http://blog.csdn.net/hsuxu/article/details/8985931

更好理解的一篇博客：http://www.jianshu.com/p/87bff5cc8d8c

**1、线程池简介：**  
    多线程技术主要解决处理器单元内多个线程执行的问题，它可以显著减少处理器单元的闲置时间，增加处理器单元的吞吐能力。      
    假设一个服务器完成一项任务所需时间为：T1 创建线程时间，T2 在线程中执行任务的时间，T3 销毁线程时间。  
  
如果：T1 + T3 远大于 T2，则可以采用线程池，以提高服务器性能。  
 一个线程池包括以下四个基本组成部分：  
                1、线程池管理器（ThreadPool）：用于创建并管理线程池，包括 创建线程池，销毁线程池，添加新任务；  
                2、工作线程（PoolWorker）：线程池中线程，在没有任务时处于等待状态，可以循环的执行任务；  
                3、任务接口（Task）：每个任务必须实现的接口，以供工作线程调度任务的执行，它主要规定了任务的入口，任务执行完后的收尾工作，任务的执行状态等；  
                4、任务队列（taskQueue）：用于存放没有处理的任务。提供一种缓冲机制。  
                  
  线程池技术正是关注如何缩短或调整T1,T3时间的技术，从而提高服务器程序性能的。它把T1，T3分别安排在服务器程序的启动和结束的时间段或者一些空闲的时间段，这样在服务器程序处理客户请求时，不会有T1，T3的开销了。  
    线程池不仅调整T1,T3产生的时间段，而且它还显著减少了创建线程的数目，看一个例子：  
    假设一个服务器一天要处理50000个请求，并且每个请求需要一个单独的线程完成。在线程池中，线程数一般是固定的，所以产生线程总数不会超过线程池中线程的数目，而如果服务器不利用线程池来处理这些请求则线程总数为50000。一般线程池大小是远小于50000。所以利用线程池的服务器程序不会为了创建50000而在处理请求时浪费时间，从而提高效率。

    代码实现中并没有实现任务接口，而是把Runnable对象加入到线程池管理器（ThreadPool），然后剩下的事情就由线程池管理器（ThreadPool）来完成了

**[java]** [view plaincopy](http://blog.csdn.net/touch_2011/article/details/6914468)

1. **package** mine.util.thread;
3. **import** java.util.LinkedList;
4. **import** java.util.List;
6. /\*\*
7. \* 线程池类，线程管理器：创建线程，执行任务，销毁线程，获取线程基本信息
8. \*/
9. **public** **final** **class** ThreadPool {
10. // 线程池中默认线程的个数为5
11. **private** **static** **int** worker\_num = 5;
12. // 工作线程
13. **private** WorkThread[] workThrads;
14. // 未处理的任务
15. **private** **static** **volatile** **int** finished\_task = 0;
16. // 任务队列，作为一个缓冲,List线程不安全
17. **private** List<Runnable> taskQueue = **new** LinkedList<Runnable>();
18. **private** **static** ThreadPool threadPool;
20. // 创建具有默认线程个数的线程池
21. **private** ThreadPool() {
22. **this**(5);
23. }
25. // 创建线程池,worker\_num为线程池中工作线程的个数
26. **private** ThreadPool(**int** worker\_num) {
27. ThreadPool.worker\_num = worker\_num;
28. workThrads = **new** WorkThread[worker\_num];
29. **for** (**int** i = 0; i < worker\_num; i++) {
30. workThrads[i] = **new** WorkThread();
31. workThrads[i].start();// 开启线程池中的线程
32. }
33. }
35. // 单态模式，获得一个默认线程个数的线程池
36. **public** **static** ThreadPool getThreadPool() {
37. **return** getThreadPool(ThreadPool.worker\_num);
38. }
40. // 单态模式，获得一个指定线程个数的线程池,worker\_num(>0)为线程池中工作线程的个数
41. // worker\_num<=0创建默认的工作线程个数
42. **public** **static** ThreadPool getThreadPool(**int** worker\_num1) {
43. **if** (worker\_num1 <= 0)
44. worker\_num1 = ThreadPool.worker\_num;
45. **if** (threadPool == **null**)
46. threadPool = **new** ThreadPool(worker\_num1);
47. **return** threadPool;
48. }
50. // 执行任务,其实只是把任务加入任务队列，什么时候执行有线程池管理器觉定
51. **public** **void** execute(Runnable task) {
52. **synchronized** (taskQueue) {
53. taskQueue.add(task);
54. taskQueue.notify();
55. }
56. }
58. // 批量执行任务,其实只是把任务加入任务队列，什么时候执行有线程池管理器觉定
59. **public** **void** execute(Runnable[] task) {
60. **synchronized** (taskQueue) {
61. **for** (Runnable t : task)
62. taskQueue.add(t);
63. taskQueue.notify();
64. }
65. }
67. // 批量执行任务,其实只是把任务加入任务队列，什么时候执行有线程池管理器觉定
68. **public** **void** execute(List<Runnable> task) {
69. **synchronized** (taskQueue) {
70. **for** (Runnable t : task)
71. taskQueue.add(t);
72. taskQueue.notify();
73. }
74. }
76. // 销毁线程池,该方法保证在所有任务都完成的情况下才销毁所有线程，否则等待任务完成才销毁
77. **public** **void** destroy() {
78. **while** (!taskQueue.isEmpty()) {// 如果还有任务没执行完成，就先睡会吧
79. **try** {
80. Thread.sleep(10);
81. } **catch** (InterruptedException e) {
82. e.printStackTrace();
83. }
84. }
85. // 工作线程停止工作，且置为null
86. **for** (**int** i = 0; i < worker\_num; i++) {
87. workThrads[i].stopWorker();
88. workThrads[i] = **null**;
89. }
90. threadPool=**null**;
91. taskQueue.clear();// 清空任务队列
92. }
94. // 返回工作线程的个数
95. **public** **int** getWorkThreadNumber() {
96. **return** worker\_num;
97. }
99. // 返回已完成任务的个数,这里的已完成是只出了任务队列的任务个数，可能该任务并没有实际执行完成
100. **public** **int** getFinishedTasknumber() {
101. **return** finished\_task;
102. }
104. // 返回任务队列的长度，即还没处理的任务个数
105. **public** **int** getWaitTasknumber() {
106. **return** taskQueue.size();
107. }
109. // 覆盖toString方法，返回线程池信息：工作线程个数和已完成任务个数
110. @Override
111. **public** String toString() {
112. **return** "WorkThread number:" + worker\_num + "  finished task number:"
113. + finished\_task + "  wait task number:" + getWaitTasknumber();
114. }
116. /\*\*
117. \* 内部类，工作线程
118. \*/
119. **private** **class** WorkThread **extends** Thread {
120. // 该工作线程是否有效，用于结束该工作线程
121. **private** **boolean** isRunning = **true**;
123. /\*
124. \* 关键所在啊，如果任务队列不空，则取出任务执行，若任务队列空，则等待
125. \*/
126. @Override
127. **public** **void** run() {
128. Runnable r = **null**;
129. **while** (isRunning) {// 注意，若线程无效则自然结束run方法，该线程就没用了
130. **synchronized** (taskQueue) {
131. **while** (isRunning && taskQueue.isEmpty()) {// 队列为空
132. **try** {
133. taskQueue.wait(20);
134. } **catch** (InterruptedException e) {
135. e.printStackTrace();
136. }
137. }
138. **if** (!taskQueue.isEmpty())
139. r = taskQueue.remove(0);// 取出任务
140. }
141. **if** (r != **null**) {
142. r.run();// 执行任务
143. }
144. finished\_task++;
145. r = **null**;
146. }
147. }
149. // 停止工作，让该线程自然执行完run方法，自然结束
150. **public** **void** stopWorker() {
151. isRunning = **false**;
152. }
153. }
154. }

测试代码：

**[java]** [view plaincopy](http://blog.csdn.net/touch_2011/article/details/6914468)

1. **package** mine.util.thread;
3. //测试线程池
4. **public** **class** TestThreadPool {
5. **public** **static** **void** main(String[] args) {
6. // 创建3个线程的线程池
7. ThreadPool t = ThreadPool.getThreadPool(3);
8. t.execute(**new** Runnable[] { **new** Task(), **new** Task(), **new** Task() });
9. t.execute(**new** Runnable[] { **new** Task(), **new** Task(), **new** Task() });
10. System.out.println(t);
11. t.destroy();// 所有线程都执行完成才destory
12. System.out.println(t);
13. }
15. // 任务类
16. **static** **class** Task **implements** Runnable {
17. **private** **static** **volatile** **int** i = 1;
19. @Override
20. **public** **void** run() {// 执行任务
21. System.out.println("任务 " + (i++) + " 完成");
22. }
23. }
24. }

运行结果：

WorkThread number:3  finished task number:0  wait task number:6  
任务 1 完成  
任务 2 完成  
任务 3 完成  
任务 4 完成  
任务 5 完成  
任务 6 完成  
WorkThread number:3  finished task number:6  wait task number:0

分析：由于并没有任务接口，传入的可以是自定义的任何任务，所以线程池并不能准确的判断该任务是否真正的已经完成（真正完成该任务是这个任务的run方法执行完毕），只能知道该任务已经出了任务队列，正在执行或者已经完成。

**2、java类库中提供的线程池简介：**

**java提供的线程池更加强大，相信理解线程池的工作原理，看类库中的线程池就不会感到陌生了。**



