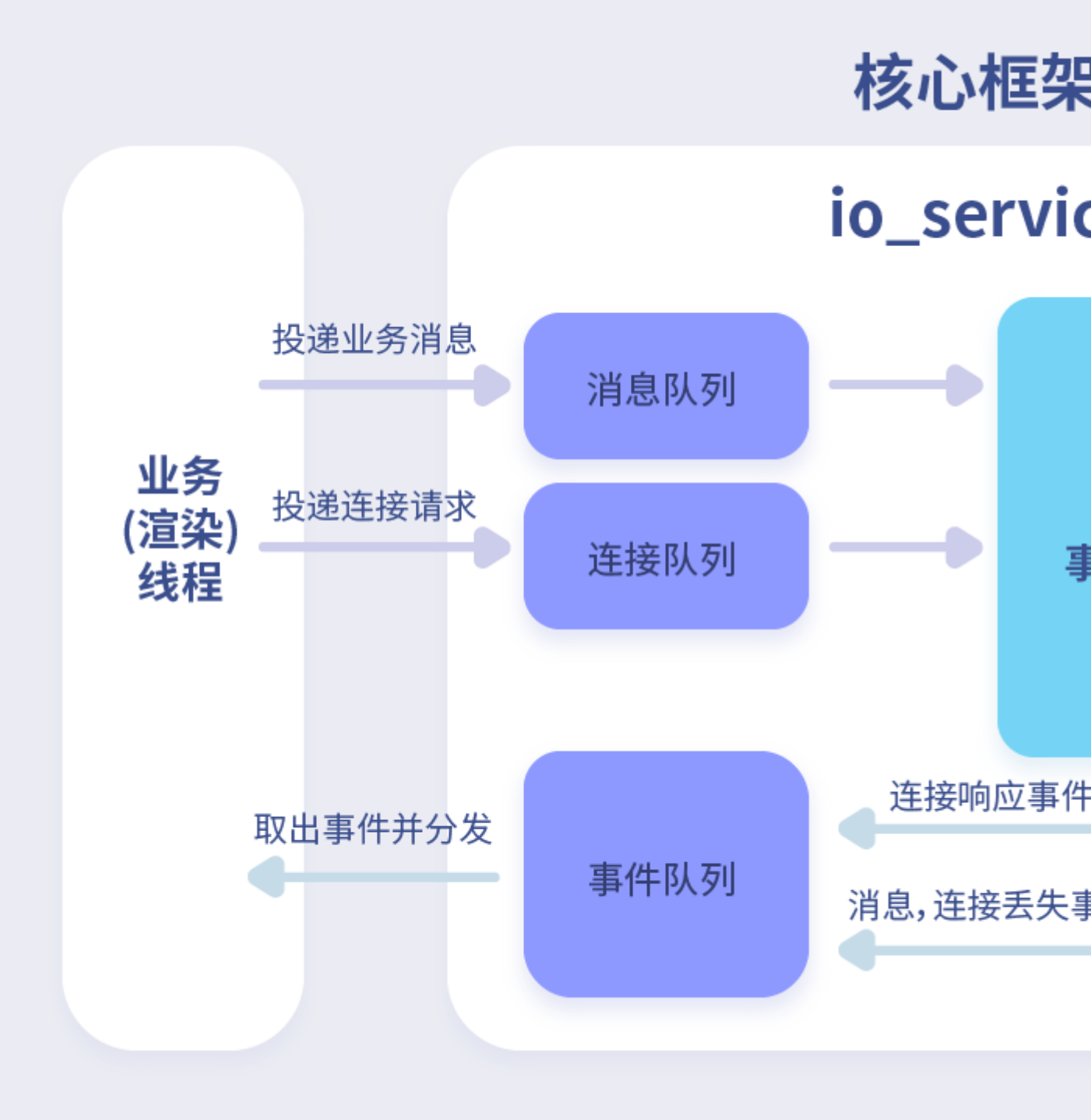
y a s i o 中文文档

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。

- 跨平台性:
  - 编译器:
    - Visual Studio 2013+
    - GCC4.7+
    - xcode9+
    - 其他支持 C++11,14,17 的编译器
  - 架构: x86, x64, ARM等。
  - 操作系统: Windows, macOS, Linux, FreeBSD, iOS, Android等。
- 术语:
  - 网络服务: `io_service`
  - 信道: `io_channel`
  - 传输会话: `io_transport`

- 框架图:



快速开始

此实例程序简单向 tool.chinaz .com发送http请求并打印响应数据。

C++

```
#include "  yasio/yasio.hpp"
#include "  yasio/obstream.hpp"
using namespace yasio;
int main()
{
    io_service service({"  tool.chinaz  .com",80}  );
    service.set_option(YOPT_S_DEFERRED_EVENT, 0); // dispatch network even
    service.start([&] (event_ptr&& ev) {
        switch (ev->kind())
        {
        case YEK_ON_PACKET: {
            auto packet = std::move(ev->packet());
            fwrite(packet.data(), packet.size(), 1, stdout);
            fflush(stdout);
            break;
        }
        case YEK_ON_OPEN:
            if (ev->status() == 0)
            {
            auto transport = ev->transport();
            if (ev->cindex() == 0)
            {
                obstream obs;
                obs.write_bytes("  GET /index.htm H TTP/1.1 r\"),n

                obs.write_bytes("  H ost: tool.chinaz  .com\r\"),n

                obs.write_bytes("  User-Agent: Moz  illa/5.0 (Windows NT 10.0; "
                            "  WOW64) AppleWebKit/537.36 (KH TML, like Gecko) "
                            "  Chrome/87.0.4820.88 Safari/537.36  r\"),n
                obs.write_bytes("  Accept: *  /*  ;q=0.8\"),n
                obs.write  bytes("  Connection: Close  r\  n\ "n\  n
```

```
local ip138 = "  tool.chinaz  .com"
local service = yasio.io_service.new({host=ip138, port=80}  )
local respdata = "  "
service:start(function(ev)
        local k = ev.kind()
        if (k == yasio.YEK_ON_PACKET) then
            respdata = respdata .. ev:packet():to_string()
        elseif k == yasio.YEK_ON_OPEN then
            if ev:status() == 0 then -- connect succeed
                local transport = ev:transport()
                local obs = yasio.obstream.new()
                obs:write_bytes("  GET / H TTP/1.1 r\"),n

                obs:write_bytes("  H ost: .. ip138 .. "\  r\"),n

                obs:write_bytes("  User-Agent: Moz  illa/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KH TML, like Gecko) Chrom
                obs:write_bytes("  Accept: *  /*  ;q=0.8\"),n
                obs:write_bytes("  Connection: Close  r\  n\ "n\  n

                service:write(transport, obs)
            end
        elseif k == yasio.YEK_ON_CLOSE then
            print("  request finish, respdata: ".."  respdata)
```

```
        end
    end)
-- Open channel 0 as tcp client and start non-blocking tcp 3 times handshake
service:open(0, yasio.YCK_TCP_CLIENT)

-- Call this function at thread which focus on the network event.
function gDispatchNetworkEvent(...)
    service:dispatch(128) -- dispatch max events is 128 per frame
end

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

测试 &示例

注意

运行Lua示例程序，会打印异常信息 `yasio - ibstream_view::consume out of range!`，这是示例程序里故意写的，请不必在意。

- 测试:
  - echo_server: TCP/UDP/KCP 回射服务器
  - echo_client: TCP/UDP/KCP 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
- 示例:
  - ftp_server: 基于yasio实现的仅支持下载的ftp服务器，点击 访问。
  - lua: Lua示例程序，包含简易的并发http请求，TCP拆包参数设置示例代码
  - xlua: xlua集成案例
  - yasio_unreal: yasio的虚幻引擎插件

编译 测试 & 示例

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

```
git clone https://github.com/yasio/yasio
cd yasio
git submodule update --init --recursive
cd build
```

```
# for xcode should be: cmake .. -GX code
cmake ..
cmake --build . --config Debug
```

io_service Class

充分利用 socket.select 多路复用模型实现网络服务，提供给上层统一的接口来进行 tcp, udp, kcp, ssl-client 通信。

语法

```
namespace yasio { inline namespace inet { class io_service; } }
```

成员

公共构造函数

Name	Description
io_service::io_service	构造1个 io_service 对象

公共方法

Name	Description
io_service::start	启动网络服务
io_service::stop	停止网络服务
io_service::is_running	判断网络服务是否运行中
io_service::is_stopping	判断网络服务是否停止中
io_service::open	打开信道
io_service::close	关闭传输会话
io_service::is_open	检测信道或会话是否打开
io_service::dispatch	分派网络事件

Name	Description
io_service::write	异步发送数据
io_service::write_to	异步发送DGRAM数据
io_service::schedule	注册定时器
io_service::init_globals	显示初始化全局数据
io_service::cleanup_globals	清理全局数据
io_service::channel_at	获取信道句柄
io_service::set_option	设置选项

注意

默认传输会话的创建会使用对象池 `obj ect_pool`

要求

头文件: yasio.hpp

i o _ s e r v i c e : : i o _ s e r v i c e

构造 `io_service` 对象。

```
io_service::io_service();

io_service::io_service(int channel_count);

io_service::io_service(const io_hostent& channel_ep);

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

参数

channel_count
信道数量。

channel_ep
信道远端地址。

channel_eps
信道远端地址数组首地址。

示例

```
#include "yasio/yasio.hpp"
int main() {
    using namespace yasio;
    io_service s1; // s1 only support 1 channel
    io_service s2(5); // s2 support 5 channels concurrency
    io_service s3(io_hostent{"github.com",443}); // s3 support 1 channel
    io_hostent hosts[] = {
        {"192.168.1.66",20336},
        {"192.168.1.88",20337},
    };
    io_service s4(hosts, YASIO_ARRAYSIZE(hosts)); // s4 support 2 channels concurrency
    return 0;
}
```

io_service::start

启动网络服务。

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

参数

cb
网络事件回调，默认情况下在 io_service::dispatch 调用者线程调度。

示例

```
#include "yasio/yasio.hpp"
int main() {
    using namespace yasio;
    auto service = yasio_shared_service(io_hostent{host="ip138.com",port=80});
    service->start([](event_ptr&& ev) {
    auto kind = ev->kind();
    if (kind == YEK_ON_OPEN)
    {
        if (ev->status() == 0)
        printf("[%d] connect succeed!",ev->cindex());
        else
        printf("[%d] connect failed!",ev->cindex());
```

```
    }
    } );
    return 0;
}
```

io_service::stop

停止网络服务。

```
void stop()
```

注意

- 当在非网络服务线程调用此函数时，会等待服务线程退出并进入 `IDLE` 状态，此状态下可以再次调用 `start` 重新启动服务。
- 当在网络服务自身线程调用了次函数时，会进入 `STOPPING` 状态，业务应该在非网络服务线程判断是否依然处于 `STOPPING` 状态来决定是否再次调用`stop`，例如:

```
if (service->is_stopping()) {
  service->stop();
}
```

io_service::is_running

判断网络服务是否在运行中。

```
bool is_running() const
```

返回值

`true`: 网络服务运行中， `false`: 网络服务处于其他状态

io_service::is_stopping

判断网络服务是否在停止中。

```
bool is_stopping() const
```

返回值

`true`: 网络服务停止中， `false`: 网络服务处于其他状态

io_service::open

打开信道。

```
bool open(size_t cindex, int kind);
```

参数

cindex
信道索引。

kind
信道类型。

返回值

`true`: 信道打开操作请求成功， `false`: 信道正在打开过程中。

注意

对于`TCP`, 将会请求内核发起非阻塞3次握手来建立可靠连接。

cindex 的值必须小于io_service初始化时的信道数量。

kind 必须是以下枚举值之一:

- `YCK_TCP_CLIENT`
- `YCK_TCP_SERVER`
- `YCK_UDP_CLIENT`
- `YCK_UDP_SERVER`
- `YCK_KCP_CLIENT`
- `YCK_KCP_SERVER`
- `YCK_SSL_CLIENT`

io_service::close

关闭信道或者传输会话。

```
void close(transport_handle_t transport);

void close(int cindex);
```

参数

transport
将要关闭的传输会话。

cindex
将要关闭的信道。

注意

对于`TCP`， 将会发起4次握手来终止连接。

io_service::is_open

判断信道或传输会话是否处于打开状态。

```
bool is_open(transport_handle_t transport) const;

bool is_open(int cindex) const;
```

参数

transport
传输会话句柄。

cindex
信道索引。

返回值

`true`: 打开， `false`: 未打开

io_service::dispatch

分派网络线程产生的事件。

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

参数

max_count
本次最大分派事件数。

注意

通常此方法应当在关心网络事件的业务逻辑线程调用, 例如Cocos2d-x的渲染线程，以及其他游戏引擎(Unity,UE4)的主逻辑线程。

此方法对于安全地更新游戏界面非常有用。

示例

```
yasio_shared_service()->dispatch(128);
```

io_service::write

向传输会话远端发送数据。

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

参数

thandle
传输会话句柄。

buffer
要发送的二进制缓冲区。

completion_handler
发送完成回调。

返回值

返回发送数据字节数, `< 0`: 说明发生错误。

注意

completion_handler 不支持 `KCP`。

空buffer会直接被忽略，也不会触发 completion_handler 。

io_service::write_to

向UDP传输会话发送数据。

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

参数

thandle
传输会话句柄。

buffer
要发送的buffer。

to
要发送的远端地址。

completion_handler
发送完成回调。

返回值

成功发送的字节数, 当 `< 0` 时说明发生错误，通常是传输会话已关闭(TCP连接断开等)。

注意

此函数仅可用于 DGRAM 传输会话，即 `UDP,KCP`。

发送完成回调 completion_handler 不支持 `KCP`。

空buffer会直接被忽略，也不会触发 completion_handler 。

io_service::schedule

注册一个定时器。

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

参数

duration
定时器超时时间，毫秒级。

cb
当定时器到期后回调。

返回值

`std::shared_ptr` 包装的定时器对象，以便用户对定时器安全地进行必要操作。

示例

```
// Register a once timer, timeout is 3 seconds.
yasio_shared_service()->schedule(std::chrono::seconds(3), [ ]->bool{
  printf("  time called!\n");
  return true;
} );

// Register a loop timer, interval is 5 seconds.
auto loopTimer = yasio_shared_service()->schedule(std::chrono::seconds(5), [ ]->bool{
  printf("  time called!\n");
  return false;
} );
```

i o _ s e r v i c e : : i n i t _ g l o b a l s

静态方法，显示地初始化全局数据

```
static void init_globals(print_fn2_t print_fn);
```

参数

print_fn
自定义网络日志打印函数。

注意

此函数是可选调用，但是当用户需要将网络日志重定向到自定义日志系统时，
则非常有用，例如重定向到UE4和U3D的日志输出。

示例

```
// yasio_uelua.cpp
// compile with: /EH sc
#include "  yasio_uelua.h"
#include "  yasio/platform/yasio_ue4.hpp"
#include "  lua.hpp"
#if defined(NS_SLUA)
using namespace NS_SLUA;
#endif
#include "  yasio/bindings/lyasio.cpp"

DECLARE_LOG_CATEGORY_EX TERN(yasio_ue4, Log, All);
DEFINE_LOG_CATEGORY(yasio_ue4);

void yasio_uelua_init(void* L)
{
  auto Ls            = (lua_State*)L;
  print_fn2_t log_cb = [  ](int level, const char* msg) {
    FString text(msg);
    const TCH AR* tstr = *text;
    UE_LOG(yasio_ue4, Log, L"  %s", tstr);
  }  ;
  io_service::init_globals(log_cb);

  luaregister_yasio(Ls);
}
void yasio_uelua_cleanup()
{
  io_service::cleanup_globals();
}
```

i o _ s e r v i c e : : c l e a n u p _ g l o b a l s

静态方法，显示地清理全局数据。

```
static void cleanup_globals();
```

注意

当用户需要卸载包含自定义日志打印回调的动态库(.dll,.so)前必须调用此函数，谨防应用程序闪退。

i o _ s e r v i c e : : c h a n n e l _ a t

通过信道索引获取信道句柄。

```
io_channel* channel_at(siz  e_tcindex) const;
```

参数

cindex
信道索引

返回值

信道句柄指针, 当索引值超出范围时，返回 `nullptr`。

io_service::set_option

设置选项。

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

参数

opt
选项枚举, 请查看 YOPT_X _X X. X

示例

```
#include "  yasio/yasio.hpp"

int main(){
    using namespace yasio;
    io_hostent hosts[  ]= {
    {"  192.168.1.66","20336"}  ,
    {"  192.168.1.88","20337"}  ,
    }  ;
    auto service = std::make_shared<io_service>(hosts, YASIO_ARRAYSIZ E(hosts));

    // for application protocol with length field, you j   ust needs set this option.
    // it's similar to j   ava netty length frame based decode.
    // such as when your protocol define as following
    //    packet.header: (header.len=12bytes)
    //          code:int16_t
    //          datalen:int32_t (not contains packet.header.len)
    //          timestamp:int32_t
    //          crc16:int16_t
    //    packet.data
    service->set_option(YOPT_C_LFBFD_PARAMS,
                        0,     // channelIndex, the channel index
                        65535, // maxFrameLength, max packet siz  e
                        2,     // lenghtFieldOffset, the offset of length field
                        4,     // lengthFieldLength, the siz  e of length field, can be 1,2,4
                        12    // lengthAdj   ustment:  if the value of length feild == packet.header.len + packet.data.len, this parame
    );
```

```
    // for application protocol without length field, j  ust sets length field siz  e to -1.
    // then io_service will dispatch any packet received from server immediately,
    // such as http request, this is default behavior of channel.
    service->set_option(YOPT_C_LFBFD_PARAMS, 1, 65535, -1, 0, 0);
    return 0;
}
```

请参阅

io_event Class

io_channel Class

io_service Options

xxsocket Class

obstream Class

ibstream_view Class

ibstream Class

io_channel Class

负责管理 TCP/SSL/UDP/KCP 连接和传输会话。

语法

```
namespace yasio { inline namespace inet { class io_channel; } }
```

成员

公共方法

Name	Description
io_channel::get_service	获取管理信道的io_service
io_channel::index	获取信道索引
io_channel::remote_port	获取信道远程端口
io_channel::bytes_transferred	获取传输字节数
io_channel::connect_id	获取连接ID

注意

当io_service对象构造后，最大信道数量不可改变，
信道句柄可通过 `io_service::channel_at` 获取。

io_channel::get_service

获取管理信道的io_service对象。

```
io_service& get_service()
```

io_channel::index

获取信道索引。

```
int index() const
```

io_channel::remote_port

获取信道远程端口.

```
u_short remote_port() const;
```

返回值

返回信道远程端口号

- 对于客户端信道表示通信的远端端口
- 对于服务端信道表示监听端口

io_channel::bytes_transferred

获取客户端接收总字节数

```
long long bytes_transferred() const;
```

返回值

从连接建立开始到当前，接收总字节数，用于统计客户端接收流量

io_channel::connect_id

获取客户端信道递增的实时连接id

```
unsigned int connect_id() const;
```

返回值

客户端信道当前连接id

请参阅

io_service Class

io_event Class

io_event Class

网络事件由 io_service 线程产生。

语法

```
namespace yasio { inline namespace inet { class io_event; } }
```

成员

公共方法

Name	Description
io_event::kind	获取事件类型
io_event::status	获取事件状态
io_event::passive	检查是否是被动事件
io_event::packet	获取事件消息包
io_event::timestamp	获取事件时间戳
io_event::transport	获取事件传输会话
io_event::transport_id	获取事件传输会话ID
io_event::transport_ud	设置或获取事件传输会话用户数据

io_event::kind

获取事件类型。

```
int kind() const;
```

返回值

事件类型，可以是以下值:

- YEK_ON_PACKET : 消息事件
- YEK_ON_OPEN : 打开事件，对于客户端信道，代表连接响应
- YEK_ON_CLOSE : 关闭事件，对于客户端信道，代表连接丢失

io_event::status

获取事件状态。

```
int status() const;
```

返回值

- 0: 正常
- 非0: 出错, 用户只需要简单打印即可。

io_event::passive

检查是否是被动事件

```
int passive() const
```

返回值

- 0: 非被动事件，包括客户端信道的连接打开、关闭事件和传输会话的消息事件。
- 1: open或close服务器信道时产生，仅当打开编译配置 YASIO_ENABLE_PASSIVE_EVENT 才会产生。
  且为保持兼容，此行为默认是关闭的。

io_event::packet

获取事件携带的消息包

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

返回值

消息包的引用, 用户可以使用 std::move 无GC方式从事件取走消息包。

io_event::timestamp

获取事件产生的微秒级时间戳。

```
highp_time_t timestamp() const;
```

返回值

和系统时间无关的微秒级时间戳。

io_event::transport

获取事件的传输会话句柄。

```
transport_handle_t transport() const;
```

返回值

返回句柄，当收到断开事件时，传输会话句柄已失效，仅可用作地址值比较。

io_event::transport_id

获取事件的传输会话ID。

```
unsigned int transport_id() const;
```

返回值

32位无符号整数范围内的唯一ID。

io_event::transport_ud

设置或获取传输会话用户数据。

```
template<typename _Uty>
_Uty io_event::transport_ud();

template<typename _Uty>
void io_event::transport_ud(_Uty uservalue);
```

注意

用户需要自己管理 userdata 的内存, 例如:

- 收到连接建立成功事件时存储userdata
- 收到连接丢失事件时清理userdata

请参阅

io_service Class

io_channel Class

obstream Class

提供二进制序列化功能。

注意

- `yasio::obstream` 等价于 `yasio::obstream_any< yasio::dynamic_extent>`
- 自3.39.0起，新增 `obstream_span` 可序列化到 `std::string` 或 `std::vector< char>`，请查看示例序列化到stl容器
- 自3.37.5起，新增 `obstream_any` 类模板具备序列化已知大小的小块栈内存能力，请查看示例: 使用栈空间序列化
- 自3.35.0起，优化为类模板basic_obstream实现，体现了C++模板强大的代码复用能力。
- `obstream` 当写入int16~int64和float/double类型时, 会自动将主机字节序转换为网络字节序。
- `fast_obstream` 不会转换任何字节序。

语法

```
namespace yasio {
template <siz e_t Extent>
using obstream_any = basic_obstream<convert_traits<network_convert_tag>, _Extent>;
using obstream     = obstream_any<dynamic_extent>;

template <siz e_t Extent>
using fast_obstream_any = basic_obstream<convert_traits<host_convert_tag>, _Extent>;
using fast_obstream     = fast_obstream_any<dynamic_extent>;
}

template <typename _Cont>
using obstream_span = basic_obstream_span<convert_traits<network_convert_tag>, _Cont>;
template <typename _Cont>
using fast_obstream_span = basic_obstream_span<convert_traits<host_convert_tag>, _Cont>;
```

成员

公共构造函数

Name	Description
obstream::obstream	构造1个 `obstream` 对象

公共方法

Name	Description
obstream::write	函数模板，写入数值
obstream::write_ix	函数模板，写入(7 bit Encoded Int)数值
obstream::write_v	写入带长度域(7 bit Encoded)的二进制数据
obstream::write_byte	写入1个字节
obstream::write_bytes	写入指定长度二进制数据
obstream::empty	检查流是否为空
obstream::data	获取流数据指针
obstream::length	获取流数据大小
obstream::buffer	获取流内部缓冲区
obstream::clear	清理流，以便复用
obstream::shrink_to_fit	释放流内部缓冲区多余内存
obstream::save	保存流二进制数据到文件系统

要求

头文件: obstream.hpp

obstream::obstream

构造 obstream 对象。

```
obstream(size_t capacity = 128);
obstream(const obstream& rhs);
obstream(obstream&& rhs);
```

obstream::write

写入数值类型

```
template<typename _Nty>
void obstream::write(_Nty value);
```

参数

value
要写入的值

注意

_Nty 实际类型可以是任意1~8字节整数类型或浮点类型。

obstream::write_ix

将32/64位整数值以7Bit Encoded Int方式压缩后写入流。

```
template<typename _Intty>
void obstream::write_ix(_Intty value);
```

参数

value
要写入的值。

注意

_Intty 类型只能是如下类型

- int32_t
- int64_t

此函数压缩编码方式兼容微软dotnet如下函数

- BinaryWriter.Write7BitEncodedInt
- BinaryWriter.Write7BitEncodedInt64

obstream::write_v

写入二进制数据，包含长度字段(7Bit Encoded Int).

```
void write_v(cxx17::string_view sv);
```

参数

sv
要写入的数据。

注意

此函数先以7Bit Encoded编码方式写入数据长度, 再调用 write_bytes 写入字节数据.

obstream::write_byte

写入1个字节。

```
void write_byte(uint8_t value);
```

参数

value
要写入的值。

注意

此函数功能等价于 `obstream::write< uint8_t>`

obstream::write_bytes

写入字节数组。

```
void write_bytes(cxx17::string_view sv);

void write_bytes(const void* data, int length);

void write_bytes(std::streamoff offset, const void* data, int length);
```

参数

sv
写入string_view包装的字节数组.

data
要写入字节数组的首地址.

length
要写入字节数组的长度.

offset
要写入字节数组的目标偏移.

注意

`offset + length` 的值必须小于 `obstream::length`

obstream::empty

判断流是否为空。

```
bool empty() const;
```

返回值

`true` 空; `false` 非空。

注意

此函数等价于 length == 0。

obstream::data

获取数据指针。

```
const char* data() const;

char* data();
```

返回值

字节流数据首地址。

obstream::length

获取流长度。

```
size_t length() const;
```

返回值

返回流中包含的总字节数。

obstream::buffer

获取流内部缓冲区。

```
const std::vector<char> &buffer() const;

std::vector<char> &buffer();
```

返回值

流内部缓冲区引用，可以使用 `std::move` 取走。

示例

```
// obstream_buffer.cpp
// compile with: /EH sc
#include "yasio/obstream.hpp"

int main( )
{
  using namespace yasio;
  using namespace cxx17;

  obstream obs;
  obs.write_v("hello world!");

  const auto& const_buffer = obs.buffer();

  // after this line, the obs will be empty
  auto move_buffer = std::move(obs.buffer());

  return 0;
}
```

obstream::clear

清理流，以便复用。

```
void clear();
```

注意

此函数不会释放buffer内存，对于高效地复用序列化器非常有用。

obstream::shrink_to_fit

释放流内部缓冲区多余内存。

```
void shrink_to_fit();
```

obstream::save

将流中的二进制字节数据保存到文件。

```
void save(const char* filename) const;
```

示例

```
// obstream_save.cpp
// compile with: /EH sc
#include "  yasio/obstream.hpp"
#include "  yasio/ibstream.hpp"

int main( )
{
   using namespace yasio;
   using namespace cxx17;

   obstream obs;
   obs.write_v("  hello world!); "
   obs.save("  obstream_save.bin)";

   ibstream ibs;
   if(ibs.load("  obstream_save.bin)") {
      // output should be: hello world!
      try {
         std::count < <ibs.read_v() < <"\  n";
      }
      catch(const std::exception& ex) {
         std::count < <"  read_v fail: "< <
```

```
            < <ex.message() < <"\ n";
        }
    }

    return 0;
}
```

使用栈内存序列化示例

注意事项

- 序列化过程中，当 `fixed_buffer` 检测到内存空间不足时会抛出 `std::out_of_range` 异常

obstream_any用法

```
#include " yasio/obstream.hpp"

int main() {
    yasio::obstream_any<128> obs; // 使用栈空间, 注意不要太大，防止栈空间溢出
    auto where = obs.push<uint16_t>();
    obs.write(3.141592654);
    obs.write(1.17723f);
    obs.write_ix<int32_t>(20201125);
    obs.write_ix<int64_t>(-9223372036854775807);
    obs.pop<uint16_t>(where);
    return 0;
}
```

obstream_span + std::array用法

```
#include " yasio/obstream.hpp"

int main() {
    std::array<char, 128> fb; // 使用栈空间, 注意不要太大，防止栈空间溢出
    yasio::obstream_span<yasio::fixed_buffer_span> obs(fb);
    auto where = obs.push<uint16_t>();
    obs.write(3.141592654);
    obs.write(1.17723f);
    obs.write_ix<int32_t>(20201125);
    obs.write_ix<int64_t>(-9223372036854775807);
    obs.pop<uint16_t>(where);
    return 0;
}
```

obstream_span + char[ ] 用法

```
#include " yasio/obstream.hpp"
```

```
int main() {
    char raw_fb[128] // 使用栈空间, 注意不要太大，防止栈空间溢出
    yasio::obstream_span<yasio::fixed_buffer_span> obs(raw_fb);
    auto where = obs.push<uint16_t>();
    obs.write(3.141592654);
    obs.write(1.17723f);
    obs.write_ix<int32_t>(20201125);
    obs.write_ix<int64_t>(-9223372036854775807);
    obs.pop<uint16_t>(where);
    return 0;
}
```

序列化到std::vector和std::string

obstream_span + std::vector用法

```
#include "yasio/obstream.hpp"

int main() {
    std::vector<char> buf;
    yasio::obstream_span<std::vector<char> >obs(buf);
    auto where = obs.push<uint16_t>();
    obs.write(3.141592654);
    obs.write(1.17723f);
    obs.write_ix<int32_t>(20201125);
    obs.write_ix<int64_t>(-9223372036854775807);
    obs.pop<uint16_t>(where);
    return 0;
}
```

obstream_span + std::string用法

```
#include "yasio/obstream.hpp"

int main() {
    std::string buf;
    yasio::obstream_span<std::string> obs(buf);
    auto where = obs.push<uint16_t>();
    obs.write(3.141592654);
    obs.write(1.17723f);
    obs.write_ix<int32_t>(20201125);
    obs.write_ix<int64_t>(-9223372036854775807);
    obs.pop<uint16_t>(where);
    return 0;
}
```

请参阅

ibstream_view Class

ibstream_view Class

提供二进制反序列化功能。

> 注意
>
> 在反序列化过程中，当剩余数据不足时会抛出 `std::out_of_range` 异常。

语法

```
namespace yasio {
// 反序列化过程，会自动转换字节序，适用于网络传输
using ibstream_view = basic_ibstream_view<endian::network_convert_tag>;

// 反序列化过程，无字节序转换，性能更快
using fast_ibstream_view = basic_ibstream_view<endian::host_convert_tag>;
}
```

成员

公共构造

Name	Description
ibstream_view::ibstream_view	构造1个 `ibstream_view` 对象

公共方法

Name	Description
ibstream_view::reset	重置待反序列化数据
ibstream_view::read	函数模板，读取数值
ibstream_view:read_ix	函数模板，读取(7 bit Encoded In)整数值int64
ibstream_view:read_v	读取带长度域(7 bit Encoded In)的二进制数据

Name	Description
ibstream_view:read_byte	读取1个字节
ibstream_view:read_bytes	读取指定长度二进制数据
ibstream_view::empty	检查流是否为空
ibstream_view::data	获取流数据指针
ibstream_view::length	获取流大小
ibstream_view::advance	向前移动流的读取游标
ibstream_view::seek	移动流的读取游标

注意

`ibstream_view` 借鉴C++17标准的 `std::string_view`, 意味着在初始化和反序列化过程中不会产生任何GC。

要求

头文件: ibstream.hpp

ibstream_view::ibstream_view

构造一个 `ibstream_view` 对象。

```
ibstream_view();

ibstream_view(const void* data, size_t size);

ibstream_view(const obstream* obs);
```

参数

data
待反序列化二进制数据首地址。

siz e
待反序列化二进制数据大小。

obs
已序列化的流。

ibstream_view::reset

重置 ibstream_view 缓冲视图。

```
void ibstream_view::reset(const void* data, siz e_tsiz e);
```

参数

data
待反序列化二进制数据首地址。

siz e
待反序列化二进制数据大小。

ibstream_view::read

从流中读取数值。

```
template<typename _Nty>
_Nty ibstream_view::read();
```

返回值

返回读到的值

注意

_Nty 实际类型可以是任意1~8字节整数类型或浮点类型。

ibstream_view::read_ix

读取7Bit Encoded Int压缩编码的整数值。

```
template<typename _Intty>
_Intty ibstream_view::read_ix();
```

返回值

32/64位整数值。

注意

_Intty 的必须是以下类型

- int32_t
- int64_t

本函数兼容于 **Microsof t d otnet**下函数

- BinaryReader.Read7BitEncodedInt()
- BinaryReader.Read7BitEncodedInt64()

i b s t r e a m_ v i e w: : r e a d _ v

读取变长二进制数据。

```
cxx17::string_view read_v();
```

返回值

返回读取到的二进制数据视图，无GC。

注意

本函数会先读取7bit Encoded Int压缩编码的长度值，再调用 read_bytes 读取二进制字节数据。

i b s t r e a m_ v i e w: : r e a d _ b y t e

读取1个字节。

```
uint8_t read_byte();
```

返回值

uint8_t值。

注意

本函数等价于 `ibstream_view::read< uint8_t>`

ibstream_view::read_bytes

读取指定长度字节数据，无GC。

```
cxx17::string_view read_bytes();
```

返回值

二进制数据的 `cxx17::string_view` 类型视图。

ibstream_view::empty

判断流是否为空。

```
bool empty() const;
```

返回值

`true` 空; `false` 流中至少包含1个字节。

注意

此方法等价于 length == 0。

ibstream_view::data

返回流数据指针。

```
const char* data() const;
```

返回值

返回指向流中第一个字节的指针。

ibstream_view::length

获取流长度。

```
size_t length() const;
```

返回值

当前流长度。

ibstream_view::advance

向前移动流读取游标。

```
void advance(ptrdiff_t offset);
```

参数

offset
要向前移动的偏移量。

注意

若*offset*传负数，则反向可移动读取游标。

ibstream_view::seek

移动读取游标。

```
ptrdiff_t seek(ptrdiff_t offset, int whence);
```

参数

offset
和*whence*相关的偏移量。

whence
含义等同于C标准库枚举值：`SEEK_SET`, `SEEK_CUR`, `SEEK_END`。

返回值

返回移动后相对于流首字节偏移。

ibstream Class

提供二进制数据加载和反序列化功能。

语法

```
namespace yasio {
using ibstream = basic_ibstream<endian::network_convert_tag>;
// 反序列化过程，无字节序转换，性能更快
using fast_ibstream = basic_ibstream<endian::host_convert_tag>;
}
```

成员

公共构造函数

Name	Description
ibstream::ibstream	构造1个 ibstream 对象

公共方法

Name	Description
ibstream::load	从文件加载流

继承层次结构

ibstream_view

ibstream

ibstream::ibstream

构造一个 ibstream 对象。

```
ibstream(std::vector<char> blob);

ibstream(const obstream* obs);
```

参数

blob
输入二进制流。

obs
已序列化的 `obstream` 对象。

ibstream::load

从文件加载.

```
bool load(const char* filename) const;
```

返回值

`true` 加载成功， `false` 加载失败。

示例

请查看 obstream::save

请参阅

obstream Class

io_service Class

xxsocket Class

封装底层bsd socket常用API，屏蔽各操作系统平台差异，yasio的起源。

特别注意

xxsocket除了 `accept_n` 以外的所有 `xxx_n` 接口均会将当前socket对象底层描述符设置为非阻塞模式，且不会恢复。

语法

```
namespace yasio { inline namespace inet { class xxsocket; } }
```

成员

Name	Description
xxsocket::xxsocket	构造一个 `xxsocket` 对象

公共方法

Name	Description
xxsocket::xpconnect	建立TCP连接
xxsocket::xpconnect_n	非阻塞方式建立TCP连接
xxsocket::pconnect	建立TCP连接
xxsocket::pconnect_n	非阻塞方式建立TCP连接
xxsocket::pserve	创建tcp服务端
xxsocket::swap	交换socket描述符句柄

Name	Description
xxsocket::open	打开socket
xxsocket::reopen	重新打开socket
xxsocket::is_open	检查socket是否打开
xxsocket::native_handle	获取socket句柄
xxsocket::release_handle	释放socket句柄控制权
xxsocket::set_nonblocking	将socket设置为非阻塞模式
xxsocket::test_nonblocking	检测socket是否为非阻塞模式
xxsocket::bind	绑定指定本地指定网卡地址
xxsocket::bind_any	绑定本地任意地址
xxsocket::listen	开始TCP监听
xxsocket::accept	接受一个TCP客户端连接
xxsocket::accept_n	非阻塞方式接受TCP连接
xxsocket::connect	建立连接
xxsocket::connect_n	非阻塞方式建立连接
xxsocket::disconnect	断开UDP和远程地址的绑定
xxsocket::send	发送数据
xxsocket::send_n	非阻塞方式发送数据
xxsocket::recv	接受数据
xxsocket::recv_n	非阻塞方式接受数据

Name	Description
xxsocket::sendto	发送DGRAM数据到指定地址
xxsocket::recvfrom	接受DGRAM数据
xxsocket::handle_write_ready	等待socket可写
xxsocket::handle_read_ready	等待socket可读
xxsocket::local_endpoint	获取socket本地地址
xxsocket::peer_endpoint	获取socket远端地址
xxsocket::set_keepalive	设置tcp keepalive
xxsocket::reuse_address	设置socket是否重用地址
xxsocket::exclusive_address	设置socket是否阻止地址重用
xxsocket::select	监听socket事件
xxsocket::shutdown	停止socket收发
xxsocket::close	关闭socket
xxsocket::tcp_rtt	获取tcp rtt.
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	将socket错误码转换为字符串
xxsocket::gai_strerror	将getaddrinfo返回值转换为字符串
xxsocket::resolve	解析域名
xxsocket::resolve_v4	解析域名包含的ipv4地址

Name	Description
xxsocket::resolve_v6	解析域名包含的ipv6地址
xxsocket::resolve_v4to6	解析域名包含的ipv4地址并转换为ipv6的V4MAPPED格式
xxsocket::resolve_tov6	解析域名包含的所有地址，ipv4地址会转换为ipv6的V4MAPPED格式
xxsocket::getipsv	获取本机支持的ip协议栈版本标志位
xxsocket::traverse_local_address	枚举本机地址

xxsocket::xxsocket

构造 `xxsocket` 对象。

```
xxsocket::xxsocket();
xxsocket::xxsocket(socket_native_type handle);
xxsocket::xxsocket(xxsocket&&right);
xxsocket::xxsocket(int af, int type, int protocol);
```

参数

handle
通过已有socket句柄构造 `xxsocket` 对象。

right
move构造函数右值引用。

af
ip协议地址类型。

protocol
协议类型，对于TCP/UDP直接传 `0` 就可以。

xxsocket::xpconnect

和远程服务器建立TCP连接。

```
int xxsocket::xpconnect(const char* hostname, u_short port, u_short local_port = 0);
```

参数

hostname
要连接服务器主机名，可以是 `IP地址` 或 `域名`。

port
要连接服务器的端口。

local_port
本地通信端口号，默认值 `0` 表示随机分配。

注意

会自动检测本机支持的ip协议栈版本。

返回值

`0`：成功， `< 0`失败，通过 `xxsocket::get_last_errno` 获取错误码。

xxsocket::xpconnect_n

和远程服务器建立TCP连接。

```
int xxsocket::xpconnect_n(const char* hostname, u_short port, const std::chrono::microseconds& wtimeout, u_short local_p
```

参数

hostname
要连接服务器主机名，可以是 `IP地址` 或 `域名`。

port
要连接服务器的端口。

local_port
本地通信端口号，默认值 `0` 表示随机分配。

wtimeout
建立连接超时时间。

返回值

`0`：成功， `< 0`失败，通过 `xxsocket::get_last_errno` 获取错误码。

注意

会自动检测本机支持的ip协议栈。

xxsocket::pconnect

和远程服务器建立TCP连接。

```
int xxsocket::pconnect(const char* hostname, u_short port, u_short local_port = 0);
int xxsocket::pconnect(const endpoint& ep, u_short local_port = 0);
```

参数

hostname
要连接服务器主机名，可以是 `IP地址` 或 `域名`。

ep
要连接服务器的地址。

port
要连接服务器的端口。

local_port
本地通信端口号，默认值 `0` 表示随机分配。

返回值

`0`：成功， `< 0`失败，通过 `xxsocket::get_last_errno` 获取错误码。

注意

不会检测本机支持的ip协议栈。

xxsocket::pconnect_n

和远程服务器建立TCP连接。

```
int pconnect_n(const char* hostname, u_short port, const std::chrono::microseconds& wtimeout, u_short local_port = 0);
int pconnect_n(const char* hostname, u_short port, u_short local_port = 0);
int pconnect_n(const endpoint& ep, const std::chrono::microseconds& wtimeout, u_short local_port = 0);
int pconnect_n(const endpoint& ep, u_short local_port = 0);
```

参数

hostname
要连接服务器主机名，可以是 `IP地址` 或 `域名`。

ep
要连接服务器的地址。

port
要连接服务器的端口。

local_port
本地通信端口号，默认值 `0` 表示随机分配。

wtimeout
建立连接超时时间。

返回值

`0`：成功， `< 0`失败，通过 `xxsocket::get_last_errno` 获取错误码。

xxsocket::pserve

开启本地TCP服务监听。

```
int pserve(const char* addr, u_short port);
int pserve(const endpoint& ep);
```

参数

addr
本机指定网卡 `IP地址`。

ep
本机地址。

port
TCP监听端口。

返回值

`0`：成功， `< 0`失败，通过 `xxsocket::get_last_errno` 获取错误码。

示例

```
// xxsocket-serve.cpp
#include < signal.h>
#include < vector>
#include "  yasio/xxsocket.hpp"
using namespace yasio;

xxsocket g_server;
static bool g_stopped = false;
void process_exit(int sig)
{
  if (sig == SIGINT)
  {
    g_stopped = true;
    g_server.close();
  }
  printf("  exit");
}
int main()
{
  signal(SIGINT, process_exit);

  if (g_server.pserve("  0.0.0.0","1219) !  =0)
    return -1;
  const char reply_msg[  ]= "  hi, I'm server  n";
  do
  {
    xxsocket cs = g_server.accept();
    if (cs.is_open())
    {
      cs.send(reply_msg, siz  eof(reply_msg) - 1);
      std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
  } while (!g_stopped);

  return 0;
}
```

x x s o c k e t : : s w a p

交换底层socket句柄。

```
xxsocket& swap(xxsocket& who);
```

参数

who
交换对象。

返回值

`xxsocket` 左值对象的引用。

xxsocket::open

打开一个socket。

```
bool open(int af = AF_INET, int type = SOCK_STREAM, int protocol = 0);
```

参数

af
地址类型，例如 `AF_INET` (ipv4)， `AF_INET6` (ipv6)。

type
socket类型， `SOCK_STREAM` (TCP), `SOCK_DGRAM` (UDP)。

protocol
协议，对于TCP/UDP，传 `0` 即可。

返回值

`true` : 成功， `false` 失败，通过 `xxsocket::get_last_errno` 获取错误码。

xxsocket::reopen

打开一个socket。

```
bool reopen(int af = AF_INET, int type = SOCK_STREAM, int protocol = 0);
```

参数

af
地址类型，例如 `AF_INET` (ipv4)， `AF_INET6` (ipv6)。

type
socket类型， `SOCK_STREAM` (TCP), `SOCK_DGRAM` (UDP)。

protocol
协议，对于TCP/UDP，传 `0` 即可。

返回值

`true` : 成功， `false` 失败，通过 `xxsocket::get_last_errno` 获取错误码。

注意

如果socket已打开，此函数会先关闭，再重新打开。

xxsocket::is_open

判断socket是否已打开。

```
bool is_open() const;
```

返回值

`true` : 已打开， `false` : 未打开。

xxsocket::native_handle

获取socket文件描述符。

```
socket_native_type native_handle() const;
```

返回值

socket文件描述符， `yasio::inet::invalid_socket` 表示无效socket。

xxsocket::release_handle

释放底层socket描述符控制权。

```
socket_native_type release_handle() const;
```

返回值

释放前的socket文件描述符

xxsocket::set_nonblocking

设置socket的非阻塞模式。

```
int set_nonblocking(bool nonblocking) const;
```

参数

nonblocking
`true`：非阻塞模式，`false`：阻塞模式。

返回值

`0`：成功，`< 0`失败，通过 `xxsocket::get_last_errno` 获取错误码。

xxsocket::test_nonblocking

检测socket是否为非阻塞模式。

```
int test_nonblocking() const;
```

返回值

`1`：非阻塞模式，`0`：阻塞模式。

注意

对于winsock2，未连接的 `SOCK_STREAM` 类型socket会返回 `-1`。

xxsocket::bind

绑定socket本机地址。

```
int bind(const char* addr, unsigned short port) const;
int bind(const endpoint& ep) const;
```

参数

addr
本机指定网卡ip地址。

port
要绑定的端口。

ep
要绑定的本机地址。

返回值

`0`：成功， `< 0`失败，通过 `xxsocket::get_last_errno` 获取错误码。

xxsocket::bind_any

绑定socket本机任意地址。

```
int bind_any(bool ipv6) const;
```

参数

ipv6
是否绑定本机任意IPv6地址

返回值

`0`：成功， `< 0`失败，通过 `xxsocket::get_last_errno` 获取错误码。

xxsocket::listen

开始监听来自TCP客户端的握手请求。

```
int listen(int backlog = SOMAX_CONN) const;
```

参数

backlog
最大监听数。

返回值

`0`：成功， `< 0`失败，通过 `xxsocket::get_last_errno` 获取错误码。

xxsocket::accept

接受一个客户端连接。

```
xxsocket accept() const;
```

返回值

和客户端通信的 `xxsocket` 对象。

xxsocket::accept_n

非阻塞方式接受一个客户端连接。

```
int accept_n(socket_native_type& new_sock) const;
```

参数

new_sock
输出参数，和客户端通信的底层socket句柄引用

返回值

`0`：成功， `< 0`失败，通过 `xxsocket::get_last_errno` 获取错误码。

注意

如果此函数返回 `0`，new_sock 会被设置为非阻塞模式。

调用此函数之前，请手动调用 `xxsocket::set_nonblocking` 将socket设置为非阻塞模式。

xxsocket::connect

建立连接。

```
int connect(const char* addr, u_short port);
int connect(const endpoint& ep);
```

参数

addr
远程主机ip地址。

port
远程主机端口。

ep
远程主机地址。

返回值

`0`：成功， `< 0`失败，通过 `xxsocket::get_last_errno` 获取错误码。

注意

TCP: 发起TCP三次握手

UDP: 建立4元组绑定

xxsocket::connect_n

建立连接。

```
int connect_n(const char* addr, u_short port, const std::chrono::microseconds& wtimeout);
int connect_n(const endpoint& ep, const std::chrono::microseconds& wtimeout);
int connect_n(const endpoint& ep);
```

参数

addr
远程主机ip地址。

port
远程主机端口。

ep
远程主机地址。

wtimeout
建立连接的超时时间。

返回值

`0`：成功， `< 0`失败，通过 `xxsocket::get_last_errno` 获取错误码。

注意

TCP: 发起TCP三次握手

UDP: 建立4元组绑定

xxsocket::disconnect

解除socket和远程主机的4元组绑定。

```
int disconnect() const;
```

返回值

`0`：成功， `< 0`失败，通过 `xxsocket::get_last_errno` 获取错误码。

注意

仅支持UDP

返回值

`0`：成功， `< 0`失败，通过 `xxsocket::get_last_errno` 获取错误码。

注意

只用于 `SOCK_DGRAM` (UDP) 类型socket。

xxsocket::send

向远端发送指定长度数据。

```
int send(const void* buf, int len, int flags = 0);
```

参数

buf
要发送数据的起始字节地址。

len
要发送数据的长度。

flags
发送数据底层标记。

返回值

`==len`：成功， `< len`失败，通过 `xxsocket::get_last_errno` 获取错误码。

xxsocket::send_n

在超时时间内尽力向远端发送指定长度数据。

```
int send_n(const void* buf, int len, const std::chrono::microseconds& wtimeout, int flags = 0);
```

参数

buf
要发送数据的起始字节地址。

len
要发送数据的长度。

wtimeout
发送超时时间。

flags
发送数据底层标记。

返回值

`==len`：成功， `< len`失败，通过 `xxsocket::get_last_errno` 获取错误码。

xxsocket::recv

从内核去除远程主机发送过来的数据。

```
int recv(void* buf, int len, int flags = 0) const;
```

参数

buf
接收数据缓冲区。

len
接收数据缓冲区长度。

flags
接收数据底层标记。

返回值

`==len`：成功， `< len`失败，通过 `xxsocket::get_last_errno` 获取错误码。

注意

此函数是否立即返回，取决于socket本身是否是 `非阻塞模式`。

xxsocket::recv_n

在超时时间内尽力从内核取出指定长度数据。

```
int recv_n(void* buf, int len, const std::chrono::microseconds& wtimeout, int flags = 0) const;
```

参数

buf
接收数据缓冲区。

len
接收数据缓冲区长度。

wtimeout
接收超时时间。

flags
接收数据底层标记。

返回值

`==len`：成功， `< len`失败，通过 `xxsocket::get_last_errno` 获取错误码。

xxsocket::sendto

向远程主机发送 `DGRAM` （UDP）数据。

```
int sendto(const void* buf, int len, const endpoint& to, int flags = 0) const;
```

参数

buf
待发送数据缓冲区。

len
待发送数据缓冲区长度。

to
发送目标地址。

flags
发送数据底层标记。

返回值

`==len`：成功， `< len`失败，通过 `xxsocket::get_last_errno` 获取错误码。

xxsocket::recvfrom

从内核去除远程主机发送过来的数据。

```
int recvfrom(void* buf, int len, endpoint& peer, int flags = 0) const;
```

参数

buf
接收数据缓冲区。

len
接收数据缓冲区长度。

peer
接收数据来源，输出参数。

flags
接收数据底层标记。

返回值

`==len` : 成功， `< len`失败，通过 `xxsocket::get_last_errno` 获取错误码。

注意

此函数是否立即返回，取决于socket本身是否是 `非阻塞模式` 。

xxsocket::handle_write_ready

等待socket可写。

```
int handle_write_ready(const std::chrono::microseconds& wtimeout) const;
```

参数

wtimeout
等待超时时间。

返回值

`0` : 超时， `1` : 成功， `< 0`: 失败，通过 `xxsocket::get_last_errno` 获取错误码。

注意

通常当内核发送缓冲区没满的情况下，此函数会立即返回。

xxsocket::handle_read_ready

等待socket可读。

```
int handle_read_ready(const std::chrono::microseconds& wtimeout) const;
```

参数

wtimeout
等待超时时间。

返回值

`0`: 超时， `1`: 成功， `< 0`: 失败，通过 `xxsocket::get_last_errno` 获取错误码。

xxsocket::local_endpoint

获取4元组通信的本地地址。

```
endpoint local_endpoint() const;
```

返回值

返回本地地址。

注意

如果没有调用过 `xxsocket::connect` 或者TCP连接3次握手未完成，那么返回的地址是 `0.0.0.0`

xxsocket::peer_endpoint

获取4元组通信的对端地址。

```
endpoint peer_endpoint() const;
```

返回值

返回本地地址。

注意

如果没有调用过 `xxsocket::connect` 或者TCP连接3次握手未完成，那么返回的地址是 `0.0.0.0`

xxsocket::set_keepalive

设置TCP底层协议的心跳参数。

```
int set_keepalive(int flag = 1, int idle = 7200, int interval = 75, int probes = 10);
```

参数

flag
`1`：开启底层协议心跳，`0`：关闭。

idle
当应用层没有任何消息交互后，启动底层协议心跳探测的最大超时时间，单位（秒）。

interval
当没有收到心跳回应时，重复发送心跳探测报时间间隔，单位（秒）。

probes
当没有收到心跳回应时，最大探测次数，超过探测次数后，会触发应用层连接断开。

返回值

`0`：成功，`< 0`失败，通过 `xxsocket::get_last_errno` 获取错误码。

示例

```
// xxsocket-keepalive.cpp
#include "  yasio/xxsocket.hpp"
using namespace yasio;
using namespace inet;

int main(){
    xxsocket client;
    if(0 == client.pconnect("  192.168.1.19","80")) {
        client.set_keepalive(1, 5, 10, 2);
    }
    return 0;
}
```

xxsocket::reuse_address

设置socket是否允许重用地址。

```
void reuse_address(bool reuse);
```

参数

reuse
是否重用。

注意

此函数一般用于服务器或者组播监听端口。

xxsocket::exclusive_address

是否明确不允许地址重用，以保护通信双方安全。

```
void exclusive_address(bool exclusive);
```

参数

exclusive
`true`：不允许， `false`：允许

注意

点击 查看 `winsock` 安全报告。

xxsocket::set_optval

设置socket选项。

```
template <typename _Ty> int set_optval(int level, int optname, const _Ty& optval);
```

参数

level
socket选项级别。

optname
选项类型。

optval
选项值。

返回值

`0`：成功， `< 0`失败，通过 `xxsocket::get_last_errno` 获取错误码。

注意

此函数同bsd socket `setsockopt` 功能相同，只是使用模板封装，更方便使用。

xxsocket::get_optval

设置socket选项。

```
template <typename _Ty> _Ty get_optval(int level, int optname) const
```

参数

level
socket选项级别。

optname
选项类型。

返回值

返回选项值。

注意

此函数同bsd socket `getsockopt` 功能相同，只是使用模板封装，更方便使用。

xxsocket::select

监听socket内核事件。

```
int select(fd_set* readfds, fd_set* writefds, fd_set* exceptfds, const std::chrono::microseconds& wtimeout)
```

参数

readfds
可读事件描述符数组。

writefds
可写事件描述符数组。

exceptfds
异常事件描述符数组。

wtimeout
等待事件超时事件。

返回值

`0`：超时，`> 0`: 成功，`< 0`: 失败，通过 `xxsocket::get_last_errno` 获取错误码。

xxsocket::shutdown

关闭TCP传输通道。

```
int shutdown(int how = SD_BOTH) const;
```

参数

how
关闭通道类型，可传以下枚举值

- `SD_SEND`：发送通道
- `SD_RECEIVE`：接受通道
- `SD_BOTH`：全部关闭

返回值

`0`：成功，`< 0`: 失败，通过 `xxsocket::get_last_errno` 获取错误码。

xxsocket::close

关闭socket，释放系统资源。

```
void close(int shut_how = SD_BOTH);
```

参数

shut_how
关闭通道类型，可以传以下枚举值

- SD_SEND : 发送通道
- SD_RECEIVE : 接受通道
- SD_BOTH : 全部关闭
- SD_NONE : 全部关闭

返回值

0 : 成功， < 0: 失败，通过 xxsocket::get_last_errno 获取错误码。

注意

如果 shut_how ！=SD_NONE，此函数会先调用 shutdown，再调用底层 close 。

xxsocket::tcp_rtt

获取TCP连接的RTT。

```
uint32_t tcp_rtt() const;
```

返回值

返回TCP的RTT时间，单位: 微秒 。

xxsocket::get_last_errno

获取最近一次socket操作错误码。

```
static int get_last_errno();
```

返回值

0 : 无错误， > 0 通过 xxsocket::strerror 转换为详细错误信息。

注意

此函数是线程安全的。

xxsocket::set_last_errno

设置socket操作错误码。

```
static void set_last_errno(int error);
```

参数

error
错误码。

注意

此函数是线程安全的。

xxsocket::not_send_error

判断是否是发送时socket出现无法继续的错误。

```
static bool not_send_error(int error);
```

参数

error
错误码。

返回值

`true`: socket正常， `false`: socket状态已经发生错误，应当关闭socket终止通讯。

注意

仅当发送操作返回值 `< 0`时，调用此函数。

xxsocket::not_recv_error

判断是否是接收时socket出现无法继续的错误。

```
static bool not_recv_error(int error);
```

参数

error
错误码。

返回值

`true`：socket正常，`false`：socket状态已经发生错误，应当关闭socket终止通讯。

注意

仅当接收操作返回值 `< 0`时，调用此函数。

xxsocket::strerror

将错误码转换为字符串。

```
static const char* strerror(int error);
```

参数

error
错误码。

返回值

错误信息的字符串。

xxsocket::gai_strerror

将 getaddrinfo 错误码转换为字符串。

```
static const char* gai_strerror(int error);
```

参数

error
错误码。

返回值

错误信息的字符串。

xxsocket::resolve

解析域名包含的所有地址。

```
int resolve(std::vector<endpoint> &endpoints, const char* hostname, unsigned short port = 0, int socktype = SOCK_STRE
```

参数

endpoints
输出参数。

hostname
域名。

port
端口。

socktype
socket类型。

返回值

`0`：无错误， `> 0` 通过 `xxsocket::strerror` 转换为详细错误信息。

xxsocket::resolve_v4

解析域名包含的IPv4地址。

```
int resolve_v4(std::vector<endpoint> &endpoints, const char* hostname, unsigned short port = 0, int socktype = SOCK_ST
```

参数

endpoints
输出参数。

hostname
域名。

port
端口。

socktype
socket类型。

返回值

`0`：无错误， `> 0` 通过 `xxsocket::strerror` 转换为详细错误信息。

xxsocket::resolve_v6

解析域名包含的IPv6地址。

```
int resolve_v6(std::vector<endpoint> &endpoints, const char* hostname, unsigned short port = 0, int socktype = SOCK_S
```

参数

endpoints
输出参数。

hostname
域名。

port
端口。

socktype
socket类型。

返回值

`0`：无错误， `> 0` 通过 `xxsocket::strerror` 转换为详细错误信息。

xxsocket::resolve_v4to6

仅解析域名包含的IPv4地址并转换为IPv6 V4MAPPED格式。

```
int resolve_v4to6(std::vector<endpoint> &endpoints, const char* hostname, unsigned short port = 0, int socktype = SOCK
```

参数

endpoints
输出参数。

hostname
域名。

port
端口。

socktype
socket类型。

返回值

`0`：无错误， `> 0` 通过 `xxsocket::strerror` 转换为详细错误信息。

xxsocket::resolve_tov6

解析域名包含的所有地址，IPv4地址会转换为IPv6 V4MAPPED格式。

```
int resolve_tov6(std::vector<endpoint> &endpoints, const char* hostname, unsigned short port = 0, int socktype = SOCK_
```

参数

endpoints
输出参数。

hostname
域名。

port
端口。

socktype
socket类型。

返回值

`0`：无错误， `> 0` 通过 `xxsocket::strerror` 转换为详细错误信息。

xxsocket::getipsv

获取本机支持的IP协议栈标志位。

```
static int getipsv();
```

返回值

- ipsv_ipv4 : 本机只支持IPv4协议。
- ipsv_ipv6 : 本机只支持IPv6协议。
- ipsv_dual_stack : 本机支持IPv4和IPv6双栈协议。

注意

当返回值支持双栈协议是，用户应当始终优先使用IPv4通信，
例如 智能手机设备 在同时开启 wifi 和 蜂窝网络 时，将会优先选择wifi，
而wifi通常是IPv4，详见: https://github.com/halx99/yasio/issues/130

示例

```
// xxsocket-ipsv.cpp
#include < vector>
#include "  yasio/xxsocket.hpp"
using namespace yasio;
int main(){
    const char* host = "  github.com;"
    std::vector<ip::endpoint> eps;
    int flags = xxsocket::get_ipsv();
    if(flags & ipsv_ipv4) {
        xxsocket::resolve_v4(eps, host, 80);
    }
    else if(flags & ipsv_ipv6) {
        xxsocket::resolve_tov6(eps, host, 80);
    }
    else {
        std::cerr < <"  Local network not available\ n;
    }
    return 0;
}
```

xxsocket::traverse_local_address

枚举本机地址。

```
static void traverse_local_address(std::function<bool(const ip::endpoint&)> handler);
```

参数

handler
枚举地址回调。

示例

```
// xxsocket-traverse.cpp
#include < vector>
#include "  yasio/xxsocket.hpp"
using namespace yasio;
int main(){
    int flags = 0;
    xxsocket::traverse_local_address([&] (const ip::endpoint& ep) -> bool {
        switch (ep.af())
        {
          case AF_INET:
            flags |  =ipsv_ipv4;
            break;
          case AF_INET6:
            flags |  =ipsv_ipv6;
            break;
        }
        return (flags == ipsv_dual_stack);
    }  );
    YASIO_LOG("  Supported ip stack flags=%d"flags);
    return flags;
}
```

请参阅

io_service Class

endpoint Class

封装底层 bsd socket 地址，支持IPv4和IPv6，包含IP地址和端口信息，可直接用于底层 socket API: bind/connect/sendto/recvfrom 。

语法

```
namespace yasio { inline namespace inet { inline namespace ip { struct endpoint; } } }
```

成员

Name	Description
endpoint::endpoint	构造一个 endpoint 对象

公共方法

Name	Description
endpoint::operator bool	检查是否是一个有效地址
endpoint::operator=	赋值运算符重载
endpoint::operator&	取地址运算符重载
endpoint::as_is	从已知地址类型构造
endpoint::as_in	从ip，端口等地址信息构造
endpoint::as_un	构造为unix domain socket地址
endpoint::as_is_raw	从连续内存地址构造
endpoint::z eroset	清零
endpoint::af	设置或获取地址类型(族)

Name	Description
endpoint::ip	设置或获取字符串形式ip
endpoint::port	设置或获取端口
endpoint::addr_v4	设置或获取IPv4地址
endpoint::is_global	检查是否是全局地址
endpoint::len	获取地址长度
endpoint::to_string	将地址转换为字符串
endpoint::format_to	格式化地址到字符串

endpoint::endpoint

构造 `endpoint` 对象。

```
endpoint::endpoint();
endpoint::endpoint(const endpoint& rhs);
explicit endpoint::endpoint(const addrinfo* ai);
explicit endpoint::endpoint(const sockaddr* sa);
explicit endpoint::endpoint(const char* str_ep);
endpoint::endpoint(const char* ip, unsigned short port);
endpoint::endpoint(uint32_t addrv4, unsigned short port);
endpoint::endpoint(int family, const void* addr, unsigned short port);
```

参数

rhs
构造endpoint的右值 endpoint 对象

ai
构造endpoint的addrinfo信息

sa
构造endpoint的sockaddr信息

str_ep
字符串表示的IP和端口信息，格式为 `127.0.0.1:2022` 或 `[ fe80::1] :2033`

ip
构造endpoint的IPv4或IPv6地址字符串

port
构造endpoint的端口

addrv4
构造endpoint的4字节无符号整数表示的IPv4地址值

family
构造endpoint的地址类型

addr
构造endpoint地址，地址类型由family决定

endpoint::operator bool

检查是否是一个有效地址。

```
explicit operator bool() const;
```

返回值

`true`：地址有效， `false`： 地址无效

endpoint::operator =

赋值运算符重载。

```
endpoint& operator=(const endpoint& rhs) const;
```

返回值

返回对象自身的引用。

示例

```
#include "  yasio/xxsocket.h"
int main() {
    yasio::endpoint ep("  127.0.0.1,"2021);
    yasio::endpoint ep1("  127.0.0.1,"2022);
    yasio::endpoint ep3{ep}  ;
    ep3 = ep1;
```

```
    return 0;
}
```

endpoint::operator &

取地址运算符重载。

```
sockaddr* operator&();
const sockaddr* operator&() const;
```

返回值

返回sockaddr* 类型，可直接用户底层bsd socket API。

注意

如果需要获取 ip::endpoint* 类型指针，请使用 std::addressof 。

示例

```
#include "  yasio/xxsocket.h"
int main() {
    yasio::endpoint ep("  127.0.0.1,"2021);
    yasio::socket sock(AF_INET, SOCK_STREAM, 0);
    ::connect(sock.native_handle(), &ep, ep.len());
    return 0;
}
```

endpoint::as_is

由已知地址信息构造。

```
endpoint& as_is(const endpoint& rhs);
endpoint& as_is(const addrinfo* info);
endpoint& as_is(const sockaddr* addr);
endpoint& as_is(const char* str_ep);
```

参数

rhs
已有 endpoint 对象

info
addrinfo* 地址信息

addr
sockaddr* 类型地址

str_ep
字符串表示的IP和端口信息，格式为 `127.0.0.1:2022` 或 `[fe80::1]:2033`

返回值

返回对象自身的引用。

endpoint::as_in

由地址类型、地址和端口参数构造。

```
endpoint& as_in(int family, const void* addr_in, u_short port);
endpoint& as_in(const char* addr, unsigned short port);
endpoint& as_in(uint32_t addrv4, u_short port);
```

参数

family
地址类型

addr_in
in地址，类型由参数 `family` 决定

addr
字符串表示的IPv4或IPv4地址

addrv4
4字节无符号整数表示的IPv4地址

port
端口号

返回值

返回对象自身的引用。

endpoint::as_un

构造为Unix domain socket的endpoint，当编译器宏 `YASIO_ENABLE_UDS` 定义时可用。

```
endpoint& as_un(const char* name);
```

参数

name
Unix domain socket本地磁盘路径，不能超过 sizeof(sockaddr::sun_path)

返回值

返回对象自身的引用。

endpoint::as_is_raw

由已知地址信息构造。

```
endpoint& as_is_raw(const void* ai_addr, size_t ai_addrlen);
```

参数

ai_addr
地址指针

ai_addrlen
地址长度

返回值

返回对象自身的引用。

endpoint::zeroset

清空地址。

```
void zeroset();
```

endpoint::af

设置或获取endpoint的地址类型。

```
void af(int v);
int af() const;
```

参数

v
地址类型： 取值 `AF_INET` 、 `AF_INET6`

返回值

返回地址类型。

endpoint::ip

设置或获取ip字符串。

```
void ip(const char* v);
std::string ip() const;
```

参数

v
字符串表示的IPv4或IPv6地址，例如: 127.0.0.1、fe80::1

返回值

返回字符串表示的IPv4或IPv6地址。

endpoint::port

设置或获取端口号。

```
void port(u_short v);
u_short port() const;
```

参数

v
端口号

返回值

返回端口号。

endpoint::addr_v4

设置或获取4字节无符号整数表示的IPv4地址。

```
void addr_v4(uint32_t addr);
uint32_t addr_v4() const;
```

参数

addr
`uint32_t` 表示的IPv4地址，会将 `endpoint` 的地址类型修改为 `AF_INET`。

返回值

返回4字节无符号整数表示的IPv4地址，如果地址类型不是IPv4则返回 `0`

endpoint::len

设置或获取地址长度。

```
void len(int n);
int len() const;
```

参数

n
地址长度

返回值

返回地址长度。

endpoint::is_global

判断地址是否为全局地址，非回环地址。

```
bool is_global() const;
```

返回值

`true`：全局地址， `false`：本机回环地址。

endpoint::to_string

转换为字符串。

```
std::string to_string(int flags = endpoint::fmt_default) const;
```

参数

flags
格式化标志位，详见: endpoint::fmt_xxx

返回值

返回字符串表示的IPv4或IPv6地址，是否包含端口取决于参数 flags 。

endpoint::format_to

赋值运算符重载。

```
size_t format_to(std::string& buf, int flags = 0) const;
size_t format_to(char* buf, size_t buf_len, int flags) const;
```

参数

buf
格式化目标缓冲区。

buf_len
格式化目标缓冲区大小，必须确保大于 `endpoint::max_fmt_len` 。

flags
格式化标志位，详见: endpoint::fmt_xxx

返回值

返回字符串表示的IPv4或IPv6地址，是否包含端口取决于参数 `flags` 。

endpoint::fmt_xxx

地址格式化标志位枚举值

```
struct endpoint {
    enum
    {
        fmt_no_local   = 1,
        fmt_no_port    = 2,
        fmt_no_port_0  = 4,
        fmt_no_un_path = 8,
        fmt_default    = fmt_no_port_0 | fmt_no_un_path,
    } ;
}
```

- fmt_no_local: 忽略回环地址
- fmt_no_port: 忽略端口
- fmt_no_port_0: 忽略0端口
- fmt_no_un_path: 忽略unix path
- fmt_default: 忽略0端口和unix path

io_service options

io_service支持的所有选项。

Name	Description
YOPT_S_DEFER_EVENT_CB	Set defer event callback params: callback:defer_event_cb_t remarks: a. User can do custom packet resolve at network thread, such as decompress and crc check. b. Return true, io_service will continue enque to event queue. c. Return false, io_service will drop the event.
YOPT_S_DEFERRED_EVENT	Set whether deferred dispatch event, default is: 1 params: deferred_event:int(1)
YOPT_S_RESOLV_FN	Set custom resolve function, native C++ ONLY params: func:resolv_fn_t*
YOPT_S_PRINT_FN	Set custom print function native C++ ONLY parmas: func:print_fn_t remarks: you must ensure thread safe of it
YOPT_S_PRINT_FN2	Set custom print function with log level parmas: func:print_fn2_t you must ensure thread safe of it
YOPT_S_EVENT_CB	Set event callback params: func:event_cb_t*
YOPT_S_TCP_KEEPALIVE	Set tcp keepalive in seconds, probes is tries. params: idle:int(7200), interal:int(75), probes:int(10)
YOPT_S_NO_NEW_TH READ	Don't start a new thread to run event loop. params: value:int(0)
YOPT_S_SSL_CACERT	Sets ssl verification cert, if empty, don't verify. params: path:const char*

Name	Description
YOPT_S_CONNECT_TIMEOUT	Set connect timeout in seconds. params: connect_timeout:int(10)
YOPT_S_CONNECT_TIMEOUTMS	Set connect timeout in milliseconds. params: connect_timeout:int(10000)
YOPT_S_DNS_CACH E_TIMEOUT	Set dns cache timeout in seconds. params: dns_cache_timeout : int(600)
YOPT_S_DNS_CACH E_TIMEOUTMS	Set dns cache timeout in milliseconds. params: dns_cache_timeout : int(600000)
YOPT_S_DNS_Q UERIES_TIMEOUT	Set dns queries timeout in seconds, default is: 5. params: dns_queries_timeout : int(5) remark: a. this option must be set before 'io_service::start' b. only works when have c-ares c. since v3.33.0 it's milliseconds, previous is seconds. d. the timeout algorithm of c-ares is complicated, usually, by default, dns queries will failed with timeout after more than 75 seconds. e. for more detail, please see: https://c-ares.haxx.se/ares_init_options.html
YOPT_S_DNS_Q UERIES_TIMEOUTMS	Set dns queries timeout in seconds, see also YOPT_S_DNS_Q UERIES_TIMEOUT
YOPT_S_DNS_Q UERIES_TRIES	Set dns queries tries when timeout reached, default is: 5. params: dns_queries_tries : int(5) remarks: a. this option must be set before 'io_service::start' b. relative option: YOPT_S_DNS_Q UERIES_TIMEOUT
YOPT_S_DNS_DIRTY	Set dns server dirty. params: reserved : int(1) remarks: a. this option only works with c-ares enabled b. you should set this option after your mobile network changed

Name	Description
YOPT_S_DNS_LIST	Set dns server list. params: servers : const char* (" xxx.xxx.xxx.xxx[:port] ,xxx.xxx.xxx.xxx[:port] ")
YOPT_C_LFBFD_FN	Sets channel length field based frame decode function. params: index:int, func:decode_len_fn_t* remark: native C++ ONLY
YOPT_C_LFBFD_PARAMS	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_adj ustment:int(0),
YOPT_C_LFBFD_IBTS	Sets channel length field based frame decode initial bytes to strip. params:index:int,initial_bytes_to_strip:int(0)
YOPT_C_REMOTE_H OST	Sets channel remote host. params: index:int, ip:const char*
YOPT_C_REMOTE_PORT	Sets channel remote port. params: index:int, port:int
YOPT_C_REMOTE_ENDPOINT	Sets channel remote endpoint. params: index:int, ip:const char* , port:int
YOPT_C_LOCAL_H OST	Sets local host for client channel only. params: index:int, ip:const char*
YOPT_C_LOCAL_PORT	Sets local port for client channel only. params: index:int, port:int
YOPT_C_LOCAL_ENDPOINT	Sets local endpoint for client channel only. params: index:int, ip:const char* , port:int
YOPT_C_MOD_FLAGS	Mods channl flags. params: index:int, flagsToAdd:int, flagsToRemove:int

Name	Description
YOPT_C_ENABLE_MCAST	Enable channel multicast mode. params: index:int, multi_addr:const char* , loopback:int
YOPT_C_DISABLE_MCAST	Disable channel multicast mode. params: index:int
YOPT_C_KCP_CONV	The kcp conv id, must equal in two endpoint from the same connection. params: index:int, conv:int
YOPT_T_CONNECT	Change 4-tuple association for io_transport_udp. params: transport:transport_handle_t remark: only works for udp client transport
YOPT_T_DISCONNECT	Dissolve 4-tuple association for io_transport_udp. params: transport:transport_handle_t remark: only works for udp client transport
YOPT_B_SOCKOPT	Sets io_base sockopt. params: io_base* ,level:int,optname:int,optval:int,optlen:int

请参阅

io_service Class

其他常用API

命名空间

```
namespace yasio { }
```

公共方法

Name	Description
yasio::host_to_network	主机字节序转网络字节序
yasio::network_to_host	网络字节序转主机字节序
yasio::xhighp_clock	获取纳秒级时间戳
yasio::highp_clock	获取微秒级时间戳
yasio::clock	获取毫秒级时间戳
yasio::set_thread_name	设置调用者线程名
yasio::basic_strfmt	格式化字符串

yasio::host_to_network

主机字节序转网络字节序。

头文件

yasio/detail/endian_portable.hpp

```
template <typename _Ty>
inline _Ty host_to_network(_Ty value);
inline int host_to_network(int value, int siz
```

参数

value
要转换的数值，value类型 _Ty 可以是任意1~8字节数值类型。

siz e
要转换的数值有效字节数，只能是1~4字节。

返回值

网络字节序数值。

示例

```
#include < stdio.h>
#include < inttypes.h>
#include "  yasio/detail/endian_portable.hpp"
int main(){
    uint16_t v1 = 0x1122;
    uint32_t v2 = 0x11223344;
    uint64_t v3 = 0x1122334455667788;
    // output will be: net.v1=2211, net.v2=44332211, net.v3=8877665544332211
    printf("  net.v1=%04PRIx16 "  , net.v2=%08PRIx32 "  , net.v3=%016PRIx64 "\  n,
        yasio::host_to_network(v1),
        yasio::host_to_network(v2),
        yasio::host_to_network(v3));
    return 0;
}
```

y a s i o : : n e t w o r k _ t o _ h o s t

网络字节序转主机字节序。

头文件

`yasio/detail/endian_portable.hpp`

```
template <typename _Ty>
inline _Ty network_to_host(_Ty value);
inline int network_to_host(int value, int siz  e
```

参数

value
要转换的数值，value类型 _Ty 可以是任意1~8字节数值类型。

siz e
要转换的数值有效字节数，只能是1~4字节。

返回值

主机字节序数值。

示例

```
#include < stdio.h>
#include < inttypes.h>
#include "  yasio/detail/endian_portable.hpp"
int main(){
    uint16_t v1 = 0x2211;
    uint32_t v2 = 0x44332211;
    uint64_t v3 = 0x8877665533442211;
    // output will be: net.v1=1122, net.v2=11223344, net.v3=1122334455667788
    printf("  host.v1=%04PRIx16 "  , host.v2=%08PRIx32 "  , host.v3=%016PRIx64 "\ n,
        yasio::network_to_host(v1),
        yasio::network_to_host(v2),
        yasio::network_to_host(v3));
    return 0;
}
```

y a s i o : : x h i g h p _ c l o c k

获取纳秒级时间戳。

头文件

yasio/detail/utils.hpp

```
template <typename _Ty = steady_clock_t>
inline highp_time_t xhighp_clock();
```

模板参数

_Ty

- yasio::steady_clock_t: 返回当前系统时间无关的时间戳
- yasio::system_clock_t: 返回系统UTC时间戳

返回值

纳秒级时间戳。

yasio::highp_clock

获取微秒级时间戳。

头文件

`yasio/detail/utils.hpp`

```
template <typename _Ty = steady_clock_t>
inline highp_time_t highp_clock();
```

模板参数

_Ty

- `yasio::steady_clock_t` : 返回当前系统时间无关的时间戳
- `yasio::system_clock_t` : 返回系统UTC时间戳

返回值

微秒级时间戳。

yasio::clock

获取毫秒级时间戳。

头文件

`yasio/detail/utils.hpp`

```
template <typename _Ty = steady_clock_t>
inline highp_time_t clock();
```

模板参数

_Ty

- `yasio::steady_clock_t` : 返回当前系统时间无关的时间戳
- `yasio::system_clock_t` : 返回系统UTC时间戳

返回值

毫秒级时间戳。

yasio::set_thread_name

设置调用者线程名。

头文件

`yasio/detail/thread_name.hpp`

```
inline void set_thread_name(const char* name);
```

参数

name
需要设置线程名称

注意

此函数用于诊断多线程程序非常有用。

yasio::basic_strfmt

函数模板，格式化字符串或者宽字符串。

头文件

`yasio/detail/strfmt.hpp`

```
template <class _Elem, class _Traits = std::char_traits<_Elem>,
        class _Alloc = std::allocator<_Elem> >
inline std::basic_string<_Elem, _Traits, _Alloc> basic_strfmt(size_t n, const _Elem* format, ...);
```

参数

n
初始buffer大小

format
格式字符串，和标准库 `printf` 相同

返回值

返回格式化后的 std::string 类型字符串。

注意

实际使用，可直接使用已经定义好的别名

- yasio::strfmt : 格式化字符串
- yasio::wcsfmt : 格式化宽字符串

示例

```
#include "  yasio/detail/strfmt.hpp"
int main() {
    std::string str1 = yasio::strfmt(64, "  My age is %d", 19);
    std::wstring str2 = yasio::wcsfmt(64, L"  My age is %d", 19);
    return 0;
}
```

yasio 宏定义

以下宏定义可以控制 `yasio` 库的某些行为，可以在 yasio/detail/config.hpp 定义或者在编译器预处理器定义

Name	Description
YASIO_H AVE_KCP	是否启用kcp传输支持，需要kcp已经在软件编译系统中，默认关闭。
YASIO_H EADER_ONLY	是否以仅头文件的方式使用yasio核心组件，默认关闭。
YASIO_SSL_BACKEND	选择SSL库以支持SSL客户端，需要软件编译系统包含OpenSSL/MbedTLS库， 3.36.0新增(同时移除YASIO_H AVE_SSL)，此宏只能取值 `1` (使用OpenSSL) 或者 `2` (使用mbedtls)。
YASIO_ENABLE_UDS	是否启用unix domain socket支持，目前仅类unix系统和win10 RS5+支持，默认关闭。
YASIO_H AVE_CARES	是否启用c-ares异步域名解析库， 当编译系统包含c-ares时可启用，有效避免每次解析域名都新开线程。 yasio有DNS缓存机制，超时时间默认10分钟， 因此无c-ares也不会造成太大的性能损耗。
YASIO_VERBOSE_LOG	是否打印详细日志，默认关闭。
YASIO_NT_COMPAT_GAI	是否启用Windows X P系统下使用 `getaddrinfo` API支持。
YASIO_USE_SPSC_Q UEUE	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，默认关闭。
YASIO_USE_SH ARED_PACKET	是否使用 `std::shared_ptr` 包装网络包，使其能在多线程之间共享，默认关闭。
YASIO_H AVE_H ALF_FLOAT	是否启用半精度浮点数支持，依赖 half.hpp。
YASIO_DISABLE_OBJ ECT_POOL	是否禁用对象池的使用，默认启用。

Name	Description
YASIO_DISABLE_CONCURRENT_SINGLETON	是否禁用并发单利类模板。
YASIO_ENABLE_PASSIVE_EVENT	是否启用服务端信道open/close事件产生，默认关闭。

yasio 粘包处理

yasio的粘包处理不仅针对TCP，对于UDP，如果发送端有组包发送机制，也是以相同的方式处理。整体来讲有两种方式:

- 通过io_service选项 YOPT_C_LFBFD_PARAMS 设置信道参数。
- 通过io_service选项 `YOPT_C_LFBFD_FN` 设置自定义包长度解码函数 decode_len_fn_t 。

> 注意
>
> 自定义解码包长度函数实现时，当从字节流中读取int值时，一定不要使用指针强转，否则可能触发ARM芯片字节对齐问题 `SIGBUS` 异常闪退。可以参考内置解码包长度函数实现 `io_channel::__builtin_decode_len` 。

YOPT_C_LFBFD_PARAMS

设置信道拆包参数。

参数

max_frame_length
最大包长度，超过将视为异常包。

length_field_offset
包长度字段相对于消息包数据首字节偏移。

length_field_length
包长度字段大小，支持1~4字节整数(uint8_t,uint16_t,uint24_t,int32_t)。

length_adj ustment
包长度调整值。通常应用层二进制协议都会设计消息头和消息体，当长度字段值包含消息头时，则此值为 `0` ，否则为 `消息头长度` 。

注意

消息包长度字段必须使用网络字节序编码。请查看内置解码包长度实现:

```
int io_channel::__builtin_decode_len(void* d, int n)
{
  int loffset = uparams_.length_field_offset;
  int lsiz e = uparams_.length_field_length;
  if (loffset > =0)
  {
```

```
    int len = 0;
    if (n >= (loffset + lsize))
    {
      ::memcpy(&len, (uint8_t*)d + loffset, lsize);
      len = yasio::network_to_host(len, lsize);
      len += uparams_.length_adjustment;
      if (len > uparams_.max_frame_length)
        len = -1;
    }
    return len;
  }
  return n;
}
```

decode_len_fn_t

信道解码消息包长度函数原型。

```
typedef std::function<int(void* d, int n)> decode_len_fn_t;
```

参数

d
待解包数据首字节地址。

n
待解包数据长度。

返回值

返回消息包数据实际长度。

- `> 0`: 解码包长度成功。
- `== 0` : 接收数据不足以解码消息包实际长度，yasio底层会继续接收数据，一旦收到新数据，会再次调用此函数。
- `< 0`: 解码包长度异常，会触发当前传输会话断开。

注意

解码消息包长度函数必须是线程安全的，例如对于Lua则不支持设置自定义解码消息包长度函数。

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
完整demo，请查看: https://github.com/yasio/yasio_unity

FAQ

重点问题解答

在iOS 14+设备上连接本地局域网主机任意端口号报错：ec=65, detail:No route to host

- 解决方案: 打开你的iPhone【设置】【隐私】【本地网络】找到你的应用，开启即可。

在iOS设备上连接服务器端口号为10161时失败，错误信息：ec=65, detail:No route to host

- 根本原因:
  - 当APP初次启动时，在ios14.1+设备上，会提示请求 `Local Network Permission` 权限，用户点拒绝。
  - 端口号 10161 是系统保留端口，用于 `SNMP via TLS` 协议。
- 解决方案: 使用其他端口号作为APP服务端口，建议使用动态/私密端口，范围： 49152~65535。
- 其他参考: https://www.speedguide.net/port.php

yasio是否依赖其他网络库?

yasio的核心代码默认不依赖任何第三方库，从作者从业踏入通信行业开始就琢磨着编写一个轻量而好用的通用网络库，xxsocket就是yasio的起源，只有异步消息发送的软中断器提取于boost.asio。

为什么使用yasio?

1. 开源社区已知比较有名的网络库有asio,libevent,libev,libuv，他们提供的都是非常基础的非阻塞多路io复用模型，并且，各平台底层会使用iocp,kqueue,epoll,select等模型，拿来做客户端网络，如连接管理，TCP粘包处理等都需要程序员自己处理。
2. yasio将连接管理，TCP拆包都封装到了底层。
3. yasio将TCP,UDP,KCP统一抽象成Transport更加方便使用。
4. yasio更轻量级，所有平台均使用select模型。

yasio是否支持非阻塞域名解析?

支持，但需要依赖c-ares库，这个库是完全实现DNS协议，并可以很好的和现有select模型结合使用，域名解析无需新开线程。如果不开启c-ares，yasio内部会为每次域名解析开线程。但会默认缓存10分钟，所以也不用担心开线程太频繁。

yasio是否支持SSL/TLS传输?

支持，但需要依赖OpenSSL或者MbedTLS。

如何从 io_event 中设置和获取 userdata

请参考：https://github.com/yasio/ftp_server/blob/master/ftp_server.cpp#L98

yasio是否处理 SIGPIPE 信号?

从 `3.30` 版本开始已处理。
如果不处理，实测过iOS在加载资源比较耗时会触发SIGPIPE直接导致APP闪退，处理后，仅仅触发TCP连接断开，错误码32，错误信息: `Broken pipe`
详见: https://github.com/yasio/yasio/issues/170

macOS 上发送UDP失败，报错误码40，错误信息 `Message too long` 怎么办?

- 原因: 数据包太大，macOS系统UDP发送缓冲区默认为 `9126` 字节。
- 解决方案: 通过socket选项将UDP发送缓冲区设置大一点，例如:
  - xxsocket接口设置方式为: `sock_udp.set_optval(SOL_SOCKET, SO_SNDBUF, (int)65535);`
  - io_service设置方式请参见: https://github.com/yasio/yasio/blob/master/tests/speed/main.cpp#L223

io_service schedule 可以多个任务吗?

可以。

std::thread 在某些嵌入式平台编译器闪退怎么办?

修改编译参数，详见: https://github.com/yasio/yasio/issues/244

xxsocket 函数命名后缀_n和_i的意思?

_n是nonblock的意思，_i是internal(新版本已去除)。

设置了YOPT_TCP_KEEPALIVE，还需要应用层心跳吗?

一般不需要，除非需要检测网络延时展示给用户，详见: https://github.com/yasio/yasio/issues/117

Lua绑定闪退怎么办?

- c++11:
  - 使用kaguya绑定库，但这个库有个问题：在绑定c++类的过程中，构造c++对象过程是先通过lua_newuserdata, 再通过placement new构造对象，在xcode clang release优化编译下会直接闪退
  - 经过bing.com搜索，发现可通过定义 `LUAI_USER_ALIGNMENT_T=max_align_t` 来解决
  - 经过测试, max_align_t在xcode clang下定义为long double类型，长度为16个字节

    - 参考: http://lua-users.org/lists/lua-l/2019-07/msg00197.html
- c++14:
    - 使用sol2绑定库，sol2可以成功解决kaguya的问题，经过查看sol2的源码，发现其在内部对lua_newuserdata返回的地址做了对齐处理，因此成功避免了clang release优化地址不对齐闪退问题
- c++17:
    - 同样使用sol2库，但xcode会提示ios11以下不支持C++17默认 `new` 操作符的 `Aligned allocation/deallocation`，通过添加编译选项-faligned-allocation可通过编译；
    - ios10以下不支持stl的shared_mutex(yasio最近做了兼容，对于Apple平台一律使用pthread实现)
- 思考：
    - Lua虚拟机实现中默认最大对齐类型是double，而 `max_align_t` 类型是C11标准才引入的： https://en.cppreference.com/w/c/types/max_align_t ，因此Lua作为ANSI C89标准兼容实现并未完美处理各个编译器地址对齐问题，而是留给了用户定义：`LUAI_USER_ALIGNMENT_T`
    - 在C++11编译系统下已经引入 `std::max_align_t` 类型: https://en.cppreference.com/w/cpp/types/max_align_t

重要: max_align_t各平台定义请查看llvm项目的 __stddef_max_align_t.h

Can't load xxx.bundle on macOS?

The file `xxx.bundle` needs change attr by command `sudo xattr -r -d com.apple.quarantine xxx.bundle`

xxsocket的resolve系列函数socktype参数作用?

- winsock实现可以忽略
- 其他操作系统如果设置为0会返回多个地址，即使直接传IP。

更多常见问题

请参阅 项目帮助台