

26 | 使用阻塞I/O和线程模型：换一种轻量的方式

2019-10-07 盛延敏

网络编程实战

[进入课程 >](#)



讲述：冯永吉

时长 10:51 大小 9.95M



你好，我是盛延敏，这里是网络编程实战第 26 讲，欢迎回来。

在前面一讲中，我们使用了进程模型来处理用户连接请求，进程切换上下文的代价是比较高的，幸运的是，有一种轻量级的模型可以处理多用户连接请求，这就是线程模型。这一讲里，我们就来了解一下线程模型。

线程（thread）是运行在进程中的一个“逻辑流”，现代操作系统都允许在单进程中运行多个线程。线程由操作系统内核管理。每个线程都有自己的上下文（context），包括一个可以唯一标识线程的 ID（thread ID，或者叫 tid）、栈、程序计数器、寄存器等。在同一个进程中，所有的线程共享该进程的整个虚拟地址空间，包括代码、数据、堆、共享库等。

在前面的程序中，我们没有显式使用线程，但这不代表线程没有发挥作用。实际上，每个进程一开始都会产生一个线程，一般被称为为主线程，主线程可以再产生子线程，这样的主线程 - 子线程对可以叫做一个对等线程。

你可能会问，既然可以使用多进程来处理并发，为什么还要使用多线程模型呢？

简单来说，在同一个进程中，线程上下文切换的开销要比进程小得多。怎么理解线程上下文呢？我们的代码被 CPU 执行的时候，是需要一些数据支撑的，比如程序计数器告诉 CPU 代码执行到哪里了，寄存器里存了当前计算的一些中间值，内存里放置了一些当前用到的变量等，从一个计算场景，切换到另外一个计算场景，程序计数器、寄存器等这些值重新载入新场景的值，就是线程的上下文切换。

POSIX 线程模型

POSIX 线程是现代 UNIX 系统提供的处理线程的标准接口。POSIX 定义的线程函数大约有 60 多个，这些函数可以帮助我们创建线程、回收线程。接下来我们先看一个简单的例子程序。

 复制代码

```
1 int another_shared = 0;
2
3 void thread_run(void *arg) {
4     int *calculator = (int *) arg;
5     printf("hello, world, tid == %d \n", pthread_self());
6     for (int i = 0; i < 1000; i++) {
7         *calculator += 1;
8         another_shared += 1;
9     }
10 }
11
12 int main(int c, char **v) {
13     int calculator;
14
15     pthread_t tid1;
16     pthread_t tid2;
17
18     pthread_create(&tid1, NULL, thread_run, &calculator);
19     pthread_create(&tid2, NULL, thread_run, &calculator);
20
21     pthread_join(tid1, NULL);
22     pthread_join(tid2, NULL);
23
24     printf("calculator is %d \n", calculator);
```

```
25     printf("another_shared is %d \n", another_shared);
26 }
```

thread_helloworld 程序中，主线程依次创建了两个子线程，然后等待这两个子线程处理完毕之后终止。每个子线程都在对两个共享变量进行计算，最后在主线程中打印出最后的计算结果。

程序的第 18 和 19 行分别调用了 `pthread_create` 创建了两个线程，每个线程的入口都是 `thread_run` 函数，这里我们使用了 `calculator` 这个全局变量，并且通过传地址指针的方式，将这个值传给了 `thread_run` 函数。当调用 `pthread_create` 结束，子线程会立即执行，主线程在此后调用了 `pthread_join` 函数等待子线程结束。

运行这个程序，很幸运，计算的结果是正确的。

 复制代码

```
1 ./thread-helloworld
2 hello, world, tid == 125607936
3 hello, world, tid == 126144512
4 calculator is 2000
5 another_shared is 2000
```

主要线程函数

创建线程

正如前面看到，通过调用 `pthread_create` 函数来创建一个线程。这个函数的原型如下：

 复制代码

```
1 int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
2                     void *(*func)(void *), void *arg);
3
4 返回：若成功则为 0，若出错则为正的 Exxx 值
```

每个线程都有一个线程 ID (tid) 唯一来标识，其数据类型为 `pthread_t`，一般是 `unsigned int`。`pthread_create` 函数的第一个输出参数 `tid` 就是代表了线程 ID，如果创建线程成功，`tid` 就返回正确的线程 ID。

每个线程都会有很多属性，比如优先级、是否应该成为一个守护进程等，这些值可以通过 `pthread_attr_t` 来描述，一般我们不会特殊设置，可以直接指定这个参数为 `NULL`。

第三个参数为新线程的入口函数，该函数可以接收一个参数 `arg`，类型为指针，如果我们想给线程入口函数传多个值，那么需要把这些值包装成一个结构体，再把这个结构体的地址作为 `pthread_create` 的第四个参数，在线程入口函数内，再将该地址转为该结构体的指针对象。

在新线程的入口函数内，可以执行 `pthread_self` 函数返回线程 `tid`。

 复制代码

```
1 pthread_t pthread_self(void)
```

◀ ▶

终止线程

终止一个线程最直接的方法是在父线程内调用以下函数：

 复制代码

```
1 void pthread_exit(void *status)
```

◀ ▶

当调用这个函数之后，父线程会等待其他所有的子线程终止，之后父线程自己终止。

当然，如果一个子线程入口函数直接退出了，那么子线程也就自然终止了。所以，绝大多数的子线程执行体都是一个无限循环。

也可以通过调用 `pthread_cancel` 来主动终止一个子线程，和 `pthread_exit` 不同的是，它可以指定某个子线程终止。

 复制代码

```
1 int pthread_cancel(pthread_t tid)
```

复制代码

```
1 int pthread_join(pthread_t tid, void ** thread_return)
```

复制代码

当调用 `pthread_join` 时，主线程会阻塞，直到对应 `tid` 的子线程自然终止。和 `pthread_cancel` 不同的是，它不会强迫子线程终止。

分离线程

一个线程的重要属性是可结合的，或者是分离的。一个可结合的线程是能够被其他线程杀死和回收资源的；而一个分离的线程不能被其他线程杀死或回收资源。一般来说，默认的属性是可结合的。

我们可以通过调用 `pthread_detach` 函数可以分离一个线程：

复制代码

```
1 int pthread_detach(pthread_t tid)
```

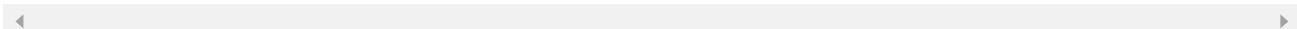
在高并发的例子里，每个连接都由一个线程单独处理，在这种情况下，服务器程序并不需要对每个子线程进行终止，这样的话，每个子线程可以在入口函数开始的地方，把自己设置为分离的，这样就能在它终止后自动回收相关的线程资源了，就不需要调用 `pthread_join` 函数了。

每个连接一个线程处理

接下来，我们改造一下服务器端程序。我们的目标是这样：每次有新的连接到达后，创建一个新线程，而不是用新进程来处理它。

 复制代码

```
1 #include "lib/common.h"
2
3 extern void loop_echo(int);
4
5 void thread_run(void *arg) {
6     pthread_detach(pthread_self());
7     int fd = (int) arg;
8     loop_echo(fd);
9 }
10
11 int main(int c, char **v) {
12     int listener_fd = tcp_server_listen(SERV_PORT);
13     pthread_t tid;
14
15     while (1) {
16         struct sockaddr_storage ss;
17         socklen_t slen = sizeof(ss);
18         int fd = accept(listener_fd, (struct sockaddr *) &ss, &slen);
19         if (fd < 0) {
20             error(1, errno, "accept failed");
21         } else {
22             pthread_create(&tid, NULL, &thread_run, (void *) fd);
23         }
24     }
25
26     return 0;
27 }
```



这个程序的第 18 行阻塞调用在 `accept` 上，一旦有新连接建立，阻塞调用返回，调用 `pthread_create` 创建一个子线程来处理这个连接。

描述连接最主要的是连接描述字，这里通过强制把描述字转换为 `void *` 指针的方式，完成了传值。如果你对这部分有点不理解，建议看一下 C 语言相关的指针部分内容。我们这里可以简单总结一下，虽然传的是一个指针，但是这个指针里存放的并不是一个地址，而是连接描述符的数值。

新线程入口函数 `thread_run` 里，第 6 行使用了 `pthread_detach` 方法，将子线程转变为分离的，也就意味着子线程独自负责线程资源回收。第 7 行，强制将指针转变为描述符数据，和前面将描述字转换为 `void *` 指针对应，第 8 行调用 `loop_echo` 方法处理这个连接的数据读写。

loop_echo 的程序如下，在接收客户端的数据之后，再编码回送出去。

 复制代码

```
1 char rot13_char(char c) {
2     if ((c >= 'a' && c <= 'm') || (c >= 'A' && c <= 'M'))
3         return c + 13;
4     else if ((c >= 'n' && c <= 'z') || (c >= 'N' && c <= 'Z'))
5         return c - 13;
6     else
7         return c;
8 }
9
10 void loop_echo(int fd) {
11     char outbuf[MAX_LINE + 1];
12     size_t outbuf_used = 0;
13     ssize_t result;
14     while (1) {
15         char ch;
16         result = recv(fd, &ch, 1, 0);
17
18         // 断开连接或者出错
19         if (result == 0) {
20             break;
21         } else if (result == -1) {
22             error(1, errno, "read error");
23             break;
24         }
25
26         if (outbuf_used < sizeof(outbuf)) {
27             outbuf[outbuf_used++] = rot13_char(ch);
28         }
29
30         if (ch == '\n') {
31             send(fd, outbuf, outbuf_used, 0);
32             outbuf_used = 0;
33             continue;
34         }
35     }
36 }
```

运行这个程序之后，开启多个 telnet 客户端，可以看到这个服务器程序可以处理多个并发连接并回送数据。单独一个连接退出也不会影响其他连接的数据收发。

 复制代码

```
1 $telnet 127.0.0.1 43211
```

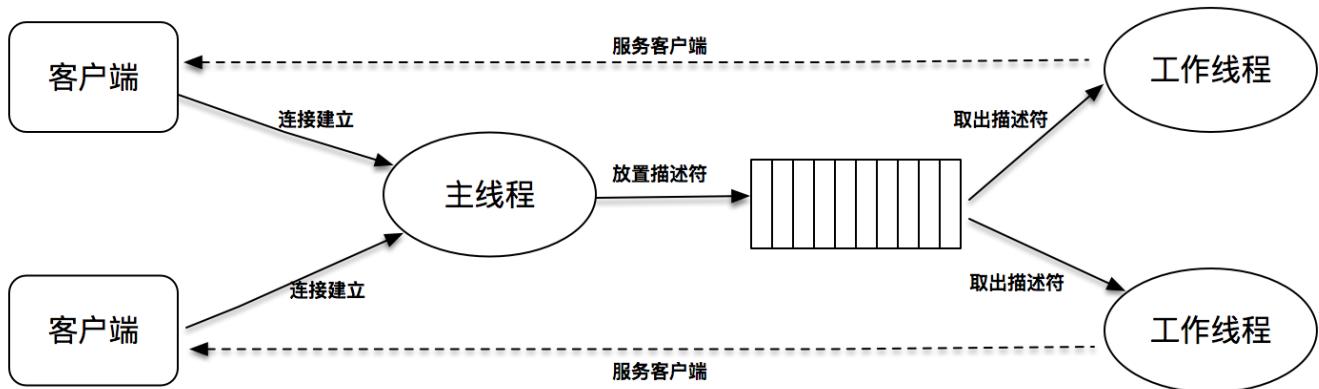
```
2 Trying 127.0.0.1...
3 Connected to localhost.
4 Escape character is '^].
5 aaa
6 nnn
7 ^]
8 telnet> quit
9 Connection closed.
```

构建线程池处理多个连接

上面的服务器端程序虽然可以正常工作，不过它有一个缺点，那就是如果并发连接过多，就会引起线程的频繁创建和销毁。虽然线程切换的上下文开销不大，但是线程创建和销毁的开销却是不小的。

能不能对这个程序进行一些优化呢？

我们可以使用预创建线程池的方式来进行优化。在服务器端启动时，可以先按照固定大小预创建出多个线程，当有新连接建立时，往连接字队列里放置这个新连接描述字，线程池里的线程负责从连接字队列里取出连接描述字进行处理。



这个程序的关键是连接字队列的设计，因为这里既有往这个队列里放置描述符的操作，也有从这个队列里取出描述符的操作。

对此，需要引入两个重要的概念，一个是锁 mutex，一个是条件变量 condition。锁很好理解，加锁的意思就是其他线程不能进入；条件变量则是在多个线程需要交互的情况下，用来线程间同步的原语。

```

1 // 定义一个队列
2 typedef struct {
3     int number; // 队列里的描述字最大个数
4     int *fd; // 这是一个数组指针
5     int front; // 当前队列的头位置
6     int rear; // 当前队列的尾位置
7     pthread_mutex_t mutex; // 锁
8     pthread_cond_t cond; // 条件变量
9 } block_queue;
10
11 // 初始化队列
12 void block_queue_init(block_queue *blockQueue, int number) {
13     blockQueue->number = number;
14     blockQueue->fd = calloc(number, sizeof(int));
15     blockQueue->front = blockQueue->rear = 0;
16     pthread_mutex_init(&blockQueue->mutex, NULL);
17     pthread_cond_init(&blockQueue->cond, NULL);
18 }
19
20 // 往队列里放置一个描述字 fd
21 void block_queue_push(block_queue *blockQueue, int fd) {
22     // 一定要先加锁, 因为有多个线程需要读写队列
23     pthread_mutex_lock(&blockQueue->mutex);
24     // 将描述字放到队列尾的位置
25     blockQueue->fd[blockQueue->rear] = fd;
26     // 如果已经到最后, 重置尾的位置
27     if (++blockQueue->rear == blockQueue->number) {
28         blockQueue->rear = 0;
29     }
30     printf("push fd %d", fd);
31     // 通知其他等待读的线程, 有新的连接字等待处理
32     pthread_cond_signal(&blockQueue->cond);
33     // 解锁
34     pthread_mutex_unlock(&blockQueue->mutex);
35 }
36
37 // 从队列里读出描述字进行处理
38 int block_queue_pop(block_queue *blockQueue) {
39     // 加锁
40     pthread_mutex_lock(&blockQueue->mutex);
41     // 判断队列里没有新的连接字可以处理, 就一直条件等待, 直到有新的连接字入队列
42     while (blockQueue->front == blockQueue->rear)
43         pthread_cond_wait(&blockQueue->cond, &blockQueue->mutex);
44     // 取出队列头的连接字
45     int fd = blockQueue->fd[blockQueue->front];
46     // 如果已经到最后, 重置头的位置
47     if (++blockQueue->front == blockQueue->number) {
48         blockQueue->front = 0;
49     }
50     printf("pop fd %d", fd);
51     // 解锁

```

```
52     pthread_mutex_unlock(&blockQueue->mutex);
53     // 返回连接字
54     return fd;
55 }
```

我在文稿里放置了 block_queue 相关的定义和实现，并在关键的地方加了一些蛛丝，有几个地方需要特别注意：

第一是记得对操作进行加锁和解锁，这里是通过 pthread_mutex_lock 和 pthread_mutex_unlock 来完成的。

第二是当工作线程没有描述字可用时，需要等待，第 43 行通过调用 pthread_cond_wait，所有的工作线程等待有新的描述字可达。第 32 行，主线程通知工作线程有新的描述符需要服务。

服务器端程序如下：

 复制代码

```
1 void thread_run(void *arg) {
2     pthread_t tid = pthread_self();
3     pthread_detach(tid);
4
5     block_queue *blockQueue = (block_queue *) arg;
6     while (1) {
7         int fd = block_queue_pop(blockQueue);
8         printf("get fd in thread, fd==%d, tid == %d", fd, tid);
9         loop_echo(fd);
10    }
11 }
12
13 int main(int c, char **v) {
14     int listener_fd = tcp_server_listen(SERV_PORT);
15
16     block_queue blockQueue;
17     block_queue_init(&blockQueue, BLOCK_QUEUE_SIZE);
18
19     thread_array = calloc(THREAD_NUMBER, sizeof(Thread));
20     int i;
21     for (i = 0; i < THREAD_NUMBER; i++) {
22         pthread_create(&(thread_array[i].thread_tid), NULL, &thread_run, (void *) &blockQueue);
23     }
24
25     while (1) {
```

```
26     struct sockaddr_storage ss;
27     socklen_t slen = sizeof(ss);
28     int fd = accept(listener_fd, (struct sockaddr *) &ss, &slen);
29     if (fd < 0) {
30         error(1, errno, "accept failed");
31     } else {
32         block_queue_push(&blockQueue, fd);
33     }
34 }
35
36     return 0;
37 }
```



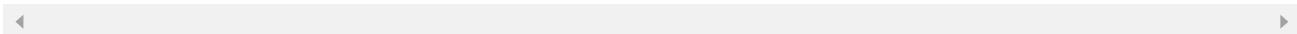
有了描述字队列，主程序变得非常简洁。第 19-23 行预创建了多个线程，组成了一个线程池。28-32 行在新连接建立后，将连接描述字加入到队列中。

7-9 行是工作线程的主循环，从描述字队列中取出描述字，对这个连接进行服务处理。

同样的，运行这个程序之后，开启多个 telnet 客户端，可以看到这个服务器程序可以正常处理多个并发连接并回显。

 复制代码

```
1 $telnet 127.0.0.1 43211
2 Trying 127.0.0.1...
3 Connected to localhost.
4 Escape character is '^]'.
5 aaa
6 nnn
7 ^]
8 telnet> quit
9 Connection closed.
```



和前面的程序相比，线程创建和销毁的开销大大降低，但因为线程池大小固定，又因为使用了阻塞套接字，肯定会出现有连接得不到及时服务的场景。这个问题的解决还是要回到我在开篇词里提到的方案上来，多路 I/O 复用加上线程来处理，仅仅使用阻塞 I/O 模型和线程是没有办法达到极致的高并发处理能力。

总结

这一讲，我们使用了线程来构建服务器端程序。一种是每次动态创建线程，另一种是使用线程池提高效率。和进程相比，线程的语义更轻量，使用的场景也更多。线程是高性能网络服务器必须掌握的核心知识，希望你能够通过本讲的学习，牢牢掌握它。

思考题

和往常一样，给你留两道思考题。

第一道，连接字队列的实现里，有一个重要情况没有考虑，就是队列里没有可用的位置了，想想看，如何对这种情况进行优化？

第二道，我在讲到第一个 hello-world 计数器应用时，说“结果是幸运”这是为什么呢？怎么理解呢？

欢迎你在评论区写下你的思考，我会和你一起思考，也欢迎把这篇文章分享给你的朋友或者同事，一起交流一下。

极客时间

网络编程实战

从底层到实战，深度解析网络编程

盛延敏

前大众点评云平台首席架构师

新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言 (3)

写留言



安排

2019-10-07

- 1.没位置可用可以选择丢弃，取出来直接关闭，等待对方重连，或者先判断队列是否有位置，没位置的话直接就不取出套接字，让它留在内核队列中，让内核处理。
- 2.存在竞态条件，需要加锁，不加锁可能大部分时间也能得到正确结果。

展开 ▼



程序水果宝

2019-10-07

thread_array = calloc(THREAD_NUMBER, sizeof(Thread));这个Thread是哪里来的，是C库的线程结构体吗？

展开 ▼



王小面

2019-10-07

利用假期时间，终于追上了进度

展开 ▼

