Инструментарий Нативных Образов Axiom NIK Pro
Руководство пользователя

AXIOM JDK

Copyright © 2019-2024 Все права защищены АО "АКСИОМ" (АКСИОМ)

Программное обеспечение АКСИОМ содержит программное обеспечение с открытым исходным кодом. Дополнительная информация о коде сторонних разработчиков доступна на сайте https://axiomjdk.ru/third_party_licenses. Для дополнительной информации о том, как получить копию исходного кода, можно обратиться по адресу info@axiomjdk.ru.

ДАННАЯ ИНФОРМАЦИЯ МОЖЕТ ИЗМЕНЯТЬСЯ БЕЗ ПРЕДВАРИТЕЛЬНОГО УВЕДОМЛЕНИЯ. АКСИОМ ПРЕДОСТАВЛЯЕТ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ "КАК ЕСТЬ" БЕЗ КАКИХ-ЛИБО ГАРАНТИЙ, АКСИОМ ПРЯМО ОТКАЗЫВАЕТСЯ ОТ ВСЕХ ПОДРАЗУМЕВАЕМЫХ ГАРАНТИЙ, ВКЛЮЧАЯ, НО НЕ ОГРАНИЧИВАЯСЬ ПОДРАЗУМЕВАЕМЫМИ ГАРАНТИЯМИ ТОВАРНОЙ ПРИГОДНОСТИ И ПРИГОДНОСТИ ДЛЯ ОПРЕДЕЛЕННОЙ ЦЕЛИ.

АКСИОМ НИ ПРИ КАКИХ ОБСТОЯТЕЛЬСТВАХ НЕ НЕСЕТ ОТВЕТСТВЕННОСТИ ЗА ЛЮБЫЕ КОСВЕННЫЕ, СЛУЧАЙНЫЕ, СПЕЦИАЛЬНЫЕ, ШТРАФНЫЕ ИЛИ КОСВЕННЫЕ УБЫТКИ, ИЛИ УБЫТКИ ОТ ПОТЕРИ ПРИБЫЛИ, ДОХОДА, ДАННЫХ ИЛИ ИСПОЛЬЗОВАНИЯ ДАННЫХ, ПОНЕСЕННЫЕ ВАМИ ИЛИ ЛЮБОЙ ТРЕТЬЕЙ СТОРОНОЙ, БУДЬ ТО В РЕЗУЛЬТАТЕ ДЕЙСТВИЯ ДОГОВОРА ИЛИ ДЕЛИКТА, ДАЖЕ ЕСЛИ АКСИОМ БЫЛО ПРЕДУПРЕЖДЕНО О ВОЗМОЖНОСТИ ТАКИХ УБЫТКОВ.

Использование любого программного продукта АКСИОМ регулируется соответствующим лицензионным соглашением, которое никоим образом не изменяется условиями данного уведомления. Программные продукты и фирменные наименования: Axiom JDK, Axiom JDK Pro, Axiom Runtime Container Pro, Axiom Linux, Libercat, Libercat Certified и АКСИОМ принадлежат АКСИОМ и их использование допускается только с разрешения правообладателя.

Товарный знак Linux® используется в соответствии с сублицензией от Linux Foundation, эксклюзивного лицензиата Линуса Торвальдса, владельца знака на всемирной основе. Java и OpenJDK являются товарными знаками или зарегистрированными товарными знаками компании Oracle и/или ее аффилированных лиц. Другие торговые марки являются собственностью их соответствующих владельцев и используются только в целях идентификации. :pp: ++

Содержание

1. Введение

Пакет разработчика Axiom NIK Pro является универсальным инструментом на базе GraalVM Open Source для многоязычного программирования и ускорения работы ваших приложений. Axiom NIK Pro имеет ту же функциональность, что и GraalVM.

Axiom NIK Pro заранее компилирует ваши Java-приложения в автономные двоичные файлы. Эти двоичные файлы меньше по размеру, запускаются быстрее, обеспечивают максимальную производительность и используют меньше памяти и меньше загружают процессор, чем приложения, работающие на виртуальной машине Java (JVM).

Axiom NIK Pro обеспечивает бесшовную реализацию многоязычных проектов, например микросервисов, таких как Spring Boot, Micronaut, Helidon и Quarkus, на различных языках программирования.

Вы можете использовать Axiom NIK Pro так же, как и любой другой комплект разработки Java в вашей IDE.

2. Требования к программным средствам

Для корректной работы инструмента native-image, доступного в каталоге bin вашей установки Axiom NIK Pro, необходимы следующие компоненты:

- файлы заголовков для библиотеки C
- glibc-devel
- zlib
- gcc и/или libstdc++-static

Эти зависимости можно установить (если они еще не установлены) с помощью менеджера пакетов на вашем компьютере.

3. Основные функции и алгоритм работы

Пакет разработчика Axiom NIK Pro преобразует ваше Java-приложение в предварительно (AOT) скомпилированный исполняемый файл, который запускается автономно и почти моментально. Благодаря совместимости с различными платформами, включая ОС Linux с библиотекой musl, решение оптимизирует потребление ресурсов и уменьшает размер приложения.

Созданный нативный исполняемый файл включает в себя классы приложения, классы его зависимостей, классы рантайм библиотеки и статически связанный собственный код из JDK. Файл запускается на той ОС для которой он собран и включает в себя необходимые компоненты, такие как управление памятью, планирование потоков и т. д.

Native Image поддерживает языки на основе JVM, например Java, Scala, Kotlin. Полученный образ может дополнительно выполнять динамические языки, такие как JavaScript, Ruby, R или Python. Чтобы сообщить сборщику о гостевом языке, используемом приложением, укажите `--language:<languageId>` для каждого гостевого языка (например, `--language:js`).

Предварительная сборка и выполнение

Во время предварительной сборки образа сборщик может выполнять пользовательский код. Этот код может иметь побочные функции, например запись значения в статическое поле класса. Таким образом, этот код выполняется во время **сборки**. Значения, записанные в статические поля этим кодом, сохраняются в выделенной памяти. **Выполнение** относится к коду и состоянию в двоичном файле во время его выполнения.

Самый простой способ увидеть разницу между концепциями сборки и выполнения — это инициализация класса. В Java класс инициализируется при первом использовании. Каждый класс Java, используемый во время сборки, инициализируется во время сборки. Обратите внимание: простая загрузка класса не обязательно приводит к его инициализации. Статический инициализатор классов, инициализированных во время сборки, выполняется на JVM, на которой выполняется сборка образа. Если класс инициализируется во время сборки, его статические поля сохраняются в созданном двоичном файле. Во время выполнения первое использование такого класса не вызывает инициализацию класса.

Пользователи могут запускать инициализацию класса во время сборки разными способами:

- Передав `--initialize-at-build-time=<class>` сборщику native-image;

- Используя класс в статическом инициализаторе класса, инициализированного во время сборки.

Native Image инициализирует часто используемые классы JDK во время сборки образа, например `java.lang.String`, `java.util.**` и т. д. Обратите внимание, что инициализация классов во время сборки является экспертной функцией.

> **Важно:**
>
> Не все классы подходят для инициализации во время сборки. Существует небольшая часть функций Java, которые не подлежат предварительной компиляции (AOT) и, следовательно, теряют преимущества в производительности. Чтобы иметь возможность создать высокооптимизированный нативный исполняемый файл, проводится агрессивный статический анализ, при котором все классы и все байт-коды, доступные во время выполнения, должны быть известны во время сборки. Поэтому невозможно загрузить новые данные, которые не были доступны во время предварительной компиляции.

В следующем примере демонстрируется разница между исполняемым кодом во время сборки и во время выполнения:

```
public class HelloWorld {
  static class Greeter {
    static {
      System.out.println("Greeter is getting ready!");
    }
    public static void greet() {
     System.out.println("Hello, World!");
    }
  }

 public static void main(String[] args) {
  Greeter.greet();
 }
}
```

Сохранив код в файле `HelloWorld.java`, компилируем и запускаем приложение на JVM:

```
javac HelloWorld.java
java HelloWorld
Greeter is getting ready!
Hello, World!
```

Теперь мы создадим нативный образ, а затем выполним его:

```
native-image HelloWorld
========================================================
GraalVM Native Image: Generating 'helloworld' (executable)...
========================================================
...
Finished generating 'helloworld' in 14.9s.
./helloworld
Greeter is getting ready!
Hello, World!
```

HelloWorld запустился и вызвал `Greeter.greet`. Это привело к инициализации Greeter и печати сообщения Greeter is getting ready!. Инициализатор класса Greeter выполняется во время выполнения образа.

Выделенная память нативного образа

Выделенная память нативного образа содержит:

- Объекты, созданные во время сборки образа, доступные из кода приложения.
- Объекты классов java.lang.Class, используемые в нативном образе.
- Константы объекта, встроенные в код метода.

При запуске, нативный образ копирует исходное содержание памяти нативного образа из двоичного файла.

Один из способов включения объектов в память образа — инициализировать классы во время сборки:

```
class Example {
  private static final String message;
  static {
    message = System.getProperty("message");
  }
  public static void main(String[] args) {
    System.out.println("Hello, World! My message is: " + message);
  }
}
```

Теперь компилируем и запускаем приложение на JVM:

```
javac Example.java
java -Dmessage=hi Example
Hello, World! My message is: hi
```

```
java -Dmessage=hello Example
Hello, World! My message is: hello
java Example
Hello, World! My message is: null
```

Если создать нативный образ, в котором класс Example инициализируется во время сборки, произойдет следующее:

```
native-image Example --initialize-at-build-time=Example -Dmessage=native
========================================================
GraalVM Native Image: Generating 'example' (executable)...
========================================================
...
Finished generating 'example' in 19.0s.
./example
Hello, World! My message is: native
./example -Dmessage=aNewMessage
Hello, World! My message is: native
```

Инициализатор класса Example был выполнен во время сборки образа. Это создало объект String для поля сообщения и сохранило его в памяти образа.

4. Создание нативного исполняемого файла

Инструмент создания нативных образов native-image принимает в качестве входных данных байт-код Java. Вы можете создать собственный исполняемый файл из файла класса, из файла JAR или из модуля.

5. Параметры native-image

Сборщику native-image необходимо указать путь к классам для всех классов, используя такой же параметр, как для запуска java: -cp за ним следует список каталогов или файлов JAR, разделенных двоеточием (:). Имя класса, содержащего main метод, является последним аргументом. Вместо этого вы можете использовать -jar и предоставить файл JAR, в котором указан основной метод (main).

Синтаксис команды native-image:

Для создания исполняемого файла класса в текущем рабочем каталоге

```
native-image [параметры] class [имя образа] [опции]
```

для создания образа из файла JAR

```
native-image [параметры] -jar jarfile [имя образа] [опции]
```

Передаваемые параметры native-image обрабатывает слева направо.

Запустите `native-image --help`, чтобы получить обзор команд и параметров.

Запустите `native-image --help-extra`, чтобы вывести справку по нестандартным параметрам, макросам и параметрам сервера.

Список параметров native-image для постройки образов

Параметр	Описание
`-cp`, `-classpath`, `--class-path <class search path of directories and zip/jar files>`	Список каталогов, JAR-архивов и ZIP-архивов, разделенных двоеточиями для поиска файлов классов.
`-D<name>=<value>`	Установить системное свойство.
`-J<flag>`	Передать <flag>непосредственно JVM, на которой запущен сборщик нативных образов.

Параметр	Описание
`-O<level>`	0 — отсутствие оптимизации или 1 — базовая оптимизация (по умолчанию).
`--verbose`	Включить подробный вывод.
`--version`	Вывести версию продукта и выйти.
`--help`	Вывести это справочное сообщение.
`--help-extra`	Вывести справку по нестандартным опциям.
`--allow-incomplete-classpath`	Разрешить сборку образа с неполным путем к классу. Сообщать об ошибках разрешения типов во время выполнения, когда к ним обращаются в первый раз, а не во время построения образа.
`--auto-fallback`	Создать нативный образ, если возможно.
`--enable-http`	Включить поддержку http в сгенерированном образе.
`--enable-https`	Включить поддержку https в сгенерированном образе.
`--enable-url-protocols`	Список URL-протоколов, разделенных запятыми, которые нужно включить.
`--features`	Разделенный запятыми список полноценных классов для реализации функций.
`--force-fallback`	Принудительное создание резервного образа.

Параметр	Описание
`--initialize-at-build-time`	Разделенный запятыми список пакетов и классов (и неявно всех их суперклассов), которые инициализируются во время сборки образа. Пустая строка обозначает все пакеты.
`--initialize-at-run-time`	Разделенный запятыми список пакетов и классов (и неявно всех их подклассов), которые должны быть инициализированы во время выполнения, а не во время сборки образа. Пустая строка в настоящее время не поддерживается.
`--install-exit-handlers`	Предоставить java.lang.Terminator обработчики выхода для исполняемых образов.
`--libc`	Использовать libc реализацию нативной библиотеки (доступны реализации glibc и musl).
`--native-compiler-options`	Использовать специальную версию компилятора C, используемую для компиляции кода запроса.
`--native-compiler-path`	Указать путь к компилятору C, используемому для компиляции и компоновки кода запроса.
`--native-image-info`	Показать информацию о наборе инструментов и настройки сборки образа.
`--no-fallback`	Создать нативный образ или сообщить об ошибке, если невозможно.
`--report-unsupported-elements-at-runtime`	Сообщать об использовании неподдерживаемых методов и полей во время выполнения при первом обращении к ним, а не об ошибке во время построения образа.
`--shared`	Создать общую библиотеку.

Параметр	Описание
`--static`	Создать статически связанный исполняемый файл (требуются статические libc и zlib библиотеки).
`--target`	Выберите целевую ОС для компиляции нативного образа (в формате - in). По умолчанию используется ОС-архитектура хоста.
`--trace-class-initialization`	Разделенный запятыми список полных имен классов, для которых отслеживается инициализация класса.
`--trace-object-instantiation`	Разделенный запятыми список полных имен классов, для которых отслеживается создание экземпляра объекта.
`-da`	Отключить ассерты с заданной степенью детализации в сгенерированном образе. Также поддерживаются варианты `-da[:[имя_пакета]|:[имя_класса]` или `-disableassertions[:[имя_пакета]|:[имя_класса]`.
`-dsa`	отключить ассерты во всех системных классах.
`-ea`	Включить ассерты с заданной степенью детализации в сгенерированном образе. Также поддерживаются варианты `-ea[:[имя_пакета]|:[имя_класса]` или `-enableassertions[:[имя_пакета]|:[имя_класса]`.
`-esa`	Включить ассерты во всех системных классах.
`-g`	Сгенерировать отладочную информацию.

Создание исполняемого файла из класса

Чтобы создать нативный исполняемый файл из файла класса Java в текущем рабочем каталоге, используйте следующую команду:

```
native-image [параметры] class [имя образа] [опции]
```

Например, создайте исполняемый файл для приложения HelloWorld следующим образом.

1. Сохраните этот код в файл с именем HelloWorld.java:

   ```
   public class HelloWorld {
    public static void main(String[] args) {
     System.out.println("Hello, Native World!");
    }
   }
   ```

2. Скомпилируйте его и создайте собственный исполняемый файл из класса Java:

   ```
   javac HelloWorld.java
   native-image HelloWorld
   ```

   Данная команда создаст нативный исполняемый файл helloWorld в текущем рабочем каталоге.

3. Запустите программу.

   ```
   ./helloWorld
   ```

Создание исполняемого файла из файла JAR

Чтобы создать нативный исполняемый файл из файла JAR в текущем рабочем каталоге, используйте следующую команду:

```
native-image [параметры] -jar jarfile [имя образа]
```

Поведение утилиты native-image по умолчанию соответствует команде java, что означает, что вы можете передавать параметры `-jar`, `-cp`, `-m` для сборки, как это обычно делается с java. Например, `java -jar App.jar someArgument` становится `native-image -jar App.jar и ./App someArgument`.

Создание исполняемого файла из модуля

Вы также можете преобразовать модульное приложение Java в нативный исполняемый файл.

Используйте следующую команду для создания нативного исполняемого файла из модуля Java:

```
native-image [параметры] --module <module>[/<mainclass>] [опции]
```

Получение уведомления о завершении процесса сборки

В зависимости от размера вашего приложения и доступных ресурсов вашего компьютера, AOT-компиляция вашего Java-приложения в нативный исполняемый файл может занять несколько минут. Если вы создаете проект в фоновом режиме, используйте команду, которая уведомит вас о завершении процесса сборки. Ниже приведены примеры команд для каждой операционной системы.

Linux

```
\# Уведомление в терминале
native-image -jar App.jar ... ; printf '\a'

\# Использование libnotify для создания уведомления на
\# рабочем столе
native-image -jar App.jar ... ; notify-send "GraalVM Native Image build
completed with exit code $?"

\# Использование Zenity, чтобы открыть окно с текстом
native-image -jar App.jar ... ; zenity --info --text="GraalVM Native Image
build completed with exit code $?"
```

macOS

```
\# Уведомление в терминале
native-image -jar App.jar ... ; printf '\a'

\# Уведомление с помощью синтезатора речи
native-image -jar App.jar ... ; say "GraalVM Native Image build completed"
```

Windows

```
REM Уведомление в терминале (для ввода ^G нажмите сочетание клавиш Ctrl+G)
native-image.exe -jar App.jar & echo ^G
```

```
REM Уведомление в отдельном окне с текстом
native-image.exe -jar App.jar & msg "%username%" GraalVM Native Image build
completed
```

6. Конфигурация сборки нативного образа

Сборщик Native Image поддерживает широкий спектр параметров для настройки процесса сборки нативного образа.

Встраивание файла конфигурации

Рекомендуемый способ конфигурирования постройки нативного образа — встроить файл native-image.properties в JAR-файл проекта. Сборщик нативных образов автоматически подберет все параметры конфигурации, предоставленные где-либо ниже местоположения ресурса META-INF/native-image/, и будет использовать их для создания аргументов командной строки создания образа.

Чтобы избежать ситуации, когда составные части проекта собираются с дублирующимися конфигурациями, рекомендуется использовать «подкаталоги» внутри META-INF/native-image. Таким образом, удастся избежать дублирующихся конфигураций для файла JAR, созданного из нескольких проектов maven. Например:

- Конфигурация для foo.jar находиться в `META-INF/native-image/foo_groupID/foo_artifactID`
- Конфигурация для bar.jar находится в `META-INF/native-image/bar_groupID/bar_artifactID`

Файл JAR, содержащий foo и bar, будет содержать обе конфигурации, не конфликтуя друг с другом. Поэтому рекомендуемая схема хранения данных конфигурации собственного образа в файлах JAR следующая:

```
META-INF/
└── native-image
  └── groupID
    └── artifactID
      └── native-image.properties
```

Обратите внимание, что использование `${.}` в файле `native-image.properties` распространяется на

расположение ресурса, которое содержит именно этот файл конфигурации. Это может быть полезно, если файл `native-image.properties` хочет ссылаться на ресурсы в своей «подпапке», например, `-H:SubstitutionResources=${.}/substitutions.json`. Всегда обязательно используйте варианты параметров, требующие ресурсов, т. е. используйте `-H:ResourceConfigurationResources` вместо `-H:ResourceConfigurationFiles`. Другие варианты, которые работают в этом контексте:

- -H:DynamicProxyConfigurationResources
- -H:JNIConfigurationResources
- -H:ReflectionConfigurationResources
- -H:ResourceConfigurationResources
- -H:SubstitutionResources
- -H:SerializationConfigurationResources

Имея такой составной файл native-image.properties, создание образа не требует каких-либо дополнительных аргументов, указанных в командной строке. Достаточно просто выполнить следующую команду:

```
$JAVA_HOME/bin/native-image -jar путь/[имя образа].jar
```

Чтобы увидеть, какие данные конфигурации применяются для построения образа, используйте команду native-image --verbose. Это покажет, откуда сборщик получает конфигурации для создания окончательных параметров командной строки для нативного образа.

Формат файла конфигурации

Файл `native-image.properties` — это обычный файл свойств Java, который можно использовать для указания конфигураций нативного образа. Поддерживаются следующие свойства.

Args

Используйте это свойство, если для правильного построения вашего проекта требуются специальные параметры командной строки нативного образа. Например, у `native-image-configure-examples/configure-at-runtime-example` есть `Args = --initialize-at-build-time=com.fasterxml.jackson.annotation.JsonProperty$Access` в его файле `own-image.properties`, чтобы гарантировать инициализацию класса `com.fasterxml.jackson.annotation.JsonProperty$Access` во время сборки образа.

JavaArgs

Иногда может возникнуть необходимость предоставить специальные параметры для JVM, на которой работает сборщик нативных образов. В этом случае можно использовать свойство `JavaArgs`.

ImageName

Это свойство можно использовать для указания определяемого пользователем имени образа. Если ImageName не используется, имя выбирается автоматически:

- `native-image -jar <name.jar>` устанавливает имя образа по умолчанию `<name>`
- `native-image -cp … fully.qualified.MainClass` устанавливает имя образа по умолчанию `fully.qualified.mainclass`

Использование ImageName не мешает пользователю позже переопределить имя через командную строку. Например, если `foo.bar` содержит `ImageName=foo_app`:

- `native-image -jar foo.bar` создает образ `foo_app`
- `native-image -jar foo.bar application` создает образ `application`

Порядок обработки аргументов

Аргументы, передаваемые в сборщик образов, обрабатываются слева направо. Это также распространяется на аргументы, которые передаются косвенно через конфигурацию нативного образа на основе `META-INF/native-image`. Предположим, у вас есть файл JAR, содержащий файл `native-image.properties` с `Args = -H:Optimize=0`. Затем, используя опцию `-H:Optimize=2` после команды `-cp <имя файла jar>`, вы можете переопределить настройку, полученную из файла JAR.

Указание параметров по умолчанию для нативного образа

Если необходимо передавать некоторые параметры для каждой сборки образа, например, чтобы всегда генерировать изображение с выводом подробной информации (`--verbose`), вы можете использовать переменную среды `NATIVE_IMAGE_CONFIG_FILE`. Если для этой переменной задан файл свойств Java, сборщик будет использовать настройку по умолчанию, определенную там, при каждом вызове. Создайте файл конфигурации и экспортируйте его с помощью следующей команды в `~/.bash_profile`:

```
NATIVE_IMAGE_CONFIG_FILE=$HOME/.native-image/default.properties
```

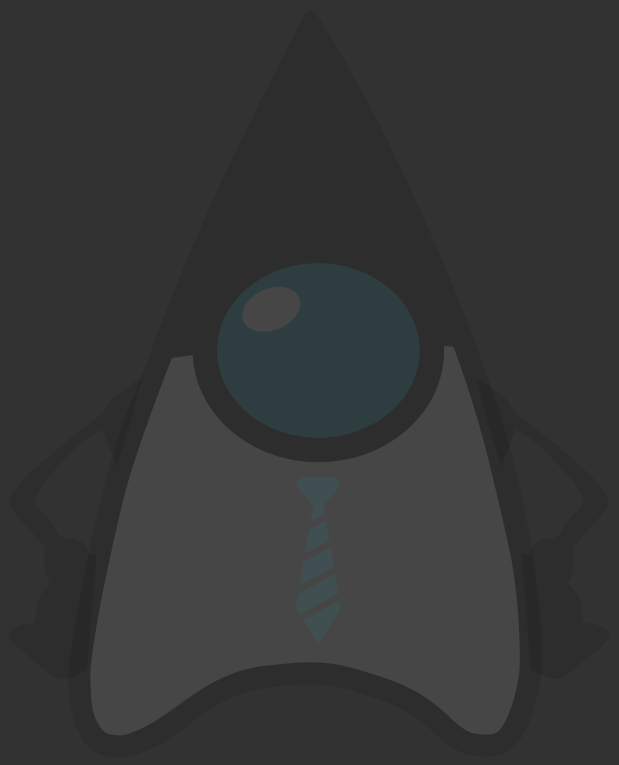
Каждый раз, когда используется native-image, он будет использовать аргументы, указанные как NativeImageArgs, а также аргументы, указанные в командной строке. Вот пример файла конфигурации, сохраненного как `~/.native-image/default.properties`:

```
NativeImageArgs = --configurations-path /home/user/custom-image-configs \
         -O1
```

Изменение пути к каталогу конфигурации

Сборщик образов по умолчанию хранит информацию о конфигурации в домашнем каталоге пользователя — `$HOME/.native-image/`. Чтобы изменить выходной каталог, установите переменную среды `NATIVE_IMAGE_USER_HOME`. Например:

```
export NATIVE_IMAGE_USER_HOME= $HOME/.local/share/native-image
```



Инструментарий Нативных Образов Axiom NIK Pro
Руководство пользователя

AXIOM JDK