第 13 章

编程语言支持

任何程序开发语言都不可能独当一面、包打天下，总有它的长处和短处。

就 C++来说，它继承了 C 语言的低级能力，能够使用指针直接操作内存，也可以嵌入汇编语言以获得最快的运行速度。它还支持面向对象和泛型编程等现代程序开发技术，可以在很高的层次上对软件建模，获得高度的抽象，几乎所有领域的软件都可以使用 C++完成开发。

但 C++也有一些缺点，它不能做到便捷地跨平台开发，许多强大的功能还需要编译器和操作系统的支持。C++程序开发周期长，复杂的语言特性让它具有陡峭的学习曲线，C++也没有垃圾回收和类型反射机制，这些都限制了它的快速应用开发的能力。

因此，虽然 C++很强大，但它没有必要也不可能事必躬亲，与其他的编程语言配合会更好地发挥它的优点同时弥补它的缺点，Python 就是这样的一种语言。虽然表面上看 Python 与 C++完全不同，但基本的编程理念却很相似：具有类似的控制流语句，面向对象，支持操作符重载和异常等等，两者具有很好的互补性。

Boost 通过 boost.python 库提供了对 Python 语言的全面支持。

13.1 python 库概述

Boost 中的 python 库已经发展到第 2 版，较早期的第 1 版有了很大的改进，可以更加方便和容易地在 Python 和 C++之间自由转换，而且功能更强大。python 库全面支持 C++和 Python 的各种特性，包括 C++到 Python 的异常转换、默认参数、关键字参数、引用和指针等等，让 C++与 Python 可以近乎完美地对接：用 C++很容易地为 python 编写扩展模块，也可以很容易地在 C++代码中执行 python 程序。

为行文方便，本章以下的描述中使用首字母大写的“Python”表示 Python 语言，首字母小写的“python”表示 boost.python 库。

13.1.1 Python 语言简介

20 世纪 80 年代末，Guido van Rossum 创建了一种新型的脚本语言，他从英国 BBC 的“Monty Python's Flying Circus”节目中摘取了一个单词作为这种新语言的名字，这就是之后风行二十余年的 Python。

Python 是一种简洁的、功能强大的、动态强类型的解释型语言。

Python 具有比 C/C++语言还要简洁的语法，使用代码缩进而不是分号来分隔语句，同时简化了许多传统的语法结构，从而具有优雅的代码格式。

Python 也有具有强大的功能，内建了多种高级数据结构，如列表、集合、元组和字典，使程序员只需要编写少量的代码就可以实现 C/C++很多行代码才能完成的工作。Python 完全支持面向过程编程和面向对象编程（实际上，Python 里的一切几乎都是对象），支持异常、名字空间、操作符重载等现代编程机制。它还有一个“包罗万象”的标准库和许多第三方库，如正则表达式、数据库、XML、电子邮件、测试、图形界面……几乎可以实现任何功能。如果想要定制功能也很容易，Python 可用 C 语言扩展底层，增添新模块。

Python 是一个动态语言，类似 Php、Perl，无需声明变量类型就可以使用。但它又是强类型的，变量一旦初始化，就不能随意改变类型。Python 不需要编译，它可以动态地解释交互运行，类似 BASIC，这大大缩短了程序的开发周期。但 Python 也能够像 Java 一样编译成字节码后运行在虚拟机（解释器）上以获得更高的效率。

Python 是可移植的，在各种操作系统上都有免费的 Python 解释器，包括 Windows、Mac OS X、Linux、UNIX 等主流平台。在这些平台上 Python 可以代替批处理、shell 或者 Perl，编写脚本程序方便日常工作，也可以开发非常复杂的应用程序或者服务程序，Python 能够做其他语言所能做的所有事情。

最初的 Python 是用 C 语言写成的，随着时代的发展，又逐渐出现了其他语言编写的 Python，如 Java 的 Jython 和 C#的 IronPython，它们不仅具有 Python 的功能，还可以调用宿主语言，更为强大。因此，传统的 Python 有时又被称为 C-Python。

下面简单展示几段 Python 代码，其中的#是 Python 的注释：

```
print 'hello python'      #输出 hello python
print 2**3                #输出 2 的立方 8
```

```
for x in range(1,5):                #循环语句，输出 1 2 3 4
  print x

x = ['abc', 123, "python"]          #一个列表，可以包含任意类型
for y in x:                         #遍历列表
  print 'list[%s]' % y             #带格式的输出

import re                           #导入正则表达式模块
reg = re.compile('a*b')             #编译一个正则表达式对象
print reg.match('ab') != None       #正则表达式匹配字符串
```

如果读者想进一步了解 Python，建议阅读推荐书目[12]，可以从因特网上免费下载。作为一个快速的开始，它可以在一天之内阅读完并掌握。

13.1.2 安装 Python 环境

Python 目前有两个主要的版本分支：2.x 和 3.x。

2.x 稳定版是 2.73，有非常丰富的第三方库，缺点是对 unicode 支持不够好，处理中文比较麻烦。

3.x 稳定版是 3.3，对 2.x 版本做了很多语法变动（例如 print 由语句变成了函数，取消了 u'XX'的字符串定义），完全支持 unicode，甚至可以使用中文定义变量名，但缺点是第三方库不够丰富。

本书使用的 Python 版本是 2.73，可以从 Python 的官方网站获取安装包，不用什么复杂的选项配置就可以完成安装。

13.1.3 编译 python 库

boost.python 库需编译才能使用，要求前提是已经安装了 Python（版本高于 2.2）环境。

编译 python 库的 bjam 命令如下：

```
bjam --with-python [--buildtype=complete] [stdlib=stlport] [stage]
```

如果使用源码嵌入工程编译的方式，则需要在编译环境里指定头文件“Python.h”的包含路径（GCC 使用-I 选项指定，VC 在 options 中设置），默认情况下是“/usr/include/python2.7”或“C:/Python27/include/”，例如：

```
g++ -c pyprebuild.cpp -I/usr/include/python2.7
```

参考 python 库的 jamfile 即可完成内嵌预编译 cpp 源文件：

```
//pyprebuild.cpp
```

```
#define BOOST_PYTHON_SOURCE
//#defineBOOST_PYTHON_NO_LIB

#include<libs/python/src/numeric.cpp>
#include<libs/python/src/list.cpp>
#include<libs/python/src/long.cpp>
#include<libs/python/src/dict.cpp>
#include<libs/python/src/tuple.cpp>
#include<libs/python/src/str.cpp>
#include<libs/python/src/slice.cpp>

#include<libs/python/src/converter/from_python.cpp>
#include<libs/python/src/converter/registry.cpp>
#include<libs/python/src/converter/type_id.cpp>
#include<libs/python/src/object/enum.cpp>
#include<libs/python/src/object/class.cpp>
#include<libs/python/src/object/function.cpp>
#include<libs/python/src/object/inheritance.cpp>
#include<libs/python/src/object/life_support.cpp>
#include<libs/python/src/object/pickle_support.cpp>
#include<libs/python/src/errors.cpp>
#include<libs/python/src/module.cpp>
#include<libs/python/src/converter/builtin_converters.cpp>
#include<libs/python/src/converter/arg_to_python_base.cpp>
#include<libs/python/src/object/iterator.cpp>
#include<libs/python/src/object/stl_iterator.cpp>
#include<libs/python/src/object_protocol.cpp>
#include<libs/python/src/object_operators.cpp>
#include<libs/python/src/wrapper.cpp>
#include<libs/python/src/import.cpp>
#include<libs/python/src/exec.cpp>
#include<libs/python/src/object/function_doc_signature.cpp>
```


把 pyprebuild.cpp 加入工程即可完成 python 库的编译工作。

13.1.4 使用 python 库

python 库位于名字空间 boost::python，为了使用 python 库，需要包含头文件 <boost/python.hpp>，如果使用源码嵌入工程的编译方式，还需要在头文件前加入宏 BOOST_PYTHON_SOURCE，即：

```
#define BOOST_PYTHON_SOURCE                    //源码嵌入工程编译方式
#include <boost/python.hpp>
using namespace boost::python;
```

13.2 嵌入 Python

我们先从 python 库最简单的用法——嵌入 Python 语句开始。这种使用方式可以调用 Python 语言的标准库和第三方库，就像拥有了一群数量庞大的库函数，让 C++不费任何力气就拥有了脚本语言的操纵能力。

不过目前 python 库的嵌入功能没有它的扩展功能那么强大，有的操作还需要调用 Python API，但可以满足基本的要求。

嵌入 Python 语言需要链接 Python 的运行库 python27.lib，它在 C:/Python27/Libs 目录下，可以在 VC 的工程属性中设置链接库选项，但最好是使用预处理指令放在源码中，如：

```
#pragma comment(lib, "python27.lib")//VC 系列编译器支持这个指令
#define BOOST_PYTHON_SOURCE
#include <boost/python.hpp>
using namespace boost::python;
```

13.2.1 初始化解释器

在 C 程序中执行 Python 语句有一个标准流程：

- 首先要调用 Python API 函数 Py_Initialize()启动 Python 解释器；
- 解释器启动后，可以使用 Py_IsInitialized()来检查解释器是否已经成功启动；
- 在完成所有的 Python 调用后，使用 Py_Finalize()来清除解释器环境。

目前的 boost.python 库不完全遵循上面的流程，建议库文档不要通过执行 Py_Finalize()来清除环境，因此在 C++中只需要调用 Py_Initialize()就可以了[①]。

Python API 相当的简陋，而 python 库并没有对它进行封装，我们可以考虑自己实现一个初始化 Python 解释器的类 pyinit，它要比单纯的 API 函数更方便好用。

pyinit 使用 Py_InitializeEx()初始化解释器，并提供 isInitialized()检查解释器的状态，version()获得 Python 解释器的版本：

```
//pyinit.hpp
#include <boost/noncopyable.hpp>
#include <boost/python.hpp>
```

① 但经作者测试，似乎调用了 Py_Finalize()也无问题。

```
class pyinit: boost::noncopyable
{
public:
  pyinit(int initsigs = 1)
  {
      assert(initsigs == 0 || initsigs == 1 );
      Py_InitializeEx(initsigs);
  }
  ~pyinit(){/*Py_Finalize() ;*/}

  bool isInitialized()
  {       return Py_IsInitialized(); }
  const char* version()
  {       return Py_GetVersion();    }
};
```

pyinit 使用了 boost::noncopyable，但并没有实现为单件，因为它没有内部的数据或状态，不需要提供全局的访问点。

读者可以在 pyinit 的基础上增加更多的功能，让它更有用。

13.2.2 封装 Python 对象

python 库使用模板类 handle 和 object 封装了 Python API 中的 PyObject 类型。handle 是一个智能指针，一般情况下我们应当优先使用 object。

object 类封装了 PyObject，内部也使用了引用计数，使用起来就像 Python 语言中的原生变量，或者是 C++11 中的 auto 和 boost.any（参见 7.7 节）。它的类摘要如下：

```
class object
{
public:
     object();
     object(object const&);

     //模板构造函数，可以接受任何类型!
     template <class T> explicit object(T const& x);
     ~object();

     object& operator=(object const&);
     PyObject* ptr() const;
};
```

模板函数 extract<type>可以把 Python 对象转换成所需的值，它的用法很像 any_cast，如果无法转换则会抛出异常。

使用 object 及其子类，我们可以在 C++中编写类似 Python 的代码。示范 object 基本用法的代码如下：

```
#pragma comment(lib, "python27.lib")
#define BOOST_PYTHON_SOURCE
#include <boost/python.hpp>
#include "pyinit.hpp"
using namespace boost::python;

int main()
{
  pyinit pinit;                              //初始化 Python 环境

  object i(10);                              //一个 Python 变量,可以是任何类型
  i = 10 *i;                                 //像 Python 里一样操作
  cout << extract<int>(i) << endl;           //用 extract 转换类型

  object s("string");                        //一个 Python 字符串变量,
  string str = extract<string>(s * 5);       //把字符串增加五倍,再转换成 string
  cout << str << endl;
}
```

在基本的 object 之外，python 库还提供了许多 Python 语言中类型的对应物，如 list、dict、tuple，它们分别对应 Python 语言中的列表、字典和元组结构，有许多操作函数，使用方法和 Python 基本相同。

示范这些 Python 对应数据结构基本用法的代码如下：

```
#pragma comment(lib, "python27.lib")
#define BOOST_PYTHON_SOURCE
#include <boost/python.hpp>
#include "pyinit.hpp"
using namespace boost::python;

int main()
{
  pyinit pinit;

  //list 类型,注意需要名字空间限定以避免与 std::list 冲突
  python::list l;
  l.append("zelda");                         //与 Python 类似的操作,添加数据
  l.append(2.236);

  assert(len(l) == 2);                       //也可以用 len()获得长度
  assert(l.count("zelda") == 1);             //count()计算成员的数量
  cout << extract<double>(l[1]) << endl;
```

```
    l.sort();                                   //排序

    //tuple 类型,同样需要名字空间限定以避免与 boost::tuple 冲突
    python::tuple t = python::make_tuple("metroid", "samus", "ridley");
    assert(len(t) == 3);
    assert(string(extract<string>(t[-2])) == "samus");
                                                //可以用 Python 里的负数进行反序索引
    l.append(t);                                //把 tuple 加入到 list 中
    assert(len(t) == 3);

    python::str s(',');                         //字符串类型
    s = s.join(t);                              //连接 tuple 里的字符串
    cout << string(extract<string>(s)) << endl;

    dict d;                                     //字典类型,不会引起名字冲突
    d["mario"] = "peach";                       //可以用任意的 key/value 值对
    d[0] = "killer7";
    assert(d.has_key(0));
    assert(len(d) == 2);
}
```

object 还有很多用法，如用成员函数 attr()访问属性，直接使用 operator()调用 Python 函数，但通常这些代码都不如直接执行 Python 语句方便，把 Python 变量交给 Python 解释器去管理更好。

13.2.3 执行 Python 语句

启动 Python 解释器后，可以使用 python 库提供的 exe()系列函数执行 Python 语句，这些函数的声明如下：

```
object eval(str expression, object globals, object locals)
object exec(str code, object globals, object locals)
object exec_file(str filename, object globals, object locals)
```

这三个函数的功能类似，都可以执行 Python 语句，但有小的不同：eval()函数计算表达式的值并返回结果，exec()执行 Python 语句并返回结果，而 exec_file()则执行一个文件中的 Python 代码。

函数接口中的 globals 和 locals 参数是 Python 中的字典结构，是语句运行的全局和局部场景，通常这两个参数可以忽略，或者取__main__模块的名字空间字典。

示范简单执行 Python 语句的代码如下：

```
cout << pinit.version() << endl;              //显示 Python 的版本
cout << extract<int>(eval("3**3")) << endl;   //计算 3 的立方
```

```
exec("print 'hello python'");                    //输出语句
```

如果在 Python 语句使用了变量，那么必须要指定 globals 参数。使用 import()函数可以导入__main__模块，用成员函数 attr()获取属性，如：

```
object main_module = import("__main__");
object main_namespace = main_module.attr("__dict__");
```

示范较复杂的 Python 语句执行的代码如下：

```
#pragma comment(lib, "python27.lib")
#define BOOST_PYTHON_SOURCE
#include <boost/python.hpp>
#include "pyinit.hpp"
using namespace boost::python;

int main()
{
  pyinit pinit;

  //获取运行所需的名字空间
  object main_ns = import("__main__").attr("__dict__");

  //执行 for 循环
  string str = "for x in range(1,5):\n"
               "\tprint x";
  exec(str.c_str(), main_ns);                    //输出 1,2,3,4

  //定义一个 Python 函数,计算 x 的 y 次方
  char *funcdef = "def power(x, y):\n"
              "\t return x**y\n"
              "print power(5, 3)\n";
  exec(funcdef, main_ns);                        //输出 125
  object f = main_ns["power"];                   //使用名字空间字典获得函数对象
  cout << extract<int>(f(4, 2)) << endl;         //用 operator()执行

  //导入 re 模块,执行正则表达式功能,输出 True
  exec("import re", main_ns);
  exec("print re.match('c*','ccc') != None", main_ns);
}
```


13.2.4 异常处理

如果执行 Python 语句发生错误，python 库会抛出 error_already_set 异常，但不含有任何信息，需要调用 API 函数 PyErr_Print()向标准输出打印具体的错误信息。

为方便使用，可以给 pyinit 类再增加一个静态成员函数 err_print()：

```
class pyinit: boost::noncopyable
{
public:
  ...
  static void err_print()
  {   PyErr_Print();  }
}
```

可以这样使用：

```
try
{
  exec("import re");                          //没有指定 globals，将发生错误
}
catch (...)
{
  pyinit::err_print();
}
```

输出的错误信息可能是这样：

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: __import__ not found
```


13.3 扩展 Python

python 库能够在 C++程序中调用 Python 语言，但它更重要的功能在于用 C++编写 Python 扩展模块，嵌入到 Python 中解释器中调用，提高 Python 的执行效率。

python 库在第 1 版的基础上做了大量的改进，充分利用了 C++的高级特性和新技术，封装和屏蔽了许多底层实现，展现给外界的是一个高度抽象、灵活和易于学习的接口。只要用户会编写 C++程序，就可以立刻为 Python 编写扩展模块。

与原始 C API 的烦琐调用步骤相比，python 库在很大程度上简化了扩展模块的编写，使编写工作就像是在写 Python 语言，而且它还很好地处理了许多原来手工编程容易出错的地方，让开发者把精力集中在模块的主要逻辑上，而不是陷入管理 Python 对象引用计数等低级细节中。

扩展 Python 我们同样需要编译 python 库，并包含头文件<boost/python.hpp>，但不必指明 Python 的运行库即可自动链接（需设定库文件搜索路径，默认是 C:\Python27\libs），即：

```
#define BOOST_PYTHON_SOURCE                   //源码嵌入工程编译方式
#include <boost/python.hpp>
```

```
using namespace boost::python;
```

13.3.1 最简单的例子

在本节中我们要实现一个最简单的功能，向 Python 导出一个无参的 hello()函数，它将打印出“hello boost python”字符串：

```
#include <string>
using std::string;

string hello_func()                                    //返回一个字符串
{ return "hello boost python";}
```

python 库编写扩展模块非常容易，由于使用了模板元编程等高级技术，代码量非常少，而且看起来非常清晰易懂。

首先我们要在 VC 中建立一个 DLL 工程，名字叫 boostpy，是一个不使用预编译头的空工程，不要忘记设置工程字符集的 Not Set 和运行库多线程的 MT 或 MTd 属性，并加入宏“_STLP_DEBUG”和“__STL_DEBUG”（如果使用了 STLport）。

然后我们在工程中新增一个 boostpy.cpp 文件，并加入 13.1.3 节定义的 python 库嵌入源码编译文件 pyprebuild.cpp。

接下来我们在 boostpy.cpp 中编写函数导出代码，使用 BOOST_PYTHON_MODULE 宏定义 Python 模块名，模块名必须与 dll 的名字相同，，当然也可以在编译后改 dll 的名字。

宏 BOOST_PYTHON_MODULE 的用法很像 test 库的单元测试套件宏，在宏 BOOST_PYTHON_MODULE 定义的模块内部我们使用 def()函数定义要导出的函数，需要指定导出的名字和 C++的函数名字，它的语法一定程度上模仿了 Python 语言的函数定义关键字 def。

下面列出了例子的全部代码：

```
#define BOOST_PYTHON_SOURCE                            //源码嵌入工程编译方式
#include <boost/python.hpp>
using namespace boost::python;                         //打开名字空间

string hello_func(){...}                               //C++函数定义

BOOST_PYTHON_MODULE(boostpy)                           //Python 模块定义开始
{

  //导出一个名字为 hello 的函数，其 doc string 是“函数说明字符串”
  def("hello", hello_func, "函数说明字符串");
}
```

不需要担心这个简短的程序中没有 DllMain()、WINAPI 等 Windows 编程中常见的 dll 导出元素，也不需要写 def 或者 exp 文件，python 库为我们在幕后做了一切，将自动生成一个完全可用的动态链接库 boostpy.dll。

但这个 dll 不能直接被 Python 环境识别，必须把后缀名改成标准的 Python 模块后缀名 pyd，即 boostpy.pyd（也可以修改 VC 工程设置，直接生成后缀是 pyd 的 dll 文件），然后放置到 Python 环境可以找到的路径下——通常是 Python 主目录或者主目录下的 lib 目录（默认即 C:/Python27 和 C:/Python27/lib）。

我们首先使用 Python 交互解释器 IDLE 测试这个最小的 Python 扩展模块，下面显示结果中的“>>>”是 IDLE 的输入提示符：

```
>>> import boostpy                          #导入 boostpy 模块
>>> boostpy.hello()                         #直接执行 hello()函数
'hello boost python'
>>> print boostpy.hello()                   #使用 print 语句输出结果
hello boost python
```

也可以使用 help()函数来查看 boostpy 模块的信息：

```
>>> help(boostpy)
Help on module boostpy:
NAME
   boostpy
FILE
   c:\python27\boostpy.pyd
FUNCTIONS
   hello(...)
      hello() -> str :
          函数说明字符串
          C++ signature :
              class stlpd_std::basic_string<char,class stlpd_std::char_traits
                <char>,class stlpd_std::allocator<char> > hello()
```

boostpy 模块也可以在 Python 脚本中运行，使用脚本运行扩展模块时不要求 pyd 模块必须在 Python 主目录下，只要和脚本在同一个目录即可。

例如下面的 test.py 脚本：

```
#coding:utf-8
#file test.py
import boostpy                              #导入 boostpy 模块
print boostpy.hello()                       #使用 print 语句输出结果
```

运行这个 Python 脚本将与 IDLE 交互解释器的运行结果一样，输出“hello boost python”。

13.3.2 导出函数

python 库使用模板函数 def()来导出 C++函数到 Python，它有多个重载形式，声明是：

```
template <class F>
void def(char const* name, F f,...);
```

def()函数要求导出的名字必须是 C 字符串（以 NULL 结束的字符数组），不能是 std::string 对象，第二个参数是类型为 F 的函数指针，这之后可以添加文档字符串以及函数的参数列表等函数的附加信息。

向 Python 导出函数的参数需要使用 python 库中的参数关键字类 arg，它的类摘要如下：

```
struct arg
{
  explicit arg (char const *name);
  template <class T>  arg &perator = (T const &value);
  arg operator,(char const *name) const;
  arg operator,(python::arg const &k) const;
};
```

arg 类的构造函数接受一个 C 字符串作为参数的导出名字，得益于 Python 的动态语言特性，我们无需指定它表示的参数的类型。

为了支持 C++和 Python 的缺省参数特性，arg 重载了 operator=，可以指定参数的缺省值。它还重载了逗号操作符，可以使用逗号表达式把 arg 参数连接起来（很像 assign 库的用法）。但在 def()函数中使用时，为了不与函数参数分隔的逗号混淆，我们需要把 arg 逗号表达式用圆括号括起来。

在上一节的基础上，我们再定义两个 C++函数用于导出：

```
string hello_to(const string& str)            //接受一个字符串参数
{ return "hello " + str;}

string hello_x(const string& str, int x)      //接受字符串和整数参数
{
  string tmp = "hello ";
  for (int i = 0 ; i < x ; ++i)
  {       tmp += str + " ";   }
  return tmp;
}
```

然后我们在宏 BOOST_PYTHON_MODULE 定义的导出模块中导出它们，并使用 arg 类定义它们在 Python 中的参数：

```
def("helloto", hello_to, arg("str"));         //定义一个参数
```

```
def("hellox", hello_x, (arg("str"), "x"));   //使用逗号操作符
```

在导出函数时参数名不一定非要与 C++中的一致，可以是任意的名字，只要它符合 Python 的命名规则即可，比如名字可以是“a_bit_long_name_of_arg_type_is_str”。

编译生成新的 pyd 文件后，可以使用下面的 Python 脚本调用验证：

```
print boostpy.helloto('boost')
print boostpy.hellox('C++', 5)
```

Python 脚本运行结果如下：

```
hello boost
hello C++ C++ C++ C++ C++
```

python 库另外提供了一个便捷函数 args()，可以用在不需要指定参数值的时候直接使用参数名字符串生成多个 arg 对象，例如：

```
def("hellox", hello_x, args("str", "x"));
```

13.3.3 导出重载函数

C++和 Python 中都有重载函数的概念，它们可以名字相同但参数和返回值不同。在向 Python 导出 C++重载函数的时候不能使用之前的 def()形式，因为无法从函数名字区分出重载函数，必须使用函数指针类型定义。

使用函数指针手工导出

我们把之前的三个 hello()系列函数都改为重载函数，名字为 hello_func()，然后我们定义三个函数指针 fp1、fp2、fp3：

```
string (*fp1)()                    = &hello_func;
string (*fp2)(const string&)       = &hello_func;
string (*fp3)(const string&, int) = &hello_func;
```

然后就可以使用 def()函数导出函数指针，由于函数指针定义已经包含了参数信息，因此我们无需特意使用 arg 类定义参数。当然使用 arg 也是允许的，可以更好地在 Python 中描述参数信息：

```
BOOST_PYTHON_MODULE(boostpy)                    //Python 模块定义开始
{
  def("hello", fp1, "doc string1");
  def("hello", fp2, "doc string2");
  def("hello", fp3, (arg("str"),arg("x")), "doc string3");
}
```

这三个函数的 Python 调用脚本如下：

```
import boostpy  #导入 boostpy 模块
print boostpy.hello()
print boostpy.hello('boost')
print boostpy.hello('C++',5)
```

使用宏自动导出

如果程序中有大量的重载函数，那么手工定义函数指针的工作将会很烦琐，因此 python 库特意提供了一个方便的宏 BOOST_PYTHON_FUNCTION_OVERLOADS，专门用于简化重载函数的定义，它可以自动产生重载函数说明，声明如下：

```
#define BOOST_PYTHON_FUNCTION_OVERLOADS(generator_name, fname, min_args, max_args)
```

宏 BOOST_PYTHON_FUNCTION_OVERLOADS 使用四个参数，第一个参数 generator_name 用于生成一个辅助类，包含重载函数的定义，第二个参数 fname 指定重载函数的名字，最后两个参数指定重载函数参数的最小和最大参数个数。

使用 BOOST_PYTHON_FUNCTION_OVERLOADS 有一点限制，要求重载函数必须具有顺序相同的参数序列，即少的参数序列是多的参数序列的子集。例如我们的 hello()系列函数的参数序列依次是：(void)、(string)、(string,int)，如果第三个函数的参数是(int,string)，那么它就不满足 BOOST_PYTHON_FUNCTION_OVERLOADS 的要求，宏只能用于前两个函数。

宏 BOOST_PYTHON_FUNCTION_OVERLOADS 可以这样使用：

```
BOOST_PYTHON_FUNCTION_OVERLOADS(hello_overloads, hello_func, 0, 2)
```

其中 hello_overloads 是我们为重载函数指定的辅助类名，hello_func 是重载函数的名字，数字 0 和 2 表示有三个重载形式，参数最少是 0 个，最多是 3 个。

在使用 def()导出前，我们还必须定义最多参数的重载函数指针类型，即：

```
typedef string (*hello_ft)(const string&, int);
```

然后我们就可以向 Python 一次性导出全部重载函数：

```
def("hello",(hello_ft)0, hello_overloads());
```

注意 def()函数中第二个参数，即函数指针参数的用法，我们把一个空指针转换成 hello_ft 函数指针类型，然后再用辅助类的临时对象 hello_overloads()以导出所有函数。

我们也可以仍然使用函数指针，但需要用最多参数的那个函数指针：

```
def("hello",fp3, hello_overloads());
```

BOOST_PYTHON_FUNCTION_OVERLOADS 将把重载函数导出为一个 Python 函数，它具有 max_args 个缺省参数，使用 Python 的 help()可以看到导出的函数说明。

BOOST_PYTHON_FUNCTION_OVERLOADS 的另一个用法是导出具有缺省参数值的函数，这种函数就像是有 *N* 个重载形式的函数，例如：

```
string hello_func(const string& str="boost", int x = 5);
typedef string (*hello_ft)(const string&, int);
def("hello",(hello_ft)0, hello_overloads());
```

混合使用手工重载和自动重载方式，我们就能够很好地减轻编写导出重载函数的工作负担，对于有相同参数序列或者缺省参数的函数使用宏 BOOST_PYTHON_FUNCTION_OVERLOADS，其他的则使用手工定义函数指针的方式。

13.3.4 导出类

python 库的另一个强大的功能是可以方便地把 C++类导出为 Python 类，这在 python 库出现前是一件非常烦琐且容易出错的工作。python 库使用一个类似 Python 语法的模板类“class_”封装了这项工作。

模板类 class_有很多成员函数，因为它必须要能够支持 C++和 Python 两种语言的各种语义，它的类摘要如下：

```
template <class T , class Bases = bases<> >
class class_ : public object
{

    //构造函数，产生 Python 的缺省初始化函数__init__
    class_(char const* name);

    //构造函数，指定初始化函数__init__
    template <class Init>  class_(char const* name, Init);

    //导出其他初始化函数
    template <class Init>  class_& def(Init);

    //导出成员函数
    template <class F>  class_& def(char const* name, F f);

    //导出 attribute
    template <class U>
    class_& setattr(char const* name, U const&);

    //导出静态成员函数
    class_& staticmethod(char const* name);
```

```
    //导出成员变量
    template <class D>
    class_& def_readonly(char const* name, D T::*pm);
    template <class D>
    class_& def_readwrite(char const* name, D T::*pm);

    //导出静态成员变量
    template <class D>
    class_& def_readonly(char const* name, D const& d);
    template <class D>
    class_& def_readwrite(char const* name, D& d);

    //导出 property
    template <class Get>
    void add_property(char const* name, Get const& fget, char const* doc=0);
    template <class Get, class Set>
    void add_property(
       char const* name, Get const& fget, Set const& fset, char const* doc=0);
    template <class Get>
    void add_static_property(char const* name, Get const& fget);
    template <class Get, class Set>
    void add_static_property(char const* name, Get const& fget, Set const& fset);
};
```

class_类的用法与 def()函数基本相同，它导出模板参数 T 类型为 Python 类，再用成员函数 def()、def_readonly()等分别导出 T 的成员函数和成员变量。

例如，我们把之前的 hello()系列函数改为一个简单的 C++类：

```
class demo_class
{
private:
  string msg;
public:
  static string s_hello;
  demo_class():msg("boost"){}                    //缺省构造函数
  string hellox(int x = 1)
  {
      string tmp = s_hello;
      for (int i = 0 ; i < x ; ++i)
      {   tmp += msg + " ";       }
      return tmp;
  }
};
string demo_class::s_hello = "hello ";          //静态成员变量初始化
```

那么使用 class_可以这样导出类：

```
BOOST_PYTHON_MODULE(boostpy)
{
  class_<demo_class>("demo", "doc string")
      .def("hello", &demo_class::hellox, arg("x")=1)
      .def_readwrite("shello", &demo_class::s_hello);
}
```

class_首先用模板参数指定了导出类 demo_class，然后在构造函数中指定了导出名和文档字符串。随后用 def()导出成员函数，用法与导出普通函数类似，但对于成员函数我们必须写出全名，并且使用取地址操作符&。最后我们使用 def_readwrite()导出了成员变量。

Python 调用脚本如下：

```
from boostpy import *          #导入 boostpy 模块
d = demo()                     #使用缺省构造函数
print d.hello()                #缺省参数 1，输出：hello boost
print d.shello                 #访问成员变量，输出：hello
d.shello = 'goodbye '          #修改成员变量
print d.hello(2)               #输出:goodbye boost boost
```

13.3.5 导出类的更多细节

本节将在 13.3.4 节的基础上讲解 python 库导出类的更多用法。

构造函数

13.3.4 节讲述导出类时我们没有指定类的构造函数，因此 Python 在创建对象时将使用缺省构造函数。但很多情况下缺省构造函数是不够的，带参数的构造函数更加常见。

我们不能使用 def()来导出构造函数，因为 C++中的构造函数不同于普通的成员函数，最重要的区别是不能取它的地址，即没有这样的语法：

```
&demo_class::demo_class
```

因此，python 库使用模板类 init<...>和 optional<...>来共同定义构造函数和构造函数中的缺省参数。它们的模板参数都是构造函数的参数类型，init 中的参数是必须出现的，而 optional 中的参数是有缺省值可以不出现的，它们的用法很像定义重载构造函数。

假设我们为 demo_class 增加一个构造函数：

```
demo_class(const string& str = "python")
{ msg = str;  }
```

那么我们既可以直接在 class_的构造函数中指定 init，也可以在 def()中指定 init：

```
class_<demo_class>("demo", "doc string",
```

```
init<optional<string> >("init doc") )
```

或：

```
class_<demo_class>("demo", "doc string")
  .def(init<optional<string> >("init doc"))
```

这两种导出形式在语义上是不同的。第一种形式指定了一个有缺省参数的构造函数，而第二种形式先指定了一个缺省构造函数，然后又指定了另外一个有缺省参数的构造函数，因此第二种形式要求被导出的类必须有缺省构造函数。

导出 property

使用 def_readonly()和 def_readwrite()我们可以导出 C++类的成员变量，同时指定它的读写属性，但这两个函数要求类的成员变量必须是 public 的。通常在 C++中很少有 public 的成员变量，它们总是被封装为 private 或者 protected 拒绝外界的直接访问，而使用访问函数来间接地存取值。

在 Python 语言中用“property”来表示类似的概念，python 库使用 add_property()和 add_static_property()来导出 property，前者用于操作对象 property，后者用于操作类 property（静态成员变量），它们的用法与 def()基本相同。

如果我们要把 demo_class 的静态成员变量 s_hello 导出为 Python 属性，首先要定义它的访问函数：

```
static void set(const string& str)
{ s_hello = str;  }
static string get()
{ return s_hello ;    }
```

因为这两个访问函数是静态成员函数，因此我们需要使用 add_static_property()，否则可以使用 add_property()：

```
class_<demo_class>("demo", "doc string")
  .add_static_property("rshello", &demo_class::get)
  .add_static_property("rwshello", &demo_class::get, &demo_class::set)
```

这样我们就导出了两个 Python property，rshello 是只读 property，而 rwshello 则可读可写。

导出 attribute

attribute 与 property 在 Python 中是两个相似又有区别的概念，python 库使用 setattr()函数来导出 attribute。它的用法与 def()相同，但导出到 Python 中的将是一个 attribute，而不是一个普通的 Method。

例如，我们可以将 get()函数改为导出成一个 attribute：

```
class_<demo_class>("demo", "doc string")
  .setattr("get", &demo_class::get);
```

staticmethod()函数配合 setattr()可以导出 C++类的静态成员函数，成为 Python 中的一个静态方法。因为静态成员函数已经被 setattr()定义过了，因此 staticmethod()只需要指定导出名。例如：

```
class_<demo_class>("demo", "doc string")
  .setattr("get", &demo_class::get)              //先定义 attribute
  .staticmethod("get");                          //再定义静态方法
```

重载成员函数

对于导出类的重载函数，可以使用手工定义成员函数指针的形式，也可以使用简化的工具宏，但使用的是另外一个宏 BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS，除了名字不同，它与 BOOST_PYTHON_FUNCTION_OVERLOADS 的用法完全相同。

例如，使用这个宏封装了有缺省参数的 hellox()的代码如下：

```
BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS(hello_overloads,hellox,0,1)
class_<demo_class>("demo", "doc string")
  .def("hello",&demo_class::hellox, hello_overloads())
```

导出继承关系

class_也支持导出 C++类的继承关系，这样子类将自动获得父类导出的 Python 属性和方法，而且在 Python 中也会有多态的特性。

导出继承关系很简单，需要使用 class_的第二个模板参数，在这里用模板类 bases 指出基类，例如，假设我们有一个 demo_class 的子类 derived，那么它的导出继承关系代码如下：

```
class derived :public demo_class{};              //简单的派生类
class_<derived, bases<demo_class> >("derived");
```

在导出继承关系时，通常让父类有一个虚的析构函数是个好主意，这可以让 Python 能够正确地使用父类指针销毁子类对象。

13.3.6 高级议题

本节讨论关于 python 库扩展用法的一些高级议题。

重载操作符

C++和 Python 都支持重载操作符，不同的是 C++使用 operator#()的形式，而 Python 则定义了若干内部方法，如_add_、_sub_等。python 库使用一个 self 对象模仿 Python 语法提供了重载操作符的支持。

我们使用 9.2 节介绍的 rational 类作为导出重载操作符的例子，代码很简单，几乎是自说明的：

```
typedef boost::rational<int> rational;                  //typedef 用来简化代码
class_<rational >("rational", "boost rational")         //缺省构造函数
  .def(init<int, int>())                                //分子分母的构造函数
  .setattr("n", &rational::numerator)                   //获得分子
  .setattr("m", &rational::denominator)                 //获得分母
  .def(self + int()).def(int() + self)                  //重载对 int 的加法
  .def(self - int())                                    //重载对 int 的加法
  .def(self + self ).def(self  - self)                  //有理数之间的加减法
  .def(self < self);                                    //比较操作符
```

对应的 Python 脚本如下:

```
from boostpy import *                                   #导入 boostpy 模块
r = rational(1, 5);                                     #有理数 1/5
print r.n(), r.m()                                      #输出 1 5
r = r + 5                                               #重载操作符加法运算
print r.n(), r.m()                                      #输出 26 5
print rational() < r                                    #缺省构造有理数 0,输出 True
```


导出枚举类型

python 库使用 enum_类来导出 C++的枚举类型，用法与 class_基本相同，下面的代码说明了它的用法：

```
enum demo_enum{sha1, md5, md2};                         //枚举几个常见的摘要算法
BOOST_PYTHON_MODULE(boostpy)                            //Python 模块定义开始
{
  enum_<demo_enum>("digest")                            //使用 value 函数导出枚举值
      .value("sha1", sha1)
      .value("md5", md5)
      .value("md2", md2);
}
```

作用域

默认情况下我们导出的所有 Python 对象都在全局作用域，但有时候 C++类型会嵌套在其他类中，而我们需要导出这种嵌套关系。

scope 类封装了 Python 中作用域的概念，一个缺省构造的 scope 对象保存了当前的作用域。scope 也可以被赋值为一个 class_对象，这时作用域将变为 class_对象所对应的域，所有接下来的导出都在这个域之内，一直持续到 scope 对象析构为止。例如：

```
struct demo1
{
  struct demo2
  {   int f(){return 10;} };
};
BOOST_PYTHON_MODULE(boostpy)                    //Python 模块定义开始
{
  scope s = class_<demo1>("demo1");             //开始一个内部作用域
  class_<demo1::demo2>("demo2")
      .def("f", &demo1::demo2::f);
}
```

其他功能

python 库还有许多其他的功能用于支持 C++语法到 Python 的转换，如导出虚函数、定义 Python 迭代器、转换 C++异常到 Python 异常、Python 的序列化模块 Pickle 等等，而且还有两个用 Python 编写的自动代码生成器 pyste 和 py++，可以大大简化扩展 Python 功能的代码编写工作。

这些功能在 Boost 的说明文档中都有详细描述，读者可以自行阅读参考。

13.4 总结

现实世界中存在很多种编程语言，C++无疑是其中最强大最富表现力的一种，但俗话说“一个篱笆三个桩，一个好汉三个帮”，C++也需要与其他的语言互相配合，共同构建复杂的软件系统。

C++可以为几乎所有的编程语言编写基础模块，为它们提供扩展功能，例如 Perl、Python、Java、Fortran 等等，但目前 Boost 程序库仅提供了对 Python 语言的支持，即 python 库。

boost.python 是一个功能非常强大的库，对 C++和 Python 的双方向转化提供了无缝的操作，它有许多复杂的用法和深入的细节，涉及 C++和 Python 两种语言的许多语法的细枝末节，本章只能讲述其中基本的一些知识。但相信读者有了这些基础知识，再深入学习 python 库将不再是难事。

python 库有两种使用方式：嵌入 Python 和扩展 Python。

我们首先讨论了在 C++中嵌入 Python 语言，这种用法需要链接 Python 运行库。python 库

使用 object 类封装了 Python 对象，并有 str、list、tuple、slice 等 Python 对应的类，可以在 C++中编写 Python 风格的代码。python 库也提供了三个函数可以直接执行 Python 语句，这种方式通常更方便，使我们可以直接执行已经写好的 Python 脚本。

python 库也为 C++扩展 Python 提供了完善的支持，几乎所有的 C++语言特性都能通过 python 库翻译到对应的 Python 模块，包括重载函数、函数缺省参数、枚举、类的构造函数、嵌套类、虚函数、继承，等等。通过使用 def()和 class_、enum_等函数和类，可以编写非常简洁的描述式代码来完成导出 C++对象的工作，而且这种导出是非侵入的，不需要对原有的代码做任何的变动。

基于 python 库可以构建出 C++与 Python 的混合软件系统：

使用 Python 的动态特性、解释能力和大量的标准模块，我们能够快速构建出可用的软件原型，然后用 C++改写其中运行效率低的部分，作为底层模块供 Python 调用。这样我们既拥有 Python 的快速开发能力，又有了 C++的运行高效率。python 库为我们提供了 C++与 Python 的任意转换能力，可以把软件系统中的任何一个模块用任意语言实现，系统中语言所占的比例因需求不同而变化，如果侧重快速开发，那么大部分代码都会是 Python 编写的，如果侧重运行效率，那么情况就会相反。

本章实现了一个比较有用的 C++工具类 pyinit，它包装了一些 Python API 函数，提供了一个方便易用的接口，可以简化嵌入 Python 的工作。