Java 中的线程池及实现

Threadpool and it's Implementation in Java

封 玮 周世平 （烟台大学计算机学院 264005）

摘要：本文探讨了应用线程池技术对 Java 多线程程序性能的优化及有效使用线程池技术的准则，研究了线程池技术的特点、工作原理、实现方法，并给出了线程池的 Java 实现实例。

关键词：多线程 线程池 Java

1 引言

Java 语言是一种面向对象的程序设计语言。用 Java 语言模拟客观世界能直接反映客观对象的行为和它们之间的关系，体现客观对象的自治特性。而 Java 语言对多线程技术的支持，又使其能够方便地描述并发操作，体现客观对象的并发特性。Java 语言没有把线程化看作是底层操作系统的工具，而是通过内置语言级的多线程控制，保证了多个并发例程可以通过并发线程编程实现，从程序角度提供了各个线程间协调工作的调度能力，从而降低了并发程序设计的难度。语言级多线程技术的支持，使程序展现出优异的性能，但也面临了一些新的问题，如：对多个线程进行调度、管理的复杂性以及线程的切换开销带来的多线程程序的低效性等等。本文探讨了应用线程池技术对 Java 多线程程序性能的优化及有效使用线程池技术的准则，研究了线程池技术的特点、工作原理、实现方法，并给出了线程池的 Java 实现实例。

2 线程池的工作原理

从本质上讲，线程池是一种对线程的管理策略。它预先创建一定数量的线程，让它们进入阻塞状态等待。需要时，只需简单地唤醒这些已存在的线程中的一个，而不是从头创建它。这样，可以对多个任务重用线程。

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，所以消除了线程创建所带来的延迟，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，就强制其他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。

线程池的实现，应能够：

* 提供工作线程池中任务的调度；
* 可灵活地调整池的大小；
* 对线程生命周期进行管理；
* 可以限制工作队列中任务的数目，以防止队列中的任务耗尽所有可用内存；
* 提供多种可用的关闭和饱和度策略（阻塞、废弃、抛出、废弃最老的、在调用者中运行等）。

3 线程池的 Java 实现

3.1 阻塞队列

我们的线程池将在阻塞队列的基础上创建。所以，应首先建立实现线程池所需要的阻塞队列。本质上，我们的阻塞队列是一个提供了出/入队、判断队列是否为空和关闭队列等操作并考虑到安全性因素的链式线程队列。

线程出队操作，考虑了队列为空的情况。队列为空时，自动阻塞，直到有对象入队。

线程入队操作，考虑了队列关闭的情况。队列关闭时，抛出例外。

程序清单 1：Blocking_queue.java

一个基于链表结构的安全的线程队列（如果试图对空队列进行出队操作将自动阻塞）。

```
import java.util.*;
public class Blocking_queue{? //定义阻塞队列类
  private LinkedList elements = new LinkedList();
? //创建基于链表结构的安全的线程队列
  private boolean closed = false; //设置队列关闭标志为 false
//如果使用一个关闭的队列，则抛出 Closed exception。
public class Closed extends RuntimeException{ //定
```

```
义"关闭队列例外"类
    private Closed(){ //定义"关闭队列"例外类的构造函数
        super( "Tried to access closed Blocking_queue");? //进行自定义的"关闭队列例外"处理
    }
  }
//将一个对象入队
  public synchronized final void enqueue( Object new_element )? throws Blocking_queue.Closed{
    if( closed )? //判断队列是否关闭
      throw new Closed();? //是,则进行自定义的"关闭队列例外"处理
      elements.addLast( new_element );? //否,在队尾加入新元素
      notify();? //唤醒等待线程
  }
//将一个元素出队,如果队列为空则阻塞(直到有对象入队)。
  public synchronized final Object dequeue() throws InterruptedException,Blocking_queue.Closed{
    try{?
      while( elements.size() <= 0 ){ //队列为空,则阻塞
        wait();? //等待,直到有新元素
        if( closed )? //判断队列是否关闭
          throw new Closed();? //是,则进行自定义的"关闭队列例外"处理
      }
      return elements.removeFirst(); //头元素出队
    }catch( NoSuchElementException e ) // 不应发生
    {
      throw new Error("Internal error (com.holub.asynch.Blocking_queue)"); //抛出错误
    }
  }
public synchronized final boolean is_empty()? { //定义同步方法判断队列是否为空
    return elements.size() > 0;? //返回布尔值,空为false,非空为 true
  }
/*
    释放一个阻塞队列会使所有被阻塞的线程(由出队方法排好等待出队的那些线程)得以释放。此时,会抛出 Blocking_queue.Closed 例外。一旦队列关闭,任何试图入队的操作也将引发 Blocking_queue.Closed 例外。
* /
  public synchronized void close()? { //定义同步的关闭方法
    closed = true;? //设置队列关闭标志为 true
    notifyAll(); //唤醒所有等待线程
  }
}
```

3.2 线程池的实现

借助于前面建立好的阻塞队列,我们可以方便地实现一个线程池类。线程池中,是一组可用以执行任意操作的线程,你可以随时要求池中的一个线程为你执行某些操作。你的请求会立刻得到响应,而操作将以后台作业形式执行。

下面,线程池类的实现。我们看到:线程池构造函数中,指定了两个参数 initial_thread_count 和 maximum_thread_count。前者,在创建线程池时,规定池中线程的个数。后者,当要求线程池为你执行某操作时,如果池中的所有线程都在忙着,你可以额外创建一些线程,但不能超出 maximum_thread_count 规定的数目。

需由线程池完成的操作,通过调用 execute()方法来告诉线程池。语句如下:

```
pool.execute( new Runnable() {public void run() {……}} );
```

线程池构造函数创建了一组池中的线程对象,并启动池中的线程。此时,这些线程的 run()方法就进入了自己的主执行循环,然后在阻塞队列 Blocking_queue(也称"工作队列")上阻塞。而 execute()方法实现的功能则是通过对阻塞队列 Blocking_queue 执行 Runnable 对象的入队操作,将这个 Runnable 对象传递给池中的线程。Runnable 对象的入队将导致一个等待的池线程对象"醒来",将这个 Runnable 对象出队,执行它的 run()方法,执行后该线程再返回阻塞队列。

程序清单 2:Thread_pool.java

线程池的一般实现。

```
/ *
```

线程池的大小可根据执行请求自动扩展。即,如果池中有可用的线程,它将被用于执行 Runnable 对象,否则,创建一个新线程(并加入到池中)执行该请求。可以规定最大线程数来限制池中线程的数量。每个线程池可组成一个线程

组(池中所有线程均在此组)。实践中,这意味着安全性管理控制——该池中的一个线程是否能够访问其他组中的线程,同时,也向你提供了一个使整个组成为守护线程的简易机制。

```
* /
import Blocking_queue; //引入阻塞队列
public class Thread_pool extends ThreadGroup{? //定义线程池类
  private final Blocking_queue pool = new Blocking_queue(); //创建阻塞对象
  private final int maximum_size; //线程池的最大线程数
  private int pool_size;? //线程池中线程个数
  private boolean has_closed = false; //设置关闭标志为 false
  private static int group_number = 0; //组号
  private static int thread_id = 0; //线程标志
/*
  池线程类。这是些待被激活的对象,它们在一个空队列上被阻塞着,你向这个队列传递一个 Runnable 对象释放一个线程,这个 Runnable 对象的 run()方法得以执行。所有的池线程对象都将是组成这个线程池的线程组的成员。
* /
  private class Pooled_thread extends Thread{ //定义池线程类
    public Pooled_thread(){ //池线程构造函数
      super( Thread_pool.this, "T" + thread_id ); //调用 Thread 类构造函数
    }
    public void run()? { //定义池线程体
      try{
        while( ! has_closed ){ //当池线程队列未关闭
          ((Runnable)( pool.dequeue() )).run(); //释放一个线程,该对象的 run()方法得以执行
        }
      }
      catch(InterruptedException e){/* 终止线程 * /}
      catch(Blocking_queue.Closed e){/* 终止线程 * /}
    }
  }
/*
```

4 结语

线程池技术是组织网络应用程序的有用工具,特别适用于网络应用中服务器接受到大量的处理任务生存期很短的请求的情况,它可以大大减少线程的创建和销毁次数,提高服务器的工作效率。但如果线程要求的运行资源比创建所花费的资源多得多,单靠减少创建所花费的资源对系统效率、性能提高不明显时,就不适合应用线程池技术,而需要借助其他的技术来改善服务器的服务性能。

人们在 Java 多线程应用程序的实现方面做了大量有益的探索和实践。值得一提的是 Doug Lea 编写的一个优秀的并发实用程序开放源码库 util.concurrent 包。它包括了线程池的高性能实现及互斥、信号量、诸如在并发访问下执行得很好的队列和散列表等集合类以及几个工作队列实现。util.concurrent 是 JSR 166 的切入点,它带来一组并发性的实用程序,这些实用程序将应用于 JDK 1.5 发行版,成为 java.util.concurrent 包基础。

参考资料

1 机械工业出版社,《Java2 核心技术 卷Ⅱ:高级特性》,2000 年 11 月。
2 机械工业出版社,《Java 程序设计教程 高级篇》,2002 年 1 月。
3 Sun Microsystems, Inc. JAVA 性能技巧和防火墙隧道技术,2002 年。
4 Sun Microsystems, Concurrent collections classes.developerWorks,2003 年 7 月。
5 Brian Goet.线程池与工作队列,developerWorks,2002 年 10 月。
6 Doug Lea. Concurrent Programming in Java. Prentice Hall,2000 年 11 月。
7 http://Java. sun. com/j2se/1. 4. 2/docs/api/, 2003 年。
8 Allen Holub,关于解决 Java 编程语言线程问题的建议,developerWorks,2000 年 10 月。
9 Alex Roetter,编写线程安全的类的准则.developerWorks,2001 年 。
10 http://www.jcp.org/en/jsr/,JSR 166 Concurrency Utilities.2003 年。
11 Allen Holub. Programming Java threads in the real world. JavaWorld,1999 年 6 月。