Extensões C para Python

Leonardo de Albernaz Ferreira

10 de Junho de 2012

1 Introdução

Atualmente contamos com várias linguagens de programação de alto nível, com uma sintaxe fácil, agradável e métodos de desenvolvimento rápido. Um exemplo disso é o Python, uma linguagem de tipagem dinâmica que conta com várias bibliotecas já estabelecidas para solução de problemas numéricos e científicos. Pela linguagem ter um alto nível abstração ela acaba com menos performance de execução do que uma linguagem de baixo nível como o C. Mesmo tendo muitas rotinas otimizadas e com uma performance comparável ao do C podemos nos deparar com alguns problemas que só podem ser resolvidos com paralelismo e computação distribuída. É aí que entra a integração do Python com C, basicamente podemos desenvolver código python que interage com rotinas C. Dessa forma podemos utilizar de outras tecnologias relativas a computação de alto desempenho no escopo do C como OpenMP e até CUDA. Construímos uma extensão para o manual e iremos utiliza-la para exemplificar o processo de desenvolvimento desse tipo de módulo.

2 O Módulo Example

A extensão de exemplo utiliza a função de potência de um numero pelo outro pow(x,y) do C para ilustrar o processo de criação, para discutirmos sua estrutura e os passos necessários para usa-lo. O diferencial presente nas extensões é o tráfego de dados entre as linguagens e para isso utilizamos alguns métodos disponibilizados no header Python.h em nossas funções. Após o desenvolvimento da nossa extensão teremos que construir e compilar o código já que este se trata de código C. O processo de compilação e construção é efetuado através de um script Python que fica responsável pelas chamadas ao compilador e outras configuradões necessárias.

2.1 Estrutura

A estrutura de um módulo C é composta de 4 partes essenciais: O include do header Python.h que contem todas as funções e informações necessárias para efetuar a integração com o python, uma função de inicialização do módulo, um array usualmente chamado de "Tabela de Métodos"contendo as informações das funções da extensão e finalmente as funções propriamente ditas. A extensão que desenvolvemos para o presente documento é a seguinte:

```
1  #include <Python.h>
2
3  static PyMethodDef ExampleMethods[] = {
4      {"pow\_c",  pow_c, METH_VARARGS, "Descri   o"},
5      {NULL, NULL, 0, NULL}          /* Sentinel */
6  };
7
8  PyMODINIT_FUNC initexamplemodule(void) {
9      (void) Py_InitModule("examplemodule", ExampleMethods);
10 }
11
12
13 static PyObject *pow_c(PyObject *self, PyObject *args) {
14   float a;
15   float b;
16   float resultado;
17
18   PyArg_ParseTuple(args, "ff", &a, &b);
19
20   resultado = pow(a,b);
21
22
23   return Py_BuildValue("f", resultado);
24 }
```

Nosso objetivo é utilizar essa extensão como qualquer outro módulo Python, através de imports e chamadas aos métodos. Vejamos como usaremos nossa extensão:

```
1 >>> from examplemodule import *
2 >>> pow_c(10,2)
3 100.0
```

2.1.1 Inicialização

Toda extensão C feita para o python possui um método de inicialização. Quando importamos alguma extensão no python é necessário que ela se inicialize, montando as informações das funções que estão disponíveis para o usuário. Para criarmos esse método de inicialização seguimos o padrão de nome que o python usa para a chamada automática, esse padrão é initnome(), em que nome é o nome da extensão. A única instrução de código que deverá haver neste método é o uso da função contida no nosso header do Python, Py_InitModule. Esta função recebe como parâmetros o nome da extensão e um array "Tabela de Métodos"com as informações de cada função da nossa extensão, assim ao inicializar o módulo, o python saberá quais os métodos que ele contem.

2.1.2 Tabela de Funções

Como falamos na inicialização, a chamada para a função Py_InitModule precisa de um array contendo as informações das nossas funções. Esta tabela é basicamente um array bi-dimensional da forma abaixo:

```
1 static PyMethodDef ExampleMethods[] = {
2     {"pow_c",  pow_c, METH_VARARGS, "Descri  o"},
3     {NULL, NULL, 0, NULL}          /* Sentinel */
4 };
```

Como vemos cada linha possui 4 posições. A primeira contem como será o nome que desejamos chamar no Python para a função no C, a segunda contem o nome da função no C, a terceira é uma flag que fala para o interpretador que convenção de chamada usar para a função C, geralmente é "METH_VARARGS"e a quarta uma descrição da função a título de documentação. Geralmente o nome na primeira posição é exatamente igual ao nome da função do C na segunda, mas isso não é necessariamente obrigatório. Poderíamos ter o seguinte caso:

```
1 static PyMethodDef ExampleMethods[] = {
2     {"potencia",  pow\_c, METH_VARARGS, "Descri  o"},
3     {NULL, NULL, 0, NULL}          /* Sentinel */
4 };
```

Assim chamaríamos no Python nossa função da seguinte maneira:

```
1 >>> from examplemodule import *
2 >>> resultado = potencia(10,2)
3 >>> resultado
4 100.0
```

A nossa tabela de dados referentes as funções tem outra convenção, que é uma uma linha no final com as posições nulas.

2.1.3 A Função pow_c(a,b)

A parte mais importante do nosso módulo de exemplo é a função pow_c, ela é o objetivo da construção da extensão. Suponhamos que a função math.pow(x,y)

do python é muito lenta em comparação com a função pow(x,y) do C. Seria útil termos uma forma de utilizar o pow do C do contexto do Python caso tivéssemos algum código que necessitasse massivamente da função pow. Então decidimos que é necessário criar uma extensão para fazer esta integração, mas há alguns cuidados que devemos tomar: os dados no python estão armazenados de uma maneira e no C de outra. É por isso que em nossa função pow_c(a,b) chamamos o método "PyArg_ParseTuple(arg, "ff", a, b)"ele é responsável por fazer a ponte entre os tipos de dados do Python e os do C.

2.1.4 PyArg_ParseTuple

Esta função possui três argumentos: Variável arg que vem do python contendo todos os dados e variáveis passadas como parâmetro, uma string contendo os tipos dessas variáveis através de uma convensão[1] e uma lista de endereços das variáveis declaradas no C deve ser passada. No caso do exemplo temos a chamada PyArg_ParseTuple(arg, "ff", a, b), onde passamos a variável "arg"advinda do python, a String "ff"que nos indica que existem duas variáveis do tipo "Float"na variável arg e os endereços das variáveis de destino a e b. É possível adaptar diversos tipos de variáveis e até objetos complexos. Vejamos uma chamada mais complexa:

```
1 int a;
2 int b;
3 float c;
4 Object object;
5 PyArg(arg, "iifO", $a, $b, $c, $object)
```

Neste caso sabemos através da String que temos dois ints, um float e um objeto. Os objetos são armazenados de uma forma mais complexa e precisam de tratamento para evitar problemas nos dados, não estaremos falando sobre este aspecto aqui.

Após termos nossas variáveis no C preenchidas com os valores vindos do Python podemos então efetuar as manipulações no C de forma padrão. É necessário um cuidado ao usar bibliotecas não nativas do C, pois elas deverão ser linkadas no processo de empacotamento da extensão.

2.1.5 Py_BuildValue

Agora que já traduzimos os dados e efetuamos as transformações com o C temos que enviar os dados para o Python. Para isso usamos o método Py_BuildValue, ele funciona de uma forma inversa ao PyArg_ParseTuple, ele agora vai pegar nossas variáveis do C e monta-las de forma que o Python consiga interpretar. Ele recebe a string com a mesma convenção do PyArg_ParseTuple e as variáveis que irão para o Python. Vejamos nossa função pow_c:

```
1 static PyObject *pow_c(PyObject *self, PyObject *args) {
```

[1] http://docs.python.org/c-api/arg.html

```
2     float a;
3     float b;
4     float resultado;
5
6     PyArg_ParseTuple(args, "ff", &a, &b);
7
8     resultado = pow(a,b);
9
10
11    return Py_BuildValue("f", resultado);
12  }
```

Pegamos as variáveis a e b do python e aplicamos nela o pow, geramos então uma terceira variável com o resultado. Este resultado queremos devolver para o python, para que prossigamos o utilizando no nosso código:

```
1 >>> from examplemodule import *
2 >>> resultado = pow_c(10,2)
3 >>> resultado
4 100.0
```

Agora fizemos tudo que é necessário fazer para desenvolvermos nosso código. Podemos prosseguir para o processo de construção e compilação da nossa extensão.

3 Construindo e Compilando[2]

Agora que temos nosso exemplo desenvolvido temos que construir e compilar a extensão para que possamos importa-lo como qualquer outra extensão. Para isso utilizamos um script python que se encarrega de chamar o compilador, linkar as bibliotecas externas e preparar nossa extensão para ser importada no Python. Esse script por convenção geralmente se chama "setup.py"e contem o seguinte formato para nosso exemplo:

```
1 from distutils.core import setup, Extension
2
3 module1 = Extension('examplemodule', sources = ['examplemodule.c'])
4
5 setup (name = 'Example',
6        version = '0.01',
7        description = 'Example',
8        ext_modules = [module1])
```

O Script basicamente importa os módulos nativos do Python que se encarregam de fazer este serviço, nomeia nosso módulo e suas informações na linha 3 explicitando quais os arquivos-fontes que o contemplam e faz a chamada da função que se encarrega do resto.

3.1 Utilizando bibliotecas C na extensão

Quando utilizamos uma outra biblioteca no C que não seja o Python.h precisamos linka-la junto no processo de construção da extensão, é necessário que enviemos a biblioteca junto do nosso módulo para quando nós chamarmos as funções no Python não ocorra problemas de referencia. Vejamos um exemplo de um módulo que utiliza OpenMP e como ficaria seu Setup.py:

```
1 from distutils.core import setup, Extension
2 module1 = Extension('orbitStatistics',
3                      sources = ['calculations.cpp'],
                           extra_compile_args=['-fopenmp'],
4       extra_link_args=['-lgomp'])
5
6 setup (name = 'Orbit Statistics C Api',
7        version = '1.0',
8        description = 'This is a alpha package',
9        ext_modules = [module1])
```

Vemos que agora temos dois parâmetros adicionais na definição da extensão: extra_compile_args=['-fopenmp'] e extra_link_args=['-lgomp']. O Primeiro diz para o compilador C que o código está utilizando a biblioteca do OpenMP e a segunda esta dizendo para linkar a biblioteca junto no processo de compilação e construção.

4 Integrando Módulo ao Python

O Script irá gerar um pacote no formato .so e um arquivo no formato .o para o código fonte. A extensão estará pronta para ser importado como qualquer outra biblioteca do Python.

5 Referências

1. http://docs.python.org/extending/
2. http://docs.python.org/extending/building.html
3. http://starship.python.net/crew/mwh/toext/