

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.

Содержание

Цели	3
Задачи	3
Java Threads	4
Несекундомер с Thread	4
ExecutorService	6
Использование корутин	8
Загрузка изображения	10
Загрузка картинки при помощи корутины	11
Использование готового решения	12
Выводы	13
Время на выполнение работы	14
Список источников	15

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 ImageView 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*.

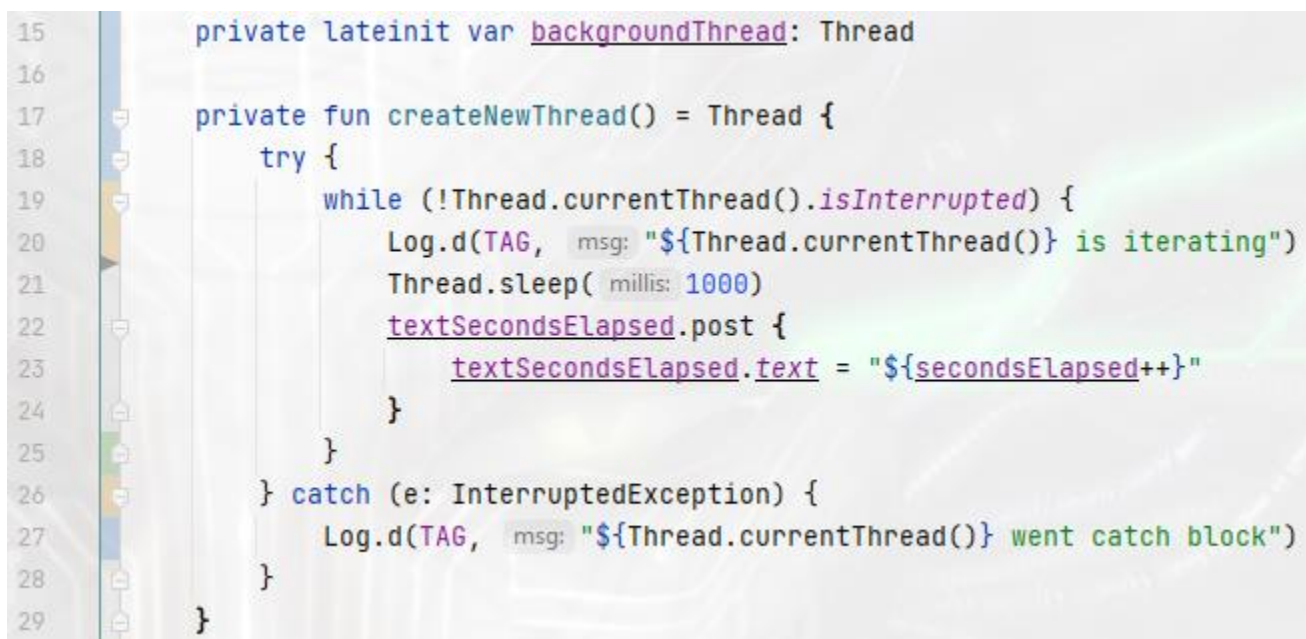
Для создания потока необходимо вызвать конструктор, в котором описать что мы хотим от нового потока. Чаще всего в конструктор передаётся экземпляр наследующий интерфейс Runnable (Runnable – то, что можно выполнить в потоке). Далее при помощи метода `run()` можем запустить код из Runnable в текущем потоке, или же в новом потоке – для этого нужен метод `start()`.

Для контролирования потока существует метод `interrupt()`. Вызов данного метода устанавливает у потока статус, что он прерван. При этом следует заметить, что вызов данного метода не завершает поток, а лишь устанавливает статус. Также следует заметить, что при обработке исключения `InterruptedException` статус потока автоматически сбрасывается. 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:

A screenshot of a code editor showing Kotlin code. The code defines a `backgroundThread` of type `Thread` and a `createNewThread` function. The function uses a `try-catch` block with a `while` loop to keep the thread running until it is interrupted. Inside the loop, it logs a message, sleeps for 1000 milliseconds, and updates a UI element `textSecondsElapsed`. The catch block logs a message when an `InterruptedException` occurs.

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

Рис. 1 Поток

Код также выполняется бесконечно, но теперь он обернут конструкцией try-catch. Если мы будем получать InterruptedException, то на консоль выведется соответствующее сообщение, а поток прекратит свою работу.

Для запуска потока вызывается функция её инициализации, потому что мы не хотим, чтобы прерванный поток начинал своё исполнение заново. Далее производим запуск потока, а для его остановки используем interrupt (interrupt = остановка из-за нашей реализации).



```
40 override fun onStart() {
41     super.onStart()
42     Log.d( tag: "TAG", msg: "onStart()")
43
44     backgroundThread = createNewThread()
45     backgroundThread.start()
46     Log.d( tag: "TAG", msg: "${backgroundThread.id}")
47
48 }
49
50 override fun onStop() {
51     super.onStop()
52     Log.d( tag: "TAG", msg: "onStop()")
53
54     backgroundThread.interrupt()
55 }
```

Рис. 2 Процесс запуска и остановки потока



```
2021-12-15 10:36:53.122 27333-27363/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-4,5,main] is iterating
2021-12-15 10:36:53.802 27333-27333/dev.wooftown.continuewatch D/ContWatch: Saving state SEC=9
2021-12-15 10:36:53.810 27333-27363/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-4,5,main] went catch block
2021-12-15 10:36:53.821 27333-27333/dev.wooftown.continuewatch D/ContWatch: Restore state SEC=9
2021-12-15 10:36:53.821 27333-27368/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-5,5,main] is iterating
2021-12-15 10:36:54.821 27333-27368/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-5,5,main] is iterating
2021-12-15 10:36:55.821 27333-27368/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-5,5,main] is iterating
2021-12-15 10:36:56.822 27333-27368/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-5,5,main] is iterating
2021-12-15 10:36:57.822 27333-27368/dev.wooftown.continuewatch D/ContWatch: Thread[Thread-5,5,main] is iterating
```

Рис. 3 Вывод сообщений для отладки

Исходя из сообщений отладки можно сделать вывод, что в один момент времени работает только один фоновый поток секундомера. 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 переданный `Runnable` асинхронно. Работает по принципу “выстрелил и забыл”. `Submit(Runnable)` — запускает поток и возвращает экземпляр класса `Future`, который можно использовать для различных действий над потоком. При помощи `Future` можно использовать метод `get()`, который насмерть блокирует текущий поток и будет ждать пока фоновый не завершится. Это плюс т. к. для использования `join()` нужно более глубокое понимание работы потоков и их использование не всегда очевидно. Также можно задать время, в течение которого ждать завершения потока, если мы боимся, что он не завершится вообще. Для общения между потоками также вместо `Runnable` можно использовать `Callable` объект. При помощи метода `shutdown()` имеется возможность завершить приём новых задач, и ожидать выполнения работы. Если мы хотим остановить одну задачу, то надо воспользоваться объектом `Future` — результат выполнения асинхронных вычислений. В данном примере используем метод `cancel(Boolean)`. Данный метод совершает попытку остановить выполнение работы. Если передать аргумент `True`, то данный поток должен быть прерван.

Для начала необходимо создать пул потоков в Application классе.

```
7 class MyApplication : Application() {  
8     val watchPool : ExecutorService = Executors.newSingleThreadExecutor()  
9 }
```

Рис. 4 Определение пула

Создаём пул с одним потоком. Далее в классе активити в коллбеке onStart() будем использовать функцию для добавления потока в очередь на исполнение посредством метода submit(), а объект класса Future будет сохранять себе, т. к. это нужно для контроля над потоком.

```
15 private lateinit var backgroundFuture: Future<*>  
16  
17 private fun submitBackground(executorService: ExecutorService) = executorService.submit {  
18     while (!executorService.isShutdown) {  
19         Log.d(TAG, msg: "${Thread.currentThread()} is iterating")  
20         Thread.sleep( millis: 1000)  
21         textSecondsElapsed.post {  
22             textSecondsElapsed.text = "${secondsElapsed++}"  
23         }  
24     }  
25 }  
26  
27 override fun onCreate(savedInstanceState: Bundle?) {  
28     super.onCreate(savedInstanceState)  
29     val binding = ActivityMainBinding.inflate(layoutInflater)  
30     textSecondsElapsed = binding.SecondsElapsed  
31     setContentView(binding.root)  
32 }  
33  
34 override fun onStart() {  
35     super.onStart()  
36     backgroundFuture = submitBackground((applicationContext as MyApplication).watchPool)  
37 }  
38  
39 override fun onStop() {  
40     super.onStop()  
41     backgroundFuture.cancel(true)  
42 }
```

Рис. 5 Запуск и остановка потока

Для остановки потока используем метод cancel для Future нашего потока. Для доступа к классу Application используем applicationContext. Ниже можно увидеть, что все потоки выполняются в одном пуле.


```

2021-12-15 11:29:05.600 27864-27879/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-1-thread-1,5,main] is iterating
2021-12-15 11:29:06.601 27864-27879/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-1-thread-1,5,main] is iterating
2021-12-15 11:29:07.602 27864-27879/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-1-thread-1,5,main] is iterating
2021-12-15 11:29:07.687 27864-27864/dev.wooftown.appexecutorservice D/ContWatch: Saving state SEC=10
2021-12-15 11:29:07.724 27864-27879/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-1-thread-1,5,main] is iterating
2021-12-15 11:29:07.724 27864-27864/dev.wooftown.appexecutorservice D/ContWatch: Restore state SEC=10
2021-12-15 11:29:08.725 27864-27879/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-1-thread-1,5,main] is iterating
2021-12-15 11:29:09.726 27864-27879/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-1-thread-1,5,main] is iterating
2021-12-15 11:29:10.727 27864-27879/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-1-thread-1,5,main] is iterating
2021-12-15 11:29:11.727 27864-27879/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-1-thread-1,5,main] is iterating
2021-12-15 11:29:12.201 27864-27864/dev.wooftown.appexecutorservice D/ContWatch: Saving state SEC=14
2021-12-15 11:29:12.222 27864-27879/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-1-thread-1,5,main] is iterating
2021-12-15 11:29:12.222 27864-27864/dev.wooftown.appexecutorservice D/ContWatch: Restore state SEC=14
2021-12-15 11:29:13.223 27864-27879/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-1-thread-1,5,main] is iterating
2021-12-15 11:29:14.224 27864-27879/dev.wooftown.appexecutorservice D/ContWatch: Thread[pool-1-thread-1,5,main] is iterating

```

Рис. 6 Результат работы приложения

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 количества корутин чем обычных потоков. Также они более безопасны с точки зрения утечки памяти.

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

Для создания корутины используются конструкторы: `launch` (создать и забыть), `async` (вернуть `promise`), `runBlocking` (заблокировать поток) и так далее. 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.


```

18     private val job = MainScope().launch { this: CoroutineScope
19         while (true) {
20             Log.d(TAG, msg: "Coroutine works")
21             delay( timeMillis: 1000)
22             textSecondsElapsed.post {
23                 textSecondsElapsed.text = "${secondsElapsed++}"
24             }
25         }
26     }
27
28     override fun onStart() {
29         super.onStart()
30         job.start()
31     }
32
33     override fun onStop() {
34         super.onStop()
35         job.cancel()
36     }
37

```

Рис. 7 Пример работы с корутиной

Обратимся к нашему любому сайту и найдём более корректное и красивое решение. При помощи `lifecycleScope` возможно сделать так, чтобы корутина работала только когда приложение RESUMED. Перепишем приложение в более красивом варианте:

```

lifecycleScope.launchWhenResumed { this: CoroutineScope
    while (isActive) {
        Log.d(TAG, msg: "Coroutine works")
        delay( timeMillis: 1000)
        textSecondsElapsed.post {
            textSecondsElapsed.text = "${secondsElapsed++}"
        }
    }
}

```

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.

```
16 class MainViewModel(application: Application) : AndroidViewModel(application) {
17
18     val bitmap: MutableLiveData<Bitmap> = MutableLiveData()
19
20     private val context = getApplication<MyApplication>()
21     private val executorService: ExecutorService = context.downloadThread
22
23     fun downloadImage(url: URL) {
24         executorService.execute {
25             Log.d(TAG, msg: "Sleeping 5sec")
26             Thread.sleep( millis: 5000)
27             Log.d(TAG, msg: "Downloading in ${Thread.currentThread()}")
28             bitmap.postValue(BitmapFactory.decodeStream(url.openConnection().getInputStream()))
29         }
30     }
31
32     companion object {
33         const val TAG = "ExecutorService"
34     }
35
36 }
```

Рис. 9 Реализованный ViewModel

Не рекомендуется передавать свой контекст в `ViewModel`, поэтому используем класс `AndroidViewModel`. Далее при помощи метода `getApplication` получаем экземпляр класса `MyApplication`.

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.

```
16 class MainViewModel : ViewModel() {
17
18     val bitmap: MutableLiveData<Bitmap> = MutableLiveData()
19
20     fun downloadImage(url: URL) {
21         viewModelScope.launch(Dispatchers.IO) { this: CoroutineScope
22             Log.d(TAG, msg: "Sleeping 2sec")
23             delay( timeMillis: 2000)
24             Log.d(TAG, msg: "Downloading in ${Thread.currentThread()}")
25             val pic = BitmapFactory.decodeStream(url.openConnection().getInputStream())
26             withContext(Dispatchers.Main){ this: CoroutineScope
27                 bitmap.value = pic
28             }
29         }
30     }
31
32     companion object {
33         const val TAG = "ExecutorService"
34     }
35
36 }
37 }
```

Рис. 10 Загрузка при помощи корутины

При помощи диспетчера корутин (переключения потоков) вместо `postValue` используем обычное присвоение.

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 прерывания потока используется метод `interrupt()`, при его вызове поток получает статус `Interrupted`. Также необходимо уделить внимание происходящему в потоке, иначе можем получить `InterruptedException`. В целом данный механизм является очень грубым при работе, поэтому следующим пунктом были рассмотрены `ExecutorServices`.

`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 при помощи методов `submit/execute` можно начать их выполнение. Для остановки потоков можно использовать `cancel()`. Для завершения работы используется `shutdown()` – ожидает пока все потоки завершаются и завершается сам.

Также была выполнена работа с `Kotlin Coroutines`. Для их запуска необходимо воспользоваться одним из билдеров. Для остановки использовался метод `cancel()`. Корутины имеют обширные возможности: управление работой посредством `Job`, различные скоупы, диспатчеры и так далее.

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 `Picasso`. Моя реализация выглядит не лучшим, 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
- <https://docs.oracle.com>