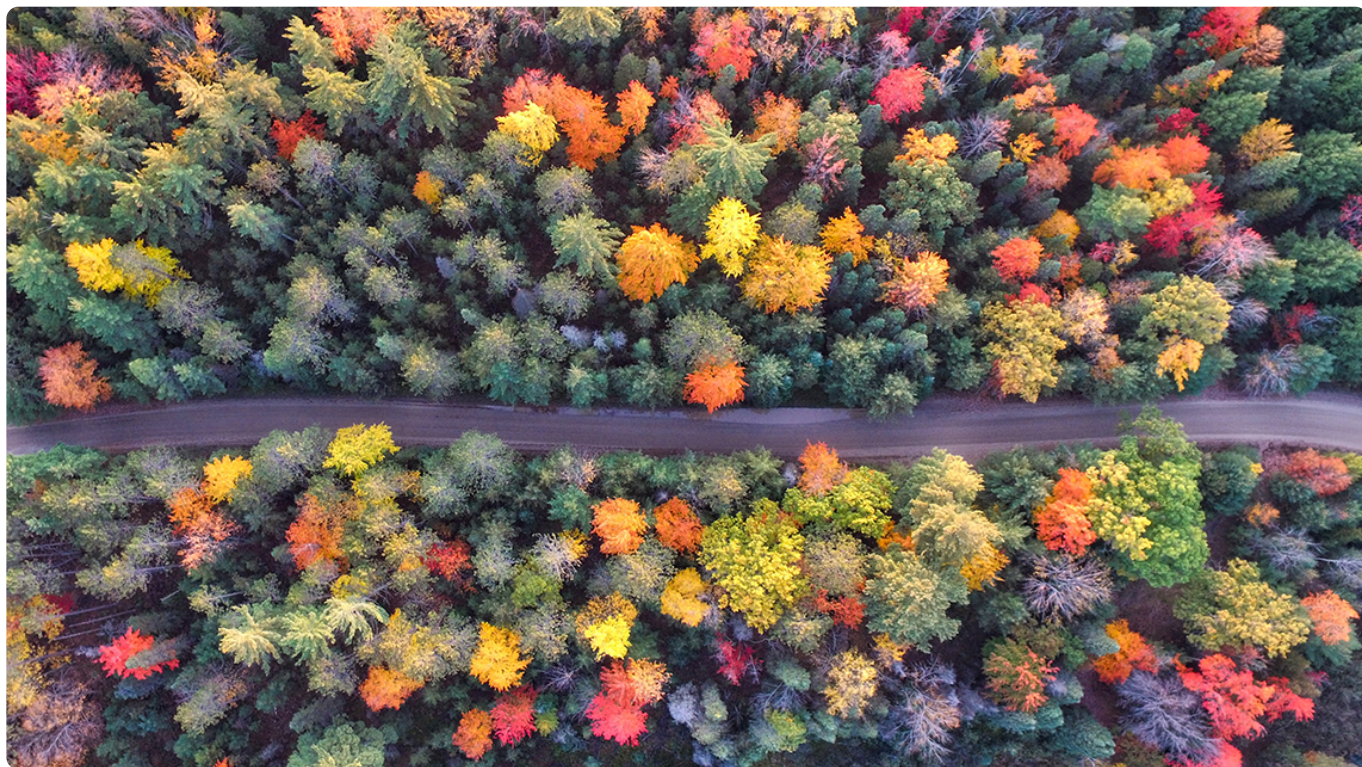


## 29 | 渐入佳境：使用epoll和多线程模型

2019-10-14 盛延敏

网络编程实战

[进入课程 >](#)



讲述：冯永吉

时长 08:05 大小 7.41M



你好，我是盛延敏，这里是网络编程实战第 29 讲，欢迎回来。


在前面的第 27 讲和第 28 讲中，我介绍了基于 poll 事件分发的 reactor 反应堆模式，以及主从反应堆模式。我们知道，和 poll 相比，Linux 提供的 epoll 是一种更为高效的事件分发机制。在这一讲里，我们将切换到 epoll 实现的主从反应堆模式，并且分析一下为什么 epoll 的性能会强于 poll 等传统的事件分发机制。

### 如何切换到 epoll

我已经将所有的代码已经放置到[GitHub](#)上，你可以自行查看或下载。


我们的网络编程框架是可以同时支持 poll 和 epoll 机制的，那么如何开启 epoll 的支持呢？

lib/event\_loop.c 文件的 event\_loop\_init\_with\_name 函数是关键，可以看到，这里是通过宏 EPOLL\_ENABLE 来决定是使用 epoll 还是 poll 的。

 复制代码


```
1 struct event_loop *event_loop_init_with_name(char *thread_name) {
2     ...
3     #ifdef EPOLL_ENABLE
4         yolanda_msgx("set epoll as dispatcher, %s", eventLoop->thread_name);
5         eventLoop->eventDispatcher = &epoll_dispatcher;
6     #else
7         yolanda_msgx("set poll as dispatcher, %s", eventLoop->thread_name);
8         eventLoop->eventDispatcher = &poll_dispatcher;
9     #endif
10    eventLoop->event_dispatcher_data = eventLoop->eventDispatcher->init(eventLoop);
11    ...
12 }
```

在根目录下的 CMakeLists.txt 文件里，引入 CheckSymbolExists，如果系统里有 epoll\_create 函数和 sys/epoll.h，就自动开启 EPOLL\_ENABLE。如果没有，EPOLL\_ENABLE 就不会开启，自动使用 poll 作为默认的事件分发机制。

 复制代码

```
1 # check epoll and add config.h for the macro compilation
2 include(CheckSymbolExists)
3 check_symbol_exists(epoll_create "sys/epoll.h" EPOLL_EXISTS)
4 if (EPOLL_EXISTS)
5     #    Linux 下设置为 epoll
6     set(EPOLL_ENABLE 1 CACHE INTERNAL "enable epoll")
7
8     #    Linux 下也设置为 poll
9     #    set(EPOLL_ENABLE "" CACHE INTERNAL "not enable epoll")
10 else ()
11     set(EPOLL_ENABLE "" CACHE INTERNAL "not enable epoll")
12 endif ()
```

但是，为了能让编译器使用到这个宏，需要让 CMake 往 config.h 文件里写入这个宏的最终值，configure\_file 命令就是起这个作用的。其中 config.h.cmake 是一个模板文件，已经预先创建在根目录下。同时还需要让编译器 include 这个 config.h 文件。include\_directories 可以帮我们达成这个目标。

 复制代码

```
1 configure_file(${CMAKE_CURRENT_SOURCE_DIR}/config.h.cmake
2               ${CMAKE_CURRENT_BINARY_DIR}/include/config.h)
3
4 include_directories(${CMAKE_CURRENT_BINARY_DIR}/include)
```

这样，在 Linux 下，就会默认使用 epoll 作为事件分发。

那么前面的[27 讲](#)和[28 讲](#)中的程序案例如何改为使用 poll 的呢？

我们可以修改 CMakeLists.txt 文件，把 Linux 下设置为 poll 的那段注释下的命令打开，同时关闭掉原先设置为 1 的命令就可以了。下面就是具体的示例代码。

 复制代码

```
1 # check epoll and add config.h for the macro compilation
2 include(CheckSymbolExists)
3 check_symbol_exists(epoll_create "sys/epoll.h" EPOLL_EXISTS)
4 if (EPOLL_EXISTS)
5     # Linux 下也设置为 poll
6     set(EPOLL_ENABLE "" CACHE INTERNAL "not enable epoll")
7 else ()
8     set(EPOLL_ENABLE "" CACHE INTERNAL "not enable epoll")
9 endif (
```

不管怎样，现在我们得到了一个 Linux 下使用 epoll 作为事件分发的版本，现在让我们使用它来编写程序吧。

## 样例程序

我们的样例程序和[第 28 讲](#)的一模一样，只是现在我们的事件分发机制从 poll 切换到了 epoll。

```
1 #include <lib/acceptor.h>
2 #include "lib/common.h"
3 #include "lib/event_loop.h"
4 #include "lib/tcp_server.h"
5
6 char rot13_char(char c) {
7     if ((c >= 'a' && c <= 'm') || (c >= 'A' && c <= 'M'))
8         return c + 13;
9     else if ((c >= 'n' && c <= 'z') || (c >= 'N' && c <= 'Z'))
10         return c - 13;
11     else
12         return c;
13 }
14
15 // 连接建立之后的 callback
16 int onConnectionCompleted(struct tcp_connection *tcpConnection) {
17     printf("connection completed\n");
18     return 0;
19 }
20
21 // 数据读到 buffer 之后的 callback
22 int onMessage(struct buffer *input, struct tcp_connection *tcpConnection) {
23     printf("get message from tcp connection %s\n", tcpConnection->name);
24     printf("%s", input->data);
25
26     struct buffer *output = buffer_new();
27     int size = buffer_readable_size(input);
28     for (int i = 0; i < size; i++) {
29         buffer_append_char(output, rot13_char(buffer_read_char(input)));
30     }
31     tcp_connection_send_buffer(tcpConnection, output);
32     return 0;
33 }
34
35 // 数据通过 buffer 写完之后的 callback
36 int onWriteCompleted(struct tcp_connection *tcpConnection) {
37     printf("write completed\n");
38     return 0;
39 }
40
41 // 连接关闭之后的 callback
42 int onConnectionClosed(struct tcp_connection *tcpConnection) {
43     printf("connection closed\n");
44     return 0;
45 }
46
47 int main(int c, char **v) {
48     // 主线程 event_loop
49     struct event_loop *eventLoop = event_loop_init();
50 }
```

```

51 // 初始化 acceptor
52 struct acceptor *acceptor = acceptor_init(SERV_PORT);
53
54 // 初始 tcp_server, 可以指定线程数目, 这里线程是 4, 说明是一个 acceptor 线程, 4 个 I/O 组
55 //tcp_server 自己带一个 event_loop
56 struct TCPserver *tcpServer = tcp_server_init(eventLoop, acceptor, onConnectionComp:
57                                             onWriteCompleted, onConnectionClosed,
58 tcp_server_start(tcpServer);
59
60 // main thread for acceptor
61 event_loop_run(eventLoop);
62 }

```

关于这个程序，之前一直没有讲到的部分是缓冲区对象 `buffer`。这其实也是网络编程框架应该考虑的部分。

我们希望框架可以对应用程序封装掉套接字读和写的部分，转而提供的是针对缓冲区对象的读和写操作。这样一来，从套接字收取数据、处理异常、发送数据等操作都被类似 `buffer` 这样的对象所封装和屏蔽，应用程序所要做的事情就会变得更加简单，从 `buffer` 对象中可以获取已接收到的字节流再进行应用层处理，比如这里通过调用 `buffer_read_char` 函数从 `buffer` 中读取一个字节。

另外一方面，框架也必须对应用程序提供套接字发送的接口，接口的数据类型类似这里的 `buffer` 对象，可以看到，这里先生成了一个 `buffer` 对象，之后将编码后的结果填充到 `buffer` 对象里，最后调用 `tcp_connection_send_buffer` 将 `buffer` 对象里的数据通过套接字发送出去。

这里像 `onMessage`、`onConnectionClosed` 几个回调函数都是运行在子反应堆线程中的，也就是说，刚刚提到的生成 `buffer` 对象，`encode` 部分的代码，是在子反应堆线程中执行的。这其实也是回调函数的内涵，回调函数本身只是提供了类似 `Handler` 的处理逻辑，具体执行是由事件分发线程，或者说是 `event loop` 线程发起的。


框架通过一层抽象，让应用程序的开发者只需要看到回调函数，回调函数中的对象，也都是如 `buffer` 和 `tcp_connection` 这样封装过的对象，这样像套接字、字节流等底层实现的细节就完全由框架来完成了。



框架帮我们做了很多事情，那这些事情是如何做到的？在第四篇实战篇，我们将一一揭开答案。如果你有兴趣，不妨先看看实现代码。


## 样例程序结果

启动服务器，可以从屏幕输出上看到，使用的是 `epoll` 作为事件分发器。

 复制代码

```
1 $./epoll-server-multithreads
2 [msg] set epoll as dispatcher, main thread
3 [msg] add channel fd == 5, main thread
4 [msg] set epoll as dispatcher, Thread-1
5 [msg] add channel fd == 9, Thread-1
6 [msg] event loop thread init and signal, Thread-1
7 [msg] event loop run, Thread-1
8 [msg] event loop thread started, Thread-1
9 [msg] set epoll as dispatcher, Thread-2
10 [msg] add channel fd == 12, Thread-2
11 [msg] event loop thread init and signal, Thread-2
12 [msg] event loop run, Thread-2
13 [msg] event loop thread started, Thread-2
14 [msg] set epoll as dispatcher, Thread-3
15 [msg] add channel fd == 15, Thread-3
16 [msg] event loop thread init and signal, Thread-3
17 [msg] event loop run, Thread-3
18 [msg] event loop thread started, Thread-3
19 [msg] set epoll as dispatcher, Thread-4
20 [msg] add channel fd == 18, Thread-4
21 [msg] event loop thread init and signal, Thread-4
22 [msg] event loop run, Thread-4
23 [msg] event loop thread started, Thread-4
24 [msg] add channel fd == 6, main thread
25 [msg] event loop run, main thread
```


开启多个 `telnet` 客户端，连接上该服务器，通过屏幕输入和服务器端交互。

 复制代码

```
1 $telnet 127.0.0.1 43211
2 Trying 127.0.0.1...
3 Connected to 127.0.0.1.
4 Escape character is '^]'.
5 fafaf
6 snsns
7 ^]
```

```
8
9
10 telnet> quit
11 Connection closed.
```

服务端显示不断地从 `epoll_wait` 中返回处理 I/O 事件。

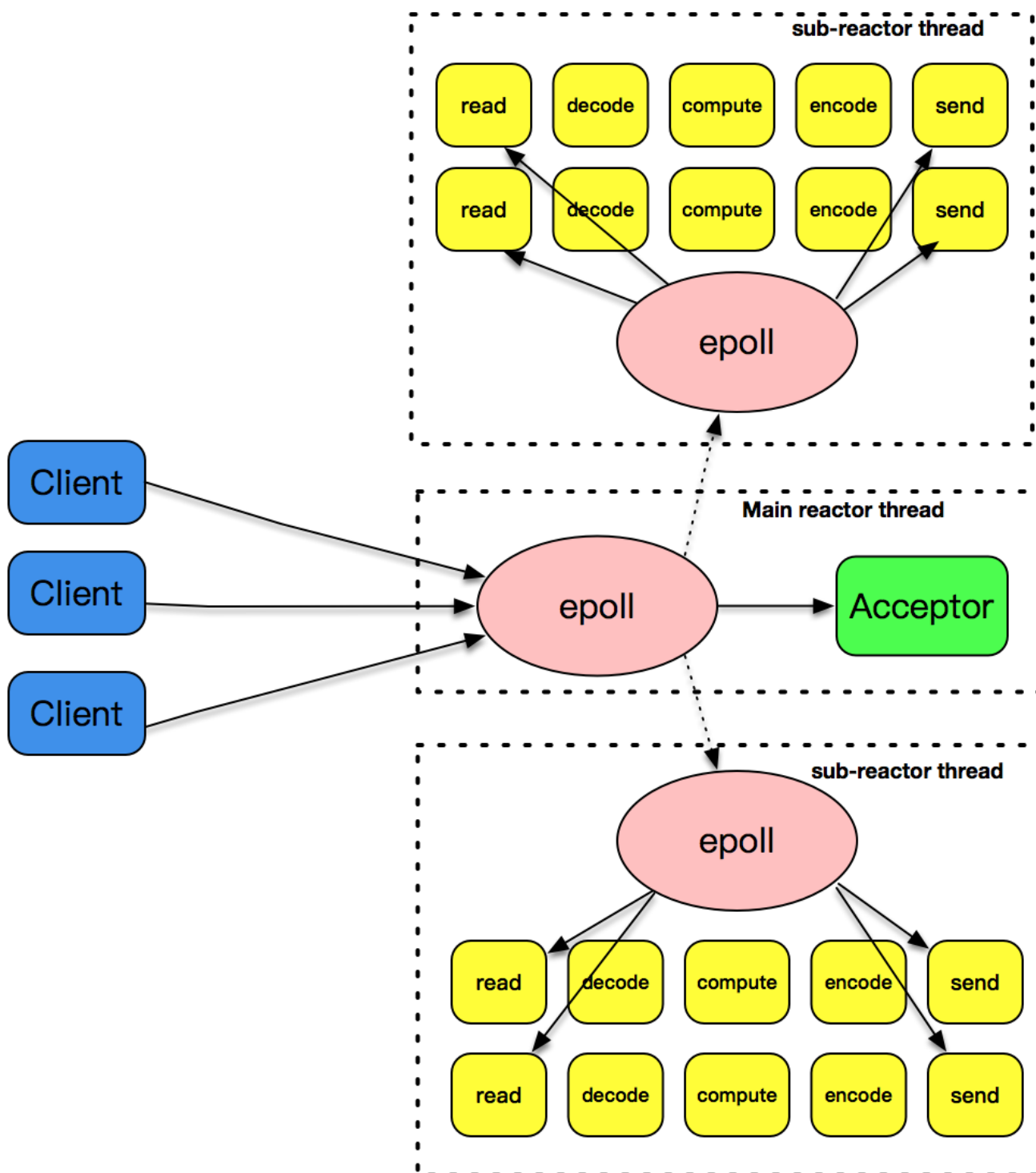
 复制代码

```
1 [msg] epoll_wait wakeup, main thread
2 [msg] get message channel fd==6 for read, main thread
3 [msg] activate channel fd == 6, revents=2, main thread
4 [msg] new connection established, socket == 19
5 connection completed
6 [msg] epoll_wait wakeup, Thread-1
7 [msg] get message channel fd==9 for read, Thread-1
8 [msg] activate channel fd == 9, revents=2, Thread-1
9 [msg] wakeup, Thread-1
10 [msg] add channel fd == 19, Thread-1
11 [msg] epoll_wait wakeup, Thread-1
12 [msg] get message channel fd==19 for read, Thread-1
13 [msg] activate channel fd == 19, revents=2, Thread-1
14 get message from tcp connection connection-19
15 afasf
16 [msg] epoll_wait wakeup, main thread
17 [msg] get message channel fd==6 for read, main thread
18 [msg] activate channel fd == 6, revents=2, main thread
19 [msg] new connection established, socket == 20
20 connection completed
21 [msg] epoll_wait wakeup, Thread-2
22 [msg] get message channel fd==12 for read, Thread-2
23 [msg] activate channel fd == 12, revents=2, Thread-2
24 [msg] wakeup, Thread-2
25 [msg] add channel fd == 20, Thread-2
26 [msg] epoll_wait wakeup, Thread-2
27 [msg] get message channel fd==20 for read, Thread-2
28 [msg] activate channel fd == 20, revents=2, Thread-2
29 get message from tcp connection connection-20
30 asfasfas
31 [msg] epoll_wait wakeup, Thread-2
32 [msg] get message channel fd==20 for read, Thread-2
33 [msg] activate channel fd == 20, revents=2, Thread-2
34 connection closed
35 [msg] epoll_wait wakeup, main thread
36 [msg] get message channel fd==6 for read, main thread
37 [msg] activate channel fd == 6, revents=2, main thread
38 [msg] new connection established, socket == 21
39 connection completed
```

```
40 [msg] epoll_wait wakeup, Thread-3
41 [msg] get message channel fd==15 for read, Thread-3
42 [msg] activate channel fd == 15, revents=2, Thread-3
43 [msg] wakeup, Thread-3
44 [msg] add channel fd == 21, Thread-3
45 [msg] epoll_wait wakeup, Thread-3
46 [msg] get message channel fd==21 for read, Thread-3
47 [msg] activate channel fd == 21, revents=2, Thread-3
48 get message from tcp connection connection-21
49 dfasfadsf
50 [msg] epoll_wait wakeup, Thread-1
51 [msg] get message channel fd==19 for read, Thread-1
52 [msg] activate channel fd == 19, revents=2, Thread-1
53 connection closed
54 [msg] epoll_wait wakeup, main thread
55 [msg] get message channel fd==6 for read, main thread
56 [msg] activate channel fd == 6, revents=2, main thread
57 [msg] new connection established, socket == 22
58 connection completed
59 [msg] epoll_wait wakeup, Thread-4
60 [msg] get message channel fd==18 for read, Thread-4
61 [msg] activate channel fd == 18, revents=2, Thread-4
62 [msg] wakeup, Thread-4
63 [msg] add channel fd == 22, Thread-4
64 [msg] epoll_wait wakeup, Thread-4
65 [msg] get message channel fd==22 for read, Thread-4
66 [msg] activate channel fd == 22, revents=2, Thread-4
67 get message from tcp connection connection-22
68 fafaf
69 [msg] epoll_wait wakeup, Thread-4
70 [msg] get message channel fd==22 for read, Thread-4
71 [msg] activate channel fd == 22, revents=2, Thread-4
72 connection closed
73 [msg] epoll_wait wakeup, Thread-3
74 [msg] get message channel fd==21 for read, Thread-3
75 [msg] activate channel fd == 21, revents=2, Thread-3
76 connection closed
```

其中主线程的 `epoll_wait` 只处理 `acceptor` 套接字的事件，表示的是连接的建立；反应堆子线程的 `epoll_wait` 主要处理的是已连接套接字的读写事件。文稿中的这幅图详细解释了这部分逻辑。





## epoll 的性能分析

epoll 的性能凭什么就要比 poll 或者 select 好呢？这要从两个角度来说明。

第一个角度是事件集合。在每次使用 poll 或 select 之前，都需要准备一个感兴趣的事件集合，系统内核拿到事件集合，进行分析并在内核空间构建相应的数据结构来完成对事件集合的注册。而 epoll 则不是这样，epoll 维护了一个全局的事件集合，通过 epoll 句柄，可以操纵这个事件集合，增加、删除或修改这个事件集合里的某个元素。要知道在绝大多数情况

下，事件集合的变化没有那么大，这样操纵系统内核就不需要每次重新扫描事件集合，构建内核空间数据结构。

第二个角度是就绪列表。每次在使用 poll 或者 select 之后，应用程序都需要扫描整个感兴趣的事件集合，从中找出真正活动的事件，这个列表如果增长到 10K 以上，每次扫描的时间损耗也是惊人的。事实上，很多情况下扫描完一圈，可能发现只有几个真正活动的事件。而 epoll 则不是这样，epoll 返回的直接就是活动的事件列表，应用程序减少了大量的扫描时间。

此外，epoll 还提供了更高级的能力——边缘触发。[第 23 讲](#)通过一个直观的例子，讲解了边缘触发和条件触发的区别。

这里再举一个例子说明一下。

如果某个套接字有 100 个字节可以读，边缘触发和条件触发都会产生 read ready notification 事件，如果应用程序只读取了 50 个字节，边缘触发就会陷入等待；而条件触发则会因为还有 50 个字节没有读取完，不断地产生 read ready notification 事件。

在边缘触发下，如果某个套接字缓冲区可以写，会无限次返回 write ready notification 事件，在这种情况下，如果应用程序没有准备好，不需要发送数据，一定需要解除套接字上的 ready notification 事件，否则 CPU 就直接跪了。

我们简单地总结一下，边缘触发只会产生一次活动事件，性能和效率更高。不过，程序处理起来要更为小心。

## 总结

本讲我们将程序框架切换到了 epoll 的版本，和 poll 版本相比，只是底层的框架做了更改，上层应用程序不用做任何修改，这也是程序框架强大的地方。和 poll 相比，epoll 从事件集合和就绪列表两个方面加强了程序性能，是 Linux 下高性能网络程序的首选。

## 思考题

最后我给大家给大家布置两道思考题：

第一道，说说你对边缘触发和条件触发的理解。

第二道，对于边缘触发和条件触发，onMessage 函数处理要注意什么？

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。

 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。

上一篇 28 | I/O多路复用进阶：子线程使用poll处理连接I/O事件

## 精选留言 (1)

 写留言



JJ

2019-10-14

边缘条件，当套接字缓冲区可写，会不断触发ready notification事件，不是应该条件触发才是这样吗？



1