Содержание

Предисловие 10

Часть I. Введение и учебный материал 13

Глава 1. Введение 14

1.1. Платформа Java и вычислительная среда 15
1.2. Роль интерфейса JNI 16
1.3. Последствия использования интерфейса JNI 17
1.4. Случаи использования интерфейса JNI 18
1.5. Развитие интерфейса JNI 19
1.6. Примеры программ 21

Глава 2. Приступая к работе 22

2.1. Общее представление 22
2.2. Объявление native-метода 24
2.3. Компиляция класса HelloWorld 25
2.4. Создание файла заголовков 25
2.5. Написание кода native-метода 26
2.6. Компиляция исходного кода на языке C и создание native-библиотеки 26
2.7. Запуск программы 27

Часть II. Справочное пособие по программированию 29

Глава 3. Основные типы, строки и массивы 30

3.1. Простой native-метод 30
3.1.1. Прототип C-функции, реализующей native-метод 31
3.1.2. Аргументы native-метода 31
3.1.3. Соответствие типов 32
3.2. Обращение к строкам 33
3.2.1. Преобразование к native-строкам 33
3.2.2. Освобождение ресурсов native-строк 35
3.2.3. Создание новых строк 35
3.2.4. Другие функции JNI для работы со строками 35
3.2.5. Новые JNI-функции для работы со строками в Java 2 SDK 1.2 37

3.2.6. Перечень JNI-функций для работы со строками 39
3.2.7. Выбор между функциями строк 40
3.3. Обращение к массивам 43
3.3.1. Обращение к массивам в языке C 43
3.3.2. Обращение к массивам простых типов 44
3.3.3. Перечень JNI-функций для работы с примитивными массивами 45
3.3.4. Выбор между функциями примитивного массива 46
3.3.5. Обращение к массивам объектов 48

Глава 4. Поля и методы 51
4.1. Обращение к полям 51
4.1.1. Алгоритм обращения к полю объекта 53
4.1.2. Дескрипторы поля 53
4.1.3. Обращение к статическим полям 54
4.2. Вызов методов 56
4.2.1. Вызов методов объекта 57
4.2.2. Формирование дескриптора метода 58
4.2.3. Обращение к статическим методам 59
4.2.4. Вызов методов суперкласса 60
4.3. Вызов конструкторов 61
4.4. Кэширование полей и идентификаторов методов 63
4.4.1. Кэширование во время использования 63
4.4.2. Кэширование в блоке статической инициализации класса 66
4.4.3. Сравнение двух способов кэширования идентификаторов 67
4.5. Эффективность использования JNI при обращении к полям и методам 68

Глава 5. Локальные и глобальные ссылки 71
5.1. Локальные и глобальные ссылки 72
5.1.1. Локальные ссылки 72
5.1.2. Глобальные ссылки 74
5.1.3. Слабые глобальные ссылки 75
5.1.4. Сравнительный анализ 76
5.2. Освобождение ссылок 77
5.2.1. Освобождение локальных ссылок 77
5.2.2. Управление локальными ссылками в Java 2 SDK 1.2 79

5.2.3. Освобождение глобальных ссылок 80
5.3. Правила управления ссылками 81

Глава 6. Исключения 85
6.1. Общее представление 85
6.1.1. Кэширование и вызов исключений в native-коде 85
6.1.2. Вспомогательные функции 87
6.2. Правильная обработка исключений 88
6.2.1. Проверка исключений 88
6.2.2. Обработка исключений 90
6.2.3. Исключения во вспомогательных функциях 91

Глава 7. Интерфейс вызова 95
7.1. Создание виртуальной машины Java 95
7.2. Компоновка native-приложений с виртуальной машиной Java 98
7.2.1. Компоновка с определенной виртуальной машиной Java 98
7.2.2. Компоновка с неизвестными виртуальными машинами 99
7.3. Подключение native-потоков 101

Глава 8. Дополнительные функции интерфейса JNI 105
8.1. Интерфейс JNI и потоки 105
8.1.1. Ограничения 105
8.1.2. Вход и выход из монитора 106
8.1.3. Реализация методов wait и notify в интерфейсе JNI 108
8.1.4. Получение указателя JNIEnv в произвольных контекстах 109
8.1.5. Соответствие с моделями потоков 110
8.2. Написание кода с многоязычной поддержкой 112
8.2.1. Создание jstrings из native-строк 112
8.2.2. Преобразование jstrings в native-строки 113
8.3. Регистрация native-методов 114
8.4. Обработчики загрузки и выгрузки библиотеки 115
8.4.1. Обработчик JNI_OnLoad 115
8.4.2. Обработчик JNI_OnUnload 117
8.5. Поддержка отражения 118
8.6. JNI-программирование в языке C++ 119

Глава 9. Использование существующих native-библиотек 121
9.1. Преобразование «один к одному» 122
9.2. Общие заглушки 125
9.3. Сравнение методов создания оболочек для native-функций 129
9.4. Реализация общих заглушек 130
9.4.1. Класс CPointer 130
9.4.2. Класс CMalloc 130
9.4.3. Класс CFunction 131
9.5. Peer-классы 136
9.5.1. Применение peer-классов в Java-платформе 137
9.5.2. Освобождение памяти native-структур 138
9.5.3. Обратные указатели на peer-объекты 140

Глава 10. Проблемные и слабые места 145
10.1. Проверка ошибок 145
10.2. Передача недействительных аргументов JNI-функциям 145
10.3. Отличие jclass от jobject 146
10.4. Приведение аргументов к типу jboolean 146
10.5. Границы между Java-приложением и native-кодом 147
10.6. Отличие идентификаторов от ссылок 149
10.7. Кэширование идентификаторов полей и методов 150
10.8. Окончания строк Unicode 152
10.9. Нарушение правил управления доступом 152
10.10. Игнорирование многоязычной поддержки 153
10.11. Сохранение ресурсов виртуальной машины 154
10.12. Чрезмерное создание локальных ссылок 155
10.13. Использование недействительных локальных ссылок 156
10.14. Использование указателя JNIEnv в потоках 156
10.15. Несоответствие модели потоков 156

Часть III. Спецификация 158

Глава 11. Обзор проекта JNI 159
11.1. Цели проекта JNI 159
11.2. Загрузка native-библиотек 160
11.2.1. Загрузчик классов 160
11.2.2. Загрузчики классов и native-библиотеки 162
11.2.3. Поиск native-библиотек 163
11.2.4. Ограничение безопасности типов 165

11.2.5. Выгрузка native-библиотек ... 166
11.3. Связывание native-методов ... 166
11.4. Соглашения о вызовах ... 168
11.5. Указатель интерфейса JNIEnv ... 168
11.5.1. Структура интерфейсного указателя JNIEnv ... 169
11.5.2. Преимущества интерфейсного указателя ... 170
11.6. Передача данных ... 171
11.6.1. Глобальные и локальные ссылки ... 172
11.6.2. Реализация локальных ссылок ... 172
11.6.3. Слабые глобальные ссылки ... 173
11.7. Обращение к объектам ... 173
11.7.1. Обращение к примитивным массивам ... 174
11.7.2. Поля и методы ... 176
11.8. Ошибки и исключения ... 177
11.8.1. Отсутствие проверок программных ошибок ... 178
11.8.2. Исключения виртуальной машины Java ... 178
11.8.3. Асинхронные исключения ... 179

Глава 12. Типы интерфейса JNI ... 181

12.1. Примитивные и ссылочные типы ... 181
12.1.1. Примитивные типы ... 181
12.1.2. Ссылочные типы ... 181
12.1.3. Тип jvalue ... 183
12.2. Идентификаторы полей и методов ... 183
12.3. Форматы строк ... 183
12.3.1. Строки UTF-8 ... 183
12.3.2. Дескрипторы классов ... 184
12.3.3. Дескрипторы полей ... 185
12.3.4. Дескрипторы методов ... 186
12.4. Константы ... 186

Глава 13. Функции интерфейса JNI ... 188

13.1. Краткий обзор JNI-функций ... 188
13.1.1. Непосредственно экспортируемые функции интерфейса вызовов ... 189
13.1.2. Интерфейс JavaVM ... 189
13.1.3. Функции в native-библиотеках ... 190
13.1.4. Интерфейс JNIEnv ... 190
13.2. Спецификация JNI-функций ... 196

Алфавитный указатель ... 271

Предисловие

В этой книге описывается интерфейс JNI. Она будет полезна тем, кто интересуется следующими вопросами:

- интеграция кода, написанного на таких языках программирования, как C и C++, в Java-приложение;
- внедрение виртуальной машины Java в существующее приложение, написанное на языках программирования C и C++;
- реализация виртуальной машины Java;
- технические вопросы организации взаимодействия между различными языками, в том числе имеющие отношение к работе со сборщиком мусора и многопоточности.

Во-первых, эта книга предназначена для разработчиков. Они с легкостью могут найти в ней необходимую информацию о том, как начать работу с интерфейсом JNI, подробное описание различных JNI-функций, а также получить полезные советы по эффективному использованию JNI. Впервые интерфейс JNI появился в 1997 году. В этой книге собран двухлетний опыт коллективной работы специалистов компании Sun Microsystems, а также опыт огромного количества разработчиков технологического сообщества Java.

Во-вторых, в книге приведено обоснование дизайна различных JNI-функций. Полное понимание структуры интерфейса не только представляет собой интерес для научного сообщества, но также является необходимым условием его эффективного использования.

В-третьих, один из разделов книги посвящен полному описанию JNI-спецификации платформы Java 2. Программисты JNI могут использовать информацию из этого раздела в качестве справочного пособия. Разработчики виртуальных машин Java обязаны придерживаться данной спецификации.

Комментарии по спецификации или вопросы об интерфейсе JNI можно присылать по адресу электронной почты: jni@java.sun.com. Чтобы получить последние обновления платформы Java 2 или последней версии Java 2 SDK, посетите наш сайт http://java.sun.com. Чтобы получить обновленную информацию о Java™ Series, включая список ошибок данного пособия, а также просмотреть перечень книг, готовящихся к публикации, посетите сайт http://java.sun.com/Series.

Результатом переговоров между компанией Sun Microsystems и обладателями лицензий на технологии Java стала разработка интерфейса JNI. Отчасти интерфейс JNI был разработан на основе интерфейса Java Runtime Interface (JRI) компании Netscape, созданного Уорреном

Харрисом (Warren Harris). Активное участие в обсуждениях проекта принимали представители компаний, занимающихся лицензиями Java-технологий, такие как: Расс Арун (Russ Arun), Ян Эллисон (Ian Ellison) и Майк Тоутонги-Тейлор (Mike Toutonghi-Taylor) – представители компании Microsoft, Патрик Берд (Patrick Beard) – представитель компании Apple, Симон Нэш (Simon Nash) – представитель компании IBM, Кен Рут (Ken Root) – представитель компании Intel.

Внутренние исследования компании Sun, проводимые Дэйвом Бауэном (Dave Bowen), Джеймсом Гослингом (James Gosling), Питером Кесслером (Peter Kessler), Тимом Линдгольмом (Tim Lindholm), Марком Рейнхольдом (Mark Reinhold), Дереком Уайтом (Derek White) и Фрэнком Йеллином (Frank Yellin), также принесли большую пользу проекту JNI. Дэйв Браун (Dave Brown), Дэйв Коннелли (Dave Connelly), Джеймс Макилри (James McIlree), Бенжамин Ренод (Benjamin Renaud) и Том Родригез (Tom Rodriguez) внесли существенный вклад в усовершенствование интерфейса JNI в Java 2 SDK 1.2. Команда тестировщиков под руководством Карлы Шроер (Carla Schroer) в Новосибирске создала тесты на совместимость для JNI. В ходе работы группа выявила слабые места, в которых исходная спецификация была неточной и неполной.

Технология JNI не смогла бы развиться и реализоваться без поддержки руководства компании в лице Дэйва Бауэна (Dave Bowen), Ларри Абраамса (Larry Abrahams), Дика Нейсса (Dick Neiss), Йона Каннегаарда (Jon Kannegaard) и Алана Бараца (Alan Baratz). Я получил полную поддержку и содействие в работе над этой книгой от моего руководителя Дэйва Бауэна (Dave Bowen).

В то время когда начиналась разработка интерфейса JNI, Тим Линдгольм (Tim Lindholm), автор книги «Спецификация виртуальной машины Java» («The Java™ Virtual Machine Specification»), возглавил работу по развитию виртуальной машины Java. Он занимался разработкой виртуальной машины и native-интерфейсов, выступал за использование интерфейса JNI и привнес точность и ясность в книгу. Он также представил первоначальный эскиз обложки данного пособия.

Многие коллеги оказали помощь при создании книги. Ананд Паланисуами (Anand Palaniswamy) работал над темой общих системных сбоев и ошибок в одном из разделов 10-й главы. Джанет Коениг (Janet Koenig) внимательно изучала предварительный проект и делилась множеством полезных идей. Бэт Стирнс (Beth Stearns) написала проект ко 2-й главе, основываясь на учебном онлайн-посо-

бии по интерфейсу JNI. Я получил ценные советы по созданию этой книги от Крега Дж. Борделона (Craig J. Bordelon), Мишеля Брандеджа (Michael Brundage), Мэри Дэйджфорд (Mary Dageforde), Джошуа Энгела (Joshua Engel) и Эллиота Хьюза (Elliott Hughes).

Благодаря Лайзе Френдли (Lisa Friendly), редактору книги «Последовательность в Java» («The Java™ Series»), эта книга была написана и опубликована. Кен Арнольд (Ken Arnold), автор книги «Язык программирования Java» («The Java™ Programming Language»), был первым, кто предложил написать книгу по интерфейсу JNI. Я очень признателен сотрудникам издательства Addison-Wesley в лице Майка Хендриксона (Mike Hendrikson) и Марины Ланг (Marina Lang) за их помощь и терпение, проявленные в процессе работы над книгой. Дайан Фрид (Diane Freed) курировала весь ход работы, начиная с момента написания до момента публикации книги.

За последние несколько лет я имел удовольствие работать с группой талантливых и преданных своему делу людей, которые занимаются разработкой программного обеспечения Java в компании Sun Microsystems, в частности с членами команд разработчиков виртуальных машин HotSpot и Sun Labs. Эта книга посвящена всем им.

Шенг Лиэнг
май 1999 года

Часть

Введение и учебный материал

Глава 1

Введение

Интерфейс JNI (Java Native Interface) является мощным инструментом платформы Java. Приложения, использующие интерфейс JNI, могут включать в себя как native-код, написанный на языках программирования C и C++, так и код, написанный на языке программирования Java. Интерфейс JNI позволяет программистам использовать всю мощь Java-платформы без отказа от своих предыдущих наработок. Поскольку интерфейс JNI является частью платформы Java, программисту достаточно будет один раз решить проблему взаимодействия имеющегося кода и Java, и он может рассчитывать на то, что это решение будет работать со всеми реализациями Java-платформ.

Эта книга является одновременно как руководством по программированию, так и справочником по JNI. Книга состоит из трех основных частей, которые включают в себя следующие главы:

- глава 1 представляет собой введение, которое дает читателям общее представление об интерфейсе JNI. В главе 2 рассматривается интерфейс JNI на простом примере. Эта глава представляет собой учебное пособие для начинающих, которые еще не знакомы с интерфейсом JNI;
- главы 3–10 являются справочным пособием по программированию, которое дает широкое представление о ряде функций интерфейса JNI. Небольшие наглядные примеры позволят выделить различные характерные особенности интерфейса JNI и продемонстрировать методы, полезные при программировании с использованием JNI;
- в главах 11–13 содержится подробное описание всех типов интерфейса JNI и его функций. Главы построены таким образом, что могут служить справочным пособием.

Книга рассчитана на широкий круг читателей, интересующихся интерфейсом JNI. Целевой аудиторией учебного раздела книги и раздела по программированию являются начинающие программисты, что же касается опытных разработчиков и специалистов по интерфейсу JNI, то их могут больше заинтересовать справочные разделы

книги. Основными читателями книги, вероятнее всего, станут разработчики, использующие JNI для написания приложений. Данная книга адресована в большей степени программистам, использующим JNI. В меньшей же степени книга будет интересна разработчикам самого интерфейса JNI или конечным пользователям приложений, которые написаны с использованием интерфейса JNI.

Предполагается, что читатель имеет базовые знания языков программирования Java, C и C++. Если же знания в данных областях отсутствуют или недостаточны, то будет полезно ознакомиться с такими книгами, как: «Язык программирования Java™», второе издание (The Java™ Programming Language, Second Edition), написанной Кэном Арнольдом (Ken Arnold) и Джеймсом Гослингом (James Gosling) (издательство «Addison-Wesley», 1998), «Язык программирования С», второе издание (The C Programming Language, Second Edition), написанной Брайаном Керниганом (Brian Kernighan) и Деннисом Ритчи (Dennis Ritchie) (издательство «Prentice Hall», 1988), а также «Язык программирования С++», третье издание (The C++ Programming Language, Third Edition), написанной Бьёрном Страуструпом (Bjarne Stroustrup) (издательство «Addison-Wesley», 1997).

Заключительный раздел данной главы посвящен происхождению, роли и этапам развития интерфейса JNI.

1.1. Платформа Java и вычислительная среда

Поскольку данная книга охватывает приложения, написанные на языке программирования Java, а также на языках программирования native-уровня (С, С++ и т. д.), то для начала необходимо определить состав программного окружения этих языков.

Java-платформа является программной средой, состоящей из виртуальной машины Java (ВМ) и интерфейса программирования приложений Java (Java API). Java-приложения пишутся на языке программирования Java и компилируются в машинно-независимый код (двоичный формат класса). Класс может быть выполнен на любой виртуальной машине Java. Java API состоит из набора определенных классов. Любая реализация Java-платформы гарантированно поддерживает язык программирования Java, виртуальную машину и API.

Понятие «вычислительная среда» (host environment) включает в себя операционную систему, под которой работает виртуальная

машина, а также набор системных (native) библиотек и инструкций процессора. *Native-приложения* пишутся на языке программирования C/C++, компилируются в бинарный код, зависящий от особенностей вычислительной среды, и связываются с native-библиотеками. Native-приложения и native-библиотеки чаще всего также зависят от особенностей вычислительной среды. Например, приложение C, написанное для одной операционной системы, не будет работать в других операционных системах.

Java-платформа, как правило, надстраивается над вычислительной средой. Например, Java Runtime Environment (JRE) является разработкой компании Sun, которая поддерживает Java-платформу на таких операционных системах, как Solaris или Windows. Java-платформа обладает рядом функций, которые позволяют приложениям быть независимыми от вычислительной среды.

1.2. Роль интерфейса JNI

При установке Java-платформы в вычислительной среде может возникнуть необходимость разрешить Java-приложениям работать непосредственно с native-кодом, написанным на других языках. Программисты стали адаптировать Java-платформу для построения приложений, которые традиционно разрабатывались как на языках програмирования C и C++, так и на языке Java. Однако из-за имеющихся наработок Java-приложения будут сосуществовать с кодом C и C++ еще долгое время.

JNI является мощным инструментом, который позволяет воспользоваться преимуществами платформы Java, при этом по-прежнему используя код, написанный на других языках. Будучи частью виртуальной машины Java, интерфейс JNI представляет собой двусторонний интерфейс, который дает возможность Java-приложениям вызывать native-код, и наоборот: native-коду пользоваться возможностями Java. Схема 1.1 иллюстрирует роль интерфейса JNI.

Интерфейс JNI был создан для решения проблем, возникающих при необходимости совмещения Java-приложений с native-кодом. Будучи двусторонним, интерфейс JNI может поддерживать два типа native-кода: native-библиотеки и native-приложения. Приведем пример:

- вы можете использовать JNI для написания *native-методов*, которые позволяют Java-приложениям вызывать функции, содержащиеся в native-библиотеках. Java-приложения вызывают native-методы точно так же, как они вызывают методы,

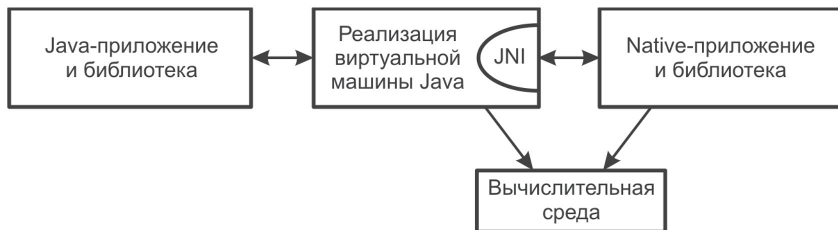

Схема 1.1 ❖ Роль интерфейса JNI

написанные на языке программирования Java. При этом native-методы реализуются при помощи другого языка и находятся в native-библиотеках;

- интерфейс JNI поддерживает *интерфейс запуска* (invocation interface), который позволяет вам встраивать виртуальную машину Java в native-приложения. Native-приложения могут связываться с native-библиотекой, которая реализует виртуальную машину Java, чтобы затем использовать интерфейс запуска для вызова компонентов программного обеспечения, написанных на языке программирования Java. Например, веб-браузер, написанный на языке программирования C, может выполнять загруженные апплеты во встроенной виртуальной машине Java.

1.3. Последствия использования интерфейса JNI

Необходимо помнить, что приложение, используя интерфейс JNI, рискует потерять два преимущества Java-платформы.

Во-первых, Java-приложения, зависящие от интерфейса JNI, не смогут работать в разных вычислительных средах. И хотя та часть приложения, которая написана на языке программирования Java, будет работать везде, в обязательном порядке необходимо перекомпилировать ту часть приложения, которая написана на native-языке.

Во-вторых, язык программирования Java является безопасным по типу (type-safe) и защищенным, в то время как native-языки (такие как C или C++) к таковым не относятся. Поэтому необходимо быть предельно внимательным при написании приложений, использующих JNI. Неправильно написанный native-метод может привести к сбою всего приложения. Именно по этой причине перед вызовом функций JNI в Java-приложениях выполняются проверки на безопасность.

Как правило, следует выстраивать приложение таким образом, чтобы native-методы были определены как можно в меньшем количестве классов. Это будет способствовать полной изоляции native-кода от оставшейся части приложения.

1.4. Случаи использования интерфейса JNI

Перед тем как начинать разрабатывать новый проект с использованием интерфейса JNI, следует вернуться на шаг назад и выяснить, существуют ли альтернативные и более подходящие решения. Как было упомянуто в предыдущем разделе, приложения, использующие JNI, имеют определенные недостатки, по сравнению с теми приложениями, которые написаны строго на языке программирования Java. Например, теряется гарантия безопасности типов языка программирования Java. Ряд альтернативных подходов Java также позволяет приложениям взаимодействовать с кодом, написанным на других языках. Например:

- Java-приложение может взаимодействовать с native-приложением через TCP/IP-соединение или с помощью других механизмов межпроцессного взаимодействия IPC (Inter-Process Communication);
- Java-приложение может подключаться к базам данных через JDBC™ API (Java DataBase Connectivity – соединение с базами данных на Java);
- Java-приложение может воспользоваться технологиями распределенных объектов, таких как Java IDL API (Java Interface Definition Language).

Общая характерная черта этих альтернативных решений заключается в том, что Java-приложение и native-код находятся в разных процессах, а в некоторых случаях на разных машинах. Разделение на процессы предполагает важное преимущество. Защита адресного пространства, поддерживаемая процессами, обеспечивает высокую степень локализации неисправности: сбой native-приложения не сразу остановит Java-приложение, с которым оно связано по протоколу TCP/IP.

Однако иногда возникает необходимость взаимодействия Java-приложения с native-кодом, который находится в том же самом процессе. В этом случае интерфейс JNI становится полезным. Рассмотрим следующие примеры:

- предположим, что приложению потребуется выполнить, к примеру, ряд специальных файловых операций, которые не под-

держивает Java API. Тогда обработка файлов посредством другого процесса станет громоздкой и неэффективной;

- возможно, возникла необходимость получить доступ к существующей native-библиотеке, но при этом недопустимо появление издержек на копирование и передачу данных через другие процессы. Загрузка native-библиотеки в этот же процесс будет более эффективным решением;
- выполнение приложения может охватывать несколько процессов, приводя тем самым к неприемлемым затратам памяти. Это обычно происходит, если процессы приложения находятся на одной и той же клиентской машине. Загрузка native-библиотеки в существующий процесс приложения потребует меньших системных ресурсов, чем запуск нового процесса и загрузка библиотеки в этот процесс;
- возможно, появилась необходимость реализовать небольшой участок критичного по времени кода на языке низкого уровня, таком как ассемблер. Если 3D-приложение тратит значительную часть времени на обработку графики, то следует написать основную часть графической библиотеки на ассемблере для достижения максимальной эффективности.

Таким образом, интерфейс JNI используется в случаях, когда возникает необходимость взаимодействия Java-приложения с native-кодом, который находится в том же самом процессе.

1.5. Развитие интерфейса JNI

Необходимость взаимодействия Java-приложений с native-кодом появилась с самых первых дней существования платформы Java. Первый релиз платформы Java – Java Development Kit (JDK™) версии 1.0 – включал интерфейс взаимодействия с native-кодом (Native Method Invocation – NMI), который позволил Java-приложениям вызывать функции, написанные на других языках, таких как C и C++. Многие сторонние приложения, а также реализации библиотек классов Java (в том числе такие, как java.lang, java.io и java.net), опирались на собственный интерфейс для доступа к системным функциям.

К сожалению, интерфейс взаимодействия с native-кодом в JDK версии 1.0 имел две основные проблемы:

- во-первых, native-код обращался к полям объектов как к членам структур языка C. Однако при этом спецификация вир-

туальной машины Java не определяет, каким образом объекты располагаются в памяти. Если реализация данной виртуальной машины Java размещает объекты иным способом, нежели это определено в интерфейсе, то необходимо перекомпилировать native-библиотеки;

- во-вторых, интерфейс взаимодействия с native-методом в JDK версии 1.0 опирается на старый сборщик мусора (gabage collector), так как native-методы могут содержать прямые указатели на объекты в виртуальной машине. Любая реализация виртуальной машины, которая использует более продвинутые алгоритмы сборки мусора, не сможет поддерживать данный интерфейс.

Интерфейс JNI был разработан как раз для решения этих проблем. Этот интерфейс поддерживается всеми реализациями виртуальных машин Java на различных платформах. Также интерфейс JNI предоставляет следующие возможности:

- любая виртуальная машина может поддерживать больший объем native-кода;
- поставщикам инструментов разработки не придется иметь дело с различными видами интерфейсов взаимодействия с native-кодом;
- самым значимым является то, что прикладным программистам достаточно написать одну версию своего native-кода, и эта версия будет работать на любых реализациях виртуальной машины Java.

Впервые поддержка JNI появилась в JDK версии 1.1. Однако в JDK 1.1 для обеспечения Java API все еще используется старый способ (такой же, как и в JDK версии 1.0) вызова native-методов. И только в Java 2 SDK (Software Development Kit) версии 1.2, известной как JDK версии 1.2, native-методы были переписаны в соответствии со стандартом JNI.

JNI – это native-интерфейс, поддерживаемый всеми реализациями виртуальной машины Java. Начиная с JDK версии 1.1, необходимо писать программы с использованием JNI. Старый интерфейс вызова native-кода поддерживается в Java 2 SDK версии 1.2, но он не будет поддерживаться в виртуальной машине Java в будущем.

Java 2 SDK версии 1.2 содержит ряд усовершенствований JNI, имеющих обратную совместимость. Все последующие изменения JNI в будущем также будут поддерживать полную бинарную совместимость.

1.6. Примеры программ

В этой книге приведено большое количество примеров программ, которые демонстрируют возможности JNI.

В основном демонстрационные программы состоят из нескольких сегментов кода, написанного на языке программирования Java, и native-кода, написанного на языках C или C++. Иногда native-код ссылается на системно-зависимые функции Solaris или Win32. В книге также приведены примеры создания JNI-программ с использованием утилит командной строки, таких как javah, поставляемых вместе с JDK и Java 2 SDK.

Необходимо помнить, что использование JNI не ограничивается конкретной вычислительной средой или определенными инструментами разработки приложений. Основное внимание книги сосредоточено не на инструментах, использующихся для сборки и запуска кода, а именно на его написании. Утилиты командной строки, поставляемые в комплекте с JDK и Java 2 SDK, являются достаточно примитивными. Сторонние инструменты могут предложить лучший способ создания приложений, использующих JNI. Рекомендуется также обращаться к информации о JNI, поставляемой со средствами разработки.

Глава 2

Приступая к работе

В этой главе рассматривается простой пример использования интерфейса JNI. Мы напишем Java-приложение, которое будет вызывать функцию C для вывода фразы «`Hello World!`».

2.1. Общее представление

В схеме 2.1 иллюстрируется процесс использования JDK или Java 2 SDK для написания простого Java-приложения, которое вызывает функцию C для вывода фразы «`Hello World!`». Этот процесс состоит из следующих шагов:

1. Создать класс (`HelloWorld.java`), объявляющий native-метод.
2. Использовать `javac` для компиляции исходного файла `HelloWorld`. В результате получаем файл класса `HelloWorld.class`. Компилятор `javac` поставляется вместе с JDK или Java 2 SDK.
3. Использовать `javah -jni` для создания заголовочного файла C, содержащего прототип функции, которая реализует native-метод. Инструмент `javah` поставляется вместе с JDK или Java 2 SDK.
4. Написать код на языке C (`HelloWorld.c`), реализующий native-метод.
5. Скомпилировать C-код в native-библиотеку `HelloWorld.dll` или `libHelloWorld.so`. При этом будут использоваться C-компилятор и компоновщик (linker), доступные в данной вычислительной среде.
6. Запустить программу `HelloWorld`, используя имеющийся интерпретатор `java` runtime. И файл класса (`HelloWorld.class`), и native-библиотека (`HelloWorld.dll` или `libHelloWorld.so`) будут загружены в момент выполнения.

Ниже эти этапы рассматриваются более подробно.

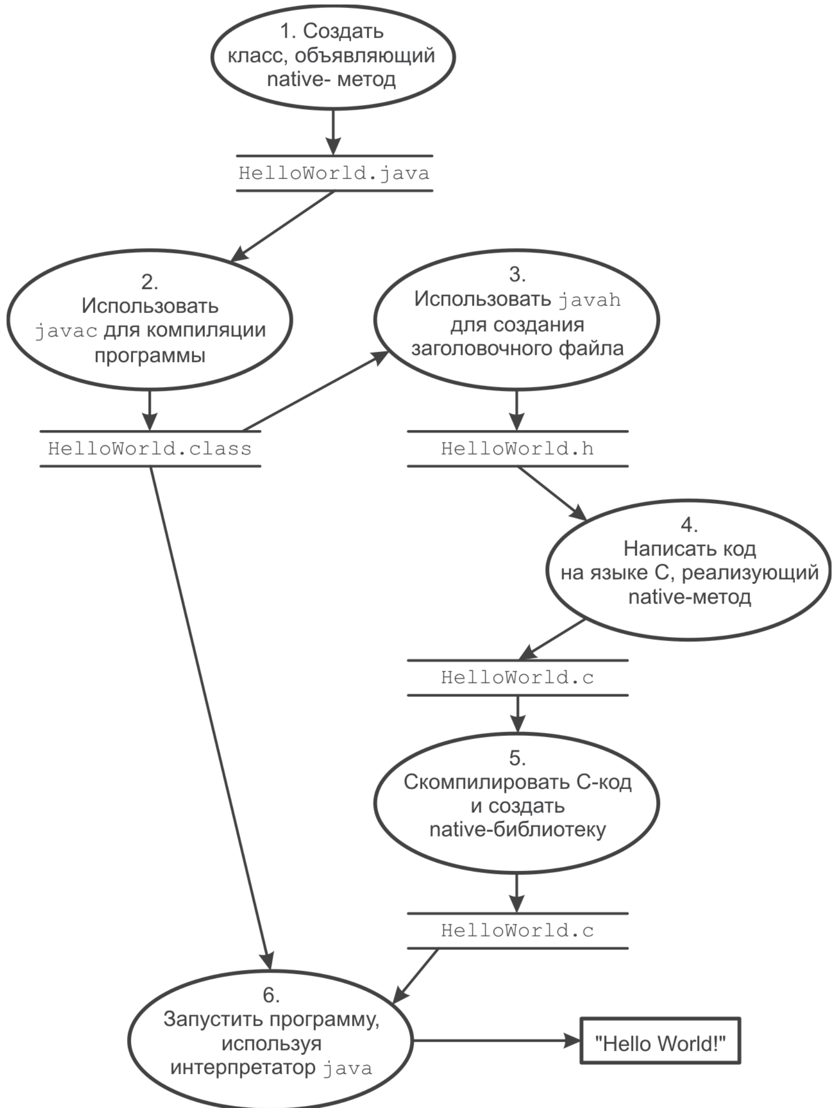

Схема 2.1 ❖ Процесс написания и запуска программы «Hello World!»

2.2. Объявление native-метода

Начнем с написания следующей программы на языке программирования Java. Программа определяет класс `HelloWorld`, который содержит native-метод `print`.

```
class HelloWorld {
    private native void print();
    public static void main(String[] args) {
        new HelloWorld().print();
    }
    static {
        System.loadLibrary("HelloWorld");
    }
}
```

Определение класса `HelloWorld` начинается с объявления native-метода `print`. За этим следует метод *main*, который создает экземпляр класса `HelloWorld` и вызывает native-метод `print` для данного экземпляра. В последней части определения класса находится статический инициализатор, который загружает native-библиотеку, содержащую реализацию native-метода `print`.

Существуют два различия между объявлением native-методов, таких как `print`, и объявлением обычных методов языка программирования Java. Объявление native-метода должно содержать модификатор `native`. Модификатор `native` указывает на то, что этот метод реализуется на другом языке. Кроме того, объявление native-метода завершается точкой с запятой (символ окончания оператора), поскольку в самом классе отсутствует реализация native-методов. Реализацию метода `print` мы выполним в отдельном файле С.

Прежде чем вызвать native-метод `print`, необходимо загрузить в память процесса native-библиотеку, содержащую реализацию метода. В данном случае мы загружаем native-библиотеку в статическом инициализаторе класса `HelloWorld`. Виртуальная машина Java автоматически запускает статический инициализатор до вызова любых методов класса `HelloWorld`, таким образом гарантируя, что native-библиотека будет загружена до того, как будет вызван native-метод `print`. Мы определили статический метод `main`, для того чтобы иметь возможность выполнить код класса `HelloWorld`. Обращение к native-методу `print` в `HelloWorld.main` происходит точно так же, как обращение к обычному методу.

`System.loadLibrary` получает имя библиотеки, находит native-библиотеку, которая соответствует этому имени, и загружает native-биб-

Конец ознакомительного фрагмента.
Приобрести книгу можно
в интернет-магазине
«Электронный универс»
e-Univers.ru