Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа интеллектуальных систем и суперкомпьютерных технологий

Лабораторная работа

Дисциплина: Проектирование мобильных приложений

Тема: Мультипоточные приложения

Выполнил студент гр. 3530901/90201 _____Е. В. Бурков
(подпись)

Принял старший преподаватель _____ А. Н. Кузнецов
(подпись)

“_____” _____ 2021 г.

Санкт-Петербург
2021

Содержание

Цели

Получить практические навыки разработки многопоточных приложений:

1. Организация обработки длительных операций в background (worker) thread:
    - Запуск фоновой операции (Coroutine/ExecutionService/Thread)
    - Остановка фоновой операции (Coroutine/ExecutionService/Thread)
2. Публикация данных из background (worker) thread в main (ui) thread.

Освоить 3 основные группы API для разработки многопоточных приложений:

1. Kotlin Coroutines
2. ExecutionService
3. Java Threads

Задачи

- Разработайте несколько альтернативных приложений "не секундомер", отличающихся друг от друга организацией многопоточной работы. Опишите все известные Вам решения.
- Создайте приложение, которое скачивает картинку из интернета и размещает ее в ImaveView в Activity. Используйте ExecutorService для решения этой задачи.
- Перепишите предыдущее приложение с использованием Kotlin Coroutines.
- Скачать изображение при помощи библиотеки на выбор.

Java Threads

Изначально решение данной задачи базировалось на использовании потоков из Java. Данный механизм предложен в пакете *java.lang*. Данные потоки являются тяжеловесными (на разных архитектурах и машинах работают по-разному).

Для создания потока необходимо вызвать конструктор в котором описать что мы хотим от него. Чаще всего внутрь него запихивается экземпляр наследующий интерфейс Runnable (что-то что можно запустить в потоке). Далее при помощи run() можем запустить код из Runnable в текущем потоке, или же создав новый поток исполнить код в нём (start()).

Для контроля над потоком имеет несколько методов. Например, interrupt() – прерывает текущий поток. Также можно проверять прерван ли поток при помощи булевой isInterrupted(). Перейдём к первой программе для демонстрации данного подхода на практике.

НЕсекундомер с Thread

Переделаем наш поток таким образом:

```
16      private var backgroundThread = Thread {
17          while (!Thread.currentThread().isInterrupted) {
18              try {
19                  Log.d(TAG, msg: "${Thread.currentThread()} is iterating")
20                  Thread.sleep( millis: 1000)
21                  textSecondsElapsed.post {
22                      textSecondsElapsed.text = "${secondsElapsed++}"
23                  }
24              } catch (e : InterruptedException){
25                  Log.d(TAG, msg: "EXCEPTION")
26              }
27
28          }
29      }
```

Рис. 1 Поток

Будем исполнять код бесконечно. Также обернём всё конструкцией try catch. Это сделано для того, чтобы, когда поток спит, а мы хотим прервать его, то появится InterruptedException, так как поток прерван, а мы продолжаем выполнять в нём действия. Запустим программу и посмотрим на логи.

```
2021-11-21 01:14:54.662 10009-10024/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-2,5,main] is iterating
2021-11-21 01:14:54.662 10009-10024/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-2,5,main] is iterating
2021-11-21 01:14:54.662 10009-10024/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-2,5,main] is iterating
2021-11-21 01:14:55.663 10009-10024/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-2,5,main] is iterating
2021-11-21 01:14:56.664 10009-10024/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-2,5,main] is iterating
2021-11-21 01:14:57.403 10009-10009/dev.wooftown.continuewatch D/ContWatch: Saving state SEC=2
2021-11-21 01:14:57.406 10009-10024/dev.wooftown.continuewatch D/ContWatch: EXCEPTION
2021-11-21 01:14:57.406 10009-10024/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-2,5,main] is iterating
2021-11-21 01:14:57.426 10009-10038/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-3,5,main] is iterating
2021-11-21 01:14:57.426 10009-10009/dev.wooftown.continuewatch D/ContWatch: Restore state SEC=2
```

Рис. 2 Вывод сообщений дебага

Видно, что наш поток не завершается, и к тому же остаётся висеть и исполняться. Добавим завершение потока в блок catch и пронаблюдаем что произойдёт.

```
2021-11-21 01:18:50.655 10273-10288/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-2,5,main] is iterating
2021-11-21 01:18:51.656 10273-10288/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-2,5,main] is iterating
2021-11-21 01:18:52.384 10273-10273/dev.wooftown.continuewatch D/ContWatch: Saving state SEC=1
2021-11-21 01:18:52.412 10273-10301/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-3,5,main] is iterating
2021-11-21 01:18:52.412 10273-10273/dev.wooftown.continuewatch D/ContWatch: Restore state SEC=1
2021-11-21 01:18:53.413 10273-10301/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-3,5,main] is iterating
2021-11-21 01:18:54.414 10273-10301/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-3,5,main] is iterating
2021-11-21 01:18:55.368 10273-10273/dev.wooftown.continuewatch D/ContWatch: Saving state SEC=3
2021-11-21 01:18:55.387 10273-10273/dev.wooftown.continuewatch D/ContWatch: Restore state SEC=3
2021-11-21 01:18:55.388 10273-10303/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-4,5,main] is iterating
2021-11-21 01:18:56.389 10273-10303/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-4,5,main] is iterating
2021-11-21 01:18:57.389 10273-10303/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-4,5,main] is iterating
```

Рис. 3 Вывод сообщений для отладки

Теперь мы уверены, что предыдущие потоки завершены, и работает только один. При работе в рамках одной Activity всё работает как надо. Для передачи данных в UI Thread использовался метод post. *Causes the Runnable to be added to the message queue. The runnable will be run on the user interface thread.*

ExecutorService

Данное решение предложено в рамках пакета *java.lang.concurrent*, его суть — это более гибкая и удобная работа с потоками внутри нашего приложения. Также имеется весьма мощный механизм канала связи для контролирования асинхронных потоков. Применим данный способ создания фонового потока в нашем приложении.

Данный класс никак не связан с компонентом ОС Android.

Для начала необходимо создать объект нашего ExecutorService. При создании можно указать максимальный размер пространства потоков и ядер. Далее необходимо запустить что-то, для этого есть несколько методов. Execute(Runnable) – запускает переданной Runnable асинхронно. Submit(Runnable) – запускает поток и возвращает экземпляр класса Future, который можно использовать для проверки завершился наш Runnable или нет. Попробуем решить задачу при помощи средств языка Котлин. В onStart() будем запускать поток, а в onStop() прекращать его работу.

```
override fun onStart() {
    executorService = Executors.newSingleThreadExecutor()
    executorService.execute {
        while (!executorService.isShutdown){
            Log.d(TAG, msg: "${Thread.currentThread()} is iterating")
            Thread.sleep( millis: 1000)
            textSecondsElapsed.post {
                textSecondsElapsed.text = "${secondsElapsed++}"
            }
        }
    }
    super.onStart()
}

override fun onStop() {
    executorService.shutdown()
    super.onStop()
}
```

Рис. 4 Предложенная реализация

В методе onStart создаём новый ExecutorService. Сразу же через лямбду задаём ему отсчёт каждую секунду, пока наш сервис не остановлен. Для прекращения работы используется shutdown(). Коротко говоря, данный метод останавливает передачу в ExecutorService новый задач, а также даёт всем работающим потокам остановиться (тут мы использовали это в условии).

```
2021-11-21 02:01:48.769 14121-14136/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-1-thread-1,5,main] is iterating
2021-11-21 02:01:49.770 14121-14136/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-1-thread-1,5,main] is iterating
2021-11-21 02:01:50.771 14121-14136/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-1-thread-1,5,main] is iterating
2021-11-21 02:01:51.772 14121-14136/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-1-thread-1,5,main] is iterating
2021-11-21 02:01:51.930 14121-14121/dev.wooftown.appexecutorservice D/ContWatch: Saving state SEC=3
2021-11-21 02:01:51.952 14121-14150/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-2-thread-1,5,main] is iterating
2021-11-21 02:01:51.952 14121-14121/dev.wooftown.appexecutorservice D/ContWatch: Restore state SEC=3
2021-11-21 02:01:52.953 14121-14150/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-2-thread-1,5,main] is iterating
2021-11-21 02:01:53.953 14121-14150/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-2-thread-1,5,main] is iterating
2021-11-21 02:01:54.954 14121-14150/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-2-thread-1,5,main] is iterating
2021-11-21 02:01:55.955 14121-14150/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-2-thread-1,5,main] is iterating
2021-11-21 02:01:56.956 14121-14150/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-2-thread-1,5,main] is iterating
2021-11-21 02:01:57.956 14121-14150/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-2-thread-1,5,main] is iterating
2021-11-21 02:01:58.957 14121-14150/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-2-thread-1,5,main] is iterating
2021-11-21 02:01:59.746 14121-14121/dev.wooftown.appexecutorservice D/ContWatch: Saving state SEC=10
2021-11-21 02:01:59.785 14121-14154/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-3-thread-1,5,main] is iterating
2021-11-21 02:01:59.785 14121-14121/dev.wooftown.appexecutorservice D/ContWatch: Restore state SEC=10
2021-11-21 02:02:00.786 14121-14154/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-3-thread-1,5,main] is iterating
2021-11-21 02:02:01.786 14121-14154/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-3-thread-1,5,main] is iterating
2021-11-21 02:02:02.787 14121-14154/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-3-thread-1,5,main] is iterating
```

Рис. 5 Результат работы приложения

Можно заметить, что при создании нового ExecutorService создаётся новый Thread Pool. В Java потоки маппируются в структуру потоков, с которыми оперирует операционная система при выдаче ресурсов. Поэтому исполняя что-то в ExecutorService мы просто добавляем задачу в наш пул.

Нет смысла переживать, что мы создаём так много Thread pool’ов, сборщик мусора с радостью удаляет все ненужные системе ресурсы.

```
2021-11-21 02:02:01.786 14121-14154/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-3-thread-1,5,main] is iterating
2021-11-21 02:02:02.787 14121-14154/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-3-thread-1,5,main] is iterating
2021-11-21 02:10:10.877 14167-14182/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-1-thread-1,5,main] is iterating
2021-11-21 02:10:11.877 14167-14182/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-1-thread-1,5,main] is iterating
2021-11-21 02:10:12.878 14167-14182/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-1-thread-1,5,main] is iterating
```

Рис. 6 Повторное появление pool-1

Использование корутин

Взглянем в сторону паттерна корутин из ЯП Kotlin. Coroutines – те же самые потоки, только имеют ряд преимуществ.

Корутины легковесны – благодаря этому появляется возможность запуска намного большего количество корутин чем обычных потоков. Также они более безопасны с точки зрения утечки памяти. Они имеют удобные механизмы связи между собой и включены во многие библиотеки Jetpack, что делает их просто какой-то вкуснятиной.

Для работы с корутинами в рамках разработки Android приложения необходимо подключить соответствующую библиотеку.

Для создания корутины можно воспользоваться билдером launch{}, он запустит новую корутину параллельно с остальным кодом. Также необходим scope, некий мост между кодом, который выполняется последовательно и кодом из корутины. Конструкция launch{} возвращает экземпляр класса Job, который может быть присоединён (join()) к другому потоку. Для остановки потока можно воспользоваться методом cancel(). Для того, чтобы приостановить выполнение корутины имеется функция delay().

Применим данные из пункта выше в нашей работе.

Рис. 7 Пример работы с корутиной

Данное решение выглядит очень странным из-за MainScope(), обратимся к нашему любому сайту и найдём что-то более подходящее. Сразу же натыкаемся на lifecycleScope, возможно, это может сделать наше приложение более красивым. Подключаем новую зависимость и переписываем приложение в более красивом варианте:

Рис. 8 Корутина с использованием lifecycleScope

Загрузка изображения

Самое время применить полученные знания в следующей задаче: необходимо загрузить изображение и поместить его в ImageView. Для этого воспользуемся ViewModel. Если мы попытаемся загрузить картинку в UI потоке, то получим соответствующую ошибку.

Для начала напишем ViewModel, который будет создавать поток для загрузки изображения.

```
class MainViewModel : ViewModel() {

    var bitmap: MutableLiveData<Bitmap?> = MutableLiveData( value: null)

    private val executorService: ExecutorService = Executors.newSingleThreadExecutor()

    fun downloadImage(url: URL) {
        executorService.execute {
            Log.d(TAG,  msg: "Downloading in ${Thread.currentThread()}")
            bitmap.postValue(BitmapFactory.decodeStream(url.openConnection().getInputStream()))
        }
    }

    override fun onCleared() {
        executorService.shutdown()
        super.onCleared()
    }

    companion object {
        const val TAG = "ExecutorService"
    }

}
```

Рис. 9 Реализованный ViewModel

В главном Activity установим наблюдателя за bitmap и теперь по нажатию кнопки у нас будет загружаться картинка, а когда загрузка завершиться она будет находиться в ImageView.

Также проверим, чтобы картинка загружалась в другом потоке, и если нам захочется перевернуть экран, то загрузка всё равно будет продолжаться. Для этого можно ограничить скорость интернета.

Загрузка картинки при помощи корутины

Решим эту же задачу при помощи корутины. Заменим метод по загрузке картинки.

```
fun downloadImage(url: URL) {
    viewModelScope.launch(Dispatchers.IO) {  this: CoroutineScope
        Log.d(TAG,  msg: "Sleeping 2sec")
        delay( timeMillis: 2000)
        Log.d(TAG,  msg: "Downloading in ${Thread.currentThread()}")
        bitmap.postValue(BitmapFactory.decodeStream(url.openConnection().getInputStream()))
    }

}
```

Рис. 10 Загрузка при помощи корутины

Использование готового решения

Испытаем библиотеку Picasso для загрузки изображения.

```
38    fun downloadImage(url: URL) {
39        Picasso.get().load(url.toString()).into(
40            object : Target {
41                override fun onBitmapLoaded(bitmap: Bitmap?, from: Picasso.LoadedFrom?) {
42                    bitmapData.postValue(bitmap)
43                }
44
45                override fun onBitmapFailed(e: Exception?, errorDrawable: Drawable?) {}
46
47                override fun onPrepareLoad(placeHolderDrawable: Drawable?) {}
48            }
49        )
50
51
52    }
```

Рис. 11 Загрузка изображения при помощи библиотеки Picasso

Честно говоря, разработчики хотели, чтобы методы использовали чуть по-другому, но я решил изменить только метод загрузки изображения в ViewModel.

Выводы

В ходе выполнения лабораторной работы было выполнено знакомство с многопоточными Android приложениями. Во всех заданиях необходимо было исполнять код не в UI потоке. Для отправки значений в UI поток использовался метод post. Также возможно использование обсерверов из ViewModel.

Первым делом вниманию подверглись Java Threads. Чтобы запустить поток необходимо вызвать метод start(). Для остановки потока используется метод interrupt(). Также необходимо уделить внимание происходящему в потоке, или может получить InterruptException. В целом данный механизм является очень деревянным, поэтому в 5 Java завезли следующее средство.

ExecutorService является некой обёрткой над обычными потоками. С его помощью можно по-разному конфигурировать исполнение кода в разных потоках, а также настроить сами потоки и пулы. Для запуска потоков необходимо создать для них пул, а после можем посылать туда сами потоки. Для завершения работы используется shutdown() – ожидает пока все потоки завершается и завершается сам, или shutdownNow() – мгновенное прекращение работы всех потоков.

Самым вкусным инструментом работы с потоками являются легковесные корутины. Благодаря скоупам и простоте реализации работать с ними одно удовольствие. Для запуска потока можно, например использовать launch(), а для прекращения работы cancel().

В последней части работы была выполнена загрузка изображения из интернета при помощи разных средств. Было обращено внимание, что приложению требуются специальные разрешения для работы с сетью. В конце опробовалась библиотека Picassso. Моя реализация выглядит убого, потому что я захотел сохранить примерную архитектуру приложения, а сами методы данной библиотеки больше подходят тем, кто использует Data Binding.

Время на выполнение работы

- 1 – 210 мин
- 2 – 60 мин
- 3 – 20 мин
- 4 – 40 мин
- Отчёт – 30 мин (форматирование)

Список источников

- https://developer.android.com
- https://github.com/andrei-kuznetsov/android-lectures
- https://github.com/wooftown/spbstu-android - листинги