1.线程池基本原理

在传统服务器结构中, 常是 有一个总的 监听线程监听有没有新的用户连接服务器, 每当有一个新的 用户进入, 服务器就开启一个新的线程用户处理这 个用户的数据包。这个线程只服务于这个用户 , 当 用户与服务器端关闭连接以后, 服务器端销毁这个线程。然而频繁地开辟与销毁线程极大地占用了系统的资源。而且在大量用户的情况下, 系统为了开辟和销毁线程将浪费大量的时间和资源。线程池提供了一个解决外部大量用户与服务器有限资源的矛盾, 线程池和传统的一个用户对应一 个线程的处理方法不同, 它的基本思想就是在程序 开始时就在内存中开辟一些线程, 线程的数目是 固定的,他们独自形成一个类, 屏蔽了对外的操作, 而服务器只需要将数据包交给线程池就可以了。当有新的客户请求到达时 , 不是新创建一个线程为其服务 , 而是从“池子”中选择一个空闲的线程为新的客户请求服务 ,服务完毕后 , 线程进入空闲线程池中。如果没有线程空闲 的 话, 就 将 数 据 包 暂 时 积 累 , 等 待 线 程 池 内 有 线 程空闲以后再进行处理。通过对多个任务重用已经存在的线程对象 , 降低了对线程对象创建和销毁的开销。当客户请求 时 , 线程对象 已 经 存 在 , 可 以 提 高 请 求 的响应时间 , 从而整体地提高了系统服务的表现。

一般来说实现一个线程池主要包括以下几个组成部分：

1）线程管理器：用于创建并管理线程池。

2）工作线程：线程池中实际执行任务的线程。在初始化线程时会预先创建好固定数目的线程在池中，这些初始化的线程一般处于空闲状态，一般不占用CPU，占用较小的内存空间。

3）任务接口：每个任务必须实现的接口，当线程池的任务队列中有可执行任务时，被空闲的工作线程调去执行（线程的闲与忙是通过互斥量实现的，跟前面文章中的设置标志位差不多），把任务抽象出来形成接口，可以做到线程池与具体的任务无关。

4）任务队列：用来存放没有处理的任务，提供一种缓冲机制，实现这种结构有好几种方法，常用的是队列，主要运用先进先出原理，另外一种是链表之类的**数据结构**，可以动态的为它分配内存空间，应用中比较灵活，下文中就是用到的链表。

下面的不在赘述百度《线程池技术在并发服务器中的应用》写的非常详细！

转自：http://blog.csdn.net/zouxinfox/article/details/3560891

什么时候需要创建线程池呢？简单的说，如果一个应用需要频繁的创建和销毁线程，而任务执行的时间又非常短，这样线程创建和销毁的带来的开销就不容忽视，这时也是线程池该出场的机会了。如果线程创建和销毁时间相比任务执行时间可以忽略不计，则没有必要使用线程池了。

下面是**Linux**系统下用**C语言**创建的一个线程池。线程池会维护一个任务链表(每个CThread_worker结构就是一个任务)。

pool_init()函数预先创建好max_thread_num个线程，每个线程执thread_routine ()函数。该函数中

```
1. while (pool->cur_queue_size == 0)
2. {
3.     pthread_cond_wait (&(pool->queue_ready),&(pool->queue_lock));
4. }
```

表示如果任务链表中没有任务，则该线程出于阻塞等待状态。否则从队列中取出任务并执行。

pool_add_worker()函数向线程池的任务链表中加入一个任务，加入后通过调用pthread_cond_signal (&(pool->queue_ready))唤醒一个出于阻塞状态的线程(如果有的话)。

pool_destroy ()函数用于销毁线程池，线程池任务链表中的任务不会再被执行，但是正在运行的线程会一直把任务运行完后再退出。

[cpp] view plain copy
print?

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <unistd.h>
4. #include <sys/types.h>
5. #include <pthread.h>
6. #include <assert.h>
7.
8. /*
9. *线程池里所有运行和等待的任务都是一个CThread_worker
10. *由于所有任务都在链表里，所以是一个链表结构
11. */
12. typedef struct worker
13. {
14.     /*回调函数，任务运行时会调用此函数，注意也可声明成其它形式*/
15.     void *(*process) (void *arg);
16.     void *arg;/*回调函数的参数*/
17.     struct worker *next;
18.
19. } CThread_worker;
20.
21.
22.
23. /*线程池结构*/
24. typedef struct
```

```
25. {
26.     pthread_mutex_t queue_lock;
27.     pthread_cond_t queue_ready;
28.
29.     /*链表结构，线程池中所有等待任务*/
30.     CThread_worker *queue_head;
31.
32.     /*是否销毁线程池*/
33.     int shutdown;
34.     pthread_t *threadid;
35.     /*线程池中允许的活动线程数目*/
36.     int max_thread_num;
37.     /*当前等待队列的任务数目*/
38.     int cur_queue_size;
39.
40. } CThread_pool;
41.
42.
43.
44. int pool_add_worker (void *(*process) (void *arg), void *arg);
45. void *thread_routine (void *arg);
46.
47.
48. //share resource
49. static CThread_pool *pool = NULL;
50. void
51. pool_init (int max_thread_num)
52. {
53.     pool = (CThread_pool *) malloc (sizeof (CThread_pool));
54.
55.     pthread_mutex_init (&(pool->queue_lock), NULL);
56.     pthread_cond_init (&(pool->queue_ready), NULL);
57.
58.     pool->queue_head = NULL;
59.
60.     pool->max_thread_num = max_thread_num;
61.     pool->cur_queue_size = 0;
62.
63.     pool->shutdown = 0;
64.
65.     pool-
>threadid = (pthread_t *) malloc (max_thread_num * sizeof (pthread_t));
66.     int i = 0;
67.     for (i = 0; i < max_thread_num; i++)
68.     {
69.         pthread_create (&(pool-
>threadid[i]), NULL, thread_routine,NULL);
70.     }
71. }
```

```
72.
73.
74.
75. /*向线程池中加入任务*/
76. int
77. pool_add_worker (void *(*process) (void *arg), void *arg)
78. {
79.     /*构造一个新任务*/
80.
    CThread_worker *newworker = (CThread_worker *) malloc (sizeof (CThread
_worker));
81.     newworker->process = process;
82.     newworker->arg = arg;
83.     newworker->next = NULL;/*别忘置空*/
84.
85.     pthread_mutex_lock (&(pool->queue_lock));
86.     /*将任务加入到等待队列中*/
87.     CThread_worker *member = pool->queue_head;
88.     if (member != NULL)
89.     {
90.         while (member->next != NULL)
91.             member = member->next;
92.         member->next = newworker;
93.     }
94.     else
95.     {
96.         pool->queue_head = newworker;
97.     }
98.
99.     assert (pool->queue_head != NULL);
100.
101.     pool->cur_queue_size++;
102.     pthread_mutex_unlock (&(pool->queue_lock));
103.     /*好了，等待队列中有任务了，唤醒一个等待线程;
104.     注意如果所有线程都在忙碌，这句没有任何作用*/
105.     pthread_cond_signal (&(pool->queue_ready));
106.     return 0;
107. }
108.
109.
110.
111. /*销毁线程池，等待队列中的任务不会再被执行，但是正在运行的线程会一直
112. 把任务运行完后再退出*/
113. int
114. pool_destroy ()
115. {
116.     if (pool->shutdown)
117.         return -1;/*防止两次调用*/
118.     pool->shutdown = 1;
```

```
119.
120.     /*唤醒所有等待线程，线程池要销毁了*/
121.     pthread_cond_broadcast (&(pool->queue_ready));
122.
123.     /*阻塞等待线程退出，否则就成僵尸了*/
124.     int i;
125.     for (i = 0; i < pool->max_thread_num; i++)
126.         pthread_join (pool->threadid[i], NULL);
127.     free (pool->threadid);
128.
129.     /*销毁等待队列*/
130.     CThread_worker *head = NULL;
131.     while (pool->queue_head != NULL)
132.     {
133.         head = pool->queue_head;
134.         pool->queue_head = pool->queue_head->next;
135.         free (head);
136.     }
137.     /*条件变量和互斥量也别忘了销毁*/
138.     pthread_mutex_destroy(&(pool->queue_lock));
139.     pthread_cond_destroy(&(pool->queue_ready));
140.
141.     free (pool);
142.     /*销毁后指针置空是个好习惯*/
143.     pool=NULL;
144.     return 0;
145. }
146.
147.
148.
149. void *
150. thread_routine (void *arg)
151. {
152.     printf ("starting thread 0x%x\n", pthread_self ());
153.     while (1)
154.     {
155.         pthread_mutex_lock (&(pool->queue_lock));
156.         /*如果等待队列为0并且不销毁线程池，则处于阻塞状态；注意
157.         pthread_cond_wait是一个原子操作，等待前会解锁，唤醒后会加锁*/
158.         while (pool->cur_queue_size == 0 && !pool->shutdown)
159.         {
160.             printf ("thread 0x%x is waiting\n", pthread_self ());
161.             pthread_cond_wait (&(pool->queue_ready), &(pool-
>queue_lock));
162.         }
163.
164.         /*线程池要销毁了*/
165.         if (pool->shutdown)
166.         {
```

```
167.             /*遇到break,continue,return等跳转语句，千万不要忘记先解锁*/
168.             pthread_mutex_unlock (&(pool->queue_lock));
169.             printf ("thread 0x%x will exit\n", pthread_self ());
170.             pthread_exit (NULL);
171.         }
172.
173.
        printf ("thread 0x%x is starting to work\n", pthread_self ());
174.
175.         /*assert是调试的好帮手*/
176.         assert (pool->cur_queue_size != 0);
177.         assert (pool->queue_head != NULL);
178.
179.         /*等待队列长度减去1，并取出链表中的头元素*/
180.         pool->cur_queue_size--;
181.         CThread_worker *worker = pool->queue_head;
182.         pool->queue_head = worker->next;
183.         pthread_mutex_unlock (&(pool->queue_lock));
184.
185.         /*调用回调函数，执行任务*/
186.         (*(worker->process)) (worker->arg);
187.         free (worker);
188.         worker = NULL;
189.     }
190.     /*这一句应该是不可达的*/
191.     pthread_exit (NULL);
192. }
193.
194. //     下面是测试代码
195.
196. void *
197. myprocess (void *arg)
198. {
199.
    printf ("threadid is 0x%x, working on task %d\n", pthread_self (),*
(int *) arg);
200.     sleep (1);/*休息一秒，延长任务的执行时间*/
201.     return NULL;
202. }
203.
204. int
205. main (int argc, char **argv)
206. {
207.     pool_init (3);/*线程池中最多三个活动线程*/
208.
209.     /*连续向池中投入10个任务*/
210.     int *workingnum = (int *) malloc (sizeof (int) * 10);
211.     int i;
212.     for (i = 0; i < 10; i++)
```

```
213.     {
214.         workingnum[i] = i;
215.         pool_add_worker (myprocess, &workingnum[i]);
216.     }
217.     /*等待所有任务完成*/
218.     sleep (5);
219.     /*销毁线程池*/
220.     pool_destroy ();
221.
222.     free (workingnum);
223.     return 0;
224. }
```

将上述所有代码放入threadpool.c文件中，

在Linux输入编译命令

$ gcc -o threadpool threadpool.c -lpthread

以下是运行结果

starting thread 0xb7df6b90

thread 0xb7df6b90 is waiting

starting thread 0xb75f5b90

thread 0xb75f5b90 is waiting

starting thread 0xb6df4b90

thread 0xb6df4b90 is waiting

thread 0xb7df6b90 is starting to work

threadid is 0xb7df6b90, working on task 0

thread 0xb75f5b90 is starting to work

threadid is 0xb75f5b90, working on task 1

thread 0xb6df4b90 is starting to work

threadid is 0xb6df4b90, working on task 2

thread 0xb7df6b90 is starting to work

threadid is 0xb7df6b90, working on task 3

thread 0xb75f5b90 is starting to work

threadid is 0xb75f5b90, working on task 4

thread 0xb6df4b90 is starting to work

threadid is 0xb6df4b90, working on task 5

thread 0xb7df6b90 is starting to work

threadid is 0xb7df6b90, working on task 6
thread 0xb75f5b90 is starting to work
threadid is 0xb75f5b90, working on task 7
thread 0xb6df4b90 is starting to work
threadid is 0xb6df4b90, working on task 8
thread 0xb7df6b90 is starting to work
threadid is 0xb7df6b90, working on task 9
thread 0xb75f5b90 is waiting
thread 0xb6df4b90 is waiting
thread 0xb7df6b90 is waiting
thread 0xb75f5b90 will exit
thread 0xb6df4b90 will exit
thread 0xb7df6b90 will exit