УДК 004

ОСОБЕННОСТИ ПРИМЕНЕНИЯ ТЕХНОЛОГИИ REACTIVEX В МОБИЛЬНЫХ ПРИЛОЖЕНИЯХ ДЛЯ ОПЕРАЦИОННОЙ СИСТЕМЫ ANDROID

© 2019 А. В. Винокуров

Воронежский институт высоких технологий (г. Воронеж, Россия)

В статье рассмотрены причины важности ReactiveX (в виде её реализации для языка Java – фреймворка RxJava) в качестве инструмента разработки асинхронного кода для Android, возможные альтернативы, описаны распространённые проблемы использования ReactiveX в приложениях Android и их практические решения.

Ключевые слова: Android, ReactiveX, RxJava, реактивное программирование, шаблоны проектирования, потоки данных.

Любой разработчик программного обеспечения, который когда-либо писал приложение для мобильных устройств, знает о важности раздельного выполнения в таком приложении операций с данными и операций пользовательского интерфейса. Как минимум, потому что выполнение определённых видов действий приложения, связанных с сетевой активностью, обращением к базе данных и т. п., в потоке пользовательского интерфейса запрещено спецификацией операционной системы и попытка их выполнить в этом потоке вызовет ошибку компиляции.

Традиционными средствами для обработки фоновых операций в Android SDK являются:

- обработчики (Handlers);
- сервисы (Services);
- загрузчики (Loaders);
- асинхронные задания (AsyncTasks).

При этом обработчики и сервисы для фоновых операций с данными использовать в их «чистом» виде нецелесообразно из-за следующих недостатков:

- синхронизированный с потоком UI блокирующий интерфейс;
- отсутствие встроенной реализации остановки выполнения потока;
- отсутствие встроенной реализации для обработки изменений в конфигурации системы;
- отсутствие встроенной реализации обработки нескольких потоков.

Сервисы, к тому же, являются системными объектами, имеющими собственный контекст, в силу чего могут передавать информацию в пользовательский интерфейс только через отправку намерений (Intent), что накладывает ограничения по составу передаваемых данных и возможностям интерфейса по их отображению.

Загрузчики и асинхронные задания, несмотря на то, что поддерживают асинхронное по отношению к пользовательскому интерфейсу выполнение кода, обладают целым рядом недостатков программного интерфейса, делающих их использование крайне нежелательным и трудоёмким для разработчиков. Так, например, использование загрузчиков требует реализации в приложении отдельного класса, унаследованного от абстрактного класса Loader или его потомков (CursorLoader, AsyncTaskLoader), а также отдельной реализации класса LoaderManager и реализации интерфейсов обратного вызова LoaderManager. LoaderCallbacks. Использование загрузчиков в Android неудобно настолько, что многие разработчики предпочитают избегать их применения везде, где это возможно.

AsyncTask – наиболее оптимальный из стандартных инструментов выполнения асинхронного кода в Android. Всё, чего требует его реализация в приложении, - это отдельный (как правило, вложенный) класс с generic-интерфейсом, унаследованный от AsyncTask, для каждой отдельной фоновой операции.

Однако для современных условий разработки, когда одно мобильное приложение должно управлять десятками, если не сотнями, фоновых действий, демонстрируя интерактивный пользовательский интерфейс, когда код,

Винокуров А. В. – Воронежский институт высоких технологий, магистр.

позволяющий это делать, должен быть создан в сжатые сроки, все эти решения не годятся. Не будет преувеличением сказать, что в инструментах Android SDK используемых ими объектно-ориентированных принципах ощущается острая нехватка нового уровня абстракции – представления действий приложения в качестве потоков данных, а не отдельных объектов и их свойств/методов.

Добавляется к этому и то обстоятельство, что встроенная седьмая версия языка Java в Android не обновляется уже много лет, хотя в сентябре 2018 года вышла уже версия Java 11. Таким образом, все вышеуказанные классы Android SDK вынуждены использовать устаревшие шаблоны и инструменты из пакета java.util.concurrent.* для управления многопоточностью в приложении. Через подключение сторонних библиотек в приложение можно добавить какую-то функциональность из Java 8 (например, лямбда-выражения), но общей картины это не меняет.

Перечисленные обстоятельства обусловили рост популярности в современной Android-разработке, во-первых, парадигмы реактивного программирования, как таковой, а, во-вторых, основанной на этой парадигме мультиплатформенной технологии ReactiveX от компании Netflix, подключаемой в виде локализованных для конкретного языка программирования или платформы библиотек (RxJava, RxAndroid, RxKotlin, RxScala и др.).

Чтобы осветить вопрос всесторонне, сделаем небольшое отступление и поясним, что же собой представляет реактивное программирование, и в чём отличия данного подхода от традиционных императивных подходов, таких как объектно-ориентированное или процедурное программирование.

Первопроходцем, давшим определение реактивному подходу в программировании (применительно к созданию интерактивных анимаций) стал Конал Эллиот из Microsoft Graphics Research Group:

«Если программирование необходимо для создания интерактивного контента, но привычный способ создания такого контента сегодня не подходит для большинства потенциальных авторов, тогда мы нуждаемся в переходе к другой форме программирования. Эта форма должна предоставлять автору полную свободу выражения того, что такое анимация, пока будут невидимо обрабатываться детали дискретного, последовательного представления. Иными словами, эта форма должна быть более *декларативной* («Что происходит?») нежели *императивной* («Как сделать?»).

В «Реактивном манифесте», предложенном инициативной группой Typesafe Inc. и подписанном в настоящее время более чем 24 тыс. разработчиками, сформулированы следующие принципы реактивных систем:

- событийно-ориентированность;
- масштабируемость;
- отказоустойчивость;
- отзывчивость.

Или совсем коротко: реактивное программирование – это программирование с асинхронными потоками данных.

Применительно к мобильному приложению это означает, прежде всего:

- использование асинхронных неблокирующих моделей обработки событий;
- обработку ошибок на уровне потоков событий;
- архитектуру приложения, разделяющую пользовательский интерфейс и работу с данными (паттерны MV*: Model-View-ViewModel, Model-View-Presenter, Model-View-Controller).

Простое и понятное воплощение эти положения нашли в функционале библиотек RxJava и RxAndroid. Рассмотрим их по порядку:

1) Асинхронная неблокирующая модель обработки событий

Весь программный интерфейс библиотек ReactiveX построен вокруг шаблона проектирования «Наблюдатель» (и родственного ему «Издатель-Подписчик»).

Наблюдаемый объект, выступающий генератором событий, реализует один из Observable-интерфейсов библиотеки RxJava (как правило, просто создаётся анонимный класс на основе стандартного объекта Callable или Runnable), а объект-наблюдатель на него подписывается и обрабатывает до 4 возможных событий: onSubscribe – событие возникновения подписки, onNext - generic-событие, возвращающее наблюдателю объект заданного типа (например, результат вызова метода call объекта Callable), onComplete – метод, сообщающий наблюдателю о том, что выполнение завершено успешно (например,

метода run объекта Runnable), onError – метод, сообщающий, что выполнение завершилось с ошибкой и возвращающий объект Throwable.

Таким образом, наблюдаемый объект выполняет свой код асинхронно по отношению к наблюдателю, который реагирует только на наступление отслеживаемых событий. С помощью методов subscribeOn и observeOn из библиотеки RxAndroid код наблюдателя передаётся для выполнения в фоновый поток ввода-вывода, а результат возвращается в поток UI, что позволяет полностью избавиться от нежелательных блокировок пользовательского интерфейса.

2) Обработка ошибок на уровне потока событий

Все незапланированные ошибки в работе наблюдателя централизованно принимаются методом onError.

3) Совместимость с архитектурными паттернами MV*

Наиболее совместимым и естественным для использования с библиотеками RxJava и RxAndroid является паттерн Model-View-Presenter (Модель – Вид - Представитель), в котором на уровне Модели находятся наблюдаемые объекты (или их фабрики), на уровне Вида находятся системные объекты, такие как активности и фрагменты, которые отвечают исключительно за построение пользовательского интерфейса и предоставляют методы для управления им, а на уровне Представителя осуществляется реализация бизнес-логики приложения путём обработки подписок на наблюдаемые объекты.

Тем не менее, несмотря на простоту и надёжность по сравнению со средствами Android SDK, использование библиотек ReactiveX в Android связано с рядом трудностей, которые потребуют от программиста аккуратности и внимания. Полагаем, что в общем виде такие проблемы можно разделить на три группы:

1) Проблемы взаимодействия с жизненным циклом приложения;

2) Проблемы, вызванные дефектами архитектуры;

3) Проблемы использования сторонних библиотек.

Наиболее значимыми и болезненными являются проблемы первой группы.

Самой типичной ошибкой, приводящей к массовым утечкам контекста в приложении и негативно сказывающейся на производительности, является отсутствие отписок от наблюдаемых объектов при вызове методов onDestroy у активности или onDestroyView у фрагмента. Под утечкой контекста следует понимать хранение приложением уничтоженной активности или фрагмента в памяти устройства, поскольку сборщик мусора виртуальной машины не может освободить память, занятую объектами, на которые существуют ссылки. Типовым решением для данной проблемы является добавление в интерфейс Представителя метода, который при каждом вызове событий onDestroy и onDestroyView будет вызывать метод unsubscribe у созданных подписок.

При этом следует помнить, что обработка каждой подписки в отдельности потребует заведения в классе Представителя отдельного поля для каждого наблюдаемого объекта, поэтому при использовании RxJava 2 целесообразно хранить все подписки с помощью объекта CompositeDisposable, у которого вызывать метод clear или dispose при уничтожении активности или фрагмента.

Второй проблемой, связанной с жизненным циклом приложения Android, является определение правильного метода жизненного цикла для возобновления подписки. Как правило, подписка, создаваемая в onCreate должна быть прекращена при вызове onDestroy, но если для целей конкретного приложения необходимо получение данных в onStart или onResume, то прекращение такой подписки в onDestroy также может привести к дефектам функциональности и утечкам памяти. Поэтому правильной рекомендацией будет уничтожать подписку в том деинициализирующем методе жизненного цикла, который корреспондирует инициализирующему методу, в котором подписка была создана.

Значительно более редкой и столь же более болезненной является проблема сохранения прогресса длительной фоновой операции в жизненном цикле приложения. Поскольку инициализирование объектов, отвечающих за RxJava-подписки происходит в жизненном цикле активности или фрагмента, то пересоздание активности или фрагмента приводит к пересозданию этих объектов. Таким образом, прекращение соответствующей подписки в деинициализирующем методе жизненного цикла лишь предотвращает утечку памяти, но не избавляет нас от необходимости

повторного запуска прерванной операции при возобновлении подписки.

Возможных выходов здесь может быть несколько:

1) Обработка подписок в статическом синглтоне или объекте класса Application. Быстрое, но плохое решение, которое чревато утечками памяти.

2) Выполнение длительной операции в Сервисе. Сервис обладает собственным контекстом и жизненным циклом, поэтому вполне способен выполнить длительную фоновую операцию независимо от поворотов экрана. Недостатком данного подхода является необходимость заводить отдельный сервис для каждой подобной операции и громоздкость реализации в целом.

3) Запрет смены ориентации экрана при выполнении длительных фоновых операций. Действительно, если программно запретить смену ориентации, то при повороте экрана метод onDestroy не будет вызван, и можно не опасаться пересоздания подписки из-за неосторожного движения пользователя. Это кажется простым и естественным решением, но на самом деле, во-первых, негативно влияет на перфоманс (спецификация Material Design требует обработки поворотов экрана), а, во-вторых, поворот экрана – распространённый, но не единственный случай, когда у активности может быть вызван метод жизненного цикла onDestroy. Фактически, программный запрет на обработку поворота экрана – это попытка скрыть проблему, а не решить её.

4) Использование ConnectableObserver в связке с Loader. Непростой по реализации, однако, как представляется, наиболее корректный способ сохранения состояния длительных фоновых процессов, вроде загрузки файла, при использовании библиотек RxJava и RxAndroid. Особенностью ConnectableObserver является управляемое получение наблюдателем наблюдаемых данных: возможность получить данные отложенно, а не непосредственно после создания подписки, возможность получения подписчиком данных, сгенерированных до момента подписки, и иные полезные функции. В свою очередь, Loader позволяет сохранить состояние ConnectableObserver при вызове метода onDestroy у соответствующей активности, и возобновить фоновую операцию непосредственно с того момента, на котором она была прервана. При этом не потребуется реализации нескольких загрузчиков и, в целом, код получится компактнее, чем при использовании средств Android SDK.

Проблемы использования RxJava могут быть связаны с недостатками проектирования конкретного приложения. Например, обработка действий пользователя с объектами, использующими адаптеры, такими как ListView, RecyclerView и GridView, может потребовать совершения каких-либо фоновых операций на уровне Модели (допустим, сохранения в локальную базу данных выбранного товара). Пока обработка действий пользователя с элементами списка ведётся в классе активности или фрагмента с помощью, например, AdapterView.OnItemClickListener проблем не возникнет. Однако если элемент списка содержит сразу несколько кнопок или иных элементов, использующих обработчики кликов, касаний и иных событий, связанных с экранным фокусом, то код этих обработчиков должен быть прописан непосредственно в классе адаптера, который ничего не знает ни о Представителе, ни о Модели. Хранение в адаптере ссылки на класс-представитель активности технически возможно, но чревато нарушением архитектурного шаблона из-за размывания сфер ответственности классов. Наиболее распространённым выходом в такой ситуации является использование PublishSubject – класса из библиотеки RxJava, обладающего свойствами как наблюдателя так и наблюдаемого, и упрощённым интерфейсом подписки. Таким образом, активность или фрагмент подписывается на получение от адаптера данных пользовательского ввода и в дальнейшем передаёт эти данные в обработку Представителю.

Проблемы при использовании сторонних библиотек могут быть вызваны применением в таких библиотеках собственных технологий асинхронного выполнения. Например, нецелесообразным представляется использование технологий ReactiveX вместе с классами библиотеки AndroidX от Google, такими как LiveData, поскольку это альтернативные технологии.

Несмотря на то, что LiveData обеспечивает независимость существования подписки от жизненного цикла активности или фрагмента, это преимущество, по нашему мнению, не окупает усложнения кода и добавления в проект избыточной

зависимости

В заключении стоит отметить, что существенным достоинством ReactiveX является возможность операций непосредственно с реактивными потоками событий, например, их фильтрация перед подачей в метод обратного вызова объекта-наблюдателя. Существующие альтернативные технологии асинхронного выполнения кода в Android, такие как Kotlin Coroutines и AndroidX, таким функционалом не обладают, хотя в остальном вполне могут выступать в качестве достойной альтернативы. Поэтому, на наш взгляд, будущее фреймворка Android за ориентированностью на парадигму реактивного программирования и интеграцией в SDK если не самих библиотек ReactiveX (хотя для этого уже создан прецедент в виде встроенной поддержки RxJava в Room Persistence Library, входящую в состав Android Framework), то их функционала.

ЛИТЕРАТУРА

1. Elliott, C. Composing Reactive Animations // Microsoft Research Graphics Group. 1998. URL: http://conal.net/fran/tutorial.htm (дата обращения 08.01.2019).

2. The Reactive Manifesto (v. 2.0). 2014. URL: https://www.reactivemanifesto.org (дата обращения 08.01.2019).

3. Andre Medeiros The introduction to reactive programming you've been missing. 2014. URL: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754 (дата обращения 08.01.2019).

4. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес; [пер. с англ.: А. Слинкин науч. ред.: Н. Шалаев]. – Санкт-Петербург [и др.] : Питер, 2014. – С. 280 – 290,

5. Эккель, Б. Философия Java. 4-е полное изд. – СПб.: Питер, 2017. – С. 78.

FEATURES OF USING THE REACTIVEX TECHNOLOGY IN ANDROID APPLICATIONS

© 2019 A. V. Vinokurov

Voronezh Institute of High Technologies (Voronezh, Russia)

The article observes reasons of using ReactiveX (with it's implementation in Java language – RxJava framework) as a tool for asynchronous programming for Android, possible alternatives, describes common problems with using ReactiveX in Android and their practical solutions.

Key words: Android, ReactiveX, RxJava, reactive programming, design patterns, data streams.