知乎 首发于 libGemfield

pybind11的最佳实践

Gemfield
A CivilNet Maintainer

关注他

49 人赞同了该文章

背景

之前的一个小项目上，Gemfield需要封装ffmpeg代码给python用户使用，当时Gemfield写了一篇文章：

Gemfield：使用pybind11 将C++代码编译为python模块
zhuanlan.zhihu.com

可以看作是pybind11的入门，内容当然是比较简单了。在这篇文章里，Gemfield将再次详细的阐述pybind11的使用。

因为Python2已经被废弃了，以及Gemfield及团队日常使用的电脑为KDE Ubuntu，所以这篇文章默认均为Linux和Python3上的pybind11的使用。另外，Gemfield还想让你知道，目前有一些开源项目，专门用来针对已有的C++代码来自动生成pybind11绑定代码：

- binder;
- AutoWIG;
- robotpy-build;

准备环境

- KDE Ubuntu 20.04;
- 安装python3-dev;
- 安装cmake;
- boost： sudo apt install libboost-dev
- pytest: python3 -m pip install pytest

完成上述环境的准备后，克隆pybind11项目：

```
mkdir build
cd build
cmake ..
make check -j 4
make install
```

pybind11是一个只有头文件的库，那么这里为啥需要编译呢？编译的都是测试代码，用来检验环境、编译器、代码、python等是否兼容。至于安装，则是通过make install将pybind11的头文件拷贝到系统目录下：

```
gemfield@ThinkPad-X1C:~/github/prototype/pybind11/build$ sudo make install
[sudo] gemfield 的密码：
[  4%] Built target cross_module_gil_utils
[  9%] Built target pybind11_cross_module_tests
[100%] Built target pybind11_tests
Install the project...
-- Install configuration: "MinSizeRel"
-- Installing: /usr/local/include/pybind11
-- Installing: /usr/local/include/pybind11/cast.h
-- Installing: /usr/local/include/pybind11/complex.h
-- Installing: /usr/local/include/pybind11/buffer_info.h
-- Installing: /usr/local/include/pybind11/common.h
-- Installing: /usr/local/include/pybind11/operators.h
-- Installing: /usr/local/include/pybind11/functional.h
-- Installing: /usr/local/include/pybind11/attr.h
-- Installing: /usr/local/include/pybind11/pytypes.h
-- Installing: /usr/local/include/pybind11/embed.h
-- Installing: /usr/local/include/pybind11/eigen.h
-- Installing: /usr/local/include/pybind11/stl.h
-- Installing: /usr/local/include/pybind11/eval.h
-- Installing: /usr/local/include/pybind11/iostream.h
-- Installing: /usr/local/include/pybind11/numpy.h
-- Installing: /usr/local/include/pybind11/pybind11.h
-- Installing: /usr/local/include/pybind11/options.h
-- Installing: /usr/local/include/pybind11/chrono.h
-- Installing: /usr/local/include/pybind11/stl_bind.h
-- Installing: /usr/local/include/pybind11/detail
-- Installing: /usr/local/include/pybind11/detail/common.h
-- Installing: /usr/local/include/pybind11/detail/typeid.h
-- Installing: /usr/local/include/pybind11/detail/internals.h
```

```
-- Installing: /usr/local/share/cmake/pybind11/pybind11Config.cmake
-- Installing: /usr/local/share/cmake/pybind11/pybind11ConfigVersion.cmake
-- Installing: /usr/local/share/cmake/pybind11/FindPythonLibsNew.cmake
-- Installing: /usr/local/share/cmake/pybind11/pybind11Common.cmake
-- Installing: /usr/local/share/cmake/pybind11/pybind11Tools.cmake
-- Installing: /usr/local/share/cmake/pybind11/pybind11NewTools.cmake
-- Installing: /usr/local/share/cmake/pybind11/pybind11Targets.cmake
```

因为是编译安装，因此我们需要手工设置PYTHONPATH环境变量：

```
export PYTHONPATH=$PYTHONPATH:/home/gemfield/github/prototype/pybind11/
```

这样是为了你后续可以使用python3 -m pybind11 --includes 命令来得到所要包含的头文件的路径：

```
gemfield@ThinkPad-X1C:~/github/prototype/pybind11/examples$ python3 -m pybind11 --incl
-I/usr/include/python3.8 -I/home/gemfield/github/prototype/pybind11/include
```

不然会报找不到Python.h的错误。如果是pip或者conda来安装的pybind11，则无需手工指定PYTHONPATH目录。因为pybind11是纯头文件库，因此只需要指定头文件路径即可。如果因为种种原因你无法通过python3 -m pybind11 --includes来获得头文件路径，则可以通过如下的方式来获取：

- 手工指定pybind11的头文件路径：-I<path-to-pybind11>/include
- 再通过python3-config命令来获取python3-dev的头文件目录： `python3-config--includes`

使用pybind11来绑定普通函数

1，照例，我们先上hello world的例子

也就是使用pybind11来绑定一个简单的函数：

```
//syszux.cpp
#include <pybind11/pybind11.h>

int syszuxAdd(int i, int j) {
```

```
PYBIND11_MODULE(syszux, m) {
    m.doc() = "pybind11 example plugin"; // optional module docstring
    m.def("syszuxAdd", &syszuxAdd, "A function which adds two numbers");
}
```

编译：

```
g++ -O3 -Wall -shared -std=c++11 -fPIC -I/usr/include/python3.8 `python3 -m pybind11 -
```

运行：

```
gemfield@ThinkPad-X1C:~$ python3
Python 3.8.2 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import syszux
>>> syszux.syszuxAdd(3,4)
7
```

2，使用pybind11::arg来定义keyword参数：

```
m.def("syszuxAdd", &syszuxAdd, pybind11::arg("i"), pybind11::arg("j"), "syszuxAdd whic
```

这个时候就可以使用syszuxAdd(i=70,j=30)的调用形式了。

3，使用默认参数：

```
m.def("syszuxAdd", &syszuxAdd,  pybind11::arg("i") = 1, pybind11::arg("j") = 2);
```

使用pybind11来绑定class

既然pybind11是用来扩展C++，那我们还是尽快实践下类的绑定吧。和普通的函数绑定相比，绑定class的时候由m.def转变为了pybind11::class_<class>.def了；另外，还需要显式的指定class的构造函数的参数类型。

```
class CivilNet {
    public:
        CivilNet(const std::string &name) : name_(name) { }
        void setName(const std::string &name) { name_ = name; }
        const std::string &getName() const { return name_; }
    private:
        std::string name_;
};


PYBIND11_MODULE(syszux, m) {
    pybind11::class_<CivilNet>(m, "CivilNet")
        .def(pybind11::init<const std::string &>())
        .def("setName", &CivilNet::setName)
        .def("getName", &CivilNet::getName);
}
```

运行：

```
gemfield@ThinkPad-X1C:~/github/prototype/pybind11/examples$ python3
Python 3.8.2 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import syszux
>>> dir(syszux)
['CivilNet', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__
>>> x = syszux.CivilNet('gemfield')
>>> x.getName
<bound method PyCapsule.getName of <syszux.CivilNet object at 0x7f4be3daedb0>>
>>> x.getName()
'gemfield'
>>> x.setName('civilnet')
>>> x.getName()
'civilnet'
```

2，模拟python风格的property

在上面的class绑定的例子中，我们并没有办法来像python中访问property的方式来访问name_私

通过def_property的定义，我们就可以像访问python的property风格那样访问name_。运行如下：

```
gemfield@ThinkPad-X1C:~/github/prototype/pybind11/examples$ python3
Python 3.8.2 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import syszux
>>> x = syszux.CivilNet("gemfield")
>>> x.getName()
'gemfield'
>>> x.name_
'gemfield'
>>> x.name_ = 'syszux'
>>> x.getName()
'syszux'
>>>
```

类似的还有class_::def_readwrite() 、class_::def_readonly()等定义。但是这样都是在已经定义好的类成员上进行读写，而python中的对象上还可以增加动态属性，就是一个class中本没有这个成员，但是直接赋值后也就产生了.....这就是动态属性，比如上面的x.name_虽然是可以按照python风格来读写了，但是你要是直接赋值给一个x.age是不行的（我们并没有定义age成员）。那怎么办呢？

使用pybind11::dynamic_attr()，代码如下所示：

```
PYBIND11_MODULE(syszux, m) {
    pybind11::class_<CivilNet>(m, "CivilNet",pybind11::dynamic_attr())
        .def(pybind11::init<const std::string &>())
        .def("setName", &CivilNet::setName)
        .def("getName", &CivilNet::getName)
        .def_property("name_", &CivilNet::getName, &CivilNet::setName);
}
```

运行如下：

```
>>> import syszux
```

```
>>> x.gender
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'syszux.CivilNet' object has no attribute 'gender'
>>> x.__dict__
{'age': 18}
```

3，继承关系的Python绑定

设想这个时候，我们引入了继承关系，定义类Syszux继承自类CivilNet：

```
class CivilNet {
    public:
        CivilNet(const std::string &name) : name_(name) { }
        void setName(const std::string &name) { name_ = name; }
        const std::string &getName() const { return name_; }
    private:
        std::string name_;
};

class Syszux : public CivilNet {
    public:
        Syszux(const std::string& name, int age) : CivilNet(name),age_(age){}
        const int getAge() const {return age_; }
        void setAge(int age) {age_ = age;}
    private:
        int age_;
};
```

这个时候怎么绑定Syszux类到python呢？直截了当的想法是：

```
PYBIND11_MODULE(syszux, m) {
    pybind11::class_<CivilNet>(m, "CivilNet",pybind11::dynamic_attr())
        .def(pybind11::init<const std::string &>())
        .def("setName", &CivilNet::setName)
        .def("getName", &CivilNet::getName)
        .def_property("name_", &CivilNet::getName, &CivilNet::setName);

    pybind11::class_<Syszux>(m, "Syszux",pybind11::dynamic_attr())
```

```
        .def("getAge", &Syszux::getAge)
        .def_property("age_", &Syszux::getAge, &Syszux::setAge)
        .def_property("name_", &Syszux::getName, &Syszux::setName);
}
```

这样虽然能够工作，但终归不是个办法。很显然，我本意只是绑定个Syszux，却要把Syszux父类的所有成员一一列出来并且绑定，而如果不这么做的话，在Python中又无法访问Syszux父类中的成员！！！怎么办呢?

有两种方法，第一种方法是把父类当作一个模板参数声明下，如pybind11::class_<Syszux, CivilNet>:

```
PYBIND11_MODULE(syszux, m) {
    pybind11::class_<CivilNet>(m, "CivilNet",pybind11::dynamic_attr())
        .def(pybind11::init<const std::string &>())
        .def("setName", &CivilNet::setName)
        .def("getName", &CivilNet::getName)
        .def_property("name_", &CivilNet::getName, &CivilNet::setName);


    // Method 1: template parameter
    pybind11::class_<Syszux, CivilNet>(m, "Syszux")
        .def(pybind11::init<const std::string &, int>())
        .def("setAge", &Syszux::setAge)
        .def("getAge", &Syszux::getAge)
        .def_property("age_", &Syszux::getAge, &Syszux::setAge);
}
```

第二种方法就是模拟python风格，在括号里应用父类的python对象:

```
PYBIND11_MODULE(syszux, m) {
    pybind11::class_<CivilNet> civilnet (m, "CivilNet");
        civilnet.def(pybind11::init<const std::string &>())
        .def("setName", &CivilNet::setName)
        .def("getName", &CivilNet::getName)
        .def_property("name_", &CivilNet::getName, &CivilNet::setName);

    //Method 2: pass parent class_ object:
    pybind11::class_<Syszux>(m, "Syszux", civilnet)
```

```
}
```

4，多态的Python绑定

当你在模块里新增一个工厂方法（只是举例）来返回一个Syszux的实例：

```
m.def("create", []() { return std::unique_ptr<CivilNet>(new Syszux("Gemfield", 18)); }
```

你会发现虽然new的是Syszux实例，但是该实例是隐藏在CivilNet指针后的，因此这种create方法得到的实例只能使用CivilNet成员，而不能使用Syszux新扩展出来的成员。用python验证下也会发现这种尴尬情况：

```
>>> import syszux
>>> x = syszux.create()
>>> type(x)
<class 'syszux.CivilNet'>
>>> x.getName()
'Gemfield'
>>> x.getAge()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'syszux.CivilNet' object has no attribute 'getAge'
>>>
```

要改变这种情况，需要在基类中添加virtual关键字，这样pybind11就认识到当前的情况了。比如像下面这样改动后：

```
class CivilNet {
......
        virtual ~CivilNet() = default;
......
};

class Syszux : public CivilNet {
......
};
```

```
>>> type(x)
<class 'syszux.Syszux'>
>>> x.getAge()
18
```

5，函数重载的Python绑定

大多数情况下，C++的代码中都会出现函数重载的情况，就像下面所示的代码：

```
class Syszux : public CivilNet {
    public:
        ......
        void setAge(int age) {age_ = age;}
        void setAge(std::string age){ages_ = age;}
    private:
        int age_;
        std::string ages_;
};
```

这个时候如果还像之前那样绑定的话，就会出现**unresolved overloaded function type**的错误。pybind11当然提供了对应的解决方案，不止一种，Gemfield这里展示更优雅的C++14的解决方案，使用pybind11::overload_cast：

```
pybind11::class_<Syszux>(m, "Syszux", civilnet)
    .def(pybind11::init<const std::string &, int>())
    .def("setAge", pybind11::overload_cast<int>(&Syszux::setAge))
    .def("getAge", &Syszux::getAge)
    .def_property("age_", &Syszux::getAge, pybind11::overload_cast<int>(&Syszux::setAg
```

当然了，编译时候的命令就需要从-std=c++11更换为-std=c++14了：

```
g++ -O3 -Wall -shared -std=c++14 \
    -fPIC -I/usr/include/python3.8 `python3 -m pybind11 --includes` syszux.cpp -o sysz
```

6，对enum的绑定

当在Python中调用pybind11绑定的C++代码时，如果这部分C++代码抛出了异常，就像下面这样：

```
class Syszux : public CivilNet {
    public:
        Syszux(const std::string& name, int age) : CivilNet(name),age_(age){}
        const int getAge() const {throw std::runtime_error("I,AM,GEMFIELD"); return ag
        void setAge(int age) {age_ = age;}
        void setAge(std::string age){ages_ = age;}
    private:
        int age_;
        std::string ages_;
};
```

还是上文那个Syszux类，在getAge函数里我手动设置抛出了runtime_error异常，那么在Python中调用这个函数时，会发生什么呢？Pybind11会将这些C++的异常转换为对应的Python异常：

```
>>> import syszux
>>> x=syszux.create()
>>> x.getAge()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: I,AM,GEMFIELD
>>> try:
...   x.getAge()
... except Exception as e:
...   print(type(e))
...
<class 'RuntimeError'>
>>>
```

pybind11定义了C++中std::exception（及其子类）到Python异常的转换关系：

`std::exception`	`RuntimeError`
`std::bad_alloc`	`MemoryError`
`std::domain_error`	`ValueError`
`std::invalid_argument`	`ValueError`
`std::length_error`	`ValueError`
`std::out_of_range`	`IndexError`
`std::range_error`	`ValueError`
`std::overflow_error`	`OverflowError`
`pybind11::stop_iteration`	`StopIteration` (used to implement custom iterators)
`pybind11::index_error`	`IndexError` (used to indicate out of bounds access in `__getitem__`, `__setitem__`, etc.)
`pybind11::value_error`	`ValueError` (used to indicate wrong value passed in `container.remove(...)`)
`pybind11::key_error`	`KeyError` (used to indicate out of bounds access in `__getitem__`, `__setitem__` in dict-like objects, etc.)

对于那些自定义的C++异常，你就需要使用pybind11::register_exception了；对于从C++中去catch来自Python的异常，你就需要了解pybind11::error_already_set了。

变量导出

使用attr函数可以将c++中的变量导出到Python模块中：

```
PYBIND11_MODULE(syszux, m) {
    m.doc() = "pybind11 example plugin"; // optional module docstring
    m.def("syszuxAdd", &syszuxAdd, pybind11::arg("i"), pybind11::arg("j"), "A function
    m.attr("gemfield") = 7030;
```

上述代码导出了两个符号：gemfield和c，运行如下：

```
>>> dir(syszux)
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'c', 'gem
>>> syszux.c
'CivilNet'
>>> syszux.gemfield
7030
```

至于上述代码中的pybind11::cast是怎么回事，可以看下一节的类型转换。

C++和Python之间的类型转换

如果你已经看到了这里，相信你对pybind11已经有了一个非常直接的感觉。在pybind11提供的方便的功能背后，很多人都会问一个问题，pybind11究竟做了什么让C++到Python的绑定变得这么容易？我想其中最重要的一点可能就是类型转换——一个Python的对象实例如何转换成了一个C++的对象实例，以及反过来一个C++的对象实例如何转换成了Python对象实例。只有实现了类型在C++和Python之间的相互转换，我们才能像浮在表面上那样自由的在C++中访问Python中定义的对象，以及在Python中访问C++中定义的对象。

其实，在本文的前几个章节，你可能已经感受到了一点，就是在绑定一个C++的函数到Python的时候，我们对于函数的参数列表和返回值都没怎么关心过！这也一定预示着一个事实：pybind11完成了这些类型在C++和Python之间的自动转换！新的问题随之而来，都有什么类型可以被pybind11自动转换呢？自己随便写的一个类可以吗？

没错，pybind11提供了三种基本的机制来实现C++和Python之间的类型转换，前两种就是个wrapper（看谁wrap谁），最后一种才是真正的类型转换（要有内存的拷贝）：

1，类型定义在C++中，而在Python中访问

假设定义了一个C++类型Syszux，pybind11自动生成Syszux的wrapper，该wrapper又符合Python的内存布局规范，从而python代码可以访问Syszux；也就是pybind11封装C++类型来兼容Python。

比如，对于前述章节的pybind11::class_封装的Syszux类，Syszux类生命周期内都是一个C++的

2，类型定义在Python中，而在C++中访问

这和第一种情况相反，这种情况下我们假设有一个Python的类型，比如tuple或者list，然后我们使用pybind11::object系列的wrapper来封装tuple、list或者其它，从而得以在C++中访问。比如下面这个例子：

```
void print_list(pybind11::list my_list) {
    for (auto item : my_list)
        std::cout << item << " ";
}
```

然后在python中：

```
>>> print_list([7, 0, 3,  0])
7 0 3 0
```

在这个调用中，Python的 `list` 并没有被本质的转换，只不过被wrap在了C++的pybind11::list类型中。像list这样支持直接被wrap的python类型还有

- handle
- object
- bool_
- int_
- float_
- str
- bytes
- tuple
- list
- dict
- slice
- none
- capsule
- iterable
- iterator
- function
- buffer

前两种情况下，我们都是在C++或者Python中有一个native的类型，然后通过wrapper来互相访问。在第三种情况下，我们在C++和Python中都使用自身native的类型——然后尝试通过复制内存的方式来转换它们。比如，在C++中定义函数：

```
void print_vector(const std::vector<int> &v) {
    for (auto item : v)
        std::cout << item << "\n";
}
```

在python中运行：

```
print_vector([1, 2, 3])
>>> 1 2 3
```

python的list是怎么跑到C++的vector上的呢？原来呀，pybind11构造了一个全新的std::vector<int>对象，然后将python list中的每个元素拷贝了过来——然后这个新构造的std::vector<int>对象再传递给print_vector函数。pybind11默认支持很多种这样的类型转换，虽然比较方便，比如上面是python list到c++ vector，也可以是Python tuple到c++ vector。到底有多少种类型支持默认转换呢，如下所示：

- int8_t, uint8_t
- int16_t, uint16_t
- int32_t, uint32_t
- int64_t, uint64_t
- ssize_t, size_t
- float, double
- bool
- char
- char16_t
- char32_t
- wchar_t
- const char *
- const char16_t *
- const char32_t *
- const wchar_t *
- std::string
- std::u16string

- std::tuple<...>
- std::reference_wrapper<...>
- std::complex<T>
- std::array<T, Size>
- std::vector<T>
- std::deque<T>
- std::valarray<T>
- std::list<T>
- std::map<T1, T2>
- std::unordered_map<T1, T2>
- std::set<T>
- std::unordered_set<T>
- std::optional<T>
- std::experimental::optional<T>
- std::variant<...>
- std::function<...>
- std::chrono::duration<...>
- std::chrono::time_point<...>
- Eigen::Matrix<...>
- Eigen::Map<...>
- Eigen::SparseMatrix<...>

总结起来就是基础类型、容器类型、std::function、std::chrono、Eigen。但是要注意这种转换中间引入了完完全全的内存拷贝！小的类型还可以，但是对于大内存的类型就非常不友好了。还有，这其中有些转换虽然是自动的，但是需要你手工添加相应的pybind11头文件，不然会在运行时报错：

```
Did you forget to `#include <pybind11/stl.h>`? Or <pybind11/complex.h>,
<pybind11/functional.h>, <pybind11/chrono.h>, etc. Some automatic
conversions are optional and require extra headers to be included
when compiling your pybind11 module.
```

4, protocol 协议

当我们在前一个小节中愉快的实践了pybind11的自动类型转换，比如下面这样：

```
void print_vector(const std::vector<int> &v) {
```

在python中运行：

```
print_vector([1, 2, 3])
>>> 1 2 3
```

我们就会发现，在python中传递list是可以的，传递tuple也是可以的，传递dict是**不**可以的，那么传递numpy的ndarray呢？我们关心numpy是因为这个东西是python中做运算的基础类型呀。然后，你只要稍微尝试下numpy作为参数，你就会发现，当ndarray是一维的话，确实可以像list一样正常传递，而如果ndarray是二维及以上的话，再想把numpy从python传递到C++的vector中，就会报如下的错误：

```
#s是Syszux的实例
>>> s.test(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: test(): incompatible function arguments. The following argument types are s
    1. (self: syszux.Syszux, arg0: List[int]) -> None

Invoked with: <syszux.Syszux object at 0x7f5fc0a4f7b0>, array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

那怎么办呢？对于Numpy这种一维是数组、二维是矩阵、三维及以上是张量的类型来说，都可以抽象为buffer，这个buffer view规范里说了，你需要包含：

- buffer的首地址；
- buffer里每个scaler的大小；
- 维度的个数（2维？3维？）
- 维度（比如：2维下的1920 x 1080）
- 步长（在每个维度上前进一个索引，所移动的内存字节数）

因此我们可以使用pybind11提供的buffer protocol来解决这个问题。现在我们就使用buffer protocol来定义一个C++类Gemfield，可以从C++传递到Python中，继而可以转换为numpy……咦？在CivilNet的SYSZUXav项目中，不正是有一个现成的例子吗：

https://github.com/CivilNet/SYSZUXav
/blob/master/src/syszuxav.cpp

为构造函数的参数，就像下面这样：

```
class Gemfield {
public:
    Gemfield(size_t rows, size_t cols) : m_rows(rows), m_cols(cols) {
        m_data = new float[rows*cols];
    }
    float *data() { return m_data; }
    size_t rows() const { return m_rows; }
    size_t cols() const { return m_cols; }
private:
    size_t m_rows, m_cols;
    float *m_data;
};
...
void test(Gemfield vi){
    std::cout<<vi.cols()<<" x "<<vi.rows()<<std::endl;
}
...
PYBIND11_MODULE(syszux, m) {
    pybind11::class_<Gemfield>(m, "Gemfield", pybind11::buffer_protocol())
        .def(pybind11::init([](pybind11::buffer const b) {
            pybind11::buffer_info info = b.request();
            if (info.format != pybind11::format_descriptor<float>::format() || info.nd
                throw std::runtime_error("Incompatible buffer format!");

            auto v = new Gemfield(info.shape[0], info.shape[1]);
            memcpy(v->data(), info.ptr, sizeof(float) * (size_t) (v->rows() * v->cols(
            return v;
        }));
......
```

然后在Python中运行：

```
>>> import syszux
>>> x = syszux.create()
>>> import numpy as np
>>> a = np.arange(10).reshape(2,5)
>>> x.test(syszux.Gemfield(np.float32(a)))
```

任何兼容protocol协议的类型。

函数返回值和传参时候的内存处理

让我们来见识下一个简单的函数绑定的例子：

```
static Gemfield gemfield(70,30);

Gemfield* getGem() {
    return &gemfield;
}

PYBIND11_MODULE(syszux, m) {
    m.def("getGem", &getGem, "get Gem");
......
```

好简单的一个pybind11绑定呀。然后运行，意想不到的事情发生了：

```
gemfield@ThinkPad-X1C:~/examples/build$ python3
>>> import syszux
>>> syszux.getGem()
<syszux.Gemfield object at 0x7f3af3951bf0>
>>>
free(): invalid pointer
Aborted (core dumped)
```

看到上面退出python解释器时的Aborted (core dumped)的crash悲剧了吗？？？谁能说说发生了什么？？？

Python和C++使用了截然不同的内存管理和对象生命周期管理。因此，pybind11提供了好几种函数返回值的内存拷贝策略，而默认策略是return_value_policy::automatic。在上面的例子中，当getGem()从python中返回时，返回类型Gemfield被wrap为一个Python类型，默认的内存拷贝策略是return_value_policy::automatic，这就导致python的wrapper认为自己是&gemfield这块内存的主人（这块内存是在static区域）。当结束python会话时，Python的内存管理器会删除Python wrapper——然后删除其管理的内存——然后C++部分也会销毁gemfield对象的内存——这就导致了双重delete——程序crash了。

这样的话全局的gemfield内存就会被reference而避免了被双重delete的命运。至此，你就明白了pybind11中提供的各种函数返回时的内存拷贝策略的重要性了。在pybind11中，一共有如下这么多的内存拷贝策略：

- return_value_policy::take_ownership
- return_value_policy::copy
- return_value_policy::move
- return_value_policy::reference
- return_value_policy::reference_internal
- return_value_policy::automatic
- return_value_policy::automatic_reference

在C++项目中使用CMake来集成pybind11

在本文的前述部分，你会发现编译的时候我们都是手动执行g++命令，如下所示：

```
g++ -O3 -Wall -shared -std=c++11 -fPIC -I/usr/include/python3.8 `python3 -m pybind11 -
```

这里面需要手工指定头文件的路径，已经让人感觉到不方便了。而在一般的C++项目中，我们一般都是使用CMake来组织工程的，那么，如何在CMake中来集成对pybind11的使用呢？pybind11当然也想到了这个问题，通过提供pybind11_add_module接口来方便用户的使用。但是要使用到pybind11_add_module接口，首先需要包含pybind11的CMake模块。C++项目中主要通过两种方式来包含：add_subdirectory和find_package。比如，上面的g++命令可以重构为如下的CMake的组织方式：

```
cmake_minimum_required(VERSION 3.4...3.18)
project(syszux LANGUAGES CXX)

find_package(pybind11 REQUIRED)
pybind11_add_module(syszux syszux.cpp)
```

这里使用了find_package。通过make VERBOSE=1，你可以观察到CMakeList.txt生成的Makefile在被make的时候，编译参数如下所示：

```
/usr/bin/c++   -Dsyszux_EXPORTS -isystem /usr/include/python3.8 -fPIC -fvisibility=hid
```

两者之间有什么区别呢？我们先来说说C++项目里使用pybind11的两种经典方式：

- 一种是pybind11本身是安装在系统目录下的，就像C++的标准库一样，这样一个或多个项目可以共同使用；这个时候，使用find_package；
- 另外一种就是pybind11仓库本身就作为三方仓库集成到你的项目中，这样recursive clone你的项目后，你的项目中本身是包含pybind11这些个头文件的；这个时候，使用add_subdirectory。

只有在find_package或者add_subdirectory成功之后，CMakeLists.txt中的pybind11_add_module才能被调用。当然pybind11_add_module还有好多参数，如有需要可以查看文档。如下所示：

```
pybind11_add_module(<name> [MODULE | SHARED] [EXCLUDE_FROM_ALL]
                    [NO_EXTRAS] [THIN_LTO] source1 [source2 ...])
```

将Python解释器集成到你的C++程序中

pybind11的目的主要是使得用户可以方便的在python中调用C++模块。然而它也可以帮助用户将python解释器集成到C++程序中，这样我们的C++程序就可以像调用函数一样调用py模块了。这种情况下，你的CMakeLists.txt就不能继续使用pybind11_add_module了，反过来，应该是link一个pybind11::embed目标。照例，我们先来个hello world：

```
#include <pybind11/embed.h> // everything needed for embedding
int main() {
    pybind11::scoped_interpreter guard{}; // start the interpreter and keep it alive
    pybind11::print("Hello, World!"); // use the Python API
}
```

很明显的就是头文件换成embed.h了。然后，CMakeLists.txt需要做如下改变，像上面说的，不是要做一个python插件了，而是要把python解释器集成到C++程序中：

```
#pybind11_add_module(syszux syszux.cpp)
#修改为
add_executable(syszux syszux.cpp)
target_link_libraries(syszux PRIVATE pybind11::embed)
```

看到了吧，链接的正是/usr/lib/x86_64-linux-gnu/libpython3.8.so 库。运行程序syszux：

```
gemfield@ThinkPad-X1C:~/examples/build$ ./syszux
Hello, World!
```

main函数中短短两行代码却也暗藏玄机，其中的pybind11::scoped_interpreter初始化的时候，也正是python解释器生命周期开始的时候；当pybind11::scoped_interpreter对象销毁的时候，也正是python解释器生命周期结束的时候。而在这之后，如果再次实例化了pybind11::scoped_interpreter对象，则python解释器的生命周期又会重新开始。

执行python代码

我们辛辛苦苦集成了python解释器并不是为了跑个hello world，最起码要运行个python代码吧。我们可以使用 `eval`，`exec`或者`eval_file` ，下面是使用exec的一个例子：

```
#include <iostream>
#include <pybind11/embed.h> // everything needed for embedding
int main() {
    pybind11::scoped_interpreter guard{};
    std::cout << "GEMFIELD: c++ part pid is: " << ::getpid() << " | and parent pid is:
    pybind11::exec(R"(
        import os
        kwargs = dict(name="Gemfield", pid=os.getpid(), ppid=os.getppid())
        message = "Hello, {name}! The python part pid is: {pid} | and parent pid is: {
        print(message)
        import time
        time.sleep(50)
    )");
}
```

运行该程序如下所示：

```
gemfield@ThinkPad-X1C:~/examples/build$ ./syszux
GEMFIELD: c++ part pid is: 1270481 | and parent pid is: 131479
Hello, Gemfield! The python part pid is: 1270481 | and parent pid is: 131479
```

知乎 首发于 libGemfield

编辑于 2020-08-29

「真诚赞赏，手留余香」

赞赏

还没有人赞赏，快来当第一个赞赏的人吧！

Python C++ pybind11

文章被以下专栏收录

libGemfield
C++、ELF、LLVM

推荐阅读

使用pybind11 将C++代码编译为python模块

背景我们大家平常使用的python实现都是cpython，所以使用C语言或者C++来写一些扩展的时候，就相当于在写cpython的插件。cpython的扩展关键在于要实现一个 PyObject*...

Gemfield

混合编程：如何用python用C++

406龚煽使

知乎 首发于 libGemfield

写下你的评论...

啦啦啦 2020-10-02

请问cmake中不需要add_executable()或者add_library()吗，我这里ide提示不加无法编译

赞

码来码去 2020-08-30

pybind11是否能把numpy对象传递到C++转为多维数组?

赞