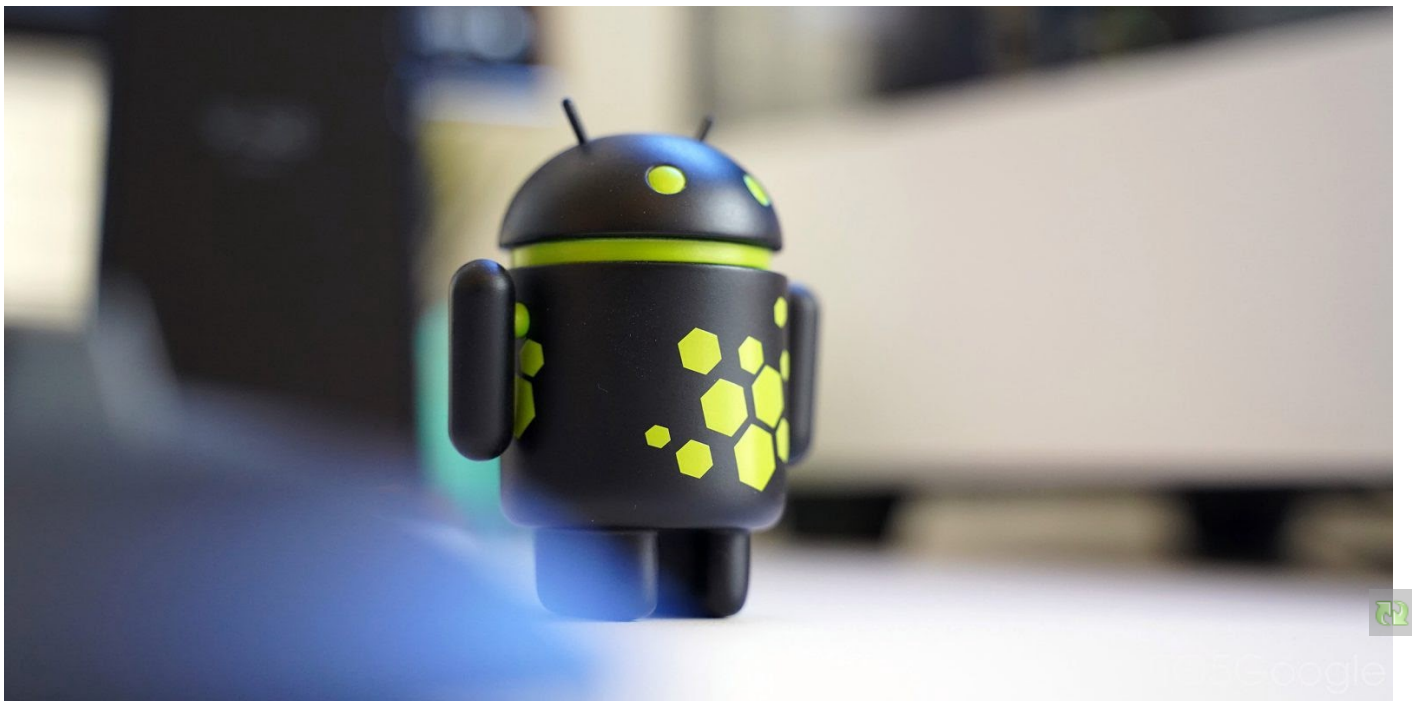
VitaliSergey вчера в 14:28

Android изнутри: сравнение Dalvik и ART

Java, Разработка мобильных приложений, Разработка под Android

Привет, Хабр! Около полугода назад я публиковал подробный «гайд» по JVM. Пост, в целом, зашел, а в комментариях спросили, не планируется ли “чего-то по андроиду”. Наконец, у меня дошли руки.



В этом посте поговорим о среде выполнения в Android. В частности, я постараюсь кратко, но емко изложить, чем отличается ART и Dalvik, и как со временем улучшились средства разработки в Android. Тема явно не новая, но, надеюсь, придется кстати тем, кто только начинает вникать. Кому интересно — добро пожаловать под кат.

Виртуальная машина

Сначала, давайте разберемся чем отличается JVM от DVM.

Java Virtual Machine — виртуальная машина, способная выполнять байт-код Java независимо от базовой платформы. Она опирается на принцип “Write once, run anywhere”. Байт-код Java может быть запущен на любой машине, способной поддерживать JVM.

Компилятор Java преобразует .java файлы в class-файлы (байт-код). Байт-код передается JVM, который компилирует его в машинный код для исполнения непосредственно на CPU.

Особенности JVM:

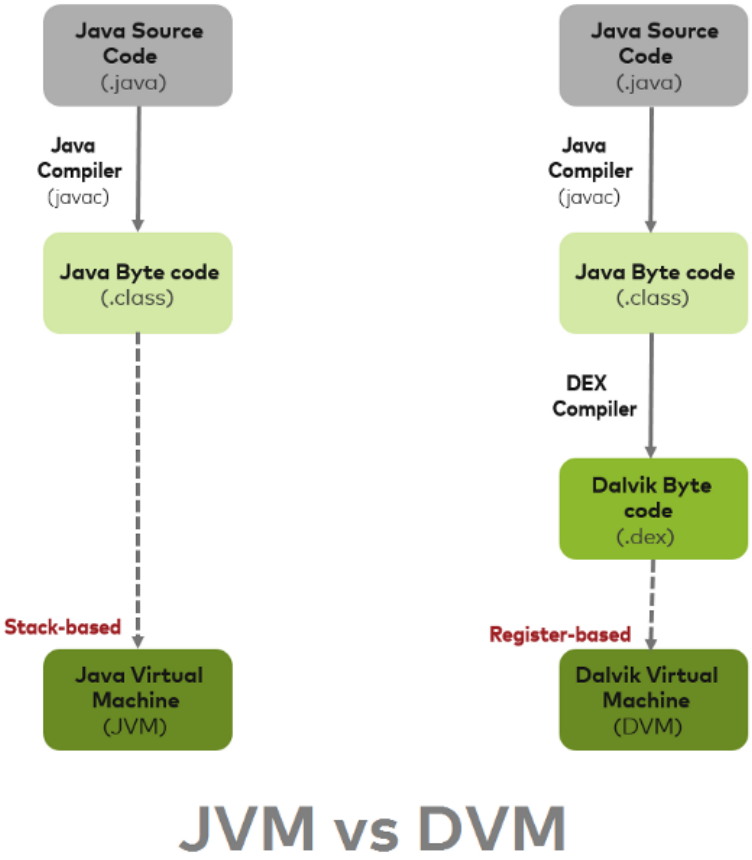
- Имеет стековую архитектуру: в качестве структуры данных, куда помещаются и хранятся методы, используется стек. Он работает по схеме LIFO или “Last in — First Out” или “Последним вошел, первым вышел”.
- Может запускать только class-файлы.
- Использует JIT-компилятор.

Dalvik Virtual Machine (DVM) — виртуальная Java машина, разработанная и написанная Дэном Борнштейном (англ. Dan Bornstein) и другими, как часть мобильной платформы Android.

Можно сказать, что Dalvik — это среда для выполнения компонентов операционной системы Android и пользовательских приложений. Каждый процесс выполняется в своём, изолированном адресном пространстве. Когда пользователь запускает приложение (либо операционная система запускает один из своих компонентов), ядро виртуальной машины Dalvik (Zygote Dalvik VM) создает отдельный, защищенный процесс в общей памяти, в котором непосредственно разворачивается VM, как среда для запуска приложения. Другими словами, изнутри Android выглядит как набор виртуальных машин Dalvik, в каждой из которых исполняется приложение.

Особенности DVM:

- Использует архитектуру на основе регистров: структура данных, куда помещаются методы, основана на регистрах процессора. За счет отсутствия операций POP и PUSH, команды в регистровой виртуальной машине выполняются быстрее аналогичных команд стековой виртуальной машины.
- Исполняет байт-код собственного формата: Android dexer (о нем поговорим ниже) преобразует class-файлы в формат .dex, оптимизированные для выполнения на Dalvik VM. В отличие от class-файла, dex-файл содержит сразу несколько классов.



JVM vs DVM

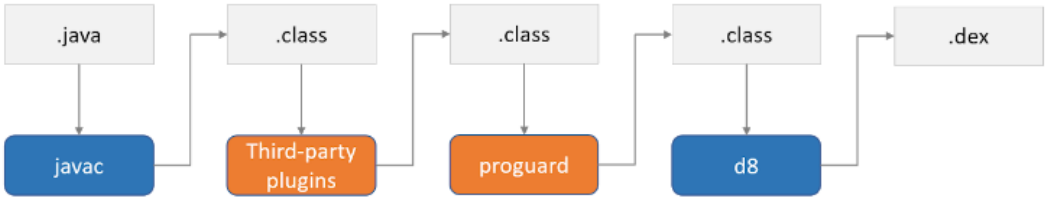
Подробно об архитектуре DVM можно почитать тут.

Android Dexer

Разработчики Android знают, что процесс преобразования Java байткода в .dex байткод для Android Runtime является ключевым шагом в создании APK. Компилятор dex в основном работает “под капотом” в повседневной разработке приложений, но он напрямую влияет на время сборки приложения, на размер файла .dex и производительность во время выполнения.

Как уже упоминалось, сам dex-файл содержит сразу несколько классов. Повторяющиеся строки и другие константы, используемые в нескольких файлах классов, включаются только для экономии места. Байт-код Java также преобразуется в альтернативный набор команд, используемый DVM. Несжатый dex-файл обычно на несколько процентов меньше по размеру, чем сжатый архив Java (JAR), полученный из тех же файлов .class.

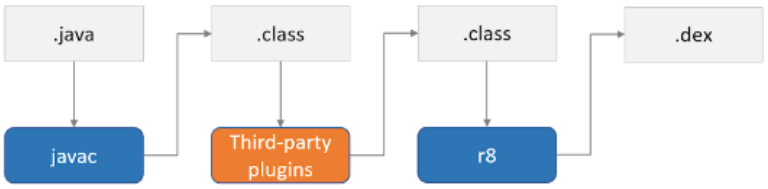
Изначально, class-файлы преобразовывались в dex-файлы с помощью встроенного DX-компилятора. Но начиная с Android Studio 3.1 и далее, компилятором по умолчанию стал D8. По сравнению с DX-компилятором, D8 компилирует быстрее и выводит dex-файлы меньшие по размеру, при этом обеспечивая более высокую производительность приложения во время исполнения. Полученный таким образом байт-код dex подвергается минификации с помощью open-source утилиты ProGuard. В итоге, мы получаем тот же dex-файл, но только меньше. Далее этот dex-файл используется для сборки apk и, наконец, для развертывания на устройстве Android.



Сборка приложения с использованием D8

Но следом за D8 в 2018 году пришел R8, который, по сути, является тем же D8, только с дополнениями.

При работе с Android Studio 3.4 и Android Gradle 3.4.0 plugin или выше, Proguard больше не используется для оптимизации кода во время компиляции. Вместо этого плагин работает по умолчанию с R8, который сам выполняет Code shrinking, Optimisation и Obfuscation. Хотя R8 предлагает только подмножество функций, предоставляемых Proguard, он позволяет совершить процесс преобразования Java байт-кода в dex-байт-код единоразово, что еще больше сокращает время сборки.



Сборка приложения с использованием R8

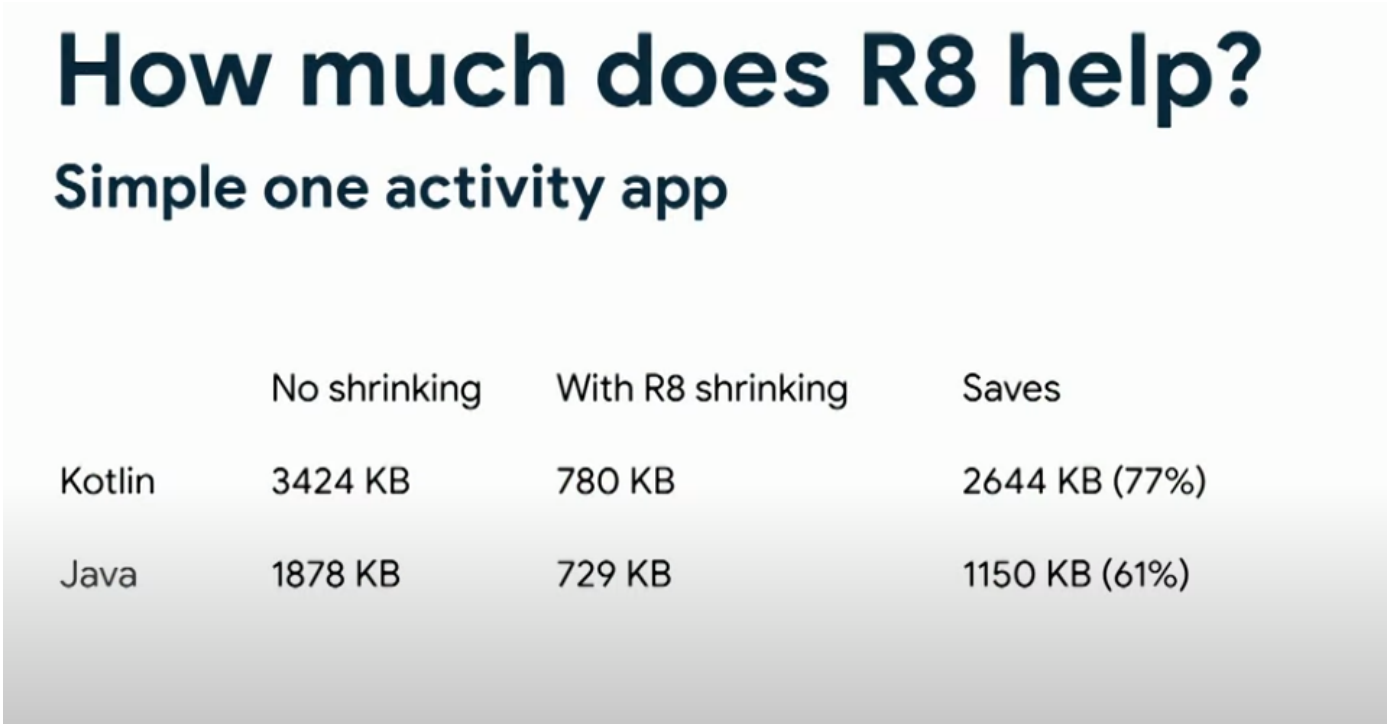
R8 и сокращение кода

Как правило, приложения используют сторонние библиотеки, такие как Jetpack, Gson, Google Play Services. Когда мы используем одну из этих библиотек, часто в приложении используется только малая часть каждой отдельной библиотеки. Без Code shrinking, весь код библиотеки сохраняется в вашем приложении.

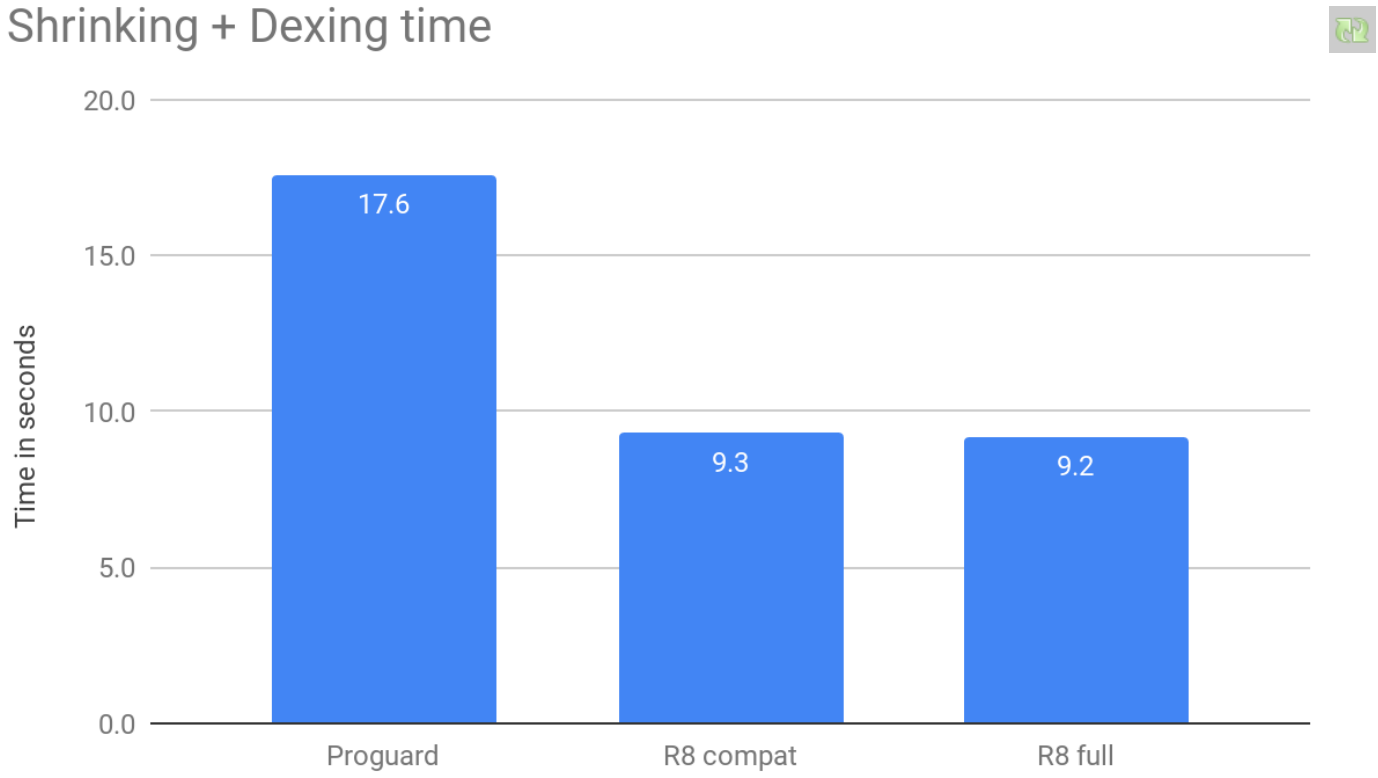
Бывает так, что для улучшения читаемости и удобства поддержки приложения разработчики используют подобный код. Например, могут быть использованы значимые имена переменных и шаблон проектирования для того, чтобы другим было удобнее разобраться в коде. Но шаблоны, как правило, приводят к бОльшему объему кода, чем это необходимо.

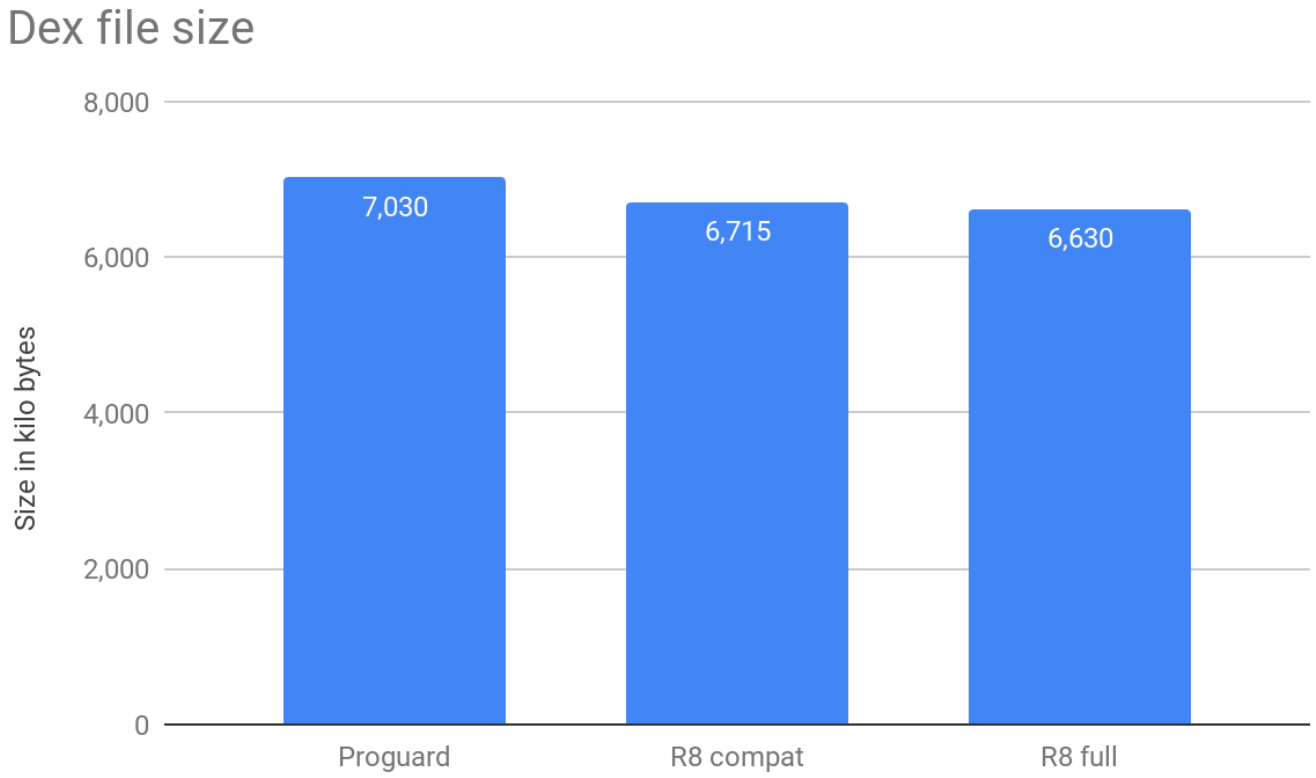
В этом случае R8 приходит на помощь. Он позволяет существенно уменьшить размер приложения, оптимизируя размер даже того кода, который действительно используется приложением.

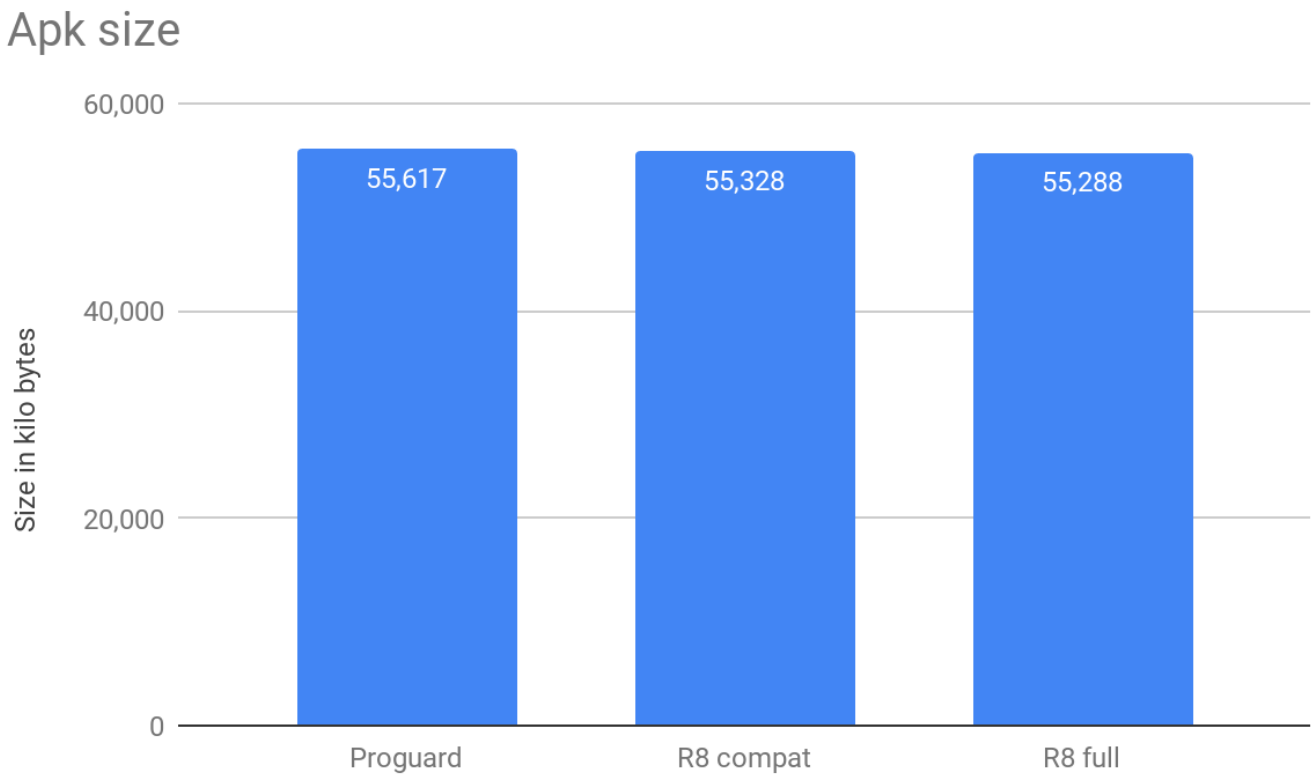
В качестве примера, ниже преведены цифры из доклада Shrinking Your App with R8, который был представлен на Android Dev Summit '19:



А вот так выглядело сравнение эффективности R8 на этапе выпуска бета-версии (взято из источника Android Developers Blog):







Детальнее можно ознакомиться в оф документации и докладе.

ART vs DVM в Android

DVM была спроектирована именно для мобильных устройств и использовалась как виртуальная

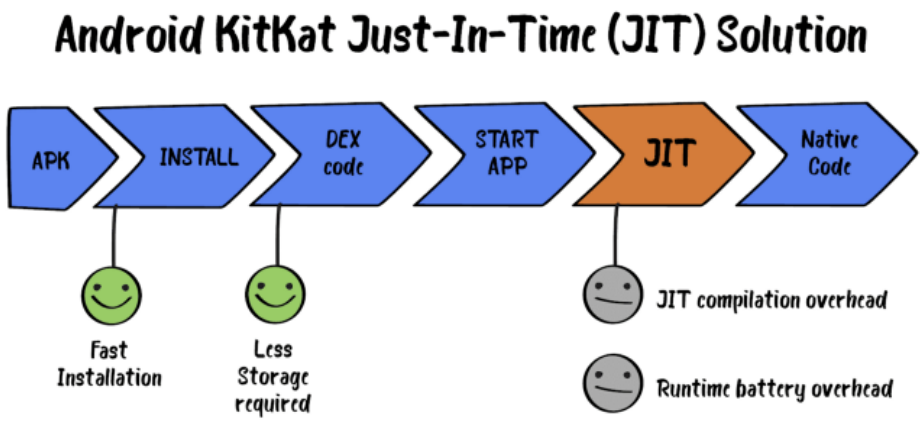
машина для запуска андроид приложений вплоть до Android 4.4 Kitkat.

Начиная с этой версии, ART был представлен как среда выполнения, а в Android 5.0 (Lollipop) ART полностью заменил Dalvik.

Основное явное отличие ART от DVM состоит в том, что ART использует AOT компиляцию, а DVM — JIT компиляцию. Не так давно ART начал использовать гибрид AOT и JIT. Далее разберем это чуть подробнее.

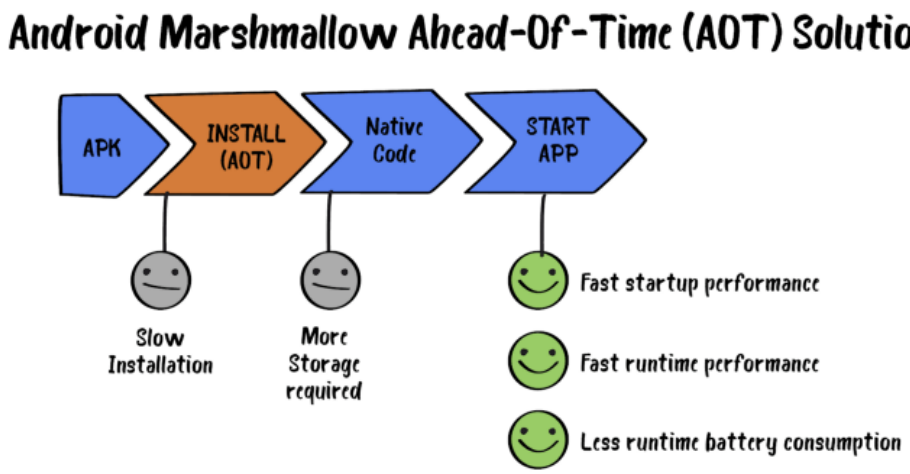
DVM

- Использует JIT компиляцию: всякий раз при запуске приложения,
- компилируется та часть кода, которая необходима для выполнения приложения. Остальная часть кода компилируется динамически. Это замедляет запуск и работу приложений, но уменьшает время установки.
- Ускоряет загрузку устройства, поскольку кеш приложения создается во время выполнения.
- Приложения, работающие на DVM, требуют меньше памяти, чем те, которые работают на ART.
- Уменьшает резерв батареи, увеличивая нагрузку на CPU.
- Dalvik является “устаревшим” и не используется на андроид версиях выше 4.4.

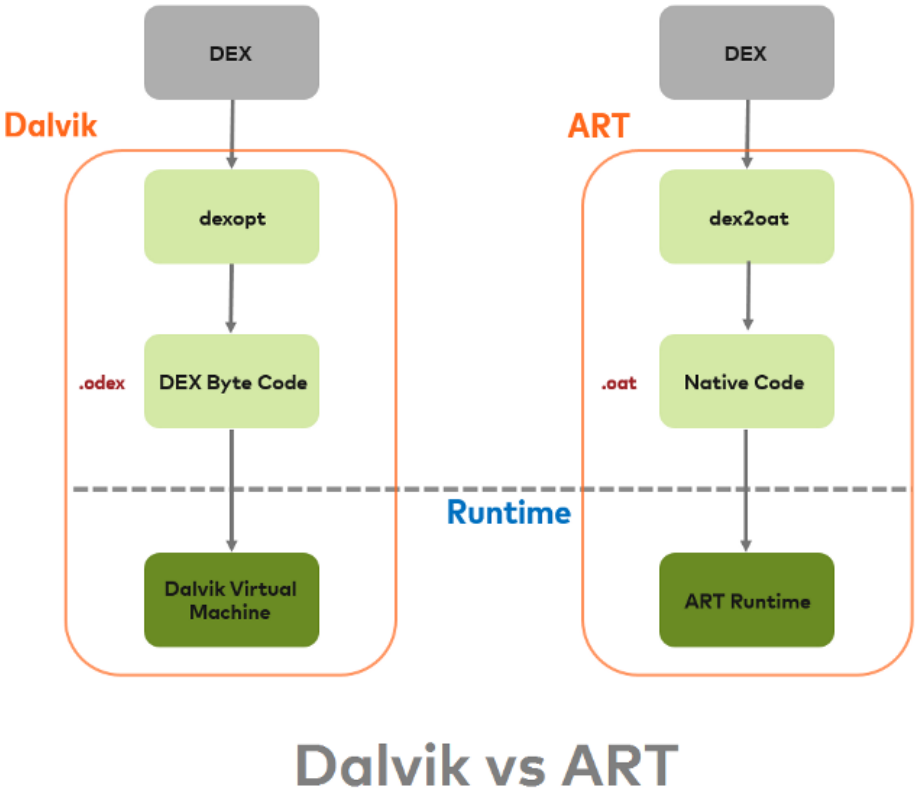


ART

- Использует AOT компиляцию, то есть компилирует весь код во время установки приложения. Это ускоряет запуск и работу приложений, но требует большего времени установки.
- Замедляет загрузку устройства, так как кеш создается во время первой загрузки.
- Ввиду использования подхода AOT компиляции, требует больше памяти в сравнении с приложениями на DVM.
- Увеличивает резерв батареи, сокращая работу процессора из-за отсутствия компиляции при выполнении приложений.
- Улучшенная Garbage Collection или сборка мусора. Во времена использования Dalvik, сборщики мусора должны были осуществить 2 прохода по куче (heap), что и приводило к плохому UX. В случае с ART, такой ситуации нет: он чистит кучу один раз для консолидации памяти.



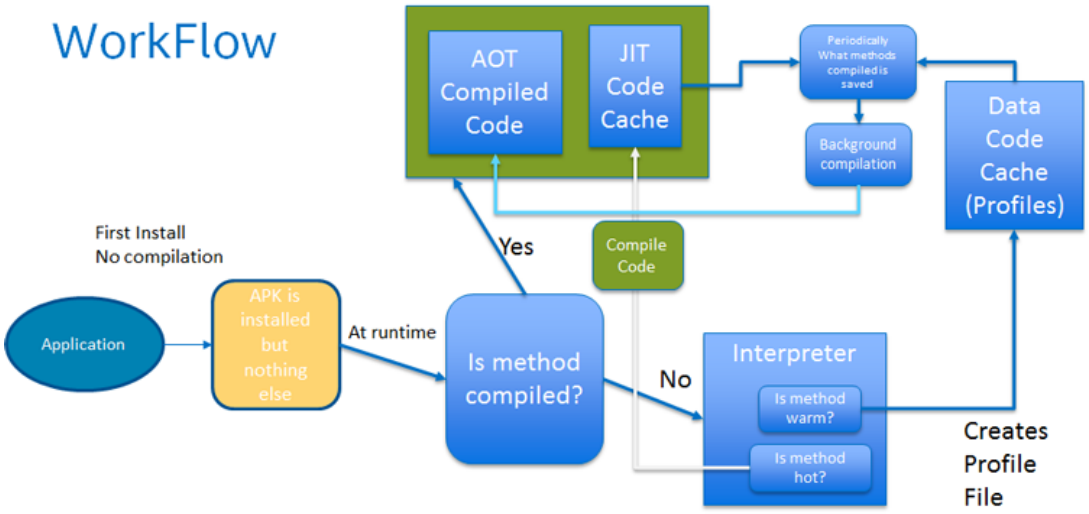
И небольшая схема Dalvik vs ART:



JIT + AOT в ART

Среда выполнения Android (ART), начиная с Android 7, включает компилятор JIT с профилированием кода. JIT-компилятор дополняет AOT компилятор и повышает производительность во время выполнения, экономит место на диске и ускоряет обновления приложений и системы.

Происходит это по следующей схеме:



Вместо того, чтобы запускать AOT-компиляцию каждого приложения на этапе установки, он запускает приложение под управлением виртуальной машины, используя JIT-компилятор (почти так же, как в Android < 5.0), но следит за тем, какие участки кода приложения выполняются чаще всего. Затем эта информация используется для AOT-компиляции данных участков кода. Последняя операция выполняется только во время бездействия смартфона, находящегося на зарядке.

Говоря простыми словами, теперь два совершенно разных подхода работают сообща, что дает свои плюсы:

- более эффективная компиляция — при запуске приложения в реальном времени компилятор имеет возможность узнать о его работе гораздо больше, чем выполняя статический анализ, и, как следствие, применяются более подходящие методы оптимизации для каждой ситуации;
- сохранение оперативной и постоянной памяти — байт-код компактнее машинного кода, а если выполнять AOT-компиляцию только отдельных участков приложения и не выполнять компиляцию приложений, которыми юзер не пользуется, можно существенно сэкономить пространство NAND-памяти;
- резкое увеличение скорости установки и первой загрузки после обновления системы — нет AOT-компиляции, нет задержки.

О релазации JIT компилятора в ART подробнее тут.

Заключение

В этой статье я постарался разобрать основные отличия Dalvik от ART, и в целом взглянуть на то, как с течением времени Android улучшал инструменты для разработки.

ART все еще находится в стадии разработки: добавляются новые фичи, чтобы улучшить опыт как для пользователей, так и для разработчиков.

Если было полезно — дайте знать в комментах.

Теги: android, dalvik, android runtime

Хабы: Java, Разработка мобильных приложений, Разработка под Android

Виталий Сергей @VitaliSergey
Архитектор решений в IntexSoft

Комментарии 8

Отслеживать новые в ☐ почте ☐ трекере

mixsture вчера в 14:37 0

Довольно странный они применили подход: сократили что-то на однократном действии (установке), но зато ухудшили на многократном действии (запуск). Да еще и от запуска пользователь ждет оперативность и экономный расход батареи, а вот от установки такого не ждут — все равно, дольше ли она на минуту или нет.
Я бы сказал, это пример, как не надо делать :)

Ответить

eastig вчера в 16:17 +1

В статье не говорится, почему вернулись к схеме Interpreter+JIT+AOT. Основная проблема с AOT была, что при любом изменении системы приходилось перекомпилировать все библиотеки и приложения, что могло занимать много времени. При переходе на Interpreter+JIT+AOT одним из критериев был не ухудшение времени запуска. После перехода выяснилось, что не весь код приложений нужно компилировать.

В ART применяется многократная AOT перекомпиляция кода в зависимости от изменения профиля исполнения. Кроме того Google Play может предоставлять некий профиль исполнения приложения, который учитывается при установке приложения.

В Android 10 и 11 случился APEX. Теперь ART может обновлять через Google Play, если производитель телефона поддерживает эту функцию.

Ответить

alexkmbkdr1 вчера в 18:52 0

А какое например изменение системы требовало перекомпиляции всех библиотек и приложений? Этот момент не понятен.

Ответить

eastig вчера в 19:15 0

Например исправили security bug в Java core или framework библиотеках. Насколько я помню любое изменение в системных компонентах требовало перекомпиляции всех приложений, но могу и ошибаться.

Ответить

qw1 сегодня в 09:00 0

Не ошибаетесь. Изменение любого jar-файла в директории /system/framework, и при следующей загрузке жди минут 10 окошко «Android is optimizing» (актуально во времена Android 5/6, сейчас уже быстрее проходит)

Ответить

eastig вчера в 16:22 +1

Кстати, в ART JIT-компилятор и AOT-компилятор — это один и тот же компилятор, который запускают либо в режиме JIT или AOT. AOT-компилятор можно запустить через утилиту dex2oat.

Ответить

house2008 вчера в 18:26 0

А почему нельзя компилировать прям на сервере и просто отдавать нужный бинарник под нужную архитектуру? Ведь AOT как раз всё равно собирал бинарник во время установки приложения. Ведь Apple так и делает, в стор заливается fat binary и потом просто раздают нужную архитектуру в зависимости от клиента, а на клиенте уже подтягиваются системные зависимости во время запуска приложения.

Ответить

eastig вчера в 19:32 0

Это затруднительно сделать в силу большого разнообразия версий Android и производителей телефонов.

Ответить

Написать комментарий

Предпросмотр Отправить Markdown (?)

САМОЕ ЧИТАЕМОЕ

Сутки | Неделя | Месяц

Что помешало экипажу Crew Dragon выйти из корабля?

+176 89,2k 50 120

Памятка для пострадавшего от слезоточивого газа/перцового баллона

+110 30,9k 194 86

Microsoft Defender начал помечать файл hosts как зловредный, если там блокируется сбор телеметрии Windows 10

+41 17,2k 25 88

Рынок соискателя или работодателя VS возрастная дискриминация

+50 17,7k 51 113

Опрос про big data в российских IT

Опрос