Boost 库 学习指南和说明文档

作者：刘刚 email:ganghust@gmail.com
个人主页 ： http://hustlg.bokee.com 2007 年 11 月 17 号

Boost 中文站

Boost库是一个经过千锤百炼、可移植、提供源代码的C++库，作为标准库的后备，是C++标准化进程的发动机之一。 Boost库由C++标准委员会库工作组成员发起，在C++社区中影响甚大，其成员已近2000人。 Boost库为我们带来了最新、最酷、最实用的技术，是不折不扣的“准”标准库。本站主要介绍Boost相关的中文技术文档。

Boost 入门

C++ Boost 学习资源列表

boost库简介
Windows和Solaris上Boost安装和编译
走进Boost(Boost使用入门)

Boost 编程技术

Boost中的智能指针

C++ Boost Thread线程编程指南 NEW

Boost 中文文档

C++ Boost库文档索引文档 NEW

C++ Boost Assign 文档 NEW
C++ Boost Regex 文档

C++ Boost Array 文档

Boost 源码剖析

Boost源码剖析之：容器赋值—assign NEW

Boost源码剖析之：型别分类器—type_traits

Boost源码剖析之：泛型指针类any之海纳百川

Boost源码剖析之：增强的std::pair——Tuple Types

C++ Boost 学习资源列表

boost的老巢 http://www.boost.org

boost的中文站： http://www.c-view.org

CSDN--Boost 系列专题 http://www.csdn.net/Subject/336/index.shtm

Linux伊甸园论坛- STL/boost专区 http://www.linuxeden.com/forum/forumdisplay.php?f=37

dozb的blog http://dozb.blogchina.com

C++ Boost 库简介

boost是一个准标准库，相当于STL的延续和扩充，它的设计理念和STL比较接近，都是利用泛型让复用达到最大化。不过对比STL，boost更加实用。STL集中在算法部分，而boost包含了不少工具类，可以完成比较具体的工作。

boost主要包含一下几个大类：字符串及文本处理、容器、迭代子(Iterator)、算法、函数对象和高阶编程、泛型编程、模板元编程、预处理元编程、并发编程、数学相关、纠错和测试、数据结构、输入/输出、跨语言支持、内存相关、语法分析、杂项。 有一些库是跨类别包含的，就是既属于这个类别又属于那个类别。

在文本处理部分，conversion/lexcial_cast类用于“用C++”的方法实现数字类型和字符串之间的转换。 主要是替代C标准库中的 atoi、 itoa之类的函数。当然其中一个最大的好处就是支持泛型了。

format库提供了对流的“printf-like”功能。printf里使用%d、%s等等的参数做替换的方法在很多情况下还是非常方便的，STL的iostream则缺乏这样的功能。format为stream增加了这个功能，并且功能比原始的printf更强。

regex，这个不多说了，正则表达式库。如果需要做字符串分析的人就会理解正则表达式有多么有用了。

spirit，这个是做LL分析的框架，可以根据EBNF规则对文件进行分析。（不要告诉我不知道什么是EBNF）。做编译器的可能会用到。一般人不太用的到。

tokenizer库。我以前经常在CSDN上看到有人问怎么把一个字符串按逗号分割成字符串数组。也许有些人很羡慕VB的split函数。现在，boost的tokenizer也有相同的功能了，如果我没记错的话，这个tokenizer还支持正则表达式，是不是很爽？

array: 提供了常量大小的数组的一个包装，喜欢用数组但是苦恼数组定位、确定数组大小等功能的人这下开心了。

dynamic_bitset，动态分配大小的bitset，我们知道STL里有个bitset，为位运算提供了不少方便。可惜它的大小需要在编译期指定。现在好了，运行期动态分配大小的bitset来了。

graph。提供了图的容器和相关算法。我还没有在程序中用到过图，需要用的人可以看看。

multi_array提供了对多维数组的封装，应该还是比较有用的。

并发编程里只有一个库，thread，提供了一个可移植的线程库，不过在Windows平台上我感觉用处不大。因为它是基于Posix线程的，在Windows里对Posix的支持不是很好。

接下来的 数学和数值 类里，包含了很多数值处理方面的类库，数学类我也不太熟，不过这里有几个类还是很有用的，比如rational分数类，random随机数类，等等。

static_assert，提供了编译器的assert功能。

test库，一个单元测试框架，非常不错。

concept_check提供了泛型编程时，对泛型量的一点检查，不是很完善，不过比没有好。

数据类型类any，一个安全的可以包含不同对象的类。把它作为容器的元素类型，那么这个容器就可以包含不同类型的元素。比用void *要安全。

compressed_pair，跟STL里的pair差不多。不过对空元素做了优化。

tuple，呵呵，也许是某些人梦寐以求的东西。可以让函数返回多个值。

跨语言支持：python，呵呵，好东东啊，可以将C++的类和函数映射给python使用。以下为几个CSDN上的关于boost.python的中文资料：http://dev.csdn.net/article/19/19828.shtm，http://dev.csdn.net/article/19/19829.shtm，http://dev.csdn.net/article/19/19830.shtm，http://dev.csdn.net/article/19/19831.shtm

pool:内存池，呵呵，不用害怕频繁分配释放内存导致内存碎片，也不用自己辛辛苦苦自己实现了。

smart_ptr:智能指针，这下不用担心内存泄漏的问题了吧。不过，C++里的智能指针都还不是十全十美的，用的时候小心点了，不要做太技巧性的操作了。

date_time，这个是平台、类库无关的实现，如果程序需要跨平台，可以考虑用这个。

timer，提供了一个计时器，虽然不是Windows里那种基于消息的计时器，不过据说可以用来测量语句执行时间。

uitlity里提供了一个noncopyable类，可以实现“无法复制”的类。很多情况下，我们需要避免一个类被复制，比如代表文件句柄的类，文件句柄如果被两个实例共享，操作上会有很多问题，而且语义上也说不过去。一般的避免实例复制的方法是把拷贝构造和operator=私有化，现在只要继承一下这个类就可以了，清晰了很多。

value_initialized：数值初始化，可以保证声明的对象都被明确的初始化，不过这个真的实用吗？似乎写这个比直接写初始化还累。呵呵，仁者见仁了。

这里面除了regex、python和test需要编译出库才能用，其他的大部分都可以直接源代码应用，比较方便。其实这些库使用都不难。最主要的原因是有些库的使用需要有相关的背景知识，比如元编程、STL、泛型编程等等。

Windows 和 Solaris 上 Boost 安装和编译

作者:大卫

0 前言

大卫注：这是当初研究boost时的笔记，最近看到论坛上有人问，所以就贴出来共享一下。其实个人认为，boost目前还不适于进行应用开发，毕竟 boost库太大了（当然，你可以只用一部分，但程序的可维护性始终是个问题），除非你想一探C++研究前沿的Meta Programming这个Generic Programming的神奇世界。强烈建议boost的研究者在研究boost之前研究一下一个小得多的模板库loki，boost中的很多让你无法理解的技术在loki库中被大量运用，并且这个库的作者专门写了

Modern C++ Design来解说该库的实现。此外，如果你要研究boost，开始时不要编译所有的库，如Python，thread，test等，因为等你花几个小时编译完了，你可能发现，你根本就用不到这些库，或者对它根本就不感兴趣，等到你研究完比较小的几个库，对boost有了充分了解的时候再来编译也不迟。

注：

开始前请确认你的OS中已经安装了适当的编译器，以下Windows环境中以Windows 2000 + VC6为例，Unix环境中以Solaris 9 + GCC 3.4.2为例；

以下以$BOOSTDIR表示boost的存放目录，请自行根据实际情况进行修改。

1 下载 Boost + 解包（略）

2 编译 jam

2.1Windows

到 $BOOSTDIR\tools\build\jam_src 下执行 build.bat 对 jam 进行编译，编译结果将存放在 $BOOSTDIR\ tools\build\jam_src\bin.ntx86下。如果你在执行该批处理程序过程中遇到问题，如报告无法找到编译器相关程序，请执行 X:\Program Files\Microsoft Visual Studio\VC98\Bin\VCVARS32.bat 以建立VC的基本环境变量。

2.2 Solaris 9

到 $BOOSTDIR\tools\build\jam_src 下执行 ./build.sh 对 jam 进行编译，编译结果将存放在 $BOOSTDIR\tools\build\jam_src\bin.solarisx86下。

3 设置环境变量

（注：这一步其实可以省略，直接在（三）中通过-s输入到命令行即可，但设置可以让命令行更清晰、简单一点。）

3.1 Windows

我的电脑点右键->属性->高级->环境变量->user variable或system variable中：

PATH最后添加bjam存放目录，如：
$BOOSTDIR\tools\build\jam_src\bin.ntx86
新建环境变量MSVCDIR，并在变量值一栏中填入VC安装目录，如：
X:\Program Files\Microsoft Visual Studio\VC98
新建环境变量：
PYTHON_ROOT=X:\Program Files\Python2.3.4
PYTHON_VERSION=2.3

3.2 Solaris 9

在.profile中PATH后添加编译后的jam的存放目录。
并增加
PYTHON_VERSION=2.3
export PYTHON_VERSION
注意，无需设置PYTHON_ROOT，Solaris下jam会自动处理。

4 编译 Boost

4.1 Windows

命令：

jam -sBOOST_ROOT=. -sTOOLS=msvc "-sBUILD=debug release <runtime-link>static/dynamic"

以上命令解释如下：

-s 即set，设置环境变量；

BOOST_ROOT boost的存放目录

TOOLS 你选择的toolset，如gcc、msvc（即vc6）、vc7.1，此外还有gcc-stlport、msvc-stlport、vc7.1-stlport，表示同时使用stlport。具体支持何种toolset，大家可以自行到$BOOSTDIR\tools\build\v1看个究竟。 BUILD 编译类型，上述选项表示编译出支持static和dynamic链接的debug和release版本（4个版本）。

编译后的lib、dll将被copy到$BOOSTDIR\bin\boost\libs目录下，但是这些lib、dll分散在不同的目录下，为了便于使用，可以在上述目录下分别查找*.lib和*.dll找出这些文件，然后将他们分别全部copy到VC的lib目录和Windows的System32目录，也可以自己建立一个专门用于存放boost的lib文件的目录，然后依次选择Tools->Options->Directories->Library files，将上述目录路径添加到VC的环境设置中。

4.2 Solaris 9

到$BOOSTDIR下执行以下命令：

jam -sBOOST_ROOT=. -sTOOLS=gcc "-sBUILD=debug release <runtime-link>static/dynamic"

但建议用如下命令：

jam -sBOOST_ROOT=. -sTOOLS=gcc "-sBUILD=release <runtime-link>dynamic speed"

这样可以极大加快编译的速度，同时，个人认为像boost这样大的库，最好还是采用动态链接以减小目标程序的size，就像libstdc++，还没有见过有人去静态链接libstdc++.a，虽然系统中提供了这个静态库。

走进 Boost [Boost 使用入门]

0 摘要

一直流传这么一个说法，想成为高手，一定要多读高手写的源代码。哪些代码是好材料呢？C++标准库的源代码？不，如果您读过，就会发现：要么是各种实现独有的表达方式让人摸不着头脑，要么是恐怖的代码风格（如到处是下划线）憋得人难受。Boost库的代码则相当清晰，注释合理，命名规范，绝对是适合阅读的典范。同时，Boost内容广泛，数值计算、泛型编程、元编程、平台API……不妨从容选择自己感兴趣的部分，细细品味。

在本文中，我们将会介绍了Boost库的下载与安装，并将体验Boost库中一个非常简单实用的组件lexcial_cast。

1 Boost 简介

Boost是什么？一套开放源代码、高度可移植的C++库。

谁发起的？C++标准委员会库工作组。所以，质量保证，不怕遇到假冒伪劣产品。

有些什么呢？瞧瞧：

正则表达式，可以与POSIX API和Perl语言处理正则表达式的功能相媲美，而且还能支持各种字符类型（如char、wchar_t，甚至还可以是自定义字符类型）；

多线程，想了很久的跨平台多线程库了；

数据结构“图”，再加上即将加入标准的hash_set、hash_map、hash_multiset、hash_multimap等等（事实上不少STL实作，如SGI STL，已经支持以上数据结构），C++对数据结构的支持已近完备；

python，没错，对Python语言的支持；

智能指针，与std::auto_ptr一起善加使用，可杜绝内存泄露，效率更不可和垃圾收集机制GC同日而语；

更有循环冗余的CRC、可轻松定义返回多个值函数的元组tuple、可容纳不同类型值的any、对标准库各方面的补充……

还在迅速扩大中，部分内容有望进入C++标准库……

2 Boost 下载和 Boost 安装

去哪下载Boost呢？英文http://www.boost.org (1)，中文http://boost.c-view.org，可以找到一个.zip或.tar.gz格式的压缩包。下载完毕后，解压到某个目录，比如boost_1_26_0，里面一般有这么几个子目录：boost、libs、more、people、status、tools，看看没问题就行了。

如果Boost更新时您懒得去下载整个压缩包，只希望更新发生变动的文件；或者您是一位跟我一样的Boost Fans，希望跟踪Boost的最新变化，不妨使用CVS方式。首先得有一个CVS客户端软件，比如CvsGui或http://sourceforge.net/projects/cvsgui/提供的WinCVS、gCVS和MacCVS，分别适用于Windows、Linux和MacOS平台。下载、安装、启动三步曲。

如果您习惯于传统CVS的命令行模式，那么可在Admin→Command Line...→Command line settings中输入下面一行2：

style="PADDING-BOTTOM: 0px"style="PADDING-BOTTOM: 0px"cvs -z3 -d:pserver:anonymous@cvs.boost.sourceforge.net:/cvsroot/boost checkout boost

勾上下面的复选框，选择本地目标目录（比如可以新建一个C:\Boost，这凭个人爱好），再点击确定即可开始更新。如果是第一次运行，则可能需要一段时间下载所有文件。当然以后更新就只需要很短的时间了。

如果您偏好GUI模式，请选择Admin→Preferences...，在General的Enter CVS ROOT中填写：

style="PADDING-BOTTOM: 0px"style="PADDING-BOTTOM: 0px"anonymous@cvs.boost.sourceforge.net:/cvsroot/boost

Authentication 选择"passwd" file on the cvs server，同时Use version选择cvs 1.10 (standard)。然后在WinCvs的HOME folder中填写或选择一个本地目标目录，点击确定。选择View→Browse Location→Change...换到本地目标目录后，在Create→Check Module...→Checkout Settings的Enter the module name and path on the server中填写boost，单击确定即可。如果这一过程中要求输入密码，不必理会，直接回车就行。这是WinCVS 1.2的情况。如果您下载的是新的版本，请注意各项设置大同小异，如前面的Authentication选择pserver、不需要设置Use version等。

然后设置编译器。以Windows常用集成环境为例。Microsoft Visual C++ 6.0，可在工具→选择→目录处把Boost的路径（如前面的boost_1_26_0）添加到Include Files搜索路径中。而对于Borland C++ Builder 5.0，则是在Project→Options→Directories/Conditionals→Include Path中添加Boost的路径。还有一种比较常用的Dev-C++ 4.0（内置GNU C++，可从http://www.bloodshed.net处免费下载），可在Options→Compile Options→Directories→C++ include files处添加Boost的路径即可。其他IDE类似。至于命令行方式，则需在编译时对相应的头文件路径参数（Borland C++ Compiler、GNU C++是-I，VC++的cl是/I）给出Boost路径。

做到这一步，恭喜您，大部分Boost库就可以用了。

为什么不是全部？首先，目前还没有一个能完全符合C++标准的编译器，所以Boost库中的组件或多或少不可用，详细信息请看Boost网站上“编译器支持情况（Compiler Status）”一文。另外，有些库需要Build相应的lib或dll文件。不过这样的库很少，主要是由于平台相关性的原因，如处理正则表达式的 regex库、支持python语言的python库等，而建构库的过程相当烦琐，需要使用Jam工具（可以简单提一下：在 tools/build/jam_src/builds目录下有三个文件win32-borlandc.mk、win32-gcc.mk、win32- visualc.mk，分别是适用于Windows平台下的Borland C++ Compiler、GNU C++和Visual C++的mak文件。如果在Unix平台，则应使用tools/build/Makefile。用命令行工具make或nmake来做出Jam执行文件，然后再用Jam来建构库，详细内容可见Boost.Build文档）。我个人的建议是，不用急着去建构lib或dll。真的需要使用这些库时，再make随库提供的mak文件即可。虽然Boost.Jam也许是Boost库未来发展的方向，不过毕竟绝大部分库都无须建构，可以直接使用。

3 Boost 组件 lexical_cast

这次我们先挑个简单实用的Boost组件，看看Boost能给我们带来怎样的便利。

3.1 字符串→数值

在CSDN论坛上经常看到询问如何在字符串类型和数值类型间进行转换的问题，也看到了许多不同的答案。下面先讨论一下从字符串类型到数值类型的转换。

如何将字符串"123"转换为int类型整数123？答案是，用标准C的库函数atoi；

如果要转换为long类型呢？标准C的库函数atol；
如何将"123.12"转换为double类型呢？标准C的库函数atod；
如果要转换为long double类型呢？标准C的库函数atold；
……

后来有朋友开始使用标准库中的string类，问这个如何转换为数值？有朋友答曰，请先转换为const char*。我很佩服作答者有数学家的思维：把陌生的问题转化成熟悉的问题。（曾经有一则笑话，好事者问数学家：知道如何烧水吗？答：知道。把水壶加满水，点火烧。又问：如果水壶里已经有水了呢？答：先倒掉，就转化为我熟悉的问题了……）

不，不，这样是C的做法，不是C++。那么，C++该怎么做呢？使用Boost Conversion Library所提供的函数lexical_cast（需要引入头文件boost/lexical_cast.hpp）无疑是最简单方便的。如:

style="PADDING-BOTTOM: 0px"style="PADDING-BOTTOM: 0px"#include <boost/lexical_cast.hpp>

```
#include <iostream>

int main()

{

        using boost::lexical_cast;

        int a = lexical_cast<int>("123");

        double b = lexical_cast<double>("123.12");

        std::cout<<a<<std::endl

        std::cout<<b<<std::endl;

        return 0;

}
```

一个函数就简洁地解决了所有的问题。

3.2 数值→字符串

那么从数值类型到字符串类型呢？

用itoa？不对吧，标准C/C++里根本没有这个函数。即使在Windows平台下某些编译器提供了该函数3，没有任何移植性不说，还只能解决int类型（也许其他函数还可以解决long、unsigned long

等类型），浮点类型又怎么办？当然，办法还是有，那就是：sprintf。

style="PADDING-BOTTOM: 0px"style="PADDING-BOTTOM: 0px"char s[100];

sprintf(s, "%f", 123.123456);

不知道诸位对C里的scanf/printf系列印象如何，总之阿炯我肯定记不住那些稀奇古怪的参数，而且如果写错了参数，就会得到莫名其妙的输出结果，调试起来可就要命了（我更讨厌的是字符数组，空间开100呢，又怕太小装不下；开100000呢，总觉得太浪费，心里憋气，好在C++标准为我们提供了string这样的字符串类）。这时候，lexical_cast就出来帮忙啦。

style="PADDING-BOTTOM: 0px"style="PADDING-BOTTOM: 0px"#include <boost/lexical_cast.hpp>

#include <string>

#include <iostream>

int main()

{

using std::string;

const double d = 123.12;

string s = boost::lexical_cast<string>(d);

std::cout<<s<<std::endl;

return 0;

}

跟前面一样简单。

3.3 异常

如果转换失败，则会有异常bad_lexical_cast抛出。该异常类是标准异常类bad_cast的子类。

style="PADDING-BOTTOM: 0px"style="PADDING-BOTTOM: 0px"#include <boost/lexical_cast.hpp>

#include <iostream>

int main()

{

using std::cout;

using std::endl;

int i;

try{

i = boost::lexical_cast<int>("abcd");

```
        }
        catch(boost::bad_lexical_cast& e)
        {
                cout<<e.what()<<endl;
                return 1;
        }
        cout<<i<<endl;
        return 0;
}
```

显然“abcd”并不能转换为一个int类型的数值，于是抛出异常，捕捉后输出“bad lexical cast: source type value could not be interpreted as target”这样的信息。

3.4 注意事项

lexical_cast依赖于字符流std::stringstream（会自动引入头文件4），其原理相当简单：把源类型读入到字符流中，再写到目标类型中，就大功告成。例如

style="PADDING-BOTTOM: 0px"style="PADDING-BOTTOM: 0px"int d = boost::lexical_cast<int>("123");

就相当于

style="PADDING-BOTTOM: 0px"style="PADDING-BOTTOM: 0px"int d;

std::stringstream s;

s<<"123";

s>>d;

既然是使用了字符流，当然就有些随之而来的问题，需要特别指出5。

由于Visual C++ 6的本地化（locale）部分实现有问题，因此如果使用了非默认的locale，可能会莫名其妙地抛出异常。当然，一般情况下我们并不需要去改变默认的locale，所以问题不是很大。

输入数据必须“完整”地转换，否则抛出bad_lexical_cast异常。例如

style="PADDING-BOTTOM: 0px"style="PADDING-BOTTOM: 0px"int i = boost::lexical_cast<int>("123.123"); // this will throw

便会抛出异常。因为“123.123”只能“部分”地转换为123，不能“完整”地转换为123.123。

浮点数的精度问题。

style="PADDING-BOTTOM: 0px"style="PADDING-BOTTOM: 0px"std::string s = boost::lexical_cast<std::string>(123.1234567);

以上语句预想的结果是得到“123.1234567”，但是实际上我们只会得到“123.123”，因为默认情况下std::stringstream的精度是6（这是C语言程序库中的“前辈”printf留下的传统）。这可以说是boost::lexical_cast的一个bug。怎么办呢？权宜之计，可以这么做：打开头文件<boost/lexical_cast.hpp>，注意对照修改6：

style="PADDING-BOTTOM: 0px"style="PADDING-BOTTOM: 0px"#include <boost/limits.hpp>

```
//...
template<typename Target, typename Source>
Target lexical_cast(Source arg) {
        //...
        Target result;
        interpreter.precision(std::numeric_limits<Source>::digits10);
        if( !(interpreter << arg) ||
        !(interpreter >> result) ||
        !(interpreter >> std::ws).eof())
        //...
}
```

即可得到正确结果。当然，理论上效率会有一点点损失，不过几乎可以忽略不计。

4 小结

我们已经体验了boost::lexcial_cast。当然，lexical_cast不仅仅局限于字符串类型与数值类型之间的转换：可在任意可输出到stringstream的类型和任意可从stringstream输入的类型间转换。这次的了解尽管很粗略，不过毕竟我们已经“走进Boost”，而不仅仅是“走近”。以后，我们可以自行领略Boost的动人之处啦。

5 注释

[1] 如果您访问Boost英文网站出现DNS错误，不妨试试http://64.226.201.52/。

[2] 请参考Boost文档中的“下载与安装说明（Boost Download and Installation）”部分。

[3] Borland C++ Builder提供了itoa，而Microsoft Visual C++提供了一个功能相同的函数，不过名字是_itoa。

[4] 有些不符合标准的标准库实现中，字符流类名是strstream，在头文件中。而标准规定的是stringstream，在头文件中。

[5] 请参考http://groups.yahoo.com/group/boost/message/15023的讨论。

[6] 非常感谢Andrew Koenig和Bjarne Stroustrup两位的指教和帮助。最开始我的想法是，指定最大精度，加入interpreter.precision(15)之类的语句，然而又担心移植性的问题。Andrew Koenig先生给出了非常明确的解释：You are quite correct that 15 is not portable across all floating-point implementations. However, it is portable across all implementations that support IEEE floating-point arithmetic, which is most computers that are in common use today. If you want to do better than that, you might consider using numeric_limits::digits10, which is the number of significant base-10 digits that can be accurately represented in a double.（中文大意是，诚然，15并非可移植到所有浮点实现中，但对于支持IEEE浮点运算的实现来说，则的确是可移植的，而且，这也是现今绝大部分计算机所使用的。如果想做得更好一点，则可以考虑使用numeric_limits::digits10，就能表示出10进制下double能精确表达的位数。）

Boost 中的智能指针

撰文 Bjorn Karlsson 翻译 曾毅最后更新：2004年6月2日欢迎来到Boost,C++革新者出类拔萃的社群。如果你在C++标准库当中找不到你所需要的，很可能Boost已经为您准备好了他们的产品。

Boost 介绍

根据Boost网站的介绍，Boost是“一个免费的，可移植的，同步评测的C++库，Boost堪称是新类库的典范,特别是其中那些能够与ISO C++标准库良好的协同工作的库。”但是Boost不仅仅是一个库的集合。它也是一个快速发展的开发者社区，这些开发者创建，使用以及参与讨论Boost库。Boost社群不仅仅是维护着这个库，而且还为它的使用者和设计者提供学习交流的场所。这个库堪称是一个设计稳固类的精典范例，在下个版本发布之前你甚至感觉不到能够有什么地方还值得改进。加入Boost邮件列表上的讨论组（或者是活跃于其中，或者只是看看别人如何讨论）是提高你对库的设计的问题和解决方案的认识的非常好的方法。Boost还提供一个人数飞速增长的Boost使用者邮件列表，这个列表关注的内容集中在使用Boost库的问题上。Boost库的质量和他的技术标准是十分令人惊异的。Boost可移植性标准确保了当你将你的代码从一个平台上移动到另一个平台上时，你的库仍然会正常工作。最近的发布版本是Boost 1.25.0，由从智能指针到正则表达式，直至可移植的线程库。Boost目前支持35个库，这些当中所有的内容都被社区的成员测试和使用过了。这些库都是可以免费使用的，它们当中的很多内容都已经被用于商业应用软件的开发。

Boost是C++社群中最为强大的一个之一。在2000名成员当中，很多的人都是世界顶级的C++程序员。这些成员之所以能够长期的参与到其中来是因为他们非常热爱同拥有最优秀的思维方法的程序设计者一同工作。他们同时很清楚他们的努力必定会对C++社群产生巨大的影响，因为你在Boost当中看到的大部分内容将成为融入未来C++标准的候选内容。

了解Boost最好的方法就是浏览Boost库。在这篇文章当中，我将向你介绍Boost的智能指针（smart pointer）库smart_ptr。smart_ptr是一个能够反映出Boost的创新以及完美设计的好例子。我建议你访问Boost的站点()来获取Boost集中的其它34个库的详细内容。

智能指针

相对来说比较小的Boost库之一便是smart_ptr。smart_ptr是我认为在C++标准中将会停止向前发展的库之一。这篇文章讨论了Boost当中的smart_ptr库。但首先，我将以一个对智能指针的简介开始。

智能指针的 30 秒介绍

智能指针是存储指向动态分配（堆）对象指针的类。除了能够在适当的时间自动删除指向的对象外，他们的工作机制很像C++的内置指针。智能指针在面对异常的时候格外有用，因为他们能够确保正确的销毁动态分配的对象。他们也可以用于跟踪被多用户共享的动态分配对象。

事实上，智能指针能够做的还有很多事情，例如处理线程安全，提供写时复制，确保协议，并且提供远程交互服务。有能够为这些ESP (Extremely Smart Pointers)创建一般智能指针的方法，但是并没有涵盖近来。（可以查看[1]来了解关于这个主题的更为深入的内容，顺便提一下，Alexandrescu现在正在考虑将他的C++库Loki提交给Boost）。

智能指针的大部分使用是用于生存期控制，阶段控制。它们使用operator->和operator*来生成原始指针，这样智能指针看上去就像一个普通指针。

这样的一个类来自标准库：std::auto_ptr。它是为解决资源所有权问题设计的，但是缺少对引用数和数组的支持。并且，std::auto_ptr在被复制的时候会传输所有权。在大多数情况下，你需要更多的和/或者是不同的功能。这时就需要加入smart_ptr类。

smart_ptr 类

在Boost中的智能指针有：

。scoped_ptr，用于处理单个对象的唯一所有权；与std::auto_ptr不同的是，scoped_ptr可以被复制。

。scoped_array，与scoped_ptr类似，但是用来处理数组的

。shared_ptr，允许共享对象所有权

。shared_array，允许共享数组所有权

scoped_ptr

scoped_ptr智能指针与std::auto_ptr不同，因为它是不传递所有权的。事实上它明确禁止任何想要这样做的企图！这在你需要确保指针任何时候只有一个拥有者时的任何一种情境下都是非常重要的。如果不去使用scoped_ptr，你可能倾向于使用std::auto_ptr，让我们先看看下面的代码：

auto_ptr MyOwnString[?]

(new string("This is mine to keep!"));

auto_ptr NoItsMine[?](MyOwnString[?]);

cout << *MyOwnString << endl; // Boom

这段代码显然将不能编译通过，因为字符串的所有权被传给了NoItsMine。这不是std::auto_ptr的设计缺陷—而是一个特性。尽管如此，当你需要MyOwnString达到上面的代码预期的工作效果的话，你可以使用scoped_ptr：

scoped_ptr MyOwnString[?]

(new string("This is mine to keep for real!"));

// Compiler error - there is no copy constructor.

scoped_ptr TryingToTakeItAnyway[?]

(MyOwnString[?]);

scoped_ptr通过从boost::noncopyable继承来完成这个行为（可以查看Boost.utility库）。不可复制类声明复制构造函数并将赋值操作符声明为private类型。

scoped_array

scoped_array与scoped_ptr显然是意义等价的，但是是用来处理数组的。在这一点标准库并没有考虑—除非你当然可以使用std::vector，在大多数情况下这样做是可以的。

用法和scoped_ptr类似：

typedef tuples::tupleint> ArrayTuple[?];

scoped_array MyArray[?](new ArrayTuple[?][10]);

tuples::get<0>(MyArray[?][5]) ="The library Tuples is also part of Boost";

tuple是元素的集合—例如两倍，三倍，和四倍。Tuple的典型用法是从函数返回多个值。Boost Tuple库可以被认为是标准库两倍的扩展，目前它与近10个tuple元素一起工作。支持tuple流，比较，赋值，卸包等等。更多关于Boost的Tuple库的信息请参考[2]和[3]。

当scoped_array越界的时候，delete[]将被正确的调用。这就避免了一个常见错误，即是调用错误的操作符delete。

shared_ptr

这里有一个你在标准库中找不到的—引用数智能指针。大部分人都应当有过使用智能指针的经历，并且已经有很多关于引用数的文章。最重要的一个细节是引用数是如何被执行的—插入，意思是说你将引用计数的功能添加给类，或者是非插入，意思是说你不这样做。Boost shared_ptr是非插入类型的，这个实现使用一个从堆中分配来的引用计数器。关于提供参数化策略使得对

任何情况都极为适合的讨论很多了，但是最终讨论的结果是决定反对聚焦于可用性。可是不要指望讨论的结果能够结束。

shared_ptr完成了你所希望的工作：他负责在不使用实例时删除由它指向的对象（pointee），并且它可以自由的共享它指向的对象（pointee）。

```
void PrintIfString2(const any& Any) {
if (const shared_ptr* s =
any_cast >(&Any)) {
cout << **s << endl;
}
}
int main(int argc, char* argv[])
{
std::vector Stuff;
shared_ptr SharedString12
(new string("Share me. By the way,
Boost.any is another useful Boost
library"));
shared_ptr SharedString22
(SharedString12);
shared_ptr SharedInt12
(new int(42));
shared_ptr SharedInt22
(SharedInt12);
Stuff.push_back(SharedString12);
Stuff.push_back(SharedString22);
Stuff.push_back(SharedInt12);
Stuff.push_back(SharedInt22);
// Print the strings
for_each(Stuff.begin(), Stuff.end(),
PrintIfString2);
Stuff.clear();
// The pointees of the shared_ptr's
```

// will be released on leaving scope

// shared_ptr的pointee离开这个范围后将被释放

return 0;

}

any 库 提 供 了 存 储 所 有 东 西 的 方 法 [2]HYPERLINK "file:///C:Documents%20and%20SettingsAdministrator 桌 面 My%20Documents 新 建 CUJhtml20.04karlsson%22%20l"[4] 。 在 包 含 类 型 中 需 要 的 是 它 们 是 可 拷 贝 构 造 的（CopyConstructible），析构函数这里绝对不能引发，他们应当是可赋值的。我们如何存储和传递“所有事物”？无区别类型（读作void*）可以涉及到所有的事物，但这将意味着将类型安全（与知识）抛之脑后。any库提供类型安全。所有满足any需求的类型都能够被赋值，但是解开的时候需要知道解开类型。any_cast是解开由any保存着的值的钥匙，any_cast与dynamic_cast的工作机制是类似的—指针类型的类型转换通过返回一个空指针成功或者失败，因此赋值类型的类型转换抛出一个异常(bad_any_cast)而失败。

shared_array

shared_array与shared_ptr作用是相同的，只是它是用于处理数组的。

shared_array MyStrings[2](new Base[20]);

深入 shared_ptr 实现

创建一个简单的智能指针是非常容易的。但是创建一个能够在大多数编译器下通过的智能指针就有些难度了。而创建同时又考虑异常安全就更为困难了。Boost::shared_ptr这些全都做到了，下面便是它如何做到这一切的。（请注意：所有的include，断开编译器处理，以及这个实现的部分内容被省略掉了，但你可以在Boost.smart_ptr当中找到它们）。

首先，类的定义：很显然，智能指针是（几乎总是）模板。

template class shared_ptr {

公共接口是：

explicit shared_ptr(T* p =0) : px(p) {

// fix: prevent leak if new throws

try { pn = new long(1); }

catch (...) { checked_delete(p); throw; }

}

现在看来，在构造函数当中两件事情是容易被忽略的。构造函数是explicit的，就像大多数的构造函数一样可以带有一个参数。另外一个值得注意的是引用数的堆分配是由一个try-catch块保护的。如果没有这个，你得到的将是一个有缺陷的智能指针，如果引用数没有能够成功分配，它

将不能正常完成它自己的工作。

~shared_ptr() { dispose(); }

析构函数执行另外一个重要任务：如果引用数下降到零，它应当能够安全的删除指向的对象（pointee）。析构函数将这个重要任务委托给了另外一个方法：dispose。

void dispose() { if (—*pn == 0)

{ checked_delete(px); delete pn; } }

正如你所看到的，引用数（pn）在减少。如果它减少到零，checked_delete在所指对象 (px)上被调用，而后引用数(pn)也被删除了。

那么，checked_delete执行什么功能呢？这个便捷的函数（你可以在Boost.utility中找到）确保指针代表的是一个完整的类型。在你的智能指针类当中有这个么？

这是第一个赋值运算符：

template shared_ptr& operator=

(const shared_ptr& r) {

share(r.px,r.pn);

return *this;

}

这是成员模版,如果不是这样，有两种情况：

1. 如果没有参数化复制构造函数，类型赋值Base = Derived无效。

2. 如果有参数化复制构造函数，类型赋值将生效，但同时创建了一个不必要的临时smart_ptr。

这再一次的展示给你为什么不应当加入你自己的智能指针的一个非常好的原因—这些都不是很明显的问题。

赋值运算符的实际工作是由share函数完成的：

void share(T* rpx, long* rpn) {

if (pn = rpn) { // Q: why not px = rpx?

// A: fails when both == 0

++*rpn; // done before dispose() in case

// rpn transitively dependent on

// *this (bug reported by Ken Johnson)

dispose();

px = rpx;

pn = rpn;

}

}

需要注意的是自我赋值（更准确地说是自我共享）是通过比较引用数完成的，而不是通过指针。为什么这样呢？因为它们两者都可以是零，但不一定是一样的。

template shared_ptr

(const shared_ptr& r) : px(r.px) { // never throws

++*(pn = r.pn);

}

这个版本是一个模版化的拷贝构造和函数。可以看看上面的讨论来了解为什么要这样做。

赋值运算符以及赋值构造函数在这里同样也有一个非模版化的版本：

shared_ptr(const shared_ptr& r) :

// never throws

px(r.px) { ++*(pn = r.pn); }

shared_ptr& operator=

(const shared_ptr& r) {

share(r.px,r.pn);

return *this;

}

reset函数就像他的名字那样，重新设置所指对象（pointee）。在将要离开作用域的时候，如果你需要销毁所指对象（pointee）它将非常方便的帮你完成，或者简单的使缓存中的值失效。

void reset(T* p=0) {

// fix: self-assignment safe

if (px == p) return;

if (—*pn == 0)

{ checked_delete(px); }

else { // allocate new reference

// counter

// fix: prevent leak if new throws

try { pn = new long; }

catch (...) {

// undo effect of —*pn above to

// meet effects guarantee

++*pn;

```
checked_delete(p);
throw;
} // catch
} // allocate new reference counter
*pn = 1;
px = p;
} // reset
```

这里仍然请注意避免潜在的内存泄漏问题和保持异常安全的处理手段。

这样你就有了使得智能指针发挥其“智能”的运算符：

```
// never throws
T& operator*() const { return *px; }
// never throws
T* operator->() const { return px; }
// never throws
T* get() const { return px; }
```

这仅仅是一个注释：有的智能指针实现从类型转换运算符到T*的转换。这不是一个好主意，这样做常会使你因此受到伤害。虽然get在这里看上去很不舒服，但它阻止了编译器同你玩游戏。

我记得是Andrei Alexandrescu说的：“如果你的智能指针工作起来和哑指针没什么两样，那它就是哑指针。”简直是太对了。

这里有一些非常好的函数，我们就拿它们来作为本文的结束吧。

```
long use_count() const
{ return *pn; } // never throws
bool unique() const
{ return *pn == 1; } // never throws
```

函数的名字已经说明了它的功能了，对么？

关于Boost.smart_ptr还有很多应当说明的（比如std::swap和std::less的特化，与std::auto_ptr榜定在一起确保兼容性以及便捷性的成员，等等），由于篇幅限制不能再继续介绍了。详细内容请参考Boost distribution ()的smart_ptr.hpp。即使没有那些其它的内容，你不认为他的确是一个非常智能的指针么？

总结

这仅仅是Boost世界的一个简短的介绍。欢迎每一个人的加入，坦率地说，我认为大多数的C++程序员有很好的理由这样做。我在这里要对Beman Dawes, David Abrahams, 以及Jens Maurer回答问题以及分享他们观点的帮助表示感谢（参见“ 来自 Boost创始人的回答 ”）。

Boost见！

注释与参考

[1] Andrei Alexandrescu. Modern C++ Design (Addison-Wesley, 2001).

[2] Boost, .

[3] Jaako Jarvi. “Tuple Types and Multiple Return Values,” C/C++ Users Journal, August 2001.

[4] Jim Hyslop和Herb Sutter. “I'd Hold Anything for You,” C/C++ Users Journal C++ Experts Forum, December 2001, .

[5] Boost邮件列表, .

[6] Boost用户邮件列表, .

[7] C++ Standard, International Standard ISO/IEC 14882.

作者简介 Bjorn Karlsson是ReadSoft专业的软件开发者,您可以通过下面的邮件与他取得联系：

bjorn.karlsson@readsoft.com.

译者注

[1]截至本文翻译结束，Boost社群发布的最新版本为Boost 1.31.0版，最新的版本发布信息可以通过下列地址了解：

http://lists.boost.org/MailArchives/boost-announce/msg00034.php

http://sourceforge.net/project/shownotes.php?release_id=214915

同时可以通过下面的地址下载所有的Boost发布版本：

http://sourceforge.net/project/showfiles.php?group_id=7586

[2]截至本文翻译结束，Boost库已经扩展到了55个，所有的库文档及其它资源可以通过下列地址获得：

http://boost.sourceforge.net/libs/libraries.htm

[3]文中结尾处提到的“来自 Boost创始人的回答 ”可以在下面的地址找到：

http://www.cuj.com/documents/s=8470/cuj0204karlsson/side1.htm

---

Set MYTITLE = Boost中的智能指针

C++ Boost Thread 编程指南

作者：dozb

C++ Boost Thread 编程指南

0 前言

标准C++线程即将到来。CUJ预言它将衍生自Boost线程库，现在就由Bill带领我们探索一下Boost线程库。

就在几年前，用多线程执行程序还是一件非比寻常的事。然而今天互联网应用服务程序普遍使用多线程来提高与多客户链接时的效率；为了达到最大的吞吐量，事务服务器在单独的线程上运行服务程序；GUI应用程序将那些费时，复杂的处理以线程的形式单独运行，以此来保证用户界面能够及时响应用户的操作。这样使用多线程的例子还有很多。

但是C++标准并没有涉及到多线程，这让程序员们开始怀疑是否可能写出多线程的C++程序。尽管不可能写出符合标准的多线程程序，但是程序员们还是会使用支持多线程的操作系统提供的多线程库来写出多线程C++程序。但是这样做至少有两个问题：这些库大部分都是用C语言完成的，如果在C++程序中要使用这些库就必须十分小心；还有，每一个操作系统都有自己的一套支持多线程的类库。因此，这样写出来得代码是没有标准可循的，也不是到处都适用的（non-portable)。Boost线程库就是为了解决所有这些问题而设计的。

Boost是由C++标准委员会类库工作组成员发起，致力于为C++开发新的类库的组织。现在它已经有近2000名成员。许多库都可以在Boost源码的发布版本中找到。为了使这些类库是线程安全的（thread-safe)，Boost线程库被创建了。

许多C++专家都投身于Boost线程库的开发中。所有接口的设计都是从0开始的，并不是C线程API的简单封装。许多C++特性（比如构造函数和析构函数，函数对象（function object)和模板）都被使用在其中以使接口更加灵活。现在的版本可以在POSIX,Win32和Macintosh Carbon平台下工作。

1 创建线程

就像std::fstream类就代表一个文件一样，boost::thread类就代表一个可执行的线程。缺省构造函数创建一个代表当前执行线程的实例。一个重载的构造函数以一个不需任何参数的函数对象作

为参数，并且没有返回值。这个构造函数创建一个新的可执行线程，它调用了那个函数对象。

起先，大家认为传统C创建线程的方法似乎比这样的设计更有用，因为C创建线程的时候会传入一个void*指针，通过这种方法就可以传入数据。然而，由于Boost线程库是使用函数对象来代替函数指针，那么函数对象本身就可以携带线程所需的数据。这种方法更具灵活性，也是类型安全（type-safe)的。当和Boost.Bind这样的功能库一起使用时，这样的方法就可以让你传递任意数量的数据给新建的线程。

目前，由Boost线程库创建的线程对象功能还不是很强大。事实上它只能做两项操作。线程对象可以方便使用==和!=进行比较来确定它们是否是代表同一个线程；你还可以调用boost::thread::join来等待线程执行完毕。其他一些线程库可以让你对线程做一些其他操作（比如设置优先级，甚至是取消线程）。然而，由于要在普遍适用（portable）的接口中加入这些操作不是简单的事，目前仍在讨论如何将这些操组加入到Boost线程库中。

Listing1展示了boost::thread类的一个最简单的用法。 新建的线程只是简单的在std::out上打印“hello,world”，main函数在它执行完毕之后结束。

例1：

style="PADDING-BOTTOM: 0px"style="PADDING-BOTTOM: 0px"#include <boost/thread/thread.hpp>

```
#include <iostream>

void hello()
{
        std::cout <<
        "Hello world, I'm a thread!"
        << std::endl;
}

int main(int argc, char* argv[])
{
        boost::thread thrd(&hello);
        thrd.join();
        return 0;
}
```

2 互斥体

任何写过多线程程序的人都知道避免不同线程同时访问共享区域的重要性。如果一个线程要改变共享区域中某个数据，而与此同时另一线程正在读这个数据，那么结果将是未定义的。为了避免这种情况的发生就要使用一些特殊的原始类型和操作。其中最基本的就是互斥体（mutex，mutual exclusion的缩写）。一个互斥体一次只允许一个线程访问共享区。当一个线程想要访问共享区时，首先要做的就是锁住（lock）互斥体。如果其他的线程已经锁住了互斥体，那么就必须先等那个线程将互斥体解锁，这样就保证了同一时刻只有一个线程能访问共享区域。

互斥体的概念有不少变种。Boost线程库支持两大类互斥体，包括简单互斥体（simple mutex)和递归互斥体（recursive mutex)。如果同一个线程对互斥体上了两次锁，就会发生死锁（deadlock），也就是说所有的等待解锁的线程将一直等下去。有了递归互斥体，单个线程就可以对互斥体多次上锁，当然也必须解锁同样次数来保证其他线程可以对这个互斥体上锁。

在这两大类互斥体中，对于线程如何上锁还有多个变种。一个线程可以有三种方法来对一个互斥体加锁：

一直等到没有其他线程对互斥体加锁。

如果有其他互斥体已经对互斥体加锁就立即返回。
一直等到没有其他线程互斥体加锁，直到超时。

似乎最佳的互斥体类型是递归互斥体，它可以使用所有三种上锁形式。然而每一个变种都是有代价的。所以Boost线程库允许你根据不同的需要使用最有效率的互斥体类型。Boost线程库提供了6中互斥体类型，下面是按照效率进行排序：

style="PADDING-BOTTOM: 0px"style="PADDING-BOTTOM: 0px"boost::mutex,

boost::try_mutex,

boost::timed_mutex,

boost::recursive_mutex,

boost::recursive_try_mutex,

boost::recursive_timed_mutex

如果互斥体上锁之后没有解锁就会发生死锁。这是一个很普遍的错误，Boost线程库就是要将其变成不可能（至少时很困难）。直接对互斥体上锁和解锁对于Boost线程库的用户来说是不可能的。mutex类通过teypdef定义在RAII中实现的类型来实现互斥体的上锁和解锁。这也就是大家知道的Scope Lock模式。为了构造这些类型，要传入一个互斥体的引用。构造函数对互斥体加锁，析构函数对互斥体解锁。C++保证了析构函数一定会被调用，所以即使是有异常抛出，互斥体也总是会被正确的解锁。

这种方法保证正确的使用互斥体。然而，有一点必须注意：尽管Scope Lock模式可以保证互斥体被解锁，但是它并没有保证在异常抛出之后贡献资源仍是可用的。所以就像执行单线程程序一样，必须保证异常不会导致程序状态异常。另外，这个已经上锁的对象不能传递给另一个线程，因为它们维护的状态并没有禁止这样做。

List2给出了一个使用boost::mutex的最简单的例子。例子中共创建了两个新的线程，每个线程都有10次循环，在std::cout上打印出线程id和当前循环的次数，而main函数等待这两个线程执行完才结束。std::cout就是共享资源，所以每一个线程都使用一个全局互斥体来保证同时只有一个线程能向它写入。

许多读者可能已经注意到List2中传递数据给线程还必须的手工写一个函数。尽管这个例子很简单，如果每一次都要写这样的代码实在是让人厌烦的事。别急，有一种简单的解决办法。函数库允许你通过将另一个函数绑定，并传入调用时需要的数据来创建一个新的函数。 List3向你展示了如何使用Boost.Bind库来简化List2中的代码，这样就不必手工写这些函数对象了。

例2：

style="PADDING-BOTTOM: 0px"style="PADDING-BOTTOM: 0px"#include <boost/thread/thread.hpp>

```
#include <boost/thread/mutex.hpp>
#include <iostream>

boost::mutex io_mutex;

struct count
{
	count(int id) : id(id) { }

	void operator()()
	{
		for (int i = 0; i < 10; ++i)
		{
			boost::mutex::scoped_lock
			lock(io_mutex);
			std::cout << id << ": "
			<< i << std::endl;
		}
	}

	int id;
};
```

```
int main(int argc, char* argv[])
{
        boost::thread thrd1(count(1));
        boost::thread thrd2(count(2));
        thrd1.join();
        thrd2.join();
        return 0;
}
```

例3： // 这个例子和例2一样，除了使用Boost.Bind来简化创建线程携带数据，避免使用函数对象

style="PADDING-BOTTOM: 0px"style="PADDING-BOTTOM: 0px"

```
#include <boost/thread/thread.hpp>
#include <boost/thread/mutex.hpp>
#include <boost/bind.hpp>
#include <iostream>

boost::mutex io_mutex;

void count(int id)
{
        for (int i = 0; i < 10; ++i)
        {
                boost::mutex::scoped_lock
                lock(io_mutex);
                std::cout << id << ": " <<
                i << std::endl;
        }
}

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

```
    boost::thread thrd1(
    boost::bind(&count, 1));
    boost::thread thrd2(
    boost::bind(&count, 2));
    thrd1.join();
    thrd2.join();
    return 0;
}
```

3 条件变量

有的时候仅仅依靠锁住共享资源来使用它是不够的。有时候共享资源只有某些状态的时候才能够使用。比方说，某个线程如果要从堆栈中读取数据，那么如果栈中没有数据就必须等待数据被压栈。这种情况下的同步使用互斥体是不够的。另一种同步的方式——条件变量，就可以使用在这种情况下。

条件变量的使用总是和互斥体及共享资源联系在一起的。线程首先锁住互斥体，然后检验共享资源的状态是否处于可使用的状态。如果不是，那么线程就要等待条件变量。要指向这样的操作就必须在等待的时候将互斥体解锁，以便其他线程可以访问共享资源并改变其状态。它还得保证从等到得线程返回时互斥体是被上锁得。当另一个线程改变了共享资源的状态时，它就要通知正在等待条件变量得线程，并将之返回等待的线程。

List4是一个使用了boost::condition的简单例子。有一个实现了有界缓存区的类和一个固定大小的先进先出的容器。由于使用了互斥体boost::mutex，这个缓存区是线程安全的。put和get使用条件变量来保证线程等待完成操作所必须的状态。有两个线程被创建，一个在buffer中放入100个整数，另一个将它们从buffer中取出。这个有界的缓存一次只能存放10个整数，所以这两个线程必须周期性的等待另一个线程。为了验证这一点，put和get在std::cout中输出诊断语句。最后，当两个线程结束后，main函数也就执行完毕了。

```
style="PADDING-BOTTOM: 0px"style="PADDING-BOTTOM: 0px"#include <boost/thread/thread.hpp>
#include <boost/thread/mutex.hpp>
#include <boost/thread/condition.hpp>
#include <iostream>

const int BUF_SIZE = 10;
const int ITERS = 100;
```

```
boost::mutex io_mutex;

class buffer
{
        public:
        typedef boost::mutex::scoped_lock
        scoped_lock;

        buffer()
        : p(0), c(0), full(0)
        {
        }

        void put(int m)
        {
                scoped_lock lock(mutex);
                if (full == BUF_SIZE)
                {
                        {
                                boost::mutex::scoped_lock
                                lock(io_mutex);
                                std::cout <<
                                "Buffer is full. Waiting..."
                                << std::endl;
                        }
                        while (full == BUF_SIZE)
                        cond.wait(lock);
                }
                buf[p] = m;
                p = (p+1) % BUF_SIZE;
                ++full;
```

```
        cond.notify_one();
    }

    int get()
    {
        scoped_lock lk(mutex);
        if (full == 0)
        {
            {
                boost::mutex::scoped_lock
                lock(io_mutex);
                std::cout <<
                "Buffer is empty. Waiting..."
                << std::endl;
            }
            while (full == 0)
            cond.wait(lk);
        }
        int i = buf[c];
        c = (c+1) % BUF_SIZE;
        --full;
        cond.notify_one();
        return i;
    }

    private:
    boost::mutex mutex;
    boost::condition cond;
    unsigned int p, c, full;
    int buf[BUF_SIZE];
};
```

```
buffer buf;

void writer()
{
	for (int n = 0; n < ITERS; ++n)
	{
		{
			boost::mutex::scoped_lock
			lock(io_mutex);
			std::cout << "sending: "
			<< n << std::endl;
		}
		buf.put(n);
	}
}

void reader()
{
	for (int x = 0; x < ITERS; ++x)
	{
		int n = buf.get();
		{
			boost::mutex::scoped_lock
			lock(io_mutex);
			std::cout << "received: "
			<< n << std::endl;
		}
	}
}
```

```
int main(int argc, char* argv[])
{
        boost::thread thrd1(&reader);
        boost::thread thrd2(&writer);
        thrd1.join();
        thrd2.join();
        return 0;
}
```

4 线程局部存储

大多数函数都不是可重入的。这也就是说在某一个线程已经调用了一个函数时，如果你再调用同一个函数，那么这样是不安全的。一个不可重入的函数通过连续的调用来保存静态变量或者是返回一个指向静态数据的指针。 举例来说，std::strtok就是不可重入的，因为它使用静态变量来保存要被分割成符号的字符串。

有两种方法可以让不可重用的函数变成可重用的函数。第一种方法就是改变接口，用指针或引用代替原先使用静态数据的地方。比方说，POSIX定义了strok_r，std::strtok中的一个可重入的变量，它用一个额外的char**参数来代替静态数据。这种方法很简单，而且提供了可能的最佳效果。但是这样必须改变公共接口，也就意味着必须改代码。另一种方法不用改变公有接口，而是用本地存储线程（thread local storage）来代替静态数据（有时也被成为特殊线程存储，thread-specific storage）。

Boost线程库提供了智能指针boost::thread_specific_ptr来访问本地存储线程。每一个线程第一次使用这个智能指针的实例时，它的初值是NULL，所以必须要先检查这个它的只是否为空，并且为它赋值。Boost线程库保证本地存储线程中保存的数据会在线程结束后被清除。

List5是一个使用boost::thread_specific_ptr的简单例子。其中创建了两个线程来初始化本地存储线程，并有10次循环，每一次都会增加智能指针指向的值，并将其输出到std::cout上（由于std::cout是一个共享资源，所以通过互斥体进行同步）。main线程等待这两个线程结束后就退出。从这个例子输出可以明白的看出每个线程都处理属于自己的数据实例，尽管它们都是使用同一个boost::thread_specific_ptr。

例5：

```
style="PADDING-BOTTOM: 0px"style="PADDING-BOTTOM: 0px"#include <boost/thread/thread.hpp>
#include <boost/thread/mutex.hpp>
#include <boost/thread/tss.hpp>
#include <iostream>
```

```
boost::mutex io_mutex;
boost::thread_specific_ptr<int> ptr;

struct count
{
	count(int id) : id(id) { }

	void operator()()
	{
		if (ptr.get() == 0)
		ptr.reset(new int(0));

		for (int i = 0; i < 10; ++i)
		{
			(*ptr)++;
			boost::mutex::scoped_lock
			lock(io_mutex);
			std::cout << id << ": "
			<< *ptr << std::endl;
		}
	}

	int id;
};

int main(int argc, char* argv[])
{
	boost::thread thrd1(count(1));
	boost::thread thrd2(count(2));
	thrd1.join();
```

```
    thrd2.join();
    return 0;
}
```

5 仅运行一次的例程

还有一个问题没有解决：如何使得初始化工作（比如说构造函数）也是线程安全的。比方说，如果一个引用程序要产生唯一的全局的对象，由于实例化顺序的问题，某个函数会被调用来返回一个静态的对象，它必须保证第一次被调用时就产生这个静态的对象。这里的问题就是如果多个线程同时调用了这个函数，那么这个静态对象的构造函数就会被调用多次，这样错误产生了。

解决这个问题的方法就是所谓的“一次实现”（once routine）。“一次实现”在一个应用程序只能执行一次。如果多个线程想同时执行这个操作，那么真正执行的只有一个，而其他线程必须等这个操作结束。为了保证它只被执行一次，这个routine由另一个函数间接的调用，而这个函数传给它一个指针以及一个标志着这个routine是否已经被调用的特殊标志。这个标志是以静态的方式初始化的，这也就保证了它在编译期间就被初始化而不是运行时。因此也就没有多个线程同时将它初始化的问题了。Boost线程库提供了boost::call_once来支持“一次实现”，并且定义了一个标志boost::once_flag及一个初始化这个标志的宏BOOST_ONCE_INIT。

List6是一个使用了boost::call_once的例子。其中定义了一个静态的全局整数，初始值为0；还有一个由BOOST_ONCE_INIT初始化的静态boost::once_flag实例。main函数创建了两个线程，它们都想通过传入一个函数调用boost::call_once来初始化这个全局的整数，这个函数是将它加1。main函数等待着两个线程结束，并将最后的结果输出的到std::cout。由最后的结果可以看出这个操作确实只被执行了一次，因为它的值是1。

```
style="PADDING-BOTTOM: 0px"style="PADDING-BOTTOM: 0px"#include <boost/thread/thread.hpp>
#include <boost/thread/once.hpp>
#include <iostream>

int i = 0;
boost::once_flag flag =
BOOST_ONCE_INIT;

void init()
{
    ++i;
```

```
}

void thread()
{
        boost::call_once(&init, flag);
}

int main(int argc, char* argv[])
{
        boost::thread thrd1(&thread);
        boost::thread thrd2(&thread);
        thrd1.join();
        thrd2.join();
        std::cout << i << std::endl;
        return 0;
}
```

6 Boost 线程库的未来

Boost线程库正在计划加入一些新特性。其中包括boost::read_write_mutex，它可以让多个线程同时从共享区中读取数据，但是一次只可能有一个线程向共享区写入数据；boost::thread_barrier，它使得一组线程处于等待状态，知道所有得线程都都进入了屏障区；boost::thread_pool,他允许执行一些小的routine而不必每一都要创建或是销毁一个线程。

Boost线程库已经作为标准中的类库技术报告中的附件提交给C++标准委员会，它的出现也为下一版C++标准吹响了第一声号角。委员会成员对Boost线程库的初稿给予了很高的评价,当然他们还会考虑其他的多线程库。他们对在C++标准中加入对多线程的支持非常感兴趣。从这一点上也可以看出，多线程在C++中的前途一片光明。

7 参考资料：

The Boost.Threads Library by Bill Kempf

Visit the Boost website at <http://www.boost.org>.

更多查看：

C++ Boost库文档索引文档

C++ Boost Assign 文档

C++ Boost 库文档索引

来源:http://dozb.blogchina.com/2157330.html

什么库用何种编译器请看 [[http://www.boost.org/status/compiler_status.html][Compiler Status]]

如何下载，建造，安装库请看 [[http://www.boost.org/more/getting_started.html][Getting Started]]
对一些库的文档的其他可选文件格式:

PDF [.zip | .gz]

Unix man pages
DocBook

XSL Formatting Objects

1 按字母顺序库列表

any - 安全，泛型的容器，包含不同类型的值, 作者 Kevlin Henney.

array - STL风格封装下的定长数组, 作者 Nicolai Josuttis.
assign - 用常数或更容易方式生成的数据填充容器, 作者 Thorsten Ottosen.
bind 和 mem_fn - 为函数/对象/指针和成员函数而被泛化的组合者, 作者 Peter Dimov.
call_traits - 实现自动判断传入参数的方式, 作者 John Maddock, Howard Hinnant, et al.
compatibility - 对不一致的标准库提供帮助, 作者 Ralf Grosse-Kunstleve and Jens Maurer.
compressed_pair - 针对pair当中空成员做了一些优化, 作者 John Maddock, Howard Hinnant, et al.
concept check - 泛型编程的工具, 作者 Jeremy Siek.
config - 帮助 boost 库的开发者配置编译器特性；不打算提供给库用户使用.
conversion - 各种类型间的转化，Numeric, polymorphic, 和 lexical casts, 作者 Dave Abrahams and Kevlin Henney.
crc - 循环沉余码, 作者 Daryle Walker.
date_time - Date-Time 库， 作者 Jeff Garland.
dynamic_bitset - std::bitset的动态长度版本, 作者 Jeremy Siek 和 Chuck Allison.
enable_if - 函数模板重载时的选择性包含, 作者 Jaakko Järvi, Jeremiah Willcock, 和 Andrew Lumsdaine.
filesystem - 方便地操作文件路径, 通过iteration访问目录, 和其他有用的文件系统操作, 作者 Beman Dawes.
format - 类型安全的 '类似printf' 格式的操作, 作者 Samuel Krempp.
function - 为延期调用和回调的函数对象的包裹, 作者 Doug Gregor.
functional - 增强的函数对象配接器, 作者 Mark Rodgers.
graph - 泛型图的组件和算法, 作者 Jeremy Siek 和 a University of Notre Dame team.
integer - 能够帮助简化对整数类型的处理。
interval - Extends the usual arithmetic functions to mathematical intervals, 作者 Guillaume Melquiond, Herv′ Brönnimann and Sylvain Pion.
in_place_factory, typed_in_place_factory- Generic in-place construction of contained objects with a variadic argument-list, 作者 Fernando Cacciola.
io state savers - 保存 I/O 状态来防止混乱的数据, 作者 Daryle Walker.
iterators - Iterator 构造框架, 配接器, 概念, 和其他, 作者 Dave Abrahams, Jeremy Siek, 和 Thomas Witt.
lambda - 在实际调用地点定义小的无名函数对象, 作者 Jaakko Järvi and Gary Powell.

math - 在数学领域的几个贡献, 作者 various authors.
math/common_factor - 最大公约数和最小公倍数, 作者 Daryle Walker.
math/octonion - Octonions, 作者 Hubert Holin.
math/quaternion - Quaternions, 作者 Hubert Holin.
math/special_functions - 数学方面的函数比如 atanh, sinc, 和 sinhc, 作者 Hubert Holin.
minmax - 标准库扩展，用于同时进行 min/max 和 min/max 元素计算, 作者 Hervé Brönnimann.
mpl - 模板元编程框架，用于编译时计算，序列化和元函数类, 作者 Aleksey Gurtovoy.
multi_array - 多维数组的容器和配接器, 作者 Ron Garcia.
multi_index - 提供对可重复键值STL兼容容器的存取接口, 作者 Joaquín M López Muñoz.
numeric/conversion - 优化的基于策略的数值变换, 作者 Fernando Cacciola.
operators - 使算法类和迭代器容易的模板, 作者 Dave Abrahams 和 Jeremy Siek.
optional - 对可选项值的可识别联合包裹, 作者 Fernando Cacciola.
pool - 内存池管理, 作者 Steve Cleary.
preprocessor - 预处理元编程工具，包含重复和递归, 作者 Vesa Karvonen 和 Paul Mensonides.
program_options - 通过命令行，配置文件和其他来源来存取配置参数, 作者 Vladimir Prus.
property map - Concepts defining interfaces which map key objects to value objects, 作者 Jeremy Siek.
python - 映射 C++ 类和函数给 Python 使用， 作者 Dave Abrahams.
random - 随机数生成的完整系统, 作者 Jens Maurer.
range - new 根基，其为建于new iterator概念之上的泛型计算, 作者 Thorsten Ottosen.
rational - 有理数类, 作者 Paul Moore.
ref - 一个工具库，用于传递引用到泛型函数, 作者 Jaako Järvi, Peter Dimov, Doug Gregor, 和 Dave Abrahams.
regex - 正则表达式库, 作者 John Maddock .
serialization - Serialization for persistence and marshalling, 作者 Robert Ramey
signals - 被管理的信号和邮槽回调的实现, 作者 Doug Gregor.
smart_ptr - 五个智能指针类模板, 作者 Greg Colvin, Beman Dawes, Peter Dimov, 和 Darin Adler.
static_assert - 静态断言 (编译时断言), 作者 John Maddock.
spirit - LL分析的框架，在嵌入式C++中根据EBNF规则对文件进行分析, 作者 Joel de Guzman and team.
string_algo - 字符串算法库, 作者 Pavol Droba .
test - 支持简单程序测试, 完整单元测试, 和程序执行监控, 作者 Gennadiy Rozental.
thread - 跨平台的线程实现。Portable C++ multi-threading, 作者 William Kempf.
timer - Event timer, progress timer, and progress display classes, 作者 Beman Dawes.
tokenizer - 把字符串或其他字符序列分解成一系列标记(tokens), 作者 John Bandela.
tribool - 3种状态的 boolean 类型库, 作者 Doug Gregor.
tuple - Ease definition of functions returning multiple values, and more, 作者 Jaakko Järvi.
type_traits - 类型的基本属性的模板, 作者 John Maddock, Steve Cleary, et al.
uBLAS - 基本线性代数，用于矩阵操作, 作者 Joerg Walter and Mathias Koch.

utility - 类 **noncopyable** 加 **checked_delete()**, **checked_array_delete()**, **next()**, **prior()** 函数模板, 加 **base-from-member idiom**, 作者 Dave Abrahams 等.
value_initialized - 为统一的语法的值初始化的包裹, 作者 Fernando Cacciola, 基于 David Abrahams 的思想.
variant - 安全, 泛型, 基于栈的，不同于联合容器, 作者 Eric Friedman and Itay Maman.

---

2 按主题库列表

2.0 字符串和文本处理(String and text processing)

conversion/lexical_cast - lexical_cast 类模板，作者 Kevlin Henney.

format - 类型安全的 '类似printf' 格式的操作, 作者 Samuel Krempp.
regex - 正则表达式库, 作者 John Maddock .
spirit - LL分析的框架，在嵌入式C++中根据EBNF规则对文件进行分析, 作者 Joel de Guzman and team.
tokenizer - 把字符串或其他字符序列分解成一系列标记(tokens), 作者 John Bandela.
string_algo - 字符串算法库, 作者 Pavol Droba .

2.1 容器(Containers)

array - STL风格封装下的定长数组, 作者 Nicolai Josuttis.

dynamic_bitset - std::bitset的动态长度版本, 作者 Jeremy Siek 和 Chuck Allison.
graph - 泛型图的组件和算法, 作者 Jeremy Siek 和 a University of Notre Dame team.
multi_array - 多维数组的容器和配接器, 作者 Ron Garcia.
multi_index - 提供对可重复键值STL兼容容器的存取接口, 作者 Joaquín M López Muñoz.
property map - Concepts defining interfaces which map key objects to value objects, 作者 Jeremy Siek.
variant - 安全, 泛型, 基于栈的，不同于联合容器, 作者 Eric Friedman and Itay Maman.

2.2 迭代器(Iterators)

graph - 泛型图的组件和算法, 作者 Jeremy Siek 和 a University of Notre Dame team.

iterators - Iterator 构造框架, 配接器, 概念, 和其他, 作者 Dave Abrahams, Jeremy Siek, 和 Thomas Witt.
operators - 使算法类和迭代器容易的模板, 作者 Dave Abrahams 和 Jeremy Siek.

tokenizer - 把字符串或其他字符序列分解成一系列标记(tokens), 作者 John Bandela.

2.3 算法(Algorithms)

graph - 泛型图的组件和算法, 作者 Jeremy Siek 和 a University of Notre Dame team.

minmax - 标准库扩展，用于同时进行 min/max 和 min/max 元素计算, 作者 Hervé Brönnimann.

string_algo - 字符串算法库, 作者 Pavol Droba .

utility - 类 **next(), prior()** 函数模板, 作者 Dave Abrahams and others.

range - new 根基，其为建于new iterator概念之上的泛型计算, 作者 Thorsten Ottosen.

2.4 函数对象和高阶编程(Function objects and higher-order programming)

bind 和 mem_fn - 为函数/对象/指针和成员函数而被泛化的组合者, 作者 Peter Dimov.

function - 为延期调用和回调的函数对象的包裹, 作者 Doug Gregor.

functional - 增强的函数对象配接器, 作者 Mark Rodgers.

lambda - 在实际调用地点定义小的无名函数对象, 作者 Jaakko Järvi 和 Gary Powell.

ref - 一个工具库，用于传递引用到泛型函数, 作者 Jaako Järvi, Peter Dimov, Doug Gregor, 和 Dave Abrahams.

signals - 被管理的信号和邮槽回调的实现, 作者 Doug Gregor.

result_of - 确定函数调用表达式的类型.

2.5 泛型编程(Generic Programming)

call_traits - 实现自动判断传入参数的方式, 作者 John Maddock, Howard Hinnant, et al.

concept check - 泛型编程的工具, 作者 Jeremy Siek.

enable_if - 函数模板重载时的选择性包含, 作者 Jaakko Järvi, Jeremiah Willcock, 和 Andrew Lumsdaine.

in_place_factory, typed_in_place_factory- Generic in-place construction of contained objects with a variadic argument-list, 作者 Fernando Cacciola.

operators - 使算法类和迭代器容易的模板, 作者 Dave Abrahams 和 Jeremy Siek.

property map - Concepts defining interfaces which map key objects to value objects, 作者 Jeremy Siek.

static_assert - 静态断言 (编译时断言), 作者 John Maddock.

type_traits - 类型的基本属性的模板, 作者 John Maddock, Steve Cleary, et al.

2.6 模板元编程(Template Metaprogramming)

mpl - 模板元编程框架，用于编译时计算，序列化和元函数类, 作者 Aleksey Gurtovoy.

static_assert - 静态断言 (编译时断言), 作者 John Maddock.

type_traits - 类型的基本属性的模板, 作者 John Maddock, Steve Cleary, et al.

2.7 预处理元编程(Preprocessor Metaprogramming)

preprocessor - 预处理元编程工具，包含重复和递归, 作者 Vesa Karvonen 和 Paul Mensonides.

2.8 并发编程(Concurrent Programming)

thread - 轻便的C++多线程库, 作者 William Kempf.

2.9 数学和数值计算(Math and numerics)

math - 在数学领域的几个贡献, 作者 various authors.

conversion/numeric_cast - numeric_cast 类模板, 作者 Kevlin Henney.
numeric/conversion - 优化的基于策略的数值变换, 作者 Fernando Cacciola.
integer - 能够帮助简化对整数类型的处理。
interval - Extends the usual arithmetic functions to mathematical intervals, 作者 Guillaume Melquiond, Hervé Brönnimann and Sylvain Pion.
math/common_factor - 最大公约数和最小公倍数, 作者 Daryle Walker.
math/octonion - Octonions, 作者 Hubert Holin.
math/quaternion - Quaternions, 作者 Hubert Holin.
math/special_functions - 数学方面的函数比如 atanh, sinc, 和 sinhc, 作者 Hubert Holin.
multi_array - 多维数组的容器和配接器, 作者 Ron Garcia.
operators - 使算法类和迭代器容易的模板, 作者 Dave Abrahams 和 Jeremy Siek.
random - 随机数生成的完整系统, 作者 Jens Maurer.
rational - 有理数类, 作者 Paul Moore.
uBLAS - 基本线性代数，用于矩阵操作, 作者 Joerg Walter and Mathias Koch.

2.10 纠错和测试(Correctness and testing)

concept check - 泛型编程的工具, 作者 Jeremy Siek.

static_assert - 静态断言 (编译时断言), 作者 John Maddock.
test - 支持简单程序测试, 完整单元测试, 和程序执行监控, 作者 Gennadiy Rozental.

2.11数据结构(Data structures)

any - 安全，泛型的容器，包含不同类型的值, 作者 Kevlin Henney.

compressed_pair - 针对pair当中空成员做了一些优化, 作者 John Maddock, Howard Hinnant, et al.
multi_index - 提供对可重复键值STL兼容容器的存取接口, 作者 Joaquín M López Muñoz.

tuple - 容易地定义可返回多个值的函数, 作者 Jaakko Järvi.
variant - 安全, 泛型, 基于栈的，不同于联合容器, 作者 Eric Friedman and Itay Maman.

2.12 输入/输出(Input/Output)

format - 类型安全的 '类似printf' 格式的操作, 作者 Samuel Krempp.

io state savers - 保存 I/O 状态来防止混乱的数据, 作者 Daryle Walker.
program_options - 通过命令行，配置文件和其他来源来存取配置参数, 作者 Vladimir Prus.
serialization - Serialization of arbitrary data for persistence and marshalling, 作者 Robert Ramey
assign - 用常数或更容易方式生成的数据填充容器, 作者 Thorsten Ottosen.

2.13 跨语言支持(Inter-language support)

python - 映射 C++ 类和函数给 Python 使用， 作者 Dave Abrahams.

2.14 内存(Memory)

pool - 内存池管理, 作者 Steve Cleary.

smart_ptr - 五个智能指针类模板，作者 Greg Colvin, Beman Dawes, Peter Dimov, 和 Darin Adler.
utility - 类 **noncopyable** 加 **checked_delete()**, **checked_array_delete()**, **next()**, **prior()** 函数模板, 加 **base-from-member idiom**, 作者 Dave Abrahams 等.

2.15解析(Parsing)

spirit - LL分析的框架，在嵌入式C++中根据EBNF规则对文件进行分析, 作者 Joel de Guzman and team.

2.16杂项(Miscellaneous)

base-from-member - Idiom to initialize a base class with a member, 作者 Daryle Walker.

compressed_pair - 针对pair当中空成员做了一些优化, 作者 John Maddock, Howard Hinnant, et al.
conversion - 各种类型间的转化，Numeric, polymorphic, 和 lexical casts, 作者 Dave Abrahams and Kevlin Henney.
numeric/conversion - 优化的基于策略的数值变换, 作者 Fernando Cacciola.
crc - 循环沉余码, 作者 Daryle Walker.
date_time - Date-Time 库， 作者 Jeff Garland.
filesystem - 方便地操作文件路径, 通过iteration访问目录, 和其他有用的文件系统操作, 作者 Beman Dawes.

optional - 对可选项值的可识别联合包裹, 作者 Fernando Cacciola.
program_options - 通过命令行，配置文件和其他来源来存取配置参数, 作者 Vladimir Prus.
timer - 事件定时器, 进度定时器, 和进度显示类, 作者 Beman Dawes.
tribool - 3种状态的 boolean 类型库, 作者 Doug Gregor.
utility - 类 **noncopyable** 加 **checked_delete()**, **checked_array_delete()**, **next()**, **prior()** 函数模板, 加 **base-from-member idiom**, 作者 Dave Abrahams 等.
value_initialized - 为统一的语法的值初始化的包裹, 作者 Fernando Cacciola, 基于 David Abrahams 的思想.

2.17 Broken compiler workarounds

compatibility - 对不一致的标准库提供帮助, 作者 Ralf Grosse-Kunstleve and Jens Maurer.

config - 帮助 boost 库的开发者配置编译器特性；不打算提供给库用户使用.

[Category suggestions from Aleksey Gurtovoy and Beman Dawes]

---

3 Boost中已废除的库

compose - Functional composition adapters for the STL, 作者 Nicolai Josuttis. Removed in Boost version 1.32. Please use Bind or Lambda instead.