электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

 # 04, апрель 2016

УДК 004.424

Разработка мобильных приложений под операционные системы Android и iOS с использованием реактивных расширений

Педченко М. О., *студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»

Чамеев Н. Л., *студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»

Научный руководитель: Гапанюк Ю. Е., к.т.н, доцент
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»
gapyu@bmstu.ru

Введение

В последнее время в среде мобильной разработки очень стремительно набирает популярность создание приложений с помощью функционального реактивного подхода, используя Reactive Extensions. Это библиотека, разработанная компанией Microsoft, позволяющая работать с асинхронными вызовами и событиями, объединяя их с помощью различных операторов, похожих на операторы запросов в LINQ. Для работы с Reactive Extensions на мобильных платформах, используются реализации RxJava (от компании Netflix) и RxSwift для Android и iOS соответственно.

В статье рассмотрена общая концепция реактивного функционального программирования и практические примеры использования RxJava и RxSwift при разработке нативных мобильных приложения под платформы Android и iOS.

Основные компоненты

Самыми базовыми компонентами в Rx, и в частности в RxJava, являются и Observer (Subscriber). С помощью этих компонентов реализуется шаблон Observable проектирования «наблюдатель» (англ. Observer). Observable является источником данных и содержит методы для управления наблюдателями (Observer), а Observer является

потребителем данных и его интерфейс имеет методы для получения оповещений от Observable.

Интерфейс Observer состоит из трех методов: onCompleted, onError и onNext. После того, как Observable порождает данные, для каждого Observer, подписанного на данный Observable будет вызван метод onNext на каждый из элементов потока данных. После этого возможен вызов методов onCompleted (при успешном завершении) или onError (если возникли какие-либо ошибки). Два этих дополнительных метода (onError и onCompleted) позволяют легко и удобно обрабатывать ошибки и окончание потока данных.

Главная идея

Rx - отличный инструмент, позволяющий главным образом упростить сложные вычисления и асинхронные части в вашем коде. Reactive происходит от слова react (реагировать), подразумевается, что система реагирует на изменения состояния. В процессе развития программного обеспечения, система становится более сложной, появляются дополнительные модули и зависимости. Возникает потребность, в реагировании на множество источников данных, а также в устойчивости данной системы.

Как правило, мы пишем код, в котором есть методы и функции. Мы вызываем их, получаем на выходе результат, обрабатываем его и используем дальше. Но некоторые процессы, такие как работа с сетью, устроены по-другому. В таких случаях методы не выполняются сразу же, они могут занимать некоторый продолжительный промежуток времени. Бывает так, что запрос не только выполняется долго, но еще и требует постепенную обработку промежуточных значений. Конечно в данном случае можно вызвать метод одного объекта, внутри которого вызвать метод еще одного объекта и передать туда данные, и так до бесконечности. Но есть и другое, более оптимальное решение проблемы - реактивные расширения.

Представьте, что у вас есть процесс, работающий долго. Также, во время выполнения он возвращает промежуточные результаты. Reactive Extensions в данном случае идеально подходят для контроля этого процесса. Rx позволяет нам создать событие и обработчиков, которые будут реагировать на них. Соответственно, Rx будут обрабатывать результат, каждый раз когда он приходит, независимо от того, промежуточный он или нет.

Реактивные расширения берут свое начало от шаблона проектирования «Observer», в котором основными сущностями являются «объект» и «наблюдатели». Каждый раз, когда «объект» меняет свое состояние, он сообщается об этом «наблюдателям». Rx

дополняет этот паттерн, внося в концепцию программирования идею потока данных, что в свою очередь позволяет детализировать поток данных. Например паттерн «Observer» обычно описывает взаимодействие целых объектов или классов, когда реактивное программирование позволяет нацелиться также и на поля класса.

Как это работает

Для визуальной демонстрации асинхронных потоков данных в реактивном программировании, обычно используются, так называемые, шариковые диаграммы (marble diagrams). Представим, что мы написали код, который выполняет асинхронный запрос на сервер. Удобно расположить все происходящее, как упорядоченную последовательность событий. Например, вначале сервер отправляет ответ об установленном соединении, затем начинает отправку данных, отвечает на запрос, и закрывает соединение. Все это происходит в порядке очереди, но асинхронно, и невозможно точно предсказать, как долго будет выполняться та или иная операция. В контексте Rx, мы устанавливаем Observable и Observer. Запрос как Observable, некий Handler, как Observer. Каждый раз когда с сервера будут приходить данных, Observer соответственно, получает их. Представим это в виде таймлайна (см. рис. 1).

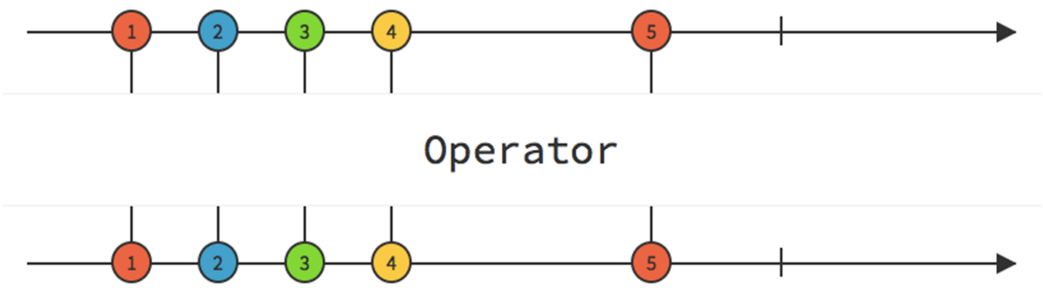

Рис. 1. Общий вид шариковой диаграммы для оператора

На диаграмме, ось X (слева направо) представляет собой время. Каждая горизонтальная линия на оси Y представляет с собой подписку на исходный поток данных, которая была инициирована либо самим пользователем, либо оператором. Далее на этих горизонтальных линиях, с помощью кругов (шариков) обозначаются сами данные, которые в данный момент времени были созданы (вызов onNext). Конец этой линии означает конец подписки, который может быть двух видов: прямой вертикальной чертой помечается отписка без ошибок (вызов onCompleted) и крестом обозначается отписка из-

за исключительной ситуации (вызов onError). Вертикальные линии, идущие от более высокой линии к более низкой представляют данные, которые отображаются соответственно с верхнего потока к нижнему. При помощи блока оператора, мы можем задать преобразование данных на пути к Observer.

Observable

Как уже ранее было отмечено, Observable является источником данных. Он представляет объект, который посылает уведомления, подобно пуш-уведомлениям (push-notification). Из особенностей стоит отметить, что существует два типа Observable – «холодные (cold)» и «горячие (hot)». «Холодный» Observable не посылает никаких уведомлений пока на него не подпишется Observer. Таким образом гарантируя, что последовательность данных будет отправлена целиком с самого начала. «Горячий» Observable, в свою очередь, может начать посылать уведомления в момент создания, следовательно, Observer, подписавшись в середине процесса, не получит данные с начала.

Observer

В Rx на Observable подписывается Observer (Subscriber), тем самым реагируя на все сообщения, которые посылает Observable. У Observer есть три метода:

- **`onNext`** - вызывается каждый раз, когда Observable посылает уведомление. Принимает как параметр объект, отправленный Observable;
- **`onCompleted`** - вызывается после финального вызоыва **`onNext`**, при отсутствии ошибок;
- **`onError`** - вызывается в тот момент, когда произошла ошибка. Останавливает процесс Observable.

Scheduler

Очень часто перед нами стоит задача использования многопоточности для того, чтобы не забивать главный поток (main thread), который обычно отвечает за интерфейс (UI). Чтобы пользователь не замечал задержек, необходимо выполнять асинхронные операции в отдельном потоке. В Rx для этого существует Scheduler. Сам по себе Scheduler представляет уровень абстракции с очередью операций и потоком, на котором они будут выполняться.

По умолчанию, в Rx Observable и вся цепочка его операторов будет функционировать на том потоке, на котором метод **`Subscribe`** вызывался.

Методы Observable для работы с многопоточностью:

- `subscribeOn` позволяет поменять Scheduler, на котором Observable работает;
- `observeOn` позволяет поменять Scheduler, на котором Observable посылает уведомления.

В общем случае механизм шедулинга выглядит так:

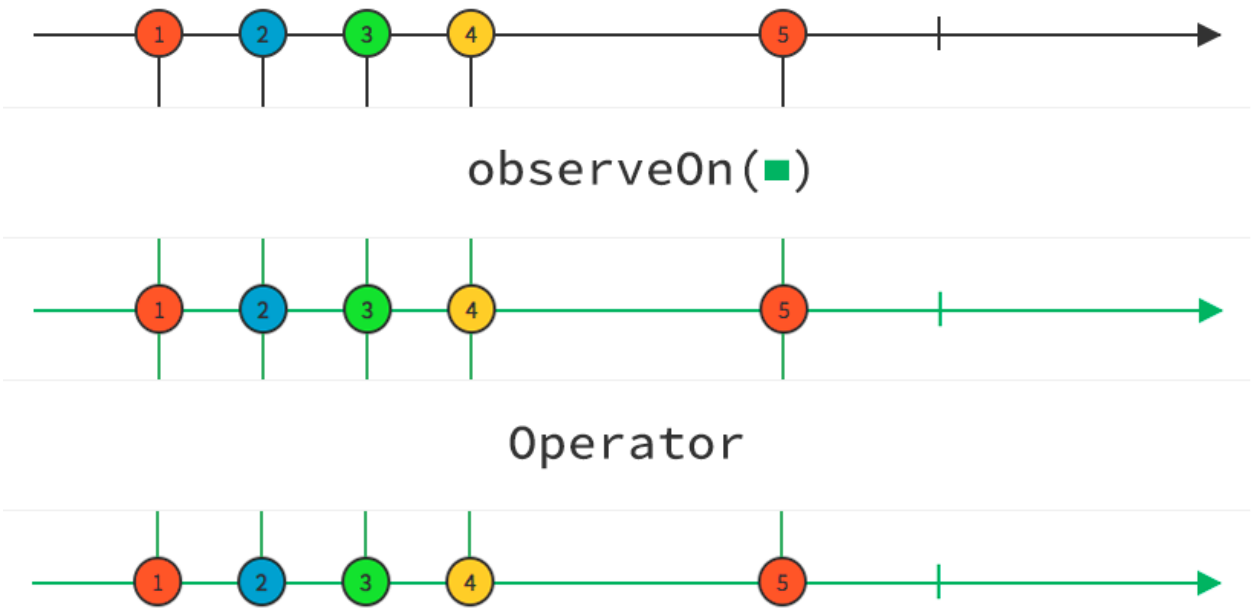

Рис. 2. Диаграмма работы оператора observeOn

Пример использования Reactive Extensions в iOS

Рассмотри использование Reactive Extensions в операционной системе iOS на простом примере. В качестве примера будет выступать приложение, которое выполняет запрос на сервер API Яндекс Предиктора и подсказывает наиболее вероятное продолжение слов или фраз, набираемых пользователем.

Нативная разработка под iOS поддерживается двумя языками программирования: Objective-C и Swift. Реализация паттерна Rx также доступна в двух вариантах. В первом случае это ReactiveCocoa, во втором - RxSwift. Поскольку Swift является более новым языком программирования, недавно выпущенным компанией Apple, было решено остановиться на нем.

Перед началом разработки, требуется подключить к проекту необходимые библиотеки. Рекомендуется использовать Xcode. Загрузку зависимостей из удаленных репозиториев будем осуществлять при помощи CocoaPods.

В корневой директории проекта необходимо создать файл с названием Podfile.

Добавим зависимости:

```
# Podfile
use_frameworks!

pod 'RxSwift', '~> 2.0.0-beta'
pod 'RxCocoa', '~> 2.0.0-beta'
pod 'RxBlocking', '~> 2.0.0-beta'
```

Поскольку мы используем Swift, необходимо явно указать в Podfile `use_frameworks!`. В противном случае мы не сможем использовать данную библиотеку в нашем проекте. Swift CocoaPod требует, чтобы мы явно писали данную директиву.

1. RxSwift

Соотвественно ядро для работы с паттерном Rx. Предоставляет интерфейс Observable, Observer, Scheduler, и операторов. А также координирует их совместную работу.

2. RxCocoa

Данная библиотека предоставляет нам набор расширений для стандартных классов Cocoa и CocoaTouch. Без данной библиотеки экземпляры стандартных классов, такие как кнопка и т.д, не смогут быть Observer и Observable.

3. RxBlocking

Данная библиотека представляет собой набор так называемых блочных операторов. Блочные операторы используются при написании unit тестов.

4. Alamofire

Alamofire – это обертка над стандартными классами для работы с сетью. Предоставляет нам удобный интерфейс для работы с REST. А также инкапсулирует некоторую дополнительную логику, такую как кэширования, что позволяет упростить написание сетевых запросов.

После этого становятся доступны все необходимые классы и можно приступать к написании программного кода.

Создадим один ViewController. В данном приложении будет всего лишь один экран, состоящий из строки ввода данных и списка предлагаемых продолжений текста, введенного в эту строку.

Создадим верстку этого экрана. В iOS элементы пользовательского интерфейса можно создавать двумя способами: с помощью Interface Builder и с помощью кода. Добавим на экран поле для ввода и список для отображения материалов в Interface Builder.

В iOS поле для ввода является объектом класса UITextField. Благодаря подключению библиотеки RxCocoa, у экземпляров данного класса повляется поле rx_text, которое как раз и позволяет нам пользоваться реактивными расширениями.

Поле rx_text является Observable, который может создавать события каждый раз, когда пользователь изменяет текст. Измененный текст получат все подписчики, которые были подписаны на данный Observable. Чтобы подписаться, нужно вызывать у Observable метод Subscribe и передать туда экземпляр объекта Observer:

```
searchTextField.rx_text
        .subscribeNext { results in
                // bind to ui
        }
```

После получения текста из поля ввода, выполним запрос на сервера Яндекса для получения списка продолжений текста. Чтобы это сделать, воспользуемся оператором switchMap. Данный оператор принимает как аргумент функцию, которая возвращает еще один Observable. SwitchMap позволяет заменить текущий поток данных на поток данных из другого Observable, который возвращается из результата функции, переданной в аргументы. В отличие от flatMap, этот оператор выполняет отписку от предыдущего Observable, если во время этого был получен новый элемент из исходного потока.

Как аргумент оператора передадим функцию, которая возвращает Observable от списка строк, созданную с помощью библиотеки Alamofire. Получаем такую последовательность операторов:

```
searchTextField.rx_text
        .switchMap(query          ->          Api.getInstance().complete(query))
        .subscribeNext(Subscriber())
```

Попробуем запустить этот пример. Текущая реализация содержит в себе несколько проблем. Рассмотрим их поподробнее и попытаемся решить.

Во-первых, при быстром наборе текста будет создаваться множество частых событий, что в итоге приводит к большому числу запросов на сервер. В данном случае пользователю не требуется получать информацию с сервера во время того, как он набирает текст в поле, поэтому такое создание запросов на каждый символ не оправдано. Это можно решить с помощью оператора throttle. Данный оператор пропускает данные, которые не вошли в некоторый временной интервал и по окончании этого интервала, если не пришли новые данные, возвращает последние пришедшие. После применения этого оператора, при печати текста в поле ввода, запрос не будет отправляться на сервер, пока пользователь не закончит печать:

```
searchTextField.rx_text
        .throttle(350)
```

```
        .switchMap(query          ->          Api.getInstance().complete(query))
        .subscribeNext(Subscriber())
```

Во-вторых, данный код будет выполняться в основном потоке. В iOS главный поток используется для отрисовки пользовательского интерфейса. Если в нем выполнять задачи, которые имеют продолжительность больше, чем время отрисовки одного кадра на экране, то пользователь заметит эти задержки: весь пользовательский интерфейс не будет реагировать пока эта задача не закончится.

В-третьих, так как все операции выполняются в потоке, отличном от главного, в методах-обработчиках подписчика, код также будет выполняться не в основном потоке. В iOS крайне не рекомендуется изменять элементы пользовательского интерфейса не в главном потоке, поэтому при попытке сделать это возникнет исключение и приложение аварийно завершит свою работу. Эту проблему также легко решить одной строкой кода: можно с помощью метода observeOn() указать шедулер, где будет выполняться код подписчика. Чтобы выполнить этот код в основном потоке, нужно указать шедулер:

В итоге получаем такую цепочку:

```
searchTextField.rx_text
        .throttle(350)
        .switchMap(query -> Api.getInstance().complete(query))
        .subcribeOn(OperationQueueScheduler)
        .observeOn(MainScheduler.sharedInstance)
        .subscribeNext(Subscriber())
```

Пример использования Reactive Extensions в Android

Рассмотрим использование Reactive Extensions в операционной системе Android на простом примере. В качестве примера будет выступать приложение, которое выполняет запрос на сервер API Яндекс Предиктора и подсказывает наиболее вероятное продолжение слов или фраз, набираемых пользователем.

Под операционную систему Android в основном разрабатывают на языках Java и C\C++. При использовании Java, для разработчика доступны практически все API Android. Поэтому, большинство приложений написано именно на этом языке программирования. В данном примере мы будем использовать реализацию Reactive Extensions под Java - RxJava, которая была разработана компанией Netflix.

Перед началом разработки, требуется подключить к проекту необходимые библиотеки. При использовании среды разработки Android Studio, сборка проекта выполняется с помощью системы автоматической сборки Gradle. Эта система поддерживает загрузку зависимостей из удаленных репозиториев Maven и по-умолчанию, без указания дополнительных адресов репозиториев, возможно нахождение зависимостей

в репозитории Maven Central. Все необходимые для работы примера библиотеки уже имеются в этом репозитории, поэтому, чтобы их подключить добавим следующие зависимости в директиву “dependencies” файла build.gradle модуля проекта:

1. **io.reactivex:rxjava**

Данная зависимость, собственно, добавляет весь функционал Reactive Extensions. Она содержит все основные классы RxJava (включая Observable, Observer, операторы и т.д.).

2. **io.reactivex:rxandroid**

Эта зависимость содержит набор специфичных для Android классов, которые могут потребоваться при использовании RxJava в Android (конкретнее, она предоставляет планировщики (schedulers), позволяющие выполнять программный код в главной потоке или на конкретном handler’e).

3. **com.jakewharton.rxbinding:rxbinding**

Эта библиотека предоставляет обертки над стандартными пользовательскими элементами управления операционной системы Android, которые создают соответствующие события в RxJava.

5. **com.squareup.retrofit**

Данная библиотека позволяет описывать какое-либо API, сделанное по идеологии REST с помощью интерфейса и аннотаций в Java. В конечном итоге, разработчик получает Java методы для работы с серверным API. Кроме всего прочего, эта библиотека позволяет создавать на основе API методов объекты Observable, на которые сразу можно подписаться.

После этого становятся доступны все необходимые классы и можно приступать к написании программного кода.

Создадим одно главное Activity (в Android Activity в большинстве случаев представляет с собой один экран, с которым пользователь может взаимодействовать) и назовем его MainActivity. В данном приложении будет всего лишь один экран, состоящий из строки ввода данных и списка предлагаемых продолжений текста, введенного в эту строку.

Создадим верстку этого экрана. В Android элементы пользовательского интерфейса можно создавать двумя способами: с помощью XML-файлов и создавая дерево элементов вручную из программного кода. В первом случае создание пользовательского интерфейса

очень похоже на создание обыкновенной веб-страницы на HTML. Добавим поле для ввода текста, список и контейнер для этих двух элементов в XML файл макета.

Статический метод textChanges у класса RxTextView принимает как аргумент EditText и возвращает объект Observable, который может создавать события каждый раз, когда пользователь изменяет текст. Измененный текст получат все подписчики, которые были подписаны на данный Observable. Чтобы подписаться, нужно вызывать у Observable метод Subscribe и передать туда экземпляр объекта Observer:

```
RxTextView.textChanges(input).subscribe(new InputSubscriber());
```

Поскольку в Java работать со строками чаще удобнее в виде типа String, а EditText возвращает текст в виде CharSequence, преобразуем этот тип в String с помощью оператора Map.

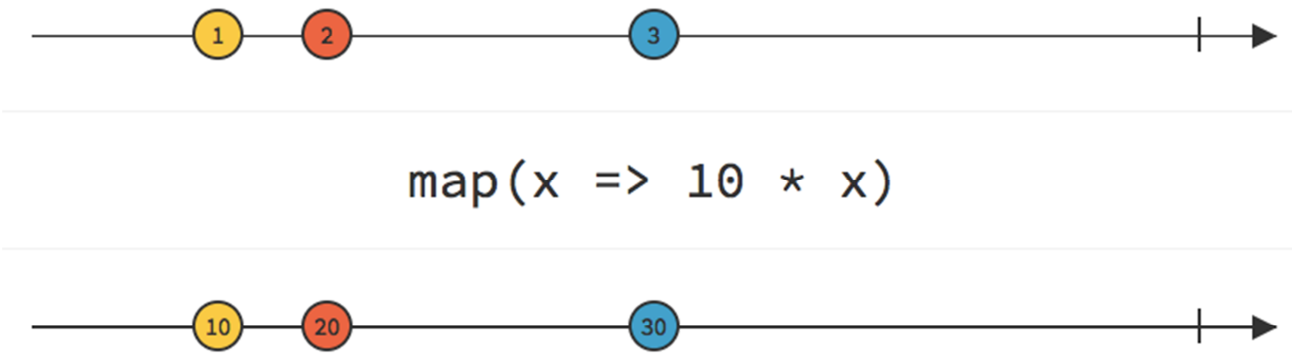

Рис. 3. Диаграмма работы оператора map.

Рассмотрим диаграмму оператора Map:

Этот оператор преобразует одни входные данные в другие с помощью функции, которая передается как аргумент в этот оператор. Чтобы применить этот оператор, добавим его в цепочку после создания объекта Observable:

```
RxTextView
                .textChanges(input)
                .map(CharSequence::toString)
                .subscribe(new InputSubscriber());
```

В итоге, на вход InputSubscriber теперь будут приходить не объекты типа CharSequence, а объекты типа String.

После получения текста из поля ввода, выполним запрос на сервера Яндекса для получения списка продолжений текста. Чтобы это сделать, воспользуемся оператором switchMap. Данный оператор принимает как аргумент функцию, которая возвращает еще один Observable. SwitchMap позволяет заменить текущий поток данных на поток данных

из другого Observable, который возвращается из результата функции, переданной в аргументы. В отличие от flatMap, этот оператор выполняет отписку от предыдущего Observable, если во время этого был получен новый элемент из исходного потока.

Как аргумент оператора передадим функцию, которая возвращает Observable от списка строк, созданную с помощью библиотеки Retrofit. Для этого отдельно было описано API с помощью интерфейса и аннотаций. После этого Retrofit создает реализацию данного интерфейса, которую можно использовать для создания запросов в сеть.

Получаем такую последовательность операторов:

```
RxTextView
                .textChanges(input)
                .map(CharSequence::toString)
                .switchMap(query -> Api.getInstance().complete(query))
                .subscribe(new InputSubscriber());
```

Рассмотрим также реализацию подписчика этой цепочки. В самом конце, на выходе, мы ожидаем список строк, которые мы сможем отобразить на экране. Поэтому, создадим класс, наследующийся от типа Subscriber<List<String>>. Он содержит три абстрактных метода, которые разработчик должен переопределить для своих целей: onCompleted, onError и onNext. Так как с сервера мы получим сразу весь список одним элементом, то для обработки конечного результата будет достаточно написать реализацию только для методов onError и onNext. В реализации метода onError просто отобразим сообщение об ошибке, а в onNext установим полученный список в элемент пользовательского интерфейса, который отображает списковые данные.

Попробуем запустить этот пример. Текущая реализация содержит в себе несколько проблем. Рассмотрим их поподробнее и попытаемся решить.

Во-первых, при быстром наборе текста будет создаваться множество частых событий, что в итоге приводит к большому числу запросов на сервер. В данном случае пользователю не требуется получать информацию с сервера во время того, как он набирает текст в поле, поэтому такое создание запросов на каждый символ не оправдано. Это можно решить с помощью оператора debounce. Данный оператор пропускает данные, которые не вошли в некоторый временной интервал и по окончании этого интервала, если не пришли новые данные, возвращает последние пришедшие. После применения этого оператора, при печати текста в поле ввода, запрос не будет отправляться на сервер, пока пользователь не закончит печать:

```
RxTextView
                .textChanges(input)
                .map(CharSequence::toString)
                .debounce(350, TimeUnit.MILLISECONDS)
                .switchMap(query -> Api.getInstance().complete(query))
                .subscribe(new InputSubscriber());
```

Во-вторых, данный код будет выполняться в основном потоке. В Android главный поток используется для отрисовки пользовательского интерфейса. Если в нем выполнять задачи, которые имеют продолжительность больше, чем время отрисовки одного кадра на экране, то пользователь заметит эти задержки: весь пользовательский интерфейс не будет реагировать пока эта задача не закончится. На самом деле, Retrofit внутри себя уже решает эту проблему, поэтому конкретно сетевой запрос на сервер будет выполняться в отдельном пуле потоков (thread pool), но мы можем явно указать где будет выполняться код с помощью метода subscribeOn(), принимающий как аргумент шедулер, который будет использоваться для выполнения операций в цепочке.

В-третьих, так как все операции выполняются в потоке, отличном от главного, в методах-обработчиках подписчика, код также будет выполняться не в основном потоке. В Android нельзя изменять элементы пользовательского интерфейса не в главном потоке, поэтому при попытке сделать это возникнет исключение и приложение аварийно завершит свою работу. Эту проблему также легко решить одной строкой кода: можно с помощью метода observeOn() указать шедулер, где будет выполняться код подписчика. Чтобы выполнить этот код в основном потоке, можно указать шедулер mainThread из класса AndroidSchedulers, который содержится в библиотеке RxAndroid.

В итоге получаем такую цепочку:

```
RxTextView
                .textChanges(input)
                .map(CharSequence::toString)
                .debounce(350, TimeUnit.MILLISECONDS)
                .switchMap(query -> Api.getInstance().complete(query))
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new InputSubscriber());
```

Заключение

Хотя технология реактивного программирования берет свое начало от функциональных языков, сейчас она активно развивается везде. Rx делает код намного более понятным, а также гибким; приложение становится более устойчивым к непредвиденным ситуациям. Реактивность - это новый подход, новое мышление в программировании, которое вносит частицу функционального программирования в ООП, таким образом, помимо объектов и их состояний, появляются потоки данных и операторы, позволяющие манипулировать этими потоками. Все это заставляет по-другому взглянуть на существующие задачи, что позволяет выявить более гибкие решения.

Список литературы

1. Czaplicki E., Chong S. Asynchronous functional reactive programming for GUIs // ACM SIGPLAN Notices. 2013. №1. p. 411-422.
2. Роджерс Р. Android. Разработка приложений: пер. с англ. М.: ЭКОМ Паблишерз, 2010. 400 с. [Rogers R. et al. Android application development. O'Reilly Media, Inc., 2010. 328 p.]
3. Eidhof C., Kugler F., Swierstra W. Functional Programming in Swift, 2014. Available at: http://www.raywenderlich.com/82599/swift-functional-programming-tutorial, accessed: 16.10.2015.
4. Daniel Lew. Grokking RxJava, Part 1. Available at: http://blog.danlew.net/2014/09/15/grokking-rxjava-part-1/, accessed : 16.10.2015.
5. Реактивное программирование. Режим доступа: https://ru.wikipedia.org/wiki/Реактивное_программирование (дата обращения: 16.10.2015).