/ko/sample/instrumentation/example/ExampleB' was successfully transformed  >MethodInterceptorExampleB intercept -  The return value of the method  'public java.math.BigDecimal java.math.BigDecimal.pow(int)' is '9.87'  and it's a BigDecimal. Return it with the opposite sign  >ExampleB run - pi.pow(2) = -9.87  >MethodInterceptorExampleB intercept -  The return value of the method  'public java.math.BigDecimal java.math.BigDecimal.negate()' is '-3.14'  and it's a BigDecimal. Return it with the opposite sign  >ExampleB run - pi.negate() = 3.14 |

Видно что после трансформации класса example.ExampleB, он действительно использует не оригинальные объекты BigDecimal, а соответствующие им прокси объекты. Из-за этого в игру вступает метод example.proxy.MethodInterceptorExampleB.intercept(...), описанный выше, и поведение программы меняется: например pi.pow(2), т.е. квадрат числа π, получается отрицательным.

## Пример С - трансформация класса: замена возвращаемого значения метода, создание Java-прокси

Логика примера находится в классе example.ExampleC. В примере создаётся объект типа Map, содержащий две пары ключ-значение, определяющие цвета вентилей для горячей и холодной воды:

|  |
| --- |
| private final static Map<Integer, String> createFaucetColorCodeMap() {  final Map<Integer, String> result = new HashMap<Integer, String>(2);  result.put(COLD\_VALVE, "blue");  result.put(HOT\_VALVE, "red");  return result;  } |

После чего пример получает из созданного объекта цвет по ключам COLD\_VALVE/HOT\_VALVE и выводит соответствующие сообщения:

|  |
| --- |
| java -jar main.jar -ex c |
| >ExampleC run - The 'cold' color code is 'blue'  >ExampleC run - The 'hot' color code is 'red' |

Мы трансформируем метод createFaucetColorCodeMap() таким образом, чтобы после создания и заполнения объекта Map, он возвращал не сам этот объект, а его прокси, который для ключа COLD\_VALVE будет возвращать значение для HOT\_VALVE и наоборот, т.е. будет вести себя так, как будто цвета поменялись местами.

На этот раз интересующий нас объект используется через интерфейс, потому можно использовать Java-прокси. Если вы не знакомы с функционалом Java-прокси, посмотрите документацию класса java.lang.reflect.Proxy в [[9]](#9eio59bdric3). Для создания прокси объектов типа Map создан класс example.proxy.MapProxyFactoryExampleC. Его метод create(Map) возвращает прокси объект, созданный из переданного в качестве аргумента оригинального объекта. Получаемые таким образом прокси объекты устроены так, что при вызове у них любого метода интерфейса Map, вызывается метод example.proxy.InvocationHandlerExampleC.invoke(...). Этот метод делает следующее: если обнаруживает что был вызван метод get(Object), которому в качестве аргумента был передан ключ COLD\_VALVE, то он возвращает значение для ключа HOT\_VALVE и наоборот; в остальных случаях метод просто возвращает значение, которое возвращает метод оригинального объекта.

Трансформирующий код в этом примере находится в классе трансформера example.transform.ClassFileTransformerExampleС, который регистрируется в классе example.agent.AgentExampleAbc. Трансформация происходит следующим образом: возвращаемое значение метода example.ExampleC.createFaucetColorCodeMap() заменяется на прокси объект, созданный из оригинального возвращаемого значения при помощи метода example.proxy.MapProxyFactoryExampleC.create(Map). Обратите внимание что при этом используется специальная комбинация символов $\_, обозначающая возвращаемое значение метода. Другие языковые расширения, поддерживаемые компиллятором Javassist, можно посмотреть в [[5]](#xec354wlzlxr).

Если запустить пример с Java-агентом, то увидим следующее:

|  |
| --- |
| java -javaagent:agent-abc.jar -jar main.jar -logl debug -ex c |
| >AbstractClassFileTransformer unsafeTransform -  Transforming class 'com/gl/vn/me/ko/sample/instrumentation/example/ExampleC'  >ClassFileTransformerExampleC doTransform -  Modifying of return value of the method  'com.gl.vn.me.ko.sample.instrumentation.example.ExampleC.createFaucetColorCodeMap()'  >AbstractClassFileTransformer unsafeTransform -  Class 'com/gl/vn/me/ko/sample/instrumentation/example/ExampleC' was successfully transformed  >InvocationHandlerExampleC invoke -  Method 'public abstract java.lang.Object java.util.Map.get(java.lang.Object)' was invoked with argument '-1' (cold). Swap it with '1' (hot)  >ExampleC run - The 'cold' color code is 'red'  >InvocationHandlerExampleC invoke -  Method 'public abstract java.lang.Object java.util.Map.get(java.lang.Object)' was invoked with argument '1' (hot). Swap it with '-1' (cold)  >ExampleC run - The 'hot' color code is 'blue' |

Видно что теперь пример вместо оригинального объекта использует прокси объект, в результате чего происходят вызовы метода example.proxy.InvocationHandlerExampleC.invoke(...), благодаря которому поведение прокси объекта отличается от поведения оригинального объекта, и мы видим например сообщение The 'cold' color code is 'red', которое отличается от сообщения The 'cold' color code is 'blue', выводимого нетрансформированным классом example.ExampleC.

## Пример D - переопределение класса: трансформация уже загруженного класса java.lang.String

Итак, наконец мы дошли до самого интересного примера. Примеры "A", "B" и "C" демонстрировали трансформацию класса при его первоначальной загрузке. В этом примере мы переопределим уже загруженный класс, входящий в состав Java Platform SE, мы переопределим класс String. Отмечу что при трансформации классов Java Platform нужно быть предельно внимательным, т.к. даже безобидные на первый взгляд изменения могут иметь серьёзные последствия.

В соответствии со спецификацией метода String.toString(), учитывая что два одинаковых строковых литерала в коде класса всегда будут представлены одним и тем же объектом String, выражение "test" == "test".toString()всегда должно быть true, и запуск примера выглядит следующим образом

|  |
| --- |
| java -jar main.jar -ex d |
| >ExampleD run - ("test" == "test".toString()) is true |

Давайте переопределим класс String таким образом, чтобы это выражение стало false. Мы изменим метод String.toString() таким образом, чтобы вместо this он возвращал new String(this). Кроме того, пусть метод String.toString() выводит сообщение о том, что он был вызван.

Как описывалось в разделе [Трансформация уже загруженного класса](#6uuec1m8a0x8), в Java Platform SE 6 появился метод Instrumentation.retransformClasses(Class<?>...), позволяющий произвести трансформацию указанных классов аналогично тому, как она происходит при их первоначальной загрузке (только в этом случае конечно действуют все те же ограничения, которые накладываются на трансформацию уже загруженных классов). В Java Platform SE 5 такого удобного метода нет, но его можно написать самостоятельно. Итак, нашим аналогом является метод InstrumentationEnvironment.retransformClasses(Class<?>[]). Метод делает следующее: для всех переданных классов получает их оригинальный байт-код и используя его вызывает метод Instrumentation.redefineClasses(ClassDefinition[]). Обратите внимание каким образом происходит получение оригинального байт-кода классов: сначала найденные объекты javassist.CtClass удаляются из javassist.ClassPool, а затем грузятся заново, таким образом мы точно избавляемся от тех изменений в классах, которые могли находиться в памяти. После получения оригинального байт-кода классов, соответствующие объекты CtClass размораживаются методом javassist.CtClass.defrost(), чтобы разрешить их дальнейшую трансформацию. Кроме того, метод retransformClasses(Class<?>[]) вначале выполнения получает эксклюзивный доступ к util.javassist.JavassistEnvironment, используя метод lock(), таким образом во время выполнения метода retransformClasses(Class<?>[]) ни один другой поток не сможет воспользоваться методами класса util.javassist.JavassistEnvironment.

Трансформирующий код находится в классе example.transform.ClassFileTransformerExampleD. На этот раз мы полностью заменяем оригинальное тело метода. Обратите внимание что для обозначения this в конструкции new String(this) в Javassist-коде, используется комбинация символов $0. Также перед использованием логгера, делается проверка что он уже был сконфигурирован (помним, что при трансформации классов Java Platform нужно быть особенно внимательным).

В отличие от предыдущих примеров, пример "D" использует свой собственный класс Java-агента: example.agent.AgentExampleD, который не только регистрирует трансформер классов, но и переопределяет класс String, вызвав его ретрансформацию. Особое внимание стоит обратить на текст манифеста JAR-файла Java-агента:

|  |
| --- |
| Manifest-Version: 1.0  Premain-Class: com.gl.vn.me.ko.sample.instrumentation.example.agent.AgentExampleD  Class-Path: instrumentation-util.jar  Can-Redefine-Classes: true |

Атрибут Can-Redefine-Classes потребовался для того, чтобы разрешить переопределение классов. Обратите внимание, что загрузку класса String производит не системный загрузчик, а первичный загрузчик, что подтверждает также специальная проверка в example.agent.AgentExampleD:

|  |
| --- |
| final ClassLoader classLoader = classToRetransform.getClassLoader();  final String classLoaderStringRepresentation = classLoader == null ? "bootstrap" : classLoader.toString(); |

В результате которой при запуске примера с Java-агентом и уровнем логирования debug, мы увидим сообщение:

|  |
| --- |
| >AgentExampleD retransformClass -  Class 'java.lang.String' was loaded via 'bootstrap' class loader. The class will be retransformed |

При переопределении класса String его загрузка опять будет произведена первичным загрузчиком.

Итак, при запуске примера с Java-агентом результат выглядит следующим образом:

|  |
| --- |
| java -javaagent:agent-d.jar=trace -jar main.jar -logl trace -ex d |
| >AgentExampleD premain - Invocation  >Agent initInstrumentationEnvironment - Instrumentation environment was initialized  >Agent registerClassFileTransformers - Class transformer 'ClassFileTransformerExampleD' was successfully added to instrumentation  >AgentExampleD retransformClass - Class 'java.lang.String' was loaded via 'bootstrap' class loader. The class will be retransformed  >AbstractClassFileTransformer unsafeTransform - Transforming class 'java/lang/String'  >ClassFileTransformerExampleD doTransform - Modifying return value of the method 'java.lang.String.toString()'  >AbstractClassFileTransformer unsafeTransform - Class 'java/lang/String' was successfully transformed  >AgentExampleD premain - Invocation finished  >java.lang.String.toString invocation for '-logl'  >java.lang.String.toString invocation for 'trace'  >java.lang.String.toString invocation for '-ex'  >java.lang.String.toString invocation for 'd'  >java.lang.String.toString invocation for 'TRACE'  >java.lang.String.toString invocation for '>%-30C{1} %M - %m%n'  >Main main - Invocation  >java.lang.String.toString invocation for 'makeConcatWithConstants'  >java.lang.String.toString invocation for 'com.gl.vn.me.ko.sample.instrumentation.example.Example'  >java.lang.String.toString invocation for 'D'  >java.lang.String.toString invocation for 'test'  >java.lang.String.toString invocation for 'makeConcatWithConstants'  >java.lang.String.toString invocation for '("test" == "test".toString()) is '  >ExampleD run - ("test" == "test".toString()) is false  >Main main - Invocation finished |

Видно что метод String.toString() действительно стал выводить сообщения о том что он был вызван, кроме того, выражение "test" == "test".toString() оказалось false.

# Лицензионное соглашение с конечным пользователем

Содержимое статьи и исходный код примеров распространяется под лицензией [WTFPL](http://www.wtfpl.net/).

# Литература

1. *"Using the BootClasspath: Tweaking the Java Runtime API"* [<http://www.tedneward.com/files/Papers/BootClasspath/index.html>]*, Ted Neward*
2. *"Understanding Class.forName: Loading Classes Dynamically from within Extensions"* [<http://www.idevelopment.info/data/Programming/java/reflection/Understanding_ClassForName.pdf>]*, Ted Neward*
3. *"Creating and Using Extensions"* [<http://docs.oracle.com/javase/tutorial/ext/basics/>]*, Oracle*
4. *"Packaging Programs in JAR Files"* [<http://docs.oracle.com/javase/tutorial/deployment/jar/>]*, Oracle*
5. *"Getting Started with Javassist"* [<http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/tutorial/tutorial.html>]*, Shigeru Chiba*
6. *"Javassist API Specification"* [<http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/html/index.html>]*, Shigeru Chiba*
7. *"CGLib API Specification",* [<http://cglib.sourceforge.net/apidocs/index.html>]
8. *"The Java Language Specification" and "The Java Virtual Machine Specification" [*[*http://docs.oracle.com/javase/specs/*](http://docs.oracle.com/javase/specs/)*], Oracle*
9. *"Java 2 Platform Standard Edition 5 API Specification"* [<http://docs.oracle.com/javase/1.5.0/docs/api/>]*, Oracle*
10. *"Java Platform Debugger Architecture Documentation"* [[http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/](http://www.google.com/url?q=http%3A%2F%2Fdocs.oracle.com%2Fjavase%2F1.5.0%2Fdocs%2Fguide%2Fjpda%2F&sa=D&sntz=1&usg=AFQjCNGSlVwgdTSj6UO89OPMqgWOt7cILg)]*, Oracle*