Android
-
asynchrónnosť

Peter Borovanský
KAI, I-18

borovan 'at' ii.fmph.uniba.sk

AsyncTask
Kotlin coroutines

Hádanka 1

```
Log.d(TAG, "Start")
val list = listOf<Int>(1,2,3,4,5,6,7,8,9,10)
val newList = list.stream().map {
    Thread.sleep(1000)
    it*it              // return it * it
}
Log.d(TAG, "End")
newList.forEach {// výpis kolekcie
  Log.d(TAG, it.toString())
}
```

```
08:59:32.832 Start
08:59:32.834 End   Start+0sec.
08:59:33.839 1
08:59:34.841 4
08:59:35.842 9
08:59:36.844 16
08:59:37.846 25
08:59:38.849 36
08:59:39.851 49
08:59:40.854 64
08:59:41.856 81
08:59:42.858 100
```

stream bez .collect() je *lenivá* kolekcia

Hádanka 2

```
Log.d(TAG, "Start")
val list = listOf<Int>(1,2,3,4,5,6,7,8,9,10)
val newList = list.stream().map {
    Thread.sleep(1000)
    it*it                // return it * it
} .collect(Collectors.toList())
Log.d(TAG, "End")
newList.forEach {// výpis kolekcie
  Log.d(TAG, it.toString())
}
```

```
09:02:23.363  Start
09:02:33.389  End   Start+10sec.
09:02:33.389  1
09:02:33.389  4
09:02:33.389  9
09:02:33.389  16
09:02:33.389  25
09:02:33.389  36
09:02:33.390  49
09:02:33.390  64
09:02:33.390  81
09:02:33.390  100
```

Hádanka 3

```
Log.d(TAG, "Start")
val list = listOf<Int>(1,2,3,4,5,6,7,8,9,10)
val newList = list.parallelStream().map {
    Thread.sleep(1000)
    it*it              // return it * it
} .collect(Collectors.toList())
Log.d(TAG, "End")
newList.forEach {// výpis kolekcie
  Log.d(TAG, it.toString())
}
```

parallelStream používa
toľko paralelizmu, koľko je #cores

```
Runtime.getRuntime()
.availableProcessors() == 4
```

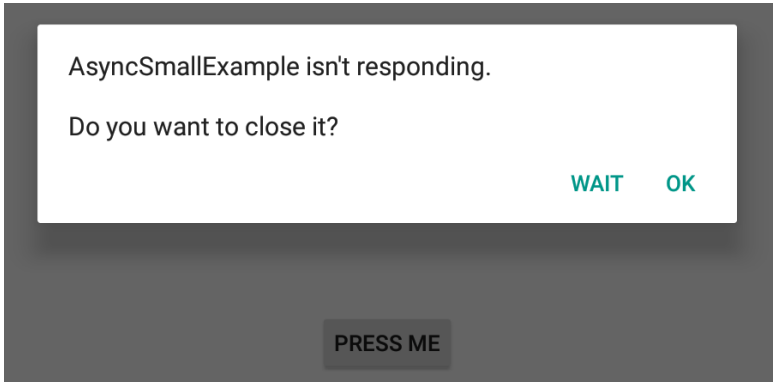
```
09:04:06.410 Start
09:04:09.420 End   Start+3sec.
09:04:09.420 1
09:04:09.420 4
09:04:09.420 9
09:04:09.420 16
09:04:09.420 25
09:04:09.420 36
09:04:09.421 49
09:04:09.421 64
09:04:09.421 81
09:04:09.422 100
```

Asynchrónne operácie

- nie je možné robiť časovo náročné operácie v hlavnom vlákne aplikácie
  - extra komplikovaný (matematický) výpočet
  - simuláciu procesu spomaľovanú napr. Thread.sleep(…)
  - trvajúce požiadavky (napr. http/sql-request), ktoré môžu trvať netriviálne dlho

- Takýto kód zablokuje hlavné vlákno, a ak vyvoláte GUI eventy (napr. pochabým klikaním v priebehu 20s), správca aplikácií usúdi, že aplikácia je mŕtva zavre ju

```
fun buttonClick(view: View) {
  var i = 0
  while (i <= 20) {
    try {
        Thread.sleep(1000)
        i++
    }
    catch (e: Exception) {
        e.printStackTrace()
  }
}
```



Project:AsyncSmallExample.zip

Async Task
(doInBackground)

Parametrizovaná trieda AsyncTask je thread-wrapper riešením problému od API-3

```
                                typ parametrov, type progresu, typ výsledku
private inner class MyTask : AsyncTask<String, Int, String>() {

   override fun onPreExecute() { … }  // vykoná sa pred doInBackground
   // celé jadro toho, čo sa má vykonávať v extra vlákne
   override fun doInBackground(vararg params: String): String {
      while (i in 0..20) {
        try {
           Thread.sleep(1000)
           publishProgress(i)
        } catch (e: Exception) { … }
        }
        return "Button Pressed"
    }

    override fun onProgressUpdate(vararg values: Int?) { … }

    override fun onPostExecute(result: String) { … } // po doInBackgr.
}
```

Project:AsyncSmallExample.zip

Async Task
(onPre/PostExecue)

```
private inner class MyTask : AsyncTask<String, Int, String>() {
    var color : Int = Color.BLACK
    override fun onPreExecute() {
        color =  ... Random Color ...
    }
    override fun doInBackground(vararg params: String): String { ...}

            // varargs je variabilný počet argumentov, ako ... v Jave
    override fun onProgressUpdate(vararg values: Int?) {
        myTextView.setTextColor(color)          // beží v main threade
        val counter = values.get(0)
        myTextView.text = "Counter = $counter"
    }

    override fun onPostExecute(result: String) {  // "Button Pressed"
        myTextView.setTextColor(color)
        myTextView.text = result
    }
}
```

Project:AsyncSmallExample.zip

Async Task
(spustenie)

Štandardne sa rôzne inštancie AsyncTask spúšťajú sériovo, kým nedobehne jedna, ostatné čakajú vo fronte

```
val task1 = MyTask().execute()  // serial run of AsyncTask
```

Ak ich chceme spustiť viacero a paralelne, tak cez POOL_EXECUTOR

```
task = MyTask().executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR)
```

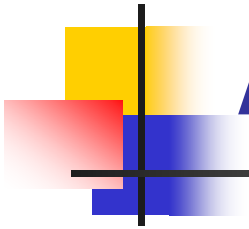
Ale počet paralelne bežiacich AsyncTaskov je limitovaný, v závislosti od počtu jadier CPU

```
val cpu_cores = Runtime.getRuntime().availableProcessors()
```

Reálne väčším problémom, že napriek popularite a jednoduchosti používania AsyncTask je od Android 11 AsyncTask zastaralý (*deprecated*)
https://www.xda-developers.com/asynctask-deprecate-android-11/amp/?__twitter_impression=true

Z toho zatiaľ nie je jasné, že ho Google odstráni, ale ...

Project:AsyncSmallExample.zip



Alternatívy

Čo je alternatíva:

- RX-library
- Java's Concurrency framework
- Kotlin coroutines od verzie Kotlin 1.3

build.gradle:

- `implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.2"`

import

- `import kotlinx.coroutines.*`

tutorial:

- https://kotlinlang.org/docs/tutorials/coroutines/coroutines-basic-jvm.html

Corutina

- je odľahčené vlákno
- non-preemptive multitasking
- 1958 zaviedli Donald Knuth a Melvin Conway
- vyskytujú sa v iných jazykoch, C#, javascript

`suspend` je modifikátor funkcie, ktorá vykonávaná v corutine môže byť pozdržaná

`await()` je čaká na hodnotu výpočtu bez blokovania corutiny.

Corutina

(Spustenie – blokujúce, neblokujúce)

.launch spustí novú corutinu podobne ako .start() Thread
.join počká na dokončenie spustenej korutiny, ako Thread

```
Log.d(TAG, "Start")
GlobalScope.launch {   // Start a coroutine, non-blocking
    delay(1000)        // wait 1s.
    Log.d(TAG, "Hello")
}
Thread.sleep(3000)     // wait for 3s.
Log.d(TAG, "Stop")
runBlocking {          // Start a coroutine, blocking
    delay(4000L)
}
Log.d(TAG, "Finish")
```

```
21:22:18.220 Start
21:22:19.225 Hello     Start+1sec.
21:22:21.222 Stop      Start+3sec.
21:22:25.225 Finish    Start+7sec.
```

https://simply-how.com/kotlin-coroutines-by-example-guide

Corutina

(suspend)

```
    Log.d(TAG, "Start")
    runBlocking {
        printHello()
    }
    Log.d(TAG, "Finish")


suspend fun printHello() {
    delay(1000L)
    Log.d(TAG,"Hello")
}
```

```
21:27:34.083 Start
21:27:35.089 Hello     Start+1sec.
21:27:35.089 Finish    Start+1sec.
```

https://simply-how.com/kotlin-coroutines-by-example-guide

Corutina

(suspend)

```
Log.d(TAG,"The main program is started")
GlobalScope.launch {
    Log.d(TAG,"Background processing started")
    delay(1000L)
    Log.d(TAG,"Background processing finished")
}
Log.d(TAG,"The main program continues")
runBlocking {
    delay(2000L)
    Log.d(TAG,"The main program is finished")
}
```

```
21:33:51.083 The main program is started
21:33:51.084 Background processing started
21:33:51.084 The main program continues
21:33:52.090 Background processing finished      Start+1sec.
21:33:53.086 The main program is finished        Start+2sec.
```

https://simply-how.com/kotlin-coroutines-by-example-guide

Corutina

(async/await)

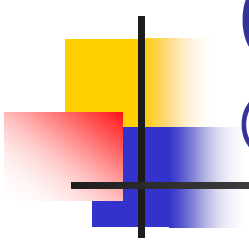
```
21:38:31.369 Awaiting computations...
21:38:32.375 Computation1 finished    Start+1sec.
21:38:33.376 Computation2 finished    Start+2sec.
21:38:33.378 The result is 3          Start+2sec.
```

.async spustí novú corutinu ktorá počíta nejaký výsledok

.await čaká na tento výsledok

```
 runBlocking {
        val result1 = async { computation1() }
        val result2 = async { computation2() }
        Log.d(TAG, "Awaiting computations...")
        val result = result1.await() + result2.await()
        Log.d(TAG, "The result is $result")
} }
suspend fun computation1(): Int {
    delay(1000L) // simulated computation
    Log.d(TAG,"Computation1 finished")
    return 1 }
suspend fun computation2(): Int {
    delay(2000L)
    Log.d(TAG,"Computation2 finished")
    return 2 }
```

https://simply-how.com/kotlin-coroutines-by-example-guide



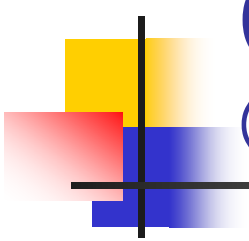
Corutina

(cancel)

```
21:44:52. Processing 0 ...
21:44:53. Processing 1 ...
21:44:54. Processing 2 ...
21:44:55. Processing 3 ...
21:44:56. Processing 4 ...
21:44:57. Processing 5 ...
21:44:58. Processing 6 ...
21:44:59. Processing 7 ...
21:45:00. Processing 8 ...
21:45:01. Processing 9 ...
21:45:02. main: The user requests the cancellation
21:45:02. main: The batch is cancelled
```

```
runBlocking {
    val job = launch { // Emulate some batch processing
        repeat(30) { i ->
            Log.d(TAG, "Processing $i ...")
            delay(1000L)
        }
    }
    delay(10000L)
    Log.d(TAG, "main: The user requests the cancellation")
    job.cancelAndJoin()
       // cancel the job and wait for it's completion
    Log.d(TAG, "main: The batch is cancelled")
}
```

https://simply-how.com/kotlin-coroutines-by-example-guide



Corutina

(withTimeout)

```
21:47:57 Processing 0 ...
21:47:58 Processing 1 ...
21:47:59 Processing 2 ...
21:48:00 Processing 3 ...
21:48:01 Processing 4 ...
21:48:02 Processing 5 ...
21:48:03 Processing 6 ...
21:48:04 Processing 7 ...
21:48:05 Processing 8 ...
21:48:06 Processing 9 ...
21:48:07 The processing return status is: null
```

```
runBlocking {
    val status = withTimeoutOrNull(10000L) {
        repeat(30) { i ->
            Log.d(TAG, "Processing $i ...")
            delay(1000L)
        }
        "Finished"
    }
    Log.d(TAG, "The processing return status is: $status")
}
```

https://simply-how.com/kotlin-coroutines-by-example-guide