

个人资料



一切都是铺垫



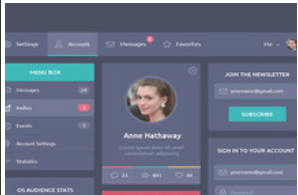
访问：15302次
积分：252
等级：BLOG > 2
排名：千里之外

原创：6篇 转载：29篇
译文：0篇 评论：0条

文章搜索



app外包公司



ui



阅读排行

- Android 实现监听开机 (2369)
- android PackageMan (1567)
- cocos2dx-lua 在coco (1167)
- Java--进程间通讯的 (970)

异步赠书：10月Python畅销书升级 【线路图】人工智能到底学什么？！ 程序员9月书讯 节后荐书：Python、PyQt5、Kotlin（评论送书）

Android中的Thread与AsyncTask的区别

标签：[asynctask](#) [android](#)

2015-07-09 17:22 254人阅读 评论(0) 收藏 举报

分类：[android开发 \(27\)](#)

Android 原生的 AsyncTask.java 是对线程池的一个封装，使用其自定义的 Executor 来调度线程的执行方式（并发还是串行），并使用 Handler 来完成子线程和主线程数据的共享。

预先了解 AsyncTask，必先对线程池有所了解。

一般情况下，如果使用子线程去执行一些任务，那么使用 new Thread 的方式会很方便的创建一个线程到主线程和子线程的通信，我们将使用 Handler（一般需要刷新 UI 的适合用到）。

如果我们创建大量的（特别是在短时间内，持续的创建生命周期较长的线程）野生线程，往往会出现问题：

1. 每个线程的创建与销毁（特别是创建）的资源开销是非常大的；
2. 大量的子线程会分享主线程的系统资源，从而会使主线程因资源受限而导致应用性能降低。

各位开发一线的前辈们为了解决这个问题，引入了线程池（ThreadPool）的概念，也就是把这些野生的线程圈养起来，统一的管理他们。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

那么线程池是如何使用的呢？

我们可以通过ThreadPoolExecutor来创建一个线程池。

```
new ThreadPoolExecutor(corePoolSize, maximumPoolSize, keepAliveTime, TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>(), handler);
```

创建一个线程池需要输入几个参数：

corePoolSize（线程池的基本大小）：当提交一个任务到线程池时，如果当前线程池中的线程数小于corePoolSize，那么线程池会自动创建新的线程来执行任务。如果调用了线程池的 **prestartAllCoreThreads** 方法，线程池会预先创建所有核心线程。
runnableTaskQueue（任务队列）：用于保存等待执行的任务的队列。



- 如何在Swift中创建自: (876)
- Android 实现纵向浏览 (855)
- Android Handler实现 (456)
- android Spinner控件i (373)
- Android Material Des (363)
- 【COCOS2DX-LUA】 (347)

- 评论排行
- 在 Swift 中使用 Coco (0)
 - cocos2dx-lua 在coco (0)
 - Android ContentProv (0)
 - android 内存优化 (0)
 - Android Handler实现 (0)
 - android j使用JNI实现 (0)
 - Android Touch事件传 (0)
 - Android 实现纵向浏览 (0)
 - Android 实现监听开机 (0)
 - Android 实现根据手 (0)

- 推荐文章
- * 【观点】第二十三期：程序员应该如何积累财富？
 - * Android检查更新下载安装
 - * 动手打造史上最简单的Recycleview 侧滑菜单
 - * TCP网络通讯如何解决分包粘包问题



app外包公司



ui

ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按 FIFO（先进先出）原则对元素进行排序。

LinkedBlockingQueue：一个基于链表结构的阻塞队列，此队列按 FIFO（先进先出）排序元素，吞吐量通常要高于 **ArrayBlockingQueue**。静态工厂方法**Executors.newFixedThreadPool()** 使用了这个队列。

SynchronousQueue：一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于**LinkedBlockingQueue**，静态工厂方法 **Executors.newCachedThreadPool** 使用了这个队列。

PriorityBlockingQueue：一个具有优先级的无限阻塞队列。

maximumPoolSize（线程池最大大小）：线程池允许创建的最大线程数。如果队列满了，并且已创建的线程数小于最大线程数，则线程池会再创建新的线程执行任务。值得注意的是如果使用了无界的任务队列这个参数就没什么效果。

ThreadFactory：用于设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字。

RejectedExecutionHandler（饱和策略）：当队列和线程池都满了，说明线程池处于饱和状态，那么必须采取一种策略处理提交的新任务。这个策略默认情况下是 **AbortPolicy**，表示无法处理新任务时抛出异常。以下是JDK1.5提供的四种策略。

- AbortPolicy**：直接抛出异常。
- CallerRunsPolicy**：只用调用者所在线程来运行任务。
- DiscardOldestPolicy**：丢弃队列里最近的一个任务，并执行当前任务。
- DiscardPolicy**：不处理，丢弃掉。

当然也可以根据应用场景需要来实现 **RejectedExecutionHandler** 接口自定义策略。如写不能处理的任务。

keepAliveTime（线程活动保持时间）：线程池的工作线程空闲后，保持存活的时间。所以如果任务很多并且每个任务执行的时间比较短，可以调大这个时间，提高线程的利用率。

TimeUnit（线程活动保持时间的单位）：可选的单位有天（**DAYS**），小时（**HOURS**），分钟（**MINUTES**），毫秒（**MILLISECONDS**），微秒（**MICROSECONDS**，千分之一毫秒）和毫微秒（**NANOSECONDS**，千分之一微秒）。

如何向线程池提交线程任务呢？

1. 我们可以使用线程池的 **execute** 提交的任务，但是 **execute** 方法没有返回值，所以无法判断任务是否被线程池执行成功：

```
threadsPool.execute(new Runnable() {
    @Override
    public void run() {
        // TODO Auto-generated method stub
    }
});
```

2. 我们也可以使用 **submit** 方法来提交任务，它会返回一个 future,那么我们可以通过这个 future 来判断任务是否执行成功，通过 future 的 **get** 方法来获取返回值，**get** 方法会阻塞住直到任务完成，而使用 **get(long timeout, TimeUnit unit)** 方法则会阻塞一段时间后立即返回，这时有可能任务没有

关闭

```
Future<Object> future = executor.submit(harReturnVa
try {
    Object s = future.get();
} catch (InterruptedException e) {
    // 处理中断异常
} catch (ExecutionException e) {
    // 处理无法执行任务异常
} finally {
    // 关闭线程池
```



```
executor.shutdown();
}
```

线程池是如何关闭的呢？

ThreadPoolExecutor 提供了两个方法，用于线程池的关闭，分别是 shutdown()和shutdownNow()，其中：

- shutdown()：不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务；
- shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任

线程池的原理？

线程池中比较重要的规则：

corePoolSize 与 maximumPoolSize

由于 ThreadPoolExecutor 将根据 corePoolSize 和 maximumPoolSize 设置的边界自动调整池，方法 execute(java.lang Runnable) 中提交时：

1. 如果运行的线程少于 corePoolSize，则创建新线程来处理请求，即使其他辅助线程是空闲...
2. 如果设置的 corePoolSize 和 maximumPoolSize 相同，则创建的线程池是大小固定的，线程数与 corePoolSize 相同，当有新请求过来时，若 workQueue 未
3. 如果运行的线程多于 corePoolSize 而少于 maximumPoolSize，则仅当队列满时才创建新线程去处理请求；
4. 如果运行的线程多于 corePoolSize 并且等于 maximumPoolSize，若队列已经满了，则通过 RejectedExecutionHandler 所指定的策略来处理新请求；
5. 如果将 maximumPoolSize 设置为基本的无界值（如 Integer.MAX_VALUE），则允许池适应任意数量的并发任务

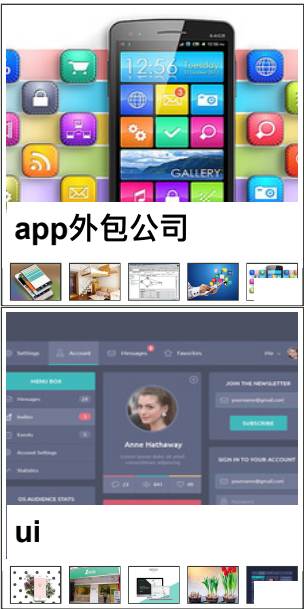
也就是说，处理任务的优先级为：

corePoolSize > workQueue > maximumPoolSize，如果三者都满了，使用 RejectedExecutionHandler 处理被拒绝的任务。

当池中的线程数大于 corePoolSize 的时候，多余的线程会等待 keepAliveTime 长的时间，如果无请求可处理就自行销毁。

workQueue线程池所使用的缓冲队列，该缓冲队列的长度决定了能略：

1. 直接提交。工作队列的默认选项是 SynchronousQueue，它将此，如果不存在可用于立即运行任务的线程，则试图把任务加入此策略可以避免在处理可能具有内部依赖性的请求集时出现锁。无界 maximumPoolSizes 以避免拒绝新提交的任务。当命令以超策略允许无界线程具有增长的可能性；



- 2. 无界队列。使用无界队列（例如，不具有预定义容量的 **LinkedBlockingQueue**）将导致在所有 **corePoolSize** 线程都忙时新任务在队列中等待。这样，创建的线程就不会超过**corePoolSize**（因此，**maximumPoolSize** 的值也就无效了）。当每个任务完全独立于其他任务，即任务执行互不影响时，适合于使用无界队列；例如，在 Web 页服务器中。这种排队可用于处理瞬态突发请求，当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性；
- 3. 有界队列。当使用有限的 **maximumPoolSizes** 时，有界队列（如 **ArrayBlockingQueue**）有助于防止资源耗尽，但是可能较难调整和控制。队列大小和最大池大小可能需要相互折衷：使用大型队列和小型池可以最大限度地降低 CPU 使用率、操作系统资源和上下文切换开销，但是可能导致人工降低吞吐量。如果任务频繁阻塞（例如，如果它们是 I/O 边界），则系统可能为超过您许可的更多线程安排时间。使用小型队列通常要求较大的池大小，CPU 使用率较高，但是可能遇到不可接受的调度开销，这样也会降低吞吐量。

ThreadFactory

使用 ThreadFactory 创建新线程。如果没有另外说明，则在同一个 ThreadGroup 中一律使用 Executors.defaultThreadFactory() 创建线程，并且这些线程具有相同的 NORM_PRIORITY 优先级和非守护进程状态。通过提供不同的 ThreadFactory，可以改变线程的名称、线程组、优先级、守护进程状态等等。如果执行 newThread 时 ThreadFactory 未能创建线程（返回 null），则执行程序将继续运行，但不能执行任何任务。

接下来我们看一下 ThreadPoolExecutor 中最重要的 **execute** 方法：

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    // 如果线程数小于基本线程数，则创建线程并执行当前任务
    if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) {
        // 如线程数大于等于基本线程数或线程创建失败，则将当前任务放到工作队列中。
        if (runState == RUNNING && workQueue.offer(command)) {
            if (runState != RUNNING || poolSize == 0)
                ensureQueuedTaskHandled(command);
        }
        // 如果线程池不处于运行中或任务无法放入队列，并且当前线程数量小于最大允许的线程数量，
        // 则创建一个线程执行任务。
        else if (!addIfUnderMaximumPoolSize(command))
            // 抛出RejectedExecutionException异常
            reject(command); // is shutdown or saturated
    }
}
```

线程池容量的动态调整？

ThreadPoolExecutor 提供了动态调整线程池容量大小的方法：**setCon**

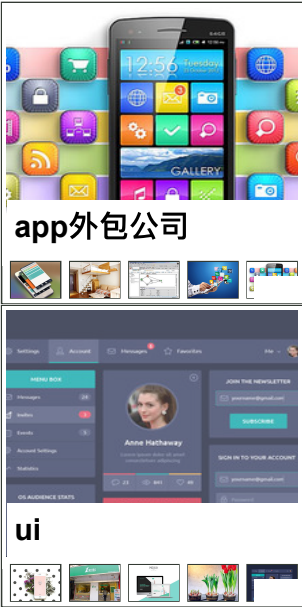
- setCorePoolSize**：设置核心池大小
- setMaximumPoolSize**：设置线程池最大能创建的线程数目大小

当上述参数从小变大时，**ThreadPoolExecutor** 进行线程赋值，还可能

线程池的监控？

通过线程池提供的参数进行监控。线程池里有一些属性在监控线程池的

taskCount：线程池需要执行的任务数量。



completedTaskCount: 线程池在运行过程中已完成的任务数量。小于或等于 **taskCount**。

largestPoolSize: 线程池曾经创建过的最大线程数量。通过这个数据可以知道线程池是否满过。如等于线程池的最大大小, 则表示线程池曾经满了。

getPoolSize: 线程池的线程数量。如果线程池不销毁的话, 池里的线程不会自动销毁, 所以这个大小只增不减。

getActiveCount: 获取活动的线程数。

通过扩展线程池进行监控。通过继承线程池并重写线程池的 **beforeExecute**, **afterExecute** 和**terminated** 方法, 我们可以在任务执行前, 执行后和线程池关闭前干一些事情。如监控任务的平均执行时间, 最大执行时间和最小执行时间等。

使用线程池的风险?

虽然线程池是构建多线程应用程序的强大机制, 但使用它并不是没有风险的。用线程池构建的应用程序容易遭受任何其它多线程应用程序容易遭受的所有并发风险, 诸如同步错误和死锁, 它还容易遭受特定于线程池的少数其它风险, 诸如与池有关的死锁、资源不足和线程泄漏。

死锁

任何多线程应用程序都有死锁风险。当一组进程或线程中的每一个都在等待一个只有该组中另一个事件时, 我们就说这组进程或线程 **死锁**了。死锁的最简单情形是: 线程 A 持有对象 X 的独占锁, 而线程 B 持有对象 Y 的独占锁, 却在等待对象 X 的锁。除非有某种方法来打破对锁的等待(支持这种方法), 否则死锁的线程将永远等下去。

虽然任何多线程程序中都有死锁的风险, 但线程池却引入了另一种死锁可能, 在那种情况下, 所有池线程都在执行已阻塞的等待队列中另一任务的执行结果的任务, 但这一任务却因为没有未被占用的线程而不能运行。用来实现涉及许多交互对象的模拟, 被模拟的对象可以相互发送查询, 这些查询接下来作为排队的任务执行。查询对象又同步等待着响应时, 会发生这种情况。

资源不足

线程池的一个优点在于: 相对于其它替代调度机制(有些我们已经讨论过)而言, 它们通常执行得很好。但只有恰当地调整了线程池大小时才是这样的。线程消耗包括内存和其它系统资源在内的大量资源。除了 Thread 对象所需的内存之外, 每个线程都需要两个可能很大的执行调用堆栈。除此以外, JVM 可能会为每个 Java 线程创建一个本机线程, 这些本机线程将消耗额外的系统资源。最后, 虽然线程之间切换的调度开销很小, 但如果有很多线程, 环境切换也可能严重地影响程序的性能。

如果线程池太大, 那么被那些线程消耗的资源可能严重地影响系统性能。在线程之间进行切换将会浪费时间, 而且使用超出比您实际需要的线程可能会引起资源匮乏问题, 因为池线程正在消耗一些资源, 而这些资源可能会被其它任务更有效地利用。除了线程自身所使用的资源以外, 服务请求时所做的工作可能需要其它资源, 例如 JDBC 连接、套接字或文件。这些也都是有限资源, 有太多的并发请求也可能引起失效, 例如不能分配 JDBC 连接。

并发错误


线程池和其它排队机制依靠使用 **wait()** 和 **notify()** 方法, 这两个方法都难知, 导致线程保持空闲状态, 尽管队列中有工作要处理。使用这些方法时, 我们上面出错。而最好使用现有的、已经知道能工作的实现, 例如 **util.concurrent** 包中的类。

线程泄漏

各种类型的线程池中一个严重的风险是线程泄漏, 当从池中除去一个线程却没有返回池时, 会发生这种情况。发生线程泄漏的一种情形出现在任务结束时。如果池类没有捕捉到它们, 那么线程只会退出而线程池的大小将会越来越多, 线程池最终就为空, 而且系统将停止, 因为没有可用的线程来处理任务。



app外包公司



ui



关闭



LINCOLN
林肯MKZ
0息购车缤纷贷 金秋更享置换礼
了解更多
中型豪华轿车 ¥28.4

有些任务可能会永远等待某些资源或来自用户的输入，而这些资源又不能保证变得可用，用户可能也已经回家了，诸如此类的任务会永久停止，而这些停止的任务也会引起和线程泄漏同样的问题。如果某个线程被这样一个任务永久地消耗着，那么它实际上就被从池除去了。对于这样的任务，应该要么只给予它们自己的线程，要么只让它们等待有限的时间。

请求过载

仅仅是请求就压垮了服务器，这种情况是可能的。在这种情形下，我们可能不想将每个到来的请求都排队到我们的工作队列，因为排在队列中等待执行的任务可能会消耗太多的系统资源并引起资源缺乏。在这种情形下决定如何做取决于您自己；在某些情况下，您可以简单地抛弃请求，依靠更高级别的协议稍后重试请求，您也可以用一个指出服务器暂时很忙的响应来拒绝请求。

参考资料：

- Java 7之多线程线程池
- 聊聊并发（三）——JAVA线程池的分析和使用
- Java 理论与实践: 线程池与工作队列

以上。

顶

0

踩

0

上一篇

Android 方向传感器与磁力计和加速度传感器之间的关系


下一篇

Android Activity 启动模式详解

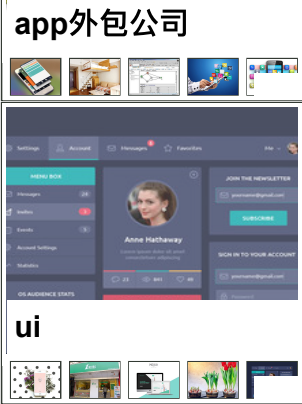
相关文章推荐

- Android中的Thread与AsyncTask的区别
 - Python全栈工程师入门指南
 - Android中的Thread与AsyncTask的区别
 - 自然语言处理在“天猫精灵”的实践应用--姜...
 - Android中的Thread与AsyncTask的区别？
 - Vue2.x基本特性解析
 - Android中的Thread与AsyncTask的区别
 - 程序员都应该掌握的Git和Github实用教程
- android Handler Thread AsyncTask httpU...
 - 深度学习项目实战-人脸检测
 - Android 不可缺少的异步（Thread、Handl...
 - Shell脚本编程
 - Android 不可缺少的异步（Thread、Handl...
 - android
 - Android
 - Android


关闭




app外包公司




ui




开发一个app多




app开发报价单



公众号开发



喷码机



app外

查看评论



林肯MKZ

0息购车缤纷贷 金秋更享置换礼

了解更多

中型豪华轿车 ¥28.4

暂无评论

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场



中型豪华轿车 ¥28.4