### 1、为什么使用线程池

为每个请求对应一个线程方法的不足是：为每个请求创建一个新线程的开销很大；为每个请求创建新线程的服务器在创建和销毁线程上花费的时间和消耗的系统资源要比花在处理实际的用户请求的时间和资源更多。容易引起资源不足，造成浪费。为解决单个任务处理时间很短而请求的数目巨大的问题，引出线程池：

通过对多个任务重用线程，线程创建的开销被分摊到了多个任务上。其好处

是，因为在请求到达时线程已经存在，所以无意中也消除了线程创建所带来的延迟，使应用程序响应更快；通过适当地调整线程池中的线程数目，也就是当请求的数目超过某个阈值时，就强制其它任何新到的请求一直等待，直到获得一个线程来处理为止，从而可以防止资源不足。

### 2、使用线程池的风险

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。

#### 2.1 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进程才能引起的事件时，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，并且在等待对象Y的锁，

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（Java 锁定不支持这种方法），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。当线程池被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。

#### 2.2 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。

#### 2.3 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以执行一项任务，而在任务完成后该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抛出一个 RuntimeException 或一个 Error 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永久减少一个。当这种情况发生的次数足够多时，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。

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。

### 3、有效使用线程池的准则

不要对那些同步等待其它任务结果的任务排队。这可能会导致上面所描述的那种形式的死锁，在那种死锁中，所有线程都被一些任务所占用，这些任务依次等待排队任务的结果，而这些任务又无法执行，因为所有的线程都很忙。

在为时间可能很长的操作使用合用的线程时要小心。如果程序必须等待诸如 I/O 完成这样的某个资源，那么请指定最长的等待时间，以及随后是失效还是将任务重新排队以便稍后执行。这样做保证了：通过将某个线程释放给某个可能成功完成的任务，从而将最终取得某些进展。

理解任务。要有效地调整线程池大小，您需要理解正在排队的任务以及它们正在做什么。它们是 CPU 限制的（CPU-bound）吗？它们是I/O 限制的（I/O-bound）吗？您的答案将影响您如何调整应用程序。如果您有不同的任务类，这些类有着截然不同的特征，那么为不同任务类设置多个工作队列可能会有意义，这样可以相应地调整每个池。

### 4. 线程池的大小设置

调整线程池的大小基本上就是避免两类错误：线程太少或线程太多。

虽然线程池大小的设置受到很多因素影响，但是这里给出一个参考公式：

最佳线程数目=（（线程等待时间+线程CPU时间）/线程CPU时间）\* CPU数目

比如平均每个线程CPU运行时间为0.5s，而线程等待时间（非CPU运行时间，比如IO）为1.5s，CPU核心数为8，那么根据上面这个公式估算得到：((0.5+1.5)/0.5)\*8=32。这个公式进一步转化为：

最佳线程数目 = （线程等待时间与线程CPU时间之比 + 1）\* CPU数目

线程等待时间所占比例越高，需要越多线程。线程CPU时间所占比例越高，需要越少线程。

### 5. 常用的几种线程池

#### 5.1 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。

这种类型的线程池特点是：

* 工作线程的创建数量几乎没有限制(其实也有限制的,数目为Interger. MAX\_VALUE), 这样可灵活的往线程池中添加线程。
* 如果长时间没有往线程池中提交任务，即如果工作线程空闲了指定的时间(默认为1分钟)，则该工作线程将自动终止。终止后，如果你又提交了新的任务，则线程池重新创建一个工作线程。
* 在使用CachedThreadPool时，一定要注意控制任务的数量，否则，由于大量线程同时运行，很有会造成系统瘫痪。

示例代码如下：

public class NewCacheThreadPoolTest {

public static void main(String[] args) {  
 // 创建一个可缓存线程池  
 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();  
 }  
 // 使用线程池执行任务  
 cachedThreadPool.execute(  
 () -> System.*out*.println(index)  
 );  
  
 }  
 }  
}

#### 5.2 newFixedThreadPool

创建一个指定工作线程数量的线程池。每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池初始的最大数，则将提交的任务存入到池队列中。

FixedThreadPool是一个典型且优秀的线程池，它具有线程池提高程序效率和节省创建线程时所耗的开销的优点。但是，在线程池空闲时，即线程池中没有可运行任务时，它不会释放工作线程，还会占用一定的系统资源。

示例代码如下：

public class NewFixedThreadPoolTest {

public static void main(String[] args) {  
 // 创建一个指定线程数量的线程池  
 ExecutorService fixedThreadPool = Executors.*newFixedThreadPool*(3);  
 for (int i = 0; i < 10; i++) {  
 final int index = i;  
 // 使用线程池执行任务  
 fixedThreadPool.execute(  
 () -> {  
 try {  
 System.*out*.println(index);  
 Thread.*sleep*(2000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 );  
 }  
 }  
}

定长线程池的大小最好根据系统资源进行设置如Runtime.getRuntime().availableProcessors()。

#### 5.3 newSingleThreadExecutor

创建一个单线程化的Executor，即只创建唯一的工作者线程来执行任务，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。如果这个线程异常结束，会有另一个取代它，保证顺序执行。单工作线程最大的特点是可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的。

示例代码如下：

public class NewSingleThreadPoolTest {

public static void main(String[] args) {  
 ExecutorService singleThreadExecutor = Executors.*newSingleThreadExecutor*();  
 for (int i = 0; i < 10; i++) {  
 final int index = i;  
 singleThreadExecutor.execute(  
 () -> {  
 try {  
 System.*out*.println(index);  
 Thread.*sleep*(2000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 );  
 }  
 }  
}

#### 5.4 newScheduleThreadPool

创建一个定长的线程池，而且支持定时的以及周期性的任务执行。

延迟3秒执行，延迟执行示例代码如下：

public class NewScheduleThreadPoolTest {

public static void main(String[] args) {  
 ScheduledExecutorService scheduledThreadPool = Executors.*newScheduledThreadPool*(5);  
 scheduledThreadPool.schedule(  
 () -> System.*out*.println("delay 3 seconds"), 3, TimeUnit.*SECONDS* );  
 }  
}

表示延迟1秒后每3秒执行一次，定期执行示例代码如下：

public class NewScheduleThreadPoolTest2 {

public static void main(String[] args) {  
 ScheduledExecutorService scheduledThreadPool = Executors.*newScheduledThreadPool*(5);  
 scheduledThreadPool.scheduleAtFixedRate(  
 () ->  
 System.*out*.println("delay 1 seconds, and execute every 3 seconds")  
 , 1, 3, TimeUnit.*SECONDS*);  
 }  
}