

< 2017年3月 >						
日	一	二	三	四	五	六
26	27	28	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	1
2	3	4	5	6	7	8

java常用的几种线程池比较

1. 为什么使用线程池

诸如 Web 服务器、数据库服务器、文件服务器或邮件服务器之类的许多服务器应用程序都面向处理来自某些远程来源的大量短小的任务。请求以某种方式到达服务器，这种方式可能是通过网络协议（例如 HTTP、FTP 或 POP）、通过 JMS 队列或者可能通过轮询数据库。不管请求如何到达，服务器应用程序中经常出现的情况是：单个任务处理的时间很短而请求的数目却是巨大的。

构建服务器应用程序的一个简单模型是：每当一个请求到达就创建一个新线程，然后在新线程中为请求服务。实际上对于原型开发这种方法工作得很好，但如果试图部署以这种方式运行的服务器应用程序，那么这种方法的严重不足就很明显。每个请求对应一个线程（**thread-per-request**）方法的不足之一是：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。

除了创建和销毁线程的开销之外，活动的线程也消耗系统资源。在一个 JVM 里创建太多的线程可能会导致系统由于过度消耗内存而用完内存或“切换过度”。为了防止资源不足，服务器应用程序需要一些办法来限制任何给定时刻处理的请求数目。

线程池为线程生命周期开销问题和资源不足问题提供了解决方案。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，所以无意中消除了线程创建所带来的延迟。这样，就可以立即为请求服务，使应用程序响应更快。而且，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

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。

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（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。

2.3 并发错误

线程池和其它排队机制依靠使用 wait() 和 notify() 方法，这两个方法都难于使用。如果编码不正确，那么可能丢失通知，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，必须格外小心。而最好使用现有的、已经知道能工作的实现，例如 util.concurrent 包。

2.4 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。

2.5 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。

3. 有效使用线程池的准则

只要您遵循几条简单的准则，线程池可以成为构建服务器应用程序的极其有效的方法：

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。幸运的是，对于大多数应用程序来说，太多和太少之间的余地相当宽。

请回忆：在应用程序中使用线程有两个主要优点，尽管在等待诸如 I/O 的慢操作，但允许继续进行处理，并且可以利用多处理器。在运行于具有 N 个处理器机器上的计算限制的应用程序中，在线程数目接近 N 时添加额外的线程可能会改善总处理能力，而在线程数目超过 N 时添加额外的线程将不起作用。事实上，太多的线程甚至会降低性能，因为它会导致额外的环境切换开销。

线程池的最佳大小取决于可用处理器的数目以及工作队列中的任务的性质。若在一个具有 N 个处理器的系统上只有一个工作队列，其中全部是计算性质的任务，在线程池具有 N 或 N+1 个线程时一般会获得最大的 CPU 利用率。

对于那些可能需要等待 I/O 完成的任务（例如，从套接字读取 HTTP 请求的任务），需要让池的大小超过可用处理器的数目，因为并不是所有线程都一直在工作。通过使用概要分析，您可以估计某个典型请求的等待时间（WT）与服务时间（ST）之间的比例。如果我们将这一比例称之为 WT/ST，那么对于一个具有 N 个处理器的系统，需要设置大约 $N * (1 + WT/ST)$ 个线程来保持处理器得到充分利用。

处理器利用率不是调整线程池大小过程中的唯一考虑事项。随着线程池的增长，您可能会碰到调度程序、可用内存方面的限制，或者其它系统资源方面的限制，例如套接字、打开的文件句柄或数据库连接等的数目。

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Integer. 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：



```
1 package test;
2 import java.util.concurrent.ExecutorService;
3 import java.util.concurrent.Executors;
4 public class ThreadPoolExecutorTest {
5     public static void main(String[] args) {
6         ExecutorService cachedThreadPool = Executors.newCachedThreadPool();
7         for (int i = 0; i < 10; i++) {
8             final int index = i;
9             try {
10                 Thread.sleep(index * 1000);
11             } catch (InterruptedException e) {
12                 e.printStackTrace();
13             }
14             cachedThreadPool.execute(new Runnable() {
15                 public void run() {
16                     System.out.println(index);
17                 }
18             });
19         }
20     }
21 }
```



5.1 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，即线程池中无可运行任务时，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：



```
package test;
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()。

5.1 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线程是活动的。

示例代码如下：



```
package test;
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();
                    }
                }
            });
        }
    }
}
```

```
    }  
    }  
    });  
    }  
    }  
}
```




5.1 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，支持定时及周期性任务执行。

延迟3秒执行，延迟执行示例代码如下：




```
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.schedule(new Runnable() {  
            public void run() {  
                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：



```
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);  
    }  
}
```



分类: [java](#)

好文要顶

关注我

收藏该文



Mr.Aaron

关注 - 0

粉丝 - 0

+加关注

0

0

« 上一篇: SSH三大框架简介
» 下一篇: \$(document).ready(function(){})和\$(window).load(function(){})的区别

posted @ 2016-12-23 10:18 Mr.Aaron 阅读(884) 评论(0) 编辑 收藏

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。

- 【推荐】50万行VC++源码：大型组态工控、电力仿真CAD与GIS源码库
- 【免费】自开发零实施的H3 BPM免费下载
- 【推荐】Google+GitHub联手打造前端工程师课程
- 【云上】在金山大米云，让云计算更简单
- 【推荐】阿里云香港云服务器65折，免备案



最新IT新闻：

- 锤子线下专卖店真要开了，但罗永浩在担心什么？
 - 她月薪5万真的是因为选择了最艰难的路？
 - 苹果新总部依旧未完工 4月之前还有很多要做
 - 高通：别叫它处理器，请叫“骁龙平台”
 - 耐克回应“气垫门”：用错宣传材料，退款并赔偿4500元
- » 更多新闻...



最新知识库文章：

- 为什么我要写自己的框架？
 - 垃圾回收原来是这么回事
 - 「代码家」的学习过程和学习经验分享
 - 写给未来的程序媛
 - 高质量的工程代码为什么难写
- » 更多知识库文章...