线程池

一、线程池简介

线程池是一种多线程处理形式，处理过程中将任务添加到队列，然后在创建线程后自动启动这些任务。线程池线程都是后台线程。每个线程都使用默认的堆栈大小，以默认的优先级运行，并处于多线程单元中。如果某个线程在托管代码中空闲（如正在等待某个事件），则线程池将插入另一个辅助线程来使所有处理器保持繁忙。如果所有线程池线程都始终保持繁忙，但队列中包含挂起的工作，则线程池将在一段时间后创建另一个辅助线程但线程的数目永远不会超过最大值。超过最大值的线程可以排队，但他们要等到其他线程完成后才启动。

二、技术背景

在面向对象编程中，创建和销毁对象是很费时间的，因为创建一个对象要获取内存资源或者其它更多资源。在 Java 中更是如此，虚拟机将试图跟踪每一个对象，以便能够在对象销毁后进行垃圾回收。所以提高服务程序效率的一个手段就是尽可能减少创建和销毁对象的次数，特别是一些很耗资源的对象创建和销毁。如何利用已有对象来服务就是一个需要解决的关键问题，其实这就是一些"池化资源"技术产生的原因。比如大家所熟悉的数据库连接池正是遵循这一思想而产生的，线程池技术同样符合这一思想。

目前，一些著名的大公司都特别看好这项技术，并早已经在他们的产品中应用该技术。比如 IBM 的 WebSphere，IONA 的 Orbix 2000 在 SUN 的 Jini 中，Microsoft 的 MTS（Microsoft Transaction Server 2.0），COM+等。

三、功能

应用程序可以有多个线程，这些线程在休眠状态中需要耗费大量时间来等待事件发生。其他线程可能进入睡眠状态，并且仅定期被唤醒以轮循更改或更新状态信息，然后

再次进入休眠状态。为了简化对这些线程的管理，每个进程提供了一个线程池，一个线程池有若干个等待操作状态，当一个等待操作完成时，线程池中的辅助线程会执行回调函数。线程池中的线程由系统管理，程序员不需要费力于线程管理，可以集中精力处理应用程序任务。

四、Executors 用法

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：

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class ThreadPoolExecutorTest {
public static void main(String[] args) {
   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() {
              public void run() {
                System.out.println(index);
              }
            });
          }
        }
      }
```

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：

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class ThreadPoolExecutorTest {
```

```
public static void main(String[] args) {
    ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);
    for (int i = 0; i < 10; i++) {
     final int index = i;
     fixedThreadPool.execute(new Runnable() {
      public void run() {
       try {
        System.out.println(index);
        Thread.sleep(2000);
       } catch (InterruptedException e) {
        e.printStackTrace();
       }
      }
     });
    }
  }
}
```

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 ()

(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：

```
public class ThreadPoolExecutorTest {
public static void main(String[] args) {
   ScheduledExecutorService scheduledThreadPool =
            Executors.newScheduledThreadPool(5);
   scheduledThreadPool.schedule(new Runnable() {
    public void run() {
      System.out.println("delay 3 seconds");
    }
   }, 3, TimeUnit.SECONDS);
 }
}
```

表示延迟 3 秒执行。

定期执行示例代码如下：

```
package test;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;
public class ThreadPoolExecutorTest {
public static void main(String[] args) {
   ScheduledExecutorService scheduledThreadPool =
          Executors.newScheduledThreadPool(5);
   scheduledThreadPool.scheduleAtFixedRate(new Runnable() {
```

```
        public void run() {
          System.out.println("delay 1 seconds, and excute every 3 seconds");
        }
      }, 1, 3, TimeUnit.SECONDS);
    }
}
```

表示延迟 1 秒后每 3 秒执行一次。

(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执行。示例代码如下：

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class ThreadPoolExecutorTest {
public static void main(String[] args) {
    ExecutorService singleThreadExecutor =
            Executors.newSingleThreadExecutor();
   for (int i = 0; i < 10; i++) {
    final int index = i;
    singleThreadExecutor.execute(new Runnable() {
      public void run() {
        try {
          System.out.println(index);
```

```
            Thread.sleep(2000);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        }
      });
    }
  }
}
```

结果依次输出，相当于顺序执行各个任务。