

[下载APP](#)

30 | 真正的大杀器：异步I/O探索

2019-10-16 盛延敏

网络编程实战

[进入课程 >](#)



讲述：冯永吉

时长 10:52 大小 9.96M



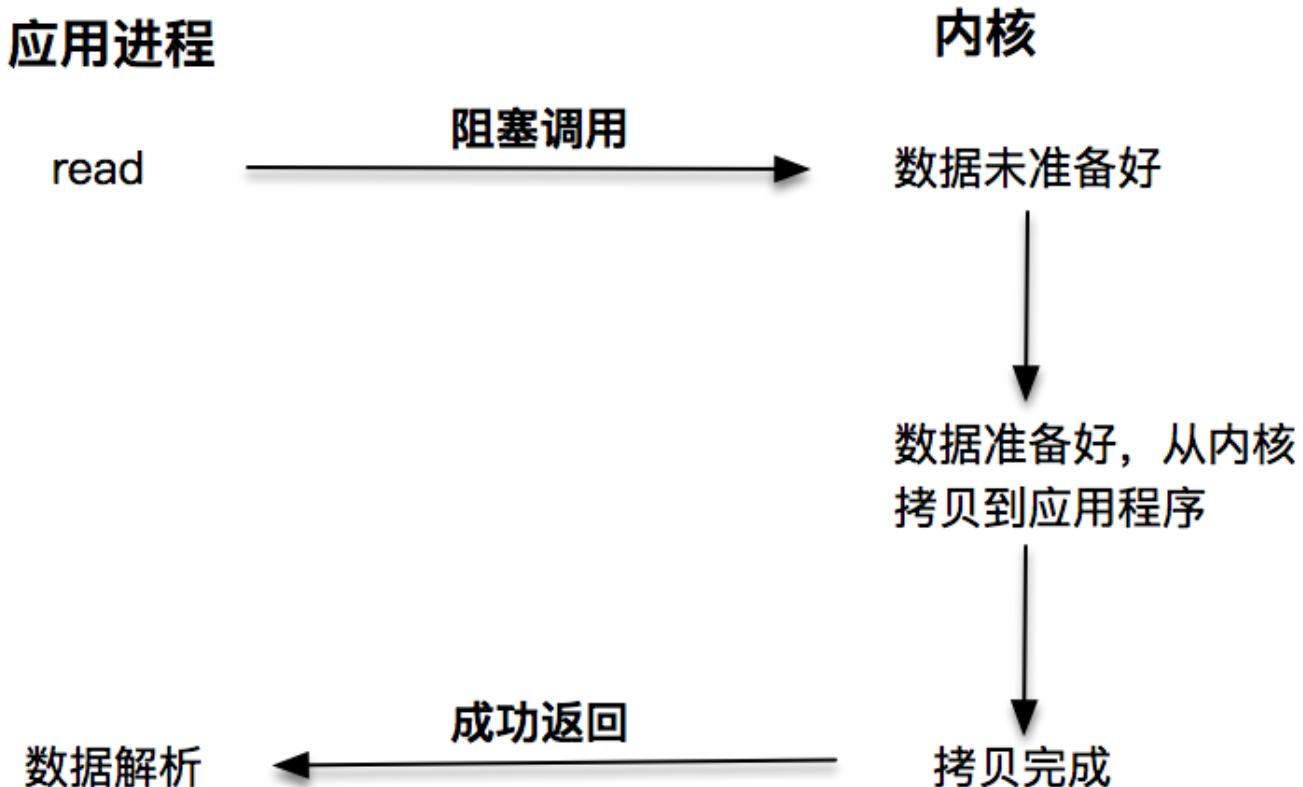
你好，我是盛延敏，这里是网络编程实战的第 30 讲，欢迎回来。

在性能篇的前几讲中，我们谈到了阻塞 I/O、非阻塞 I/O 以及像 select、poll、epoll 等 I/O 多路复用技术，并在此基础上结合线程技术，实现了以事件分发为核心的 reactor 反应堆模式。你或许还听说过一个叫做 Proactor 的网络事件驱动模式，这个 Proactor 模式和 reactor 模式到底有什么区别和联系呢？在今天的内容中，我们先讲述异步 I/O，再一起揭开以异步 I/O 为基础的 proactor 模式的面纱。

阻塞 / 非阻塞 VS 同步 / 异步

尽管在前面的课程中，多少都涉及到了阻塞、非阻塞、同步、异步的概念，但为了避免看见这些概念一头雾水，今天，我们就先来梳理一下这几个概念。

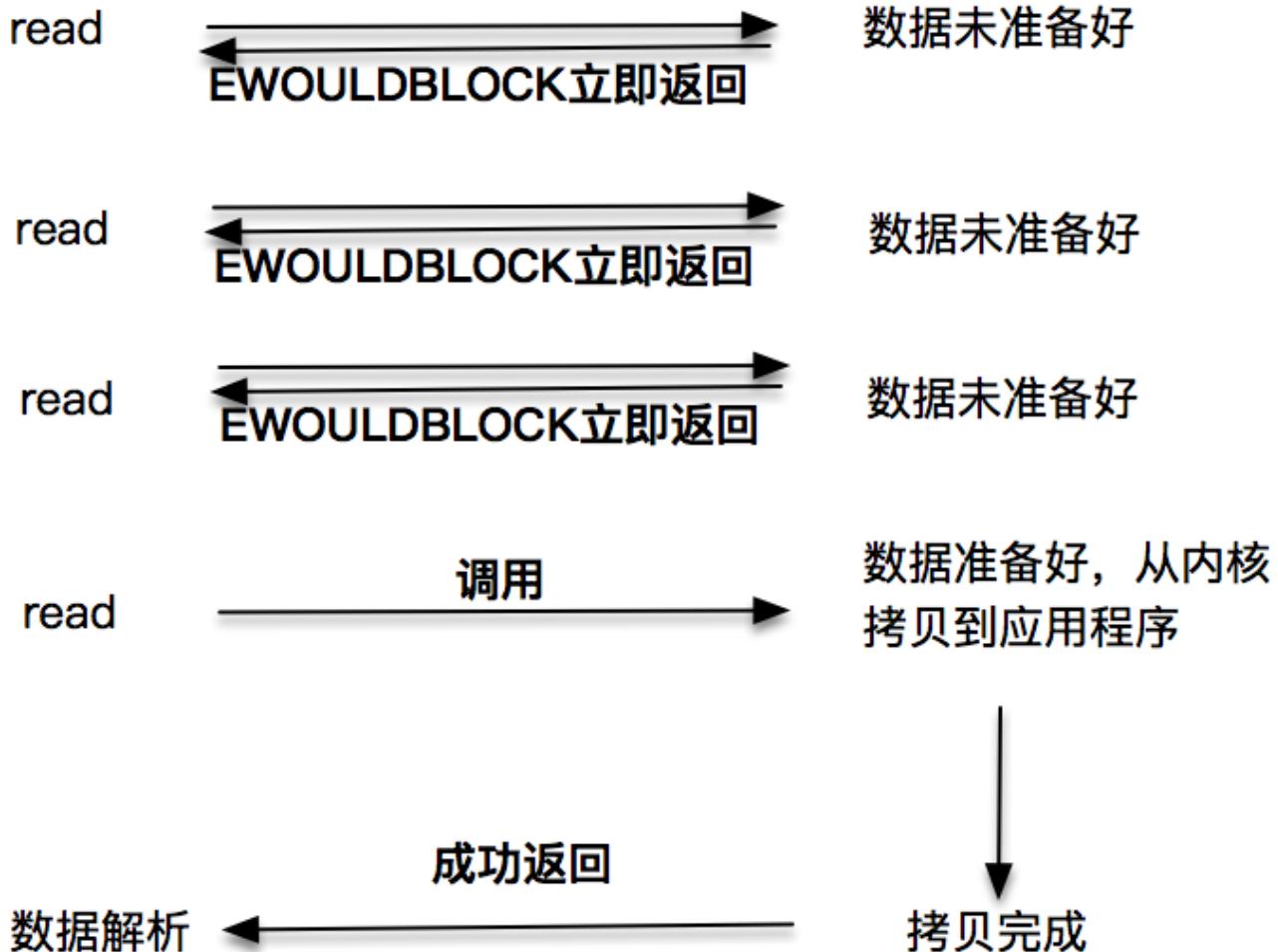
第一种是阻塞 I/O。阻塞 I/O 发起的 read 请求，线程会被挂起，一直等到内核数据准备好，并把数据从内核区域拷贝到应用程序的缓冲区中，当拷贝过程完成，read 请求调用才返回。接下来，应用程序就可以对缓冲区的数据进行数据解析。



第二种是非阻塞 I/O。非阻塞的 read 请求在数据未准备好的情况下立即返回，应用程序可以不断轮询内核，直到数据准备好，内核将数据拷贝到应用程序缓冲，并完成这次 read 调用。注意，这里最后一次 read 调用，获取数据的过程，**是一个同步的过程。这里的同步指的是内核区域的数据拷贝到缓存区这个过程**。

应用进程

内核

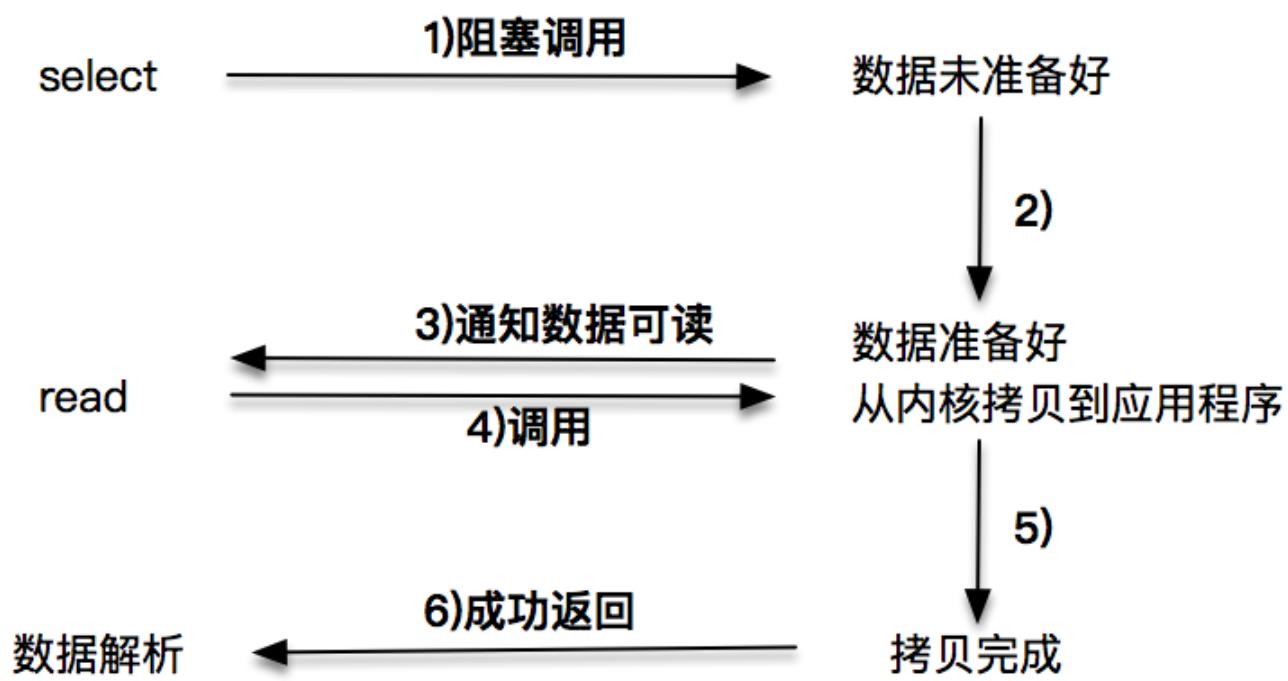


每次让应用程序去轮询内核的 I/O 是否准备好，是一个不经济的做法，因为在轮询的过程中应用进程啥也不能干。于是，像 `select`、`poll` 这样的 I/O 多路复用技术就隆重登场了。通过 I/O 事件分发，当内核数据准备好时，再通知应用程序进行操作。这个做法大大改善了应用进程对 CPU 的利用率，在没有被通知的情况下，应用进程可以使用 CPU 做其他的事情。

注意，这里 `read` 调用，获取数据的过程，也是一个同步的过程。

应用进程

内核



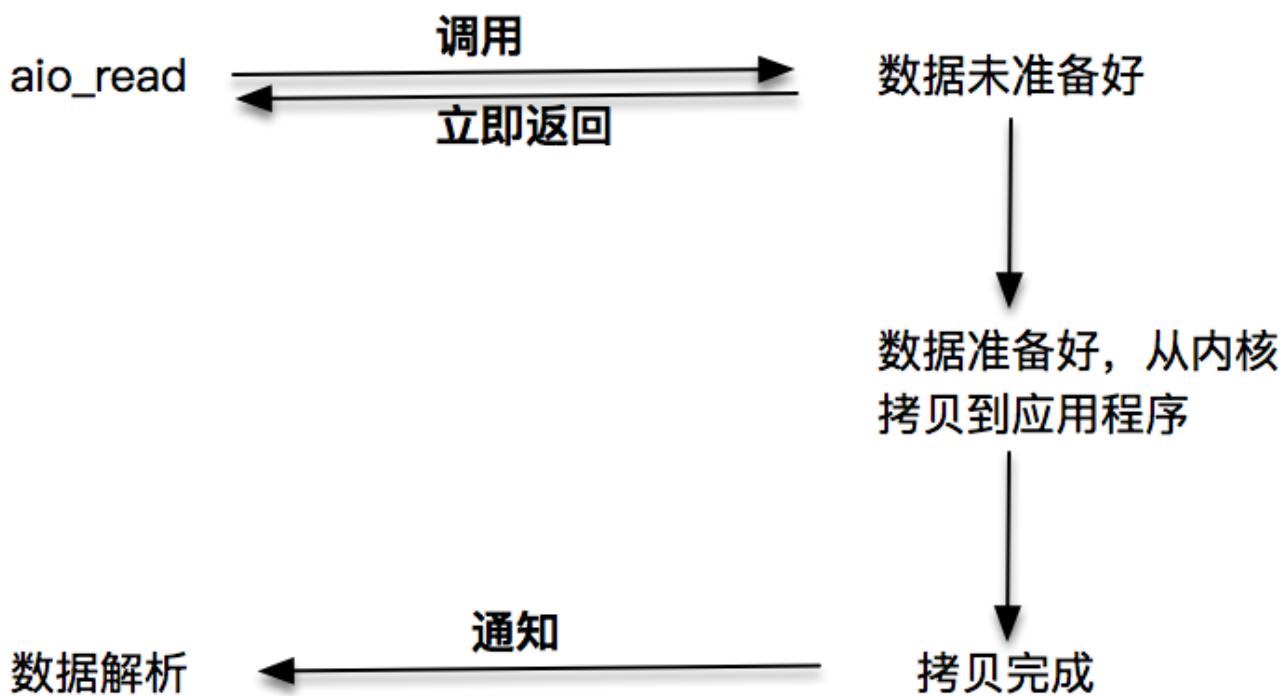
第一种阻塞 I/O 我想你已经比较了解了，在阻塞 I/O 的情况下，应用程序会被挂起，直到获取数据。第二种非阻塞 I/O 和第三种基于非阻塞 I/O 的多路复用技术，获取数据的操作不会被阻塞。

无论是第一种阻塞 I/O，还是第二种非阻塞 I/O，第三种基于非阻塞 I/O 的多路复用都是同步调用技术。为什么这么说呢？因为同步调用、异步调用的说法，是对于获取数据的过程而言的，前面几种最后获取数据的 `read` 操作调用，都是同步的，在 `read` 调用时，内核将数据从内核空间拷贝到应用程序空间，这个过程是在 `read` 函数中同步进行的，如果内核实现的拷贝效率很差，`read` 调用就会在这个同步过程中消耗比较长的时间。

而真正的异步调用则不用担心这个问题，我们接下来就来介绍第四种 I/O 技术，当我们发起 `aio_read` 之后，就立即返回，内核自动将数据从内核空间拷贝到应用程序空间，这个拷贝过程是异步的，内核自动完成的，和前面的同步操作不一样，应用程序并不需要主动发起拷贝动作。

应用进程

内核



还记得[第 22 讲](#)中讲到的去书店买书的例子吗？基于这个例子，针对以上的场景，我们可以这么理解。

第一种阻塞 I/O 就是你去了书店，告诉老板你想要某本书，然后你就一直在那里等着，直到书店老板翻箱倒柜找到你想要的书。

第二种非阻塞 I/O 类似于你去了书店，问老板有没有一本书，老板告诉你没有，你就离开了。一周以后，你又来这个书店，再问这个老板，老板一查，有了，于是你买了这本书。

第三种基于非阻塞的 I/O 多路复用，你来到书店告诉老板：“老板，到货给我打电话吧，我再来付钱取书。”

第四种异步 I/O 就是你连去书店取书的过程也想省了，你留下地址，付了书费，让老板到货时寄给你，你直接在家里拿到就可以看了。

这里放置了一张表格，总结了以上几种 I/O 模型。

同步I/O	阻塞I/O	非阻塞I/O	select、poll、epoll等多路复用+非阻塞I/O
异步I/O	异步I/O (aio)		

aio_read 和 aio_write 的用法

听起来，异步 I/O 有一种高大上的感觉。其实，异步 I/O 用起来倒是挺简单的。下面我们看下一个具体的例子：

 复制代码

```
1 #include "lib/common.h"
2 #include <aio.h>
3
4 const int BUF_SIZE = 512;
5
6 int main() {
7     int err;
8     int result_size;
9
10    // 创建一个临时文件
11    char tmpname[256];
12    snprintf(tmpname, sizeof(tmpname), "/tmp/aio_test_%d", getpid());
13    unlink(tmpname);
14    int fd = open(tmpname, O_CREAT | O_RDWR | O_EXCL, S_IRUSR | S_IWUSR);
15    if (fd == -1) {
16        error(1, errno, "open file failed ");
17    }
18
19    char buf[BUF_SIZE];
20    struct aiocb aiocb;
21
22    // 初始化 buf 缓冲，写入的数据应该为 0xfafa 这样的，
23    memset(buf, 0xfa, BUF_SIZE);
24    memset(&aiocb, 0, sizeof(struct aiocb));
25    aiocb.aio_fildes = fd;
26    aiocb.aio_buf = buf;
27    aiocb.aio_nbytes = BUF_SIZE;
28
29    // 开始写
30    if (aio_write(&aiocb) == -1) {
31        printf(" Error at aio_write(): %s\n", strerror(errno));
32        close(fd);
33        exit(1);
34    }
35
36    // 因为是异步的，需要判断什么时候写完
37    while (aio_error(&aiocb) == EINPROGRESS) {
38        printf("writing... \n");
39    }
40
41    // 判断写入的是否正确
42    err = aio_error(&aiocb);
43    result_size = aio_return(&aiocb);
```

```
44     if (err != 0 || result_size != BUF_SIZE) {
45         printf(" aio_write failed() : %s\n", strerror(err));
46         close(fd);
47         exit(1);
48     }
49
50     // 下面准备开始读数据
51     char buffer[BUF_SIZE];
52     struct aiocb cb;
53     cb.aio_nbytes = BUF_SIZE;
54     cb.aio_fildes = fd;
55     cb.aio_offset = 0;
56     cb.aio_buf = buffer;
57
58     // 开始读数据
59     if (aio_read(&cb) == -1) {
60         printf(" air_read failed() : %s\n", strerror(err));
61         close(fd);
62     }
63
64     // 因为是异步的，需要判断什么时候读完
65     while (aio_error(&cb) == EINPROGRESS) {
66         printf("Reading... \n");
67     }
68
69     // 判断读是否成功
70     int numBytes = aio_return(&cb);
71     if (numBytes != -1) {
72         printf("Success.\n");
73     } else {
74         printf("Error.\n");
75     }
76
77     // 清理文件句柄
78     close(fd);
79     return 0;
80 }
```

这个程序展示了如何使用 aio 系列函数来完成异步读写。主要用到的函数有：

aio_write: 用来向内核提交异步写操作；

aio_read: 用来向内核提交异步读操作；

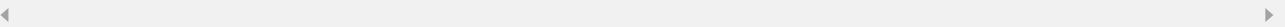
aio_error: 获取当前异步操作的状态；

aio_return: 获取异步操作读、写的字节数。

这个程序一开始使用 aio_write 方法向内核提交了一个异步写文件的操作。第 23-27 行是这个异步写操作的结构体。结构体 aiocb 是应用程序和操作系统内核传递的异步申请数据结构，这里我们使用了文件描述符、缓冲区指针 aio_buf 以及需要写入的字节数 aio_nbytes。

 复制代码

```
1 struct aiocb {  
2     int      aio_fildes;          /* File descriptor */  
3     off_t    aio_offset;         /* File offset */  
4     volatile void *aio_buf;       /* Location of buffer */  
5     size_t   aio_nbytes;        /* Length of transfer */  
6     int      aio_reqprio;        /* Request priority offset */  
7     struct sigevent  aio_sigevent; /* Signal number and value */  
8     int      aio_lio_opcode;     /* Operation to be performed */  
9 };
```



这里我们用了一个 0xfa 的缓冲区，这在后面的演示中可以看到结果。

30-34 行向系统内核申请了这个异步写操作，并且在 37-39 行查询异步动作的结果，当其结束时在 42-48 行判断写入的结果是否正确。

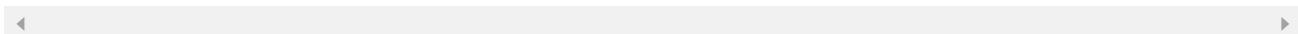
紧接着，我们使用了 aio_read 从文件中读取这些数据。为此，我们准备了一个新的 aiocb 结构体，告诉内核需要把数据拷贝到 buffer 这个缓冲区中，和异步写一样，发起异步读之后在第 65-67 行一直查询异步读动作的结果。

接下来运行这个程序，我们看到屏幕上打印出一系列的字符，显示了这个操作是有内核在后台帮我们完成的。

 复制代码

```
1 ./aio01  
2 writing...  
3 writing...  
4 writing...  
5 writing...  
6 writing...  
7 writing...  
8 writing...  
9 writing...  
10 writing...  
11 writing...
```

```
12 writing...
13 writing...
14 writing...
15 writing...
16 Reading...
17 Reading...
18 Reading...
19 Reading...
20 Reading...
21 Reading...
22 Reading...
23 Reading...
24 Reading...
25 Success.
```



打开 /tmp 目录下的 aio_test_xxxx 文件，可以看到，这个文件成功写入了我们期望的数据。

```
1 00000000: fafa fafa fafa fafa fafa fafa fafa fafa .....
2 00000010: fafa fafa fafa fafa fafa fafa fafa fafa .....
3 00000020: fafa fafa fafa fafa fafa fafa fafa fafa 它...同...同...同...同...同...
4 00000030: fafa fafa fafa fafa fafa fafa fafa fafa .....
5 00000040: fafa fafa fafa fafa fafa fafa fafa fafa .....
6 00000050: fafa fafa fafa fafa fafa fafa fafa fafa .....
7 00000060: fafa fafa fafa fafa fafa fafa fafa fafa .....
8 00000070: fafa fafa fafa fafa fafa fafa fafa fafa .....
```

请注意，以上的读写，都不需要我们在应用程序里再发起调用，系统内核直接帮我们做好了。

Linux 下 socket 套接字的异步支持

aio 系列函数是由 POSIX 定义的异步操作接口，可惜的是，Linux 下的 aio 操作，不是真正的操作系统级别支持的，它只是由 GNU libc 库函数在用户空间借由 pthread 方式实现的，而且仅仅针对磁盘类 I/O，套接字 I/O 不支持。

也有很多 Linux 的开发者尝试在操作系统内核中直接支持 aio，例如一个叫做 Ben LaHaise 的人，就将 aio 实现成功 merge 到 2.5.32 中，这部分能力是作为 patch 存在的，但是，它依旧不支持套接字。

Solaris 倒是有真正的系统级别的 aio，不过还不是很确定它在套接字上的性能表现，特别是和磁盘 I/O 相比效果如何。

综合以上结论就是，Linux 下对异步操作的支持非常有限，这也是为什么使用 epoll 等多路分发技术加上非阻塞 I/O 来解决 Linux 下高并发高性能网络 I/O 问题的根本原因。

Windows 下的 IOCP 和 Proactor 模式

和 Linux 不同，Windows 下实现了一套完整的支持套接字的异步编程接口，这套接口一般被叫做 IOCompletionPort(IOCP)。

这样，就产生了基于 IOCP 的所谓 Proactor 模式。

和 Reactor 模式一样，Proactor 模式也存在一个无限循环运行的 event loop 线程，但是不同于 Reactor 模式，这个线程并不负责处理 I/O 调用，它只是负责在对应的 read、write 操作完成的情况下，分发完成事件到不同的处理函数。

这里举一个 HTTP 服务请求的例子来说明：

1. 客户端发起一个 GET 请求；
2. 这个 GET 请求对应的字节流被内核读取完成，内核将这个完成事件放置到一个队列中；
3. event loop 线程，也就是 Proactor 从这个队列里获取事件，根据事件类型，分发到不同的处理函数上，比如一个 http handle 的 onMessage 解析函数；
4. HTTP request 解析函数完成报文解析；
5. 业务逻辑处理，比如读取数据库的记录；
6. 业务逻辑处理完成，开始 encode，完成之后，发起一个异步写操作；
7. 这个异步写操作被内核执行，完成之后这个异步写操作被放置到内核的队列中；
8. Proactor 线程获取这个完成事件，分发到 HTTP handler 的 onWriteCompleted 方法执行。

从这个例子可以看出，由于系统内核提供了真正的“异步”操作，Proactor 不会再像 Reactor 一样，每次感知事件后再调用 read、write 方法完成数据的读写，它只负责感知事件完成，并由对应的 handler 发起异步读写请求，I/O 读写操作本身是由系统内核完成的。和前面看到的 aio 的例子一样，这里需要传入数据缓冲区的地址等信息，这样，系统内核才可以自动帮我们把数据的读写工作完成。

无论是 Reactor 模式，还是 Proactor 模式，都是一种基于事件分发的网络编程模式。**Reactor 模式是基于待完成的 I/O 事件，而 Proactor 模式则是基于已完成的 I/O 事件**，两者本质，都是借由事件分发的思想，设计出可兼容、可扩展、接口友好的一套程序框架。

总结

和同步 I/O 相比，异步 I/O 的读写动作由内核自动完成，不过，在 Linux 下目前仅仅支持简单的基于本地文件的 aio 异步操作，这也使得我们在编写高性能网络程序时，首选 Reactor 模式，借助 epoll 这样的 I/O 分发技术完成开发；而 Windows 下的 IOCP 则是一种异步 I/O 的技术，并由此产生了和 Reactor 齐名的 Proactor 模式，借助这种模式，可以完成 Windows 下高性能网络程序设计。

思考题

和往常一样，给大家布置两道思考题：

1. 你可以查一查 Linux 的资料，看看为了在内核层面支持完全的异步 I/O，Linux 的世界里都发生了什么？
2. 在例子程序里，`aio_error` 一直处于占用 CPU 轮询异步操作的状态，有没有别的方法可以改进一下，比如挂起调用者、设置超时时间等？

欢迎你在评论区写下你的思考，也欢迎把这篇文章分享给你的朋友或者同事，一起交流进步一下。



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。

上一篇 29 | 漫入佳境：使用epoll和多线程模型

下一篇 31 | 性能篇答疑--epoll源码深度剖析

精选留言 (13)

 写留言

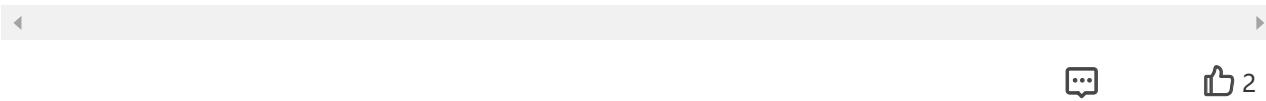


传说中的成大大

2019-10-16

看第二遍理解了reactor和proactor的区别前者是同步 有消息到达时调用应用程序的回调，应用程序自己调用read 同步取得数据,而后者是内核异步数据读取完成之后才调用应用程序的回调

作者回复：我理解Linux下标榜的proactor其实都是伪的。



fackgc17

2019-10-16

Linux 的 AIO 机制可能后面逐渐不用了，可以关注 5.1 的 io_uring 机制，大杀器

作者回复: 赞, 学习了。

 传说中的成大大
2019-10-19

issue和mr是啥意思啊, 没接触到过呢!

展开 ▾

 Steiner
2019-10-18

应该这么说吧, 同步和异步是指数据准备过程, 阻塞非阻塞是数据获取过程

作者回复: 也可以这么说吧。

 传说中的成大大
2019-10-17

```
在poll-server-onethread程序中 onMessage回调里面调用 char *run_cmd(char *cmd) {  
    char *data = malloc(16384);  
    bzero(data, sizeof(data));  
    FILE *fdp;  
    const int max_buffer = 256;...}
```

展开 ▾

作者回复: 从onMessage如何调到run_cmd的?

 传说中的成大大
2019-10-17

我把github上的代码进行了改进, 收到消息时执行run_cmd 用来实现ls pwd ...的shell命令, 但是总是提示: not found 原谅我抄的代码, 只是对代码进行了逻辑修改, 百度了半天都解决不了这个问题

作者回复: 贴你的代码过来, 大家一起会诊。



沉淀的梦想

2019-10-17

Proactor中所谓的队列，我的理解是一个Block Queue，给aio注册一个回调函数，回调函数的内容是往BlockQueue中放置一个通知，然后event loop线程苏醒，获取到这个通知后进行分发，不知道理解的对不对？

还有一个疑问POSIX的aio库要怎么注册回调？Java里面的aio有这个功能，感觉linux也...

展开 ▾

作者回复: Java是一个跨OS的语言，AIO的实施需充分调用OS参与，我理解可能对windows支持的比较好，Linux支持的一般吧。



阿西吧

2019-10-16

还有比异步IO更好的吗？

展开 ▾

作者回复: 我知道的这个已经把该办的事情都办了。



传说中的成大大

2019-10-16

我也想知道应该怎么取设计和封装接口函数 类等等

展开 ▾



传说中的成大大

2019-10-16

老师 你好 我要怎么样才能像你一样设计一个服务器框架呢？我需要哪些知识储备呢？

作者回复: 把我的代码看懂，然后搞清楚原理，自己试着慢慢撸一个。



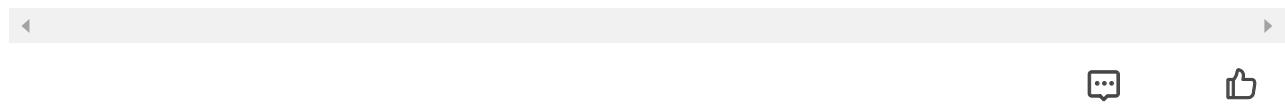
程序水果宝

2019-10-16

看了最近几篇文章以后个人感觉应该把反应堆、epoll、异步和同步的函数列出来配合着它们的功能讲，很有可能不懂的地方都在那些封装的函数里面，像main函数里面的内容反而给出链接加注释就可以了，这样可能会让人的理解更加深刻一些。还有实验结果也不用列这么多，这些完全可以由自己去实验。

展开 ▼

作者回复: 感谢你的建议。时间有限，做出来的内容可能没有办法满足所有人的需求。在第四篇里可能会解答你的大部分疑惑，如果有进一步的问题，我可以在答疑中统一回复，解答大家的疑惑。

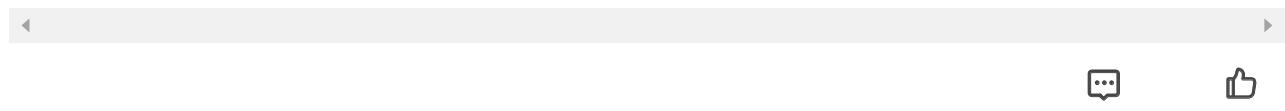


传说中的成大大

2019-10-16

而突然又理解到了同步i/o和异步i/o的问题 比如我调用read函数 在read函数返回之前数据被拷贝到缓冲区这个过程就是同步i/o的操作 像后面的aio系列函数 是在函数调用后 内核把数据拷贝到应用层缓冲区 这个就叫异步

作者回复: 你真的悟道了，哈哈:)



传说中的成大大

2019-10-16

再第二遍读的时候 我突然理清了 阻塞/非阻塞 io 和同步/异步io 这里提到的都是跟i/o操作相关 我又想起了线程的同步和异步 跟阻塞和非阻塞 没有半毛钱的关系啊。

作者回复: 好像有点悟道的意思.....

