yasio Documentation

发布 *latest*

yasio

2020 年 07 月 17 日

Contents

yasio 是一个轻量级跨平台的异步 *socket* 库，专注于客户端和基于各种游戏引擎的游戏客户端网络服务，支持 *windows & linux & apple & android & win10-universal*。

重要: 由于作者还没找到比较完美的 kcp 实现方案，目前对 kcp 的封装尚处于 experimental 实验阶段，详见：speedtest。

- yasio GitHub
- yasio 文档
  - English
  - 简体中文
  - 下载 PDF
- yasio 官网：https://yasio.org/

CHAPTER 1

成功案例

红警 OL 手游项目：用于客户端网络传输，并且随着该项目于 2018 年 10 月 17 日由腾讯游戏发行正式上线后稳定运行于上千万移动设备上。

CHAPTER 2

特点

yasio 借鉴著名的 boost 网络库 asio, 在保持轻量级的情况下，具备以下特点:

- 支持 IPv6_only 网络。
- 支持处理多个连接的所有网络事件。
- 支持微秒级定时器。
- 支持 TCP 粘包处理，业务完全不必关心。
- 支持处理由于 EINTR 信号导致收发数据中断的情形。
- 支持 Lua 绑定。
- 支持 Cocos2d-x jsb 绑定。
- 支持 CocosCreator jsb2.0 绑定。
- 支持 Unity3D C# 绑定。
- 支持组播。
- 支持 SSL 客户端，基于 OpenSSL。
- 支持非阻塞域名解析，基于 c-ares。
- 支持 header only 集成方式，意味着无需编译，直接像 c++ stl 一样包含头文件即可使用，使用预编译宏 *YASIO_HEADER_ONLY* 开启。
- 跨平台性:
  - 编译器:

- Visual Studio 2013+
- GCC4.7+
- xcode9+
- 其他支持 C++11,14,17 标准的编译器
- 架构: x86, x64, ARM 等。
- 操作系统: Windows, macOS, Linux, iOS, Android 等。

CHAPTER 3

用法一览

此简单例子向 ip138.com 发送 http 请求并打印返回结果

C++

```
#include "yasio/yasio.hpp"
#include "yasio/obstream.hpp"
using namespace yasio;
using namespace yasio::inet;

int main()
{
  io_service service({"www.ip138.com", 80});
  service.set_option(YOPT_S_DEFERRED_EVENT, 0); // 直接在网络线程分派网络事件
  service.start([&](event_ptr&& ev) {
    switch (ev->kind())
    {
      case YEK_PACKET: {
        auto packet = std::move(ev->packet());
        fwrite(packet.data(), packet.size(), 1, stdout);
        fflush(stdout);
        break;
      }
      case YEK_CONNECT_RESPONSE:
```

(下页继续)

(续上页)

```
        if (ev->status() == 0)
        {
          auto transport = ev->transport();
          if (ev->cindex() == 0)
          {
            obstream obs;
            obs.write_bytes("GET /index.htm HTTP/1.1\r\n");

            obs.write_bytes("Host: www.ip138.com\r\n");

            obs.write_bytes("User-Agent: Mozilla/5.0 (Windows NT 10.0; "
                            "WOW64) AppleWebKit/537.36 (KHTML, like Gecko) "
                            "Chrome/79.0.3945.117 Safari/537.36\r\n");
            obs.write_bytes("Accept: */*;q=0.8\r\n");
            obs.write_bytes("Connection: Close\r\n\r\n");

            service.write(transport, std::move(obs.buffer()));
          }
        }
        break;
      case YEK_CONNECTION_LOST:
        printf("The connection is lost.\n");
        break;
    }
  });
  // open channel 0 as tcp client
  service.open(0, YCK_TCP_CLIENT);
  getchar();
}
```

Lua

```
local ip138 = "www.ip138.com"
local service = yasio.io_service.new({host=ip138, port=80})
local respdata = ""
-- 传入网络事件处理函数启动网络服务线程，网络事件有：消息包，连接响应，连接丢失
service:start(function(ev)
        local k = ev.kind()
        if (k == yasio.YEK_PACKET) then
            respdata = respdata .. ev:packet():to_string()
```

(下页继续)

(续上页)

```
        elseif k == yasio.YEK_CONNECT_RESPONSE then
            if ev:status() == 0 then -- status 为 0 表示连接建立成功
                local transport = ev:transport()
                local obs = yasio.obstream.new()
                obs:write_bytes("GET / HTTP/1.1\r\n")

                obs:write_bytes("Host: " .. ip138 .. "\r\n")

                obs:write_bytes("User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64)␣
↪AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.117 Safari/537.36\r\n")
                obs:write_bytes("Accept: */*;q=0.8\r\n")
                obs:write_bytes("Connection: Close\r\n\r\n")

                service:write(transport, obs)
            end
        elseif k == yasio.YEK_CONNECTION_LOST then
            print("request finish, respdata: " ..  respdata)
        end
    end)
-- 将信道 0 作为 TCP 客户端打开，并向服务器发起异步连接，进行 TCP 三次握手
service:open(0, yasio.YCK_TCP_CLIENT)

-- 由于 lua_State 和渲染对象，不支持在其他线程操作，因此分派网络事件封装为全局 Lua 函数，并且
以下函数应该在主线程或者游戏引擎渲染线程调用
function gDispatchNetworkEvent(...)
    service:dispatch(128) -- 每帧最多处理 128 个网络事件
end

_G.yservice = service -- Store service to global table as a singleton instance
```

更多使用实例，请参考 tests & examples :

- tests:
  - echo_server: TCP/UDP 回射服务器
  - echo_client: TCP, UDP 回射客户端
  - ssltest: SSL 测试客户端，请求 github.com 主页并打印返回数据
  - tcptest: TCP 测试程序
  - speedtest: TCP,UDP,KCP 本机传输速率测试程序
  - mcast: 组播测试程序

- examples:
  - ftp_server: 基于 yasio 实现的仅支持下载的 ftp 服务器，点击 访问
  - lua: lua 样例程序，包含并发 http 请求，TCP 拆包实例代码
  - DemoU3D: Unity3D xLua 集成 Demo
  - DemoUE4: 虚幻 4 集成 Demo

CHAPTER 4

构建 tests & examples

- 确保已安装支持 C++11 标准的编译器，例如 msvc, gcc, clang 等
- 确保已安装 git, cmake
- 运行如下命令:

```
git clone https://github.com/yasio/yasio
cd yasio
git submodule update --init --recursive
cd build
# 注意，这里 xcode 需要运行: cmake .. -GXcode
cmake ..
cmake --build . --config Debug
```

CHAPTER 5

yasio 异步网络服务的三要素

- io_channel: 信道，负责连接管理，建立，关闭，重连，启动 TCP/UDP 服务监听
- io_transport: 传输会话，类似文件句柄，用于实际数据收发，由框架内部产生并通过网络事件返回
- io_service：核心网络服务，在内部采用 反应堆模式（Reactor）处理所有网络事件，对用户提供 前置器模式（Proactor）使用方式，即用户只管投递异步消息发送和处理框架返回的消息事件即可，无需关心具体发送、接收和拆包细节

CHAPTER 6

API 文档

6.1 yasio API 文档

6.1.1 yasio 常用接口

yasio::highp_clock

函数模板，获取微秒级稳定时钟或者系统时钟。

```
template <typename _Ty>
inline long long highp_clock();
```

Example

C++

```
long long t1 = yasio::highp_clock<yasio::steady_clock_t>(); // 获取微秒级 CPU 时钟
long long t2 = yasio::highp_clock<yasio::system_clock_t>(); // 获取微秒级 UTC 时间
```

yasio::xhighp_clock

函数模板，获取纳秒级稳定时钟或者系统时钟。

```
template <typename _Ty>
inline long long xhighp_clock();
```

Example

C++

```
long long t1 = yasio::xhighp_clock<yasio::steady_clock_t>(); // 获取纳秒级 CPU 时钟
long long t2 = yasio::xhighp_clock<yasio::system_clock_t>(); // 获取纳秒级 UTC 时间
```

6.1.2 yasio CLASSES

io_service

yasio 的核心类，提供 TCP,UDP,KCP 异步网络服务，以独立线程处理所有网络事件。

命名空间

```
namespace yasio::inet
```

成员

公共构造函数:

函数名	函数说明
io_service::io_service	构造 io_service 对象

公共方法:

函数名	函数说明
io_service::start	启动网络服务
io_service::stop	停止网络服务
io_service::is_running	判断网络线程是否运行
io_service::dispatch	在调用者线程分派网络事件（包，连接响应，连接丢失）
io_service::set_option	设置选项
io_service::open	打开信道
io_service::is_open	判断信道或传输会话是否已打开
io_service::close	关闭信道或 Transport
io_service::write	异步写入数据
io_service::write_to	异步写入数据, 仅用于非绑定状态的 UDP
io_service::schedule	启动定时器
io_service::builtin_resolv	内置域名解析
io_service::channel_at	根据信道索引获取信道对象
io_service::init_globals	显示初始化 io_service 全局状态

io_service::io_service

构造 io_service 对象，共有 4 个重载版本

```
io_service::io_service()
io_service::io_service(int channel_count)
io_service::io_service(const io_hostent& channel_eps)
io_service::io_service(const io_hostent* channel_eps, int channel_count)
```

Example

C++

```
io_service s1; // s1 对象仅支持 1 个信道
io_service s2(5); // s2 对象仅支持 5 个信道并发网络处理，即可以同时处理 5 个客户端连接或开 5
个 TCP server
io_service s3(io_hostent{"github.com", 443}); // s3 对象仅支持 1 个信道
io_hostent hosts[] = {
  {"192.168.1.66", 20336},
  {"192.168.1.88", 20337},
};
io_service s4(hosts, YASIO_ARRAYSIZE(hosts)); // s4 支持 2 个信道
```

io_service::start

启动网络服务

```
void start(io_event_cb_t cb)
```

Parameters

cb

接收网络事件的回调函数

Example

C++

```
auto service = yasio_shared_service(io_hostent{host="ip138.com", port=80});
service->start([](event_ptr&& ev) {
  auto kind = ev->kind();
  if (kind == YEK_CONNECT_RESPONSE)
  {
    if (ev->status() == 0)
      printf("[%d] connect succeed.\n", ev->cindex());
    else
      printf("[%d] connect failed!\n", ev->cindex());
  }
});
```

io_service::stop

停止网络服务

```
void stop()
```

io_service::is_running

判断网络服务是否运行

```
bool is_running() const
```

io_service::dispatch

在调用者线程分派网络事件

```
void dispatch(int max_count)
```

Parameters

max_count
每次调用分派最大网络事件数, 通常 128 足够

Example

C++

```
// 通常在 OpenGL 或 cocos 和 unity 等游戏引擎渲染线程调用,
// 以便在特定网络消息回调里安全地更新界面逻辑。
yasio_shared_service()->dispatch(128);
```

io_service::set_option

设置选项, 是可变参接口，根据 opt 不同，参数个数和类型不同

```
void set_option(int opt, ...)
```

Parameters

opt
选项类型，以 YOPT_ 开头的枚举值, 详见: *io_service::options*

Example

C++

```
io_hostent hosts[] = {
  {"192.168.1.66", 20336},
  {"192.168.1.88", 20337},
};
```

(下页继续)

(续上页)

```
io_service* service = new io_service(hosts, YASIO_ARRAYSIZE(hosts));

// 对于有包长度字段的协议，对于 tcp 自定义二进制协议，强烈建议设计包长度字段，并设置此选项，业务无须关心粘包问题
service->set_option(YOPT_C_LFBFD_PARAMS,
                    0,     // channelIndex, 信道索引
                    65535, // maxFrameLength, 最大包长度
                    0,     // lenghtFieldOffset, 长度字段偏移，相对于包起始字节
                    4, // lengthFieldLength, 长度字段大小，支持 1 字节，2 字节，3 字节，4 字节
                    0 // lengthAdjustment: 如果长度字段字节大小包含包头，则为 0， 否则，这里 = 包头大小
);

// 对于没有包长度字段设计的协议，例如 http， 设置包长度字段为-1，
// 那么底层服务收到多少字节就会传回给上层多少字节
service->set_option(YOPT_C_LFBFD_PARAMS, 1, 65535, -1, 0, 0);
```

io_service::open

打开信道

```
void open(size_t cindex, int kind)
```

Parameters

cindex

信道索引

kind

信道类型，可取值:

- YCK_TCP_CLIENT
- YCK_TCP_SERVER
- YCK_UDP_CLIENT
- YCK_UDP_SERVER
- YCK_KCP_CLIENT

- YCK_KCP_SERVER
- YCK_SSL_CLIENT

Example

C++

```
// 将信道 0 作为 TCP 客户端打开，发起 TCP 三次握手和服务器建立连接
yasio_shared_service()->open(0, YCK_TCP_CLIENT);
```

io_service::is_open

判断信道或传输会话是否已打开, 共有 2 个重载版本，分别用于信道和传输会话

```
bool is_open(transport_handle_t) const
bool is_open(int cindex) const
```

io_service::close

关闭信道或传输会话

```
void close(transport_handle_t transport)
void close(int cindex)
```

Parameters

transport
传输会话句柄

io_service::write

异步发送数据

```
int write(transport_handle_t thandle, std::vector<char> buffer,
              io_completion_cb_t completion_handler = nullptr)
```

Parameters

thandle
传输会话句柄

buffer
要发送的数据

completion_handler
发送完成回调，在网络线程调度, 原型: *io_completion_cb_t*

io_service::write_to

非阻塞发送 UDP 数据

```
int write_to(transport_handle_t thandle, std::vector<char> buffer,
                          const ip::endpoint& to, io_completion_cb_t completion_handler␣
↪= nullptr)
```

Parameters

thandle
传输会话句柄

buffer
要发送的数据, 空 buffer，直接忽略，也不会触发 completion_handler

to
发送远端地址

completion_handler
发送完成回调，在网络线程调度，kcp 暂不支持此回调, 原型: *io_completion_cb_t*

Remark

只能用于未使用 connect 绑定过 4 元组的 UDP socket.

io_service::schedule

注册定时器

```
highp_timer_ptr schedule(const std::chrono::microseconds& duration, timer_cb_t cb)
```

Parameters

duration

指定定时器超时时间

cb

定时器超时回调函数，在网络线程调度

Return Value

定时器引用计数的智能指针，用户可持有此指针对定时器进行操作

Example

C++

```
// 注册一个 3 秒后超时的一次性计时器，超时后定时器会被自动销毁
yasio_shared_service()->schedule(std::chrono::seconds(3), []()->bool{
  printf("time called!\n");
  return true;
});

// 注册一个每隔 5 秒循环执行的计时器
auto loopTimer = yasio_shared_service()->schedule(std::chrono::seconds(5), []()->bool{
  printf("time called!\n");
  return false;
});
```

io_service::builtin_resolv

内置域名解析，会自动判断本地主机 ipv6 网络环境情况

```
int builtin_resolv(std::vector<ip::endpoint>& endpoints, const char* hostname,
                                   unsigned short port = 0)
```

Parameters

endpoints
存储域名解析结果地址列表

hostname
域名

port
端口

Return Value

返回 0 成功，-1 失败

io_service::channel_at

根据信道索引获取信道对象

```
io_channel* channel_at(size_t cindex) const
```

Parameters

cindex
信道索引

io_service::init_globals

显示初始化 io_service 全局状态, 静态成员函数，非必须调用，可在程序启时调用

```
static void init_globals(print_fn_t print_fn)
```

Parameters

print_fn
自定义打印函数，

Remark

- 如果不希望重定向初始化日志，则无需调用此函数，否则可调用，例如重定向 yasio 日志到 UE4 或 Unity3D 编辑器输出窗口
- 如果不显示调用此函数，则 yasio 内部自动初始化

io_completion_cb_t

Prototype

```
typedef std::function<void(int ec, size_t bytes_transferred)> io_completion_cb_t;
```

Parameters

ec
错误码
UDP: 如果发送出错，可通过此错误码决定是否关闭 transport
TCP: 始终为 0，用户无需关心

bytes_transferred
传输完成字节数
UDP: 如果出错则可能为 0 或者小于请求发送字节数
TCP: 始终等于请求发送的字节数，因此用户可忽略

io_service::options

枚举值	参数说明
YOPT_S_DEFERRED_EVENT	设置是否使用事件队列延迟分派网络事件, 参数 deferred:int，默认值 1
YOPT_S_RESOLV_FN	设置自定义域名解析回调，参数类型 resolv_fn_t*
YOPT_S_PRINT_FN	[废弃] 请使用 io_service::init_globals 替代，设置打印函数, 参数类型 print_fn_t*
YOPT_S_EVENT_CB	设置网络事件回调, 参数类型 io_event_cb_t*
YOPT_S_TCP_KEEPALIVE	设置 TCP 底层心跳, 参数: idle:int(7200), interal:int(75), probes:int(10)
YOPT_S_NO_NEW_THREAD	设置是否禁用线程，直接在阻塞在 start_service 调用者线程处理网络事件，参数类型 int, 默认值 0
YOPT_S_SSL_CACERT	设置 ssl 客户端证书，参数类型 const char*
YOPT_S_CONNECT_TIMEOUT	设置 TCP 客户端连接超时，参数类型 int，默认值 10(s)
YOPT_S_DNS_CACHE_TIMEOUT	设置 DNS 解析缓存超时时间，参数:L dns_cache_timeout:int 默认值 600(s)
YOPT_S_DNS_QUERIES_TIMEOUT	设置 DNS 解析超时时间，参数类型: dns_resov_timeout:int，默认值 10(s)，仅当启用 c-ares 异步域名解析时才有效
YOPT_C_LFBFD_FN	设置信道自定义长度解析函数，用于 TCP 底层拆包，参数: cindex:int, decode_len_fn_t*
YOPT_C_LFBFD_PARAMS	设置信道基于 netty 的 LengthBasedFrameDecoder 拆包参数，参数: cindex:int, max_frame_length:int, length_field_offset:int, length_field_length:int, length_adjustment:int
YOPT_C_LOCAL_PORT	

io_event

网络事件回调返回此对象。

命名空间

```
namespace yasio::inet
```

成员

公共方法:

io_event::kind

获取事件类型

```
int kind() const
```

Return Value

- YEK_PACKET：消息事件
- YEK_CONNECT_RESPONSE：连接响应事件
- YEK_CONNECTION_LOST：连接断开事件

io_event::status

获取事件类型

```
int status() const
```

Return Value

0: 正常，非 0：出错
业务只需要处理 0 和非 0 的情况，无需关心具体状态码，非 0 时可选择做简单打印记录

io_event::packet

获取事件中的完整消息包

```
std::vector<char>& packet()
```

io_event::transport

获取事件到传输会话句柄

```
transport_handle_t transport()
```

io_event::timestamp

返回事件产生的微秒级时间戳

```
long long timestamp() const
```

io_event::transport_id()

获取传输会话 ID，可用于日志跟踪

```
unsigned int id() const
```

Return Value

返回 ID 是全局自增的，可保证在 32 位整数最大范围内保证唯一, (uint32_t)-1 视为无效 ID

io_event::transport_udata

安全地获取和设置用户变量，所有类型网络事件，调用次函数都是安全的

```
_Uty io_event::transport_udata()
void io_event::transport_udata(_Uty uservalue)
```

Remark

注意: 生命周期需要使用者自己维护，收到 connect success 存储 userdata, 收到 connect lost 事件，清理 userdata

io_channel

信道，负责连接管理，固定对象，用户不可构造, 可通过 `io_service::cindex_to_handle` 接口获取

命名空间

```
namespace yasio::inet
```

成员

公共方法:

函数名	函数说明
io_channel::get_service	获取管理本信道的 io_service 对象
io_channel::index	获取索引
io_channel::remote_port	获取远程端口，客户端：连接端口，服务器：监听端口

io_channel::get_service

获取管理本信道的 io_service 对象

```
io_service& get_service()
```

io_channel::index

```
int index() const
```

io_channel::remote_port

获取远程端口。
客户端：连接端口
服务器：监听端口

```
u_short remote_port() const
```

io_transport

传输会话，负责数据传输，通过 io_event::transport 接口中获取。

注解: io_transport 所有公共方法，如需调用，请在使用接口 `io_service::is_open` 返回 true 时调用。

命名空间

```
namespace yasio::inet
```

成员

公共方法:

函数名	函数说明
io_transport::local_endpoint	获取本地地址
io_transport::peer_endpoint	获取对端地址
io_transport::get_context	获取管理会话的信道

io_transport::local_endpoint

返回本地 IP 和端口

```
ip::endpoint local_endpoint() const
```

io_transport::peer_endpoint

返回通信对端 IP 和端口

```
ip::endpoint peer_endpoint() const
```

io_transport::get_context

获取管理会话的信道对象

```
io_channel* get_context() const
```

obstream

二进制写入流，可以非常方便地进行二进制编码，写入数值类型时自动转换为网络字节序

命名空间

```
namespace yasio
```

成员

函数名	函数说明
obstream::obstream	构造 obstream 对象

公共方法:

函数名	函数说明
obstream::write_i	函数模板，写入数值数据，会自动转化为网络字节序
obstream::write_i24	写入 24 位有符号整数
obstream::write_u24	写入 24 位无符号整数
obstream::write_7b	写入 7bit Encoded Int 变长存储
obstream::write_v	写入字节流数据，会先写入 7bit 编码的变长长度域
obstream::write_v32	写入字节流数据，使用 32bit 长度域
obstream::write_v16	写入字节流数据，使用 16bit 长度域
obstream::write_v8	写入字节流数据，使用 8bit 长度域
obstream::write_byte	写入一个字节
obstream::write_bytes	写入字节流数据，不含长度域
obstream::buffer	获取 stream 中所有字节流数据
obstream::empty	判断 stream 是否为空
obstream::length	获取 stream 总字节数
obstream::data	获取字节数据指针
obstream::pwrite_i	在指定偏移位置写入数值数据，会自动转化为网络字节序
obstream::swrite_i	静态方法，在指定地址写入数值数据，会自动转化为网络字节序
obstream::sub	截取子 stream，返回新 obstream 对象

Example

C++

```
obstream obs;

// 内存布局 |-8bits-|
obs.write_i<int8_t>(12);

// 内存布局 |-8bits-|-32bits-|
obs.push32();

// 内存布局 |-8bits-|-32bits-|-16bits-|
obs.write_i<int16_t>(52013);

// 内存布局 |-8bits-|-32bits-|-16bits-|-8bits(length of the string)-|-88bits(the string)-
↪|
obs.write_v8("hello world");

// 将整包长度字节数写入最近一次 push 保留的 4 字节内存，完成封包，内存布局不变
obs.pop32(obs.length());
```

ibstream_view 和 ibstream

二进制读取流，ibstream_view 借鉴 C++17 标准引入的 `std::string_view` 无 GC 读取二进制数据，读取数值类型时自动转换为主机字节序，如果需要使用深拷贝对象，使用 ibstream 即可，ibstream 继承了所有 ibstream_view 关于二进制读取的接口。

命名空间

```
namespace yasio
```

成员

函数名	函数说明
ibstream_view::ibstream_view	构造 ibstream_view 对象

公共方法:

函数名	函数说明
ibstream_view::reset	重置输入数据
ibstream_view::read_i	函数模板，写入数值数据，会自动转化为网络字节序
ibstream_view::read_i24	读取 24 位有符号整数
ibstream_view::read_u24	读取 24 位无符号整数
ibstream_view::read_7b	读取 7bit Encoded Int 变长存储
ibstream_view::read_v	读取字节流数据，会先读取 7bit 编码的变长长度域
ibstream_view::read_v32	读取字节流数据，使用 32bit 长度域
ibstream_view::read_v16	读取字节流数据，使用 16bit 长度域
ibstream_view::read_v8	读取字节流数据，使用 8bit 长度域
ibstream_view::read_byte	读取一个字节
ibstream_view::read_bytes	读取字节流数据，无长度域
ibstream_view::seek	移动流指针，和系统文件 io lseek 用法相同
ibstream_view::length	获取 stream 总字节数
ibstream_view::data	获取字节数据指针
ibstream_view::sread_i	静态方法，在指定地址读取数值数据，会自动转化为网络字节序

Example

C++

```
ibstream_view ibs(event->packet().data(), event->packet().size());

// 读取 1 字节整数
int8_t cmd = ibs.read_i<int8_t>();

// 跳过 4 字节长度域
ibs.seek(4, SEEK_CUR);

int16_t seq = ibs.read_i<int16_t>();
cxx17::string_view content = ibs.read_v();
```

xxsocket

xxsocket 是跨平台 socket 工具类，屏蔽了各操作系统平台差异，提供统一接口，是 yasio 网络库的起源, 支持 C++11 move 语义。

命名空间

```
namespace yasio::inet
```

成员

公共构造函数:

函数名	函数说明
xxsocket::xxsocket	构造 xxsocket 对象

公共方法:

函数名	函数说明
xxsocket::xpconnect	发起 TCP 连接，支持本地 ipv4/ipv6 检测
xxsocket::xpconnect_n	发起 TCP 连接，支持本地 ipv4/ipv6 检测, 支持超时
xxsocket::pconnect	发起 TCP 连接
xxsocket::pconnect_n	发起 TCP 连接, 支持超时
xxsocket::pserv	启动 TCP 服务器
io_service::swap	交换 socket 文件描述符
xxsocket::open	打开套接字
xxsocket::reopen	重新打开套接字
xxsocket::is_open	判断套接字是否已打开
xxsocket::native_handle	返回内部 xxsocket 管理的 socket 文件描述符
xxsocket::detach	释放对套接字的所有权
xxsocket::set_nonblocking	设置 socket 是否为非阻塞状态
xxsocket::test_nonblocking	检测 socket 是否是非阻塞状态（不可靠的）
xxsocket::bind	绑定套接字，指定地址
xxsocket::bind_any	绑定套接字, 任意地址
xxsocket::listen	启动 TCP 服务监听
xxsocket::accept	接受一个客户端连接，必须 listen 之后才可以调用此接口
xxsocket::accept_n	接受一个客户端连接，必须 listen 之后才可以调用此接口, 支持超时
xxsocket::connect	发起 TCP 连接, 和 pconnect,xpconnect 不同的是，套接字必须是已打开状态
xxsocket::connect_n	和 connect 相同，支持超时
xxsocket::send	发送数据
xxsocket::send_n	发送数据，支持超时
xxsocket::recv	从内核读取对端发来的数据
xxsocket:recv_n	从内核读取对端发来的数据，支持超时
xxsocket:sendto	发送 UDP 包到指定地址

下页

表 1 – 续上页

函数名	函数说明
xxsocket:recvfrom	从内核读取发送到已绑定本地端口的 UDP 套接字数据
xxsocket::handle_write_ready	等待 socket 可写，只有当发送缓冲区满了的时候，才会阻塞
xxsocket::handle_read_ready	等待 socket 可读唤醒
xxsocket::local_endpoint	获取本地地址，socket 必须已经成功 connect，才能正确返回
xxsocket::peer_endpoint	获取对端地址，socket 必须已经成功 connect，才能正确返回
xxsocket::set_keepalive	设置 TCP 协议心跳参数
xxsocket::reuse_address	设置 socket 是否重用地址/端口
xxsocket::set_optval	设置 socket 选项，函数模板封装 setsockopt，会自动计算大小
xxsocket::get_optval	获取 socket 选项，函数模板封装 getsockopt，会自动计算大小
xxsocket::select	封装 bsd socket.select，使用略有差异
xxsocket::alive	判断 socket 是否处于正常状态
xxsocket::shutdown	封装 bsd socket.shutdown
xxsocket::close	关闭套接字，回收套接字资源
xxsocket::get_last_errno	获取最近 socket 错误码
xxsocket::set_last_errno	设置最近 socket 错误码
xxsocket::strerror	获取 socket 错误码的文本描述信息
xxsocket::gai_strerror	获取 getaddrinfo 错误码的文本描述信息
xxsocket::resolve	静态方法，域名解析, 解析所有地址
xxsocket::resolve_v4	静态方法，域名解析, 解析 ipv4 地址
xxsocket::resolve_v6	静态方法，域名解析, 解析 ipv6 地址
xxsocket::resolve_v4to6	静态方法，域名解析, 只解析 ipv4 地址，并根据 V4MAPPED 规则转换为 ipv6 地址
xxsocket::resolve_tov6	静态方法，域名解析, 解析所有地址，但 ipv4 地址会根据 V4MAPPED 规则转换为 ipv6

6.1.3 yasio Interop

为了支持 Unity C#，yasio 提供了 C 语言接口导出，详见: https://github.com/yasio/yasio/blob/master/yasio/bindings/yasio_ni.cpp

可参考 xlua 集成案例: https://github.com/c4games/xLua

yasio NI API

dotnet API

```
// Integrate yasio to unity3d xlua, uncomment instead "yasio-ni", for integration detail,
↪ see https://github.com/c4games/xlua
// #if (UNITY_IPHONE || UNITY_WEBGL || UNITY_SWITCH) && !UNITY_EDITOR
//         public const string LIBNAME = "__Internal";
```

(下页继续)

(续上页)

```
// #else
//         public const string LIBNAME = "xlua";
// #endif
const string LIBNAME = "yasio-ni";

public delegate void YNIEventDelegate(uint emask, int cidx, IntPtr sid, IntPtr bytes,␣
↪int len);
public delegate int YNIResolvDelegate(string host, IntPtr sbuf);
public delegate void YNIPrintDelegate(string msg);

[DllImport(LIBNAME, CallingConvention = CallingConvention.Cdecl)]
public static extern void yasio_init_globals(YNIPrintDelegate fnPrint);

/// <summary>
/// Start a low level socket io service
/// </summary>
/// <param name="strParam">
/// format: "ip:port;ip:port;ip:port"
/// </param>
/// <param name="d"></param>
/// <returns></returns>
[DllImport(LIBNAME, CallingConvention = CallingConvention.Cdecl)]
public static extern void yasio_start(int channel_count, YNIEventDelegate d);

[DllImport(LIBNAME, CallingConvention = CallingConvention.Cdecl)]
public static extern void yasio_open(int cindex, int cmask);

[DllImport(LIBNAME, CallingConvention = CallingConvention.Cdecl)]
public static extern void yasio_close(int cindex);

[DllImport(LIBNAME, CallingConvention = CallingConvention.Cdecl)]
public static extern void yasio_close_handle(IntPtr sid);

[DllImport(LIBNAME, CallingConvention = CallingConvention.Cdecl)]
public static extern void yasio_write(IntPtr sid, byte[] bytes, int len);

[DllImport(LIBNAME, CallingConvention = CallingConvention.Cdecl)]
public static extern void yasio_dispatch(int count);

[DllImport(LIBNAME, CallingConvention = CallingConvention.Cdecl)]
```

(下页继续)

(续上页)

```
public static extern void yasio_set_option(int opt, string strParam);

[DllImport(LIBNAME, CallingConvention = CallingConvention.Cdecl)]
public static extern void yasio_set_resolv_fn(YNIResolvDelegate fnResolv);

[DllImport(LIBNAME, CallingConvention = CallingConvention.Cdecl)]
public static extern void yasio_set_print_fn(YNIPrintDelegate fnPrint);

[DllImport(LIBNAME, CallingConvention = CallingConvention.Cdecl)]
public static extern void yasio_stop();

[DllImport(LIBNAME, CallingConvention = CallingConvention.Cdecl)]
public static extern long yasio_highp_time();

[DllImport(LIBNAME, CallingConvention = CallingConvention.Cdecl)]
public static extern void yasio_memcpy(IntPtr destination, IntPtr source, uint length);

/// <summary>
/// The yasio constants
/// </summary>
public enum YEnums
{
    #region Channel mask enums, copy from yasio.hpp
    YCM_CLIENT = 1,
    YCM_SERVER = 1 << 1,
    YCM_TCP = 1 << 2,
    YCM_UDP = 1 << 3,
    YCM_KCP = 1 << 4,
    YCM_SSL = 1 << 5,
    YCK_TCP_CLIENT = YCM_TCP | YCM_CLIENT,
    YCK_TCP_SERVER = YCM_TCP | YCM_SERVER,
    YCK_UDP_CLIENT = YCM_UDP | YCM_CLIENT,
    YCK_UDP_SERVER = YCM_UDP | YCM_SERVER,
    YCK_KCP_CLIENT = YCM_KCP | YCM_CLIENT | YCM_UDP,
    YCK_KCP_SERVER = YCM_KCP | YCM_SERVER | YCM_UDP,
    YCK_SSL_CLIENT = YCM_SSL | YCM_CLIENT | YCM_TCP,
    #endregion

    #region Event kind enums, copy from yasio.hpp
    YEK_CONNECT_RESPONSE = 1,
```

(下页继续)

(续上页)

```
    YEK_CONNECTION_LOST,
    YEK_PACKET,
    #endregion

    #region Channel flags
    /* Whether setsockopt SO_REUSEADDR and SO_REUSEPORT */
    YCF_REUSEADDR = 1 << 9,

    /* For winsock security issue, see:
       https://docs.microsoft.com/en-us/windows/win32/winsock/using-so-reuseaddr-and-so-
↪exclusiveaddruse
    */
    YCF_EXCLUSIVEADDRUSE = 1 << 10,
    #endregion

    #region All supported options by native, copy from yasio.hpp
    // Set with deferred dispatch event, default is: 1
    // params: deferred_event:int(1)
    YOPT_S_DEFERRED_EVENT = 1,

    // Set custom resolve function, native C++ ONLY
    // params: func:resolv_fn_t*
    YOPT_S_RESOLV_FN,

    // Set custom print function, native C++ ONLY, you must ensure thread safe of it.
    // parmas: func:print_fn_t,
    YOPT_S_PRINT_FN,

    // Set custom print function
    // params: func:io_event_cb_t*
    YOPT_S_EVENT_CB,

    // Set tcp keepalive in seconds, probes is tries.
    // params: idle:int(7200), interal:int(75), probes:int(10)
    YOPT_S_TCP_KEEPALIVE,

    // Don't start a new thread to run event loop
    // value:int(0)
    YOPT_S_NO_NEW_THREAD,
```

(下页继续)

(续上页)

```
    // Sets ssl verification cert, if empty, don't verify
    // value:const char*
    YOPT_S_SSL_CACERT,

    // Set connect timeout in seconds
    // params: connect_timeout:int(10)
    YOPT_S_CONNECT_TIMEOUT,

    // Set dns cache timeout in seconds
    // params: dns_cache_timeout : int(600),
    YOPT_S_DNS_CACHE_TIMEOUT,

    // Set dns queries timeout in seconds, only works when have c-ares
    // params: dns_queries_timeout : int(10)
    YOPT_S_DNS_QUERIES_TIMEOUT,

    // Sets channel length field based frame decode function, native C++ ONLY
    // params: index:int, func:decode_len_fn_t*
    YOPT_C_LFBFD_FN = 101,

    // Sets channel length field based frame decode params
    // params:
    //     index:int,
    //     max_frame_length:int(10MBytes),
    //     length_field_offset:int(-1),
    //     length_field_length:int(4),
    //     length_adjustment:int(0),
    YOPT_C_LFBFD_PARAMS,

    // Sets channel length field based frame decode initial bytes to strip, default is 0
    // params:
    //     index:int,
    //     initial_bytes_to_strip:int(0)
    YOPT_C_LFBFD_IBTS,

    // Sets channel remote host
    // params: index:int, ip:const char*
    YOPT_C_REMOTE_HOST,

    // Sets channel remote port
```

(下页继续)

(续上页)

```
    // params: index:int, port:int
    YOPT_C_REMOTE_PORT,

    // Sets channel remote endpoint
    // params: index:int, ip:const char*, port:int
    YOPT_C_REMOTE_ENDPOINT,

    // Sets local host for client channel only
    // params: index:int, ip:const char*
    YOPT_C_LOCAL_HOST,

    // Sets local port for client channel only
    // params: index:int, port:int
    YOPT_C_LOCAL_PORT,

    // Sets local endpoint for client channel only
    // params: index:int, ip:const char*, port:int
    YOPT_C_LOCAL_ENDPOINT,

    // Sets channl flags
    // params: index:int, flagsToAdd:int, flagsToRemove:int
    YOPT_C_MOD_FLAGS,

    // Enable channel multicast mode
    // params: index:int, multi_addr:const char*, loopback:int
    YOPT_C_ENABLE_MCAST,

    // Disable channel multicast mode
    // params: index:int
    YOPT_C_DISABLE_MCAST,

    // Change 4-tuple association for io_transport_udp
    // params: transport:transport_handle_t
    // remark: only works for udp client transport
    YOPT_T_CONNECT,

    // Dissolve 4-tuple association for io_transport_udp
    // params: transport:transport_handle_t
    // remark: only works for udp client transport
    YOPT_T_DISCONNECT,
```

(下页继续)

(续上页)

```
    // Sets io_base sockopt
    // params: io_base*,level:int,optname:int,optval:int,optlen:int
    YOPT_B_SOCKOPT = 201,
    #endregion
};
```

6.1.4 yasio 宏定义

以下宏定义可以控制 yasio 的行为, 可以在 yasio/detail/config.hpp 更改或者在编译器预处理器定义

宏定义	说明
YASIO_HEADER_ONLY	是否以仅头文件的方式使用 yasio 核心组件，默认关闭
YASIO_VERBOSE_LOG	是否打印详细日志，默认关闭
YASIO_USE_SPSC_QUEUE	是否使用 SPSC(单生产者单消费者) 队列，仅当只有一个线程调用 io_service::write 时放可启用
YASIO_DISABLE_OBJECT_POOL	是否禁用对象池的使用，默认启用
YASIO_ENABLE_ARES_PROFILER	是否启用异步域名解析性能日志，默认关闭
YASIO_HAVE_CARES	是否启用 c-ares 异步域名解析库
YASIO_HAVE_KCP	是否启用 kcp 传输支持
YASIO_HAVE_SSL	是否启用 ssl 客户端支持
YASIO_DISABLE_CONCURRENT_QUEUE	是否禁用并发单利类模板