Скрытые возможности Java: декомпиляция, исправление и обратная разработка

Русский перевод книги Алекса Калиновского о декомпиляции, обфускации, исправлении байт-кода, исследовании среды выполнения и защите приложений Java.

Больше крутых материалов в моём телеграм-канале @grogrigs

Глава 1. Начало работы

В этой главе

• Обзор методов: когда и зачем применять каждый из них
• Повышение производительности с помощью файловых менеджеров
• Назначение и структура демонстрационного приложения
• Короткий тест
• Краткие итоги

Обзор методов: когда и зачем применять каждый из них

В таблице 1.1 кратко представлены методы, которые подробнее рассматриваются в соответствующих главах. Используйте её как путеводитель, который поможет начать работу с этой книгой.

Таблица 1.1. Обзор методов

Глава	Метод	Для чего полезен
2	Декомпиляция классов	Восстановление утраченного исходного кода. Изучение реализации какой-либо функции или приёма. Устранение неполадок в недокументированном коде. Срочное исправление ошибок в промышленном или стороннем коде. Оценка возможных способов взлома вашего кода.
3	Обфускация классов	Защита байт-кода от декомпиляции. Защита содержащейся в байт-коде интеллектуальной собственности. Предотвращение взлома приложений.
4	Взлом непубличных методов и переменных класса	Получение доступа к существующим, но не предоставленным извне возможностям. Изменение значений внутренних переменных.
5	Замена и исправление классов приложения	Изменение реализации класса без повторной сборки всей библиотеки. Изменение функций стороннего приложения или каркаса.

Таблица 1.1. Обзор методов (продолжение)

Глава	Метод	Для чего полезен
6	Эффективная трассировка	Создание приложений, которые легко сопровождать и в которых легко устранять неполадки. Изучение внутреннего устройства приложения. Добавление отладочных сведений в существующие приложения для понимания деталей реализации.
7	Управление безопасностью Java	Добавление или снятие ограничений на доступ к критически важным системным ресурсам.
8	Исследование среды выполнения	Определение значений системных свойств. Определение сведений о системе, например числа процессоров и ограничений памяти. Определение конфигурации сети.
9	Взлом кода с помощью нестандартных отладчиков	Взлом приложений, в которых отсутствует качественная трассировка. Анализ потока управления многопоточных приложений. Взлом обфусцированных приложений.
10	Использование профилировщиков для анализа приложения во время выполнения	Исследование использования кучи и частоты сборки мусора с целью повысить производительность. Просмотр распределения объектов и ссылок для поиска и устранения утечек памяти. Исследование распределения и синхронизации потоков для поиска проблем с блокировками и гонками данных и повышения производительности. Исследование приложения во время выполнения для лучшего понимания его внутренней структуры.
11	Нагрузочное тестирование для поиска и устранения проблем масштабируемости	Создание автоматизированных тестовых сценариев, имитирующих нагрузку на систему. Анализ того, насколько хорошо приложение отвечает требованиям к уровню обслуживания, таким как масштабируемость, доступность и аварийное переключение.
12	Обратная разработка приложений	Взлом элементов пользовательского интерфейса, таких как сообщения, предупреждения, подсказки, изображения, значки, меню и цвета.
13	Методы подслушивания	Перехват HTTP-обмена между браузером и веб-сервером. Перехват обмена между клиентом и сервером RMI. Перехват SQL-инструкций и значений из драйвера JDBC.

Таблица 1.1. Обзор методов (продолжение)

Глава	Метод	Для чего полезен
14	Управление загрузкой классов	Реализация собственного загрузчика классов для управления способом загрузки классов и источником, из которого они загружаются. Применение собственного загрузчика классов для инструментирования байт-кода на лету. Программное создание классов во время выполнения.
15	Замена и исправление основных классов Java	Изменение реализации системных классов для изменения основного поведения. Расширение основных возможностей JDK в соответствии с потребностями приложения. Исправление ошибок в реализации JDK.
16	Перехват потока управления	Корректная реакция на системные ошибки, например нехватку памяти и переполнение стека. Перехват вывода в `System.out` и `System.err`. Перехват вызовов `System.exit()`. Реакция на завершение работы JVM. Перехват любого вызова метода, выделения объекта или события жизненного цикла потока посредством JVMPI.
17	Понимание и изменение байт-кода	Изменение реализации класса на уровне байт-кода. Программное создание байт-кода. Инструментирование байт-кода для добавления новой логики.
18	Полный контроль посредством исправления нативного кода	Исправление реализации нативных функций. Расширение поведения JVM на самом низком уровне.
19	Защита коммерческих приложений от взлома	Защита конфиденциальной информации с помощью средств криптографии Java. Обеспечение целостности данных с помощью цифровых подписей. Реализация безопасной лицензионной политики для разблокировки функций коммерческих приложений.

Повышение производительности с помощью файловых менеджеров

Рассматриваемые здесь методы призваны повысить производительность разработки. В конечном счёте именно качество и производительность отличают опытных программистов от начинающих, а поскольку эта книга должна помочь читателям стать специалистами, я считаю своим долгом представить несколько инструментов повышения производительности. Как взлом, так и обычная разработка требуют работы с файлами и каталогами, и правильно выбранный инструмент способен значительно её облегчить. Разумеется, только вам решать, пользоваться ли тем или иным инструментом. Следует помнить, что большинство инструментов требуют первоначальных затрат на установку, настройку и обучение, не говоря уже о возможной плате за лицензию. Однако, как и в случае с большинством инструментов, эти вложения окупаются очень быстро.

Мы рассмотрим две развитые альтернативы сочетанию Windows Explorer, Notepad/Text Editor и `CMD.EXE`. Основное внимание будет уделено Windows, поскольку именно на этой платформе ведётся большая часть разработки на Java, однако средства повышения производительности могут существовать и для других платформ. Начинать книгу по углублённому изучению Java с разговора о Notepad и `CMD.EXE` может показаться нелепым, но многие разработчики, которых я встречал, всё ещё пользуются ими, поэтому я хочу предложить лучшую альтернативу.

Windows Explorer - простая оболочка, понятная обычным пользователям, однако она не способна помочь с задачами, которые приходится выполнять программисту. Самый простой пример - создание и запуск файла `.bat`. При использовании стандартного интерфейса Windows пришлось бы перейти в нужный каталог, с помощью мыши пройти несколько диалоговых окон для создания нового файла, а затем открыть этот файл в Notepad и отредактировать его. Запустить файл можно было бы двойным щелчком, но весь вывод и сообщения об ошибках исчезли бы вместе с окном CMD, которое Explorer открыл автоматически. Поэтому лучше открыть `CMD.EXE`, перейти в каталог и уже оттуда запустить файл. В итоге приходится иметь дело с тремя открытыми окнами, которые не синхронизированы и никак не связаны между собой. Более удобная альтернатива - встроенное программное обеспечение для управления файлами, объединяющее интерфейс навигации по каталогам, текстовый редактор, командную строку для запуска сценариев, поддержку архивов и множество функций, упрощающих повседневные задачи.

FAR и Total Commander

File and Archive Manager (FAR) и Total Commander - развитые файловые менеджеры для Windows, история которых восходит ко временам DOS и Norton Commander. Они распространяются как условно-бесплатные программы, поэтому ими можно пользоваться без ограничения срока до тех пор, пока вы не будете готовы зарегистрировать их за небольшую плату. В них предусмотрено множество функций для поиска файлов, изменения файловых атрибутов и работы с несколькими файлами и каталогами. Встроенная поддержка сети и FTP позволяет показывать удалённые FTP-сайты на панели, которая выглядит точно так же, как панель локального каталога. В FAR есть мощный встроенный редактор, который можно настроить для цветовой подсветки. Обе среды предлагают обширные наборы сочетаний клавиш, а FAR поддерживает подключаемые модули. Оба инструмента позволяют просматривать содержимое архивов, например файлов JAR и Zip, на панели так же, как содержимое вложенного каталога. Благодаря этому программы наподобие WinZip становятся ненужными. Более того, пользователю не приходится иметь дело с разными текущими каталогами и интерфейсами, как при работе с неинтегрированными программами. В таблице 1.2 перечислены возможности FAR и Total Commander и приведено их непосредственное сравнение.

Таблица 1.2. Сравнение FAR и Total Commander

Возможность	FAR	Total Commander
Создание, копирование, просмотр, редактирование и удаление файлов и папок	Отлично	Отлично
Внутреннее и внешнее средство просмотра или редактор	Отлично	Хорошо (нет внутреннего редактора)
Непосредственный просмотр содержимого архивов (JAR, Zip и т. п.)	Отлично	Отлично
Широкие возможности настройки функций и интерфейса	Отлично	Хорошо
Внешний вид и поведение в стиле Windows	Плохо	Хорошо
Настраиваемые пользовательские меню	Отлично	Отлично
Встроенная поддержка сети Windows	Да	Да
Встроенный клиент FTP	Да	Да
Сочетания клавиш	Отлично	Хорошо
Фильтры имён файлов	Отлично	Отлично
Быстрый просмотр	Удовлетворительно	Отлично
История команд, папок, просмотра и редактирования	Отлично	Хорошо
Настраиваемые сопоставления файлов	Отлично	Хорошо
Подсветка файлов	Отлично	Отлично
Объём занимаемой памяти	4-14 Мбайт	4-10 Мбайт
API подключаемых модулей и наличие различных модулей	Отлично (более 500 подключаемых модулей)	Отсутствует
Стоимость регистрации одной копии	25 долларов	28 долларов
Общая оценка	Отлично	Хорошо

Хотя оба инструмента служат более удобной альтернативой Windows Explorer, дополненному другими программами, FAR оказывается мощнее. Даже в стандартной комплектации он предоставляет больше функций и сильнее повышает производительность, чем Total Commander. Кроме того, благодаря более чем 500 подключаемым модулям, написанным другими разработчиками, его возможности практически безграничны. Недостаток FAR - не слишком привлекательный пользовательский интерфейс, хотя к нему можно привыкнуть. Total Commander, показанный на рисунке 1.1, больше похож на Windows Explorer; если вам не нужны предельная настраиваемость и функциональность, он может оказаться лучшим выбором.

Какими бы ни были ваши предпочтения, обязательно попробуйте встроенный файловый менеджер, даже если поначалу он покажется сложным. В долгосрочной перспективе это окупится!

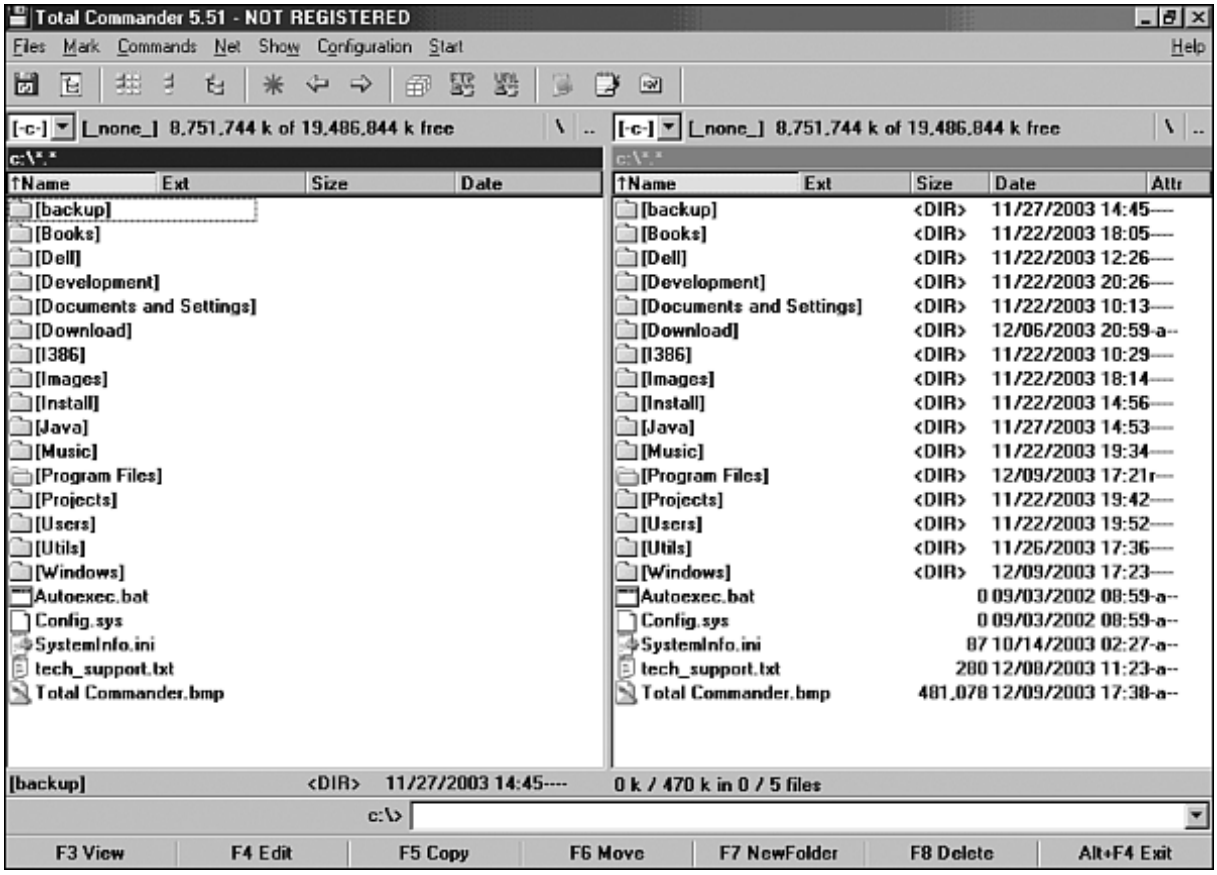

Рисунок 1.1. Total Commander

Интегрированные среды разработки Java

Большинство методов из этой книги не требуют писать много кода, а с таким инструментом, как FAR, можно без труда выполнить все необходимые задачи. Однако интегрированные среды разработки (IDE) заметно упрощают программирование, поэтому в этом разделе дан краткий обзор ведущих IDE и рекомендация по выбору среды.

Сегодня применительно к IDE вопрос состоит не в том, нужно ли вообще ими пользоваться, а в том, какую IDE выбрать. Во многом выбор определяется опытом разработки и личными предпочтениями, поэтому я не стану долго обсуждать разные варианты. Две ведущие бесплатные IDE - Eclipse (eclipse.org), продвигаемая IBM, и NetBeans (netbeans.org), продвигаемая Sun. Обе хороши, хотя Eclipse пользуется несколько большей популярностью и располагает более широким сообществом последователей. Лучшие коммерческие IDE - IntelliJ IDEA, Borland JBuilder и Oracle JDeveloper.

Поскольку вы будете заниматься низкоуровневым программированием и взломом, лучше всего выбрать гибкую IDE с небольшим потреблением памяти. Лично я предпочитаю IDEA за её гибкость, интуитивно понятный интерфейс, изобилие сочетаний клавиш и функций рефакторинга. Она не бесплатна, поэтому, если вы не можете позволить себе лицензию, моей второй рекомендацией будет Eclipse.

Назначение и структура демонстрационного приложения

На протяжении большей части книги мы будем работать с одним и тем же демонстрационным приложением. Оно не отличается особой сложностью, но содержит основной набор компонентов, встречающихся в большинстве автономных программ на Java. В этом разделе описаны само приложение и его реализация.

Chat - простая реализация чата на Java, работающего по TCP/IP. Она позволяет пользователям обмениваться мгновенными сообщениями по сети. Chat хранит историю разговора и выделяет отправленные и полученные сообщения разными цветами. В приложении есть строка меню и диалоговое окно About. Chat можно запустить с помощью сценария `chat.bat` из каталога `distrib/bin`. Работающее приложение Chat показано на рисунке 1.2.

Chat реализовано с помощью Java Swing для пользовательского интерфейса и RMI для сетевого обмена. Во время работы каждый экземпляр Chat создаёт внутрипроцессный реестр RMI, который другие экземпляры используют для отправки сообщений пользователю. Пользователь должен ввести имя узла того адресата, которому он хочет отправить сообщение. При отправке сообщения Chat находит удалённый объект сервера и вызывает его метод. В целях тестирования сообщения можно отправлять на `localhost`: в этом случае одно и то же сообщение добавляется в разговор и как отправленное, и как полученное.

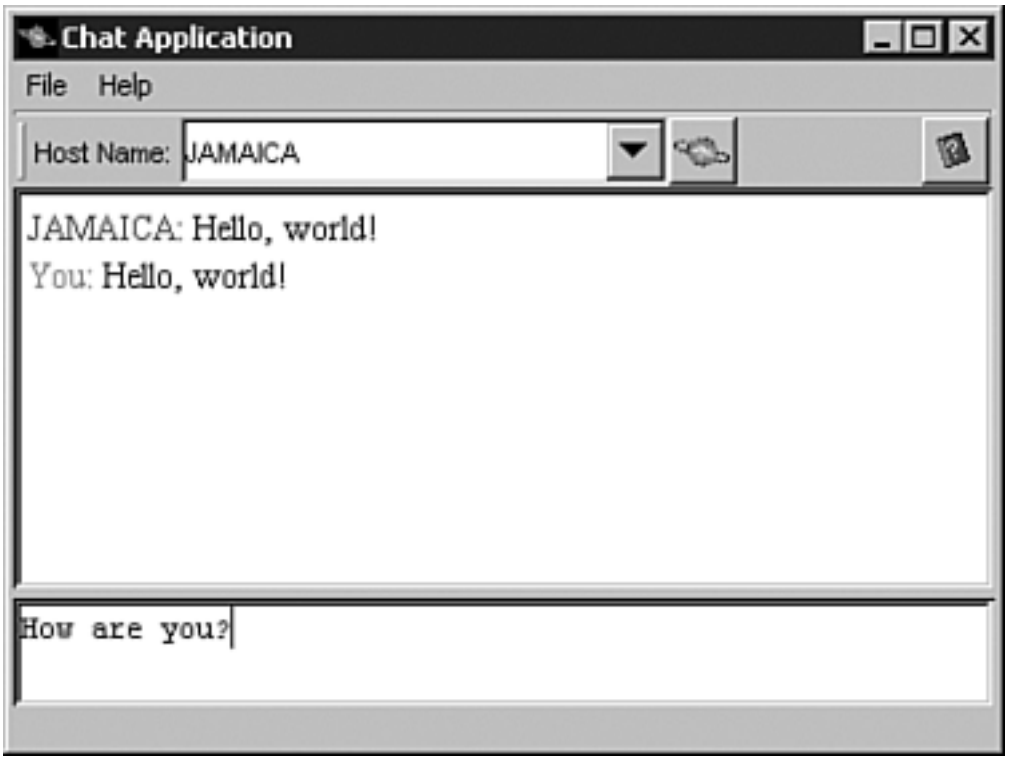

Рисунок 1.2. Приложение Chat

Диаграмма классов UML приложения Chat показана на рисунке 1.3.

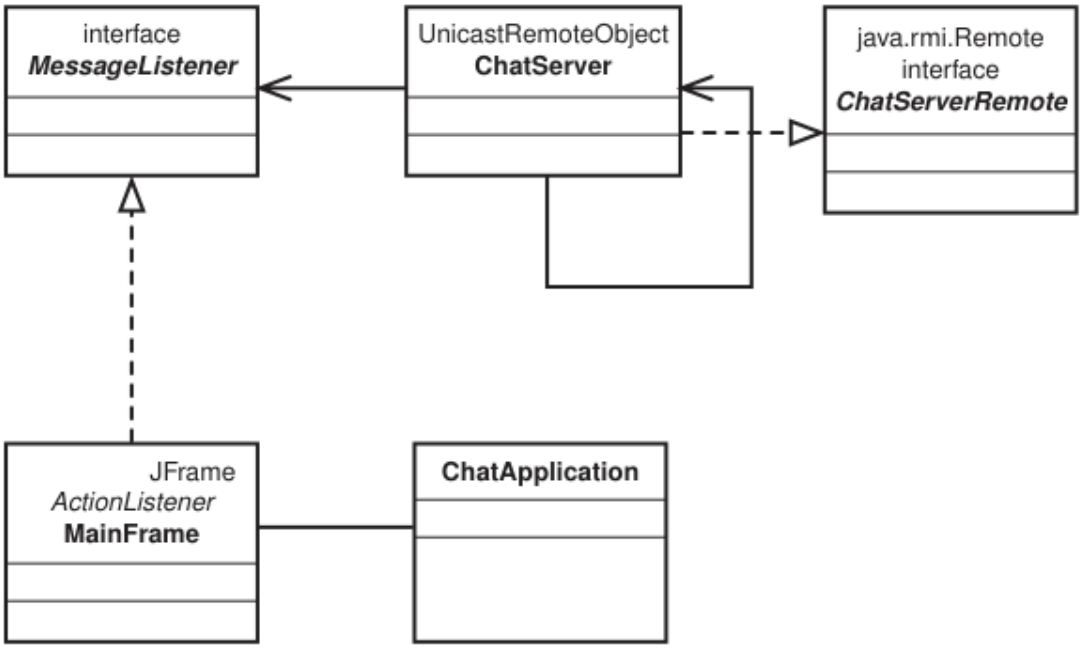

Рисунок 1.3. Диаграмма классов Chat

Структура каталогов Chat соответствует стандартам де-факто для разработки приложений на Java. «Домашняя» папка каталогов приложения называется `CovertJava`. Содержащиеся в ней подкаталоги перечислены в таблице 1.3.

Таблица 1.3. Структура каталогов приложения Chat

Имя каталога	Описание
`bin`	Содержит сценарии, а также сценарии разработки и тестирования
`build`	Содержит файл Ant `build.xml` и другие файлы, связанные со сборкой
`classes`	Каталог вывода компилятора для файлов `.class`
`distrib`	Содержит приложение в форме дистрибутива
`distrib\bin`	Содержит сценарии запуска приложения
`distrib\conf`	Содержит файлы конфигурации, например файлы политики Java
`distrib\lib`	Содержит библиотеки, используемые для запуска приложения
`distrib\patches`	Содержит исправления классов
`lib`	Содержит библиотеки, используемые для сборки приложения
`src`	Содержит исходные файлы приложения

Приложение Chat можно собрать с помощью Ant, используя файл `build.xml` из каталога `build`.

Короткий тест

1. Какие методы можно использовать, чтобы узнать о внутренней реализации приложения?

2. Какие методы можно использовать, чтобы изменить внутреннюю реализацию приложения?

3. Какие методы можно использовать, чтобы перехватить обмен данными между клиентом и сервером?

4. Какие приложения Windows заменяют FAR и Total Commander?

Краткие итоги

• Различные методы, представленные в этой книге, позволяют изучать внутреннюю реализацию, чтобы взломать приложение или работать с JDK на системном уровне.

• Интегрированные приложения для управления файлами повышают производительность и стоят вложенных средств.

Глава 2. Декомпиляция классов

> «Когда ничто другое не помогает, прочитайте руководство».
>
> *Законы Мерфи о технике*

В этой главе

• Когда следует выполнять декомпиляцию
• Лучшие декомпиляторы
• Декомпиляция класса
• Благодаря чему возможна декомпиляция?
• Возможные проблемы с декомпилированным кодом
• Короткий тест
• Краткие итоги

Когда следует выполнять декомпиляцию

В идеальном мире декомпиляция, вероятно, была бы не нужна, за исключением тех случаев, когда требуется узнать, как другие люди, не любящие писать хорошую документацию, реализовали определённую возможность. Однако в реальном мире нередко возникают ситуации, в которых непосредственное обращение к исходному коду оказывается лучшим, если не единственным, решением. Вот некоторые причины выполнить декомпиляцию:

• восстановление случайно утраченного исходного кода;
• изучение реализации какой-либо функции или приёма;
• устранение неполадок в приложении или библиотеке, для которых нет хорошей документации;
• срочное исправление ошибок в стороннем коде, исходный текст которого отсутствует;
• изучение способов защиты собственного кода от взлома.

Декомпиляция восстанавливает исходный код из байт-кода Java. Это процесс, обратный компиляции, и он возможен благодаря стандартной, хорошо документированной структуре байт-кода. Подобно тому как компилятор создаёт байт-код из исходного текста, декомпилятор позволяет получить исходный текст из имеющегося байт-кода. Декомпиляция - мощный способ изучения логики реализации при отсутствии документации и исходного кода. Именно поэтому многие поставщики программ прямо запрещают декомпиляцию и обратную разработку в лицензионном соглашении. Если вы не уверены в законности своих действий, обязательно проверьте лицензионное соглашение или получите явное разрешение поставщика.

Кто-то может возразить, что прибегать к таким крайним мерам, как декомпиляция, не следует и что поддержку и исправление ошибок нужно оставлять поставщикам байт-кода. Но в профессиональной среде разработчик приложения отвечает за безупречную работу его функций. Пользователям и руководству безразлично, находится ошибка в вашем или в стороннем коде. Им важно, чтобы проблема была устранена, и отвечать за это придётся вам. Предпочтительным способом должно быть обращение к поставщику стороннего кода. Однако в срочной ситуации, когда решение необходимо предоставить за считаные часы, умение работать с байт-кодом даст вам дополнительное преимущество перед коллегами, а возможно, принесёт ещё и премию.

Лучшие декомпиляторы

Чтобы приступить к декомпиляции, нужны подходящие инструменты. Хороший декомпилятор способен создать исходный код, который почти не уступает оригиналу, скомпилированному в байт-код. Одни декомпиляторы бесплатны, другие распространяются на коммерческой основе. Хотя я поддерживаю принципы, лежащие в основе коммерческого программного обеспечения, чтобы я стал им пользоваться, оно должно давать заметные преимущества перед бесплатными аналогами. В случае декомпиляторов мне не удалось обнаружить недостатка функций у бесплатных программ, поэтому лично я рекомендую бесплатный инструмент, например JAD или JODE. В таблице 2.1 перечислены некоторые распространённые декомпиляторы и кратко описано качество каждого из них.

> **Истории с передовой.** В Riggs Bank мы готовились ввести в эксплуатацию очень крупное и важное приложение J2EE, развёрнутое в кластере серверов приложений одного из ведущих поставщиков J2EE. Несколько команд ждали готовности промышленной среды, но по непонятной причине сервер приложений не запускался на некоторых узлах. Одна и та же установка работала на одних машинах, но завершалась сбоем на других, сообщая об ошибочном URL конфигурации. Хуже того, URL из сообщения об ошибке не удавалось найти ни в одном файле конфигурации, сценарии оболочки или переменной среды.
>
> Несколько дней мы безуспешно пытались устранить проблему, и ситуация была готова взорваться: нескольким командам грозил срыв критически важного срока. Копирование и повторная установка сервера приложений не помогли, и в конце концов мы решили найти в его библиотеках класс, создававший сообщение об ошибке. Декомпиляция этого класса и ещё нескольких использовавших его классов показала, что URL программно формировался на основе каталога установки сервера. Каталог установки определялся выполнением команды Unix `pwd`. Оказалось, что на узлах, где возникал сбой, отсутствовало разрешение на выполнение `pwd`, но вводящее в заблуждение сообщение об ошибке никак на это не указывало. Исправление разрешений заняло несколько минут, а весь процесс с момента обнаружения и декомпиляции класса - менее часа. Так надвигающаяся катастрофа превратилась в крупный успех ИТ-команды.

Приведённые URL могут со временем устареть, поэтому обычно проще всего найти домашнюю страницу декомпилятора и последнюю доступную для загрузки версию с помощью поиска Google.

Очень важный критерий - качество поддержки декомпилятором более сложных конструкций языка, например внутренних классов и анонимных реализаций. Хотя формат байт-кода почти не менялся со времён JDK 1.1, важно пользоваться декомпилятором, который авторы часто обновляют. Новые возможности языка в JDK 1.5 потребуют обновления декомпиляторов, поэтому обязательно проверяйте дату выпуска используемой версии.

Таблица 2.1. Декомпиляторы

Инструмент и оценка	Лицензия	Описание
JAD - отлично	Бесплатно для некоммерческого использования	JAD - очень быстрый, надёжный и развитый декомпилятор. Он полностью поддерживает внутренние классы, анонимные реализации и другие сложные возможности языка. Создаваемый код отличается чистотой, а директивы импорта хорошо упорядочены. В нескольких других средах декомпиляции консольная версия JAD используется как движок.
JODE - отлично	Общественная лицензия GNU	JODE - очень хороший декомпилятор, написанный на Java и доступный на SourceForge.net вместе с полным исходным кодом. Он может быть не таким быстрым и распространённым, как JAD, но даёт превосходные результаты, временами даже более чистые, чем JAD. Образовательную ценность доступного исходного кода самого декомпилятора трудно переоценить.
Mocha - удовлетворительно	Бесплатно	Mocha - первый широко известный декомпилятор, вызвавший множество юридических споров, но вместе с тем и волну энтузиазма. Mocha наглядно показал, что исходный код Java можно восстановить почти в первоначальном виде. Сообщество разработчиков приветствовало это открытие, а юридические отделы встретили его с опасением. Общедоступный код не обновлялся с 1996 года, хотя, предположительно, Borland обновила его и встроила в JBuilder.

Хотя на рынке можно найти и другие декомпиляторы, JAD и JODE, безусловно, достаточно хороши и поэтому широко применяются. Многие продукты предоставляют графический пользовательский интерфейс (GUI), но для самой работы используют входящий в комплект декомпилятор. Например, Decafe, DJ и Cavaj представляют собой GUI-инструменты, поставляемые вместе с JAD, и потому не вошли в обзор. В остальной части книги для получения исходного кода мы будем использовать консольную версию JAD. В большинстве случаев консольный декомпилятор - это всё, что вам нужно. Если же вы предпочитаете GUI, обязательно убедитесь, что в его основе лежит надёжный декомпилятор, например JAD или JODE.

Декомпиляция класса

Если вы прежде не пользовались декомпиляторами, давайте посмотрим, насколько хорошо они справляются со своей задачей. Мы будем работать с немного расширенной версией класса `MessageInfo`, который Chat использует для отправки текста сообщения и его атрибутов на удалённый узел. Файл `MessageInfoComplex.java`, приведённый в листинге 2.1, содержит анонимный внутренний класс (`MessageInfoPK`) и метод `main()`, демонстрирующие несколько более сложных случаев декомпиляции.

```
package covertjava.decompile;

/**
 * MessageInfo используется для отправки дополнительных сведений с каждым
 * сообщением по сети. В настоящее время он содержит имя узла, с которого
 * поступило сообщение, и имя отправившего его пользователя.
 */
public class MessageInfoComplex implements java.io.Serializable {
    String hostName;
    String userName;

    public MessageInfoComplex(String hostName, String userName) {
        this.hostName = hostName;
        this.userName = userName;
    }

    /**
     * @return имя узла, отправившего сообщение
     */
    public String getHostName() {
        return hostName;
    }
    /**
     * @return имя пользователя, отправившего сообщение
     */
    public String getUserName() {
        return userName;
    }
```

Листинг 2.1. Исходный код `MessageInfoComplex` *(начало)*

```
    /**
     * Вспомогательный метод для получения строки, наилучшим образом
     * идентифицирующей пользователя.
     * @return имя, которым следует обозначать пользователя,
     *         отправившего это сообщение
     */
    public String getDisplayName() {
        return getUserName() + " (" + getHostName() + ")";
    }

    /**
     * Создать идентификатор сообщения, который можно использовать
     * для распознавания этого сообщения в базе данных.
     * Формат: <ID><UserName><HostName>. Имена ограничены 8 символами.
     * Пример: для Kalinovsky/JAMAICA будет создано 443651_Kalinovs_JAMAICA.
     */
    public String generateMessageId() {
        StringBuffer id = new StringBuffer(22);
        String systemTime = "" + System.currentTimeMillis();
        id.append(systemTime.substring(0, 6));

        if (getUserName() != null && getUserName().length() > 0) {
            // Добавить имя пользователя, если оно указано
            id.append('_');
            int maxChars = Math.min(getUserName().length(), 8);
            id.append(getUserName().substring(0, maxChars));
        }
```

```
        if (getHostName() != null && getHostName().length() > 0) {
            // Добавить имя узла, если оно указано
            id.append('_');
            int maxChars = Math.min(getHostName().length(), 7);
            id.append(getHostName().substring(0, maxChars));
        }

        return id.toString();
    }
    /**
     * Включить пример анонимного внутреннего класса.
     */
    public static void main(String[] args) {
        new Thread(new Runnable() {
```

Листинг 2.1. Исходный код `MessageInfoComplex` *(продолжение)*

```
            public void run() {
                System.out.println("Running test");
                MessageInfoComplex info =
                    new MessageInfoComplex("JAMAICA", "Kalinovsky");
                System.out.println("Message id = " + info.generateMessageId());
                info = new MessageInfoComplex(null, "JAMAICA");
                System.out.println("Message id = " + info.generateMessageId());
            }
        }).start();
    }

    /**
     * Внутренний класс, который можно использовать как первичный ключ
     * для MessageInfoComplex.
     */
    public static class MessageInfoPK implements java.io.Serializable {
        public String id;
    }
}
```

Листинг 2.1. Исходный код `MessageInfoComplex` *(окончание)*

После компиляции `MessageInfoComplex.java` с помощью `javac` и параметров по умолчанию мы получаем три файла классов: `MessageInfoComplex.class`, `MessageInfoComplex$MessageInfoPK.class` и `MessageInfoComplex$1.class`. Как вам, возможно, известно, внутренние и анонимные классы появились в Java в JDK 1.1, однако одной из целей проектирования было сохранение совместимости формата байт-кода с более ранними версиями Java. Поэтому эти конструкции языка порождают в определённой мере самостоятельные классы, хотя связь с родительским классом сохраняется. На последнем этапе проверки мы запустим декомпилятор для файла класса, а затем сравним созданный исходный код с оригиналом. Если вы загрузили и установили JAD и добавили его в путь поиска, программу можно запустить следующей командой:

```
jad MessageInfoComplex.class
```

Завершив работу, JAD создаёт файл `MessageInfoComplex.jad`. Мы переименовали его в `MessageInfoComplex_FullDebug.jad`; содержимое файла показано в листинге 2.2.

```
// Декомпилировано Jad v1.5.7g. Copyright 2000 Pavel Kouznetsov.
// Домашняя страница Jad: http://www.geocities.com/SiliconValley/Bridge/8617/jad.html
// Параметры декомпилятора: packimports(3)
// Имя исходного файла: MessageInfoComplex.java
```

Листинг 2.2. Декомпилированный код `MessageInfoComplex` *(начало)*

```
package covertjava.decompile;

import java.io.PrintStream;
import java.io.Serializable;

public class MessageInfoComplex
    implements Serializable
{
    public static class MessageInfoPK
        implements Serializable
    {
        public String id;

        public MessageInfoPK()
        {
        }
    }

    public MessageInfoComplex(String hostName, String userName)
    {
        this.hostName = hostName;
        this.userName = userName;
    }

    public String getHostName()
    {
        return hostName;
    }

    public String getUserName()
    {
        return userName;
    }
    public String getDisplayName()
    {
        return getUserName() + " (" + getHostName() + ")";
    }

    public String generateMessageId()
```

Листинг 2.2. Декомпилированный код `MessageInfoComplex` *(продолжение)*

```
    {
        StringBuffer id = new StringBuffer(22);
        String systemTime = "" + System.currentTimeMillis();
        id.append(systemTime.substring(0, 6));
        if(getUserName() != null && getUserName().length() > 0)
        {
            id.append('_');
            int maxChars = Math.min(getUserName().length(), 8);
            id.append(getUserName().substring(0, maxChars));
        }
        if(getHostName() != null && getHostName().length() > 0)
        {
            id.append('_');
            int maxChars = Math.min(getHostName().length(), 7);
            id.append(getHostName().substring(0, maxChars));
        }
        return id.toString();
    }
```

```
    public static void main(String args[])
    {
        (new Thread(new Runnable() {
            public void run()
            {
                System.out.println("Running test");
                MessageInfoComplex info =
                    new MessageInfoComplex("JAMAICA", "Kalinovsky");
                System.out.println("Message id = " + info.generateMessageId());
                info = new MessageInfoComplex(null, "JAMAICA");
                System.out.println("Message id = " + info.generateMessageId());
            }
        })).start();
    }

    String hostName;
    String userName;
}
```

Листинг 2.2. Декомпилированный код `MessageInfoComplex` *(окончание)*

Уделите несколько минут просмотру созданного кода. Как видите, он почти на 100% совпадает с оригиналом! Порядок объявлений переменных, методов и внутреннего класса, а также форматирование различаются, но логика совершенно одинакова. Комментарии тоже потеряны, однако хорошо написанный код Java, подобный нашему, и без того понятен, не правда ли?

В нашем случае результат оказался хорошим, поскольку `javac` включает полные отладочные сведения при использовании параметра `-g`. Если исходный код скомпилировать без отладочных сведений (с параметром `-g:none`), декомпилированный код станет менее понятным: например, будут утрачены имена параметров методов и локальных переменных. Ниже показаны конструктор и использующий локальные переменные метод класса `MessageInfoComplex`, скомпилированного без отладочных сведений:

```
public MessageInfoComplex(String s, String s1)
{
    hostName = s;
    userName = s1;
}

public String generateMessageId()
{
    StringBuffer stringbuffer = new StringBuffer(22);
    String s = "" + System.currentTimeMillis();
    stringbuffer.append(s.substring(0, 6));
    if(getUserName() != null && getUserName().length() > 0)
    {
        stringbuffer.append('_');
        int i = Math.min(getUserName().length(), 8);
        stringbuffer.append(getUserName().substring(0, i));
    }
    if(getHostName() != null && getHostName().length() > 0)
    {
        stringbuffer.append('_');
        int j = Math.min(getHostName().length(), 7);
        stringbuffer.append(getHostName().substring(0, j));
    }
    return stringbuffer.toString();
}
```

Благодаря чему возможна декомпиляция?

Исходный код Java, в отличие от исходного кода C/C++, компилируется не в двоичный машинный код. Результатом компиляции исходного текста Java становится промежуточный байт-код - не зависящее от платформы представление исходного кода. После загрузки байт-код можно интерпретировать или компилировать, поэтому преобразование высокоуровневого языка программирования в низкоуровневый машинный код выполняется в два этапа. Именно промежуточный этап делает декомпиляцию байт-кода Java почти безошибочной. Байт-код несёт всю существенную информацию исходного файла. Хотя комментарии и форматирование утрачиваются, все методы, переменные и программная логика, разумеется, сохраняются. Поскольку байт-код не является машинным языком самого низкого уровня, формат кода остаётся близок к исходному. Спецификация JVM определяет набор инструкций, соответствующих операторам и ключевым словам языка Java, поэтому такой фрагмент кода Java:

```
public String getDisplayName() {
    return getUserName() + " (" + getHostName() + ")";
}
```

представляется следующим байт-кодом:

```
0 new #4 <java/lang/StringBuffer>
3 dup
4 aload_0
5 invokevirtual #5 <covertjava/decompile/MessageInfoComplex.getUserName>
8 invokestatic #6 <java/lang/String.valueOf>
11 invokestatic #6 <java/lang/String.valueOf>
14 invokespecial #7 <java/lang/StringBuffer.<init>>
17 ldc #8 < (>
19 invokevirtual #9 <java/lang/StringBuffer.append>
22 aload_0
23 invokevirtual #10 <covertjava/decompile/MessageInfoComplex.getHostName>
26 invokevirtual #9 <java/lang/StringBuffer.append>
29 ldc #11 <)>
31 invokevirtual #9 <java/lang/StringBuffer.append>
34 invokestatic #6 <java/lang/String.valueOf>
37 invokestatic #6 <java/lang/String.valueOf>
40 areturn
```

Формат байт-кода подробно рассматривается в главе 17 «Понимание и изменение байт-кода», однако сходство заметно уже при беглом взгляде на байт-код. Декомпилятор загружает его и пытается восстановить исходный текст на основе инструкций байт-кода. Имена методов и переменных класса обычно сохраняются, тогда как имена параметров методов и локальных переменных теряются. Если имеются отладочные сведения, они предоставляют декомпилятору имена параметров и номера строк, делая восстановленный исходный файл ещё понятнее.

Возможные проблемы с декомпилированным кодом

В большинстве случаев декомпиляция создаёт читаемый файл, который можно изменить и снова скомпилировать. Однако иногда результат декомпиляции не удаётся скомпилировать повторно. Такое возможно, если байт-код был обфусцирован, а заданные обфускатором имена создают неоднозначность при компиляции. Байт-код проверяется во время загрузки, но средства проверки предполагают, что целый ряд ошибок уже был обнаружен компилятором. Поэтому проверка байт-кода не так строга, как компиляция, и обфускаторы могут воспользоваться этим для более надёжной защиты интеллектуальной собственности. Например, ниже приведён результат JAD для анонимного внутреннего класса из метода `main()` класса `MessageInfoComplex`, обфусцированного с помощью Zelix ClassMaster:

```
static class c
    implements Runnable
{
    public void run()
    {
        boolean flag = a.b;
        System.out.println(a("*4%p\002\026&kj\016\0135"));
        b b1 = new b(a("2\000\006_\";\0"), a("3 'w\005\02778u\022"));
        System.out.println(a("5$8m\n\037$kw\017X|k").concat(String.valueOf
            (String.valueOf(b1.d()))));
        b1 = new b(null, a("2\000\006_\";\0"));
        System.out.println(a("5$8m\n\037$kw\017X|k").concat(String.valueOf
            (String.valueOf(b1.d()))));
        if(flag)
            b.c = !b.c;
    }

    private static String a(String s)
    {
        char ac[];
        int i;
        int j;
        ac = s.toCharArray();
        i = ac.length;
        j = 0;
        if(i > 1) goto _L2; else goto _L1
_L1:
        ac;
        j;

_L10:
        JVM INSTR dup2 ;
        JVM INSTR caload ;
        j % 5;
        JVM INSTR tableswitch 0 3: default 72
        //                       0 52
        //                       1 57
        //                       2 62
        //                       3 67;
        goto _L3 _L4 _L5 _L6 _L7
_L4:
        0x78;
        goto _L8
_L5:
        65;
        goto _L8
_L6:
        75;
        goto _L8
_L7:
        30;
        goto _L8
_L3:
        107;
_L8:
```

```
        JVM INSTR ixor ;
        (char);
        JVM INSTR castore ;
        j++;
        if(i != 0) goto _L2; else goto _L9
_L9:
        ac;
        i;
        goto _L10
_L2:
        if(j >= i)
            return new String(ac);
        if(true) goto _L1; else goto _L11
_L11:
    }
}
```

Как видите, это полный провал, нисколько не похожий на исходный код Java. Ещё тревожнее то, что JAD создал исходный код, который невозможно скомпилировать. Два других декомпилятора сообщили об ошибке в файле класса. Само собой разумеется, JVM без проблем распознаёт и загружает рассматриваемый байт-код. Обфускация подробно рассматривается в главе 3 «Обфускация классов».

Мощный способ защиты интеллектуальной собственности состоит в кодировании файлов классов и применении собственного загрузчика классов, который декодирует их при загрузке. В этом случае декомпиляторы нельзя использовать ни для одного класса приложения, кроме точки входа и самого загрузчика. Кодирование не является непреодолимой защитой, но значительно усложняет взлом. Сначала злоумышленнику придётся декомпилировать загрузчик классов, чтобы понять механизм декодирования, затем декодировать все файлы классов, и только после этого он сможет приступить к декомпиляции. Сведения о наилучших способах защиты интеллектуальной собственности в приложениях Java приведены в главе 19 «Защита коммерческих приложений от взлома».

Короткий тест

1. По каким причинам выполняют декомпиляцию байт-кода?

2. Какие параметры компилятора влияют на качество декомпиляции и каким образом?

3. Почему декомпилированный байт-код Java почти идентичен исходному коду?

4. Как защитить байт-код от декомпиляции?

Краткие итоги

• Декомпиляция создаёт из байт-кода исходный код, почти идентичный оригиналу.

• Декомпиляция - мощный способ изучения логики реализации при отсутствии документации и исходного кода. Однако лицензионное соглашение может прямо запрещать декомпиляцию и обратную разработку.

• Для декомпиляции необходимо загрузить и установить декомпилятор.

• Декомпиляция классов Java эффективна благодаря тому, что байт-код представляет собой промежуточный этап между исходным и машинным кодом.

• Хороший обфускатор способен сделать декомпилированный код крайне трудным для чтения и понимания.

Глава 3. Обфускация классов

> «Любая достаточно развитая технология неотличима от магии».
>
> *Законы Мерфи о технике*

В этой главе

• Защита идей, лежащих в основе вашего кода
• Обфускация как средство защиты интеллектуальной собственности
• Преобразования, выполняемые обфускаторами
• Лучшие обфускаторы
• Возможные проблемы и распространённые решения
• Обфускация приложения Chat с помощью Zelix KlassMaster
• Взлом обфусцированного кода
• Короткий тест
• Краткие итоги

Защита идей, лежащих в основе вашего кода

Обратная разработка и взлом существуют с самых первых дней разработки программного обеспечения. Более того, кража или воспроизведение чужих идей всегда были самым простым способом создания конкурентоспособных продуктов. Разумеется, есть и вполне приемлемый способ опираться на прежние открытия других людей, и пока эти люди не возражают, он прекрасно работает. Однако большинство изобретателей и исследователей хотели бы получить признание и, возможно, финансовое вознаграждение за свой труд. Проще говоря, им тоже нужно выплачивать ипотеку и ездить в отпуск.

Хороший способ защитить интеллектуальную собственность - получить авторские права и патенты на уникальные особенности работы. Это, безусловно, рекомендуется для изобретений и крупных открытий, потребовавших значительных вложений в исследования и разработку. Оформление авторских прав на программу - довольно простой и экономичный процесс, но он защищает только «оригинальный» код приложения, а не заложенные в нём идеи. Другие люди не смогут без разрешения автора взять защищённый авторским правом код и использовать его в своих приложениях, но собственная реализация той же функции нарушением считаться не будет. Патенты обеспечивают гораздо более надёжную защиту, поскольку охватывают идеи и алгоритмы, а не конкретную реализацию, однако подача патентной заявки обходится дорого, а получение патента может занять годы.

Реален ли риск взлома вашего приложения? Если в нём есть хорошие идеи, то безусловно. На момент написания книги большинство широко освещавшихся случаев обратной разработки не относилось к продуктам Java, однако вот выдержка из сообщения одного поставщика Java-продуктов, DataDirect Technologies:

> **РОКВИЛЛ, штат Мэриленд, 1 июля 2002 года.** DataDirect Technologies, Inc., ведущий поставщик средств подключения к данным, подала иск против i-net Software GmbH, заявив о нарушении авторских прав и условий договора. DataDirect Technologies добивается как предварительного, так и постоянного судебного запрета, чтобы воспрепятствовать

> дальнейшим попыткам i-net продвигать и продавать продукты, которые, по мнению DataDirect Technologies, были незаконно получены путём обратной разработки её продуктов.

DataDirect Technologies утверждает, что конкурент выполнил обратную разработку её продукта, и тем не менее даже сегодня этот продукт лишь минимально защищён от декомпиляции.

В реальном мире авторское право на код и патент на подход не обеспечивают достаточной защиты, если конкурент или злоумышленник может без труда изучить реализацию по исходному коду. Вопросы юридической защиты рассматриваются в отдельной главе, а пока сосредоточимся на разумных способах защиты интеллектуальной собственности (ИС) приложений Java.

Обфускация как средство защиты интеллектуальной собственности

Обфускация - процесс преобразования байт-кода в менее понятную человеку форму с целью затруднить обратную разработку. Обычно он включает удаление всех отладочных сведений, например таблиц переменных и номеров строк, а также переименование пакетов, классов и методов с присвоением им имён, созданных машиной. Более развитые обфускаторы идут дальше: изменяют поток управления кода Java, перестраивая существующую логику и вставляя фиктивный код, который никогда не будет выполнен. Основное условие обфускации состоит в том, что преобразования не должны нарушать корректность байт-кода или изменять предоставляемые извне функции.

Обфускация возможна по тем же причинам, что и декомпиляция: байт-код Java стандартизован и хорошо документирован. Обфускаторы загружают файлы классов Java, разбирают их формат, а затем применяют преобразования в соответствии с поддерживаемыми возможностями. После выполнения всех преобразований байт-код сохраняется в новом файле класса. Новый файл имеет другую внутреннюю структуру, но ведёт себя точно так же, как исходный.

Обфускаторы особенно необходимы для продуктов и технологий, в которых логика реализации передаётся пользователю. Так происходит со страницами HTML и JavaScript, где продукт распространяется в форме исходного кода. Положение Java ненамного лучше: хотя программы обычно распространяются в виде двоичного байт-кода, декомпилятор, подобный описанному в предыдущей главе, способен получить исходный текст, почти не уступающий оригиналу.

Преобразования, выполняемые обфускаторами

Стандартов обфускации не существует, поэтому уровень защиты зависит от качества обфускатора. В следующих разделах представлены некоторые функции, часто встречающиеся в обфускаторах. На примере метода `sendMessage` класса `ChatServer` мы покажем, как каждое преобразование влияет на декомпилированный код. Исходный код `sendMessage` приведён в листинге 3.1.

```
public void sendMessage(String host, String message) throws Exception {
    if (host == null || host.trim().length() == 0)
        throw new Exception("Please specify host name");

    System.out.println("Sending message to host " + host + ": " + message);
    String url = "//" + host + ":" + this.registryPort + "/chatserver";
    ChatServerRemote remoteServer = (ChatServerRemote)Naming.lookup(url);
    MessageInfo messageInfo = new MessageInfo(this.hostName, this.userName);
    remoteServer.receiveMessage(message, messageInfo);
    System.out.println("Message sent to host " + host);
}
```

Листинг 3.1. Исходный код `sendMessage`

Удаление отладочных сведений

Байт-код Java может содержать вставленные компилятором сведения, которые помогают отлаживать выполняющийся код. Сведения, добавленные `javac`, могут включать некоторые или все из следующих элементов: номера строк, имена переменных и имена исходных файлов. Для выполнения класса отладочные сведения не нужны, но отладчики используют их, чтобы сопоставлять байт-код с исходным текстом. Декомпиляторы применяют эти сведения для более точного восстановления исходного кода. При наличии в файле класса полных отладочных сведений декомпилированный код почти идентичен оригиналу. Когда отладочные сведения удаляются, сохранённые в них имена теряются и декомпиляторам приходится создавать собственные. В нашем случае после удаления сведений параметры `sendMessage` назывались бы `s1` и `s2`, а не `host` и `message`.

Искажение имён

Разработчики дают пакетам, классам и методам осмысленные имена. Реализация сервера нашего демонстрационного чат-приложения называется `ChatServer`, а метод, отправляющий сообщение другому пользователю, - `sendMessage`. Хорошие имена чрезвычайно важны для разработки и сопровождения, но ничего не значат для JVM. Среде выполнения Java (JRE) безразлично, называется ли `sendMessage` именем `goShopping` или `abcdefg`: она всё равно вызовет и выполнит этот метод. Переименовывая понятные человеку осмысленные имена в бессмысленные, созданные машиной, обфускаторы значительно затрудняют понимание декомпилированного кода.

То, что прежде называлось `ChatServer.sendMessage`, превращается в `d.a`; когда множество классов и методов получает одинаковые имена, следить за декомпилированным кодом становится чрезвычайно трудно. Хороший обфускатор пользуется полиморфизмом, чтобы ещё сильнее запутать дело. Три метода, которые в исходном коде имели разные имена и сигнатуры и выполняли разные задачи, в обфусцированном коде могут получить одно общее имя. Поскольку их сигнатуры различаются, это не нарушает спецификацию языка Java, но усиливает путаницу в декомпилированном коде. В листинге 3.2 показан пример декомпилированного `sendMessage` после обфускации, которая удалила отладочные сведения и исказила имена.

```
public void a(String s, String s1)
    throws Exception
{
    if(s == null || s.trim().length() == 0)
    {
        throw new Exception("Please specify host name");
    } else
    {
        System.out.println(String.valueOf(String.valueOf((
            new StringBuffer("Sending message to host ")
            ).append(s).append(": ").append(s1))));
        String s2 = String.valueOf(String.valueOf((
            new StringBuffer("//")).append(s).append(":")
            .append(b).append("/chatserver")));
        b b1 = (b)Naming.lookup(s2);
        MessageInfo messageinfo = new MessageInfo(e, f);
        b1.receiveMessage(s1, messageinfo);
        System.out.println("Message sent to host ".concat(
            String.valueOf(String.valueOf(s))));
        return;
    }
}
```

Листинг 3.2. Декомпилированный `sendMessage` *после искажения имён*

Кодирование строк Java

Строки Java хранятся в байт-коде как обычный текст. Большинство хорошо написанных приложений содержит трассировку, создающую журналы выполнения для отладки и аудита. Даже если имена классов и методов изменены, строки, записываемые методами в файл журнала или консоль, могут выдать назначение метода.

В нашем случае `ChatServer.sendMessage` выводит трассировочное сообщение следующим выражением:

```
System.out.println("Sending message to host " + host + ": " + message);
```

Даже если `ChatServer.sendMessage` переименован в `d.a`, по такой строке трассировки в теле декомпилированного метода сразу понятно, что он делает. Однако если строка закодирована в байт-коде, декомпилированная версия класса выглядит так:

```
System.out.println(String.valueOf(String.valueOf((new
    StringBuffer(a("A\025wV6|\0279_:a\003xU:2\004v\0227}\003m\022"))
    ).append(s).append(a("(P")).append(s1))));
```

Если внимательно посмотреть на закодированную строку, видно, что сначала она передаётся методу `a()`, который декодирует её и возвращает читаемую строку методу `System.out.println()`. Кодирование строк - мощная возможность, которая должна присутствовать в обфускаторе коммерческого уровня.

Изменение потока управления

Представленные выше преобразования затрудняют обратную разработку обфусцированного кода, но не меняют фундаментальную структуру кода Java. Кроме того, они никак не защищают алгоритмы и поток управления программы, а именно в них обычно заключается важнейшая часть новшества. Показанная выше декомпилированная версия `ChatServer.sendMessage` всё ещё достаточно понятна. Видно, что сначала код проверяет входные данные и при ошибке создаёт исключение. Затем он находит удалённый объект сервера и вызывает его метод.

Лучшие обфускаторы способны преобразовывать поток выполнения байт-кода, вставляя фиктивные условные инструкции и переходы `goto`. Это может несколько замедлить выполнение, но повышение защиты ИС, возможно, стоит такой небольшой цены. В листинге 3.3 показано, во что превратился `sendMessage` после применения всех рассмотренных выше преобразований.

```
public void a(String s, String s1)
    throws Exception
{
    boolean flag = MessageInfo.c;
    s;
    if(flag) goto _L2; else goto _L1
_L1:
    JVM INSTR ifnull 29;
    goto _L3 _L4
_L3:
    s.trim();
```

Листинг 3.3. Декомпилированный `sendMessage` *после всех преобразований (начало)*

```
_L2:
    if(flag) goto _L6; else goto _L5
_L5:
    length();
    JVM INSTR ifne 42;
    goto _L4 _L7
_L4:
    throw new Exception(a("\002)qUe7egDs1,rM6:*g@6<$yQ"));
_L7:
    System.out.println(String.valueOf(String.valueOf((
        new StringBuffer(a("\001 zP\177<\"4Ys!6uSsr1{\024~=6`\024"))
        ).append(s).append(a("he")).append(s1))));
    String.valueOf(String.valueOf(
        (new StringBuffer(a("}j"))).append(s).append(":")
        .append(b).append(a("}&|Ub! fBs "))));
_L6:
    String s2;
    s2;
    covertjava.chat.b b1 = (covertjava.chat.b)Naming.lookup(s2);
    MessageInfo messageinfo = new MessageInfo(e, f);
    b1.receiveMessage(s1, messageinfo);
    System.out.println(a("\037 gGw5 4Gs<14@yr-{Gbr").concat(String.valueOf
        (String.valueOf(s))));
    if(flag)
        b.c = !b.c;
    return;
}
```

Листинг 3.3. Декомпилированный `sendMessage` *после всех преобразований (окончание)*

Вот это уже полная, но весьма действенная неразбериха! `sendMessage` - довольно небольшой метод с простой условной логикой. Если бы обфускация потока управления применялась к более сложному методу с циклами `for`, инструкциями `if` и локальными переменными, результат был бы ещё эффективнее.

Вставка повреждённого кода

Вставка повреждённого кода - довольно сомнительный приём, который некоторые обфускаторы используют, чтобы воспрепятствовать декомпиляции обфусцированных классов. Он основан на вольном толковании спецификации байт-кода Java средой выполнения Java. JRE не обеспечивает строгое соблюдение всех правил проверки формата байт-кода, и это позволяет обфускаторам внедрять в файлы классов некорректный байт-код. Внедрённый код не мешает выполнению исходного кода, но попытка декомпилировать файл класса заканчивается ошибкой или, в лучшем случае, созданием путаного исходного текста, переполненного ключевыми словами `JVM INSTR`. В листинге 3.3 показано, как декомпилятор может обработать повреждённый код. Риск этого метода состоит в том, что повреждённый код может не работать в версии JVM, которая строже следует спецификации.

Даже если сегодня это не создаёт проблем в большинстве JVM, позднее ситуация может измениться.

Удаление неиспользуемого кода (сокращение)

В качестве дополнительного преимущества большинство обфускаторов удаляет неиспользуемый код, уменьшая размер приложения. Например, если в классе `A` есть метод `m()`, который не вызывается ни одним классом, код `m()` удаляется из байт-кода `A`. Эта возможность особенно полезна для кода, загружаемого через Интернет или устанавливаемого в незащищённых средах.

Оптимизация байт-кода

Ещё одно дополнительное преимущество, о котором заявляют создатели обфускаторов, - возможная оптимизация кода. Поставщики утверждают, что объявление методов, не имеющих модификатора `final`, как `final` там, где это возможно, и небольшие усовершенствования кода способны ускорить выполнение. Оценить реальный прирост производительности трудно, а большинство поставщиков не публикует показатели. Важно отметить, что с каждым новым выпуском JIT-компиляторы становятся мощнее. Поэтому такие операции, как финализация методов и удаление мёртвого кода, скорее всего, в любом случае выполняются JIT-компилятором.

Лучшие обфускаторы

Существует множество обфускаторов, и большинство из них предлагает один и тот же набор основных функций. В таблице 3.1 приведены лишь несколько самых популярных бесплатных и коммерческих продуктов.

Таблица 3.1. Популярные обфускаторы

Характеристика	KlassMaster	ProGuard	Retro Guard	Dash-O	Jshrink
Версия	4.1	1.7	1.1.13	2.x	2.0
Цена	199-399 долларов	Бесплатно	Бесплатно	895-2995 долларов	95 долларов
Удаление отладочных сведений	Да	Да	Да	Да	Да
Искажение имён	Да	Да	Да	Да	Да
Кодирование строк Java	Да	Нет	Нет	Нет	Да
Изменение потока управления	Да	Нет	Нет	Нет	Нет
Вставка повреждённого кода	Да	Нет	Нет	Нет	Нет
Удаление неиспользуемого кода (сокращение)	Да	Да	Нет	Да	Да
Оптимизация байт-кода	Нет	Нет	Нет	Да	Да
Гибкость языка сценариев и управления обфускацией	Отлично	Отлично	Хорошо	Не оценивалось	Хорошо
Восстановление трассировок стека	Да	Да	Нет	Нет	Нет

Для коммерческих приложений, содержащих интеллектуальную собственность, я рекомендую Zelix KlassMaster прежде всего благодаря уникальной обфускации потока управления. Этот приём делает обфусцированный код действительно трудным для взлома, поэтому продукт стоит каждого заплаченного за него доллара. На момент написания это единственный известный обфускатор с такой возможностью. ProGuard бесплатно доступен на <www.sourceforge.net> и является лучшим выбором для экономного пользователя, чьи приложения не требуют защиты коммерческого уровня.

Возможные проблемы и распространённые решения

Обфускация - достаточно безопасный процесс, который должен сохранять функции приложения. Однако в некоторых случаях выполняемые обфускаторами преобразования могут непреднамеренно нарушить работу прежде исправного кода. В следующих разделах рассматриваются распространённые проблемы и рекомендуемые решения.

Динамическая загрузка классов

Переименование пакетов, классов, методов и переменных работает нормально, пока имя последовательно изменяется во всей системе. Обфускаторы следят за тем, чтобы все статические ссылки в байт-коде были обновлены в соответствии с новым именем. Однако если код динамически загружает класс с помощью `Class.forName()` или `ClassLoader.loadClass()`, передавая исходное имя класса, может возникнуть исключение `ClassNotFound`. Современные обфускаторы достаточно хорошо обрабатывают такие случаи и пытаются заменить строки в соответствии с новыми именами. Но если строка создаётся во время выполнения или читается из файла свойств, обфускатор не способен её обработать. Хорошие обфускаторы создают файл журнала с предупреждениями, указывающими на код, который может вызвать проблемы во время выполнения.

> **Истории с передовой.** Самый новаторский продукт CreamTec - WebCream, бесплатно загружаемый из Интернета. Бесплатная редакция ограничена пятью одновременно работающими пользователями; чтобы увеличить их число, необходимо приобрести коммерческую лицензию. Я вырос на Украине и знал многих людей, которые предпочли бы взломать лицензирование и превратить бесплатную редакцию в неограниченную, обычно стоящую тысячи долларов. В CreamTec мы использовали простой бесплатный обфускатор, почти ничего не делавший, кроме искажения имён. Мы считали его достаточным, пока один мой друг, воспринимающий коммерческие программы с ограниченной функциональностью как личное оскорбление, не взломал наш код лицензирования менее чем за 15 минут. Намёк оказался более чем понятным, и мы решили приобрести Zelix KlassMaster, чтобы защитить продукт настолько хорошо, насколько возможно. После применения агрессивной обфускации потока управления и нескольких дополнительных уловок наш друг уже не смог столь же легко добраться до кода лицензирования, а поскольку тратить несколько дней на его изучение он не захотел, то отказался от попыток.

Самое простое решение - настроить обфускатор так, чтобы он сохранял имена динамически загружаемых классов. Содержимое класса, включая методы, переменные и код, по-прежнему можно преобразовывать.

Рефлексия

Для рефлексии необходимо знать имена методов и полей во время компиляции, поэтому обфускация влияет и на неё. Обязательно используйте хороший обфускатор и просматривайте предупреждения в файле журнала. Как и при динамической загрузке классов, если обфускация вызывает ошибки во время выполнения, из неё необходимо исключить имена методов или полей, на которые ссылаются `Class.getMethod` или `Class.getField`.

Сериализация

Сериализованные объекты Java содержат данные экземпляра и сведения о классе. Если версия или структура класса меняется, при десериализации может возникнуть исключение. Обфусцированные классы можно сериализовать и десериализовать, но попытка обфусцированного класса десериализовать экземпляр необфусцированного класса завершится неудачей. Эта проблема встречается нечасто; обычно её можно решить, исключив сериализуемые классы из обфускации или не смешивая сериализованные классы.

Нарушение соглашений об именовании

Переименование методов может нарушить шаблоны проектирования, например Enterprise JavaBeans (EJB), где разработчик компонента обязан предоставить методы с определёнными именами и сигнатурами. Методы обратного вызова EJB, такие как `ejbCreate` и `ejbRemove`, не определены суперклассом или интерфейсом. Предоставление этих методов с конкретной сигнатурой - всего лишь соглашение, предписанное спецификацией EJB и контролируемое контейнером. Изменение имён методов обратного вызова нарушает соглашение об именовании и делает компонент непригодным к использованию. Имена таких методов всегда следует исключать из обфускации.

Трудности сопровождения

И последнее, но не менее важное: обфускация затрудняет сопровождение приложений и устранение неполадок. Обработка исключений Java позволяет эффективно изолировать ошибочный код, а трассировка стека обычно даёт хорошее представление о том, что и где пошло не так. Сохранение отладочных сведений об именах исходных файлов и номерах строк позволяет среде выполнения сообщить точное место возникновения ошибки в коде. Неосторожная обфускация может лишить разработчика этой возможности и затруднить отладку: вместо настоящих имён классов и номеров строк он увидит лишь обфусцированные имена.

В обфусцированном коде следует сохранять как минимум сведения о номерах строк. Хорошие обфускаторы создают журнал преобразований, в том числе сопоставление исходных имён классов и методов с их обфусцированными аналогами. Ниже приведён фрагмент файла журнала, созданного Zelix KlassMaster для класса `ChatServer`:

```
Class: public covertjava.chat.ChatServer    =>    covertjava.chat.d
Source: "ChatServer.java"

FieldsOf: covertjava.chat.ChatServer
hostName    =>    e
protected static instance    =>    a
messageListener    =>    d
protected registry    =>    c
protected registryPort    =>    b
userName    =>    f

MethodsOf: covertjava.chat.ChatServer
public static getInstance()    =>    a
public getRegistry(int)    =>    a
public init()    =>    b
public receiveMessage(java.lang.String, covertjava.chat.MessageInfo)
    NameNotChanged
public sendMessage(java.lang.String, java.lang.String)    =>    a
public setMessageListener(covertjava.chat.MessageListener)    =>    a
```

Итак, если в трассировке стека исключения присутствует метод `covertjava.chat.d.b`, по журналу можно установить, что первоначально он назывался `init` и находился в классе, который первоначально назывался `covertjava.chat.ChatServer`. Если исключение возникло в `covertjava.chat.d.a`, однозначно определить исходное имя метода не удастся, поскольку ему соответствует несколько вариантов (вот она, сила перегрузки). Именно поэтому номера строк так важны. С помощью файла журнала и номера строки в исходном файле можно быстро найти проблемную область кода приложения.

Некоторые обфускаторы предоставляют служебную программу для восстановления трассировок стека. Это удобный способ получить настоящую трассировку из обфусцированной. Обычно такая программа применяет тот же метод, которым мы только что воспользовались, но автоматизирует работу, так почему бы не сэкономить время? Она также позволяет перемешивать номера строк для дополнительной защиты.

Обфускация приложения Chat с помощью Zelix KlassMaster

Хотя у каждого обфускатора свой формат настройки преобразований, все они поддерживают общий набор функций. В приложении Chat нет передовых алгоритмов или ожидающих патента изобретений, но оно дорого нашему сердцу, поэтому мы воспользуемся Zelix KlassMaster, чтобы защитить его от любопытных взглядов злоумышленников и воров.

Сначала получим копию Zelix KlassMaster и установим её на локальную машину. Напомним, что домашний каталог приложения Chat мы называем `CovertJava`. Затем скопируем `ZKM.jar` из каталога установки KlassMaster в каталог `lib` нашего проекта, чтобы иметь возможность обращаться к нему из сценария. Проще всего создать сценарий обфускации с помощью GUI KlassMaster. Из каталога `lib` запустим GUI командой:

```
java -jar ZKM.jar
```

В появившемся начальном вспомогательном диалоговом окне выберем Set Classpath. Затем укажем библиотеки времени выполнения используемой JDK, а в появившемся после этого диалоговом окне Open Classes выберем `CovertJava/lib/chat.jar`. KlassMaster должна загрузить все классы приложения Chat, после чего станет доступен просмотр внутренней структуры байт-кода. Экран должен выглядеть примерно так, как показано на рисунке 3.1.

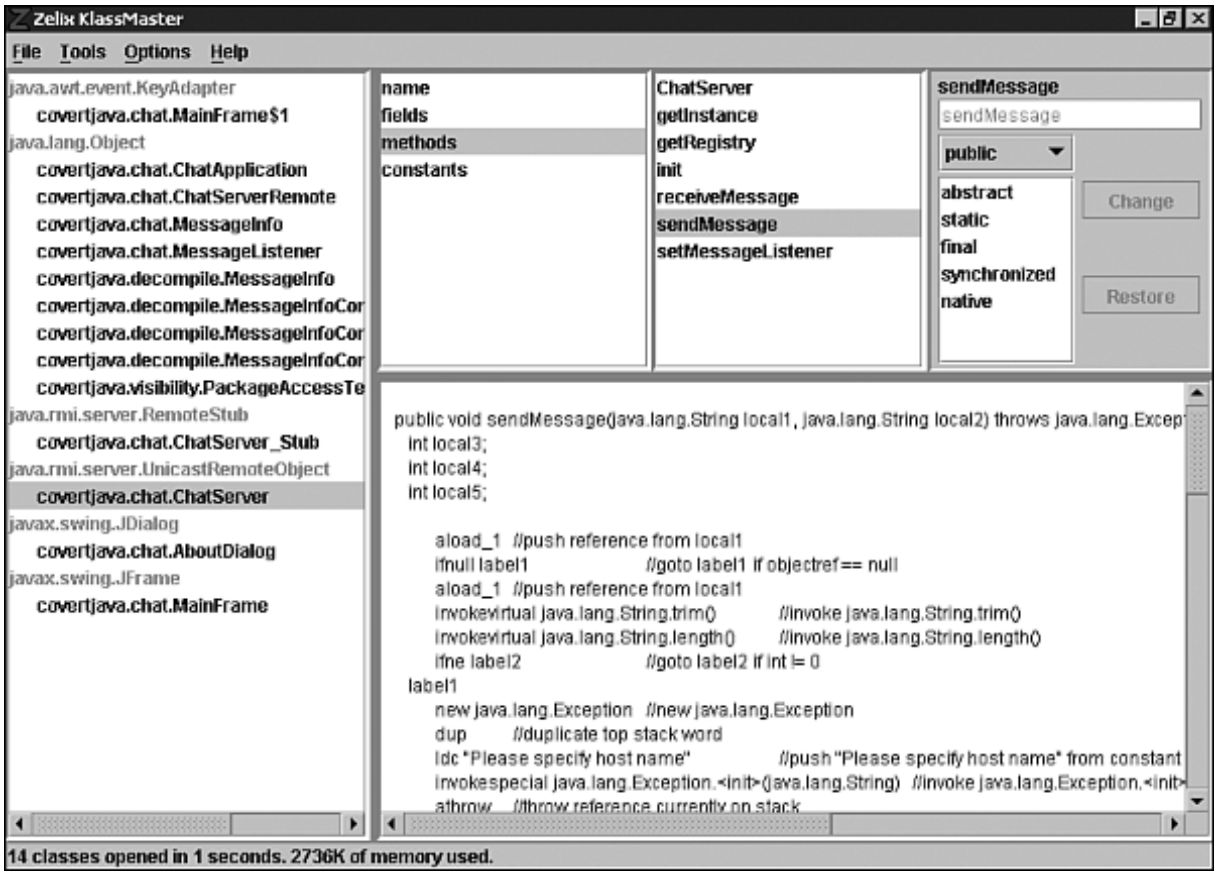

Рисунок 3.1. Классы Chat, загруженные в GUI KlassMaster

При работе с GUI сразу становится видно, насколько гибка KlassMaster. Можно вручную менять имена классов, методов и полей; изменять видимость классов и методов; объявлять методы `final`; менять текстовые строки и выполнять другие полезные действия. KlassMaster пытается распространить изменения по всему загруженному коду: если на метод с изменённым именем ссылаются другие классы, ссылки в них также обновляются. После внесения всех изменений классы можно сохранить в текущем виде либо сначала сократить и обфусцировать. Загруженные в среду GUI классы можно изменять и после обфускации, хотя я не могу придумать, зачем это кому-либо понадобится. Подробное описание функций KlassMaster и способов их применения приведено в руководстве пользователя.

Хорошо написанное приложение Java предоставляет сценарии сборки, поэтому встроим обфускацию в наш сценарий. Сначала с помощью GUI KlassMaster создадим сценарий обфускации, затем вручную изменим его, чтобы сделать гибче. Сценарий вполне можно написать вручную либо скопировать и изменить пример. Запустим GUI, выберем ZKM Script Helper в меню Tools, а затем выполним следующие действия:

1. Прочитайте инструкции на странице Introductory Page и нажмите Next.

2. На странице Classpath Statement выберите `rt.jar` и нажмите Next.

3. На странице Open Statement перейдите к `CovertJava/distrib/chat.jar` и нажмите >, чтобы выбрать файл для открытия. Нам нужен только один файл, поскольку в нём упакованы все классы приложения. Нажмите Next.

4. На странице TrimExclude Statement по умолчанию уже заданы исключения для случаев, в которых обфускация, вероятнее всего, приведёт к ошибке. Например, переименование методов класса реализации EJB делает его непригодным к использованию, поэтому EJB по умолчанию исключаются.

5. На странице Trim Statement установите флажки Delete Source File Attributes и Delete Deprecated Attributes, чтобы избавиться от отладочных сведений, затем нажмите Next.

6. В раскрывающемся списке Don't Change Main Class Name на странице Exclude Statement выберите `covertjava.chat.ChatApplication`, чтобы сохранить это имя. Благодаря этому записи манифеста JAR останутся действительными, а пользователи по-прежнему смогут запускать чат с помощью понятного человеку имени.

7. На странице Obfuscate Statement выберите Aggressive в списке Obfuscate Control Flow. Затем выберите Aggressive в списке Encrypt String Literals и Scramble в списке Line Number Tables. Это обеспечит коду достаточную защиту, но позволит впоследствии преобразовывать трассировки стека. Убедитесь, что установлен флажок Produce a Change Log File, и нажмите Next.

8. На странице SaveAll Statement перейдите к `CovertJava/distrib`, создайте подкаталог `obfuscated`, выберите его в качестве каталога вывода и нажмите Next.

9. На следующей странице должен появиться текст сценария и возможность сохранить его в каталог. Сохраните сценарий под именем `obfuscate_script.txt` в каталоге `CovertJava/build` и завершите работу GUI.

Полученный сценарий должен выглядеть примерно так, как показано в листинге 3.4.

```
/*****************************************************************************/
/* Generated by Zelix KlassMaster 4.1.1 ZKM Script Helper 2003.08.13 17:03:43 */
/*****************************************************************************/
```

Листинг 3.4. Сценарий обфускации, созданный GUI (начало)

```
classpath   "c:\java\jdk1.4\jre\lib\rt.jar"
            "c:\java\jdk1.4\jre\lib\sunrsasign.jar"
            "c:\java\jdk1.4\jre\lib\jsse.jar"
            "c:\java\jdk1.4\jre\lib\jce.jar"
            "c:\java\jdk1.4\jre\lib\charsets.jar";

open        "C:\Projects\CovertJava\distrib\chat.jar";

trim        deleteSourceFileAttributes=true
            deleteDeprecatedAttributes=true
            deleteUnknownAttributes=false;

exclude     covertjava.chat.^ChatApplication^ public static main(java.lang.String[]);

obfuscate   changeLogFileIn=""
```

```
            changeLogFileOut="ChangeLog.txt"
            obfuscateFlow=aggressive
            encryptStringLiterals=aggressive
            lineNumbers=scramble;

saveAll     archiveCompression=all "C:\Projects\CovertJava\distrib\obfuscated";
```

Листинг 3.4. Сценарий обфускации, созданный GUI (окончание)

Стоит заменить абсолютные пути к файлам относительными, чтобы вместо `C:\Projects\CovertJava\distrib\chat.jar` сценарий открывал `distrib\chat.jar`. Наконец, встроим обфускацию в процесс сборки, объявив собственную задачу и добавив цель, которая её вызывает. KlassMaster написана на Java, поэтому её можно вызывать из любого сценария сборки. К счастью, она предоставляет класс-обёртку для интеграции с Ant, так что нам достаточно добавить в `build.xml` приложения Chat следующий код:

```
<!-- Определить задачу, которая запустит Zelix KlassMaster для обфускации классов -->
<taskdef name="obfuscate" classname="ZKMTask" classpath="${basedir}/lib/ZKM.jar"/>

...

<!-- Определить цель, создающую обфусцированную версию Chat -->
<target name="obfuscate" depends="release">
    <obfuscate scriptFileName="${basedir}/build/obfuscate_script.txt"
        logFileName="${basedir}/build/obfuscate_log.txt"
        trimLogFileName="${basedir}/build/obfuscate_trim_log.txt"
        defaultExcludeFileName="${basedir}/build/obfuscate_defaultExclude.txt"
        defaultTrimExcludeFileName="${basedir}/build/obfuscate_defaultTrimExclude.txt"
        defaultDirectoryName="${basedir}"
    />
</target>
```

Теперь можно запустить Ant для цели `obfuscate`. Если сборка завершится успешно, в `CovertJava/distrib/obfuscated` будет создан новый файл `chat.jar`. Он содержит обфусцированную версию Chat, которую по-прежнему можно запустить командой `java -jar chat.jar`. Уделите несколько минут изучению содержимого этого JAR и попробуйте декомпилировать некоторые классы.

Прежде чем закончить разговор об использовании KlassMaster, я хотел бы привести ещё несколько примеров синтаксиса файла сценария для исключения классов и членов классов из обфускации. Формат из таблицы 3.2 можно применять в инструкциях сценария обфускации, принимающих имена в качестве параметров. Язык сценариев ZKM поддерживает подстановочные знаки, например `*` (любая последовательность символов) и `?` (любой один символ), а также логические операции, например `||` (или) и `!` (не). Подробное объяснение и полный синтаксис приведены в документации KlassMaster.

Таблица 3.2. Часто используемые шаблоны имён KlassMaster

Синтаксис	Чему соответствует
`package1.package2.`	Именам пакетов `package1` и `package2`. Другие имена пакетов и дочерние элементы `package2` не соответствуют шаблону.
`*.`	Всем именам пакетов приложения.
`Class1`	Имени класса `Class1`.
`package1.Class1`	Имени `Class1` в пакете `package1`, но не имени `package1`.
`package1.^Class1`	Именам `Class1` и `package1`.
`package1.^Class1^ method1()`	Именам `package1`, `Class1` и `method1` без параметров.
`package1.^Class1^ method1(*)`	Именам `package1`, `Class1` и всех перегруженных версий `method1`.

Взлом обфусцированного кода

После столь долгого разговора о защите интеллектуальной собственности с помощью обфускации следует сказать несколько слов о прочности этой защиты. Затрудняет ли хороший обфускатор взлом приложения? Безусловно. Гарантирует ли он, что приложение не будет взломано? Вовсе нет!

Если обфускация потока управления не применяется, читать обфусцированный код и работать с ним не так уж трудно. Главное - найти хорошую исходную точку для декомпиляции. В главе 2 «Декомпиляция классов» представлено несколько методов обратной разработки приложений, но обфускация способна нейтрализовать многие из них. Например, самый эффективный способ найти исходную точку - искать текст в файлах классов. При кодировании строк поиск ничего не даст, поскольку строки не хранятся как обычный текст. Имена пакетов и классов тоже больше нельзя использовать для изучения структуры приложения и выбора хорошей исходной точки. Технически всё ещё возможно декомпилировать точку входа приложения среднего размера и двигаться по потоку управления, но на практике это неосуществимо.

Для кода с обфусцированным потоком управления самый разумный способ изучить реализацию приложения - воспользоваться старым добрым отладчиком. Большинство IDE имеет средства отладки, однако в нашем случае потребуется мощный отладчик, способный работать без исходного кода. Чтобы найти хорошую исходную точку для декомпиляции, приложение нужно запустить в режиме отладки. В Java есть стандартный программный интерфейс для отладчиков под названием Debugger API (кто бы мог подумать!), поддерживающий как локальную, так и удалённую отладку. Удалённая отладка позволяет отладчику подключиться к приложению, работающему в режиме отладки, и является предпочтительным способом взлома приложения. Хорошие отладчики показывают подробные сведения о выполняющихся потоках, стеки вызовов каждого потока, загруженные классы и находящиеся в памяти объекты. Они позволяют устанавливать точки останова и трассировать выполнение методов. Главное при работе с обфусцированными приложениями - использовать обычный интерфейс (UI или программный API), чтобы перейти к интересующей функции, а затем с помощью отладчика выяснить, какой класс или классы её реализуют. После определения классов их можно декомпилировать и изучить, как описано в главе 2. Работа с отладчиками подробно рассматривается в главе 9 «Взлом кода с помощью нестандартных отладчиков».

Короткий тест

1. Какие средства позволяют защитить интеллектуальную собственность в приложениях Java?

2. Какие преобразования обфускаторов обеспечивают наиболее надёжную защиту?

3. Какие преобразования могут вызвать каждую из возможных проблем, перечисленных в этой главе?

4. Какой способ изучения обфусцированного кода наиболее эффективен?

Краткие итоги

• Обфускация может быть лучшим способом защиты интеллектуальной собственности в байт-коде Java.

• Обфускаторы выполняют некоторые или все из следующих преобразований: удаляют отладочные сведения, искажают имена, кодируют строки, изменяют поток управления, вставляют повреждённый код, удаляют неиспользуемый код и оптимизируют байт-код.

• Обфускация затрудняет сопровождение, но эти трудности можно свести к минимуму правильной настройкой обфускатора.

• Обфусцированный код по-прежнему остаётся читаемым, если не применяются обфускация потока управления и кодирование строк.

Глава 4. Взлом непубличных методов и переменных класса

> «Заработать можно заставить что угодно, если достаточно долго с ним возиться. Если же возиться ещё дольше, вы его сломаете».
>
> *Законы Мерфи о технике*

В этой главе

• Проблема инкапсуляции
• Доступ к членам класса с пакетной и защищённой видимостью
• Доступ к закрытым членам класса
• Короткий тест
• Краткие итоги

Проблема инкапсуляции

Инкапсуляция - один из столпов объектно-ориентированного программирования. Её назначение состоит в отделении интерфейса от реализации и обеспечении модульности компонентов приложения. Обычно рекомендуется объявлять члены данных закрытыми или защищёнными и предоставлять открытые функции доступа и изменения, также известные как функции чтения и записи. Иногда также рекомендуют объявлять внутренние методы реализации закрытыми или открытыми, чтобы защитить класс от неправильного использования. Следование принципу инкапсуляции помогает создать более качественное приложение, но порой становится препятствием для такого применения, которого разработчик класса не предусмотрел.

В наших экспериментах будет использоваться `java.awt.BorderLayout`. Возможно, это когда-нибудь побудит инженеров JavaSoft добавить открытые методы. Исходный код `BorderLayout` мы получим из файла `src.jar` в каталоге установки JDK.

Доступ к членам класса с пакетной и защищённой видимостью

Сначала продемонстрируем, как легко получить доступ к переменным и методам с пакетной видимостью. В примере используется переменная с пакетной видимостью, но этот приём столь же хорошо работает и с защищённой видимостью. Переменная или метод имеют пакетную видимость, если при объявлении не указано явное ключевое слово видимости, такое как `public`, `protected` или `private`. `BorderLayout` хранит компонент, добавленный с ограничением `BorderLayout.CENTER`, в переменной `center`, объявленной следующим образом:

```
package java.awt;

public class BorderLayout implements LayoutManager2, java.io.Serializable {
    ...
    Component center;
    ....
}
```

Напомним, что члены с пакетной видимостью доступны объявившему их классу и всем классам того же пакета. В нашем примере любой класс пакета `java.awt` может напрямую обратиться к переменной `center`. Поэтому простое решение состоит в том, чтобы создать в пакете `java.awt` вспомогательный класс `AwtHelper` и через него обращаться к членам экземпляров `BorderLayout` с пакетной видимостью. В `AwtHelper` есть открытая функция, которая принимает экземпляр `BorderLayout` и возвращает компонент для заданного ограничения компоновки:

> **Истории с передовой.** WebCream - продукт, преобразующий приложения Java AWT и Swing в интерактивные веб-сайты HTML. Для этого он эмулирует графическую среду для приложения с графическим пользовательским интерфейсом (GUI), выполняющегося на стороне сервера, перехватывает текущее верхнее окно и преобразует его в страницу HTML. Чтобы создать HTML, WebCream перебирает все контейнеры и пытается воспроизвести компоновки Java с помощью таблиц HTML. Одна из компоновок, которую должен поддерживать WebCream, - `BorderLayout`. Для контейнера с `BorderLayout` модулю визуализации HTML нужно знать, какой дочерний компонент добавлен в область South, какой в North и так далее. `BorderLayout` хранит эти сведения в переменных-членах `south`, `north` и других и даже предоставляет метод `getChild()`, возвращающий компонент. Проблема состоит в том, что переменные объявлены с пакетной видимостью, а метод `getChild` - как закрытый. Чтобы обойти отсутствие открытого доступа к дочерним компонентам `BorderLayout`, инженерам WebCream пришлось применить методы взлома, описанные в этой главе.

```
package java.awt;

public class AwtHelper {
    public static Component getChild(BorderLayout layout, String key) {
        Component result = null;

        if (key == BorderLayout.NORTH)
            result = layout.north;
        else if (key == BorderLayout.SOUTH)
            result = layout.south;
        else if (key == BorderLayout.WEST)
            result = layout.west;
        else if (key == BorderLayout.EAST)
            result = layout.east;
        else if (key == BorderLayout.CENTER)
            result = layout.center;

        return result;
    }
}
```

Напишем тестовый класс `covertjava.visibility.PackageAccessTest`, который с помощью `AwtHelper` получает экземпляр разделённой панели из `MainFrame` приложения Chat. Больше всего нас интересует следующий фрагмент исходного кода:

```
Container container = createTestContainer();

if (container.getLayout() instanceof BorderLayout) {
    BorderLayout layout = (BorderLayout)container.getLayout();
    Component center = AwtHelper.getChild(layout, BorderLayout.CENTER);
    System.out.println("Center component = " + center);
}
```

Мы получаем компоновку контейнера и, если это `BorderLayout`, с помощью `AwtHelper` извлекаем центральный компонент. В центре `MainFrame` приложения Chat находится разделённая панель, поэтому при правильном коде в системной консоли должен появиться экземпляр `JSplitPane`. При запуске `PackageAccessTest` мы получаем следующее исключение:

```
java.lang.SecurityException: Prohibited package name: java.awt
```

Исключение создаётся потому, что `java.awt` считается системным пространством имён, которым обычные классы пользоваться не должны. При попытке взломать член стороннего класса с пакетной видимостью этого бы не произошло, но мы намеренно выбрали системный класс, чтобы показать реальный пример.

Единственная возможная проблема при применении этого приёма к несистемному пространству имён, например `com.mycompany.mypackage`, возникает, если пакет запечатан. Чтобы добавить вспомогательный класс в запечатанный пакет, нужен тот же приём, который применяется для добавления исправленного класса в главе 5 «Замена и исправление классов приложения».

Добавить системный класс несколько сложнее, поскольку такие классы загружаются и обрабатываются иначе, чем классы приложений. Системные классы всесторонне рассматриваются в главе 16 «Перехват потока управления». Пока достаточно сказать, что класс, добавляемый в системный пакет, должен находиться в пути загрузки основных классов. Каталог или файл JAR можно поместить в начало или конец этого пути с помощью параметра `-Xbootclasspath` в командной строке `java`. Поскольку у приложения Chat уже есть подкаталог `patches`, воспользуемся им и для системных классов. Изменим `build.xml`, чтобы переместить каталог `java.lang` с `AwtHelper` в `distrib/patches`, и создадим в `distrib/bin` новый сценарий `package_access_test.bat`:

```
@echo off
set CLASSPATH=..\lib\chat.jar
java -Xbootclasspath/p:..\patches covertjava.visibility.PackageAccessTest
```

Запуск `package_access_test.bat` даёт следующий результат:

```
C:\Projects\CovertJava\distrib\bin>package_access_test.bat
Testing package-visible access
Center component = javax.swing.JSplitPane[,0,0,0x0,...]
```

Необходимость помещать классы в путь загрузки основных системных классов несколько усложняет развёртывание, поскольку требует изменить сценарий запуска. Например, веб-приложение, развёртываемое в веб-контейнере наподобие Tomcat или WebLogic, уже нельзя просто развернуть через консоль или каталог развёртывания приложений. Сценарий запуска сервера приложений придётся изменить, добавив параметр `-Xbootclasspath`. Другой недостаток этого приёма заключается в том, что он не работает с закрытыми членами. И последнее, но не менее важное: добавление классов в пакеты может нарушать лицензионное соглашение. Именно так обстоит дело с `BorderLayout`, поскольку один из разделов лицензионного соглашения Java компании Sun прямо запрещает добавлять классы в пакеты, имена которых начинаются с `java`. В следующем разделе представлена другая возможность, устраняющая некоторые из этих проблем.

Доступ к закрытым членам класса

Закрытые члены доступны только объявившему их классу. Это одно из основных правил языка Java, обеспечивающих инкапсуляцию. Но так ли это на самом деле? Всегда ли правило соблюдается? Если вы подумали: «Раз этот человек об этом пишет, значит, какая-то лазейка должна быть», то оказались правы. Компилятор Java контролирует закрытость членов во время компиляции. Поэтому другие классы не могут содержать статических ссылок на закрытые методы и переменные класса.

Однако в Java есть мощный механизм рефлексии, позволяющий запрашивать метаданные экземпляров и классов и обращаться к полям и методам во время выполнения. Поскольку рефлексия динамична, проверки времени компиляции к ней неприменимы. Вместо этого среда выполнения Java полагается на диспетчер безопасности, если он существует, чтобы проверить достаточность привилегий вызывающего кода для конкретного вида доступа. Диспетчер безопасности обеспечивает достаточную защиту, поскольку все функции API рефлексии перед выполнением своей логики делегируют проверку ему. Ослабляет эту защиту то, что диспетчер безопасности часто не задан. По умолчанию он отсутствует, и если код явно не устанавливает стандартный или собственный диспетчер, проверки управления доступом во время выполнения не действуют. Даже установленный диспетчер безопасности обычно настраивается файлом политики, который можно расширить, разрешив доступ к API рефлексии.

Если вы внимательно посмотрели на класс `BorderLayout`, то могли заметить, что в нём уже есть метод, возвращающий дочерний компонент по ключу положения. Неудивительно, что он называется `getChild` и имеет следующую сигнатуру:

```
private Component getChild(String key, boolean ltr)
```

Это хорошая новость: писать собственную реализацию не требуется. Проблема в том, что метод объявлен закрытым и вызвать его через открытый метод невозможно. Чтобы воспользоваться существующим кодом JDK, нужно вызвать `BorderLayout.getChild()` посредством API рефлексии. Мы используем ту же структуру теста, что и в предыдущем разделе. Но на этот раз тестовый класс вместо `AwtHelper` делегирует работу собственной вспомогательной функции `getChild()`:

```
public class PrivateAccessTest {
    public static void main(String[] args) throws Exception {
        Container container = createTestContainer();

        if (container.getLayout() instanceof BorderLayout) {
            BorderLayout layout = (BorderLayout)container.getLayout();
            Component center = getChild(layout, BorderLayout.CENTER);
            System.out.println("Center component = " + center);
        }
        ...
    }

    public static Component getChild(BorderLayout layout, String key)
        throws Exception {
        Class[] paramTypes = new Class[]{String.class, boolean.class};
        Method method = layout.getClass().getDeclaredMethod("getChild", paramTypes);

        // По умолчанию закрытые методы недоступны
        method.setAccessible(true);
        Object[] params = new Object[] {key, new Boolean(true)};
        Object result = method.invoke(layout, params);
```

```
            return (Component)result;
        }
        ...
    }
```

Реализация `getChild()` составляет суть приёма. Она получает объект метода посредством рефлексии, а затем вызывает `setAccessible(true)`. Значение `true` подавляет проверку управления доступом и разрешает вызов метода. Остальная часть метода представляет собой обычное применение API рефлексии. Запуск `covertjava.visibility.PrivateAccessTest` даёт тот же результат, что и в предыдущем разделе:

```
C:\Projects\CovertJava\distrib\bin>private_access_test.bat
Testing private access
Center component = javax.swing.JSplitPane[,0,0,0x0,...]
```

Это оказалось пугающе просто. Если диспетчер безопасности установлен посредством `System.setSecurityManager` или из командной строки, как бывает в большинстве серверов приложений и связующего программного обеспечения, придётся потрудиться чуть больше. Если запустить наш тест, передав в командной строке `java` параметр `-Djava.security.manager`, мы получим следующее исключение:

```
java.security.AccessControlException: access denied
(java.lang.RuntimePermission accessDeclaredMembers)
```

Чтобы код работал при установленном диспетчере безопасности, необходимо предоставить разрешения на доступ к объявленным членам посредством рефлексии и на подавление проверок доступа. Для этого добавим файл политики Java, предоставляющий нашей базе кода два разрешения:

```
grant {
    permission java.lang.RuntimePermission "accessDeclaredMembers";
    permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
};
```

Наконец, создадим в каталоге `distrib\bin` новый тестовый сценарий `private_access_test.bat`, который добавляет параметр командной строки `java.security.policy` для установки нашего файла политики:

```
set CLASSPATH=..\lib\chat.jar
set JAVA_ARGS=%JAVA_ARGS% -Djava.security.manager
set JAVA_ARGS=%JAVA_ARGS% -Djava.security.policy=../conf/java.policy
java %JAVA_ARGS% covertjava.visibility.PrivateAccessTest
```

Если файл политики уже установлен, нашу инструкцию `grant` нужно вставить в него. Файлы безопасности Java позволяют подключать дополнительные файлы политики с помощью атрибута `policy.url.n`. Безопасность Java и файлы политики подробно рассматриваются в главе 7 «Управление безопасностью Java».

Приём, основанный на API рефлексии, можно применять и для доступа к членам с пакетной и защищённой видимостью. Благодаря этому больше не нужно вставлять вспомогательные классы в сторонние пакеты. Недостаток API рефлексии - общеизвестная медлительность, поскольку ему приходится работать со сведениями времени выполнения и, возможно, проходить несколько проверок безопасности. Когда важна скорость, для членов с пакетной и защищённой видимостью предпочтительнее вспомогательные классы. Ещё одна возможность - сериализовать экземпляр в поток массива байтов, а затем разобрать поток и получить значения переменных-членов. Очевидно, это утомительный процесс, который не работает с переходными полями.

Короткий тест

1. Какой приём позволяет получить значение защищённой переменной?

2. Какой приём позволяет получить значение закрытой переменной?

3. Каковы преимущества и недостатки каждого приёма?

Краткие итоги

• Доступ можно получить даже к методам и переменным, которые не объявлены открытыми.

• К члену с пакетной или защищённой видимостью можно обратиться, вставив в его пакет вспомогательный класс либо применив API рефлексии.

• К члену с закрытой видимостью можно обратиться с помощью API рефлексии.

• Если установлен диспетчер безопасности, политику Java необходимо изменить так, чтобы разрешить API рефлексии неограниченный доступ.

Глава 5. Замена и исправление классов приложения

> «Сколько инженеров-программистов нужно, чтобы заменить лампочку?»
>
> «Ни одного. Мы опишем это в руководстве».
>
> «Ни одного. Это аппаратная проблема».
>
> «Один, но он будет свободен только в 2010 году».
>
> «Двое. Один всегда уходит посреди проекта».
>
> «Четверо. Один спроектирует изменение, второй реализует, третий задокументирует, а четвёртый потом будет его сопровождать».

В этой главе

- Что делать, когда мы испробовали все пути, но потерпели неудачу?
- Поиск класса, который необходимо исправить
- Пример ситуации, требующей исправления
- Исправление класса для реализации новой логики
- Перенастройка приложения для загрузки и использования исправленного класса
- Исправление запечатанных пакетов
- Короткий тест
- Краткие итоги

Что делать, когда мы испробовали все пути, но потерпели неудачу?

Практически каждый разработчик когда-либо пользовался библиотекой или компонентом, созданным другим человеком. Практически каждый разработчик когда-либо и доходил из-за библиотеки до такого отчаяния, что был готов разыскать человека, решившего объявить метод закрытым, и попытаться образумить его. Что ж, большинство из нас не зашло бы так далеко, однако возможность изменить вещи, которые делают нашу жизнь невыносимой, безусловно, была бы полезна. Дело не в том, что библиотеки пишут злые люди; просто даже самые блестящие проектировщики не способны предвидеть все возможные способы, которыми другие разработчики захотят воспользоваться их кодом.

Разумеется, всегда лучше решать вопросы мирным путём. Если можно убедить поставщика изменить код или вы сами вправе это сделать, непременно так и поступите. Но уже один тот факт, что вы читаете эту книгу, доказывает: в реальной жизни традиционный подход работает не всегда. И вот тут начинается самое интересное. Итак, когда следует прибегать к замене и исправлению классов? Ниже перечислены несколько ситуаций, требующих хакерского подхода:

• **Вы используете стороннюю библиотеку, в которой есть нужная возможность, но она не предоставлена через открытый API.** Например, до JDK 1.4 Java Swing не предоставляла метода для получения списка слушателей `JComponent`. Компонент хранил слушателей в переменной с пакетной видимостью без открытого доступа, поэтому программно выяснить, имеются ли у него слушатели событий, было невозможно.

• **Вы используете сторонний класс или интерфейс, но предоставляемые им функции недостаточно гибки для вашего приложения.** Простое изменение API может сэкономить несколько дней работы или оказаться единственным решением проблемы. В таком случае вас устраивает 99% библиотеки, но оставшийся 1% мешает эффективно ею пользоваться.

• **В используемом продукте или API есть ошибка, а ждать исправления от поставщика нельзя.** Например, JRun 3.0 содержал ошибку определения версии JVM в HP UX. Разбирая строку версии, сообщаемую средой выполнения Java, он ошибочно заключал, что работает под управлением старой версии JDK, и отказывался запускаться.

• **Вам нужна очень тесная интеграция с продуктом, но его архитектура недостаточно открыта для ваших требований.** Многие каркасы отделяют интерфейсы от реализации. Внутри программы доступ к функциям осуществляется через интерфейсы, а реализацию предоставляют экземпляры конкретных классов. Основные библиотеки Java по большей части позволяют указывать классы реализации через системные свойства. Так обстоит дело с AWT Toolkit и анализатором SAX: классы реализации можно задать системными свойствами `java.awt.toolkit` и `org.xml.sax.driver` соответственно. Взлом потребуется, если нужно предоставить другой класс реализации для библиотеки, не имеющей средств настройки.

• **Вы пользуетесь сторонним кодом, но ожидаемая функция не работает.** Вы не знаете, вызвано ли это неправильным использованием или ошибкой в коде. В документации проблема не упоминается, а обходного решения нет. Временная вставка отладочной трассировки и сообщений в сторонний код поможет понять происходящее в системе.

• **Возникла срочная проблема в промышленной среде, которую необходимо устранить.** При этом нельзя рисковать повторным развёртыванием нового кода в промышленной среде. Для решения требуется небольшое изменение кода, затрагивающее лишь несколько классов.

При работе со сторонним кодом вы можете нарушить лицензионное соглашение, поэтому обязательно прочитайте его и для надёжности согласуйте действия с юридическим отделом. Законы об авторском праве могут соблюдаться очень строго, а изменение стороннего кода часто незаконно. Получите у поставщика разрешение на реализацию решения, а не принимайте ответственность за взлом на себя.

Хорошая новость в том, что при использовании представленного в этой главе метода вы не вносите прямых изменений в библиотеку или продукт. Вы не вмешиваетесь в их код, а предоставляете замену функции, которая вас не устраивает. В каком-то смысле это похоже на создание производного класса от класса поставщика и переопределение метода, хотя здесь легко ступить на скользкую дорожку. Оставив юридические вопросы в стороне, посмотрим, как это сделать.

Поиск класса, который необходимо исправить

Прежде всего нужно определить, какой код следует исправить. Иногда это совершенно очевидно, и конкретный класс или интерфейс известен сразу. Если вы считаете себя слишком умным, чтобы слушать объяснение поиска кода, смело переходите к разделу об исправлении. В противном случае устройтесь поудобнее и познакомьтесь с несколькими способами достижения результата.

Общий подход

Общий метод поиска исправляемого класса состоит в том, чтобы найти исходную точку и двигаться по последовательности выполнения, пока не будет обнаружен код, который требуется изменить. Если нужного кода нет поблизости от исходной точки, необходимо выбрать новую исходную точку и повторить процесс.

> **Истории с передовой.** Компания AT&T Wireless обновляла систему ввода заказов для активации сотовых телефонов. Написанную на Java систему требовалось перенести с JDK 1.2 на 1.3. Обновление имело решающее значение из-за исправлений ошибок в Swing и других пакетах Java, а также потому, что пользователи ожидали более удобного решения. Повышение производительности и оптимизация потребления памяти в JDK 1.3 также были важными основаниями для обновления. Разработка велась в Windows, но промышленной средой служила HP UX. Когда все ошибки были исправлены, а модульные тесты под JDK 1.3 завершены, новую версию Java установили на промежуточный сервер HP UX для начала формального интеграционного тестирования.
>
> К сожалению, выяснилось, что сервер приложений, использовавшийся для служб J2EE, содержал ошибку определения версии JDK. Ошибка возникала при разборе строки версии из системных свойств, но поскольку в Windows и Unix эта строка различалась, проблема проявилась только на этапе интеграционного тестирования. Сервер приложений отказывался работать в HP, полагая, что выполняется под более старой версией Java. Возвращаться к JDK 1.2 было уже поздно, а готового исправления не существовало. Чтобы спасти проект, инженеры декомпилировали класс сервера приложений, проверявший версию, и самостоятельно исправили ошибку. После развёртывания исправления сервер запустился и безупречно работал в промышленной среде. К сожалению, в награду никого из инженеров не отправили на Ямайку, как обещало руководство, но такова жизнь.

Для быстрого результата крайне важна хорошая исходная точка. Иногда выбор начального класса достаточно очевиден. Например, при исправлении API или логики исходной точкой будет интерфейс или класс, который вы хотите изменить. Если закрытый метод класса нужно сделать открытым, исходной точкой станет этот класс. Если требуется устранить ошибку, вызывающую исключение Java, исходной точкой будет класс на вершине трассировки стека.

В любой ситуации после определения исходного класса следует получить его исходный код, при необходимости декомпилировав байт-код, а затем перейти к классу, который действительно нужно исправить. Этот процесс напоминает движение по диаграмме последовательности: вы начинаете с только что описанного метода и исследуете каждый вызываемый по пути класс. В крупных системах с сотнями классов может потребоваться определить несколько исходных точек и выбрать ту, которая обеспечивает кратчайший путь к изменяемому коду.

Поиск текстовых строк

В большой сложной системе есть десятки пакетов и сотни классов. Без ясной исходной точки легко заблудиться, пытаясь проследить логику приложения. Вспомните код запуска такого сервера приложений, как WebLogic. При запуске WebLogic выполняет сотни задач и использует для этого множество потоков, и даже при неограниченном запасе кофе я не советовал бы взламывать его, двигаясь от класса `weblogic.Server`.

Самый надёжный подход в таких случаях - текстовый поиск строки, которая, как известно, находится рядом с целевым классом. Хорошо написанные продукты и библиотеки можно настроить на запись обширных отладочных сведений в файл журнала. Помимо очевидной пользы для сопровождения и устранения неполадок, это облегчает поиск кода, отвечающего за нужную функцию. Если настроить приложение на подробное журналирование последовательности выполнения, а затем воспроизвести проблему, последнее успешное или первое ошибочное сообщение журнала позволит определить исходную точку. Как вам, возможно, известно, байт-код хранит строки как обычный текст, а значит, во всех файлах `.class` можно искать подстроку, встреченную в журнале. Предположим, что при использовании каркаса безопасности для некоторых имён создаётся исключение с текстом `Invalid username`. Причина отказа неизвестна, как и решение. Если трассировка стека недоступна, проще всего найти `Invalid username` во всех файлах `.class` каркаса. Скорее всего, во всём коде обнаружится один или два экземпляра, и декомпиляция файла класса позволит понять первопричину. Точно так же во всех файлах классов можно искать имя метода или класса, метку GUI, подстроку страницы HTML или любую другую строку, которая, по вашему мнению, встроена в код Java.

Работа с обфусцированным кодом

Хуже всего, когда приходится иметь дело с обфусцированным кодом. Хороший обфускатор переименовывает пакеты, классы, методы и переменные. Лучшие продукты на рынке даже кодируют строки Java, поэтому поиск сообщения трассировки может ничего не дать. Тогда задача превращается в адский труд по поэтапному изучению приложения. Здесь требуется более творческий подход, иначе поиск будет напоминать поиски иголки в стоге сена. Знание принципов обфускации помогает ориентироваться. Хотя обфускатор вправе изменить имена классов и методов приложения, с системными классами он этого сделать не может. Например, если библиотека проверяет наличие файла и создаёт исключение при его отсутствии, двоичный поиск строки исключения может ничего не дать, если обфускатор достаточно умён, чтобы её закодировать. Однако поиск `File` или `FileInputStream` способен привести к связанному коду. Аналогично, если приложение неправильно читает системную дату или время, можно искать `java.util.Date` или метод `getTime` класса `Calendar`. Главная проблема в том, что обфусцированные классы после декомпиляции не всегда удаётся скомпилировать заново. Дополнительные сведения приведены в главе 2 «Декомпиляция классов».

Пример ситуации, требующей исправления

Мы изменим представленное ранее приложение Chat так, чтобы в окне разговора показывались имя пользователя и имя узла, а не только имя узла. Как вы помните, исходное приложение перед каждым полученным сообщением выводит имя узла и двоеточие, как показано на рисунке 5.1.

Так реализовать служебную программу проще, но пользователям, безусловно, важнее видеть, от какого человека поступило сообщение, а не с какого компьютера оно отправлено. Chat допускает бесплатное использование и усовершенствование, но его исходный код отсутствует.

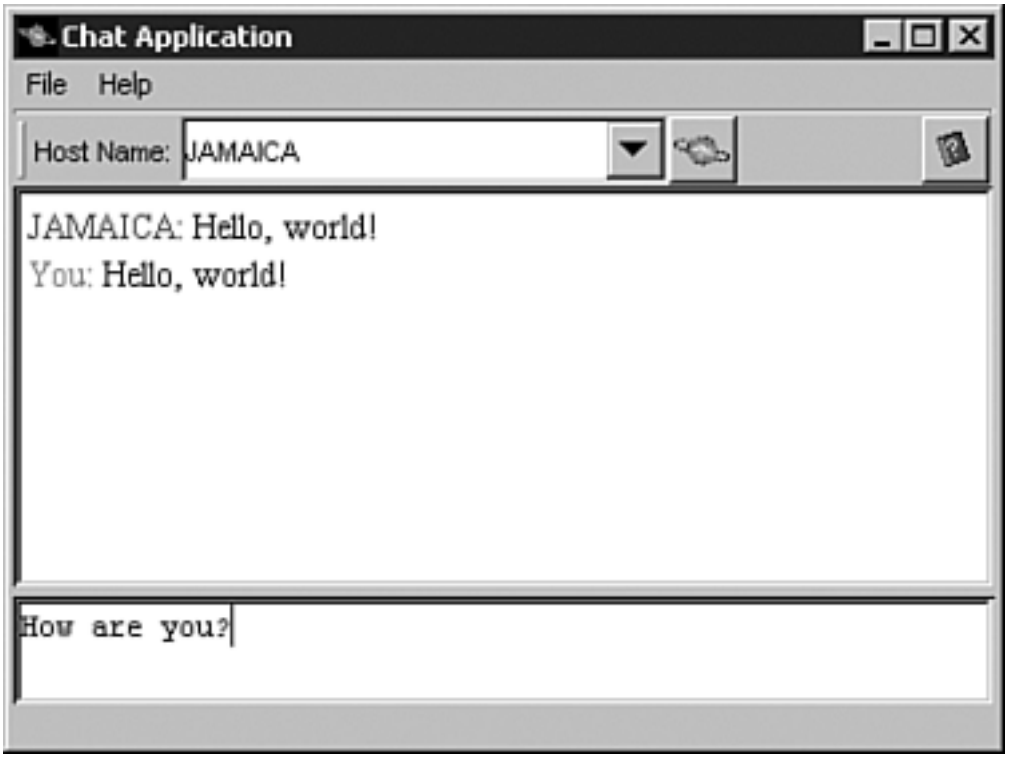

Рисунок 5.1. Главное окно исходного приложения Chat

Как обычно бывает с приложениями Java, байт-код поставляется в одном или нескольких файлах JAR, поэтому сначала нужно создать рабочий каталог и распаковать в него все библиотеки JAR. Это облегчает навигацию и даёт прямой доступ к файлам `.class`, которые являются предметом нашего исследования. Создав рабочий каталог и выполнив `jar xf chat.jar`, мы увидим следующие файлы:

```
images
AboutDialog.class
ChatApplication.class
ChatServer.class
ChatServerRemote.class
MainFrame.class
MainFrame$1.class
MessageInfo.class
MessageListener.class
```

Попробуем все описанные выше способы поиска исходной точки и посмотрим, какой из них лучше всего подходит для этого приложения.

Использование имени класса

К счастью, байт-код не обфусцирован, поэтому можно посмотреть на имена классов и попытаться выбрать победителя. Пятисекундного изучения должно хватить, чтобы прийти к выводу: лучшим кандидатом для первого просмотра является `MainFrame`. В декомпилированном коде видно, что вся запись разговора выполняется методом `appendMessage`, который выглядит так:

```
void appendMessage(String message, MessageInfo messageInfo) {
    if (messageInfo == null) {
        this.conversation.append("<font color=\"red\">");
        this.conversation.append("You");
    }
    else {
        this.conversation.append("<font color=\"blue\">");
        this.conversation.append(messageInfo.getDisplayName());
    }
    this.conversation.append(": ");
    this.conversation.append("</font>");
    this.conversation.append(message);
    this.conversation.append("<br>");
    this.txtConversation.setText(this.conversation.toString() +
        "</BODY></HTML>");
}
```

В реализации метода для получения имени отправителя используется метод `getDisplayName()` класса `MessageInfo`. Это приводит нас к декомпиляции класса `MessageInfo`, чтобы получить показанную ниже реализацию `getDisplayName`:

```
public String getDisplayName() {
    return getHostName();
}
```

Бинго! Мы выяснили, что пользовательский интерфейс Chat полагается на `MessageInfo`, а текущая реализация использует только имя узла. Следовательно, наша задача - исправить `MessageInfo.getDisplayName()`, чтобы он использовал и имя узла, и имя пользователя.

Поиск текстовых строк

Представим, что Chat - большое приложение, содержащее более 500 классов во множестве пакетов. Надеяться угадать нужный класс по имени - всё равно что надеяться на правильную работу программы после первой компиляции. Для получения исходной точки нужен более надёжный метод. Служебная программа Chat пишет неплохие сообщения журнала, поэтому попробуем воспользоваться ими. Запустив её, отправим сообщение другому пользователю, получим ответ и увидим в консоли Java следующий вывод:

```
Initializing the chat server...
Trying to get the registry on port 1149
Registry was not running, trying to create one...
ChatApplication server initialized
Sending message to host JAMAICA: test
Received message from host JAMAICA
```

Нетрудно догадаться, что новое сообщение добавляется в историю разговора при отправке или получении. Также вполне очевидно, что такие сведения, как имя узла-отправителя или узла-получателя, не являются частью статической строки. Поэтому в качестве критерия поиска по всем файлам `.class` рабочего каталога используем `Received message from host`. Поиск даёт один файл, `ChatServer.class`, который мы немедленно декомпилируем в `ChatServer.jad`. Поиск строки в декомпилированном исходном коде приводит к следующему методу `receiveMessage()`:

```
public void receiveMessage(String message, MessageInfo messageInfo)
    throws RemoteException
{
    System.out.println("Received message from host " +
        messageInfo.getHostName());
    if(messageListener != null)
        messageListener.messageReceived(message, messageInfo);
}
```

Поиск `messageListener` в `ChatServer.jad` показывает, что это интерфейс, а экземпляр слушателя задаётся методом `setMessageListener()`. Теперь у нас два варианта. Можно найти классы, реализующие `MessageListener`, и выяснить, какой из них, если реализаций несколько, связан с `ChatServer`. Другой подход основан на том, что имена методов Java хранятся в байт-коде как текст. Поскольку код не обфусцирован, можно найти `setMessageListener()` во всех файлах классов. Мы воспользуемся вторым способом. В нашем случае поиск возвращает два класса: `ChatServer` и `MainFrame`. Мы заключаем, что только `MainFrame` выступает слушателем `ChatServer`, и декомпилируем его. Остальное исследование выполняется точно так же, как в предыдущем разделе, где `MainFrame` был найден по имени класса. В демонстрационном приложении угадать исходную точку по имени оказалось быстрее, но здесь присутствует определённая доля удачи. Поиск по сообщениям журнала надёжнее и лучше работает для большинства реальных приложений.

Использование стека вызовов для навигации по логике приложения

Многие проблемы Java проявляются через исключения. Исключения могут создаваться классами среды выполнения Java или самим приложением, и обычно сообщения об ошибке вместе с типом исключения достаточно для решения проблемы. Но эта книга существует именно потому, что в жизни не всё просто. Можно получить `NullPointerException` или исключение без сообщения об ошибке, а при работе со сторонним кодом не иметь ни малейшего представления о способе обхода. Если лицензия не запрещает декомпилировать исходный код либо сам исходный код доступен, можно начать поиск значительно менее болезненным способом.

Самый простой и надёжный способ понять причину исключения - использовать стек вызовов. Следует знать, что операционные системы отслеживают вызовы методов с помощью стека. Если метод A вызывает метод B, сведения об A помещаются в стек. Если B затем вызывает C, в стек также помещаются сведения о B. Когда каждый метод возвращает управление, по стеку определяется, какой метод должен продолжить выполнение. В Java получить доступ к стеку вызовов можно через отладчик, вызвав `printStackTrace()` для исключения, либо с помощью метода `Thread.dumpStack()`. Для серверных приложений использование отладчика обычно слишком обременительно, поэтому в нашем примере я предлагаю два последних способа. Метод исключения `printStackTrace()` широко распространён и никого не удивит. Метод потока `dumpStack` встречается реже, но чрезвычайно полезен, если нужно выяснить, кто вызывает метод во время выполнения. Просто добавьте:

```
Thread.dumpStack();
```

в тело исследуемого метода и запустите приложение. При каждом вызове метода вы будете точно знать, как выполнение к нему пришло и какие другие методы следует исследовать.

Стек вызовов подробнее рассматривается в главе 16 «Перехват потока управления».

Исправление класса для реализации новой логики

Теперь, когда известно, что именно нужно исправить, сама работа относительно проста. Получите исходный файл непосредственно из дистрибутива или декомпилируйте файл `.class`, предварительно убедившись, что лицензия этого не запрещает. Для удобства сопровождения все исправленные классы следует держать в отдельном каталоге, например `patches`. Воспроизведите структуру каталогов, соответствующую структуре пакета класса, и внесите изменения с помощью любимой IDE либо обычного редактора и консольного компилятора. Помните, что библиотеку, скорее всего, когда-нибудь придётся обновить, поэтому каждый вставленный фрагмент кода необходимо снабдить комментариями. Тогда после получения новой версии исходного класса изменения будет легко повторить.

По той же причине изменения следует изолировать и держать вместе в одном месте файла. Если добавляется заметный объём кода, подумайте о создании вспомогательного класса и делегировании ему работы, чтобы как можно меньше менять исправляемый файл.

В нашем примере нужно предоставить новую реализацию метода getDisplayName(), использующую шаблон UserName (HostName). Создадим в каталоге установки приложения новый каталог patches с подкаталогом covertjava.chat, скопируем туда MessageInfo.jad и переименуем его в MessageInfo.java. В MessageInfo уже есть метод getUserName(), поэтому требуется лишь соединить две строки. Перепишем getDisplayName() следующим образом:

```
public String getDisplayName() {
    return getUserName() + " (" + getHostName() + ")";
    // *** исходная логика исправлена Alex Kalinovsky согласно новым требованиям
    // return getHostName();
}
```

Затем скомпилируем MessageInfo.java. Теперь можно обновить приложение, чтобы новая логика вступила в силу. Изменения также следует задокументировать для последующего сопровождения кода.

Перенастройка приложения для загрузки и использования исправленного класса

Именно эта задача заставляет весь приём работать. Нужно добиться, чтобы JVM использовала предоставленную нами исправленную версию класса вместо старой. Это приводит нас к краеугольному камню программирования на Java - правильной настройке CLASSPATH. Да, всё настолько просто. Нужно лишь убедиться, что архив Zip или JAR либо каталог с исправленной версией класса находится в пути поиска классов раньше архива или каталога с оригиналом. В сценарии запуска приложения, настраивающем CLASSPATH, этот архив или каталог следует указать первым. Если держать все исправления вместе, но отдельно от исправляемых продуктов и библиотек, их легко сопровождать. Получив новую версию библиотеки, можно перезаписать старые файлы JAR, не опасаясь потерять исправления. Разумеется, всё нужно снова протестировать, а поскольку исправление опирается не только на открытые интерфейсы, но и на закрытые реализации, его может потребоваться привести в соответствие с новой библиотекой. И последнее: если понадобится объяснить кому-либо, что, чёрт возьми, вы сделали с системой, можно указать на каталог patches и документацию.

Как говорится, «всё не то, чем кажется». Иногда можно думать, что система загружает новый класс, хотя в действительности загружается старая версия. Лучший способ убедиться в использовании новой версии - добавить в новый класс отладочную трассировку или хотя бы грубый вызов System.out.println(). Запустив приложение, обязательно проверьте появление трассировки, после чего её можно удалить. В демонстрационном коде мы добавили статический инициализатор, выводящий сообщение о действующем исправлении:

```
static {
    // Записать в журнал факт действия исправления
    System.out.println("Loaded patched MessageInfo");
}
```

Чтобы увидеть изменения в действии, обновим сценарий запуска Chat, включив каталог `patches`. Скопируем `bin/chat.bat` в `bin/chat_patched.bat` и откроем копию в редакторе. Затем изменим инициализацию `CLASSPATH`, чтобы каталог исправлений предшествовал файлу приложения `chat.jar`:

```
set CLASSPATH=..\patches;..\lib\chat.jar
```

Сохраним файл и запустим его. Отправим несколько тестовых сообщений на `localhost`; если всё сделано правильно, новый экран будет выглядеть так, как на рисунке 5.2.

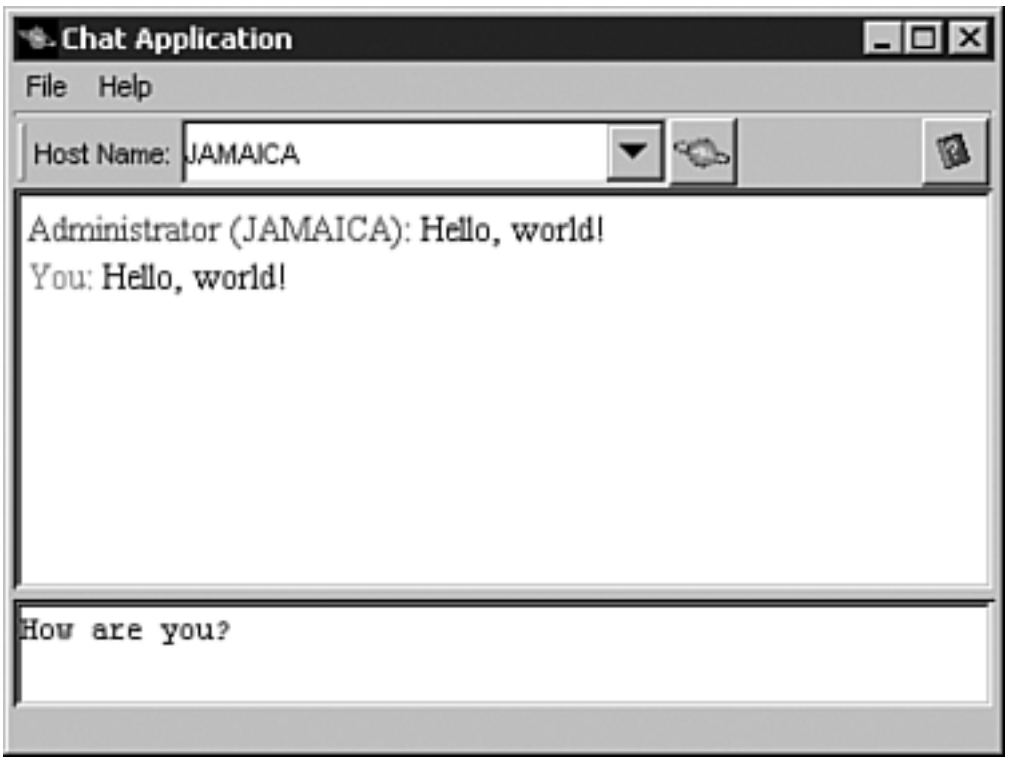

Рисунок 5.2. Главное окно исправленного приложения Chat

Если исправляется системный класс, задача становится сложнее. Системным называется класс, загруженный из пути основных классов, например `java.lang.String`. Даже если поместить исправленную версию первой в `CLASSPATH`, она не заменит оригинал. Исправление системных классов рассматривается в главе 15 «Замена и исправление основных классов Java».

Исправление запечатанных пакетов

Java поддерживает понятие запечатанных пакетов. Если пакет запечатан, все его классы должны загружаться из одного файла JAR. Для разработчиков приложений и поставщиков инструментов это замечательная возможность, призванная предотвратить только что изученный приём. Чтобы запечатать пакет, в файле манифеста JAR необходимо задать атрибут `Sealed` со значением `true`. Если файл `chat.jar` приложения Chat запечатан, запуск `bin\chat_patched.bat` создаёт следующее исключение:

```
Exception in thread "main" java.lang.ExceptionInInitializerError
at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
at sun.reflect.NativeConstructorAccessorImpl.newInstance

(NativeConstructorAccessorImpl.java:39)
at sun.reflect.DelegatingConstructorAccessorImpl.newInstance
(DelegatingConstructorAccessorImpl.java:27)
at java.lang.reflect.Constructor.newInstance(Constructor.java:274)
...
Caused by: java.lang.SecurityException: sealing violation: package covertjava.chat is
sealed
at java.net.URLClassLoader.defineClass(URLClassLoader.java:225)
at java.net.URLClassLoader.access$100(URLClassLoader.java:54)
at java.net.URLClassLoader$1.run(URLClassLoader.java:193)
at java.security.AccessController.doPrivileged(Native Method)
```

К несчастью для поставщиков и к счастью для взломщиков и любознательных людей вроде нас, запечатанные пакеты часто легко вскрыть. Достаточно изменить значение атрибута `Sealed` в манифесте JAR на `false` либо извлечь содержимое JAR в рабочий каталог и изменить сценарий запуска так, чтобы вместо исходного файла JAR использовался этот каталог. Это пример благого намерения, которое так и не было реализовано всерьёз.

Короткий тест

1. Какие подходы можно использовать для поиска исправляемого класса?

2. Почему класс можно искать по тексту и в каких случаях этот способ не работает?

3. Как получить стек вызовов в произвольной точке приложения, находящейся не внутри блока `catch`?

4. Каковы требования к установке исправления?

Краткие итоги

Чтобы исправить логику приложения, необходимо:

• найти класс, содержащий нужную логику;
• получить исходный код класса;
• изменить исходный код, реализовав новую логику;
• скомпилировать и развернуть новый файл класса;
• обновить среду запуска приложения так, чтобы новый класс загружался первым.

Глава 6. Эффективная трассировка

> «Неработающая сложная система неизменно оказывается результатом развития простой системы, которая прекрасно работала».
>
> *Законы Мерфи о технике*

В этой главе

- Введение в трассировку
- Трассировка как эффективный способ изучения программы
- Инструменты и API трассировки и журналирования
- Что следует и чего не следует делать при трассировке
- Короткий тест
- Краткие итоги

Введение в трассировку

Трассировка - это запись отладочных сообщений в поток вывода во время выполнения приложения. Она в реальном времени создаёт зафиксированный след операций, выполняемых работающим приложением. Для трассировки приложение не обязательно запускать в режиме отладки. Кроме того, поскольку сообщения трассировки обычно записываются в файл журнала, она предоставляет сведения для изучения поведения приложения и устранения неполадок. Трассировка используется уже много лет, но особую популярность приобрела в серверном программировании. Небольшое приложение легко загрузить в отладчик и выполнить по шагам, исследуя проблему. Распределённые системы обычно значительно сложнее на стороне сервера и часто работают внутри продуктов связующего программного обеспечения, например сервера приложений или веб-сервера. Из-за этого пользоваться отладчиком непрактично, а пошаговое выполнение может нарушить многопоточную работу. Для работающих промышленных приложений отладка вообще невозможна, поэтому трассировка остаётся единственным правдоподобным способом получить сведения о происходящем в системе.

Трассировка вставляется в код почти так же, как комментарии, а качественная трассировка делает комментарии ненужными. Поскольку трассировка приводит к вызовам методов и записи сообщений в поток, она может обходиться дорого, особенно внутри циклов и часто вызываемых методов. Каркасы журналирования разрабатываются так, чтобы создавать минимальные накладные расходы при выполнении кода, и позволяют уменьшать уровень сообщений трассировки, записываемых в поток вывода. Однако важно понимать: даже не слишком хорошо оптимизированная трассировка всё равно лучше полного её отсутствия. Если проблема возникает в промышленном приложении без трассировки, единственным источником сведений остаются отзывы пользователей, которые, как всем известно, часто ненадёжны или неточны. Грамотная трассировка позволяет проследить действие пользователя и ответ приложения, чего в большинстве случаев достаточно для определения источника проблемы. В каждую запись следует включать идентификатор пользователя, чтобы журнал можно было фильтровать по операциям конкретного человека.

До JDK 1.4 в Java не было стандартного API журналирования. Учитывая популярность и важность журналирования для приложений J2EE, это несколько прискорбно. Из-за столь поздней стандартизации каждому поставщику и многим разработчикам приложений пришлось создавать собственную версию API трассировки и журналирования. Сегодня часто встречаются разные версии API журналирования, иногда даже внутри одного приложения. Например, приложение J2EE, работающее в WebLogic, может использовать Log4J для собственных отладочных трассировок, но API журналирования WebLogic для сообщений о критических ошибках системного уровня. Все API журналирования поддерживают уровни сообщений, определяющие, сколько сообщений записывается в файл журнала.

> **Истории с передовой.** Несколько лет назад крупный клиент попросил меня оценить текущее состояние его проекта и порекомендовать улучшения. Дела у клиента шли плохо: и руководство, и команда разработчиков были недовольны медленным продвижением и неспособностью быстро устранять дефекты. Изучение кода показало отсутствие систематической трассировки. В лучшем случае разработчики изредка пользовались `System.out`, чтобы вывести значение переменной, но работающее приложение напоминало чёрную дыру: никто не знал, что происходит в любой конкретный момент. Даже простую проблему было трудно исправить, поскольку условия, приведшие к ней, а иногда и само её происхождение оставались неизвестны.
>
> Поэтому моей первой рекомендацией было потратить три дня на добавление журналирования в код. Вся команда отложила текущие задачи и прошлась по коду, добавляя трассировку, которая выводила имена вызываемых методов и бизнес-параметры, например имена пользователей, номера заказов и идентификаторы записей базы данных. Эти три дня окупились почти сразу. Благодаря эффективной трассировке находить и исправлять дефекты стало легко даже без отладки кода. Если часть приложения была написана кем-то другим, трассировка всё равно позволяла понять процесс выполнения. Что ещё важнее, после ввода системы в эксплуатацию файл журнала стал главным источником сведений о текущем состоянии системы и операциях, выполненных за день.

Трассировка как эффективный способ изучения программы

Мы уже говорили о важности трассировки для устранения неполадок в крупных системах. Сведения из журналов трассировки не менее ценны для понимания потока выполнения приложения и определения исходных точек обратной разработки. Поскольку трассировка предназначена для создания понятной человеку истории операций продукта, читать её легче, чем декомпилированный код Java. Файл журнала представляет снимок уже выполненной работы. Поэтому, если ошибка вызывает исключение без трассировки стека, можно включить самый подробный уровень трассировки, воспроизвести ошибку и найти последнее сообщение, попавшее в журнал до исключения. Тот же приём применим для поиска кода, который нужно исправить. Если приложение не предоставляет достаточной трассировки, в классы можно добавить сообщения и даже дампы стеков потоков посредством исправления. Работа утомительна, но может оказаться единственным эффективным способом изучения обфусцированного кода, особенно после обфускации потока управления. Трассировка устраняет необходимость гадать, поскольку её сообщения служат бесспорным доказательством того, что определённый фрагмент кода выполнялся в конкретный момент.

Если приложение ещё не выводит сообщения журнала, добавление трассировки может занять много времени и, признаем откровенно, это не та задача, о которой мечтает хороший разработчик.

Быстрый и грубый способ взглянуть на состояние времени выполнения - вывести стек вызовов текущего потока из исследуемого метода, как описано в главе 5 «Замена и исправление классов приложения». Напомним: добавление вызова метода `dumpStack()` класса `java.lang.Thread` заставляет его вывести имена методов, образующих стек вызовов. Всего одна строка кода позволяет хорошо понять, какие вызовы привели к выполнению рассматриваемого метода.

Инструменты и API трассировки и журналирования

Сегодня господствуют два API журналирования. Первый - Apache Log4J, появившийся в 1999 году в группе AlphaWorks компании IBM. С тех пор он прошёл долгий путь и сегодня, вероятно, является самым развитым и гибким API журналирования. Он бесплатен и может распространяться как отдельный файл JAR, совместимый со всеми версиями JDK начиная с 1.1. Узнаваемость бренда Apache и прекрасный набор функций Log4J принесли ему огромную популярность среди разработчиков Java; его можно считать стандартом де-факто журналирования Java.

Второй каркас - Java Logging API компании Sun, ставший официальным стандартом. Хорошо это или плохо, JCP решила не использовать Log4J, а создать новый стандарт. Java Logging API также мощен и гибок, хотя в нём отсутствуют некоторые развитые функции Log4J, в том числе:

• запись в системные журналы, например журнал событий Windows NT и Unix Syslog;
• ротация файлов журнала по дате;
• шаблоны форматирования в стиле функции языка C `printf`;
• повторная загрузка файла конфигурации.

Повторная загрузка файла конфигурации - важная возможность для высокодоступных приложений. Обычно конфигурация читается при инициализации каркаса, то есть, как правило, во время запуска приложения. Изменения уровней и категорий журналирования не подхватываются до перезапуска приложения. При устранении неполадок в работающей промышленной системе частые перезапуски исключены, поэтому без повторной загрузки уровень журналирования приходится выбирать до запуска приложения.

Для взлома и обратной разработки Log4J явно предпочтительнее, поскольку работает со всеми версиями JDK. Загрузка и документация в Интернете доступны по адресу jakarta.apache.org.

Что следует и чего не следует делать при трассировке

Трассировка чрезвычайно важна для крупных распределённых приложений, но при неосторожном использовании способна заметно снизить производительность. Ниже приведены простые правила эффективной трассировки.

Что следует делать

• Используйте трассировку как можно шире.

• Убедитесь, что каждая запись содержит время, а также имя класса и метода, создавших её.

• Записывайте трассировку до и после критических участков кода, например вызовов внешних систем, обращений к базе данных и диску. Это позволяет легко определить точку сбоя.

• Включайте значения параметров и контекстных переменных, помогающие понять контекст выполнения в момент записи. Например, идентификатор пользователя в каждой записи позволяет отфильтровать журнал по операциям, выполненным от имени конкретного пользователя.

• При первом перехвате исключения записывайте его вместе с трассировкой стека.

• Обязательно используйте разные уровни трассировки. Критический уровень предназначен для системных сообщений и ошибок, а уровень отладки - для многочисленных сообщений, не слишком важных для устранения неполадок. Это позволяет управлять размером журнала и накладными расходами приложения.

Чего не следует делать

• Не используйте `System.out.println` для записи трассировки. Это создаёт постоянные накладные расходы и лишает гибкости. Вместо него применяйте каркас журналирования.

• Не выводите исключение с помощью `Exception.printStackTrace()`, поскольку этот метод всегда пишет в `System.out`. Используйте методы каркаса журналирования.

• Не вставляйте трассировку в циклы с сотнями или тысячами повторений.

• Не вставляйте трассировку в часто вызываемые небольшие методы.

Ещё одна нежелательная практика - соединять строки оператором + при передаче параметров каркасу журналирования. Даже если уровень трассировки повышен, среда выполнения всё равно выполнит дорогостоящую операцию: выделит новый буфер для результирующей строки и скопирует в него строки-аргументы. Поэтому вместо:

```
logger.debug("Message received from host: " + host + ", text: " + text)
```

используйте:

```
if (logger.isDebugEnabled() == true)
    logger.debug("Message received from host: " + host + ", text: " + text)
```

Выполнение второго фрагмента обходится значительно дешевле, поскольку строки соединяются только при включённой отладочной трассировке.

Короткий тест

1. Чем трассировка отличается от отладки?

2. Как использовать трассировку для взлома приложения?

3. Почему возможность повторной загрузки конфигурации журналирования во время выполнения так важна?

4. Чем опасно использование оператора + для соединения строк?

Краткие итоги

• Трассировка - это запись отладочных сообщений в поток вывода во время выполнения приложения.

• Трассировка вставляется в код приложения в виде вызовов API.

• Трассировка создаёт накладные расходы, но они оправданы упрощением устранения неполадок.

• Добавление трассировки помогает понять поток выполнения приложения.

• И Log4J, и Java Logging API являются хорошими каркасами журналирования. Log4J более развит и может использоваться с версиями JDK до 1.1, поэтому лучше подходит для взлома.

Глава 7. Управление безопасностью Java

> «Эксперт - это человек, который знает всё больше о всё меньшем, пока наконец не знает абсолютно всё ни о чём».
>
> *Законы Мерфи о технике*

В этой главе

• Обзор безопасности Java
• Обход проверок безопасности
• Короткий тест
• Краткие итоги

Обзор безопасности Java

Java с самого начала проектировалась как безопасный язык. Безопасность включает такие возможности языка, как проверка границ индексов массивов, проверка байт-кода и управляемый доступ к критически важным системным ресурсам, например файлам и сведениям о пользователе. Первоначально модель безопасности предполагала, что всем локально установленным классам следует предоставлять доступ к системным ресурсам, а весь байт-код, загруженный по сети, необходимо ограничивать в доступе к конфиденциальной информации и операциям. По мере развития Java граница между локальными и удалёнными классами несколько размылась. Серверы приложений и веб-серверы полагаются на модель безопасности, чтобы один компонент, например веб-приложение, не нарушал работу другого. Подписанные апплеты позволяют загруженному коду обращаться к локальным файлам, буферу обмена и системным свойствам.

Различные основные классы во время выполнения используют каркас безопасности для проверки того, разрешено ли вызывающей стороне выполнить запрошенную операцию. В центре модели безопасности находится экземпляр `java.lang.SecurityManager`, служащий фасадом для пакета `java.security`. В Java доступ к системным сведениям или ресурсу представляется разрешением. Например, `PropertyPermission` представляет доступ к системным свойствам. Перед выполнением логики метода основные классы обращаются к диспетчеру безопасности, чтобы проверить наличие у вызывающей стороны нужных разрешений. Работу механизма проще всего понять на примере реализации метода `System.setProperty()`, приведённой в листинге 7.1.

```
public static String setProperty(String key, String value) {
    if (key == null) {
        throw new NullPointerException("key can't be null");
    }
    if (key.equals("")) {
        throw new IllegalArgumentException("key can't be empty");
    }
    if (security != null)
        security.checkPermission(new PropertyPermission(key, "write"));
    return (String) props.setProperty(key, value);
}
```

Листинг 7.1. `System.setProperty`

Сначала реализация получает системный диспетчер безопасности. Если он установлен и параметры корректны, диспетчер проверяет, предоставлено ли разрешение `PropertyPermission` на запись в указанный ключ. Метод `checkPermission` диспетчера безопасности создаёт исключение `SecurityException`, если разрешение отсутствует. Новое значение системного свойства задаётся только после успешного возврата из `checkPermission`. В таблице 7.1 перечислены операции, контролируемые диспетчером безопасности.

Таблица 7.1. Операции, защищаемые диспетчером безопасности

Тип	**Защищаемые операции**
Система	Доступ к пакетам. Доступ к переменным-членам и методам класса. Загрузка нативных библиотек. Завершение работы системы. Запуск и остановка потоков. Создание собственных загрузчиков классов.
Ввод, вывод, сеть	Создание, удаление, чтение и запись файлов. Обход каталогов. Создание сокетов. Загрузка классов с сетевого URL.
AWT/Swing	Доступ к буферу обмена. Доступ к системной очереди событий. Отображение окон без предупреждения. Получение задания печати.

Сведения о том, какие разрешения предоставлены каким классам, хранятся в файлах политики Java. Сначала из каталога `${java.home}/lib/security`, где `${java.home}` - каталог установки JRE, загружается общесистемный файл `java.policy`. Затем, если он существует, загружается `${user.home}/.java.policy`. Собственный файл политики Java можно задать в командной строке параметром `-Djava.security.policy`. Подробное описание безопасности и разрешений Java находится по адресу java.sun.com.

Безопасность Java включает API шифрования (JCE), аутентификации и авторизации (JAAS), поддержки защищённых сокетов (JSSE) и многие другие. Эти дополнительные API предоставляют мощные средства защиты информации, но в обычном приложении напрямую не используются. Мы сосредоточимся на возможностях безопасности, с которыми разработчик приложения сталкивается чаще всего.

Обход проверок безопасности

С точки зрения этой книги нас прежде всего интересует обход проверок безопасности, выполняемых средой Java. Строгость модели безопасности определяется тремя сценариями конфигурации:

• диспетчер безопасности не установлен;
• диспетчер безопасности установлен со стандартной политикой;
• диспетчер безопасности установлен с собственной политикой.

> **Истории с передовой.** Мы планировали воспользоваться сторонней технологией развёртывания, чтобы установить приложение Swing на тысячи пользовательских компьютеров. Выбранный продукт поддерживал автоматические обновления, для чего приложение требовалось запускать через его средство запуска, а не как самостоятельную программу. Для автоматического обновления по HTTP продукт устанавливал собственный обработчик HTTP, конфликтовавший с обработчиком HTTP с поддержкой SSL, который использовало наше приложение. Java не поддерживает динамическое переключение обработчиков протоколов, поскольку хранит и повторно использует экземпляры реализаций из закрытого кэша. Единственным способом обойти проблему была принудительная очистка кэша при запуске приложения. Мы применили описанный в главе 4 «Взлом непубличных методов и переменных класса» приём доступа к закрытым членам, для которого необходимы диспетчер безопасности и собственный файл политики. Вместе с приложением мы поставили файл политики, разрешающий неограниченный доступ к членам классов через API рефлексии, и после очистки кэша при запуске добились совместной работы продукта с нашим кодом.

Диспетчер безопасности не установлен

Если диспетчер безопасности не установлен, проверки просто не выполняются. Как видно из листинга 7.1, разрешения проверяются только при ненулевом экземпляре системного диспетчера. По умолчанию приложения Java, но не апплеты и приложения Web Start, работают без диспетчера безопасности, поэтому код имеет практически неограниченный доступ к системным сведениям и ресурсам. Например, без диспетчера любой класс может прочитать имя учётной записи пользователя из системного свойства `user.name`. Поэтому для приложений, требующих защищённой среды, отсутствие диспетчера безопасности не рекомендуется.

Диспетчер безопасности установлен со стандартной политикой

Установка диспетчера безопасности включает проверку разрешений на доступ к сведениям и ресурсам. Это стандартная модель для апплетов и приложений Web Start, работающих в песочнице. Диспетчер можно установить параметром командной строки `java -Djava.security.manager` или вызовом `System.setSecurityManager()`. Если собственный файл политики не задан, загружается и применяется стандартный. Стандартная политика разрешает лишь небольшое число действий, создавая сравнительно защищённую среду выполнения. Ниже приведена сокращённая версия стандартного файла политики с предоставляемыми разрешениями:

```
grant codeBase "file:${java.home}/lib/ext/*" {
    permission java.security.AllPermission;
};

grant {
    // Разрешает любому потоку остановить себя методом
    // java.lang.Thread.stop(), не принимающим аргументов.
    permission java.lang.RuntimePermission "stopThread";

    // Разрешает всем прослушивать непривилегированные порты.
    permission java.net.SocketPermission "localhost:1024-", "listen";

    // «Стандартные» свойства, доступные всем для чтения.
    permission java.util.PropertyPermission "java.version", "read";
    ...
}
```

Если разрешение не предоставлено явно, действие запрещено. Например, доступ к закрытому методу через рефлексию требует `RuntimePermission` типа `accessDeclaredMembers` и `ReflectPermission` типа `suppressAccessChecks`. При запуске теста из главы 4 с установленным диспетчером безопасности приложение завершается с исключением безопасности.

Разрешения можно предоставить коду несколькими способами. Проще всего создать собственный файл политики безопасности Java со всеми необходимыми разрешениями и указать его в командной строке свойством `-Djava.security.policy`. Этот подход был показан в главе 4.

Диспетчер безопасности установлен с собственной политикой

А что, если диспетчер безопасности установлен, а собственный файл политики уже задан параметром `-D`? Так часто бывает с серверами приложений и веб-серверами, которые полагаются на безопасность Java для изоляции приложений J2EE. Дополнительные разрешения можно предоставить ещё двумя способами. Разумеется, можно просто отредактировать собственный файл политики продукта, выполняющего ваш код, или системный файл политики, однако это неаккуратное решение.

Лучше назвать свой файл политики `.java.policy` и поместить его в домашний каталог пользователя. Среда выполнения Java сначала загружает системный файл политики из `${java.home}`, а затем пользовательский из `${user.home}`. Изменять существующие сценарии или файлы не понадобится, благодаря чему изменения остаются изолированными и легко сопровождаются.

Ещё одна возможность - добавить ссылку на свой файл политики в файл `java.security`, находящийся в подкаталоге `${java.home}/lib/security`. В этом файле описана конфигурация безопасности и указано, какие файлы политики и в каком порядке загружать. Там уже есть ссылки на системный и пользовательский файлы; чтобы подключить свой, добавьте следующую строку:

```
policy.url.3=file:/C:/Projects/CovertJava/distrib/conf/java.policy
```

Обязательно используйте путь, соответствующий расположению файла на вашем компьютере. Число `3` - следующий свободный индекс в списке URL файлов политики.

Короткий тест

1. Какой класс служит фасадом для каркаса безопасности Java?
2. Какую роль играют разрешения?
3. Какие файлы и в каком порядке загружаются для определения предоставляемых разрешений?

Краткие итоги

• Безопасность Java управляет доступом к системным сведениям и ресурсам посредством разрешений.

• Диспетчер безопасности находится в центре модели безопасности. Если он не установлен, проверки не выполняются.

• Разрешения можно предоставлять в файлах политики Java.

• Файлы политики можно указывать в командной строке или через файл конфигурации безопасности Java.

Глава 8. Исследование среды выполнения

> «Логика - это систематический метод уверенно приходить к неправильному выводу».
>
> *Законы Мерфи о технике*

В этой главе

- Ценность понимания среды выполнения
- Системные свойства
- Сведения о системе
- Сведения о памяти
- Сведения о сети
- Доступ к переменным среды
- Короткий тест
- Краткие итоги

Ценность понимания среды выполнения

Взлом систем и устранение неполадок в приложениях часто требуют изменения различных системных параметров. Например, чтобы установить исправленную версию класса в главе 5 «Замена и исправление классов приложения», мы изменили значение переменной `CLASSPATH`. В главе 7 «Управление безопасностью Java» было показано, как установить диспетчер безопасности посредством системного свойства. Понимание среды выполнения и знание точных значений системных параметров позволяет опираться в работе на факты, а не на предположения. В этой главе вы узнаете, как получать значения системных свойств, проверять наличие диспетчера безопасности и извлекать различные сведения о памяти и сети.

Большая часть полезных сведений легко доступна через классы API Java, если знать, где искать. Из-за кроссплатформенной природы Java сведения о базовом оборудовании и физических ресурсах ограничены. В этой главе мы создадим класс исследования, который собирает и выводит все полезные сведения о среде. Его можно вызвать из работающего приложения или запустить отдельно, указав `covertjava.snoop.RuntimeInfo` как главный класс либо воспользовавшись `snoop_re.bat`. Например, серверы приложений обычно запускаются через последовательность пакетных файлов или сценариев оболочки, настраивающих переменные среды и параметры приложения. Код запуска сервера может изменить стандартное состояние среды, и единственный надёжный способ узнать текущие значения - вывести их посредством трассировки.

Системные свойства

Системные свойства - пары «имя-значение», содержащие сведения о среде выполнения. В них входят данные об операционной системе, поставщике и версии JVM, пути загрузки классов и библиотек, домашний и текущий каталоги пользователя и другие полезные свойства. Значение каждого свойства можно найти в Интернете, и я рекомендую с ними познакомиться. Хорошая исходная точка - JavaDoc класса `java.lang.System` по адресу java.sun.com. В листинге 8.1 показано, как вывести все системные свойства.

```
public void printSystemProperties(PrintStream stream) {
    StringBuffer buffer = new StringBuffer(1000);
    Properties props = System.getProperties();

    for (Enumeration keys = props.keys(); keys.hasMoreElements();;) {
        String key = (String)keys.nextElement();
        buffer.append(key);
        buffer.append("=");
        buffer.append(System.getProperty(key));
        buffer.append('\n');
    }
    stream.print(buffer.toString());
}
```

Листинг 8.1. Исходный код `printSystemProperties`

Выполнение этого метода даёт примерно следующий результат:

```
java.vm.version=1.4.1-b21
java.vm.vendor=Sun Microsystems Inc.
java.vendor.url=http://java.sun.com/
java.vm.name=Java HotSpot(TM) Client VM
...
```

Системные свойства можно изменять методом `System.setProperty()`. Если диспетчер безопасности установлен, он проверяет наличие у вызывающей стороны разрешения на запись в нужное свойство. Некоторые свойства читаются только при инициализации и потому фактически ведут себя как доступные только для чтения. Например, свойство `java.class.path` описывает путь загрузки классов. Хотя его значение можно изменить, желаемого результата это не даст, потому что системный загрузчик классов читает его лишь один раз и последующего изменения не видит. Управление безопасностью Java подробно рассматривалось в главе 7.

Сведения о системе

Системные свойства очень хорошо охватывают сведения о среде. Однако внимания заслуживают ещё несколько ключевых аспектов системы. Мы хотим выяснить, задан ли диспетчер безопасности, чтобы оценить защищённость среды. Также важно знать, какой загрузчик использовался для загрузки класса, чтобы понять, насколько свободно можно управлять загрузкой. Наконец, из соображений лицензирования и производительности нужно узнать число доступных процессоров. В листинге 8.2 приведён исходный код, получающий эти сведения.

```
public void printSystemInfo(PrintStream stream) {
    StringBuffer buffer = new StringBuffer(200);
    buffer.append("Security manager: ");
    buffer.append(System.getSecurityManager() == null ?
        "null" : System.getSecurityManager().getClass().getName());
    buffer.append('\n');

    buffer.append("Class loader for this class: ");
    ClassLoader classLoader = this.getClass().getClassLoader();
    buffer.append(classLoader == null ?
        "null" : classLoader.getClass().getName());
    buffer.append('\n');
    buffer.append("Number of available processors to JVM: ");
    buffer.append(Runtime.getRuntime().availableProcessors());
    buffer.append('\n');
    stream.println(buffer.toString());
}
```

Листинг 8.2. Исходный код `printSystemInfo`

Сведения о памяти

JVM управляет памятью, доступной приложению Java. Когда приложение запрашивает дополнительную память, JVM сначала проверяет, можно ли удовлетворить запрос за счёт свободной памяти, уже выделенной процессу. Если свободной памяти недостаточно, JVM может принудительно запустить сборку мусора, пытаясь освободить память, занятую мёртвыми объектами. Если сборка не освобождает достаточно памяти, JVM пытается получить её у операционной системы, увеличивая общий выделенный объём. Она продолжает получать память от ОС до разрешённого максимума, который по умолчанию равен 16 Мбайт, но может быть задан параметром `-Xmx`.

Управление памятью имеет решающее значение для стабильности и производительности приложений Java. Подробности сборки мусора и эффективного использования памяти выходят за рамки книги, но вы узнаете, как получать размеры различных пулов памяти. Лучше всего наблюдать за использованием памяти приложения с помощью профилировщика, например JProble или OptimizeIt. Однако для них приложение требуется запустить в режиме отладки, что неприемлемо в промышленных системах. Инструменты нового поколения, например Wily Introscope и Borland Server Trace, способны с минимальными накладными расходами наблюдать за памятью любого работающего приложения и отображать её использование. Но они опираются на тот же метод, который применим и мы, и могут оказаться слишком дорогими для многих приложений Java. В листинге 8.3 показана реализация метода, выводящего текущий профиль памяти JVM.

```
public void printMemoryInfo(PrintStream stream) {
    StringBuffer buffer = new StringBuffer(200);
    buffer.append("Maximum memory allowed for JVM: ");
    buffer.append(toMb(Runtime.getRuntime().maxMemory()));
    buffer.append(" Mb\n");
    buffer.append("Memory currently allocated in JVM: ");
    buffer.append(toMb(Runtime.getRuntime().totalMemory()));
    buffer.append(" Mb\n");
    buffer.append("Free memory in JVM: ");
    buffer.append(toMb(Runtime.getRuntime().freeMemory()));
    buffer.append(" Mb\n");
    stream.println(buffer.toString());
}
```

Листинг 8.3. Исходный код `printMemoryInfo`

Вспомогательная функция `toMb` преобразует переданное значение из байтов в мегабайты с точностью до двух знаков. Она находится в классе `RuntimeInfo`.

Сведения о сети

Приложение Java способно узнать о сетевой среде не так уж много. Java - кроссплатформенный язык высокого уровня, поэтому плохо подходит для реализации низкоуровневых служебных программ ОС и драйверов. Практически единственные полезные сведения, доступные из среды выполнения, - имя и IP-адрес локального узла. Их можно использовать при лицензировании, когда лицензии выдаются на отдельный сервер и привязываются к узлу. Дополнительные сведения о базовой конфигурации сети и оборудования можно получить через JNI и библиотеки C, выполняющие нативные вызовы ОС. В листинге 8.4 показано получение сведений об узле.

```
public void printNetworkInfo(PrintStream stream) {
    StringBuffer buffer = new StringBuffer(200);
    InetAddress localhost = null;

    try {
        localhost = java.net.InetAddress.getLocalHost();
        buffer.append("Local host name: ");
        buffer.append(localhost.getHostName());
        buffer.append('\n');
        buffer.append("Local host IP address: ");
        buffer.append(localhost.getHostAddress());
    }
    catch (UnknownHostException ex) {
        buffer.append("*** Failed to detect network properties " +
            "due to UnknownHostException: ");
        buffer.append(ex.getMessage());
    }
    stream.println(buffer.toString());
}
```

Листинг 8.4. Исходный код `printNetworkInfo`

Доступ к переменным среды

Переменные среды - удобный способ параметризации приложения. Например, каталог установки приложения может различаться на разных узлах, а надёжного способа определить место установки в Java нет. Поэтому часто определяют переменную среды APP_HOME и передают её приложению Java.

Не следует использовать слишком много переменных среды, поскольку это усложняет программирование пакетных файлов и сценариев оболочки. Лучше хранить настройки в конфигурационных файлах формата свойств Java или XML, а переменной среды задавать расположение этих файлов. Правильный способ получить переменные среды в Java - передать их в командной строке ключом -D. Для ясности я предпочитаю добавлять ко всем их именам префикс env. Следующий пример показывает, как передать значения переменных среды Windows TEMP и COMPUTERNAME создаваемой в этой главе служебной программе исследования:

```
set JAVA_ARGS=%JAVA_ARGS% -Denv.temp=%TEMP%
set JAVA_ARGS=%JAVA_ARGS% -Denv.computername=%COMPUTERNAME%
java %JAVA_ARGS% covertjava.snoop.RuntimeInfo
```

Затем приложение Java может получить значение переменной методом System.getProperty(), используя имя, заданное в командной строке. В нашем примере значение TEMP можно получить вызовом System.getProperty("env.temp").

Короткий тест

1. В каких случаях следует исследовать среду выполнения?

2. Какие сведения о среде выполнения доступны через стандартный API Java?

3. Как получить недоступные через стандартные API сведения, зависящие от ОС или оборудования?

Краткие итоги

• Получение снимка значений среды - надёжный способ понять конфигурацию работающего приложения.

• Системные свойства, доступные через метод System.getProperty(), предоставляют большую часть сведений о среде выполнения.

• Снимок объёмов памяти помогает оценить эффективность управления памятью приложения в промышленной среде.

• Сетевые сведения ограничиваются именем и IP-адресом локального узла.

• Значения переменных среды можно передавать приложению Java параметром командной строки -D.

Глава 9. Взлом кода с помощью нестандартных отладчиков

> «Любая простая теория будет сформулирована самым сложным образом».
>
> *Законы Мерфи о технике*

В этой главе

• Понимание внутреннего устройства неизвестных приложений
• Обычные отладчики и их ограничения
• Взлом с помощью всеведущего отладчика
• Короткий тест
• Краткие итоги

Понимание внутреннего устройства неизвестных приложений

Писать главу об использовании обычного отладчика - значит почти оскорблять интеллект читателя. Если вы не способны самостоятельно разобраться с отладчиком, вероятно, стоит подумать о смене профессии. Поэтому эта глава посвящена нестандартному инструменту, который предлагает иной взгляд внутрь работающих приложений. Хотя я говорю об отладке, на самом деле вы научитесь разбираться во внутреннем устройстве неизвестных приложений. Обычно предпочтительнее работать с исходным кодом, однако в некоторых случаях отладчик незаменим, в частности когда:

• вы работаете с большим приложением, в котором нет хорошей трассировки;
• чтение исходного кода не даёт ясного понимания внутренней логики из-за сложной иерархии объектов и неуклюжих приёмов программирования;
• вы работаете с приложением, подвергнутым агрессивной обфускации.

Обычные отладчики и их ограничения

Java с самого начала проектировалась с учётом отладки. Стандартный Java Debug API опирается на представление об удалённом наблюдателе за отлаживаемым процессом. Этот стандарт позволяет поставщикам создавать собственные отладчики с самым разным набором функций. Некоторые почти не выходят за рамки установки точек останова и пошагового выполнения кода приложения. Более развитые отладчики предоставляют дополнительные возможности, например условные точки останова, приостанавливающие выполнение при определённом значении переменной или создании определённого исключения.

Обычные отладчики помогают писать код и исправлять ошибки, но малоэффективны при обратной разработке или устранении неполадок в сложном приложении. Их главная проблема заключается в том, что сведения показываются только для текущего момента, а как только этот момент проходит, безвозвратно теряются. Для эффективной работы обычного отладчика по всему коду требуется стратегически расставить точки останова, что, разумеется, утомительно, особенно если код плохо знаком.

Предположим, вы получили приложение Chat и хотите узнать, как оно обрабатывает входящие сообщения. Его можно загрузить и запустить в режиме отладки, но откуда знать, где поставить точку останова? Даже в небольшом приложении вроде Chat пошаговое выполнение не поможет, потому что сообщения доставляются асинхронно в потоке RMI. Только декомпиляция и внимательное изучение кода способны привести к методу `receiveMessage`, который определён в `ChatServerRemote` и реализован в `ChatServer`. Если попытаться выполнить обратную разработку версии Chat, обфусцированной с помощью Zelix KlassMaster, задача станет, мягко говоря, трудной. Нужен более подходящий инструмент. На сцену выходит Omniscient Debugger.

Взлом с помощью всеведущего отладчика

Всеведущая отладка позволяет записать состояние выполняющейся программы, а затем вернуться назад во времени и изучить прежние состояния. Она отличается от обычной тем, что пользовательское приложение запускается не в режиме отладки и потому не может быть приостановлено. Идея всеведущего отладчика состоит в записи максимально возможного объёма сведений о потоках и переменных, их значениях, стандартных потоках ввода и вывода и загруженных классах. После записи приложение можно остановить или закрыть. Затем отладчик позволяет проследить выполнение с начала, с конца либо с определённого момента, установленного по вызову метода или сообщению в стандартном потоке вывода. Благодаря этому точки останова не нужны, а асинхронно обработанные операции точно не будут пропущены.

В настоящее время единственная известная мне реализация этого подхода - Omniscient Debugger (ODB) Била Лямбды. Он распространяется по лицензии GPL и доступен по адресу lambdacs.com. Программа ещё требует существенной доработки, особенно в части пользовательского интерфейса, но с записью и исследованием справляется поразительно хорошо. ODB применяет метод декорирования байт-кода, вставляя код, который записывает необходимые для отладки сведения. Мы представим, что ничего не знаем о внутренней реализации Chat, и с помощью Omniscient Debugger Била выясним, как Chat обрабатывает входящие сообщения.

Запись выполнения Chat

Начнём с записи выполнения Chat в ODB. Дистрибутив Covert Java содержит последнюю на момент написания книги версию ODB. Она находится в каталоге `lib` и используется по умолчанию. Можно проверить наличие более новой версии и воспользоваться ею. Запустите `debugChat.bat` из каталога `distrib\bin`; этот пакетный файл настраивает среду ODB и предписывает при запуске выполнить `covertjava.chat.ChatApplication`. После загрузки ODB открывает главное окно и запускает Chat. Когда программа запросит расположение исходного кода `ChatApplicaton`, нажмите No Source Available, чтобы ограничиться байт-кодом. Окно отладчика после инициализации показано на рисунке 9.1.

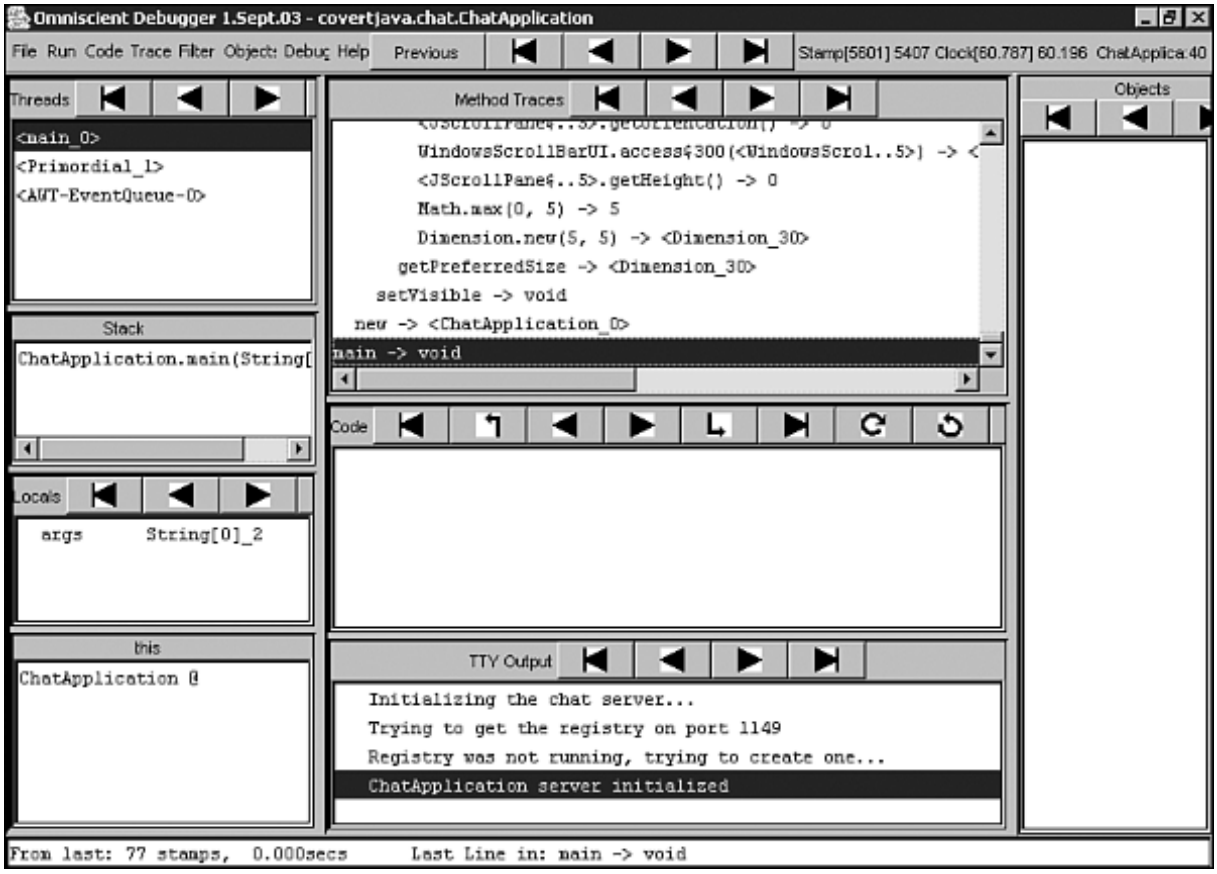

Рисунок 9.1. Окно ODB после инициализации

Пользовательский интерфейс ODB не самый интуитивный, но после привыкания в нём можно эффективно перемещаться. Слева от меню в самом верху находится панель Stamp. ODB записывает выполнение программы как последовательность временных меток, а эта панель позволяет двигаться по ней назад и вперёд. Сведения о состоянии приложения отображаются на нескольких панелях:

• **Threads.** Показывает все записанные потоки. Если перед именем потока стоит `--`, в текущий момент времени он ещё не создан. Завершившийся поток помечается как Dead.
• **Stack.** Показывает стек вызовов выбранного потока.
• **Locals.** Показывает имена и значения локальных переменных выбранного метода на панели стека вызовов или Method Traces.
• **This.** Показывает результат вызова `toString()` для объекта, чей метод выбран.
• **Method Traces.** Показывает последовательность вызовов методов, включая вложенные вызовы с отступами.
• **Code.** Показывает исходный код, если он доступен.

• **TTY Output.** Показывает перехваченные вызовы `System.out.println` (в настоящее время другие методы вывода ODB не поддерживает).
• **Objects.** Показывает строковые представления наблюдаемых объектов.

Попытаемся осмыслить сведения на рисунке 9.1. Видно, что после инициализации в Chat есть три потока. Один из них, `<main_0>`, сейчас выбран и не имеет стека вызовов, поскольку фактически уже завершился; однако методы, выполнявшиеся в нём, видны на панели Method Traces. Прокручивая эту панель, мы получаем полный журнал имён методов и даже значений параметров. Щелчок по методу на Method Traces переносит к моменту его выполнения и обновляет остальные панели соответствующими сведениями. Аналогично, щелчок по сообщению на TTY Output позволяет увидеть, кто и когда его вывел. Чтобы проследить выполнение программы с начала, нажмите кнопку First Timestamp of Any Thread на панели Stamp, похожую на кнопку перемотки видеомагнитофона, а затем пошагово двигайтесь кнопкой Next Timestamp, похожей на Play.

Навигация по коду обработки сообщений

Теперь мы готовы взломать код обработки сообщений приложения Chat. Сначала необходимо обойти ошибку ODB, мешающую перехватывать сведения о динамически загружаемых классах. Нажмите Alt+Tab, чтобы перейти к окну Debug Controller с кнопкой Stop Recording и несколькими флажками параметров записи. Нажмите Stop Recording; когда надпись изменится на Start Recording, нажмите кнопку ещё раз. Затем нажмите Alt+Tab, чтобы открыть окно Chat. Введите `localhost` в поле Host Name и `Hello` в поле текста сообщения. Нажмите Enter и убедитесь, что сообщение появилось в области разговора Chat. Поскольку Chat был запущен из отладчика, ввод и реакция приложения записаны, и теперь можно вернуться к окну отладчика для изучения трассировок. Нажмите Last Timestamp Recorded (Any Thread) на панели Stamp, чтобы перемотать отображение к концу записанной последовательности. На панели TTY Output должно появиться больше строк. Щёлкните строку `Received message from host...`. Окно отладчика должно выглядеть примерно так, как на рисунке 9.2.

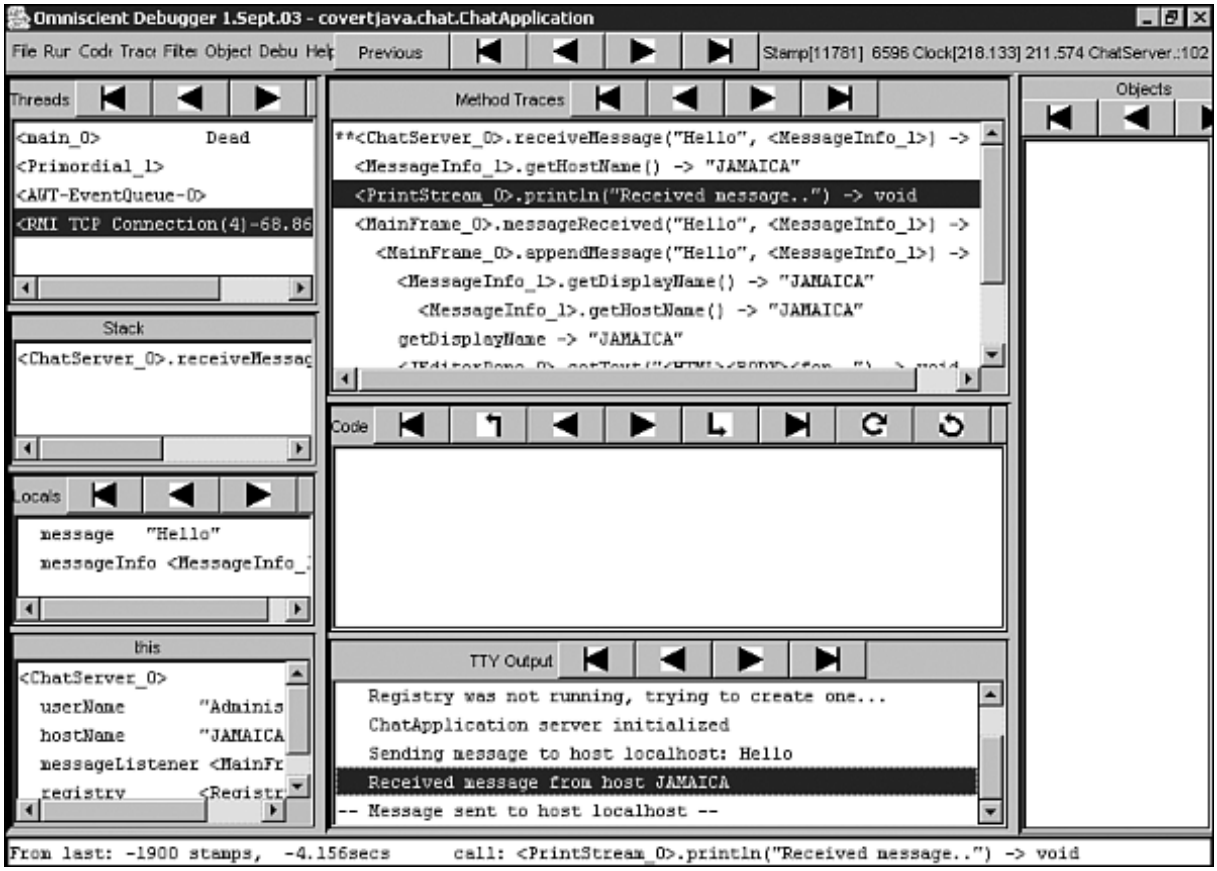

Рисунок 9.2. Окно ODB после отправки сообщения

По сведениям на панелях можно заключить, что сообщение вывел метод `ChatServer.receiveMessage`, выполнявшийся в потоке RMI TCP Connection. Это логично, поскольку, как нам известно, Chat доставляет сообщения через RMI. Прокручивая Method Traces, можно точно увидеть, что выполнялось до и после вызова `System.out.println`. Видно, что текст сообщения извлекается из класса `MessageInfo` и добавляется в `JEditorPane` внутри `MainFrame`. Даже без исходного кода мы достаточно хорошо понимаем логику обработки сообщений Chat.

Задачу упростило сообщение трассировки `Received message from host...`, которое непосредственно привело к методу обработки. А если бы такой подсказки не было? К тому же результату ведёт несколько подходов. После записи действий можно сначала посмотреть на Threads. Знание архитектуры Java поможет решить, какие потоки исследовать. Например, если знать, что удалённый обмен приложений Java, вероятно, использует RMI, а RMI обрабатывает входящие вызовы в собственном потоке, можно искать соответствующие потоки. Неудивительно, что после перемотки отладчика к концу отправки сообщения виден новый поток `RMI TCP Connection...`. Щелчок по нему на Threads показывает выполнявшиеся методы, и, вуаля, в вершине трассировки снова находится `ChatServer.receiveMessage`.

Другой хороший способ найти логику обработки сообщений - функция Search в ODB. Она не слишком развита, но позволяет искать тестовую строку на панели трассировки. В нашем случае известно, что текст сообщения был `Hello`. Если непонятно, в каком потоке выполнялась обработка, можно включить Any Thread OK в меню Code отладчика. Затем выберите Search в меню Trace и введите `Hello`. После нажатия Enter отладчик найдёт и выберет содержащую строку трассировку, которой в нашем случае, как вы уже догадались, будет `ChatServer.receiveMessage`.

После нахождения класса и метода логику реализации можно изучать по исходному или декомпилированному коду. Теперь приложение можно запустить в режиме отладки под обычным отладчиком и наблюдать «живые» данные, поскольку мы знаем, где поставить точки останова.

Использование ODB для взлома обфусцированной версии Chat

В главах 3 «Обфускация классов» и 4 «Взлом непубличных методов и переменных класса» обсуждались обфускация и трудности взлома приложений, подвергнутых агрессивной обфускации. Декомпиляция обфусцированного кода утомительна, а понять внутреннее устройство большого приложения может оказаться невозможно. Такой отладчик, как ODB, - лучший инструмент для работы с подобными приложениями: вместо попыток вообразить последовательность вызовов можно просто изучить её запись. Для обфусцированного кода применимы те же способы поиска исходной точки, что и для обычных классов. Чтобы это показать, повторим исследование обработки сообщений Chat, но теперь с обфусцированной версией.

Изменим `CLASSPATH` внутри `debugChat.bat`, заменив `lib\chat.jar` на `lib\obfuscated\chat.jar`. Затем запустим отладчик и выполним те же шаги записи, что были описаны выше. Отправив сообщение, перемотаем отладчик к концу записи с помощью панели Stamp. Сравнение панелей со сведениями для необфусцированного Chat показывает, что принципиально они одинаковы. Хотя имена классов и методов Chat изменились, стеки вызовов и значения параметров остались прежними. Не изменились и системные сведения, например основные классы Java и имена потоков. В листинге 9.1 показано содержимое Method Traces для обфусцированного Chat.

```
**<d_0>.receiveMessage("Hello", <MessageInfo_1>) -> void
<MessageInfo_1>.a() -> "JAMAICA"
<PrintStream_0>.println("Received message..") -> void
<f_0>.a("Hello", <MessageInfo_1>) -> void
<f_0>.b("Hello", <MessageInfo_1>) -> void
<MessageInfo_1>.b() -> "JAMAICA"
<MessageInfo_1>.a() -> "JAMAICA"
b -> "JAMAICA"
<JEditorPane_0>.setText("<HTML><BODY><fon..") -> void
b -> void
a -> void
receiveMessage -> void
```

Листинг 9.1. Содержимое панели Method Traces

Мы по-прежнему можем использовать отладчик, чтобы выяснить, какой метод вывел сообщение `Received message from host...` или какая строка трассировки метода содержит `Hello`. Это даёт большое преимущество при поиске исходной точки для изучения приложения.

Короткий тест

1. Зачем использовать отладчик при взломе?

2. Каковы ограничения обычных отладчиков?

3. Какой принцип лежит в основе реализации ODB?

4. Какие подходы позволяют находить логику реализации в ODB?

Краткие итоги

• Отладчик может сократить путь к логике реализации в большом приложении или обфусцированном коде.

• Главные ограничения обычных отладчиков - необходимость устанавливать точки останова в ключевых местах и отсутствие истории изменений состояния и вызовов методов.

• ODB записывает временные метки выполняющегося приложения и позволяет перемещаться по ним, чтобы понять внутреннюю логику.

• ODB - лучший подход к взлому обфусцированных приложений.

Глава 10. Использование профилировщиков для анализа приложения во время выполнения

> «Ни одну ошибку нельзя правильно исправить после 16:30 в пятницу. Правильное решение станет самоочевидным в 9:15 утра в понедельник».
>
> *Законы Мерфи о технике*

В этой главе

• Почему и когда следует применять профилирование
• Лучшие профилировщики Java
• Исследование использования кучи и частоты сборки мусора для повышения производительности
• Просмотр размещения объектов и ссылок для поиска и устранения утечек памяти
• Исследование распределения и синхронизации потоков
• Поиск дорогостоящих методов для повышения производительности
• Исследование приложения во время выполнения с помощью дампа потоков
• Короткий тест
• Краткие итоги

Почему и когда следует применять профилирование

Термином «профилирование» традиционно называют измерение времени выполнения методов для поиска и устранения узких мест производительности. Однако в мире Java это понятие расширилось и теперь охватывает сбор различных показателей, а также отладку потоков и объектов во время выполнения. Вот несколько причин использовать профилировщик для приложения Java:

• исследовать использование кучи и частоту сборки мусора с целью повысить производительность;
• просматривать размещение объектов и ссылки для поиска и устранения утечек памяти;
• исследовать распределение и синхронизацию потоков, чтобы находить проблемы блокировок и гонок данных и повышать производительность;
• выявлять дорогостоящие методы для повышения производительности;
• исследовать приложение во время выполнения, чтобы лучше понять его внутреннюю структуру.

Профилирование обычно выполняют после этапа разработки, и если вы прежде не пользовались профилировщиком, результаты могут открыть вам глаза. Три главные высокоуровневые цели профилирования - повысить производительность приложения, исправить ошибки, которые трудно найти в коде, и понять, что происходит в приложении во время выполнения бизнес-логики.

Лучшие профилировщики Java

Для эффективной работы нужны правильные инструменты. Три самых известных профилировщика Java - JProbe Suite компании Quest Software, первоначально разработанный KLGroup, OptimizeIt Suite компании Borland и JProfiler компании ej-technologies. Все три хороши и почти одинаково успешно справляются с задачей. Мне нравится, что JProfiler объединяет все функции в одном приложении, а не разделяет профилирование, отладку памяти и потоков между отдельными инструментами. JProbe, вероятно, был первым инструментом такого рода и по-прежнему остаётся лучшим. Помимо функций других средств, он предоставляет удобное окно графа кучи, где объекты показаны прямоугольными узлами, а по графу объектов можно перемещаться разными способами. Другие инструменты показывают граф только как дерево, что упрощает отображение, но снижает наглядность. Эта и другие дополнительные функции JProbe делают его моим предпочтительным инструментом для отладки памяти. В остальной части главы мы будем в основном пользоваться им, но вы вправе выбрать инструмент по вкусу или имеющейся лицензии.

Исследование использования кучи и частоты сборки мусора для повышения производительности

Сборка мусора - замечательная возможность Java, избавляющая разработчиков от необходимости явно освобождать выделенные объекты. Цена этой простоты - накладные расходы во время работы сборщика. Последние версии JVM применяют сборку мусора по поколениям, способную выполняться асинхронно, однако даже после этих усовершенствований накладные расходы остаются значительными. Влияние сборки мусора на производительность приложения легко недооценить.

JProbe, как и другие профилировщики, показывает сводный график кучи работающего приложения. Он позволяет наблюдать за общим объёмом выделенной и доступной свободной памяти, как показано на рисунке 10.1.

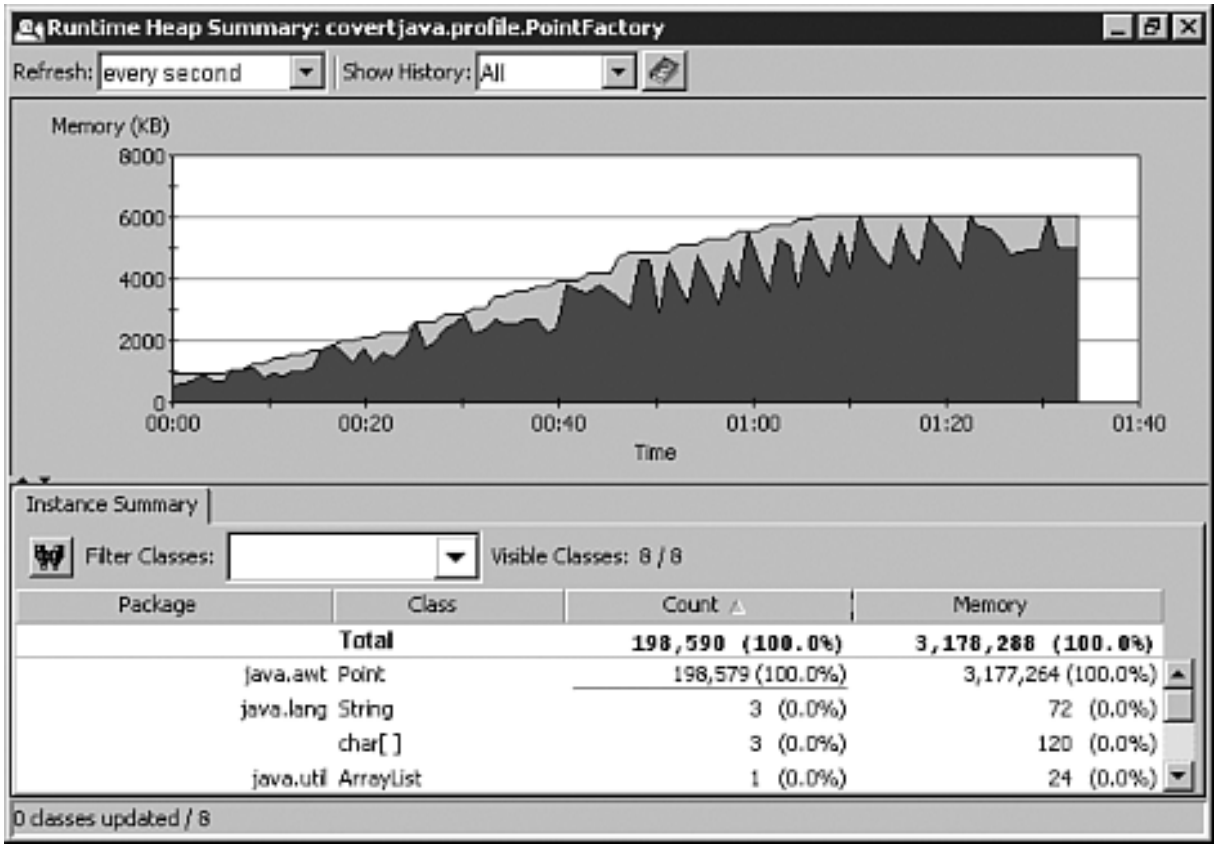

Рисунок 10.1. Сводный график кучи во время выполнения при неоптимизированном размере кучи

Когда свободной памяти недостаточно для удовлетворения запроса, JVM запускает сборку мусора первого поколения, освобождая память неиспользуемых объектов. Первое поколение рассматривает только недавно выделенные объекты, избегая длительного перебора всего графа объектов. Если освобождённой памяти недостаточно, запускается полная сборка мусора. Она начинается с корней дерева объектов, которыми обычно являются объекты, доступные из активных потоков и статических переменных, и определяет все объекты, достижимые из корня. Объекты, недостижимые из корня, то есть не входящие в граф ссылок, начинающийся корневым объектом, помечаются для сборки. Если полная сборка не освобождает достаточно памяти, JVM проверяет, разрешено ли выделить дополнительную память операционной системы. Если достигнут предельный объём или ОС не может предоставить память, создаётся исключение `OutOfMemory`.

Как видите, создание множества новых объектов способно привести к сложным и длительным операциям. Вместо выполнения бизнес-логики JVM может расходовать процессорное время на управление памятью. На рисунке 10.1 показано использование памяти приложением с неоптимизированным максимальным размером кучи. Каждое падение объёма используемой кучи является результатом сборки мусора. Видно, что JVM вынуждена часто запускать сборщик из-за малого объёма свободной памяти. Максимальный размер кучи задаётся параметром `-Xmx` в командной строке `java`. При увеличении максимума с 5 до 16 Мбайт сводный график принимает вид, показанный на рисунке 10.2.

> **Истории с передовой.** Одно из приложений Java, использовавшихся в Riggs Bank, перестало запускаться после резкого увеличения объёма отображаемых данных. Пользователи ждали инициализации до 30 минут, но она так и не завершалась. Исследование запуска выявило признаки бесконечных циклов сборки мусора: высокую загрузку процессора при отсутствии дискового или сетевого обмена. После увеличения максимального размера кучи со 128 до 256 Мбайт приложение стало инициализироваться менее чем за 30 секунд!

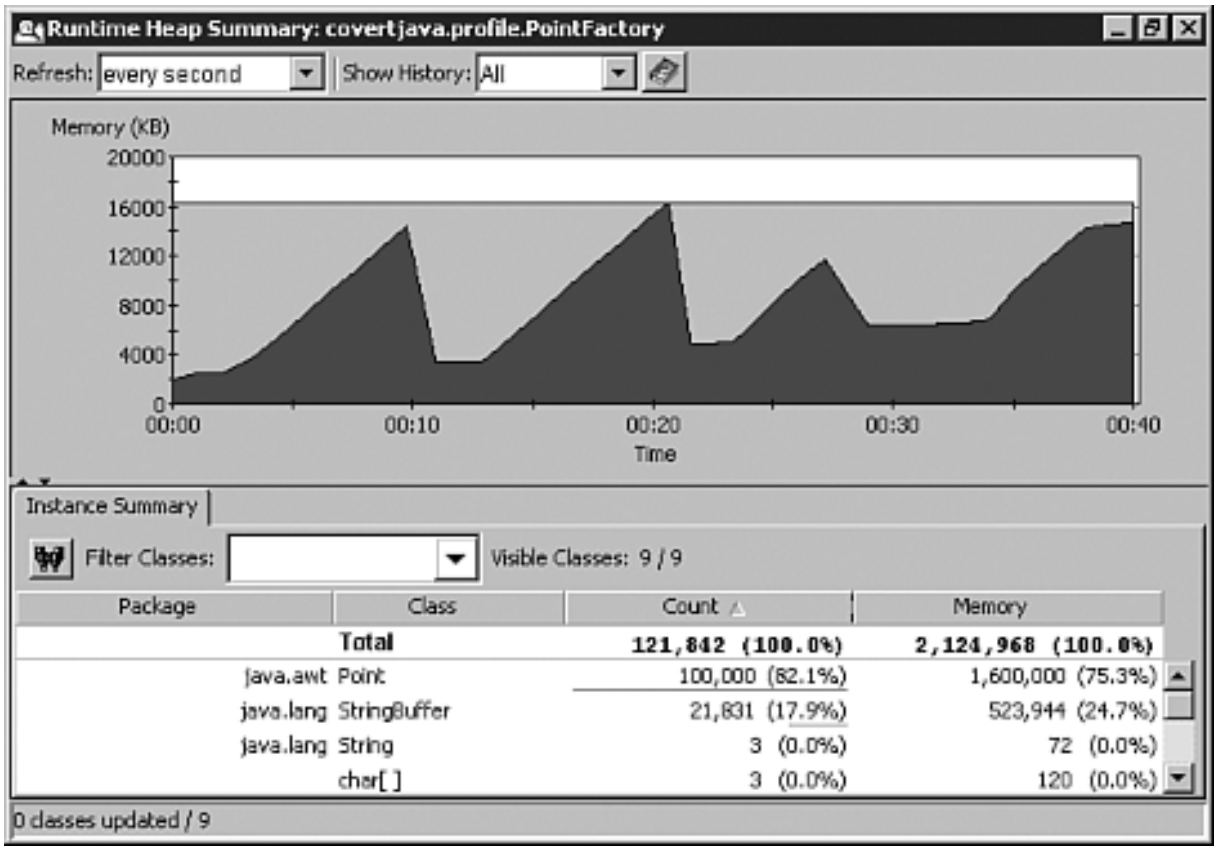

Рисунок 10.2. Сводный график кучи во время выполнения при оптимизированном размере кучи

Производительность приложения значительно выросла, поскольку сборка мусора выполнялась всего 3 раза вместо почти 30. Фиксированного правила определения оптимального размера кучи нет. Иногда несколько циклов сборки выполняются почти так же быстро, как один цикл, которому приходится перебирать больше объектов. Единственный надёжный способ найти лучшую настройку конкретного приложения - экспериментировать с начальными и максимальными размерами кучи.

Просмотр размещения объектов и ссылок для поиска и устранения утечек памяти

Утечки памяти в приложениях Java давно перестали быть неожиданностью. Это не те же утечки, которыми изобилуют языки более низкого уровня, например C++, но память всё равно не возвращается в свободный пул. Как вы уже узнали, сборщик Java освобождает память, занятую недостижимыми объектами. Если на объект существует ссылка, он не подлежит сборке, даже если больше никогда не используется. Например, объект, помещённый в массив и никогда из него не удаляемый, навсегда остаётся в памяти. Такие объекты, иногда называемые задержавшимися, со временем могут поглотить всю свободную память и вызвать исключение `OutOfMemory`. Проблема усиливается, когда объект ссылается на большое дерево других объектов, что часто встречается в приложениях.

Ещё одна опасность - задержавшиеся объекты, представляющие ресурсы, например соединения с базой данных или временные файлы. Чем дольше объект остаётся в памяти, тем дороже сборка мусора, поскольку ей приходится обходить больше объектов. Профилировщики предоставляют лучшие средства поиска задержавшихся объектов и определения причин, мешающих их сборке.

Рассмотрим простой пример, создающий задержавшиеся объекты, и найдём их с помощью JProbe. Предположим, мы разрабатываем графический редактор и нуждаемся в фабрике, предоставляющей слой абстракции для создания объектов `Point`. Фрагмент реализации фабрики показан в листинге 10.1.

```
public class PointFactory {
    protected ArrayList points = new ArrayList();

    public Point createPoint(int x, int y) {
        Point point = new Point(x, y);
        this.points.add(point);
        return point;
    }

    public void removePoint(Point point) {
        this.points.remove(point);
    }
}
```

Листинг 10.1. Реализация `PointFactory`

Метод `createPoint` создаёт экземпляры `Point` и помещает их в массив. `removePoint` удаляет точку из массива, а вместе с `PointFactory` поставляется тестовый метод, создающий через фабрику несколько точек. Его код приведён в листинге 10.2.

```
public void printTestPoints() {
    for (int i = 0; i < 5; i++) {
        Point point = createPoint(i, i);
        System.out.println("Point = " + point);
    }
}
```

Листинг 10.2. Метод проверки создания точек

Метод `main` класса `PointFactory` показывает диалоговое окно с кнопкой Print Test Points. Нажатие кнопки вызывает `printTestPoints()` класса `PointFactory`. Реализация кажется совершенно корректным кодом Java, но её выполнение приводит к утечкам памяти. Если увеличить размер выделяемого объекта и число повторений цикла, всю доступную память можно быстро исчерпать. Вы, несомненно, уже поняли причину утечки. В нашем примере она очевидна, поскольку есть один класс менее чем с 50 строками кода, но при сотнях классов со сложной логикой проблема была бы далеко не столь заметна. Здесь профилировщики становятся незаменимыми.

Чтобы исследовать управление памятью демонстрационного кода, настроим `covertjava.profile.PointFactory` как самостоятельное приложение в JProbe Memory Debugger. После запуска начальное представление показывает сводку кучи и список классов. Поскольку приложение небольшое, в списке видно лишь несколько классов, но в крупной программе можно применить фильтр по имени класса или пакета. Когда появится диалоговое окно приложения, по списку классов и числу экземпляров видно, что объекты `java.awt.Point` ещё не созданы. Нажмём Print Test Points и вернёмся к представлению классов. Теперь видно пять созданных экземпляров `Point` и пять экземпляров `StringBuffer`. Это очевидный результат вызова `printTestPoints`, поэтому удивляться нечему. JProbe позволяет принудительно запустить сборку мусора; сделаем это. После её завершения экземпляры `StringBuffer` исчезнут, но точки останутся в памяти. Повторение теста начиная с кнопки Print Test Points увеличит число экземпляров `Point` до 10. Утечка подтверждена, и теперь выясним, почему точки задерживаются в памяти.

После обнаружения неосвобождаемого экземпляра есть несколько способов найти препятствие для сборки мусора: вручную изучить исходный код, просмотреть дерево ссылок в профилировщике или найти в нём все пути к корню.

Просмотр исходного кода и поиск всех мест создания экземпляра помогает выяснить, какие ссылки не выходят из области видимости и не обнуляются. Обычно профилировщик показывает точку

выделения экземпляра, что облегчает работу. Это прямолинейный подход, способный занять много времени. Более развитый, а иногда единственно осуществимый способ - сделать снимок кучи во время выполнения и просматривать в нём нужный объект. Вместо ручного отслеживания возможных ссылок в исходном коде снимок позволяет перемещаться по фактическим ссылкам времени выполнения, удерживающим объект в памяти. Большинство профилировщиков показывает ссылающиеся объекты и объекты, на которые ссылается рассматриваемый объект. Очень мощная возможность - отображение всех путей от объекта до корня. Графы объектов на снимке могут быть сложны для навигации. Если интересует один объект, весь граф не нужен; достаточно увидеть, какие объекты мешают его сборке. Напомним, сборщик освобождает объект только тогда, когда он недостижим из корней. Следовательно, если выявить и устранить все пути от объекта к корню, он станет пригоден для сборки.

Перед созданием снимка кучи всегда полезно запросить сборку мусора. Она удалит объекты, на которые больше нет ссылок, и упростит поиск утечек. В JProbe для этого выберите Request Garbage Collection в меню Program. Затем создайте снимок, выбрав Take Heap Snapshot в том же меню. Когда JProbe захватит снимок, выберите Class View в меню Snapshot. Теперь можно изучать состояние кучи и графы объектов. Нас интересуют `java.awt.Point`, поэтому щёлкнем этот класс правой кнопкой в Class View и выберем Instance Detail View. Новый экран показывает все находящиеся в памяти экземпляры `Point` и позволяет просматривать деревья входящих и исходящих ссылок. Дерево ссылающихся объектов состоит из иерархии предков, для которых экземпляр прямо или косвенно является дочерним: непосредственных родителей, родителей родителей и так далее. Перемещение по дереву даёт хорошее представление об удерживаемых ссылках. Для нашего экземпляра `Point` видно, что на него ссылается `ArrayList`, на который, в свою очередь, ссылается `PointFactory`.

У JProbe есть другое представление, которым я предпочитаю пользоваться. Оно называется Reference Graph и нагляднее показывает граф объектов. Чтобы открыть его для `Point`, щёлкните экземпляр правой кнопкой и выберите Instance Referrers and References. Появится окно, где все объекты представлены именованными прямоугольниками, а выбранный объект находится в центре. Поскольку мы выясняем, почему точки задерживаются в памяти, нажмите значок Paths to Root на панели графа. Результат показан на рисунке 10.3.

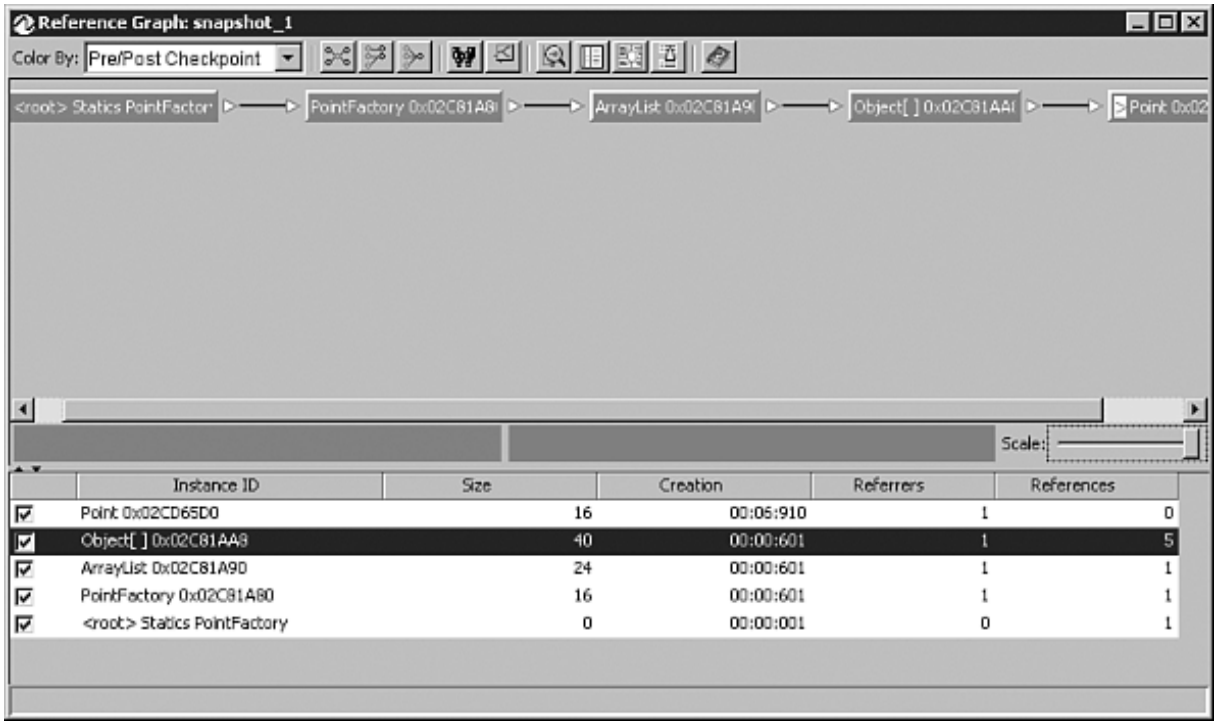

Рисунок 10.3. Граф путей от точки к корню

Мы снова видим, что на объект Point ссылается ArrayList класса PointFactory. Можно заключить, что точки остаются в памяти потому, что никогда не удаляются из массива. В реализации printTestPoint() видно, что метод вызывает createPoint, который сохраняет созданный Point в массиве points. Для исправления нужно добавить в printTestPoint вызов removePoint, как показано в листинге 10.3.

```
public void printTestPoints() {
    for (int i = 0; i < 5; i++) {
        Point point = createPoint(i, i);
        System.out.println("Point = " + point);
        removePoint(point);
    }
}
```

Листинг 10.3. Исправленная реализация проверки создания точек

Исследование распределения и синхронизации потоков

Многопоточным приложениям по самой их природе угрожают проблемы синхронизации. В одном потоке код выполняется в том же порядке, в каком записан в исходном файле. При нескольких потоках одновременно выполняются параллельные операции, а каждый поток может быть прерван, чтобы процессор перешёл к операциям следующего. Порядок и точки прерываний практически случайны и потому непредсказуемы. Большинство приложений Java многопоточно, и хотя встроенное представление о потоках скрывает сложности их распределения, оно не защищает от обычных проблем: гонок данных, взаимных блокировок и зависания потоков.

Гонка данных возникает, когда два потока одновременно пытаются обратиться к общему ресурсу и изменить его. Типичный пример потенциальной гонки - изменение остатка банковского счёта без синхронизации. Допустим, класс BankAccount предоставляет методы getBalance() и setBalance() для чтения и записи текущего остатка, а также deposit() для внесения или снятия средств. Реализация deposit() сначала получает текущий остаток, затем прибавляет сумму и устанавливает новое значение. Если два потока, thread1 и thread2, одновременно вносят средства, thread1 может прочитать остаток и прибавить сумму, но прежде чем записать результат, быть прерванным потоком thread2. Тот прочитает старый остаток и увеличит его, а затем уступит thread1. thread1 запишет рассчитанное значение, но после этого thread2 продолжит выполнение и перезапишет его собственным.

В таком случае сумма, внесённая thread1, будет потеряна. Клиентам, возможно, понравилась бы подобная аномалия при снятии средств, но с внесением они её не потерпят. Профилировщики, например JProbe Threadalizer, способны обнаруживать различные виды гонок и сообщать о них. Обычное решение - добавить синхронизацию, защищающую доступ к данным. Чтобы исправить внесение средств, соответствующий метод следует объявить ключевым словом synchronized. Синхронизация гарантирует, что блокировку получит только один поток, а любой другой, пытающийся получить ту же блокировку, будет остановлен до её освобождения владельцем. Вместо синхронизированных методов можно объявить объект блокировки и передавать его синхронизированному блоку, как в листинге 10.4.

```
protected Object balanceLock = new Object();
protected double balance;

...

public void deposit(double amount) {
    synchronized (balanceLock) {
        double balance = getBalance();
        balance += amount;
        setBalance(balance);
    }
}
```

Листинг 10.4. Защита доступа к данным синхронизированным блоком

Синхронизация создаёт накладные расходы, поэтому применять её следует только при необходимости. Кроме того, не каждая обнаруженная профилировщиком гонка является проблемой, которую нужно устранять. Например, одновременное чтение остатка несколькими потоками безвредно, поэтому `getBalance()` не обязательно синхронизировать. В многоуровневом приложении синхронизацию можно переложить на другие уровни. Например, уровень изоляции транзакций базы данных способен предотвратить параллельное чтение и запись одних данных.

Взаимная блокировка возникает, когда поток ожидает блокировку, захваченную другим потоком, но никогда им не освобождаемую. Это распространённая проблема многопоточных приложений, способная полностью остановить программу. Например, если `thread1` получает `lock1`, а затем ждёт `lock2`, удерживаемую `thread2`, взаимная блокировка возникнет, если `thread2` попытается получить `lock1`, не освободив сначала `lock2`. Рассмотрим демонстрационный класс с двумя блокировками и двумя методами, которые синхронизируют работу с их помощью. В листинге 10.5 фактическая работа неважна: нас интересует логика синхронизации.

```
public Object lock1 = new Object();
public Object lock2 = new Object();

public void method1() {
    synchronized (lock1) {
        synchronized (lock2) {
            doSomething();
        }
    }
}

public void method2() {
    synchronized (lock2) {
        synchronized (lock1) {
            doSomething();
        }
    }
}
```

Листинг 10.5. Код с возможностью взаимной блокировки

Оба метода используют `lock1` и `lock2`, но получают их в разном порядке. При выполнении в отдельных потоках `method1` может войти в первый синхронизированный блок и захватить `lock1`. Если его прервёт `method2`, тот войдёт в свой внешний блок и захватит `lock2`. После продолжения выполнения возникает взаимная блокировка, поскольку оба потока бесконечно ждут занятые блокировки. Класс `ThreadTest` пакета `covertjava.profile` демонстрирует этот сценарий. Его запуск создаёт взаимную блокировку, мешающую корректно завершить JVM, потому что два потока, порождённые `main()`, никогда не возвращаются.

В примере проблему легко заметить при внимательном изучении `ThreadTest`, но в реальном приложении с сотнями классов это было бы значительно труднее. Хорошие профилировщики обнаруживают взаимные блокировки и сообщают, что к ним привело. JProbe Threadalizer от Quest и OptimizeIt Thread Debugger от Borland умеют их находить, но OptimizeIt также предоставляет

наглядную временную шкалу выполнения потоков, поэтому воспользуемся им. Запустите OptimizeIt и настройте `ThreadTest` как приложение с главным классом `covertjava.profile.ThreadTest`. В диалоговом окне параметров обязательно установите Auto-start Analyzer, чтобы автоматически начать анализ работающей программы. Запустите `ThreadTest` из меню или кнопкой панели; через несколько секунд OptimizeIt должен показать потоки, как на рисунке 10.4.

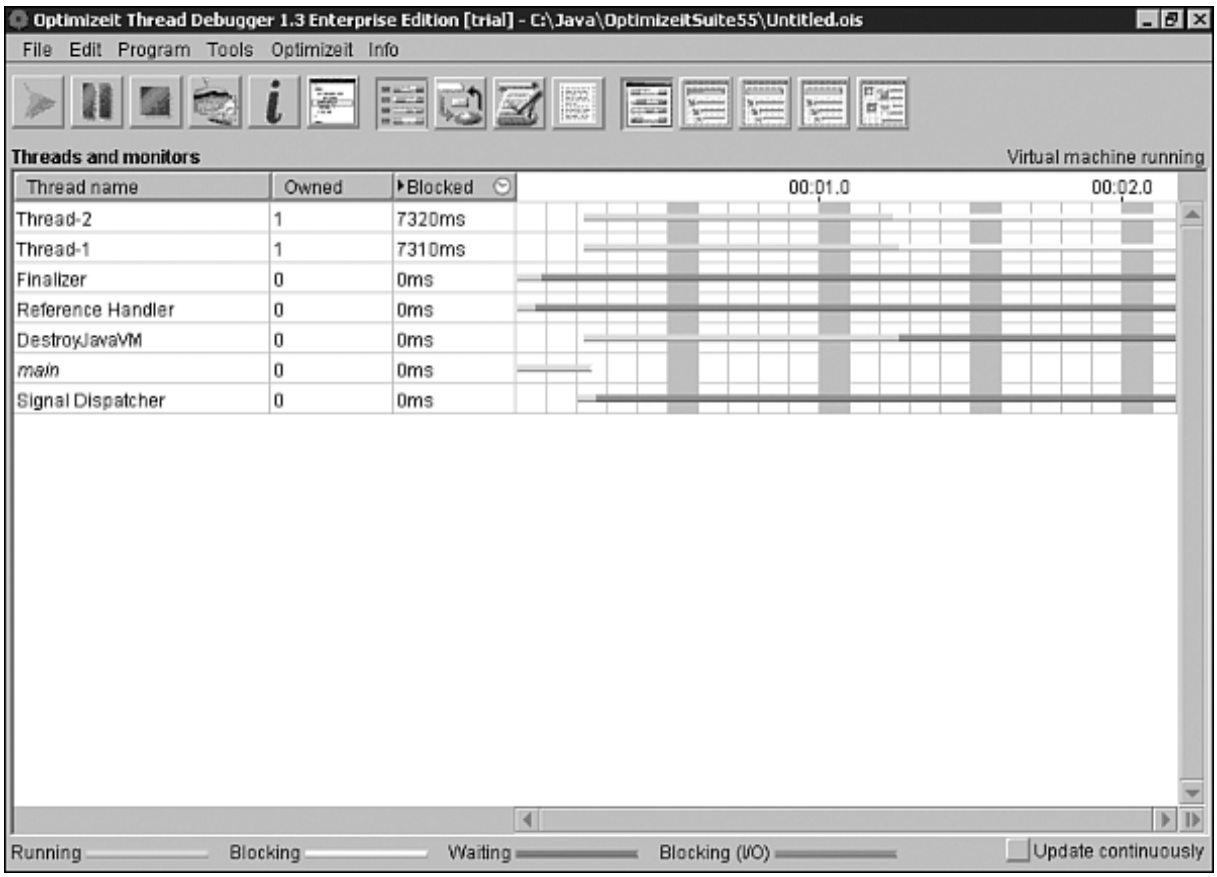

Рисунок 10.4. Представление потоков работающего `ThreadTest` *в OptimizeIt*

Представление показывает все потоки и временную шкалу их состояний. Зелёный цвет означает выполнение, жёлтый - блокировку в ожидании занятого другим потоком монитора, красный - ожидание уведомления, фиолетовый - блокировку в нативном коде. `ThreadTest` порождает два потока со стандартными именами `Thread-1` и `Thread-2`. На временной шкале видно, что после короткого выполнения они блокируются навсегда. Остальные потоки запускаются JVM для системных задач, поэтому сосредоточимся на потоках приложения. OptimizeIt показывает как удерживаемые каждым потоком блокировки, так и блокировки, которых он ждёт. Поскольку мы включили анализатор, воспользуемся его подсказкой. Через меню File перейдём к Analyzer, затем нажмём Stop Recording. Инструмент должен обнаружить одну взаимную блокировку с описанием `Locking order mismatch`. Выбор записи покажет подробные сведения о потоках и удерживаемых блокировках. В нашем случае текст гласит: `Thread Thread-2 enters monitor java.lang.Object 0xabf9280 first and then java.lang.Object 0xabf72f0 while thread Thread-1 enters the same monitors in the opposite order`. OptimizeIt даже показывает строки кода, где были захвачены блокировки, поэтому исправить проблему легко.

Чтобы избежать взаимной блокировки, блокировки следует получать в одном порядке. В нашем случае `method2` должен сначала получить `lock1`, а затем `lock2`. И последнее: хотя анализатор обнаруживает многие распространённые ошибки, ведущие к взаимной блокировке, он не способен выявить любую возможную ситуацию. Хорошее проектирование и надёжные приёмы программирования значительно помогают предотвратить будущие проблемы.

Зависанием потока называют ситуацию, когда поток ждёт уведомления, которое никогда не приходит. Это происходит, если поток вызывает для объекта `wait()`, но никакой другой поток не вызывает для того же объекта `notify()`. Зависание - ещё одна проблема многопоточности, обнаруживаемая профилировщиком.

Поиск дорогостоящих методов для повышения производительности

Оптимизация производительности - сложная тема, далеко выходящая за рамки главы. Производительность Java прошла долгий путь, и современные JIT-компиляторы обеспечивают превосходную скорость и оптимизацию кода. Самый важный фактор конечной скорости - качество проектирования и реализации приложения. Известные ловушки и типичные обходные решения способны повысить производительность на сотни процентов. Нередко переписывание нескольких ключевых методов ускоряет отклик приложения в 10 раз. Оптимизация начинается с поиска узких мест. Естественно, производительность следует учитывать уже при проектировании, хотя жертвовать чистотой архитектуры ради ожидаемого небольшого выигрыша не рекомендуется. По мере разработки и модульного тестирования код следует профилировать, выясняя, какие методы расходуют больше всего времени. Основная часть профилирования обычно выполняется при интеграционном тестировании.

Профилирование собирает различные статистические показатели выполнения, позволяющие понять, на что фактически уходит время. Ниже перечислены самые полезные показатели JProbe:

• **Время метода.** Время выполнения данного метода без учёта времени вызванных им методов.
• **Число вызовов метода.** Количество вызовов; позволяет выявить методы, оптимизация которых сильнее всего повлияет на общую производительность.
• **Среднее время метода.** Среднее время выполнения, равное времени метода, делённому на число вызовов; позволяет находить самые медленные методы.
• **Накопленное время.** Общее время выполнения метода с учётом вызванных им методов; позволяет определить методы, потребляющие основную часть процессорного времени.
• **Среднее число объектов на метод.** Среднее количество объектов, созданных внутри метода. Поскольку сборка мусора сильно влияет на производительность Java, сокращение числа создаваемых объектов может ускорить выполнение.

После сбора и анализа статистики можно оптимизировать плохо работающие части приложения.

Исследование приложения во время выполнения с помощью дампа потоков

Поскольку профилировщик требует запуска JVM в режиме отладки, он непригоден для промышленных приложений. Было бы полезно получать снимок потоков и их работы для любого приложения, даже запущенного в обычном режиме. К счастью, JVM можно приказать создать полный дамп потоков с их состояниями и стеками вызовов. В Unix выполните `kill -3 <pid>`, где `<pid>` - идентификатор процесса JVM; в Solaris также работает `kill -QUIT <pid>`. В Windows нажмите Ctrl+Break в окне консоли. Дамп Chat приведён в листинге 10.6.

```
Full thread dump:

"RMI ConnectionExpiration-[10.241.11.244:14512]" daemon prio=7
tid=0x8ad0b68 nid=0x748 waiting on monitor [0x119df000..0x119dfdbc]
at java.lang.Thread.sleep(Native Method)
at sun.rmi.transport.tcp.TCPChannel$Reaper.run(TCPChannel.java:522)
at java.lang.Thread.run(Thread.java:479)

"TimerQueue" daemon prio=5 tid=0x8ac15d8 nid=0x8a0
waiting on monitor [0x1177f000..0x1177fdbc]
at java.lang.Object.wait(Native Method)
at javax.swing.TimerQueue.run(TimerQueue.java:228)
at java.lang.Thread.run(Thread.java:479)

"AWT-EventQueue-0" prio=7 tid=0x8a1ac28 nid=0x148
waiting on monitor [0x10f3f000..0x10f3fdbc]
at java.lang.Object.wait(Native Method)
at java.lang.Object.wait(Object.java:415)
at java.awt.EventQueue.getNextEvent(EventQueue.java:255)
at java.awt.EventDispatchThread.pumpOneEventForHierarchy(EventDispatchThread...)
at java.awt.EventDispatchThread.pumpEventsForHierarchy(EventDispatchThread...)
at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:88)
at java.awt.EventDispatchThread.run(EventDispatchThread.java:80)

"Signal Dispatcher" daemon prio=10 tid=0x802388 nid=0x2b0
waiting on monitor [0..0]
```

Листинг 10.6. Частичный дамп потоков Chat (начало)

```
"Finalizer" daemon prio=9 tid=0x7fe368 nid=0x640
waiting on monitor [0x8c4f000..0x8c4fdbc]
at java.lang.Object.wait(Native Method)
at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:103)
at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:118)
at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:157)

...
```

Листинг 10.6. Частичный дамп потоков Chat (окончание)

Листинг 10.6 показывает некоторые потоки, работающие внутри JVM Chat. Например, видно, что поток соединения RMI спит в методе `run` класса `sun.rmi.transport.tcp.TCPChannel$Reaper`. Поток `AWT-EventQueue` ждёт следующего события, а поток финализатора ожидает монитор методом `wait()`.

Дампы потоков очень полезны при взаимной блокировке работающего приложения, если её причина неочевидна. Сведения о потоках и ожидаемых ими мониторах - лучшая возможная подсказка. В Unix дамп также включает переменные среды, сведения об ОС, дамп мониторов и множество других полезных данных.

Короткий тест

1. Зачем профилировать приложение?
2. Почему сборка мусора столь сильно влияет на производительность?
3. Что может вызывать утечки памяти в приложении Java?
4. Как найти и устранить утечки памяти?
5. Какие проблемы распространены в многопоточных приложениях?
6. Какой процесс следует использовать для повышения производительности приложения?

Краткие итоги

• Профилировщики предоставляют развитые средства решения таких проблем, как утечки памяти, гонки данных, взаимные блокировки и низкая производительность.
• Исследование использования кучи и частоты сборки мусора подсказывает способы повышения производительности.
• Просмотр размещения объектов и ссылок во время выполнения помогает находить и устранять утечки памяти.
• Исследование распределения и синхронизации потоков помогает находить проблемы блокировок и гонок данных и повышает производительность.
• Выявление дорогостоящих методов при профилировании повышает производительность приложения.
• Исследование приложения в профилировщике позволяет понять его внутреннюю структуру.
• Дамп потоков, создаваемый JVM в ответ на `SIGQUIT`, предоставляет полезные сведения о потоках и мониторах приложения, не работающего в режиме отладки.

Глава 11. Нагрузочное тестирование для поиска и устранения проблем масштабируемости

> «Ошибки в программе невозможно обнаружить никому, кроме конечного пользователя».
>
> *Законы Мерфи о технике*

В этой главе

• Важность нагрузочного тестирования
• Нагрузочное тестирование серверов RMI с помощью JUnit
• Нагрузочное тестирование с помощью JMeter
• Короткий тест
• Краткие итоги

Важность нагрузочного тестирования

К серверным приложениям предъявляются требования к уровню обслуживания, определяющие доступность, масштабируемость и аварийное переключение:

• **Доступность** задаёт требования ко времени непрерывной работы, то есть продолжительность работы приложения без перезапуска.
• **Масштабируемость** определяет способность приложения обеспечивать тот же уровень обслуживания при увеличении числа запросов.
• **Аварийное переключение** определяет способность приложения сохранять уровень обслуживания при отказе одного из компонентов.

Обычный цикл разработки предусматривает время для модульного и интеграционного тестирования, обычно сосредоточенного на функциях, но не всегда оставляет время на испытание нагрузкой. Цель нагрузочного тестирования - оценить, насколько производительность системы под нагрузкой отвечает требованиям к уровню обслуживания. Разумеется, время отклика любой системы ухудшается с ростом нагрузки, но пока оно укладывается в заданные требования, система считается масштабируемой.

Игнорировать нагрузочное тестирование рискованно, особенно если приложение должно обслуживать сотни или тысячи пользователей. При большой аудитории маленькая проблема превращается в крупную, поскольку затрагивает множество людей. Некоторые сбои возникают лишь при определённой нагрузке. Это возможно в операциях, зависящих от таких ресурсов, как потоки, соединения с базой данных и память. Большинство проблем многопоточных приложений проявляется при определённом числе параллельных запросов. Только испытание нагрузкой и длительная работа системы воспроизводят промышленную среду, поэтому до ввода системы в эксплуатацию такое тестирование совершенно необходимо. Наконец, оно показывает, как приложение отреагирует на атаки отказа в обслуживании и попытки взлома. Нагрузочное тестирование выявляет не способ построения приложения, а его долговечность. Оно помогает извлечь больше пользы из методов других глав. Например, профиль приложения под нагрузкой отличается от профиля без нагрузки.

Имитировать нагрузку на систему непросто, поэтому важно выбрать правильные инструменты. На рынке есть множество продуктов, большинство из которых предназначено для веб-сайтов, обслуживающих содержимое HTML по HTTP. Помимо HTTP серверные приложения Java работают с другими протоколами, например JRMP для клиентов RMI и IIOP для клиентов CORBA и RMI. Лучшие инструменты стоят дорого, однако альтернативы с открытым исходным кодом предоставляют большинство базовых функций. Мы испытаем работающее по RMI приложение Chat с помощью открытого каркаса JUnit. Затем применим другой открытый инструмент, JMeter, к веб-приложению WebCream с браузерными клиентами.

> **Истории с передовой.** Мы создавали веб-приложение для 5000 пользователей. Первый выпуск пропустил несколько сроков, поэтому после разработки и модульного тестирования не осталось времени ни на автоматические сценарии, ни на полноценное нагрузочное тестирование. Несколько тестировщиков вручную имитировали некоторую нагрузку, после чего развёртывание, со скрещёнными пальцами, получило разрешение. Приложение ввели в эксплуатацию в примечательное время, в два часа ночи, из-за окна обслуживания сервера, а измученная команда отправилась домой в надежде отдохнуть несколько дней. Но к 11 утра, когда пользователи начали обращаться к сайту, сервер приложений перестал отвечать. Перезапуск помогал на несколько часов, затем взаимная блокировка повторялась. Руководство пришло в ярость, раздавались угрозы найти виновного и уволить его. Наконец выяснилось, что причиной был многопоточный код обработки исключений и аварийного восстановления. При исключении во время запроса код анализировал его, иногда пытался восстановиться, а затем повторял операцию. При множестве параллельных запросов последовательность анализа, восстановления и повтора превращалась в бесконечный цикл. Своевременное нормальное нагрузочное тестирование обнаружило бы ошибку без ущерба для тысяч пользователей и репутации команды.

Основные принципы имитации нагрузки почти всегда одинаковы. Сначала работающим приложением или программно создаётся тестовый сценарий. Затем на его основе создают виртуальных пользователей или клиентов, запускаемых во множестве потоков для одновременного обращения к серверу. Сервер воспринимает виртуальных пользователей как настоящий трафик, а контроль правильности и времени ответа даёт результаты испытания.

Нагрузочное тестирование серверов RMI с помощью JUnit

JUnit - открытый проект Java, предоставляющий каркас написания и выполнения модульных тестов. Он поощряет создание тестового кода, утверждающего корректность функций приложения. Тест JUnit - класс Java, который компилируется и выполняется для проверки кода приложения. Такой подход сочетает автоматическое повторное тестирование с простотой синхронного сопровождения тестового и прикладного кода. И наконец, разработчики пишут Java, а не нажимают кнопки в отладчиках и тестовых инструментах, что, вероятно, немало способствует популярности каркаса.

Для работы с JUnit разработчик пишет тест, расширяющий `junit.framework.TestCase` или реализующий `junit.framework.Test`. Тест вызывает проверяемые классы и утверждает соответствие возвращаемых значений ожидаемым. Например, тест банковского счёта получает текущий остаток, вносит сумму и проверяет, что новый остаток равен прежнему плюс внесённая сумма. Качество теста прямо пропорционально усердию разработчика. Следует охватить все возможные сценарии, включая ошибочные. После написания тесты компилируются и выполняются отдельно или группами. JUnit хорошо документирован и прост в изучении; если вы с ним не работали, потратьте пару часов на руководство и примеры. Не забудьте обновить резюме: хорошие

руководители увидят в этом признак качественного разработчика. Каркас и документация бесплатно доступны на junit.org. Остальная часть раздела посвящена нагрузочному тесту Chat на основе JUnit.

Chat не создавался неразрушимым, но и не предназначался для сотен пользователей. Если он выдерживает от трёх до шести одновременных пользователей, этого, вероятно, достаточно для демонстрационного приложения. В любом нагрузочном тесте верхнюю планку следует ставить немного выше ожидаемого максимума. Поэтому в примере мы потребуем поддержки 10 виртуальных клиентов, одновременно отправляющих сообщения одному приложению Chat. Вызовы также нужно разнести во времени, имитируя реальное поведение. Это означает отправку запросов примерно в одно время, а не точно одновременно. Иногда «одновременными» называют клиентов, отправляющих запрос строго в один момент, а «параллельными» - поддерживающих разговор с сервером, но посылающих запросы примерно в одно время. В однопроцессорной системе истинной параллельной обработки нет, поэтому различия между этими понятиями фактически не существует.

Сначала разработаем тест, имитирующий отправку пользователем одного сообщения целевому узлу. Затем создадим оболочку, которая порождает несколько виртуальных пользователей, многократно отправляющих сообщения. Для гибкости разрешим задавать число имитируемых одновременных пользователей, количество повторов и задержку, разносящую вызовы во времени.

Тест написан в классе `covertjava.loadtest.ChatTestCase`. Он расширяет `TestCase`, а основная логика находится в методе `testSendMessage()`, приведённом в листинге 11.1.

```
public void testSendMessage() {
    logger.info("Sending test message...");
    try {
        StringBuffer message = new StringBuffer();
        message.append("[ChatTestCase@");
        message.append(Integer.toHexString(this.hashCode()));
        message.append("] Test ");
        message.append(messagesSent++);

        ChatServer.getInstance().sendMessage(this.host, message.toString());
        logger.info("Sent message successfully");
    }
    catch (Exception e) {
        e.printStackTrace();
        assertTrue("Exception: " + e.getMessage(), false);
    }
}
```

Листинг 11.1. Исходный код `testSendMessage`

`ChatTestCase` создаёт тестовое сообщение и отправляет его целевому узлу через `ChatServer`. Ключевой момент - вызов `assertTrue()` в блоке `catch` со вторым параметром `false`, сообщающий JUnit о неудаче. В противном случае метод просто возвращает управление, что означает успех. Такая проверка, разумеется, далека от идеального подтверждения правильной обработки сообщения сервером Chat. Она не проверяет разбор сообщения и правильное добавление в окно истории. Однако для испытания сетевого обмена и пропускной способности удалённого сервера это вполне достойный приём, достаточный для иллюстрации.

Следующий шаг - создание набора с экземплярами `ChatTestCase`. Это выполняется в классе `ChatLoadTest`; код показан в листинге 11.2.

```
ActiveTestSuite suite = new ActiveTestSuite();
for (int i = 0; i < clientsNumber; i++) {
```

Листинг 11.2. Создание набора тестов (начало)

```
    Test test = new ChatTestCase();
    test = new DelayedTest(test, (int)(Math.random()*lagTime));
    test = new RepeatedTest(test, repeatRuns);
    suite.addTest(test);
}
```

Листинг 11.2. Создание набора тестов (окончание)

Такие параметры, как `clientsNumber` и `lagTime`, читаются из файла свойств. Наборы JUnit служат контейнерами тестов и упрощают их совместный запуск. `ActiveTestSuite` одновременно запускает все тесты и ждёт их завершения. JUnit применяет шаблон декоратора, чтобы добавлять тестам функции. Например, `RepeatedTest` многократно выполняет заданный поток. Для разнесения выполнения мы добавили декоратор `DelayedTest`, который перед запуском засыпает на заданное время. Итог листинга 11.2 - `clientsNumber` клиентов, которые после случайной задержки одновременно отправят сообщения.

Тест JUnit можно запускать разными способами, но мы воспользуемся Swing GUI, чтобы видеть ход испытания. Если Chat на `localhost` ещё не работает, запустим его файлом `CovertJava\distrib\bin\chat.bat`. Затем откроем GUI JUnit и выполним набор с помощью `loadtestJUnit.bat` из каталога `CovertJava\bin`. Вскоре после начала GUI должен выглядеть примерно так, как на рисунке 11.1.

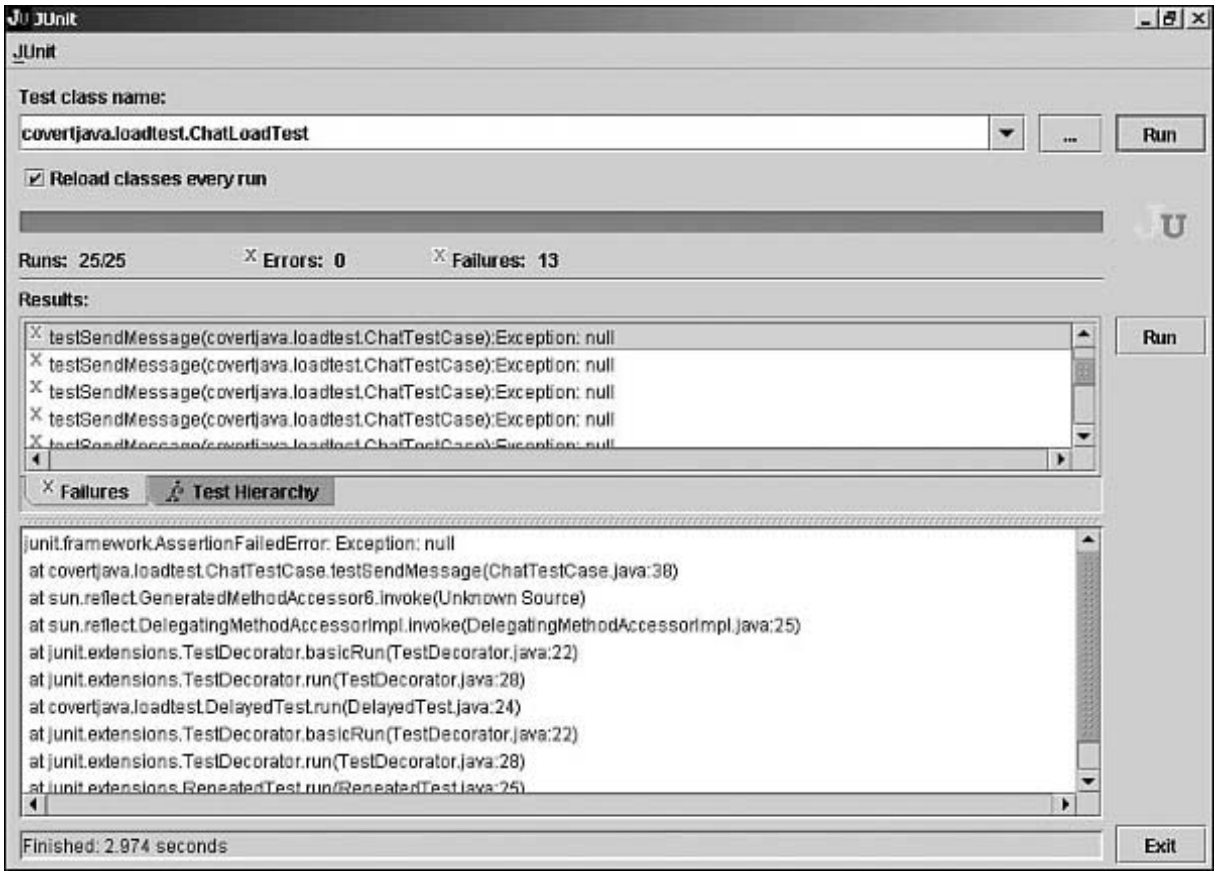

Рисунок 11.1. GUI JUnit, показывающий ход тестирования

Большинство тестов завершилось неудачей, а на панели результатов видно сообщение `testSendMessage(covertjava.loadtest.ChatTestCase): Exception java.lang.NullPointerException: null`. В приложении Chat область истории превратилась в беспорядок, а в консоли появилось несколько `NullPointerException`. Вот и первые плоды

нагрузочного тестирования. Если уменьшить число клиентов до двух, тесты проходят успешно, следовательно, проблема связана с многопоточностью. В главах 9 «Взлом кода с помощью нестандартных отладчиков» и 10 «Использование профилировщиков для анализа приложения во время выполнения» представлены способы поиска и устранения ошибок параллельного выполнения. Попробуйте применить их к Chat.

Для достаточно опытных читателей раскрою причину: новые сообщения неправильно добавляются в историю. Swing не является потокобезопасным, и взаимодействовать с его компонентами обычно рекомендуется из потока диспетчеризации событий AWT. Проектировщики Swing пожертвовали надёжностью ради скорости, и их трудно винить, поскольку производительность Swing давно подвергалась критике. Chat получает входящие сообщения в потоке RMI и передаёт обработку классу `MainFrame`. Метод `appendMessage` класса `MainFrame`, добавляющий сообщение в `JEditorPane`, не использует синхронизацию. Поэтому при одновременной отправке несколькими пользователями несколько потоков пытаются записать данные в один `JEditorPane`. Это классическая проблема перезаписи данных, устраняемая объявлением `appendMessage` синхронизированным. После изменения повторный нагрузочный тест проходит чисто, и работу можно с радостью считать выполненной.

Преимущество JUnit для нагрузочного тестирования - простота и возможность использовать уже написанные тесты. Кроме того, он эффективен для серверов RMI, поскольку автоматическая запись сценариев инструментами нагрузочного тестирования не всегда даёт сопровождаемый результат.

Нагрузочное тестирование с помощью JMeter

В предыдущем разделе показано нагрузочное тестирование приложений Java с помощью JUnit. Разработка была проста, и хотя модных графиков и диаграмм мы не получили, основная работа выполнена неплохо. Однако подход ограничен: виртуального клиента приходится писать вручную, а JUnit умеет лишь запускать тест во множестве потоков. А если нужно проверить веб-приложение с интерфейсом HTML? Обычно пользователи работают в браузере, а сервер реализует интерфейс сервлетами и JSP. JUnit не подходит, поскольку виртуальные пользователи должны поддерживать функции браузера: сеансы, файлы cookie, формы HTTP и прочее. Другой недостаток JUnit - отсутствие осязаемого доказательства успеха или неудачи. Опытный разработчик знает, какое влияние на руководство оказывают отчёты, диаграммы и графики. Одним словом, нужен инструмент получше.

Нагрузочное тестирование - огромный прибыльный рынок с множеством хороших продуктов. Сегодня их основные функции почти одинаковы: автоматическая запись виртуальных пользователей, программное создание сценариев, поддержка нескольких языков и протоколов и, как вы догадались, множество эффектных графиков и отчётов. Нередко именно графики влияют на окончательный выбор. Отметим двух лидеров: Mercury Load Runner и Rational Test Suite. Load Runner - превосходный проверенный временем инструмент предельной гибкости, содержащий практически любую возможную функцию. Особенно важна запись и настройка сценариев виртуальных пользователей, иногда полностью устраняющая необходимость писать код. Он даже способен записывать обмен на уровне протокола для HTTP- и RMI-клиентов. Поскольку ценность нагрузочных испытаний справедливо считается высокой, инструменты могут быть очень дорогими. Проверка кластера серверов сотнями виртуальных пользователей обходится в тысячи долларов лицензий. Мы рассмотрим открытую альтернативу JMeter, бесплатно доступную на сайте Apache. Она далеко не так отполирована и универсальна, но предоставляет сходные основные функции и достаточна для большинства веб-приложений. Не сомневайтесь: несколько графиков и отчёт мы тоже создадим.

Обзор JMeter

JMeter предназначен для нагрузочного тестирования и измерения производительности веб-сайтов и приложений Java. Он поддерживает серверы с соединениями HTTP и FTP, но также может проверять базы данных, сценарии Perl и объекты Java. JMeter предоставляет базовую запись сценария через прокси-сервер и требует глубокого понимания протокола и реализации сервера. Создание плана требует ручной работы, но, как и JUnit, это та сложная задача, которая действительно нравится разработчикам. Поскольку большинство новых приложений имеет тонкий клиент, с помощью JMeter мы испытаем веб-продукт WebCream.

В JMeter есть GUI для создания плана и движок его выполнения. План служит контейнером элементов конфигурации и логических контроллеров, представляющих настройки и действия. Тест создаётся визуально и в результате представляет собой дерево вложенных элементов, описывающее испытание и его выполнение. HTTP Proxy Server, записывающий действия браузера, помогает быстро получить черновой план. Независимо от способа начала необходимо знать элементы JMeter:

• **Thread Group** задаёт число виртуальных пользователей, параллельно выполняющих вложенные узлы, время нарастания для разнесения запуска и продолжительность теста.
• **Listeners** визуализируют результаты и отслеживают выполнение. Например, Graph Results добавляет график времени ответа.

• **Configuration Elements** добавляют зависящие от протокола диспетчеры и стандартные настройки семплеров. Для поддержки HTTP-сеансов через cookie необходимо добавить HTTP Cookie Manager.
• **Assertions** проверяют корректность ответа сервера. Само получение ответа не означает успеха. Можно искать подстроку или точное совпадение; при неудаче утверждение считается проваленным и показывается Assertion Results.
• **Preprocessors** выполняются перед операцией. Например, User Parameters определяет и инициализирует переменные HTTP-запроса.
• **Postprocessors** выполняются после операции. Например, Regular Expression Extractor извлекает заголовок из возвращённой HTML-страницы.
• **Timers** вводят запуск по расписанию и разнесение операций.

У группы потоков могут быть дополнительные узлы:

• **Logic Controllers** задают поток управления вложенных узлов. Например, Loop Controller многократно выполняет вложенные узлы.
• **Samplers** отправляют запрос проверяемому приложению и выполняют фактические вызовы сервера. Сейчас JMeter поддерживает FTP, HTTP, SOAP, Java, JDBC и LDAP.

Созданный план можно выполнить на локальной машине. Несколько машин можно настроить как удалённые серверы JMeter и управлять ими из одного GUI, имитируя больше клиентов, чем выдерживает одна машина.

Обзор WebCream

WebCream - уникальный инструмент Java, автоматически предоставляющий веб-доступ к GUI-приложениям и апплетам Java. Разработчики создают интерфейс с помощью AWT и Swing и одновременно получают HTML-доступ. WebCream можно представить как динамический преобразователь Java в HTML, на лету переводящий окна и диалоги GUI в HTML, а затем имитирующий действия веб-страницы как события GUI, сохраняя исходную логику. Он не требует менять существующие формы или бизнес-логику и изучать API. Благодаря браузерному интерфейсу, динамическому содержимому и сеансам WebCream хорошо подходит для нагрузочного тестирования HTTP с помощью JMeter.

WebCream поставляется со встроенным Tomcat и демонстрационным приложением; немного поработайте с ними, чтобы познакомиться с продуктом. Обязательно изучайте исходный HTML каждой созданной страницы. Постарайтесь понять используемые URL, способ передачи данных серверу и общие шаблоны страниц. Стандартная редакция бесплатна, но ограничена пятью параллельными пользователями, поэтому во время тестов может потребоваться перезапускать Tomcat, если вы не хотите ждать истечения сеансов. Главная страница демонстрации показывает окно с тремя кнопками: Login Dialog, Tabs and Table и Tree Dialog, как на рисунке 11.2.

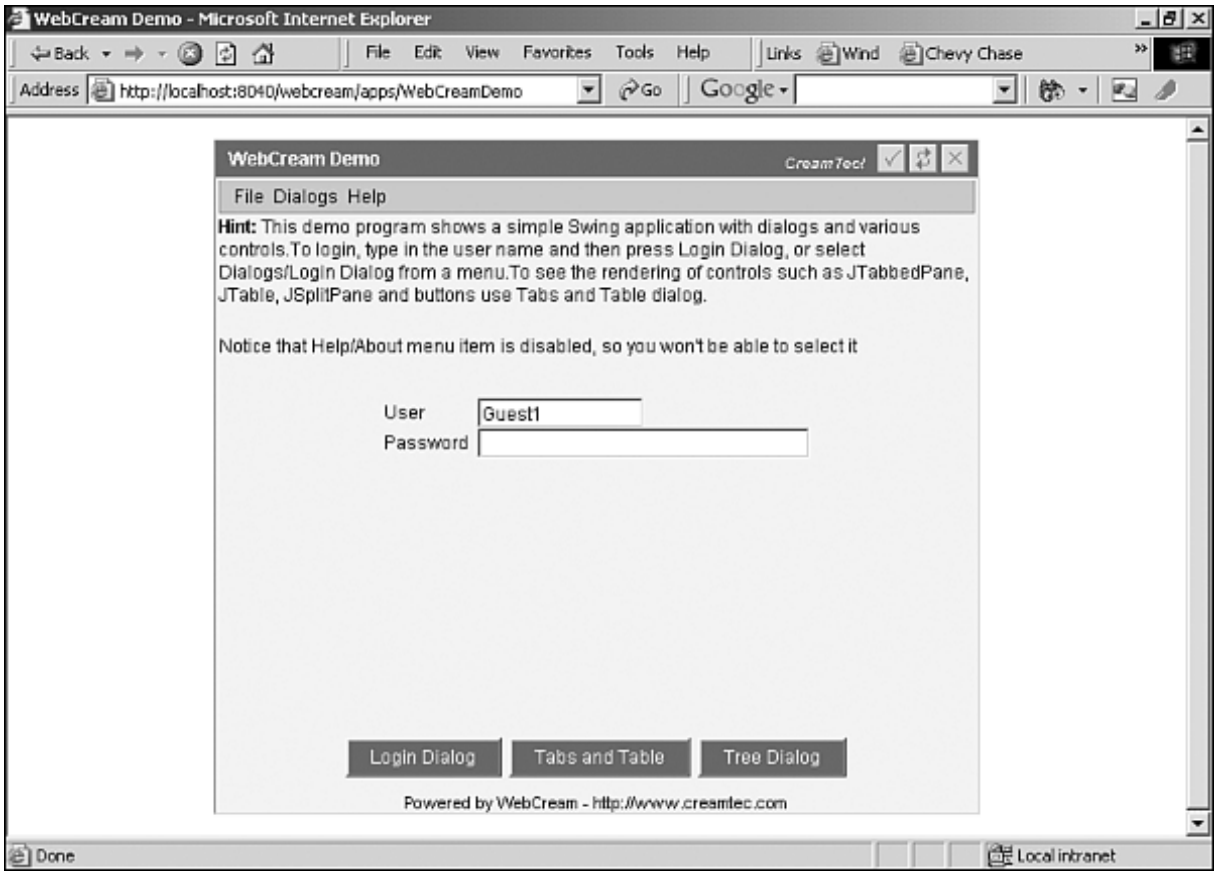

Рисунок 11.2. Главная HTML-страница демонстрационного приложения WebCream

Нажатие Login Dialog открывает страницу с диалогом ввода имени и пароля и выбора домена. Если нажать OK, сведения о входе передаются главной странице. В тесте мы ограничимся этой функцией.

Создание плана веб-теста

Пора приступить к работе. Загрузите и установите JMeter и WebCream. Запустите GUI файлом `JMeter\bin\jmeter.bat`. Поскольку мы проверяем WebCream, запустите встроенный Tomcat файлом `WebCream\bin\startServer.bat`.

Начальное дерево JMeter показывает пустой Test Plan и WorkBench. WorkBench - место для временных узлов и экспериментов, поэтому сосредоточимся на плане. Для имитации нескольких одновременных пользователей добавим группу потоков: щёлкните Test Plan правой кнопкой и выберите Add, Thread Group. Сначала зададим один поток для простоты, позднее изменим число на 5.

Чтобы имитировать реальное взаимодействие, пользователь должен делать паузы перед запросами. Сто параллельных пользователей не означают сто одновременных запросов, ведь люди читают ответы и заполняют формы, а иногда уходят за кофе. JMeter предоставляет Gaussian Random Timer. Добавьте его группе потоков и задайте постоянную задержку 300 миллисекунд с отклонением 100 миллисекунд. Перед следующим шагом JMeter будет приостанавливать поток как минимум на 300 миллисекунд.

До создания HTTP-запросов сделаем приготовления. WebCream поддерживает HTTP-сеанс посредством cookie браузера. Мы имитируем браузер, поэтому добавим группе HTTP Cookie Manager. Одного этого элемента достаточно, чтобы JMeter сохранял cookie; вручную определять их не нужно.

Наконец, используем HTTP Request Defaults, чтобы один раз задать общие сведения всех запросов: протокол HTTP, сервер `localhost`, путь `/webcream/apps/WebCreamDemo` и порт `8040`. В исходном HTML WebCream заметны три скрытых параметра каждой страницы, показанные в листинге 11.3.

```
<input type="hidden" name="__RequestId" value="1">
<input type="hidden" name="__WindowTitle" value="WebCream Demo">
<input type="hidden" name="__Action" value="">
```

Листинг 11.3. Общие параметры формы WebCream

При переходе по страницам значения меняются, отражая последовательный номер запроса и заголовок окна. Параметр `__Action` сообщает серверу нужное действие; например, для закрытия окна клиент задаёт `close`. Поскольку параметры отправляются с каждым запросом, разумно включить их в HTTP Request Defaults. Добавьте три параметра в GUI, пока оставив значения пустыми. Мы вернёмся к ним после знакомства с JMeter. Полученное дерево показано на рисунке 11.3.

Мы имитируем открытие демонстрации в браузере, нажатие Login Button, переход к странице входа и нажатие OK для возврата. Чтобы выявить утечки и проблемы многопоточности, пользователь несколько раз откроет и закроет диалог.

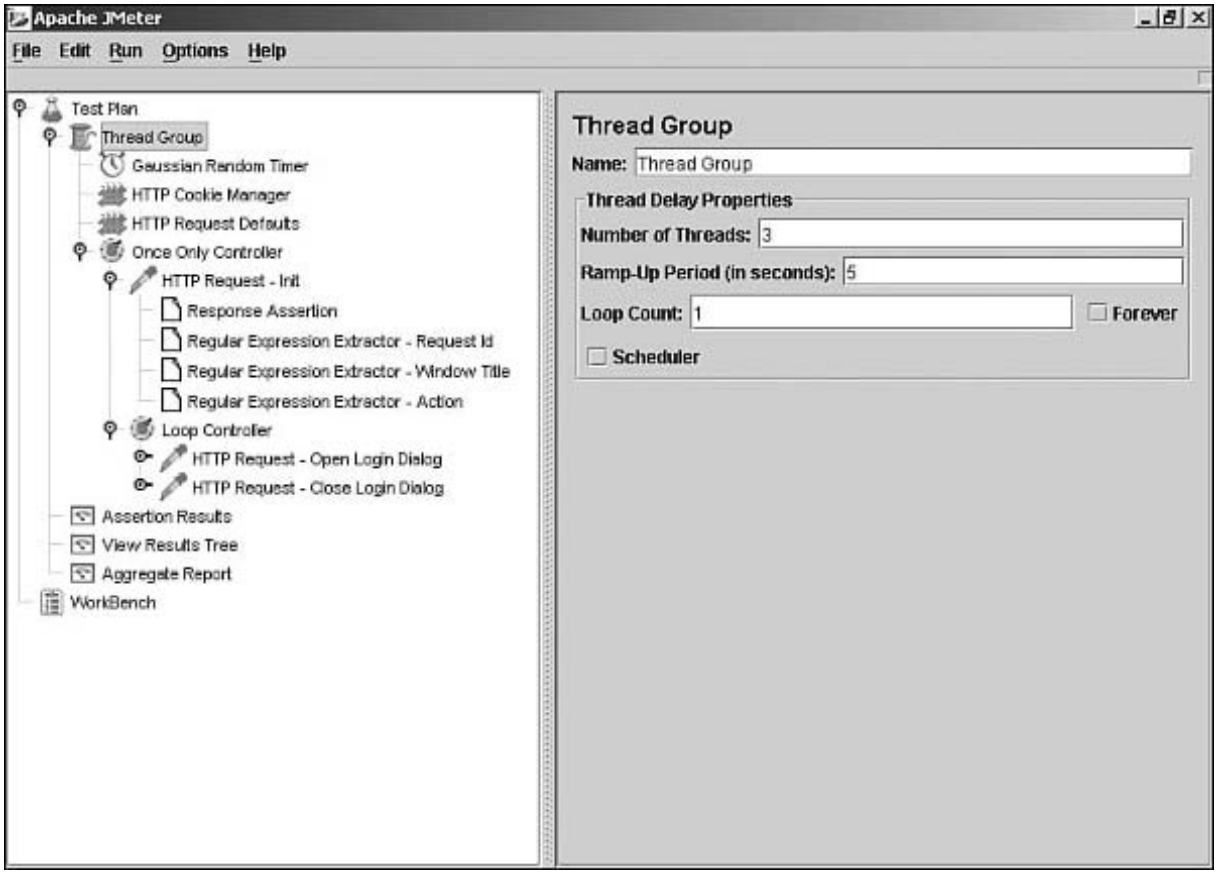

Рисунок 11.3. Дерево плана JMeter, шаг 1

Для краткости добавьте Once Only Controller группе потоков, чтобы объединить запросы семплера в поддерево и отделить от конфигурации. Затем добавьте контроллеру HTTP Request Sampler, назовите его HTTP Request - Init и, поскольку это начальный запрос, выберите GET, оставив остальное пустым. Сведения о сервере поступят из Request Defaults. Мы создали шаг, эквивалентный вводу:

```
http://localhost:8040/webcream/apps/WebCreamDemo
```

в адресную строку браузера и нажатию Enter. Сервер должен вернуть главную страницу. Проверим ответ, добавив Response Assertion узлу HTTP Request - Init. Утверждение может быть простой подстрокой или несколькими регулярными выражениями. Мы лишь проверим, что ответ - страница с заголовком WebCream Demo. При написании плана полезно держать соответствующую страницу открытой в браузере. Укажите, что ответ должен содержать:

```
<title>WebCream Demo</title>
```

Текущее дерево показано на рисунке 11.4.

Теперь перейдём к открытию и закрытию диалога. Это нужно повторить несколько раз, поэтому сначала добавьте Loop Controller внутрь Once Only Controller и задайте 5 повторов. Затем добавьте HTTP Request Sampler, имитирующий нажатие Login Dialog. Чтобы понять отправляемые данные, откройте исходный код главной страницы. Поиск `Login Dialog` приводит к листингу 11.4.

```
<form name="WebCreamForm" method="POST"
    action="http://localhost:8040/webcream/apps/WebCreamDemo"
    onSubmit="onSubmitForm()"
>
...
<input type=button
    name="JButton31266642"
    value="Login Dialog"
    class=button
    OnClick=javascript:doSubmit('/button/JButton31266642')
    style="position:absolute;left:84;top:5;width:103;height:26"
>
...
</form>
```

Листинг 11.4. Исходный HTML кнопки Login

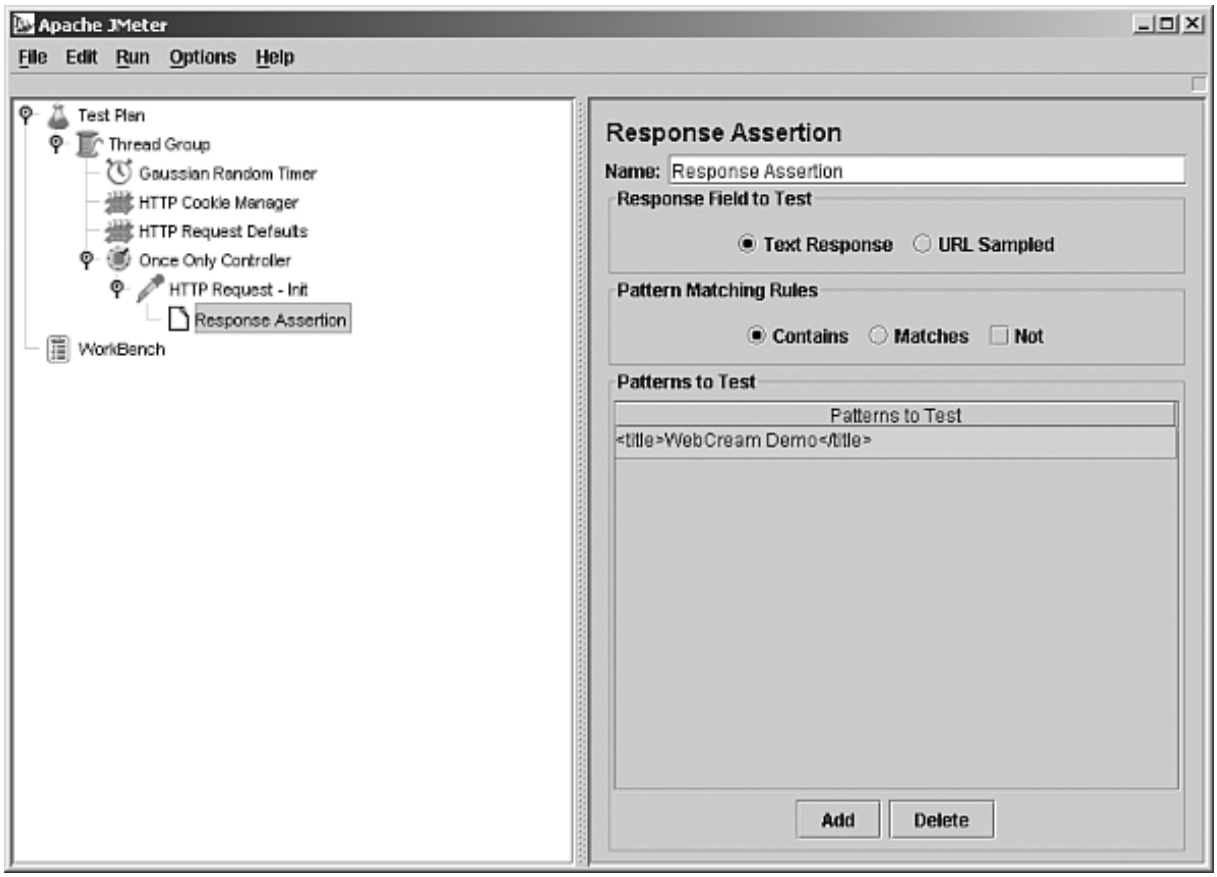

Рисунок 11.4. Дерево плана JMeter, шаг 2

Даже небольшое знакомство с HTML показывает, что кнопка вызывает JavaScript `doSubmit`, передавая `'/button/JButton31266642'`. Поиск `doSubmit` на странице и в подключённых файлах находит в `webcream_misc.js` следующий фрагмент:

```
function doSubmit(action) {
    window.document.WebCreamForm.__Action.value=action;
    onSubmitForm();
    window.document.WebCreamForm.submit();
}
```

Следовательно, нажатие Login Dialog задаёт `__Action` равным `/button/JButton31266642` и отправляет форму серверу. При нескольких обращениях через браузер видно, что строка действия меняется. Она всегда начинается с `/button/JButton`, но остальные цифры создаются автоматически. Динамическое содержимое есть у большинства веб-страниц, поэтому применяемый к WebCream приём полезен и для других приложений.

Для динамического содержимого JMeter нужны переменные и регулярные выражения. Regular Expression Extractor - постпроцессор, применяющий выражение к ответу семплера и помещающий извлечённое значение в переменную. Регулярные выражения - мощный механизм работы с текстом; если вы с ними незнакомы, рекомендую изучить. В Интернете множество справочников и учебников, а O'Reilly выпустила книгу *Mastering Regular Expressions*.

Инициализированную переменную можно передавать другим элементам, например семплерам. Извлечём три параметра HTTP Request Defaults: `__RequestId`, `__WindowTitle` и `__Action`. Щёлкните Http Request - Init правой кнопкой и добавьте Regular Expression Extractor, затем дополните его имя текстом - `Request Id`. В параметрах укажите `__RequestId` как Reference Name, чтобы результат сохранялся в одноимённой переменной. По HTML листинга 11.3 составим выражение:

```
name="__RequestId" value="(\d+)"
```

Статические символы однозначно находят определение, маска `\d+` означает любое число цифр, а скобки выделяют результирующую часть. Нам нужно только числовое значение, поэтому они окружают `\d+`.

Шаблон экстрактора создаёт строку из найденных совпадений и может содержать статический текст и `$n$` как место n-го совпадения. Мы ожидаем одно совпадение, поэтому укажем `$1$`; извлечение `__RequestId` готово.

Аналогично добавьте экстракторы для `__WindowTitle` и `__Action`. Самое время потренироваться и придумать выражения самостоятельно. На всякий случай облегчу задачу: для `__WindowTitle` подходит:

```
name="__WindowTitle" value="(.+)"
```

с шаблоном `$1$`. Для `__Action`:

```
name="(JButton\d+)" value="Login Dialog"
```

и шаблон `/button/$1$`. Главное - получить строку, пригодную как значение параметра следующего запроса. Текущее дерево показано на рисунке 11.5.

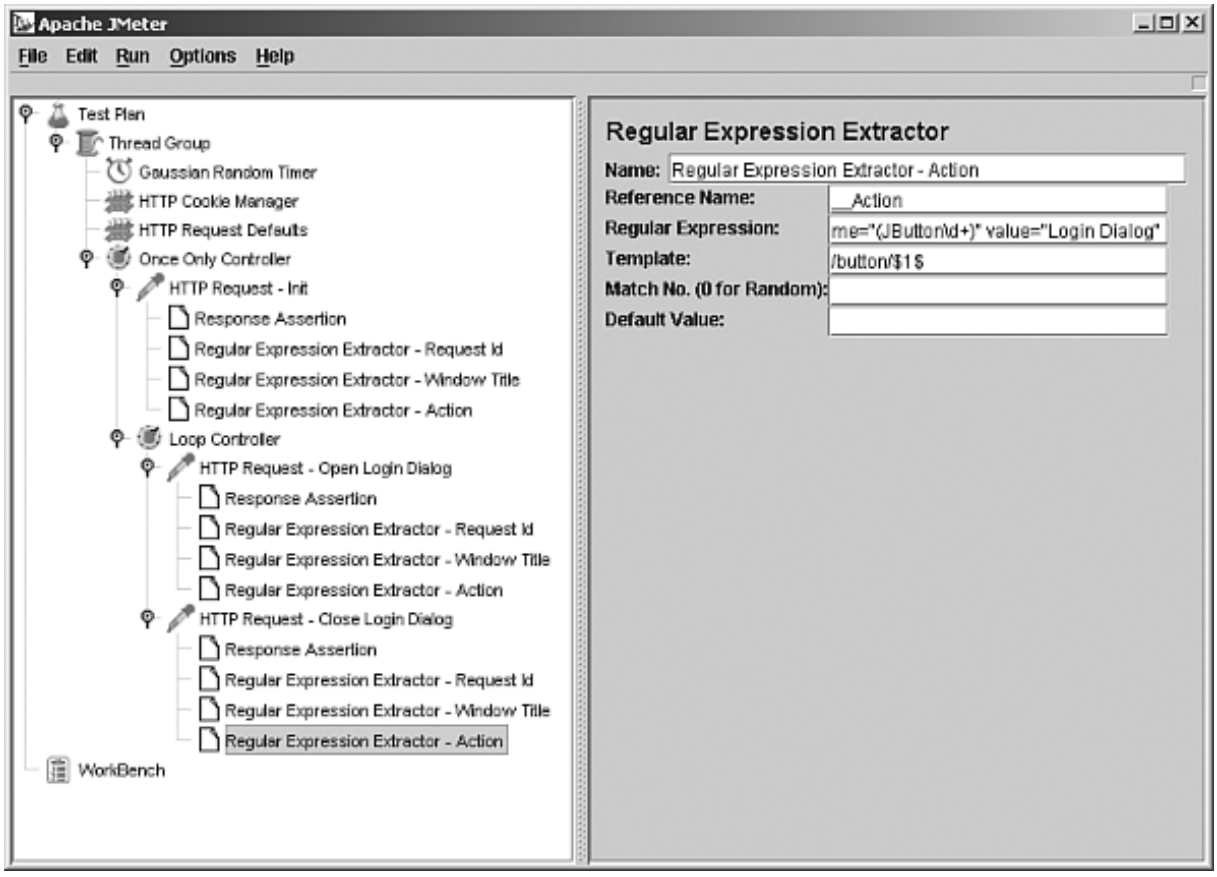

Рисунок 11.5. Дерево плана JMeter, шаг 3

Теперь можно использовать извлечённые переменные как параметры HTTP-запросов. Мы уже определили параметры в HTTP Request Defaults, поэтому вернёмся к нему и зададим значения. Значение переменной JMeter получают синтаксисом `${name}`. Укажите `${__RequestId}` для `__RequestId`, `${__WindowTitle}` для `__WindowTitle` и `${__Action}` для `__Action`.

Почти готово. Перед запуском добавим слушатели для наблюдения. При проверке сценария особенно полезны View Results Tree и Assertion Results. Первый показывает каждый запрос с данными запроса и ответа, второй быстро показывает неудачные утверждения. В окончательном сценарии View Results Tree оставлять не следует из-за накладных расходов на сбор и хранение всех данных. Наконец, добавьте Aggregate Report для сводки выполнения и показателей производительности каждого запроса.

Если все шаги выполнены правильно, план будет выглядеть как на рисунке 11.6.

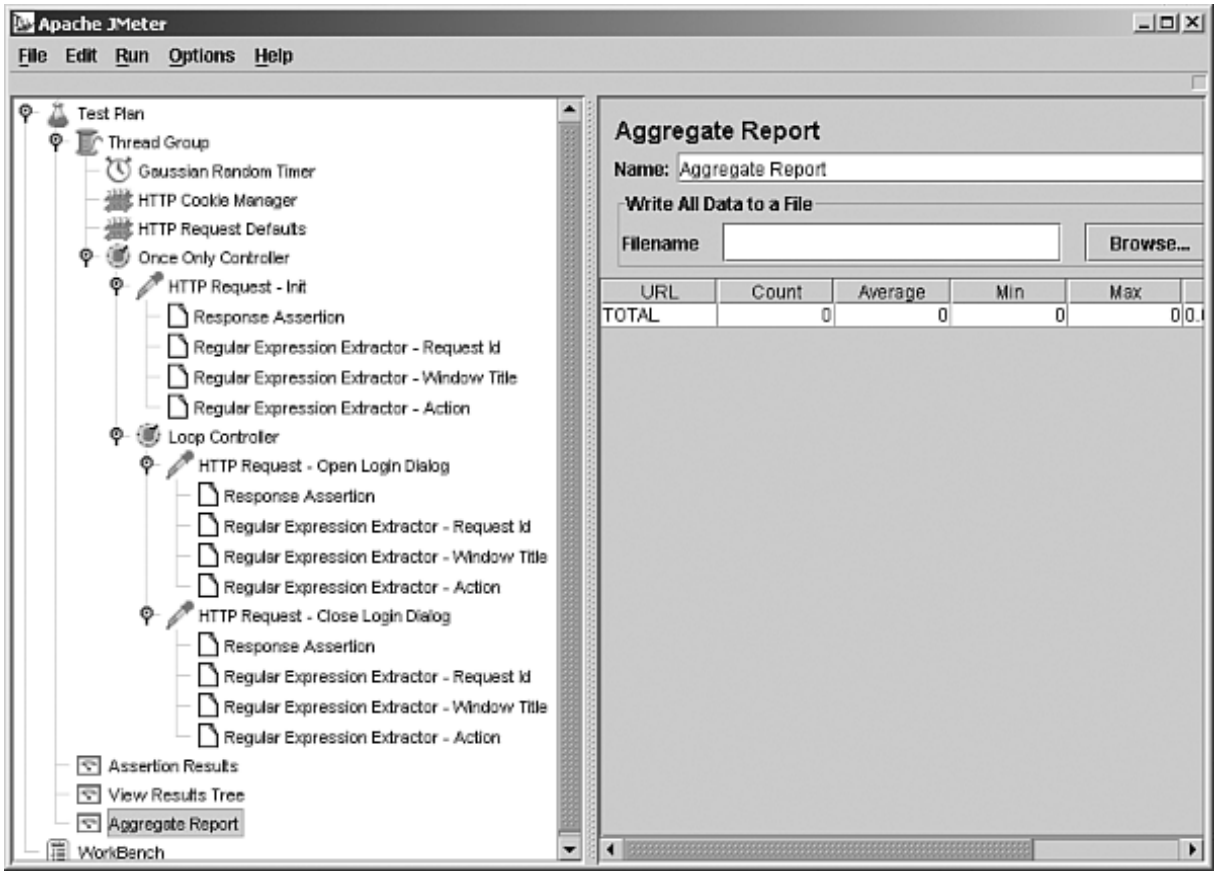

Рисунок 11.6. Завершённое дерево плана JMeter для WebCream

Убедитесь, что Tomcat WebCream работает, и обрушьте всю ярость JMeter, выбрав Start в меню Run. Через несколько минут тест завершится, а результаты появятся в узлах слушателей.

Короткий тест

1. Какова цель нагрузочного тестирования?
2. Чем отличаются одновременные и параллельные пользователи?
3. Какие клиентские протоколы можно тестировать с помощью JUnit?
4. Как тесты JUnit утверждают корректность ответа?
5. Какие клиентские протоколы можно тестировать с помощью JMeter?
6. Какие узлы конфигурации и семплеров используются в планах JMeter?
7. Как наблюдать за ходом и результатами выполняющегося плана JMeter?
8. Каковы преимущества и недостатки JUnit и JMeter?

Краткие итоги

• Цель нагрузочного тестирования - оценить соответствие производительности системы требованиям к уровню обслуживания под нагрузкой.
• Коммерческие и открытые продукты могут автоматически записывать виртуальных пользователей, программно создавать сценарии, поддерживать несколько языков и протоколов и создавать графики и отчёты.
• JUnit - открытый каркас, пригодный для простых нагрузочных испытаний серверов RMI или обычных серверов Java. Тест JUnit представляет собой компилируемый и выполняемый класс Java.
• JMeter - открытый инструмент нагрузочного тестирования и измерения производительности веб-сайтов и приложений Java. Он поддерживает HTTP и FTP, а также базы данных, сценарии Perl и объекты Java.
• JMeter поддерживает динамическое содержимое посредством экстракторов регулярных выражений и переменных.
• Слушатели JMeter создают отчёты о производительности и графики.

Глава 12. Обратная разработка приложений

> «Если бы строители строили здания так, как программисты пишут программы, первый же появившийся дятел уничтожил бы цивилизацию».
>
> *Законы Мерфи о технике*

В этой главе

• Элементы пользовательского интерфейса и ресурсы
• Взлом текста
• Взлом изображений
• Взлом файлов конфигурации
• Короткий тест
• Краткие итоги

Элементы пользовательского интерфейса и ресурсы

В главе 5, «Замена и исправление классов приложения», мы говорили об исправлении классов для изменения логики приложения. В этой главе мы обсудим изменение таких элементов пользовательского интерфейса, как сообщения, предупреждения, приглашения, изображения, значки, меню и цвета. Здесь применяются те же принципы, которые описаны в главе 5. Первая задача - найти ресурс, который нужно изменить. Затем можно внести изменение и обновить приложение, чтобы новый ресурс использовался вместо старого. В таблице 12.1 показано соответствие между элементами пользовательского интерфейса и ресурсами, которые необходимо исправлять.

Тип элемента	Примеры элементов	Ресурс Java
Текст	Сообщение диалогового окна, предупреждение, текст исключения	Строка Java внутри файла класса или текст в файле конфигурации
Изображение	Заставка, значок, фоновое изображение	Файлы JPG или GIF (либо файлы других форматов изображений) в каталоге приложения или внутри файла JAR
Визуальный элемент	Компоновка окна, состав меню, цвет	Файл класса Java, содержащий соответствующий элемент
Параметр конфигурации	Программные ограничения, например максимальное число соединений, срок действия, число потоков	Файл класса Java, если значение жёстко задано, файлы `.properties` или файлы `.xml`, если значение настраивается
Звук	Фоновая музыка, звуковой эффект	Файлы WAV, AIFF или AU, представляющие звуковой фрагмент

Таблица 12.1. Соответствия между элементами пользовательского интерфейса и ресурсами

Обучение на примере - один из самых простых способов усвоить сведения, поэтому возьмём гипотетическое приложение, а затем переделаем его по собственной прихоти. Во многих предыдущих главах вам уже встречалось приложение Chat, и, продолжая добрую традицию, для иллюстрации каждого приёма мы будем взламывать Chat.

Взлом текста

В Chat текст `Send message` используется и как текст пункта меню, и как подсказка ToolTip для кнопки панели инструментов. Хотя он превосходно передаёт смысл действия, молодое поколение пользователей может счесть его немодным. Чтобы угодить этой аудитории, взломаем Chat так, чтобы вместо `Send message` использовался текст `Unleash the fury`. Мы будем работать с распространяемой версией приложения, делая вид, будто исходный код нам недоступен.

Поскольку Chat поставляется в файле JAR, прежде всего нужно распаковать `CovertJava/distrib/lib/chat.jar` командой `jar` во временный рабочий каталог. Затем необходимо найти классы, задающие текстовые строки, которые сейчас отображает Chat. С помощью FAR или средства поиска выполняем двоичный поиск текста `Send message` во всех файлах рабочего каталога и его подкаталогов. Результатом поиска должен стать файл `MainFrame.class`, который мы декомпилируем, как описано в главе 2, «Декомпиляция классов». Поиск текста `Send message` в декомпилированном исходном коде приводит к методу `jbInit`, фрагмент которого показан в листинге 12.1.

```
private void jbInit()
    throws Exception
{
    ...
    btnSend.setToolTipText("Send message");
    btnHelp.setIcon(imageHelp);
    ...
    menuEditSend.setText("Send message");
    menuEditSend.addActionListener(this);
    ...
}
```

Листинг 12.1. Фрагмент `jbInit`*, показывающий текстовые строки*

Применяя приём исправления, представленный в главе 5, можно просто заменить `Send message` на `Unleash the fury` в декомпилированном исходном коде и добавить его в Chat как исправление. Это было достаточно просто, и, к счастью, в большинстве реальных приложений всё должно быть столь же просто. Сложности могут возникнуть, если код приложения обфусцирован с кодированием строк, но знания, полученные в предыдущих главах, не позволят этому помешать нам найти класс для исправления.

Хорошие приложения не задают текстовые строки в коде жёстко. Вместо этого сообщения хранятся в пакетах ресурсов или файлах конфигурации, что упрощает сопровождение и локализацию. Если строки не закодированы, двоичный поиск по содержимому рабочего каталога всё равно даст нужные результаты. Найдя файл, содержащий строку, его можно обновить новым текстом. После этого приложение необходимо упаковать заново, чтобы изменение вступило в силу.

Взлом изображений

Работать с изображениями немного сложнее, чем с текстом. В отличие от текста, изображение нельзя просто найти, поскольку неизвестно, что использовать в качестве критерия поиска. Поэтому первый шаг - выяснить имя изображения и определить, как приложение его загружает, после чего можно применить исправление. Прямого и простого способа найти имя изображения нет. Вероятнее всего, приложение загружает его в коде, относящемся к пользовательскому интерфейсу, но разработчики нередко используют каркас или класс-диспетчер, который выполняет подготовительную работу, необходимую для загрузки изображения в Java. Опыт подсказывает изучить развёрнутое приложение (то есть не хранящееся в JAR) или, возможно, найти все файлы `.jpg` и `.gif`. Чаще всего имя изображения лучше всего подсказывает, где оно используется. Например, заставка может называться примерно `splash.gif`, а значок кнопки «Создать» на панели инструментов - `new.gif`. В обычном приложении Java изображений не так много, поэтому нужное должно найтись без труда. Очевидно, что просмотр изображения подтвердит, того ли кандидата вы нашли. (Мне ведь не обязательно было это говорить, правда?)

А что, если изображений слишком много, а программист, написавший приложение, использовал такие имена файлов, как `img0045.gif`? По размеру всё ещё можно догадаться, какое изображение вам нужно, либо просто последовательно просмотреть их все. Помните, что приложение может хранить изображения в другом файле JAR, поэтому при поиске нужно открыть их все.

Теперь рассмотрим худший случай: вы видите, что приложение отображает изображение, и отчаянно хотите его изменить, но, просмотрев все упакованные с приложением графические файлы, так его и не находите. Определить, откуда оно берётся, можно несколькими способами. Поскольку простой путь не сработал, придётся копнуть глубже и добраться до места в коде, где загружается изображение. Приёмы обратной разработки уже рассматривались в предыдущих главах, а в качестве отправной точки я предлагаю искать в файлах классов строки `.gif` или `.jpg`. Я никогда не видел приложения, которое хранит файлы GIF с расширением `.txt`, и искренне надеюсь никогда не дожить до такого дня. Например, чтобы изменить изображение в диалоговом окне «О программе» приложения Chat, можно найти в рабочем каталоге все файлы GIF и JPG. Это приведёт нас в каталог `covertjava.chat.images`, содержащий несколько изображений. Поскольку их там совсем немного, можно просто просмотреть файлы и определить, какой из них используется в диалоговом окне «О программе». Либо можно было искать строку `.gif`, что снова привело бы к файлу класса `MainFrame`. Декомпилировав его, мы обнаружили бы, что для диалогового окна «О программе» Chat использует `saturn_small.gif`.

Наконец, можно искать вызов метода `getImage` класса `java.awt.Toolkit`, поскольку это самый распространённый способ загрузки изображения. Ниже приведена пара примеров кода Java, загружающего изображение:

```
/**
 * Creates an icon from an image contained in "resources/images"
 * directory inside a .jar file
 */
public ImageIcon createImageIcon(String fileName) throws Exception
{
    String path = "/resources/images/" + fileName;
    return new ImageIcon(getClass().getResource(path));
}

/**
 * Loads an image from "images" directory on disk
 */
public Image loadImage(String fileName) throws Exception
{
    return Toolkit.getToolkit().getImage("images/" + filename);
}
```

Найдя код, загружающий изображение, вы должны суметь понять, откуда оно берётся. Если изображение не упаковано вместе с приложением, скорее всего, доступ к нему осуществляется по внешнему URL наподобие mycompany.com. Теперь вы, образованный хакер, точно знаете, как подступиться к задаче. Загрузите изображение с этого URL с помощью веб-браузера, сохраните его локально, отредактируйте и упакуйте с приложением. Затем исправьте загружающий его класс и вместо URL с HTTP используйте либо URL `jar:file/` для загрузки из своего JAR, либо `Toolkit.getImage()` для загрузки из каталога. Вуаля - готово.

Взлом файлов конфигурации

Файлы конфигурации - всего лишь ещё один вид ресурсов приложения, и подход к их взлому такой же, как для текста и изображений. Речь не о хранящихся в каталоге приложения конфигурационных файлах, которые легко изменить, а о файлах, которые могут находиться внутри файла `.jar` или `.zip` и не обязательно предназначены для редактирования. Как и прежде, распакуем архив приложения командой `jar` и изучим структуру каталогов в поисках подозрительных файлов. Если структура каталогов велика, можно найти все файлы с расширением `.properties` или `.xml`. Скорее всего, такой простой просмотр даст результат. Если нет, можно вернуться к описанным в предыдущих главах приёмам обратной разработки, чтобы найти участок кода, использующий этот параметр. Затем можно проследить путь до кода, загружающего конфигурацию, и получить достаточно сведений для выбора стратегии исправления.

Короткий тест

1. Предположим, вы используете библиотеку, которая при перехвате обобщённого исключения выводит сообщение `Unknown error`. Как заменить его текстом `Internal error, please contact technical support`?
2. Как изменить диалоговое окно «О программе» приложения Chat, чтобы на изображении было ваше имя?
3. Как выяснить, можно ли настроить установленное используемым вами приложением максимальное число одновременных соединений?

Краткие итоги

• Для взлома элементов пользовательского интерфейса необходимо найти и исправить ресурсы, из которых они были созданы.
• Для взлома текста необходимо найти файл CLASS или файл конфигурации (`.properties`, `.xml` либо `.resource`), в котором он задан, и изменить этот файл.
• Для взлома изображений необходимо найти файл изображения (обычно GIF или JPG).
• Взлом файлов конфигурации сводится к простому редактированию текста после того, как найден файл, задающий интересующие свойства.

Глава 13. Методы подслушивания

> «Можно многое заметить, просто наблюдая».
>
> *Первый закон Берры*

В этой главе

• Определение подслушивания
• Подслушивание HTTP
• Подслушивание протокола RMI
• Подслушивание драйвера JDBC и операторов SQL
• Короткий тест
• Краткие итоги

Определение подслушивания

Предыдущие главы были в основном посвящены работе с байт-кодом и ресурсами приложения. Многоуровневые приложения, господствующие на стороне сервера, дают дополнительные направления для обратной разработки и взлома. Уровни приложения обычно развёртывают как отдельные процессы, обменивающиеся данными по сетевым протоколам. Например, веб-браузер, отображающий интерфейс HTML на рабочей станции пользователя, взаимодействует с веб-сервером по протоколу передачи гипертекста (HTTP). В свою очередь веб-сервер обычно взаимодействует с сервером приложений посредством RMI или IIOP. Сервер приложений использует JDBC для взаимодействия с базой данных. Классическая схема развёртывания распределённых многоуровневых приложений Java показана на рисунке 13.1.

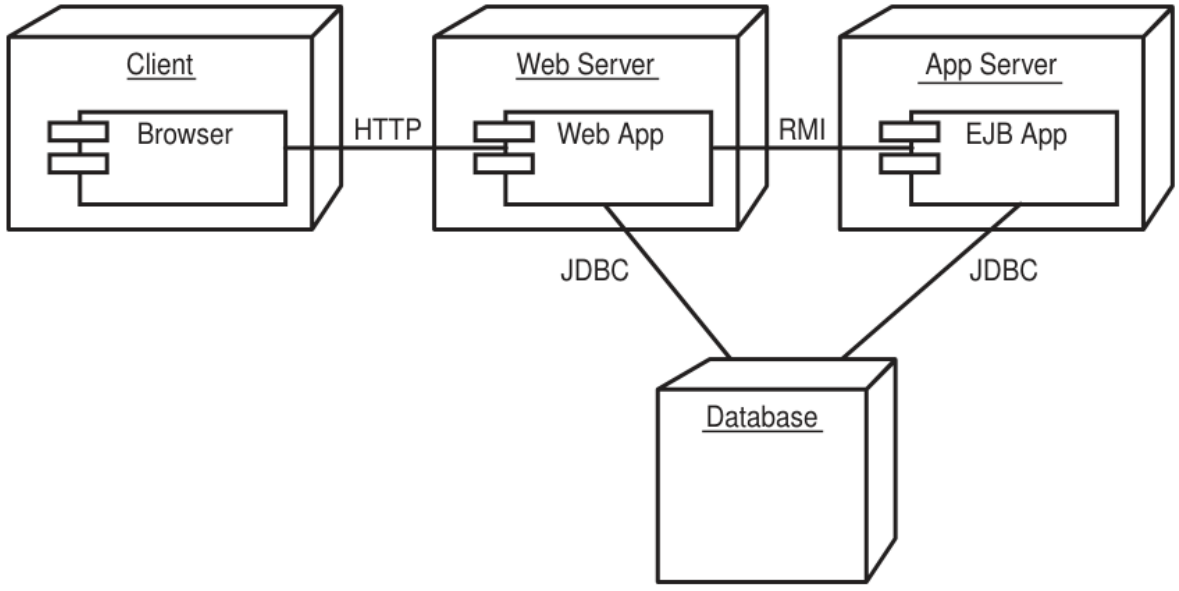

Рисунок 13.1. Схема развёртывания многоуровневого приложения

В этой главе представлены несколько приёмов подслушивания разговора между распределёнными уровнями. Подслушивание - это перехват и регистрация обмена сообщениями между клиентом и сервером. Оно может помочь в устранении неполадок или настройке производительности сложной распределённой системы, а также позволяет понять устройство приложения и принципы его взаимодействия.

Подслушивание HTTP

Сначала рассмотрим подслушивание веб-приложений и веб-служб, использующих HTTP в качестве транспортного протокола для взаимодействия со своими клиентами. Сообщения HTTP состоят из заголовка и содержимого, передаваемого по сети через TCP/IP. Чтобы клиент мог разговаривать с сервером, он должен знать имя узла или IP-адрес сервера и порт, который тот прослушивает. Сообщения HTTP передаются как обычный текст, который человек легко читает и понимает.

Чтобы подслушать обмен между клиентом и сервером, необходимо перехватить их сообщения. Один из способов - поместить между клиентом и сервером посредника, который трассирует и туннелирует сообщения HTTP. Другой способ - следить за взаимодействием на уровне сетевого протокола, отбирая сообщения, которыми обмениваются клиент и сервер. В следующих разделах мы рассмотрим оба варианта. Серверным приложением вновь станет WebCream, поскольку оно предоставляет статическое и динамическое содержимое HTML посредством сервлетов и страниц JSP.

Использование туннеля для захвата обмена сообщениями HTTP

В данном контексте туннель - это псевдосервер, помещённый между настоящими клиентом и сервером для перехвата и трассировки обмена сообщениями. Вместо прямого разговора с сервером клиент перенастраивается так, чтобы отправлять сообщения туннелю; туннель настраивается на передачу запросов серверу от имени клиента и пересылку ответов сервера обратно клиенту. Туннель записывает передаваемые сообщения на экран или в файл, позволяя хакеру либо разработчику изучить подробности разговора. Текстовая природа HTTP и отсутствие в нём состояния делают этот протокол хорошим кандидатом для туннелирования.

Для туннелирования запросов браузера к серверу Tomcat приложения WebCream мы воспользуемся утилитой TCPMON, распространяемой с проектом Apache AXIS. Хотя TCPMON далеко не самая сложная программа туннелирования, она бесплатна и хорошо справляется с большинством задач. Загрузите AXIS с веб-сайта Apache и, если у вас нет сценария запуска TCPMON, скопируйте подкаталог `CovertJava\bin\axis` в каталог, где установлен AXIS. TCPMON принимает три параметра командной строки: прослушиваемый порт, узел и номер порта сервера, которому нужно передавать запросы. Поскольку Tomcat приложения WebCream работает на порте 8040, запустим TCPMON на порте 8000 и укажем ему пересылать данные на порт 8040 узла `localhost`.

Чтобы WebCream поддерживал туннелирование через AXIS, необходимо внести несколько простых изменений в конфигурацию. WebCream создаёт формы страниц с полностью определёнными URL для отправки, включающими имя узла и номер порта. Мы хотим, чтобы весь трафик проходил через TCPMON, поэтому необходимо заставить WebCream всегда отправлять формы на порт 8000, а не на порт 8040, используемый по умолчанию. Для этого добавьте следующие две строки в `WebCreamconf\WebCreamDemo.properties` (дополнительные сведения см. в документации WebCream):

```
html.docsURL=http://localhost:8040/webcream
html.submitURL=http://localhost:8000/webcream/apps/WebCreamDemo
```

Запустите Tomcat приложения WebCream с помощью `WebCream\bin\startServer.bat`. Откройте любимый веб-браузер и введите в адресной строке:

```
http://localhost:8000/webcream/apps/WebCreamDemo
```

Обратите внимание: мы используем порт 8000, на котором работает TCPMON, а не направляем браузер непосредственно к Tomcat, работающему на порте 8040.

Браузер должен отобразить главную страницу демонстрационного приложения WebCream. Переключитесь в TCPMON и убедитесь, что перехваченный запрос виден на верхней (или левой) панели. TCPMON может довольно медленно определять, что запрос передан полностью, поэтому дождитесь, пока состояние запроса на верхней панели изменится на `Done`. Вернитесь в браузер и нажмите кнопку `Login Dialog`, чтобы открыть следующую страницу; затем введите `Neo` как имя пользователя и `Wakeup` как пароль и нажмите `OK`. (Впрочем, вы можете выбрать собственные имя пользователя и пароль.) После переключения на экран TCPMON он должен выглядеть примерно так, как показано на рисунке 13.2.

Теперь можно изучить запросы и данные, перехваченные TCPMON. На верхней панели показаны три сообщения, что соответствует числу страниц, запрошенных браузером. Самые интересные сведения находятся на нижних левой и правой панелях. Левая панель показывает заголовок и данные запроса, а правая - заголовок и содержимое ответа. Изучение различных атрибутов и элементов содержимого позволяет понять обмен данными между браузером и сервером. Например, выбрав третий запрос на верхней панели и просмотрев его данные, можно увидеть настоящие значения имени входа (`Neo`) и пароля (`Wakeup`). В заголовке виден файл cookie `JSESSIONID`, который можно использовать для захвата сеанса пользователя.

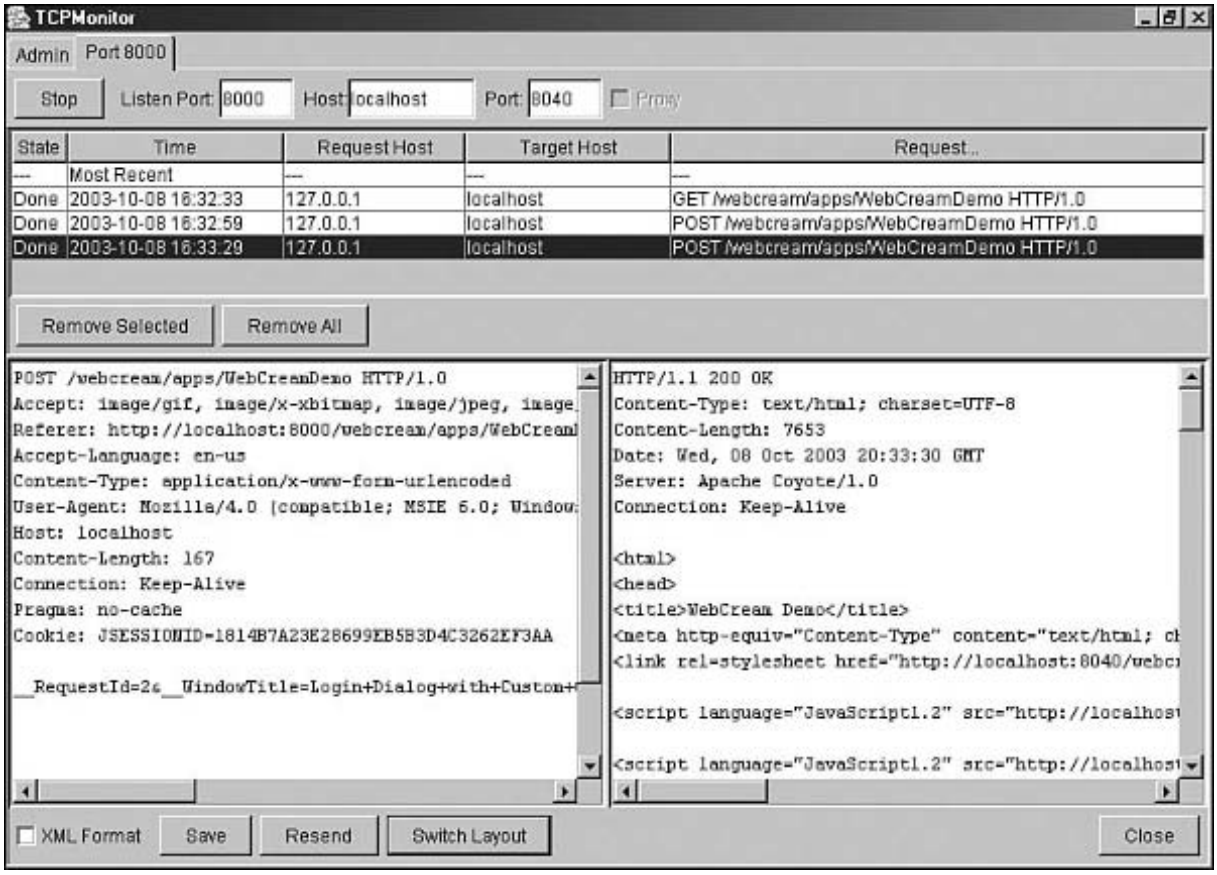

Рисунок 13.2. TCPMON показывает перехваченные запросы

Использование сетевого анализатора для захвата обмена сообщениями HTTP

Туннелирование - простой и действенный способ увидеть, что в действительности передаётся между клиентом и сервером. Оно хорошо подходит для отладки и изучения веб-приложений, но требует перенастроить клиент (а иногда даже сервер, как пришлось сделать с WebCream), чтобы тот отправлял запросы агенту туннелирования, а не непосредственно серверу. Второй приём позволяет подслушивать на уровне сетевого протокола и лишён ограничений туннелирования.

Любое распределённое взаимодействие проходит через слой протоколов, поддерживаемых JVM, операционной системой и сетевым драйвером. Протоколы образуют стек: протоколы более высокого уровня полагаются на протоколы более низкого уровня при выполнении основных задач. Так, HTTP опирается на TCP, который, в свою очередь, опирается на IP. Одно сообщение HTTP может быть представлено несколькими низкоуровневыми пакетами IP, а сетевые уровни должны разбирать, а затем вновь собирать эти пакеты. Большинство физических сетей состоит из взаимосвязанных сегментов Ethernet с рабочими станциями. Внутри сегмента пакеты одного узла передаются всем узлам независимо от адреса узла назначения. Для связи с компьютером вне сегмента маршрутизаторы перенаправляют пакеты в другие сегменты. Суть этого грубого обзора в следующем: когда приложение на одном узле взаимодействует с удалённым приложением на другом узле, пакеты протокола до достижения цели должны пройти через ряд других сетевых узлов.

Сетевое прослушивание и наблюдение используют этот принцип, чтобы шпионить за обменом данными. Обычно сетевой узел принимает только предназначенные ему пакеты, игнорируя все остальные. Однако узел может работать в неразборчивом режиме, принимая все пакеты независимо от адреса назначения. Тогда инженер или хакер может изучать пакеты и их содержимое и получать представление о взаимодействии приложений.

Работа на уровне протокола может быть трудной и долгой. Поскольку HTTP очень распространён, существуют продукты, упрощающие подслушивание обмена HTTP. Мы рассмотрим HTTP Sniffer компании EffeTech, условно-бесплатное приложение для Windows. Попытаемся выполнить ту же задачу перехвата взаимодействия браузера с Tomcat приложения WebCream и посмотрим, отличается ли результат от TCPMON. Один из недостатков сетевого анализатора состоит в том, что он не работает, когда клиент и сервер выполняются на одном узле. Пакеты не передаются сетевому драйверу, поэтому анализатор не способен захватывать запросы и ответы протокола. Таким образом, для этого и следующего раздела необходимо обеспечить работу сервера Tomcat на другом сетевом узле, не там, где работает браузер. Если другой компьютер вам недоступен, вместо WebCream можно использовать любой веб-сайт.

Загрузите и установите HTTP Sniffer от EffeTech. Он устанавливает WinPcap - библиотеку захвата пакетов, которая добавляется в Windows как драйвер устройства для получения необработанных данных с сетевой платы. Поскольку Tomcat приложения WebCream работает на порте 8040, этот порт необходимо добавить в фильтр анализатора с помощью пункта `Filter` меню `Sniffer`. Если запись не даст результатов, возможно, придётся сменить текущий сетевой адаптер анализатора, но пока оставьте значение по умолчанию. Запустите Tomcat приложения WebCream на удалённом сервере, а затем откройте веб-браузер. Обратите внимание: анализатор может работать на одном компьютере с браузером, на одном компьютере с веб-сервером или на любом компьютере, подключённом к тому же сегменту Ethernet, что и узлы браузера либо сервера. Начните запись с помощью меню `Sniffer` или панели инструментов и введите URL сервера в браузере. При проверке с WebCream откройте главный экран, нажмите кнопку `Login Dialog`, чтобы перейти на следующую страницу, введите `Neo` как имя пользователя и `Wakeup` как пароль, затем нажмите `OK`. После остановки записи в HTTP Sniffer его экран должен выглядеть так, как показано на рисунке 13.3.

Способ представления и тип перехваченных сведений почти такие же, как в TCPMON. Помимо более опрятного пользовательского интерфейса, HTTP Sniffer показывает другие полученные браузером с сервера ресурсы, например файлы GIF и JavaScript. И снова, изучив данные запроса, можно получить значения параметров формы, в том числе имя пользователя и пароль. Прелесть этого подхода в том, что нам не пришлось ничего делать с целевым приложением, но мы захватили полный журнал взаимодействия клиента и сервера.

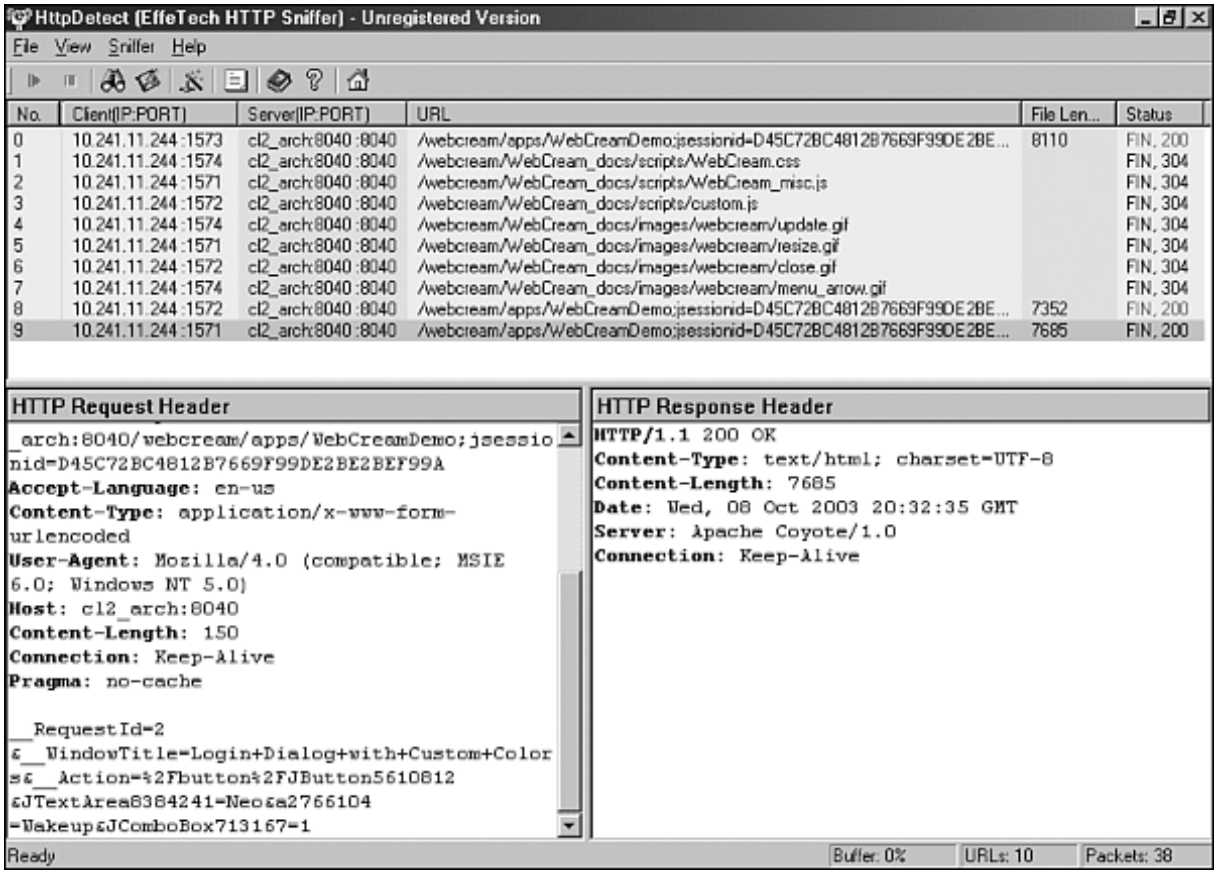

Рисунок 13.3. HTTP Sniffer показывает перехваченные запросы

Защита веб-приложений от подслушивания

За короткое время вы научились подслушивать пользовательские интерфейсы на основе браузера. Лёгкость, с которой можно получить значения потенциально конфиденциальных параметров, довольно тревожна. Несколько предосторожностей способны значительно осложнить взлом веб-приложения. Самый простой и действенный способ защитить целостность данных и обезопасить взаимодействие клиента и сервера - использовать защищённый протокол передачи гипертекста (HTTPS). Этот протокол требует, чтобы до отправки любых данных HTTP клиент установил с сервером защищённый канал посредством протокола защищённых сокетов (SSL). После установления канала обмен сообщениями происходит так же, как при HTTP. Преимущество HTTPS в том, что все данные зашифрованы, поэтому даже при перехвате сетевого пакета расшифровать его практически невозможно. SSL создаёт дополнительную нагрузку, способную замедлить сервер, поэтому для высокопроизводительных приложений весь обмен по HTTPS может оказаться непрактичным. Хороший компромисс - применять HTTPS выборочно или продолжать использовать HTTP, но шифровать конфиденциальное содержимое на уровне приложения. Если пользовательским интерфейсом служит веб-браузер, шифрование и расшифрование на стороне клиента можно выполнять функциями JavaScript.

Подслушивание протокола RMI

Удалённый вызов методов Java (JRMI) использует либо специфичный для Java протокол удалённого вызова методов Java (JRMP), либо межброкерный интернет-протокол (IIOP), чтобы отправлять двоичные сообщения удалённым узлам. JRMP и IIOP используют для передачи сообщений по сети протокол управления передачей и интернет-протокол (TCP/IP), поэтому канал связи можно прослушивать по сети. Теоретически можно написать туннель, который получает и регистрирует сообщения, прежде чем передать их предполагаемому получателю, но это была бы очень утомительная работа. Поэтому для подслушивания RMI мы воспользуемся сетевым анализатором. К сожалению, ни один инструмент не имеет встроенной поддержки высокоуровневых протоколов Java, подобной поддержке HTTP в HTTP Sniffer. Значит, придётся работать на более низком уровне, изучая пакеты TCP и IP, представляющие вызовы RMI целиком или частично. Потренируемся шпионить за разговором между пользователями двух приложений Chat.

Транспортный протокол RMI

Для представления передаваемого формата RMI использует понятие потока. Внутри взаимодействия с ним связаны два потока - выходной и входной. Они сопоставляются соответствующим потокам сокета и применяются для отправки и подтверждения сообщений. Выходной поток состоит из заголовка и следующей за ним последовательности сообщений. Если используется HTTP, выходной поток вкладывается в сообщение HTTP. Заголовок содержит обозначение протокола и атрибуты, описывающие тип используемого протокола. Суть вызова RMI находится в разделе сообщений. Выходными сообщениями могут быть вызовы методов, удалённые проверки связи или трафик сборки мусора; входными - результат вызова либо подтверждение проверки связи.

Для выполнения удалённого вызова RMI использует протокол сериализации объектов Java, который преобразует имя метода и значения параметров в двоичную структуру, передаваемую по сети. Поэтому на двоичном уровне все удалённые вызовы имеют одинаковый формат. Для подслушивания RMI знать особенности транспорта не обязательно, но дополнительные сведения доступны по адресу java.sun.com. Для нашей задачи достаточно знать, что при обмене сообщениями через RMI между двумя приложениями Java создаётся и используется низкоуровневое соединение TCP/IP с входным и выходным потоками.

Использование сетевого анализатора для перехвата сообщений RMI

Один из самых популярных инструментов сетевого анализа - Ethereal. Он бесплатен и перенесён на Unix и Windows. Он способен анализировать почти любой мыслимый протокол (кроме RMI), и, хотя его пользовательский интерфейс несколько груб, за отсутствием лучшего инструмента он нам подойдёт. Ethereal использует WinPcap - ту же библиотеку, с помощью которой HTTP Sniffer захватывает сетевые пакеты. Загрузите, установите и запустите Ethereal. Начните запись сетевого трафика, просто чтобы составить представление о сведениях, которые он способен захватить. Откройте браузер и посетите несколько веб-сайтов; затем остановите запись и посмотрите на элементы верхней панели. Вы увидите несколько пакетов. Наша первая задача - настроить инструмент так, чтобы он показывал только интересующие нас сведения.

Начнём с открытия диалогового окна `Capture Options`, которое появляется после выбора `Start` в меню `Capture`. Снимите флажок `Capture Packets in Promiscuous Mode`, установите `Update List of Packets in Real Time` и убедитесь, что все флажки `Stop Capture After` сняты. Поскольку сетевое прослушивание требует, чтобы клиент и сервер работали на разных узлах, запустите Chat на двух компьютерах. Нажмите `OK`, чтобы подтвердить параметры захвата и начать запись. Затем переключитесь в Chat и отправьте другому узлу сообщение `Hi Alex`. (Я не обижусь, если вместо моего имени вы укажете своё.) Отправьте ещё одно сообщение и запомните его текст (моё второе сообщение - `IT jobs are becoming boring`); он понадобится позже. Затем вернитесь в Ethereal и остановите захват.

Мы захватили множество пакетов, и, чтобы увидеть разговор в Chat, нужен способ отобрать пакеты, несущие фрагменты отправленных пользователями текстовых сообщений. Для начала можно найти пакет со строкой, которая, как нам известно, была отправлена во время записи. Выберите `Find Frame` в меню `Edit`, чтобы открыть диалоговое окно `Search`. Введите часть отправленного сообщения (я бы ввёл `Alex`) и обязательно выберите вариант `String` в группе переключателей `Find Syntax`. Тем самым мы указываем Ethereal искать строку в любой части пакета. Нажмите `OK`, после чего должен быть выбран пакет, содержащий эту строку. На нижних панелях должно отображаться содержимое пакета, в котором среди двоичных данных должна присутствовать искомая строка. Поскольку один вызов RMI может быть разбит на несколько сообщений TCP/IP, обычно полезно применить функцию `Follow TCP Stream`, которая вновь собирает поток и показывает его в отдельном окне. Щёлкните выбранный кадр пакета правой кнопкой мыши и выберите `Follow TCP Stream`.

Теперь перед нами двоичная версия пакета RMI. Она выглядит несколько загадочно, но с небольшим терпением из неё можно извлечь кое-какие сведения. Пакет начинается заголовком `JRMI`, указывающим, что для транспорта RMI используется протокол JRMI. Далее следуют IP-адрес исходного узла, идентификатор объекта сервера и различные сведения распределённой сборки мусора (DGC). Содержимое сообщения находится в самом конце. В моём случае видна строка `Hi Alex`, за которой следует `covertjava.chat.MessageInfo`; мы предполагаем, что это имя класса сообщения. После неё идут другие параметры сообщения, например имя узла и имя пользователя.

Изучив поток TCP/IP, представляющий отправленное Chat сообщение RMI, можно предположить, что последующие сообщения будут иметь тот же формат. Иными словами, хотя текст сообщения может измениться, заголовок и формат сообщения останутся прежними. Если предположение верно, сообщения Chat можно искать по подстроке из заголовка или формата сообщения. Чтобы проверить эту теорию, попробуем найти строку `covertjava.chat.MessageInfo` во всех кадрах с помощью диалогового окна `Find Frame`. Если внизу экрана задана строка фильтра, сбросьте фильтр. Затем перейдите к первому кадру и выполните поиск. Первый найденный мной кадр содержал строку `Hi Alex`, а после поиска следующего кадра я обнаружил сообщение с текстом

`IT jobs are becoming boring`. Это подтверждает наше предположение о Chat и позволяет следить за разговором пользователей.

Подобный подход применим и к другим приложениям. Ethereal работает на очень низком уровне, но при должном усердии и небольшом анализе можно подслушивать сетевое взаимодействие любых приложений.

Защита приложений RMI от подслушивания

Из полученных знаний о сетевом прослушивании можно заключить, что полностью предотвратить подслушивание сетевого взаимодействия невозможно. В конце концов, данные должны передаваться по проводам и проходить через несколько сетевых узлов, а значит, могут быть перехвачены. Единственный способ защитить данные - зашифровать их. SSL предоставляет функции промышленного уровня для защиты сетевых каналов и может применяться при взаимодействии RMI. Расширение защищённых сокетов Java (JSSE) представляет собой набор API, позволяющих приложениям Java использовать SSL для шифрования данных, аутентификации и обеспечения целостности сообщений. RMI разрешает приложениям предоставлять для экспортируемых объектов собственные фабрики сокетов, используемые вместо стандартных сокетов TCP/IP. Применение фабрик SSL из JSSE как собственных фабрик сокетов и для клиента, и для сервера эффективно защищает канал. Пример сочетания RMI и SSL приведён на веб-сайте JavaSoft по адресу:

java.sun.com

Взаимодействие через SSL создаёт дополнительную нагрузку, которая может быть излишней для всех возможных видов обмена данными между клиентом и сервером. Возможно, лучше продолжить использовать стандартные фабрики сокетов, но шифровать конфиденциальные части данных средствами JSSE. Пример применения шифрования Java показан в главе 19, «Защита коммерческих приложений от взлома».

Подслушивание драйвера JDBC и операторов SQL

Большинство серверных приложений использует базу данных для хранения и получения данных. Знание того, как приложение взаимодействует с базой данных и какие операторы SQL применяет, может значительно помочь в настройке производительности или обратной разработке. Безусловно преобладающей технологией хранения данных является JDBC, и в ближайшее время это не изменится. Чтобы использовать JDBC, приложение должно определить и загрузить драйвер, получить соединение, а затем выполнять операторы для обновлений и запросов. С точки зрения настройки производительности и обратной разработки наиболее интересны структура и параметры операторов SQL. Общеизвестно, что производительность базы данных во многом зависит от эффективности операторов SQL приложения. Анализ SQL и времени выполнения может привести к улучшениям на стороне приложения, базы данных или с обеих сторон.

Теоретически можно пройти по исходному или байт-коду приложения и собрать все операторы SQL, хранящиеся как строки. Разумеется, это может быть сложно для приложений с большим числом операторов или динамически формирующих SQL. Можно также положиться на саму базу данных, которая регистрирует операторы SQL. Хотя это, несомненно, лучше анализа кода приложения, потребуются административные права на базу данных, а если одну базу используют несколько приложений, задача может оказаться непростой.

API JDBC предоставляет способ регистрировать операции с базой данных на уровне драйвера посредством метода `setLogWriter` класса `DriverManager`. Он выводит такие сведения, как регистрация драйверов, URL для создания соединений с базой данных и имена классов

соединений. Проблема журналирования диспетчера драйверов в том, что оно не даёт наиболее важных сведений: операторов SQL и переданных базе данных значений. Я перенаправил журналирование драйвера JDBC в `System.out` и проверил его простой программой, которая регистрирует драйвер, получает соединение с базой данных и выполняет несколько параметризованных операторов `SELECT`. В выводе журнала драйвера в листинге 13.1 нет и следа выполненных программой операторов `SELECT`.

```
DriverManager.initialize: jdbc.drivers = null
JDBC DriverManager initialized
registerDriver: driver[className=com.sybase.jdbc2.jdbc.SybDriver...]
registerDriver: driver[className=com.sybase.jdbc2.jdbc.SybDriver...]
DriverManager.deregisterDriver: com.sybase.jdbc2.jdbc.SybDriver@1d4c61c
registerDriver: driver[className=com.sybase.jdbc2.jdbc.SybDriver...]
DriverManager.getConnection("jdbc:sybase:Tds:he1unx142:2075/pslwrkdb1")
trying driver[className=com.sybase.jdbc2.jdbc.SybDriver...]
getConnection returning driver[className=
com.sybase.jdbc2.jdbc.SybDriver...]
```

Листинг 13.1. Вывод журналирования драйвера JDBC

> **Истории с передовой.** В Riggs Bank мы начали использовать Wily Introscope для наблюдения за производительностью кластера серверов. Introscope - приложение Java, которое собирает показатели производительности во время выполнения и позволяет сохранять их в реляционной базе данных для последующего анализа. Построенному на хороших идеях Introscope недоставало отточенности, а с нашим объёмом данных о производительности он просто не мог работать с историческими данными. Используемые Introscope таблицы выросли до миллионов записей, и мы подозревали, что причиной проблемы стали неэффективное устройство базы данных и операторы SQL. Техническая поддержка почти ничего не могла предложить, кроме: «Храните меньше данных или обновите компьютер». Чтобы оживить продукт, мы решили шпионить за SQL, используемым для выполнения запросов. Зарегистрировав и проанализировав операторы SQL, мы смогли добавить индексы к таблицам, оптимизировать устройство базы данных и перенастроить хранение данных в продукте. В результате производительность запросов выросла в десять раз, что стало большой победой команды разработчиков.

Чтобы надёжно подслушивать вызовы базы данных, необходимо заменить драйвер JDBC оболочкой, которая регистрирует оператор, а затем делегирует его «настоящему» драйверу. Есть прекрасная старая максима: хорошая технология или продукт делает «простые вещи лёгкими, а сложные возможными». P6Spy, который мы применим для журналирования JDBC, как раз это демонстрирует. Меньше чем за десять минут, после минимальных изменений конфигурации, он позволяет регистрировать вызовы базы данных существующих приложений без каких-либо изменений кода. P6Spy - бесплатное приложение с открытым исходным кодом, доступное на SourceForge.

После загрузки и установки P6Spy файлы `p6spy.jar` и `spy.properties` необходимо поместить в `CLASSPATH` целевого приложения. Чтобы активировать шпион, применяемый приложением класс настоящего драйвера следует заменить классом драйвера-шпиона. Имя настоящего драйвера нужно задать в `spy.properties`, чтобы шпион мог делегировать ему вызовы. Остальные параметры приложения, относящиеся к базе данных, менять не требуется. Например, если приложение соединяется с базой Oracle через стандартный драйвер, имя класса настоящего драйвера - `oracle.jdbc.driver.OracleDriver`. Чтобы активировать шпион, в файле конфигурации приложения это имя следует заменить на `com.p6spy.engine.spy.P6SpyDriver`, а в `spy.properties` настоящий драйвер нужно задать так:

```
realdriver=oracle.jdbc.driver.OracleDriver
```

После перезапуска приложения создаётся журнал с перехваченными вызовами базы данных. Например, настроив P6Spy для стандартного демонстрационного приложения JDK `TableExample` и введя в таблице несколько операторов `SELECT`, я получил следующий журнал:

```
1066073757578|16|0|statement||SELECT * FROM TAB
1066073780718|0|0|statement||SELECT * FROM TAB where tname='Alex'
```

P6Spy использует Log4J и допускает широкую настройку. Дополнительные сведения см. в документации продукта.

Короткий тест

1. Каковы общие подходы к подслушиванию?
2. Почему взаимодействие HTTP легко подслушивать?
3. Как можно защитить взаимодействие HTTP?
4. Благодаря чему возможно сетевое прослушивание?
5. Как подслушивать взаимодействие RMI?
6. Какой способ перехвата операторов SQL в JDBC самый надёжный?

Краткие итоги

• Подслушивание - это перехват и регистрация обмена сообщениями между клиентом и сервером. Оно возможно потому, что взаимодействие должно пересекать границы процессов или компьютеров.
• Туннелирование - приём, при котором между клиентом и сервером помещается посредник, а клиент перенастраивается на отправку сообщений посреднику. Посредник передаёт запросы серверу, пересылает ответ обратно клиенту и попутно регистрирует обмен.
• Подслушивать HTTP легко благодаря простой текстовой природе протокола и широкому разнообразию инструментов.
• Сетевое прослушивание возможно потому, что пакеты сетевых протоколов должны перемещаться между узлами. Неразборчивый сетевой узел принимает все пакеты независимо от адреса назначения, позволяя анализатору перехватывать сообщения, проходящие через сеть.
• Подслушивать RMI можно с помощью такого сетевого анализатора, как Ethereal. Для этого требуется изучать низкоуровневые сетевые пакеты, несущие двоичное содержимое сообщений RMI.
• Подслушивание JDBC легко осуществляется с помощью оболочек над настоящим драйвером и объектами соединения. Оболочки регистрируют каждый оператор, прежде чем передать его настоящим классам драйвера.

Глава 14. Управление загрузкой классов

> «Никто не замечает, когда всё идёт хорошо».
>
> *Закон жалоб Циммермана*

В этой главе

• Внутреннее устройство JVM с точки зрения загрузки классов
• Написание собственного загрузчика классов
• Короткий тест
• Краткие итоги

Загрузчики классов загружают и инициализируют классы и интерфейсы в виртуальной машине Java (JVM). В этой главе дан обзор данного процесса и показано, как разработать собственный загрузчик классов, позволяющий декорировать загружаемый байт-код.

Внутреннее устройство JVM с точки зрения загрузки классов

Виртуальная машина Java, работающая поверх операционной системы (ОС), предоставляет уровень абстракции приложениям, написанным на Java. Спецификация языка и стандартизованные интерфейсы программирования приложений (API) обеспечивают межплатформенную разработку и развёртывание, поскольку приложения не обращаются непосредственно к ОС с платформенно-зависимыми вызовами. Большая часть основных API Java сама написана на Java; лишь небольшая доля машинного кода вызывается через JNI. Приложения Java могут содержать байт-код и машинные динамические библиотеки. В Windows машинные библиотеки распространяются как динамически подключаемые библиотеки (`.dll`), а в Unix - как разделяемые библиотеки (`.so`). Чтобы выполнить приложение Java, JVM применяет загрузчик классов, который загружает и инициализирует системные и прикладные классы, что может привести к загрузке машинных библиотек приложения. JVM с точки зрения загрузки классов показана на рисунке 14.1.

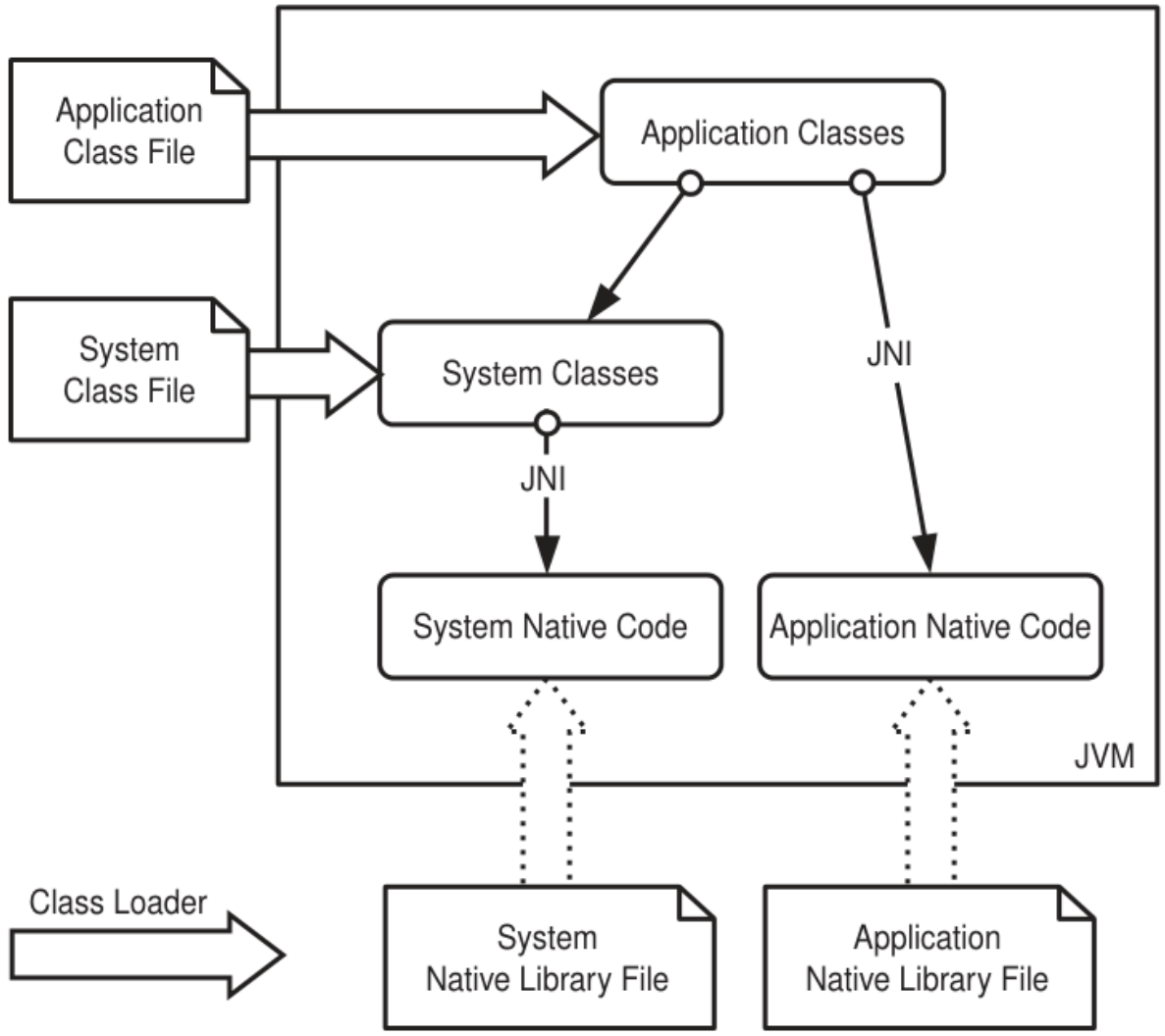

Рисунок 14.1. JVM с точки зрения загрузки классов

Хотя обычному приложению не требуется заботиться о загрузке и инициализации, возможность управлять этим процессом используется в таких приёмах, как инструментирование байт-кода во время выполнения и защита целостности байт-кода.

Процесс загрузки приложения начинается с начального класса, переданного средству запуска Java (обычно как параметр командной строки среде выполнения Java). Все родительские классы и любые классы, на которые ссылается метод `main()` начального класса, лениво загружаются и инициализируются по мере обращения к ссылкам. Если явно не указано иное, для загрузки упомянутого класса используется загрузчик класса, который на него ссылается. Если загрузчики классов загружают классы Java, спросите вы, то чем загружаются сами загрузчики классов? Ответ - начальным загрузчиком классов, реализованным в машинном коде и применяемым для загрузки таких основных классов Java, как `java.lang.Object` и `java.lang.ClassLoader`. Загрузчик расширений служит для загрузки библиотек расширений, обычно из каталога `lib/ext` среды JRE, причём помещённые туда файлы JAR автоматически становятся доступны приложениям Java. Загрузчик классов приложения создаётся внутри средством запуска JVM для загрузки классов из стандартного `CLASSPATH`. Наконец, загрузчик, называемый системным загрузчиком классов, обычно совпадает с загрузчиком классов приложения.

Загрузчики классов организованы в цепочку, где дочерний загрузчик позволяет родительскому найти класс, прежде чем пытается найти его сам. Обычно загрузчик классов сначала проверяет, не

загрузил и не инициализировал ли он уже этот класс. Если класс ещё не загружен, загрузчик пытается создать или загрузить его и, если находит, инициализировать в JVM. Начальный загрузчик находится в корне иерархии загрузчиков классов и соответствует значению `null`, тогда как системный загрузчик по умолчанию становится родителем делегирования для новых загрузчиков. В таблице 14.1 перечислены загрузчики классов и сведения об источниках данных классов.

Загрузчик классов	**Источник данных классов**
Начальный	Каталоги и файлы JAR, перечисленные в системном свойстве `sun.boot.class.path`, которое по умолчанию включает основные классы среды выполнения в `rt.jar` и несколько других стандартных JAR. Значением можно управлять с помощью параметра командной строки `-Xbootclasspath` средства запуска Java.
Расширений	Каталоги, перечисленные в системном свойстве `java.ext.dirs`, которое по умолчанию указывает на каталог `lib/ext` среды JRE. Его можно явно задать в командной строке с помощью `-Djava.ext.dirs=<path>`.
Приложения	Каталоги и JAR, перечисленные в системном свойстве `java.class.path`, которое по умолчанию задаётся из переменной среды `CLASSPATH`.

Таблица 14.1. Загрузчики классов и источники их файлов CLASS

Чтобы во время выполнения заглянуть в иерархию загрузчиков классов, напишем простой класс, который выводит загрузчик, применённый для его загрузки, и цепочку его родителей. Код класса показан в листинге 14.1.

```
public class PrintClassLoaders {
    public static void main(String[] args) {
        System.out.println("System class loader = " +
            ClassLoader.getSystemClassLoader());
        System.out.println("Thread context class loader = " +
            Thread.currentThread().getContextClassLoader());
        System.out.println("Class loader hierarchy for this class:");

        String padding = "";
        ClassLoader cl = PrintClassLoaders.class.getClassLoader();
        while (cl != null) {
            System.out.println(padding + cl);
            cl = cl.getParent();
            padding += "    ";
        }
        System.out.println(padding + "null (bootstrap class loader)");
    }
}
```

Листинг 14.1. Исходный код `PrintClassLoaders`

Класс `PrintClassLoaders` находится в пакете `covertjava.classloader`. Его выполнение даёт следующий вывод:

```
System class loader = sun.misc.Launcher$AppClassLoader@12f6684
Thread context class loader = sun.misc.Launcher$AppClassLoader@12f6684
Class loader hierarchy for this class:
sun.misc.Launcher$AppClassLoader@12f6684
    sun.misc.Launcher$ExtClassLoader@f38798
        null (bootstrap class loader)
```

Из вывода можно заключить, что иерархия загрузчиков классов соответствует порядку, в котором среда выполнения пытается найти и загрузить класс. Экземпляр `AppClassLoader`, возвращаемый вызовом `getClassLoader()` из начального класса, является загрузчиком, связанным с `PrintClassLoaders`. Этот же экземпляр используется как системный загрузчик, возвращаемый

ClassLoader.getSystemClassLoader(), а его непосредственный родитель - загрузчик расширений ExtClassLoader. Родителем загрузчика расширений служит null, что подразумевает начальный загрузчик. Поскольку загрузчики делегируют работу родителю, прежде чем пытаются загрузить класс сами, классы в начальном пути классов всегда загружаются начальным загрузчиком. Если класс там не найден, следующим проверяется путь классов расширений. Путь классов приложения проверяется, только если класс не найден ни в одном из этих путей. Если его не находит и загрузчик приложения, выбрасывается печально известное исключение ClassNotFound.

Необходимо ясно понимать несколько важных положений о загрузке классов:

• Загрузчик, который загрузил и определил класс в процессе, связывается с ним и называется определяющим загрузчиком классов.
• Загруженный класс определяется не только именем, но парой из имени и определяющего загрузчика. Два класса с одним именем, но разными загрузчиками считаются различными.
• Если класс A ссылается на ещё не загруженный класс B, для загрузки B используется определяющий загрузчик A.

Таким образом, если класс был загружен из начального пути начальным загрузчиком, он не способен ссылаться на классы из пути приложения, если только явно не обращается к ним через системный или собственный загрузчик. С потоком связан загрузчик классов, который можно получить методом getContextClassLoader(). Обычно ему назначается загрузчик класса, запустившего поток. Иногда для доступа к классам из другого источника вместо определяющего загрузчика нужно использовать контекстный загрузчик потока. Это может потребоваться сервлетам, загруженным собственным загрузчиком веб-контейнера. Загрузчик контейнера может быть настроен на загрузку только классов веб-приложения, и тогда классы из системного пути окажутся недоступны.

Написание собственного загрузчика классов

Java позволяет приложениям управлять созданием или загрузкой классов посредством собственных, то есть пользовательских, загрузчиков. Например, веб-серверы и серверы приложений предоставляют каждому развёрнутому приложению отдельный загрузчик, чтобы лучше изолировать логические приложения, работающие в одной JVM. Напомним: даже два класса, загруженные из одного файла, считаются различными, если их определили разные загрузчики. Когда у каждого приложения собственный загрузчик, статические переменные, которые так часто служат для реализации шаблона «одиночка» и хранения глобальных данных, становятся общими в пределах приложения, а не всей JVM. Собственные загрузчики также позволяют веб-серверам и серверам приложений перезагружать классы без перезапуска JVM. Ещё один мощный приём - применение собственного загрузчика для создания классов на лету или изменения двоичной структуры данных класса до его определения в среде выполнения Java.

В этом разделе мы создадим в пакете covertjava.classloader собственный загрузчик DecoratingClassLoader, позволяющий на лету изменять загруженные двоичные данные классов. После написания этот класс можно будет использовать для поддержки зашифрованных файлов классов или декорирования байт-кода, которые рассматриваются далее в книге. Пока сосредоточимся на загрузке классов из заданного пользователем пути и вызове метода обратного вызова, который получает возможность изучить и изменить двоичные данные. Код будет в общих чертах основан на реализации загрузчика классов в JUnit. Все загрузчики должны расширять абстрактный класс java.lang.ClassLoader и как минимум переопределять метод findClass, чтобы загружать или создавать класс по заданному имени. Реализация findClass в DecoratingClassLoader показана в листинге 14.2.

```
protected Class findClass(String name) throws ClassNotFoundException {
    System.out.println("DecoratingClassLoader loading class " + name);
    byte[] bytes = getClassBytes(name);
    if (getDecorator() != null)
```

Истории с передовой. WebCream на лету преобразует приложения Java в веб-сайты HTML. Одна из важнейших функций этого продукта - возможность выполнять несколько виртуальных клиентов внутри одной JVM. Многопоточность позволяет легко запустить в JVM несколько экземпляров приложения Java, но в итоге приложения совместно используют классы. Это приводит к ошибкам, поскольку статические инициализаторы выполняются только для первого клиента, а статические члены общие для всех клиентов. Для чёткого разделения виртуальных клиентов WebCream применяет собственный загрузчик классов. Каждый виртуальный клиент получает отдельный экземпляр загрузчика, который загружает все классы приложения. Поскольку JVM определяет класс по сочетанию загрузчика и имени, каждый виртуальный клиент получает собственные класс и статические данные.

```
        bytes = getDecorator().decorateClass(name, bytes);
    return defineClass(name, bytes, 0, bytes.length);
}
```

Листинг 14.2. Реализация `findClass`

Метод `findClass` вызывается только в том случае, если класс не найден ни одним родительским загрузчиком в иерархии. Благодаря этому основные классы наподобие `java.lang.Object` загружаются начальным загрузчиком. После загрузки двоичных данных класса декоратору предоставляется возможность дополнить байт-код. Затем байты класса передаются методу `defineClass` класса `java.lang.ClassLoader`, который преобразует двоичное представление класса во внутренний объект `Class`. Чтение двоичных данных класса выполняется в методе `getClassBytes()`, показанном в листинге 14.3.

```
protected byte[] getClassBytes(String className)
    throws ClassNotFoundException
{
    byte[] bytes = null;
    for (int i = 0; i < classPathItems.size(); i++) {
        String path = (String)classPathItems.get(i);
        String fileName = className.replace('.', '/') + ".class";
        if (isJar(path) == true) {
            bytes = loadJarBytes(path, fileName);
        }
        else {
            bytes = loadFileBytes(path, fileName);
        }
        if (bytes != null)
            break;
    }
    if (bytes == null)
        throw new ClassNotFoundException(className);
    return bytes;
}
```

Листинг 14.3. Реализация `getClassBytes`

Метод перебирает элементы настроенного пути классов и пытается найти и загрузить данные файла `.class` в соответствующем каталоге. Если данные класса не найдены, выбрасывается `ClassNotFoundException`. `DecoratingClassLoader` принимает путь классов как параметр конструктора, а конструктор по умолчанию инициализирует его путь значением системного свойства `decorate.class.path`. Полный исходный код `DecoratingClassLoader` находится в каталоге `CovertJava/src/covertjava/classloader`.

Чтобы использовать собственный загрузчик, его нужно явно указать для загрузки класса, недоступного родительским загрузчикам. Это можно сделать, передав экземпляр загрузчика методу `Class.forName()` или непосредственно вызвав метод `loadClass()` собственного загрузчика. Помните, что загрузчики делегируют загрузку родительской цепочке, и если родитель может найти класс, метод `findClass()` дочернего загрузчика не вызывается. Поэтому самый чистый способ позволить собственному загрузчику загрузить класс - обеспечить отсутствие этого класса в системном, начальном и расширенном путях классов. Этот вариант часто выбирают реализации веб-серверов и серверов приложений, предупреждающие, что классы приложения не следует помещать в системный `CLASSPATH`. Если требуется полный контроль над загрузкой, можно переопределить метод `loadClass(String name, boolean resolve)`. Он вызывается для начала загрузки класса, а стандартная реализация сначала позволяет загрузить класс родителю. После переопределения собственный загрузчик вызывается для каждого класса в системе независимо от того, где находятся данные класса. Другой вариант - использовать начальный загрузчик как родителя, передав конструктору `ClassLoader` значение `null`. Однако лучше всего убрать из системного `CLASSPATH` те JAR и каталоги с классами, которые должен загружать собственный загрузчик: это обеспечивает разделение системных и пользовательских классов.

Чтобы проверить делегирование `DecoratingClassLoader`, добавим декоратор `PrintingClassDecorator`, который просто выводит имя класса, для которого вызван (см. листинг 14.4).

```
public class PrintingClassDecorator implements ClassDecorator {
    public byte[] decorateClass(String name, byte[] bytes) {
        System.out.println("Processed bytes for class " + name);
        return bytes;
    }
}
```

Листинг 14.4. Реализация `PrintingClassDecorator`

Чтобы увидеть `DecoratingClassLoader` в действии, испытаем его на классе `PrintClassLoaders`, рассмотренном ранее. Понадобится класс запуска, который создаёт экземпляр собственного загрузчика, а затем загружает с его помощью тестовый класс. Класс `DecoratingLauncher` из пакета `covertjava.classloader`

принимает имя запускаемого тестового класса и загружает его через `DecoratingClassLoader`. Затем посредством API отражения он вызывает функцию `main()` тестового класса. Метод `main()` класса `DecoratingLauncher` показан в листинге 14.5.

```
public static void main(String[] args) throws Exception {
    if (args.length < 1) {
        System.out.println("Missing command line parameter <main class>");
        System.out.println("Syntax: DecoratingLauncher " +
            "[-Ddecorate.class.path=<path>] <main class> [<arg1>, [<arg2>]..]");
        System.exit(1);
    }

    DecoratingClassLoader decoratingClassLoader = new DecoratingClassLoader();
    decoratingClassLoader.setDecorator(new PrintingClassDecorator());
    Class mainClass = Class.forName(args[0], true, decoratingClassLoader);
    String[] mainArgs = new String[args.length - 1];
    System.arraycopy(args, 1, mainArgs, 0, args.length - 1);
    invokeMain(mainClass, mainArgs);
}
```

Листинг 14.5. Метод `main()` *класса* `DecoratingLauncher`

Обратите внимание: экземпляр нашего загрузчика передаётся методу `forName()`. Перед выполнением теста нужно удалить класс `PrintClassLoaders` из `CLASSPATH` и поместить его в каталог, на который будет указывать системное свойство `decorate.class.path`. Придётся изменить цель `release` в `build.xml`, чтобы до создания JAR перемещать

covertjava.classloaders.PrintClassLoaders в каталог CovertJava/lib/classes. Также нужно создать в каталоге bin новый пакетный файл, который вызывает средство запуска и передаёт ему PrintClassLoaders как параметр. Содержимое пакетного файла показано в листинге 14.6.

```
@echo off
rem Demonstration of using custom class loader on PrintClassLoaders class
call setEnv.bat
set JAVA_OPTS=-Ddecorate.class.path=..\lib\classes %JAVA_OPTS%
java %JAVA_OPTS% covertjava.classloader.DecoratingLauncher ^
covertjava.classloader.PrintClassLoaders
```

Листинг 14.6. Код classLoaderTest.bat

Выполнение classLoaderTest.bat даёт следующий вывод:

```
Processed bytes for class covertjava.classloader.PrintClassLoaders
System class loader = sun.misc.Launcher$AppClassLoader@12f6684
Thread context class loader = sun.misc.Launcher$AppClassLoader@12f6684
Class loader hierarchy for this class:
covertjava.classloader.DecoratingClassLoader@ad3ba4
    sun.misc.Launcher$AppClassLoader@12f6684
        sun.misc.Launcher$ExtClassLoader@f38798
            null (bootstrap class loader)
```

Из вывода видно, что загрузчик, связанный с классом PrintClassLoaders, является экземпляром DecoratingClassLoader. В качестве родителя DecoratingClassLoader получил системный загрузчик классов, а остальная иерархия не изменилась.

Короткий тест

1. Каково назначение загрузчика классов?
2. Что такое начальный загрузчик и какие классы он загружает?
3. Что такое загрузчик расширений и какие классы он загружает?
4. Какие классы считаются одинаковыми внутри JVM?
5. Для чего можно использовать собственный загрузчик классов?
6. Можно ли с помощью собственного загрузчика загружать все классы, включая основные классы Java?
7. Почему DecoratingClassLoader использует собственный путь классов?

Краткие итоги

• Загрузчики классов загружают и инициализируют классы и интерфейсы в JVM.
• Процесс загрузки приложения начинается с начального класса, переданного средству запуска. Все родительские классы и любые классы, на которые ссылается метод `main()` начального класса, лениво загружаются и инициализируются по мере обращения к ссылкам.
• Начальный загрузчик реализован в машинном коде и применяется для загрузки таких основных классов Java, как `java.lang.Object` и `java.lang.ClassLoader`.

• Загрузчики классов организованы в цепочку, где дочерний загрузчик позволяет родительскому найти класс, прежде чем пытается найти его сам.
• Загруженный класс определяется не только именем, но парой из имени и определяющего загрузчика классов.
• Собственные загрузчики позволяют приложениям Java управлять загрузкой классов. Они применяются для перезагрузки классов, разделения логических приложений внутри одной JVM и декорирования либо создания байт-кода на лету.

Глава 15. Замена и исправление основных классов Java

> «Путь без препятствий, вероятно, никуда не ведёт».
>
> *Дефальк*

В этой главе

- Зачем это нужно?
- Исправление основных классов Java с помощью начального пути классов
- Пример исправления `java.lang.Integer`
- Короткий тест
- Краткие итоги

Зачем это нужно?

В главе 5, «Замена и исправление классов приложения», мы говорили об исправлении классов Java для изменения или расширения лежащей в их основе логики. Представленные там приёмы работают для классов приложений и библиотек, загружаемых системным либо собственным загрузчиком. Однако попытки применить их к основным классам в пакете, имя которого начинается с `java`, не дают результата: продолжает использоваться исходная версия класса. В главе 14, «Управление загрузкой классов», подробно рассматривалось, как загружаются классы, и после небольшого размышления становится понятно, почему для системных классов требуется другой подход. Напомним, что системные классы загружает машинный начальный загрузчик, который не использует переменную среды `CLASSPATH`. Хотя общий подход к исправлению системных классов подобен исправлению классов приложения, есть несколько тонких различий, и именно им посвящена эта глава.

Действительно ли нужно исправлять основные классы? За свою карьеру мне приходилось исправлять классы приложений гораздо чаще, чем системные классы. Одна из причин, возможно, в том, что основные классы хорошо спроектированы и к настоящему времени созрели до формы, устраивающей большинство разработчиков. Тем не менее время от времени можно наткнуться на недостаток основного класса, для которого нет хорошего обходного решения.

Определённо не рекомендуется использовать исправление основных классов Java как постоянное решение. Оно влечёт правовые последствия (лицензия JDK запрещает изменять основные классы) и может потребовать дополнительной работы при переходе на новую версию JDK. Однако этот приём даёт разработчику гораздо больше контроля. С его помощью можно вставлять трассировку в код JDK и временно менять реализацию основной логики в соответствии с потребностями приложения. И последнее, но не менее важное: это просто здорово, и вооружиться столь мощным приёмом не помешает. Только обязательно прочтите лицензионное соглашение, прежде чем встать на этот путь.

Исправление основных классов Java с помощью начального пути классов

Как уже упоминалось, подход к исправлению основных классов подобен подходу для классов приложений. Необходимо получить исходный файл класса, который требуется исправить. JDK предусмотрительно распространяется с исходным кодом (спасибо, Sun!), поэтому чаще всего код можно просто взять из `src.jar`. Обратите внимание, что некоторые системные классы поставляются без исходного кода; это справедливо для классов внутри пакета `sun` и других непубличных пакетов. Файлы классов можно декомпилировать, как описано в главе 2, «Декомпиляция классов», однако при этом необходимо соблюдать лицензионное соглашение.

> **Истории с передовой.** Я работал над продуктом WebCream, способным запускать несколько виртуальных клиентов Swing внутри одной JVM. Во время испытаний выяснилось, что после некоторого времени работы JVM блокируется и новые клиенты больше не могут инициализироваться. Исследование с помощью дампов потоков JVM, описанных в главе 10, «Профилировщики для анализа приложения во время выполнения», показало, что блокировка происходит при вызове метода `getTreeLock()` класса `java.awt.Component`. Реализация `getTreeLock()` просто возвращает переменную, объявленную в `Component` так:

```
static final Object LOCK = new AWTTreeLock();
```

> Таким образом, AWT использует глобальную блокировку, общую для всех компонентов, и если один поток своевременно не освобождает монитор блокировки, никакой другой поток не может выполнять операции AWT и Swing. Разработчики Java сделали это для предотвращения гонок при повторной компоновке, но для такого продукта, как WebCream, это настоящий убийца масштабируемости. В тот момент немедленным решением стало исправление класса `java.awt.Component`, чтобы вместо глобальной блокировки он использовал блокировку, относящуюся к конкретному виртуальному клиенту. После установки исправления о блокировках виртуальных клиентов больше не сообщалось.

Получив исходный код, можно вставить новую логику. Скомпилируйте класс так же, как любой другой, и, пожалуйста, постарайтесь не добавить ошибок. Теперь, когда у вас есть новая версия байт-кода, остаётся указать JVM использовать её вместо исходного байт-кода. Этого можно добиться, управляя начальным путём классов, как объяснялось в главе 14. Начальный загрузчик использует этот путь для поиска основных классов. По умолчанию путь включает только `rt.jar` и, возможно, несколько других системных библиотек. Файл `rt.jar`, расположенный в `JRE_HOME\lib`, содержит большинство основных классов, поэтому, если исходного кода класса нет и нужно найти его байт-код, сначала проверьте `rt.jar`. Начальный путь классов задаётся параметром `-Xbootclasspath` в командной строке средства запуска Java. Выполнение `java -X` выводит следующую справку:

```
C:\CovertJava\java -X

-Xbootclasspath:<directories and zip/jar files separated by ;>
    set search path for bootstrap classes and resources
-Xbootclasspath/a:<directories and zip/jar files separated by ;>
    append to end of bootstrap class path
-Xbootclasspath/p:<directories and zip/jar files separated by ;>
    prepend in front of bootstrap class path
...
```

Параметр командной строки позволяет задать или дополнить начальный путь классов. Поскольку нас интересует замена существующего класса, используем `-Xbootclasspath/p:`, чтобы добавить каталог с исправлениями перед стандартным путём. При запуске JVM с этим параметром вместо исходного класса применяется исправленный.

Пример исправления java.lang.Integer

Чтобы применить теорию на практике, напишем простое исправление для `java.lang.Integer`. По причинам, неизвестным сообществу Java, объект `Integer` неизменяем. После задания значения его нельзя изменить. Вероятно, замысел состоял в том, чтобы объекты `Integer` вели себя подобно объектам `String`: если нужно изменить значение, представленное объектом `Integer`, следует создать новый экземпляр и использовать его вместо старого. Проблема такого подхода в неэффективном расходе памяти приложениями, которым нужны динамические коллекции целых чисел. Java не предоставляет классов коллекций для примитивных типов, поэтому единственный способ получить динамический массив целых чисел - использовать `java.util.Array` из экземпляров `Integer`. Если сохранённое целое требуется изменить, необходимо создать новый экземпляр `Integer` и поместить его в массив на место прежнего значения. Разумеется, выделение памяти и последующая сборка мусора создают значительные накладные расходы. Гораздо лучше было бы изменить внутреннее значение объекта `Integer`. Однако поскольку `java.lang.Integer` неизменяем, единственное законное обходное решение - создать и использовать собственный класс, имитирующий `Integer` и предоставляющий метод `setValue()`.

Тем не менее мы исправим существующий класс `java.lang.Integer` и наделим его методом `setValue()`. Сделаем это исключительно из академического интереса и для практики проповедуемых нами приёмов, поскольку не хотим нарушать лицензионное соглашение Java. Из исходного кода `java.lang.Integer` видно, что значение объекта хранится в закрытом поле `value`. Поэтому нужно скопировать исходный файл в каталог `CovertJava\src\java\lang` и вставить метод `setValue` (см. листинг 15.1).

```
public void setValue(int value) {
    this.value = value;
}
```

Листинг 15.1. Исходный код метода `setValue()`

Следующий шаг - создать тестовый класс `CorePatchTest`, обращающийся к только что добавленному методу `setValue()`. Код тестового класса показан в листинге 15.2.

```
package covertjava.patching;

public class CoreClassTest {
    public static void main(String[] args) {
        Integer i = new Integer(10);
        System.out.println("Old value = " + i);
        i.setValue(100);
        System.out.println("New value = " + i);
    }
}
```

Листинг 15.2. Использование исправленного `java.lang.Integer`

Компиляция классов, использующих исправленные версии основных классов, может быть несколько хитрой, если открытый интерфейс основного класса изменился. Попытка запустить `javac` для нашего тестового класса приводит к ошибке, поскольку в реализации `Integer` из JDK нет метода `setValue()`. По этой причине применить Ant для компиляции исправленного `java.lang.Integer` невозможно. Самый простой обходной путь - вручную скомпилировать

исправленный Integer с помощью javac, а затем скопировать файл класса в каталог CovertJava/distrib/patches. Теперь компилятор можно настроить так, чтобы он использовал в нашем проекте исправленную версию Integer: для этого исправленный класс помещается в начальном пути классов перед исходной версией.

javac, как и задача javac в Ant, принимает параметр -bootclasspath, позволяющий переопределить стандартный начальный путь классов. Однако при переопределении начального пути для javac необходимо указать расположение rt.jar и всех остальных системных библиотек. В результате сценарии сборки начинают зависеть от пути установки JDK или переменных среды. Более простой способ - передать -Xbootclasspath/p: той JVM, в которой работает Ant, чтобы не переопределять стандартный путь, а всего лишь добавить элемент перед ним. Сценарий ant.bat передаёт параметры командной строки вызову Java через переменную среды ANT_OPTS, чем мы и воспользуемся, добавив следующую строку в CovertJava\bin\build.bat:

```
ANT_OPTS=-Xbootclasspath/p:..\distrib\patches
```

Теперь с помощью Ant можно собрать проект и библиотеки дистрибутива (цель release). Последняя задача после сборки проекта - создать в каталоге CovertJava\bin пакетный файл corePatchTest.bat, выполняющий CorePatchTest. Чтобы гарантировать применение исправленной версии Integer, снова передаём Java параметр -Xbootclasspath. Соответствующий исходный код corePatchTest.bat показан в листинге 15.3.

```
set JAVA_OPTS=-Xbootclasspath/p:..\distrib\patches
java %JAVA_OPTS% covertjava.patching.CoreClassTest
```

Листинг 15.3. Выполнение теста исправления основного класса

corePatchTest.bat даёт следующий вывод:

```
Old value: 10
New value: 100
```

Вуаля! В нашей копилке появился ещё один приём.

Короткий тест

1. Можете ли вы придумать случай, когда захотелось бы исправить основной класс?
2. Чем процесс исправления основных классов отличается от исправления классов приложения?
3. Зачем нужно изменять начальный путь классов?

Краткие итоги

• Исправление основных классов Java может помочь при отладке и изучении JVM.
• Основные классы всегда загружаются начальным загрузчиком, который ищет байт-код в начальном пути классов.
• Чтобы исправить основной класс, новую версию необходимо поместить в начальном пути перед старой.
• Чтобы скомпилировать класс, использующий исправленную версию основного класса с изменённым открытым интерфейсом, исправленную версию необходимо указать в начальном пути классов компилятора Java.

Глава 16. Перехват потока управления

> «Нет ничего настолько простого, чтобы его нельзя было понять неправильно».
>
> *Закон Фримена*

В этой главе

• Определение потока управления
• Перехват системных ошибок
• Перехват системных потоков
• Перехват вызова `System.exit`
• Реакция на завершение работы JVM с помощью перехватчиков
• Перехват методов динамическим заместителем
• Интерфейс профилировщика виртуальной машины Java
• Короткий тест
• Краткие итоги

Определение потока управления

Поток управления - это последовательность выполнения методов и инструкций потоком. Виртуальная машина Java (JVM) выполняет инструкции байт-кода Java в том порядке, в котором они находятся в файле класса. Поток управления можно программировать условными конструкциями, например `if`, `else` и `for`, или вызовом метода. Перехват потока управления подразумевает знание выполняемой инструкции или метода и возможность изменить ход выполнения во время работы. Например, может потребоваться перехватить вызов `System.exit()`, чтобы не дать JVM завершить работу. Пока вы не слишком воодушевились возможностями, сразу уточню ожидания. Прямого способа перехватить любую инструкцию или вызов метода в работающей JVM нет, если только она не запущена в режиме профилирования. Методы выполняются JIT-компилятором, и стандартного API Java, позволяющего добавить к вызовам методов слушатель или перехватчик, не существует. Однако мы рассмотрим несколько косвенных подходов к перехвату управления в распространённых сценариях. Кроме того, мы изучим интерфейс профилировщика JVM, с помощью которого в режиме отладки можно перехватить любой вызов.

Перехват системных ошибок

Системные ошибки сообщаются JVM при ненормальных условиях, предположительно находящихся вне управления приложения. Они выбрасываются как подклассы `java.lang.Error` и поэтому не объявляются, то есть могут быть выброшены любым методом, даже если его сигнатура явно их не объявляет. К системным относятся ошибки виртуальной машины, например `OutOfMemoryError` и `StackOverflowError`, ошибки связывания, такие как `ClassFormatError` и `NoSuchMethodError`, и другие сбои. По принятому соглашению программисты приложений должны перехватывать только экземпляры `java.lang.Exception`, а значит, такое состояние, как нехватка памяти, не обнаруживается логикой обработки ошибок приложения. Для большинства реальных приложений это нежелательно: даже если при ошибке ничего нельзя сделать, приложение обычно должно записать её в файл журнала и попытаться освободить удерживаемые ресурсы. Хорошее проектное решение - поместить на вершине стека вызовов основных потоков приложения блок `try-catch`, перехватывающий `java.lang.Error` или `java.lang.Throwable` и передающий управление методу, который анализирует ошибочное состояние, регистрирует его и пытается аккуратно завершить работу. Например:

```
public static void main(String[] args) {
    try {
        // Execute application logic
        runApplication();
    }
    catch (Throwable x) {
        // Log error and attempt clean shutdown
        onFatalError(x);
    }
}
```

Если JVM не хватает памяти или класс не найден, приложение может попытаться исправить положение, освободив содержимое кэшей либо отключив функцию, затронутую отсутствующими классами. Всё лучше, чем постыдно исчезнуть без следа.

Перехват системных потоков

До того как журналирование стало фактическим требованием к приложениям Java, отладочную трассировку часто выводили с помощью `System.out.println`. Недостатки этого подхода многочисленны и очевидны. Такие трассировки после написания нельзя включать или отключать без изменения кода. Хотя поток вывода приложения можно перенаправить в файл для постоянного хранения, ротация отсутствует, а поскольку файл остаётся открытым, его невозможно удалить до завершения приложения (поэтому размер файла может стать непомерным). При работе с унаследованным кодом Java, испещрённым вызовами `System.out.println()`, часто возникает задача заменить их вызовами каркаса журналирования (обсуждение журналирования и трассировки см. в главе 6, «Эффективная трассировка»). Важно также захватывать стандартный поток ошибок, в который выводят данные такие методы, как `printStackTrace()` класса `Exception`. Одно из изящных решений - перехватывать вывод в `System.out` и `System.err` и вместо этого отправлять его в файл журнала.

Приём основан на том, что системный поток вывода можно перенаправить в собственный `PrintStream` с помощью метода `setOut` класса `java.lang.System`. `PrintStream` - класс-декоратор вокруг экземпляра `OutputStream`, отвечающего за собственно вывод. Следовательно, задача состоит в том, чтобы разработать перенаправляющий `OutputStream`, который пишет в файл журнала вместо стандартного вывода процесса, а затем назначить его потоку `System.out`.

Мы разработаем класс LogOutputStream, расширяющий java.io.OutputStream и записывающий вывод в файл журнала посредством Apache Log4J. Каркас ввода-вывода Java спроектирован очень хорошо, и все методы OutputStream в конце концов делегируют работу единственному методу write(), принимающему целочисленный параметр. LogOutputStream накапливает в StringBuffer символы, получаемые методом write(int), а после обнаружения разделителя строк записывает весь буфер на диск через Log4J. Единственная хитрость реализации - обнаружение конца строки. Как вам, несомненно, известно, в Unix конец строки отмечается одним символом: \n (новая строка). В Windows конец строки отмечается сочетанием двух символов: \r и \n (возврат каретки и новая строка). Чтобы написать действительно межплатформенный код Java, необходимо опираться на системное свойство line.separator. Поскольку это свойство является строкой, реализация должна искать подстроку, а не сравнивать символы. Наша реализация оптимизирована так, чтобы сначала сравнением символов проверить возможный конец строки, а затем поиском подстроки убедиться, что это действительно конец. Переопределённый метод write() показан в листинге 16.1.

```
public void write(int b) throws IOException {
    char ch = (char)b;
    this.buffer.append(ch);
    if (ch == this.lineSeparatorEnd) {
        // Check on a char by char basis for speed
        String s = buffer.toString();
        if (s.indexOf(lineSeparator) != -1) {
            // The whole separator string is written
            logger.info(s.substring(0, s.length() - lineSeparator.length()));
            buffer.setLength(0);
        }
    }
}
```

Листинг 16.1. Метод write() *класса* LogOutputStream

Здесь logger - ссылка на статическую переменную типа org.apache.log4j.Logger, объявленную в LogOutputStream следующим образом:

```
static Logger logger = Logger.getLogger(LogOutputStream.class.getName());
```

Таким образом, весь вывод в System.out перенаправляется каркасу Log4J как сообщения уровня INFO от класса LogOutputStream. Чтобы увидеть наш класс в действии, нужно настроить файловый добавитель в log4j.properties и установить перехватчик, как показано в листинге 16.2.

```
public static void main(String[] args) {
    System.out.println("Installing the interceptor...");
    PrintStream out = new PrintStream(new LogOutputStream(), true);
    System.setOut(out);
    System.out.println("Hello, world");
    System.out.println("Done");
}
```

Листинг 16.2. Установка перехвата System.out

При выполнении метода main() класса LogOutputStream сообщение Installing the interceptor... появляется в консоли, а сообщения Hello, world и Done записываются в файл журнала. Тот же перехватчик можно установить для потока System.err. Для гибкости его можно параметризовать, чтобы конструктор принимал уровень журналирования и имя потока. Аналогичным образом поток System.In можно программно задать методом System.setIn(), чтобы передавать приложению требуемый ввод.

Перехват вызова System.exit

Процесс JVM обычно завершается, когда не остаётся активных потоков. Потоки, работающие как демоны (`Thread.isDaemon() == true`), не мешают завершению JVM. В многопоточных приложениях, включая графические интерфейсы Swing и серверы RMI, добиться аккуратного завершения естественным окончанием всех потоков нелегко. Для принудительного завершения JVM и процесса часто вызывают `System.exit()`. Полагаться на `System.exit()` стало обычным делом даже в не очень сложных программах; хотя это облегчает жизнь разработчику приложения, такой вызов может создать проблему для продуктов среднего уровня, например веб-серверов и серверов приложений. Неосторожный вызов `System.exit()` из веб-приложения способен, например, остановить процесс веб-сервера и, возможно, лишить пользователей доступа к другим веб-приложениям и статическим страницам HTML. Так с системными администраторами не подружишься, а каждый хороший разработчик знает цену здоровых отношений с этой командой.

В этом разделе рассматривается простой способ перехватить вызов `System.exit()` и предотвратить завершение JVM. Этот приём можно обнаружить, изучив исходный код метода `exit()` класса `java.lang.System`. Первым делом метод проверяет, установлен ли диспетчер безопасности. Если установлен, метод проверяет наличие у вызывающей стороны разрешения на завершение JVM. Поэтому наша задача - установить собственный диспетчер безопасности (или изменить политику безопасности, если диспетчер уже установлен), запрещающий завершение до тех пор, пока оно не будет явно разрешено. Класс `InterceptingSecurityManager` из пакета `covertjava.intercept` расширяет класс `SecurityManager` и переопределяет метод `isExitAllowed()`, чтобы управлять завершением JVM. Внутренний флаг, который можно задать методом `setExitAllowed()`, определяет, разрешено ли JVM завершиться. Если завершение запрещено, для изменения потока управления выбрасывается непроверяемое `SecurityException`. Метод `main()` в листинге 16.3 показывает установку перехватывающего диспетчера безопасности и его влияние на ход выполнения.

```
public static void main(String[] args) {
    InterceptingSecurityManager secManager = new InterceptingSecurityManager();
    System.setSecurityManager(secManager);
    try {
        System.out.println("Run some logic...");
        System.exit(1);
    }
    catch (Throwable x) {
        if (x instanceof SecurityException)
            System.out.println("Intercepted System.exit()");
        else
            x.printStackTrace();
    }
    System.out.println("Run more logic...");
    secManager.setExitAllowed(true);
    System.out.println("Finished");
}
```

Листинг 16.3. Перехват `System.exit()`

Для простоты примера настоящая деловая логика, которая обычно вызывалась бы внутри блока `try`, заменена сообщением `Run some logic...`. Главное - перехватить класс `Throwable`, а не обычный `Exception`, поскольку перехваченный `System.exit()` сообщается как непроверяемое исключение. Выполнение метода `main()` из листинга 16.3 даёт следующий вывод:

```
Run some logic...
Intercepted System.exit()
Run more logic...
Finished
Process terminated with exit code 0
```

Вместо завершения JVM после вызова `System.exit()` внутри блока `try` программа продолжает работать, пока завершение не будет разрешено.

Реакция на завершение работы JVM с помощью перехватчиков

В предыдущем разделе показано, как перехватить программную попытку завершить JVM вызовом `System.exit()`. Иногда завершение JVM инициирует пользователь командой `kill` в Unix или сигналом `Ctrl+C` в Windows. JVM также может завершиться из-за выхода пользователя из системы или завершения работы ОС. Может ли программа Java перехватить сигнал завершения? Нет, перехватить его она не может, но способна отреагировать. Начиная с JDK 1.3 приложение может установить перехватчик завершения методом `addShutdownHook()` класса `java.lang.Runtime`. Перехватчики завершения - это инициализированные, но не запущенные экземпляры `java.lang.Thread`. При завершении JVM все потоки перехватчиков запускаются параллельно с остальными потоками JVM. Перехватчикам доступен весь API Java, но они должны учитывать уязвимое состояние JVM. Эти потоки не должны выполнять длительные операции и должны быть потокобезопасными. Нельзя рассчитывать на доступность системных служб, поскольку те сами могут находиться в процессе завершения. Подходящая задача для перехватчика - записать строку в файл журнала перед его закрытием и освободить прочие ресурсы, например открытые соединения с базой данных и файлы. Пример установки перехватчика завершения показан в листинге 16.4.

```
public static void main(String[] args) {
    Runtime.getRuntime().addShutdownHook(new Thread() {
        public void run() {
            handleJVMShutdown();
        }
    });
}

public static void handleJVMShutdown() {
    // Record the shutdown and close all resources
}
```

Листинг 16.4. Установка перехватчика завершения

Перехват методов динамическим заместителем

Иногда до и после вызова метода требуется выполнить некоторую обработку. Она может включать трассировку имени метода и значений параметров, измерение времени выполнения или даже предоставление альтернативной реализации. Предположим, вы разрабатываете графический редактор, в котором основные фигуры представлены интерфейсами `Line`, `Circle`, `Rectangle` и `Curve`. Добавить трассировку всех методов этих интерфейсов можно несколькими способами. Можно пройти по каждой функции и тщательно вставить вызовы трассировки. Либо можно написать класс-заместитель, реализующий каждый интерфейс, выводящий трассировку и делегирующий работу исходной реализации. Это более чистый подход, поскольку отладочный код отделён от реализации, но он требует большого объёма однообразного программирования. Интересная и не слишком известная альтернатива - динамический заместитель, использующий отражение для перехвата вызовов методов. Пакет `java.lang.reflection` предлагает интерфейс `InvocationHandler` и класс `Proxy`, которые вместе позволяют динамически создавать экземпляр, реализующий несколько интерфейсов, заданных во время выполнения. Такой подход не требует определять при компиляции интерфейсы, реализуемые заместителем. После создания заместитель можно привести к любому из интерфейсов, указанных при его создании, а любой вызов метода этих интерфейсов передаётся единственному методу заместителя, `invoke`. Единственное требование к классу динамического заместителя - реализация интерфейса `InvocationHandler`, который определяет метод `invoke`.

Разработаем для Chat динамический заместитель, который трассирует вызовы слушателя сообщений. Напомним, что Chat связывает главное окно с сервером RMI посредством интерфейса `MessageListener`. Хотя у `MessageListener` только один метод, этого достаточно для иллюстрации. Поместим динамический заместитель между экземплярами `MainFrame` и `ChatServer`, чтобы добавить трассировку вызовов методов. Создадим в пакете `covertjava.intercept` класс `TracingProxy`, реализующий интерфейс `InvocationHandler`. Заместитель будет делегировать вызовы методов настоящему объекту, поэтому конструктор примет целевой объект как параметр. Объявление класса `TracingProxy` и его конструктор показаны в листинге 16.5.

```
public class TracingProxy implements InvocationHandler {
    protected Object target;

    public TracingProxy(Object target) {
        this.target = target;
    }
    ...
}
```

Листинг 16.5. Объявление `TracingProxy`

Обратите внимание, что трассирующий заместитель принимает цель как тип `java.lang.Object`. Это ключевой момент: класс заместителя не связан с `MessageListener`, поэтому может использоваться с любым интерфейсом.

Теперь необходимо написать метод `invoke()` из интерфейса `InvocationTarget`. Он принимает три параметра: сам объект заместителя, метод и массив параметров метода. Наша реализация выводит имя метода, а затем делегирует вызов цели, переданной конструктору заместителя. Этот код показан в листинге 16.6.

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    Object result;
    try {
        System.out.println("Entering " + method.getName());
        result = method.invoke(target, args);
    }
    catch (InvocationTargetException e) {
        throw e.getTargetException();
    }
    finally {
        System.out.println("Leaving " + method.getName());
    }
    return result;
}
```

Листинг 16.6. Реализация метода `invoke()`

Теперь заместитель готов к испытанию. Чтобы увидеть его в действии, создадим экземпляр `TracingProxy`, инициализированный экземпляром `MainFrame` приложения Chat как целью. Затем создадим объект `java.lang.reflect.Proxy`, который динамически реализует интерфейс `MessageListener` и делегирует вызовы экземпляру трассирующего заместителя. Наконец, передадим заместитель отражения серверу Chat, приведя его к интерфейсу `MessageListener`. Соответствующий код Java показан в листинге 16.7.

```
public static void main(String[] args) throws Exception {
    ChatServer chatServer = ChatServer.getInstance();
    chatServer.setMessageListener(new MainFrame(false));

    TracingProxy listener = new TracingProxy(chatServer.getMessageListener());
    Object proxy = Proxy.newProxyInstance(
        chatServer.getClass().getClassLoader(),
        new Class[] {MessageListener.class},
        listener
    );
    chatServer.setMessageListener((MessageListener)proxy);

    MessageInfo messageInfo = new MessageInfo("localhost", "alex");
    chatServer.receiveMessage("Test message", messageInfo);
    System.exit(0);
}
```

Листинг 16.7. Использование динамического заместителя

Выполнение метода `main()` класса `TracingProxy` даёт следующий вывод:

```
C:\Projects\CovertJava\classes>java covertjava.intercept.TracingProxy
Received message from host localhost
Entering messageReceived
Leaving messageReceived
```

Таким образом, мы смогли перехватить вызов метода `messageReceived`, не реализуя интерфейс `MessageInfo`. Динамические заместители также полезны при разработке каркасов и инструментов, когда требуется взаимодействовать с классами, типы которых неизвестны во время компиляции. Вместо генерации и компиляции статических классов-заместителей каркасы могут использовать динамические заместители как связующий слой между компонентами.

Интерфейс профилировщика виртуальной машины Java

Многообещающим нововведением стал интерфейс профилировщика виртуальной машины Java (JVMPI), стандартизующий взаимодействие профилировщика с JVM. Впервые он был открыт в JDK 1.2.2, а затем расширен в JDK 1.4. Этот API представляет собой двусторонний интерфейс, определяющий, как виртуальная машина должна уведомлять агент профилировщика о происходящих внутри неё событиях, например запуске потоков, вызовах методов и выделении памяти. Он также определяет средства, с помощью которых профилировщик получает сведения о состоянии JVM и задаёт интересующие его события. Агент профилировщика работает внутри JVM, а все методы API являются функциями в стиле C, вызываемыми через JNI. Для доступа к API JVM необходимо запустить с параметром `-XrunProfilerLibrary`, где `ProfilerLibrary` - имя загружаемой машинной библиотеки. Несколько жаль, что интерфейса JVMPI на основе Java нет, а подробности реализаций на C и JNI выходят за рамки книги. Тем не менее ниже перечислены самые интересные события, которые можно перехватывать:

• `JVMPI_EVENT_CLASS_LOAD` - отправляется при загрузке класса.
• `JVMPI_EVENT_CLASS_LOAD_HOOK` - отправляется после загрузки данных класса загрузчиком, но до создания внутреннего представления класса. Это даёт профилировщику возможность декорировать или инструментировать байт-код.
• `JVMPI_EVENT_METHOD_ENTRY` - отправляется при входе в метод.
• `JVMPI_EVENT_METHOD_EXIT` - отправляется при выходе из метода.
• `JVMPI_EVENT_THREAD_START` - отправляется при запуске потока.
• `JVMPI_EVENT_THREAD_END` - отправляется после завершения потока.

Полное справочное описание JVMPI доступно по адресу java.sun.com.

Короткий тест

1. Почему и где в приложении важно использовать `java.lang.Throwable`?
2. Как перенаправить вывод системного потока ошибок в базу данных?
3. Как перехватить вызов `System.exit()`?
4. Как приложение Java, работающее как служба, может закрыть все соединения с базой данных при завершении работы компьютера?
5. Какие события можно получать через JVMPI?

Краткие итоги

• Хорошего способа перехватить поток управления в Java нет. JVMPI предоставляет самые широкие возможности вмешательства в выполнение, но требует программирования JNI.
• Системные ошибки сообщаются как необъявленные и могут быть перехвачены в виде экземпляров `java.lang.Throwable`.
• Стандартные системные потоки вывода и ошибок можно программно перенаправить в собственный `PrintStream`.
• Вызов `System.exit()` можно перехватить, установив собственный `SecurityManager`, запрещающий завершение, пока оно не разрешено явно.
• Приложения могут выполнять код при завершении JVM с помощью перехватчиков завершения. Эти перехватчики являются потоками, которые JVM запускает после получения сигнала завершения.
• JVMPI предоставляет огромные возможности управления средой выполнения, загрузкой классов и выполнением методов.

Глава 17. Понимание и модификация байт-кода

> «Каждое решение порождает новые проблемы».
>
> *Пятое следствие Мерфи*

В этой главе

• Основы байт-кода
• Просмотр файлов классов с помощью обозревателя байт-кода jClassLib
• Набор инструкций JVM
• Формат файла класса
• Инструментирование и генерация байт-кода
• Сравнение модификации байт-кода с AOP и динамическими заместителями
• Короткий тест
• Краткие итоги

Основы байт-кода

В главе 2, «Декомпиляция классов», был дан краткий обзор байт-кода и его назначения в Java. Как вам, несомненно, известно, байт-код является промежуточной ступенью между исходным и машинным кодом, обеспечивающей межплатформенное выполнение программ Java. Байт-код определён спецификацией виртуальной машины Java (java.sun.com), где также описаны понятия языка, формат файла класса, требования к виртуальной машине Java (JVM) и другие важные стороны языка программирования Java. Строгое соблюдение спецификации обеспечивает переносимость и повсеместное выполнение приложений, скомпилированных в байт-код. JVM, работающая поверх операционной системы, отвечает за предоставление среды выполнения и преобразование инструкций байт-кода Java в машинные инструкции.

Большинство представленных ранее в книге приёмов взлома требовало получить исходный код и манипулировать им для изменения поведения приложения. В этой главе мы будем работать на уровне байт-кода, а не исходного кода. Мы узнаем, как просматривать структуры данных файла класса, инструментировать (расширять) существующий байт-код и программно создавать новые классы. Изменение на уровне байт-кода даёт следующие преимущества:

• Не требуется получать исходный код или декомпилировать байт-код, а затем снова компилировать исходный код.
• Байт-код можно создавать или инструментировать загрузчиком классов на лету, по мере загрузки классов в JVM.
• Автоматизировать генерацию байт-кода проще и быстрее, чем генерацию исходного кода, поскольку требуется меньше этапов и не нужно запускать компилятор. Например, Hibernate создаёт код хранения данных для классов Java во время выполнения.
• Инструменты могут применять инструментирование байт-кода для добавления логики, которой не требуется присутствовать в исходных файлах. Например, некоторые реализации аспектно-ориентированного программирования (AOP) вставляют в байт-код собственные атрибуты и инструментируют методы для поддержки AOP.

Следующие два раздела дают краткое введение в те части спецификации JVM, которые относятся к байт-коду. Хотя полезно ознакомиться с работой JVM и форматом файла класса, для реализации представленных в этой главе приёмов это не строго обязательно. Если терпение не относится к вашим известным добродетелям, а чтение материала, похожего на спецификацию, сравнимо для вас с написанием пользовательской документации к своему коду, можете пропустить следующие два раздела и сразу перейти к разделу «Инструментирование и генерация байт-кода».

Просмотр файлов классов с помощью обозревателя байт-кода jClassLib

Обозреватель байт-кода, поставляемый с бесплатной библиотекой jClassLib, - превосходная графическая утилита для просмотра содержимого файла класса. На левой панели она показывает иерархическое представление структуры файла, а на правой - содержимое выбранного элемента. На рисунке 17.1 jClassLib отображает содержимое `SimpleClass` из пакета `covertjava.bytecode`.

Обозреватель байт-кода jClassLib не позволяет изменять файл класса, но прекрасно визуализирует структуры, представленные в следующих разделах. Полезный способ изучать байт-код - сопоставлять его инструкции с конструкциями и операторами исходного кода. Обозреватель также можно использовать для отладки генерации и инструментирования байт-кода, которыми мы займёмся в конце главы.

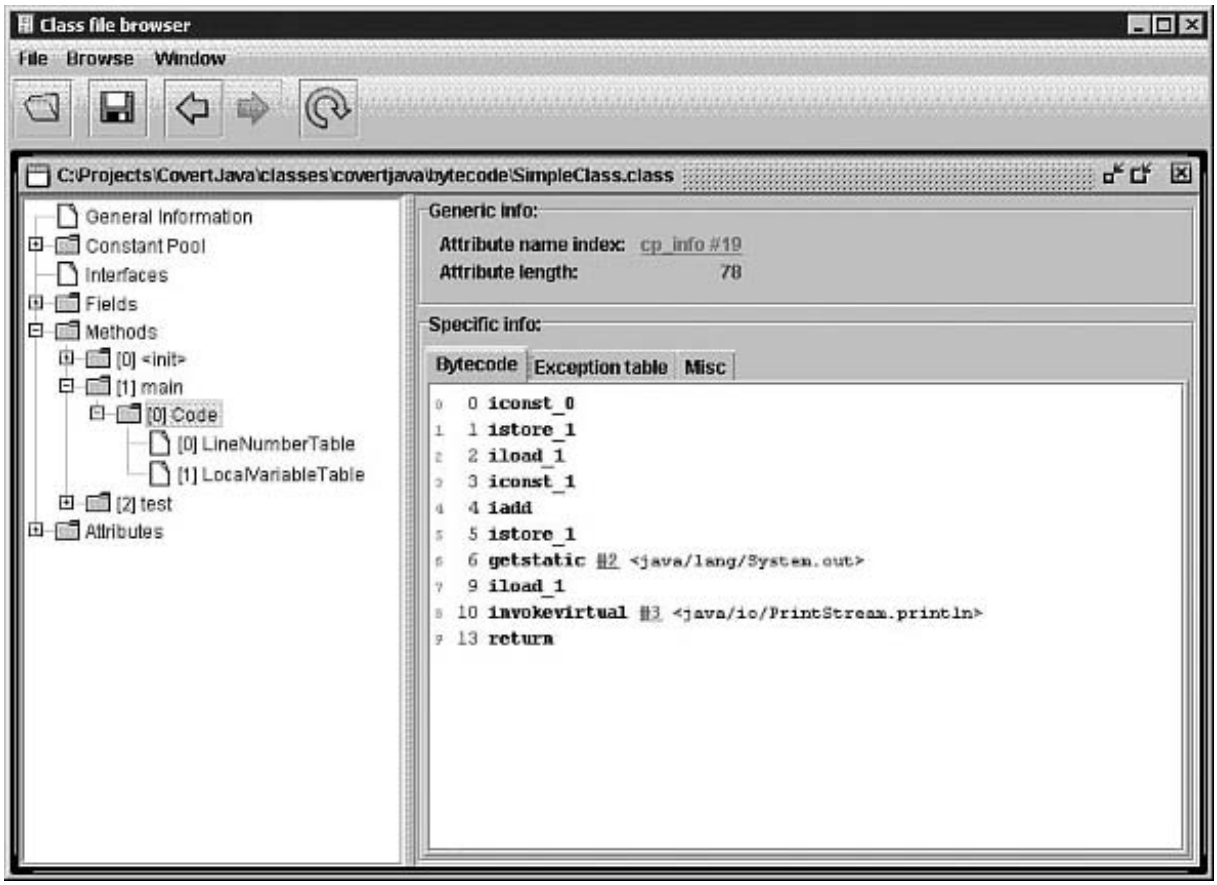

Рисунок 17.1. Обозреватель байт-кода jClassLib

Набор инструкций JVM

Исходные файлы Java компилируются в двоичные файлы классов определённого формата. Логика каждого метода Java представлена набором примитивных инструкций JVM, определённых её спецификацией. Инструкции JVM - базовые команды, подобные машинному коду. Каждая инструкция JVM состоит из кода операции (opcode), за которым следуют ноль или более операндов, представляющих параметры операции. В файле класса инструкции хранятся как двоичный поток, представляющий атрибут `Code` метода. Код операции занимает 1 байт, после которого могут следовать байты с данными операндов. Например, исходный код из листинга 17.1 представлен набором инструкций из листинга 17.2.

> **Истории с передовой.** Hibernate - бесплатная высокопроизводительная служба объектно-реляционного хранения и запросов для Java. Одно из главных преимуществ Hibernate - возможность прозрачно сохранять объекты Java. Вместо утомительных вызовов JDBC разработчики пишут файл XML с отображением объектов на схему базы данных, а Hibernate выполняет всю черновую работу. Служба хранения использует отражение и генерацию байт-кода во время выполнения, чтобы не мешать отладке в IDE и инкрементальной компиляции. Hibernate подчёркивает, что применение библиотеки Apache Byte Code Engineering Library, а позднее библиотеки генерации байт-кода CGLIB, для манипуляций с байт-кодом позволяет избежать накладных расходов API отражения Java.

```
int i = 0;
i = i + 1;
System.out.println(i);
```

Листинг 17.1. Пример исходного кода Java

```
0 iconst_0
1 istore_1
2 iinc 1 by 1
5 getstatic #21 <java/lang/System.out>
8 iload_1
9 invokevirtual #27 <java/io/PrintStream.println>
12 return
```

Листинг 17.2. Представление примера исходного кода в байт-коде

Большинство инструкций очень просты, и сопоставить их с представленным ими исходным кодом легко. Например, `iconst_0` задаёт целочисленную константу со значением 0, а `istore_1` сохраняет значение с вершины стека (в нашем случае 0) в локальную переменную, заданную индексом (в нашем случае `i`). Более интересный сценарий - вызов метода. Как видно из листингов, до вызова метода `println` имя статического поля класса (`System.out`) и значение параметра (`i`) помещаются в стек операндов. Подробные сведения об инструкциях можно получить из спецификации JVM, но это выходит за рамки книги. Полезно познакомиться с инструкциями и их операндами, хотя мы и будем применять каркас, предоставляющий уровень абстракции над байт-кодом. Для инструментирования и генерации байт-кода необходимо программно строить наборы инструкций, поэтому хотя бы базовое понимание набора инструкций и его соответствия Java обязательно.

Формат файла класса

Формат двоичного файла класса предписан спецификацией JVM. Он описывается рядом структур данных, представляющих сам класс, его методы, поля и атрибуты. Для манипуляций с байт-кодом необходимо изучить соглашения об именовании различных элементов и формат основных структур данных.

Дескрипторы полей и методов

Java поддерживает перегруженные методы, связывая метод с дескриптором, созданным на основе принимаемых параметров. Благодаря этому внутри `print(int i)` и `print(char ch)` хранятся как два разных метода. Преобразование имён выполняется по соглашению, предписанному спецификацией JVM, и поскольку в байт-коде хранятся преобразованные имена, здесь можно на него взглянуть.

Дескрипторы полей и методов кодируются на основе их типов. В таблице 17.1 приведены объявленные типы Java и соответствующие типы дескрипторов полей, используемые в байт-коде.

Объявленный тип	**Тип дескриптора**
`byte`	`B`
`char`	`C`
`double`	`D`
`float`	`F`
`int`	`I`
`long`	`J`
`short`	`S`
`boolean`	`Z`
Экземпляр `Classname`	`L<Classname>;`
`[]` (одно измерение массива)	`[`

Таблица 17.1. Коды типов полей

В таблице 17.2 приведены примеры объявлений Java и их дескрипторов в байт-коде.

Объявление типа	**Тип дескриптора**
`int number;`	`I`
`int[][] numbers;`	`[[I`
`Object reference;`	`Ljava.lang.Object;`

Таблица 17.2. Примеры типов дескрипторов

Дескрипторы методов создаются в следующем формате:

```
([<param1>[...<paramN>]])<return>
```

где:

• <param1> ... <paramN> - необязательные дескрипторы типов параметров;
• <return> - дескриптор возвращаемого типа или V, если метод возвращает void.

Например, метод, объявленный как

```
Integer getIntProperty(String propertyName, int defaultValue)
```

будет иметь дескриптор метода

```
(Ljava.lang.String;I)Ljava.lang.Integer;
```

Некоторые специальные методы имеют предопределённые имена. Статические инициализаторы называются <clinit>, а инициализаторы экземпляров и конструкторы - <init>.

Структура файла класса

Каждый класс Java определяется двоичным потоком, который обычно хранится в файле класса и состоит из 8-битных байтов. Содержимое потока описано псевдоструктурой из спецификации JVM, приведённой в листинге 17.3. Хотя сведений может показаться слишком много, структуры из этого и следующих разделов помогут понять последующую генерацию и инструментирование байт-кода.

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Листинг 17.3. Структура `ClassFile`

Для ясности спецификация JVM определяет псевдотипы u1, u2 и u4, представляющие соответственно беззнаковые типы размером 1, 2 и 4 байта. В таблице 17.3 перечислены поля структуры ClassFile и их значения.

Поле	Описание
`magic`	Маркер формата файла класса. Всегда имеет значение `0xCAFEBABE`.
`minor_version`, `major_version`	Версия JVM, для которой скомпилирован файл класса. JVM может поддерживать меньшие основные версии, но не выполняет большие.
`constant_pool_count`	Число элементов массива пула констант. Первый элемент зарезервирован для внутреннего применения JVM, поэтому допустимые значения `constant_pool_count` начинаются с 1.
`constant_pool[]`	Массив структур переменной длины, представляющих строковые константы, имена классов и полей и другие константы.
`access_flags`	Маска модификаторов, используемых в объявлениях класса или интерфейса. Допустимые модификаторы: `ACC_PUBLIC`, `ACC_FINAL`, `ACC_SUPER`, `ACC_INTERFACE` и `ACC_ABSTRACT`.
`this_class`	Индекс элемента массива `constant_pool`, описывающего данный класс.
`super_class`	Ноль или индекс элемента массива `constant_pool`, описывающего суперкласс данного класса. Для класса значение 0 означает, что суперклассом является `java.lang.Object`.
`interfaces_count`	Число родительских интерфейсов данного класса или интерфейса.
`interfaces[]`	Массив индексов элементов `constant_pool`, описывающих родительские интерфейсы данного класса.
`fields_count`	Число элементов массива полей.
`fields[]`	Массив структур переменной длины, описывающих поля, объявленные в данном классе.
`methods_count`	Число элементов массива методов.
`methods[]`	Массив структур переменной длины, описывающих методы, объявленные в данном классе, включая байт-код методов.
`attributes_count`	Число элементов массива атрибутов.
`attributes[]`	Массив структур переменной длины, объявляющих атрибуты файла класса. К стандартным относятся `SourceFile`, `LineNumberTable` и другие. JVM обязана игнорировать неизвестные ей атрибуты.

Таблица 17.3. Поля `ClassFile`

Пул констант заслуживает дополнительного внимания, поскольку часто используется другими структурами. Любая текстовая строка в классе Java независимо от её природы хранится в одном и том же пуле констант. Сюда входят имя класса, имена полей и методов, имена вызываемых классом классов и методов и строковые литералы внутри кода Java. Всякий раз, когда нужно использовать имя или строку, на неё ссылаются по индексу в пуле констант. Пул констант - массив структур `cp_info`, общий формат которых показан в листинге 17.4.

```
cp_info {
    u1 tag;
    u1 info[];
}
```

Листинг 17.4. Структура элемента пула констант

Настоящие элементы пула имеют структуру, соответствующую тегу. Например, строка задаётся структурой `CONSTANT_String`, а ссылка на поле - `CONSTANT_Fieldref`. Перечень структур и их содержимое можно найти в спецификации JVM.

Структура `ClassFile` использует ещё три структуры: `field_info`, `method_info` и `attribute_info`. `field_info` подобна `method_info`, поэтому в листинге 17.5 показана только структура `method_info`.

```
method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Листинг 17.5. Структура `method_info`

Значения полей `method_info` приведены в таблице 17.4.

Поле	Описание
`access_flags`	Маска модификаторов, описывающих доступность и свойства метода, включая `static`, `final`, `synchronized`, `native` и `abstract`.
`name_index`	Индекс элемента массива `constant_pool`, представляющего имя метода.
`descriptor_index`	Индекс элемента массива `constant_pool`, представляющего дескриптор метода.
`attributes_count`	Число элементов массива атрибутов.
`attributes[]`	Массив атрибутов метода. К атрибутам, определённым спецификацией JVM, относятся `Code` и `Exceptions`. Неизвестные JVM атрибуты игнорируются.

Таблица 17.4. Поля `method_info`

Атрибуты

Атрибуты используются в структурах `ClassFile`, `field_info`, `method_info` и `Code_attribute` для предоставления дополнительных сведений, зависящих от типа структуры. Например, атрибуты класса включают имя исходного файла и отладочные сведения, а атрибуты метода - байт-код и исключения. В листинге 17.6 показана структура `attribute_info`, а в таблице 17.5 перечислены её поля.

```
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

Листинг 17.6. Структура `attribute_info`

Поле	Описание
`attribute_name_index`	Индекс элемента `constant_pool`, представляющего имя атрибута.
`attribute_length`	Длина массива `attribute_info` в байтах.
`attribute_info`	Двоичное содержимое атрибута.

Таблица 17.5. Поля `attribute_info`

Компиляторам и постпроцессорам разрешено определять и именовать новые атрибуты при условии, что они не влияют на семантику класса. Например, реализации AOP могут использовать атрибуты байт-кода для хранения аспектов, определённых для класса.

Проверка байт-кода

Когда компилятор преобразует исходный код Java в байт-код, он тщательно проверяет синтаксис, применение ключевых слов и операторов и другие возможные ошибки. Это обеспечивает допустимость и безопасность созданного байт-кода. При загрузке класса в JVM выполняется упрощённое подмножество проверок, удостоверяющее, что файл класса имеет правильный формат и не был подделан. Например, средство проверки байт-кода убеждается, что первые 4 байта содержат магическое число, а атрибуты имеют правильную длину. Оно проверяет, что от финальных классов не наследуются, а поля и методы содержат правильные ссылки на пул констант, а также выполняет ряд других проверок.

Инструментирование и генерация байт-кода

Мы дошли до момента, когда наконец можно положить руки на клавиатуру и заняться кое-чем занятным. Теперь, когда вы достаточно знаете о байт-коде, можно реализовать два наиболее распространённых вида манипуляций с ним. Очевидно, что непосредственная работа с двоичным содержимым файла класса утомительна. Чтобы облегчить задачу, воспользуемся открытой библиотекой Apache Byte Code Engineering Library (BCEL).

Обзор BCEL

Домашняя страница BCEL находится по адресу jakarta.apache.org; там можно загрузить двоичный дистрибутив, исходный код и руководство. Библиотека предоставляет объектно-ориентированный API для работы со структурами и полями, из которых состоит класс. Она позволяет прочитать существующий файл класса и представить его иерархией объектов; преобразовать представление класса, добавляя поля, методы и двоичный код; программно создавать новые классы с нуля. Представление класса можно сохранить в файл или передать JVM как массив байтов для инструментирования и генерации на лету. BCEL даже поставляется с загрузчиком классов, который можно применять для динамического инструментирования или создания классов во время выполнения. Диаграмма основных классов BCEL показана на рисунке 17.2.

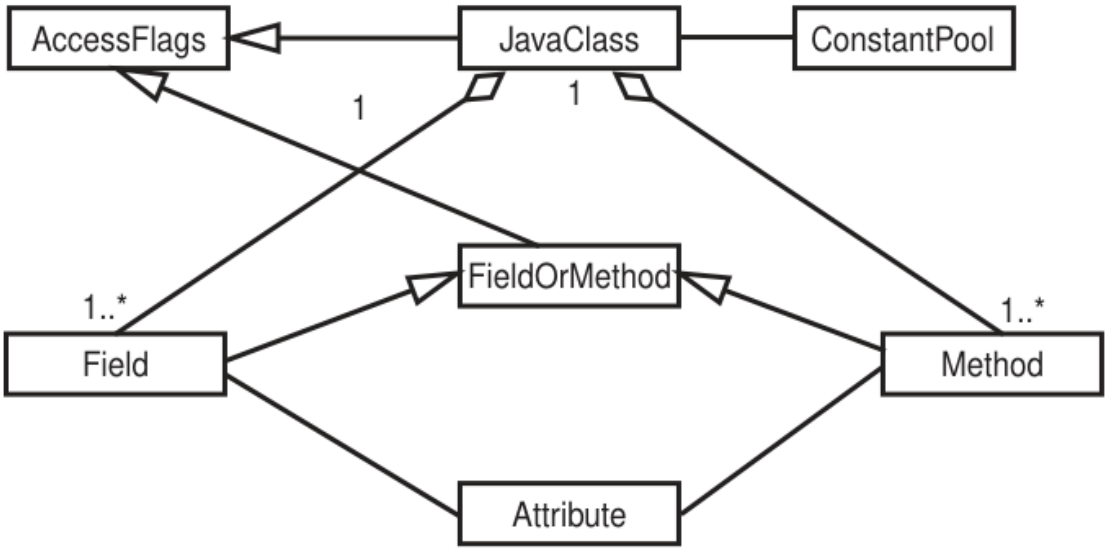

Рисунок 17.2. Диаграмма основных классов BCEL

В таблице 17.6 кратко описаны основные классы, которыми мы воспользуемся. Подробные сведения доступны в JavaDoc библиотеки BCEL.

Класс BCEL	Описание
`JavaClass`	Представляет существующий класс Java. Содержит поля, методы, атрибуты, пул констант и другие структуры данных класса.
`Field`	Представляет структуру `field_info`.
`Method`	Представляет структуру `method_info`.
`ConstantPool`	Представляет пул констант, содержащийся в классе.
`ClassGen`	Динамически создаёт новый класс. Может быть инициализирован существующим классом.
`FieldGen`	Динамически создаёт новое поле. Может быть инициализирован существующим полем.
`MethodGen`	Динамически создаёт новый метод. Может быть инициализирован существующим методом.
`ConstantPoolGen`	Динамически создаёт новый пул констант. Может быть инициализирован существующим пулом констант.
`InstructionFactory`	Создаёт инструкции для вставки в байт-код.
`InstructionList`	Хранит список инструкций байт-кода.
`Instruction`	Представляет инструкцию, например `iconst_0` или `invokevirtual`.

Таблица 17.6. Основные классы BCEL

Как видно, большинство классов прямо соответствуют терминам и структурам данных, определённым в спецификации JVM.

Инструментирование методов

Инструментирование - это вставка нового байт-кода или дополнение существующего байт-кода класса. Продукты, выдающие показатели производительности работающих приложений Java, используют инструментирование для сбора данных. Чтобы получить практический опыт, разработаем каркас, создающий журнал вызовов методов во время выполнения. Omniscient Debugger, рассмотренный в главе 9, «Взлом кода нестандартными отладчиками», применяет подобный приём для записи выполнения программы с целью последующего просмотра. Регистрация вызовов методов во время выполнения даёт подробный журнал кода, исполняемого JVM.

Для проверки реализации воспользуемся классом `SimpleClass` из пакета `covertjava.bytecode`, метод `main` которого показан в листинге 17.7.

```
public static void main(String[] args) {
    int i = 0;
    i = i + 1;
    System.out.println(i);
}
```

Листинг 17.7. Метод `main()` *класса* `SimpleClass`

Чтобы пример оставался простым, мы не будем писать полный каркас журналирования вызовов. Вместо этого ограничимся классом `InvocationRegistry` со статическим методом, показанным в листинге 17.8.

```
public static void methodInvoked(String methodName) {
    System.out.println("*** method invoked " + methodName);
}
```

Листинг 17.8. Точка входа в каркас журналирования методов

`methodInvoked()` - точка входа в каркас журналирования методов, используемая для регистрации вызова. Для каждого потока он может хранить стек вызовов методов, который сохраняется или выводится в конце работы приложения. Пока реализация просто выводит имя метода, показывая, что для этого метода был вызван каркас.

Теперь, когда основа заложена, можно приступить к реализации класса, инструментирующего байт-код методов. Назовём его `MethodInstrumentor`, а его метод `main()` будет принимать из командной строки имя класса и методов, которые требуется инструментировать. При выполнении `MethodInstrumentor` загрузит указанный класс, инструментирует методы, имена которых соответствуют заданному шаблону регулярного выражения, добавив вызов `InvocationRegistry.methodInvoked()`, а затем сохранит класс под новым именем. При выполнении новая версия класса должна регистрировать вызовы своих методов в реестре.

`MethodInstrumentor` находится в пакете `covertjava.bytecode`, и разрабатывать его мы будем сверху вниз. Метод `main()` класса `MethodInstumentor` показан в листинге 17.9.

```
public static void main(String[] args) throws IOException {
    if (args.length != 2) {
        System.out.println("Syntax: MethodInstrumentor " +
            "<full class name> <method name pattern>");
        System.exit(1);
    }

    JavaClass cls = Repository.lookupClass(args[0]);
    MethodInstrumentor instrumentor = new MethodInstrumentor();
    instrumentor.instrumentWithInvocationRegistry(cls, args[1]);
    cls.dump("new_" + cls.getClassName() + ".class");
}
```

Листинг 17.9. Метод `main()` *класса* `MethodInstrumentor`

Проверив синтаксис командной строки, `MethodInstrumentor` пытается загрузить заданный класс посредством класса `Repository` из BCEL. `Repository` ищет и загружает класс в пути классов приложения; это лишь один из множества способов загрузить класс через BCEL. По какой-то необъяснимой причине при ошибке BCEL возвращает `null`, а не выбрасывает исключение, но ради ясности кода мы не будем это проверять. После загрузки класса создаётся экземпляр `MethodInstrumentor` и вызывается его метод `instrumentWithInvocationRegistry()`

для выполнения преобразований. По завершении класс сохраняется в файл под новым именем. Рассмотрим реализацию `instrumentWithInvocationRegistry` в листинге 17.10.

```
public void instrumentWithInvocationRegistry(JavaClass cls,
    String methodPattern) {
    ConstantPoolGen constants = new ConstantPoolGen(cls.getConstantPool());
    Method[] methods = cls.getMethods();
    for (int i = 0; i < methods.length; i++) {
        // Instrument all methods that match the given criteria
        if (Pattern.matches(methodPattern, methods[i].getName())) {
            methods[i] = instrumentMethod(cls, constants, methods[i]);
        }
    }
    cls.setMethods(methods);
    cls.setConstantPool(constants.getFinalConstantPool());
}
```

Листинг 17.10. Реализация `instrumentWithInvocationRegistry`

Поскольку мы добавим вызов метода из другого класса, на него необходимо сослаться по имени. Напомним, что все имена хранятся в пуле констант, а значит, в существующий пул придётся

добавить новые константы. Для добавления элементов в структуры BCEL нужно использовать классы-генераторы с суффиксом `Gen` в имени. Код создаёт экземпляр `ConstantPoolGen`, изначально заполненный константами существующего пула, а затем перебирает все методы, используя мощь регулярных выражений, чтобы проверить, какие из них следует инструментировать. После обработки всех методов класс обновляется новыми методами и новым пулом констант. Собственно инструментирование выполняет метод `instrumentMethod()`, показанный в листинге 17.11.

```
public Method instrumentMethod(JavaClass cls, ConstantPoolGen constants,
    Method oldMethod) {
    System.out.println("Instrumenting method " + oldMethod.getName());
    MethodGen method = new MethodGen(oldMethod, cls.getClassName(), constants);
    InstructionFactory factory = new InstructionFactory(constants);
    InstructionList instructions = new InstructionList();

    // Append two instructions representing a method call
    instructions.append(new PUSH(constants, method.getName()));
    Instruction invoke = factory.createInvoke(

        "covertjava.bytecode.InvocationRegistry",
        "methodInvoked",
        Type.VOID,
        new Type[] {new ObjectType("java.lang.String")},
        Constants.INVOKESTATIC
    );
    instructions.append(invoke);
    method.getInstructionList().insert(instructions);
    instructions.dispose();
    return method.getMethod();
}
```

Листинг 17.11. Реализация `instrumentMethod()`

Как видно, `instrumentMethod()` программно создаёт инструкции байт-кода, соответствующие вызову метода. Самый простой способ выбрать правильные инструкции JVM и их параметры - сначала написать код на Java, скомпилировать, а затем с помощью чего-либо наподобие обозревателя jClassLib посмотреть, как он преобразуется в байт-код. После этого соответствующий байт-код можно построить объектами BCEL.

Первым делом `instrumentMethod()` создаёт объект `MethodGen`, хранящий новый байт-код. Затем создаются фабрика инструкций и список для их хранения. Если вы внимательно читали главу и экспериментировали с обозревателем байт-кода jClassLib, то, возможно, помните, что вызов метода Java представлен несколькими инструкциями байт-кода. Сначала параметры метода нужно поместить в стек операндов, а затем передать управление методу инструкцией `invokevirtual` (пример байт-кода вызова метода см. в листинге 17.2). Именно это необходимо вставить в код метода перед существующим байт-кодом. При непосредственной работе с байт-кодом пришлось бы вставить в пул две константы: `covertjava.bytecode.InvocationRegistry` для имени класса и `methodInvoked` для имени метода. К счастью, BCEL делает это за нас, поскольку высокоуровневые классы `InstructionFactory` и `PUSH` автоматически добавляют константы в пул. После создания инструкции добавляются в список. По завершении генерации кода список вставляется в инструкции создаваемого метода, а структура метода возвращается.

Чтобы проверить инструментирование, скомпилируйте классы и запустите `MethodInstrumentor` для `SimpleClass.class` следующей командой:

```
java covertjava.bytecode.MethodInstrumentor covertjava.bytecode.SimpleClass .*
```

В текущем каталоге должен появиться новый файл класса new_covertjava.bytecode.SimpleClass.class. Скопируйте его в каталог classes, заменив существующий файл SimpleClass.class, а затем выполните метод main() класса SimpleClass. Если всё работает, в консоли появится:

```
C:\Projects\CovertJava\classes>java covertjava.bytecode.SimpleClass
*** method invoked main
1
```

Как видно, инструментированный класс начинает с вызова InvocationRegistry, выводящего первую строку, а затем выполняет собственное тело, которое выводит 1.

Генерация классов

Наша вторая задача - научиться программно создавать новый класс. Как упоминалось ранее, это полезно для связующего программного обеспечения и каркасов, желающих избежать генерации исходного кода. В примере создадим генератор объекта-значения, который содержит все поля заданного класса, но не содержит методов. Объект-значение - распространённый шаблон проектирования распределённых приложений для передачи данных по сети. Следует признать, что наш генератор создаст очень грубую версию объектов-значений, но мы немного усложним задачу, обеспечив генерацию только полей, значения которых должны сохраняться.

Вновь используем SimpleClass как подопытного кролика. SimpleClass определяет пять полей, показанных в листинге 17.12.

```
public int number;
protected String name;
private Thread myThread;
static String className;
transient String transientName;
```

Листинг 17.12. Поля SimpleClass

Напишем в пакете covertjava.bytecode класс ClassGenerator, принимающий два параметра командной строки: полностью определённое имя класса и шаблон регулярного выражения для имён копируемых полей. Метод main() класса ClassGenerator показан в листинге 17.13.

```
public static void main(String[] args) throws IOException {
    if (args.length != 2) {
        System.out.println("Syntax: ClassGenerator " +
            "<full class name> <field name pattern>");
        System.exit(1);
    }

    JavaClass sourceClass = Repository.lookupClass(args[0]);
    ClassGenerator generator = new ClassGenerator();
    JavaClass valueClass = generator.generateValueObject(sourceClass, args[1]);
    valueClass.dump(valueClass.getClassName() + ".class");
}
```

Листинг 17.13. Метод main() *класса* ClassGenerator

Как и в MethodInstrumentor, реализация проверяет синтаксис командной строки, загружает класс, а затем вызывает метод generateValueObject(), показанный в листинге 17.14.

```
public JavaClass generateValueObject(
    JavaClass sourceClass,
    String fieldPattern)
{
    String newName = sourceClass.getClassName() + "Value";
    ClassGen classGen = new ClassGen(
        newName,
        "java.lang.Object",
        newName,
        Constants.ACC_PUBLIC | Constants.ACC_SUPER,
        new String[] { "java.io.Serializable" });

    Field[] fields = sourceClass.getFields();
    for (int i = 0; i < fields.length; i++) {
        if (Pattern.matches(fieldPattern, fields[i].getName())) {
            int skipFlags = Constants.ACC_STATIC | Constants.ACC_TRANSIENT;
            if ((fields[i].getAccessFlags() & skipFlags) == 0) {
                fields[i].setAccessFlags(Constants.ACC_PUBLIC);
                addField(classGen, fields[i]);
            }
        }
    }
    return classGen.getJavaClass();
}
```

Листинг 17.14. Метод `generateValueObject()` *класса* `ClassGenerator`

Сначала реализация создаёт экземпляр `ClassGen`, представляющий создаваемый класс. Его имя совпадает с именем класса-параметра с добавленным суффиксом `Value`. Он расширяет `java.lang.Object` и реализует `java.io.Serializable`. Затем реализация перебирает поля класса-параметра в поисках имён, соответствующих заданному критерию. С помощью битовой маски реализация исключает статические и временные поля и копирует подходящие поля в создаваемый класс. Для простоты модификатор доступа создаваемого поля задаётся как `public`. После завершения генерации представление класса возвращается вызывающей стороне, которая сохраняет его на диске.

Выполнение `ClassGenerator` для `SimpleClass` создаёт в текущем каталоге файл `covertjava.bytecode.SimpleClassValue.class`. В листинге 17.15 показана декомпилированная версия класса.

```
package covertjava.bytecode;

import java.io.Serializable;

public class SimpleClassValue
    implements Serializable
{
    public int number;
    public String name;
    public Thread myThread;
}
```

Листинг 17.15. Декомпилированная версия класса `SimpleClassValue`

Вуаля! Для `SimpleClassValue` созданы все подходящие поля `SimpleClass`.

Библиотека ASM

Новый открытый проект, набирающий популярность, - библиотека манипуляций с байт-кодом ASM, размещённая по адресу asm.objectweb.org. Она предназначена для тех же задач, что и BCEL, но заявляет значительно более высокую производительность благодаря иному подходу к реализации. BCEL создаёт полное дерево объектов, представляющее двоичный файл класса вплоть до отдельных инструкций байт-кода. Поэтому для одного файла класса потенциально могут быть созданы сотни объектов, что приводит к снижению производительности. Хотя объект для каждого атрибута файла класса удобен, при манипуляциях с байт-кодом во время выполнения такой подход может стать дорогим, если инструментируются тысячи классов.

ASM использует шаблон проектирования «посетитель», чтобы не создавать объекты без необходимости. Предоставляемый каркасом анализатор класса вызывает пользовательский класс-посетитель, передавая данные методов и полей как параметры. Для большинства параметров реализация посетителя просто передаёт их следующему посетителю, сохраняя данные в двоичной форме. Для тех полей или методов, которые нужно изменить, реализация посетителя получает от каркаса объектное представление, а затем манипулирует объектом. Благодаря этому большая часть байт-кода остаётся в двоичной форме, а накладные расходы на производительность минимальны.

Если минимальные накладные расходы инструментирования на производительность важны, ASM лучше BCEL. Если выше ценятся ясность и простота реализации, я рекомендую BCEL.

Сравнение модификации байт-кода с AOP и динамическими заместителями

Теперь, когда вы научились модифицировать байт-код, можно сравнить этот приём с другими подходами к расширению возможностей во время выполнения. В главе 16, «Перехват потока управления», были представлены динамические заместители, позволяющие перехватывать методы любого интерфейса без статической реализации этого интерфейса. Хотя динамические заместители просто писать и легко использовать, их главный недостаток в том, что они работают только с интерфейсами, а не с классами, и требуют явного создания в вызывающем коде. Так, чтобы использовать динамический заместитель с Chat, нам пришлось вызвать метод `setMessageListener()` класса `ChatServer` и установить заместитель. Если бы исходного кода Chat не было, сделать это без декомпиляции не удалось бы. Изменять код приложения допустимо во время разработки, но это не подходит для стороннего кода или интеграции во время выполнения. В отличие от динамического заместителя, модификация байт-кода не требует менять модифицируемый код при компиляции.

AOP - развивающаяся технология добавления сквозных свойств объектам и методам - является чистым и хорошо структурированным дополнением к традиционному программированию. С помощью аспектов легко добавить такие возможности, как трассировка вызовов методов либо предварительная и последующая обработка. AOP чётко отделяет реализацию логики программы от инфраструктурных задач, например трассировки, профилирования, безопасности и прочих. Аспекты определяются в отдельных файлах, компилируемых и обрабатываемых вместе с кодом приложения. Реализации AOP используют инструментирование байт-кода для вставки дополнительного поведения. В этом они больше похожи на рассмотренную в главе модификацию байт-кода, чем на динамические заместители. AOP - высокоуровневый подход, которому недостаёт гибкости непосредственной инженерии байт-кода. Когда это уместно, аспекты могут быть самым простым способом добавить скрытую логику в существующее приложение.

Короткий тест

1. По каким причинам манипулируют байт-кодом?
2. Что такое код операции и как операнды передаются инструкции байт-кода?
3. Как выглядел бы дескриптор метода Java с именем `getCount()`, объявленного как `public Object[] getObjects(String name, char type)`?
4. Из каких структур состоит файл класса?
5. Какие основные классы BCEL используются для инструментирования или генерации класса?
6. Какой атрибут метода необходимо изменить для инструментирования его байт-кода?

Краткие итоги

• Манипуляции с байт-кодом полезны для генерации кода, инструментирования существующих классов и расширения поведения классов без изменения их исходного кода.
• Формат файла класса Java и допустимые инструкции определены спецификацией JVM.
• Логика каждого метода Java представлена набором примитивных инструкций JVM, являющихся базовыми командами и весьма похожих на машинный код.
• Двоичный формат файла класса представлен псевдоструктурами, определёнными в спецификации JVM и включающими данные о классе, полях, методах, атрибутах и других свойствах.
• Apache Byte Code Engineering Library (BCEL) предоставляет объектно-ориентированный API для работы со структурами и полями, из которых состоит класс.
• Инструментирование - это вставка нового байт-кода или дополнение существующего байт-кода класса.

Глава 18. Полный контроль с помощью исправления машинного кода

> «У каждого человека есть замысел, который не сработает».
>
> *Закон Хоу*

В этой главе

- Зачем и когда исправлять машинный код
- Использование машинного кода в виртуальной машине Java
- Общие подходы к исправлению машинных методов
- Исправление машинного кода на платформе Windows
- Исправление машинного кода на платформах Unix
- Короткий тест
- Краткие итоги

Зачем и когда исправлять машинный код

Мы рассмотрели различные приёмы замены, исправления и обратной разработки классов Java. Все эти приёмы требуют работы на уровне исходного или байт-кода, поэтому ограничивают наши возможности высокоуровневым миром Java. Виртуальная машина Java (JVM) взаимодействует с операционной системой (ОС) посредством машинных библиотек, а значит, все низкоуровневые операции написаны не на Java и ими нельзя манипулировать представленными приёмами. Например, `System.currentTimeMillis()` является машинным методом, а все методы `ClassLoader` делегируют настоящее определение класса машинному методу `defineClass0`. Хотя исправить класс Java обычно проще и чище, в некоторых случаях остаётся только исправлять машинный код. В этой главе представлены несколько низкоуровневых приёмов исправления машинного кода, которые вместе с предыдущими приёмами дают полный контроль над JVM.

Прежде чем пачкать руки машинными исправлениями, я хотел бы отметить два важных момента. Первый касается законности предстоящей работы. Как уже говорилось в книге, вы сами обязаны проверить, не запрещены ли обратная разработка и исправление лицензионным соглашением продукта, с которым вы работаете. Кража чужой интеллектуальной собственности не только незаконна, но и неэтична, поэтому я настоятельно рекомендую применять представленные приёмы только с добрыми намерениями. Второй момент: работа с машинным кодом требует уверенного знания языка C, базового понимания машинных инструкций и знакомства с форматами двоичных файлов. На разных платформах двоичные файлы имеют разные форматы, и даже два разных компилятора могут создавать разные исполняемые файлы для одной платформы. Например, объектные файлы компилятора Microsoft C отличаются от файлов компилятора Borland C. Исправление двоичного кода требует вставлять машинные инструкции в существующий машинный код и манипулировать двоичным файлом. Это похоже на путешествие по неизведанным водам: будьте готовы к трудностям и не ожидайте, что всё заработает с первого раза. Отсутствие общего, чётко определённого формата и сложность работы с необработанными машинными инструкциями приводят к нехватке хороших инструментов, так сильно помогавших нам прежде. Например, ни один декомпилятор не способен получить код C из двоичного исполняемого файла.

Для этой главы требуются:

• понимание языка C;
• умение писать и компилировать машинные библиотеки для целевой платформы;
• базовое знание машинных инструкций и языка ассемблера;
• некоторое знакомство с машинным интерфейсом Java (JNI).

Использование машинного кода в виртуальной машине Java

Большая часть кода, выполняющегося внутри JVM, включая основные классы, написана на Java. Это совершенно разумно, поскольку Java чиста, безопасна и не зависит от платформы. Однако в какой-то момент JVM требуется взаимодействовать с оборудованием, для чего она полагается на ОС. Низкоуровневые операции, например чтение блока байтов с жёсткого диска или создание сетевого сокета, передаются машинным библиотекам, выполняющим специфичные для ОС вызовы. На рисунке 14.1 в главе 14, «Управление загрузкой классов», была показана упрощённая схема загрузки JVM классов и машинного кода. Чаще всего машинные библиотеки просто платформенно-зависимым образом делегируют вызов операционной системе. Машинные библиотеки для Java можно писать только на языке C, а доступ к ним осуществляется через JNI.

Обзор JNI

Для межплатформенности Java должна использовать уровень абстракции между собой и операционной системой. Этот уровень реализован в наборе машинных библиотек, доступных через JNI. JNI - спецификация, описывающая определение машинных методов в Java и предоставление их реализации в библиотеках C. Иными словами, JNI задаёт договор между классами Java и машинными библиотеками.

Сторона договора Java проста: чтобы объявить машинный метод, достаточно добавить к его объявлению ключевое слово `native` и завершить объявление точкой с запятой. Предположим, программе Java нужно узнать параметры памяти, например общий объём физической и виртуальной памяти и объём доступной физической и виртуальной памяти локального компьютера. Класс `java.lang.Runtime` способен сообщить параметры памяти только для JVM, но не общие свойства памяти, поэтому придётся обратиться к ОС машинным вызовом. Для этого напишем класс Java `OSMemoryInfo` с набором машинных методов. Вот объявление метода, возвращающего общий объём физической памяти:

```
public native static long getPhysicalTotal();
```

После объявления метод можно скомпилировать и использовать в других классах Java. Попытка выполнить его приводит к `java.lang.UnsatisfiedLinkError`, поскольку реализация `getPhysicalTotal()` ещё не предоставлена. Для выполнения машинных методов объявляющий их класс Java должен загрузить машинную библиотеку с реализацией метода. Машинные библиотеки зависят от ОС, поэтому для каждой платформы, где должно работать приложение, необходима отдельная версия. Библиотека загружается только по имени, поскольку расширение зависит от платформы. В Windows имена библиотек заканчиваются на `.dll`, а в Unix - на `.so`. В листинге 18.1 показана загрузка библиотеки `OSMemoryInfo`.

```
public class OSMemoryInfo {
    static {
        try {
            System.loadLibrary("OSMemoryInfo");
        } catch (Exception x) {
            System.err.println("Error while loading native library");
            x.printStackTrace(System.err);
            System.exit(1);
        }
    }
    ...
}
```

Листинг 18.1. Загрузка машинной библиотеки из класса Java

Библиотеку загружает статический инициализатор, выполняемый при первой загрузке класса в JVM. Этот шаг завершает договор со стороны Java и приводит нас к стороне машинного кода.

Чтобы выполнить класс `OSMemoryInfo`, JVM должна получить библиотеку с реализациями всех машинных методов. Расположение библиотеки определяется платформенно-зависимым путём поиска. В Windows путь включает текущий каталог и каталоги, заданные переменной среды `PATH`. В Unix путь поиска определяется переменной среды, имя которой зависит от разновидности Unix. Например, в Solaris она называется `LS_LIBRARY_PATH`, а в HP UX - `SH_LIB_PATH`. Имя машинной библиотеки также зависит от ОС. В Windows наша библиотека называлась бы `OSMemoryInfo.dll`, а в Unix - `OSMemoryInfo.so`. Библиотека должна экспортировать функции, соответствующие имени и синтаксису объявления машинных методов класса Java. JNI задаёт отображение типов C на типы Java и предоставляет развитые механизмы доступа к объектам Java, выбрасывания исключений и манипуляций с типами данных. Например, функция C, реализующая показанный ранее метод Java `getPhysicalTotal()`, должна объявляться так:

```
JNIEXPORT jlong JNICALL
Java_covertjava_nativecode_OSMemoryInfo_getPhysicalAvail(JNIEnv *, jclass);
```

Пример реализации JNI

Учиться на примере эффективнее всего, поэтому поработаем с классом `OSMemoryInfo` из предыдущего раздела. Напомним, что он предназначен для получения сведений о памяти от операционной системы через JNI. В нём четыре машинных метода, возвращающих общий и доступный объёмы физической и виртуальной памяти. Все методы имеют синтаксис, показанный в листинге 18.1, а полный исходный код класса находится в `CovertJava/src/covertjava/nativecode/OSMemoryInfo.java`.

Проще всего получить правильный синтаксис функций C, соответствующих машинным методам Java, с помощью утилиты `javah`. Она создаёт заголовочный файл C на основе предоставленного файла класса Java. Для каждого машинного метода класса Java `javah` создаёт сигнатуру функции в выходном заголовочном файле C. Выполнение `javah` для класса `covertjava.bytecode.OSMemoryInfo` создаёт файл `covertjava_nativecode_OSMemoryInfo.h`, который также находится в каталоге `CovertJava/src/covertjava/nativecode`. Уделите минуту объявлениям функций и тому, как типы данных Java отображаются на типы C.

Следующий шаг - написать тела четырёх функций, объявленных в `covertjava_nativecode_OSMemoryInfo.h`. Для краткости рассмотрим только реализацию Windows, поскольку реализация Unix отличается лишь функцией, вызываемой в ОС. Все четыре функции используют одну функцию API Win32, `GlobalMemoryStatusEx`, возвращающую множество сведений о памяти ОС. Тела функций написаны в `OSMemoryInfo.c`, расположенном в каталоге `CovertJava/src/covertjava/nativecode`. В листинге 18.2 показана реализация `Java_covertjava_nativecode_OSMemoryInfo_getPhysicalTotal()`.

```
JNIEXPORT jlong JNICALL
Java_covertjava_nativecode_OSMemoryInfo_getPhysicalTotal
(JNIEnv *env, jclass cls)
{
    MEMORYSTATUSEX memStat;
    memStat.dwLength = sizeof (memStat);
    if (GlobalMemoryStatusEx(&memStat) == 0 && (*env) != 0) {
        jclass exceptionCls = (*env)->FindClass(env, "java/lang/Exception");
        char msg[100];
        sprintf(msg,
            "Failed to get memory information from the OS, error code %li",
            (long)GetLastError());
        if (exceptionCls != 0)  /* Raise Java exception */
            (*env)->ThrowNew(env, exceptionCls, msg);
        return -1;
    }
    return (jlong) (int) memStat.ullTotalPhys;
}
```

Листинг 18.2. Машинная реализация `getPhysicalTotal()`

Здесь нет ничего сложного: всего лишь вызов функции Win32, проверка ошибки и возврат результата. В духе идеологии Java созданная нами функция C выбрасывает `java.lang.Exception`, если вызов API Win32 заканчивается неудачей.

После написания тел функций можно собрать динамически подключаемую библиотеку Windows (DLL). Я буду использовать компилятор MSVC, который фактически бесплатно поставляется при загрузке Windows SDK и .Net SDK. Файл makefile для сборки DLL находится в каталоге `CovertJava/build`, а пакетный файл `CovertJava/bin/build_native.bat` запускает `nmake.exe`. Вы можете выбрать любой компилятор и способ сборки, но я рекомендую компилятор Microsoft по причинам, изложенным далее. Если хотите пересобрать машинные библиотеки, обязательно обновите все пути внутри `build_native.bat`.

Теперь можно выполнить метод `main()` класса `OSMemoryInfo`, который выводит значения, полученные от машинных методов. Выполнение файла `CovertJava/bin/OsMemoryInfo.bat`, вызывающего метод `main()`, дало на моём компьютере следующий результат:

```
C:\Projects\CovertJava\bin>OsMemoryInfo.bat
Total     Physical Memory: 535121920
Available Physical Memory: 199958528
Total     Virtual  Memory: 2147352576
Available Virtual  Memory: 1960931328
```

Теперь у нас есть работающая реализация JNI, с которой можно экспериментировать.

Общие подходы к исправлению машинных методов

Зная основные принципы взаимодействия кода Java с машинным кодом и архитектуру JNI, можно рассмотреть способы переопределения машинных функций. Как и при исправлении байт-кода, цель состоит в перехвате вызова машинного метода и предоставлении собственной реализации. Исправление должно быть прозрачным для вызывающей стороны и не требовать изменений клиентского кода Java. Рассмотрим три подхода с их достоинствами и недостатками.

Исправление объявления метода Java

Самое простое решение - исправить класс Java, объявляющий метод: удалить ключевое слово `native` и заменить его реализацией Java. Реализация может делегировать работу вспомогательному классу с настоящей логикой метода. Несмотря на простоту, этот способ наиболее эффективен и должен быть первым выбором. Поскольку все изменения выполняются на уровне Java, не нужно погружаться в программирование C и манипуляции двоичными файлами. Сложность возникает, если машинный вызов действительно нужен, но необходимо изменить часть его логики. Предположим, появилось новое требование создавать пользователя ОС в группе `Users`, а не `Administrators`. Здесь не обойтись без машинного метода, взаимодействующего с ОС. Но даже тогда исходный метод Java можно исправить, сделав немашинным, а затем заставить его вызвать машинный метод. Машинный метод реализуется в собственной машинной библиотеке под другим именем и создаёт пользователя на уровне ОС. Исправление объявления неприменимо лишь тогда, когда лицензионное соглашение запрещает обратную разработку классов Java, но не ограничивает изменение машинных библиотек.

Подмена машинных библиотек

Второй подход - заменить исходную машинную библиотеку подставной, экспортирующей те же функции. Подставные функции делегируют работу исходным, если только не требуется альтернативная реализация. Подставная библиотека действует как умный заместитель исходной и способна выполнять предварительную, последующую обработку и полностью переопределять вызовы методов. Этот подход хорошо работает, если функций в библиотеке немного или исправлять нужно большинство экспортируемых методов. Поскольку вся работа выполняется на C, это сравнительно простой подход, не требующий менять ни классы Java, ни двоичный машинный код. Если экспортируемых функций много, написание подставной библиотеки становится утомительным. Как и при исправлении объявления метода Java, возможна проблема, когда нужно сохранить часть логики исходного машинного метода. Подход практически следует принципу «всё или ничего»: либо вы делегируете работу исходному методу, либо нет.

Исправление машинного кода

Помните один из вопросов, над которыми мы размышляли в главе 15, «Замена и исправление основных классов Java»? Он звучал так: «Что делать, когда мы перепробовали все пути, но потерпели неудачу?» Не думаю, что эту фразу станут цитировать в Интернете, но в некотором смысле именно об этом вся книга. Два предыдущих подхода дают чистые и сравнительно простые решения для исправления машинного кода, но не исполняют обещание «полного контроля». Для полного контроля необходимо уметь взламывать машинные библиотеки и исправлять код подобно тому, как мы работали с байт-кодом. Третий подход делает именно это: он основан на изучении двоичного формата библиотеки, поиске изменяемого машинного кода и его исправлении новой логикой. Это нелёгкий путь, поэтому сначала я рекомендую два первых подхода. Исправление машинного кода зависит от платформы и требует глубокого понимания формата исполняемых файлов, языка ассемблера и адресации процессора. Но и награда велика. Изучаемый приём применим к любому исполняемому файлу, не только библиотекам JNI. Он также даёт представление о форматах исполняемых файлов и о том, как операционная система загружает и выполняет программы. В следующих разделах рассматривается исправление машинного кода в Windows и Unix.

Исправление машинного кода на платформе Windows

Для понимания этого раздела необходимы базовые знания языка ассемблера и некоторое знакомство с форматом Portable Executable. Взлом и исправление довольно популярны среди игроков и студентов, поэтому существует множество утилит, значительно упрощающих задачу в Windows. Вместо ручного редактирования двоичного кода и вставки новых машинных инструкций мы можем поручить низкоуровневое исправление утилитам и библиотекам.

Формат Portable Executable

Формат Windows Portable Executable (PE) отчасти основан на общем формате объектных файлов Unix (COFF). Он описывает двоичную структуру исполняемого файла, способного работать в любой совместимой с Win32 ОС. К исполняемым относятся файлы EXE, DLL, SCR, VxD и других типов. По структуре файл PE во многом подобен архиву JAR или Zip, содержащему другие файлы или разделы. В PE есть заголовок DOS, заголовок PE и таблица разделов, за которой следует ряд разделов, представляющих различные ресурсы, например текст, данные и ресурсы пользовательского интерфейса. В таблице 18.1 показана структура файла PE.

Элемент	Описание
Заголовок DOS MZ	Обеспечивает обратную совместимость, чтобы при запуске в MS-DOS файл распознавался как допустимый исполняемый.
Заглушка DOS	Небольшая встроенная программа, обычно просто выводящая строку о необходимости запустить файл в Win32.
Заголовок PE	Содержит различные сведения о части PE файла, например число разделов и адреса точек входа.
Таблица разделов	Массив структур, описывающих каждый раздел. Структуры содержат такие сведения, как атрибут раздела, смещение в файле и виртуальное смещение.
Раздел `.text`	Содержит двоичный код программы.
Раздел `.data`	Содержит инициализированные данные.
Раздел `.idata`	Содержит таблицу импорта.
Раздел `.edata`	Содержит таблицу экспорта.
Отладочные символы	Различные отладочные сведения, например номера строк.

Таблица 18.1. Структура файла PE

Прекрасный способ изучить внутреннюю структуру переносимого исполняемого файла - открыть его в утилите PE Explorer. Это хорошо написанная условно-бесплатная программа, показывающая в графическом окне заголовки, разделы и содержимое известных разделов PE. В PE Explorer также входит дизассемблер для изучения машинного кода файла. Бесплатную ознакомительную версию можно загрузить с heaventools.com. Например, загрузив созданный ранее `OSMemoryInfo.dll` в PE Explorer, можно увидеть разделы и экспорт DLL. В экспорте видно, что DLL предоставляет четыре функции с преобразованными именами. Функция `Java_covertjava_nativecode_OSMemoryInfo_getPhysicalTotal` экспортируется как `Java_covertjava_nativecode_OSMemoryInfo_getPhysicalTotal@8`. Следуя соглашению `__stdcall`, компилятор C автоматически добавил ко всем функциям знак `@` и число байтов, занимаемых параметрами в стеке.

Поскольку нас интересует исправление логики функции, необходимо просмотреть соответствующий ей машинный код. Исходный код C компилируется непосредственно в двоичный машинный код. В отличие от байт-кода Java, который должен дополнительно компилироваться или интерпретироваться JIT-компилятором, машинный код выполняется процессором напрямую. Прямое следствие: скомпилированный исполняемый файл работает только на той архитектуре процессора, для которой собран. Косвенное следствие: простого способа декомпилировать машинный код обратно в исходный нет. Они очень различаются; стандарта представления конструкций языка C машинными инструкциями не существует, а каждый компилятор выполняет собственные оптимизации, ещё больше затрудняющие декомпиляцию. Поэтому единственный способ обратной разработки двоичных исполняемых файлов - работать на уровне языка ассемблера. Ассемблер является читаемым человеком представлением машинных инструкций. Он очень примитивен, но его код непосредственно соответствует тому, как процессор будет его выполнять. Мы не станем писать код на ассемблере, но если вы хотите узнать о нём больше, возьмите книгу на Amazon.com или прочитайте документацию в Интернете. Для архитектур Intel я рекомендую книгу Кипа Р. Ирвина *Assembly Language for Intel-Based Computers* (Prentice Hall, ISBN: 0130910139).

Попробуем найти внутри двоичного файла код функции `Java_covertjava_nativecode_OSMemoryInfo_getPhysicalTotal` посредством PE Explorer. Если ещё не сделали этого, загрузите, установите и запустите PE Explorer, затем загрузите в него

OSMemoryInfo.dll. Посмотрите экспорт, чтобы увидеть имена функций, предоставляемых DLL. Затем запустите Disassembler из меню Tools со стандартными параметрами. Вы увидите синий экран с панелями разнообразных сведений. На главной панели отображается дизассемблированный код точки входа DLL. Поскольку нас интересует код getPhysicalTotal(), быстро найдём его с помощью поиска. Выберите Find в меню Search и введите getPhysicalTotal в текстовое поле диалогового окна Find. На панели Name List должен выделиться элемент Java_covertjava_nativecode_OSMemoryInfo_getPhysicalTotal, а на главной панели появится его дизассемблированный код, как на рисунке 18.1.

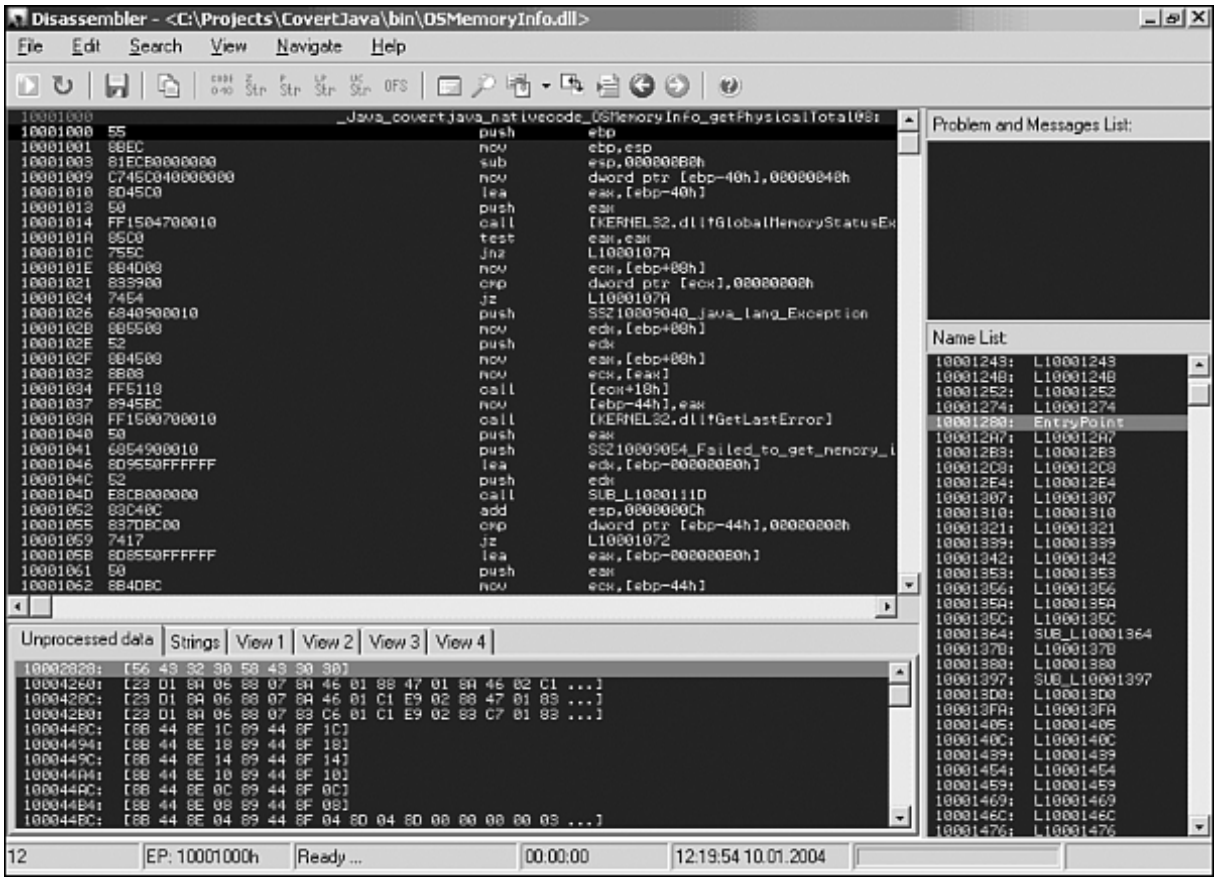

Рисунок 18.1. PE Explorer показывает дизассемблированный код getPhysicalTotal()

При базовом понимании ассемблера можно заметить, что функция начинает с сохранения указателя стека и выделения в стеке места под локальные переменные. Затем она вызывает GlobalMemoryStatusEx из модуля KERNEL32.dll и проверяет, равно ли возвращённое значение 0. Если результат равен 0, функция проверяет, равен ли 0 параметр env метода getPhysicalTotal(); если нет, она форматирует сообщение об ошибке и вызывает подпрограмму, выбрасывающую исключение. В противном случае возвращается значение из локальной структуры, заполненной GlobalMemoryStatusEx. Затем указатель стека восстанавливается, и функция возвращает управление. Мы видим практически взаимно однозначное соответствие коду C тела функции, поскольку getPhysicalTotal использует лишь примитивные операции, например сравнение и вызовы функций. Теперь мы готовы исправить этот код новой логикой.

Исправление машинной функции утилитой Function Replacer

Как уже говорилось, исправление машинной функции подразумевает поиск её двоичного кода и замену части новым кодом или отводом к новому коду. Отводом может служить простая ассемблерная инструкция `JMP` на адрес начала новых инструкций или фрагмент кода, загружающий динамическую библиотеку и вызывающий из неё процедуру. Исправление нужно применять осторожно, чтобы не нарушить состояние регистров и стека вызовов. Ещё один тонкий вопрос - судьба кода, перекрытого кодом отвода. Если исходный код выполнять не требуется, код исправления можно записать поверх исходных инструкций. Но если исправление дополняет исходную логику предварительной или последующей обработкой, исходный код необходимо перенести в другое место, прежде чем заменять отводом. Как видно, двоичное исправление - довольно сложный и хрупкий процесс, требующий тщательного анализа состояния вызывающей стороны и вызываемого кода. Поэтому первым выбором я рекомендую исправлять объявление метода Java или подменять библиотеку целиком.

Надёжных инструментов, способных безопасно выполнять двоичное исправление, нет. Единственная приличная утилита, которую мне удалось найти и с переменным успехом использовать (она не работала с JDK 1.4), - Function Replacer, написанная участником группы программистов Execution с ярким именем Death. Её можно загрузить с веб-сайта группы Execution, который сейчас размещён по адресу execution.cjb.net. Идея утилиты идеально соответствует нашим требованиям. Function Replacer заменяет экспортируемую функцию одной DLL Win32 экспортируемой функцией другой DLL. Замещающая функция должна иметь то же число параметров и тот же стиль вызова, чтобы сохранить состояние стека. Мы используем эту утилиту для исправления метода `getPhysicalTotal()` из `OSMemoryInfo.dll` заглушкой из другой DLL, жёстко запрограммированной всегда возвращать 10. Исходный код исправления показан в листинге 18.3.

```
JNIEXPORT jlong JNICALL Java_covertjava_nativecode_OSMemoryInfo_getPhysicalTotal
(JNIEnv *env, jclass cls)
{
    return (jlong) (int) 10;
}
```

Листинг 18.3. Исходный код исправления `getPhysicalTotal()`

DLL с исправлением называется `OSMemoryInfoPatch.dll` и заранее собрана для этой книги. Её можно пересобрать сценарием `CovertJava/bin/build_native.bat`, если установлен компилятор C и сценарий сборки обновлён для него. Создайте резервную копию `OSMemoryInfo.dll` и запустите Function Replacer. В интерфейсе укажите `OSMemoryInfo.dll` как `To-Be-Patched DLL` и выберите `Java_covertjava_nativecode_OSMemoryInfo_getPhysicalTotal@8` (второй элемент списка) как заменяемую функцию. Укажите `OSMemoryInfoPatch.dll` как `Replacer DLL` и выберите `Java_covertjava_nativecode_OSMemoryInfo_getPhysicalTotal@8` как замещающую функцию. Нажмите `Replace Function` и убедитесь, что утилита не сообщает об ошибках. Теперь попытайтесь выполнить приложение Java и проверьте, сработало ли исправление. Убедитесь, что текущая версия JDK - 1.2 или 1.3, и запустите `CovertJava/bin/OSMemoryInfo.bat`. На моём компьютере получился следующий вывод:

```
C:\Projects\CovertJava\bin>OsMemoryInfo.bat
Total     Physical Memory: 10
Available Physical Memory: 318607360
Total     Virtual  Memory: 2147352576
Available Virtual  Memory: 1992871936
```

Вместо настоящего объёма физической памяти моего компьютера, 535121920, машинный метод Java теперь возвращает 10. Исправление сработало, поэтому исследуем стоящее за ним волшебство. Function Replacer записывает загрузочный код поверх исходного кода метода и вставляет вызов замещающей процедуры. Загрузочный код в начале исходной функции загружает

DLL с исправлением вызовом API `LoadLibrary()` и находит замещающую функцию посредством `GetProcAddress()`. Это стандартный способ динамической загрузки DLL в Win32. После нахождения замещающей функции управление передаётся ей инструкцией `JMP`. Ассемблерный код загрузчика показан в листинге 18.4.

```
push    esi
call    osmemory.10001006
pop     esi
sub     esi,401005
lea     eax,dword ptr ds:[esi+40102c]
push    eax
call    dword ptr ds:[<&kernel32.LoadLibraryA>]
push    ebx
lea     ebx,dword ptr ds:[esi+401042]
push    ebx
push    eax
call    dword ptr ds:[<&kernel32.GetProcAddress>]
pop     ebx

pop     esi
; OSMemoryInfoPatch._java_covertjava_nativecode_osmemoryinfo_GetPhysicalTotal@8
jmp     eax
; Define strings for library and patch function name
db     ...
```

Листинг 18.4. Исправленный ассемблерный код `getPhysicalTotal()`

Поскольку управление передаётся инструкцией `JMP`, замещающая процедура возвращает его непосредственно вызывающей стороне, а не загрузочному коду. Анализ кода помогает понять ограничения устройства Function Replacer. Размер загрузочного кода зависит от длины имени DLL и функции исправления, поэтому подход не работает для очень маленьких машинных функций. Поскольку загрузочный код перекрывает исходный, вызвать исходную функцию невозможно.

Ещё одна проблема Function Replacer: при работе исправления с JDK 1.4.2 утилита вызывает аварийное завершение JVM. Хотя ассемблерный код допустим и исправленная DLL без проблем загружается программами C, похоже, он вмешивается во внутреннее состояние JVM. Function Replacer упрощает исправление, но утилита ненадёжна. Поэтому рассмотрим альтернативный подход: вручную реализуем и установим исправление с помощью мощной библиотеки.

Ручное исправление с помощью библиотеки Microsoft Detours

Detours - библиотека Microsoft для работы с файлами PE на двоичном уровне и перехвата функций во время выполнения. Это надёжный и хорошо написанный каркас, который можно использовать в программах C. Основные возможности библиотеки Detours:

• **Перехват функций во время выполнения.** Функции перехватываются в памяти во время работы, а не на диске. Это более чистый подход, который также может помочь обойти некоторые ограничения лицензионных соглашений.
• **Вызов исходной функции.** Detours сохраняет код исправляемой функции. В отличие от Function Replacer, библиотека Detours до перезаписи сохраняет машинные инструкции исходной функции в сущность, называемую трамплином. Это позволяет выполнять предварительную и последующую обработку вокруг исходной функции.
• **Малый размер отвода.** Отвод реализован как `JMP` к логике исправления, занимает лишь 4 байта и поэтому работает даже с очень короткими функциями.
• **Редактирование таблицы импорта для вставки DLL.** Detours предоставляет функции редактирования таблицы импорта исполняемого файла PE. Это полезно для вставки DLL, реализующей и устанавливающей исправление как отвод целевой функции. Изменения импорта сохраняются в файл на диске.

• **Чистый высокоуровневый API C.** Библиотека хорошо спроектирована и довольно проста в использовании. Понимание архитектуры Win32 всё ещё требуется, но писать на ассемблере не нужно. Исправление и отвод пишутся как функции C, а перехват устанавливается всего несколькими строками кода.

Библиотеку Detours можно бесплатно загрузить с research.microsoft.com. Она поставляется с хорошей документацией и множеством примеров, а поскольку книга посвящена Java, мы не станем тратить время на написание кода C. В листинге 18.5 показаны несколько ключевых фрагментов примера, который исправляет функцию Win32 `Sleep` и измеряет общее время сна программы.

```
/* Declare a Sleep() trampoline using Detours macro */
DETOUR_TRAMPOLINE(VOID WINAPI UntimedSleep(DWORD dwMilliseconds), Sleep);

/* DLL entry point that installs and removes a detour for Sleep */
BOOL WINAPI DllMain(HINSTANCE hinst, DWORD dwReason, LPVOID reserved)
{
    if (dwReason == DLL_PROCESS_ATTACH) {
        printf("slept.dll: Starting.\n");
        Verify((PBYTE)Sleep);
        printf("\n");
        fflush(stdout);
        DetourFunctionWithTrampoline((PBYTE)UntimedSleep, (PBYTE)TimedSleep);
    }
    else if (dwReason == DLL_PROCESS_DETACH) {
        DetourRemove((PBYTE)UntimedSleep, (PBYTE)TimedSleep);
        printf("slept.dll: Removed trampoline, slept %d ticks.\n", dwSlept);
        fflush(stdout);
    }
    return TRUE;
}

/* This is a patch for Sleep() that measures the total time spent sleeping */
VOID WINAPI TimedSleep(DWORD dwMilliseconds)
{
    DWORD dwBeg = GetTickCount();
    UntimedSleep(dwMilliseconds);
    DWORD dwEnd = GetTickCount();
    InterlockedExchangeAdd(&dwSlept, dwEnd - dwBeg);
}
```

Листинг 18.5. Основные этапы использования библиотеки Detours

Код листинга 18.5 устанавливает отвод (исправление) `TimedSleep()` для функции `Sleep()`. Исходную функцию `Sleep()` по-прежнему можно вызвать через трамплин `UntimedSleep()`. Чтобы применить Detours к функции JNI, нужно написать замещающую функцию с той же сигнатурой, что и целевая, и поместить её в DLL. Функция `DllMain()` этой DLL должна установить отвод посредством `DetourFunctionWithTrampoline()`, после чего DLL нужно вставить первым элементом импорта в DLL или EXE, содержащий исправляемую функцию JNI.

Исправление машинного кода на платформах Unix

Исправлять двоичные файлы в мире Unix гораздо труднее, чем в Windows. Unix - разнообразная платформа с множеством аппаратных архитектур и программных стандартов, поэтому такие низкоуровневые задачи, как дизассемблирование исполняемого файла и редактирование машинного кода, требуют разных реализаций для разных архитектур. Например, к распространённым архитектурам процессоров Unix относятся SPARC в Sun Solaris, PA-RISC или Itanium в HP UX, RS/6000 или PPC в IBM AIX и Intel в Linux. У каждого процессора свой набор инструкций, поэтому двоичные файлы непереносимы между архитектурами. Это означает, что общего дизассемблера, преобразующего машинный код в ассемблер на всех платформах, нет. Для каждой платформы существуют бесплатные и коммерческие дизассемблеры, но их качество и удобство сильно различаются. Одна из лучших утилит - IDA Pro (datarescue.com), поддерживающая множество типов процессоров. Она работает только в Windows, но заявляет способность дизассемблировать двоичные файлы большинства распространённых аппаратных архитектур.

С программными стандартами положение ненамного лучше. Существует множество стандартов форматов исполняемых файлов, среди которых сегодня наиболее заметны Common Object File Format (COFF) и Executable and Linking Format (ELF). COFF традиционно использовался в системах Unix. У него есть ограничения и ему недостаёт гибкости, поэтому более современный ELF постепенно его заменяет. И COFF, и ELF подобны формату PE компании Microsoft. В таблице 18.2 показана высокоуровневая структура формата ELF с точки зрения связывания.

Элемент	Описание
Заголовок ELF	Содержит различные сведения о файле, например число разделов и адреса точек входа.
Таблица заголовков программы (необязательная)	Указывает расположение и описание сегментов.
Раздел 1	Данные, относящиеся к разделу 1. Это могут быть машинные инструкции, данные, таблица символов и так далее.
Раздел N	Данные, относящиеся к разделу N.
Таблица заголовков разделов	Массив структур, описывающих атрибуты каждого раздела: имя, тип, начальный адрес раздела и способ интерпретации сведений.

Таблица 18.2. Структура файла ELF

Исправление двоичных файлов требует чтения и записи. Работу с файлами ELF можно упростить библиотекой `libelf`. Она предоставляет набор высокоуровневых функций C для манипуляций с исполняемыми файлами, разделяемыми библиотеками, объектными и другими файлами формата ELF.

`libelf` доступна для Solaris, HP-UX, AIX и Linux; вероятнее всего, её можно найти и для других разновидностей Unix. Поскольку `libelf` является библиотекой общего назначения, она не предоставляет функций исправления, найденных нами в Microsoft Detours. `libelf` даёт удобный способ найти исправляемый код и обновить исполняемый файл изменениями, но вставлять ассемблерные инструкции и, возможно, реализовывать трамплин приходится вручную.

Подход к исправлению разделяемых библиотек Unix с машинным кодом совпадает с применённым в Windows. Необходимо найти и дизассемблировать машинный код целевой функции. Затем его можно перезаписать новым кодом или инструкцией `JMP` к новому коду. Новую логику можно также реализовать в разделяемой библиотеке, динамически загружаемой исправлением. Если сигнатура функции и соглашение о вызовах совпадают, параметры передаются и возврат происходит

правильно. Для проектирования конкретного ассемблерного кода обратитесь к документации целевого процессора.

Короткий тест

1. Какую роль JNI играет в архитектуре Java?
2. Какие этапы необходимо выполнить, чтобы реализовать и выполнить машинный метод?
3. Для каждого из трёх подходов к исправлению машинных методов перечислите достоинства и недостатки.
4. К какому разделу файла PE нужно обратиться, чтобы получить машинный код?
5. Почему Function Replacer не работает с машинными функциями, состоящими всего из нескольких машинных инструкций?
6. Можно ли при реализации отвода в ассемблерном коде передать управление исправлению инструкцией `CALL` вместо `JMP`? Объясните почему.
7. Какие преимущества предлагает библиотека Detours по сравнению с Function Replacer?
8. Какие форматы исполняемых файлов преобладают в Unix?
9. Как исправить машинную функцию в Unix?

Краткие итоги

• Исправление машинного кода даёт предельный контроль над JVM, поскольку позволяет изменять поведение на самом низком уровне. Оно основано на изучении двоичного формата библиотеки, поиске изменяемого машинного кода и его исправлении новой логикой.
• JNI - спецификация, описывающая определение машинных методов в Java и предоставление их реализации в машинных библиотеках.
• Машинные методы Java требуют разработать на языке C динамическую (разделяемую) библиотеку, загружаемую JVM во время выполнения.
• Самый простой подход к машинному исправлению - исправить объявляющий метод класс Java, удалить ключевое слово `native` и предоставить новую реализацию метода на Java.
• Подмена машинной библиотеки делегирующим заместителем предлагает второй вариант исправления машинного кода. Подставная библиотека реализуется на C без изменений классов Java. Исходная библиотека переименовывается, а новая получает её прежнее имя.
• В Windows экспортируемую функцию одной DLL можно исправить экспортируемой функцией другой DLL с помощью такой утилиты, как Function Replacer. Она проста в использовании, но имеет ограничения и проблемы с надёжностью.
• Microsoft Detours - библиотека для работы с файлами PE на двоичном уровне и перехвата функций во время выполнения. Это надёжный и хорошо написанный каркас для ручного исправления в программах C.
• Исполняемые файлы Unix обычно соответствуют формату COFF или ELF. Общий подход к исправлению библиотек Unix подобен подходу Windows.
• `libelf` - широко используемая библиотека манипуляций исполняемыми файлами формата ELF в Unix.

Глава 19. Защита коммерческих приложений от взлома

> «Мерфи был оптимистом».
>
> *Постулат Бека*

В этой главе

• Постановка целей защиты приложения
• Защита данных с помощью архитектуры криптографии Java
• Защита дистрибутива приложения от взлома
• Реализация лицензирования для открытия функций приложения
• Короткий тест
• Краткие итоги

Постановка целей защиты приложения

На протяжении книги мы рассматривали разнообразные приёмы обратной разработки, взлома, подслушивания и вскрытия. Во многих главах я призывал читателя к совести и добросовестности, прося не нарушать интеллектуальные права авторов программ. Хотя у большинства пользователей здоровые моральные ценности, те немногие остальные способны причинить большой ущерб. В этой главе даются практические советы о защите приложений Java от взлома и реализации модели распространения коммерческих программных продуктов.

Обычное приложение Java поставляется как комплект (чаще всего в виде файла JAR или Zip, но иногда как исполняемая программа установки), содержащий библиотеки Java и машинные библиотеки, файлы конфигурации, документацию и различные файлы ресурсов. Сегодня принято бесплатно предлагать общественности версию программы без излишеств, а полный набор функций предоставлять только в лицензированных версиях. Другая стратегия привлечения потенциальных покупателей - ограниченный ознакомительный период с доступом ко всем функциям. После его окончания коммерческие функции приложения отключаются до покупки лицензии. Эту стратегию применяет Borland при распространении JBuilder X. Сначала он работает как 30-дневная ознакомительная версия Enterprise Edition, а затем превращается в JBuilder X Foundation с ограниченными возможностями. Многие поставщики корпоративного программного обеспечения бесплатно предлагают продукты для разработки или полагаются на честность пользователей и страх судебного преследования, побуждающий приобрести правильную лицензию. Какую бы модель лицензирования и распространения ни выбрал поставщик, он жизненно заинтересован собирать лицензионные платежи и получать выручку. Простые приёмы этой главы достаточно хорошо страхуют соблюдение модели лицензирования.

Важно понимать: добиться абсолютной защиты от хакеров практически невозможно, особенно если приложение можно загрузить из Интернета. Даже если самые сильные алгоритмы безопасности применяются для создания и шифрования конфиденциальных данных, например серийного номера, хороший хакер с доступом к приложению может исправить код проверки и полностью её обойти. В предыдущих главах показано, насколько легко находить и взламывать классы Java; если ставки высоки, можно вскрыть даже машинный код. Поэтому ключ успешного механизма защиты -

сделать взлом слишком трудным для 95% обычных пользователей и заставить оставшиеся 5% опытных хакеров потратить на него значительное время. Иными словами, цель состоит в том, чтобы купить лицензию было дешевле, чем тратить время на взлом защиты. Ещё один жизненно важный аспект - предотвратить лёгкое распространение взломанных версий в Сети и не позволить хакерам выпускать собственные лицензии на продукт.

Начнём с основных сторон безопасности и криптографии. На примерах с API криптографии Java вы научитесь шифровать и расшифровывать сведения шифрами, защищать целостность данных свёрткой сообщения и реализовывать надёжный механизм лицензирования асимметричными парами ключей. Кроме этих мер мы рассмотрим несколько приёмов защиты основных файлов приложения от взлома и исправления.

Защита данных с помощью архитектуры криптографии Java

Слово «криптография» происходит от древнегреческих слов *kryptos* («скрытый») и *graphein* («писать»). Криптография позволяет преобразовать читаемые сведения в непонятный код, который можно передавать открыто, а затем вернуть в исходную форму. Шифрование - процесс кодирования читаемых сведений, а расшифрование - процесс извлечения читаемых сведений из кода. Ещё одна часто используемая криптографическая служба создаёт хеш, или свёртку сообщения, чтобы проверить, не менялось ли сообщение после отправки. Криптографические службы реализуются различными математическими алгоритмами. По предоставляемой службе алгоритмы делятся на три основные категории: свёртка сообщений, шифрование/расшифрование и подписание.

Алгоритмы свёртки не изменяют содержимое сообщения, а создают на основе его содержимого и секретного ключа уникальный хеш. Ключом может быть что угодно: число, передаваемое как параметр вычислительному алгоритму, строка символов как пароль или последовательность байтов. Отправитель конфиденциального сообщения вычисляет свёртку с секретным ключом и посылает её вместе с сообщением. Получатель с тем же секретным ключом вычисляет свёртку полученного сообщения, и если она не совпадает со свёрткой отправителя, содержимое считается скомпрометированным. Перехватившая сообщение третья сторона может просмотреть и изменить его содержимое, но без секретного ключа не сможет пересчитать свёртку. Тем самым сохраняется целостность обмена.

Алгоритмы шифрования и расшифрования защищают конфиденциальные сведения, которые может перехватить третья сторона. Содержимое сообщения изменяется секретным ключом, образуя результат, который практически невозможно вернуть к исходному содержимому. Перехватившая сообщение третья сторона не сможет расшифровать его без ключа.

Алгоритмы шифрования и расшифрования делятся на две категории: с симметричными и асимметричными ключами. Симметричные алгоритмы требуют, чтобы у отправителя и получателя был один и тот же ключ для шифрования и расшифрования. Иногда их называют двусторонними, поскольку обе операции выполняются одним ключом. Надёжность защиты, очевидно, зависит от того, насколько хорошо ключи защищены от третьих сторон. Асимметричные алгоритмы используют для преобразований пары ключей. Пара состоит из открытого и закрытого ключа. Такой алгоритм называют односторонним: сведения, зашифрованные открытым ключом, можно расшифровать только закрытым, а для сведений, зашифрованных закрытым ключом, требуется открытый. Обычно открытый ключ свободно доступен всему миру, а закрытый владелец держит в секрете. Для клиент-серверных и серверных приложений это безопаснее симметричных алгоритмов, поскольку в дистрибутив клиентского приложения нужно включать только открытый ключ.

Обычный пример применения асимметричного алгоритма - браузер, которому требуется установить защищённое соединение с веб-сервером. Браузер получает открытый ключ для кодирования сведений, отправляемых веб-серверу. Веб-сервер расшифровывает их своим секретным закрытым ключом, но перехватившая сообщение третья сторона не сможет расшифровать его открытым ключом. Сервер шифрует закрытым ключом сведения для браузера, поэтому весь обмен защищён. Симметричные алгоритмы намного быстрее асимметричных, из-за чего их часто применяют совместно. Например, реализация SSL устанавливает сеанс асимметричным алгоритмом. После создания защищённого канала генерируются и передаются симметричные ключи для шифрования передаваемых данных.

Подписание - это создание сравнительно короткой цифровой подписи на основе данных произвольного размера с помощью закрытого ключа асимметричного алгоритма. Отправитель создаёт подпись и передаёт её вместе с сообщением. Получатель проверяет целостность сообщения открытым ключом и подписью. Как и свёртка сообщения, подпись математически уникальна для заданных данных, поэтому при изменении данных подпись не соответствует содержимому. Подлинность обеспечивается асимметричным алгоритмом, где закрытый ключ есть только у отправителя. Чтобы третья сторона не могла подделать открытый ключ и выдать его за ключ отправителя, широко применяются цифровые сертификаты. Цифровой сертификат содержит открытый ключ отправителя, подписанный закрытым ключом доверенного центра сертификации (CA). Например, браузеры заранее настроены доверять Verisign (verisign.com) как CA. Компания, желающая разрешить пользователям устанавливать защищённый канал связи со своим веб-сервером, должна отправить открытый ключ в Verisign и получить цифровой сертификат. Полученный сертификат устанавливается на веб-сервере и передаётся браузеру в начале взаимодействия. Браузер проверяет подлинность сертификата открытым ключом Verisign и устанавливает защищённое соединение, только если проверка успешна.

Обзор архитектуры криптографии Java

Архитектура криптографии Java (JCA) предоставляет полную и надёжную реализацию криптографических служб и алгоритмов. Подобно большинству API Java, JCA предоставляет интерфейсы, определяющие независимое от поставщика взаимодействие приложения со службами. Расширения криптографии Java (JCE), когда-то бывшие отдельным модулем, начиная с JDK 1.4 входят в J2SE. JCE поставляется с провайдером Sun, реализующим наиболее распространённые алгоритмы, например `HmacSHA1` для защищённого хеширования и DES для генерации пар ключей и подписания. Из-за экспортных ограничений правительства США некоторые алгоритмы, в том числе RSA, не входят в JDK. Помимо Sun JCE богатый набор алгоритмов реализуют другие превосходные открытые пакеты. Bouncy Castle (bouncycastle.org) и Cryptix (cryptix.org) предоставляют бесплатные реализации на Java.

Основные классы JCA находятся в пакете `javax.crypto`, хотя классы и интерфейсы для работы со свёртками сообщений расположены в `java.security`. Домашняя страница Java Security (java.sun.com) - хорошая отправная точка для подробностей различных тем безопасности с точки зрения Java. Домашняя страница JCE java.sun.com даёт высокоуровневый обзор JCE и ссылки на относящиеся к JCE страницы, например справочное руководство. Если вам нравится погружаться глубоко, рекомендую купить книгу о безопасности Java, поскольку тема обширна и интересна. Книга Джейми Яворски и Пола Перрона *Java Security Handbook* (Sams Publishing, ISBN: 0672316021) всесторонне рассматривает разнообразные темы безопасности. Следующие разделы главы сосредоточены на практическом применении безопасности для защиты приложений Java от взлома.

Защита сообщений Chat с помощью JCA

Снова воспользуемся пресловутым приложением Chat, чтобы показать самые полезные способы защиты конфиденциальности пользователей и интеллектуальной собственности автора. Поскольку Chat отправляет сообщения по сети, разговор пользователей можно перехватить и подслушать. Поэтому очевидная отправная точка защиты Chat - защита содержимого передаваемых сообщений.

Напомним, что экземпляры Chat на разных узлах обмениваются сообщениями посредством RMI поверх TCP/IP. Это не так плохо, как HTML поверх HTTP, поскольку двоичные потоки TCP/IP подслушать намного труднее текстового HTTP. Но, как вы видели в главе 13, «Методы подслушивания», с подходящими инструментами хакер способен слушать разговор и читать содержимое сообщений. Главная причина возможности подслушивания в том, что строки внутри сериализованных объектов Java остаются текстом. Поэтому защита сообщений Chat требует шифровать строки. Для Chat подойдёт почти любое шифрование: сообщения двоичные, и если строки не распознаются человеком, они не выделяются в теле сообщения (см. главу 13). Работает даже простая операция XOR над символами. Теоретически самый безопасный способ защитить канал RMI - собственные фабрики сокетов, создающие сокеты SSL. Однако нас интересует универсальный способ защиты данных, поэтому мы напишем код с API криптографии Java.

Первое проектное решение - выбрать алгоритм. Асимметричные алгоритмы обычно защищают лучше, поскольку закрытый ключ недоступен широкой публике. Но для шифрования сообщений Chat асимметричный алгоритм не подходит. Приложение Chat на настольном компьютере должно и шифровать отправляемые сообщения, и расшифровывать получаемые. Асимметричный алгоритм потребовал бы поставлять с Chat оба ключа, закрытый и открытый. Это фактически сводит на нет дополнительную защиту, поэтому следует применить симметричное шифрование: оно быстрее и проще в реализации.

Второе проектное решение - выбрать провайдера безопасности. Провайдер даёт конкретную реализацию алгоритма. Чтобы не распространять с Chat дополнительные библиотеки, сначала проверим алгоритмы Sun JCE, поставляемого с JRE. Sun JCE поддерживает следующие алгоритмы шифров: Data Encryption Standard (DES), DESede и PBEWithMD5AndDES. DES - широко применяемый стандарт, принятый правительством США. Хотя известны способы вскрыть его большой вычислительной мощностью, большинству приложений он даёт достаточную защиту. DESede, также известный как множественный DES, применяет несколько ключей DES для повышения стойкости. PBEWithMD5AndDES использует сочетание алгоритмов, включая шифрование по паролю из стандарта PKCS#5 и свёртку сообщений из алгоритмов MD5 и DES. Благодаря стандартизации JCA клиентский код, использующий эти алгоритмы, почти не зависит от выбранного алгоритма. Мы выберем PBEWithMD5AndDES, поскольку он даёт самую сильную защиту из трёх.

Картина стоит тысячи слов, а в мире программирования исходный код стоит тысячи картин. Весь исходный код этой главы находится в пакете `covertjava.protect`. Начнём с класса, предоставляющего приложению Chat службы шифрования. В листинге 19.1 показан конструктор `covertjava.protect.Encryptor`.

```
import javax.crypto.*;
import javax.crypto.spec.*;

public Encryptor(char[] password) throws Exception {
    PBEKeySpec keySpec = new PBEKeySpec(password);
    SecretKeyFactory keyFactory =
        SecretKeyFactory.getInstance("PBEWithMD5AndDES");
    secretKey = keyFactory.generateSecret(keySpec);
    PBEParameterSpec paramSpec = new PBEParameterSpec(this.keyParams, this.iter_count);

    this.encCipher = Cipher.getInstance("PBEWithMD5AndDES");
    this.encCipher.init(Cipher.ENCRYPT_MODE, secretKey, paramSpec);
    this.decCipher = Cipher.getInstance("PBEWithMD5AndDES");
    this.decCipher.init(Cipher.DECRYPT_MODE, secretKey, paramSpec);
}
```

Листинг 19.1. Подготовка шифров для шифрования и расшифрования

Разберём исходный код и поймём, что он делает. Конструктор `Encryptor` принимает пароль как массив символов. Алгоритм PBEWithMD5AndDES использует три параметра: соль и число итераций передаются для инициализации DES, а пароль применяется для шифрования по PKCS#5. Алгоритм шифрования в JCE представлен классом `javax.crypto.Cipher`. Программа получает экземпляр шифра вызовом `Cipher.getInstance()`, принимающим имя алгоритма (перегруженный метод может также принять имя провайдера).

Алгоритмы безопасности часто требуют параметров от клиентского кода. Параметры используются в математических вычислениях при шифровании и представляют секретное начальное значение или пароль, необходимый для последующего расшифрования. Хотя большинство требующих параметров алгоритмов способны использовать стандартные значения провайдера, настоятельно рекомендуется инициализировать их собственными значениями.

Инициализировать наш шифр можно двумя способами. Первый - передать параметры алгоритма, например соль и число итераций. Второй - передать уже созданный ключ. Если передавать ключ, его пришлось бы поставлять с дистрибутивом Chat, что упростило бы его извлечение хакерами. Параметры алгоритма, являющиеся обычными числами, можно жёстко задать в коде Java и разместить в разных классах. Обфускация сделает код очень трудным для чтения, поэтому мы выберем параметры, а не ключ. Объявление параметров алгоритма внутри класса `Encryptor` показано в листинге 19.2.

```
public class Encryptor {
    private static byte[] keyParams = {
        (byte)0x10, (byte)0x15, (byte)0x01, (byte)0x04,
        (byte)0x55, (byte)0x06, (byte)0x72, (byte)0x01
    };
    private static int iter_count = 20;
    ...
}
```

Листинг 19.2. Параметры PBEWithMD5AndDES, используемые `Encryptor`

В реальном приложении параметры лучше поместить в другой класс или создавать на лету генератором случайных чисел с жёстко заданным начальным значением. Это усложнило бы взлом, но мы сохраним простоту. Возвращаясь к листингу 19.1, видим, что первый блок создаёт спецификацию ключа на основе переданного пароля. Спецификация ключа - промежуточная форма данных ключа, по которой фабрика создаёт секретный ключ. Поскольку все экземпляры Chat используют одинаковые параметры алгоритма и пароль, созданные ключи совпадают. Поэтому сообщения, зашифрованные одним экземпляром Chat, можно расшифровать другим. После создания секретного ключа получаются два экземпляра шифра: один инициализируется для шифрования, другой - для расшифрования.

После создания экземпляр `Encryptor` готов шифровать и расшифровывать. Большинство криптографических алгоритмов работают с необработанными байтами. У класса шифра есть два метода, `update()` и `doFinal()`, которыми можно зашифровать массив байтов. Например, если имеется инициализированный шифр `cipher` и массив байтов `data`, данные шифруются так:

```
byte[] ecryptedData = cipher.doFinal(data);
```

В JCE есть служебный класс `SealedObject`, который обёртывает любой сериализуемый объект и применяет предоставленный шифр для шифрования или расшифрования обёрнутого объекта при сериализации. Поскольку Chat отправляет сообщения как объекты, `SealedObject` лучше необработанных байтов: он предоставляет более высокоуровневый API шифрования. Два метода `Encryptor` для шифрования и расшифрования экземпляров `java.io.Serializable` показаны в листинге 19.3.

```
public Serializable encryptObject(Serializable object) throws Exception {
    return new SealedObject(object, this.encCipher);
}

public Object decryptObject(Serializable object) throws Exception {
    return ((SealedObject)object).getObject(this.decCipher);
}
```

Листинг 19.3. Методы, реализующие шифрование и расшифрование

Как видно, `SealedObject` делает реализацию тривиальной. Рассмотренный класс `Encryptor` теперь можно применить в Chat. Вместо передачи друг другу экземпляров `covertjava.chat.MessageInfo` Chat перед отправкой вызывал бы `Encryptor`, получая зашифрованную версию сообщения. При получении нового сообщения Chat извлекал бы объект `MessageInfo` из запечатанного объекта посредством `Encryptor`. Этот код пришлось бы поместить в методы `sendMessage()` и `receiveMessage()` класса `ChatServer`, но ради экономии времени и места мы этого делать не станем. Метод `main()` класса `Encryptor` показывает самопроверку, записывающую текстовую строку в файл и читающую её обратно.

Защита дистрибутива приложения от взлома

Шифрование передаваемых данных защищает сведения от подслушивания на уровне протокола. Это оберегает сведения пользователя, но не интеллектуальную собственность программы, например алгоритмы, шаблоны проектирования и код. Многие приёмы обратной разработки из книги легко применить для вскрытия коммерческого продукта и открытия функций, которые иначе потребовали бы покупки лицензии. Для лицензий, выдаваемых по числу узлов установки программы, ещё одна угроза исходит от неэтичной организации, покупающей самую дешёвую лицензию на один узел, а затем развёртывающей её на множестве узлов. В этом разделе рассматриваются несколько приёмов защиты дистрибутива от взлома и обеспечения оплаты согласно модели лицензирования.

Защита байт-кода от декомпиляции

В главе 2, «Декомпиляция классов», показано, насколько легко получить исходный код из байт-кода Java и что чаще всего декомпилированный код практически взаимно однозначно соответствует исходному. В главе 3, «Обфускация классов», подробно рассказано, как защищать байт-код от декомпиляции. Очевидно, что общая защита сильна настолько, насколько силён реализующий её код. Можно использовать самый сильный алгоритм шифрования данных, но если код декомпилируется и исправляется за 30 минут, шифрование можно просто закомментировать.

Обфускация, обфускация, обфускация. Это единственный надёжный способ защитить байт-код, а следовательно, интеллектуальную собственность приложения. Для наилучших результатов критически важна обфускация потока управления, рассмотренная в главе 3. Предельная мера против декомпиляции байт-кода - скомпилировать приложение Java в машинный исполняемый файл. Мы видели сложность обратной разработки и исправления машинного кода, и каким бы хорошим ни был обфускатор байт-кода, машинный код вскрыть гораздо труднее. К сожалению, к настоящему времени большинство поставщиков компиляторов Java в машинный код либо прекратили существование, либо перестали активно поддерживать продукты. Улучшения JIT и рост скорости процессоров обеспечивают приложениям Java достаточную производительность, устраняя необходимость компилировать их в машинный код. Вероятно, лучшие реализации для Windows предлагают TowerJ и Excelsior, но я советую проявить осторожность и тщательно испытать скомпилированное приложение, чтобы убедиться в правильной работе всех функций. Для большинства приложений Java агрессивный обфускатор, например Zelix KlassMaster, вероятно, лучше компиляции в машинные двоичные файлы.

Защита байт-кода от взлома

Как бы хорошо ни сработал обфускатор, байт-код всё ещё можно декомпилировать. А если его можно декомпилировать, его можно изменить и исправить приложение. Чтобы усилить защиту, рассмотрим несколько идей проверок безопасности, оберегающих классы от исправления.

В книге мы разработали различные приёмы взлома и исправления. Мы обсуждали декомпиляцию и последующее исправление целых классов, доступ к защищённым и закрытым методам и работу с системным начальным путём классов. При неправильном применении эти приёмы способны повредить интеллектуальной собственности внутри приложения Java. Здесь рассмотрим приёмы, значительно затрудняющие взлом, и меры противодействия каждому из них.

Взлом непубличных методов и переменных класса (глава 4)

Самое простое решение - запечатать JAR приложения. Запечатывание гарантирует, что все классы пакета происходят из одного источника кода. Это означает, что хакер не сможет поместить собственные классы в пакеты, предоставленные JAR. JAR запечатывается добавлением следующей строки в файл манифеста:

```
Sealed: true
```

Сам JAR необходимо защитить от изменений. Как вы легко его запечатываете, так хакер легко снимет печать. Ему достаточно распаковать содержимое JAR во временный каталог, удалить атрибут `Sealed`, а затем снова упаковать. Java поддерживает подписанные JAR, способные защитить содержимое от изменений цифровой подписью каждого класса. Это хорошо работает для подписанных апплетов, загружаемых и проверяемых браузером. Проблема в том, что сам подписанный JAR не защищён, поэтому хакер снова может распаковать файл, удалить манифест с цифровыми подписями и собрать его заново. Хотя JAR больше не будет считаться подлинным и происходящим от настоящего поставщика, он прекрасно выполнится и будет работать. Поэтому нужен способ убедиться, что содержимое дистрибутива не изменялось; мы рассмотрим его в следующем разделе.

Замена и исправление классов приложения (глава 5)

Запечатывание JAR помогает и против этого приёма. Для дополнительной защиты можно добавить проверку, утверждающую, что класс действительно загружен из JAR дистрибутива приложения, а не стороннего JAR. Реализация этого простого метода находится в `covertjava.protect.IntegrityProtector`. Исходный код `assertClassSource()` показан в листинге 19.4.

```
public void assertClassSource(Class cls, String jarName) {
    // Class loader should not be null
    if (cls.getClassLoader() == null)
        throw new InternalError(BOOT_CLASSLOADER);

    String name = cls.getName();
    int lastDot = name.lastIndexOf('.');
    if (lastDot != -1)
        name = name.substring(lastDot + 1);
    URL url = cls.getResource(name + ".class");
    if (url == null)
        throw new InternalError(FAILED_TO_GET_URL);
    name = url.toString();
    if (name.startsWith("jar:") == false || name.indexOf(jarName + "!") == -1)
        throw new InternalError(UNEXPECTED_JAR);
}
```

Листинг 19.4. Проверка исходного JAR класса

Первый оператор `if` проверяет, что загрузчик заданного класса не является начальным, сравнивая его с `null`. Такое утверждение допустимо, поскольку известно, что классы Chat не помещаются в начальный путь и должны загружаться стандартным загрузчиком средства запуска приложения.

Остальной код получает URL источника файла CLASS, использованного для создания заданного класса. Вот URL, возвращаемый `Class.getResource()` для класса `MessageInfo`:

```
jar:file:/C:/Projects/CovertJava/distrib/lib/chat.jar!/covertjava/chat/MessageInfo.class
```

URL показывает, что класс загружен из файла `chat.jar` в каталоге `C:/Projects/CovertJava/distrib/lib`. Получив URL, `assertClassSource()` проверяет, что тот начинается с `jar:` и содержит имя JAR, переданное как параметр метода. Если утверждение ложно, выбрасывается непроверяемая ошибка `InternalError`, прерывающая выполнение. Возможно, проверка не абсолютно безошибочна, но должна отразить большинство попыток исправления. Для использования этой защиты Chat должен вызывать `assertClassSource()` для ключевых классов, являющихся основными кандидатами на исправление. Добавим новый класс `ProtectedChatApplication` как альтернативную точку входа Chat. `ProtectedChat` расширит `covertjava.chat.ChatApplication` и будет применять различные разработанные в главе механизмы защиты. Код листинга 19.5 показывает часть метода `main()` класса `ProtectedChatApplication`, проверяющую происхождение `LicenseManager`.

```
public static void main(String[] args) throws Exception {
    LicenseManager licenseManager = new LicenseManager("conf/chat.license");
    IntegrityProtector protector = new IntegrityProtector();
    protector.assertClassSource(licenseManager.getClass(), "/lib/chat.jar");
    ...
}
```

Листинг 19.5. Проверка происхождения `LicenseManager`

Метод `main()` проверяет, что класс `LicenseManager` загружен из файла `chat.jar`, находящегося в подкаталоге `lib`. Подобные проверки следует вставить в другие классы Chat. Чем их больше, тем больше труда потребуется хакеру для вскрытия приложения.

Манипуляции безопасностью Java (глава 7)

Манипуляции безопасностью Java позволяют хакерам получать доступ к защищённым, пакетным и закрытым членам класса и обходить другие проверки, обычно принудительно выполняемые диспетчером безопасности. Если приложение устанавливает диспетчер безопасности или использует собственный файл политики, следует вставить проверки наличия диспетчера и применения правильного файла политики. Диспетчер можно получить методом `System.getSecurityManager()`, а исходный файл политики проверить по значению системного свойства `java.security.policy`. Сам файл политики защищается от изменений приёмом защиты содержимого приложения, описанным далее.

Обратная разработка приложений (глава 12)

В зависимости от типа ресурса требуется защита байт-кода или содержимого приложения. Такие ресурсы, как строки пунктов меню и сообщения об ошибках, часто жёстко заданы в байт-коде, тогда как изображения и мультимедийные файлы обычно находятся в отдельном каталоге или внутри JAR. Защита байт-кода рассмотрена ранее, а защита содержимого приложения представлена в следующем разделе.

Управление загрузкой классов (глава 14)

Собственные загрузчики дают большую власть, поскольку способны манипулировать байт-кодом на лету. Это, конечно, не распространённый приём взлома, но если нужно защитить класс приложения от манипуляций с байт-кодом во время выполнения, вместо системного загрузчика можно установить и использовать заранее определённый собственный. Затем в разных местах кода приложения можно проверять, загружен ли класс ожидаемым загрузчиком.

Понимание и модификация байт-кода (глава 17)

Для модификации байт-кода требуется либо собственный загрузчик, выполняющий её на лету, либо статическое изменение файлов CLASS приложения. В предыдущем абзаце мы обсудили предотвращение применения стороннего загрузчика, а в следующем разделе описана защита файлов дистрибутива.

Защита содержимого приложения от взлома

Под содержимым приложения здесь понимаются распространяемые с ним файлы: библиотеки JAR и Zip, изображения, файлы конфигурации и прочее. Проверка неизменности ключевых файлов критически важна для защиты целостности приложения, поскольку большинство приёмов взлома требует изменить какой-либо файл. Самый важный вид файла для защиты целостности - архив JAR. Мы уже рассмотрели приём, проверяющий загрузку класса из ожидаемого JAR. Теперь разработаем класс, позволяющий приложению утверждать, что ни один его файл не был подделан.

Самый прямой способ проверить целостность содержимого - перебрать файлы дистрибутива и проверить их атрибуты, например размер и время изменения. Для предельной защиты приложение может вычислить контрольную сумму содержимого файла и сравнить её с контрольной суммой исходных файлов, полученной при подготовке дистрибутива. Мы разработаем в пакете `covertjava.protect` класс `IntegrityProtector` и используем его для защиты пресловутого Chat. Ради краткости ограничимся проверкой длины файлов, хотя её легко расширить другими атрибутами и хешем содержимого. `IntegrityProtector` перебирает список основных файлов приложения, вычисляет их общий размер в байтах, а затем получает контрольную сумму алгоритмом свёртки сообщений. Затем добавим в Chat файл конфигурации с версией и свёрткой дистрибутива. Хранить свёртку вместо общей длины файлов значительно затрудняет взлом. Наконец, при запуске Chat применим `IntegrityProtector`, чтобы проверить совпадение текущей контрольной суммы файлов с исходной суммой из файла конфигурации.

Первая задача - решить, какие файлы Chat следует защищать от изменений. Нельзя просто включить все файлы, поскольку некоторые предназначены для изменения конечным пользователем. Например, `bin/setenv.bat` можно изменить, задав особый домашний каталог Chat или другой порт. `conf/log4j.properties` может измениться, если пользователь настроит уровни журналирования. Однако `lib/chat.jar` и `conf/java.policy` никогда не должны отличаться от исходных версий (если только мы не отправляем заказчику исправления, вместе с которыми можно предоставить новые контрольные суммы). В примере защитим лишь основные файлы дистрибутива Chat:

```
conf\java.policy
lib\chat.jar
lib\log4j-1.2.8.jar
```

Для гибкости список основных файлов будем читать из файла конфигурации `ChatFileList.class`. Чтобы запутать хакеров, списку дано расширение `.class`, хотя содержимое текстовое. Во время разработки файл будет находиться в `CovertJava/conf`, но мы изменим `build.xml`, чтобы копировать `ChatFileList.class` в каталог `covertjava/protect` вместе с классами пакета `covertjava.protect`. Задача `<copy>`, добавленная в `build.xml`, показана в листинге 19.6.

```
<copy todir="classes/covertjava/protect">
    <fileset dir="conf">
        <include name="ChatFileList.class"/>
    </fileset>
</copy>
```

Листинг 19.6. Копирование списка файлов в каталог дистрибутива

Дополнительные 15 минут на маскировку списка под обычный файл класса стоят усилий, поскольку делают тривиальный метод защиты менее очевидным хакеру. Теперь можно продолжить разработку класса `IntegrityProtector` в пакете `covertjava.protect`. Сначала нужно написать несколько вспомогательных методов, которые читают содержимое текстового файла (например, `ChatFileList.class` для Chat) и разбирают его в массив строк. Открыв `IntegrityProtector.java` в каталоге `src/covertjava/protect`, можно увидеть реализации вспомогательных методов `readFilePathsFromResource()`, `readFilePathsFromFile()` и `readFilePathsFromString()`. Теперь напишем метод, создающий контрольную сумму для заданного списка путей. Добавим в `IntegrityProtector` метод `getFilesCheckSum()` с реализацией из листинга 19.7.

```
public String getFilesCheckSum(String[] paths, char separator,
    String installPath) throws Exception {
    long totalSize = 0;
    for (int i = 0; i < paths.length; i++) {
        String path = paths[i];
        if (separator != File.separatorChar)
            path = path.replace(separator, File.separatorChar);
        path = installPath + File.separatorChar + path;
        totalSize += new File(path).length();
    }
    byte[] checkSum = toByteArray(totalSize);
    MessageDigest sha = MessageDigest.getInstance("SHA-1");
    checkSum = sha.digest(checkSum);
    BASE64Encoder encoder = new BASE64Encoder();
    return encoder.encode(checkSum);
}
```

Листинг 19.7. Вычисление контрольной суммы списка файлов

Метод перебирает массив имён файлов, прибавляя размер каждого файла в байтах к общему размеру. После вычисления общий размер преобразуется в массив байтов вспомогательным методом `toByteArray()`. Затем `getFilesCheckSum()` получает экземпляр алгоритма свёртки

SHA-1 (Secure Hash Algorithm, предоставляемый Sun JCE) и вычисляет хеш общего размера. Поскольку контрольную сумму нужно хранить в текстовом файле, байты необходимо

преобразовать в распознаваемые человеком символы ASCII. Нельзя просто привести переменные byte к типу char, поскольку получатся непечатаемые символы (например, байт 7 даст звуковой сигнал, а байт 8 - возврат на один символ). Стандартное решение - кодирование base64. Оно использует подмножество ASCII всего из 64 печатаемых символов. Подмножество включает символы A-Z и a-z, цифры 0-9 и несколько других безопасных символов, например знаки препинания. Поскольку символов меньше, base64 выделяет на символ 6 бит вместо 8 бит ASCII. Поэтому 3 байта входных данных кодируются в 4 байта выходных. IntegrityProtector применяет класс Base64Encoder из пакета sun.misc, чтобы получить печатное представление контрольной суммы файла.

Теперь, умея получать контрольную сумму, используем её для проверки целостности установки Chat. Напишем метод main() класса IntegrityProtector, выводящий контрольную сумму для заданного списка файлов. Его тело показано в листинге 19.8.

```
public static void main(String[] args) throws Exception {
    if (args.length != 1) {
        System.out.println("Syntax: IntegrityProtector " +
            "[-Dhome=<path to home>] <file list path>");
        System.exit(1);
    }
    IntegrityProtector protector = new IntegrityProtector();
    String[] paths = protector.readFilePathsFromFile(args[0]);
    String homePath = System.getProperty("home", "..");
    String checksum = protector.getFilesCheckSum(paths, '\\', homePath);
    System.out.println("Checksum = [" + checksum + "]");
}
```

Листинг 19.8. Вывод контрольной суммы файла

Метод main() принимает один параметр, задающий файл списка (в нашем случае conf/ChatFileList.class), и необязательный параметр домашнего каталога файлов (в нашем случае distrib). Для удобства в каталог CovertJava/bin включён пакетный файл getChatChecksum.bat, который выводит контрольную сумму Chat через IntegrityProtecor. Выполнение getChatChecksum.bat после сборки дистрибутива Chat целью Ant release даёт:

```
Checksum = [gLmBOKQe88gLrC9vaSjBarf2Rfw=]
```

Каждый раз, когда основной файл Chat меняется (например, при повторной сборке lib/chat.jar), контрольная сумма становится другой. Но если размеры файлов не изменились, сумма остаётся прежней, что потенциально создаёт дыру в защите. Поэтому контрольная сумма настоящего содержимого файлов безопаснее; однако большинство хакеров не станет заботиться о сохранении размера, так что для Chat сработает даже наш простой механизм.

Теперь можно добавить в каталог conf приложения Chat файл конфигурации chat.properties. В нём сохраним контрольную сумму дистрибутива как значение свойства chat.versionInfo. И снова мы избегаем понятного имени свойства, чтобы затруднить взлом. Последняя задача - при запуске Chat проверять текущую контрольную сумму файлов по сумме из файла конфигурации. Выполняющая это часть метода main() класса ProtectedChatApplication показана в листинге 19.9.

```
public static void main(String[] args) throws Exception {
    String homePath = System.getProperty("chat.home");
    String propPath = homePath + File.separator + "conf" +
        File.separator + "chat.properties";
    AppProperties props = new AppProperties(propPath);
    String checkSum = props.getProperty("chat.versionInfo");
    IntegrityProtector protector = new IntegrityProtector();
    String[] paths = protector.readFilePathsFromResource("ChatFileList.class");
    protector.assertFilesIntegrity(paths, '\\', homePath, checkSum);
}
```

Листинг 19.9. Сравнение текущей контрольной суммы с суммой дистрибутива

После чтения исходной суммы и списка защищённых файлов метод использует `assertFilesIntegrity()` класса `IntegrityProtector` для проверки целостности содержимого. Метод `assertFilesIntegrity()` из листинга 19.10 просто вызывает `getFilesChecksum()` для заданного списка и выбрасывает `InternalError`, если вычисленная сумма не совпадает с исходной.

```
public void assertFilesIntegrity(String[] paths, char separator,
    String installPath, String checkSum) throws Exception {
    String installCheckSum = getFilesCheckSum(paths, separator, installPath);
    if (installCheckSum.equals(checkSum) == false)
        throw new InternalError("Some of the installation files are corrupt");
}
```

Листинг 19.10. Реализация `IntegrityProtector.assertFilesIntegrity()`

Закончив программирование, можно испытать защиту. Файлы Chat, предоставленные с книгой, поставляются с правильной контрольной суммой. Вы должны суметь запустить Chat файлом `chat_protected.bat` из каталога `distrib/bin`. Убедитесь, что на вашем компьютере открывается главное окно Chat. Теперь притворимся, что взламываем Chat, изменив `java.policy` в каталоге `distrib/conf`. Откройте файл, добавьте новую строку и сохраните. Убедитесь, что длина изменилась, и снова запустите `chat_protected.bat`. Должно появиться следующее исключение:

```
Exception in thread "main" java.lang.InternalError:
Some of the installation files are corrupt
    at covertjava.protect.IntegrityProtector.assertFilesIntegrity(...)
    at covertjava.protect.ProtectedChatApplication.main(...)
```

Поскольку длина файла изменилась, вычисленная контрольная сумма больше не совпадает с исходной, и `IntegrityProtector` выбрасывает ошибку. Если бы мы распространяли защищённую реализацию Chat, то, разумеется, не включили бы `getChatChecksum.bat`, удалили бы `distrib/bin/chat.bat` и метод `main()` класса `ChatApplication`. Это обеспечило бы единственный способ запуска Chat через класс `ProtectedChatApplication`. Чтобы защищённый Chat работал с новой версией файлов, потребовалось бы получить новую контрольную сумму и задать её значением свойства `chat.versionInfo` в `chat.properties`.

Реализация лицензирования для открытия функций приложения

В этом разделе рассматриваются современные способы лицензирования приложений, а затем разработка каркаса лицензирования для коммерчески распространяемого приложения.

Современные модели лицензирования программ

Распространением современных программ управляют несколько преобладающих моделей лицензирования. Условия распространения и использования обычно изложены в лицензионном соглашении конечного пользователя (EULA), поставляемом с продуктом. Хотя каждый поставщик сам формулирует условия, модели можно разделить на следующие три категории.

Коммерческие программы с закрытым исходным кодом

Это традиционная модель распространения коммерческих программ. К ней относятся собственнические продукты, например Microsoft Windows; программы, доступные для бесплатного ознакомления, например ItelliJ IDEA; продукты с бесплатной версией ограниченной функциональности, например Borland JBuilder и BEA WebLogic. Бесплатная версия с ограничениями становится всё популярнее у поставщиков Java, поскольку разработчики хотят хорошо почувствовать продукт до решения о покупке. Если продукт написан хорошо, пользователи привыкают к нему и в конце концов часто покупают полнофункциональную версию.

Коммерческие программы с открытым исходным кодом

Эта развивающаяся модель набирает популярность и позволяет конечным пользователям не только загружать и использовать программу, но и получать исходный код. Условия обычно разрешают бесплатную разработку и развёртывание, но могут требовать оплаты документации, технической поддержки или расширенных функций. Наиболее заметные примеры этой категории - сервер приложений JBoss и база данных MySQL.

Бесплатные программы с открытым исходным кодом

Программы этой категории бесплатно и без ограничений предоставляются общественности. Самая распространённая лицензия свободного открытого программного обеспечения - General Public License (GPL), разрешающая использовать программу и её исходный код в коммерческих и некоммерческих целях. Существуют варианты GPL и другие открытые лицензии, например Apache, которые могут налагать некоторые ограничения, но все они стремятся к основной идее свободы для всех.

Реализация лицензирования для открытия коммерческих функций

Когда с продуктом предоставляется исходный код, программные ограничения для принудительного соблюдения модели лицензирования, очевидно, бессмысленны. Любой пользователь легко снимет ограничения функций, изменив исходный код и пересобрав продукт. Однако большинство продуктов сегодня поставляется без исходного кода, поэтому программное принуждение к соблюдению политики лицензирования может способствовать продажам. Мы разработаем класс `LicenseManager`, который создаёт защищённые серийные номера на основе среды заказчика и типа лицензии. Класс воспользуется асимметричным алгоритмом, чтобы серийные номера мог выпускать только поставщик программы. Даже если управление лицензиями вам не требуется, раздел полезен практическим примером API безопасности и криптографии Java.

При выборе конструкции лицензирования необходимо продумать самый эффективный способ предотвращать нарушения политики, не ухудшая впечатления заказчика. Например, лицензия, открывающая коммерческие функции без привязки к среде конечного пользователя, небезопасна. Её проще выдавать, поскольку пользователю не нужно отправлять сведения поставщику, но если кто-то разместит такую лицензию в Интернете, она может превратиться в кошмар распространения. Лицензию следует привязать к параметру среды заказчика, например имени узла, IP-адресу или имени домена. Тогда даже попавшая в Интернет лицензия не будет работать в среде, для которой не выдавалась. Ещё одно важное соображение - возможность открывать ограниченные функции через файл лицензии без сопровождения и сборки нескольких версий программы. Время истечения внутри файла полезно, если поставщик хочет выдать временную лицензию для ознакомления.

Например, если бы у Chat был коммерческий потенциал, мы могли бы распространять бесплатную версию с ограничениями, позволяющую отправлять обычные текстовые сообщения одному пользователю за раз. Затем можно было бы реализовать дополнительные функции: текст HTML, цвета, списки друзей, смайлики и изображения. Теоретически мы могли бы собирать два экземпляра Chat: ограниченный и полнофункциональный. Но на практике сопровождать такой код очень трудно, поэтому, подобно большинству поставщиков, мы предпочли бы единую кодовую базу полной версии. Чтобы ограничить доступ к коммерческой функции, необходимо в ключевых местах кода проверять наличие файла лицензии и разрешение функции. При неудаче проверок функция отключается. Тогда Chat без файла лицензии можно было бы предложить для бесплатной загрузки. Если пользователь захотел бы расширенные функции нашего замечательного приложения, ему пришлось бы купить лицензию. После получения оплаты мы выпустили бы лицензию по имени узла пользователя и отправили ему файл. При следующем запуске Chat прочитал бы лицензионные сведения и включил приобретённые функции.

Сведения о разрешённых и отключённых функциях можно зашифровать в файле лицензии, но тогда его трудно читать и сопровождать. Хранить параметры обычным текстом проще, но это подобно кусочку курицы перед голодным крокодилом. Например, лицензия может быть выдана определённому узлу, но даже самый неискушённый пользователь способен скопировать файл на другой узел и изменить имя. Самое чистое решение - создать защищённую цифровую подпись параметров лицензии и хранить её в файле вместе с параметрами обычным текстом. Подпись создаётся служебной программой лицензирования с закрытым ключом асимметричного алгоритма. При запуске приложение применяет открытый ключ алгоритма и подпись для проверки подлинности сведений из файла лицензии. Ограничения снимаются только после успешной проверки.

> **Истории с передовой.** WebCream - популярный инструмент Java, динамически преобразующий приложения и апплеты с графическим интерфейсом Swing в интерактивные веб-сайты HTML. Это коммерческий продукт компании CreamTec. Чтобы убедить разработчиков оценить продукт, мы решили сделать стандартную версию WebCream бесплатной. Стандартная версия полнофункциональна и способна выполнять полное преобразование. Для продвижения коммерческих лицензий были введены

ограничения на число одновременных пользователей и некоторые расширенные возможности настройки.

Изначально мы собирали три разных редакции из слегка различавшегося кода. При множестве платформ и версий установщика сборка выпусков занимала больше дня. Чтобы упростить сопровождение и укладываться в жёсткие сроки, мы решили поддерживать одну кодовую базу для коммерческой и бесплатной редакций. В бесплатной редакции коммерческие функции просто блокировались и открывались лишь при наличии файла лицензии. Этот файл содержит зашифрованные сведения о разрешённых лицензией узлах, числе одновременных пользователей и доступе к коммерческим функциям. Подход значительно упростил распространение и управление несколькими редакциями.

Создание файла лицензии

Приступим к разработке универсального диспетчера лицензий и применим его к Chat. Для простоты файл лицензии будет иметь формат свойств Java. Лицензирование основано на трёх параметрах: имени узла, IP-адресе и дате истечения. Поскольку я не хотел тратить время на написание всех прибыльных функций Chat, этого простого примера достаточно. Создайте в `CovertJava/distrib/conf` новый файл `chat.license`, как в листинге 19.11.

```
host=localhost
ip=172.24.109.159
expires=2005/1/1
serial=
```

Листинг 19.11. Файл лицензии Chat

Укажите свои имя узла и IP-адрес, а значение свойства `serial` пока оставьте пустым: к нему мы вернёмся. Необходимо написать два класса: один для создания лицензии, другой для проверки. Мы не хотим включать код генерации лицензий в дистрибутив, поэтому нужны два класса. Поскольку оба должны читать сведения из файла, разумно унаследовать один от другого. Начнём с класса `LicenseManager` в пакете `covertjava.protect`. Определим поля-члены для свойств лицензии (см. листинг 19.12).

```
public class LicenseManager {
    private String host;
    private String ip;
    private Date expires;
    ...
}
```

Листинг 19.12. Объявление `LicenseManager`

Затем дадим ему конструктор, принимающий имя файла лицензии и заполняющий внутренние поля сведениями из него, как показано в листинге 19.13.

```
public LicenseManager(String licenseFileName) throws Exception {
    this.licenseProps = new AppProperties(home+File.separator+licenseFileName);
    this.host = licenseProps.getProperty("host");
    this.ip = licenseProps.getProperty("ip");
    String expiresString = licenseProps.getProperty("expires");
    this.expires = this.dateFormat.parse(expiresString);
    ...
}
```

Листинг 19.13. Инициализация `LicenseManager`

Чтобы защитить параметры лицензии от изменений, нужна цифровая подпись. Все алгоритмы JCE работают с массивами байтов, поэтому добавим метод `getLicenseString()`, возвращающий единое представление всех свойств лицензии. Его исходный код показан в листинге 19.14.

```
protected String getLicenseString() throws Exception {
    return this.host + this.ip + this.expires;
}
```

Листинг 19.14. Единое представление свойств лицензии

Пока оставим `LicenseManager` и начнём работать над `LicenseGenerator`. Он должен расширять `LicenseManager` и предоставлять методы создания цифровой подписи. В качестве серийного номера используем цифровую подпись в кодировке base64. Для подписи также нужна пара ключей асимметричного алгоритма. Ключи можно создать утилитой JDK `keytool` или программно посредством API безопасности Java. С `keytool` ключи создаются и экспортируются всего несколькими командами, но ради академического интереса выберем программный подход. Сначала нужно выбрать алгоритм и длину ключа. Стандартные асимметричные алгоритмы - DSA и RSA. Оба при правильном размере дают достаточную защиту, но мы используем DSA, поскольку он непосредственно поддерживается Sun JCE из состава JRE. Размер ключа прямо влияет на сложность шифрования: чем длиннее ключ, тем труднее его вскрыть. Каждый бит удваивает время взлома. Современный процессор вскрывает 16-битные ключи за минуты, а 1024-битные считаются невскрываемыми, поскольку даже вся вычислительная мощность Земли потребовала бы миллионов лет (как говорят). Поскольку расшифрование не выполняется в реальном времени, используем размер 1024 бита. В листинге 19.15 показан метод `LicenseGenerator`, создающий пару ключей.

```
public void generateKeys() throws Exception {
    KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
    SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
    random.setSeed(System.currentTimeMillis());
    keyGen.initialize(1024, random);
    KeyPair pair = keyGen.generateKeyPair();

    String publicKeyPath = home + File.separator + "conf"
        + File.separator + "key_public.ser";
    byte[] bytes = pair.getPublic().getEncoded();
    FileOutputStream stream = new FileOutputStream(publicKeyPath);

    stream.write(bytes);
    stream.close();
    ...
}
```

Листинг 19.15. Создание ключей алгоритма DSA

Сначала реализация получает экземпляр генератора пар DSA. Генератор необходимо инициализировать размером ключа (1024 бита) и провайдером случайных чисел (мы используем `SecureRandom` компании Sun). Затем ключи создаются вызовом `generateKeyPair()`. После создания пары остаётся сохранить открытый и закрытый ключи на диск. В листинге 19.15 показано сохранение открытого ключа в `key_public.ser` в каталоге `Conf`. Не показанная оставшаяся часть метода таким же образом сохраняет закрытый ключ в `key_private.ser`.

Очевидно, ключи нужно создать лишь однажды. У `LicenseGenerator` есть метод `main()`, вызывающий `generateKeys()` или `generateSerialNumber()` в зависимости от параметров командной строки. Реализацию генерации ключей мы уже видели, теперь рассмотрим серийный номер. Как говорилось ранее, это цифровая подпись единого набора свойств лицензии в кодировке base64. Метод `generateSerialNumber()` из листинга 19.16 делает именно это.

```
public String generateSerialNumber() throws Exception {
    String licenseString = getLicenseString();
    byte[] serialBytes = licenseString.getBytes(CHARSET);
    serialBytes = getSignature(serialBytes);
    BASE64Encoder encoder = new BASE64Encoder();
    return encoder.encode(serialBytes);
}
```

Листинг 19.16. Создание серийного номера

Байты строки серийного номера передаются методу `getSignature()`, а его результат преобразуется в строку классом `BASE64Encoder`. Это приводит нас к реализации цифрового подписания алгоритмом DSA, показанной в листинге 19.17.

```
private byte[] getSignature(byte[] serialBytes) throws Exception {
    String privateKeyPath = home + File.separator + "conf" +
        File.separator + "key_private.ser";
    FileInputStream stream = new FileInputStream(privateKeyPath);
    byte[] encodedPrivateKey = new byte[stream.available()];
    stream.read(encodedPrivateKey);

    PKCS8EncodedKeySpec pubKeySpec = new PKCS8EncodedKeySpec(encodedPrivateKey);
    KeyFactory keyFactory = KeyFactory.getInstance("DSA");
    PrivateKey key = keyFactory.generatePrivate(pubKeySpec);
    Signature dsa = Signature.getInstance("SHA1withDSA");
    dsa.initSign(key);
    dsa.update(serialBytes);
    return dsa.sign();
}
```

Листинг 19.17. Цифровое подписание с DSA

Мы подписываем сведения лицензии закрытым ключом, чтобы никто другой не мог создавать лицензии. Закрытый ключ читается из файла в массив байтов `encodedPrivateKey`. Затем класс `PKCS8EncodedKeySpec` из пакета `java.security.spec` преобразует двоичное представление во внутреннее представление ASN.1. После этого фабрика DSA преобразует представление в экземпляр `PrivateKey`. Имея ключ и байты серийного номера, мы получаем экземпляр безопасного хеша с алгоритмом DSA (`SHA1withDSA`), передаём параметры и создаём цифровую подпись методом `sign()`. Выполнение сценария генератора `licenseGenerator.bat` для `chat.license` из каталога `distrib/conf` даёт:

```
C:\CovertJava\bin>licenseGenerator.bat -serial distrib/conf/chat.license
License information read:
host=localhost
ip=172.24.109.159
expires=Sat Jan 01 00:00:00 EST 2005
Serial=[MC0CFBiEzKka0pnEQSlDyKxbHy+gE1+zAhUAlxPlWyAXcCDcoWSRY/Kk/xAkvTQ=]
```

Теперь созданный серийный номер нужно скопировать и вставить значением свойства `serial` в `chat.license`. Создание лицензии завершено.

Проверка файла лицензии

Необходимо расширить Chat: прочитать серийный номер и проверить, не подделаны ли параметры лицензии. Поскольку серийный номер Chat фактически является цифровой подписью параметров, нужно написать метод, использующий открытый ключ пары для проверки подписи. Добавим в написанный ранее `LicenseManager` метод `verifySerialNumber()`. Для проверки цифровой подписи, созданной закрытым ключом, метод должен применить открытый ключ. Свойства лицензии и серийный номер были прочитаны конструктором `LicenseManager` и сохранены в переменных-членах. Исходный код `verifySerialNumber()` показан в листинге 19.18.

```
public void verifySerialNumber(String keyFileName) throws Exception {
    String keyFilePath = this.home + File.separator + keyFileName;
    FileInputStream stream = new FileInputStream(keyFilePath);
    byte[] encodedPubKey = new byte[stream.available()];
    stream.read(encodedPubKey);
    X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(encodedPubKey);
    KeyFactory keyFactory = KeyFactory.getInstance("DSA");
    PublicKey publicKey = keyFactory.generatePublic(pubKeySpec);

    byte[] licenseData = getLicenseString().getBytes(CHARSET);
    String encodedSig = this.licenseProps.getProperty("serial");
    if (encodedSig == null || encodedSig.length() == 0)
        throw new InternalError("Serial number is missing");
    BASE64Decoder decoder = new BASE64Decoder();
    byte[] serialSig = decoder.decodeBuffer(encodedSig);
    Signature signature = Signature.getInstance("SHA1withDSA");
    signature.initVerify(publicKey);
    signature.update(licenseData);
    if (signature.verify(serialSig) == false)
        throw new InternalError("Invalid serial number");
}
```

Листинг 19.18. Проверка серийного номера

Сначала метод читает содержимое файла открытого ключа и с помощью класса `X509EncodedKeySpec` и фабрики DSA преобразует двоичное представление ключа в экземпляр интерфейса `PublicKey`. Затем единое представление параметров лицензии, возвращаемое `getLicenseString()`, преобразуется в массив байтов. Серийный номер читается как значение свойства `serial` из файла лицензии и, если он не отсутствует, декодируется из base64 классом `BASE64Decoder`. Получается экземпляр алгоритма подписания `SHA1withDSA`, которому передаются открытый ключ и данные лицензии. Наконец, метод `verify()` алгоритма проверяет, является ли серийный номер правильной цифровой подписью лицензионных данных. Если нет, сообщается исключение `InternalError`.

Чтобы встроить проверку лицензии в Chat, нужно добавить вызов `verifySerialNumber()` в метод `main()` класса `ProtectedChat`. Поскольку в `main()` уже есть экземпляр `LicenseManager`, достаточно добавить блок из листинга 19.19.

```
public static void main(String[] args) throws Exception {
    ...
    // Check license information
    licenseManager.verifySerialNumber("conf/key_public.ser");
    if (licenseManager.isHostAllowed() == false)
        throw new Exception("Host is not allowed by the license");
    if (licenseManager.isLicenseExpired() == true)
        throw new Exception("The license is expired");
}
```

Листинг 19.19. Вызов проверки лицензии

Если метод `verifySerialNumber()` класса `LicenseManager` не выбрасывает исключение, проверяются имя узла и срок действия лицензии. Имя проверяется простым сравнением строки из файла лицензии с именем узла, где работает Chat. Срок проверяется столь же простым

сравнением текущей системной даты с прочитанной датой истечения. Коммерческие функции Chat открываются, только если обе проверки успешны. Пока код `LicenseManager` и `ProtectedChat` не взломан, механизм лицензирования довольно надёжен.

Интересный способ вставить лицензионные проверки - инструментирование байт-кода из главы 17, «Понимание и модификация байт-кода». Вместо ручных вызовов методов `LicenseManager` по классам приложения можно разработать служебный постпроцессор, декорирующий ключевые методы кодом проверки лицензии. Утилита выполняется после компиляции исходного кода, но до помещения в JAR дистрибутива. Вставленный байт-код выбрасывает исключение или возвращает ошибку, если лицензия недействительна или функция не разрешена. Так логика приложения чётко отделяется от кода лицензирования.

Веб-активация и регистрация лицензии

Описанный механизм лицензирования обеспечивает значительную защиту от пиратства. Но если проверка взломана, отследить распространение скомпрометированного продукта трудно, особенно при его появлении в Интернете. Хорошая стратегия - дублировать вызовы методов проверки по всему коду приложения. В примере Chat создаёт и использует `LicenseManager` только в методе `main()` класса `ProtectedChat`. Для дополнительной защиты те же методы следует вызывать в коде `MainFrame` или `ChatServer`. Ещё одна мера - активация и регистрация через Интернет.

Идея веб-регистрации состоит в том, что при каждом запуске приложение соединяется с веб-сайтом поставщика и проверяет действительность лицензии. Это позволяет поставщику отслеживать число установленных версий и отключать лицензии или сборки, которые заведомо взломаны. Подключение к поставщику также даёт дополнительные преимущества, например автоматические обновления и сбор статистики использования.

Но сетевое подключение не следует считать основным способом активации и регистрации. Продукты могут устанавливаться и работать в контролируемой изолированной среде за корпоративными брандмауэрами, полностью закрывающими Интернет. Отправка сведений о заказчике на веб-сайт поставщика также может вызвать опасения о конфиденциальности.

Короткий тест

1. Чем различаются алгоритмы свёртки сообщения, шифрования и подписания?
2. В чём разница между симметричными и асимметричными алгоритмами?
3. Как защитить содержимое электронного письма, отправляемого через Интернет? Какие классы JCE потребуются?
4. Какие меры можно принять для защиты содержимого приложения от взлома?
5. Мы использовали алгоритм свёртки для вычисления контрольной суммы файлов дистрибутива Chat. Стала бы контрольная сумма безопаснее при использовании симметричного или асимметричного алгоритма? Почему?
6. Какие логические шаги нужны для получения цифровой подписи симметричным алгоритмом?
7. Какие логические шаги нужны для получения цифровой подписи асимметричным алгоритмом?

Краткие итоги

• Криптография позволяет преобразовать читаемые сведения в непонятный код, который можно открыто передать, а затем вернуть в исходную форму.
• Шифрование - процесс кодирования читаемых сведений, а расшифрование - извлечение читаемых сведений из кода.
• Алгоритмы свёртки сообщений не изменяют содержимое сообщения, а создают уникальный хеш на основе содержимого и секретного ключа.

• Алгоритмы шифрования и расшифрования скрывают конфиденциальные сведения, которые может перехватить третья сторона. Содержимое изменяется секретным ключом, образуя результат, который без ключа практически невозможно вернуть в исходную форму.
• Подписание означает создание сравнительно короткой цифровой подписи на основе данных произвольного размера с помощью закрытого ключа асимметричного алгоритма. Отправитель создаёт подпись и передаёт её с сообщением. Подлинность обеспечивается асимметричными алгоритмами, где закрытый ключ есть только у отправителя.
• Архитектура криптографии Java (JCA) предоставляет полную и надёжную реализацию криптографических служб и алгоритмов.
• Алгоритмы шифрования, свёртки сообщений и подписания можно применять для защиты взаимодействия между уровнями распределённого приложения.
• Большинство алгоритмов безопасности требует параметров, например пароля или ключей. Обычно они работают с двоичными данными.
• После обфускации байт-кода проверка целостности файлов дистрибутива - важнейшая мера защиты приложения от взлома.
• Самый эффективный способ реализовать лицензирование, открывающее коммерческие функции, - предоставить текстовый файл лицензии с цифровой подписью, созданной асимметричным алгоритмом.