Contents

一种基于 io_uring 和 C++20 协程的通用异步 IO 框架实现

本次课设所有代码均使用作者编写的通用异步 IO 框架 ZIO，其提供了现代风格的异步 IO 编程体验。遂本次报告开始先介绍本框架原理和实现。

OrbitZore/libzio: https://github.com/OrbitZore/libzio/tree/master

技术背景说明

io_uring

io_uring 是 2019 年进入 Linux 内核（5.0）的一个新高性能异步 IO 设施，被设计用来替代原有的 POSIX 异步函数（`aio_`系列），是继 C10k 之后新时代 C10m 问题的一个解决方案。

io_uring 设计了三个全新的系统调用（`io_uring_setup`，`io_uring_enter`和`io_uring_register`）。应用程序调用`io_uring_setup`在用户程序的内存空间里创建两个无锁循环队列（submit queue 和 completion queue），应用程序通过写入 submit queue 写 IO 操作；读 completion queue 读取 IO 操作返回值。内核读 submit queue 执行 IO 操作；写 completion queue 返回 IO 操作返回值。应用程序可通过调用`io_uring_enter`手动通知内核消费 submit queue/阻塞等待 completion queue/消费 completion queue。可以使用`io_uring_register`注册缓冲区供 io_uring 使用。

和 select/poll/epoll 这三种多路复用技术对比来说，io_uring 是真正的异步 IO；select 维护 fdset，poll 维护 fd 数组，epoll 维护红黑树。而 io_uring 仅仅维护两个无锁循环队列。简单而高效。io_uring 现在已经成为 LinuxIO 的一个常用设施，在 OceanBase 数据库大赛等高性能 IO 场合里经常能见到 io_uring 的身影。

io_uring 作者 Jens Axboe 将这三个系统调用封装成 C 库 liburing 并鼓励使用 liburing 而不是裸系统调用。本框架也将使用 liburing 编写。

C++20 协程

从根本上说，C++20 协程是函数对象上的语法糖。编译器将围绕您的协程函数生成一个代码框架。此代码框架依赖于用户定义的返回和承诺类型。

C++20 协程提供了三个一元从右到左结合的操作符`co_return`,`co_yield`和`co_await`。任何一个包含三个操作符的函数都被视为协程，插入进协程代码框架。

按传统协程分类方法来看 C++20 协程是无栈协程。之所以内置于 C++ 语言是为了能更好的让编译器放手进行优化，从而避免编译器陷入各种用于保存上下文的内联汇编中。和传统协程方案对比来说 C++20 协程具有协程额外占用内存极小，协程切换速度快的优势。

本框架将使用 C++20 协程结合 liburing 实现异步网络 IO/文件 IO 并在语言层实现类似 Python asyncio 库的语法。

源代码文件功能说明

```
├── uringtest.h
├── zio_ip.hpp
├── zio_buffer.hpp
├── zio_http.hpp
└── zio_types.hpp
```

uringtest 实现了整体异步框架。

zio_ip 提供了基于 socket 设施的 IP 网络支持。

zio_buffer 提供了基于缓冲区的读 IO。

zio_http 基于 zio_buffer 提供了简单 http 支持。

zio_types 提供了几个方便编程的 C++20 概念。

uringtest 功能说明

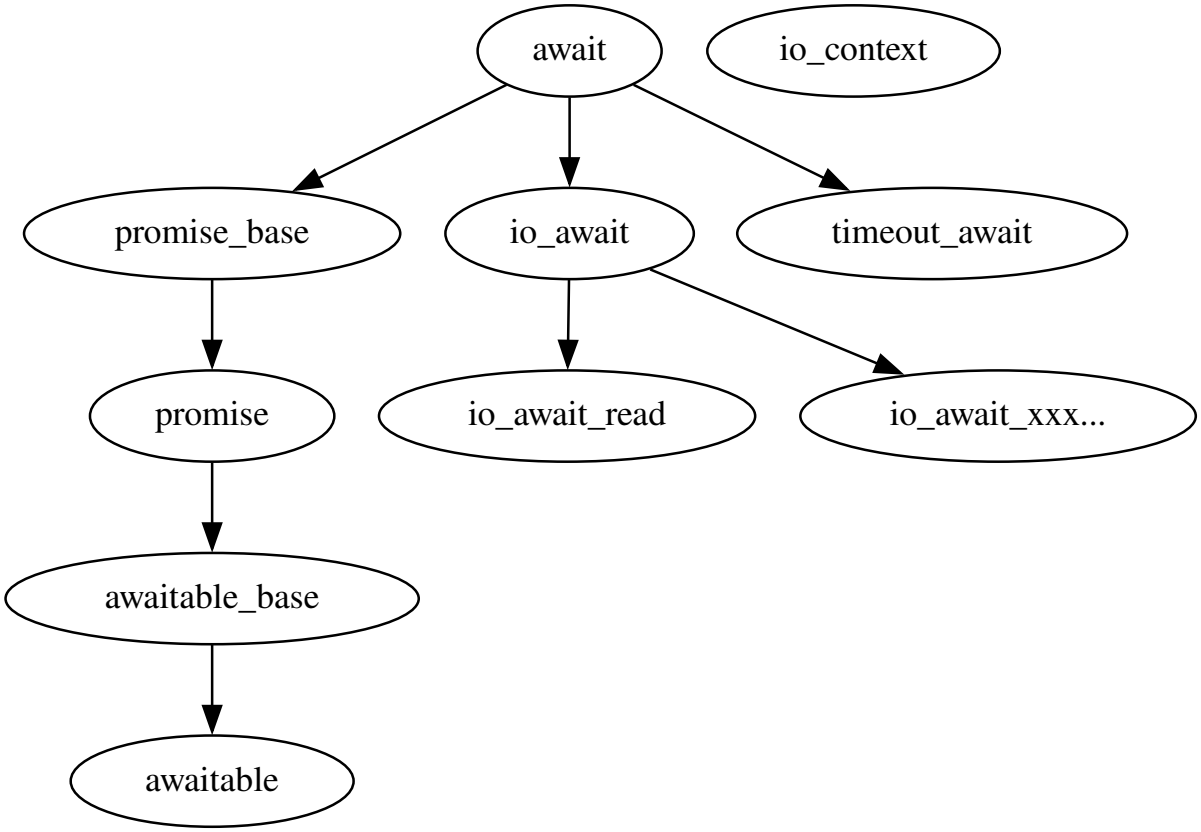


Figure 1: 类型继承图

await 记录了通用等待体的行为（所属 io_context，完成状态，等待此等待体的 promise 对象，内存是否由 io_context 管理）

每个对 await 的继承都是一种等待体的实现，其在 io_context 中的事件循环都有对应代码处理。

promise_base 记录协程对象，io_await 记录 IO 操作，timeout_await 记录超时可等待体。

io_context 管理事件循环，驱动 liburing 进行 IO 事件并等待结果。

zio_ip 实现说明

```
template <IP_Address Address>
inline optional<Address> from_url(const char* a);
struct ipv4 : public sockaddr_in {
  static inline constexpr auto DEFAULT_AF=AF_INET;
  static inline constexpr auto AF();
  static inline constexpr unsigned int length();
```

```
  static inline constexpr void resize(unsigned int);
  inline void set_port(uint16_t hport);
  static inline optional<ipv4> from_url(const char* a);
  static inline ipv4 from_pret(const char* c, uint16_t port);
};
struct ipv6 : public sockaddr_in6 {
  static inline constexpr auto DEFAULT_AF=AF_INET6;
  static inline constexpr auto AF();
  static inline constexpr unsigned int length();
  static inline constexpr void resize(unsigned int);
  inline void set_port(uint16_t hport);
  static inline optional<ipv6> from_url(const char* a);
  static inline ipv6 from_pret(const char* c, uint16_t port);
  string inline to_pret() const;
};
union ipvx{
  ipv4 v4;
  ipv6 v6;
  inline auto AF();
  static inline constexpr void resize(unsigned int);
  inline unsigned int length();
  inline void set_port(uint16_t hport);
  string inline to_pret() const;
};
struct tcp {
  static inline constexpr auto SOCK = SOCK_STREAM;
  static inline constexpr auto PROTO = IPPROTO_TCP;
  template <IP_Address T>
  using acceptor = zio::acceptor<T, zio::ip::tcp>;
  template <IP_Address T>
  static auto async_connect(T&& addr);
};
struct udp {
  static constexpr auto SOCK = SOCK_DGRAM;
  static constexpr auto PROTO = IPPROTO_UDP;
  template <IP_Address T>
  using acceptor = zio::acceptor<T, zio::ip::udp>;
  template <IP_Address T>
  static auto async_connect(T&& addr);
  template <IP_Address T>
  static auto open(T&& addr);
};
```

封装 getaddrinfo/inet_pton 等函数，sockaddr_in/sockaddr_in6 等结构体。并提供 ipvx 通用 ip 地址类，tcp/udp 等方便使用的命名空间。

zio_http 实现说明

```
namespace zio::http {
namespace url {
inline string encode(string_view a);
inline string decode(string_view a);
}  // namespace url
```

```
namespace STATUS_CODE {
inline map<int, string> status_code_to_name = {
    {200, "OK"},
    {404, "Not Found"},
};
}
namespace MIME {
enum data_type {
  html,
  file,
};
inline map<data_type, string> data_type_name = {
    {html, "text/html; charset=utf-8"},
    {file, "application/octet-stream"}};
}  // namespace MIME

struct http_request {
  string _data, content;
  string_view method, url, version;
  unordered_map<string_view, string_view> header;
  inline void prase();
  inline awaitable<void> recv_content(
      zio::buffer::zio_istream<ip::ipv4, ip::tcp>& s);
  inline static awaitable<http_request> read_request(
      zio::buffer::zio_istream<ip::ipv4, ip::tcp>& s);
};

inline string make_http_header(int status_code,
                               size_t size,
                               MIME::data_type type,
                               vector<string> extra = {});
}  // namespace zio::http
```

实现 url 编解码，http 状态吗，MIME 类型，http 请求分析与内容接收，http 头制作。

实验 1 网络聊天室

实验目的

基于 TCP/IP，设计一个类似早期 IRC 的网络聊天室。

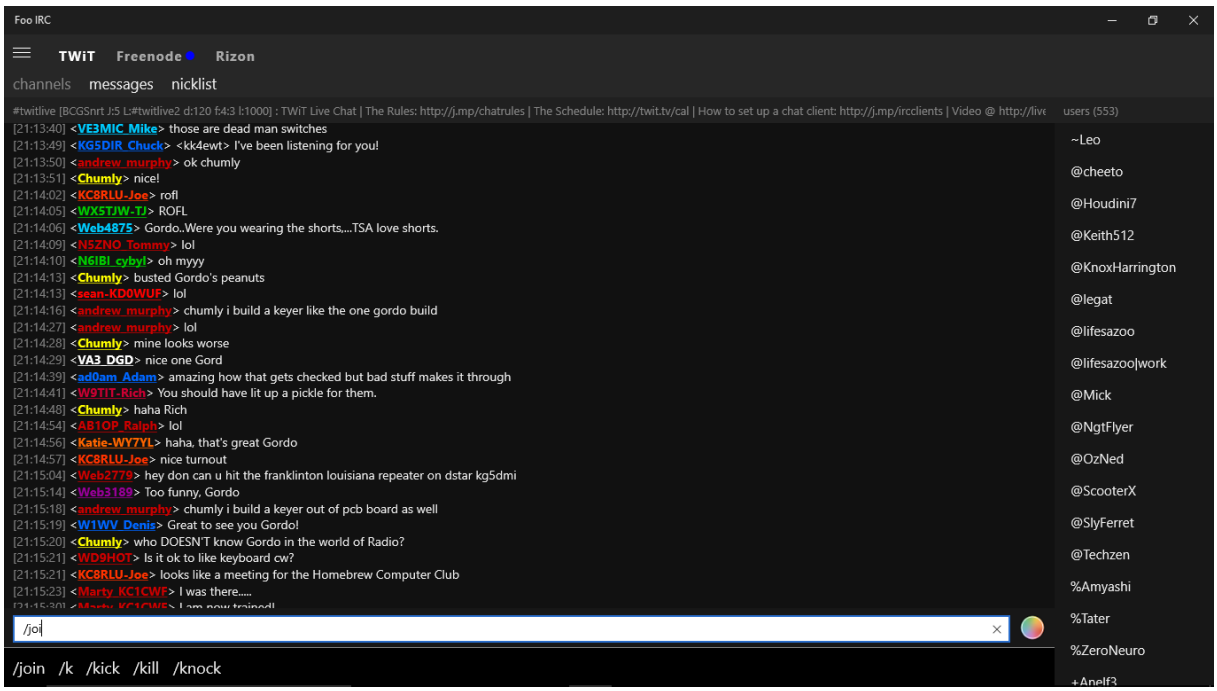


Figure 2: IRC

得益于异步框架的设计，本程序天生异步无阻塞。无需 fork/pthread_create 等进程/线程创建即可流畅的实现网络聊天室。

总体设计

`int main(int argc,char* argv[])`处理传入参数并初始化 io_context，解析后传递给`awaitable<void> server(ipv4 addr)`。`server`监听 ipv4 地址，如果传入连接成功建立，创建 session 并传递连接，在 io_context 中注册`session::run()`开始处理单个连接。

`session::session()`构造函数内将本 session 插入进`list<session*> sessions`，用于查找当前在线的所有`session`。然后初始化 session 用户名。`session::run()`发送欢迎消息并开启消息循环，检测输入回车时将消息传递给`awaitable<bool> handle(shared_ptr<string> message)`。`handle`函数检查消息是否为命令，如果是则进行对应处理，不是则广播至`sessions`记录的全体连接。

如果在`session::run()`中发现连接已断开，则`delete this`，`session::~session()`析构函数删除 sessions 内本实例指针。

详细设计

主程序设计

```
awaitable<void> server(ipv4 addr) {
  tcp::acceptor<ipv4> ac(addr);
  while (auto con = co_await ac.async_accept()) {
    ctx.reg((new session(std::move(con)))->run());
  }
}

int main(int argc, char* argv[]) {
  assert(argc >= 2);
  if (auto ip = ipv4::from_url(argv[1])) {
    cerr << ip->to_pret() << endl;
    ctx.reg(server(*ip));
    ctx.run();
  } else {
    cerr << "url prase error!" << endl;
  }
}
```

main 函数处理命令行参数和初始化 io_context，并传递给 server 函数。server 函数监听连接，并传递给 session::run()。

每个连接设计

```
struct session {
  shared_ptr<string> name;
  connection<ipv4, tcp> con;
  list<session*>::iterator i;
  session(connection<ipv4, tcp> con);
  ~session();
  awaitable<void> welcome();
  awaitable<void> run() {
    {
      co_await welcome();
      broadcast(system_name,make_shared<string>(name->substr(0, name->size() -
          3) +" come chatroom!\n"));
    }
    auto message = make_unique<string>();
    char buffer[1024];
    co_await [&]() -> awaitable<void> {
      while (1) {
        int n1 = co_await con.async_recv(buffer, 1024);
        if (n1 <= 0)
          co_return;
        for (int i = 0; i < n1; i++) {
          *message += buffer[i];
          if (buffer[i] == '\n') {
            if (co_await handle(std::exchange(message, make_unique<string>()))
                {
```

```
                co_return;
              }
              continue;
            }
          }
        }
      }();
      delete this;
    }
    awaitable<bool> handle(shared_ptr<string> message) {
      if (message->starts_with("/help")) {
        co_await con.async_send(help_string, strlen(help_string));
      } else if (message->starts_with("/nick")) {
        auto name_view = strip(string_view(*message).substr(6, 6));
        if (name_view.size()) {
          name = unique_ptr<string>(new string(name_view));
          name->resize(6, ' ');
          *name += " > ";
        }
        auto message = get_date() + *system_name + "changed name to " +name->
            substr(0, name->size() - 3) + "\n";
        co_await send(con.fd, std::move(message));
      } else if (message->starts_with("/online")) {
        vector<shared_ptr<string>> names;
        for (auto i : sessions)
          names.emplace_back(i->name);
        co_await send(con.fd,merge_message(system_name, make_shared<string>("
            Online User:\n")));
        for (auto name : names) {
          co_await send(con.fd,merge_message(system_name,make_shared<string>(name
              ->substr(0, name->size() - 3) + "\n")));
        }
      } else if (message->starts_with("/exit")) {
        co_await send(con.fd, merge_message(*system_name, "Have a good day!\n"));
        co_return true;
      } else
        broadcast(name, message);
      co_return false;
    }
};
```

session::run()开启单个用户的消息循环，如果检测到是一行输入则传递给session::handle()检测是否为对应指令，如果是则进行对应处理，不是则广播至其他所有用户。session::run()根据session::handle()返回值判断是否要关闭连接。

聊天室整体设计

```
struct session;
list<session*> sessions;
io_context ctx;
awaitable<void> send(int fd, string line) {
  co_await async_write(fd, line.data(), line.size());
}
```

```
awaitable<void> send(int fd, shared_ptr<string> line) {
  co_await async_write(fd, line->data(), line->size());
}
void broadcast(shared_ptr<string> name, shared_ptr<string> message) {
  auto line = merge_message(name, message);
  for (auto session : sessions)
    ctx.reg(send(session->con.fd, line));
}
```

broadcast 遍历所有 sessions 发送消息。

实验结果与分析

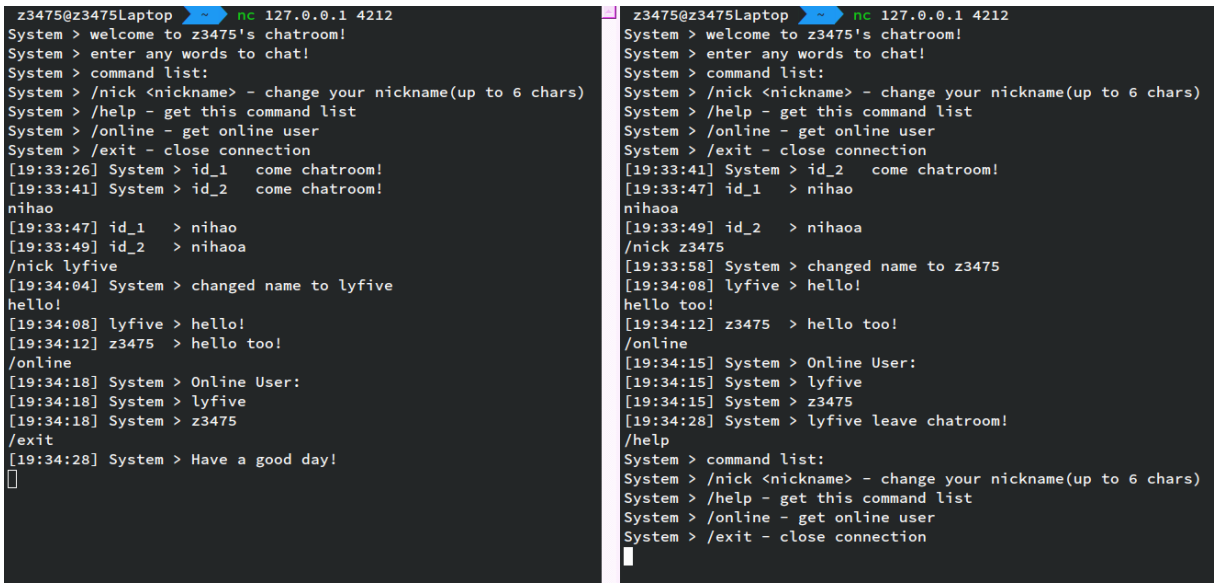


Figure 3: 两客户端聊天室日志

成功实现聊天室功能，并实现查看在线列表和切换用户名功能。

小结与心得体会

在编写 C++ 异步网络程序时一定要注意变量的生命周期问题。对于网络聊天室程序来说，单个 session 的生命周期等同于连接的建立和关闭。不同 session 的生命周期势必会出现交错的情况，编写程序时需要注意避免出现 double free 或者 use after free 的情况。

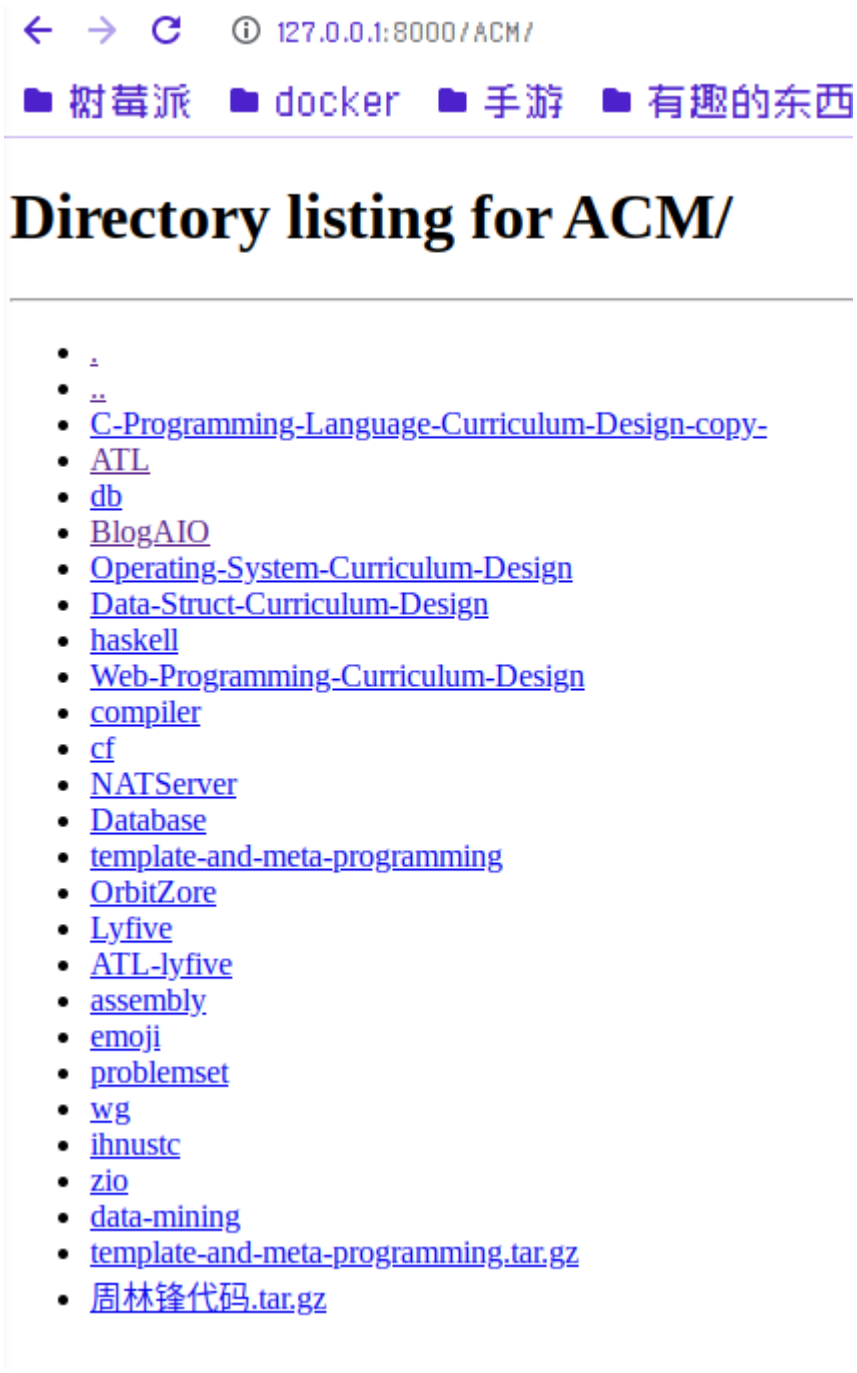
在使用聊天室的过程中，用户输入缓冲区可能会被其他用户聊天给打断显示。经过一番调查之后发现单终端没有很好的解决办法，MUD 和各大 IRC 皆是如此。除非使用特定程序分理出两个缓冲区，这超出了本次课设的范围。

实验 2 简单 http 服务器

实验目的

基于 TCP/IP，设计一个类似 python http.server 的静态文件分享 http 服务器。

本次实验主要参考 RFC7230 https://www.rfc-editor.org/rfc/rfc7230 实现 http/1.1 服务器。



得益于异步框架的设计，本程序天生异步无阻塞。无需 fork/pthread_create 等进程/线程创建即可流畅的实现 http/1.1 服务器。

总体设计

主要有四个函数，`server(ip::ipv4 ip, filesystem::path path)`监听 ipv4 地址。将建立的连接传递 给`http_server(connection<ip::ipv4, ip::tcp> con,filesystem::path path)`，http_server 提供 http/1.1 实现（默认长连接）。如果是 GET 请求则按对应 url 访问对应路径，根据路径是否为目录调用`write_html`写 html 信息或是文件调用`write_file`向客户端写文件。

```
awaitable<void> write_html(connection<ip::ipv4, ip::tcp>& con,
                           string content,
                           int status_code = 200);
awaitable<void> write_file(connection<ip::ipv4, ip::tcp>& con,
                           filesystem::path filename,
                           int status_code = 200);
awaitable<void> http_server(connection<ip::ipv4, ip::tcp> con,
                            filesystem::path path);
awaitable<void> server(ip::ipv4 ip, filesystem::path path);
int main(int argc, char* argv[]);
```

详细设计

main(),server() 设计

```
awaitable<void> server(ip::ipv4 ip, filesystem::path path) {
  auto acceptor = ip::tcp::acceptor<ip::ipv4>(ip);
  while (true) {
    auto x = co_await acceptor.async_accept();
    if (x) {
      timeval timeout;
      timeout.tv_sec = 3;
      timeout.tv_usec = 0;
      if (x.setopt(SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof timeout) < 0 ||
          x.setopt(SOL_SOCKET, SO_SNDTIMEO, &timeout, sizeof timeout) < 0) {
        perror("setsocketopt error!");
      }

      ctx.reg(http_server(std::move(x), path));
    }
  }
}
int main(int argc, char* argv[]) {
  if (argc != 4) {
    cerr << "Usage: simple http file server\n";
    cerr << " <listen_address> <listen_port>\n";
    cerr << " <shared_directory>\n";
    cerr << "Version: 1.0\n";
    cerr << "Compiler: " << COMPILER_NAME << "\n";
    return 1;
  }
  signal(SIGPIPE, SIG_IGN);
  ctx.reg(
      server(ip::ipv4::from_pret(argv[1], *tools::to_int(argv[2])), argv[3]));
```

```
30    ctx.run();
31  }
```

main()函数负责初始化命令行参数，在main()函数里将 SIGPIPE 信号置忽略，防止在之后写文件时客户端关闭导致系统发送 SIGPIPE 信号默认关闭程序。

server()函数负责初始化连接套接字并传递给http_server()开启 http 服务器。

http_server() 设计

```
1  awaitable<void> http_server(connection<ip::ipv4, ip::tcp> con,filesystem::path
       path) {
2    buffer::zio_istream is(std::move(con));
3    while (is.con) {
4      try {
5        auto req = co_await http::http_request::read_request(is);
6        if (!req)
7          co_return;
8        if (req.url.front() == '/')
9          req.url.remove_prefix(1);
10       auto dirpath = path / http::url::decode(req.url);
11       int status_code;
12       if (is_directory(dirpath)) {
13         vec<pair<string, string>> folders = {{".", "."}, {"..", ".."}}, files;
14         for (auto i : filesystem::directory_iterator(dirpath)) {
15           string filename = i.path().filename();
16           (i.is_directory() ? folders : files).push_back({http::url::encode(
                 filename), filename});
17         }
18         co_await write_html(is.con, format(html_template, req.url, req.url,
               folders, files),status_code = 200);
19       } else if (is_regular_file(dirpath)) {
20         co_await write_file(is.con, dirpath, status_code = 200);
21       } else {
22         co_await write_html(is.con, "", status_code = 404);
23       }
24       cerr << format("[{}] \"{} /{} {} {}\"\n", tools::format_now("%c"), req.
             method, req.url, req.version, status_code);
25     } catch (const exception& e) {
26       cerr << e.what() << endl;
27       co_return;
28     }
29   }
30 }
```

http_server 初始化带缓冲区读入，每次读一个请求。根据 URL 判断对应路径是否为目录，如果是目录就进行类似ls的操作，然后写入到 html；如果是文件则直接写入文件。

write_html() 和 write_file() 实现

```
awaitable<void> write_html(connection<ip::ipv4, ip::tcp>& con,
                           string content,
                           int status_code = 200) {
  auto header =
      http::make_http_header(status_code, content.size(), http::MIME::html);
  int n = co_await con.async_write(header.data(), header.size());
  if (n < header.size())
    co_return;
  n = co_await con.async_write(content.data(), content.size());
  if (n < content.size())
    co_return;
}
```

write_html 将 content 当作 html 格式内容，制作 http 头信息并一并写出。

```
awaitable<void> write_file(connection<ip::ipv4, ip::tcp>& con,
                           filesystem::path filename,
                           int status_code = 200) {
  size_t filesize = file_size(filename), available = 0;
  auto fd = open(filename.c_str(), O_RDONLY);
  if (fd < 0)
    status_code = 404;
  auto header = http::make_http_header(status_code, filesize, http::MIME::file)
      ;
  int n = co_await con.async_write(header.data(), header.size());
  if (n < header.size())
    co_return;
  if (fd < 0)
    co_return;
  array<char, 16 * 1024> data;  // buffer size 16k
  while ((filesize -= available) &&
         (available = co_await async_read(fd, &data[0], 16 * 1024))) {
    int c = 0;
    while (c < available) {
      int n = co_await con.async_write(&data[c], available-c);
      if (n < 0) {
        cerr << format("Errorno [ {} ]: {}\n", n, strerror(-n));
        co_return;
      }
      c += n;
    }
  }
  close(fd);
}
```

write_file 将打开文件，制作 http 头信息并发送，初始化 16k 缓冲区进行文件和套接字的同步 IO。

实验结果与分析

```
z3475@z3475Laptop ~/ACM/zio/uring master ± build/examples/zio_example_http_server 0.0.0.0 8000 .
[Wed Dec 28 18:23:10 2022] "GET / HTTP/1.1 200"
[Wed Dec 28 18:23:12 2022] "GET /service-worker.js HTTP/1.1 404"
Errorno [ -32 ]: Broken pipe
[Wed Dec 28 18:23:13 2022] "GET /liburingtest.so HTTP/1.1 200"
[Wed Dec 28 18:23:14 2022] "GET /service-worker.js HTTP/1.1 404"
[Wed Dec 28 18:23:39 2022] "GET /examples/ HTTP/1.1 200"
[Wed Dec 28 18:23:40 2022] "GET /service-worker.js HTTP/1.1 404"
[Wed Dec 28 18:23:41 2022] "GET /examples/tcp_chatroom.cpp HTTP/1.1 200"
[Wed Dec 28 18:23:42 2022] "GET /service-worker.js HTTP/1.1 404"
[Wed Dec 28 18:24:12 2022] "GET /service-worker.js HTTP/1.1 404"
[Wed Dec 28 18:24:41 2022] "GET /CMakeLists.txt HTTP/1.1 200"
[Wed Dec 28 18:25:07 2022] "GET /CMakeLists.txt HTTP/1.1 200"
```

Figure 4: 浏览器访问后台消息

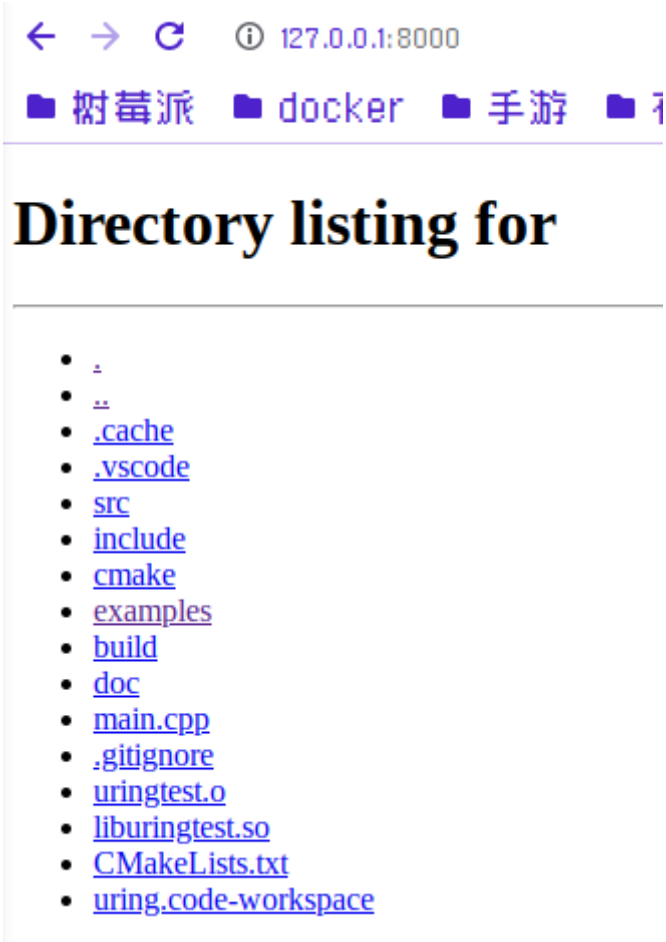


Figure 5: 浏览器访问

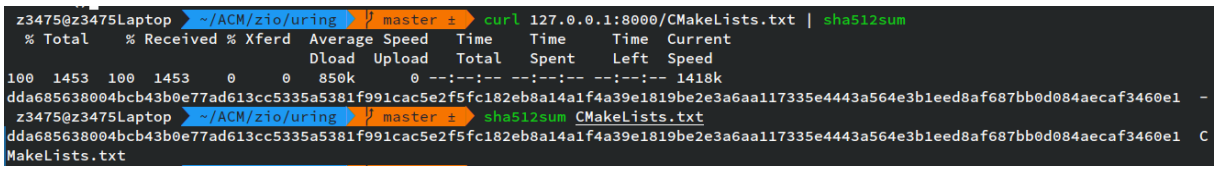


Figure 6: 下载文件并与本地文件比较

成功实现 http 服务器

小结与心得体会

http/1.1 头信息是文本式的，比较方便人类阅读，实现起来也很简单。http/2.0 为了提高头信息解析速度将文本式的头信息拆分成分组发送，http/3.0 更是直接基于 UDP 建立了可靠的消息传送协议 QUIC。可见 http 为了解决速度慢/并发数低/连接延迟高等问题，发展是越来越贴近传输层的。

实验 3 简单 socks5 服务器

实验目的

设计一个 socks5 服务器，并可直接用于浏览器代理。

本次实验主要参考 RFC1928 https://www.rfc-editor.org/rfc/rfc1928 实现 socks5 服务器。

得益于异步框架的设计，本程序天生异步无阻塞。无需 fork/pthread_create 等进程/线程创建即可流畅的实现 socks5 服务器。

总体设计

socks5 是一种代理协议，在连接双方成功建立 socks5 通讯时，接下来的流量完全代表代理的流量。

socks5 建立连接时分为两步，第一步是认证方法选择，第二步是尝试建立代理。在第一步客户端率先发送支持的认证方法，服务端读取并选择一个支持的认证方法。在第二步客户端发送代理连接请求/绑定端口请求，服务器进行对应处理并发送状态。之后代理连接成功建立。

因为大多数应用不需要绑定端口，所以本次实验只实现了连接请求。如果是 UDP 的 socks5 代理则需要在 UDP 对应端口同步监听，代理建立过程和 TCP 一致。同样因为大多数应用建立在 TCP 上，所以本次实验只实现了 TCP socks5 服务器。

详细设计

数据结构设计

首先根据两步四次服务器/客户端交互设计对应数据格式和输入函数。

```
struct selection_request_message {
  char ver, nmethods;
  vector<char> methods;
  template <types::Address Address, types::Protocol Protocol>
  awaitable<bool> read_from(connection<Address, Protocol>& con);
};

struct selection_response_message {
  char ver{0x05}, method{0x00};
  template <types::Address Address, types::Protocol Protocol>
  awaitable<bool> write_to(connection<Address, Protocol>& con);
};

struct socks_request {
  char ver;
  char cmd;
  char rsv;
  char atype;
  vector<char> dst_addr;
  array<char, 2> dst_port;
  template <types::Address Address, types::Protocol Protocol>
  awaitable<bool> read_from(connection<Address, Protocol>& con);
};

struct socks_response {
  char ver{0x05};
  char rep;
  char rsv{0x00};
  char atype;
  vector<char> bind_addr;
  array<char, 2> bind_port;
  template <types::Address Address, types::Protocol Protocol>
  awaitable<bool> write_to(connection<Address, Protocol>& con);
};
```

main(),server(),sock_server() 函数设计

```
int main(int argc, char* argv[]) {
  if (argc != 3) {
    cerr << "Usage: simple socks proxy server\n";
    cerr << " <listen_address> <listen_port>\n";
    cerr << "Version: 1.0\n";
    return 1;
  }
  signal(SIGPIPE, SIG_IGN);
  ctx.reg(server(ip::ipv4::from_pret(argv[1], *tools::to_int(argv[2]))));
  ctx.run();
```

```
}
```

`main()`函数处理命令行参数并传递给`server()`，初始化 io_context。

```
awaitable<void> server(ip::ipv4 ip) {
  auto acceptor = ip::tcp::acceptor<ip::ipv4>(ip);
  while (true) {
    auto x = co_await acceptor.async_accept();
    if (x) {
      timeval timeout;
      timeout.tv_sec = 3;
      timeout.tv_usec = 0;
      if (x.setopt(SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof timeout) < 0 ||
          x.setopt(SOL_SOCKET, SO_SNDTIMEO, &timeout, sizeof timeout) < 0) {
        perror("setsocketopt error!");
      }
      ctx.reg(socks_server(std::move(x)));
    }
  }
}
```

`server()`函数监听连接并将连接传递给`socks_server()`函数。

```
awaitable<void> socks_server(connection<ip::ipv4, ip::tcp> con) {
  selection_request_message srm;
  //step 1
  if (!co_await srm.read_from(con))
    co_return;
  selection_response_message sqm;
  sqm.method = 0;
  if (!co_await sqm.write_to(con))
    co_return;

  //step 2
  socks_request sr;
  if (!co_await sr.read_from(con))
    co_return;
  if (sr.cmd == 0x01) {  // connect command
    ip::ipvx ipvx;
    memset(&ipvx, 0, sizeof(ipvx));
    if (sr.atype == 0x01) {  // ipv4
      ipvx.v4.sin_family = ip::ipv4::AF();
      memcpy(&ipvx.v4.sin_addr, sr.dst_addr.data(), 4);
      memcpy(&ipvx.v4.sin_port, sr.dst_port.data(), 2);
    } else if (sr.atype == 0x04) {  // ipv6
      ipvx.v6.sin6_family = ip::ipv6::AF();
      memcpy(&ipvx.v6.sin6_addr, sr.dst_addr.data(), 16);
      memcpy(&ipvx.v6.sin6_port, sr.dst_port.data(), 2);
    } else if (sr.atype == 0x03) {  // domain
      string s(sr.dst_addr.begin(), sr.dst_addr.end());
      ipvx.v4 = *ip::ipv4::from_url(s.data());
      ipvx.v4.sin_port = htons((unsigned char)sr.dst_port[0] << 8 |
                               (unsigned char)sr.dst_port[1]);
    }
    auto con2 = co_await async_connect<ip::ipvx, ip::tcp>(ipvx);
    socks_response sp;
```

```
    if (con2) {
      cerr<<"From "<<con.getpeer().to_pret()<<" to "<<con2.getpeer().to_pret()
          <<endl;
      sp.rep = 0x00;
      auto local_addr = con2.getaddr();
      if (local_addr.AF() == ip::ipv4::AF()) {
        sp.atype = 0x01;
        sp.bind_addr.resize(4);
        memcpy(&sp.bind_addr[0], &local_addr.v4.sin_addr, 4);
        memcpy(&sp.bind_port[0], &local_addr.v4.sin_port, 2);
      } else {
        sp.atype = 0x04;
        sp.bind_addr.resize(16);
        memcpy(&sp.bind_addr[0], &local_addr.v6.sin6_addr, 16);
        memcpy(&sp.bind_port[0], &local_addr.v6.sin6_port, 2);
      }
      co_await sp.write_to(con);
      ctx.reg(proxy(std::move(con), std::move(con2)));
    } else {
      sp.rep = 0x01;
      co_await sp.write_to(con);
    }
  }
}
```

socks_server()函数进行 socks5 握手并进行连接处理，成功建立连接后传递给proxy()函数。接下来由proxy()函数实现两连接的互传功能。

proxy() 函数和 send() 函数

```
awaitable<void> send(int fd, unique_ptr<array<char, 1024>> ap, int n) {
  co_await async_write(fd, ap->data(), n);
  co_return;
}

template <types::Address Address1,
          types::Protocol Protocol1,
          types::Address Address2,
          types::Protocol Protocol2>
awaitable<void> proxy(connection<Address1, Protocol1> con1,
                      connection<Address2, Protocol2> con2) {
  if (!con1) co_return;
  if (!con2) co_return;
  unique_ptr<array<char, 1024>> con1r = make_unique<array<char, 1024>>(),
                                con2r = make_unique<array<char, 1024>>();
  auto read1 = con1.async_read(con1r->begin(), 1024);
  auto read2 = con2.async_read(con2r->begin(), 1024);
  bool f1 = true, f2 = true;
  while (f1 || f2) {
    co_await wait_any(read1, read2);
    if (f1 && !read1) {
      if (read1.get_return() <= 0)
        f1 = false;
```

```
24        else {
25          ctx.reg(send(con2.fd, std::move(con1r), read1.get_return()));
26          con1r = make_unique<array<char, 1024>>();
27          read1 = con1.async_read(con1r->begin(), 1024);
28        }
29      }
30      if (f2 && !read2) {
31        if (read2.get_return() <= 0)
32          f2 = false;
33        else {
34          ctx.reg(send(con1.fd, std::move(con2r), read2.get_return()));
35          con2r = make_unique<array<char, 1024>>();
36          read2 = con2.async_read(con2r->begin(), 1024);
37        }
38      }
39    }
40  }
```

为了避免进行async_write()时写入数据被新async_read覆盖。这里在每次async_read时创建一个新的缓冲区，如果需要async_write则将缓冲区传递给send()函数，缓冲区所有权移交给send()函数。

proxy()函数内部同时等待读两个连接的数据操作。如果成功读入则写入给对面连接并创建一个新读取操作。

实验结果与分析

```
z3475@z3475Laptop ~/ACM/zio/uring  master ± curl -v www.baidu.com -x socks5://127.0.0.1:1242
* Uses proxy env variable no_proxy == '0.0.0.0/8,10.0.0.0/8,100.64.0.0/10,127.0.0.0/8,169.254.0.0/16,172.16.0.0/12,192.0.0.0/24,192
.0.2.0/24,192.88.99.0/24,192.168.0.0/16,198.18.0.0/15,198.51.100.0/24,203.0.113.0/24,224.0.0.0/3,::1/128,fc00::/7,fe80::/10'
*   Trying 127.0.0.1:1242...
* Connected to 127.0.0.1 (127.0.0.1) port 1242 (#0)
* SOCKS5 connect to IPv4 14.215.177.39:80 (locally resolved)
* SOCKS5 request granted.
* Connected to 127.0.0.1 (127.0.0.1) port 1242 (#0)
> GET / HTTP/1.1
> Host: www.baidu.com
> User-Agent: curl/7.87.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Accept-Ranges: bytes
< Cache-Control: private, no-cache, no-store, proxy-revalidate, no-transform
< Connection: keep-alive
< Content-Length: 2381
< Content-Type: text/html
< Date: Wed, 28 Dec 2022 19:14:28 GMT
< Etag: "588604dd-94d"
< Last-Modified: Mon, 23 Jan 2017 13:27:57 GMT
< Pragma: no-cache
< Server: bfe/1.0.8.18
< Set-Cookie: BDORZ=27315; max-age=86400; domain=.baidu.com; path=/
<
<!DOCTYPE html>
<!--STATUS OK--><html> <head><meta http-equiv=content-type content=text/html;charset=utf-8><meta http-equiv=X-UA-Compatible content
=IE=Edge><meta content=always name=referrer><link rel=stylesheet type=text/css href=http://s1.bdstatic.com/r/www/cache/bdorz/baidu.
min.css><title>百度一下，你就知道</title></head> <body link=#0000cc> <div id=wrapper> <div id=head> <div class=head_wrapper> <div c
lass=s_form> <div class=s_form_wrapper> <div id=lg> <img hidefocus=true src=//www.baidu.com/img/bd_logo1.png width=270 height=129>
</div> <form id=form name=f action=//www.baidu.com/s class=fm> <input type=hidden name=bdorz_come value=1> <input type=hidden name=
ie value=utf-8> <input type=hidden name=f value=8> <input type=hidden name=rsv_bp value=1> <input type=hidden name=rsv_idx value=1>
 <input type=hidden name=tn value=baidu><span class="bg s_ipt_wr"><input id=kw name=wd class=s_ipt value maxlength=255 autocomplete
=off autofocus></span><span class="bg s_btn_wr"><input type=submit id=su value=百度一下 class="bg s_btn"></span> </form> </div> </d
iv> <div id=u1> <a href=http://news.baidu.com name=tj_trnews class=mnav>新闻</a> <a href=http://www.hao123.com name=tj_trhao123 cla
```

Figure 7: curl 测试

```
z3475@z3475Laptop ~/ACM/zio/uring/build  master ± examples/zio_example_socks_proxy 0.0.0.0 1242
From 127.0.0.1:41172 to 14.215.177.39:80
```

Figure 8: curl 测试后台状态

```
 z3475@z3475Laptop  ~/ACM/zio/uring/build  master ±  examples/zio_example_socks_proxy 0.0.0.0 1242
From 127.0.0.1:41172 to 14.215.177.39:80
From 127.0.0.1:60792 to 119.100.50.33:443
From 127.0.0.1:60806 to 119.100.50.33:443
From 127.0.0.1:60820 to 119.100.50.33:443
From 127.0.0.1:60824 to 119.100.50.33:443
From 127.0.0.1:60826 to 14.215.177.38:443
From 127.0.0.1:60830 to 175.6.206.33:443
From 127.0.0.1:60836 to 113.219.142.36:443
From 127.0.0.1:60838 to 111.170.26.38:443
From 127.0.0.1:60848 to 180.101.49.57:443
From 127.0.0.1:44112 to 180.101.49.186:443
From 127.0.0.1:44126 to 47.103.132.209:443
From 127.0.0.1:44136 to 14.215.177.38:443
From 127.0.0.1:44148 to 14.215.177.38:443
From 127.0.0.1:44160 to 14.215.177.38:443
From 127.0.0.1:44170 to 175.6.243.36:443
From 127.0.0.1:44172 to 182.106.158.36:443
From 127.0.0.1:44186 to 113.219.142.36:443
From 127.0.0.1:44196 to 113.219.142.36:443
From 127.0.0.1:44200 to 113.219.142.36:443
From 127.0.0.1:44208 to 113.219.142.36:443
From 127.0.0.1:44218 to 113.219.142.36:443
From 127.0.0.1:44226 to 113.219.142.36:443
```

Figure 9: 浏览器访问 www.baidu.com 测试后台状态

成功实现了 socks5 代理。

小结与心得体会

梦开始的地方。