**使用Boost.Asio编写通信程序**

* **[asio](http://cpp.ezbty.org/category/cpp/asio" \o ")**
* [**boost**](http://cpp.ezbty.org/category/cpp/boost)
* [**同步**](http://cpp.ezbty.org/category/cpp/%E5%90%8C%E6%AD%A5)
* [**异步**](http://cpp.ezbty.org/category/cpp/%E5%BC%82%E6%AD%A5)
* [**网络编程**](http://cpp.ezbty.org/category/cpp/%E7%BD%91%E7%BB%9C%E7%BC%96%E7%A8%8B)
* [**输入输出**](http://cpp.ezbty.org/category/cpp/%E8%BE%93%E5%85%A5%E8%BE%93%E5%87%BA)
* [**通信**](http://cpp.ezbty.org/category/cpp/%E9%80%9A%E4%BF%A1)

**摘要**：本文通过形像而活泼的语言简单地介绍了Boost::asio库的使用，作为asio的一个入门介绍是非常合适的，可以给人一种新鲜的感觉，同时也能让体验到asio的主要内容。本文来自网络，原文在[这里](http://www.cppprog.com/2009/0720/133.html)。

**目录 [**[**隐藏**](http://cpp.ezbty.org/content/science_doc/%E4%BD%BF%E7%94%A8boostasio%E7%BC%96%E5%86%99%E9%80%9A%E4%BF%A1%E7%A8%8B%E5%BA%8F)**]**

1. [ASIO的同步方式](http://cpp.ezbty.org/content/science_doc/%E4%BD%BF%E7%94%A8boostasio%E7%BC%96%E5%86%99%E9%80%9A%E4%BF%A1%E7%A8%8B%E5%BA%8F#ASIO)
   1. [自我介绍](http://cpp.ezbty.org/content/science_doc/%E4%BD%BF%E7%94%A8boostasio%E7%BC%96%E5%86%99%E9%80%9A%E4%BF%A1%E7%A8%8B%E5%BA%8F#header-1)
   2. [示例代码](http://cpp.ezbty.org/content/science_doc/%E4%BD%BF%E7%94%A8boostasio%E7%BC%96%E5%86%99%E9%80%9A%E4%BF%A1%E7%A8%8B%E5%BA%8F#header-2)
   3. [小结](http://cpp.ezbty.org/content/science_doc/%E4%BD%BF%E7%94%A8boostasio%E7%BC%96%E5%86%99%E9%80%9A%E4%BF%A1%E7%A8%8B%E5%BA%8F#header-3)
2. [ASIO的异步方式](http://cpp.ezbty.org/content/science_doc/%E4%BD%BF%E7%94%A8boostasio%E7%BC%96%E5%86%99%E9%80%9A%E4%BF%A1%E7%A8%8B%E5%BA%8F#ASIO-4)
   1. [自我介绍](http://cpp.ezbty.org/content/science_doc/%E4%BD%BF%E7%94%A8boostasio%E7%BC%96%E5%86%99%E9%80%9A%E4%BF%A1%E7%A8%8B%E5%BA%8F#header-5)
   2. [示例代码](http://cpp.ezbty.org/content/science_doc/%E4%BD%BF%E7%94%A8boostasio%E7%BC%96%E5%86%99%E9%80%9A%E4%BF%A1%E7%A8%8B%E5%BA%8F#header-6)
   3. [小结](http://cpp.ezbty.org/content/science_doc/%E4%BD%BF%E7%94%A8boostasio%E7%BC%96%E5%86%99%E9%80%9A%E4%BF%A1%E7%A8%8B%E5%BA%8F#header-7)
3. [ASIO的“便民措施”](http://cpp.ezbty.org/content/science_doc/%E4%BD%BF%E7%94%A8boostasio%E7%BC%96%E5%86%99%E9%80%9A%E4%BF%A1%E7%A8%8B%E5%BA%8F#ASIO-8)
   1. [端点](http://cpp.ezbty.org/content/science_doc/%E4%BD%BF%E7%94%A8boostasio%E7%BC%96%E5%86%99%E9%80%9A%E4%BF%A1%E7%A8%8B%E5%BA%8F#header-9)
   2. [超时](http://cpp.ezbty.org/content/science_doc/%E4%BD%BF%E7%94%A8boostasio%E7%BC%96%E5%86%99%E9%80%9A%E4%BF%A1%E7%A8%8B%E5%BA%8F#header-10)
   3. [统一读写接口](http://cpp.ezbty.org/content/science_doc/%E4%BD%BF%E7%94%A8boostasio%E7%BC%96%E5%86%99%E9%80%9A%E4%BF%A1%E7%A8%8B%E5%BA%8F#header-11)
   4. [基于流的操作](http://cpp.ezbty.org/content/science_doc/%E4%BD%BF%E7%94%A8boostasio%E7%BC%96%E5%86%99%E9%80%9A%E4%BF%A1%E7%A8%8B%E5%BA%8F#header-12)
4. [用ASIO编写UDP通信程序](http://cpp.ezbty.org/content/science_doc/%E4%BD%BF%E7%94%A8boostasio%E7%BC%96%E5%86%99%E9%80%9A%E4%BF%A1%E7%A8%8B%E5%BA%8F#ASIOUDP)
5. [用ASIO读写串行口](http://cpp.ezbty.org/content/science_doc/%E4%BD%BF%E7%94%A8boostasio%E7%BC%96%E5%86%99%E9%80%9A%E4%BF%A1%E7%A8%8B%E5%BA%8F#ASIO-13)
   1. [演示代码](http://cpp.ezbty.org/content/science_doc/%E4%BD%BF%E7%94%A8boostasio%E7%BC%96%E5%86%99%E9%80%9A%E4%BF%A1%E7%A8%8B%E5%BA%8F#header-14)

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)。

**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。

**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.txt</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.txt</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;  
}