# 第二十一章 并发

并发具有可论证的确定性，但是实际上具有不可确定性

## 21.1 并发的多面性

### 21.1.1 更快的执行

**阻塞**：程序中某个任务因为程序控制范围之外的某些条件（通常是I/O）而导致不能继续执行，那么我们就说这个任务或线程被阻塞了。

事件驱动编程

### 21.1.2 改进代码设计

并发需要付出代价，包含复杂性代价。

## 21.2 基本的线程机制

### 21.2.1 定义任务

线程可以驱动任务，要想定义任务，只要实现Runnable接口并编写run()方法，使得该任务可移植性你的命令。

*/\*\*  
 \** ***@author*** *tiantian.yuan  
 \** ***@Description*** *<p>显示火箭发射之前的倒计时</p>  
 \** ***@date*** *2017/8/31 11:35  
 \*/***public class** LiftOff **implements** Runnable{  
 **protected int countDown** = 10;  
 **private static int** *taskCount* = 0;  
 **private final int id** = *taskCount*++;  
  
 **public** LiftOff() {  
 }  
  
 **public** LiftOff(**int** countDown) {  
 **this**.**countDown** = countDown;  
 }  
  
 **public** String status(){  
 **return "#"** + **id** + **"("** + (**countDown** > 0 ? **countDown** : **"LiftOff!"**) + **")."**;  
 }  
 **public void** run() {  
 **while** (**countDown**-- > 0) {  
 System.***out***.println(status());  
 Thread.*yield*();  
 }  
 }  
}

Thread.yield()的调用是对线程调度器(Java线程机制的一部分，可以将CPU从一个线程转义给另外一个线程)的一种建议，

**package** com.study.tian.thread;  
  
*/\*\*  
 \** ***@author*** *tiantian.yuan  
 \** ***@Description*** *<p></p>  
 \** ***@date*** *2017/8/31 11:42  
 \*/***public class** MainThread {  
 **public static void** main(String[] args) {  
 LiftOff launch = **new** LiftOff();  
 launch.run();  
 }  
}

当一个类实现了Runnable接口，它必须具有run()方法。但是这个方法并无特殊之处--它不会产生任何内在的线程能力。要实现线程行为，你必须显式地将一个任务附着到线程上

### 21.2.2 Thread类

将Runnable对象转变为工作任务的传统方式是把它交给一个Thread构造器，下面的示例展示了如何使用Thread来驱动LiftOff对象

**package** com.study.tian.thread;  
  
*/\*\*  
 \** ***@author*** *tiantian.yuan  
 \** ***@Description*** *<p></p>  
 \** ***@date*** *2017/8/31 11:49  
 \*/***public class** BasicThreads {  
 **public static void** main(String[] args) {  
 Thread t = **new** Thread(**new** LiftOff());  
 Thread t2 = **new** Thread(**new** LiftOff());  
 t.start();  
 t2.start();  
 System.***out***.println(**"Watting for LiftOff"**);  
 }  
}

### 21.2.3 使用Executor

**package** com.study.tian.thread;  
  
**import** java.util.concurrent.ExecutorService;  
**import** java.util.concurrent.Executors;  
  
*/\*\*  
 \** ***@author*** *tiantian.yuan  
 \** ***@Description*** *<p></p>  
 \** ***@date*** *2017/8/31 14:23  
 \*/***public class** CacheThreadPool {  
 **public static void** main(String[] args) {  
 ExecutorService service = Executors.*newCachedThreadPool*();  
 **for** (**int** i = 0;i<5;i++) {  
 service.execute(**new** LiftOff());  
 }  
 service.shutdown();  
 }  
}

shutdown()方法的调用可以防止新人物提交给这个跟Executor当前线程(驱动main的线程)将继续运行在shutdown()被调用之前提交所有的任务。这个程序将在Executor中所有任务完成之后尽快退出

**package** com.study.tian.thread;  
  
**import** java.util.concurrent.ExecutorService;  
**import** java.util.concurrent.Executors;  
  
*/\*\*  
 \** ***@author*** *tiantian.yuan  
 \** ***@Description*** *<p></p>  
 \** ***@date*** *2017/8/31 14:41  
 \*/***public class** FixedThreadPool {  
 **public static void** main(String[] args) {  
 ExecutorService service = Executors.*newFixedThreadPool*(5);  
 **for** (**int** i = 0 ;i < 5;i++) {  
 service.execute(**new** LiftOff());  
 }  
 service.shutdown();  
 }  
}

FixedThreadPool 使用了有限的线程集来执行所提交的任务

有了FixedThreadPool，你就可以一次性预先执行代价高昂的线程分配，因而也就可以限制线程的数量。这可以节省时间，因为你不用每个任务都固定地付出创建线程的开销。

注意，在任何线程池中，现有线程在可能的情况下，都会被复用。

CachedThreadPool在程序执行过程中通常会创建与所需数量相同的线程，然后在它回收复用旧线程时才停止创建新线程，因此它是合理的Executor的首选。但这种方式会引起问题，我们就需要切换到FixedThreadPool了。

SingleThreadPool就像是线程数量为1的FixedThreadPool。这对于你希望在另一个线程中连续运行的任何事物(长期存活的任务)来说，都是很有用的，它对于在线程中运行的短任务也同样很方便

**package** com.study.tian.thread;  
  
**import** java.util.concurrent.ExecutorService;  
**import** java.util.concurrent.Executors;  
  
*/\*\*  
 \** ***@author*** *tiantian.yuan  
 \** ***@Description*** *<p></p>  
 \** ***@date*** *2017/8/31 15:50  
 \*/***public class** SingleThreadExecutor {  
 **public static void** main(String[] args) {  
 ExecutorService service = Executors.*newSingleThreadExecutor*();  
 **for** (**int** i = 0; i < 5; i++) {  
 service.execute(**new** LiftOff(100));  
 }  
 service.shutdown();  
 }  
}

如果向SingleThreadExecutor提交了多个任务，那么这些任务将排队，每个任务会在下一个任务开始之前运行结束，所有的任务使用相同的线程。所以SingleThreadExecutor会序列化所有提交给他的任务，并会维护它的悬挂任务队列。

### 21.2.4 从任务中产生返回值

Runnable是执行工作的独立任务，但是它不反回任何值。

Callable是Java 1.5 引入的一种具有类型参数的泛型，它的类型参数表示的是从方法call()的值，并且必须使用ExecutorsService的submit()方法调用它

**public class** CallableDemo {  
 **public static void** main(String[] args) {  
 ExecutorService service = Executors.*newCachedThreadPool*();  
 List<Future<String>> results = **new** ArrayList<Future<String>>();  
 **for** (**int** i = 0;i<10;i++) {  
 results.add(service.submit(**new** TaskWithResult(i)));  
 }  
 **for** (Future<String> future : results) {  
 **try** {  
 System.***out***.println(future.get());  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **catch** (ExecutionException e) {  
 e.printStackTrace();  
 }**finally** {  
 service.shutdown();  
 }  
 }  
 }  
}

submit()方法会产生Future对象，它用Callable返回结果的特定类型进行了参数化。

isDone()方法可以用来查询Furture是否已经完成。

当任务完成时，它具有一个结果，你可以调用get()方法来获取结果

你也可以不用isDone()进行检查就直接调用get()，在这种情况下，get()将会阻塞。直至结果准备完善。

你还可以试图调用get()来获取结果之前，先调用具有超时的get()。或者调用isDone()来查看任务是否完成。

### 21.2.5 休眠

影响任务行为的一种简单方法是调用sleep()，这将使任务终止执行给定时间。

对于sleep()调用可能调用InterruptedException异常，并需要在run中捕获，因为异常不能跨线程传播回main线程，所以你必须在本线程处理所有在任务内部产生的异常。

TimeUnit.***MILLISECONDS***.sleep(100);

Java 1.5 引入了TimeUnit的sleep，允许你指定延迟的时间单元

### 21.2.6 优先级

线程的优先级将该线程的重要性传递给调度器。尽管CPU处理现有线程集的顺序时不确定的，但是调度器将倾向于让优先权更高的线程先执行。

然而，这并不是意味着优先权较低的线程将得不到执行（也就是说，优先权不会导致死锁）优先级较低的线程仅仅是执行的频率较低，

在绝大多数时间里，所有的线程都应该以默认优先级运行。试图操作线程优先级通常是一种错误。

在一个任务内部，通过调用Thread.currentThread()来获取对驱动该任务的Thread对象的引用。

优先级是在run()的开头部分设定的，在构造器中设置他们不会有任何好处，因为Executor在此刻还没有开始执行任务

尽管JDK有10个优先级，但它并与多数操作系统都不能映射得很好。比如，windows有7个优先级且不是固定的，所以映射关系也是不确定的。Sun的solaris有231个优先级。唯一可移植的方法是当调整优先级的时候，只使用MAX\_PRIORITY、NORM\_PRIORITY和MIN\_PRIORITY三种级别。

### 21.2.7 让步

调用yield()方法做出按时，本任务已经做完，可以让出本线程的CPU执行权，不过这只是个按时，没有任何机制保证它会被采纳。当调用yield()方法时，你也是在建议具有相同优先级的其它线程可以运行。

但是，大体上,对于任何重要的控制或在调整应用时，都不能依赖yield()。实际上yield()经常被误用。

### 21.2.8 后台线程

所谓后台(daemon)线程,是指在程序运行的时候，在后台提供一种通用服务的线程，并且这种线程并不属于程序中不可或缺的部分。因此，当所有的非后台线程结束时，程序也就中指了，同时会杀死进程中的所有后台线程。反过来说，只要任何非后台线程还在运行，程序就不会终止！

public class SimpleDaemon implements Runnable{  
  
 public void run() {  
 try {  
 while (true) {  
 TimeUnit.MILLISECONDS.sleep(100);  
 System.out.println(Thread.currentThread() + " " + this);  
 }  
 } catch (InterruptedException e) {  
 System.out.println("sleep interrupt");  
 }  
 }  
  
 public static void main(String[] args) {  
 for (int i= 0;i<10;i++) {  
 Thread daemon = new Thread(new SimpleDaemon());  
 daemon.setDaemon(true);  
 daemon.start();  
 }  
 System.out.println("All daemon started");  
 try {  
 TimeUnit.MILLISECONDS.sleep(175100);  
 } catch (InterruptedException e) {  
 System.out.println("sleep error");  
 }  
 }  
}

1. 必须在线程启动之前调用setDaemon()方法，才能把它设置为后台线程
2. 一旦main()完成其工作，就没什么能阻止程序终止了

通过编写定制的ThreadFactory可以定制由Executor的线程和属性

public class DaemonThreadFactory implements ThreadFactory{

@Override

public Thread newThread(Runnable r) {

Thread thread = new Thread(r);

thread.setDaemon(true);

return thread;

}

}

这与普通的ThreadFactory的唯一差异就是它将后台状态全部设置为true.你可以用新的DaemonThreadFactory作为参数传递给Executors.newCachedThreadPool()

public class DaemonFromFactory implements Runnable {

public void run() {

try {

while (true) {

TimeUnit.MILLISECONDS.sleep(100);

System.out.println(Thread.currentThread() + " " + this);

}

} catch (InterruptedException e) {

System.out.println("Interrupt");

}

}

public static void main(String[] args) throws InterruptedException {

ExecutorService service = Executors.newCachedThreadPool(new DaemonThreadFactory());

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

service.execute(new DaemonFromFactory());

}

System.out.println("All daemons started");

TimeUnit.MILLISECONDS.sleep(500);

}

}

Daemon线程被设置成了后台线程,然后派生了许多子线程,这些线程并没有被设置成后台线程,不过他们的确是后台线程.

### 21.2.9 编码的变体

在构造器中启动线程会很有问题，因为另一个任务很可能在构造器结束之前开始执行，这意味着该任务能够访问处于不稳定状态的对象。

### 21.2.11 术语

A线程在B线程之上调用join()方法，其效果是B等待一段时间直到A线程结束之后才继续执行。

如果B线程上，A线程调用join()，B线程将会被挂起，直到目标线程A结束才恢复（即B.isAlive()返回为假）。

join()也可以带上一个超时参数（单位可以是毫秒，或者毫秒和纳秒），这样如果目标线程在这段时间到期时还没有结束的话，join()方法总能返回。

对join()方法的调用可以被中断，做法是在调用interrupt()方法，这时需要用到try-catch子句.

注意，Java SE5 的java.util.concurrent类库包含诸如CyclicBarrier这样的工具，它们可能比最初的线程类库中的join()更加合适。

## 21.3 共享受限资源

### 21.3.1不正确的访问资源

### 21.3.2 解决共享资源竞争

对于并发工作，需要防止两个任务访问相同的资源，至少不能在关键阶段出现这种情况。

防止这种冲突的方法就是当资源被一个任务使用时，在其上加锁。当某一资源的任务锁定这项资源，使其它任务在被其解锁之前，就无法访问它，在其被解锁之时，另一个任务就可以锁定并使用它，依次类推。

序列化访问共享资源，意味着在一个时间点上只允许一个任务访问共享资源。因为锁语句产生了相排斥的效果，所以这种机制常常称为互斥量(mutex)。

Java以提供关键字synchronized的形式，为防止资源冲突提供了内置支持。当任务要执行被synchronized关键字保护的代码片段时，它将检查锁是否可用，然后获取锁，执行代码，释放锁。

要控制对共享资源的访问，得先把它包装进一个对象。然后把所有要访问这个资源的方法标记为synchronized。如果某个任务对于一个对标记为synchronized的方法调用中，那么在这个线程从该方法返回之前，其它所有要调用类中任何标记为synchronized方法的线程都会被阻塞。

注意,在使用并发时,将共享的属性设置为private是非常重要的,否则,synchronized关键字就不能防止其它任务访问此属性,这样就会产生冲突.