# boost.ASIO

2011-09-07 16:32

1. ASIO 的同步方式
http://hi.baidu.com/walker20100000/blog/item/a0003cdebdcdabf477c63854.html
Boost.Asio 是一个跨平台的网络及底层 IO 的 C++编程库，它使用现代 C++手法实现了统一的异步调用模型。
ASIO 的同步方式

ASIO 库能够使用 TCP、UDP、ICMP、串口来发送/接收数据，下面先介绍 TCP 协议的读写操作。对于读写方式，ASIO 支持同步和异步两种方式，首先登场的是同步方式，下面请同步方式自我介绍一下。
自我介绍

大家好！我是同步方式！

我的主要特点就是执着！所有的操作都要完成或出错才会返回，不过偶的执着被大家称之为阻塞，实在是郁闷~~（场下一片嘘声），其实这样 也是有好处的，比如逻辑清晰，编程比较容易。

在服务器端，我会做个 socket 交给 acceptor 对象，让它一直等客户端连进来，连上以后再通过这个 socket 与客户端通信， 而所有的通信都是以阻塞方式进行的，读完或写完才会返回。

在客户端也一样，这时我会拿着 socket 去连接服务器，当然也是连上或出错了才返回，最后也是以阻塞的方式和服务器通信。

有人认为同步方式没有异步方式高效，其实这是片面的理解。在单线程的情况下可能确实如此，我不能利用耗时的网络操作这段时间做别的事 情，不是好的统筹方法。不过这个问题可以通过多线程来避免，比如在服务器端让其中一个线程负责等待客户端连接，连接进来后把 socket 交给另外的线程去 和客户端通信，这样与一个客户端通信的同时也能接受其它客户端的连接，主线程也完全被解放了出来。

我的介绍就有这里，谢谢大家！
示例代码

好，感谢同步方式的自我介绍，现在放出同步方式的演示代码(起立鼓掌!)。

服务器端
```
#include <iostream>
#include <boost/asio.hpp>

int main(int argc, char* argv[])
```

```
{
using namespace boost::asio;
// 所有 asio 类都需要 io_service 对象
io_service iosev;
ip::tcp::acceptor acceptor(iosev,
ip::tcp::endpoint(ip::tcp::v4(), 1000));
for(;;)
{
// socket 对象
ip::tcp::socket socket(iosev);
// 等待直到客户端连接进来
acceptor.accept(socket);
// 显示连接进来的客户端
std::cout << socket.remote_endpoint().address() << std::endl;
// 向客户端发送 hello world!
boost::system::error_code ec;
socket.write_some(buffer("hello world!"), ec);

// 如果出错，打印出错信息
if(ec)
{
std::cout <<
boost::system::system_error(ec).what() << std::endl;
break;
}
// 与当前客户交互完成后循环继续等待下一客户连接
}
return 0;
}
```

客户端

```
#include <iostream>
#include <boost/asio.hpp>

int main(int argc, char* argv[])
{
using namespace boost::asio;

// 所有 asio 类都需要 io_service 对象
io_service iosev;
// socket 对象
ip::tcp::socket socket(iosev);
// 连接端点，这里使用了本机连接，可以修改 IP 地址测试远程连接
ip::tcp::endpoint ep(ip::address_v4::from_string("127.0.0.1"), 1000);
```

```
// 连接服务器
boost::system::error_code ec;
socket.connect(ep,ec);
// 如果出错，打印出错信息
if(ec)
{
std::cout << boost::system::system_error(ec).what() << std::endl;
return -1;
}
// 接收数据
char buf[100];
size_t len=socket.read_some(buffer(buf), ec);
std::cout.write(buf, len);

return 0;
}
```

小结

从演示代码可以得知

ASIO 的 TCP 协议通过 boost::asio::ip 名 空间下的 tcp 类进行通信。
IP 地址（address,address_v4,address_v6）、 端口号和协议版本组成一个端点（tcp::endpoint）。用于在服务器端生成 tcp::acceptor 对 象，并在指定端口上等待连接；或者在客户端连接到指定地址的服务器上。
socket 是 服务器与客户端通信的桥梁，连接成功后所有的读写都是通过 socket 对 象实现的，当 socket 析 构后，连接自动断 开。
ASIO 读写所用的缓冲区用 buffer 函 数生成，这个函数生成的是一个 ASIO 内部使用的缓冲区类，它能把数组、指针（同时指定大 小）、std::vector、std::string、boost::array 包装成缓冲区类。
ASIO 中的函数、类方法都接受一个 boost::system::error_code 类 型的数据，用于提供出错码。它可以转换成 bool 测试是否出错，并通过 boost::system::system_error 类 获得详细的出错信息。另外，也可以不向 ASIO 的函数或方法提供 boost::system::error_code，这时如果出错的话就会直 接抛出异常，异常类型就是 boost::system:: system_error(它是从 std::runtime_error 继承的)。

2.ASIO 的异步方式

2.ASIO 的异步方式

嗯？异步方式好像有点坐不住了，那就请异步方式上场，大家欢迎...
自我介绍

大家好，我是异步方式

和同步方式不同，我从来不花时间去等那些龟速的 IO 操作，我只是向系统说一声要做什么，然后就可以做其它事去了。如果系统完成了操作， 系统就会通过我之前给它的回调对象来通知我。

在 ASIO 库中，异步方式的函数或方法名称前面都有“async_” 前缀，函数参数里会要求放一个回调函数（或仿函数）。异步操作执行 后不管有没有完成都会立即返回，这时可以做一些其它事，直到回调函数（或仿函数）被调用，说明异步操作已经完成。

在 ASIO 中很多回调函数都只接受一个 boost::system::error_code 参数，在实际使用时肯定是不够的，所以一般 使用仿函数携带一堆相关数据作为回调，或者使用 boost::bind 来绑定一堆数据。

另外要注意的是，只有 io_service 类的 run()方法运行之后回调对象才会被调用，否则即使系统已经完成了异步操作也不会有任务动作。
示例代码

好了，就介绍到这里，下面是我带来的异步方式 TCP Helloworld 服务器端：

```
#include <iostream>
#include <string>
#include <boost/asio.hpp>
#include <boost/bind.hpp>
#include <boost/smart_ptr.hpp>

using namespace boost::asio;
using boost::system::error_code;
using ip::tcp;

struct CHelloWorld_Service
{
CHelloWorld_Service(io_service &iosev)
:m_iosev(iosev),m_acceptor(iosev, tcp::endpoint(tcp::v4(), 1000))
{}

void start()
{
// 开始等待连接（非阻塞）
boost::shared_ptr<tcp::socket> psocket(new tcp::socket(m_iosev));
// 触发的事件只有 error_code 参数，所以用 boost::bind 把 socket 绑定进去
m_acceptor.async_accept(*psocket,
boost::bind(&CHelloWorld_Service::accept_handler, this, psocket, _1));
}

// 有客户端连接时 accept_handler 触发
```

```
void accept_handler(boost::shared_ptr<tcp::socket> psocket, error_code ec)
{
if(ec) return;
// 继续等待连接
start();
// 显示远程 IP
std::cout << psocket->remote_endpoint().address() << std::endl;
// 发送信息(非阻塞)
boost::shared_ptr<std::string> pstr(new std::string("hello async world!"));
psocket->async_write_some(buffer(*pstr),
boost::bind(&CHelloWorld_Service::write_handler, this, pstr, _1, _2));
}

// 异步写操作完成后 write_handler 触发
void write_handler(boost::shared_ptr<std::string> pstr, error_code ec,
size_t bytes_transferred)
{
if(ec)
std::cout<< "发送失败!" << std::endl;
else
std::cout<< *pstr << " 已发送" << std::endl;
}

private:
io_service &m_iosev;
ip::tcp::acceptor m_acceptor;
};

int main(int argc, char* argv[])
{
io_service iosev;
CHelloWorld_Service sev(iosev);
// 开始等待连接
sev.start();
iosev.run();

return 0;
}
```

小结

在这个例子中，首先调用 sev.start()开始接受客户端连接。由于 async_accept 调用后立即返回，start()方法 也就马上完成了。sev.start()在瞬间返回后 iosev.run()开始执行，iosev.run()方法是一个循环，负责分发异步回调事件，只 有所有异步操作全部完成才会返

回。

这 里有个问题，就是要保证 start()方法中 m_acceptor.async_accept 操作所用的 tcp::socket 对象在整个异步操作期间保 持有效(不然系统底层异步操作了一半突然发现 tcp::socket 没了，不是拿人家开涮嘛-_-!!!)，而且客户端连接进来后这个 tcp::socket 对象还有用呢。这里的解决办法是使用一个带计数的智能指针 boost::shared_ptr，并把这个指针作为参数绑定到回调函数上。

一旦有客户连接，我们在 start()里给的回调函数 accept_handler 就会被调用，首先调用 start()继续异步等待其它客户端的连接，然后使用绑定进来的 tcp::socket 对象与当前客户端通信。

发送数据也使用了异步方式(async_write_some)， 同样要保证在整个异步发送期间缓冲区的有效性，所以也用 boost::bind 绑定了 boost::shared_ptr。

对于客户端也一样，在 connect 和 read_some 方法前加一个 async_前缀，然后加入回调即可，大家自己练习写一写。

3.ASIO 的“便民措施”

3.ASIO 的“便民措施”

asio 中提供一些便利功能，如此可以实现许多方便的操作。
端点

回到前面的客户端代码，客户端的连接很简单，主要代码就是两行：

```
...
// 连接
socket.connect(endpoint,ec);
...
// 通信
socket.read_some(buffer(buf), ec);
```

不过连接之前我们必须得到连接端点 endpoint,也就是服务器地址、端口号以及所用的协议版本。

前面的客户端代码假设了服务器使用 IPv4 协议，服务器 IP 地址为 127.0.0.1，端口号为 1000。实际使用的情况是，我们经常只能知道服务器网络 ID，提供的服务类型，这时我们就得使用 ASIO 提供的 tcp::resolver 类来取得服务器的端点了。

比如我们要取得 163 网站的首页，首先就要得到“www.163.com”服务器的 HTTP 端点：

```
io_service iosev;
ip::tcp::resolver res(iosev);
ip::tcp::resolver::query query("www.163.com","80"); //www.163.com 80 端口
```

```
ip::tcp::resolver::iterator itr_endpoint = res.resolve(query);
```

这里的 itr_endpoint 是一个 endpoint 的迭代器，服务器的同一端口上可能不止一个端点，比如同时有 IPv4 和 IPv6 两种。现在，遍历这些端点，找到可用的：

```
// 接上面代码
ip::tcp::resolver::iterator itr_end; //无参数构造生成 end 迭代器
ip::tcp::socket socket(iosev);
boost::system::error_code ec = error::host_not_found;
for(;ec && itr_endpoint!=itr_end;++itr_endpoint)
{
socket.close();
socket.connect(*itr_endpoint, ec);
}
```

如果连接上，错误码 ec 被清空，我们就可以与服务器通信了：

```
if(ec)
{
std::cout << boost::system::system_error(ec).what() << std::endl;
return -1;
}
// HTTP 协议，取根路径 HTTP 源码
socket.write_some(buffer("GET <a href="http://www.163.com"
title="http://www.163.com">http://www.163.com</a> HTTP/1.0 "));
for(;;)
{
char buf[128];
boost::system::error_code error;
size_t len = socket.read_some(buffer(buf), error);
// 循环取数据，直到取完为止
if(error == error::eof)
break;
else if(error)
{
std::cout << boost::system::system_error(error).what() << std::endl;
return -1;
}

std::cout.write(buf, len);
}
```

当所有 HTTP 源码下载了以后，服务器会主动断开连接，这时客户端的错误码得到 boost::asio::error::eof，我们要根据它来判定是否跳出循环。

ip::tcp::resolver::query 的构造函数接受服务器名和服务名。前面的服务名我们直接使

用了端口号"80"，有时 我们也可以使用别名，用记事本打开%windir%\system32\drivers\etc\services 文件（Windows 环境），可以看到 一堆别名及对应的端口，如：

```
echo 7/tcp # Echo
ftp 21/tcp # File Transfer Protocol (Control)
telnet 23/tcp # Virtual Terminal Protocol
smtp 25/tcp # Simple Mail Transfer Protocol
time 37/tcp timeserver # Time
```

比如要连接 163 网站的 telnet 端口（如果有的话），可以这样写：

```
ip::tcp::resolver::query query("www.163.com","telnet");
ip::tcp::resolver::iterator itr_endpoint = res.resolve(query);
```

超时

在网络应用里，常常要考虑超时的问题，不然连接后半天没反应谁也受不了。

ASIO 库提供了 deadline_timer 类来支持定时触发，它的用法是：

```
// 定义定时回调
void print(const boost::system::error_code& /*e*/)
{
std::cout << "Hello, world! ";
}

deadline_timer timer;
// 设置 5 秒后触发回调
timer.expires_from_now(boost::posix_time::seconds(5));
timer.async_wait(print);
```

这段代码执行后 5 秒钟时打印 Hello World!

我们可以利用这种定时机制和异步连接方式来实现超时取消：

```
deadline_timer timer;
// 异步连接
socket.async_connect(my_endpoint, connect_handler/*连接回调*/);
// 设置超时
timer.expires_from_now(boost::posix_time::seconds(5));
timer.async_wait(timer_handler);
...
// 超时发生时关闭 socket
void timer_handler()
{
socket.close();
}
```

最后不要忘了 io_service 的 run()方法。
统一读写接口

除了前面例子所用的 tcp::socket 读写方法（read_some, write_some 等）以外，ASIO 也提供了几个读写函数，主要有这么几个：
read、write、read_until、write_until
当然还有异步版本的
async_read、async_write、async_read_until、async_write_until
这些函数可以以统一的方式读写 TCP、串口、HANDLE 等类型的数据流。

我们前面的 HTTP 客户端代码可以这样改写：

```
...
//socket.write_some(buffer("GET <a href="http://www.163.com"
title="http://www.163.com">http://www.163.com</a> HTTP/1.0 "));
write(socket,buffer("GET <a href="http://www.163.com"
title="http://www.163.com">http://www.163.com</a> HTTP/1.0 "));
...
//size_t len = socket.read_some(buffer(buf), error);
size_t len = read(socket, buffer(buf), transfer_all() ,error);
if(len) std::cout.write(buf, len);
```

这个 read 和 write 有多个重载，同样，有错误码参数的不会抛出异常而无错误码参数的若出错则抛出异常。

本例中 read 函数里的 transfer_all()是一个称为 CompletionCondition 的对象，表示读取/写入直接缓 冲区装满或出错为止。另一个可选的是 transfer_at_least(size_t)，表示至少要读取/写入多少个字符。

read_until 和 write_until 用于读取直到某个条件满足为止,它接受的参数不再是 buffer，而是 boost::asio:: streambuf。

比如我们可以把我们的 HTTP 客户端代码改成这样：

```
boost::asio::streambuf strmbuf;
size_t len = read_until(socket,strmbuf," ",error);
std::istream is(&strmbuf);
is.unsetf(std::ios_base::skipws);
// 显示 is 流里的内容
std::copy(std::istream_iterator<char>(is),
std::istream_iterator<char>(),
std::ostream_iterator<char>(std::cout));
```

基于流的操作

对于 TCP 协议来说，ASIO 还提供了一个 tcp::iostream。用它可以更简单地实现我们的 HTTP

客户端:

```
ip::tcp::iostream stream("www.163.com", "80");
if(stream)
{
// 发送数据
stream << "GET <a href="http://www.163.com"
title="http://www.163.com">http://www.163.com</a> HTTP/1.0 ";
// 不要忽略空白字符
stream.unsetf(std::ios_base::skipws);
// 显示 stream 流里的内容
std::copy(std::istream_iterator<char>(stream),
std::istream_iterator<char>(),
std::ostream_iterator<char>(std::cout));
}
```

用 ASIO 编写 UDP 通信程序

ASIO 的 TCP 协议通过 boost::asio::ip 名空间下的 tcp 类进行通信，举一返三:ASIO 的 UDP 协议通过 boost::asio::ip 名空间下的 udp 类进行通信。

我 们知道 UDP 是基于数据报模式的，所以事先不需要建立连接。就象寄信一样，要寄给谁只要写上地址往门口的邮箱一丢，其它的事各级邮局 包办；要收信用只要看看自家信箱里有没有信件就行（或问门口传达室老大爷）。在 ASIO 里，就是 udp::socket 的 send_to 和 receive_from 方法（异步版本是 async_send_to 和 asnync_receive_from）。

下面的示例代码是从 ASIO 官方文档里拿来的（实在想不出更好的例子了:-P）：

服务器端代码

```
//
// server.cpp
// ~~~~~~~~~~
//
// Copyright (c) 2003-2008 Christopher M. Kohlhoff
// (chris at kohlhoff dot com)
//
// Distributed under the Boost Software License, Version 1.0.
// (See accompanying
// file LICENSE_1_0.txt or
// copy at <a href="http://www.boost.org/LICENSE_1_0.txt"
title="http://www.boost.org/LICENSE_1_0.txt">http://www.boost.org/LICENSE_1_0.t
xt</a>)
//

#include <ctime>
```

```
#include <iostream>
#include <string>
#include <boost/array.hpp>
#include <boost/asio.hpp>

using boost::asio::ip::udp;

std::string make_daytime_string()
{
using namespace std; // For time_t, time and ctime;
time_t now = time(0);
return ctime(&now);
}

int main()
{
try
{
boost::asio::io_service io_service;
// 在本机 13 端口建立一个 socket
udp::socket socket(io_service, udp::endpoint(udp::v4(), 13));

for (;;)
{
boost::array<char, 1> recv_buf;
udp::endpoint remote_endpoint;
boost::system::error_code error;
// 接收一个字符，这样就得到了远程端点(remote_endpoint)
socket.receive_from(boost::asio::buffer(recv_buf),
remote_endpoint, 0, error);

if (error && error != boost::asio::error::message_size)
throw boost::system::system_error(error);

std::string message = make_daytime_string();
// 向远程端点发送字符串 message(当前时间)
boost::system::error_code ignored_error;
socket.send_to(boost::asio::buffer(message),
remote_endpoint, 0, ignored_error);
}
}
catch (std::exception& e)
{
std::cerr << e.what() << std::endl;
```

```
}

return 0;
}
```

客户端代码

```
//
// client.cpp
// ~~~~~~~~~~
//
// Copyright (c) 2003-2008 Christopher M. Kohlhoff
// (chris at kohlhoff dot com)
//
// Distributed under the Boost Software License, Version 1.0.
// (See accompanying file LICENSE_1_0.txt or
// copy at <a href="http://www.boost.org/LICENSE_1_0.txt"
title="http://www.boost.org/LICENSE_1_0.txt">http://www.boost.org/LICENSE_1_0.t
xt</a>)
//

#include <iostream>
#include <boost/array.hpp>
#include <boost/asio.hpp>

using boost::asio::ip::udp;

int main(int argc, char* argv[])
{
try
{
if (argc != 2)
{
std::cerr << "Usage: client <host>" << std::endl;
return 1;
}

boost::asio::io_service io_service;
// 取得命令行参数对应的服务器端点
udp::resolver resolver(io_service);
udp::resolver::query query(udp::v4(), argv[1], "daytime");
udp::endpoint receiver_endpoint = *resolver.resolve(query);

udp::socket socket(io_service);
socket.open(udp::v4());
```

```
// 发送一个字节给服务器，让服务器知道我们的地址
boost::array<char, 1> send_buf = { 0 };
socket.send_to(boost::asio::buffer(send_buf), receiver_endpoint);
// 接收服务器发来的数据
boost::array<char, 128> recv_buf;
udp::endpoint sender_endpoint;
size_t len = socket.receive_from(
boost::asio::buffer(recv_buf), sender_endpoint);

std::cout.write(recv_buf.data(), len);
}
catch (std::exception& e)
{
std::cerr << e.what() << std::endl;
}

return 0;
}
```

用 ASIO 读写串行口

ASIO 不仅支持网络通信，还能支持串口通信。要让两个设备使用串口通信，关键是要设置好正确的参数，这些参数是：波特率、奇偶校验 位、停止位、字符大小和流量控制。两个串口设备只有设置了相同的参数才能互相交谈。

ASIO 提供了 boost::asio::serial_port 类，它有一个 set_option(const SettableSerialPortOption& option)方法就是用于设置上面列举的这些参数的，其中的 option 可以是：

serial_port::baud_rate 波特率，构造参数为 unsigned int
serial_port::parity 奇偶校验，构造参数为 serial_port::parity::type，enum 类型，可以是 none, odd, even。
serial_port::flow_control 流量控制，构造参数为 serial_port::flow_control::type，enum 类型，可以是 none software hardware
serial_port::stop_bits 停止位,构造参数为 serial_port::stop_bits::type,enum 类型，可以是 one onepointfive two
serial_port::character_size 字符大小，构造参数为 unsigned int

演示代码

```
#include <iostream>
#include <boost/asio.hpp>
#include <boost/bind.hpp>
```

```
using namespace std;
using namespace boost::asio;

int main(int argc, char* argv[])
{
io_service iosev;
// 串口 COM1, Linux 下为 “/dev/ttyS0”
serial_port sp(iosev, "COM1");
// 设置参数
sp.set_option(serial_port::baud_rate(19200));
sp.set_option(serial_port::flow_control(serial_port::flow_control::none));
sp.set_option(serial_port::parity(serial_port::parity::none));
sp.set_option(serial_port::stop_bits(serial_port::stop_bits::one));
sp.set_option(serial_port::character_size(8));
// 向串口写数据
write(sp, buffer("Hello world", 12));

// 向串口读数据
char buf[100];
read(sp, buffer(buf));

iosev.run();
return 0;
}
```

上面这段代码有个问题，read(sp, buffer(buf))非得读满 100 个字符才会返回，串口通信有时我们确实能知道对方发过来的字符长度，有时候是不能的。

如果知道对方发过来的数据里有分隔符的话（比如空格作为分隔），可以使用 read_until 来读，比如：

```
boost::asio::streambuf buf;
// 一直读到遇到空格为止
read_until(sp, buf, ' ');
copy(istream_iterator<char>(istream(&buf)>>noskipws),
istream_iterator<char>(),
ostream_iterator<char>(cout));
```

另外一个方法是使用前面说过的异步读写+超时的方式，代码如下：

```
#include <iostream>
#include <boost/asio.hpp>
#include <boost/bind.hpp>

using namespace std;
using namespace boost::asio;
```

```
void handle_read(char *buf,boost::system::error_code ec,
std::size_t bytes_transferred)
{
cout.write(buf, bytes_transferred);
}

int main(int argc, char* argv[])
{
io_service iosev;
serial_port sp(iosev, "COM1");
sp.set_option(serial_port::baud_rate(19200));
sp.set_option(serial_port::flow_control());
sp.set_option(serial_port::parity());
sp.set_option(serial_port::stop_bits());
sp.set_option(serial_port::character_size(8));

write(sp, buffer("Hello world", 12));

// 异步读
char buf[100];
async_read(sp, buffer(buf), boost::bind(handle_read, buf, _1, _2));
// 100ms 后超时
deadline_timer timer(iosev);
timer.expires_from_now(boost::posix_time::millisec(100));
// 超时后调用 sp 的 cancel()方法放弃读取更多字符
timer.async_wait(boost::bind(&serial_port::cancel, boost::ref(sp)));

iosev.run();
return 0;
}
```

boost：：asio：：error 的用法浅析

boost：：asio：：error 的用法浅析

boost：：asio：：error 的用法浅析

作者：转载自：asio 分享学习快乐更新时间：2009-8-2

一般而言我们创建用于接收 error 的类型大多声明如下：

boost::system::error_code error 我们用这个类型去接受在函数中产生的错误

如

```
socket.connect( endpoint, error);
```

如果连接失败，错误类型会保存到 error 中，比如连接主机失败可能会返回这样的错误

```
boost::asio::error::host_not_found;
```

通过 if （error）检测到 error 后，抛出异常

```
throw boost::system::system_error(error);
```

需要注意的是，我们的 error 被 转化成 system_error 了

显示错误很简单了，std::cout << e.what()

就哦啦。

大致的异常都是这个步骤进行的，

然而还有一点在异步调用的时候

产生的异常 error 的传递是个问题，因为异步会立刻返回，局部变量是会被销毁的，

boost::asio::placeholders::error,将会保存异常的状态，这样我们使用异步调用时如

socket::async_write_some 的时候不用自己创建 boost::system::error_co error 了，直接使用

boost::asio::placeholders::error 作为参数即可，

同理，我们 sync_write_some 需要返回读写数据的大小，令人开心的是

boost::asio::placeholders::bytes_transferred 直接作为参数就可以保存数据大小。

实例如下：

```
boost::asio::async_write(socket_, boost::asio::buffer(message_),
boost::bind(&tcp_connection::handle_write, shared_from_this(),
boost::asio::placeholders::error,
boost::asio::placeholders::bytes_transferred));
}
```

参考手册上说的很明确，
boost::asio::placeholders::error,boost::asio::placeholders::bytes_transferred
就是为异步调用使用 bind 的时候设计的。

当然了 boost::system::error_co error 还用有用的，同步调用的时候我们就用它作为参数

如：

```
boost::system::error_code error;

size_t len = socket.read_some(boost::asio::buffer(buf), error);
```

同样在异步调用的回调 handle 中也用它作参数如

```
void handle_write(const boost::system::error_code& /*error*/,

size_t /*bytes_transferred*/)
{
}
```

总结就是说异步就用
boost::asio::placeholders::error,boost::asio::placeholders::bytes_transferred

同步就用 boost::system::error_code