**Министерство образования и науки Российской Федерации**

**Федеральное государственное автономное образовательное учреждение высшего**

**профессионального образования Московский физико-технический институт (национальный исследовательский университет)**

**Факультет управления и прикладной математики**

**Кафедра теоретической кибернетики и методов оптимального управления**

**На правах рукописи**

**УДК 539.12**

**Михальченко Егор Викторович**

Сравнительный анализ Android приложения, использующего для отображения экранов

Fragment либо View

Выпускная квалификационная магистерская работа

**Направление подготовки 03.04.01 Прикладная математика и** **физика**

|  |  |  |
| --- | --- | --- |
| **Научный руководитель** |  | **Петрухин В.А.** |
| **Студент** |  | **Михальченко Е.В.** |

**Москва 2019 г.**

**Содержание**

1. **Введение**
2. **Обзор исследования**
   1. Что именно нужно проверить
   2. Как планируется проверять (обзор методов исследования)
3. **Обзор основных технологий реализации**
   1. Объект Android Fragment [Fragment](https://developer.android.com/guide/components/fragments)
   2. Библиотека Conductor [Exploring Conductor](https://android.jlelse.eu/exploring-conductor-android-app-development-without-fragments-part-1-fce1eab0c9df) [Conductor](https://github.com/bluelinelabs/Conductor)

# Объект Android View (в том числе Invalidations, Layouts, and Performance) [View](https://developer.android.com/reference/android/view/View) [Custom View](https://developer.android.com/guide/topics/ui/custom-components) [Custom View 2](https://tuhub.ru/posts/realizatsiya-custom-view-komponenta-v-android)

* 1. Объект Android Activity [Activity](https://developer.android.com/reference/android/app/Activity) [Introduction to Activities](https://developer.android.com/guide/components/activities/intro-activities)
  2. Объект Android Application и взаимодействие с Android OS (как стартует и что может повлиять) [Android architecture](https://xakep.ru/2014/05/21/excurse-in-android-architecture/)
  3. Single Activity [medium](https://medium.com/rosberryapps/a-single-activity-android-application-why-not-fa2a5458a099)

1. **Исследование применимости**
   1. Работа с жизненным циклом
   2. Интеграция с различными архитектурами ( [android architectures](https://academy.realm.io/posts/mvc-vs-mvp-vs-mvvm-vs-mvi-mobilization-moskala/) )
   3. Результаты замены Fragment’ов на Controller’ы (run tests regression after add conductor)
   4. Гибкость использования
   5. Возможные проблемы
2. **Тестирование:**
   1. Обзор методов тестирования
      1. Параметры, влияющие на результаты тестов
      2. Фреймворки [Inspect GPU rendering speed and overdraw](https://developer.android.com/topic/performance/rendering/inspect-gpu-rendering#profile_rendering) [Benchmarking](https://blog.mindorks.com/improving-android-app-performance-with-benchmarking) [Benchmark 2](https://developer.android.com/studio/profile/benchmark) [Firebase](https://firebase.google.com/docs/perf-mon) [Android Profiler](https://developer.android.com/studio/profile/android-profiler) [Memory allocations](https://developer.android.com/studio/profile/memory-profiler) [Espresso](https://developer.android.com/training/testing/espresso) [Espresso 2](https://developer.android.com/training/testing/ui-testing/espresso-testing) [UI Automator](https://developer.android.com/training/testing/ui-automator) [UI Automator 2](https://developer.android.com/training/testing/ui-testing/uiautomator-testing) [Overview of system tracing](https://developer.android.com/topic/performance/tracing)
      3. Выбранные фреймворки
   2. Проводимые тесты
      1. Временные тесты
      2. GPU Rendering тесты [Analyze with Profile GPU Rendering](https://developer.android.com/topic/performance/rendering/profile-gpu)
      3. Профилирование (часть от [Android Profiler](https://developer.android.com/studio/profile/android-profiler))
   3. Проблемы тестирования
      1. Библиотека Benchmark
      2. Библиотека Espresso (нужен idle)
      3. Количество тестов (цикличность, также с idle)
      4. -
3. **Побочные выводы**
4. **Заключение**

**Введение**

Научная актуальность

Различные проблемы с эксплуатацией решений по отображению экрана всплывали с самого начала.

В первых версиях это были такие компоненты как Activity (далее активити) [1], и каждый новый экран это был новый экземпляр объекта данного класса

Далее появились более легковесные решения – Fragment (далее фрагменты) [2], которые могли существовать на базе той же Activity, те быть её частью, но уже новые экраны – это были иногда Activity, иногда Fragment’ы. Были разные подходы (разные потому что фрагменты имели различные преимущества и в зависимости от необходимости тех или иных преимуществ в конкретном так или иначе могли строить архитектуру), но чаще всего делали одну активити под определенную цепочку экранов, а сами эти экраны были фрагментами [3].

Архитектурные решения менялись, в старых находили недостатки и новые приходили упрощать жизнь при разработке. И так со временем появился подход: Single Activity [см далее]. Процесс построения экранов, когда на всё многоэкранное приложение существует только одна активити, которая, по сути, является контейнером для фрагментов.

И вот, учитывая, что у всех решений есть плюсы и минусы, некоторые решили, что фрагменты использовать это слишком усложнённо (у него, например, методов жизненного цикла выделяют даже больше, чем у активити; есть различные вопросы и к реализации, и др.). То разработчики не из Google (а все выше перечисленные решения располагались в стандартных библиотеках Android, который и разрабатывает Google) решили сделать более легковесную реализацию экранов, названную Conductor [4]. Это библиотека, которая основана на View (далее вью) [5], её используют абсолютно все вышеперечисленные решения, это в рамках экранов является строительной единицей.

О данной библиотеке отзываются лестно в том числе такие люди как (см [reddit](https://www.reddit.com/r/androiddev/comments/4em04y/whats_the_verdict_on_conductor_why_use_it_and_why/)): Jake Wharton (самый известный разработчик Android, создал много open-source фреймворком, которыми пользуются почти все android разработчики ) и Hannes Dorfmann (также очень известный android энтузиаст, который пишет много развивающих статей и распространяет/внедряет новые подходы, более оптимальные в android разработке).

Так как о данном решении разработчики что-то слышали краем уха, но относятся с недоверием, потому что это не внутренняя библиотека платформы, но решение выглядит привлекательным [4], было решено провести анализ данного решения. А «контрольной группой» выступить распространенное на текущий момент решение Single Activity, использующее Fragment’ы.

Практическую значимость, цель и задачи проводимого исследования

Практическая значимость данного исследования заключается в том, чтобы выяснить, как соотносится решение в виде Conductor (View) с проблемами разработчиков в сравнении с решением в виде Fragment. Сюда ведь входит не только измерение основных действий для решений отображения экранов, но и также гибкости настройки решения, масштабируемости и, далеко не в последнюю очередь по значимости, удобство применения разработчиками.

Тема не нова, но проводимых сколь-нибудь комплексных исследований не найдено. Даже если какие-то разработчики что-то для себя пытался выяснять, то это не было представлено широкой публике.

**Обзор исследования**

**Что именно нужно проверить**

Нам необходимо понять, какие есть различия в приложениях написанном на Fragment и на Controller (View).

В первую очередь это будут проверки обычных пользовательских действий в приложении, нет ли видимых глазу человека проблем. Это затронет в том числе и проверки на время между действиями пользователя и реакцией приложения на них, различия в реакциях видимые и невидимые.

Далее необходимо будет проверить на сколько разработчику удобно применять фрагмент или контроллер при решении поставленных продуктом задач. На сколько продуманны интерфейсы на текущий момент, на сколько сложно их дополнять, на сколько просто допустить ошибку при их использовании (а это прямо пропорционально количеству багов при использовании библиотеки).

**Как планируется проверять**

Для нахождения ответов на поставленные вопросы следует использовать одно и то же приложение, с одним лишь различием – фрагменты оно использует или вью. Это нужно, чтобы никакие другие отличающиеся элементы не могли вносить дополнительные расхождения в результатах. Android OS это очень большая и сложная система. Ведь она предполагает работу на крайне ограниченных ресурсах, также следует учитывать, что мобильная индустрия довольно быстро и сильно развивается последние десятилетия и как и любая другая система написана людьми. Поэтому следует по возможности свести различия к минимуму при проведении тестов.

При написании приложения специально для анализа этого приложения очень часто не возникает проблем, которые могли бы быть при написании enterprise решений для решения конкретных реальных задач. Поэтому следует выбрать как минимум среднего размера приложение с большим количеством экранов. Open source более менее серьёзных проектов достаточно много.

Следует не забывать, что при тестировании следует покрыть как можно больше часто используемых кейсов, поэтому необходимо smoke тестирование этих приложений, пройдясь по основных экранам и затронуть важный функционал.

И напоследок перед остановкой на конкретном приложении следует ознакомиться с технологиями, используемыми при его написании. Не стоит использовать крайне редкие фреймворки и подходы, так как мы хотим выяснить, возможно ли применение исследуемой технологии большинством разработчиков.

Изначально было около 7 кандидатов. Несколько отпало на этапе исследования функционала (он оказывался часто слишком недостаточным, те приложение не очень большое), далее несколько отпало на этапе исследования структуры экранов (довольно часто они были очень однообразными, так как часто и методы решения различных задач похожи). В конце концов осталось 2 приложения схожие по объему и разнообразию используемых подходов построения экранов. Понадобилось начать пробовать внедрять Conductor в оба (беглый взгляд не позволяет рассмотреть end-to-end цепочки выполнения кода), чтобы понять, на сколько распространённые подходы применяются при написании приложения.

Итог: было выбрано open source приложения от компании Google “The Google I/O 2019” [6]. Плюсом среди прочих этого приложения является то, что достаточно много android разработчиков c ним знакомо. И поэтому, если они захотят познакомиться с Conductor посредством моей работы, то они легко смогут погрузиться в работу.

**Обзор основных технологий реализации**

Для того, чтобы определиться, на какие моменты следует обращать внимание при исследовании и лучше понять результаты тестов, необходимо понимать, как устроено Android приложение. Данная глава не даст широкого представления об android разработке, но погрузит в основные понятие и объекты приложений.

**Объект Android Fragment**

Фрагмент (класс Fragment ) представляет поведение или часть пользовательского интерфейса в операции (класс Activity ). Разработчик может объединить несколько фрагментов в одну операцию для построения многопанельного пользовательского интерфейса и повторного использования фрагмента в нескольких операциях. Фрагмент можно рассматривать как модульную часть операции. Такая часть имеет свой жизненный цикл и самостоятельно обрабатывает события ввода. Кроме того, ее можно добавить или удалить непосредственно во время выполнения операции. Это нечто вроде вложенной операции, которую можно многократно использовать в различных операциях.

Фрагмент всегда должен быть встроен в операцию, и на его жизненный цикл напрямую влияет жизненный цикл операции. Например, когда операция приостановлена, в том же состоянии находятся и все фрагменты внутри нее, а когда операция уничтожается, уничтожаются и все фрагменты. Однако пока операция выполняется (это соответствует состоянию возобновлена жизненного цикла), можно манипулировать каждым фрагментом независимо, например добавлять или удалять их. Когда разработчик выполняет такие транзакции с фрагментами, он может также добавить их в стек переходов назад, которым управляет операция. Каждый элемент стека переходов назад в операции является записью выполненной транзакции с фрагментом. Стек переходов назад позволяет пользователю обратить транзакцию с фрагментом (выполнить навигацию в обратном направлении), нажимая кнопку Назад.

Когда фрагмент добавлен как часть макета операции, он находится в объекте ViewGroup внутри иерархии представлений операции и определяет собственный макет представлений. Разработчик может вставить фрагмент в макет операции двумя способами. Для этого следует объявить фрагмент в файле макета операции как элемент <fragment> или добавить его в существующий объект ViewGroup в коде приложения. Впрочем, фрагмент не обязан быть частью макета операции. Можно использовать фрагмент без интерфейса в качестве невидимого рабочего потока операции.

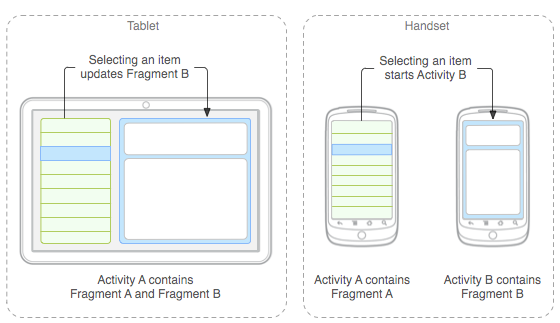
В этом документе показано, как построить приложение, использующее фрагменты. В частности, обсуждается, как фрагменты могут поддерживать свое состояние, когда они добавляются в стек переходов назад операции, использовать события совместно с операцией и другими фрагментами внутри нее, выводить данные в строку действий операции и т.д.

Философия проектирования

Фрагменты впервые появились в Android версии 3.0 (API уровня 11), главным образом, для обеспечения большей динамичности и гибкости пользовательских интерфейсов на больших экранах, например, у планшетов. Поскольку экраны планшетов гораздо больше, чем у смартфонов, они предоставляют больше возможностей для объединения и перестановки компонентов пользовательского интерфейса. Фрагменты позволяют делать это, избавляя разработчика от необходимости управлять сложными изменениями в иерархии представлений. Разбивая макет операции на фрагменты, разработчик получает возможность модифицировать внешний вид операции в ходе выполнения и сохранять эти изменения в стеке переходов назад, которым управляет операция.

Например, новостное приложение может использовать один фрагмент для показа списка статей слева, а другой—для отображения статьи справа. Оба фрагмента отображаются за одну операцию рядом друг с другом, и каждый имеет собственный набор методов обратного вызова жизненного цикла и управляет собственными событиями пользовательского ввода. Таким образом, вместо применения одной операции для выбора статьи, а другой — для чтения статей, пользователь может выбрать статью и читать ее в рамках одной операции, как на планшете, изображенном на рисунке 1.

Следует разрабатывать каждый фрагмент как модульный и повторно используемый компонент операции. Поскольку каждый фрагмент определяет собственный макет и собственное поведение со своими обратными вызовами жизненного цикла, разработчик может включить один фрагмент в несколько операций. Поэтому он должен предусмотреть повторное использование фрагмента и не допускать, чтобы один фрагмент непосредственно манипулировал другим. Это особенно важно, потому что модульность фрагментов позволяет изменять их сочетания в соответствии с различными размерами экранов. Если приложение должно работать и на планшетах, и на смартфонах, можно повторно использовать фрагменты в различных конфигурациях макета, чтобы оптимизировать взаимодействие с пользователем в зависимости от доступного размера экрана. Например, на смартфоне может возникнуть необходимость в разделении фрагментов для предоставления однопанельного пользовательского интерфейса, если разработчику не удается поместить более одного фрагмента в одну операцию.



Создание фрагмента

Для создания фрагмента необходимо создать подкласс класса Fragment (или его существующего подкласса). Класс Fragment имеет код, во многом схожий с кодом Activity . Он содержит методы обратного вызова, аналогичные методам операции, такие как onCreate() , onStart() , onPause() и onStop() . На практике, если требуется преобразовать существующее приложение Android так, чтобы в нем использовались фрагменты, достаточно просто переместить код из методов обратного вызова операции в соответствующие методы обратного вызова фрагмента.

Как правило, необходимо реализовать следующие методы жизненного цикла:

onCreate()

Система вызывает этот метод, когда создает фрагмент. В своей реализации разработчик должен инициализировать ключевые компоненты фрагмента, которые требуется сохранить, когда фрагмент находится в состоянии паузы или возобновлен после остановки.

onCreateView()

Система вызывает этот метод при первом отображении пользовательского интерфейса фрагмента на дисплее. Для прорисовки пользовательского интерфейса фрагмента следует возвратить из этого метода объект View , который является корневым в макете фрагмента. Если фрагмент не имеет пользовательского интерфейса, можно возвратить null.

onPause()

Система вызывает этот метод как первое указание того, что пользователь покидает фрагмент (это не всегда означает уничтожение фрагмента). Обычно именно в этот момент необходимо фиксировать все изменения, которые должны быть сохранены за рамками текущего сеанса работы пользователя (поскольку пользователь может не вернуться назад).

большинстве приложений для каждого фрагмента должны быть реализованы, как минимум, эти три метода. Однако существуют и другие методы обратного вызова, которые следует использовать для управления различными этапами жизненного цикла фрагмента. Все методы обратного вызова жизненного цикла подробно обсуждаются в разделе Управление жизненным циклом фрагмента.

Существует также ряд подклассов, которые, возможно, потребуется расширить вместо использования базового класса Fragment :

DialogFragment

Отображение перемещаемого диалогового окна. Использование этого класса для создания диалогового окна является хорошей альтернативой вспомогательным методам диалогового окна в классе Activity . Дело в том, что он дает возможность вставить

агментов, что позволяет пользователю вернуться к закрытому фрагменту.

ListFragment

Отображение списка элементов, управляемых адаптером (например, SimpleCursorAdapter ), аналогично классу ListActivity . Этот класс предоставляет несколько методов для управления списком представлений, например, метод обратного вызова onListItemClick() для обработки нажатий.

PreferenceFragment

Отображение иерархии объектов Preference в виде списка, аналогично классу PreferenceActivity . Этот класс полезен, когда в приложении создается операция «Настройки».

Создание макета

Параметр container , передаваемый методу onCreateView() , является родительским классом ViewGroup (из макета операции), в который будет вставлен макет фрагмента. Параметр savedInstanceState является классом Bundle , который предоставляет данные о предыдущем экземпляре фрагмента во время возобновления фрагмента (восстановление состояния подробно обсуждается в разделе Управление жизненным циклом фрагмента).

Метод inflate() принимает три аргумента:

- Идентификатор ресурса макета, раздувание которого следует выполнить.

- Объект класса ViewGroup , который должен стать родительским для макета после раздувания. Передача параметра container необходима для того, чтобы система смогла применить параметры макета к корневому представлению раздутого макета, определяемому родительским представлением, в которое направляется макет.

- Логическое значение, показывающее, следует ли прикрепить макет к объекту ViewGroup (второй параметр) во время раздувания. (В данном случае это false, потому что система уже вставляет раздутый макет в объект container , ипередача значения true создала бы лишнюю группу представления в окончательном макете).

Мы увидели, как создавать фрагмент, предоставляющий макет. Теперь необходимо добавить фрагмент в операцию.

Добавление фрагмента в операцию

Как правило, фрагмент добавляет часть пользовательского интерфейса в операцию, и этот интерфейс встраивается в общую иерархию представлений операции. Разработчик может добавить фрагмент в макет операции двумя способами:

*объявив фрагмент в файле макета операции*.

В этом случае можно указать свойства макета для фрагмента, как будто он является представлением.

Атрибут android:name в элементе <fragment> определяет класс Fragment , экземпляр которого создается в макете.

Когда система создает этот макет операции, она создает экземпляр каждого фрагмента, определенного в макете, и для каждого вызывает метод onCreateView() , чтобы получить макет каждого фрагмента. Система вставляет объект View , возвращенный фрагментом, непосредственно вместо элемента <fragment> .

*Примечание.* Каждый фрагмент должен иметь уникальный идентификатор, который система сможет использовать для восстановления фрагмента в случае перезапуска операции. (Что касается разработчика, он может использовать этот идентификатор для захвата фрагмента с целью выполнения транзакций с ним, например, чтобы удалить его). Предоставить идентификатор фрагменту можно тремя способами:

- указать атрибут android:id с уникальным идентификатором;

- указать атрибут android:tag с уникальной строкой;

- ничего не предпринимать, чтобы система использовала идентификатор контейнерного представления.

*или программным образом, добавив фрагмент в существующий объект ViewGroup .*

В любой момент выполнения операции разработчик может добавить фрагменты в ее макет Для этого достаточно указать объект ViewGroup в котором следует разместить фрагмент.

Для выполнения транзакций с фрагментами внутри операции (таких как добавление, удаление или замена фрагмента) необходимо использовать API-интерфейсы из FragmentTransaction . Экземпляр класса FragmentTransaction можно получить от объекта Activity следующим образом:

FragmentManager fragmentManager = getFragmentManager()

FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();

После этого можно добавить фрагмент методом add() , указав добавляемый фрагмент и представление, в которое он должен быть добавлен. Например:

ExampleFragment fragment = new ExampleFragment(); fragmentTransaction.add(R.id.fragment\_container, fragment); fragmentTransaction.commit();

Первый аргумент, передаваемый методу add() , представляет собой контейнерный объект ViewGroup для фрагмента, указанный при помощи идентификатора ресурса. Второй параметр — это фрагмент, который нужно добавить.

Выполнив изменения с помощью FragmentTransaction , необходимо вызвать метод commit() , чтобы они вступили в силу.

Добавление фрагмента, не имеющего пользовательского интерфейса

Пример, приведенный выше, демонстрирует, как добавлять в операцию фрагмент с предоставлением пользовательского интерфейса. Однако можно использовать фрагмент и для реализации фонового поведения операции без какого-либо дополнительного пользовательского интерфейса.

Чтобы добавить фрагмент без пользовательского интерфейса, добавьте фрагмент из операции, используя метод add(Fragment, String) (передав ему уникальный строковый «тег» для фрагмента вместо идентификатора представления). Фрагмент будет добавлен, но, поскольку он не связан с представлением в макете операции, он не будет принимать вызов метода onCreateView() . Поэтому в реализации этого метода нет необходимости.

Передача строкового тега свойственна не только фрагментам без пользовательского интерфейса, поэтому можно передавать строковые теги и фрагментам, имеющим пользовательский интерфейс. Однако, если у фрагмента нет пользовательского интерфейса, то строковый тег является единственным способом его идентификации. Если впоследствии потребуется получить фрагмент от операции, нужно будет вызвать метод findFragmentByTag() .

Пример операции, использующей фрагмент в качестве фонового потока, без пользовательского интерфейса, приведен в образце кода FragmentRetainInstance.java , входящем в число образцов в SDK (и доступном при помощи Android SDK Manager). Путь к нему в системе —

<sdk\_root>/APIDemos/app/src/main/java/com/example/android/apis/app/FragmentRetainInstance.java .

Управление фрагментами

Для управления фрагментами в операции нужен класс FragmentManager . Чтобы получить его, следует вызвать метод getFragmentManager() из кода операции.

Ниже указаны действия, которые позволяет выполнить FragmentManager :

- получать фрагменты, имеющиеся в операции, с помощью метода findFragmentById() (для фрагментов, предоставляющих пользовательский интерфейс в макете операции) или findFragmentByTag() (как для фрагментов, имеющих пользовательский интерфейс, так и для фрагментов без него);

- снимать фрагменты со стека переходов назад методом popBackStack() (имитируя нажатие кнопки Назад пользователем);

- регистрировать процесс-слушатель изменений в стеке переходов назад при помощи метода addOnBackStackChangedListener() .

Дополнительные сведения об этих и других методах приводятся в документации по классу FragmentManager.

Как было показано в предыдущем разделе, можно использовать класс FragmentManager для открытия FragmentTransaction , что позволяет выполнять транзакции с фрагментами, например, добавление и удаление.

Выполнение транзакций с фрагментами

Большим достоинством использования фрагментов в операции является возможность добавлять, удалять, заменять их и выполнять другие действия с ними в ответ на действия пользователя. Любой набор изменений, вносимых в операцию, называется транзакцией. Ее можно выполнить при помощи API-интерфейсов в FragmentTransaction . Каждую транзакцию можно сохранить в стеке переходов назад, которым управляет операция. Это позволит пользователю перемещаться назад по изменениям во фрагментах (аналогично перемещению назад по операциям).

Экземпляр класса FragmentTransaction можно получить от FragmentManager , например, так:

FragmentManager fragmentManager = getFragmentManager();

FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();

Каждая транзакция является набором изменений, выполняемых одновременно. Разработчик может указать все изменения, которые ему нужно выполнить в данной транзакции, вызывая методы add() , remove() и replace() . Затем, чтобы применить транзакцию к операции, следует вызвать метод commit() .

Впрочем, до вызова метода commit() у разработчика может возникнуть необходимость вызвать метод addToBackStack() , чтобы добавить транзакцию в стек переходов назад по транзакциям фрагмента. Этим стеком переходов назад управляет операция, что позволяет пользователю вернуться к предыдущему состоянию фрагмента, нажав кнопку Назад.

Например, следующий код демонстрирует, как можно заменить один фрагмент другим, сохранив при этом предыдущее состояние в стеке переходов назад:

//Create new fragment and transaction

Fragment newFragment = new ExampleFragment();

FragmentTransaction transaction = getFragmentManager().beginTransaction();

//Replace whatever is in the fragment\_container view with this fragment, and add the transaction to the back stack transaction.replace(R.id.fragment\_container, newFragment); transaction.addToBackStack(null);

//Commit the transaction

transaction.commit();

В этом коде объект newFragment замещает фрагмент (если таковой имеется), находящийся в контейнере макета, на который указывает идентификатор R.id.fragment\_container . В результате вызова метода addToBackStack() транзакция замены сохраняется в стеке переходов назад, чтобы пользователь мог обратить транзакцию и вернуть предыдущий фрагмент, нажав кнопку Назад.

Если в транзакцию добавить несколько изменений (например, еще раз вызвать add() или remove() ), а затем вызвать addToBackStack() , все изменения, примененные до вызова метода commit() , будут добавлены в стек переходов назад как одна транзакция, и кнопкаНазад обратит их все вместе.

Порядок добавления изменений к объекту FragmentTransaction не играет роли за следующими исключениями:

- метод commit() должен быть вызван в последнюю очередь;

- если в один контейнер добавляется несколько фрагментов, то порядок их добавления определяет порядок, в котором они появляются в иерархии видов.

Если при выполнении транзакции, удаляющей фрагмент, не вызвать метод addToBackStack() , при фиксации транзакции фрагмент уничтожается, и пользователь теряет возможность вернуться к нему. В то же время, если вызвать addToBackStack() при удалении фрагмента, фрагмент перейдет в состояние остановки и будет возобновлен, если пользователь вернется к нему.

*Совет*. К каждой транзакции с фрагментом можно применить анимацию перехода, вызвав setTransition() до фиксации.

Вызов метода commit() не приводит к немедленному выполнению транзакции. Метод запланирует ее выполнение в потоке пользовательского интерфейса операции (в «главном» потоке), как только у потока появится возможность для этого. Впрочем, при необходимости можно вызвать executePendingTransactions() из потока пользовательского интерфейса, чтобы транзакции, запланированные методом commit() были выполнены немедленно. Как правило, в этом нет необходимости, за исключением случаев, когда транзакция является зависимостью для заданий в других потоках.

*Внимание!* Фиксировать транзакцию методом commit() можно только до того, как операциясохранит свое состояние (после того, как пользователь покинет ее). Попытка

зафиксировать транзакцию после этого момента вызовет исключение. Дело в том, что состояние после фиксации может быть потеряно, если понадобится восстановить операцию. В ситуациях, в которых потеря фиксации не критична, следует вызывать commitAllowingStateLoss().

Взаимодействие с операцией

Хотя Fragment реализован как объект, независимый от класса Activity , и может быть использован внутри нескольких операций, конкретный экземпляр фрагмента напрямую связан с содержащей его операцией.

В частности, фрагмент может обратиться к экземпляру Activity с помощью метода getActivity() и без труда выполнить такие задачи, как поиск представления в макете операции:

View listView = getActivity().findViewById(R.id.list);

Аналогичным образом операция может вызывать методы фрагмента, получив ссылку на объект Fragment от FragmentManager с помощью метода findFragmentById() или findFragmentByTag() . Например:

ExampleFragment fragment = (ExampleFragment) getFragmentManager().findFragmentById(R.id.example\_fragment);

Создание обратного вызова события для операции

В некоторых случаях необходимо, чтобы фрагмент использовал события совместно с операцией. Хороший способ реализации этого состоит в том, чтобы определить интерфейс обратного вызова внутри фрагмента и потребовать от контейнерной операции его реализации. Когда операция примет обратный вызов через этот интерфейс, она сможет обмениваться информацией с другими фрагментами в макете по мере необходимости.

Пусть, например, у новостного приложения имеются два фрагмента в одной операции: один для отображения списка статей (фрагмент A), а другой—для отображения статьи (фрагмент B). Тогда фрагмент A должен сообщать операции о том, что выбран пункт списка, чтобы она могла сообщить фрагменту B о необходимости отобразить статью. В этом случае интерфейс

OnArticleSelectedListener объявляется во фрагменте A:

public static class FragmentA extends ListFragment {

...

Container Activity must implement this interface public interface OnArticleSelectedListener {

public void onArticleSelected(Uri articleUri);

}

...

}

Тогда операция, содержащая этот фрагмент, реализует интерфейс OnArticleSelectedListener и переопределит метод onArticleSelected() , чтобы извещать фрагмент B о событии исходящем от фрагмента A Чтобы контейнерная операция наверняка

реализовала этот интерфейс, метод обратного вызова onAttach() во фрагменте A (который система вызывает при добавлении фрагмента в операцию) создает экземпляр класса OnArticleSelectedListener , выполнив приведение типа объекта Activity , который передается методу onAttach() :

public static class FragmentA extends ListFragment { OnArticleSelectedListener mListener;

...

@Override

public void onAttach(Activity activity) {

super.onAttach(activity);

try {

mListener = (OnArticleSelectedListener) activity;

} catch (ClassCastException e) {

throw new ClassCastException(activity.toString() + " must implement OnArticleSelectedListener");

}

}

...

}

Если операция не реализовала интерфейс, фрагмент генерирует исключение ClassCastException . В случае успеха элемент mListener будет содержать ссылку на реализацию интерфейса OnArticleSelectedListener в операции, чтобы фрагмент A мог использовать события совместно с операцией, вызывая методы, определенные интерфейсом OnArticleSelectedListener . Например, если фрагмент A является расширением класса ListFragment , то всякий раз, когда пользователь нажимает элемент списка, система вызывает onListItemClick() во фрагменте. Этот метод, в свою очередь, вызывает метод onArticleSelected() , чтобы использовать событие совместно с операцией:

public static class FragmentA extends ListFragment { OnArticleSelectedListener mListener;

...

@Override

public void onListItemClick(ListView l, View v, int position, long id) {

//Append the clicked item's row ID with the content provider Uri

Uri noteUri = ContentUris.withAppendedId(ArticleColumns.CONTENT\_URI, id);

//Send the event and Uri to the host activity

mListener.onArticleSelected(noteUri);

}

...

}

Управление жизненным циклом фрагмента

Управление жизненным циклом фрагмента во многом аналогично управлению жизненным циклом операции. Как и операция, фрагмент может существовать в одном из трех состояний:

Resumed

Фрагмент виден во время выполнения операции.

Paused

На переднем плане выполняется и находится в фокусе другая операция, но операция, содержащая данный фрагмент, по-прежнему видна (операция переднего плана частично прозрачна или не занимает весь экран).

Stopped

Фрагмент не виден. Либо контейнерная операция остановлена, либо фрагмент удален из нее, но добавлен в стек переходов назад. Остановленный фрагмент по-прежнему активен (вся информация о состоянии и элементах сохранена в системе). Однако он больше не виден пользователю и будет уничтожен в случае уничтожения операции.

Здесь снова просматривается аналогия с операцией: разработчик может сохранить состояние фрагмента с помощью Bundle на случай, если процесс операции будет уничтожен, а разработчику понадобится восстановить состояние фрагмента при повторном создании операции. Состояние можно сохранить во время выполнения метода обратного вызова onSaveInstanceState() во фрагменте и восстановить его во время выполнения onCreate() , onCreateView() или onActivityCreated() . Дополнительные сведения о сохранении состояния приводятся в документе Операции.

Самое значительное различие в ходе жизненного цикла между операцией и фрагментом состоит в принципах их сохранения в соответствующих стеках переходов назад. По умолчанию операция помещается в управляемый системой стек переходов назад для операций, когда она останавливается (чтобы пользователь мог вернуться к ней с помощью кнопки Назад, как описано в статье Задачи и стек переходов назад). В то же время, фрагмент помещается в стек переходов назад, управляемый операцией, только когда разработчик явно запросит сохранение конкретного экземпляра, вызвав метод addToBackStack() во время транзакции, удаляющей фрагмент.

В остальном управление жизненным циклом фрагмента очень похоже на управление жизненным циклом операции. Поэтому практические рекомендации по управлению жизненным циклом операций применимы и к фрагментам. При этом разработчику необходимо понимать, как жизненный цикл операции влияет на жизненный цикл фрагмента.

*Внимание!* Если возникнет необходимость в объекте Context внутри объекта класса Fragment, можно вызвать методgetActivity(). Однако разработчик должен быть внимательным и вызывать метод getActivity() только когда фрагмент прикреплен к операции. Если фрагмент еще не прикреплен или был откреплен в конце его жизненного цикла, метод getActivity() возвратит null.

Согласование с жизненным циклом операции

Жизненный цикл операции, содержащей фрагмент, непосредственным образом влияет на жизненный цикл фрагмента, так что каждый обратный вызов жизненного цикла операции приводит к аналогичному обратного вызову для каждого фрагмента. Например, когда операция принимает вызов onPause() , каждый ее фрагмент принимает onPause() .

Однако у фрагментов есть несколько дополнительных методов обратного вызова жизненного цикла, которые обеспечивают уникальное взаимодействие с операцией для выполнения таких действий, как создание и уничтожение пользовательского интерфейса фрагмента. Вот эти методы:

onAttach()

Вызывается, когда фрагмент связывается с операцией (ему передается объект Activity ).

onCreateView()

Вызывается для создания иерархии представлений, связанной с фрагментом.

onActivityCreated()

Вызывается, когда метод onCreate() , принадлежащий операции, возвращает управление.

onDestroyView()

Вызывается при удалении иерархии представлений, связанной с фрагментом

onDetach()

Вызывается при разрыве связи фрагмента с операцией.

Зависимость жизненного цикла фрагмента от содержащей его операции иллюстрируется рисунком 3. На этом рисунке можно видеть, что очередное состояние операции определяет, какие методы обратного вызова может принимать фрагмент. Например, когда операция принимает свой метод обратного вызова onCreate() , фрагмент внутри этой операции принимает всего лишь метод обратного вызова onActivityCreated() .

Когда операция переходит в состояние «возобновлена», можно свободно добавлять в нее фрагменты и удалять их. Таким образом, жизненный цикл фрагмента может быть независимо изменен, только пока операция остается в состоянии «возобновлена».

Однако, когда операция выходит из этого состояния, продвижение фрагмента по его жизненному циклу снова осуществляется операцией.

**Библиотека Conductor**

Небольшая, но полнофункциональная платформа, позволяющая создавать приложения Android на основе View. Conductor предоставляет облегченную оболочку для стандартных представлений Android, которая выполняет практически все, что вам нужно:

Плюсы, заявляемые разработчиками:

- Простая интеграция

- Приложения с одним действием без использования фрагментов

- Простое, но мощное управление жизненным циклом

- Навигация и обратная обработка

- Красивые переходы между видами

- Сохранение состояния

- Обратные вызовы для onActivityResult, onRequestPermissionsResult и т. д.

- Подходит для любых архитектур: MVP / MVVM / MVI / VIPER / MVC

Компоненты фреймворка

Controller - это оболочка View, которая предоставит вам все функции управления жизненным циклом. Думайте об этом как о более легкой и более предсказуемой альтернативе фрагмента с более простым в управлении жизненным циклом.

маршрутизатор

Router реализует навигацию и обработку backstack для контроллеров. Объекты маршрутизатора прикреплены к Activity / содержат пары ViewGroup. Маршрутизаторы не визуализируют напрямую или не отправляют Views в контейнер ViewGroup, а вместо этого переносят эту ответственность на ControllerChangeHandler, указанный в данной транзакции.

ControllerChangeHandlers отвечают за переключение представления одного контроллера на представление другого. Они могут быть полезны для выполнения анимаций и переходов между контроллерами. Несколько стандартных ControllerChangeHandlers включены.

RouterTransaction

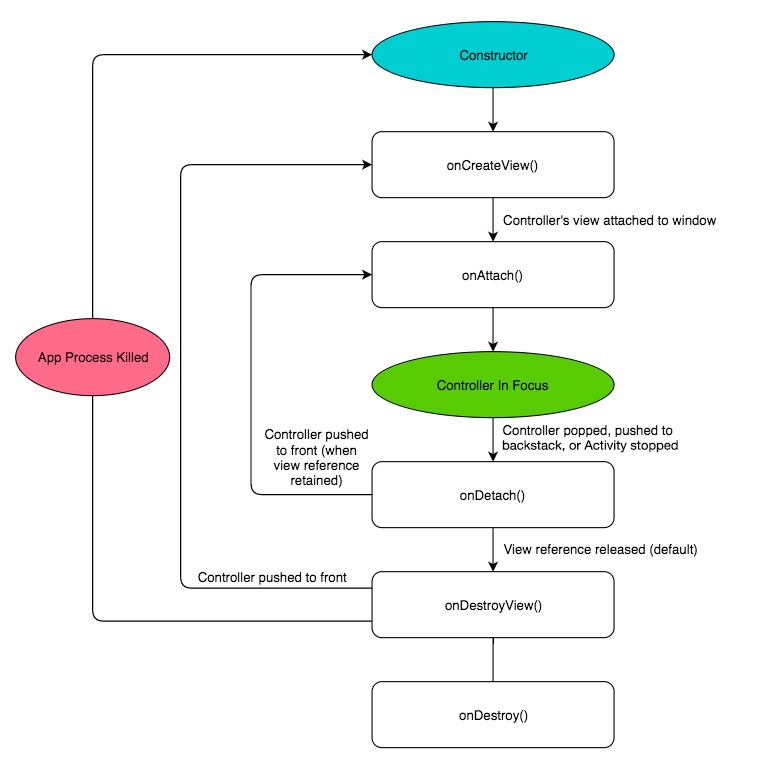
Транзакции используются для определения данных о добавлении контроллеров. RouterTransactions используются для передачи контроллера к маршрутизатору с указанным ControllerChangeHandlers, в то время как ChildControllerTransactions используются для добавления дочерних контроллеров

Контроллер против Фрагмента

Проводник был сделан для решения проблем, обозначенных фрагментами. В этом разделе рассматриваются основные различия между контроллерами и фрагментами Conductor.

Жизненный цикл

Следующая диаграмма иллюстрирует жизненный цикл контроллеров Conductor.



Как видно из диаграммы, жизненный цикл контроллеров намного проще по сравнению с фрагментами (заинтересованные читатели могут просмотреть полную диаграмму жизненного цикла фрагмента, созданную Square здесь). Всякий раз, когда контроллер помещается в маршрутизатор, вызывается его конструктор, если он не существует. Затем вызывается onCreateView, чтобы создать представление для отображения. Затем onAttach вызывается, когда контроллер присоединен к его ViewGroup. Всякий раз, когда текущий Контроллер извлекается из заднего стека, или новый Контроллер вставляется, или активность хоста останавливается, контроллер отсоединяется от своего хоста ViewGroup и, таким образом, выполняется обратный вызов onDetach. По умолчанию Conductor освобождает ссылку на представление Контроллера после его отсоединения, и, следовательно, выполняется обратный вызов onDestroyView.

Контроллеры могут быть настроены на сохранение своих представлений после их отсоединения. Это полезно для контроллеров с иерархиями представлений, которые дорого разбирать и перестраивать. В этом случае onDestroyView не будет вызываться после onDetach, и, следовательно, onCreateView также не будет вызываться, если контроллер снова возвращался на передний план.

Асинхронность. В отличие от FragmentTransaction, транзакции контроллера выполняются немедленно. Это исключает вероятность появления неизвестных состояний в ваших приложениях.

Контроллеры сохраняются во время изменения ориентации, что аналогично Fragment.setRetainInstance (true). Это чрезвычайно полезно в случае, если устройство поворачивается (и, таким образом, текущий вид уничтожается), когда в фоновом режиме выполняется длинная операция (например, сетевой запрос), чтобы получить данные для текущего вида. В таких случаях, поскольку контроллер не уничтожен, он дает вам прекрасную возможность продолжить отслеживать ход операции и обновлять представление, когда результат становится доступным. Например, если используется MVP, презентатор, который можно использовать для отслеживания операции и обновления представления, может быть создан как переменная экземпляра контроллера и, следовательно, пережить изменения ориентации.

По сравнению с фрагментами информации о Conductor гораздо меньше, поэтому наша цель и выяснить на практике, как он соотносится с базовой технологией.

**Объект Android View**

Теперь узнаем поподробнее об объекте, используемом обоими технологиями.

Этот класс представляет основной строительный блок для компонентов пользовательского интерфейса. Представление занимает прямоугольную область на экране и отвечает за рисование и обработку событий. Представление - это базовый класс для виджетов, которые используются для создания интерактивных компонентов пользовательского интерфейса (кнопок, текстовых полей и т. Д.). Подкласс ViewGroup является базовым классом для макетов, которые представляют собой невидимые контейнеры, которые содержат другие виды (или другие ViewGroups) и определяют их свойства макета.

Использование представлений

Все виды в окне расположены в одном дереве. Вы можете добавить представления либо из кода, либо указав дерево представлений в одном или нескольких файлах макета XML. Существует много специализированных подклассов представлений, которые действуют как элементы управления или способны отображать текст, изображения или другой контент.

После того, как вы создали дерево представлений, обычно вы можете выполнить несколько типов общих операций:

• Установить свойства: например, установить текст TextView. Доступные свойства и методы, которые их устанавливают, будут различаться среди разных подклассов представлений. Обратите внимание, что свойства, которые известны во время сборки, могут быть установлены в файлах макета XML.

• Установить фокус: платформа будет обрабатывать движущийся фокус в ответ на ввод пользователя. Чтобы перевести фокус на конкретное представление, вызовите requestFocus ().

• Настройка прослушивателей: представления позволяют клиентам настраивать прослушиватели, которые будут уведомлены, когда с представлением произойдет что-то интересное. Например, все представления позволят вам настроить слушателя на уведомление, когда представление приобретает или теряет фокус. Вы можете зарегистрировать такого слушателя, используя setOnFocusChangeListener (android.view.View.OnFocusChangeListener). Другие подклассы представления предлагают более специализированные слушатели. Например, кнопка предоставляет слушателю уведомление клиентов при нажатии кнопки.

• Установить видимость: вы можете скрыть или показать виды, используя setVisibility (int).

*Примечание.* Платформа Android отвечает за измерение, компоновку и отрисовку видов. Вы не должны вызывать методы, которые выполняют эти действия для представлений самостоятельно, если вы фактически не реализуете ViewGroup.

Реализация пользовательского представления

Чтобы реализовать пользовательское представление, вы обычно начинаете с предоставления переопределений для некоторых стандартных методов, которые инфраструктура вызывает для всех представлений. Вам не нужно переопределять все эти методы. Фактически, вы можете начать с переопределения onDraw (android.graphics.Canvas)

Идентификаторы

Представления могут иметь целочисленный идентификатор, связанный с ними. Эти идентификаторы обычно назначаются в файлах XML макета и используются для поиска определенных представлений в дереве представлений.

Идентификаторы представлений не обязательно должны быть уникальными по всему дереву, но рекомендуется убедиться, что они, по крайней мере, уникальны в той части дерева, которую вы ищете.

Позиция

Геометрия вида - это геометрия прямоугольника. Вид имеет местоположение, выраженное в виде пары левых и верхних координат, и два измерения, выраженные в виде ширины и высоты. Единицей расположения и размеров является пиксель.

Можно получить местоположение представления, вызвав методы getLeft () и getTop (). Первый возвращает левую или X координату прямоугольника, представляющего вид. Последний возвращает координату вершины или Y прямоугольника, представляющего вид. Оба эти метода возвращают местоположение представления относительно его родителя. Например, когда getLeft () возвращает 20, это означает, что представление расположено в 20 пикселях справа от левого края его прямого родителя.

Кроме того, предлагается несколько удобных методов, чтобы избежать ненужных вычислений, а именно getRight () и getBottom (). Эти методы возвращают координаты правого и нижнего краев прямоугольника, представляющего вид. Например, вызов getRight () похож на следующие вычисления: getLeft () + getWidth () (см. Размер для получения дополнительной информации о ширине.)

Размер, отступы и поля

Размер представления выражается шириной и высотой. Вид фактически обладает двумя парами значений ширины и высоты.

Первая пара известна как измеренная ширина и измеренная высота. Эти измерения определяют, насколько большим должно быть представление в пределах его родителя (см. Layout для более подробной информации.) Измеренные измерения можно получить, вызвав getMeasuredWidth () и getMeasuredHeight ().

Вторая пара просто называется шириной и высотой, а иногда шириной и высотой рисования. Эти размеры определяют фактический размер вида на экране, во время рисования и после макета. Эти значения могут, но не обязательно, отличаться от измеренной ширины и высоты. Ширина и высота могут быть получены путем вызова getWidth () и getHeight ().

Чтобы измерить его размеры, представление принимает во внимание его дополнение. Заполнение выражается в пикселях для левой, верхней, правой и нижней частей вида. Заполнение может использоваться для смещения содержимого представления на определенное количество пикселей. Например, левый отступ 2 будет сдвигать содержимое представления на 2 пикселя справа от левого края. Заполнение может быть установлено с использованием метода setPadding (int, int, int, int) или setPaddingRelative (int, int, int, int) и запрошено путем вызова getPaddingLeft (), getPaddingTop (), getPaddingRight (), getPaddingBottom (), getPaddingStart ( ), getPaddingEnd ().

Даже если представление может определять отступы, оно не обеспечивает поддержки полей. Однако группы просмотра предоставляют такую ​​поддержку. Обратитесь к ViewGroup и ViewGroup.MarginLayoutParams для получения дополнительной информации.

Макет

Макет представляет собой двухпроходный процесс: этап измерения и этап макета. Проход измерения реализуется в мере (int, int) и является обходом дерева представлений сверху вниз. Каждый вид проталкивает спецификации размеров вниз по дереву во время рекурсии. В конце прохода измерения каждое представление сохраняет свои измерения. Второй проход происходит в макете (int, int, int, int) и также выполняется сверху вниз. Во время этого прохода каждый родитель отвечает за размещение всех своих дочерних элементов, используя размеры, вычисленные в проходе измерения.

Когда метод view () возвращается, его значения getMeasuredWidth () и getMeasuredHeight () должны быть установлены вместе со значениями для всех потомков этого представления. Измеренная ширина представления и измеренные значения высоты должны учитывать ограничения, накладываемые родителями представления. Это гарантирует, что в конце прохождения меры все родители примут все измерения своих детей. Родительское представление может вызывать measure () более одного раза для своих дочерних элементов. Например, родитель может измерить каждого ребенка один раз с неопределенными измерениями, чтобы выяснить, насколько большим они хотят быть, а затем снова вызвать measure () для них с фактическими числами, если сумма всех неограниченных размеров детей слишком велика или слишком мала.

Для передачи измерения используются два класса. Класс MeasureSpec используется представлениями, чтобы сообщить своим родителям, как они хотят измеряться и позиционироваться. Базовый класс LayoutParams просто описывает, насколько большим должно быть представление для ширины и высоты. Для каждого измерения можно указать одно из:

- Точное число

- MATCH\_PARENT, что означает, что представление хочет быть таким же большим, как его родитель (минус отступ)

- WRAP\_CONTENT, что означает, что представление хочет быть достаточно большим, чтобы вместить его содержимое (плюс заполнение).

Существуют подклассы LayoutParams для разных подклассов ViewGroup. Например, AbsoluteLayout имеет свой собственный подкласс LayoutParams, который добавляет значения X и Y.

MeasureSpecs используются для передачи требований вниз по дереву от родителя к потомку. MeasureSpec может быть в одном из трех режимов:

- UNSPECIFIED: это используется родителем, чтобы определить желаемое измерение дочернего представления. Например, LinearLayout может вызывать measure () для своего дочернего элемента с высотой, установленной на UNSPECIFIED, и шириной EXACTLY 240, чтобы выяснить, какой высоты дочернему представлению нужно присвоить ширину 240 пикселей.

Точно: это используется родителем, чтобы наложить точный размер на ребенка. Ребенок должен использовать этот размер и гарантировать, что все его потомки будут соответствовать этому размеру.

- AT\_MOST: это используется родителем, чтобы наложить максимальный размер на ребенка. Ребенок должен гарантировать, что он и все его потомки будут соответствовать этому размеру.

Чтобы инициировать макет, вызовите requestLayout (). Этот метод обычно вызывается представлением о себе, когда он считает, что больше не может вписываться в свои текущие границы.

Отрисовка

Рисование выполняется путем обхода дерева и записи команд рисования любого вида, который необходимо обновить. После этого на экран выводятся команды рисования всего дерева, обрезанные до вновь поврежденной области.

Дерево в основном записывается и рисуется по порядку, родители рисуются раньше (то есть позади) своих детей, а братья и сестры рисуются в том порядке, в котором они появляются на дереве. Если вы устанавливаете фон для рисования для View, то View будет рисовать его перед вызовом метода onDraw (). Дочерний порядок рисования может быть переопределен с помощью ViewGroup # setChildrenDrawingOrderEnabled (boolean) в ViewGroup и с помощью setZ (float) пользовательских значений Z}, установленных в Views.

Чтобы заставить представление рисовать, вызовите invalidate ().

Обработка событий и создание потоков

Основной цикл представления выглядит следующим образом:

Событие приходит и отправляется в соответствующий вид. Представление обрабатывает событие и уведомляет всех слушателей.

Если в ходе обработки события, возможно, потребуется изменить границы представления, представление вызовет requestLayout ().

Аналогично, если в ходе обработки события может потребоваться изменить внешний вид представления, представление вызовет invalidate ().

Если были вызваны requestLayout () или invalidate (), каркас позаботится об измерении, разметке и построении дерева соответствующим образом.

Примечание. Все дерево представлений является однопоточным. Вы всегда должны быть в потоке пользовательского интерфейса при вызове любого метода в любом представлении. Если вы работаете с другими потоками и хотите обновить состояние представления из этого потока, вам следует использовать обработчик.

Фокусировка

Фреймворк будет обрабатывать обычное движение фокуса в ответ на пользовательский ввод. Это включает в себя изменение фокуса при удалении или скрытии видов или при появлении новых видов. Представления указывают на их готовность сфокусироваться через метод isFocusable (). Чтобы изменить, может ли представление получить фокус, вызовите setFocusable (boolean). В сенсорном режиме (см. Примечания ниже) представления указывают, хотят ли они по-прежнему фокусироваться через isFocusableInTouchMode () и могут изменить это через setFocusableInTouchMode (логическое значение).

Движение фокуса основано на алгоритме, который находит ближайшего соседа в заданном направлении. В редких случаях алгоритм по умолчанию может не соответствовать предполагаемому поведению разработчика. В этих ситуациях вы можете предоставить явные переопределения, используя следующие атрибуты XML в файле макета:

nextFocusDown

nextFocusLeft

nextFocusRight

nextFocusUp

Чтобы получить конкретный вид, чтобы сфокусироваться, вызовите requestFocus ().

Сенсорный режим

Когда пользователь перемещается по пользовательскому интерфейсу с помощью клавиш со стрелками, таких как D-панель, необходимо сосредоточить внимание на элементах, которые можно активировать, таких как кнопки, чтобы пользователь мог видеть, что будет вводить. Однако, если устройство обладает сенсорными возможностями, и пользователь начинает взаимодействовать с интерфейсом, касаясь его, больше нет необходимости всегда выделять или фокусировать внимание на определенном виде. Это мотивирует режим взаимодействия под названием «сенсорный режим».

Для сенсорного устройства, когда пользователь касается экрана, устройство переходит в сенсорный режим. С этого момента будут доступны только те виды, для которых isFocusableInTouchMode () имеет значение true, например виджеты для редактирования текста. Другие виды, которые являются сенсорными, например кнопки, не будут фокусироваться при прикосновении; они будут только запускать слушателей по щелчку.

Каждый раз, когда пользователь нажимает клавишу направления, такую ​​как направление D-панели, устройство просмотра выходит из режима касания и находит изображение, чтобы сфокусироваться, чтобы пользователь мог возобновить взаимодействие с пользовательским интерфейсом, не касаясь экрана снова.

Состояние сенсорного режима поддерживается во всех действиях. Вызовите isInTouchMode (), чтобы увидеть, находится ли устройство в данный момент в сенсорном режиме.

Промотка

Платформа обеспечивает базовую поддержку представлений, которые хотят прокручивать свой контент. Это включает в себя отслеживание смещения прокрутки X и Y, а также механизмы рисования полос прокрутки. Посмотрите scrollBy (int, int), scrollTo (int, int) и awakenScrollBars () для получения дополнительной информации.

Теги

В отличие от идентификаторов, теги не используются для идентификации представлений. По сути, теги - это дополнительная часть информации, которая может быть связана с представлением. Чаще всего они используются для удобства хранения данных, связанных с представлениями, в самих представлениях, а не в виде отдельной структуры.

Теги могут быть указаны со значениями последовательности символов в XML макета в виде одного тега с использованием атрибута android: tag или нескольких тегов с использованием дочернего элемента <tag>:

<View ...

android: tag = "@ string / mytag\_value" />

<View ...>

<tag android: id = "@ + id / mytag"

android: value = "@ string / mytag\_value" />

</ View>

Теги также могут быть указаны с произвольными объектами из кода с использованием setTag (java.lang.Object) или setTag (int, java.lang.Object).

Темы

По умолчанию представления создаются с использованием темы объекта Context, предоставленного их конструктору; однако другая тема может быть указана с помощью атрибута android: theme в XML-макете или путем передачи ContextThemeWrapper конструктору из кода.

Когда атрибут android: theme используется в XML, указанная тема применяется поверх темы контекста инфляции (см. LayoutInflater) и используется как для самого представления, так и для любых дочерних элементов.

В следующем примере оба вида будут созданы с использованием темной цветовой схемы материала; однако, поскольку используется тема наложения, которая определяет только подмножество атрибутов, значение android: colorAccent, определенное в теме контекста инфляции (например, тема «Активность»), будет сохранено.

<LinearLayout

...

Android: theme = "@ android: theme / ThemeOverlay.Material.Dark">

<View ...>

</ LinearLayout>

Свойства

Класс View предоставляет свойство ALPHA, а также несколько свойств, связанных с преобразованием, таких как TRANSLATION\_X и TRANSLATION\_Y. Эти свойства доступны как в форме «Свойство», так и в методах setter / getter с аналогичными именами (например, setAlpha (float) для ALPHA). Эти свойства можно использовать для установки постоянного состояния, связанного с этими связанными с рендерингом свойствами в представлении. Свойства и методы также можно использовать в сочетании с анимациями на основе аниматора, которые описаны более подробно в разделе «Анимация».

Анимация

Начиная с Android 3.0, предпочтительным способом анимации представлений является использование API-пакетов пакета android.animation. Эти основанные на Animator классы изменяют фактические свойства объекта View, такие как alpha и translationX. Это поведение отличается от поведения классов, основанных на Animation до 3.0, которые вместо этого анимируют только то, как изображение отображается на дисплее. В частности, класс ViewPropertyAnimator делает анимацию этих свойств View особенно простой и эффективной.

Кроме того, вы можете использовать классы анимации до 3.0, чтобы анимировать, как визуализируются представления. Вы можете прикрепить объект Animation к представлению, используя setAnimation (android.view.animation.Animation) или startAnimation (android.view.animation.Animation). Анимация может изменять масштаб, вращение, перевод и альфа-представление вида во времени. Если анимация прикреплена к представлению, у которого есть дочерние элементы, анимация повлияет на все поддерево, укорененное этим узлом. Когда анимация запускается, фреймворк позаботится о перерисовке соответствующих видов, пока анимация не завершится.

Безопасность

Иногда важно, чтобы приложение могло проверить, что действие выполняется с полным знанием и согласием пользователя, таким как предоставление запроса на разрешение, совершение покупки или нажатие на рекламу. К сожалению, вредоносное приложение может попытаться обмануть пользователя, не подозревая об этом, выполнить эти действия, скрывая предназначенную цель представления. В качестве исправления платформа предлагает механизм сенсорной фильтрации, который можно использовать для повышения безопасности представлений, обеспечивающих доступ к конфиденциальным функциям.

Чтобы включить сенсорную фильтрацию, вызовите setFilterTouchesWhenObscured (boolean) или установите атрибут макета android: filterTouchesWhenObscured в значение true. Когда эта функция включена, платформа будет отбрасывать касания, полученные, когда окно представления скрыто другим видимым окном. В результате, представление не будет получать прикосновения всякий раз, когда над окном представления появляется всплывающее окно, диалоговое окно или другое окно.

Для более детального контроля безопасности рассмотрите возможность переопределения метода onFilterTouchEventForSecurity (android.view.MotionEvent) для реализации собственной политики безопасности. Смотрите также MotionEvent # FLAG\_WINDOW\_IS\_OBSCURED.

Пользовательские компоненты просмотра

Android предлагает сложную и мощную компонентную модель для создания вашего пользовательского интерфейса, основанную на фундаментальных классах компоновки: View и ViewGroup. Начнем с того, что платформа включает в себя множество предварительно созданных подклассов View и ViewGroup - называемых виджетами и макетами соответственно - которые вы можете использовать для создания своего пользовательского интерфейса.

Частичный список доступных виджетов включает Button, TextView, EditText, ListView, CheckBox, RadioButton, Gallery, Spinner и более специализированный AutoCompleteTextView, ImageSwitcher и TextSwitcher.

В число доступных макетов входят LinearLayout, FrameLayout, RelativeLayout и другие. Для большего количества примеров, см. Common Layout Objects.

Если ни один из готовых виджетов или макетов не отвечает вашим потребностям, вы можете создать свой собственный подкласс View. Если вам нужно всего лишь внести небольшие изменения в существующий виджет или макет, вы можете просто создать подкласс виджета или макета и переопределить его методы.

Создание ваших собственных подклассов View дает вам точный контроль над внешним видом и функциями элемента экрана. Чтобы дать представление об элементе управления, который вы получаете с помощью пользовательских представлений, вот несколько примеров того, что вы можете сделать с ними:

- Вы можете создать полностью настраиваемый тип представления, например, регулятор «громкости», отображаемый с использованием 2D-графики и напоминающий аналоговый электронный элемент управления.

- Вы можете объединить группу компонентов View в новый отдельный компонент, возможно, сделать что-то вроде ComboBox (комбинация всплывающего списка и свободного текстового поля ввода), двухпанельный элемент управления селектором (левая и правая панели со списком в каждый, где вы можете переназначить, какой элемент в каком списке), и так далее.

- Вы можете изменить способ визуализации компонента EditText на экране (в «Учебном блокноте» это эффективно используется для создания размеченной страницы блокнота).

- Вы можете захватывать другие события, такие как нажатия клавиш, и обрабатывать их по-своему (например, для игры).

В следующих разделах объясняется, как создавать собственные представления и использовать их в вашем приложении. Для получения подробной справочной информации см. Класс View.

Основной подход

Вот краткий обзор того, что вам нужно знать, чтобы начать создавать собственные компоненты View:

Расширьте существующий класс или подкласс View своим собственным классом.

Переопределите некоторые методы из суперкласса. Методы суперкласса для переопределения начинаются с 'on', например, onDraw (), onMeasure () и onKeyDown (). Это похоже на события on ... в Activity или ListActivity, которые вы переопределяете для жизненного цикла и других функций.

Используйте свой новый класс расширения. После завершения ваш новый класс расширения может использоваться вместо представления, на котором он основан.

Совет: классы расширения могут быть определены как внутренние классы внутри действий, которые их используют. Это полезно, потому что оно контролирует доступ к ним, но не является необходимым (возможно, вы хотите создать новое общедоступное представление для более широкого использования в вашем приложении).

Полностью настраиваемые компоненты

Полностью настраиваемые компоненты могут использоваться для создания графических компонентов, которые отображаются по вашему желанию. Возможно, графический измеритель VU, который выглядит как старый аналоговый датчик, или текстовый текст «поешь в длинную», где прыгающий шар движется вдоль слов, чтобы ты мог петь вместе с караоке-машиной. В любом случае, вы хотите что-то, что встроенные компоненты просто не будут делать, независимо от того, как вы их объедините.

К счастью, вы можете легко создавать компоненты, которые выглядят и ведут себя любым способом, ограниченным, возможно, только вашим воображением, размером экрана и доступной вычислительной мощностью (помните, что в конечном итоге ваше приложение может работать на чем-то значительно меньшем мощность, чем ваша настольная рабочая станция).

Чтобы создать полностью настроенный компонент:

1. Неудивительно, что самым общим представлением, которое вы можете расширить, является View, поэтому вы обычно начинаете с расширения этого, чтобы создать новый суперкомпонент.
2. Вы можете предоставить конструктор, который может принимать атрибуты и параметры из XML, и вы также можете использовать свои собственные такие атрибуты и параметры (возможно, цвет и диапазон измерителя VU или ширину и демпфирование стрелки и т. Д.)
3. Возможно, вы захотите создать свои собственные слушатели событий, методы доступа и модификаторы свойств, а также, возможно, более сложное поведение в вашем классе компонентов.
4. Вы почти наверняка захотите переопределить onMeasure (), а также, вероятно, потребуется переопределить onDraw (), если вы хотите, чтобы компонент показал что-то. Хотя оба имеют поведение по умолчанию, onDraw () по умолчанию ничего не делает, а onMeasure () по умолчанию всегда устанавливает размер 100x100, что, вероятно, не то, что вам нужно.
5. Другие on ... методы также могут быть переопределены при необходимости.

Расширить onDraw () и onMeasure ()

Метод onDraw () предоставляет вам Canvas, на котором вы можете реализовать все, что захотите: 2D-графику, другие стандартные или пользовательские компоненты, стилизованный текст или все, что вы только можете придумать.

*Примечание*. Это не относится к трехмерной графике. Если вы хотите использовать трехмерную графику, вы должны расширить SurfaceView вместо View и рисовать из отдельного потока. Посмотрите пример GLSurfaceViewActivity для деталей.

onMeasure () немного сложнее. onMeasure () является важной частью контракта рендеринга между вашим компонентом и его контейнером. Функция onMeasure () должна быть переопределена, чтобы эффективно и точно сообщать об измерениях содержащихся в ней частей. Это немного усложняется требованиями пределов от родителя (которые передаются в метод onMeasure ()) и требованием вызывать метод setMeasuredDimension () с измеренными шириной и высотой после их вычисления. Если вам не удастся вызвать этот метод из переопределенного метода onMeasure (), результатом будет исключение во время измерения.

На высоком уровне реализация onMeasure () выглядит примерно так:

1. Переопределенный метод onMeasure () вызывается со спецификациями мер ширины и высоты (параметры widthMeasureSpec и heightMeasureSpec, оба являются целочисленными кодами, представляющими размеры), которые должны рассматриваться как требования к ограничениям измерений ширины и высоты, которые вы должны произвести. Полная ссылка на виды ограничений, которые могут потребоваться этими спецификациями, может быть найдена в справочной документации в View.onMeasure (int, int) (эта справочная документация также довольно хорошо объясняет всю операцию измерения).
2. Метод onMeasure () вашего компонента должен вычислить ширину и высоту измерения, которые потребуются для визуализации компонента. Он должен стараться не выходить за пределы переданных спецификаций, хотя он может их превысить (в этом случае родитель может выбрать, что делать, в том числе вырезать, прокрутить, вызвать исключение или попросить onMeasure () повторить попытку, возможно с другими характеристиками измерения).
3. После вычисления ширины и высоты метод setMeasuredDimension (int width, int height) должен быть вызван с вычисленными измерениями. Невыполнение этого условия приведет к исключению.

Составные элементы управления

Если вы не хотите создавать полностью настраиваемый компонент, а вместо этого хотите собрать повторно используемый компонент, состоящий из группы существующих элементов управления, то создание Составного компонента (или Составного элемента управления) может соответствовать всем требованиям. В двух словах, это объединяет ряд более элементарных элементов управления (или представлений) в логическую группу элементов, которые можно рассматривать как одну вещь. Например, поле со списком может рассматриваться как комбинация однострочного поля EditText и смежной кнопки с прикрепленным списком PopupList. Если вы нажимаете кнопку и выбираете что-то из списка, оно заполняет поле EditText, но пользователь также может ввести что-то непосредственно в EditText, если они того пожелают.

В Android на самом деле есть два других представления, доступных для этого: Spinner и AutoCompleteTextView, но независимо от этого концепция Combo Box представляет собой простой для понимания пример.

Чтобы создать составной компонент

1. Обычной отправной точкой является какой-либо макет, поэтому создайте класс, расширяющий макет. Возможно, в случае поля со списком мы могли бы использовать LinearLayout с горизонтальной ориентацией. Помните, что другие макеты могут быть вложены внутрь, поэтому составной компонент может быть произвольно сложным и структурированным. Обратите внимание, что, как и в случае с Activity, вы можете использовать декларативный (основанный на XML) подход к созданию содержащихся компонентов или программно вкладывать их из своего кода.
2. В конструкторе нового класса возьмите те параметры, которые ожидает суперкласс, и передайте их сначала конструктору суперкласса. Затем вы можете настроить другие представления для использования в вашем новом компоненте; Здесь вы можете создать поле EditText и PopupList. Обратите внимание, что вы также можете ввести свои собственные атрибуты и параметры в XML, которые могут быть извлечены и использованы вашим конструктором.
3. Вы также можете создавать прослушиватели для событий, которые могут генерироваться вашими содержащимися представлениями, например, метод прослушивателя для элемента списка. Нажмите «Прослушиватель», чтобы обновить содержимое EditText, если сделан выбор списка.
4. Вы также можете создать свои собственные свойства с помощью методов доступа и модификаторов, например, разрешить первоначальное задание значения EditText в компоненте и запросить его содержимое при необходимости.
5. В случае расширения макета вам не нужно переопределять методы onDraw () и onMeasure (), поскольку макет будет иметь поведение по умолчанию, которое, скорее всего, будет работать нормально. Тем не менее, вы все равно можете переопределить их, если вам нужно.
6. Вы можете переопределить другие методы on ..., например onKeyDown (), чтобы, возможно, выбрать определенные значения по умолчанию из всплывающего списка поля со списком при нажатии определенной клавиши.

Подводя итог, можно сказать, что использование макета в качестве основы для пользовательского элемента управления имеет ряд преимуществ, в том числе:

* Вы можете указать макет, используя декларативные XML-файлы, как на экране активности, или вы можете программно создавать представления и вкладывать их в макет из своего кода.
* Методы onDraw () и onMeasure () (плюс большинство других методов on ...), вероятно, будут иметь подходящее поведение, поэтому вам не придется их переопределять.
* В конце концов, вы можете очень быстро создать произвольно сложные составные представления и повторно использовать их, как если бы они были одним компонентом.

Изменение существующего типа представления

Существует еще более простой вариант для создания пользовательского представления, которое полезно при определенных обстоятельствах. Если есть компонент, который уже очень похож на то, что вы хотите, вы можете просто расширить этот компонент и просто переопределить поведение, которое вы хотите изменить. Вы можете делать все то же самое, что и с полностью настроенным компонентом, но, начав с более специализированного класса в иерархии View, вы также можете получить много поведения бесплатно, которое, вероятно, выполняет именно то, что вы хотите.

Например, приложение NotePad демонстрирует множество аспектов использования платформы Android. Среди них расширение представления EditText для создания выровненного блокнота. Это не идеальный пример, и API для этого могут измениться, но он демонстрирует принципы.

Если вы еще этого не сделали, импортируйте образец NotePad в Android Studio (или просто посмотрите на источник, используя предоставленную ссылку). В частности, обратите внимание на определение LinedEditText в файле NoteEditor.java.

Вот некоторые вещи, которые нужно отметить в этом файле:

1. Определение

Класс определяется следующей строкой:

открытый статический класс LinedEditText расширяет EditText

LinedEditText определяется как внутренний класс в действии NoteEditor, но он является общедоступным, так что при желании к нему можно получить доступ как NoteEditor.LinedEditText из-за пределов класса NoteEditor.

Он является статическим, то есть он не генерирует так называемые «синтетические методы», которые позволяют ему получать доступ к данным из родительского класса, что, в свою очередь, означает, что он действительно ведет себя как отдельный класс, а не как нечто сильно связанное с NoteEditor. Это более чистый способ создания внутренних классов, если они не нуждаются в доступе к состоянию из внешнего класса, сохраняют размер сгенерированного класса и позволяют легко использовать его из других классов.

Он расширяет EditText, который является представлением, которое мы выбрали для настройки в этом случае. Когда мы закончим, новый класс сможет заменить обычное представление EditText.

1. Инициализация класса

Как всегда супер называется первым. Кроме того, это не конструктор по умолчанию, а параметризованный. EditText создается с этими параметрами, когда он надувается из файла макета XML, поэтому наш конструктор должен как взять их, так и передать их конструктору суперкласса.

1. Переопределенные методы

В этом примере переопределяется только один метод, onDraw (), но вам может потребоваться переопределить другие при создании собственных пользовательских компонентов.

Для этого примера переопределение метода onDraw () позволяет нам рисовать синие линии на холсте представления EditText (холст передается в переопределенный метод onDraw ()). Метод super.onDraw () вызывается до его завершения. Метод суперкласса должен быть вызван, и в этом случае мы делаем это в конце после того, как нарисовали линии, которые мы хотим включить.

1. Используйте пользовательский компонент

Теперь у нас есть пользовательский компонент, но как мы можем его использовать?

Пользовательский компонент создается как общий вид в XML, а класс указывается с использованием полного пакета. Также обратите внимание, что на определенный внутренний класс ссылаются с помощью нотации NoteEditor $ LinedEditText, которая является стандартным способом ссылки на внутренние классы в языке программирования Java.

Если ваш пользовательский компонент View не определен как внутренний класс, то вы можете, альтернативно, объявить компонент View с именем элемента XML и исключить атрибут класса.

Обратите внимание, что класс LinedEditText теперь является отдельным файлом класса. Когда класс вложен в класс NoteEditor, этот метод не будет работать.

Другими атрибутами и параметрами в определении являются те, которые передаются в конструктор пользовательских компонентов, а затем передаются в конструктор EditText, поэтому они представляют собой те же параметры, которые вы использовали бы для представления EditText. Обратите внимание, что есть возможность добавить и свои собственные параметры, и мы коснемся этого снова ниже.

И это все, что нужно сделать. По общему признанию, это простой случай, но суть в том, что создание пользовательских компонентов настолько сложно, насколько это необходимо.

Более сложный компонент может переопределять еще больше методов ... и вводить некоторые собственные вспомогательные методы, существенно настраивая его свойства и поведение. Единственным ограничением является ваше воображение и то, что вам нужно сделать компонент.

**Объект Android Activity**

Также стоит немного узнать про контейнер, в котором и будут наши экраны находиться.

Класс Activity является важнейшим компонентом приложения для Android, а способ запуска и объединения действий является фундаментальной частью модели приложения платформы. В отличие от парадигм программирования, в которых приложения запускаются методом main (), система Android инициирует код в экземпляре Activity, вызывая конкретные методы обратного вызова, которые соответствуют определенным этапам его жизненного цикла.

Этот документ знакомит с концепцией действий, а затем предоставляет некоторые простые рекомендации о том, как с ними работать. Для получения дополнительной информации о передовых методах проектирования вашего приложения см. Руководство по архитектуре приложения.

Концепция деятельности

Мобильное приложение отличается от своего настольного аналога тем, что взаимодействие пользователя с приложением не всегда начинается в одном месте. Вместо этого путешествие пользователя часто начинается недетерминированно. Например, если вы откроете приложение электронной почты на главном экране, вы можете увидеть список электронных писем. Напротив, если вы используете приложение для социальных сетей, которое затем запускает ваше почтовое приложение, вы можете перейти непосредственно к экрану почтового приложения для составления электронного письма.

Класс Activity предназначен для облегчения этой парадигмы. Когда одно приложение вызывает другое, вызывающее приложение вызывает действие в другом приложении, а не приложение как единое целое. Таким образом, действие служит точкой входа для взаимодействия приложения с пользователем. Вы реализуете действие как подкласс класса Activity.

Действие предоставляет окно, в котором приложение рисует свой пользовательский интерфейс. Это окно обычно заполняет экран, но может быть меньше экрана и плавать поверх других окон. Как правило, одно действие реализует один экран в приложении. Например, одно из действий приложения может реализовывать экран «Предпочтения», в то время как другое действие реализует экран «Выбор фотографии».

Большинство приложений содержат несколько экранов, что означает, что они содержат несколько действий. Как правило, одно действие в приложении указывается в качестве основного действия, которое является первым экраном, который появляется, когда пользователь запускает приложение. Затем каждое действие может начать другое действие, чтобы выполнить различные действия. Например, основным действием в простом приложении электронной почты может быть экран, на котором отображается почтовый ящик. Оттуда основное действие может запустить другие действия, которые предоставляют экраны для таких задач, как написание электронных писем и открытие отдельных электронных писем.

Хотя действия работают вместе, чтобы сформировать единый пользовательский интерфейс в приложении, каждое действие слабо связано с другими действиями; обычно есть минимальные зависимости между действиями в приложении. Фактически, действия часто запускают действия, принадлежащие другим приложениям. Например, приложение браузера может запустить действие «Поделиться» приложения для социальных сетей.

Чтобы использовать действия в вашем приложении, вы должны зарегистрировать информацию о них в манифесте приложения и надлежащим образом управлять жизненными циклами действий.

Управление жизненным циклом деятельности [7]

В течение своей жизни деятельность проходит через ряд состояний. Вы используете серию обратных вызовов для обработки переходов между состояниями. В следующих разделах представлены эти обратные вызовы.

OnCreate ()

Вы должны реализовать этот обратный вызов, который срабатывает, когда система создает вашу деятельность. Ваша реализация должна инициализировать основные компоненты вашей деятельности: например, ваше приложение должно создавать представления и привязывать данные к спискам здесь. Наиболее важно, что именно здесь вы должны вызвать setContentView (), чтобы определить макет для пользовательского интерфейса действия.

Когда onCreate () завершается, следующим обратным вызовом всегда является onStart ().

OnStart ()

Когда onCreate () завершается, действие переходит в состояние «Запущено», и действие становится видимым для пользователя. Этот обратный вызов содержит то, что составляет окончательную подготовку к выходу на передний план и превращению в интерактив.

onResume ()

Система вызывает этот обратный вызов непосредственно перед тем, как действие начинает взаимодействовать с пользователем. В этот момент активность находится на вершине стека активности и захватывает весь ввод пользователя. Большая часть основных функций приложения реализована в методе onResume ().

Обратный вызов onPause () всегда следует за onResume ().

OnPause ()

Система вызывает onPause (), когда действие теряет фокус и переходит в состояние «Приостановлено». Это состояние возникает, когда, например, пользователь нажимает кнопку «Назад» или «Последние». Когда система вызывает onPause () для вашей активности, это технически означает, что ваша активность все еще частично видна, но чаще всего это указывает на то, что пользователь покидает активность, и действие скоро перейдет в состояние Остановлено или Возобновлено.

Операция в состоянии «Приостановлено» может продолжать обновлять пользовательский интерфейс, если пользователь ожидает обновления пользовательского интерфейса. Примеры такого действия включают в себя показ экрана навигационной карты или воспроизведение медиаплеера. Даже если такие действия теряют фокус, пользователь ожидает, что его пользовательский интерфейс продолжит обновление.

Вы не должны использовать onPause () для сохранения данных приложения или пользователя, сетевых вызовов или выполнения транзакций базы данных. Для получения информации о сохранении данных см. Раздел Сохранение и восстановление состояния активности.

По завершении выполнения onPause () следующим обратным вызовом является onStop () или onResume (), в зависимости от того, что происходит после того, как действие переходит в состояние «Приостановлено».

OnStop ()

Система вызывает onStop (), когда активность больше не видна пользователю. Это может произойти из-за того, что действие уничтожается, запускается новое действие или существующее действие переходит в возобновленное состояние и закрывает остановленное действие. Во всех этих случаях остановленная активность больше не видна вообще.

Следующим обратным вызовом, который вызывает система, является либо onRestart (), если действие возвращается для взаимодействия с пользователем, либо onDestroy (), если это действие полностью завершается.

onRestart ()

Система вызывает этот обратный вызов, когда действие в состоянии Остановлено собирается перезапустить. onRestart () восстанавливает состояние активности с момента ее остановки.

Этот обратный вызов всегда сопровождается onStart ().

OnDestroy ()

Система вызывает этот обратный вызов до того, как действие будет уничтожено.

Этот обратный вызов является последним, который получает действие. onDestroy () обычно реализуется, чтобы гарантировать, что все ресурсы действия высвобождаются, когда действие или процесс, содержащий его, уничтожается.

**Исследование применимости**

**Работа с жизненным циклом**

**Интеграция с различными архитектурами**

**Результаты замены Fragment’ов на Controller’ы**

**Гибкость использования**

**Возможные проблемы**

**Тестирование**

**Обзор методов тестирования**

**Параметры, влияющие на результаты тестов**

**Фреймворки**

**Выбранные фреймворки**

**Проводимые тесты**

**Временные тесты**

**GPU Rendering тесты**

**Профилирование**

**Проблемы тестирования**

**Библиотека Benchmark**

**Библиотека Espresso**

**Количество тестов**

**Заключение**

Выводы тут может сделать только тот, кто будет использовать фреймворк Conductor, так как выводы получились двоякие.

Часть исследований показывает, что это:

- [lifecycle] удобнее в использовании, тк меньше методов жизненного цикла, а соответственно и меньше необходимо следить за правильностью тех или иных данных и состояний

- [testing] удобнее в использовании, тк из-за менее тяжеловесной инициализации (меньше объектов внутри, за методами интерфейса взаимодействия, нужно инициализировать, меньше проверок делать, функций вызывать) быстрее происходит инициализация, что влияет очень эффективно на тестирование (быстрее необходимые объекты появляются для распознавания тестовым фреймворком)

- [performance] чуть более эффективно по производительности (тесты инициализации различных типов экранов нам это демонстрируют)

Но также часть исследований показывает, что:

- [architecture] преимуществ в использовании нет, тк Fragment’ы более интегрированы в среду Android и те инструменты, которые упрощают жизнь разработчикам и есть для фрагментов, отсутствуют для Controller’ов в силу небольшой распространенности исследуемого фреймворка

- [performance] никаких преимуществ в производительности (тесты отрисовки экранов/gpu тесты) фреймворк не даёт, а те тесты, что демонстрируют это, отражают лишь незначительное различие

Исходя из проделанной работы и полученных результатов я готов лишь сделать вывод, что данная технология крайне интересна и может быть как минимум равновесной альтернативой текущим подходам. И мои выводы подтверждает обзор рынка мобильных приложений, который демонстрирует, что Conductor хоть и изредка, но используется в production решениях, и о ней откликаются положительно как рядовые разработчики, так и лидеры индустрии.

Весь код и ресурсы можно посмотреть тут:

<https://github.com/GorniyGor/iosched-conductor>

Процесс перевода с Fragment на Conductor’s Controller вы можете увидеть тут:

<https://github.com/GorniyGor/iosched-conductor/pull/1>

Код тестов вы можете увидеть

тут для версии переписанной на Conductor:

<https://github.com/GorniyGor/iosched-conductor/pull/3>

и тут для изначальноей версии:

<https://github.com/GorniyGor/iosched-conductor/pull/2>

Различий в них нет, две версии были сделаны лишь для удобства запуска на соответствующих версиях (Fragment/Conductor) приложения.

**Список литературы**

<https://github.com/google/iosched>

<https://developer.android.com/guide/components/fragments>

<https://android.jlelse.eu/exploring-conductor-android-app-development-without-fragments-part-1-fce1eab0c9df>

<https://github.com/bluelinelabs/Conductor>

<https://developer.android.com/reference/android/view/View#startAnimation(android.view.animation.Animation)>

<https://developer.android.com/guide/topics/ui/custom-components>

<https://developer.android.com/guide/components/activities/intro-activities>

**Приложение**

