13 多路转接 epoll

I/O 多路转接之 poll [选学]

poll 函数接口

```
C
#include <poll.h>

int poll(struct pollfd *fds, nfds_t nfds, int timeout);

// pollfd 结构
struct pollfd {
   int   fd;         /* file descriptor */
   short events;     /* requested events */
   short revents;    /* returned events */
};
```

参数说明

- fds 是一个 poll 函数监听的结构列表. 每一个元素中, 包含了三部分内容: 文件描述符, 监听的事件集合, 返回的事件集合.
- nfds 表示 fds 数组的长度.
- timeout 表示 poll 函数的超时时间, 单位是毫秒(ms).

events 和 revents 的取值:

事　件	描　述	是否可作为输入	是否可作为输出
POLLIN	数据（包括普通数据和优先数据）可读	是	是
POLLRDNORM	普通数据可读	是	是
POLLRDBAND	优先级带数据可读（Linux 不支持）	是	是
POLLPRI	高优先级数据可读，比如 TCP 带外数据	是	是
POLLOUT	数据（包括普通数据和优先数据）可写	是	是

POLLWRNORM	普通数据可写	是	是
POLLWRBAND	优先级带数据可写	是	是
POLLRDHUP	TCP 连接被对方关闭，或者对方关闭了写操作。它由 GNU 引入	是	是
POLLERR	错误	否	是
POLLHUP	挂起。比如管道的写端被关闭后，读端描述符上将收到 POLLHUP 事件	否	是
POLLNVAL	文件描述符没有打开	否	是

返回结果

- 返回值小于 0, 表示出错;
- 返回值等于 0, 表示 poll 函数等待超时;
- 返回值大于 0, 表示 poll 由于监听的文件描述符就绪而返回.

socket 就绪条件

同 select

poll 的优点

不同于 select 使用三个位图来表示三个 fdset 的方式，poll 使用一个 pollfd 的指针实现.

- pollfd 结构包含了要监视的 event 和发生的 event，不再使用 select“参数-值”传递的方式. 接口使用比 select 更方便.
- poll 并没有最大数量限制 (但是数量过大后性能也是会下降).

poll 的缺点

poll 中监听的文件描述符数目增多时

- 和 select 函数一样，poll 返回后，需要轮询 pollfd 来获取就绪的描述符.
- 每次调用 poll 都需要把大量的 pollfd 结构从用户态拷贝到内核中.
- 同时连接的大量客户端在一时刻可能只有很少的处于就绪状态, 因此随着监视的描述符数量的增长, 其效率也会线性下降.

poll 示例: 使用 poll 监控标准输入

```
C
#include <poll.h>
```

```
#include <unistd.h>
#include <stdio.h>

int main() {
  struct pollfd poll_fd;
  poll_fd.fd = 0;
  poll_fd.events = POLLIN;

  for (;;) {
    int ret = poll(&poll_fd, 1, 1000);
    if (ret < 0) {
      perror("poll");
      continue;
    }
    if (ret == 0) {
      printf("poll timeout\n");
      continue;
    }
    if (poll_fd.revents == POLLIN) {
      char buf[1024] = {0};
      read(0, buf, sizeof(buf) - 1);
      printf("stdin:%s", buf);
    }
  }
}
```

I/O 多路转接之 epoll

epoll 初识

按照 man 手册的说法: **是为处理大批量句柄而作了改进的 poll**.

它是在 2.5.44 内核中被引进的(epoll(4) is a new API introduced in Linux kernel 2.5.44)

它几乎具备了之前所说的一切优点，被公认为 Linux2.6 下性能最好的多路 I/O 就绪通知方法.

epoll 的相关系统调用

epoll 有 3 个相关的系统调用.

epoll_create

```
C
```

```
int epoll_create(int size);
```

创建一个 epoll 的句柄.

- 自从 linux2.6.8 之后，size 参数是被忽略的.
- 用完之后, 必须调用 close()关闭.

epoll_ctl

```
C
int epoll_ctl(int epfd, int op, int fd, struct epoll_event
*event);
```

epoll 的事件注册函数.

- 它不同于 select()是在监听事件时告诉内核要监听什么类型的事件, 而是在这里先注册要监听的事件类型.
- 第一个参数是 epoll_create()的返回值(epoll 的句柄).
- 第二个参数表示动作，用三个宏来表示.
- 第三个参数是需要监听的 fd.
- 第四个参数是告诉内核需要监听什么事.

第二个参数的取值:

- `EPOLL_CTL_ADD`：注册新的 fd 到 epfd 中;
- `EPOLL_CTL_MOD`：修改已经注册的 fd 的监听事件;
- `EPOLL_CTL_DEL`：从 epfd 中删除一个 fd;

struct epoll_event 结构如下:

```
typedef union epoll_data
{
  void *ptr;
  int fd;
  uint32_t u32;
  uint64_t u64;
} epoll_data_t;

struct epoll_event
{
  uint32_t events;  /* Epoll events */
  epoll_data_t data;  /* User data variable */
} __EPOLL_PACKED;
```

events 可以是以下几个宏的集合:

- EPOLLIN ：表示对应的文件描述符可以读 (包括对端 SOCKET 正常关闭);
- EPOLLOUT : 表示对应的文件描述符可以写;
- EPOLLPRI : 表示对应的文件描述符有紧急的数据可读 (这里应该表示有带外数据到来);
- EPOLLERR : 表示对应的文件描述符发生错误;
- EPOLLHUP : 表示对应的文件描述符被挂断;
- EPOLLET ：将 EPOLL 设为边缘触发(Edge Triggered)模式, 这是相对于水平触发(Level Triggered)来说的.
- EPOLLONESHOT：只监听一次事件, 当监听完这次事件之后, 如果还需要继续监听这个 socket 的话, 需要再次把这个 socket 加入到 EPOLL 队列里.

epoll_wait

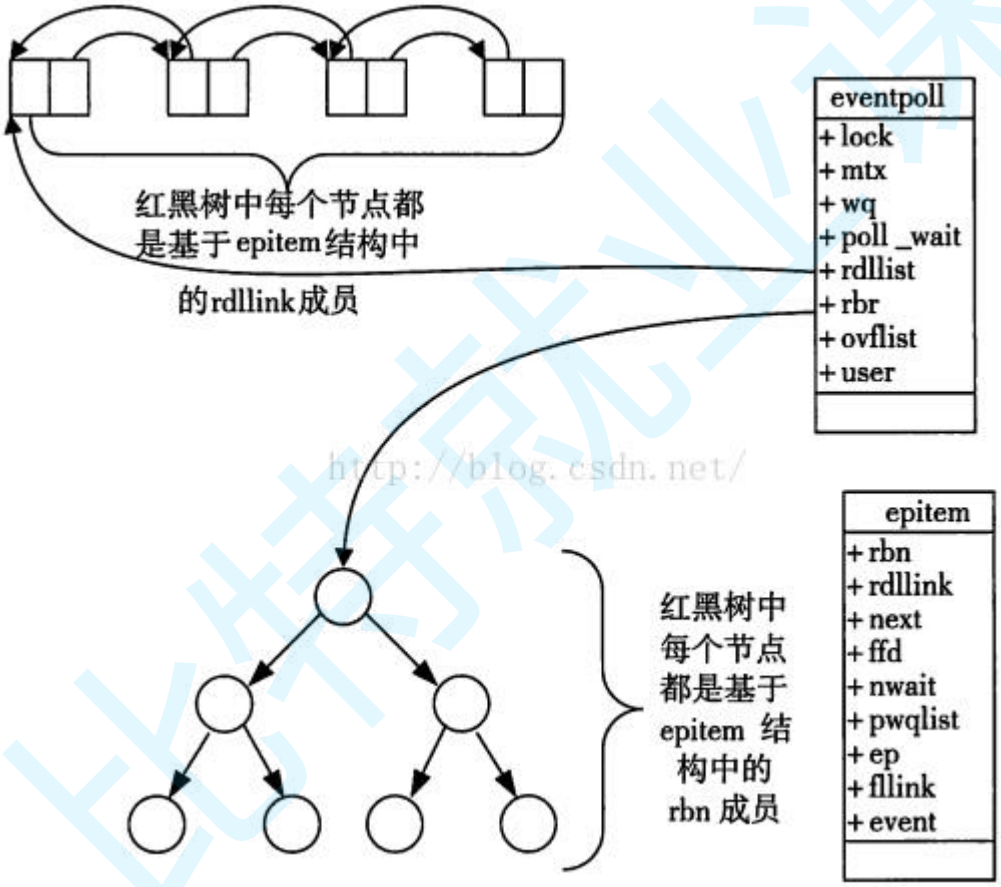
```
C
int epoll_wait(int epfd, struct epoll_event * events, int
maxevents, int timeout);
```

收集在 epoll 监控的事件中已经发送的事件.

- 参数 events 是分配好的 epoll_event 结构体数组.

- epoll 将会把发生的事件赋值到 events 数组中 (events 不可以是空指针，内核只负责把数据复制到这个 events 数组中，不会去帮助我们在用户态中分配内存).
- maxevents 告之内核这个 events 有多大，这个 maxevents 的值不能大于创建 epoll_create()时的 size.
- 参数 timeout 是超时时间 (毫秒，0 会立即返回，-1 是永久阻塞).
- 如果函数调用成功，返回对应 I/O 上已准备好的文件描述符数目，如返回 0 表示已超时, 返回小于 0 表示函数失败.

epoll 工作原理



- 当某一进程调用 epoll_create 方法时，Linux 内核会创建一个 eventpoll 结构体，这个结构体中有两个成员与 epoll 的使用方式密切相关.

```
C
struct eventpoll{
    ....
    /*红黑树的根节点，这颗树中存储着所有添加到 epoll 中的需要监控的事件
```

```
*/
    struct rb_root  rbr;
    /*双链表中则存放着将要通过 epoll_wait 返回给用户的满足条件的事件*/
    struct list_head rdlist;
    ....
};
```

- 每一个 epoll 对象都有一个独立的 eventpoll 结构体，用于存放通过 epoll_ctl 方法向 epoll 对象中添加进来的事件.
- 这些事件都会挂载在红黑树中，如此，重复添加的事件就可以通过红黑树而高效的识别出来(红黑树的插入时间效率是 lgn，其中 n 为树的高度).
- 而所有添加到 epoll 中的事件都会与设备(网卡)驱动程序建立回调关系，也就是说，当响应的事件发生时会调用这个回调方法.
- 这个回调方法在内核中叫 ep_poll_callback,它会将发生的事件添加到 rdlist 双链表中.
- 在 epoll 中，对于每一个事件，都会建立一个 epitem 结构体.

```
C
struct epitem{
    struct rb_node  rbn;//红黑树节点
    struct list_head    rdllink;//双向链表节点
    struct epoll_filefd  ffd;  //事件句柄信息
    struct eventpoll *ep;    //指向其所属的 eventpoll 对象
    struct epoll_event event; //期待发生的事件类型
}
```

- 当调用 epoll_wait 检查是否有事件发生时，只需要检查 eventpoll 对象中的 rdlist 双链表中是否有 epitem 元素即可.
- 如果 rdlist 不为空，则把发生的事件复制到用户态，同时将事件数量返回给用户. 这个操作的时间复杂度是 O(1).

总结一下, epoll 的使用过程就是三部曲:

- 调用 epoll_create 创建一个 epoll 句柄;
- 调用 epoll_ctl, 将要监控的文件描述符进行注册;

- 调用 epoll_wait, 等待文件描述符就绪;

epoll 的优点(和 select 的缺点对应)

- 接口使用方便: 虽然拆分成了三个函数, 但是反而使用起来更方便高效. 不需要每次循环都设置关注的文件描述符, 也做到了输入输出参数分离开
- 数据拷贝轻量: 只在合适的时候调用 `EPOLL_CTL_ADD` 将文件描述符结构拷贝到内核中, 这个操作并不频繁(而 select/poll 都是每次循环都要进行拷贝)
- 事件回调机制: 避免使用遍历, 而是使用回调函数的方式, 将就绪的文件描述符结构加入到就绪队列中, epoll_wait 返回直接访问就绪队列就知道哪些文件描述符就绪. 这个操作时间复杂度 O(1). 即使文件描述符数目很多, 效率也不会受到影响.
- 没有数量限制: 文件描述符数目无上限.

注意!!

网上有些博客说, epoll 中使用了内存映射机制

- 内存映射机制: 内核直接将就绪队列通过 mmap 的方式映射到用户态. 避免了拷贝内存这样的额外性能开销.

这种说法是不准确的. 我们定义的 struct epoll_event 是我们在用户空间中分配好的内存. 势必还是需要将内核的数据拷贝到这个用户空间的内存中的.

请同学们对比总结 select, poll, epoll 之间的优点和缺点(重要, 面试中常见).

epoll 工作方式

你妈喊你吃饭的例子

```
C
你正在吃鸡，眼看进入了决赛圈，你妈饭做好了，喊你吃饭的时候有两种方式：
1．如果你妈喊你一次，你没动，那么你妈会继续喊你第二次，第三次...(亲妈，水平触发)
2．如果你妈喊你一次，你没动，你妈就不管你了(后妈，边缘触发)
```

epoll 有 2 种工作方式-水平触发(LT)和边缘触发(ET)

假如有这样一个例子:

- 我们已经把一个 tcp socket 添加到 epoll 描述符

- 这个时候 socket 的另一端被写入了 2KB 的数据
- 调用 epoll_wait，并且它会返回. 说明它已经准备好读取操作
- 然后调用 read, 只读取了 1KB 的数据
- 继续调用 epoll_wait.....

水平触发 Level Triggered 工作模式

epoll 默认状态下就是 LT 工作模式.

- 当 epoll 检测到 socket 上事件就绪的时候, 可以不立刻进行处理. 或者只处理一部分.
- 如上面的例子, 由于只读了 1K 数据, 缓冲区中还剩 1K 数据, 在第二次调用 `epoll_wait` 时, `epoll_wait` 仍然会立刻返回并通知 socket 读事件就绪.
- 直到缓冲区上所有的数据都被处理完, `epoll_wait` 才不会立刻返回.
- 支持阻塞读写和非阻塞读写

边缘触发 Edge Triggered 工作模式

如果我们在第 1 步将 socket 添加到 epoll 描述符的时候使用了 EPOLLET 标志, epoll 进入 ET 工作模式.

- 当 epoll 检测到 socket 上事件就绪时, 必须立刻处理.
- 如上面的例子, 虽然只读了 1K 的数据, 缓冲区还剩 1K 的数据, 在第二次调用 `epoll_wait` 的时候, `epoll_wait` 不会再返回了.
- 也就是说, ET 模式下, 文件描述符上的事件就绪后, 只有一次处理机会.
- ET 的性能比 LT 性能更高(`epoll_wait` 返回的次数少了很多). Nginx 默认采用 ET 模式使用 epoll.
- 只支持非阻塞的读写

select 和 poll 其实也是工作在 LT 模式下. epoll 既可以支持 LT, 也可以支持 ET.

对比 LT 和 ET

LT 是 epoll 的默认行为.

使用 ET 能够减少 epoll 触发的次数. 但是代价就是强逼着程序猿一次响应就绪过程中就把所有的数据都处理完.

相当于一个文件描述符就绪之后, 不会反复被提示就绪, 看起来就比 LT 更高效一些. 但

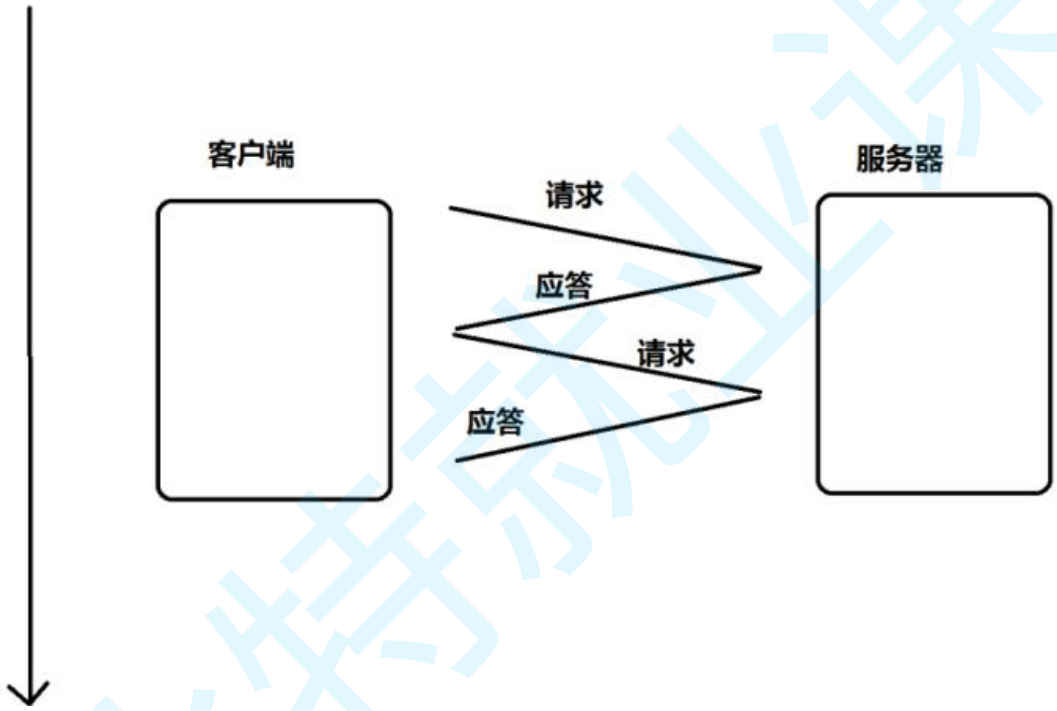
是在 LT 情况下如果也能做到每次就绪的文件描述符都立刻处理, 不让这个就绪被重复提示的话, 其实性能也是一样的.

另一方面, ET 的代码复杂程度更高了.

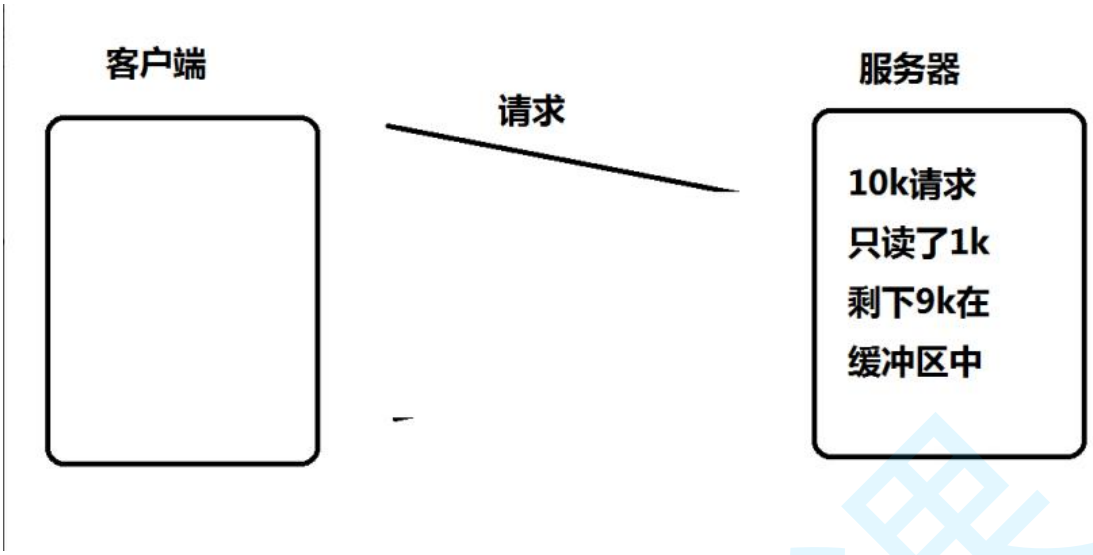
理解 ET 模式和非阻塞文件描述符

使用 ET 模式的 epoll, 需要将文件描述设置为非阻塞. 这个不是接口上的要求, 而是 "工程实践" 上的要求.

假设这样的场景: 服务器接收到一个 10k 的请求, 会向客户端返回一个应答数据. 如果客户端收不到应答, 不会发送第二个 10k 请求.



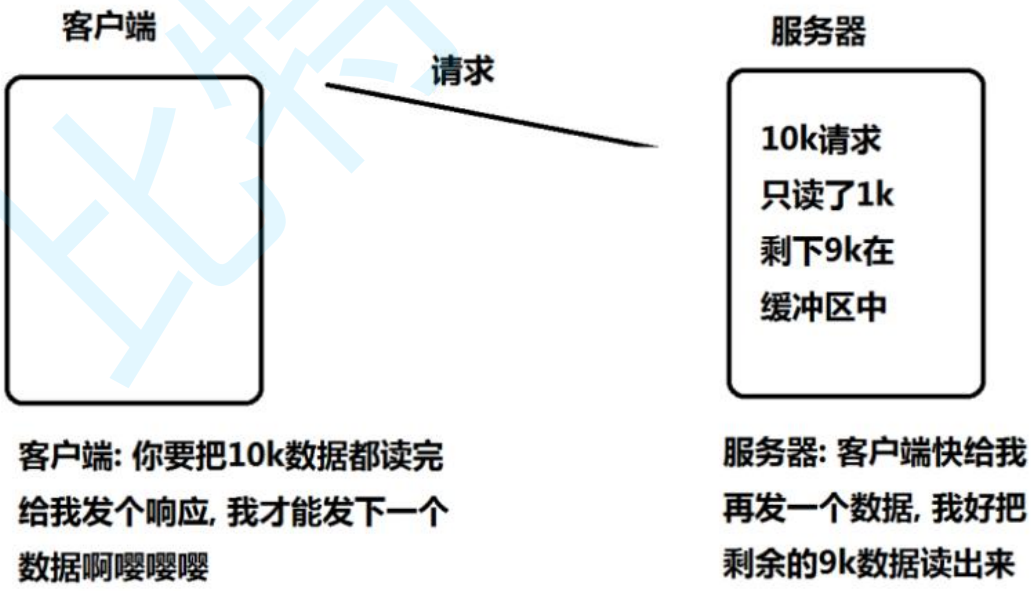
如果服务端写的代码是阻塞式的 read, 并且一次只 read 1k 数据的话(read 不能保证一次就把所有的数据都读出来, 参考 man 手册的说明, 可能被信号打断), 剩下的 9k 数据就会待在缓冲区中.



此时由于 epoll 是 ET 模式, 并不会认为文件描述符读就绪. `epoll_wait` 就不会再次返回. 剩下的 9k 数据会一直在缓冲区中. 直到下一次客户端再给服务器写数据. `epoll_wait` 才能返回

但是问题来了.

- 服务器只读到 1k 个数据, 要 10k 读完才会给客户端返回响应数据.
- 客户端要读到服务器的响应, 才会发送下一个请求
- 客户端发送了下一个请求, `epoll_wait` 才会返回, 才能去读缓冲区中剩余的数据.



所以, 为了解决上述问题(阻塞 read 不一定能一下把完整的请求读完), 于是就可以使用非阻塞轮训的方式来读缓冲区, 保证一定能把完整的请求都读出来.

而如果是 LT 没这个问题. 只要缓冲区中的数据没读完, 就能够让 `epoll_wait` 返回文件描述符读就绪.

epoll 的使用场景

epoll 的高性能, 是有一定的特定场景的. 如果场景选择的不适宜, epoll 的性能可能适得其反.

> - 对于多连接, 且多连接中只有一部分连接比较活跃时, 比较适合使用 epoll.

例如, 典型的一个需要处理上万个客户端的服务器, 例如各种互联网 APP 的入口服务器, 这样的服务器就很适合 epoll.

如果只是系统内部, 服务器和服务器之间进行通信, 只有少数的几个连接, 这种情况下用 epoll 就并不合适. 具体要根据需求和场景特点来决定使用哪种 IO 模型.

epoll 中的惊群问题(选学)

惊群问题有些面试官可能会问到. 建议同学们课后自己查阅资料了解一下问题的解决方案.

参考 http://blog.csdn.net/fsmiy/article/details/36873357

epoll 示例: epoll 服务器(LT 模式)

tcp_epoll_server.hpp

```
C
//////////////////////////////////////////////////////
// 封装一个 Epoll 服务器, 只考虑读就绪的情况
//////////////////////////////////////////////////////

#pragma once
#include <vector>
#include <functional>
#include <sys/epoll.h>
#include "tcp_socket.hpp"

typedef std::function<void (const std::string&, std::string*
resp)> Handler;

class Epoll {
public:
  Epoll() {
```

```
    epoll_fd_ = epoll_create(10);
  }

  ~Epoll() {
    close(epoll_fd_);
  }

  bool Add(const TcpSocket& sock) const {
    int fd = sock.GetFd();
    printf("[Epoll Add] fd = %d\n", fd);
    epoll_event ev;
    ev.data.fd = fd;
    ev.events = EPOLLIN;
    int ret = epoll_ctl(epoll_fd_, EPOLL_CTL_ADD, fd, &ev);
    if (ret < 0) {
      perror("epoll_ctl ADD");
      return false;
    }
    return true;
  }

  bool Del(const TcpSocket& sock) const {
    int fd = sock.GetFd();
    printf("[Epoll Del] fd = %d\n", fd);
    int ret = epoll_ctl(epoll_fd_, EPOLL_CTL_DEL, fd, NULL);
    if (ret < 0) {
      perror("epoll_ctl DEL");
      return false;
    }
    return true;
  }

  bool Wait(std::vector<TcpSocket> *output) const {
    output->clear();
    epoll_event events[1000];
    int nfds = epoll_wait(epoll_fd_, events, sizeof(events) /
sizeof(events[0]), -1);
    if (nfds < 0) {
      perror("epoll_wait");
      return false;
    }
    // [注意!] 此处必须是循环到 nfds, 不能多循环
    for (int i = 0; i < nfds; ++i) {
      TcpSocket sock(events[i].data.fd);
```

```
      output->push_back(sock);
    }
    return true;
  }

private:
  int epoll_fd_;
};

class TcpEpollServer {
public:
  TcpEpollServer(const std::string& ip, uint16_t port) : ip_(ip),
port_(port) {

  }

  bool Start(Handler handler) {
    // 1. 创建 socket
    TcpSocket listen_sock;
    CHECK_RET(listen_sock.Socket());
    // 2. 绑定
    CHECK_RET(listen_sock.Bind(ip_, port_));
    // 3. 监听
    CHECK_RET(listen_sock.Listen(5));
    // 4. 创建 Epoll 对象, 并将 listen_sock 加入进去
    Epoll epoll;
    epoll.Add(listen_sock);
    // 5. 进入事件循环
    for (;;) {
      // 6. 进行 epoll_wait
      std::vector<TcpSocket> output;
      if (!epoll.Wait(&output)) {
        continue;
      }
      // 7. 根据就绪的文件描述符的种类决定如何处理
      for (size_t i = 0; i < output.size(); ++i) {
        if (output[i].GetFd() == listen_sock.GetFd()) {
          // 如果是 listen_sock, 就调用 accept
          TcpSocket new_sock;
          listen_sock.Accept(&new_sock);
          epoll.Add(new_sock);
        } else {
          // 如果是 new_sock, 就进行一次读写
```

```
          std::string req, resp;
          bool ret = output[i].Recv(&req);
          if (!ret) {
            // [注意!!] 需要把不用的 socket 关闭
            // 先后顺序别搞反. 不过在 epoll 删除的时候其实就已经关闭
socket 了
            epoll.Del(output[i]);
            output[i].Close();
            continue;
          }
          handler(req, &resp);
          output[i].Send(resp);
        }  // end for
      }  // end for (;;)
    }
    return true;
  }

private:
  std::string ip_;
  uint16_t port_;
};
```

dict_server.cc 只需要将 server 对象的类型改成 TcpEpollServer 即可.

epoll 示例: epoll 服务器(ET 模式)

基于 LT 版本稍加修改即可

1. 修改 tcp_socket.hpp, 新增非阻塞读和非阻塞写接口
2. 对于 accept 返回的 new_sock 加上 EPOLLET 这样的选项

注意: 此代码暂时未考虑 listen_sock ET 的情况. 如果将 listen_sock 设为 ET, 则需要非阻塞轮询的方式 accept. 否则会导致同一时刻大量的客户端同时连接的时候, 只能 accept 一次的问题.

tcp_socket.hpp

```
C
  // 以下代码添加在 TcpSocket 类中
  // 非阻塞 IO 接口
  bool SetNoBlock() {
    int fl = fcntl(fd_, F_GETFL);
    if (fl < 0) {
```

```
      perror("fcntl F_GETFL");
      return false;
    }
    int ret = fcntl(fd_, F_SETFL, fl | O_NONBLOCK);
    if (ret < 0) {
      perror("fcntl F_SETFL");
      return false;
    }
    return true;
  }

  bool RecvNoBlock(std::string* buf) const {
    // 对于非阻塞 IO 读数据，如果 TCP 接受缓冲区为空，就会返回错误
    // 错误码为 EAGAIN 或者 EWOULDBLOCK，这种情况也是意料之中，需要重
试
    // 如果当前读到的数据长度小于尝试读的缓冲区的长度，就退出循环
    // 这种写法其实不算特别严谨(没有考虑粘包问题)
    buf->clear();
    char tmp[1024 * 10] = {0};
    for (;;) {
      ssize_t read_size = recv(fd_, tmp, sizeof(tmp) - 1, 0);
      if (read_size < 0) {
        if (errno == EWOULDBLOCK || errno == EAGAIN) {
          continue;
        }
        perror("recv");
        return false;
      }
      if (read_size == 0) {
        // 对端关闭，返回 false
        return false;
      }
      tmp[read_size] = '\0';
      *buf += tmp;
      if (read_size < (ssize_t)sizeof(tmp) - 1) {
        break;
      }
    }
    return true;
  }

  bool SendNoBlock(const std::string& buf) const {
    // 对于非阻塞 IO 的写入，如果 TCP 的发送缓冲区已经满了，就会出现出
```

```
错的情况
    // 此时的错误号是 EAGAIN 或者 EWOULDBLOCK. 这种情况下不应放弃治疗
    // 而要进行重试
    ssize_t cur_pos = 0;  // 记录当前写到的位置
    ssize_t left_size = buf.size();
    for (;;) {
      ssize_t write_size = send(fd_, buf.data() + cur_pos,
left_size, 0);
      if (write_size < 0) {
        if (errno == EAGAIN || errno == EWOULDBLOCK) {
          // 重试写入
          continue;
        }
        return false;
      }
      cur_pos += write_size;
      left_size -= write_size;
      // 这个条件说明写完需要的数据了
      if (left_size <= 0) {
        break;
      }
    }
    return true;
  }
```

tcp_epoll_server.hpp

```
C
//////////////////////////////////////////////////
// 封装一个 Epoll ET 服务器
// 修改点:
// 1. 对于 new sock, 加上 EPOLLET 标记
// 2. 修改 TcpSocket 支持非阻塞读写
// [注意!] listen_sock 如果设置成 ET, 就需要非阻塞调用 accept 了
// 稍微麻烦一点, 此处暂时不实现
//////////////////////////////////////////////////

#pragma once
#include <vector>
#include <functional>
#include <sys/epoll.h>
#include "tcp_socket.hpp"
```

```
typedef std::function<void (const std::string&, std::string*
resp)> Handler;

class Epoll {
public:
  Epoll() {
    epoll_fd_ = epoll_create(10);
  }

  ~Epoll() {
    close(epoll_fd_);
  }

  bool Add(const TcpSocket& sock, bool epoll_et = false) const {
    int fd = sock.GetFd();
    printf("[Epoll Add] fd = %d\n", fd);
    epoll_event ev;
    ev.data.fd = fd;
    if (epoll_et) {
      ev.events = EPOLLIN | EPOLLET;
    } else {
      ev.events = EPOLLIN;
    }
    int ret = epoll_ctl(epoll_fd_, EPOLL_CTL_ADD, fd, &ev);
    if (ret < 0) {
      perror("epoll_ctl ADD");
      return false;
    }
    return true;
  }

  bool Del(const TcpSocket& sock) const {
    int fd = sock.GetFd();
    printf("[Epoll Del] fd = %d\n", fd);
    int ret = epoll_ctl(epoll_fd_, EPOLL_CTL_DEL, fd, NULL);
    if (ret < 0) {
      perror("epoll_ctl DEL");
      return false;
    }
    return true;
  }

  bool Wait(std::vector<TcpSocket> *output) const {
    output->clear();
```

```
    epoll_event events[1000];
    int nfds = epoll_wait(epoll_fd_, events, sizeof(events) /
sizeof(events[0]), -1);
    if (nfds < 0) {
      perror("epoll_wait");
      return false;
    }
    // [注意!] 此处必须是循环到 nfds, 不能多循环
    for (int i = 0; i < nfds; ++i) {
      TcpSocket sock(events[i].data.fd);
      output->push_back(sock);
    }
    return true;
  }

private:
  int epoll_fd_;
};

class TcpEpollServer {
public:
  TcpEpollServer(const std::string& ip, uint16_t port) : ip_(ip),
port_(port) {

  }

  bool Start(Handler handler) {
    // 1. 创建 socket
    TcpSocket listen_sock;
    CHECK_RET(listen_sock.Socket());
    // 2. 绑定
    CHECK_RET(listen_sock.Bind(ip_, port_));
    // 3. 监听
    CHECK_RET(listen_sock.Listen(5));
    // 4. 创建 Epoll 对象, 并将 listen_sock 加入进去
    Epoll epoll;
    epoll.Add(listen_sock);
    // 5. 进入事件循环
    for (;;) {
      // 6. 进行 epoll_wait
      std::vector<TcpSocket> output;
      if (!epoll.Wait(&output)) {
        continue;
      }
```

```
      // 7. 根据就绪的文件描述符的种类决定如何处理
      for (size_t i = 0; i < output.size(); ++i) {
        if (output[i].GetFd() == listen_sock.GetFd()) {
          // 如果是 listen_sock, 就调用 accept
          TcpSocket new_sock;
          listen_sock.Accept(&new_sock);
          epoll.Add(new_sock, true);
        } else {
          // 如果是 new_sock, 就进行一次读写
          std::string req, resp;
          bool ret = output[i].RecvNoBlock(&req);
          if (!ret) {
            // [注意!!] 需要把不用的 socket 关闭
            // 先后顺序别搞反. 不过在 epoll 删除的时候其实就已经关闭
socket 了
            epoll.Del(output[i]);
            output[i].Close();
            continue;
          }
          handler(req, &resp);
          output[i].SendNoBlock(resp);
          printf("[client %d] req: %s, resp: %s\n",
output[i].GetFd(),
              req.c_str(), resp.c_str());
        }  // end for
      }  // end for (;;)
    }
    return true;
  }

private:
  std::string ip_;
  uint16_t port_;
};
```