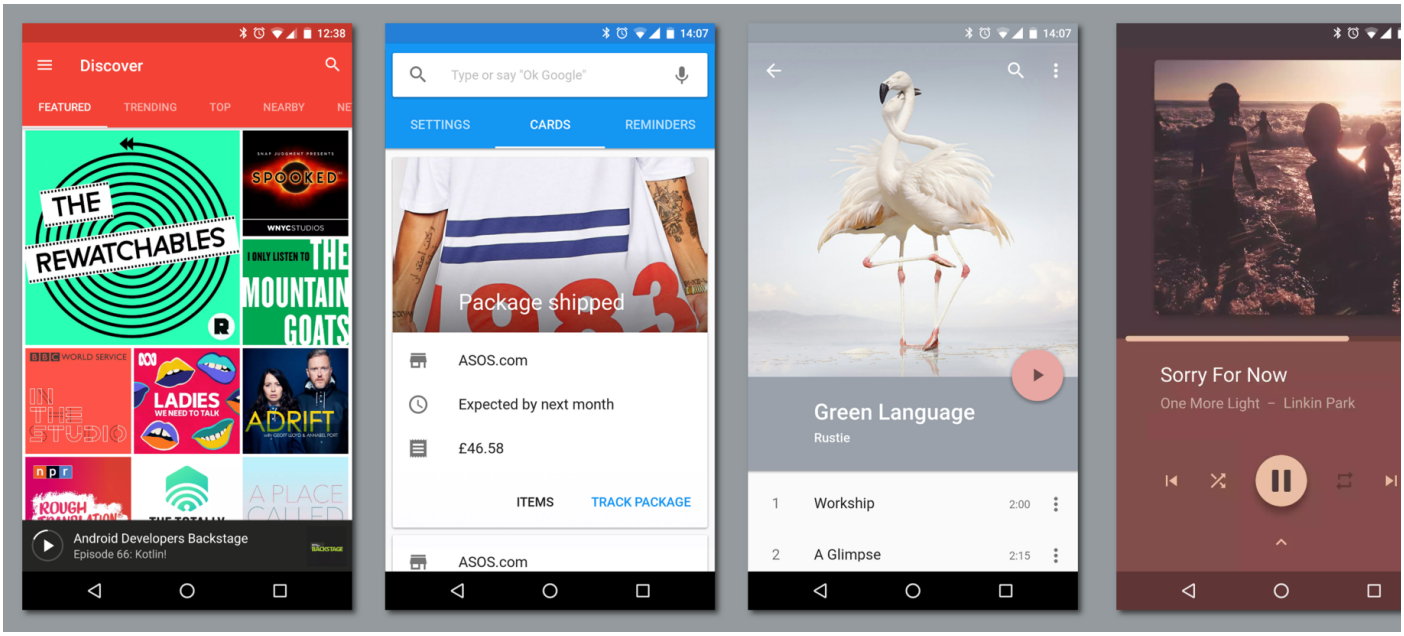
401,35
Рейтинг

Solar Security
Безопасность по имени Солнце

bugaevc 26 сентября в 14:07 Разработка

Как работает Android, часть 3

Разработка под Android, Блог компании Solar Security



В этой статье я расскажу о компонентах, из которых состоят приложения под Android, и об идеях, которые стоят за этой архитектур

Статьи серии:

- Как работает Android, часть 1
- Как работает Android, часть 2
- Как работает Android, часть 3
- ...

Web vs desktop

Если задуматься об отличиях современных веб-приложений от «обычных» десктопных приложений, можно — среди недостатков – выделить несколько преимуществ веба:

- Веб-приложения переносимы между архитектурами и платформами, *как и Java*.
- Веб-приложения не требуют установки и всегда обновлены, *как и Android Instant Apps*.

Кроме того, веб-приложения существуют в виде страниц, которые могут ссылаться друг на друга — как в рамках одного сайта, так между сайтами. При этом страница на одном сайте не обязана ограничиваться ссылкой только на главную страницу другого, она м ссылаться на конкретную страницу внутри другого сайта (это называется deep linking). Ссылаясь друг на друга, отдельные сайты объединяются в общую сеть, веб.

Несколько копий одной страницы — например, несколько профилей в социальной сети — могут быть одновременно открыты в нескольких вкладках браузера. Интерфейс браузера рассчитан на переключение между одновременными сессиями (вкладками), а

между отдельными сайтами — в рамках одной вкладки вы можете перемещаться по ссылкам (и вперёд-назад по истории) между разными страницами разных сайтов.

Всё это противопоставляется «десктопу», где каждое приложение работает отдельно и часто независимо от других — и в этом пла как устроены приложения в Android, гораздо ближе к вебу, чем к «традиционным» приложениям.

Activities & intents

Основной вид компонентов приложений под Android — это **activity**. Activity — это один «экран» приложения. Activity можно сравни страницей в вебе и с окном приложения в традиционном оконном интерфейсе.

▸ Про окна

Например, в приложении для электронной почты (email client) могут быть такие activity, как Inbox Activity (список входящих писем), Activity (чтение одного письма), Compose Activity (написание письма) и Settings Activity (настройки).

Как и страницы одного сайта, activity одного приложения могут запускаться как друг из друга, так и независимо друг от друга (дру приложениями). Если в вебе на другую страницу обращаются по URL (ссылке), то в Android activity запускаются через intent'ы.

Intent — это сообщение, которое указывает системе, что нужно «сделать» (например, открыть данный URL, написать письмо на да адрес, позвонить на данный номер телефона или сделать фотографию).

Приложение может создать такой intent и передать его системе, а система решает, какая activity (или другой компонент) будет его выполнять (handle). Эта activity запускается системой (в существующем процессе приложения или в новом, если он ещё не запуще передаётся этот intent, и она его выполняет.

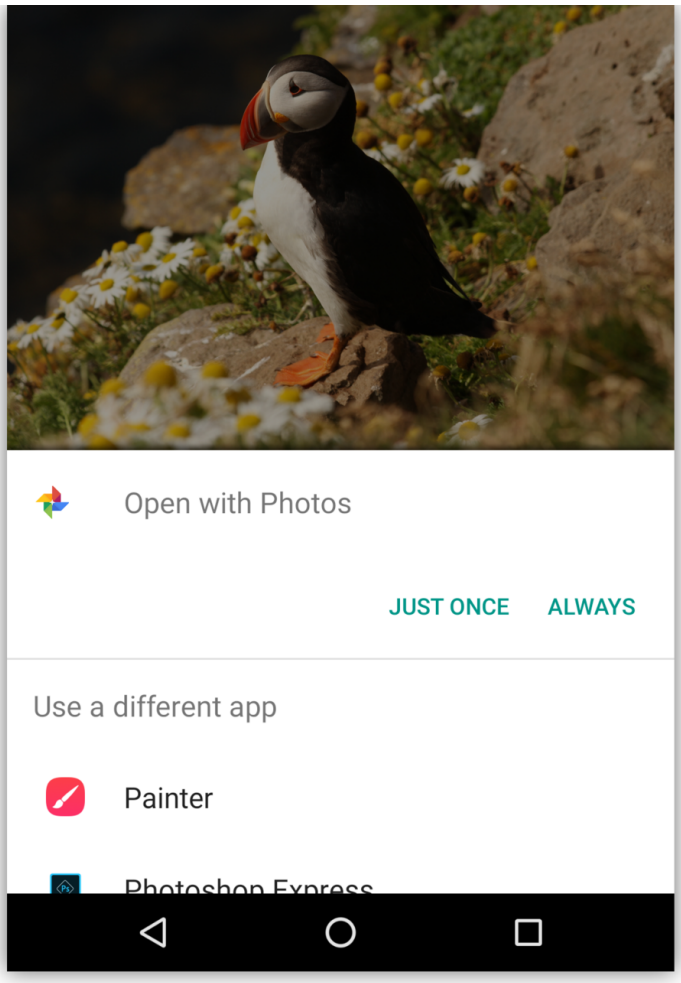
Стандартный способ создавать intent'ы — через соответствующий класс в Android Framework. Для работы с activity и intent'ами из командной строки в Android есть команда `am` — обёртка над стандартным классом Activity Manager:

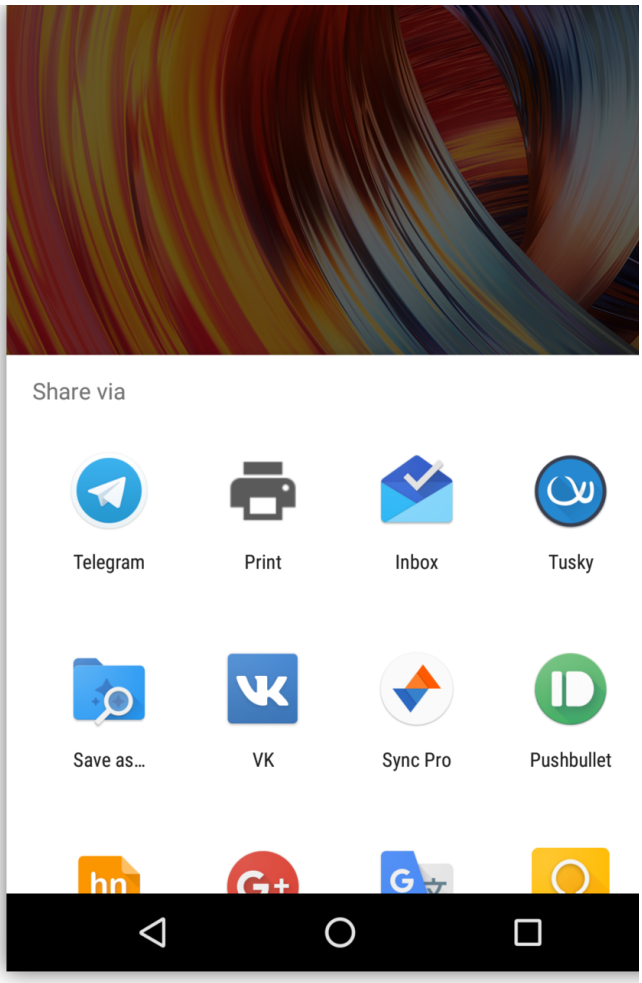
```
# передаём -a ACTION -d DATA
# открыть сайт
$ am start -a android.intent.action.VIEW -d http://example.com
# позвонить по телефону
$ am start -a android.intent.action.CALL -d tel:+7-916-271-05-83
```

Intent'ы могут быть *явными* (explicit) и *неявными* (implicit). Явный intent указывает идентификатор конкретного компонента, который нужно запустить — чаще всего это используется, чтобы запустить из одной activity другую внутри одного приложения (при этом int может даже не содержать другой полезной информации).

Неявный intent обязательно должен указывать действие, которое нужно сделать. Каждая activity (и другие компоненты) указывают манифесте приложения, какие intent'ы они готовы обрабатывать (например, `ACTION_VIEW` для ссылок с доменом `https://example.com` Система выбирает подходящий компонент среди установленных и запускает его.

Если в системе есть несколько activity, которые готовы обработать intent, пользователю будет предоставлен выбор. Обычно это случается, когда установлено несколько аналогичных приложений, например несколько браузеров или фоторедакторов. Кроме тог приложение может явно попросить систему показать диалог выбора (на самом деле при этом переданный intent оборачивается в н intent с `ACTION_CHOOSER`) — это обычно используется для создания красивого диалога Share:





Кроме того, activity может вернуть результат в вызвавшую её activity. Например, activity в приложении-камере, которая умеет обрабатывать intent «сделать фотографию» (`ACTION_IMAGE_CAPTURE`) возвращает сделанную фотографию в ту activity, которая создал intent.

При этом приложению, содержащему исходную activity, не нужно разрешение на доступ к камере.

Таким образом, правильный способ приложению под Android сделать фотографию — это не потребовать разрешения на доступ к к и использовать Camera API, а создать нужный intent и позволить системному приложению-камере сделать фото. Аналогично, вмест использования разрешения `READ_EXTERNAL_STORAGE` и прямого доступа к файлам пользователя стоит дать пользователю возможност выбрать файл в системном файловом менеджере (тогда исходному приложению будет разрешён доступ именно к этому файлу).

> A unique aspect of the Android system design is that any app can start another app's component. For example, if you want the user to capture a photo with the device camera, there's probably another app that does that and your app can use it instead of developing an activity to capture a photo yourself. You don't need to incorporate or even link to the code from the camera app. Instead, you can simply start the activity in the camera app that captures a photo. When complete, the photo is even returned to your app so you can use it. To t user, it seems as if the camera is actually a part of your app.

При этом «системное» приложение — не обязательно то, которое было предустановлено производителем (или автором сборки An Все установленные приложения, которые умеют обрабатывать данный intent, в этом смысле равны между собой. Пользователь мож выбрать любое из них в качестве приложения по умолчанию для таких intent'ов, а может выбирать нужное каждый раз. Выбранное приложение становится «системным» в том смысле, что пользователь выбрал, чтобы именно оно выполняло все задачи (то есть int такого типа, возникающие в системе.

Само разрешение на доступ к камере нужно только тем приложениям, которые реализуют свой интерфейс камеры — например, собственно приложения-камеры, приложения для видеозвонков или дополненной реальности. Наоборот, обыкновенному мессенд доступ к камере «чтобы можно было фото отправлять» *не нужен*, как не нужен и доступ к совершению звонков приложению *крупн банка*.

▾ Про лончер

Этой логике подчиняются даже такие «части системы», как, например, домашний экран (лончер, launcher). Лончер — это специал приложение со своими activity (которые используют специальные флаги вроде `excludeFromRecents` и `launchMode="singleTask"`).

Нажатие кнопки «домой» создаёт intent категории `HOME`, который дальше проходит через обычный механизм обработки intent'ов - том числе, если в системе установлено несколько лончеров и ни один не выбран в качестве лончера по умолчанию, система отобразит диалог выбора.

«Запуск» приложения из лончера тоже происходит через intent. Лончер создаёт явный intent категории `LAUNCHER`, который «обрабатывается» запуском основной activity приложения.

Приложение может иметь *несколько* activity, которые поддерживают такой intent, и отображаться в лончере несколько раз (при эт может понадобиться указать им разную `taskAffinity`). Или не иметь ни одной и не отображаться в лончере вообще (но по-прежне отображаться в полном списке установленных приложений в настройках). «Обычные» приложения так делают довольно редко; са известный пример такого поведения — Google Play Services.

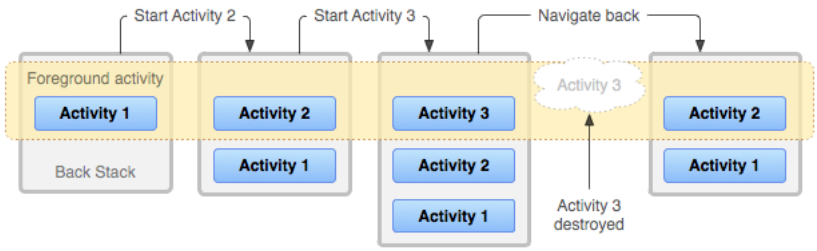
Многие операционные системы делятся на собственно операционную систему и приложения, установленные поверх, ничего друг о друге не знающие и не умеющие взаимодействовать. Система компонентов и intent'ов Android позволяет приложениям, *по-прежне абсолютно ничего друг о друге не зная*, составлять для пользователя один интегрированный системный user experience — установленные приложения *реализуют части* одной большой системы, они составляют из себя систему. И это, с одной стороны, происходит прозрачно для пользователя, с другой — представляет неограниченные возможности для кастомизации.

По-моему, это очень красиво.

Tasks & back stack

Как я уже говорил, в браузере пользователь может переключаться не между сайтами, а между вкладками, история каждой из кото может содержать много страниц разных сайтов. Аналогично, в Android пользователь может переключаться между *задачами* (tasks) которые отображаются в виде карточек на recents screen. Каждая задача представляет собой back stack — несколько activity, «наложенных» друг на друга.

Когда одна activity запускает другую, новая activity помещается в стек поверх старой. Когда верхняя activity в стеке завершается — например, когда пользователь нажимает системную кнопку «назад» — предыдущая activity в стеке снова отображается на экране.



Каждый стек может включать в себя activity из разных приложений, и несколько копий одной activity могут быть одновременно отк рамках разных задач или даже внутри одного стека.

При запуске новой activity могут быть указаны специальные флаги, такие как `singleTop`, `singleTask`, `singleInstance` и `CLEAR_TOP`, кото модифицируют этот механизм. Например, приложения-браузеры обычно разрешают запуск только одной копии своей основной ac и для переключения между открытыми страницами реализуют собственную систему вкладок. С другой стороны, Custom Tabs — пр activity в браузере (чаще всего Chrome), которая ведёт себя почти «как обычно», то есть показывает только одну страницу, но позв одновременно открывать несколько своих копий.

App lifecycle

Одно из основных ограничений встраиваемых и мобильных устройств — небольшое количество оперативной памяти (RAM). Если современные флагманские устройства уже оснащаются несколькими *гигабайтами* оперативной памяти, то в первом смартфоне на Android, HTC Dream (он же T-Mobile G1), вышедшем в сентябре 2008 года, её было всего 192 мегабайта.

Проблема ограниченной памяти дополнительно осложняется тем, что в мобильных устройствах, в отличие от «обычных» компьюте не используются swap-разделы (и swap-файлы) — в том числе и из-за низкой (по сравнению с SSD и HDD) скорости доступа к SD-к и встроенной флеш-памяти, где они могли бы размещаться. Начиная с версии 4.4 KitKat, Android использует zRAM swap, то есть эффективно сжимает малоиспользуемые участки памяти. Тем не менее, проблема ограниченной памяти остаётся.

Если все процессы представляют собой для системы чёрный ящик, лучшая из возможных стратегия поведения в случае нехватки свободной памяти — принудительно завершать («убивать») какие-то процессы, что и делает Linux Out Of Memory (OOM) Killer. Но Android *знает*, что происходит в системе, ему известно, какие приложения и какие их компоненты запущены, что позволяет реализ гораздо более «умную» схему освобождения памяти.

Во-первых, когда свободная память заканчивается, Android явно просит приложения освободить ненужную память (например, сбр кэш), вызывая методы `onTrimMemory`/`onLowMemory`. Во-вторых, Android может эффективно проводить сборку мусора в фоновых приложениях, освобождая память, которую они больше не используют (на уровне Java), при этом не замедляя работу текущего приложения.

Но основной механизм освобождения памяти в Android — это **завершение наименее используемых приложений**. Система автоматически выбирает приложения, наименее важные для пользователя (например, те, из которых пользователь давно ушёл), да компонентам шанс дополнительно освободить ресурсы, вызывая такие методы, как `onDestroy`, и завершает их, полностью освобож, используемую ими память и ресурсы.

Если пользователь возвращается в activity приложения, завершённого системой из-за нехватки памяти, эта activity запускается сно При этом **перезапуск происходит прозрачно для пользователя**, поскольку activity сохраняет своё состояние при завершении (`onSaveInstanceState`) и восстанавливает его при последующем запуске. Реализованные в Android Framework виджеты используют э механизм, чтобы автоматически сохранить состояние интерфейса (UI) при перезапуске — с точностью до введённого в EditText тек положения курсора, позиции прокрутки (scroll) и т.д. Разработчик приложения может дополнительно реализовать сохранение и восстановление каких-то ещё данных, специфичных для этого приложения.

Подчеркну, что Android может перезапускать приложения не полностью, а *покомпонентно*, оставляя неиспользуемые части завершёнными — например, из двух копий одной activity одна может быть перезапущена, а другая остаться завершённой.

С точки зрения пользователя этот механизм похож на использование swap: в обоих случаях при возвращении в *выгруженную* часть приложения приходится немного подождать, пока она загружается снова — в одном случае, с диска, в другом — пересоздаётся по сохранённому состоянию.

Именно этот механизм автоматического перезапуска и восстановления состояния создаёт у пользователя ощущение, что приложе «запущены всегда», избавляя его от необходимости явно запускать и закрывать приложения и сохранять введённые в них данные.

Services

Приложениям может потребоваться выполнять действия, не связанные напрямую ни с какой activity, в том числе, продолжать дела фоне, когда все activity этого приложения завершены. Например, приложение может скачивать из сети большой файл, обрабатыва фотографии, воспроизводить музыку, синхронизировать данные или просто поддерживать TCP-соединение с сервером для получе уведомлений.

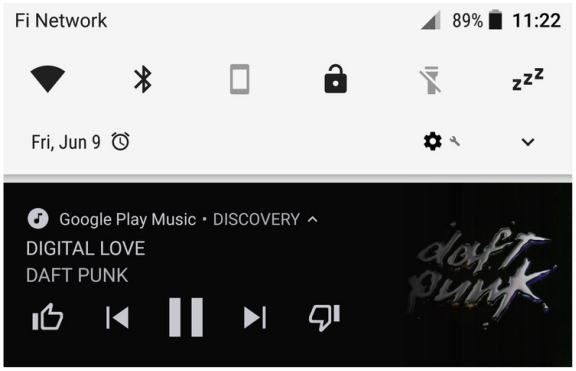
Такую функциональность нельзя реализовывать, просто запуская отдельный поток — это было бы для системы чёрным ящиком; в т числе, процесс был бы завершён при завершении всех activity, независимо от состояния таких фоновых операций. Вместо этого Ar предлагает использовать ещё один вид компонентов — **сервис**.

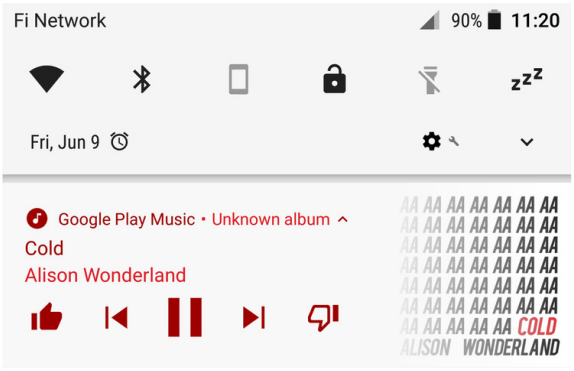
Сервис нужен, чтобы сообщить системе, что в процессе приложения выполняются действия, которые не являются частью activity э приложения. Сам по себе сервис не означает создание отдельного потока или процесса — его точки входа (entry points) запускают основном потоке приложения. Обычно реализация сервиса запускает дополнительные потоки и управляет ими самостоятельно.

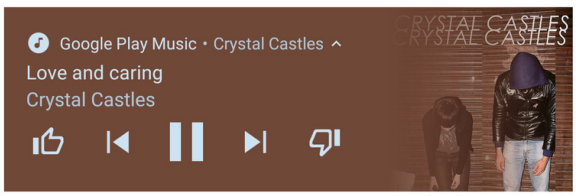
Сервисы во многом похожи на activity: они тоже запускаются с помощью intent'ов и могут быть завершены системой при нехватке памяти.

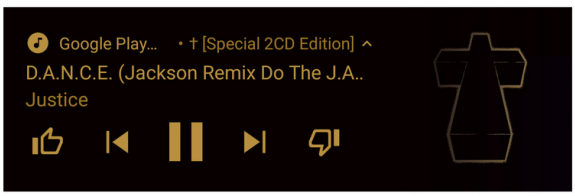
Запущенные сервисы могут быть в трёх состояниях:

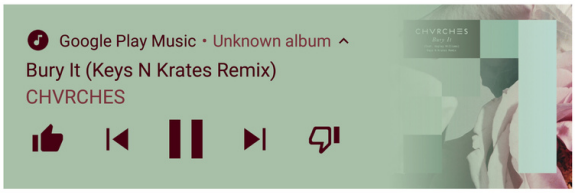
- **Foreground** service — сервис, выполняющий действие, состояние которого важно для пользователя, например, загрузка файла воспроизведение музыки. Такой сервис обязан отображать уведомление в системной шторке уведомлений (примеры: состоян загрузки, название текущей песни и управление воспроизведением). Система считает такой сервис примерно настолько же ва для пользователя, как и текущая activity, и завершит его только в крайнем случае.

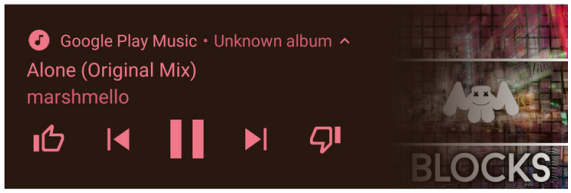


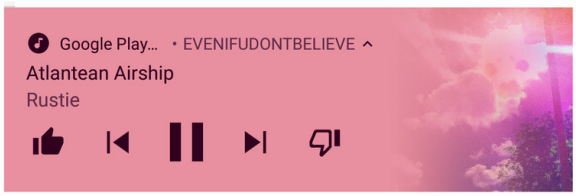


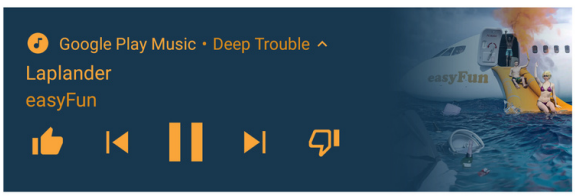












- **Background** service — сервис, выполняющий фоновое действие, состояние которого не интересует пользователя (чаще всего, синхронизацию). Такие сервисы могут быть завершены при нехватке памяти с гораздо большей вероятностью. В старых версия Android большое количество одновременно запущенных фоновых сервисов часто становилось причиной «тормозов»; начиная версии 8.0 Oreo, Android серьёзно ограничивает использование фоновых сервисов, принудительно завершая их через несколь минут после того, как пользователь выходит из приложения.

- **Bound** service — сервис, обрабатывающий входящее Binder-подключение. Такие сервисы предоставляют какую-то функционал для других приложений или системы (например, `WallpaperService` и Google Play Services). В этом случае система может автоматически запускать сервис при подключении к нему клиентов и останавливать его при их отключении.

Рекомендуемый способ выполнять фоновые действия — использование JobScheduler, системного механизма планирования фонов работы. JobScheduler позволяет приложению указать критерии запуска сервиса, такие как:

- **Доступность сети**. Здесь приложение может указать, требуется ли этому сервису наличие сетевого подключения, и если да, т возможна ли работа в роуминге или при использовании лимитного (metered) подключения.
- **Подключение к источнику питания**, что позволяет сервисам выполняться, не «сажая батарею».
- **Бездействие** (idle), что позволяет сервисам выполняться, пока устройство не используется, не замедляя работу во время акти использования.
- **Обновления контента** — например, появление новой фотографии.
- **Период и крайний срок** запуска — например, очистка кэша может производиться ежедневно, а синхронизация событий в кале — каждый час.

JobScheduler планирует выполнение (реализованное как вызов через Binder) зарегистрированных в нём сервисов в соответствии с указанными критериями. Поскольку JobScheduler — общесистемный механизм, он учитывает при планировке критерии зарегистрированных сервисов всех установленных приложений. Например, он может запускать сервисы по очереди, а не

одновременно, чтобы предотвратить резкую нагрузку на устройство во время использования, и планировать периодическое выпол нескольких сервисов небольшими группами (batch), чтобы предотвратить постоянное энергозатратное включение-выключение радиооборудования.

▾ Про TCP-соединение

Как можно заметить, использование JobScheduler не может заменить собой одного из вариантов использования фоновых сервис поддержания TCP-соединения с сервером для получения push-уведомлений. Если бы Android предоставлял приложениям такую возможность, устройству пришлось бы держать *все приложения*, соединяющиеся со своими серверами, запущенными *всё время*, это, конечно, невозможно.

Решение этой проблемы — специальные **push-сервисы**, самый известный из которых — Firebase Cloud Messaging от Google (быв Google Cloud Messaging).

Клиентская часть FCM реализована в приложении Google Play Services. Это приложение, которое специальным образом исключае из обычных ограничений на фоновые сервисы, поддерживает *одно* соединение с серверами Google. Разработчик, желающий отправить своему приложению push-уведомление, пересылает его через серверную часть FCM, после чего приложение Play Serv получив сообщение, передаёт его приложению, которому оно предназначено.

Такая схема позволяет, с одной стороны, мгновенно доставлять push-уведомления всем приложениям (не дожидаясь следующег периода синхронизации), с другой стороны, не держать множество приложений одновременно запущенными.

Broadcast receivers & content providers

Кроме activity и сервисов, у приложений под Android есть два других вида компонентов, менее интересных для обсуждения — это broadcast receiver'ы и content provider'ы.

Broadcast receiver — компонент, позволяющий приложению принимать broadcast'ы, специальный вид сообщений от системы или д приложений. Исходно broadcast'ы, как следует из названия, в основном использовались для рассылки широковещательных сообще всем подписавшимся приложениям — например, система посылает сообщение `AIRPLANE_MODE_CHANGED` при включении или отключе самолётного режима.

Сейчас вместо подписки на такие broadcast'ы, как `NEW_PICTURE` и `NEW_VIDEO`, приложения должны использовать JobScheduler. Broadc используются либо для более редких событий (таких как `BOOT_COMPLETED`), либо с явными intent'ами, то есть именно в качестве сообщения от одного приложения к другому.

Content provider — компонент, позволяющий приложению предоставлять другим приложениям доступ к данным, которыми оно управляет. Пример данных, доступ к которым можно получить таким образом — список контактов пользователя.

При этом приложение может хранить сами данные каким угодно образом, в том числе на устройстве в виде файлов, в настоящей б данных (SQLite) или запрашивать их с сервера по сети. В этом смысле content provider — это унифицированный интерфейс для дос данным, независимо от формы их хранения.

Взаимодействие с content provider'ом во многом похоже на доступ к удалённой базе данных через REST API. Приложение-клиент запрашивает данные по URI (например, `content://com.example.Dictionary.provider/words/42`) через ContentResolver. Приложение-с определяет, к какому именно набору данных был сделан запрос, используя `UriMatcher`, и выполняет запрошенное действие (query, update, delete).

Именно поверх content provider'ов реализован Storage Access Framework, позволяющий приложениям, хранящим файлы в облаке (например, Dropbox и Google Photos) предоставлять доступ к ним остальным приложениям, не занимая место на устройстве полно копией всех хранящихся в облаке файлов.

В следующей статье я расскажу о процессе загрузки Android, о содержимом файловой системы, о том, как хранятся данные пользователя и приложений, и о root-доступе.

Метки: android internals, android, lifecycle, activity, intent, jobscheduler

36,0 Карма 101,7 Рейтинг 92 Подписчики

Сергей Бугаев @bugaevc
Разработчик

Facebook

ПОХОЖИЕ ПУБЛИКАЦИИ

3 августа в 14:46
Как работает Android, часть 1
+90 76,9k 783 49

Комментарии 39

dimonkin 26.09.17 в 15:00

Спасибо. Недавно на собеседовании поднялся вопрос: зачем нужно переносить работу в сервис, если он не foreground и не bound? Ведь гарантий продолжения работы после выхода из активити нет? Ответ — система убьет процесс с запущенным сервисом с меньшей вероятностью. (где это использовать — другой вопрос:))

Также к этому можно добавить про сочетание использования сервисов с WakeLock`ом.

bugaevc 26.09.17 в 15:22

Не просто с меньшей вероятностью.

Во-первых, это *логично* и *правильно*: это независимое действие, а не часть какой-то activity. (Аналогично: зачем разделять программу на функции и классы?)

Во-вторых, некоторые гарантии выполнения после выхода из activity всё-таки есть:

> When an app goes into the background, it has a window of several minutes in which it is still allowed to create and use services. At the end of th window, the app is considered to be idle. At this time, the system stops the app's background services, just as if the app had called the services `Service.stopSelf()` methods. Under certain circumstances, a background app is placed on a temporary whitelist for several minutes. While ar app is on the whitelist, it can launch services without limitation, and its background services are permitted to run.

nekdenis 27.09.17 в 06:58

все прекрасно звучит, пока у тебя не Android Oreo
developer.android.com/about/versions/oreo/background.html#services

bugaevc 27.09.17 в 09:13

Я именно про Oreo и говорю, и выше процитировал именно ту страницу, на которую вы дали ссылку.

forceLain 26.09.17 в 18:53

> Такая схема позволяет, с одной стороны, мгновенно доставлять push-уведомления всем приложениям (не дожидаясь следующего период синхронизации), с другой стороны, не держать множество приложений одновременно запущенными.

Простите за офф-топик, но такая схема еще позволяет доминировать на рынке прошивок. На самом деле, телефон без установленных на не Google Play Services лишается практически всех нужны для нормальной работы приложения механизмов: пушей, карт, локации, activity recognition и т.п.

bugaevc 26.09.17 в 19:22

Да, это пример lock-in'a. С другой стороны, ничто не мешает кому-нибудь разработать свободный (или просто свой) аналог Play Services использовать его.

Doomland 26.09.17 в 19:19

Спасибо и за эту часть статьи. Было сложно, но я справился, осилил и осознал. Жду следующей части, так как там для меня из анонса как р будет ВЕСЬМА интересно. Хотелось бы услышать подробнее о родной файловой системе для Андроида, а так же о поддержке других файло систем. А так же о возможности (лёгкости / сложности) получения рута, и почему его получали через эксплойты.

scrow 26.09.17 в 22:45

Невероятно интересный цикл статей. Спасибо вам!

Alexeyvk 27.09.17 в 09:11

С удовольствием прочел весь цикл, ждем продолжения, спасибо!

ITMOBIT 27.09.17 в 12:00

Большое спасибо за информацию, тема познавательная

point212 27.09.17 в 15:19

Офигенски. Именно такого рода инфморации мне всегда недоставало.
Пытался читать наверное с десяток книжек по архитектуре андроида. Но там всё так затянуто, и разбавлено водой что я не успел узнать ни прежде чем мне надоело читать.

Статьи отличные. Сжато, кратко, доступно.

aquamakc 27.09.17 в 15:39

Я правильно понимаю, что всякие вацапы с вайберами работают, как службы в фоне удерживая постоянное соединение с сервером ожидая входящих сообщений, а GUI активности запускаются только при необходимости. Причём при удалении активити из памяти службы продолж работать? Или GUI для них — это отдельное приложение, которое взаимодействует с сервисами?

> **bugaevc** 27.09.17 в 16:13
>
> Да, именно в этом и есть смысл разделения на activity (GUI), которые могут запускаться, когда нужно, завершаться и перезагружаться; и сервисы, которые остаются запущенными даже без activity. Нет, всё это происходит в рамках одного приложения, конечно.
>
> Но, как я рассказал в статье, Android 8 не позволяет таким приложениям поддерживать соединение в фоне (и вообще серьёзно ограничи фоновые сервисы); вместо этого нужно использовать push-сервисы вроде Firebase Cloud Messaging. Вероятно, многие из этих приложени так и делают.
>
> С другой стороны, пока что Viber и WhatsApp работают на Oreo в режиме совместимости (у обоих target SDK < 26), то есть на них эти ограничения не распространяются.
>
> > **aquamakc** 27.09.17 в 16:23
> >
> > > Нет, всё это происходит в рамках одного приложения, конечно.
> >
> > А есть вообще возможность разделения на разные приложения? Межпроцессорное взаимодействие по примеру тех-же PIPE`ов в десктопных системах или WCF из мира .Net?
> >
> > > **bugaevc** 27.09.17 в 16:34
> > >
> > > Конечно есть! Обычные pipe-ы никуда не делись (Android is Linux), правда совсем не очевидно, как их прокидывать из одного приложения в другое. Основной механизм IPC на Android — Binder, про который я рассказывал в первой статье.
> > >
> > > Я опять приведу в качестве примера Google Play Services — приложение, которое не имеет собственного интерфейса, но реализует разнообразную функциональность, которой пользуются (подключаясь к нему через Binder) другие приложения.

ivan_kov 27.09.17 в 15:46

Спасибо за статью. Нет ли в ваших планах раскрыть тему callback hell в Андроид?

> **bugaevc** 27.09.17 в 15:54
>
> Честно говоря, именно с callback hell я в Android не встречался, но есть и много других, мягко говоря, сложностей, с которыми приходитс сталкиваться разработчику — в основном это вопросы архитектуры приложения и правильной работы с lifecycle (в том числе фрагментов Решается это использованием реактивных библиотек вроде RxJava или новых Android Architecture Components. Подробнее об этом я собираюсь рассказать в одной из следующих статей.

vitraj_petropavl 27.09.17 в 19:16

Большое спасибо! Очень интересно было бы услышать о Firebase Cloud Messaging, как его использовать на практике? Подозреваю в MIUI эт поможет решить проблему с фоновой синхронизацией, так как сторонние приложения узнают о том, что пора синхронизировать что-то от Google Play Services(получат команду и инициализируют полноценное соединение)?

bugaevc 28.09.17 в 11:28

Туториал по практическому использованию FCM в приложениях под Android есть на официальном сайте, но там нет ничего особенно интересного. FCM/GCM — не какое-то новое решение, большинство приложений их уже использует, так что проблемы с синхронизацией вероятно, вызваны чем-то другим.

vitraj_petropavl 28.09.17 в 12:26

Просьба не кидаться камнями, я начинающий разработчик и хочу внедрить более современный(правильный) и надежный механизм синхронизации в свое приложение(синхронизатор буфера обмена между девайсами), который будет использовать гуглосервисы и тем самым экономить заряд, трафик и вообще быть более правильным с точки зрения dev. Спасибо за цикл статей, очень интересно!

Mikenik 01.10.17 в 16:11

> Но основной механизм освобождения памяти в Android — это завершение наименее используемых компонентов приложений (в основном activity)

Это неверно, если только не писать многопроцессное приложение. Завершается процесс целиком.
stackoverflow.com/questions/34834490/android-memory-management-granularity-activity-or-process

bugaevc 01.10.17 в 17:47

К сожалению, этот вопрос плохо описан в документации. Насколько я понимаю, здесь есть такие особенности:

- при нехватке памяти система действительно завершает процессы целиком, при этом она может вызвать, а может и не вызвать `onDestroy()` у запущенных компонентов;
- когда activity завершается не прямо перед завершением всего процесса (а, например, при повороте экрана), `onDestroy()` вызываетс обязательно, но освобождение памяти самой activity происходит через обычный механизм сборки мусора (в том числе, если сохрани лишнюю ссылку, будет activity leak).

Но здесь нужно помнить, что запуск и завершение процесса — это всё-таки более низкий уровень, чем жизненный цикл компонентов приложения. Процитирую первую статью:

> Android максимально абстрагирует понятие *приложение запущено* как от пользователя, так и от разработчика. Конечно, процесс приложения нужно запускать и останавливать, но Android делает это автоматически

То есть на этом более высоком уровне — как я описал в статье — может так получиться, что из двух копий activity одна запущена, а друга полностью завершена — например, если пользователь вернулся в одну из этих копий после того, как (на низком уровне) процесс был завершён из-за временной нехватки памяти, а потом запущен снова.

Mikenik 02.10.17 в 00:47

Согласен, официальная документация как раз является главной причиной подобного заблуждения. Но не стоит его поддерживать и распространять, называя «основным механизмом освобождения памяти в Android», т.к. достаточно достоверно известно (ответы Dian Hackborn и Ian Lake на stackoverflow, а также из собственных тестов), что подобного механизма в Android вообще пока нет.

bugaevc 02.10.17 в 17:42

Наверно, я не очень удачно сформулировал. *Конечно же*, основной механизм освобождения памяти в Android — завершение всего неиспользуемого. Вопрос только в том, может ли приложение перейти из состояния «activity1 работает, acivity2 работает» напряму состояние «activity1 работает, acivity2 завершена», или только через состояние «activity1 заврешена, acivity2 завершена, процесс у

Переформулировал этот кусок в статье.

Mikenik 03.10.17 в 14:49

Теперь другое дело. Однако, есть ещё замечание:

> даёт их компонентам шанс дополнительно освободить ресурсы, вызывая такие методы, как onDestroy, и завершает их, полност освобождая используемую ими память и ресурсы

Насколько мне известно, это тоже неверно и onDestroy в таком случае не вызывается или как минимум не стоит на это рассчиты Например из статьи той же Dianne Hackborn:

> Once Android determines that it needs to remove a process, it does this brutally, simply force-killing it. The kernel can then immediatel reclaim all resources needed by the process, without relying on that application being well written and responsive to a polite request to exit.

Да и в документации повсюду написано, что onDestroy никому и никогда не гарантирован.

bugaevc 03.10.17 в 17:01

Рассчитывать не стоит и не гарантирован (когда по памяти завершают, иначе гарантирован), но в документации явно написано система может его вызвать в такой ситуации:

> `onDestroy()`
>
> Called before the activity is destroyed. This is the final call that the activity receives. The system either invokes this callback because the activity is finishing due to someone's calling `finish()`, **or because the system is temporarily destroying the process containing the activity to save space**. You can distinguish between these two scenarios with the `isFinishing()` method. The system may also call this method when an orientation change occurs, and then immediately call `onCreate()` to recreate the process (and the components that it contains) in the new orientation.

Mikenik 03.10.17 в 18:43

Да, но принципиально, что может не вызвать, соответственно рассчитывать на этот метод как освобождающий ресурсы или извещающий о том, что сейчас приложение окончательно убьют не имеет смысла. А абзац как раз про механизмы освобождения памяти.

Wildwilcat 02.10.17 в 12:06

Спасибо, интересно

Ktulkhu_Triediniy 03.10.17 в 15:49

Долгожданное продолжение отличного цикла статей. Спасибо!

Damir79 03.10.17 в 16:18

Спасибо за статьи! Очень интересно!
Вопрос есть, я столкнулся с проблемой передачи данных по сети.
ru.stackoverflow.com/questions/725580/android-http-connection-refused
Мой сервис, каждые 10 секунд отправляет данные на сервер. Сервис Sticky.
На Андроид 4.1.2 все работает хорошо, а на Андроид 5.1.1 нет. Как только система уничтожит сервис при нехватке памяти, то после перезапуска сервиса отправка данных на сервер перестает работать. Если вручную удалить приложение из списка задач и запустить снова, то отправка начинает работать. Также отправка возобновляется, если я снова жму иконку программы в списке задач. Насколько я понял из статьи с 5.0 версии Андроида используется новая JVM. Не с этим ли связана эта проблема? Как мне ее решить? Запускать activity и service в разных процессах? У меня возникло ощущение, что система прибивая процесс и запуская его снова, оставляет заблокированными сетевые порты.

bugaevc 03.10.17 в 16:48

Нет, переход на ART вряд ли повлиял на работу сетевых соединений.

(Этот абзац не специфичен для Android, а верен на любой Linux-системе) Действительно, система может освобождать занятые порты не сразу после завершения державшего их процесса, а только после отправки всех буферизованных данных (это называется socket linger, и с этим часто борются с помощью SO_REUSEADDR), но это актуально для портов с заранее известным номером. Если вы просто делаете POST-запрос, система автоматически назначит вам подходящий свободный порт, и этой проблемы не возникнет.

Хабр — не Тостер, но думаю, вам стоит посмотреть с помощью сниффера, в чём разница в успешных и неуспешных запросах, как вам и порекомендовали на StackOverflow.

Ну и не могу не упомянуть, что отправка данных на сервер каждые 10 секунд — это, мягко говоря, не лучшая идея. Создавать для этого каждый раз новый сокет *и забывать его закрыть* — тоже.

Damir79 04.10.17 в 06:39

Спасибо за ответ! Да, попробую в направлении закрытия сокетов порыть. Хотя я пробовал использовать хваленую библиотеку async-h ничего не меняется. Сниффером смотрел, система создает каждый раз новый порт. После перезапуска сервиса, http запросы в телеф висят в состоянии SYN-SENT, т.е. запрос на открытие порта отправляется на сервер, но обратный пакет SYN-ACK похоже блокируется системой.

Damir79 04.10.17 в 09:27

Порылся в направлении сокетов. Все корректно закрывается, портов открытых нет, сниффер показывает пустоту. Лишь мелькает запро раз в 10 секунд SYN_SENT и все (на 5.1.1). На 4.1.2 работает без сбоев.

Damir79 04.10.17 в 06:38

Удалено

ArtRoman 04.10.17 в 16:52

За несколько лет разработки под Android я запомнил, что система не богами писалась, и никто ничем не обязан. Даже если стандартные ме lifecycle приложения должны вызываться, то наверняка будут, но не обязаны. Если броадкасты работают, то они могут перестать работать п очередного обновления системы. Если даже они не указаны как sticky, то при подписке могут приходить события «changed» с текущим состоянием, т.е. sticky-поведением. Всегда надёжнее проверить каждый метод на практике, чем надеяться что оно всегда работает как опи в документации. Никто не гарантирует, что support-библиотека всегда будет отрабатывать корректно на всех устройствах, а не будет привс к крашу приложения (привет, Samsung). Нельзя заранее узнать, будет ли тот или иной кодек или контейнер поддерживаться устройством, особенно с включенным аппаратным ускорением (привет, китайские производители). Нельзя заранее узнать, сколько видео одновременно может воспроизводиться. Кстати, если на андроиде 4.4 и ниже инициализировать штатный VideoView и удалять его много раз подряд, то че какое-то время в системе кончится лимит транзакций биндера, при любом системном обращении будет выброшен TransactionTooLargeExce (репа с деталями). Недавно столкнулся с низкоуровневой сетевой ошибкой «ENOBUFS (No buffer space available)» при долгой работе устрой с частым реконнектом к серверу, независимо от версии системы. И если разобраться с самой системой обычно можно благодаря открытос системы и наличию исходного кода, то всё гораздо интереснее с вещами, которые предоставляет железо и вендоры устройства.

Damir79 05.10.17 в 14:36

Да, странная вещь у меня получилась. Если сервис словил ошибку соединения, то он будет периодически слать запросы на подключение серверу (SYN_SENT), а запросы подтверждения (SYN_ACK) почему-то будут блокироваться. Стоит только активность запустить, сразу же соединение устанавливается. Может это на моих Xiaomi (5.1.1, 6.0.1) какая-то особенность.

Damir79 05.10.17 в 14:43

Какие продукты написал? Есть что на Google Play посмотреть? Я только год в Андроиде, свою первую прогу написал, выложил в сеть. См а у пользователей связь рвется через некоторое время. Вот и нашел эту особенность с потерей сети в сервисах. Сам Google рекомендуе использовать FCM сервис оповещений для фоновых приложений. Хотя я смотрю у WhatsApp соединение с XMPP сервером постоянное. К они умудряются это делать в фоне непонятно.

ArtRoman 05.10.17 в 17:06

В основном коммерческая разработка. По рвущемуся коннекту – есть аналогичное, при выключении экрана активное соединение рвё через 10-15 минут, и больше не переподключается, несмотря на всевозможные wake-lock'и, только активация экрана помогает. Это оптимизации со стороны системы, чтобы фоновые приложения не жрали аккумулятор, трафик и прочие ресурсы. Для периодических подключений рекомендуют использовать JobScheduler, либо пуш-уведомления через GCM (нынче Firebase). В качестве костыля у меня работает программная активация экрана (через wakelock) при потере коннекта.

Damir79 06.10.17 в 07:06

Спасибо за подсказку! GCM уже начал изучать, а вот про wakelock даже в голову не приходило. Пишут с 8 андроида фоновым сетев соединениям вообще трындец настанет, так что похоже надо на GCM по любому переходить.

+19 9,9k 50 10

ИНТЕРЕСНЫЕ ПУБЛИКАЦИИ

Моделирование объектов, функций и операций. Мереологические отношения между объектами данного типа

+6 417 7 0

На пути к естественному интеллекту

+11 1,4k 19 4

Хаскель — ход конем II GT

+9 1,3k 6 4

Простая Scada на Python и Arduino

+11 2,7k 29 4

Sportiduino. Система электронной отметки для спортивного ориентирования. Часть 2 GT

+18 1,7k 16 0