链滴

Android 线程池使用介绍

作者：RustFisher

原文链接：https://ld246.com/article/1631345469189

来源网站：链滴

本文主要使用kotlin，讨论Android开发中的线程池用法。

我们想使用线程的时候，可以直接创建子线程并启动

```
Thread { Log.d("rfDev", "rustfisher said: hello") }.start()
```

如果有大量的异步任务，不想每次都创建子线程。有没有什么把子线程统一管理的方法?

遇到这样的情况，我们可以考虑线程池。线程池解决两个问题：需要执行大量异步任务的时候，减轻个异步任务的调用开销，提高性能。另外它还能够限制和管理子线程。每个**ThreadPoolExecutor**都护了一些统计数据，例如已执行的任务数量。

有大量异步任务的时候，可以考虑使用线程池。

预置线程池

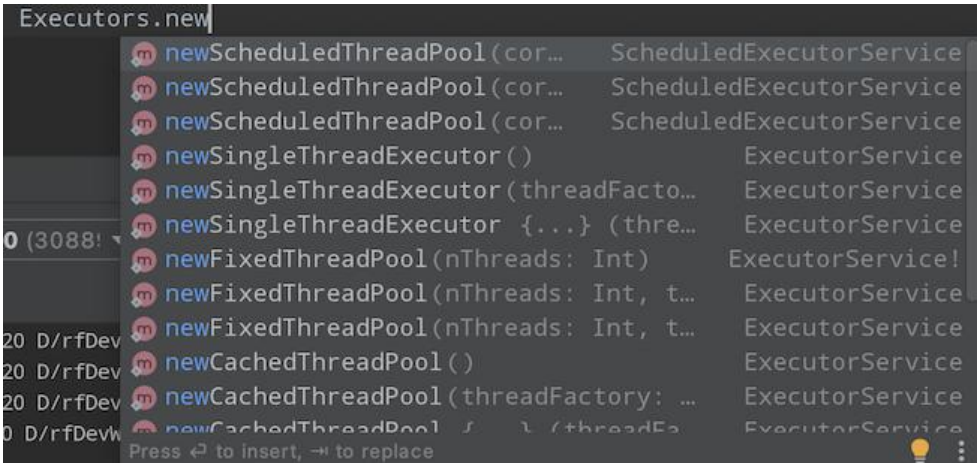
代码参考 Android API 29

ThreadPoolExecutor提供了很多参数，方便开发者调控。线程池的设计者建议开发者使用以下几个厂方法，Android中主要有5种

- newCachedThreadPool() 不限制数量的线程池，能自动回收线程
- newFixedThreadPool(int nThreads) 固定数量的线程池
- newSingleThreadExecutor() 单一的子线程
- newScheduledThreadPool(int corePoolSize) 能执行延时任务或者周期性任务
- newWorkStealingPool() 工作窃取线程池

实际上我们在Android Studio里输入 Executors.new的时候，会跳出很多个提示选项。

Executors.new 的智能提示



可缓存线程池

用 Executors.newCachedThreadPool获得一个可缓存线程池对象，然后让它执行任务。

```
val tp: ExecutorService = Executors.newCachedThreadPool()
tp.submit { Log.d(TAG, "rustfisher: cached线程池执行任务 3") }
```

可缓存线程池会在需要的时候创建新的子线程。当原有的线程可用的时候，会复用现有线程。

这个机制适用于执行多个短期异步任务。任务比较小，但是数量大。

调用 execute方法会先尝试复用已有的可用线程。如果当前没有线程，会新建一个线程并把它添加到里。

超过60秒没有使用的线程会被停止并移除。因此即便长时间不用这个线程池，也不会造成多大的开销。

定长线程池

使用 newFixedThreadPool(int nThreads)示例

```
val fixedTp: ExecutorService = Executors.newFixedThreadPool(4)
fixedTp.submit { Log.d(TAG, "rustfisher 定长线程池执行任务") }
```

静态方法里传入了一个int参数 nThreads，表示最大线程数量。

如果当前所有线程都在忙，又有新的任务添加进来。那么任务会在队列中等待，直到有可用的线程来理任务。

如果有的线程遇到错误而停止了，要执行任务的话，会创建新的线程补上位置。

池里的线程会一直存活，直到线程池停止（ExecutorService#shutdown）。

单一线程池

```
val singleTp: ExecutorService = Executors.newSingleThreadExecutor()
singleTp.submit { Log.d(TAG, "单一线程池执行任务") }
```

只拥有1个子线程。任务队列不限制任务数量。如果线程遇到问题停止了，接下来又要处理任务时，新建一个线程来处理。

它能保证任务会按顺序处理，同一时间只能处理1个任务。

单一线程池创建后，不能动态修改线程数量。不像 newFixedThreadPool(1)的定长线程池可以修改程数。

计划任务线程池

```
val scheduleTp: ScheduledExecutorService = Executors.newScheduledThreadPool(3)
```

计划任务线程池能够执行延迟任务和周期任务。

延迟任务

需要设定延时与时间单位

```
scheduleTp.schedule({ Log.d(TAG, "计划任务1 runnable") }, 300, TimeUnit.MILLISECONDS)
scheduleTp.schedule(Callable { Log.d(TAG, "计划任务2 callable") }, 400, TimeUnit.MILLISECONDS)
```

周期任务

主要涉及到2个方法 scheduleAtFixedRate和 scheduleWithFixedDelay。

假设任务时间小于周期时间，则按给定周期时间来进行。这两个方法表现一致。

假设任务执行时间大于周期时间，这两个方法有点不同

- scheduleAtFixedRate执行完上一个任务后，用时超过了周期时间，会立刻执行下一个任务。
- scheduleWithFixedDelay在上一个任务执行完毕后，还会等待周期时间，再去执行下一个任务。

工作窃取线程池

Android SDK 大于等于24，有一种新的线程池，暂且称为“工作窃取线程池”，或者叫“灵活调度程池”。

```
if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.N) {
    Executors.newWorkStealingPool()
}
```

线程池维护足够的线程来支持给定的并行度（parallelism level），可能会用多个队列来减少争用。并行度对应的是活跃的线程最大数，或者能处理任务的线程最大数。

线程的实际数量可能会动态增减。工作窃取线程池不保证按提交顺序来处理任务。

执行任务

执行任务的时候可以传入**Runnable**和**Callable**，前面用的都是**Runnable**。

用**Callable**的例子

```
tp.submit(Callable { "OK" })
```

无返回值任务的调用

无返回值任务用**Callable**和**Runnable**都行。

```
val tp: ExecutorService = Executors.newCachedThreadPool()
tp.submit { Log.d(TAG, "rustfisher: cached线程池submit runnable") }
tp.execute { Log.d(TAG, "rustfisher: cached线程池execute runnable") }
tp.submit(Callable { Log.d(TAG, "rustfisher: cached线程池submit callable") })

tp.shutdown() // 最后记得用完后停掉线程池
```

有返回值任务的调用

有返回值的任务需要**Callable**接口。

submit

调用 submit方法时会返回一个**Future**对象。通过**Future**的 get()方法可拿到返回值。这里需要注意 get()是阻塞的，完成任务后，能拿到返回值。

```
val tp: ExecutorService = Executors.newCachedThreadPool()
val future = tp.submit(Callable {
    return@Callable "callable的返回值"
})
Log.d(TAG, "future get之前 isDone: ${future.isDone}, isCancelled: ${future.isCancelled}")
val res = future.get()
Log.d(TAG, "future get之后 isDone: ${future.isDone}, isCancelled: ${future.isCancelled}")
Log.d(TAG, "future get: $res")
```

运行log

```
future get之前 isDone: false, isCancelled: false
future get之后 isDone: true, isCancelled: false
future get: callable的返回值
```

invokeAll

对于列表里的任务，可以使用 invokeAll(Collection<? extends Callable<T>> tasks)，返回一个**Future**的列表。

作为对比，给其中一个任务加上延时。

invokeAll示例

```
val tp: ExecutorService = Executors.newFixedThreadPool(5)
val callList = arrayListOf<Callable<String>>(
        Callable {
            Log.d(TAG, "task1 ${Thread.currentThread()}")
            return@Callable "rust"
        },
        Callable {
            Log.d(TAG, "task2 ${Thread.currentThread()}")
            Thread.sleep(1500) // 加上延时
            return@Callable "fisher"
        },
        Callable {
            Log.d(TAG, "task3 ${Thread.currentThread()}")
            return@Callable "列表里面的任务"
        },
)
Log.d(TAG, "invokeAll 准备提交任务")
val futureList = tp.invokeAll(callList)
Log.d(TAG, "invokeAll 已提交任务")
futureList.forEach { f ->
    Log.d(TAG, "任务列表执行结果 ${f.get()}") // 这里会阻塞 别在ui线程里get
}
```

运行log，可以看到提交任务后，经过延时，拿到了运行结果。注意看 invokeAll前后的时间。invokeAll会阻塞当前线程。使用的时候必须小心，不要在ui线程中调用。

```
2021-09-11 14:40:07.062 16914-16914/com.rustfisher.tutorial2020 D/rfDevTp: invokeAll 准备提交任务
2021-09-11 14:40:07.063 16914-19230/com.rustfisher.tutorial2020 D/rfDevTp: task1 Thread[pool-4-thread-1,5,main]
```

```
2021-09-11 14:40:07.063 16914-19231/com.rustfisher.tutorial2020 D/rfDevTp: task2 Thread[pool-4-thread-2,5,main]
2021-09-11 14:40:07.063 16914-19232/com.rustfisher.tutorial2020 D/rfDevTp: task3 Thread[pool-4-thread-3,5,main]
2021-09-11 14:40:08.563 16914-16914/com.rustfisher.tutorial2020 D/rfDevTp: invokeAll 已提任务
2021-09-11 14:40:08.563 16914-16914/com.rustfisher.tutorial2020 D/rfDevTp: 任务列表执行果 rust
2021-09-11 14:40:08.563 16914-16914/com.rustfisher.tutorial2020 D/rfDevTp: 任务列表执行果 fisher
2021-09-11 14:40:08.563 16914-16914/com.rustfisher.tutorial2020 D/rfDevTp: 任务列表执行果 列表里面的任务
```

提交了3个任务，在3个不同的子线程中执行。

invokeAny

invokeAny(Collection<? extends Callable<T>> tasks)也是接收**Callable**集合。

然后返回最先执行结束的任务的值，其它未完成的任务将被正常取消掉不会有异常。

invokeAny示例

```
val tp: ExecutorService = Executors.newCachedThreadPool()
val callList = arrayListOf<Callable<String>>(
        Callable {
            Thread.sleep(1000) // 设计延时
            return@Callable "rust"
        },
        Callable {
            Thread.sleep(400)
            return@Callable "fisher"
        },
        Callable {
            Thread.sleep(2000)
            return@Callable "列表里面的任务"
        },
)
Log.d(TAG, "invokeAny 提交任务")
val res = tp.invokeAny(callList)
Log.d(TAG, "执行结果 $res")
```

```
2021-09-11 14:04:55.253 14066-14066/com.rustfisher.tutorial2020 D/rfDevTp: invokeAny 提任务
2021-09-11 14:04:55.654 14066-14066/com.rustfisher.tutorial2020 D/rfDevTp: 执行结果 fisher
```

观察log可以看到，最后执行的是“fisher”这个任务。

停止线程池

使用完毕后，记得终止线程池

```
/*ExecutorService*/ shutdown()
shutdownNow()
```

shutdown()在已提交的任务后面创建一个停止命令，并且不再接受新的任务。如果线程池已经停止，调用这个方法将不生效。

shutdownNow()方法尝试停止所有执行中的任务，停下等待中的任务。并且返回等待执行的任务列表ist<Runnable>。

参考：

- 线程介绍 https://an.rustfisher.com/java/thread/thread-class-intro/
- Android使用线程池 https://an.rustfisher.com/android/concurrent/thread-pool-use-intro/