ГЛАВА 11

Расширенная отладка и анализ

К этому моменту, изучая разработку под Android, вы уже наверняка построили несколько приложений и могли столкнуться с неожиданным их поведением. В этой главе мы рассмотрим различные способы отладки приложений, которые позволяют выяснить, что в них работает не так, как было задумано. Несмотря на то что существуют методы делать это без использования подключаемого модуля Eclipse под названием ADT, в этой главе мы собираемся применять именно его.

Перспектива (т.е. внешний вид окна с редакторами и представлениями) отладки (Debug perspective) в Eclipse является стандартной и не предназначена специально для программирования под Android. Тем не менее, вы должны знать, что с ней можно делать. Перспектива DDMS поддерживает несколько очень полезных средств, которые помогают отлаживать приложения. Она включает следующие представления: Devices (Устройства), позволяющее просматривать, к чему произведено подключение; Emulator Control (Управление эмулятором) для отправки телефонных вызовов, SMS-сообщений и GPS-координат; File Explorer (Проводник файлов), предназначенное для просмотра/передачи файлов на устройство; Threads (Потоки), Heap (Куча) и Allocation Tracker (Отслеживание выделения памяти) для анализа внутренних особенностей приложения. Мы также рассмотрим перспективу Hierarchy View (Представление иерархии), которая позволяет проходить по действительной структуре представлений выполняющегося приложения. После этого мы приведем краткий обзор представления Traceview (Трассировка), которое существенно упрощает анализ файла с дампом приложения. Наконец, мы опишем класс `StrictMode`, используемый для перехвата нарушений политик, что позволяет отловить проектные ошибки, которые могут привести к ухудшению работы пользователей.

Включение расширенной отладки

При проведении тестирования в эмуляторе подключаемый модуль ADT заботится о полной настройке среды, поэтому для использования будут доступны все инструменты, которые рассматриваются в этой главе.

Об отладке приложений на реальном устройстве необходимо знать два момента. Первый из них — приложение должно быть установлено как поддерживающее отладку. Для этого нужно добавить атрибут `android:debuggable="true"` к дескриптору `<application>` в файле `AndroidManifest.xml`. К счастью, ADT справляется с этим самостоятельно. При создании отладочных сборок для эмулятора или развертывании Eclipse прямо на устройство упомянутый атрибут устанавливается в `true` подключае-

мым модулем ADT. При экспорте приложения для создания его производственной версии ADT известно о том, что устанавливать `debuggable` в `true` не следует. Обратите внимание, что если установить этот атрибут вручную в файле `AndroidManifest.xml`, он останется в таком состоянии. Второй момент, который следует знать, связан с тем, что устройство должно быть переведено в режим отладки через USB. Чтобы найти этот параметр, перейдите на экран Settings (Параметры) устройства, выберите пункт Application (Приложения), а затем Development (Разработка). Удостоверьтесь, что флажок Enable USB Debugging (Включить отладку через USB) отмечен.

Перспектива Debug

Хотя объект `LogCat` очень удобен для просмотра журнальных сообщений, вас наверняка интересует большая степень контроля и больший объем информации во время выполнения приложения. Отладка в Eclipse исключительно проста и подробно описана на множестве сайтов в Интернете. Таким образом, мы не будем здесь особо вдаваться в детали, а рассмотрим лишь некоторые полезные возможности, которые в ней доступны.

- Настройка в коде точек останова, при достижении которых выполнение прекращается, но может быть возобновлено.
- Проверка значений переменных.
- Выполнение с обходом и заходом в строки кода.
- Подключение отладчика к уже выполняющемуся приложению.
- Отключение от ранее подключенного приложения.
- Просмотр трассировки стека.
- Просмотр списка потоков.
- Просмотр `LogCat`.

На рис. 11.1 показано, как выглядит перспектива Debug.

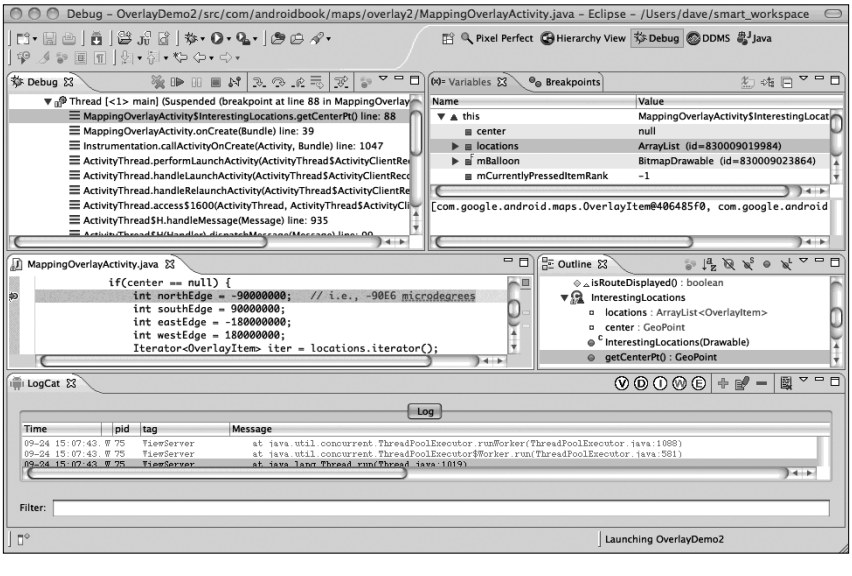

Рис. 11.1. Перспектива Debug

Запустить отладку приложения можно из перспективы Java (где пишется код), щелкнув правой кнопкой мыши и выбрав в контекстном меню пункт Debug As⇨Android Application (Отладить как⇨Android-приложение); это приведет к запуску приложения. Для проведения отладки может понадобиться переключение на перспективу Debug.

Перспектива DDMS

Аббревиатура DDMS обозначает Dalvik Debug Monitor Server (Сервер монитора отладки Dalvik). Эта перспектива позволяет разобраться в работе приложений, запускаемых в эмуляторе или на устройстве, давая возможность просматривать потоки и память, а также собирать статистику по выполнению приложения. На рис. 11.2 показано, как это может выглядеть на рабочей станции. Хотя в оставшейся части раздела используется термин *устройство*, имеется в виду устройство или эмулятор.

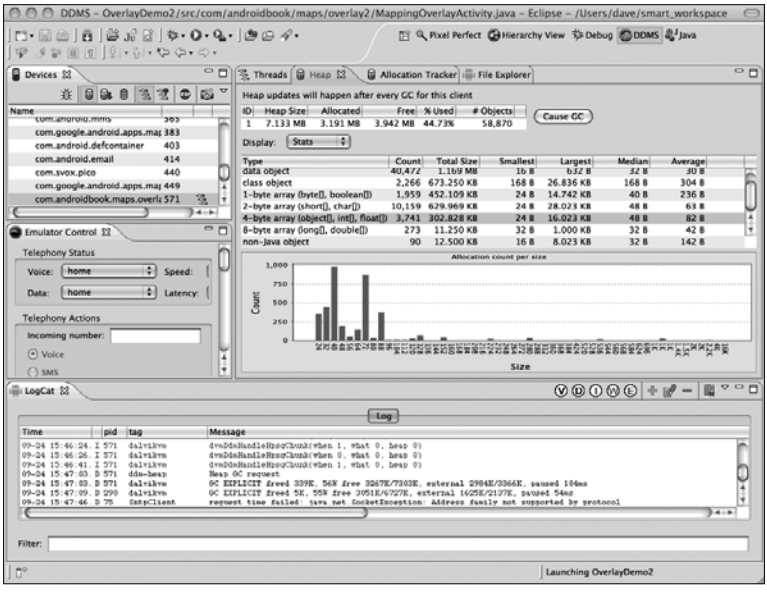

Рис. 11.2. Перспектива DDMS

В левом верхнем углу экранного снимка на рис. 11.2 видно представление Devices (Устройства). В нем перечислены все устройства, подключенные к рабочей станции (одновременно может быть подключено множество устройств или эмуляторов), а после раскрытия какого-то устройства — все приложения, доступные для отладки. В данном конкретном случае просматривается эмулятор, поэтому для отладки доступно множество приложений (несмотря на отсутствие исходного кода). На реальном устройстве можно наблюдать только несколько приложений или вообще ни одного. Не забывайте, что для отладки производственной версии приложения на реальном устройстве в файле `AndroidManifest.xml` может понадобиться установить `android:debuggable` в `true`.

В представлении Devices имеются кнопки для запуска отладки приложения, а также кнопки для обновления информации о куче, получения файла HPROF (Heap and CPU Profiling Agent — агент профилирования памяти кучи и центрального процессора), вызова сборщика мусора (garbage collection — GC), обновления списка потоков, запуска профилирования методов, останова процесса и получения снимка экрана устройства. Ниже эти кнопки будут описаны более подробно.

Кнопка с небольшим зеленым жуком (первая слева) запускает отладку выбранного приложения. Щелчок на ней переносит в только что описанную перспективу Debug. Важно отметить, что есть возможность подключения отладчика к уже запущенному приложению. Вы можете привести приложение в состояние, с которого необходимо начать отладку, после чего щелкнуть на этой кнопке. С этого момента будут активизироваться точки останова, и можно будет просматривать значения переменных, а также пошагово выполнять код.

Следующая по порядку кнопка служит для просмотра памяти кучи выполняющегося процесса. Желательно, чтобы приложения использовали минимально необходимый для их работы объем памяти и не выделяли память слишком часто. Подобно кнопке запуска отладки, необходимо выбрать приложение и щелкнуть на кнопке Update Heap (Обновить кучу). Должны выбираться только приложения, которые в настоящий момент отлаживаются. На вкладке Heap (Куча) справа (см. рис. 11.2) теперь можно щелкнуть на кнопке Cause GC (Запустить сборщик мусора), чтобы собрать сведения о памяти в куче. Вверху отображаются итоговые результаты, а ниже — детальные сведения. Кроме того, для каждого типа и размера выделенной памяти можно просмотреть дополнительные детали.

Кнопка Dump HPROF File (Получить файл HPROF) позволяет получить файл HPROF. Если в системе установлен подключаемый модуль для анализа памяти Eclipse Memory Analyzer (MAT), файл HPROF будет обработан, а результаты отображены. Это мощный способ обнаружения утечек памяти. По умолчанию файл HPROF открывается в Eclipse. Этим управляет один из параметров в диалоговом окне Preferences (Настройка), в узле Android⇨DDMS, где можно выбрать сохранение файла HPROF вместо его открытия.

Кнопка Update Threads (Обновить вкладку для потоков) вызывает заполнение вкладки Threads (Потоки), расположенной справа, текущим набором потоков из выбранного приложения. Это удобный способ слежения за созданием и уничтожением потоков, а также выяснения того, что происходит на уровне потоков внутри приложения. Ниже списка потоков можно видеть, в каком состоянии находится поток, отслеживая то, что выглядит похожим на трассировку стека (объекты, ссылка на файл исходного кода и номер строки).

Кнопка Start Method Profiling (Начать профилирование методов) позволяет собрать информацию о методах внутри приложения, включая количество вызовов и продолжительность выполнения. Необходимо щелкнуть на этой кнопке, поработать в приложении и затем щелкнуть на кнопке еще раз (надпись на кнопке будет переключаться между Start Method Profiling и Stop Method Profiling (Остановить профилирование методов)). Щелчок на Stop Method Profiling приводит к тому, что Eclipse отобразит представление Traceview, которое будет рассматриваться далее в этой главе.

Кнопка Stop (Остановить) с изображением соответствующего ей знака позволяет остановить выбранный процесс. Это жесткий останов приложения, не похожий на нажатие кнопки Back (Назад), которое оказывает влияние только на активность. В данном случае приложение исчезает.

Кнопка с изображением фотоаппарата вызывает захват текущего состояния экрана устройства, независимо от того, какое приложение выбрано в представлении Devices. Полученное изображение можно обновлять, поворачивать или копировать.

Наконец, возле кнопки с изображением фотоаппарата имеется меню, которое содержит функции всех перечисленных кнопок, а также дополнительный элемент Reset adb (Сброс `adb`). Выбор этого пункта приводит к перезапуску сервера `adb`, который взаимодействует с устройствами, в случае утери синхронизации или доступа к устройству. Это должно вызвать обновление списка устройств в представлении Devices. Другой способ сброса сервера `adb` предусматривает выдачу следующей пары команд в окне инструментов:

```
adb kill-server
adb start-server
```

В правой части окна на рис. 11.2 можно заметить вкладку Allocation Tracker (Отслеживание выделения памяти). Она позволяет запустить отслеживание отдельных выделений памяти. После щелчка на кнопке Start Tracking (Начать отслеживание) необходимо поработать с приложением и затем щелкнуть на кнопке Get Allocations (Получить выделения памяти). Отобразится список выделений памяти за истекший период времени, в котором можно щелкать на любом выделении для просмотра информации, откуда оно поступило (класс, метод, ссылка на файл исходного кода и номер строки). Щелчок на кнопке Stop Tracking (Остановить отслеживание) позволяет сбросить этот список и начать его заново.

Перспектива DDMS также имеет представления File Explorer (Проводник файлов) и Emulator Control (Управление эмулятором), так что у вас есть возможность моделировать получение входящих телефонных звонков, SMS-сообщений или GPS-координат. Представление File Explorer позволяет просматривать файловую систему на устройстве и даже обмениваться файлами между устройством и рабочей станцией. Дополнительные сведения о File Explorer будут представлены в главе 24, в разделе, посвященном SD-картам. Представление Emulator Control будет использоваться в главах 22 и 23.

Перспектива Hierarchy View

В этой перспективе производится подключение к запущенному экземпляру приложения в эмуляторе (не на реальном устройстве), после чего можно исследовать представления в приложении, их структуру и свойства. Для начала понадобится выбрать интересующее приложение. Иерархия затем может быть отображена различными способами, как показано на рис. 11.3.

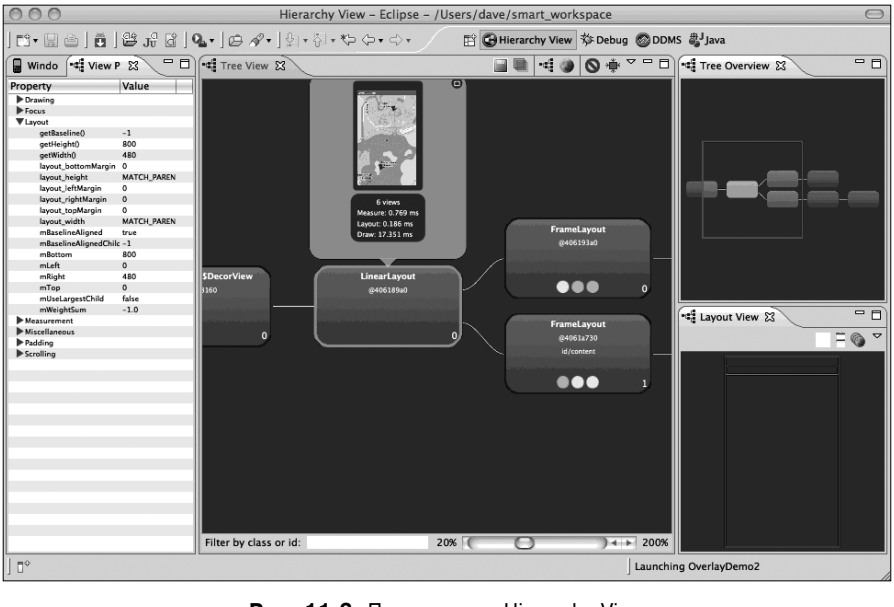

Рис. 11.3. Перспектива Hierarchy View

Здесь имеется возможность навигации по структуре, проверки свойств и выяснения, что представлений в приложении определено ровно столько, сколько необходимо. Например, при наличии множества компоновок, возможно, имеет смысл заменить их единственной компоновкой `RelativeLayout`.

Вероятно, вы отметили наличие трех цветных кружков в представлениях по центру окна. Они отражают оценку производительности этого представления в терминах измерения, компоновки и рисования представления (включая внутренние представления). Цвета являются относительными, поэтому красный кружок не обязательно означает что-то неприемлемое, но, несомненно, указывает на то, чему следует уделить дополнительное внимание.

Кроме того, обратите внимание на выбранное представление и сведения над ним. Сведения содержат не только экранный снимок этого представления, но также и показания времени измерения, компоновки и рисования представления. Это действительно важные числа для анализа представления с целью внесения в него усовершенствований. Помимо упомянутого ранее сокращения числа компоновок, можно внести изменения в инициализацию представлений и оптимизировать их рисование. Если в коде создается множество объектов, во избежание накладных расходов имеет смысл поискать возможность их повторного использования. Для выполнения длительных по времени работ лучше применять потоки или `AsyncTask`.

Подключаемый модуль Pixel Perfect View

Подобно Hierarchy View, получать изображение экрана можно и в Pixel Perfect View. Этот подключаемый модуль Eclipse предлагает средство для просмотра изображений с возможностью увеличения масштаба, которое позволяет разглядывать изображение вплоть до отдельных точек с ассоциированными с ними цветами. Это средство интересно тем, что с его помощью можно накладывать изображения друг на друга (например, макет экрана и реальный его снимок) с целью сравнения. При попытках воспроизведения конкретного внешнего вида Pixel Perfect View является полезным инструментом, помогающим выяснить, все ли делается правильно.

Представление Traceview

Ранее было показано, как можно собрать статистические данные по выполнению методов в приложении. С помощью DDMS можно провести профилирование методов, после которого отобразится представление трассировки Traceview с полученными результатами. На рис. 11.4 показано, как это выглядит.

Используя описанные ранее приемы, можно собрать результаты профилирования для всех методов в приложении. Кроме того, посредством класса `android.os.Debug` можно получить более специфичную трассировочную информацию для Android-приложения. Этот класс предоставляет метод для запуска трассировки (`Debug.startMethodTracing("базовое_имя")`) и метод для ее останова (`Debug.stopMethodTracing()`). Android создаст на SD-карте устройства файл трассировки с именем `базовое_имя.trace`. Код для запуска и останова трассировки помещается вокруг интересующего участка, и это ограничивает объем данных, собираемых в файле трассировки. Файл трассировки затем можно скопировать на рабочую станцию и просмотреть его с помощью инструмента `traceview`, находящегося в подкаталоге `tools` каталога Android SDK; этому инструменту понадобится передать единственный аргумент — имя файла трассировки. В главе 24 приведено подробное изложение всех аспектов, связанных с SD-картой, включая извлечение из нее нужных файлов.

Вы заметите, что результаты анализа отражают, какие методы вызывались, насколько часто, и сколько времени каждый из них отработал. Сведения организованы по потокам, с цветовым кодированием. Используйте этот экран для нахождения методов, которые слишком долго выполняются или слишком часто вызываются.

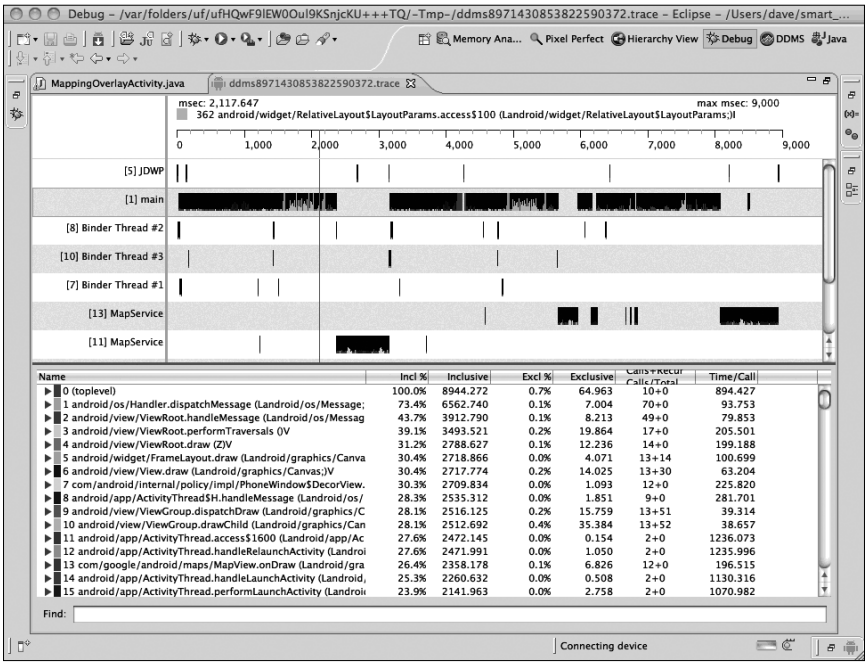

Рис. 11.4. Представление Traceview

Команда adb

Доступно несколько других средств отладки, которые можно использовать в командной строке (или в окне инструментов). Команда `adb` (Android Debug Bridge — мост отладки Android) позволяет устанавливать, обновлять и удалять приложения. В эмуляторе или на устройстве можно запустить командную оболочку, из которой затем выдавать подмножество команд Linux, предоставляемых Android. Например, можно просматривать файловую систему, выводить список процессов, читать журнал и даже подключаться к базам данных SQLite для выполнения команд SQL. Команды SQLite рассматривались в главе 4. В качестве примера, следующая команда (выданная в окне инструментов) запускает командную оболочку в эмуляторе:

```
adb -e shell
```

Обратите внимание на использование `-e` для указания на эмулятор. При подключении к устройству будет применяться `-d`. Внутри командной оболочки эмулятора будут доступны повышенные права Linux, тогда как на реальном устройстве — нет. Это значит, что эмулятор позволяет работать с базами данных SQLite, а реальное устройство не позволяет, даже внутри собственного приложения. Ввод `adb` без аргументов приведет к отображению описания доступных возможностей команды `adb`.

Консоль эмулятора

Еще один прием отладки предусматривает запуск консоли эмулятора (Emulator Console), которая, как должно быть понятно, работает только с эмулятором. Сначала, имея запущенный эмулятор, необходимо ввести в окне инструментов следующую команду:

```
telnet localhost порт#
```

Здесь *порт#* — это номер порта, который прослушивает эмулятор. Значение *порт#* обычно отображается в заголовке окна эмулятора и часто равно 5554. После открытия консоли эмулятора можно вводить команды для моделирования событий GPS, SMS-сообщений и даже изменений в состоянии батареи и сети. В разделе "Ссылки" в конце главы приведена ссылка на описание команд консоли эмулятора.

Класс StrictMode

В версии Android 2.3 появилось новое отладочное средство, которое называется классом StrictMode. Согласно Google, это средство использовалось для внесения сотен усовершенствований в приложения Google, доступные для Android. Так что же делает класс StrictMode? Он будет сообщать о нарушениях политик, имеющих отношение к потокам и виртуальной машине. Когда обнаруживается нарушение политики, выдается предупреждение, которое будет включено в трассировку стека, чтобы показать, в каком состоянии находилось приложение, когда произошло это нарушение политики. Получив такое предупреждение, можно принудительно завершить работу приложения или же просто зафиксировать предупреждение в журнале и предоставить заботу о нем самому приложению.

Политики StrictMode

В настоящее время со StrictMode доступны два типа политик. Первый тип относится к потокам и предназначен в основном для запуска не в главном потоке (в потоке пользовательского интерфейса). Чтение и запись на диск выполнять в главном потоке не рекомендуется; то же самое касается и доступа в сеть. В Google добавили в StrictMode привязки к дисковому и сетевому коду, так что если включить StrictMode для одного из потоков, и этот поток выполнит операцию с диском или сетью, будет выдано предупреждение. Понадобится выбрать аспекты, о которых ThreadPolicy должен предупреждать, а также метод предупреждения. Некоторые нарушения, которые могут интересовать, включают медленные вызовы, операции чтения с диска, операции записи на диск и операции доступа к сети. В качестве метода предупреждения может быть выбрана запись в LogCat, отображение диалогового окна, включение подсветки экрана, запись в журнальный файл DropBox или аварийное завершение приложения. Наиболее часто применяемыми вариантами являются запись в LogCat и аварийное завершение приложения. В листинге 11.1 приведен пример настройки StrictMode для политик, связанных с потоком.

Листинг 11.1. Установка объекта ThreadPolicy внутри StrictMode

```
StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
    .detectDiskReads()
    .detectDiskWrites()
    .detectNetwork()
    .penaltyLog()
    .build());
```

Обратите внимание, что класс Builder делает настройку StrictMode действительно простой. Все методы Builder, которые определяют политику, возвращают ссылку на объект Builder, поэтому вызовы таких методов могут быть соединены вместе в цепочку, как показано в листинге 11.1. Последний вызванный метод, build(), возвращает объект ThreadPolicy, который ожидается методом setThreadPolicy() класса StrictMode. Также следует отметить, что setThreadPolicy() является статическим методом, поэтому создавать объект StrictMode не требуется. Внутренне метод set-

`ThreadPolicy()` использует текущий поток для политики, так что последующие действия в потоке будут проверяться на предмет `ThreadPolicy` с выдачей предупреждений в случае необходимости. В приведенном примере политика определяется для выдачи предупреждений при операциях чтения с диска, записи на диск и доступа к сети, и она предусматривает занесение сообщений в `LogCat`. Вместо специфических методов обнаружения можно было бы воспользоваться методом `detectAll()`. Кроме того, допускается применять другие или дополнительные методы предупреждения. Например, можно было бы использовать метод `penaltyDeath()` для аварийного завершения приложения после записи предупреждения `StrictMode` в `LogCat` (как результат вызова метода `penaltyLog()`).

Включить `StrictMode` для потока достаточно один раз. Следовательно, если включить `StrictMode` в начале метода `onCreate()` главной активности, которая выполняется в главном потоке, `StrictMode` будет отслеживать все, что происходит в главном потоке. В зависимости от вида нарушений, которые планируется обнаруживать, может оказаться достаточно включения `StrictMode` внутри первой активности. Включить `StrictMode` в приложении можно также, расширив класс `Application` и добавив настройку `StrictMode` в метод `onCreate()` своего класса приложения. Все, что выполняется в потоке, может потенциально пользоваться `StrictMode`, но вызывать код настройки где-либо еще не понадобится — одного раза достаточно.

Помимо `ThreadPolicy`, в `StrictMode` имеется объект `VmPolicy`. Он позволяет выявлять утечки памяти, возникающие из-за того, что объект SQLite или любой объект `Closeable` финализируется без его явного закрытия. Как показано в листинге 11.2, объект `VmPolicy` создается с помощью похожего класса `Builder`. Одно отличие между `VmPolicy` и `ThreadPolicy` состоит в том, что объект `VmPolicy` не может выдавать предупреждение в виде диалогового окна.

Листинг 11.2. Установка объекта `VmPolicy` внутри `StrictMode`

```
StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
    .detectLeakedSqlLiteObjects()
    .penaltyLog()
    .penaltyDeath()
    .build());
```

Отключение `StrictMode`

Поскольку `StrictMode` устанавливается для потока, нарушения будут обнаруживаться даже при переходе потока управления от объекта к объекту. Когда возникает нарушение, связанное с кодом, который выполняется в главном потоке, то в выяснении причины, почему это произошло, поможет трассировка стека. Затем можно предпринять действия по решению проблемы, переместив этот код в собственный фоновый поток. Вы можете даже решить оставить все, как есть — это ваше право. Разумеется, в производственной версии приложения имеет смысл `StrictMode` отключить, чтобы не вызвать аварийное завершение приложения из-за предупреждения.

Отключить `StrictMode` в производственной версии приложения можно парой способов. Самый прямолинейный из них — удалить вызовы, но это может затруднить дальнейшую разработку приложения. Кроме того, всегда можно определить булевскую переменную уровня приложения и проверять ее значение перед вызовом кода `StrictMode`. Тогда установка этой переменной в `false` перед выпуском производственной версии обеспечит отключение `StrictMode`. Более элегантный способ предусматривает использование режима отладки приложения, как определено в файле

AndroidManifest.xml. В дескрипторе <application> внутри этого файла имеется атрибут android:debuggable. Чтобы отлаживать приложение, значение этого атрибута необходимо установить в true; объект ApplicationInfo получает набор флагов, из которого в коде можно проверить флаг, соответствующий включенному режиму отладки. В листинге 11.3 продемонстрирован такой прием в действии; когда приложение находится в режиме отладки, StrictMode включен (и выключен в противном случае).

Листинг 11.3. Установка `StrictMode` только для режима отладки

```
// Возврат, если приложение не находится в режиме отладки
ApplicationInfo appInfo = context.getApplicationInfo();
int appFlags = appInfo.flags;
if ((appFlags & ApplicationInfo.FLAG_DEBUGGABLE) != 0) {
   // Настройка режима StrictMode
}
```

Вспомните, что подключаемый модуль ADT установит атрибут android:debuggable в true при запуске версии приложения, находящейся на этапе разработки, в эмуляторе или на устройстве; это, в свою очередь, приведет к включению StrictMode в предыдущем коде. Когда приложение экспортируется для создания производственной версии, ADT установит атрибут android:debuggable в false.

Тем не менее, все это не работает в версиях, предшествующих Android 2.3. Для явного использования StrictMode потребуется развернуть среду Android 2.3 или последующей версии. В более старых версиях Android будут возникать ошибки, поскольку класс StrictMode в них не определен.

Использование `StrictMode` со старыми версиями Android

Чтобы использовать StrictMode со старыми версиями Android (до 2.3), можно прибегнуть к услугам рефлексии, вызывая методы StrictMode косвенно, когда они доступны, и предпринимая обходные действия в противном случае. В листинге 11.4 продемонстрирован простейший подход; здесь вызывается специальный метод, созданный как раз для работы со старыми версиями Android.

Листинг 11.4. Использование `StrictMode` с помощью рефлексии

```
try {
   Class sMode = Class.forName("android.os.StrictMode");
   Method enableDefaults = sMode.getMethod("enableDefaults");
   enableDefaults.invoke(null);
}
catch(Exception e) {
   // StrictMode на этом устройстве не поддерживается, пропуск его
   Log.v("StrictMode", "... not supported. Skipping...");
}
```

Приведенный код определяет, существует ли класс StrictMode, и если это так, вызывает метод enableDefaults() этого класса. Если же класс StrictMode не найден, инициируется блок catch с исключением ClassNotFoundException. В случае существования StrictMode никаких исключений возникать не должно, потому что enableDefaults() является одним из методов этого класса. Метод enableDefaults() настраивает StrictMode на обнаружение чего угодно с записью предупреждений о нарушениях в LogCat. Поскольку этот вызываемый метод класса StrictMode является статическим, в качестве его первого аргумента передается null.

Временами сообщения об абсолютно всех нарушениях не нужны. Вполне допустимо настраивать `StrictMode` на потоках, отличных от главного, и может понадобиться получать меньший объем предупреждений. Пусть, например, в отслеживаемом потоке допускаются операции чтения с диска. В таком случае можно либо не вызывать метод `detectDiskReads()` на объекте `Builder`, либо вызвать метод `detectAll()` и затем `permitDiskReads()` на `Builder`. Для каждой политики предусмотрен аналогичный метод разрешения (`permit`*XXX*`()`).

В если нужно использовать `StrictMode`, но приложение выполняется под управлением версии Android, предшествующей 2.3, то существует ли решение? Конечно, существует! Если класс `StrictMode` для приложения не доступен, при попытке обращения к нему возникнет ошибка `VerifyError`. Поместив `StrictMode` в оболочку класса и перехватывая упомянутую ошибку, класс `StrictMode` можно игнорировать в случае недоступности и работать с ним, если он есть. В листинге 11.5 приведен пример класса `StrictModeWrapper`, который можно добавить в свое приложение, а в листинге 11.6 показан код настройки `StrictMode` внутри приложения.

Листинг 11.5. Использование `StrictMode` в версиях Android, предшествующих 2.3

```
public class StrictModeWrapper {
    public static void init(Context context) {
        // Проверка, установлен ли атрибут android:debuggable в true.
        int appFlags = context.getApplicationInfo().flags;
        if ((appFlags & ApplicationInfo.FLAG_DEBUGGABLE) != 0) {
            StrictMode.setThreadPolicy(
                new StrictMode.ThreadPolicy.Builder()
                    .detectDiskReads()
                    .detectDiskWrites()
                    .detectNetwork()
                    .penaltyLog()
                    .build());
            StrictMode.setVmPolicy(
                new StrictMode.VmPolicy.Builder()
                    .detectLeakedSqlLiteObjects()
                    .penaltyLog()
                    .penaltyDeath()
                    .build());
        }
    }
}
```

Как видите, код выглядит подобно показанному ранее, и включает все, изученное до сих пор. И, наконец, чтобы настроить `StrictMode` в приложении, понадобится добавить код, показанный в листинге 11.6.

Листинг 11.6. Обращение к `StrictMode` в версиях Android, предшествующих 2.3

```
try {
    StrictModeWrapper.init(this);
}
catch(Throwable throwable) {
    Log.v("StrictMode", "... is not available. Punting...");
}
```

Обратите внимание, что код, приведенный в листинге 11.6, может быть расположен внутри локального контекста любого объекта, например, в методе `onCreate()` главной

активности. Этот код будет работать под управлением любого выпуска Android. Второй способ условного обращения к `StrictMode` обеспечивает больший контроль, поскольку проще вызвать все необходимые методы и пропустить средства, которые не нужны. Ранний способ предусматривал только использование `enableDefaults()`, но рефлексия для одного метода не является слишком сложной.

Пример приложения, использующего `StrictMode`

В качестве учебного упражнения, войдите в Eclipse и создайте копию одного из разработанных ранее приложений. Укажите для цели сборки версию 2.3 или выше, чтобы был доступен класс `StrictMode`. Однако установите `minSDKVersion` во что-либо, меньшее 2.3. Затем добавьте новый класс в папку `src`, используя код из листинга 11.5. Внутрь метода `onCreate()` активности, которая запускается первой, поместите код, приведенный в листинге 11.6; запустите полученное приложение в эмуляторе под управлением версии Android, предшествующей 2.3, и далее — под управлением Android 2.3 или одной из последующих версий. Когда класс `StrictMode` не доступен, в `LogCat` должны появиться сообщения, указывающие на это, тем не менее, приложение должно продолжить свое нормальное выполнение. Когда же класс `StrictMode` доступен, во время работы приложения в `LogCat` можно наблюдать нерегулярные сообщения о нарушениях. Если вы попробуете описанный подход на примере приложения, ориентированного на версию, предшествующую Android 2.3, скажем, NotePad, весьма вероятно появление сообщений о нарушении политик.

Ссылки

Ниже перечислены некоторые полезные ссылки на темы для дополнительного изучения.

- `http://developer.android.com/guide/developing/tools/index.html`. Документация для разработчиков по инструментам отладки в Android.
- `http://developer.android.com/guide/developing/devices/emulator.html#console`. Синтаксис и использование команд консоли эмулятора (Emulator Console). Это позволит применять интерфейс командной строки для моделирования событий, предназначенных приложению, которое выполняется в эмуляторе.
- `www.eclipse.org/mat/`. Проект Eclipse под названием MAT (Memory Analyzer — анализатор памяти). Этот подключаемый модуль можно использовать для чтения файлов HPROF, создаваемых средствами DDMS.

Резюме

Ниже перечислены темы, которые были рассмотрены в этой главе.

- Настройка среды Eclipse и физического устройства для отладки приложений.
- Перспектива Debug, позволяющая останавливать приложения для проверки значений переменных, а также пошагово выполнять код.
- Перспектива DDMS, которая содержит несколько инструментов для исследования потоков, памяти и обращений к памяти, а также для получения снимков экрана и генерации событий, отправляемых эмулятору.
- Сброс сервера `adb` из DDMS и из командной строки.
- Перспектива Hierarchy View, которая отображает структуру представлений выполняющегося приложения и включает показатели, помогающие настраивать и отлаживать приложение.

- Представление Traceview, которое показывает, какие методы вызываются во время выполнения приложения, а также статистические данные, которые помогают идентифицировать проблемные методы, требующие внимания.
- Команда `adb`, которую можно использовать для входа в устройство и просмотра его содержимого.
- Консоль эмулятора (Emulator Console), позволяющая взаимодействовать с эмулятором из командной строки. Возможности написания сценариев способствуют автоматизации выполнения рутинных действий.
- Специальный класс `StrictMode`, предназначенный для проверки, что приложение не выполняет в главном потоке не рекомендованные действия, такие как операции дискового ввода-вывода или обращения к сети.

Вопросы для самоконтроля

Ниже перечислены вопросы, ответы на которые помогут закрепить знания, полученные в этой главе.

1. Верно ли следующее: чтобы можно было отлаживать приложение, нужно явно установить атрибут `android:debuggable` дескриптора `<application>` в `true` внутри файла `AndroidManifest.xml`?
2. Какие четыре вещи можно делать с приложением внутри перспективы Debug в Eclipse?
3. Можно ли подключить к Eclipse более одного устройства и/или эмулятора одновременно? Если да, то где выбирается приложение, предназначенное для работы?
4. Какое средство DDMS используется для сбора статистических данных по текущим выделениям памяти в приложении?
5. Как определить, сколько потоков выполняется в приложении?
6. Как выяснить количество вызовов конкретного метода в приложении и общее время выполнения этого метода?
7. Где можно получить изображение экрана устройства?
8. Какая перспектива Eclipse используется для анализа структуры представлений в приложении?
9. Что обозначается тремя цветными кружками в этой перспективе? Означает ли желтый цвет наличие крупной проблемы? А красный?
10. Если вы видите желтый или красный кружок и желаете узнать, насколько проблемной является ситуация, то что потребуется предпринять для просмотра действительных числовых значений показателей?
11. Что понадобится сделать, когда нужно просмотреть профили методов, но видеть все методы целого приложения нежелательно?
12. Как создать командную оболочку Linux внутри запущенного эмулятора?
13. Можно ли это сделать на реальном устройстве? Если да, существуют ли какие-то ограничения на команды, выполняемые на реальном устройстве?
14. Как выяснить номер порта эмулятора, чтобы подключиться к нему с использованием консоли эмулятора?
15. Какие два главных аспекта проверяет класс `StrictMode`?