1. [序言](http://wiki.woodpecker.org.cn/moin/Boost" \l "head-c08365b8def7108269c81c21675b9424274cf23d)
2. [编译:VC2005注意](http://wiki.woodpecker.org.cn/moin/Boost#head-30d7247306b57c6bae058aa68b13bf19d07eacbb)
3. [Asio 网络库](http://wiki.woodpecker.org.cn/moin/Boost#head-33075201187f468f843dd3c4a6abd51a84e63a07)
   1. [网络库:VC2005注意](http://wiki.woodpecker.org.cn/moin/Boost#head-595b01cdbd8eb628689c40405f9831de27d045a4)
   2. [同步Timer](http://wiki.woodpecker.org.cn/moin/Boost#head-a24d7514a656cc0e3961c2f560dd6eda5e0eef99)
   3. [异步Timer](http://wiki.woodpecker.org.cn/moin/Boost#head-6b4c3414fe12f2410cef07f7dcb6dbb6fce10025)
   4. [回调函数的参数](http://wiki.woodpecker.org.cn/moin/Boost#head-e44289723876436a8301395febfe0219502e8931)
   5. [成员函数作为回调函数](http://wiki.woodpecker.org.cn/moin/Boost#head-c6876ff526ef494aabfe9d6ac2703b19cc030b1c)
   6. [多线程回调同步](http://wiki.woodpecker.org.cn/moin/Boost#head-99fe06fbb8967d6982fbfc1ddccd58afd57250ce)
   7. [TCP客户端:对准时间](http://wiki.woodpecker.org.cn/moin/Boost#head-14b9a3f2af1bde4d5477a1335152d3f582ca7d4a)
   8. [TCP同步时间服务器](http://wiki.woodpecker.org.cn/moin/Boost#head-ea1a8a1031b51f07080ff4a99ca1eaba714ccf8f)
4. [Filesystem Library 文件系统](http://wiki.woodpecker.org.cn/moin/Boost#head-4339b3ec71766eec88e8686c7ffa91c30fa48e91)

参考:BOOST文档

* -- 欢迎转载,但请保留引用网址以获得更新

**1. 序言**

现在学的东西很容易忘记,写这篇文章的目的是能让我在需要时快速找回当时的感觉. Let's BOOST THE WORLD .

**2. 编译:VC2005注意**

在 属性->C/C++->预处理器->预处理定义 中加入

\_CRT\_SECURE\_NO\_DEPRECATE;

来屏蔽不必要的警告

**3. Asio 网络库**

Boost.Asio是利用当代C++的先进方法,跨平台,异步I/O模型的C++网络库.

**3.1. 网络库:VC2005注意**

在 属性->C/C++->命令行 中加入

-DBOOST\_REGEX\_NO\_LIB

来防止自动连接.

**3.2. 同步Timer**

本章介绍asio如何在定时器上进行阻塞等待(blocking wait).

实现,我们包含必要的头文件.

所有的asio类可以简单的通过include "asio.hpp"来调用.

#include <iostream>

#include <boost/asio.hpp>

此外,这个示例用到了timer,我们还要包含Boost.Date\_Time的头文件来控制时间.

#include <boost/date\_time/posix\_time/posix\_time.hpp>

使用asio至少需要一个boost::asio::io\_service对象.该类提供了访问I/O的功能.我们首先在main函数中声明它.

int main()

{

boost::asio::io\_service io;

下一步我们声明boost::asio::deadline\_timer对象.这个asio的核心类提供I/O的功能(这里更确切的说是定时功能),总是把一个io\_service对象作为他的第一个构造函数,而第二个构造函数的参数设定timer会在5秒后到时(expired).

boost::asio::deadline\_timer t(io, boost::posix\_time::seconds(5));

这个简单的示例中我们演示了定时器上的一个阻塞等待.就是说,调用boost::asio::deadline\_timer::wait()的在创建后5秒内(注意:不是等待开始后),timer到时之前不会返回任何值.

一个deadline\_timer只有两种状态:到时,未到时.

如果boost::asio::deadline\_timer::wait()在到时的timer对象上调用,会立即return.

t.wait();

最后,我们输出理所当然的"Hello, world!"来演示timer到时了.

std::cout << "Hello, world!\n";

return 0;

}

完整的代码:

#include <iostream>

#include <boost/asio.hpp>

#include <boost/date\_time/posix\_time/posix\_time.hpp>

int main()

{

boost::asio::io\_service io;

boost::asio::deadline\_timer t(io, boost::posix\_time::seconds(5));

t.wait();

std::cout << "Hello, world!\n";

return 0;

}

**3.3. 异步Timer**

#include <iostream>

#include <asio.hpp>

#include <boost/date\_time/posix\_time/posix\_time.hpp>

asio的异步函数会在一个异步操作完成后被回调.这里我们定义了一个将被回调的函数.

void print(const asio::error& /\*e\*/)

{

std::cout << "Hello, world!\n";

}

int main()

{

asio::io\_service io;

asio::deadline\_timer t(io, boost::posix\_time::seconds(5));

这里我们调用asio::deadline\_timer::async\_wait()来异步等待

t.async\_wait(print);

最后,我们必须调用asio::io\_service::run().

asio库只会调用那个正在运行的asio::io\_service::run()的回调函数.

如果asio::io\_service::run()不被调用,那么回调永远不会发生.

asio::io\_service::run()会持续工作到点,这里就是timer到时,回调完成.

别忘了在调用 asio::io\_service::run()之前设置好io\_service的任务.比如,这里,如果我们忘记先调用asio::deadline\_timer::async\_wait()则asio::io\_service::run()会在瞬间return.

io.run();

return 0;

}

完整的代码:

#include <iostream>

#include <asio.hpp>

#include <boost/date\_time/posix\_time/posix\_time.hpp>

void print(const asio::error& /\*e\*/)

{

std::cout << "Hello, world!\n";

}

int main()

{

asio::io\_service io;

asio::deadline\_timer t(io, boost::posix\_time::seconds(5));

t.async\_wait(print);

io.run();

return 0;

}

**3.4. 回调函数的参数**

这里我们将每秒回调一次,来演示如何回调函数参数的含义

#include <iostream>

#include <asio.hpp>

#include <boost/bind.hpp>

#include <boost/date\_time/posix\_time/posix\_time.hpp>

首先,调整一下timer的持续时间,开始一个异步等待.显示,回调函数需要访问timer来实现周期运行,所以我们再介绍两个新参数

* 指向timer的指针
* 一个int\*来指向计数器

void print(const asio::error& /\*e\*/,

asio::deadline\_timer\* t, int\* count)

{

我们打算让这个函数运行6个周期,然而你会发现这里没有显式的方法来终止io\_service.不过,回顾上一节,你会发现当asio::io\_service::run()会在所有任务完成时终止.这样我们当计算器的值达到5时(0为第一次运行的值),不再开启一个新的异步等待就可以了.

if (\*count < 5)

{

std::cout << \*count << "\n";

++(\*count);

然后,我们推迟的timer的终止时间.通过在原先的终止时间上增加延时,我们可以确保timer不会在处理回调函数所需时间内的到期.

(原文:By calculating the new expiry time relative to the old, we can ensure that the timer does not drift away from the whole-second mark due to any delays in processing the handler.)

t->expires\_at(t->expires\_at() + boost::posix\_time::seconds(1));

然后我们开始一个新的同步等待.如您所见,我们用把print和他的多个参数用boost::bind函数合成一个的形为void(const asio::error&)回调函数(准确的说是function object).

在这个例子中, boost::bind的asio::placeholders::error参数是为了给回调函数传入一个error对象.当进行一个异步操作,开始boost::bind时,你需要使用它来匹配回调函数的参数表.下一节中你会学到回调函数不需要error参数时可以省略它.

t->async\_wait(boost::bind(print,

asio::placeholders::error, t, count));

}

}

int main()

{

asio::io\_service io;

int count = 0;

asio::deadline\_timer t(io, boost::posix\_time::seconds(1));

和上面一样,我们再一次使用了绑定asio::deadline\_timer::async\_wait()

t.async\_wait(boost::bind(print,

asio::placeholders::error, &t, &count));

io.run();

在结尾,我们打印出的最后一次没有设置timer的调用的count的值

std::cout << "Final count is " << count << "\n";

return 0;

}

完整的代码:

#include <iostream>

#include <asio.hpp>

#include <boost/bind.hpp>

#include <boost/date\_time/posix\_time/posix\_time.hpp>

void print(const asio::error& /\*e\*/,

asio::deadline\_timer\* t, int\* count)

{

if (\*count < 5)

{

std::cout << \*count << "\n";

++(\*count);

t->expires\_at(t->expires\_at() + boost::posix\_time::seconds(1));

t->async\_wait(boost::bind(print,

asio::placeholders::error, t, count));

}

}

int main()

{

asio::io\_service io;

int count = 0;

asio::deadline\_timer t(io, boost::posix\_time::seconds(1));

t.async\_wait(boost::bind(print,

asio::placeholders::error, &t, &count));

io.run();

std::cout << "Final count is " << count << "\n";

return 0;

}

**3.5. 成员函数作为回调函数**

本例的运行结果和上一节类似

#include <iostream>

#include <boost/asio.hpp>

#include <boost/bind.hpp>

#include <boost/date\_time/posix\_time/posix\_time.hpp>

我们先定义一个printer类

class printer

{

public:

构造函数有一个io\_service参数,并且在初始化timer\_时用到了它.用来计数的count\_这里同样作为了成员变量

printer(boost::asio::io\_service& io)

: timer\_(io, boost::posix\_time::seconds(1)),

count\_(0)

{

boost::bind同样可以出色的工作在成员函数上.众所周知,所有的非静态成员函数都有一个隐式的this参数,我们需要把this作为参数bind到成员函数上.和上一节类似,我们再次用bind构造出void(const boost::asio::error&)形式的函数.

注意,这里没有指定boost::asio::placeholders::error占位符,因为这个print成员函数没有接受一个error对象作为参数.

timer\_.async\_wait(boost::bind(&printer::print, this));

}

在类的折构函数中我们输出最后一次回调的conut的值

~printer()

{

std::cout << "Final count is " << count\_ << "\n";

}

print函数于上一节的十分类似,但是用成员变量取代了参数.

void print()

{

if (count\_ < 5)

{

std::cout << count\_ << "\n";

++count\_;

timer\_.expires\_at(timer\_.expires\_at() + boost::posix\_time::seconds(1));

timer\_.async\_wait(boost::bind(&printer::print, this));

}

}

private:

boost::asio::deadline\_timer timer\_;

int count\_;

};

现在main函数清爽多了,在运行io\_service之前只需要简单的定义一个printer对象.

int main()

{

boost::asio::io\_service io;

printer p(io);

io.run();

return 0;

}

完整的代码:

#include <iostream>

#include <boost/asio.hpp>

#include <boost/bind.hpp>

#include <boost/date\_time/posix\_time/posix\_time.hpp>

class printer

{

public:

printer(boost::asio::io\_service& io)

: timer\_(io, boost::posix\_time::seconds(1)),

count\_(0)

{

timer\_.async\_wait(boost::bind(&printer::print, this));

}

~printer()

{

std::cout << "Final count is " << count\_ << "\n";

}

void print()

{

if (count\_ < 5)

{

std::cout << count\_ << "\n";

++count\_;

timer\_.expires\_at(timer\_.expires\_at() + boost::posix\_time::seconds(1));

timer\_.async\_wait(boost::bind(&printer::print, this));

}

}

private:

boost::asio::deadline\_timer timer\_;

int count\_;

};

int main()

{

boost::asio::io\_service io;

printer p(io);

io.run();

return 0;

}

**3.6. 多线程回调同步**

本节演示了使用boost::asio::strand在多线程程序中进行回调同步(synchronise).

先前的几节阐明了如何在单线程程序中用boost::asio::io\_service::run()进行同步.如您所见,asio库确保 仅当 当前线程调用boost::asio::io\_service::run()时产生回调.显然,仅在一个线程中调用boost::asio::io\_service::run() 来确保回调是适用于并发编程的.

一个基于asio的程序最好是从单线程入手,但是单线程有如下的限制,这一点在服务器上尤其明显:

* 当回调耗时较长时,反应迟钝.
* 在多核的系统上无能为力

如果你发觉你陷入了这种困扰,可以替代的方法是建立一个boost::asio::io\_service::run()的线程池.然而这样就允许回调函数并发执行.所以,当回调函数需要访问一个共享,线程不安全的资源时,我们需要一种方式来同步操作.

#include <iostream>

#include <boost/asio.hpp>

#include <boost/thread.hpp>

#include <boost/bind.hpp>

#include <boost/date\_time/posix\_time/posix\_time.hpp>

在上一节的基础上我们定义一个printer类,此次,它将并行运行两个timer

class printer

{

public:

除了声明了一对boost::asio::deadline\_timer,构造函数也初始化了类型为boost::asio::strand的strand\_成员.

boost::asio::strand可以分配的回调函数.它保证无论有多少线程调用了boost::asio::io\_service::run(),下一个回调函数仅在前一个回调函数完成后开始,当然回调函数仍然可以和那些不使用boost::asio::strand分配,或是使用另一个boost::asio::strand分配的回调函数一起并发执行.

printer(boost::asio::io\_service& io)

: strand\_(io),

timer1\_(io, boost::posix\_time::seconds(1)),

timer2\_(io, boost::posix\_time::seconds(1)),

count\_(0)

{

当一个异步操作开始时,用boost::asio::strand来 "wrapped(包装)"回调函数.boost::asio::strand::wrap()会返回一个由boost::asio::strand分配的新的handler(句柄),这样,我们可以确保它们不会同时运行.

timer1\_.async\_wait(strand\_.wrap(boost::bind(&printer::print1, this)));

timer2\_.async\_wait(strand\_.wrap(boost::bind(&printer::print2, this)));

}

~printer()

{

std::cout << "Final count is " << count\_ << "\n";

}

多线程程序中,回调函数在访问共享资源前需要同步.这里共享资源是std::cout 和count\_变量.

void print1()

{

if (count\_ < 10)

{

std::cout << "Timer 1: " << count\_ << "\n";

++count\_;

timer1\_.expires\_at(timer1\_.expires\_at() + boost::posix\_time::seconds(1));

timer1\_.async\_wait(strand\_.wrap(boost::bind(&printer::print1, this)));

}

}

void print2()

{

if (count\_ < 10)

{

std::cout << "Timer 2: " << count\_ << "\n";

++count\_;

timer2\_.expires\_at(timer2\_.expires\_at() + boost::posix\_time::seconds(1));

timer2\_.async\_wait(strand\_.wrap(boost::bind(&printer::print2, this)));

}

}

private:

boost::asio::strand strand\_;

boost::asio::deadline\_timer timer1\_;

boost::asio::deadline\_timer timer2\_;

int count\_;

};

main函数中boost::asio::io\_service::run()在两个线程中被调用:主线程,一个boost::thread线程.

正如单线程中那样,并发的boost::asio::io\_service::run()会一直运行直到完成任务.后台的线程将在所有异步线程完成后终结.

int main()

{

boost::asio::io\_service io;

printer p(io);

boost::thread t(boost::bind(&boost::asio::io\_service::run, &io));

io.run();

t.join();

return 0;

}

完整的代码:

#include <iostream>

#include <boost/asio.hpp>

#include <boost/thread.hpp>

#include <boost/bind.hpp>

#include <boost/date\_time/posix\_time/posix\_time.hpp>

class printer

{

public:

printer(boost::asio::io\_service& io)

: strand\_(io),

timer1\_(io, boost::posix\_time::seconds(1)),

timer2\_(io, boost::posix\_time::seconds(1)),

count\_(0)

{

timer1\_.async\_wait(strand\_.wrap(boost::bind(&printer::print1, this)));

timer2\_.async\_wait(strand\_.wrap(boost::bind(&printer::print2, this)));

}

~printer()

{

std::cout << "Final count is " << count\_ << "\n";

}

void print1()

{

if (count\_ < 10)

{

std::cout << "Timer 1: " << count\_ << "\n";

++count\_;

timer1\_.expires\_at(timer1\_.expires\_at() + boost::posix\_time::seconds(1));

timer1\_.async\_wait(strand\_.wrap(boost::bind(&printer::print1, this)));

}

}

void print2()

{

if (count\_ < 10)

{

std::cout << "Timer 2: " << count\_ << "\n";

++count\_;

timer2\_.expires\_at(timer2\_.expires\_at() + boost::posix\_time::seconds(1));

timer2\_.async\_wait(strand\_.wrap(boost::bind(&printer::print2, this)));

}

}

private:

boost::asio::strand strand\_;

boost::asio::deadline\_timer timer1\_;

boost::asio::deadline\_timer timer2\_;

int count\_;

};

int main()

{

boost::asio::io\_service io;

printer p(io);

boost::thread t(boost::bind(&boost::asio::io\_service::run, &io));

io.run();

t.join();

return 0;

}

**3.7. TCP客户端:对准时间**

#include <iostream>

#include <boost/array.hpp>

#include <boost/asio.hpp>

本程序的目的是访问一个时间同步服务器,我们需要用户指定一个服务器(如time-nw.nist.gov),用IP亦可.

(译者注:日期查询协议,这种时间传输协议不指定固定的传输格式，只要求按照ASCII标准发送数据。)

using boost::asio::ip::tcp;

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

{

try

{

if (argc != 2)

{

std::cerr << "Usage: client <host>" << std::endl;

return 1;

}

用asio进行网络连接至少需要一个boost::asio::io\_service对象

boost::asio::io\_service io\_service;

我们需要把在命令行参数中指定的服务器转换为TCP上的节点.完成这项工作需要boost::asio::ip::tcp::resolver对象

tcp::resolver resolver(io\_service);

一个resolver对象查询一个参数,并将其转换为TCP上节点的列表.这里我们把argv[1]中的sever的名字和要查询字串daytime关联.

tcp::resolver::query query(argv[1], "daytime");

节点列表可以用 boost::asio::ip::tcp::resolver::iterator 来进行迭代.iterator默认的构造函数生成一个end iterator.

tcp::resolver::iterator endpoint\_iterator = resolver.resolve(query);

tcp::resolver::iterator end;

现在我们建立一个连接的sockert,由于获得节点既有IPv4也有IPv6的.所以,我们需要依次尝试他们直到找到一个可以正常工作的.这步使得我们的程序独立于IP版本

tcp::socket socket(io\_service);

boost::asio::error error = boost::asio::error::host\_not\_found;

while (error && endpoint\_iterator != end)

{

socket.close();

socket.connect(\*endpoint\_iterator++, boost::asio::assign\_error(error));

}

if (error)

throw error;

连接完成,我们需要做的是读取daytime服务器的响应.

我们用boost::array来保存得到的数据,boost::asio::buffer()会自动根据array的大小暂停工作,来防止缓冲溢出.除了使用boost::array,也可以使用char [] 或std::vector.

for (;;)

{

boost::array<char, 128> buf;

boost::asio::error error;

size\_t len = socket.read\_some(

boost::asio::buffer(buf), boost::asio::assign\_error(error));

当服务器关闭连接时,boost::asio::ip::tcp::socket::read\_some()会用boost::asio::error::eof标志完成, 这时我们应该退出读取循环了.

if (error == boost::asio::error::eof)

break; // Connection closed cleanly by peer.

else if (error)

throw error; // Some other error.

std::cout.write(buf.data(), len);

}

如果发生了什么异常我们同样会抛出它

}

catch (std::exception& e)

{

std::cerr << e.what() << std::endl;

}

运行示例:在windowsXP的cmd窗口下

输入:upload.exe time-a.nist.gov

输出:54031 06-10-23 01:50:45 07 0 0 454.2 UTC(NIST) \*

完整的代码:

#include <iostream>

#include <boost/array.hpp>

#include <asio.hpp>

using asio::ip::tcp;

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

{

try

{

if (argc != 2)

{

std::cerr << "Usage: client <host>" << std::endl;

return 1;

}

asio::io\_service io\_service;

tcp::resolver resolver(io\_service);

tcp::resolver::query query(argv[1], "daytime");

tcp::resolver::iterator endpoint\_iterator = resolver.resolve(query);

tcp::resolver::iterator end;

tcp::socket socket(io\_service);

asio::error error = asio::error::host\_not\_found;

while (error && endpoint\_iterator != end)

{

socket.close();

socket.connect(\*endpoint\_iterator++, asio::assign\_error(error));

}

if (error)

throw error;

for (;;)

{

boost::array<char, 128> buf;

asio::error error;

size\_t len = socket.read\_some(

asio::buffer(buf), asio::assign\_error(error));

if (error == asio::error::eof)

break; // Connection closed cleanly by peer.

else if (error)

throw error; // Some other error.

std::cout.write(buf.data(), len);

}

}

catch (std::exception& e)

{

std::cerr << e.what() << std::endl;

}

return 0;

}

**3.8. TCP同步时间服务器**

#include <ctime>

#include <iostream>

#include <string>

#include <asio.hpp>

using asio::ip::tcp;

我们先定义一个函数返回当前的时间的string形式.这个函数会在我们所有的时间服务器示例上被使用.

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

{

asio::io\_service io\_service;

新建一个asio::ip::tcp::acceptor对象来监听新的连接.我们监听TCP端口13,IP版本为V4

tcp::acceptor acceptor(io\_service, tcp::endpoint(tcp::v4(), 13));

这是一个iterative server,也就是说同一时间只能处理一个连接.建立一个socket来表示一个和客户端的连接, 然后等待客户端的连接.

for (;;)

{

tcp::socket socket(io\_service);

acceptor.accept(socket);

当客户端访问服务器时,我们获取当前时间,然后返回它.

std::string message = make\_daytime\_string();

asio::write(socket, asio::buffer(message),

asio::transfer\_all(), asio::ignore\_error());

}

}

最后处理异常

catch (std::exception& e)

{

std::cerr << e.what() << std::endl;

}

return 0;

}

运行示例:运行服务器,然后运行上一节的客户端,在windowsXP的cmd窗口下

输入:client.exe 127.0.0.1

输出:Mon Oct 23 09:44:48 2006

完整的代码:

#include <ctime>

#include <iostream>

#include <string>

#include <asio.hpp>

using asio::ip::tcp;

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

{

asio::io\_service io\_service;

tcp::acceptor acceptor(io\_service, tcp::endpoint(tcp::v4(), 13));

for (;;)

{

tcp::socket socket(io\_service);

acceptor.accept(socket);

std::string message = make\_daytime\_string();

asio::write(socket, asio::buffer(message),

asio::transfer\_all(), asio::ignore\_error());

}

}

catch (std::exception& e)

{

std::cerr << e.what() << std::endl;

}

return 0;

}