15.6 从C语言中调用Python代码¶

问题¶

你想在C中安全的执行某个Python调用并返回结果给C。 例如，你想在C语言中使用某个Python函数作为一个回调。

解决方案¶

在C语言中调用Python非常简单，不过设计到一些小窍门。 下面的C代码告诉你怎样安全的调用：

```
#include <Python.h>

/* Execute func(x,y) in the Python interpreter.  The
   arguments and return result of the function must
   be Python floats */

double call_func(PyObject *func, double x, double y) {
  PyObject *args;
  PyObject *kwargs;
  PyObject *result = 0;
  double retval;

  /* Make sure we own the GIL */
  PyGILState_STATE state = PyGILState_Ensure();

  /* Verify that func is a proper callable */
  if (!PyCallable_Check(func)) {
    fprintf(stderr,"call_func: expected a callable\n");
    goto fail;
  }
  /* Build arguments */
  args = Py_BuildValue("(dd)", x, y);
  kwargs = NULL;

  /* Call the function */
  result = PyObject_Call(func, args, kwargs);
  Py_DECREF(args);
  Py_XDECREF(kwargs);

  /* Check for Python exceptions (if any) */
  if (PyErr_Occurred()) {
    PyErr_Print();
    goto fail;
  }

  /* Verify the result is a float object */
  if (!PyFloat_Check(result)) {
    fprintf(stderr,"call_func: callable didn't return a float\n");
    goto fail;
  }

  /* Create the return value */
  retval = PyFloat_AsDouble(result);
  Py_DECREF(result);

  /* Restore previous GIL state and return */
  PyGILState_Release(state);
```

```
  return retval;

fail:
  Py_XDECREF(result);
  PyGILState_Release(state);
  abort();   // Change to something more appropriate
}
```

要使用这个函数，你需要获取传递过来的某个已存在Python调用的引用。 有很多种方法可以让你这样做， 比如将一个可调用对象传给一个扩展模块或直接写C代码从已存在模块中提取出来。

下面是一个简单例子用来掩饰从一个嵌入的Python解释器中调用一个函数：

```
#include <Python.h>

/* Definition of call_func() same as above */
...

/* Load a symbol from a module */
PyObject *import_name(const char *modname, const char *symbol) {
  PyObject *u_name, *module;
  u_name = PyUnicode_FromString(modname);
  module = PyImport_Import(u_name);
  Py_DECREF(u_name);
  return PyObject_GetAttrString(module, symbol);
}

/* Simple embedding example */
int main() {
  PyObject *pow_func;
  double x;

  Py_Initialize();
  /* Get a reference to the math.pow function */
  pow_func = import_name("math","pow");

  /* Call it using our call_func() code */
  for (x = 0.0; x < 10.0; x += 0.1) {
    printf("%0.2f %0.2f\n", x, call_func(pow_func,x,2.0));
  }
  /* Done */
  Py_DECREF(pow_func);
  Py_Finalize();
  return 0;
}
```

要构建例子代码，你需要编译C并将它链接到Python解释器。 下面的Makefile可以教你怎样做（不过在你机器上面需要一些配置）。

```
all::
        cc -g embed.c -I/usr/local/include/python3.3m \
          -L/usr/local/lib/python3.3/config-3.3m -lpython3.3m
```

编译并运行会产生类似下面的输出：

```
0.00 0.00
0.10 0.01
```

```
0.20 0.04
0.30 0.09
0.40 0.16
...
```

下面是一个稍微不同的例子，展示了一个扩展函数， 它接受一个可调用对象和其他参数，并将它们传递给 `call_func()` 来做测试：

```
/* Extension function for testing the C-Python callback */
PyObject *py_call_func(PyObject *self, PyObject *args) {
  PyObject *func;

  double x, y, result;
  if (!PyArg_ParseTuple(args,"Odd", &func,&x,&y)) {
    return NULL;
  }
  result = call_func(func, x, y);
  return Py_BuildValue("d", result);
}
```

使用这个扩展函数，你要像下面这样测试它：

```
>>> import sample
>>> def add(x,y):
...     return x+y
...
>>> sample.call_func(add,3,4)
7.0
>>>
```

讨论¶

如果你在C语言中调用Python，要记住最重要的是C语言会是主体。 也就是说，C语言负责构造参数、调用Python函数、检查异常、检查类型、提取返回值等。

作为第一步，你必须先有一个表示你将要调用的Python可调用对象。 这可以是一个函数、类、方法、内置方法或其他任意实现了 `__call__()` 操作的东西。 为了确保是可调用的，可以像下面的代码这样利用 `PyCallable_Check()` 做检查：

```
double call_func(PyObject *func, double x, double y) {
  ...
  /* Verify that func is a proper callable */
  if (!PyCallable_Check(func)) {
    fprintf(stderr,"call_func: expected a callable\n");
    goto fail;
  }
  ...
```

在C代码里处理错误你需要格外的小心。一般来讲，你不能仅仅抛出一个Python异常。 错误应该使用C代码方式来被处理。在这里，我们打算将对错误的控制传给一个叫 `abort()` 的错误处理器。 它会结束掉整个程序，在真实环境下面你应该要处理的更加优雅些（返回一个状态码）。 你要记住的是在这里C是主角，因此并没有跟抛出异常相对应的操作。 错误处理是你在编程时必须要考虑的事情。

调用一个函数相对来讲很简单——只需要使用 `PyObject_Call()` ， 传一个可调用对象给它、一个参数元组和一个可选的关键字字典。 要构建参数元组或字典，你可以使用 `Py_BuildValue()` ,如下：

```
double call_func(PyObject *func, double x, double y) {
  PyObject *args;
  PyObject *kwargs;

  ...
  /* Build arguments */
  args = Py_BuildValue("(dd)", x, y);
  kwargs = NULL;

  /* Call the function */
  result = PyObject_Call(func, args, kwargs);
  Py_DECREF(args);
  Py_XDECREF(kwargs);
  ...
```

如果没有关键字参数，你可以传递NULL。当你要调用函数时， 需要确保使用了 `Py_DECREF()` 或者 `Py_XDECREF()` 清理参数。 第二个函数相对安全点，因为它允许传递NULL指针（直接忽略它）， 这也是为什么我们使用它来清理可选的关键字参数。

调用万Python函数之后，你必须检查是否有异常发生。 `PyErr_Occurred()` 函数可被用来做这件事。 对对于异常的处理就有点麻烦了，由于是用C语言写的，你没有像Python那么的异常机制。 因此，你必须要设置一个异常状态码，打印异常信息或其他相应处理。 在这里，我们选择了简单的 `abort()` 来处理。另外，传统C程序员可能会直接让程序奔溃。

```
...
/* Check for Python exceptions (if any) */
if (PyErr_Occurred()) {
  PyErr_Print();
  goto fail;
}
...
fail:
  PyGILState_Release(state);
  abort();
```

从调用Python函数的返回值中提取信息通常要进行类型检查和提取值。 要这样做的话，你必须使用Python对象层中的函数。 在这里我们使用了 `PyFloat_Check()` 和 `PyFloat_AsDouble()` 来检查和提取Python浮点数。

最后一个问题是对于Python全局锁的管理。 在C语言中访问Python的时候，你需要确保GIL被正确的获取和释放了。 不然的话，可能会导致解释器返回错误数据或者直接奔溃。 调用 `PyGILState_Ensure()` 和 `PyGILState_Release()` 可以确保一切都能正常。

```
double call_func(PyObject *func, double x, double y) {
  ...
  double retval;

  /* Make sure we own the GIL */
  PyGILState_STATE state = PyGILState_Ensure();
  ...
  /* Code that uses Python C API functions */
  ...
  /* Restore previous GIL state and return */
  PyGILState_Release(state);
  return retval;

fail:
```

```
 PyGILState_Release(state);
 abort();
}
```

一旦返回， `PyGILState_Ensure()` 可以确保调用线程独占Python解释器。 就算C代码运行于另外一个解释器不知道的线程也没事。 这时候，C代码可以自由的使用任何它想要的Python C-API 函数。 调用成功后，PyGILState_Release()被用来讲解释器恢复到原始状态。

要注意的是每一个 `PyGILState_Ensure()` 调用必须跟着一个匹配的 `PyGILState_Release()` 调用——即便有错误发生。 在这里，我们使用一个 `goto` 语句看上去是个可怕的设计， 但是实际上我们使用它来讲控制权转移给一个普通的exit块来执行相应的操作。 在 `fail:` 标签后面的代码和Python的 `fianl:` 块的用途是一样的。

如果你使用所有这些约定来编写C代码，包括对GIL的管理、异常检查和错误检查， 你会发现从C语言中调用Python解释器是可靠的——就算再复杂的程序，用到了高级编程技巧比如多线程都没问题。