ascs 开发文档

ascs（https://github.com/youngwolf-project/ascs.git）是一套基于asio（http://think-async.com/Asio/）的c/s框架，以帮助开发者快速的搭建一个c/s系统，支持tcp和udp，ipv4和ipv6以及ssl；  
ascs自带可修改大小的消息缓存；  
ascs通过打包解包器，将c/s中用户协议相关的东西剥离开来，开发者可以很方便的使用自己的打包解包器而无需修改ascs代码；  
ascs附带原生定时器，以及对象管理（对象池，对象重用等）；  
  
请注意，ascs仅支持异步收发数据。

# 一：先看一个最简单的例子

#include <iostream>

#include <ascs/ext/tcp.h>

using namespace ascs;

using namespace ascs::tcp;

using namespace ascs::ext::tcp;

int main(int argc, const char\* argv[]) {

service\_pump sp;

server s(sp);

single\_client c(sp);

sp.start\_service();

while(sp.is\_running())

{

std::string str;

std::cin >> str;

if (!str.empty())

{

c.send\_msg("client says: " + str, false);

s.broadcast\_msg("server says: " + str, false);

}

}

return 0;

}

编译（以后不再说明）：

g++ -o main -std=c++11 -pthread main.cpp

然后启动运行，随便输入点什么，客户端会把输入发到服务端，服务端也会把输入发到客户端（注：这个例子中我们演示一个服务端对应一个客户端）。

那么后台到底做了些什么呢？

1. 服务端设置地址，由于例子中未显示设置（调用ascs::tcp::server\_base::set\_server\_addr或者定义ASCS\_SERVER\_PORT宏），所以采用默认的地址（IP为0.0.0.0，不支持宏定义）：

#ifndef ASCS\_SERVER\_PORT

#define ASCS\_SERVER\_PORT 5050

#endif

1. 客户端设置连接地址（服务器地址），由于例子中未显示设置（调用ascs::tcp::client\_socket\_base::set\_server\_addr或者定义ASCS\_SERVER\_IP 和ASCS\_SERVER\_PORT宏），所以采用默认的地址：

#ifndef ASCS\_SERVER\_IP

#define ASCS\_SERVER\_IP "127.0.0.1"

#endif

#ifndef ASCS\_SERVER\_PORT

#define ASCS\_SERVER\_PORT 5050

#endif

1. ascs::tcp::server\_base服务端开始监听并接受连接；
2. ascs::tcp::client\_socket\_base主动发起对服务器的连接；
3. 发送数据通过ascs::socket::send\_msg和ascs::tcp::server\_base::broadcast\_msg实现，示例中已经体现；
4. 对于收到数据的显示，示例中没有写，所以采用了ascs::socket的默认实现（即简单打印消息内容及长度）。

注：server是一个ascs::tcp::server\_base，只是简化了模板参数（用默认参数），single\_client是一个ascs::tcp::single\_client\_base，只是简化了模板参数（用默认参数），继承关系是：ascs::single\_client\_base->ascs::tcp::client\_socket\_base->ascs::tcp::socket\_base->ascs::socket；

另外ascs::tcp::server\_base又是一个容器，支持多条连接，它用ascs::tcp::server\_socket\_base来与ascs::tcp::client\_socket\_base对应表示一条连接的另一端，继承关系是：ascs::tcp::server\_socket\_base->ascs::tcp::socket\_base->ascs::socket。

# 二：我想处理数据怎么办（客户端）？

基于第一个示例（只列出了变动的地方）：

class my\_client : public single\_client

{

public:

…; //实现single\_client需要的构造函数

protected:

virtual bool on\_msg\_handle(out\_msg\_type& msg) {…;} //你的代码

};

int main(int argc, const char\* argv[]) {

service\_pump sp;

server s(sp);

my\_client c(sp);

…; //略

}

# 三：我想处理数据怎么办（服务端）？

基于第一个示例（只列出了变动的地方）：

class my\_server\_socket : public server\_socket

{

public:

…; //实现server\_socket需要的构造函数

protected:

virtual bool on\_msg\_handle(out\_msg\_type& msg) {…;} //你的代码

};

int main(int argc, const char\* argv[]) {

service\_pump sp;

server\_base<my\_server\_socket> s(sp);

single\_client c(sp);

…; //略

}

# 四：我想支持多条连接怎么办（客户端）？

基于第一个示例（只列出了变动的地方）：

int main(int argc, const char\* argv[]) {

service\_pump sp;

server s(sp);

client c(sp);

c.add\_socket(); //link #1

c.add\_socket(); //link #2

…; //还能添加更多

if (!str.empty())

{

c.broadcast\_msg("client says: " + str, false);

s.broadcast\_msg("server says: " + str, false);

}

}

}

注：client是一个ascs::tcp::client\_base，只是简化了模板参数（用默认参数），同时它又是一个容器，支持多条连接，它用ascs::tcp::client\_socket\_base来与ascs::tcp::server\_socket\_base对应表示一条连接的另一端，继承关系是：ascs::tcp::client\_socket\_base->ascs::tcp::socket\_base->ascs::socket。

# 五：我想支持多条连接怎么办（服务端）？

服务端天生支持多条连接，无论你愿意与否。

# 六：我想处理数据并支持多条连接怎么办（客户端）？

class my\_client\_socket : public client\_socket

{

public:

…; //实现client\_socket需要的构造函数

protected:

virtual bool on\_msg\_handle(out\_msg\_type& msg) {…;} //你的代码

};

int main(int argc, const char\* argv[]) {

service\_pump sp;

server s(sp);

client\_base<my\_client\_socket> c(sp);

c.add\_client(); //link #1

c.add\_client(); //link #2

…; //还能添加更多

if (!str.empty())

{

c.broadcast\_msg("client says: " + str, false);

s.broadcast\_msg("server says: " + str, false);

}

}

}

# 七：我想使用自己的打包解包器怎么办（两种方法）？

1. 定义ASCS\_DEFAULT\_PACKER和ASCS\_DEFAULT\_UNPACKER宏；

single\_client c(sp); //只支持一条连接的客户端

client c(sp); //支持多条连接的客户端

server s(sp); //服务端

1. typedef client\_socket\_base<my\_packer, my\_unpacker> my\_client\_socket;

single\_client\_base<my\_client\_socket> c(sp); //只支持一条连接的客户端

client\_base<my\_client\_socket> c(sp); //支持多条连接的客户端

typedef server\_socket\_base<my\_packer, my\_unpacker> my\_server\_socket;

server\_base<my\_server\_socket> s(sp); //服务端

# 八：我想在server\_socket中调用server的某些功能怎么办（两种方法）？

1. 让server成为全局对象；
2. class my\_i\_server : public i\_server

{

public:

virtual void server\_fun() = 0;

};

class my\_server\_socket : public server\_socket\_base<ASCS\_DEFAULT\_PACKER, ASCS\_DEFAULT\_UNPACKER, my\_i\_server> {}

在my\_server\_socket中，当你需要调用server的server\_fun函数的时候，直接：get\_server().server\_fun(); ；

class my\_server : public server\_base< my\_server\_socket, object\_pool< my\_server\_socket >, my\_i \_server> {}

并在my\_server中实现server\_fun() 纯虚函数；

# 九：我想查找某条连接怎么办（两种方法）？

1. 调用object\_type find(uint\_fast64\_t id)，复杂度O(1)；
2. 调用do\_something\_to\_one，复杂度O(N)：

server s(sp); //client c(sp) 也一样

server::object\_ctype socket\_ptr;

s.do\_something\_to\_one([&](const auto& item)->bool {

//假设get\_username()是你的查找标识，你需要添加它，ascs中没有。

if (item->get\_username() == “abc”)

{

socket\_ptr = item;

return true; //结束遍历

}

return false;

});

注意：容器类才需要查找某条连接，它们分别是server\_base, client\_base, ssl:server\_base and ssl::client\_base.

# 十：定时器

1. 由于几乎所有的类都继承自ascs::socket或者ascs::object\_pool（除了缓存类，打包解包器类，工具类等），所以它们天生带了定时器功能，直接调用set\_timer即可开启一个定时器，这种方法一定要注意定时器的ID必须从父类的TIMER\_END开始（直接父类不一定有，但往上总是能追溯到ascs::socket或者ascs::object\_pool，而在代码的书写上不需要知道是第几层父类定义的TIMER\_END，写直接父类::TIMER\_END即可，编译器能找到正确的定义），并且也应该定义一个自己的TIMER\_END，以便让你的类被继承时，继承者也能取到正确的TIMER\_END值。另外，定时器ID取值范围为[0, 65536)。
2. 自己申明一个timer成员变量，这样想用什么ID都可以，但范围仍然是[0, 65536)，具体如下（回调返回false终止定时器，否则回调结束后重启计时器）：

service\_pump sp; //可以与server或者client共用

timer<tracked\_executor> t(sp);

t.set\_timer(0, 100, [](typename timer<tracked\_executor>::tid id) {return true;});

timer<executor> tt(sp);

tt.set\_timer(0, 100, [](timer<typename executor>::tid id) {return true;});

这里我们创建了两种定时，t是带追踪功能的，tt则不带，所谓追踪功能，就是能判断是否还有异步调用（这里指定时器）未完成（期满或被cancel），这在优雅退出时有用。

1. 对于第一种方案，定时器是否是追踪的，取决于宏ASCS\_DELAY\_CLOSE是否为0。对于第二种方案，退出时，需要用户去停止t和tt里面所有的定时器。

# 十一：Cygwin 和 Mingw 支持

在32bit Cygwin下，直接make即可完成编译。

在64bit Cygwin下，由于asio的原因，无法完成编译。

在64bit Mingw（posix-seh）下，只能通过如下命令完成编译，以demo client为例：

g++ -Wall -fexceptions -std=c++17 -O2 -DNDEBUG -DASIO\_STANDALONE -DASIO\_NO\_DEPRECATED -pthread -I../../../asio/asio/include -I../../include/ client.cpp -s -o client.exe -lstdc++ -lws2\_32 -lwsock32

在64bit Mingw（win32-seh）下，无法完成编译（因为无法使用std线程）。

在32bit Mingw下，无法完成编译（我想可能原因是32bit Mingw已经太久远且不再维护）。