https://www.cnblogs.com/zhujiabin/p/5404771.html

2017/12/1

结合 并发编程实战 第8章看

绍**new Thread的弊端及Java四种线程池的使用**，对Android同样适用。本文是基础篇，后面会分享下线程池一些高级功能。

**1、new Thread的弊端**  
执行一个异步任务你还只是如下new Thread吗？

[复制代码](javascript:void(0);)

new Thread(new Runnable() {

@Override

public void run() {

// TODO Auto-generated method stub

}

}).start();

[复制代码](javascript:void(0);)

那你就out太多了，new Thread的弊端如下：

a. 每次new Thread新建对象性能差。  
b. 线程缺乏统一管理，可能无限制新建线程，相互之间竞争，及可能占用过多系统资源导致死机或oom。  
c. 缺乏更多功能，如定时执行、定期执行、线程中断。  
相比new Thread，Java提供的四种线程池的好处在于：  
a. 重用存在的线程，减少对象创建、消亡的开销，性能佳。  
b. 可有效控制最大并发线程数，提高系统资源的使用率，同时避免过多资源竞争，避免堵塞。  
c. 提供定时执行、定期执行、单线程、并发数控制等功能。

**2、Java 线程池**

Java通过Executors提供四种线程池，分别为：  
newCachedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。  
newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。  
newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。  
newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

**(1). newCachedThreadPool**创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。示例代码如下：

[复制代码](javascript:void(0);)

ExecutorService cachedThreadPool = Executors.newCachedThreadPool();

for (int i = 0; i < 10; i++) {

final int index = i;

try {

Thread.sleep(index \* 1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

cachedThreadPool.execute(new Runnable() {

@Override

public void run() {

System.out.println(index);

}

});

}

[复制代码](javascript:void(0);)

线程池为无限大，当执行第二个任务时第一个任务已经完成，会复用执行第一个任务的线程，而不用每次新建线程。

**(2). newFixedThreadPool**  
创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。示例代码如下：

[复制代码](javascript:void(0);)

ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);

for (int i = 0; i < 10; i++) {

final int index = i;

fixedThreadPool.execute(new Runnable() {

@Override

public void run() {

try {

System.out.println(index);

Thread.sleep(2000);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

});

}

[复制代码](javascript:void(0);)

因为线程池大小为3，每个任务输出index后sleep 2秒，所以每两秒打印3个数字。

定长线程池的大小最好根据系统资源进行设置。如Runtime.getRuntime().availableProcessors()。可参考[PreloadDataCache](http://www.trinea.cn/android/preloaddatacache%e6%94%af%e6%8c%81%e9%a2%84%e5%8f%96%e7%9a%84%e6%95%b0%e6%8d%ae%e7%bc%93%e5%ad%98%ef%bc%8c%e4%bd%bf%e7%94%a8%e7%ae%80%e5%8d%95%ef%bc%8c%e6%94%af%e6%8c%81%e5%a4%9a%e7%a7%8d%e7%bc%93/)。

**(3) newScheduledThreadPool**  
创建一个定长线程池，支持定时及周期性任务执行。延迟执行示例代码如下：

[复制代码](javascript:void(0);)

ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(5);

scheduledThreadPool.schedule(new Runnable() {

@Override

public void run() {

System.out.println("delay 3 seconds");

}

}, 3, TimeUnit.SECONDS);

[复制代码](javascript:void(0);)

表示延迟3秒执行。

定期执行示例代码如下：

[复制代码](javascript:void(0);)

scheduledThreadPool.scheduleAtFixedRate(new Runnable() {

@Override

public void run() {

System.out.println("delay 1 seconds, and excute every 3 seconds");

}

}, 1, 3, TimeUnit.SECONDS);

[复制代码](javascript:void(0);)

表示延迟1秒后每3秒执行一次。

ScheduledExecutorService比Timer更安全，功能更强大，后面会有一篇单独进行对比。

**(4)、newSingleThreadExecutor**  
创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。示例代码如下：

[复制代码](javascript:void(0);)

ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();

for (int i = 0; i < 10; i++) {

final int index = i;

singleThreadExecutor.execute(new Runnable() {

@Override

public void run() {

try {

System.out.println(index);

Thread.sleep(2000);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

});

}

[复制代码](javascript:void(0);)

结果依次输出，相当于顺序执行各个任务。

现行大多数GUI程序都是单线程的。Android中单线程可用于[数据库操作](http://www.trinea.cn/android/database-performance/)，文件操作，应用批量安装，应用批量删除等不适合并发但可能IO阻塞性及影响UI线程响应的操作。

**线程池的作用：**

线程池作用就是限制系统中执行线程的数量。  
     根 据系统的环境情况，可以自动或手动设置线程数量，达到运行的最佳效果；少了浪费了系统资源，多了造成系统拥挤效率不高。用线程池控制线程数量，其他线程排 队等候。一个任务执行完毕，再从队列的中取最前面的任务开始执行。若队列中没有等待进程，线程池的这一资源处于等待。当一个新任务需要运行时，如果线程池 中有等待的工作线程，就可以开始运行了；否则进入等待队列。

**为什么要用线程池:**

1.减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。

2.可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为消耗过多的内存，而把服务器累趴下(每个线程需要大约1MB内存，线程开的越多，消耗的内存也就越大，最后死机)。

Java里面线程池的顶级接口是Executor，但是严格意义上讲Executor并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是ExecutorService。

比较重要的几个类：

|  |  |
| --- | --- |
| ExecutorService | 真正的线程池接口。 |
| ScheduledExecutorService | 能和Timer/TimerTask类似，解决那些需要任务重复执行的问题。 |
| ThreadPoolExecutor | ExecutorService的默认实现。 |
| ScheduledThreadPoolExecutor | 继承ThreadPoolExecutor的ScheduledExecutorService接口实现，周期性任务调度的类实现。 |

要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，很有可能配置的线程池不是较优的，因此在Executors类里面提供了一些静态工厂，生成一些常用的线程池。

**1. newSingleThreadExecutor**

创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

2.**newFixedThreadPool**

创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

**3. newCachedThreadPool**

创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，

那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。

4.**newScheduledThreadPool**

创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

**实例**

**1：newSingleThreadExecutor·**

MyThread.java

[复制代码](javascript:void(0);)

public class MyThread extends Thread {

@Override

public void run() {

System.out.println(Thread.currentThread().getName() + "正在执行。。。");

}

}

[复制代码](javascript:void(0);)

TestSingleThreadExecutor.java

[复制代码](javascript:void(0);)

public class TestSingleThreadExecutor {

public static void main(String[] args) {

//创建一个可重用固定线程数的线程池

ExecutorService pool = Executors. newSingleThreadExecutor();

//创建实现了Runnable接口对象，Thread对象当然也实现了Runnable接口

Thread t1 = new MyThread();

Thread t2 = new MyThread();

Thread t3 = new MyThread();

Thread t4 = new MyThread();

Thread t5 = new MyThread();

//将线程放入池中进行执行

pool.execute(t1);

pool.execute(t2);

pool.execute(t3);

pool.execute(t4);

pool.execute(t5);

//关闭线程池

pool.shutdown();

}

}

[复制代码](javascript:void(0);)

**输出结果**

[复制代码](javascript:void(0);)

pool-1-thread-1正在执行。。。

pool-1-thread-1正在执行。。。

pool-1-thread-1正在执行。。。

pool-1-thread-1正在执行。。。

pool-1-thread-1正在执行。。。

[复制代码](javascript:void(0);)

**2：newFixedThreadPool**

TestFixedThreadPool.Java

[复制代码](javascript:void(0);)

publicclass TestFixedThreadPool {

publicstaticvoid main(String[] args) {

//创建一个可重用固定线程数的线程池

ExecutorService pool = Executors.newFixedThreadPool(2);

//创建实现了Runnable接口对象，Thread对象当然也实现了Runnable接口

Thread t1 = new MyThread();

Thread t2 = new MyThread();

Thread t3 = new MyThread();

Thread t4 = new MyThread();

Thread t5 = new MyThread();

//将线程放入池中进行执行

pool.execute(t1);

pool.execute(t2);

pool.execute(t3);

pool.execute(t4);

pool.execute(t5);

//关闭线程池

pool.shutdown();

}

}

[复制代码](javascript:void(0);)

**输出结果**

[复制代码](javascript:void(0);)

pool-1-thread-1正在执行。。。

pool-1-thread-2正在执行。。。

pool-1-thread-1正在执行。。。

pool-1-thread-2正在执行。。。

pool-1-thread-1正在执行。。。

[复制代码](javascript:void(0);)

3：**newCachedThreadPool**

TestCachedThreadPool.java

[复制代码](javascript:void(0);)

public class TestCachedThreadPool {

public static void main(String[] args) {

//创建一个可重用固定线程数的线程池

ExecutorService pool = Executors.newCachedThreadPool();

//创建实现了Runnable接口对象，Thread对象当然也实现了Runnable接口

Thread t1 = new MyThread();

Thread t2 = new MyThread();

Thread t3 = new MyThread();

Thread t4 = new MyThread();

Thread t5 = new MyThread();

//将线程放入池中进行执行

pool.execute(t1);

pool.execute(t2);

pool.execute(t3);

pool.execute(t4);

pool.execute(t5);

//关闭线程池

pool.shutdown();

}

}

[复制代码](javascript:void(0);)

输出结果：

[复制代码](javascript:void(0);)

pool-1-thread-2正在执行。。。

pool-1-thread-4正在执行。。。

pool-1-thread-3正在执行。。。

pool-1-thread-1正在执行。。。

pool-1-thread-5正在执行。。。

[复制代码](javascript:void(0);)

4：**newScheduledThreadPool**

TestScheduledThreadPoolExecutor.java

[复制代码](javascript:void(0);)

public class TestScheduledThreadPoolExecutor {

public static void main(String[] args) {

ScheduledThreadPoolExecutor exec = new ScheduledThreadPoolExecutor(1);

exec.scheduleAtFixedRate(new Runnable() {//每隔一段时间就触发异常

@Override

publicvoid run() {

//throw new RuntimeException();

System.out.println("================");

}

}, 1000, 5000, TimeUnit.MILLISECONDS);

exec.scheduleAtFixedRate(new Runnable() {//每隔一段时间打印系统时间，证明两者是互不影响的

@Override

publicvoid run() {

System.out.println(System.nanoTime());

}

}, 1000, 2000, TimeUnit.MILLISECONDS);

}

}

[复制代码](javascript:void(0);)

输出结果

[复制代码](javascript:void(0);)

================

8384644549516

8386643829034

8388643830710

================

8390643851383

8392643879319

8400643939383

[复制代码](javascript:void(0);)