使用 Boost.Python 構建混合系統

Building Hybrid Systems with Boost.Python

Author:	David Abrahams
Contact:	dave@boost-consulting.com
Organization:	Boost Consulting
Date:	2003-03-19
Author:	Ralf W. Grosse-Kunstleve
Translation:	王志勇(JerryWang_cn@msn.com)

Table of Contents

概要

Boost.Python 是一個開源 C++ 庫，她提供了一個簡明的 IDL 式的接口用於綁定 C++ 類和函數到 Python。 得益於 C++ 編譯期的內部處理(譯註：原文是 introspection，我不知道怎麼翻譯合適)和最近開發的元編程(metaprogramming)技術，成就了 Boost.Python 不需引入一種新的語法而只用純 C++ 的實現。Boost.Python 豐富的特性集合以及她的高 階接口使得工程師像混合系統(譯註：hybrid system，我聽說過油/電混合動力系統)那樣做打包的事情成為可能，並且程序員讓在應用 C++ 高效的

編譯期多態性以及 Python 非常方便的運行期多態性的時候獲得易用性和一致性；

介紹

Python 和 C++ 在很多方面相當的不同：C++ 一般編譯為機器碼，Python 是解釋處理的。Python 的動態類型系統(dynamic type system)作為語言具有靈活性的基礎常常被提及，然而 C++ 的靜態類型系統是她效率的基石。C++ 擁有一種複雜和難以理解的編譯期元語言(compile-time meta-language)，然而在 Python 裡頭幾乎所有事情都發生在運行期。

但是對於很多程序員來說，這些很大的不同也意味著 Python 和 C++ 可以完美的互補。Python 程序內性能瓶頸的部分可以用 C++ 重寫以帶來最高的運行速度， 並且強大的 C++ 庫的作者們可以選擇 Python 作為中間件語言，利用她靈活的系統集成能力。此外， 表面上的不同也掩蓋了一些非常相似之處：

- C 語言家族的控制結構(if, while, for...)
- 支持面向對象、函數化編程以及普通語言(these are both multi-paradigm programming languages.)
- 全面的操作符重載機制，重視為了代碼可讀性和表達性而提高語法可變性
- 高層概念，例如集合和迭代器(collections and iterators)
- 上層封裝機制(C++: namespace, Python: modules)以便支持重用庫的設計
- 異常捕獲機制用於錯誤情況管理
- C++ idioms in common use, such as handle/body classes and reference-counted smart pointers mirror Python reference semantics

提供給 Python 豐富的『C' 互操作 API，應當尊許一個法則：導出 C++ 的類型和函數接口給 Python 的時候，儘量用和 C++ 相似的接口。然而，Python 自己提供的 C++ 集成接口 是非常貧乏的。比較 C++ 和 Python ，『C' 只有非常基本的抽象能力，而且不支持異常處理機制。『C' 擴展模塊的作者被要求手動管理 Python 的引用計數，那 是非常麻煩而且容易出錯的事情。傳統的擴展模塊往往包含大量的重複的『樣板代碼』，使得代碼難以維護，特別是當你封裝一個發展中的 API(譯註：指還未完善的 API)的時候。

這些限制導致出現了大量的 Python 封裝系統。 SWIG 大概是最流行的一種用於集成 C/C++ 和 Python 的系統。最近出現的一種是 SIP，特別設計用於集成 Python 和 Qt 圖形用戶接口。SWIG 和 SIP 都引入了他們自己的專門語言用於定製語言間綁定。這麼做有一定的好處，但是你不得不去處理三種語言(Python 、C/C++以及引入的接口語言)，所以也帶來了實際困難。CXX 則演示了另外一種有趣的 選擇。她證明了至少部分的 Python 的 'C' API 可以被更友好的 C++ 接口封裝。然而，不像 SWIG 和 SIP 那樣，CXX 不包含對封裝 C++ 類和 Python 新類型(new Python types)的支持。

Boost.Python 的特性和目標與很多其它的封裝系統是一樣的。 就是說 Boost.Python 企圖提供最大化的易用性和靈活性，但是並不引入一種獨立的封裝語言。取而代之的是，她提供高級的 C++ 接口給用戶用於封裝 C++ 類和函數，並且通過靜態元程序(static metaprogramming)管理大量內部的複雜性。Boost.Python 超越了早期 封裝系統提供的特性，包括：

- 支持能夠被 Python 重載的 C++ 虛函數。
- 對於低階的 C++ 指針和引用(low-level C++ pointers and references)，提供全面的生命期管理機制。

- 支持把擴展功能封裝為 package，通過中心的註冊機製作語言間類型轉換。
- 通過一種安全和易用的方法，用於引入 Python 強大的序列化引擎(pickle)。
- 與 C++ 對 lvalues、rvalues 的處理機制一致，所以可以對 C++ 以及 Python 的類型系統深入的理解（譯註：明白了一個就明白了另外一個）。

開發 Boost.Python 最主要的目的是，通過使用 C++ 的編譯期內部處理(原文是：introspection)，大量的傳統擴展模塊樣板代碼可以被排除。每個封裝的 C++ 參數必須從一個 Python 對象中取得，根據參數的類型使用不同的過程(procedure)進行處理。同樣地，函數的返回類型決定了返回值怎樣從 C++ 到 Python 進行轉換。當然，參數和返回值的類型是每個函數類型的一部分，這就是 Boost.Python 得出大部分所需信息的源頭。

這種方法引入了*用戶指導封裝*：在純 C++ 的框架範圍內，使得直接導出(到 Python)的信息和被封裝的源代碼一樣多成為了可能，一些額外的信息由用戶顯式 地提供。通常這種『指導』是機械化的幾乎不需要用戶實際的干涉。因為接口的規範是用和表述代碼同樣的全特性語言描述的，用戶擁有了空前的能力，當他想進行控制的時候。

Boost.Python 的設計目標

Boost.Python 的主要設計目標是讓用戶可以只通過 C++編譯器，就能在 Python 內使用 C++ 類和函數。簡單說，用戶感覺就是直接從 Python 裡面操作 C++ 對象。

然而，不逐個轉換所有接口也是很重要的：每種語言的習慣必須被尊重。例如，儘管 C++ 和 Python 都有迭代器(iterator)的概念，他們的表達方式卻很不同。Boost.Python 不得不具有能把他們結合在一起的能力。

把 Python 用戶從 C++ 接口裡的瑣碎錯誤中隔離開必須是可能的，例如訪問已經被刪除了的對象。同樣的，必須使 C++ 用戶從低階 Python 'C' API 隔離開來，用更好的選擇去替換掉像手工進行*引用計數管理*和*新的 PyObject 指針管理*這些事情，他們都是容易導致錯誤的 'C' 接口。

支持基於組件的開發也是至關重要的，所以，一個擴展模塊中的 C++ 導出類型可以被傳遞到另外一個模塊導出的函數內應用，而且不丟失任何重要信息，例如 C++ 的繼承關係。

最後，所有的封裝必須是*非干擾性*的，不能修改甚至只是『查看』原始的 C++ 代碼。現存的 C++ 庫對於只有 header 文件和二進制文件的第三方來說，已經是可封裝的了。

Hello Boost.Python World

現在先來個 Boost.Python 的預覽，看看她是如何改進 Python 的原有機制的。下面是我們可能要用於 Python 的函數：

```
char const* greet(unsigned x)
{
   static char const* const msgs[] = { "hello", "Boost.Python", "world!" };

   if (x > 2)
       throw std::range_error("greet: index out of range");
```

```
   return msgs[x];
}
```

使用 Python 的 'C' API 封裝這個函數，我們需要這樣做：

```
extern "C" // all Python interactions use 'C' linkage and calling convention
{
    // Wrapper to handle argument/result conversion and checking
    PyObject* greet_wrap(PyObject* args, PyObject * keywords)
    {
         int x;
         if (PyArg_ParseTuple(args, "i", &x))    // extract/check arguments
         {
             char const* result = greet(x);      // invoke wrapped function
             return PyString_FromString(result); // convert result to Python
         }
         return 0;                               // error occurred
    }

    // Table of wrapped functions to be exposed by the module
    static PyMethodDef methods[] = {
        { "greet", greet_wrap, METH_VARARGS, "return one of 3 parts of a
greeting" }
        , { NULL, NULL, 0, NULL } // sentinel
    };

    // module initialization function
    DL_EXPORT init_hello()
    {
        (void) Py_InitModule("hello", methods); // add the methods to the module
    }
}
```

現在，這是使用 Boost.Python 的封裝代碼：

```
#include <boost/python.hpp>
using namespace boost::python;
BOOST_PYTHON_MODULE(hello)
{
    def("greet", greet, "return one of 3 parts of a greeting");
}
```

這裡演示了如何使用：

```
>>> import hello
>>> for x in range(3):
...     print hello.greet(x)
...
hello
```

```
Boost.Python
world!
```

實際上'C' API 的版本更冗長，it's worth noting a few things that it doesn't handle correctly:

- 原始函數接受一個無符號整數參數，然而 Python 'C' API 只提供了提取有符號整數的方式給我們。如果我們傳遞一個負數給 `hello.greet`，Boost.Python 版本會 拋出一個 Python 異常，但是另外一個版本將會執行下去，不管 C++ 的實現中在什麼時候轉換負整數到無符號數(通常封裝成很大的數)，然後傳遞**不正確的**轉換過的參數到被封裝的函數。
- 這帶給了我們第二個問題：如果 C++ 的 `greet()`函數被一個比 2 大的參數調用，它會拋出一個異常。典型地，如果一個 C++ 異常通過'C'編譯器生成的代碼 的邊界進行傳遞，會引起崩潰(crash)。像你在第一個版本中看到的那樣，那裡沒有阻止它(crash)發生的 C++ 機制。Boost.Python 封裝的函數自動包含了一個異常處理層，它能通過轉換未捕獲的 C++ 異常到對應的 Python 異常以保護 Python 用戶。
- 有點更微妙的限制是：使用 Python 'C' API 進行參數轉換的示例只能用一種方式取得整數 x。PyArg_ParseTuple 無法轉換 Python 的 `long` 對象(任意精度整數)它正好適合一個 `unsigned int` 而不是 `signed long`，也不能通過一個封裝的帶有用戶顯式定義的 `operator unsigned int()` 的 C++ 類來轉換 。Boost.Python 的動態類型轉換註冊(dynamic type conversion registry)允許用戶添加任意的轉換方法。

Library Overview

這一部分描述了庫的主要特性。為了避免混亂，庫的實現細節被省略了。

導出 Classes

C++ 類和結構是用同樣簡潔的接口導出的：

```
struct World
{
    void set(std::string msg) { this->msg = msg; }
    std::string greet() { return msg; }
    std::string msg;
};
```

下面的代碼會導出它到我們的擴展模塊：

```
#include <boost/python.hpp>
BOOST_PYTHON_MODULE(hello)
{
    class_<World>("World")
        .def("greet", &World::greet)
        .def("set", &World::set)
    ;
}
```

儘管這些代碼有某種 pythonic familiarity(譯註：或許是 Python 風格的意思)，人們有時還是發現這種語法有點令人迷惑，因為它看上去不像他們過去使用的 C++ 代碼。其實，這就是標準 C++ 的實現。由於他們的靈活的語法和操作符重載，C++ 和 Python 在定義 domain-specific (sub)languages (DSLs) 上是非常出色的，那就是我們在 Boost.Python 裡面做的。把它拆開看：

```
class_<World>("World")
```

構造一個未命名的 `class_<World>` 類型的對象並且傳遞 `"World"` 到它的構造函數。這就在擴展模塊裡面創造了一個新的 Python class 叫作 `World`，並且把它在 Boost.Python 類型轉換註冊 (type conversion registry) 裡頭和 C++ 類型 `World` 關聯起來了。我們可能也會寫下：

```
class_<World> w("World");
```

但是那會顯得更冗長，因為我們不得不再次通過 `w` 去調用 `def()` 成員函數：

```
w.def("greet", &World::greet)
```

在原來示例中的成員訪問形式——『點』(dot) 沒什麼特別的：C++ 允許在一個表達式的任何一邊寫下任何數量的空白，把『點』放在每行代碼的開始允許我們連續的調用成員函數，因為我們喜歡統一形式的語法。另外一個關鍵的、允許實現鏈式語法的事實是 `class_<>` 成員函數都返回一個到 `*this` 的引用 (reference)。

所以這個示例等於：

```
class_<World> w("World");
w.def("greet", &World::greet);
w.def("set", &World::set);
```

這種形式偶爾是有用的，以便用這種方式分解 Boost.Python 的類封裝，但是文章剩下的部分將會使用簡潔的語法。

這裡是封裝類的使用：

```
>>> import hello
>>> planet = hello.World()
>>> planet.set('howdy')
>>> planet.greet()
'howdy'
```

構造函數 (Constructors)

由於我們的 `World` 類只是一個 `struct`，它有一個隱式的無參數 (空的) 的構造函數。Boost.Python 缺省的會公開這個構造函數 (給 Python)，所以我們可以這樣寫：

```
>>> planet = hello.World()
```

然而，在任何語言裡面的良好設計的類都會需要構造函數參數——用於建立他們的不變量 (invariants)。不像 Python，她的 `__init__` 只是一個特定命名的方法，在 C++ 裡構造函數不能像普通成員函數那樣被掌控。特別地，我們不能取他們的地址：`&World::World` 是一個錯誤。

(Boost.Python)庫提供了一種不同的接口以指定構造函數，像這樣：

```
struct World
{
    World(std::string msg); // added constructor
    ...
```

我們可以像下面這樣更改我們的封裝代碼：

```
class_<World>("World", init<std::string>())
    ...
```

當然，一個C++類可以有額外的構造函數，而且我們可以通過更多的`def() init<...>`實例把它們導出：

```
class_<World>("World", init<std::string>())
    .def(init<double, double>())
    ...
```

Boost.Python 允許封裝的函數、成員函數和構造函數被重載以反映他們在C++中的重載關係(to be overloaded to mirror C++ overloading).

數據成員和屬性(Data Members and Properties)

任何 public 的C++數據成員都可以容易地導出成`readonly`或`readwrite`屬性：

```
class_<World>("World", init<std::string>())
    .def_readonly("msg", &World::msg)
    ...
```

也可以直接在 Python 內部使用：

```
>>> planet = hello.World('howdy')
>>> planet.msg
'howdy'
```

這不會造成在`World`實例內增加一個`__dict__`的結果，這麼做可以在封裝大型數據結構時節省內存。實際上，根本沒有`__dict__`實例被創造除非顯式地在 Python 裡面給它增加屬性。Boost.Python 把這種能力感激於 Python 2.2 的類型系統，特別是描述符接口(descriptor interface)和`property`類型。

在C++裡，具有 public 屬性的數據成員被認為是一種糟糕的設計，因為它們破壞了封裝性，並且風格指導通常指示使用"getter"和"setter"函數作為代替。在 Python 裡，對應`__getattr__`和`__setattr__`，從 2.2 開始`property`意味著屬性訪問是一個程序員可用的，封裝性更好的語法工具。Boost.Python 通過使 Python 的`property`直接被創建並且對用戶可用，彌合了這種語言習慣上的縫隙。即使`msg`是 private 的，我們還是可以把它作為屬性(attribute)給 Python 使用，通過：

```
class_<World>("World", init<std::string>())
```

```
    .add_property("msg", &World::greet, &World::set)
    ...
```

上面的示例和 Python 2.2+內使用 properties 的用法是一樣的：

```
>>> class World(object):
...     __init__(self, msg):
...         self.__msg = msg
...     def greet(self):
...         return self.__msg
...     def set(self, msg):
...         self.__msg = msg
...     msg = property(greet, set)
```

操作符重載(Operator Overloading)

具有編寫針對用戶定義數據類型的算術操作符(arithmetic operators)的能力已經成為一個數學計算語言主要的成功因素，像 NumPy 這樣成功的包證明了在擴展模塊裡導出操作符的威力。Boost.Python 提供了一種很簡單的機制以實現封裝操作符重載。下面的例子是一個 Boost 有理數庫封裝內部的代碼片斷：

```
class_<rational<int> >("rational_int")
  .def(init<int, int>()) // constructor, e.g. rational_int(3,4)
  .def("numerator", &rational<int>::numerator)
  .def("denominator", &rational<int>::denominator)
  .def(-self)        // __neg__ (unary minus)
  .def(self + self)  // __add__ (homogeneous)
  .def(self * self)  // __mul__
  .def(self + int()) // __add__ (heterogenous)
  .def(int() + self) // __radd__
  ...
```

這裡的魔法是應用一種簡化的表達式模板("expression templates") [VELD1995]，一種原來用於開發 high-performance matrix algebra expressions 的技術。它的本質是利用重載的操作符構造一個類型以表示計算，而不是立即進行計算工作。In matrix algebra, dramatic optimizations are often available when the structure of an entire expression can be taken into account, rather than evaluating each operation "greedily". Boost.Python 使用同樣的技術構建一個適當的 Python 方法對象，這基於在表達式內包含 `self`。

繼承(Inheritance)

C++繼承關係可以用 Boost.Python 描述，通過添加一個可選的 `bases<...>` 參數到 `class_<...>` 模板參數列表，像下面這樣：

```
class_<Derived, bases<Base1,Base2> >("Derived")
    ...
```

這有兩種作用：

1. 當類 `class_<...>` 被創建的時候，Boost.Python 在註冊項(registry)裡面查找 `Base1` 和 `Base2` 對應的 Python 類型對象，並且把他們作為新的 Python `Derived` 類型對象的基類，所以 `Base1` 和 `Base2` 類型的方法自動成為 `Derived` 類型的成員。因為註冊項(registry)是全局的，所以即使 `Derived` 是(和 Base1/Base2)在不同的模塊裡頭也有作用。
2. 從 `Derived` 到它的基類的轉換也被添加的 Boost.Python 的註冊項裡。因而可以在每個包含了 `Derived` 實例的對象內部，調用 封裝的 C++ 方法所需要(指向或引用到)的每個基類類型。`class T` 的被封裝的成員函數被看作有一個隱式的第一個參數 `T&`，所以那些用以允許基類方法被派生類調用的轉換是必要的。

當然從封裝的 C++ 類實例派生出新的 Python 對象也是可能的。因為 Boost.Python 使用 new-style class system，他們和 Python 內建類型的工作方式很像。有一個重大的細節上的不同之處：內建類型通常通過 `__new__` 函數建立他們自己的不變量(invariants)，所以派生類在使用(基類的)方法前不需要調用基類的 `__init__` ：

```
>>> class L(list):
...       def __init__(self):
...           pass
...
>>> L().reverse()
>>>
```

因為 C++ 的對象構造是一個單步操作，C++ 不能構造(對象)實例數據直到參數可用。在 `__init__` 函數裡：

```
>>> class D(SomeBoostPythonClass):
...       def __init__(self):
...           pass
...
>>> D().some_boost_python_method()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: bad argument type for built-in operation
```

它會出錯，因為在 D 實例內部 Boost.Python 找不到 `SomeBoostPythonClass` 的實例數據；`D` 的 `__init__` 函數遮蔽了基類的構造。移除 `D` 的 `__init__` 函數或者在 `SomeBoostPythonClass.__init__(...)`內部顯式的調用它都是正確的。

虛函數(irtual Functions)

在 Python 裡面， 從擴展類型派生出新的類型不是有趣的事情，除非他們能從 C++ 被多態地使用。換句話說，Python 方法的實現應當看上去是重載 C++ 虛函數的實現，當從 C++ 通過基類的指針/引用調用(這個虛函數)的時候。因為唯一的改變一個虛函數行為的辦法是在派生類內重載它，用戶必須創建一個特殊的派生類以**轉發**(dispatch)**實現**這種虛函數的多態性：

```
//
// interface to wrap:
//
class Base
{
 public:
    virtual int f(std::string x) { return 42; }
    virtual ~Base();
};

int calls_f(Base const& b, std::string x) { return b.f(x); }

//
// Wrapping Code
//

// Dispatcher class
struct BaseWrap : Base
{
    // Store a pointer to the Python object
    BaseWrap(PyObject* self_) : self(self_) {}
    PyObject* self;

    // Default implementation, for when f is not overridden
    int f_default(std::string x) { return this->Base::f(x); }
    // Dispatch implementation
    int f(std::string x) { return call_method<int>(self, "f", x); }
};


...
    def("calls_f", calls_f);
    class_<Base, BaseWrap>("Base")
        .def("f", &Base::f, &BaseWrap::f_default)
        ;
```

現在，這裡是一些 Python 演示代碼：

```
>>> class Derived(Base):
...     def f(self, s):
...          return len(s)
...
>>> calls_f(Base(), 'foo')
42
>>> calls_f(Derived(), 'forty-two')
9
```

關於**轉發類**需要注意：

- The key element which allows overriding in Python is the `call_method` invocation, which uses the same global type conversion registry as the C++ function wrapping does to convert its arguments from C++ to Python and its return type from Python to C++.
- Any constructor signatures you wish to wrap must be replicated with an initial `PyObject*` argument
- The dispatcher must store this argument so that it can be used to invoke `call_method`
- The `f_default` member function is needed when the function being exposed is not pure virtual; there's no other way `Base::f` can be called on an object of type `BaseWrap`, since it overrides `f`.

Deeper Reflection on the Horizon?

Admittedly, this formula is tedious to repeat, especially on a project with many polymorphic classes. That it is neccessary reflects some limitations in C++'s compile-time introspection capabilities: there's no way to enumerate the members of a class and find out which are virtual functions. At least one very promising project has been started to write a front-end which can generate these dispatchers (and other wrapping code) automatically from C++ headers.

Pyste is being developed by Bruno da Silva de Oliveira. It builds on GCC_XML, which generates an XML version of GCC's internal program representation. Since GCC is a highly-conformant C++ compiler, this ensures correct handling of the most-sophisticated template code and full access to the underlying type system. In keeping with the Boost.Python philosophy, a Pyste interface description is neither intrusive on the code being wrapped, nor expressed in some unfamiliar language: instead it is a 100% pure Python script. If Pyste is successful it will mark a move away from wrapping everything directly in C++ for many of our users. It will also allow us the choice to shift some of the metaprogram code from C++ to Python. We expect that soon, not only our users but the Boost.Python developers themselves will be "thinking hybrid" about their own code.

序列化(Serialization)

序列化的的含義是把內存中的對象轉換成能夠存儲到磁盤或者通過網絡連接發送的形式。序列化後生成的對象(大多數時候是一種字符串)能被重新轉化到原始對象。一個好的序列化系統會自動的轉化整個對象體系。Python 的`pickle` 模塊就是這樣一個系統。它得益於這種語言強大的運行期內部處理(譯註:instrospection)能力，幾乎能序列化任意用戶定義的對象。只需要通過加入一些簡單 的、非打擾式的處理，這種強大機制就能夠擴展到為封裝的 C++對象工作。下面是一個例子:

```
#include <string>

struct World
{
```

```
    World(std::string a_msg) : msg(a_msg) {}
    std::string greet() const { return msg; }
    std::string msg;
};

#include <boost/python.hpp>
using namespace boost::python;

struct World_picklers : pickle_suite
{
  static tuple
  getinitargs(World const& w) { return make_tuple(w.greet()); }
};

BOOST_PYTHON_MODULE(hello)
{
    class_<World>("World", init<std::string>())
        .def("greet", &World::greet)
        .def_pickle(World_picklers())
    ;
}
```

現在，我們創建一個`World`對象並且把它放在磁盤上休息：

```
>>> import hello
>>> import pickle
>>> a_world = hello.World("howdy")
>>> pickle.dump(a_world, open("my_world", "w"))
```

然後，可能是在不同的計算機上不同的操作系統的不同的一個腳本上，我們這樣用：

```
>>> import pickle
>>> resurrected_world = pickle.load(open("my_world", "r"))
>>> resurrected_world.greet()
'howdy'
```

當然，使用`cPickle`（譯註：cPickle是更高效率的一種pickle實現）模塊可以更快速的處理。

Boost.Python的`pickle_suite`完全支持標準Python文檔定義的`pickle`協議。像Python的__getinitargs__函數那樣，pickle_suite的getinitargs()函數負責創建argument tuple用以重建pickle過的對象。Python pickling協議的其他元素，__getstate__ and __setstate__可以通過C++ getstate和setstate函數選擇提供。C++的靜態類型系統允許庫確保在編譯期避免無意義的函數合併（例如：getstate卻沒有setstate）被應用。

使更複雜的C++對象能夠被序列化要比上面的示例需要更多的工作。幸運的是`object`接口（查看下一部分）在代碼可管理性上非常地有幫助。

Object 接口(Object interface)

有經驗的C語言擴展模塊接口作者應該很熟悉`PyObject*`，手動引用計數(reference-counting)，而且需要記住哪個API返回"新的"（擁有的）引用或者"借來的"（raw）引用。這些限制不僅僅是很麻煩，重要的這也是主要的錯誤源，特別是在異常的表示(presence of exceptions)上。

Boost.Python提供了一個`object`類，能夠自動進行引用計數並且提供從任意C++對象到Python對象的轉換。這對於想成為擴展模塊作者的人來說，極大的減少了學習困難。

從任何其他類型創建一個`object`是非常簡單的：

```
object s("hello, world");  // s manages a Python string
```

`object`可以和所有其他數據類型進行模板化的交互(templated interactions)，並且能夠自動完成到python的轉換。這些都進行得非常自然以至於它很容易被忽略掉：

```
object ten_Os = 10 * s[4]; // -> "oooooooooo"
```

在上面的示例裡，`4`和`10`在進行索引操作和乘法操作調用(indexing and multiplication operations)前，被轉化為Python對象。

`extract<T>` class 模板能夠用來轉換Python對象到C++類型：

```
double x = extract<double>(o);
```

如果任何一側的轉換不能進行，一個適當的exception將會在運行期被拋出。

`object`類型與Python內建類型的『副本』如：`list`，`dict`，`tuple`等等成為一套。這使得從C++轉換到這些高階類型變得方便操作：

```
dict d;
d["some"] = "thing";
d["lucky_number"] = 13;
list l = d.keys();
```

它的工作方式和看上去的樣子幾乎和一般的Python代碼一樣，但是它是純C++的。當然我們可以封裝接受或者返回`object`實例的C++函數。

混合地思考(Thinking hybrid)

由於在組合不同的編程語言時實際上的和心理上的困難，通常在開始先確定單獨的一種語言。對於任何應用程序來說，性能上的考慮決定了在核心算法上使用編譯語言(compiled language)。不幸的是，由於靜態類型系統的複雜性，我們為運行期性能所付出的代價通常在開發期極大的增加。經驗顯示：相對於開發同等的Python代碼來說，寫出可維護的C++代碼通常需要更長時間和更多努力工作得來的經驗。即使當開發者們用編譯語言(compiled language)感覺很舒服的時候，他們也常常為他們的系統增加某種類型的腳本層，因為他們的用戶可以獲得同樣的使用腳本語言的好處。

Boost.Python 讓我們可以*混合地思考*。Python可以作為一些應用程序的快速原型；她的易用性和巨大的標準庫給了我們到一個工作中的系統的一個開始。如果有必要，這些工作代碼可以用來揭示熱點比

率(譯註：意思是發現哪些代碼運行最頻繁或者佔用時間/資源最多)。為了最大化提高性能，那些(熱點)可以被C++重新實現，然後用Boost.Python把他們綁定到現有的高階過程(higher-level procedure)中。

當然，自上而下的過程不是那麼吸引人，如果從開始就有許多代碼不得不改成用C++實現。幸運的是Boost.Python允許我們應用自下而上的過程。我們曾經應用這種過程非常成功地開發了一個科學軟件的工具箱。這個工具箱的開始的時候主要是一個帶有Boost.Python綁定的C++類，過了一段時間，成長的部分主要集中在C++的部分。然而由於這個工具箱越來越複雜，越來越多的新特性可以在Python內被實現。

This figure shows the estimated ratio of newly added C++ and Python code over time as new algorithms are implemented. We expect this ratio to level out near 70% Python. Being able to solve new problems mostly in Python rather than a more difficult statically typed language is the return on our investment in Boost.Python. The ability to access all of our code from Python allows a broader group of developers to use it in the rapid development of new applications.

開發歷史(Development history)

The first version of Boost.Python was developed in 2000 by Dave Abrahams at Dragon Systems, where he was privileged to have Tim Peters as a guide to "The Zen of Python". One of Dave's jobs was to develop a Python-based natural language processing system. Since it was eventually going to be targeting embedded hardware, it was always assumed that the compute-intensive core would be rewritten in C++ to optimize speed and memory footprint [1]. The project also wanted to test all of its C++ code using Python test scripts [2]. The only tool we knew of for binding C++ and Python was SWIG, and at the time its handling of C++ was weak. It would be false to claim any deep insight into the possible advantages of Boost.Python's approach at this point. Dave's interest and expertise in fancy C++ template tricks had just reached the point where he could do some real damage, and Boost.Python emerged as it did because it filled a need and because it seemed like a cool thing to try.

This early version was aimed at many of the same basic goals we've described in this paper, differing most-noticeably by having a slightly more cumbersome syntax and by lack of special support for operator overloading, pickling, and component-based development. These last three features were quickly added by Ullrich Koethe and Ralf Grosse-Kunstleve [3], and other enthusiastic contributors arrived on the scene to contribute enhancements like support for nested modules and static member functions.

By early 2001 development had stabilized and few new features were being added, however a disturbing new fact came to light: Ralf had begun testing Boost.Python

on pre-release versions of a compiler using the EDG front-end, and the mechanism at the core of Boost.Python responsible for handling conversions between Python and C++ types was failing to compile. As it turned out, we had been exploiting a very common bug in the implementation of all the C++ compilers we had tested. We knew that as C++ compilers rapidly became more standards-compliant, the library would begin failing on more platforms. Unfortunately, because the mechanism was so central to the functioning of the library, fixing the problem looked very difficult.

Fortunately, later that year Lawrence Berkeley and later Lawrence Livermore National labs contracted with Boost Consulting for support and development of Boost.Python, and there was a new opportunity to address fundamental issues and ensure a future for the library. A redesign effort began with the low level type conversion architecture, building in standards-compliance and support for component-based development (in contrast to version 1 where conversions had to be explicitly imported and exported across module boundaries). A new analysis of the relationship between the Python and C++ objects was done, resulting in more intuitive handling for C++ lvalues and rvalues.

The emergence of a powerful new type system in Python 2.2 made the choice of whether to maintain compatibility with Python 1.5.2 easy: the opportunity to throw away a great deal of elaborate code for emulating classic Python classes alone was too good to pass up. In addition, Python iterators and descriptors provided crucial and elegant tools for representing similar C++ constructs. The development of the generalized `object` interface allowed us to further shield C++ programmers from the dangers and syntactic burdens of the Python 'C' API. A great number of other features including C++ exception translation, improved support for overloaded functions, and most significantly, CallPolicies for handling pointers and references, were added during this period.

In October 2002, version 2 of Boost.Python was released. Development since then has concentrated on improved support for C++ runtime polymorphism and smart pointers. Peter Dimov's ingenious`boost::shared_ptr` design in particular has allowed us to give the hybrid developer a consistent interface for moving objects back and forth across the language barrier without loss of information. At first, we were concerned that the sophistication and complexity of the Boost.Python v2 implementation might discourage contributors, but the emergence of Pyste and several other significant feature contributions have laid those fears to rest. Daily questions on the Python C++-sig and a backlog of desired improvements show that the library is getting used. To us, the future looks bright.

總結(Conclusions)

Boost.Python achieves seamless interoperability between two rich and

complimentary language environments. Because it leverages template metaprogramming to introspect about types and functions, the user never has to learn a third syntax: the interface definitions are written in concise and maintainable C++. Also, the wrapping system doesn't have to parse C++ headers or represent the type system: the compiler does that work for us.

Computationally intensive tasks play to the strengths of C++ and are often impossible to implement efficiently in pure Python, while jobs like serialization that are trivial in Python can be very difficult in pure C++. Given the luxury of building a hybrid software system from the ground up, we can approach design with new confidence and power.

引用(Citations)

[VELD1995] T. Veldhuizen, "Expression Templates," C++ Report, Vol. 7 No. 5 June 1995, pp. 26-31. http://osl.iu.edu/~tveldhui/papers/Expression-Templates/exprtmpl.html

脚註(Footnotes)

[1] In retrospect, it seems that "thinking hybrid" from the ground up might have been better for the NLP system: the natural component boundaries defined by the pure python prototype turned out to be inappropriate for getting the desired performance and memory footprint out of the C++ core, which eventually caused some redesign overhead on the Python side when the core was moved to C++.

[2] We also have some reservations about driving all C++ testing through a Python interface, unless that's the only way it will be ultimately used. Any transition across language boundaries with such different object models can inevitably mask bugs.

[3] These features were expressed very differently in v1 of Boost.Python