4.1. Многопоточность в Android приложениях

Сайт: Samsung Innovation Campus
Курс: Мобильная разработка на Kotlin
Книга: 4.1. Многопоточность в Android приложениях

Напечатано:: Murad Rezvan
Дата: понедельник, 3 июня 2024, 17:53

Оглавление

1. Введение в многопоточность

В ходе разработки Android приложений, так или иначе приходится сталкиваться с необходимостью иметь несколько потоков.

Поток можно представить как последовательность команд программы, которая претендует на использование процессора вычислительной системы для своего выполнения. Потоки одной и той же программы работают в общем адресном пространстве и, тем самым, разделяют (совместно используют) данные программы.

При запуске в приложении создается создает основной поток выполнения *main thread*. Он координирует отправку событий в виджеты пользовательского интерфейса, а также отвечает за связь с компонентами Android UI. Важно избегать использование *main thread* в любых операциях, которые могут привести к его блокировке. Такие операции, как например обращения к базам данных или загрузка определенных компонентов из Интернета, выступают характерными действий, которые не следует выполнять в основном потоке. При запуске таких операций в основном потоке пользовательский интерфейс не будет ни на что реагировать до их завершения. Именно поэтому они обычно выносятся в отдельный поток.

В Android существует множество способов создания и управления потоками, а также множество сторонних библиотек, которые делают управление потоками гораздо более удобным и быстрым в разработке. В данном разделе учебника мы познакомимся с некоторыми типичными решениями для многопоточных приложений в Android с использованием языка программирования Kotlin.

В Android можно классифицировать потоки на два ключевых типа:

- потоки связанные с активностью или фрагментом (они привязаны к жизненному циклу активности или фрагмента и завершаются сразу после их уничтожения);
- потоки не связанные с активностью или фрагментом (они могут продолжать работу за пределами жизни активности или фрагмента, из которых они были созданы).

2. Потоки UI и Worker

Поток пользовательского интерфейса (UI-поток) - это основной поток выполнения для приложения. Именно здесь выполняется большая часть кода вашего приложения. Все компоненты вашего приложения (Activity, Service, ContentProvider, BroadcastReceiver) создаются в этом потоке, и все системные вызовы этих компонентов выполняются нем. Кроме этого UI-поток отвечает за обновление элементов разметки приложения.

Рабочий поток (Worker-поток) - это поток, в котором можно выполнять обработку, которая не должна прерывать какие-либо изменения, происходящие в UI-потоке. Worker-потоки не владеют пользовательским интерфейсом и не взаимодействует с ним.

Для создания новых потоков нам доступен стандартный функционал класса Thread из стандартной библиотеки. Обычно первичным потоком будет поток, который владеет и управляет пользовательским интерфейсом. Затем можно запустить один или несколько рабочих потоков, для выполнения определенных задач. Эти потоки не изменяют пользовательский интерфейс напрямую.

Если нам нужно изменить компонент пользовательского интерфейса, например изменить текст в текстовом представлении, нам нужно использовать UI-поток.

Создание потока в Kotlin аналогично созданию потока в Java. Мы могли бы создать наследник класса `Thread` следующим образом:

```
class MyThread: Thread() {
    public override fun run() {
        println("Running")
    }
}
```

Или мы можем реализовать интерфейс `Runnable`:

```
class MyRunnable: Runnable {
    public override fun run() {
         println("Running")
    }
}
```

Так же, как в Java, мы можем выполнить запустить его, вызвав метод `start()`:

```
val thread = MyThread()
thread.start()

val threadWithRunnable = Thread(MyRunnable())
threadWithRunnable.start()
```

В Kotlin есть стандартная библиотечная функция `thread`:

```
public fun thread(
    start: Boolean = true,
    isDaemon: Boolean = false,
    contextClassLoader: ClassLoader? = null,
    name: String? = null,
    priority: Int = -1,
    block: () -> Unit): Thread
```

Ее можно использовать следующим образом:

```
thread {
    Thread.sleep(1000)
    println("test")
}
```

Она имеет множество дополнительных параметров.

- `start` - немедленно запустить поток;
- `isDaemon` - для создания потока как потока демона;
- `contextClassLoader` - загрузчик классов, используемый для загрузки классов и ресурсов;
- `name` - установка имени потока;

- `priority` - установка приорита потока.

Реализация UI-потока

Если поток запущен и необходимо обновить элемент пользовательского интерфейса, можно воспользоваться функцией `runOnUIThread()`. Далее приведем пример приложения, где `TextView` меняется каждую секунду. Сначала на экране необходимо поместить приветственное сообщение. После нажатия кнопки в текстовом представлении будут показывать два сообщения «First message» и «Second message» попеременно каждую секунду.

В файл разметки *activity_main.xml* добавим следующий код:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/tv1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:gravity="center"
        android:text="Welcome"
        android:textSize="50sp" />

    <Button
        android:id="@+id/btnStart"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Start"
        android:layout_below="@id/tv1"
        android:layout_centerHorizontal="true"
        />

</RelativeLayout>
```

В файл *MainActivity* добавим следующий код.

```
package ru.samsung.myitacademy.mdev.uithreadexample

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.widget.Button
import android.widget.TextView

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?)
    {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // элементы разметки
        val tv = findViewById<TextView>(R.id.tv1)
        val btn = findViewById<Button>(R.id.btnStart)
        val msg1 = "First message"
        val msg2 = "Second message"

        // слушатель для кнопки
        btn.setOnClickListener{
            // объявляем главный поток
            Thread(Runnable {
                while (true) {
                    // обновление TextView
                    runOnUiThread{ tv.text = msg1 }
                    // останавливаем поток на одну секунду
                    Thread.sleep(1000)

                    // обновление TextView
                    runOnUiThread{ tv.text = msg2 }

                    // останавливаем поток на одну секунду
                    Thread.sleep(1000)
                }
            }).start()
        }
    }
}
```

В слушателе в основном потоке создается бесконечный цикл, и с использованием UI-потока текст в виджете изменяется каждую секунду. При этом цикл `while` должен быть объявлен внутри потока. Если поток объявлен внутри цикла, программа работать не будет.

3. Синхронизация потоков

Использование асинхронных потоков может привести к ошибочному выполнению. Для разрешения таких проблем используется синхронизация. Механизм синхронизации основывается на концепции монитора.

Монитор — это специальный механизм, обеспечивающий управление взаимодействием процессов и их состоянием. Монитор можно представить себе как ключ от комнаты сейфовых ячеек в банке, а потоки можно представить как клиентов в банке. Когда первый клиент получил ключ от комнаты, дверь за ним закрывается, и все остальные желающие работать с сейфами ждут, пока первый клиент не вернет ключ. Это называется «поток захватил монитор». Когда первый клиент вышел, ключ (монитор) может быть передан следующему клиенту. Это называется «поток освободил монитор».

В отличие от Java, Kotlin не имеет ключевого слова `synchronized`. Следовательно, для синхронизации нескольких фоновых потоков используется аннотация `@Synchronized` или встроенная функция стандартной библиотеки `synchronized()`.

```
// синхронизированная
@Synchronized fun myFunction() {
}

fun myOtherFunction() {
    // синхронизированный блок
    synchronized(this) {
    }
}
```

Аннотация `@Synchronized` и функция `synchronized()` используют концепцию блокировки монитора. Вы можете рассматривать монитор как особый токен, который поток может получить или заблокировать для получения монопольного доступа к объекту.

4. Корутины в Kotlin

Корутины — это новый способ написания асинхронного, неблокирующего кода. Корутины можно представить как облегчённый поток. Так же как и потоки, корутины могут работать параллельно, взаимодействовать между собой, ожидать друг друга. Ключевым отличием является тот факт, что корутины легковесны. Аналогичный код с созданием и запуском потока потребует много больше памяти. Корутины необходимо использовать в случае, если требуется скачать что-то из сети, извлечь данные из базы данных или просто выполнить долгие вычисления и при этом не заблокировать интерфейс пользователю.

Поддержка корутин встроена в Kotlin, но все классы и интерфейсы находятся в отдельной библиотеке. kotlinx.coroutines - это обширная библиотека, разработанная компанией JetBrains, которая содержит в себе множество сопрограмм. Для их использования нужно добавить зависимость в gradle:

```
// x.x.x версия корутин
dependencies
{
  implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:x.x.x"
  implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:x.x.x"
}
```

Главным отличительным признаком сопрограмм является то, что они могут быть приостановлены без блокирования потока. Еще одно отличие заключается в том, что сопрограммы не могут быть приостановлены на произвольной инструкции, а только в так называемых точках остановки (приостановки), которые вызываются в специально маркируемых функциях. Приостановка происходит в случае вызова функции со специальным модификатором `suspend`:

```
suspend fun doSomething(foo: Foo): Bar {
    ...
}
```

Такие функции называются функциями остановки (приостановки), поскольку их вызовы могут приостановить выполнение сопрограммы. Функции остановки могут иметь параметры и возвращать значения точно так же, как и все обычные функции, но они могут быть вызваны только из сопрограмм или других функций остановки. Функции остановки не могут быть вызваны из обычной функции, поэтому для них предусмотрено несколько специальных функции запуска сопрограммы, которые позволяют вызывать функцию остановки из обычной области, не требующей приостановки:

- `runBlocking`: запускает новую сопрограмму и блокирует текущий поток до его завершения,
- `launch`: запускает новую сопрограмму и возвращает ссылку на нее как на объект класса `Job`,
- `async`: запускает новую сопрограмму и возвращает ссылку на нее как объект `Deferred <T>`. Он должен использоваться вместе с функцией `await`, которая ожидает результата, не блокируя поток.

Пример

Сопрограммы используют внутренние фоновые потоки, поэтому по умолчанию они не запускаются в потоке пользовательского интерфейса приложения Android. Следовательно, если вы попытаетесь изменить содержимое пользовательского интерфейса вашего приложения из сопрограммы, вы столкнетесь с ошибкой во время выполнения. К счастью, запустить сопрограмму в потоке пользовательского интерфейса довольно просто: вам просто нужно передать объект UI в качестве аргумента вашему билдеру сопрограмм. Давайте работу сопрограммы на простом примере. Следующий фрагмент кода приводит к тому, что нажатие на кнопку должно изменить текст на экране после определенной задержки, не блокируя поток, в котором он выполняется, во время ожидания.

```
class SimpleCoroutinesActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        //init stuff

        fab.setOnClickListener {
            launch(UI) {
                setTextAfterDelay(2, "Hello from a coroutine!")
            }
        }
    }

    private suspend fun setTextAfterDelay(seconds: Long, text: String) {
        delay(seconds, TimeUnit.SECONDS)
        textView.text = text
    }
}
```

В `OnClickListener` мы используем `launch` и передаем UI типа `CoroutineDispatcher` в качестве его параметра. Параметр UI обеспечивает выполнение блока кода в основном потоке.

Внутри блока `launch` мы вызываем функцию остановки `setTextAfterDelay`, которая определена ниже. Внутри нашей функции остановки вызывается функция `delay` , которая задерживает сопрограмму, не блокируя поток. По прошествии определенного времени функция продолжает выполнение и устанавливает текст в `TextView`.

Возвращаемым значением функции `launch()` является объект `Job`, который можно использовать для управления сопрограммой. Например, вы можете вызвать его метод `join()`, чтобы дождаться завершения сопрограммы. Точно так же вы можете вызвать его метод `cancel()`, чтобы немедленно отменить сопрограмму.

Использование функции `launch()` очень похоже на создание нового потока с объектом `Runnable`, главным образом потому, что вы не можете вернуть из него никакого значения. Если вы хотите иметь возможность вернуть значение из вашей сопрограммы, вы должны создать его с помощью функции `async()`.

Функция `async()` возвращает объект `Deferred`, который, как и объект `Job`, позволяет вам управлять сопрограммой. Однако это также позволяет вам использовать функцию `await()` для ожидания результата сопрограммы без блокировки текущего потока.

Например, рассмотрим следующие сопрограммы, которые используют функцию приостановки `fetchWebsiteContents()` и возвращают длины содержимого двух разных адресов веб-страниц:

```
val jobForLength1 = async {
    fetchWebsiteContents("https://myitacademy.ru/partners/").length
}

val jobForLength2 = async {
    fetchWebsiteContents("https://myitacademy.ru/news/").length
}
```

С помощью приведенного выше кода обе сопрограммы запустятся немедленно и будут работать параллельно.

Если вы теперь хотите использовать возвращенные длины, вы должны вызвать метод `await()` для обоих объектов `Deferred`. Однако, поскольку метод `await()` тоже является функцией приостановки, вы должны убедиться, что вызываете его из другой сопрограммы.

В следующем коде показано, как вычислить сумму двух длин, используя новую сопрограмму, созданную с помощью функции `launch()` и отобразить эту сумму в виджете `myTextView`:

```
launch(UI) {
    val sum = jobForLength1.await() + jobForLength2.await()
    myTextView.text = "Downloaded $sum bytes!"
}
```

Приведенный выше код на первый взгляд может показаться обыденным, но он не только может ожидать завершения двух фоновых операций без использования обратных вызовов, но и может делать это в потоке пользовательского интерфейса приложения, не блокируя его.

Далее приведем пример, где с помощью функции `suspend delay()`, которая является неблокирующим эквивалентом метода `Thread.sleep()` организуем движение виджета в цикле. Ниже приведен пример сопрограммы, которая увеличивает координату x виджета `TextView` каждые 400 мс, создавая эффект, похожий на выделение:

```
launch(UI) {
    while(myTextView.x < 800) {
        myTextView.x += 10
        delay(400)
    }
}
```

Конечно это далеко не полный перечень применения сопрограмм в программировании Android приложений. Корутины активно задействуются совместно с архитектурными компонентами.

Резюмируя вышеизложенное стоит отметить, что при разработке приложений для Android крайне важно выполнять длительные операции в фоновых потоках. Для этого целесообразно задействовать корутины (сопрограммы). Чтобы узнать больше о сопрограммах, вы можете обратиться к официальной документации.

5. Пример 4.1

Разработаем приложение для загрузки картинки из интернета использованием корутин. На экране приложения будет отображаться кнопка "Download Image", виджет `ProgressBar` для отображения процесса загрузки, `ImageView` для отображения картинки, а также два `TextView` для отображения URL и URI картинки.

Разметка главной активности может выглядеть следующим образом:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="8dp"
        android:text="Download Image"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <ProgressBar
        android:id="@+id/progressBar"
        style="?android:attr/progressBarStyle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:visibility="invisible"
        app:layout_constraintBottom_toBottomOf="@+id/button"
        app:layout_constraintStart_toEndOf="@+id/button"
        app:layout_constraintTop_toTopOf="@+id/button" />

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="0dp"
        android:layout_height="350dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="1.0"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/tvDownload"
        tools:srcCompat="@tools:sample/avatars" />

    <TextView
        android:id="@+id/tvDownload"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        tools:text="Download URL"
        android:padding="8dp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/button" />

    <TextView
        android:id="@+id/tvSaved"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginTop="8dp"
        tools:text="Saved Uri"
        android:padding="8dp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/imageView" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Для использования корутин в файл gradle необходимо добавить зависимости:

```
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.7'
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.7'
```

Так как в приложении будет происходить загрузка картинки из Интернета, в файл манифеста проекта необходимо добавить строчку:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Код *MainActivity.kt* будет следующим:

```
package ru.samsung.myitacademy.mdev.coroutineexample

import android.content.Context
import android.content.ContextWrapper
import android.graphics.Bitmap
import android.graphics.BitmapFactory
import android.net.Uri
import android.os.Bundle
import android.view.View
import androidx.appcompat.app.AppCompatActivity
import kotlinx.android.synthetic.main.activity_main.*
import kotlinx.coroutines.*
import java.io.File
import java.io.FileOutputStream
import java.io.IOException
import java.io.OutputStream
import java.net.URL
import java.util.*


class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val context = this

        // URL изображения для загрузки
        val urlImage:URL = URL("https://github.com/emityakov/4.1_pictures/blob/master/pic_4.1.png?raw=true")

        // показать URL изображения в текстовом поле
        tvDownload.text = urlImage.toString()


        button.setOnClickListener {
            it.isEnabled = false
            progressBar.visibility = View.VISIBLE

            // асинхронная задача для получения / загрузки изображения с URL-адреса
            val result: Deferred<Bitmap?> = GlobalScope.async {
                urlImage.toBitmap()
            }

            GlobalScope.launch(Dispatchers.Main) {
                // получаем загруженное изображение
                val bitmap : Bitmap? = result.await()

                // если скачалось, сохраняем во внутреннем хранилище
                bitmap?.apply {
                    // получаем сохраненное изображение
                    val savedUri : Uri? = saveToInternalStorage(context)

                    // отображаем изображение
                    imageView.setImageURI(savedUri)

                    // выводим сохраненный URI изображения в текстовом представлении
                    tvSaved.text = savedUri.toString()
                }

                it.isEnabled = true
                progressBar.visibility = View.INVISIBLE
            }
        }
    }
}


// функция расширения для получения / загрузки растрового изображения с URL-адреса
fun URL.toBitmap(): Bitmap?{
    return try {
        BitmapFactory.decodeStream(openStream())
    }catch (e:IOException){
        null
```

```
    }
}


// функция расширения для сохранения изображения во внутренней памяти
fun Bitmap.saveToInternalStorage(context : Context):Uri?{
    // получить экземпляр оболочки контекста
    val wrapper = ContextWrapper(context)

    // инициализация нового файла
    // нижняя строка возвращает каталог во внутренней памяти
    var file = wrapper.getDir("images", Context.MODE_PRIVATE)

    // создаем файл для сохранения изображения
    file = File(file, "${UUID.randomUUID()}.jpg")

    return try {
        //получаем поток вывода файла
        val stream: OutputStream = FileOutputStream(file)

        // сжимаем растровое изображение
        compress(Bitmap.CompressFormat.JPEG, 100, stream)

        // освобождаем поток
        stream.flush()

        // закрываем поток
        stream.close()

        // возвращаем сохраненное uri изображения
        Uri.parse(file.absolutePath)
    } catch (e: IOException){ // catch the exception
        e.printStackTrace()
        null
    }
}
```

После запуска приложения и нажатия кнопки получим следующий результат:

Начать тур для пользователя на этой странице