САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

ПЕТРА ВЕЛИКОГО

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Институт компьютерных наук и технологий

Высшая школа интеллектуальных систем и суперкомпьютерных технологий

Лабораторная работа №6

Дисциплина

«Проектирование мобильных приложений»

Вариант 18

выполнил:

Пименов С. В.

группа: 3530901/90201

преподаватель:

Кузнецов А. Н.

Санкт-Петербург

2021

Цели

* Получить практические навыки разработки многопоточных приложений
* Освоить 3 основные группы API для разработки многопоточных приложений:

Kotlin Coroutines

ExecutionService

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.
* Скачайте изображение при помощи библиотеки на выбор: Glide, picasso или fresco.

Решение при помощи Java Threads

Для реализации с помощью Threads сначала создается конструктор, в который передается то, что мы можем выполнить в потоке.

Далее при помощи run() можно запустить код из конструктора в текущем потоке или в новом потоке – при помощи start().

Также у Java Threads имеется метод interrupt(), который устанавливает у потока флаг isInterrupted() = true, при этом не завершая поток.

Рассмотрим созданную мной реализацию:

|  |
| --- |
| Листинг 1 Реализация при помощи Java Thread |
| private lateinit var backgroundThread: Thread  fun createThread() = Thread **{** try {  while(!Thread.currentThread().*isInterrupted*) {  Log.d(*TAG*, "${Thread.currentThread()} running now")  Thread.sleep(1000)  textSecondsElapsed.post **{** textSecondsElapsed.*text* = "Seconds Elapsed: ${secondsElapsed++}" **}** }  }  catch (e: InterruptedException) {  Log.d(*TAG*, "${Thread.currentThread()} caught exception")  } **}**  override fun onStart() {  super.onStart()  backgroundThread = createThread()  backgroundThread.start() }  override fun onStop() {  super.onStop()  backgroundThread.interrupt() } |

Код будет исполняться, пока у потока флаг isInterrupted() = false. Если мы получим InterruptedExeption – то в logcat выведется сообщение, а поток при этом прекратит свою работу – за это отвечает конструкиця try {} catch {}, при этом следует указать, что при обработке исключения статус потока сбрасывается автоматически.

Мы используем функцию createThread(), для того, чтобы новый поток создавался, если приложение было свернуто, а после развернуто. В противном случае прерванный поток начинал бы свое выполнение заново. Для остановки используем interrupt – это позволяет нам выйти из цикла while.

|  |
| --- |
|  |
| Рис. 1 Сообщения отладки |

Решение при помощи ExecutorService

ExecutorService является наследником интерфейса Executor. Он является описанием особого Executor’a и позволяет получить java.util.concurrent.Future, чтобы отслеживать процесс выполнения. Также у нас имеется специальная фабрика java.util.concurrent.Executors, которая нам позволяет создавать доступные по умолчанию реализации ExecutorService.

В фабрике Executors существуют различные методы: например, мы можем создать фиксированный пул потоков размером 2 при помощи Executors.newFixedThreadPool(2) илисоздать пул всего из одного потока при помощи newSingleThreadExecutor, также можно создать пул с кэшированием – newCachedThreadPool, в котором потоки убираются из пула, если они простаивали одну минуту.

Реализован ExecutorService при помощи BlokingQueue, в которую помещаются задачи, и из которой эти самые задачи выполняются.

Рассмотрим созданную мной реализацию:

|  |
| --- |
| Листинг 2 Реализация при помощи ExecutorService |
| private val pool = Executors.newSingleThreadExecutor() private lateinit var future: Future<\*>  fun submit (executor: ExecutorService): Future<\*> = executor.submit **{** try {  while(true) {  Log.d(*TAG*, "${Thread.currentThread()} running now")  Thread.sleep(1000)  textSecondsElapsed.post **{** textSecondsElapsed.*text* = "Seconds Elapsed: ${secondsElapsed++}" **}** }  }  catch (e: InterruptedException) {  Log.d(*TAG*, "${Thread.currentThread()} caught exception")  } **}**  override fun onStart() {  super.onStart()  future = submit (pool) }  override fun onStop() {  super.onStop()  future.cancel(true) } |

Для начала создаем пул с одним потоком, в onStart() добавляем поток в очередь при помощи submit(). Для остановки потока используется метод cancel, с аргументом true, который останавливает выполнение работы.

|  |
| --- |
|  |
| Рис 2. Сообщения отладки |

Наблюдаем, что все потоки выполняются в одном пуле.

Решение при помощи Coroutines

Корутины – аналог Java Threads со своими преимуществами. Во-первых: можно запустить намного больше корутин, чем потоков, во-вторых: они более безопасны с точки зрения утечки памяти. Корутины, с другой стороны, выглядят как простой последовательный код, пряча всю сложность внутри библиотек. В то же время они предоставляют возможность запускать асинхронный код без всяких блокировок, что открывает большие возможности для различных приложений.

Для создания корутин используется конструктор:

* launch {} – ничего не возвращает
* async{} - возвращает экземпляр Deferred, в котором имеется функция await(), которая возвращает результат корутины, прямо как Future в Java, где мы делаем future.get() для получения результата.
* runBlocking - запускает новую корутину, блокирует текущий поток и ждёт пока выполнится блок кода.
* launchWhenResumed
* и т.д.

Coroutine Builder в качестве возвращаемого значения отдаёт нам подкласс класса Job. С её помощью мы можем управлять жизненным циклом корутины.

Старт при помощи start(), отмена методом cancel(), ждать завершение Job при помощи join(), приостановка выполнения корутины методом delay().

Рассмотрим созданную мной реализацию:

|  |
| --- |
| Листинг 3 Решение при помощи корутин |
| override fun onCreate(savedInstanceState: Bundle?) {  …  *lifecycleScope*.launchWhenResumed **{** while (*isActive*) {  Log.d(ContentValues.*TAG*, "${Thread.currentThread()} running now")  delay(1000)  textSecondsElapsed.post **{** textSecondsElapsed.*text* = "Seconds Elapsed: ${secondsElapsed++}" **}** }  **}** } |

При помощи lifecycleScope сделаем так, чтобы корутина работала только когда приложение isResumed(), в противном случае нам нужно было бы переопределять методы onStart() и onStop(), запуская и прекращая в них работу корутины.

Загрузка изображения в фоновом потоке

Для начала реализуем класс Download, который создает поток для загрузки изображения.

|  |
| --- |
| Листинг 4. Класс Download |
| class Download: ViewModel() {  private val pool : ExecutorService = Executors.newSingleThreadExecutor()  val bitmap: MutableLiveData<Bitmap> = MutableLiveData()   fun downloadImage(url: URL) {  pool.execute **{** Log.d(*TAG*, "Downloading in ${Thread.currentThread()}")  bitmap.postValue(BitmapFactory.decodeStream(url.openConnection().getInputStream()))  **}** } |

|  |
| --- |
| Листинг 5. Класс MainActivity |
| class MainActivity : AppCompatActivity() {  override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  val binding = ActivityMainBinding.inflate(*layoutInflater*)  setContentView(binding.*root*)  val download: Download by *viewModels*()  binding.button1.setOnClickListener **{** download.downloadImage(URL("https://thispersondoesnotexist.com/image")) **}** download.bitmap.observe(this) **{**binding.imageView.setImageBitmap(**it**)**}** } } |

По нажатию кнопки картинка будет загружаться, а когда она загрузится – она попадет в ImageView, за это отвечает download.bitmap.observe. Также я проверил, что при повороте экрана загрузка продолжается.

Загрузка при помощи корутины

В реализации с помощью корутин заменим метод downloadImage() в классе Download.

|  |
| --- |
| Листинг 5. Замена в методе |
| fun downloadImage(url: URL) {  *viewModelScope*.*launch*(Dispatchers.IO) **{** Log.d(*TAG*, "Downloading in ${Thread.currentThread()}")  bitmap.postValue(BitmapFactory.decodeStream(url.openConnection().getInputStream()))    **}** } |

Также вместо postValue можно было использовать присвоение при помощи диспетчера корутин, выглядит это следующим образом:

|  |
| --- |
| Листинг 6. Использование диспетчера корутин |
| val image = BitmapFactory.decodeStream(url.openConnection().getInputStream()) withContext(Dispatchers.Main) **{** bitmap.*value* = image  **}** |

Загрузка при помощи библиотеки Picasso

|  |
| --- |
| Листинг 6. Загрузка при помощи Picasso |
| binding.button1.setOnClickListener **{** Picasso.get().load("https://thispersondoesnotexist.com/image").into(binding.imageView) **}** |

Стороння библиотека позволила нам загрузить изображение буквально в одну строку. В листинге 6 видно, что изображение загружается по ссылке, а нажатие на кнопку обрабатывается и изображение добавляется в binding.imageView.

Сравнение threads, executor\*, coroutines

|  |  |  |  |
| --- | --- | --- | --- |
|  | Java Threads | ExecutorService | Kotlin Coroutines |
| Общее | Поток в Java представлен в виде экземпляра класса java.lang.Thread. Экземпляры класса Thread в Java сами по себе не являются потоками, это лишь своего рода API для низкоуровневых потоков, которыми управляет JVM и операционная система. Когда при помощи java launcher'а мы запускаем JVM, она создает главный поток с именем main и ещё несколько служебных потоков. | ExecutorService исполняет асинхронный код в одном или нескольких потоках. Создание инстанса ExecutorService'а делается либо вручную через конкретные имплементации (ScheduledThreadPoolExecutor или ThreadPoolExecutor), но проще будет использовать фабрики класса Executors. | Описано в решении с помощью Coroutines |
| Создание | Первый способ – создание своего наследника.  **public** **class** HelloWorld{  **public** **static** **class** MyThread **extends** Thread {  @Override  **public** **void** run() {  System.out.println("Hello, World!");  }  }  **public** **static** **void** main(String []args){  Thread thread = **new** MyThread();  thread.start();  }  }  Однако, у такого способа есть свои минусы: в иерархию классов включается Thread, нарушается принцип единственной ответственности.  Второй способ – запуск через Runnable.  **public** **static** **void** main(String []args){  Runnable task = () -> {  System.out.println("Hello, World!");  };  Thread thread = **new** Thread(task);  thread.start();  } | Сначала необходимо создать пул потоков, подробнее про разные варианты было сказано при решении задачи при помощи ExecutorService.  Также при создании есть возможность настроить множество параметров, что является преимуществом по сравнению с Threads. | Описано в решении с помощью Coroutines |
| Запуск | Запуск задачи выполняется в методе run, а запуск потока в методе start. Не стоит их путать, т.к. если мы запустим метод run напрямую, никакой новый поток не будет запущен. Именно метод start просит JVM создать новый поток. | ExecutorService.submit(Runnable)  Executor.execute(Runnable) | Описано в решении с помощью Coroutines |
| Завершение | Установка флага isInterrupted = true с помощью Thread.interrupt().  Смена флага isActive с помощью метода disable(). | Submit -> Future.cancel  Execute -> shutdown (прием новых закрывается, идет обработка старых)  E -> shutdownNow (остановка всех потоков) | Или с помощью жизненного цикла объекта или при помощи метода cancel() у Job |
| Передача в UI поток | Activity.runOnUiThread(Runnable)  View.post(Runnable) | Аналогично с Threads | Те же, что и у Threads, плюс переключение контекстов (пример в листинге 6) |

Выводы

В ходе данной ЛР мы познакомились со спецификой работы с многопоточными приложениями, а именно: изучили Java Threads, Kotlin Coroutines, ExecutorService, решили с помощью них одну и ту же задачу и выявили различия между ними.

Также была реализована загрузка изображения при помощи Coroutines и сторонней библиотеки Picasso. Не удивительно, что библиотека справилась с этой задачей без нагромождения кода, ведь она заточена под такие задачи. Кроме этого, я узнал, что приложению требуются специальные разрешения для работы с сетью – это меня удивило.

Github: https://github.com/pimenov01/lab6