# 第 2 章　程序设计

## 2.1　从 C++ 到 Python

### 2.1.1　包装函数

一个 Cython 包装函数分为三部分：

- 将输入的 Python 对象转换为特定类型的 C++ 变量；
- 调用 C/C++ 接口，传入转换后的输入参数；
- 将接口的返回值转换为 Python 对象。

Cython 能自动处理一部分 C/C++ 类型与 Python 对象间的类型转换，包括基本数值类型、字符串和部分 STL 容器。对于其它类型，包括数组、指针、枚举、类、结构体和联合，CPP2PY 为其专门生成类型转换代码。函数和类方法都以这种方式得到包装。

类的数据成员在 Python 侧被包装为类的属性，CPP2PY 专门为其生成 getter 函数，如果该数据成员或变量是可变的，还会生成 setter 函数。例如，对于如下 C++ 声明，

```
struct Point {
    int x;
    const char * LUCKY;
}
```

将生成对应 Python 接口，

```
class Point:
    @property
    def x(self) -> np.int32: ...
    @x.setter
    def x(self, x: np.int32): ...
    @property
    def LUCKY(self) -> str: ...
```

容易看出，getter 是参数为空，返回值类型为对应变量类型的方法，而 setter 的返回值为空，唯一的参数类型为相应变量类型。生成它们的逻辑与普通的函数或方法完全一致。

CPP2PY 支持 C/C++ 的全局变量，然而，背后的实现机制并不直观。当你在

Python 中写下这段代码时，

```
pi = 3.14
a = pi
```

只有一个对象包含浮点数 3.14，而 `a` 和 `pi` 都是它的名称。这种赋值机制与 C/C++ 完全不同——对于后者来说，一个变量指向一段内存，赋值表示将数据复制到该内存位置。因此，没有直接的方法将 C 中的变量赋值映射到 Python 中的变量赋值。CPP2PY 处理全局变量的方法是创建一个特殊的全局对象，将全局变量包装为该对象的属性。

这样，CPP2PY 以一种优雅的方式统一封装了变量、函数、方法和数据成员。

### 2.1.2 代理类

Cython 提供了两种定义类的语法，一种是普通的 Python 类，另一种被称为扩展类。与 Python 类相比，扩展类使用 C 结构体，而不是 Python 字典来存储字段和方法，因而具有显著的性能优势。扩展类中可以存储 C 类型字段，CPP2PY 用它来封装类、结构体与联合。

CPP2PY 在 Cython 端为 C++ 类生成代理类，即同名的 Python 扩展类，其中保存了指向 C++ 对象的指针和标记指针所有权的布尔属性，在构造函数中分配内存并获取所有权，在析构函数中检查所有权、释放内存。代理类机制使得 Python 用户能以非常自然的方式访问 C++ 数据结构。

```
def __cinit__(self):
    self.thisptr = NULL
    self.owner = True

def __dealloc__(self):
    if self.owner and self.thisptr != NULL:
        del self.thisptr
        self.thisptr = NULL
```

美中不足的是，与普通 Python 类相比，扩展类只支持有限的面向对象语法。类的静态数据成员不受支持，CPP2PY 将它们视作带作用域的变量；扩展类间的多重继承不受支持，也不允许在子类中重定义 C 类型字段。CPP2PY 以一种间接的方式处理继承——拷贝超类所有的方法和数据成员到派生类中。这种实现的优点是避免生成过于复杂的代码，也支持了多重继承，而缺点则是，类的继承关系将无法从 C++ 映射到 Python 中，第三章中详细阐述了这种妥协带来的限制。

为了计算出一个类所有的超类和派生类，CPP2PY 根据类的继承关系做拓扑排序。

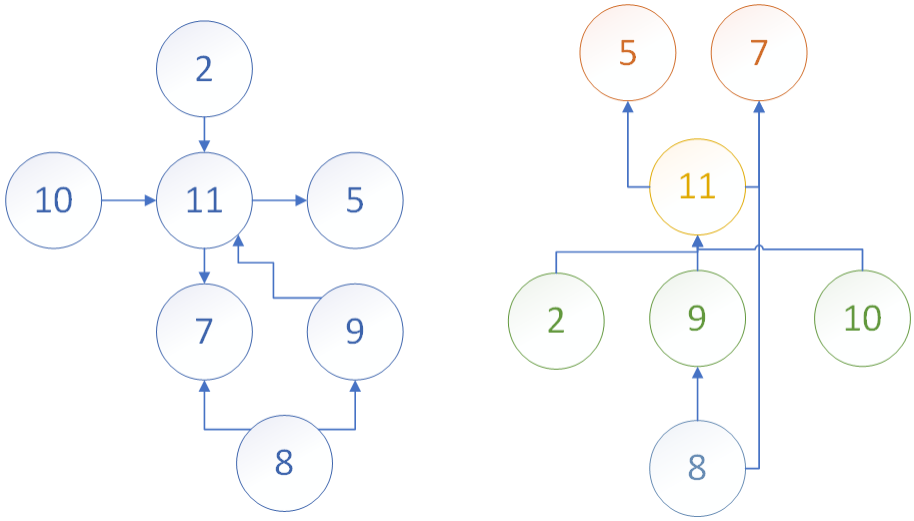


图 2.1　拓扑排序前后的复杂继承关系图

为了说明拓扑排序的效果，考虑如图 2.1 左侧所示的复杂继承关系，图中每个节点都代表一个类，每条边由子类指向它的父类，在左图中想要找到类 8 所有的超类并不容易。根据类的父子关系对它们拓扑排序后，得到的结果如图 3 右侧所示。可以立即看出，类 11 依赖于类 5 和类 7，而类 8 依赖于类 9、类 11，类 5 和类 7。类 5 和类 7 的拓扑序最靠前，而类 11 次之，类 8 的拓扑序排在最后，一个类的超类在拓扑序列必定排在它前面，派生类必定排在后面。

得到拓扑序后，只需按顺序逐个将每个类的所有父类接口拷贝到类中，即可完成拷贝超类接口的任务。

## 2.2　模块设计

CPP2PY 以 Python 语言编写，其模块设计如图2.2 所示。程序主要包含五个子模块，分别是 Config、Parser、TypeSystem、PostProcessor 和 Generator；此外还依赖于两个外部库，用于解析 C++ 代码的 libclang 和用于编译目标代码的 Cython。模块的划分遵循”高内聚、低耦合“的设计原则，是在项目编写实践中逐步确定的。

Config 模块定义了控制程序行为的输入参数，libclang 负责解析 C++ 头文件，生成源文件的抽象语法树 (Abstract Syntax Tree)。Parser 模块解析抽象语法树，从中提取所有必要信息，作为 PostProcessor 的输入。

TypeSystem 建立在 libclang 的类型系统之上。该模块还定义了在 C++ 和 Python

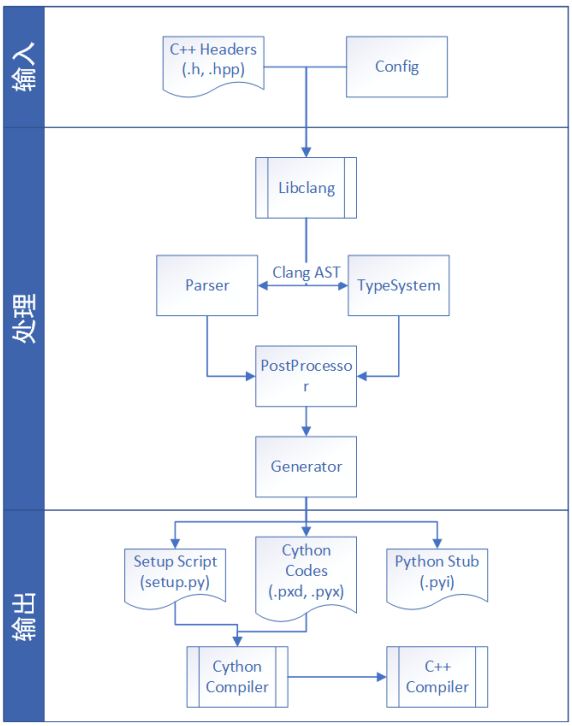


图 2.2　模块划分与数据流图

之间转换数据的类型转换器，并对相当一部分类型内置了转换功能。通过继承类型转换器抽象类，用户可以重新定义类型转换的行为，或者使 CPP2PY 支持新的数据类型。

PostProcessor 模块接受 Parser 模块的输入，处理类的继承、函数重载等问题，并为函数、方法和变量绑定类型转换器。

Generator 接受 PostProcessor 的输入，输出目标代码。生成的文件包括 Cython 代码、Python 构建脚本和存根。

其中，Cython 代码文件分成两部分，所有 C++ 接口的声明在 Cython 声明文件（`.pxd`）中，而 Cython 实现文件（`.pyx`）包含了封装这些接口的代码。这样设计使得 C++ 接口与 Python 接口不在同一个作用域内，避免了名称冲突，也给用户二次开发提供了更多的便利。

Python 存根文件与普通 Python 文件的区别是不包含具体实现。利用它，Python IDE 就能为 CPP2PY 生成的二进制代码库提供类型注解和代码补全。得益于此，用户的开发体验将大大提高。不过值得一提的是，作为动态类型语言，Python 并不在乎对象的实际类型，而只关心它是否实现了特定的属性或方法。这种类型系统通常被称为“鸭子类型”。

构建脚本指明了目标代码编译、链接的参数。用户可以根据自己的需要修改

构建脚本，虽然 CPP2PY 只在 Ubuntu 上进行了测试，但得益于 Python 构建工具 setuptools，它生成的代码能方便地在不同平台上编译，达到“一次生成，处处构建”的效果。

从 C++ 头文件生成 Cython 包装代码，并没有确定的标准，但 CPP2PY 有一个设计原则，那就是是避免生成过于复杂的代码，不追求完全保留 C++ 的语法和语义。

## 2.3 基于 libclang 的 C++ 头文件解析

众所周知，C++ 的语法复杂多变，包含了数种编程范式，不计其数的边界情况和庞大的历史包袱。Clang 是 LLVM 的编译前端，支持 C、C++、Objective-C 三种语言的解析。libclang[8] 提供了访问 Clang 抽象语法树的 Python 接口，CPP2PY 依赖它完成 C++ 语法解析任务。

Clang 抽象语法树的组成如图 2.3 所示，根节点可视为顶层命名空间，它的子节点类型包括命名空间、枚举、类、结构体、联合、变量、函数、类型别名和宏；而类节点的子节点类型又包括数据成员、静态数据成员、方法、构造函数等……根据这样的父子节点关系，Parser 模块层层迭代遍历抽象语法树。

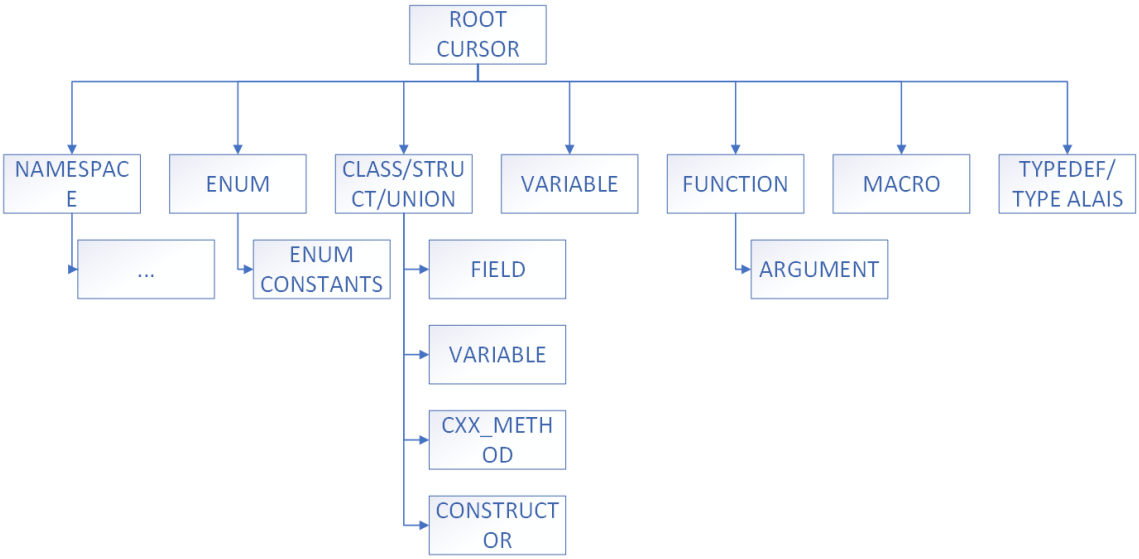


图 2.3 Clang 抽象语法树示意图

在解析过程中，C++ 的结构体和联合可以被视作特殊的类，它们的独特之处仅仅在于结构体的成员和继承方式默认为公有，而类默认为私有；联合的所有数据成员共用同一块内存。同样的道理，通过不同语法定义的类型别名也没有实质

上的差别。在 C++ 中，通过类的对象访问其非公有成员是不可能的，因此类的任何非公有成员都将被忽略。

解析宏是一件困难的事，因为宏不带有任何额外的类型信息。一个宏可能定义了一段过程、一个类型别名、也可能递归生成了非常复杂的代码。CPP2PY 只能识别定义了简单字面量的宏，包括数值类型和字符串，忽略它解析不了的宏定义。

## 2.4 软件测试

Coverage report: 93%

| Module | statements | missing | excluded | coverage |
|---|---|---|---|---|
| cpp2py/__init__.py | 5 | 0 | 0 | 100% |
| cpp2py/config.py | 56 | 2 | 0 | 96% |
| cpp2py/generator/__init__.py | 3 | 0 | 0 | 100% |
| cpp2py/generator/decl.py | 70 | 4 | 0 | 94% |
| cpp2py/generator/impl.py | 46 | 0 | 0 | 100% |
| cpp2py/generator/stub.py | 6 | 0 | 0 | 100% |
| cpp2py/main.py | 79 | 14 | 0 | 82% |
| cpp2py/parser/__init__.py | 2 | 0 | 0 | 100% |
| cpp2py/parser/libclang.py | 31 | 3 | 0 | 90% |
| cpp2py/parser/parser.py | 206 | 20 | 0 | 90% |
| cpp2py/parser/parser_types.py | 80 | 1 | 0 | 99% |
| cpp2py/parser/utils.py | 45 | 9 | 0 | 80% |
| cpp2py/process/__init__.py | 1 | 0 | 0 | 100% |
| cpp2py/process/func.py | 133 | 0 | 0 | 100% |
| cpp2py/process/postprocess.py | 150 | 8 | 0 | 95% |
| cpp2py/typesystem/__init__.py | 2 | 0 | 0 | 100% |
| cpp2py/typesystem/cxxtypes.py | 62 | 0 | 0 | 100% |
| cpp2py/typesystem/type_conversion.py | 235 | 26 | 0 | 89% |
| cpp2py/utils.py | 51 | 1 | 0 | 98% |
| **Total** | **1263** | **88** | **0** | **93%** |

*coverage.py v6.3.3, created at 2022-05-24 10:17 +0800*

图 2.4　软件测试覆盖率

合理的软件测试既能保证程序的正确运行，又能提高设计和开发的生产力。CPP2PY 的测例分单元测试和集成测试两类。单元测试主要测试程序的功能性模块，如拓扑排序、宏文本解析等。而集成测试以 C++ 代码为输入，测试 CPP2PY 能否正确地解析输入、生成目标代码并编译运行、实现预期的封装效果。

CPP2PY 经过了充分的测试，代码覆盖率达 93%。测试环境为 Ubuntu 20.04，软件版本为 Python 3.8.10、Cython 0.29.28、libclang 12。

# 第 3 章　功能介绍

## 3.1　运行 CPP2PY

CPP2PY 能以命令行工具和 Python API 两种方式运行，其主要输入参数如表 3.1 所示。这些输入参数可分为四类，指定输入输出文件信息的基本参数、传递给 Libclang 的输入解析参数、传递给构建脚本的编译参数、以及控制代码生成行为的其它参数。其中前三类参数的语义大多是显而易见的，第四类参数将在下文中介绍。

表 3.1　CPP2PY 的主要输入参数

| 参数名 | 参数解释 | 默认值 |
|---|---|---|
| `headers` | C++ 头文件路径 | - |
| `sources` | C++ 实现文件路径 | 空 |
| `modulename` | 生成模块名 | - |
| `setup_filename` | 生成的构建脚本名 | setup.py |
| `target` | 生成文件路径 | 当前路径 |
| `encoding` | 文件编码 | UTF8 |
| `incdirs` | Libclang 头文件包含路径 | 空 |
| `libclang_flags` | 传递给 Libclang 的额外参数，如 `-DDEBUG` | 空 |
| `libraries` | 外部链接库 | 空 |
| `library_dirs` | 外部链接库路径 | 空 |
| `compiler_flags` | 编译选项 | 空 |
| `build` | 是否编译 Cython 代码 | `True` |
| `generate_stub` | 是否生成 Python 存根文件 (.pyi) | `True` |
| `global_vars` | 全局变量和宏所在对象名 | `cvar` |
| `registered_converters` | 自定义的类型转换器 | 空 |
| `renames_dict` | 自定义的函数和变量重命名规则 | 空 |

通过命令行指令调用 CPP2PY，是打包简单 C/C++ 程序的不二之选。例如，下列指令将由头文件 modules.h 和代码文件 modules.cpp 组成的 C++ 项目打包成 Python 包，并命名为 cppproj。

```
cpp2py modules.h --sources modules.cpp --modname cppproj
```

如果有较为复杂的需求，如链接已有的动态共享库，则可以编写 Python 脚本，通过 API 调用 CPP2PY。下例指定了动态共享库的路径，如果不这样做的话，将在运行时因找不到共享库而引发导入异常。

```
from cpp2py import Config, make_cython_extention

config = Config(
    # ...
    library_dirs=[library_dir]
)
make_cython_extention(config)
```

## 3.2 函数

当类型转换器模块提供了所有参数和返回值的处理方法时，C/C++ 函数被包装为 Python 函数。默认情况下，CPP2PY 支持的数据类型如表 3.2 所示。

### 3.2.1 可扩展的类型转换器

CPP2PY 构建了一套具有可扩展性的类型转换器，用户可以通过继承抽象类 `AbstractTypeConverter` 定义新的类型转换器，向程序提供新类型的转换方法，或重定义内置类型的转换行为。子类必须重写以下抽象方法：

- `matches`：根据参数类型和参数名判断本类型转换器是否适用于该参数，或根据返回值类型和函数名判断本类型转换器是否适用于该返回值。
- `input_type_decl`：Cython 包装函数的输入参数类型；
- `python_to_cpp`：规定如何将输入的 Python 对象转换为 C++ 参数；
- `cpp_call_arg`：规定调用 C++ 接口时如何传递参数；
- `return_output`：规定如何将 C++ 接口的返回值转换为 Python 对象并返回。

在五个抽象方法中，第二至第四个是用来处理函数参数的，而第五个用于处理返回值。除此之外，`AbstractTypeConverter` 还定义了一些非抽象方法，必要时用户可以通过重写它们扩展类型转换器的功能。

- `add_imports`：规定需要额外导入的包，默认为空；
- `pysign_type_decl`：规定在 Python 存根文件中如何标注类型，默认为 `Any`；

表 3.2　数据在 C++ 与 Python 间的转换

| Python => | C++ | => Python |
|---|---|---|
| - | `void` | `None` |
| 基本数值类型 | 基本数值类型，包括布尔、整数、浮点、复数（`std::complex`） | 基本数值类型 |
| `array.array`, `numpy.ndarray` | 基本数值类型指针 | 被指向的数据 |
| `Iterable` | 基本数值类型定长数组 | `list` |
| `str` | 字符串（`char *`, `std::string`） | `str` |
| `Iterable[str]` | 字符数组（`char **`） | 被指向的字符串 |
| `Mapping`/`Iterable` | STL 容器① | `set`/`list`/`dict`/`tuple` |
| 枚举类 | 枚举 | 枚举类 |
| 类 | 类、结构体、联合② | 类 |
| 类 | 类、结构体、联合的指针和二重指针 | 类 |

① 支持 `vector`/`set`/`unordered_set`/`map`/`unordered_map`/`pair` 与基本数值类型、字符串的递归组合。
② 作为返回值时要求类、结构体、联合具有拷贝构造函数。

在 C 程序中，常常用 `void *` 类型的变量充当泛型指针。有鉴于此，CPP2PY 内置了一个可选的 `void *` 类型转换器 `VoidPtrConverter`，继承该类，指定 `void *` 的实际指针类型，再将类型转换器子类以参数 `registered_converters` 输入 Config 模块，就能让 CPP2PY 以处理基本数据类型指针的方式处理 `void *`。

```
from cpp2py import VoidPtrConverter

class DataConverter(VoidPtrConverter):
    def real_type(self) -> str:
        return "double"
```

### 3.2.2　基本类型指针和数组

Python 定义了一套包装底层连续内存的 C 层级标准，称为缓存协议。遵照缓存协议设计的数组对象之间可以快速转换，而不需要内存的复制，几乎知名 Python 库中的数组对象都实现了缓存协议，例如 Python 标准库 `array`，数值计算库 NumPy 中的多维数组。缓存协议是 Python 庞大科学计算生态的基石。

基于缓存协议，Cython 提供了一种特殊的数据结构，带类型的内存视图（typed memoryview），能够与 C 数组、Python 数组或 NumPy 数组相互转换。CPP2PY 使用这种数据结构来向 C 接口中传递基本数据类型的指针。

举例而言，假如有如下函数定义，

```
double norm2(double[] vec, unsigned size);
void increment(int* i);
```

用户将能在 Python 端以标准库数组或 NumPy 数组的形式向底层接口传递数组和指针参数。

```
>>> from array import array
>>> arr = array('d', [4., 3.])
>>> norm2(arr, len(arr))
5.0
>>> import numpy as np
>>> arr = np.array([1], dtype=np.int32)
>>> increment(arr); arr
array([2], dtype=int32)
```

如果输入的数组元素类型与底层接口不匹配，将引发异常。

### 3.2.3 引用和常量

在 Python 中，没有与右值引用和常量相对应的概念，因此函数参数和返回值的右值引用标识符 `&&` 和常量标识符 `const` 将被忽略。

对于左值引用参数，只有当它指向一个类时，对引用的修改才会反映到输入参数上，因为此时 C++ 接口的输入参数直接来自于代理类，而在其他情况下，输入的 Python 对象会被先转换成 C++ 变量，对 C++ 变量的修改不会反作用到原 Python 对象上。

CPP2PY 不特殊处理引用类型返回值。当然，用户可以通过自定义类型转换器来修改 CPP2PY 的行为。

### 3.2.4 默认参数

CPP2PY 支持默认参数语法，受支持的默认值类型包括基本数据类型，即整数、浮点数、布尔值，和字符串。

```
double mult(double i = 5, int j = 6ull) { return i * j; }
```

对应的 Python 接口，

```
>>> mult()
30.0
>>> mult(j = 4, i = 1.2)
4.8
```

如果一个参数的默认值不被 CPP2PY 所支持，那它左侧参数的默认值也将被忽略。举个例子，

```
int divide(int a = 1, int* b = nullptr);
```

对应的 Python 接口中，参数 a 的默认值也将被忽略。

### 3.2.5 函数重载与重命名字典

函数重载是 C++ 的一大特色，但 Cython 只支持声明 C++ 重载函数，不支持重载 Python 函数。默认情况下，CPP2PY 只会选取一个受类型转换器支持的函数、方法和构造函数进行封装，而忽略它的重载版本。

为了解决这个问题，CPP2PY 引入了重命名字典的概念，用户可以以 API 参数的形式对函数进行重命名，比如，如果你有这样两个函数，

```
void foo(int c) { cout << c << endl; };
void foo(char *c) { cout << c << endl; };
```

默认情况下，CPP2PY 只会包装两个重载函数中的一个，但是通过重命名字典，就可以将它们区分开来。

```
renames_dict = {
    ("foo", "void(int)"): "foo_int",
    ("foo", "void(char*)"): "foo_str",
}
```

重命名字典的键为一个元组，元组的第一个元素为函数名，第二个元素为函数类型，对应的值为函数的新名称；元组也可以只由一个元素构成，在这种情况

下，CPP2PY 会重命名所有名称相匹配的函数。

```
>>> foo_int(3)
3
>>> foo_str("3")
3
```

重命名字典也可作用于类的方法，但不能作用于类的构造函数。

```
class A {
    void foo(int c);
    void foo(char *c);
}

/*
renames_dict = {
    ("A::foo", "void(int)"): "foo_int",
    ("A::foo", "void(char*)"): "foo_str",
}
*/
```

## 3.3 全局变量与宏

为了提供对 C/C++ 全局变量的访问，CPP2PY 创建了一个特殊的对象 `cvar`。例如，如下 C++ 全局变量，

```
const int My_variable = 5;
double density = 0.1;
```

在 Python 中的接口如下，

```
>>> cvar.My_variable
5
>>> cvar.density
0.1
>>> cvar.density = 0.9
```

给常量赋值，或赋的值类型与 C++ 变量类型不匹配，将引发异常。

```
>>> cvar.My_variable = 0
AttributeError: attribute 'My_variable' of
```

```
    'globals._cvar' objects is not writable
>>> cvar.density = "Hello"
TypeError: must be real number, not str
```

CPP2PY 能解析定义了基本数值类型和字符串字面量的宏，并将它们视作不带作用域的常量。

```
#define GET_LUCKY "We're up all night to get lucky"
```

如果我们不想使用 `cvar` 这个名字来访问 `GET_LUCKY`，可以通过命令行或 API 参数指定一个新的名称。

```
$ cpp2py [filename] --globals myvar

>>> myvar.GET_LUCKY
"We're up all night to get lucky"
```

即便模块中没有任何需要包装的客体，CPP2PY 也会创建 `cvar` 对象。

## 3.4 枚举

定义一组具有相关语义的常量时，枚举非常有用。典型的例子包括星期几（周日到周六）和课程成绩（“A”到“D”和“F”）。在 C++ 中，枚举实质上是在一定作用域下的整数常量。而在 Python 中，枚举类通过继承标准库中的枚举基类实现。

CPP2PY 将 C/C++ 中的枚举映射为 Python 枚举类，受支持的语法包括带作用域枚举，嵌套枚举等，唯一不受支持的语法是匿名枚举。对于如下 C++ 代码，

```
enum class Suit { Diamonds,
    Hearts,
    Clubs,
    Spades };
typedef enum { Hit,
    Miss } Result;

Result guess_card(Suit suit)
{
    if (suit == Suit::Clubs) {
        return Hit;
    }
    return Miss;
```

```
}
```

其对应的 Python 接口如下，

```
>>> Result.Miss
<Result.Miss: 1>
>>> guess_card(cppenum.Suit.Clubs)
<Result.Hit: 0>
```

## 3.5 类、结构体与联合

C++ 中的类、结构体和联合被包装在 Python 代理类中，一个代理类对象包括两部分，分别是包含着指向底层 C++ 对象指针的 Python 对象，以及底层的 C++ 对象。以下所有对类的讨论同样适用于结构体和联合。

CPP2PY 将 C++ 类中的方法、数据成员映射为 Python 代理类中的方法和属性，常量数据成员将被映射为不可变属性。

静态方法的语法在 Python 中得到了保留，但静态数据成员将被包装为全局变量。例如，对于下列声明，

```
class A {
public:
    static int count;
    static int get_count() { return count; }
};
```

在 Python 端以访问全局变量的方式访问类的静态数据成员，

```
>>> cvar.count = 5
>>> A.get_count()
5
```

CPP2PY 能够解析嵌套类，但是不支持匿名结构体和匿名联合。

### 3.5.1 类的构造

CPP2PY 会忽略掉 C++ 类的拷贝构造函数和移动构造函数。此外，构造函数的重载不被支持，在受类型转换器支持的所有构造函数中，只有一个会被包装为

代理类的构造函数。

根据 C++ 语法规则，如果一个类没有显式声明任何构造函数，CPP2PY 将自动为其添加一个默认构造函数。

有些类确实没有可用的构造函数，比如具有纯虚方法的抽象类，以及没有公开构造函数的类。在这种情况下，CPP2PY 将在 Python 端生成一个特殊的构造函数，在 Python 对象初始化时抛出异常。

```
class AbstractClass {
protected:
    AbstractClass() { }

public:
    virtual ~AbstractClass() { }
    virtual double square() = 0;
};
```

尝试在 Python 端构造抽象类，会引发异常。

```
>>> AbstractClass()
TypeError: Can't instantiate class AbstractClass
    for no available constructors.
```

### 3.5.2 继承

CPP2PY 处理继承的方式是将超类的公有方法和数据成员拷贝到派生类中。这种做法保证了生成代码的简洁，也能处理多重继承。但代价是类的继承关系将不能得到保持。以下列代码为例，

```
class Shape {
public:
    virtual ~Shape() {};
    virtual double area() = 0;
    virtual double perimeter() = 0;
};

class Circle : public Shape {
    int radius;

public:
    Circle(double radius) : radius(radius) {};
    double area();
    double perimeter();
};
```

```
class Square : public Shape {
    double size;

public:
    Square(double size) : size(size) {};
    double area();
    double perimeter();
};
```

对应 Python 代理类的继承关系将不能成立。

```
>>> issubclass(Circle, Shape), issubclass(Square, Shape)
(False, False)
```

在 C++ 中，常常用基类的指针包裹派生类对象，这种多态性在函数参数中仍能得到保持，这个特性是利用 Cython 的混合类型（Fused Type）语法实现的。

```
# void print_shape(Shape* shape);
>>> print_shape(Circle(radius=2))
area = 12.5664
perimeter = 12.5664
>>> print_shape(Square(size=2))
area = 4
perimeter = 8
```

但如果类似的情况出现在函数返回值中，CPP2PY 就无法处理，因为它不能确定对象的实际类型，而 Python 又没有将基类对象转换成派生类对象的语法。在这种情况下，用户需要根据需求修改 CPP2PY 生成的 Cython 代码。

对代理类来说，只有公有继承是有意义的，因为私有继承和保护继承不能带来新的公有成员。

CPP2PY 不支持以 `using` 声明将基类非公有成员公有化的语法。

```
class Derive : public Base {
public:
    using Base::foo;    // foo is private in Base
}
```

表 3.3　重载运算符的转换

| 类型 | C++ | Python |
|---|---|---|
| 数值运算 | `+`，`-`，`*`，`/`，`%` | `__add__`、`__sub__`、`__mul__`、`__truediv__`、`__mod__` |
| 就地数值运算 | `+=`，`-=`，`*=`，`/=`，`%=` | `__iadd__`、`__isub__`、`__imul__`、`__itruediv__`、`__imod__` |
| 位运算 | `&`，`\|`，`~`，`^`，`<<`，`>>` | `__and__`、`__or__`、`__invert__`、`__xor__`、`__lshift__`、`__rshift__` |
| 就地位运算 | `&=`，`\|=`，`^=`，`<<=`，`>>=` | `__iand__`、`__ior__`、`__ixor__`、`__ilshift__`、`__irshift__` |
| 比较运算 | `<`，`>`，`<=`，`>=`，`==`，`!=` | `__lt__`、`__gt__`、`__le__`、`__ge__`、`__eq__`、`__ne__` |
| 函数调用 | `()` | `__call__` |
| 其他运算符 | | 通过重命名字典指定 |

### 3.5.3　运算符重载

C++ 类的重载运算符将被尽可能的映射到 Python 中。Python 类的每个运算符都对应一个约定的魔法方法，魔法方法的特点是以双下划线开头和结尾。

然而，由于两种语言不同的语法设计，CPP2PY 不能自动处理某些运算符的包装。首先，Python 不允许重载逻辑运算符 `and(&&)`、`or(||)`、`not (!)` 和赋值运算符，这类算符将会被忽略；其次，C++ 下标运算符对应 Python 中的两个魔法方法，`__getitem__` 和 `__setitem__`，用户应该通过重命名字典进行指定。另外，就地运算符的语义差别也值得注意，在 C++ 中，`a += b` 表示 `a.operator+=(b)`，而 Python 在执行就地操作时还将执行一步额外的赋值操作，即 `a = a.operator+=(b)`。

重载运算符在 C++ 和 Python 间的转换关系如表 3.3 所示。CPP2PY 只能处理以类方法形式实现的运算符重载，以友元函数实现的运算符重载将会被忽略。

### 3.5.4　对象的所有权管理

C/C++ 和 Python 以两种截然不同的方式管理对象的生命周期。在 C++ 中，每一块的堆内存都需要被手动释放；而在 Python 中，对象的生命周期完全由垃圾回收器掌控，当一个对象的引用计数为 0 时，它占用的内存将被回收，具体的回收时机由垃圾回收器在运行时确定。

使用代理类时需要注意的一个主要问题是包装对象的内存管理。考虑下面的

C++ 代码，

```
class Foo {
        ...
};

class Spam {
public:
    Foo *value;
    ...
};
```

假如在 Python 中这样使用这些类，

```
f = Foo()
s = Spam()
s.value = f
g = s.value
g = 4
del f
```

现在，考虑由此产生的内存管理问题。创建代理类对象时，既创建了代理类实例，又创建了底层的 C++ 实例，f 和 s 都是如此。赋值语句 `s.value = f` 将 f 内部的指针存储到 s 底层的 C++ 对象中。`g = s.value` 创建了第二个 `Foo` 代理类对象，它也包裹了 f 的指针。现在，两个代理类对象和一个 C++ 对象共享了同一个指针。对 g 重新赋值，旧的 g 值引用计数为 0，这将导致一个代理类对象被回收；而紧接着 `del f` 又销毁了一个代理类对象。此时 `s.value` 仍然保存着 f 内部的指针，它是否还有效呢？

为了解决这个问题，CPP2PY 为代理类创建了 `owner` 属性来控制其对底层 C++ 对象的所有权。当代理类对象被析构时，只有 `owner` 属性为真时，底层的 C++ 对象才会被销毁。当一个新的代理类对象被创建时，`owner` 被设置为真。当对象的指针被返回时，它被一个代理类包装，如果该指针来自一个 `getter` 函数，那代理类对象将不具有所有权。

仍以刚才的代码为例，只要在析构 f 之前将其 `owner` 属性置为否，则 f 析构后 `s.value` 就仍是合法的。

```
>>> f = Foo(); s = Spam(); s.value = f; g = s.value
>>> g.owner
False
>>> f.owner
```

```
True
```

## 3.6 其它语法特性

### 3.6.1 命名空间

命名空间是 C++ 用来解决名称冲突问题的语法。一个标识符可在多个命名空间中定义，不同的命名空间的同名标识符互不相干。

CPP2PY 能解析 C++ 的命名空间。但是命名空间既不会出现在模块中，也不会导致模块被分解成子模块。如果你的程序存在多个命名空间，要小心名称冲突问题。函数和方法的名称冲突可以用重命名字典解决，而对于其它标识符的名称冲突问题，只能通过修改 C++ 源代码来手动解决。

### 3.6.2 类型别名

CPP2PY 支持解析以 `typedef`、宏和 `using` 定义的类型别名，但不支持模板别名，以下列接口为例，

```
typedef double mytype;
#define MYTYPE mytype
using mytype2 = const MYTYPE;

mytype2 fun(mytype d) { return d + 1.0; }
```

对应到 Python 中，

```
>> fun(2)
3.0
```

但需要注意的是，CPP2PY 不会将类型别名映射到 Python 端。

### 3.6.3 异常

C++ 和 Python 都支持异常的抛出和捕获，在跨语言项目中，统一的错误处理风格有助于编写更简洁有效的程序。

得益于 Cython，CPP2PY 支持将 C++ 异常传递到 Python 端，并将一部分 C++ 的内置异常类映射为 Python 的内置异常类，映射关系如表 3.4 所示。

表 3.4　异常在 C++ 和 Python 间的转换

| C++ | Python |
| --- | --- |
| `std::bad_alloc` | `MemoryError` |
| `std::bad_cast`、`std::bad_typeid` | `TypeError` |
| `std::domain_error`、`std::invalid_argument` | `ValueError` |
| `std::ios_base::failure` | `IOError` |
| `std::out_of_range` | `IndexError` |
| `std::overflow_error` | `OverflowError` |
| `std::range_error`、`std::underflow_error` | `ArithmeticError` |
| 其他异常 | `RuntimeError` |

### 3.6.4　兼容 C

CPP2PY 默认生成 C++ 风格接口，比如使用 `new` 而不是 `malloc` 来构造结构体实例，并将调用 C++ 编译器编译目标代码。而 C++ 编译器为了支持重载、命名空间等语法特性，会对函数和方法做名称修饰（name mangling），如果你的项目是以 C 语言编写的，这将在模块导入时引发链接错误。

有一种简单的方式来让 C 项目获得 CPP2PY 的支持，即在需要封装的接口前添加 `extern "C"` 链接声明。