Chapter 1

Section 1

Section 2

Chapter 2

Section 1

Section 2

Chapter 3

Section 1

Section 2

Chapter 4

Section 1

Section 2

Chapter 5

Section 1

Section 2

Chapter 6

Section 1

Section 2

Here is an inline example, $ \pi(\theta) $,

an equation,

$$\nabla f(x) \in \mathbb{R}^n,$$

and a regular $ symbol.

Define $f(x)$:

$$f(x) = x^2 x \in \R$$

```
graph TD
  A[ Anyone ] -->|Can help | B( Go to github.com/yuzutech/kroki )
  B --> C{ How to contribute? }
  C --> D[ Reporting bugs ]
  C --> E[ Sharing ideas ]
  C --> F[ Advocating ]
```

pybind11——无缝连接C+11和Python

pybind11是一个只有头文件的轻量级库，它在导出C++类型到Python的同时，也导出Python类型到C++中，其主要目的是建立现有C++代码的Python绑定。它与David Abrahams的Boost.Python库目的和语法相似，都是通过编译期内省来推断类型信息，以最大程度地降低传统扩展模块中的重复样板代码。

Boost.Python的问题主要在于Boost本身，这也是我创建一个类似项目的原因。Boost是一套庞大且复杂的工具库，它几乎兼容所有的C++编译器。但这种兼容性是有成本的：为了支持那些极其古老且充满BUG的编译器版本，Boost不得不使用各种晦涩难懂的模板技巧与变通方法。现在，支持C++11的编译器已经被广泛使用，这种沉重结构已成为一种过大且不必要的依赖。

你可以把pybind11库想象成Boost.Python的一个小型独立版本，其中所有与python绑定生成无关的内容都被删除了。不算注释，pybind11核心头文件大约只有4K行代码，并且它只依赖于Python（2.7或3.5+，或PyPy）和C ++标准库。由于C++11语言的新特性(特别是元组、lambda函数和可变参数模板），这种紧凑的实现才成为可能。自创建以来，这个库已经在很多方面超越了Boost.Python，多数常见情况下pybind11使得python绑定代码变得非常简单。

1.1 核心特性

pybind11可以将以下C++核心特性映射到Python：

- 函数入参和返回值可以是自定义数据结构的值、引用或者指针；
- 类成员方法和静态方法；
- 重载函数；
- 类成员变量和静态变量；
- 任意异常类型；
- 枚举；
- 回调函数；
- 迭代器和ranges；
- 自定义操作符；
- 单继承和多重继承；
- STL数据结构；
- 智能指针；
- Internal references with correct reference counting；

- 可以在Python中扩展带虚函数（和纯虚函数）的C++类;

1.2 好用的功能

除了上述核心功能外，pybind11还提供了一些好用的功能：

- 支持Python2.7, 3.5+, PyPy/PyPy3 7.3与实现无关的接口。
- 可以绑定带捕获参数的lambda函数，lambda捕获的数据存储生成的Python函数对象中。
- pybind11使用C++11移动构造函数和移动运算符，尽可能有效的转换自定义数据类型。(pybind11 uses C++11 move constructors and move assignment operators whenever possible to efficiently transfer custom data types.)
- 通过Python的buffer协议，可以很轻松地获取自定义类型的内存指针。这样，我们可以很方便地在C++矩阵类型（如Eigen）和NumPy之间快速转换，而无需昂贵的拷贝操作。
- pybind11可以自动将函数矢量化，以便它们透明地应用于以NumPy数组为参数的所有条目。
- 只需几行代码就可以支持Python基于切片的访问和赋值操作。
- 使用时只需要包含几个头文件即可，不用链接任何其他的库。
- 相比Boost.Python，生成的库文件更小，编译更快。
- 使用 `constexpr` 在编译器与计算函数签名，进一步减小了库文件大小。
- 可以轻松地让C++类型支持Python pickle和unpickle操作。

1.3 支持的编译器

1. Clang/LLVM 3.3以上 (Apple Xcode’s clang需要5.0.0以上版本)
2. GCC 4.8以上
3. Microsoft Visual Studio 2015 Update 3以上
4. Intel classic C++ compiler 18 or newer (ICC 20.2 tested in CI)
5. Cygwin/GCC (previously tested on 2.5.1)
6. NVCC (CUDA 11.0 tested in CI)
7. NVIDIA PGI (20.9 tested in CI)

1.4 □□

This project was created by Wenzel Jakob. Significant features and/or improvements to the code were contributed by Jonas Adler, Lori A. Burns, Sylvain Corlay, Eric Cousineau, Aaron Gokaslan, Ralf Grosse-Kunstleve, Trent Houliston, Axel Huebl, @hulucc, Yannick Jadoul, Sergey Lyskov Johan Mabille, Tomasz Miąsko, Dean Moldovan, Ben Pritchard, Jason Rhinelander, Boris Schäling, Pim Schellart, Henry Schreiner, Ivan Smirnov, Boris Staletic, and Patrick Stewart.

We thank Google for a generous financial contribution to the continuous integration infrastructure used by this project.

1.5 □□

See the contributing guide for information on building and contributing to pybind11.

1.6 License

pybind11 is provided under a BSD-style license that can be found in the LICENSE file. By using, distributing, or contributing to this project, you agree to the terms and conditions of this license.

改动日志

主要介绍了各个发布版本增加的功能、改进点，以及修复的BUG。暂时不翻译吧，有兴趣的可以看官方文档。

[illegible]

3. 安装说明

我们可以在pybind/pybind11 on GitHub获取到pybind11的源码。推荐pybind11开发者使用下面介绍的前三种方法之一，来获取pybind11。

3.1 以子模块的形式集成

当你的项目使用Git管理时，你可以将pybind11当做一个子模块嵌入到你的项目中。在你的git仓库，使用以下命令即可包含pybind11：

```
git submodule add -b stable ../../pybind/pybind11 extern/pybind11
git submodule update --init
```

这里假设你将项目的依赖放在了`extern`目录下，并且使用GitHub。如果你没有使用GitHub，可以使用完整的https或ssh URL来代替上面的相对URL`../../pybind/pybind11`。一些服务器可能需要`.git`扩展（GitHub不用）。

到这一步后，你可以直接include `extern/pybind11/include`目录即可。或者，你可以使用各种集成工具（见Build System一章)来包含pybind11。

3.2 通过**PyPI**来集成

你可以使用pip，通过PyPI来下载Pybind11的Python包，里面包含了源码已经CMake文件。像这样：

```
pip install pybind11
```

这样pybind11将以标准的Python包的形式提供。如果你想在root环境下直接使用pybind11，可以这样做：

```
pip install "pybind11[global]"
```

如果你使用系统自带的Python来安装，我们推荐在root环境下安装。这样会在 `/usr/local/include/pybind11` 和 `/usr/local/share/cmake/pybind11` 添加文件，除非你想这样。还是推荐你只在虚拟环境或你的 `pyproject.toml` 中使用。

3.3 通过conda-forge集成

You can use pybind11 with conda packaging via conda-forge:

```
conda install -c conda-forge pybind11
```

3.4 通过vcpkg集成

你可以通过Microsoft vcpkg依赖管理工具来下载和安装pybind11：

```
git clone https://github.com/Microsoft/vcpkg.git
cd vcpkg
./bootstrap-vcpkg.sh
./vcpkg integrate install
vcpkg install pybind11
```

3.5 通过brew全局安装

brew包管理（Homebrew on macOS, or Linuxbrew on Linux）有pybind11包。这样安装：

```
brew install pybind11
```

3.6 其他方法

Other locations you can find pybind11 are listed here; these are maintained by various packagers and the community.

4. 首次尝试（First steps）

本章将演示pybind11的基本特性。在开始前，请确保正确配置了编译pybind11测试用例的开发环境。

4.1 编译测试用例

Linux/macOS

在Linux上，你需要安装python-dev或python3-dev包和cmake。在macOS上，系统自带了所需的python版本，还需要安装cmake。

在安装好依赖项之后，运行下面的脚本：

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

脚本的最后一行将编译并运行测试用例。

Windows

在Windows上，需要支持C++11的Visual Studio版本（15及其以上）。

> Note：在Visual Studio 2017(MSVC 14.1)上使用C++17时，pybind11需要添加标识 `/permissive-` 来让编译器强制标准一致。在Visual Studio 2019上，不做强制要求，但同样建议添加。

使用以下命令编译和运行测试用例：

```
mkdir build
cd build
cmake ..
cmake --build . --config Release --target check
```

[illegible]Visual Studio[illegible]

> Note[illegible]Python[illegible]i386[illegible]x86_64[illegible]x86_64[illegible]vs[illegible] `cmake -A x64 ..`[illegible]

4.2 [illegible]

[illegible]

```
#include <pybind11/pybind11.h>
namespace py = pybind11;
```

[illegible]

4.3 [illegible]

[illegible]pybind11[illegible]

```
int add(int i, int j) {
    return i + j;
}
```

[illegible] `example.cpp` [illegible]

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

int add(int i, int j) {
    return i + j;
}

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

`PYBIND11_MODULE` 会创建一个函数，它在Python中使用 `import` 语句时被调用。宏的第一个参数是模块名（example），不使用引号包住；第二个参数是类型为 `py::module_` 的变量（m），它是创建绑定的主要接口。`module_::def()` 方法，则会生成add函数的Python绑定代码。

> Note：我们只需要少量的代码就可以将函数暴露给Python，函数入参和返回值相关的细节都由模板元编程自动推断。这种方式和语法是借用Boost.Python的，尽管底层实现完全不同。

pybind11是一个head-only库，它不需要链接任何库，也没有魔法般的中间转换步骤。在Linux上，示例可以使用下面的命令进行编译：

```
c++ -O3 -Wall -shared -std=c++11 -fPIC $(python3 -m pybind11 --includes)
example.cpp -o example$(python3-config --extension-suffix)
```

> Note：如果你使用子模块的方式包含pybind11代码，这里需要使用 `$(python3-config --includes) -Iextern/pybind11/include` 代替 `$(python3 -m pybind11 --includes)` 。原因在后续章节会解释。

如果需要更多有关于Linux和MacOS上所需编译标志的详细信息，请参阅手动构建章节。有关完整的跨平台编译说明，请参阅构建系统章节。

编译上面的C++代码后，我们会得到一个二进制模块文件，直接使用 `import` 导入模块到Python中。假设编译好的模块位于当前目录下，Python交互示例代码如下：

```
>>> import example
>>> example.add(1, 2)
3L
>>>
```

4.4 关键字参数

这里，我们对上面的C++代码做一点改造，就可以通知Python关于参数的名称（如本例中的“i”和“j”）。

```
m.def("add", &add, "A function which adds two numbers",
      py::arg("i"), py::arg("j"));
```

arg是可用于将元数据传递到module::def()的几个特殊标记类之一。使用上面修改后的代码，我们可以在调用函数时使用关键字参数，以增加代码可读性，特别是对那些带有多个参数的函数。

```
import example
example.add(i=1, j=2)  #3L
```

关键字名称也会在文档的函数签名中显示：

```
>>> help(example)

....

FUNCTIONS
    add(...)
        Signature : (i: int, j: int) -> int

        A function which adds two numbers
```

还可以使用更加简短的方式给参数命名：

```
// regular notation
m.def("add1", &add, py::arg("i"), py::arg("j"));
// shorthand
using namespace pybind11::literals;
m.def("add2", &add, "i"_a, "j"_a);
```

后缀 `_a` 会生成一个等价于 `arg` 方法的字面量。使用这个后缀时，需要调用 `using namespace pybind11::literals` 来声明后缀所在的命名空间。这样除了 `literals` 外，不会从pybind11命名空间引入其他不必要的东西。

4.5 默认参数

现在我们需要绑定一个带默认参数的函数：

```
int add(int i = 1, int j = 2) {
    return i + j;
}
```

pybind11不能自动地提取默认参数，因为它不属于函数类型信息的一部分。我们需要借助 `arg` 来实现这一功能：

```
m.def("add", &add, "A function which adds two numbers",
      py::arg("i") = 1, py::arg("j") = 2);
```

默认值同样也会在文档中展示：

```
>>> help(example)

....

FUNCTIONS
    add(...)
        Signature : (i: int = 1, j: int = 2) -> int

        A function which adds two numbers
```

更简短的声明方式：

```
// regular notation
m.def("add1", &add, py::arg("i") = 1, py::arg("j") = 2);
// shorthand
m.def("add2", &add, "i"_a=1, "j"_a=2);
```

4.6 导出变量

我们可以使用 `attr` 函数来注册需要导出到Python模块中的C++变量。内建类型和常规对象（后面会细讲）会在指定attriutes时自动转换，也可以使用 `py::cast` 来显式转换。

~~~
PYBIND11_MODULE(example, m) {
    m.attr("the_answer") = 42;
    py::object world = py::cast("World");
    m.attr("what") = world;
}
``


Python⊠⊠⊠⊠⊠⊠
```pyhton
>>> import example
>>> example.the_answer
42
>>> example.what
'World'
~~~

4.7 支持的数据类型

原生支持大量数据类型，完美适用于函数参数，参数值通常直接返回或者经过py::cast处理再返回。有关完整概述，请参阅类型转换部分。(A large number of data types are supported out of the box and can be used seamlessly as functions arguments, return values or with py::cast in general. For a full overview, see the Type conversions section.)

5. [illegible]

5.1 [illegible]

[illegible]C++[illegible] `Pet` [illegible]

```
struct Pet {
    Pet(const std::string &name) : name(name) { }
    void setName(const std::string &name_) { name = name_; }
    const std::string &getName() const { return name; }

    std::string name;
};
```

[illegible]

```
#include <pybind11/pybind11.h>
namespace py = pybind11;

PYBIND11_MODULE(example, m) {
    py::class_<Pet>(m, "Pet")
        .def(py::init<const std::string &>())
        .def("setName", &Pet::setName)
        .def("getName", &Pet::getName);
}
```

- `class_` [illegible]C++ class[illegible] struct[illegible] `init()` [illegible]Python[illegible]

```
>>> import example
>>> p = example.Pet("Molly")
>>> print(p)
<example.Pet object at 0x10cd98060>
>>> p.getName()
u'Molly'
>>> p.setName("Charly")
>>> p.getName()
u'Charly'
```

> **See also**：静态成员函数需要使用 `class_::def_static` 来绑定。

5.2 关键字参数和默认参数

可以使用第4章讨论的语法来指定关键字和默认参数，详见第4章相关章节。

5.3 绑定匿名函数

使用 `print(p)` 打印对象信息时，上面的例子会得到一些基本无用的信息。

```
>>> print(p)
<example.Pet object at 0x10cd98060>
```

我们可以绑定一个工具函数到 `__repr__` 方法，来返回可读性好的摘要信息。在不改变Pet类的基础上，使用一个匿名函数来完成这个功能是一个不错的选择。

```
py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string &>())
    .def("setName", &Pet::setName)
    .def("getName", &Pet::getName)
    .def("__repr__",
        [](const Pet &a) {
            return "<example.Pet named '" + a.name + "'>";
        });
```

通过上面的修改，Python中的输出如下：

```
>>> print(p)
<example.Pet named 'Molly'>
```

pybind11支持无状态和有状态的lambda闭包，即lambda表达式的 `[]` 是否带捕获参数。

5.4 成员变量

使用 `class_::def_readwrite` 方法可以导出公有成员变量，使用 `class_::def_readonly` 方法则可以导出只读成员。

```
py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string &>())
    .def_readwrite("name", &Pet::name)
    // ... remainder ...
```

Python中使用示例如下：

```
>>> p = example.Pet("Molly")
>>> p.name
u'Molly'
>>> p.name = "Charly"
>>> p.name
u'Charly'
```

假设 `Pet::name` 是一个私有成员变量，向外提供setter和getters方法。

```
class Pet {
public:
    Pet(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;
};
```

可以使用 `class_::def_property()`(只读成员使用 `class_::def_property_readonly()`)来定义并私有成员，并生成相应的setter和geter方法：

```
py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string &>())
    .def_property("name", &Pet::getName, &Pet::setName)
    // ... remainder ...
```

只写属性通过将read函数定义为nullptr来实现。

> **see also**: 相似的方法 `class_::def_readwrite_static()`, `class_::def_readonly_static()` `class_::def_property_static()`, `class_::def_property_readonly_static()` 用于绑定静态变量和属性。

5.5 动态属性

原生的Pyhton类可以动态地获取新属性：

```
>>> class Pet:
...     name = "Molly"
...
>>> p = Pet()
>>> p.name = "Charly"  # overwrite existing
>>> p.age = 2  # dynamically add a new attribute
```

默认情况下，从C++导出的类不支持动态属性，其可写属性必须是通过 `class_::def_readwrite` 或 `class_::def_property` 定义的。试图设置其他属性将产生错误：

```
>>> p = example.Pet()
>>> p.name = "Charly"  # OK, attribute defined in C++
>>> p.age = 2  # fail
AttributeError: 'Pet' object has no attribute 'age'
```

要让C++类也支持动态属性，我们需要在 `py::class_` 的构造函数添加 `py::dynamic_attr` 标识：

```
py::class_<Pet>(m, "Pet", py::dynamic_attr())
    .def(py::init<>())
    .def_readwrite("name", &Pet::name);
```

这样，之前报错的代码就能够正常运行了。

```
>>> p = example.Pet()
>>> p.name = "Charly"  # OK, overwrite value in C++
>>> p.age = 2  # OK, dynamically add a new attribute
>>> p.__dict__  # just like a native Python class
{'age': 2}
```

需要提醒一下，支持动态属性会带来小小的运行时开销。不仅仅因为增加了额外的 `__dict__` 属性，还因为处理循环引用时需要花费更多的垃圾收集跟踪花销。但是不必担心这个问题，因为原生Python类也有同样的开销。默认情况下，pybind11导出的类比原生Python类效率更高，使能动态属性也只是让它们处于同等水平而已。

5.6 继承与向下转型

现在有两个具有继承关系的类：

```
struct Pet {
    Pet(const std::string &name) : name(name) { }
    std::string name;
};

struct Dog : Pet {
    Dog(const std::string &name) : Pet(name) { }
    std::string bark() const { return "woof!"; }
};
```

pybind11提供了两种方法来指明继承关系：1）将C++基类作为派生类 `class_` 的模板参数；2）将基类名作为 `class_` 的参数绑定到派生类。两种方法是等效的。

```
py::class_<Pet>(m, "Pet")
   .def(py::init<const std::string &>())
   .def_readwrite("name", &Pet::name);

// Method 1: template parameter:
py::class_<Dog, Pet /* <- specify C++ parent type */>(m, "Dog")
    .def(py::init<const std::string &>())
    .def("bark", &Dog::bark);

// Method 2: pass parent class_ object:
py::class_<Dog>(m, "Dog", pet /* <- specify Python parent type */)
    .def(py::init<const std::string &>())
    .def("bark", &Dog::bark);
```

指明继承关系后，派生类实例将获得两者的字段和方法：

```
>>> p = example.Dog("Molly")
>>> p.name
u'Molly'
>>> p.bark()
u'woof!'
```

上面的例子是一个常规非多态的继承关系，表现在Python就是：

```
// ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
m.def("pet_store", []() { return std::unique_ptr<Pet>(new Dog("Molly")); });
```

```
>>> p = example.pet_store()
>>> type(p)  # `Dog` instance behind `Pet` pointer
Pet          # no pointer downcasting for regular non-polymorphic types
>>> p.bark()
AttributeError: 'Pet' object has no attribute 'bark'
```

`pet_store`函数返回了一个Dog实例，但由于基类并非多态类型，Python只识别到了Pet。在C++中，一个类至少有一个虚函数才会被视为多态类型。pybind11会自动识别这种多态机制。

```
struct PolymorphicPet {
    virtual ~PolymorphicPet() = default;
};

struct PolymorphicDog : PolymorphicPet {
    std::string bark() const { return "woof!"; }
};

// Same binding code
py::class_<PolymorphicPet>(m, "PolymorphicPet");
py::class_<PolymorphicDog, PolymorphicPet>(m, "PolymorphicDog")
    .def(py::init<>())
    .def("bark", &PolymorphicDog::bark);

// Again, return a base pointer to a derived instance
m.def("pet_store2", []() { return std::unique_ptr<PolymorphicPet>(new
PolymorphicDog); });
```

```
>>> p = example.pet_store2()
>>> type(p)
PolymorphicDog  # automatically downcast
>>> p.bark()
u'woof!'
```

pybind11会自动地将一个指向多态基类的指针，向下转型为实际的派生类类型。这和C++常见的情况不同，我们不仅可以访问基类的虚函数，还能获取到通过基类看不到的，具体的派生类的方法和属性。

5.7 重载方法

重载方法即拥有相同的函数名，但入参不一样的函数：

```
struct Pet {
    Pet(const std::string &name, int age) : name(name), age(age) { }

    void set(int age_) { age = age_; }
    void set(const std::string &name_) { name = name_; }

    std::string name;
    int age;
};
```

我们在绑定`Pet::set`时会报错，因为编译器并不知道用户想选择哪个重载方法。我们需要添加具体的函数指针来消除歧义。绑定多个函数到同一个Python名称，将会自动创建函数重载链。Python将会依次匹配，找到最合适的重载函数。

```
py::class_<Pet>(m, "Pet")
   .def(py::init<const std::string &, int>())
   .def("set", static_cast<void (Pet::*)(int)>(&Pet::set), "Set the pet's
age")
   .def("set", static_cast<void (Pet::*)(const std::string &)>(&Pet::set),
"Set the pet's name");
```

在函数的文档描述中，我们可以看见重载的函数签名：

```
>>> help(example.Pet)

class Pet(__builtin__.object)
 |  Methods defined here:
 |
 |  __init__(...)
 |      Signature : (Pet, str, int) -> NoneType
 |
 |  set(...)
 |      1. Signature : (Pet, int) -> NoneType
 |
 |      Set the pet's age
 |
 |      2. Signature : (Pet, str) -> NoneType
 |
 |      Set the pet's name
```

如果你的编译器支持C++14，也可以使用下面的语法来转换重载函数：

```
py::class_<Pet>(m, "Pet")
    .def("set", py::overload_cast<int>(&Pet::set), "Set the pet's age")
    .def("set", py::overload_cast<const std::string &>(&Pet::set), "Set the
pet's name");
```

这里，`py::overload_cast`仅需指定函数类型，不用给出返回值类型，以避免原语法带来的不必要的干扰(`void (Pet::*)`)。如果是基于const的重载，需要使用`py::const`标识。

```
struct Widget {
    int foo(int x, float y);
    int foo(int x, float y) const;
};

py::class_<Widget>(m, "Widget")
   .def("foo_mutable", py::overload_cast<int, float>(&Widget::foo))
   .def("foo_const",   py::overload_cast<int, float>(&Widget::foo,
py::const_));
```

如果你想在仅支持c++11的编译器上使用`py::overload_cast`语法，可以使用`py::detail::overload_cast_impl`来代替：

```
template <typename... Args>
using overload_cast_ = pybind11::detail::overload_cast_impl<Args...>;

py::class_<Pet>(m, "Pet")
    .def("set", overload_cast_<int>()(&Pet::set), "Set the pet's age")
    .def("set", overload_cast_<const std::string &>()(&Pet::set), "Set the
pet's name");
```

> Note: 如果想定义多个重载的构造函数，使用`.def(py::init<...>())`语法依次定义就好，指定关键字和默认参数的机制也还是生效的。

5.8 枚举和内部类型

现在有一个含有枚举和内部类型的类：

```
struct Pet {
    enum Kind {
        Dog = 0,
        Cat
    };

    struct Attributes {
        float age = 0;
    };

    Pet(const std::string &name, Kind type) : name(name), type(type) { }

    std::string name;
    Kind type;
    Attributes attr;
};
```

绑定代码如下所示：

```
py::class_<Pet> pet(m, "Pet");

pet.def(py::init<const std::string &, Pet::Kind>())
    .def_readwrite("name", &Pet::name)
    .def_readwrite("type", &Pet::type)
    .def_readwrite("attr", &Pet::attr);

py::enum_<Pet::Kind>(pet, "Kind")
    .value("Dog", Pet::Kind::Dog)
    .value("Cat", Pet::Kind::Cat)
    .export_values();

py::class_<Pet::Attributes> attributes(pet, "Attributes")
    .def(py::init<>())
    .def_readwrite("age", &Pet::Attributes::age);
```

为确保嵌套类型 `Kind` 和 `Attributes` 在 `Pet` 的作用域中创建，我们必须向 `enum_` 和 `class_` 的构造函数提供 `Pet` `class_` 实例。 `enum_::export_values()` 用来导出枚举项到父作用域，C++11 的强枚举类型需要跳过这点。

```
>>> p = Pet("Lucy", Pet.Cat)
>>> p.type
Kind.Cat
>>> int(p.type)
1L
```

枚举类型的枚举项会被导出到类 `__members__` 属性中：

```
>>> Pet.Kind.__members__
{'Dog': Kind.Dog, 'Cat': Kind.Cat}
```

`name` 属性可以返回枚举值的名称的unicode字符串，`str(enum)` 也可以做到，但两者的实现目标不同。下面的例子展示了两者的差异：

```
>>> p = Pet("Lucy", Pet.Cat)
>>> pet_type = p.type
>>> pet_type
Pet.Cat
>>> str(pet_type)
'Pet.Cat'
>>> pet_type.name
'Cat'
```

> Note: 当我们给 `enum_` 的构造函数增加 `py::arithmetic()` 标识时，pybind11将创建一个支持基本算术运算和位运算（如比较、或、异或、取反等）的枚举类型。
>
> ```
> py::enum_<Pet::Kind>(pet, "Kind", py::arithmetic())
>    ...
> ```
>
> 默认情况下，省略这些可以节省内存空间。

6. 构建系统

后续再翻译。

7. [illegible]

[illegible]4[illegible]5[illegible]Python[illegible]

7.1 [illegible]

Python[illegible]C++[illegible]no-trivial[illegible]Python[illegible]C++[illegible]pybind11[illegible]`model::def()`[illegible]`class_def()`[illegible]`return_value_policy::automatic`[illegible]

[illegible]

```
/* Function declaration */
Data *get_data() { return _data; /* (pointer to a static data structure) */ }
...

/* Binding code */
m.def("get_data", &get_data); // <-- KABOOM, will cause crash when called from Python
```

[illegible]Python[illegible]`get_data()`[illegible]C++[illegible]Python[illegible]`return_value_policy::automatic`[illegible]pybind11[illegible]`_data`[illegible]

[illegible]Python[illegible]`_data`[illegible]Python[illegible]pybind11[illegible]C++[illegible]operator delete()[illegible]

[illegible]`return_value_policy::reference`[illegible]

```
m.def("get_data", &get_data, py::return_value_policy::reference);
```

[illegible]pybind11[illegible]

[illegible]	[illegible]
`return_value_policy::take_ownership`	[illegible]0[illegible]Pyhton[illegible]delete[illegible]
`return_value_policy::copy`	[illegible]Python[illegible]
`return_value_policy::move`	[illegible] `std::move` [illegible]Python[illegible]
`return_value_policy::reference`	[illegible]C++[illegible]Pyth [illegible]C++[illegible]
`return_value_policy::reference_internal`	[illegible] `this` [illegible] `self` [illegible]referenc [illegible] `keep_alive<0, 1>` [illegible]Python[illegible] `def_property` [illegible] `def_readwrite` [illegible]getter[illegible]
`return_value_policy::automatic`	[illegible] `return_value_policy::take_owner` [illegible] `return_value_policy::copy` [illegible] `py::cla` [illegible]
`return_value_policy::automatic_reference`	[illegible] `return_value_policy::reference` [illegible]C++[illegible]Python[illegible] `pybind11/stl.h` [illegible]casters[illegible]

[illegible]

```
class_<MyClass>(m, "MyClass")
    .def_property("data", &MyClass::getData, &MyClass::setData,
                  py::return_value_policy::copy);
```

在技术层面，上述代码会将策略同时应用于getter和setter函数，但是setter函数并不关心返回值策略，这样做仅仅出于语法简洁的考虑。或者，你可以通过`cpp_function`构造函数来传递目标参数：

```
class_<MyClass>(m, "MyClass")
    .def_property("data"
        py::cpp_function(&MyClass::getData, py::return_value_policy::copy),
        py::cpp_function(&MyClass::setData)
    );
```

注意：代码使用无效的返回值策略将导致未初始化内存或多次free数据结构，这将导致难以调试的、不确定的问题和段错误。因此，花点时间来理解上面表格的各个选项是值得的。

提示：

1. 上述策略的另一个重点是，他们仅可以应用于pybind11还不知晓的实例，这时策略将澄清返回值的生命周期和所有权问题。当pybind11已经知晓参数（通过其在内存中的类型和地址来识别），它将返回已存在的Python对象封装，而不是创建一份拷贝。
2. 下一节将讨论上面表格之外的调用策略，他涉及到返回值和函数参数的引用关系。
3. 可以考虑使用智能指针来代替复杂的调用策略和生命周期管理逻辑。智能指针会告诉你一个对象是否仍被C++或Python引用，这样就可以消除各种可能引发crash或未定义行为的矛盾。对于返回智能指针的函数，没必要指定返回值策略。

7.2 附加的调用策略

除了以上的返回值策略外，进一步指定调用策略可以表明参数间的依赖关系，确保函数调用的稳定性。

保活（keep alive）

当一个C++容器对象包含另一个C++对象时，我们需要使用该策略。`keep_alive<Nurse, Patient>`表明至少在索引Nurse被回收前，索引Patient应该被保活。0表示返回值，1及以上表示参数索引。1表示隐含的参数this指针，而常规参数索引从2开始。当Nurse的值在运行前被检测到为None时，调用策略将什么都不做。

当nurse不是一个pybind11注册类型时，实现依赖于创建对nurse对象弱引用的能力。如果nurse对象不是pybind11注册类型，也不支持弱引用，程序将会抛出异常。

如果你使用一个错误的参数索引，程序将会抛出“Could not cativate keep_alive!“警告的运行时异常。这时，你应该review你代码中使用的索引。

参见下面的例子：一个list append操作，将新添加元素的生命周期绑定到添加的容器对象上：

```
py::class_<List>(m, "List").def("append", &List::append, py::keep_alive<1, 2>
());
```

为了一致性，构造函数的实参索引也是相同的。索引1仍表示this指针，索引0表示返回值（构造函数的返回值被认为是void）。下面的示例将构造函数入参的生命周期绑定到被构造对象上。

```
py::class_<Nurse>(m, "Nurse").def(py::init<Patient &>(), py::keep_alive<1, 2>
());
```

> Note: `keep_alive` 与Boost.Python中的 `with_custodian_and_ward` 和 `with_custodian_and_ward_postcall` 相似。

Call guard

`call_guard<T>` 策略允许任意T类型的scope guard应用于整个函数调用。示例如下：

```
m.def("foo", foo, py::call_guard<T>());
```

上面的代码等价于：

```
m.def("foo", [](args...) {
    T scope_guard;
    return foo(args...); // forwarded arguments
});
```

仅要求模板参数T是可构造的，如 `gil_scoped_release` 就是一个非常有用的类型。

`call_guard` 支持同时制定多个模板参数，`call_guard<T1, T2, T3 ...>` 。构造顺序是从左至右，析构顺序则相反。

> See also: `test/test_call_policies.cpp` 含有更丰富的示例来展示 `keep_alive` 和 `call_guard` 的用法。

7.3 以**Python**对象作为参数

pybind11通过简单的C++封装类，公开了绝大多数Python类型。这些封装类也可以在绑定代码宏作为函数参数使用，这样我们就可以在C++侧使用原生的python类型。举个遍历Python dict的例子：

```
void print_dict(const py::dict& dict) {
    /* Easily interact with Python types */
    for (auto item : dict)
        std::cout << "key=" << std::string(py::str(item.first)) << ", "
                  << "value=" << std::string(py::str(item.second)) <<
std::endl;
}

// it can be exported as follow:
m.def("print_dict", &print_dict);
```

在Python中使用如下：

```
>>> print_dict({"foo": 123, "bar": "hello"})
key=foo, value=123
key=bar, value=hello
```

7.4 接收***args**和****kwatgs**参数

Python的函数可以接收任意数量的参数和关键字参数：

```
def generic(*args, **kwargs):
    ...  # do something with args and kwargs
```

我们也可以通过pybind11来创建这样的函数：

```
void generic(py::args args, const py::kwargs& kwargs) {
    /// .. do something with args
    if (kwargs)
        /// .. do something with kwargs
}

/// Binding code
m.def("generic", &generic);
```

`py::args`继承自`py::tuple`，`py::kwargs`继承自`py::dict`。

更多示例参考`test/test_kwargs_and_defualts.cpp`。

7.5 再探默认参数

前面的章节已经讨论了默认参数的基本用法。关于实现有一个值得关注的点，就是默认参数在声明时就被转换为Python对象了。看看下面的例子：

```
py::class_<MyClass>("MyClass").def("myFunction", py::arg("arg") =
SomeType(123));
```

这个例子里，必须保证SomeType类型已经被binding了（通过py::class_），不然就会抛出异常。

另一个值得注意的事情就是，生成的函数签名将使用对象的`__repr__`方法来处理默认参数值。如果对象没有提供该方法，那么函数签名将不能直观的看出默认参数值。

```
FUNCTIONS

|  myFunction(...)
|      Signature : (MyClass, arg : SomeType = <SomeType object at
0x101b7b080>) -> NoneType
```

要处理这个问题，我们需要定义`SomeType.__repr__`方法，或者使用`arg_v`给默认参数手动添加方便阅读的注释。

~~~~
py::class_<MyClass>("MyClass")
    .def("myFunction", py::arg_v("arg", SomeType(123), "SomeType(123)"));
``


☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒
```c++
py::class_<MyClass>("MyClass")
    .def("myFunction", py::arg("arg") = static_cast<SomeType *>(nullptr));
~~~~

7.6 Keyword-only参数

Python3提供了keyword-only参数（在函数定义中使用 `*` 作为匿名参数）：

```
def f(a, *, b):  # a can be positional or via keyword; b must be via keyword
    pass

f(a=1, b=2)  # good
f(b=2, a=1)  # good
f(1, b=2)  # good
f(1, 2)  # TypeError: f() takes 1 positional argument but 2 were given
```

pybind11提供了 `py::kw_only` 对象来实现相同的功能：

```
m.def("f", [](int a, int b) { /* ... */ },
      py::arg("a"), py::kw_only(), py::arg("b"));
```

注意，该特性不能与 `py::args` 一起使用。

7.7 Positional-only参数

python3.8引入了Positional-only参数语法，pybind11通过 `py::pos_only()` 来提供相同的功能：

```
m.def("f", [](int a, int b) { /* ... */ },
       py::arg("a"), py::pos_only(), py::arg("b"));
```

现在，你不能通过关键字来给定 `a` 参数。该特性可以和keyword-only参数一起使用。

7.8 Non-converting参数

有些参数可能支持类型转换，如：

- 通过 `py::implicitly_convertible<A,B>()` 进行隐式转换
- 将整形变量传给入参为浮点类型的函数
- 将非复数类型（如float）传给入参为 `std::complex<float>` 类型的函数
- Calling a function taking an Eigen matrix reference with a numpy array of the wrong type or of an incompatible data layout.

有时这种转换并不是我们期望的，我们可能更希望绑定代码抛出错误，而不是转换参数。通过 `py::arg` 来调用 `.noconvert()` 方法可以实现这个事情。

```
m.def("floats_only", [](double f) { return 0.5 * f; },
py::arg("f").noconvert());
m.def("floats_preferred", [](double f) { return 0.5 * f; }, py::arg("f"));
```

尝试进行转换时，将抛出 `TypeError` 异常：

```
>>> floats_preferred(4)
2.0
>>> floats_only(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: floats_only(): incompatible function arguments. The following
argument types are supported:
    1. (f: float) -> float

Invoked with: 4
```

该方法可以与缩写符号 `_a` 和默认参数配合使用，像这样 `py::arg().noconvert()` 。

7.9 允许/禁止空参数

当函数接受由 `py::class_` 注册的C++类型的指针或shared holder(如指针指针等)，pybind11允许将Python的None传递给函数，等同于C++中传递nullptr给函数。

我们可以使用 `py::arg` 对象的 `.none` 方法来显式地使能或禁止该行为。

```
py::class_<Dog>(m, "Dog").def(py::init<>());
py::class_<Cat>(m, "Cat").def(py::init<>());
m.def("bark", [](Dog *dog) -> std::string {
    if (dog) return "woof!"; /* Called with a Dog instance */
    else return "(no dog)"; /* Called with None, dog == nullptr */
}, py::arg("dog").none(true));
m.def("meow", [](Cat *cat) -> std::string {
    // Can't be called with None argument
    return "meow";
}, py::arg("cat").none(false));
```

这样，Python调用 `bark(None)` 将返回 `"(no dog)"` ，调用 `meow(None)` 将抛出异常 `TypeError` 。

```
>>> from animals import Dog, Cat, bark, meow
>>> bark(Dog())
'woof!'
>>> meow(Cat())
'meow'
>>> bark(None)
'(no dog)'
>>> meow(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: meow(): incompatible function arguments. The following argument
types are supported:
    1. (cat: animals.Cat) -> str

Invoked with: None
```

在不显式指定的情况下，默认支持传递 `None` 。

Note: Even when `.none(true)` is specified for an argument, `None` will be converted to a `nullptr` *only* for custom and opaque types. Pointers to built-in types (`double *`, `int *`, …) and STL types (`std::vector<T> *`, …; if `pybind11/stl.h` is included) are copied when converted to C++ (see Overview) and will not allow `None` as argument. To pass optional argument of these copied types consider using `std::optional<T>`

7.10 重载解析顺序

当一个函数或者方法拥有多个重载时，pybind11 通过两个步骤来决定重载调用的次序。第一步尝试不做类型匹配各个重载函数。如果没有匹配到，第二步将允许类型转换再匹配一次（显示调用 `py::arg().noconvert()` 禁用类型转换的函数除外）。

如果两步都失败了，将抛出异常 `TypeError` 。

在上述两步中，重载函数将以 pybind11 中注册的顺序依次遍历。如果函数定义中增加了 `py::prepend()` 的标识，该重载函数将最先被遍历。

Note: pybind11 不会根据重载参数的数量或类型来排优先级。换言之，pybind11 不会将仅需一次类型转换的函数排在需要三次转换的函数前面，仅仅会将不需要类型转换的重载函数排在至少需要一次类型转换的函数前面。

8. 类

本章将在第五章的基础上，进一步讲解类的绑定方法。

8.1 在Python中重载虚函数

假设有一个含有虚函数的C++类或接口，我们想在Python中重载虚函数。

```cpp
class Animal {
public:
    virtual ~Animal() { }
    virtual std::string go(int n_times) = 0;
};

class Dog : public Animal {
public:
    std::string go(int n_times) override {
        std::string result;
        for (int i=0; i<n_times; ++i)
            result += "woof! ";
        return result;
    }
};
```

现在有一个普通函数，它调用任意Animal实例的`go()`函数。

```cpp
std::string call_go(Animal *animal) {
    return animal->go(3);
}
```

pybind11绑定代码如下:

```
PYBIND11_MODULE(example, m) {
    py::class_<Animal>(m, "Animal")
        .def("go", &Animal::go);

    py::class_<Dog, Animal>(m, "Dog")
        .def(py::init<>());

    m.def("call_go", &call_go);
}
```

但是，这样绑定不可扩展，当我们尝试继承Animal类时会提示“No constructor defined!“，因为Animal无法构造。这时，我们需要类似于“跳板(trampoline)“的工具来重定向虚函数调用到Python中。

我们可以在Python中定义一个新的Animal类作为辅助跳板：

```
class PyAnimal : public Animal {
public:
    /* Inherit the constructors */
    using Animal::Animal;

    /* Trampoline (need one for each virtual function) */
    std::string go(int n_times) override {
        PYBIND11_OVERRIDE_PURE(
            std::string, /* Return type */
            Animal,      /* Parent class */
            go,          /* Name of function in C++ (must match Python name)
*/
            n_times      /* Argument(s) */
        );
    }
};
```

定义纯虚函数时需要使用 `PYBIND11_OVERRIDE_PURE` 宏，而有默认实现的虚函数则使用 `PYBIND11_OVERRIDE` 。 `PYBIND11_OVERRIDE_PURE_NAME` 和 `PYBIND11_OVERRIDE_NAME` 宏的功能类似，主要用于C函数名和Python函数名不一致的时候。以 `__str__` 为例：

```
std::string toString() override {
  PYBIND11_OVERRIDE_NAME(
      std::string, // Return type (ret_type)
      Animal,      // Parent class (cname)
      "__str__",   // Name of method in Python (name)
      toString,    // Name of function in C++ (fn)
  );
}
```

Animal类的绑定代码也需要一些微调：

```
PYBIND11_MODULE(example, m) {
    py::class_<Animal, PyAnimal /* <--- trampoline*/>(m, "Animal")
        .def(py::init<>())
        .def("go", &Animal::go);

    py::class_<Dog, Animal>(m, "Dog")
        .def(py::init<>());

    m.def("call_go", &call_go);
}
```

pybind11通过向`class_`指定额外的模板参数PyAnimal，让我们可以在Python中继承Animal类。

接下来，我们可以像往常一样定义构造函数。绑定时我们需要使用真实类，而不是辅助类。

```
py::class_<Animal, PyAnimal /* <--- trampoline*/>(m, "Animal");
    .def(py::init<>())
    .def("go", &PyAnimal::go); /* <--- THIS IS WRONG, use &Animal::go */
```

但是，上面的改动可以让我们在Python中继承Animal类，而不能继承Dog类。后续章节将会在此基础上进一步改进。

下面的Python代码展示了我们继承并重载了`Animal::go`方法，并通过虚函数来调用它：

```
from example import *
d = Dog()
call_go(d)     # u'woof! woof! woof! '
class Cat(Animal):
    def go(self, n_times):
        return "meow! " * n_times

c = Cat()
call_go(c)   # u'meow! meow! meow! '
```

如果你在派生的Python类中自定义了一个构造函数，你必须保证显示调用C++构造函数(通过 `__init__`)，不管它是否为默认构造函数。否则，实例属于C++那部分的内存就未初始化，可能导致未定义行为。在pybind11 2.6版本中，这种错误将会抛出 `TypeError` 异常。

```
class Dachshund(Dog):
    def __init__(self, name):
        Dog.__init__(self)  # Without this, a TypeError is raised.
        self.name = name

    def bark(self):
        return "yap!"
```

注意必须显式地调用 `__init__` ，而不应该使用 `supper()` 。在一些简单的线性继承中， `supper()` 或许可以正常工作；一旦你混合Python和C++类使用多重继承，由于Python MRO和C++的机制，一切都将崩溃。

Note:

当重载函数返回一个pybind11从Python中转换过来的类型的引用或指针时，有些限制条件需要注意下：

- because in these cases there is no C++ variable to reference (the value is stored in the referenced Python variable), pybind11 provides one in the PYBIND11_OVERRIDE macros (when needed) with static storage duration. Note that this means that invoking the overridden method on *any* instance will change the referenced value stored in *all* instances of that type.
- Attempts to modify a non-const reference will not have the desired effect: it will change only the static cache variable, but this change will not

propagate to underlying Python instance, and the change will be replaced the next time the override is invoked.

8.2 虚函数与继承

综合考虑虚函数与继承时，你需要为每个你允许在Python派生类中重载的方法提供重载方式。下面我们扩展Animal和Dog来举例：

```cpp
class Animal {
public:
    virtual std::string go(int n_times) = 0;
    virtual std::string name() { return "unknown"; }
};
class Dog : public Animal {
public:
    std::string go(int n_times) override {
        std::string result;
        for (int i=0; i<n_times; ++i)
            result += bark() + " ";
        return result;
    }
    virtual std::string bark() { return "woof!"; }
};
```

上节涉及到的Animal辅助类仍是必须的，为了让Python代码能够继承 `Dog` 类，我们也需要为 `Dog` 类增加一个跳板类，来实现 `bark()` 和继承自Animal的 `go()` ` `name()` 等重载方法（即便Dog类并不直接重载name方法）。

```
class PyAnimal : public Animal {
public:
    using Animal::Animal; // Inherit constructors
    std::string go(int n_times) override {
PYBIND11_OVERRIDE_PURE(std::string, Animal, go, n_times); }
    std::string name() override { PYBIND11_OVERRIDE(std::string, Animal,
name, ); }
};
class PyDog : public Dog {
public:
    using Dog::Dog; // Inherit constructors
    std::string go(int n_times) override { PYBIND11_OVERRIDE(std::string,
Dog, go, n_times); }
    std::string name() override { PYBIND11_OVERRIDE(std::string, Dog, name,
); }
    std::string bark() override { PYBIND11_OVERRIDE(std::string, Dog, bark,
); }
};
```

> 注意到 `name()` 和 `bark()` 尾部的逗号，这用来说明辅助类的函数不带任何参数。当函数至少有一个参数时，应该省略尾部的逗号。

注册一个继承已经在pybind11中注册的带虚函数的类，同样需要为其添加辅助类，即便它没有定义或重载任何虚函数：

```
class Husky : public Dog {};
class PyHusky : public Husky {
public:
    using Husky::Husky; // Inherit constructors
    std::string go(int n_times) override {
PYBIND11_OVERRIDE_PURE(std::string, Husky, go, n_times); }
    std::string name() override { PYBIND11_OVERRIDE(std::string, Husky, name,
); }
    std::string bark() override { PYBIND11_OVERRIDE(std::string, Husky, bark,
); }
};
```

我们可以使用模板辅助类将简化这类重复的绑定工作，这对有多个虚函数的基类尤其有用：

```
template <class AnimalBase = Animal> class PyAnimal : public AnimalBase {
public:
    using AnimalBase::AnimalBase; // Inherit constructors
    std::string go(int n_times) override {
PYBIND11_OVERRIDE_PURE(std::string, AnimalBase, go, n_times); }
    std::string name() override { PYBIND11_OVERRIDE(std::string, AnimalBase,
name, ); }
};
template <class DogBase = Dog> class PyDog : public PyAnimal<DogBase> {
public:
    using PyAnimal<DogBase>::PyAnimal; // Inherit constructors
    // Override PyAnimal's pure virtual go() with a non-pure one:
    std::string go(int n_times) override { PYBIND11_OVERRIDE(std::string,
DogBase, go, n_times); }
    std::string bark() override { PYBIND11_OVERRIDE(std::string, DogBase,
bark, ); }
};
```

这样，我们只需要一个辅助方法来定义虚函数和纯虚函数的重载了。只是这样编译器就需要生成许多额外的方法和类。

下面我们在pybind11中注册这些类：

```
py::class_<Animal, PyAnimal<>> animal(m, "Animal");
py::class_<Dog, Animal, PyDog<>> dog(m, "Dog");
py::class_<Husky, Dog, PyDog<Husky>> husky(m, "Husky");
// ... add animal, dog, husky definitions
```

注意，Husky不需要一个专门的辅助类，因为它没定义任何新的虚函数和纯虚函数的重载。

Python中的使用示例：

```
class ShihTzu(Dog):
    def bark(self):
        return "yip!"
```

8.3 [illegible]

8.3.1 [illegible]

[illegible]Python[illegible]C++[illegible]Python[illegible]

[illegible]Python[illegible]

[illegible]pybind11[illegible]`py::init_alias<Args, ...>()`[illegible]`py::init<Args, ...>()`[illegible]

> **See also**[illegible]See the file `tests/test_virtual_functions.cpp` for complete examples showing both normal and forced trampoline instantiation.

[illegible]

[illegible]C++[illegible]C++[illegible]

[illegible]Python[illegible] Limitations involving reference arguments[illegible]

`get_override()`[illegible]Python[illegible]Consider for example a C++ method which has the signature `bool myMethod(int32_t& value)`, where the return indicates whether something should be done with the `value`. This can be made convenient on the Python side by allowing the Python function to return `None` or an `int`:

```
bool MyClass::myMethod(int32_t& value)
{
    pybind11::gil_scoped_acquire gil;  // Acquire the GIL while in this scope.
    // Try to look up the overridden method on the Python side.
    pybind11::function override = pybind11::get_override(this, "myMethod");
    if (override) {  // method is found
        auto obj = override(value);  // Call the Python function.
        if (py::isinstance<py::int_>(obj)) {  // check if it returned a Python integer type
            value = obj.cast<int32_t>();  // Cast it and assign it to the value.
            return true;  // Return true; value should be used.
        } else {
            return false;  // Python returned none, return false.
        }
    }
    return false;  // Alternatively return MyClass::myMethod(value);
}
```

8.4 定制构造函数

前面章节介绍了绑定构造函数的方法，但它仅仅在C++侧刚好有对应的函数时才能正常工作。为了扩展到更通用的情况，pybind11可以绑定工厂方法作为构造函数。如下所示：

```
class Example {
private:
    Example(int); // private constructor
public:
    // Factory function:
    static Example create(int a) { return Example(a); }
};

py::class_<Example>(m, "Example")
    .def(py::init(&Example::create));
```

虽然可以直接绑定 `create` 方法，有时将其在Python侧将其作为构造函数公开更为合适。这可以通过调用 `.def(py::init(...))` 来完成，只需将对应的函数（返回一个新实例，如create）作为参数传入 `py::init()` 即可。同样的，用这个方法我们也可以传入一个函数，它返回新实例的原始指针或持有者（如``std::unique_ptr`）。如下所示：

```
class Example {
private:
    Example(int); // private constructor
public:
    // Factory function - returned by value:
    static Example create(int a) { return Example(a); }

    // These constructors are publicly callable:
    Example(double);
    Example(int, int);
    Example(std::string);
};

py::class_<Example>(m, "Example")
    // Bind the factory function as a constructor:
    .def(py::init(&Example::create))
    // Bind a lambda function returning a pointer wrapped in a holder:
    .def(py::init([](std::string arg) {
        return std::unique_ptr<Example>(new Example(arg));
    }))
    // Return a raw pointer:
    .def(py::init([](int a, int b) { return new Example(a, b); }))
    // You can mix the above with regular C++ constructor bindings as well:
    .def(py::init<double>())
    ;
```

当Python侧调用这些构造函数时，pybind11将调用工厂函数，并将返回的C++示例存储到Python实例中。

当与重载函数跳板类结合使用时，有两种方法。第一种方法是跳板类增加一个构造函数，函数接受原类的右值引用，这样我们可以从原类的工厂函数构造跳板类的实例。第二种方法是使用 `py::init()` 提供原类和跳板类两个工厂函数。

你也可以指定一个工厂函数，它总是返回跳板类的实例，这与 `py::init_alias<...>` 的行为类似。

下面的示例展示了这两种方法：

```
#include <pybind11/factory.h>
class Example {
public:
    // ...
    virtual ~Example() = default;
};
class PyExample : public Example {
public:
    using Example::Example;
    PyExample(Example &&base) : Example(std::move(base)) {}
};
py::class_<Example, PyExample>(m, "Example")
    // Returns an Example pointer.  If a PyExample is needed, the Example
    // instance will be moved via the extra constructor in PyExample, above.
    .def(py::init([]() { return new Example(); }))
    // Two callbacks:
    .def(py::init([]() { return new Example(); } /* no alias needed */,
                  []() { return new PyExample(); } /* alias needed */))
    // *Always* returns an alias instance (like py::init_alias<>())
    .def(py::init([]() { return new PyExample(); }))
    ;
```

大括号初始化

`pybind11` 潜在地使用C++11的大括号初始化来调用目标类的构造函数，这意味着它也可以绑定隐式的构造函数：

```
struct Aggregate {
    int a;
    std::string b;
};

py::class_<Aggregate>(m, "Aggregate")
    .def(py::init<int, const std::string &>());
```

> Note: 大括号初始化优先匹配带列表初始化的重载构造函数。极少数情况下会出问题，你可以使用 `py::init(...)` 传入一个构造新对象的匿名函数来处理这个问题。

8.5 非公有析构函数

如果一个类拥有私有或保护的析构函数（例如单例类），通过pybind11绑定类时编译器将会报错。本质的问题是 `std::unique_ptr` 智能指针负责管理实例的生命周期需要引用析构函数，即便没有资源需要回收。Pybind11提供了辅助类 `py::nodelete` 来禁止对析构函数的调用。这种情况下，C++侧负责析构对象避免内存泄漏就十分重要。

```
/* ... definition ... */

class MyClass {
private:
    ~MyClass() { }
};

/* ... binding code ... */

py::class_<MyClass, std::unique_ptr<MyClass, py::nodelete>>(m, "MyClass")
    .def(py::init<>())
```

8.6 在析构函数中调用Python

在析构函数中调用Python函数出错时，会抛出异常 `error_already_set` 。如果异常在析构函数外抛出，将会调用 `std::terminate()` 来终结程序。因此，类析构函数必须捕获所有 `error_already_set` 类型的异常，并使用 `error_already_set::discard_as_unraisable()` 来丢弃Python异常。

任意Python函数都可能抛出异常。比如一个Python生成器停止生成条目时，Pyhton将抛出 `StopIteration` 异常，如果生成器的堆栈持有C++对象的最后一个引用时，它将传递异常到C++析构函数。

```
class MyClass {
public:
    ~MyClass() {
        try {
            py::print("Even printing is dangerous in a destructor");
            py::exec("raise ValueError('This is an unraisable exception')");
        } catch (py::error_already_set &e) {
            // error_context should be information about where/why the
occurred,
            // e.g. use __func__ to get the name of the current function
            e.discard_as_unraisable(__func__);
        }
    }
};
```

> Note: pybind11不支持将C++析构函数标识为 `noexcept(false)` 。

8.7 隐式转换

假设有A和B两个类，A可以直接转换为B。

```
py::class_<A>(m, "A")
    /// ... members ...

py::class_<B>(m, "B")
    .def(py::init<A>())
    /// ... members ...

m.def("func",
    [](const B &) { /* .... */ }
);
```

如果想func函数传入A类型的参数a，Pyhton侧需要这样写 `func(B(a))` ，而C++则可以直接使用 `func(a)` ，自动将A类型转换为B类型。

这种情形下（B有一个接受A类型参数的构造函数），我们可以使用如下声明来让Python侧也支持类似的隐式转换：

```
py::implicitly_convertible<A, B>();
```

> Note: A到B的隐式转换仅在通过pybind11绑定了B类型的条件下有效。
>
> 为了防止失控的递归调用，隐式转换时不可重入的：an implicit conversion invoked as part of another implicit conversion of the same type (i.e. from `A` to `B`) will fail.

8.8 静态属性

静态属性也可以像普通属性一样公开getter和setter方法。隐式的self参数仍然存在，并在Python中用于传递Python `type` 子类实例。我们通常在C++侧忽略这个参数，下面的例子演示了如何使用lambda表达式做为getter函数，并忽略self参数。

```
py::class_<Foo>(m, "Foo")
    .def_property_readonly_static("foo", [](py::object /* self */) { return
Foo(); });
```

8.9 重载操作符

假设有这样一个类 `Vector2` ，它通过重载操作符实现了向量加法和标量乘法。

```
class Vector2 {
public:
    Vector2(float x, float y) : x(x), y(y) { }

    Vector2 operator+(const Vector2 &v) const { return Vector2(x + v.x, y + v.y); }
    Vector2 operator*(float value) const { return Vector2(x * value, y * value); }
    Vector2& operator+=(const Vector2 &v) { x += v.x; y += v.y; return *this; }
    Vector2& operator*=(float v) { x *= v; y *= v; return *this; }

    friend Vector2 operator*(float f, const Vector2 &v) {
        return Vector2(f * v.x, f * v.y);
    }

    std::string toString() const {
        return "[" + std::to_string(x) + ", " + std::to_string(y) + "]";
    }
private:
    float x, y;
};
```

操作符绑定代码如下：

```
#include <pybind11/operators.h>

PYBIND11_MODULE(example, m) {
    py::class_<Vector2>(m, "Vector2")
        .def(py::init<float, float>())
        .def(py::self + py::self)
        .def(py::self += py::self)
        .def(py::self *= float())
        .def(float() * py::self)
        .def(py::self * float())
        .def(-py::self)
        .def("__repr__", &Vector2::toString);
}
```

`.def(py::self * float())` 是如下代码的简短标记：

```
.def("__mul__", [](const Vector2 &a, float b) {
    return a * b;
}, py::is_operator())
```

8.10 支持pickle

Python的 `pickle` 模块提供了强大的将Python对象图到二进制数据流的序列化和反序列化的功能。pybind11也提供了 `py::pickle()` 定义来支持pickle和unpickle C++类。现在有这样一个类：

```
class Pickleable {
public:
    Pickleable(const std::string &value) : m_value(value) { }
    const std::string &value() const { return m_value; }

    void setExtra(int extra) { m_extra = extra; }
    int extra() const { return m_extra; }
private:
    std::string m_value;
    int m_extra = 0;
};
```

Python中通过定义 `__setstate__` 和 `__getstate__` 使能pciking支持。对于pybind11类，可以使用 `py::pickle()` 来绑定这两个函数：

```
py::class_<Pickleable>(m, "Pickleable")
    .def(py::init<std::string>())
    .def("value", &Pickleable::value)
    .def("extra", &Pickleable::extra)
    .def("setExtra", &Pickleable::setExtra)
    .def(py::pickle(
        [](const Pickleable &p) { // __getstate__
            /* Return a tuple that fully encodes the state of the object */
            return py::make_tuple(p.value(), p.extra());
        },
        [](py::tuple t) { // __setstate__
            if (t.size() != 2)
                throw std::runtime_error("Invalid state!");

            /* Create a new C++ instance */
            Pickleable p(t[0].cast<std::string>());

            /* Assign any additional state */
            p.setExtra(t[1].cast<int>());

            return p;
        }
    ));
```

`py::pickle()` 中的 `__setstate__` 部分遵循与 `py::init()` 单参数版本相同的规则，返回值可以是一个值，指针或者holder type。

Python中使用示例如下:

```
try:
    import cPickle as pickle  # Use cPickle on Python 2.7
except ImportError:
    import pickle

p = Pickleable("test_value")
p.setExtra(15)
data = pickle.dumps(p, 2)
```

> Note: Note that only the cPickle module is supported on Python 2.7.

The second argument to `dumps` is also crucial: it selects the pickle protocol version 2, since the older version 1 is not supported. Newer versions are also fine—for instance, specify `-1` to always use the latest available version. Beware: failure to follow these instructions will cause important pybind11 memory allocation routines to be skipped during unpickling, which will likely lead to memory corruption and/or segmentation faults.

8.11 深拷贝支持

Python通常在赋值中使用引用。有时需要一个真正的拷贝，以防止修改所有的拷贝实例。Python的 `copy` 模块提供了这样的拷贝能力。

在Python3中，带pickle支持的类自带深拷贝能力。但是，自定义 `__copy__` 和 `__deepcopy__` 方法能够提高拷贝的性能。在Python2.7中，由于pybind11只支持cPickle，要想实现深拷贝，用户必须实现这个两个方法。

对于一些简单的类，可以使用拷贝构造函数来实现深拷贝。如下所示：

```
py::class_<Copyable>(m, "Copyable")
    .def("__copy__",  [](const Copyable &self) {
        return Copyable(self);
    })
    .def("__deepcopy__", [](const Copyable &self, py::dict) {
        return Copyable(self);
    }, "memo"_a);
```

Note: 本例中不会复制动态属性。

8.12 多重继承

pybind11支持绑定多重继承的类，只需在将所有基类作为 `class_` 的模板参数即可：

```
py::class_<MyType, BaseType1, BaseType2, BaseType3>(m, "MyType")
   ...
```

[illegible]holder[illegible]pybind11[illegible]

[illegible]Python[illegible]C++[illegible]C++[illegible]Python[illegible]

[illegible]pybind11[illegible] `multiple_inheritance` [illegible]

```
py::class_<MyType, BaseType2>(m, "MyType", py::multiple_inheritance());
```

[illegible]

8.13 [illegible]Module-local[illegible]

pybind11[illegible]

```
// In the module1.cpp binding code for module1:
py::class_<Pet>(m, "Pet")
    .def(py::init<std::string>())
    .def_readonly("name", &Pet::name);

// In the module2.cpp binding code for module2:
m.def("create_pet", [](std::string name) { return new Pet(name); });
```

```
>>> from module1 import Pet
>>> from module2 import create_pet
>>> pet1 = Pet("Kitty")
>>> pet2 = create_pet("Doggy")
>>> pet2.name()
'Doggy'
```

[illegible]Python[illegible]

在某些例子中，这也会引起冲突。例如，有两个不相干的模块使用了同一个C++外部库，而且他们各自提供了这个库的自定义绑定。当Python程序同时（直接或间接地）导入两个库时，由于外部类型的定义冲突而导致错误。

```
// dogs.cpp

// Binding for external library class:
py::class<pets::Pet>(m, "Pet")
    .def("name", &pets::Pet::name);

// Binding for local extension class:
py::class<Dog, pets::Pet>(m, "Dog")
    .def(py::init<std::string>());
```

```
// cats.cpp, in a completely separate project from the above dogs.cpp.

// Binding for external library class:
py::class<pets::Pet>(m, "Pet")
    .def("get_name", &pets::Pet::name);

// Binding for local extending class:
py::class<Cat, pets::Pet>(m, "Cat")
    .def(py::init<std::string>());
```

```
>>> import cats
>>> import dogs
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: generic_type: type "Pet" is already registered!
```

为避开这点，你可以想 `py::class_` 传递 `py::module_local()` 属性，将外部类绑定到模块内部。

```
// Pet binding in dogs.cpp:
py::class<pets::Pet>(m, "Pet", py::module_local())
    .def("name", &pets::Pet::name);
```

```
// Pet binding in cats.cpp:
py::class<pets::Pet>(m, "Pet", py::module_local())
    .def("get_name", &pets::Pet::name);
```

这样，Python侧的 `dogs.Pet` 和 `cats.Pet` 就是两个不同的类，两个模块也能顺利地同时导入，互不干扰。有两点需要注意的是：1）外部模块不能返回或转换 `Pet` 示例到Python（除非他们也提供自己内部的绑定）；2）在Python的视角来看，他们就是两个截然不同的类。

注意，这个局部性仅作用于C++到Python方向。传递一个 `py::module_local` 类型到C++函数，在module-local类看来仍是合理的。这意味着，下面的函数添加到任意哪个模块（不限于cats和dogs两个模块），它将可以通过 `dogs.Pet` 或 `cats.Pet` 参数来调用。

```
m.def("pet_name", [](const pets::Pet &pet) { return pet.name(); });
```

举个例子，假设上述函数被添加到 `cats.cpp` ， `dogs.cpp` 和 `frogs.cpp` （ `frogs.cpp` 没有绑定 `Pets` 类）。

```
>>> import cats, dogs, frogs  # No error because of the added
py::module_local()
>>> mycat, mydog = cats.Cat("Fluffy"), dogs.Dog("Rover")
>>> (cats.pet_name(mycat), dogs.pet_name(mydog))
('Fluffy', 'Rover')
>>> (cats.pet_name(mydog), dogs.pet_name(mycat), frogs.pet_name(mycat))
('Rover', 'Fluffy', 'Fluffy')
```

即便其他模块已经全局地注册了相同的类型，我们还是可以使用 `py::module_local()` 来注册到另一个模块：在module-local定义的模块，所有C++势力将被转为关联的Python类型。在其他模块，这个实例则被转为全局地Python类型。

> Note: STL bindings (as provided via the optional `pybind11/stl_bind.h` header) apply `py::module_local` by default when the bound type might conflict with other modules; see Binding STL containers for details.
>
> The localization of the bound types is actually tied to the shared object or binary generated by the compiler/linker. For typical modules created with `PYBIND11_MODULE()` , this distinction is not significant. It is possible, however, when Embedding the interpreter to embed multiple modules in the same binary (see Adding embedded modules). In such a case, the localization will apply across all embedded modules within the same binary.

8. 14 绑定protected成员函数

通常不可能向Python公开protected 成员函数：

```
class A {
protected:
    int foo() const { return 42; }
};

py::class_<A>(m, "A")
    .def("foo", &A::foo); // error: 'foo' is a protected member of 'A'
```

因为非公有成员函数意味着外部不可调用。但我们还是希望在Python派生类中使用protected 函数。我们可以通过下面的方式来实现：

```
class A {
protected:
    int foo() const { return 42; }
};

class Publicist : public A { // helper type for exposing protected functions
public:
    using A::foo; // inherited with different access modifier
};

py::class_<A>(m, "A") // bind the primary class
    .def("foo", &Publicist::foo); // expose protected methods via the
publicist
```

因为 `&Publicist::foo` 和 `&A::foo` 准确地说是同一个函数（相同的签名和地址），仅仅是获取方式不同。 `Publicist` 的唯一意图，就是将函数的作用域变为 `public` 。

如果是希望公开在Python侧重载的 `protected` 虚函数，可以将publicist pattern与之前提到的trampoline相结合：

```
class A {
public:
    virtual ~A() = default;

protected:
    virtual int foo() const { return 42; }
};

class Trampoline : public A {
public:
    int foo() const override { PYBIND11_OVERRIDE(int, A, foo, ); }
};

class Publicist : public A {
public:
    using A::foo;
};

py::class_<A, Trampoline>(m, "A") // <-- `Trampoline` here
    .def("foo", &Publicist::foo); // <-- `Publicist` here, not `Trampoline`!
```

8.15 绑定final类

在C++11中，我们可以使用 `findal` 关键字来确保一个类不被继承。 `py::is_final` 属性则可以用来确保一个类在Python中不被继承。底层的C++类型不需要定义为final。

```
class IsFinal final {};

py::class_<IsFinal>(m, "IsFinal", py::is_final());
```

在Python中试图继承这个类，将导致错误：

```
class PyFinalChild(IsFinal):
    pass

TypeError: type 'IsFinal' is not an acceptable base type
```

8.16 定制自动向下转型

如前面“继承与自动转型”一节中解释的，pybind11内置了对C++多态的动态类型的处理。Sometimes, you might want to provide this automatic downcasting behavior when creating bindings for a class hierarchy that does not use standard C++ polymorphism, such as LLVM. As long as there’s some way to determine at runtime whether a downcast is safe, you can proceed by specializing the `pybind11::polymorphic_type_hook` template:

```
enum class PetKind { Cat, Dog, Zebra };
struct Pet {   // Not polymorphic: has no virtual methods
    const PetKind kind;
    int age = 0;
  protected:
    Pet(PetKind _kind) : kind(_kind) {}
};
struct Dog : Pet {
    Dog() : Pet(PetKind::Dog) {}
    std::string sound = "woof!";
    std::string bark() const { return sound; }
};

namespace pybind11 {
    template<> struct polymorphic_type_hook<Pet> {
        static const void *get(const Pet *src, const std::type_info*& type) {
            // note that src may be nullptr
            if (src && src->kind == PetKind::Dog) {
                type = &typeid(Dog);
                return static_cast<const Dog*>(src);
            }
            return src;
        }
    };
} // namespace pybind11
```

When pybind11 wants to convert a C++ pointer of type `Base*` to a Python object, it calls `polymorphic_type_hook<Base>::get()` to determine if a downcast is possible. The `get()` function should use whatever runtime information is available to determine if its `src` parameter is in fact an instance of some class `Derived` that inherits from `Base`. If it finds such a `Derived`, it sets `type = &typeid(Derived)` and returns a pointer to the `Derived` object that contains `src`. Otherwise, it just returns `src`, leaving `type` at its default value of nullptr. If you set `type` to a type that pybind11 doesn’t know about, no downcasting will occur, and the original `src` pointer will be used with its static type `Base*`.

It is critical that the returned pointer and `type` argument of `get()` agree with each other: if `type` is set to something non-null, the returned pointer must point to the start of an object whose type is `type`. If the hierarchy being exposed uses only single inheritance, a simple `return src;` will achieve this just fine, but in the general case, you must cast `src` to the appropriate derived-class pointer (e.g. using `static_cast<Derived>(src)`) before allowing it to be returned as a `void*`.

8.17 访问类型对象

我们可以从已注册的C++类，获取到类型对象：

```
py::type T_py = py::type::of<T>();
```

也可以直接使用`py::type::of(ob)`来获取任意Python对象的类型，跟Python中的`type(ob)`一样。

9. 异常

9.1 C++内置异常到Python异常的转换

当Python通过pybind11调用C++代码时，pybind11将捕获C++异常，并将其翻译为对应的Python异常后抛出。这样Python代码就能够处理它们。

pybind11定义了`std::exception`及其标准子类，和一些特殊异常到Python异常的翻译。由于它们不是真正的Python异常，所以不能使用Python C API来检查。相反，它们是纯C++异常，当它们到达异常处理器时，pybind11将其翻译为对应的Python异常。

Exception thrown by C++	Translated to Python exception type
`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::key_error`	`KeyError` (used to indicate out of bounds access in `__getitem__`, `__setitem__` in dict-like objects, etc.)

Exception thrown by C++	Translated to Python exception type
`pybind11::value_error`	`ValueError` (used to indicate wrong value passed in `container.remove(...)`)
`pybind11::type_error`	`TypeError`
`pybind11::buffer_error`	`BufferError`
`pybind11::import_error`	`ImportError`
`pybind11::attribute_error`	`AttributeError`
Any other exception	`RuntimeError`

异常翻译不是双向的。即上述异常不会捕获源自Python的异常。Python的异常，需要捕获`pybind11::error_already_set`。

这里有个特殊的异常，当入参不能转化为Python对象时，`handle::call()`将抛出`cast_error`异常。

9.2 注册自定义异常翻译

如果上述默认异常转换策略不够用，pybind11也提供了注册自定义异常翻译的支持。类似于pybind11 class，异常翻译也可以定义在模块内或global。要注册一个使用C++异常的`what()`方法将C++到Python的异常转换，可以使用下面的方法：

```
py::register_exception<CppExp>(module, "PyExp");
```

这个调用在指定模块创建了一个名称为PyExp的Python异常，并自动将CppExp相关的异常转换为PyExp异常。

相似的函数可以注册模块内的异常翻译：

```
py::register_local_exception<CppExp>(module, "PyExp");
```

方法的第三个参数handle可以指定异常的基类：

```
py::register_exception<CppExp>(module, "PyExp", PyExc_RuntimeError);
py::register_local_exception<CppExp>(module, "PyExp", PyExc_RuntimeError);
```

这样，PyExp异常可以捕获PyExp和RuntimeError。

Python内置的异常类型可以参考Python文档Standard Exceptions，默认的基类为 `PyExc_Exception` 。

`py::register_exception_translator(translator)` 和 `py::register_local_exception_translator(translator)` 提供了更高级的异常翻译功能，它可以注册任意的异常类型。函数接受一个无状态的回调函数 `void(std::exception_ptr)` 。

C++异常抛出时，注册的异常翻译类将以注册时相反的顺序匹配，优先匹配模块内翻译类，然后再是全局翻译类。

Inside the translator, `std::rethrow_exception` should be used within a try block to re-throw the exception. One or more catch clauses to catch the appropriate exceptions should then be used with each clause using `PyErr_SetString` to set a Python exception or `ex(string)` to set the python exception to a custom exception type (see below).

To declare a custom Python exception type, declare a `py::exception` variable and use this in the associated exception translator (note: it is often useful to make this a static declaration when using it inside a lambda expression without requiring capturing).

The following example demonstrates this for a hypothetical exception classes `MyCustomException` and `OtherException`: the first is translated to a custom python exception `MyCustomError`, while the second is translated to a standard python RuntimeError:

```
static py::exception<MyCustomException> exc(m, "MyCustomError");
py::register_exception_translator([](std::exception_ptr p) {
    try {
        if (p) std::rethrow_exception(p);
    } catch (const MyCustomException &e) {
        exc(e.what());
    } catch (const OtherException &e) {
        PyErr_SetString(PyExc_RuntimeError, e.what());
    }
});
```

Multiple exceptions can be handled by a single translator, as shown in the example above. If the exception is not caught by the current translator, the previously registered one gets a chance.

If none of the registered exception translators is able to handle the exception, it is handled by the default converter as described in the previous section.

9.3 Local vs Global Exception Translators

When a global exception translator is registered, it will be applied across all modules in the reverse order of registration. This can create behavior where the order of module import influences how exceptions are translated.

If module1 has the following translator:

```
py::register_exception_translator([](std::exception_ptr p) {
  try {
      if (p) std::rethrow_exception(p);
  } catch (const std::invalid_argument &e) {
      PyErr_SetString("module1 handled this")
  }
}
```

and module2 has the following similar translator:

```
py::register_exception_translator([](std::exception_ptr p) {
  try {
      if (p) std::rethrow_exception(p);
  } catch (const std::invalid_argument &e) {
      PyErr_SetString("module2 handled this")
  }
}
```

then which translator handles the invalid_argument will be determined by the order that module1 and module2 are imported. Since exception translators are applied in the reverse order of registration, which ever module was imported last will “win” and that translator will be applied.

If there are multiple pybind11 modules that share exception types (either standard built-in or custom) loaded into a single python instance and consistent error handling behavior is needed, then local translators should be used.

Changing the previous example to use `register_local_exception_translator` would mean that when invalid_argument is thrown in the module2 code, the module2 translator will always handle it, while in module1, the module1 translator will do the same.

9.4 在C++中处理Python异常

当C++调用Python函数时（回调函数或者操作Python对象），若Python有异常抛出，pybind11会将Python异常转化为 `pybind11::error_already_set` 类型的异常，它包含了一个C++字符串描述和实际的Python异常。`error_already_set` 用于将Python异常传回Python（或者在C++侧处理）。

Exception raised in Python	Thrown as C++ exception type
Any Python `Exception`	`pybind11::error_already_set`

举个例子：

```
try {
    // open("missing.txt", "r")
    auto file = py::module_::import("io").attr("open")("missing.txt", "r");
    auto text = file.attr("read")();
    file.attr("close")();
} catch (py::error_already_set &e) {
    if (e.matches(PyExc_FileNotFoundError)) {
        py::print("missing.txt not found");
    } else if (e.matches(PyExc_PermissionError)) {
        py::print("missing.txt found but not accessible");
    } else {
        throw;
    }
}
```

该方法并不适用与C++到Python的翻译，Python侧抛出的异常总是被翻译为 `error_already_set`.

```
try {
    py::eval("raise ValueError('The Ring')");
} catch (py::value_error &boromir) {
    // Boromir never gets the ring
    assert(false);
} catch (py::error_already_set &frodo) {
    // Frodo gets the ring
    py::print("I will take the ring");
}

try {
    // py::value_error is a request for pybind11 to raise a Python exception
    throw py::value_error("The ball");
} catch (py::error_already_set &cat) {
    // cat won't catch the ball since
    // py::value_error is not a Python exception
    assert(false);
} catch (py::value_error &dog) {
    // dog will catch the ball
    py::print("Run Spot run");
    throw;  // Throw it again (pybind11 will raise ValueError)
}
```

9.5 处理Python C API的错误

尽可能地使用pybind11 wrappers代替直接调用Python C API。如果确实需要直接使用Python C API，除了需要手动管理引用计数外，还必须遵守pybind11的错误处理协议。

在调用Python C API后，如果Python返回错误，需要调用 `throw py::error_already_set();` 语句，让pybind11来处理异常并传递给Python解释器。这包括对错误设置函数的调用，如 `PyErr_SetString` 。

```
PyErr_SetString(PyExc_TypeError, "C API type error demo");
throw py::error_already_set();

// But it would be easier to simply...
throw py::type_error("pybind11 wrapper type error");
```

也可以调用 `PyErr_Clear` 来忽略错误。

任何Python错误必须被抛出或清除，否则Python/pybind11将处于无效的状态。

9.6 异常链（raise from）

在Python 3.3中，引入了指示异常是由其他异常引发的机制：

```
try:
    print(1 / 0)
except Exception as exc:
    raise RuntimeError("could not divide by zero") from exc
```

pybind11 2.8版本，你可以使用 `py::raise_from` 函数来完成相同的事。它设置当前Python错误指示器，所以要继续传播异常，你应该 `throw py::error_already_set()`（Python 3 only）。

```
try {
    py::eval("print(1 / 0"));
} catch (py::error_already_set &e) {
    py::raise_from(e, PyExc_RuntimeError, "could not divide by zero");
    throw py::error_already_set();
}
```

9.7 处理unraiseable异常

如果Python调用的C++析构函数或任何标记为 `noexcept(true)` 的函数抛出了异常，该异常不会传播出去。如果它们在调用图中抛出或捕捉不到任何异常，c++运行时将调用std::terminate()立即终止程序。

类似的，在类 `__del__` 方法引发的Python异常也不会传播，但被Python作为unraisable错误记录下来。在Python 3.8+中，将触发system hook，并记录auditing event日志。

任何noexcept函数应该使用try-catch代码块来捕获 `error_already_set` （或其他可能出现的异常）。pybind11包装的Python异常并非真正的Python异常，它是pybind11捕获并转化的C++异常。noexcept函数不能传播这些异常。我们可以将它们转换为Python异常，然后丢弃 `discard_as_unraisable` ，如下所示。

```
void nonthrowing_func() noexcept(true) {
    try {
        // ...
    } catch (py::error_already_set &eas) {
        // Discard the Python error using Python APIs, using the C++ magic
        // variable __func__. Python already knows the type and value and of the
        // exception object.
        eas.discard_as_unraisable(__func__);
    } catch (const std::exception &e) {
        // Log and discard C++ exceptions.
        third_party::log(e);
    }
}
```

10. 智能指针

10.1 std::unique_ptr

给定一个带Python绑定的类 `Example` ，我们可以像下面一样返回它的unique pointer智能指针实例：

```
std::unique_ptr<Example> create_example() { return std::unique_ptr<Example>
(new Example()); }

m.def("create_example", &create_example);
```

没其他需要特殊处理的地方。需要注意的是，虽然允许返回unique_ptr对象，但是将其作为函数入参是非法的。例如，pybind11不能处理下列函数签名。

```
void do_something_with_example(std::unique_ptr<Example> ex) { ... }
```

上面的签名意味着Python需要放弃对象的所有权，并将其传递给该函数，这通常是不可能的（对象可能在别处被引用）。

10.2 std::shared_ptr

`class_` 可以传递一个表示持有者类型的模板类型，它用于管理对象的引用。在不指定的情况下，默认为 `std::unique_ptr<Type>` 类型，这意味着当Python的引用计数为0时，将析构对象。该模板类型可以指定为其他的智能指针或引用计数包装类，像下面我们就使用了 `std::shared_ptr` ：

```
py::class_<Example, std::shared_ptr<Example> /* <- holder type */> obj(m,
"Example");
```

注意，每个类仅能与一个持有者类型关联。

使用持有者类型的一个潜在的障碍就是，你需要始终如一的使用它们。猜猜下面的绑定代码有什么问题？

```
class Child { };

class Parent {
public:
   Parent() : child(std::make_shared<Child>()) { }
   Child *get_child() { return child.get(); }  /* Hint: ** DON'T DO THIS **
*/
private:
    std::shared_ptr<Child> child;
};

PYBIND11_MODULE(example, m) {
    py::class_<Child, std::shared_ptr<Child>>(m, "Child");

    py::class_<Parent, std::shared_ptr<Parent>>(m, "Parent")
       .def(py::init<>())
       .def("get_child", &Parent::get_child);
}
```

下面的Python代码将导致未定义行为（类似段错误）。

```
from example import Parent
print(Parent().get_child())
```

问题在于`Parent::get_child()`返回类`Child`实例的指针，但事实上这个经由`std::shared_ptr<...>`管理的实例，在传递原始指针时就丢失了。这个例子中，pybind11将创建第二个独立的`std::shared_ptr<...>`声明指针的所有权。最后，对象将被free两次，因为两个shared指针没法知道彼此的存在。

有两种方法解决这个问题：

1. 对于智能指针管理的类型，永远不要在函数如参数或返回值中使用原始指针。换句话说，在任何需要使用该类型指针的地方，使用它们指定的持有者类型代替。这个例子中`get_child()`可以这样修改：

   ```
   std::shared_ptr<Child> get_child() { return child; }
   ```

2. 定义`Child`时指定`std::enable_shared_from_this<T>`作为基类。这将在`Child`的基础上增加一点信息，让pybind11认识到这里已经存在一个`std::shared_ptr<...>`，并与之交互。修改示例如下：

```
class Child : public std::enable_shared_from_this<Child> { };
```

10.3 自定义智能指针

pybind11支持开箱即用的 `std::unique_ptr` 和 `std::shared_ptr` 。对于其他自定义的智能指针，可以使用下面的宏使能透明转换（transparent conversions）。它必须在其他绑定代码之前在顶层名称空间中声明：

```
PYBIND11_DECLARE_HOLDER_TYPE(T, SmartPtr<T>);
```

宏的第一个参数为占位符名称，用作第二个参数的模板参数。因此，你可以使用任意的标识符（不要使用你的代码中已经存在的类型），只需保持两边一致即可。

宏也可以接收第三个可选的bool类型参数，默认为false。

```
PYBIND11_DECLARE_HOLDER_TYPE(T, SmartPtr<T>, true);
```

如果 `SmartPtr<T>` 总是从 `T*` 指针初始化，不存在不一致的风险（如多个独立的 `SmartPtr<T>` 认为他们是 `T*` 指针的唯一拥有者）。当 `T` 实例使用侵入式引用计数时，应设定为 `true` 。

在使用该特性前，请先阅读 General notes regarding convenience macros。

默认情况下，pybind11假定自定义智能指针具有标准接口，如提供 `.get()` 成员函数来获取底层的原始指针。如果没有，则需要指定 `holder_helper`：

```
// Always needed for custom holder types
PYBIND11_DECLARE_HOLDER_TYPE(T, SmartPtr<T>);

// Only needed if the type's `.get()` goes by another name
namespace pybind11 { namespace detail {
    template <typename T>
    struct holder_helper<SmartPtr<T>> { // <-- specialization
        static const T *get(const SmartPtr<T> &p) { return p.getPointer(); }
    };
}}
```

上述特化告诉pybind11，自定义 `SmartPtr` 通过 `.getPointer()` 提供 `.get()` 接口。

see also: 文件 `tests/test_smart_ptr.cpp` 提供了一个展示如何使用自定义引用计数holder类型的详细示例。

11. [illegible]

[illegible]pybind11[illegible]C++[illegible]Python[illegible]

1. [illegible]C++[illegible]pybind11[illegible]Python[illegible]
2. [illegible]Python[illegible]C++[illegible]
3. C++[illegible]C++[illegible]Python[illegible]Python[illegible]pybind11[illegible] [illegible]Python[illegible]C++[illegible]C++[illegible]Python[illegible] pybind11[illegible]

[illegible]

11.1 [illegible]

1. Native type in C++, wrapper in Python

[illegible]“[illegible]”[illegible]`py::class_`[illegible]C++[illegible]C++[illegible]`py::class_`[illegible]Python[illegible]C++[illegible]Python[illegible]pybind11[illegible]C++[illegible]Python[illegible]

2. Wrapper in C++, native type in Python

[illegible]Python[illegible]tuple[illegible]list[illegible]C++[illegible]`py::object`[illegible]

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

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

Python的list仅仅是包裹在了C++ `py::list` 类里，并没有仅仅任何转换。它的核心任然是一个Python对象。拷贝一个 `py::list` 会像Python中一样增加引用计数。将对象返回到Python侧，将去掉这层封装。

3. Converting between native C++ and Python types

前面两种情况，我们在一种语言中使用原生类型，而在另一种语言中使用它的包装类型。现在，我们在两侧都使用原生类型，并对他们进行类型转换。

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

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

这个例子中，pybind11将创建一个 `std::vector<int>` 实例，并从Python list中拷贝每个元素。然后将该实例传递给 `print_vector` 。同样的事情发生在另一个方向：新建了一个list，并从C++的vector中获取元素值。

如下表所示，多数转换是开箱即用的。他们相当方便，但请记住一点，这些转换是基于数据拷贝的。这对小型的不变的类型相当友好，对于大型数据结构则相当昂贵。这可以通过自定义包装类型重载自动转换来规避（如上面提到的方法1）。This requires some manual effort and more details are available in the Making opaque types section.

内置转换的列表

下面基础数据类型是开箱即用的（有些可能需要include额外的头文件）。

Data type	Description	Header file
`int8_t`, `uint8_t`	8-bit integers	`pybind11/pybind11.h`
`int16_t`, `uint16_t`	16-bit integers	`pybind11/pybind11.h`
`int32_t`, `uint32_t`	32-bit integers	`pybind11/pybind11.h`
`int64_t`, `uint64_t`	64-bit integers	`pybind11/pybind11.h`

Data type	Description	Header file
`ssize_t`, `size_t`	Platform-dependent size	`pybind11/pybind11.h`
`float`, `double`	Floating point types	`pybind11/pybind11.h`
`bool`	Two-state Boolean type	`pybind11/pybind11.h`
`char`	Character literal	`pybind11/pybind11.h`
`char16_t`	UTF-16 character literal	`pybind11/pybind11.h`
`char32_t`	UTF-32 character literal	`pybind11/pybind11.h`
`wchar_t`	Wide character literal	`pybind11/pybind11.h`
`const char *`	UTF-8 string literal	`pybind11/pybind11.h`
`const char16_t *`	UTF-16 string literal	`pybind11/pybind11.h`
`const char32_t *`	UTF-32 string literal	`pybind11/pybind11.h`
`const wchar_t *`	Wide string literal	`pybind11/pybind11.h`
`std::string`	STL dynamic UTF-8 string	`pybind11/pybind11.h`
`std::u16string`	STL dynamic UTF-16	`pybind11/pybind11.h`

Data type	Description	Header file
	string	
`std::u32string`	STL dynamic UTF-32 string	`pybind11/pybind11.h`
`std::wstring`	STL dynamic wide string	`pybind11/pybind11.h`
`std::string_view`, `std::u16string_view`, etc.	STL C++17 string views	`pybind11/pybind11.h`
`std::pair<T1, T2>`	Pair of two custom types	`pybind11/pybind11.h`
`std::tuple<...>`	Arbitrary tuple of types	`pybind11/pybind11.h`
`std::reference_wrapper<...>`	Reference type wrapper	`pybind11/pybind11.h`
`std::complex<T>`	Complex numbers	`pybind11/complex.h`
`std::array<T, Size>`	STL static array	`pybind11/stl.h`
`std::vector<T>`	STL dynamic array	`pybind11/stl.h`
`std::deque<T>`	STL double-ended queue	`pybind11/stl.h`
`std::valarray<T>`	STL value array	`pybind11/stl.h`
`std::list<T>`	STL linked list	`pybind11/stl.h`

Data type	Description	Header file
`std::map<T1, T2>`	STL ordered map	`pybind11/stl.h`
`std::unordered_map<T1, T2>`	STL unordered map	`pybind11/stl.h`
`std::set<T>`	STL ordered set	`pybind11/stl.h`
`std::unordered_set<T>`	STL unordered set	`pybind11/stl.h`
`std::optional<T>`	STL optional type (C++17)	`pybind11/stl.h`
`std::experimental::optional<T>`	STL optional type (exp.)	`pybind11/stl.h`
`std::variant<...>`	Type-safe union (C++17)	`pybind11/stl.h`
`std::filesystem::path<T>`	STL path (C++17) [1]	`pybind11/stl.h`
`std::function<...>`	STL polymorphic function	`pybind11/functional.h`
`std::chrono::duration<...>`	STL time duration	`pybind11/chrono.h`
`std::chrono::time_point<...>`	STL date/time	`pybind11/chrono.h`
`Eigen::Matrix<...>`	Eigen: dense matrix	`pybind11/eigen.h`
`Eigen::Map<...>`	Eigen: mapped memory	`pybind11/eigen.h`

Data type	Description	Header file
`Eigen::SparseMatrix<...>`	Eigen: sparse matrix	`pybind11/eigen.h`

11.2 Strings, bytes and Unicode conversions

> Note: 本节讨论的string处理基于Python3 strings。对于python2.7，使用 `unicode` 替换 `str` ， `str` 替换 `bytes` 。Python2.7用于最好使用 `from __future__ import unicode_literals` 避免无意间使用 `str` 代替 `unicode` 。

11.2.1 传递Python strings到C++

当向一个接收 `std::string` 或 `char *` 参数的函数传递Python的 `str` 时，pybind11会将Python字符串编码为UTF-8。所有的Python `str` 都能够用UTF-8编码，所以这个操作不会失败。

C++语言是encoding agnostic。程序员负责处理编码，最简单的做法就是每处都使用UTF-8。

```
m.def("utf8_test",
    [](const std::string &s) {
        cout << "utf-8 is icing on the cake.\n";
        cout << s;
    }
);
m.def("utf8_charptr",
    [](const char *s) {
        cout << "My favorite food is\n";
        cout << s;
    }
);
```

```
>>> utf8_test("🎂")
utf-8 is icing on the cake.
🎂

>>> utf8_charptr("🍕")
My favorite food is
🍕
```

> Note: 有些终端模拟器不支持UTF-8或emoji字体，上面的例子可能无法显示。

无论C++函数的参数是传值或引用，是否是const，结果都是一样的。

向C++传递bytes对象

向接收 `std::string` 或 `char *` 类型参数的C++函数传递Python bytes对象无需转换。在Python3上，如果想要函数只接收bytes，不接收str，可以声明参数类型为 `py::bytes` 。

11.2.2 向Python返回C++ 字符串

当C++函数返回 `std::string` 或 `char*` 参数给Python调用者时，pybind11会将字符串以UTF-8格式解码给原生Python str，类似于Python中的 `bytes.decode('utf-8')` 。如果隐式转换失败，pybind11将会抛出异常 `UnicodeDecodeError` 。

```
m.def("std_string_return",
    []() {
        return std::string("This string needs to be UTF-8 encoded");
    }
);
```

```
>>> isinstance(example.std_string_return(), str)
True
```

因为UTF-8包含纯ASCII，返回一个纯ASCII字符串到Python没有任何问题。否则就需要确保编码是有效的UTF-8 。

> Warning: 隐式转换假定 `char *` 字符串以null为结束符。若不是，将导致缓冲区溢出。

显式转换

如果C++代码构造了一个非UTF-8的string字符串，可以执行显式转换并返回 `py::str` 对象。显式转换与隐式转换的开销相同。

```
// This uses the Python C API to convert Latin-1 to Unicode
m.def("str_output",
    []() {
        std::string s = "Send your r\xe9sum\xe9 to Alice in HR"; // Latin-1
        py::str py_s = PyUnicode_DecodeLatin1(s.data(), s.length());
        return py_s;
    }
);
```

```
>>> str_output()
'Send your résumé to Alice in HR'
```

Python C API提供了一些内置的编解码方法可以使用。也可以使用第三方库如libiconv 来转换UTF-8。

不使用类型转换来返回C++字符串

如果C++ `std::string` 中的数据不表示文本，则应该以 `bytes` 的形式传递给Python，这时我们可以返回一个 `py::btyes` 对象。

```
m.def("return_bytes",
    []() {
        std::string s("\xba\xd0\xba\xd0");  // Not valid UTF-8
        return py::bytes(s);  // Return the data without transcoding
    }
);
```

```
>>> example.return_bytes()
b'\xba\xd0\xba\xd0'
```

注意：pybind11可以将bytes无需编码地转换为 `std::string`，但不能不经编码地隐式转换 `std::string` 到bytes。

```
m.def("asymmetry",
    [](std::string s) {  // Accepts str or bytes from Python
        return s;  // Looks harmless, but implicitly converts to str
    }
);
```

```
>>> isinstance(example.asymmetry(b"have some bytes"), str)
True

>>> example.asymmetry(b"\xba\xd0\xba\xd0")  # invalid utf-8 as bytes
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xba in position 0:
invalid start byte
```

11.2.3 宽字符串

向入参为 `std::wstring`、`wchar_t*`、`std::u16string` 或 `std::u32string` 的C++函数传递Python str对象，str将被编码为UTF-16或UTF-32（具体哪种取决于C++编译器的支持）。当C++函数返回这些类型的字符串到Python str时，需要保证字符串是合法的UTF-16或UTF-32。

```
#define UNICODE
#include <windows.h>

m.def("set_window_text",
    [](HWND hwnd, std::wstring s) {
        // Call SetWindowText with null-terminated UTF-16 string
        ::SetWindowText(hwnd, s.c_str());
    }
);
m.def("get_window_text",
    [](HWND hwnd) {
        const int buffer_size = ::GetWindowTextLength(hwnd) + 1;
        auto buffer = std::make_unique< wchar_t[] >(buffer_size);

        ::GetWindowText(hwnd, buffer.data(), buffer_size);

        std::wstring text(buffer.get());

        // wstring will be converted to Python str
        return text;
    }
);
```

> 警告：带 `--enable-unicode=ucs2` 选项编译的Python 2.7和3.3版本可能不支持上述的宽字符串。

多字节编码地字符串，如Shift-JIS，必须转换为UTF-8/16/32后，再返回给Python。

11.2.4 字符类型

向一个入参为字符类型（char, wchar_t）的C++函数，传递Python str，C++函数将接收str的首字符。如果字符串超过一个Unicode字符长度，将忽略尾部字节。

当C++返回一个字符变量时，它将被转化为单字符的str变量。

```
m.def("pass_char", [](char c) { return c; });
m.def("pass_wchar", [](wchar_t w) { return w; });
```

```
example.pass_char("A")
'A'
```

虽然C++可以将整数转换为字符类型（`char c = 0x65`），pybind11并不会隐式转换Python整数到字符类型。可以使用`chr()` Python函数来将整数转换为字符。

```
>>> example.pass_char(0x65)
TypeError

>>> example.pass_char(chr(0x65))
'A'
```

如果需要使用8-bit整数，可使用`int8_t`或`uint8_t`作为参数类型。

11.2.5 Grapheme clusters

A single grapheme may be represented by two or more Unicode characters. For example ‘é’ is usually represented as U+00E9 but can also be expressed as the combining character sequence U+0065 U+0301 (that is, the letter ‘e’ followed by a combining acute accent). The combining character will be lost if the two-character sequence is passed as an argument, even though it renders as a single grapheme.

```
>>> example.pass_wchar("é")
'é'

>>> combining_e_acute = "e" + "\u0301"

>>> combining_e_acute
'é'

>>> combining_e_acute == "é"
False

>>> example.pass_wchar(combining_e_acute)
'e'
```

Normalizing combining characters before passing the character literal to C++ may resolve *some* of these issues:

```
>>> example.pass_wchar(unicodedata.normalize("NFC", combining_e_acute))
'é'
```

In some languages (Thai for example), there are graphemes that cannot be expressed as a single Unicode code point, so there is no way to capture them in a C++ character type.

11.2.6 c++17 string_view

C++17 string views are automatically supported when compiling in C++17 mode. They follow the same rules for encoding and decoding as the corresponding STL string type (for example, a `std::u16string_view` argument will be passed UTF-16-encoded data, and a returned `std::string_view` will be decoded as UTF-8).

11.3 STL容器

11.3.1 自动转换

包含头文件 `pybind11/stl.h` 后，自动支持 `std::vector<>` / `std::deque<>` / `std::list<>` / `std::array<>` / `std::valarray<>` , `std::set<>` / `std::unordered_set<>` , 和 `std::map<>` / `std::unordered_map<>` 到Python `list` , `set` 和 `dict` 的类型转换。 `std::pair<>` 和 `std::tuple<>` 类型转换在 `pybind11/pybind11.h` 中已经支持。

隐式转换的主要缺点就是Python和C++之间的容器类型转换都需要拷贝数据，这对程序语义和性能有一定的影响。后续章节将介绍如何避免该问题。

> Note: 这些类型任意嵌套都是可以的。

11.3.2 C++17库的容器

`pybind11/stl.h` 支持C++17的 `std::optional<>` 和 `std::variant<>`，C++14的 `std::experimental::optional<>`。

C++11中也存在这些容器的其他版本，如Boost中。pybind11提供了一个简单的方法 `type_caster` 来处理这些类型：

```
// `boost::optional` as an example -- can be any `std::optional`-like
container
namespace pybind11 { namespace detail {
    template <typename T>
    struct type_caster<boost::optional<T>> :
optional_caster<boost::optional<T>> {};
}}
```

上述内容应放到头文件中，并在需要的地方包含它们。Similarly, a specialization can be provided for custom variant types:

```
// `boost::variant` as an example -- can be any `std::variant`-like container
namespace pybind11 { namespace detail {
    template <typename... Ts>
    struct type_caster<boost::variant<Ts...>> :
variant_caster<boost::variant<Ts...>> {};

    // Specifies the function used to visit the variant -- `apply_visitor`
instead of `visit`
    template <>
    struct visit_helper<boost::variant> {
        template <typename... Args>
        static auto call(Args &&...args) ->
decltype(boost::apply_visitor(args...)) {
            return boost::apply_visitor(args...);
        }
    };
}} // namespace pybind11::detail
```

The `visit_helper` specialization is not required if your `name::variant` provides a `name::visit()` function. For any other function name, the specialization must be included to tell pybind11 how to visit the variant.

Warning: When converting a `variant` type, pybind11 follows the same rules as when determining which function overload to call (Overload resolution order), and so the same caveats hold. In particular, the order in which the `variant`'s alternatives are listed is important, since pybind11 will try conversions in this order. This means that, for example, when converting `variant<int, bool>`, the `bool` variant will never be selected, as any Python `bool` is already an `int` and is convertible to a C++ `int`. Changing the order of alternatives (and using `variant<bool, int>`, in this example) provides a solution.

11.3.3 制作opaque类型

pybind11严重依赖于模板匹配机制来转换STL类型的参数和返回值，如vector，链表，哈希表等。甚至会递归处理，如lists of hash maps of pairs of elementary and custom types。

然而，这个方法的一个基本限制就是，Python和C++类型的转换涉及到拷贝操作，这妨碍了pass-by-reference的语义。

假设我们绑定如下函数：

```
void append_1(std::vector<int> &v) {
   v.push_back(1);
}
```

在Python中调用它：

```
>>> v = [5, 6]
>>> append_1(v)
>>> print(v)
[5, 6]
```

如你所见，通过引用传递STL数据时，修改并不会传递到Python侧。相同的场景表现在通过`def_readwrite`或`def_readonly`函数公开STL数据结构时：

```
/* ... definition ... */

class MyClass {
    std::vector<int> contents;
};

/* ... binding code ... */

py::class_<MyClass>(m, "MyClass")
    .def(py::init<>())
    .def_readwrite("contents", &MyClass::contents);
```

[illegible]append[illegible]

```
>>> m = MyClass()
>>> m.contents = [5, 6]
>>> print(m.contents)
[5, 6]
>>> m.contents.append(7)
>>> print(m.contents)
[5, 6]
```

[illegible]pybind11[illegible] `PYBIND11_MAKE_OPAQUE(T)` [illegible]*opaque*[illegible]opaque[illegible] `std::vector<int>` [illegible]opaque[illegible]

```
PYBIND11_MAKE_OPAQUE(std::vector<int>);
```

[illegible] `type_caster` [illegible] `std::vector<int>` [illegible]opaque[illegible] `class_` [illegible]Python[illegible]

```
py::class_<std::vector<int>>(m, "IntVector")
    .def(py::init<>())
    .def("clear", &std::vector<int>::clear)
    .def("pop_back", &std::vector<int>::pop_back)
    .def("__len__", [](const std::vector<int> &v) { return v.size(); })
    .def("__iter__", [](std::vector<int> &v) {
       return py::make_iterator(v.begin(), v.end());
    }, py::keep_alive<0, 1>()) /* Keep vector alive while iterator is used */
    // ....
```

11.3.4 绑定STL容器

公开STL容器作为一个Python对象时一个相当常见的需求，因此pybind11也提供了一个可选的头文件`pybind11/stl_bind.h`来做这件事。映射容器会尽可能的去匹配Python对应类型的行为。下面示例将展示该机制的使用方法：

```
// Don't forget this
#include <pybind11/stl_bind.h>

PYBIND11_MAKE_OPAQUE(std::vector<int>);
PYBIND11_MAKE_OPAQUE(std::map<std::string, double>);

// ...

// later in binding code:
py::bind_vector<std::vector<int>>(m, "VectorInt");
py::bind_map<std::map<std::string, double>>(m, "MapStringDouble");
```

绑定STL容器时，pybind11会根据容器元素的类型来决定该容器是否应该局限于模块内（参考Module-local class bindings特性）。如果容器元素的类型不是已经绑定的自定义类型且未标识`py::module_local`，那么容器绑定将应用`py::module_local`。这包括数值类型、strings、Eigen类型，和其他在绑定STL容器时还未绑定的类型。module-local绑定的意图是为了避免模块间的潜在的冲突（如，两个独立的模块都试图绑定`std::vector<int>`）。

通过指定属性`py::module_local()`或`py_module_local(false)`，你也可以强制绑定的STL容器公开在模块内或全局：

```
py::bind_vector<std::vector<int>>(m, "VectorInt", py::module_local(false));
```

注意：这样的全局绑定可能会导致模块无法加载，如果其他的模块也绑定了相同的容器类型（如`std::vector<int>`）并以加载。

11.4 函数对象

要使能以下属性，需要包含 `pybind11/functional.h` 。

回调并传递匿名函数

C++11标准引入了功能强大的lambda函数和泛函对象 `std::function<>` 。lambda函数有两种类型：无状态lambda函数类似于指向一块匿名代码的函数指针，而有状态lambda函数还依赖于存储在lambda闭包对象中的被捕获的参数。

这里有一个接收任意函数签名为 `int -> int` 的函数类型参数（有状态或无状态）：

```
int func_arg(const std::function<int(int)> &f) {
    return f(10);
}
```

下面的例子更复杂，它的入参是一个函数对象，并返回一个同样类型的函数对象。返回值是一个有状态的lambda函数，它捕获了 `f` 参数：

```
std::function<int(int)> func_ret(const std::function<int(int)> &f) {
    return [f](int i) {
        return f(i) + 1;
    };
}
```

在C++回调函数中使用python具名参数，需要使用 `py::cpp_function` 包裹，类似于下面的代码：

```
py::cpp_function func_cpp() {
    return py::cpp_function([](int i) { return i+1; },
       py::arg("number"));
}
```

包含 `pybind11/functional.h` 头文件后，可以简单地直接为上述函数生成绑定代码：

```
#include <pybind11/functional.h>

PYBIND11_MODULE(example, m) {
    m.def("func_arg", &func_arg);
    m.def("func_ret", &func_ret);
    m.def("func_cpp", &func_cpp);
}
```

Python中交互示例如下:

```
$ python
>>> import example
>>> def square(i):
...     return i * i
...
>>> example.func_arg(square)
100L
>>> square_plus_1 = example.func_ret(square)
>>> square_plus_1(4)
17L
>>> plus_1 = func_cpp()
>>> plus_1(number=43)
44L
```

> Warning
>
> 请记住在从C++传递函数对象到Python的过程中（反向亦然），将生成一些包装代码来两种语言的函数调用。这种翻译自然会稍微增加函数调用的开销。当一个函数在Python和C++之间来回拷贝多次时，包装层数会不断累积，会明显降低性能。
>
> 这里有个例外：一个无状态函数作为参数传递给在Python中公开的另一个C++函数时，将不会有额外的开销。Pybind11将从封装的函数中提取C++函数指针，以回避潜在地C++ -> Python -> C++的往返。

11.5 Chrono

包含 `pybind11/chrono` 将使能C++11 chrono和Python datatime对象将的自动转换，还支持python floats（从 `time.monotonic()` 或 `time.perf_counter()` 获取的）和 `time.process_time()` 到durations的转

换。

11.5.1 C++11时钟的概览

使用这些转换时容易混淆的点是，C++11中提供的各种时钟的差异。C++11标准定义了三种时钟类型，用户也可以根据自身需求定义自己的时钟类型。这些时钟有着不用的属性，与Python之间转换时也会获得不同的结果。

标准定义的第一种时钟 `std::chrono::system_clock`。它测量当前的时间和日期。但是，这个时钟会随着操作系统的时钟变化而改变。例如，在系统时间与时间服务器同步时，这个时钟也会跟着改变。这对计时功能来说很糟糕，但对测量wall time还是有用的。

标准定义的第二种时钟 `std::chrono::steady_clock`。这个时钟以稳定的速度跳动，从不调整。这非常实用于计时功能，但与实际时间和日志并不一致。这个时间通常是你操作系统已经运行的时间，虽然不是必须的。这个时钟永远不会与系统时钟相同，因为系统时钟可以改变，但steady_clock不能。

标准定义的第二种时钟 `std::chrono::high_resolution_clock`。它是系统中分辨率最高的时钟，通常是system clock 或 steady clock的一种，也可以有自己独立的时钟。需要注意的是，你在Python中获取到的该时钟的转换值，可能存在差异，这取决于系统的实现。如果它是系统时钟的一种，Python将得到datetime对象，否则将得到timedelta对象。

11.5.2 提供的转换

C++到Python

- `std::chrono::system_clock::time_point` → `datetime.datetime`
- `std::chrono::duration` → `datetime.timedelta`
- `std::chrono::[other_clocks]::time_point` → `datetime.timedelta`

Python到C++

- `datetime.datetime` or `datetime.date` or `datetime.time` → `std::chrono::system_clock::time_point`
- `datetime.timedelta` → `std::chrono::duration`
- `datetime.timedelta` → `std::chrono::[other_clocks]::time_point`
- `float` → `std::chrono::duration`
- `float` → `std::chrono::[other_clocks]::time_point`

11.6 Eigen

没接触过Eigen，先不翻译。

11.7 自定义类型转换

在极少数情况下，程序可能需要一些pybind11没有提供的自定义类型转换，这需要使用到原始的Python C API。这是相当高级的使用方法，只有熟悉Python引用计数复杂之处的专家才能使用。

The following snippets demonstrate how this works for a very simple `inty` type that that should be convertible from Python types that provide a `__int__(self)` method.

```
struct inty { long long_value; };

void print(inty s) {
    std::cout << s.long_value << std::endl;
}
```

The following Python snippet demonstrates the intended usage from the Python side:

```
class A:
    def __int__(self):
        return 123


from example import print

print(A())
```

To register the necessary conversion routines, it is necessary to add an instantiation of the `pybind11::detail::type_caster<T>` template. Although this is an implementation detail, adding an instantiation of this type is explicitly allowed.

```
namespace pybind11 { namespace detail {
    template <> struct type_caster<inty> {
    public:
        /**
         * This macro establishes the name 'inty' in
         * function signatures and declares a local variable
         * 'value' of type inty
         */
        PYBIND11_TYPE_CASTER(inty, _("inty"));

        /**
         * Conversion part 1 (Python->C++): convert a PyObject into a inty
         * instance or return false upon failure. The second argument
         * indicates whether implicit conversions should be applied.
         */
        bool load(handle src, bool) {
            /* Extract PyObject from handle */
            PyObject *source = src.ptr();
            /* Try converting into a Python integer value */
            PyObject *tmp = PyNumber_Long(source);
            if (!tmp)
                return false;
            /* Now try to convert into a C++ int */
            value.long_value = PyLong_AsLong(tmp);
            Py_DECREF(tmp);
            /* Ensure return code was OK (to avoid out-of-range errors etc)
*/
            return !(value.long_value == -1 && !PyErr_Occurred());
        }

        /**
         * Conversion part 2 (C++ -> Python): convert an inty instance into
         * a Python object. The second and third arguments are used to
         * indicate the return value policy and parent object (for
         * ``return_value_policy::reference_internal``) and are generally
         * ignored by implicit casters.
         */
        static handle cast(inty src, return_value_policy /* policy */, handle
/* parent */) {
            return PyLong_FromLong(src.long_value);
        }
    };
}} // namespace pybind11::detail
```

Note: A `type_caster<T>` defined with `PYBIND11_TYPE_CASTER(T, ...)` requires that `T` is default-constructible (`value` is first default constructed and then `load()` assigns to it).

Warning: When using custom type casters, it’s important to declare them consistently in every compilation unit of the Python extension module. Otherwise, undefined behavior can ensue.

12. Python C++接口

pybind11通过简单的C++包装公开了Python类型和函数，这使得我们可以方便的在C++中调用Python代码，而无需借助Python C API。

12.1 Python类型

12.1.1 可用的封装

所有主要的Python类型通过简单C++类封装公开出来了，可以当做参数参数来使用。包括：`handle`，`object`，`bool_`，`int_`，`float_`，`str`，`bytes`，`tuple`，`list`，`dict`，`slice`，`none`，`capsule`，`iterable`，`iterator`，`function`，`buffer`，`array`，和`array_t`.

> Warning: Be sure to review the Gotchas before using this heavily in your C++ API.

12.1.2 在C++中实例化复合Python类型

字典对象可以通过`dict`构造函数来初始化：

```
using namespace pybind11::literals; // to bring in the `_a` literal
py::dict d("spam"_a=py::none(), "eggs"_a=42);
```

tuple对象可以通过`py::make_tuple()`来构造：

```
py::tuple tup = py::make_tuple(42, py::none(), "spam");
```

每个元素都会被转换为支持的Python类型，

simple namespace可以这样实例化：

```
using namespace pybind11::literals;  // to bring in the `_a` literal
py::object SimpleNamespace =
py::module_::import("types").attr("SimpleNamespace");
py::object ns = SimpleNamespace("spam"_a=py::none(), "eggs"_a=42);
```

namespace的属性可以通过 `py::delattr()`，`py::getattr()` 和 `py::setattr()` 来修改。Simple namespaces可以作为类实例的轻量级替代。

12.1.3 相互转换

混合编程时，通常需要将任意C++类型转换为Python类型，可以使用 `py::cast()` 来实现：

```
MyClass *cls = ...;
py::object obj = py::cast(cls);
```

反方向可以使用以下语法：

```
py::object obj = ...;
MyClass *cls = obj.cast<MyClass *>();
```

转换失败时，两个方向都会抛出 `cast_error` 异常。

12.1.4 在C++中访问Python库

在C++中也可以导入Python标准库或Python环境（sys.path)可找到的库的对象。示例如下：

```
// Equivalent to "from decimal import Decimal"
py::object Decimal = py::module_::import("decimal").attr("Decimal");

// Try to import scipy
py::object scipy = py::module_::import("scipy");
return scipy.attr("__version__");
```

12.1.5 调用Python函数

通过 `operator()` 可以调用Python类、函数和方法。

```
// Construct a Python object of class Decimal
py::object pi = Decimal("3.14159");

// Use Python to make our directories
py::object os = py::module_::import("os");
py::object makedirs = os.attr("makedirs");
makedirs("/tmp/path/to/somewhere");
```

One can convert the result obtained from Python to a pure C++ version if a `py::class_` or type conversion is defined.

```
py::function f = <...>;
py::object result_py = f(1234, "hello", some_instance);
MyClass &result = result_py.cast<MyClass>();
```

12.1.6 调用Python对象的方法

使用 `.attr` 可以调用对象的Python方法。

```
// Calculate e^π in decimal
py::object exp_pi = pi.attr("exp")();
py::print(py::str(exp_pi));
```

In the example above `pi.attr("exp")` is a *bound method*: it will always call the method for that same instance of the class. Alternately one can create an *unbound method* via the Python class (instead of instance) and pass the `self` object explicitly, followed by other arguments.

```
py::object decimal_exp = Decimal.attr("exp");

// Compute the e^n for n=0..4
for (int n = 0; n < 5; n++) {
    py::print(decimal_exp(Decimal(n));
}
```

12.1.7 关键字参数

支持关键字参数，Python语法示例如下：

```
def f(number, say, to):
    ...  # function code


f(1234, say="hello", to=some_instance)  # keyword call in Python
```

C++中则可以这样写：

```
using namespace pybind11::literals; // to bring in the `_a` literal
f(1234, "say"_a="hello", "to"_a=some_instance); // keyword call in C++
```

12.1.8 拆包参数

拆包参数 `*args` 和 `**kwargs` 可以与其他参数混合使用：

```
// * unpacking
py::tuple args = py::make_tuple(1234, "hello", some_instance);
f(*args);

// ** unpacking
py::dict kwargs = py::dict("number"_a=1234, "say"_a="hello",
"to"_a=some_instance);
f(**kwargs);

// mixed keywords, * and ** unpacking
py::tuple args = py::make_tuple(1234);
py::dict kwargs = py::dict("to"_a=some_instance);
f(*args, "say"_a="hello", **kwargs);
```

Generalized unpacking according to PEP448 is also supported:

```
py::dict kwargs1 = py::dict("number"_a=1234);
py::dict kwargs2 = py::dict("to"_a=some_instance);
f(**kwargs1, "say"_a="hello", **kwargs2);
```

12.1.9 隐式转换

当使用涉及Python类型的C++接口，或调用Python函数，返回 `object` 类型的对象时，会涉及到子类（如dict）的隐式转换。通过 `operator[]` 或 `obj.attr()` 返回代理对象也是如此。转型到子类可以提供代码的可读性，并允许向需要特定子类类型而不是通用 `object` 类型的C++函数传值。

```
#include <pybind11/numpy.h>
using namespace pybind11::literals;

py::module_ os = py::module_::import("os");
py::module_ path = py::module_::import("os.path");  // like 'import os.path
as path'
py::module_ np = py::module_::import("numpy");  // like 'import numpy as np'

py::str curdir_abs = path.attr("abspath")(path.attr("curdir"));
py::print(py::str("Current directory: ") + curdir_abs);
py::dict environ = os.attr("environ");
py::print(environ["HOME"]);
py::array_t<float> arr = np.attr("ones")(3, "dtype"_a="float32");
py::print(py::repr(arr + py::int_(1)));
```

对 `object` 子类的隐式转型，不需要向自定义类那样显式调用 `obj.cast()` 。

Note

If a trivial conversion via move constructor is not possible, both implicit and explicit casting (calling `obj.cast()`) will attempt a “rich” conversion. For instance, `py::list env = os.attr("environ");` will succeed and is equivalent to the Python code `env = list(os.environ)` that produces a list of the dict keys.

12.1.10 处理异常

Python异常将会包装为 `py::error_already_set` 后抛出。详见前面的章节“在C++中处理Python异常” 。

12.1.11 Gotchas

Default-Constructed Wrappers

通过包装类型的默认构造函数，不能得到有效的Python对象（不是 `py::none` ），和 `PyObject*` 空指针一样。可以通过 `static_cast<bool>(my_wrapper)` 来检查。

Assigning py::none() to wrappers

你可能想在C++函数中使用类似 `py::str` 和 `py::dict` 类型的参数，并给它们 `py::none` 默认值。但是，最好的情况是它会因为 `None` 无法转型为该类型而失败；最坏的情况是它会默默工作但会破坏你想要的类型（如Python中 `py::str(py::none)` 会返回None）。

12.2 NumPy

12.2.1 缓冲协议（buffer protocol）

Python支持插件库间以一种极其通用且便利方式进行数据交换。类型可以公开缓冲区视图，以提供对内部原始数据进行快速直接访问。假设我们想绑定下面的简单的Matrix类:

```
class Matrix {
public:
    Matrix(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;
};
```

下面的绑定代码将Matrix作为一个buffer对象公开，使得Matrices可以转型为NumPy arrays。甚至可以完全避免拷贝操作，类似python语句 `np.array(matrix_instance, copy = False)` 。

```
py::class_<Matrix>(m, "Matrix", py::buffer_protocol())
   .def_buffer([](Matrix &m) -> py::buffer_info {
        return py::buffer_info(
            m.data(),                               /* Pointer to buffer */
            sizeof(float),                          /* Size of one scalar */
            py::format_descriptor<float>::format(), /* Python struct-style
format descriptor */
            2,                                      /* Number of dimensions
*/
            { m.rows(), m.cols() },                 /* Buffer dimensions */
            { sizeof(float) * m.cols(),             /* Strides (in bytes) for
each index */
              sizeof(float) }
        );
    });
```

要使新类型支持缓冲协议，需要在 `py:class_` 构造函数中指定 `py::buffer_protocol()` 的标识，并调用 `def_buffer()` 方法定义一个通过给定的matrix实例创建 `py::buffer_info` 描述对象。 `py::buffer_info` 的内容反映了Python缓冲协议的规范。

```
struct buffer_info {
    void *ptr;
    py::ssize_t itemsize;
    std::string format;
    py::ssize_t ndim;
    std::vector<py::ssize_t> shape;
    std::vector<py::ssize_t> strides;
};
```

要想创建一个支持Python buffer对象为参数的C++函数，可以简单实用 `py::buffer` 作为函数参数之一。buffer对象会存在多种配置，因此通常在需要在函数体中进行安全检查。下面的例子，将展示如果定义一个双精度类型的Eigen矩阵的自定义构造函数，支持从兼容buffer对象来初始化（如NumPy matrix）。

```
/* Bind MatrixXd (or some other Eigen type) to Python */
typedef Eigen::MatrixXd Matrix;

typedef Matrix::Scalar Scalar;
constexpr bool rowMajor = Matrix::Flags & Eigen::RowMajorBit;

py::class_<Matrix>(m, "Matrix", py::buffer_protocol())
    .def(py::init([](py::buffer b) {
        typedef Eigen::Stride<Eigen::Dynamic, Eigen::Dynamic> Strides;

        /* Request a buffer descriptor from Python */
        py::buffer_info info = b.request();

        /* Some sanity checks ... */
        if (info.format != py::format_descriptor<Scalar>::format())
            throw std::runtime_error("Incompatible format: expected a double
array!");

        if (info.ndim != 2)
            throw std::runtime_error("Incompatible buffer dimension!");

        auto strides = Strides(
            info.strides[rowMajor ? 0 : 1] / (py::ssize_t)sizeof(Scalar),
            info.strides[rowMajor ? 1 : 0] / (py::ssize_t)sizeof(Scalar));

        auto map = Eigen::Map<Matrix, 0, Strides>(
            static_cast<Scalar *>(info.ptr), info.shape[0], info.shape[1],
strides);

        return Matrix(map);
    }));
```

作为参考，Eigen数据类型的 `def_buffer()` 方法类似于如下结构：

```
.def_buffer([](Matrix &m) -> py::buffer_info {
    return py::buffer_info(
        m.data(),                               /* Pointer to buffer */
        sizeof(Scalar),                         /* Size of one scalar */
        py::format_descriptor<Scalar>::format(), /* Python struct-style
format descriptor */
        2,                                      /* Number of dimensions */
        { m.rows(), m.cols() },                 /* Buffer dimensions */
        { sizeof(Scalar) * (rowMajor ? m.cols() : 1),
          sizeof(Scalar) * (rowMajor ? 1 : m.rows()) }
                                                /* Strides (in bytes) for
each index */
    );
 })
```

关于绑定Eigen类型更简单的方法(尽管有一些限制)，请参阅Eigen部分。

12.2.2 Arrays

将上述代码中的 `py::buffer` 替换为 `py::array` ，我们可以限制函数只接收NumPy array（而不是任意满足缓冲协议的Python类型）。

在很多场合，我们希望函数只接受特定数据类型的NumPy array，可以使用 `py::array_t<T>` 来实现。如下所示，函数需要一个双精度浮点类型的NumPy array。

```
void f(py::array_t<double> array);
```

当上面的函数被其他类型（如int）调用时，绑定代码将试图将输入转型为期望类型的NumPy array。该特性需要包含 `pybind11/numpy.h` 头文件。该文件不依赖与NumPy的头文件，因此可以独立于NumPy编译。运行时需要NumPy版本大于1.7.0。

NumPy array的数据并不保证密集排布；此外，数据条目可以以任意的行列跨度分隔。有时，我们需要函数仅接受C（行优先）或Fortran（列优先）次序的密集排布数组。这就需要指定第二个模板参数为 `py::array::c_style` 或 `py::array::f_style` 。

```
void f(py::array_t<double, py::array::c_style | py::array::forcecast> array);
```

`py::array::forcecast` 参数为第二个模板参数的默认值。它确保将不支持的参数转型为满足指定需要的数组，而不是试图匹配下一个函数数重载。

arrays有一些基于NumPy API的方法：

- `.dtype()` 返回数组元素的类型。
- `.strides()` 返回数组strides的指针。
- `.squeeze()` 从给定数组的形状中删除一维的条目。
- `.view(dtype)` 返回指定dtype类型的数组视图。
- `.reshape({i, j, ...})` 返回指定shape的数组视图。 `.resize({})` 也可以。
- `.index_at(i, j, ...)` 获取数组指定所以的元素。

还有几种获取引用的方法(如下所述)。

12.2.3 结构体类型

为了让 `py::array_t` 可以使用结构体类型，首先我们需要注册这个类型的内存布局。这可以通过 `PYBIND11_NUMPY_DTYPE` 宏来实现。

```
struct A {
    int x;
    double y;
};

struct B {
    int z;
    A a;
};

// ...
PYBIND11_MODULE(test, m) {
    // ...

    PYBIND11_NUMPY_DTYPE(A, x, y);
    PYBIND11_NUMPY_DTYPE(B, z, a);
    /* now both A and B can be used as template arguments to py::array_t */
}
```

结构体需要是由基础算术类型、 `std::complex` ，之前已经注册的子结构体类型， `arrays` 这些类型组成。支持C++数组和 `std::array` 。虽然有静态断言来防止不支持结构体类型的注册，使用者仍需负责地只使用plain结构体，这样可以安全的操作原始内存，而不会范围不变量。

12.2.4 向量化函数

假设我们想要将一个如下签名的函数绑定到Python，想让他既能接收常规参数，又能接收任意NumPy数组参数（向量、矩阵、多维数组）。

```
double my_func(int x, float y, double z);
```

包含 `pybind11/numpy.h` 后，这很好实现：

```
m.def("vectorized_func", py::vectorize(my_func));
```

这样将对数组中每个元素调用函数进行处理。与 `numpy.vectorize()` 一类方案相比，该方案显著的优势是：元素处理的循环完全在c++端运行，编译器可以将其压缩成一个紧凑的、优化后的循环。函数函数值将返回NumPy 数组类型 `numpy.dtype.float64` 。

```
x = np.array([[1, 3], [5, 7]])
y = np.array([[2, 4], [6, 8]])
z = 3
result = vectorized_func(x, y, z)
```

标量 `z` 将透明地复制4次。输入数组 `x` 和 `y` 将自动转型为正确的类型（从 `numpy.dtype.int64` 转到需要的 `numpy.dtype.int32` 和 `numpy.dtype.float32` ）。

Note

只有传值或常量引用的算术类型、复数、POD类型才能向量化，其他参数将原样传递。带右值引用参数的函数不能向量化。

如果计算太过复杂而无法对其进行量化，就需要手动创建和访问缓冲区内容。下面的代码展示了这该如何进行。(the code is somewhat contrived, since it could have been done more simply using `vectorize`).

```
#include <pybind11/pybind11.h>
#include <pybind11/numpy.h>

namespace py = pybind11;

py::array_t<double> add_arrays(py::array_t<double> input1,
py::array_t<double> input2) {
    py::buffer_info buf1 = input1.request(), buf2 = input2.request();

    if (buf1.ndim != 1 || buf2.ndim != 1)
        throw std::runtime_error("Number of dimensions must be one");

    if (buf1.size != buf2.size)
        throw std::runtime_error("Input shapes must match");

    /* No pointer is passed, so NumPy will allocate the buffer */
    auto result = py::array_t<double>(buf1.size);

    py::buffer_info buf3 = result.request();

    double *ptr1 = static_cast<double *>(buf1.ptr);
    double *ptr2 = static_cast<double *>(buf2.ptr);
    double *ptr3 = static_cast<double *>(buf3.ptr);

    for (size_t idx = 0; idx < buf1.shape[0]; idx++)
        ptr3[idx] = ptr1[idx] + ptr2[idx];

    return result;
}

PYBIND11_MODULE(test, m) {
    m.def("add_arrays", &add_arrays, "Add two NumPy arrays");
}
```

12.2.5 直接访问

出于性能方面的考虑，尤其是处理大型数组时，我们通常希望直接访问数组元素（已确定索引有效），而不需要在每次访问时进行内部维数和边界检查。为了规避这些检查，`array` 和 `array_t<T>` 模板类提供了不带检查的代理类 `unchecked<N>` 和 `mutable_unchecked<N>`，其中 `N` 为数组所需的维数。

```
m.def("sum_3d", [](py::array_t<double> x) {
    auto r = x.unchecked<3>(); // x must have ndim = 3; can be non-writeable
    double sum = 0;
    for (py::ssize_t i = 0; i < r.shape(0); i++)
        for (py::ssize_t j = 0; j < r.shape(1); j++)
            for (py::ssize_t k = 0; k < r.shape(2); k++)
                sum += r(i, j, k);
    return sum;
});
m.def("increment_3d", [](py::array_t<double> x) {
    auto r = x.mutable_unchecked<3>(); // Will throw if ndim != 3 or
flags.writeable is false
    for (py::ssize_t i = 0; i < r.shape(0); i++)
        for (py::ssize_t j = 0; j < r.shape(1); j++)
            for (py::ssize_t k = 0; k < r.shape(2); k++)
                r(i, j, k) += 1.0;
}, py::arg().noconvert());
```

要从 `array` 对象获取代理，你必须同时制定数据类型和维数作为模板参数，如 `auto r = myarray.mutable_unchecked<float, 2>()` 。

如果在编译时不知道维度的数量，则可以省略维度模板参数（像这样 `arr_t.unchecked()` 、 `arr.unchecked<T>()` ）。这同样可以工作，只是会导致代码优化较少而有略微新能损失。

注意，返回的代理类时直接引用array的数据，只在构造时读取shape, strides, writeable flag。您必须确保所引用的数组在返回对象的持续时间内不会被销毁或reshape, typically by limiting the scope of the returned instance.

The returned proxy object supports some of the same methods as `py::array` so that it can be used as a drop-in replacement for some existing, index-checked uses of `py::array`:

- `.ndim()` returns the number of dimensions
- `.data(1, 2, ...)` and `r.mutable_data(1, 2, ...)` returns a pointer to the `const T` or `T` data, respectively, at the given indices. The latter is only available to proxies obtained via `a.mutable_unchecked()`.
- `.itemsize()` returns the size of an item in bytes, i.e. `sizeof(T)`.
- `.ndim()` returns the number of dimensions.
- `.shape(n)` returns the size of dimension `n`

- `.size()` returns the total number of elements (i.e. the product of the shapes).
- `.nbytes()` returns the number of bytes used by the referenced elements (i.e. `itemsize()` times `size()`).

12.2.6 省略号

Python 3 provides a convenient `...` ellipsis notation that is often used to slice multidimensional arrays. For instance, the following snippet extracts the middle dimensions of a tensor with the first and last index set to zero. In Python 2, the syntactic sugar `...` is not available, but the singleton `Ellipsis` (of type `ellipsis`) can still be used directly.

```
a = ...  # a NumPy array
b = a[0, ..., 0]
```

The function `py::ellipsis()` function can be used to perform the same operation on the C++ side:

```
py::array a = /* A NumPy array */;
py::array b = a[py::make_tuple(0, py::ellipsis(), 0)];
```

12.2.7 内存视图

当我们只想提供C/C++ buffer的访问接口而不用构造类对象时，我们可以返回一个 `memoryview` 对象。假设我们希望公开 `2*4 uint8_t` 数组的 `memoryview` 时，可以这样做：

```
const uint8_t buffer[] = {
    0, 1, 2, 3,
    4, 5, 6, 7
};
m.def("get_memoryview2d", []() {
    return py::memoryview::from_buffer(
        buffer,                                    // buffer pointer
        { 2, 4 },                                  // shape (rows, cols)
        { sizeof(uint8_t) * 4, sizeof(uint8_t) }   // strides in bytes
    );
})
```

这样提供的`memoryview`不归Python管理，使用者有责任管理buffer的生命周期。在C++测删除buffer后继续使用创建的`memoryview`对象将导致未定义行为。

我们也可以使用`memoryview::from_memory`创建一个一维连续数组的内存视图：

```
m.def("get_memoryview1d", []() {
    return py::memoryview::from_memory(
        buffer,               // buffer pointer
        sizeof(uint8_t) * 8   // buffer size
    );
})
```

> Note: `memoryview::from_memory` is not available in Python 2.

12.3 实用工具

12.3.1 在C++中使用Python print函数

C++中通常使用`std::out`输出，而Python中则通常使用`print`。因为这些方法使用不同的缓冲区，混合使用它们可能会导致输出顺序问题。为解决这个问题，pybind11提供了`py::print`函数将输出写到Python的`sys.stdout`中。

函数包含了Python `print`一样的`sep`, `end`, `file`, `flush`等参数。

```
py::print(1, 2.0, "three"); // 1 2.0 three
py::print(1, 2.0, "three", "sep"_a="-"); // 1-2.0-three

auto args = py::make_tuple("unpacked", true);
py::print("->", *args, "end"_a="<-"); // -> unpacked True <-
```

12.3.2 从ostream捕获标准输出

C++库通常使用 `std::cout` 和 `std::cerr` 来打印输出，但它们和Python的标准 `sys.stdout` 和 `sys.stderr` 不能很好的协同工作。使用 `py::print` 代替库的打印是不现实的。我们可以将库函数的输出重定向到相应的Python streams来处理该问题：

```
#include <pybind11/iostream.h>

...

// Add a scoped redirect for your noisy code
m.def("noisy_func", []() {
    py::scoped_ostream_redirect stream(
        std::cout,                               // std::ostream&
        py::module_::import("sys").attr("stdout") // Python output
    );
    call_noisy_func();
});
```

Warning

`pybind11/iostream.h` 的实现不是线程安全的。多线程并发写入重定向的ostream将导致数据竞争和潜在的缓冲区溢出。因此，目前要求所有(可能的)并发重定向写入ostream都要有互斥锁保护。

此方法会对输出流进行刷新，并在 `scoped_ostream_redirect` 被销毁时根据需要进行刷新。这允许实时地重定向输出，比如输出到Jupyter notebook。C++流和Python输出这两个参数是可选的，不指定时默认为标准输出。`py::scoped_estream_redirect <scoped_estream_redirect>` 是作用于标准错误的。可以通过 `py::call_guard` 来简便设置。

```
// Alternative: Call single function using call guard
m.def("noisy_func", &call_noisy_function,
      py::call_guard<py::scoped_ostream_redirect,
                     py::scoped_estream_redirect>());
```

The redirection can also be done in Python with the addition of a context manager, using the `py::add_ostream_redirect() <add_ostream_redirect>` function:

```
py::add_ostream_redirect(m, "ostream_redirect");
```

The name in Python defaults to `ostream_redirect` if no name is passed. This creates the following context manager in Python:

```
with ostream_redirect(stdout=True, stderr=True):
    noisy_function()
```

It defaults to redirecting both streams, though you can use the keyword arguments to disable one of the streams if needed.

12.3.3 从字符串和文件执行Python表达式

pybind11 provides the `eval`, `exec` and `eval_file` functions to evaluate Python expressions and statements. The following example illustrates how they can be used.

```
// At beginning of file
#include <pybind11/eval.h>

...

// Evaluate in scope of main module
py::object scope = py::module_::import("__main__").attr("__dict__");

// Evaluate an isolated expression
int result = py::eval("my_variable + 10", scope).cast<int>();

// Evaluate a sequence of statements
py::exec(
    "print('Hello')\n"
    "print('world!');",
    scope);

// Evaluate the statements in an separate Python file on disk
py::eval_file("script.py", scope);
```

C++11 raw string literals are also supported and quite handy for this purpose. The only requirement is that the first statement must be on a new line following the raw string delimiter `R"(`, ensuring all lines have common leading indent:

```
py::exec(R"(
    x = get_answer()
    if x == 42:
        print('Hello World!')
    else:
        print('Bye!')
    )", scope
);
```

13. 内嵌解释器

虽然pybind11主要聚焦于使用C++扩展Python，反过来也是可以的，可以内嵌Python解释器到C++程序中。前面章节讲解的pybind11内容仍然适用。本节将介绍嵌入所需的一些额外内容。

13.1 准备开始

创建一个内嵌解释器的程序，可以在Cmake中添加 `pybind11::embed` 来支持。

```
cmake_minimum_required(VERSION 3.4)
project(example)

find_package(pybind11 REQUIRED)  # or `add_subdirectory(pybind11)`

add_executable(example main.cpp)
target_link_libraries(example PRIVATE pybind11::embed)
```

`main.cpp` 的基本结构如下：

```
#include <pybind11/embed.h> // everything needed for embedding
namespace py = pybind11;

int main() {
    py::scoped_interpreter guard{}; // start the interpreter and keep it
alive

    py::print("Hello, World!"); // use the Python API
}
```

需要在使用任意Python API前初始化解释器，包括pybind11 Python函数和类。RAII guard类 `scoped_interpreter` 可用来管理解释器的生命周期。在guard类销毁时，解释器将会关闭并占用的内存。必须在所有Python函数前调用它。

13.2 执行Python代码

在12.3.3节中，我们介绍了可以使用 `eval` ，`exec` 和 `eval_file` 函数来执行Python表达式或语句。下面的例子展示了附带解释器来执行Python代码的流程：

```
#include <pybind11/embed.h>
namespace py = pybind11;

int main() {
    py::scoped_interpreter guard{};

    py::exec(R"(
        kwargs = dict(name="World", number=42)
        message = "Hello, {name}! The answer is {number}".format(**kwargs)
        print(message)
    )");
}
```

也可以使用pybind11 API来实现相同的功能（参考12章）。

```
#include <pybind11/embed.h>
namespace py = pybind11;
using namespace py::literals;

int main() {
    py::scoped_interpreter guard{};

    auto kwargs = py::dict("name"_a="World", "number"_a=42);
    auto message = "Hello, {name}! The answer is
{number}"_s.format(**kwargs);
    py::print(message);
}
```

两种方法也可以混合使用：

```
#include <pybind11/embed.h>
#include <iostream>

namespace py = pybind11;
using namespace py::literals;

int main() {
    py::scoped_interpreter guard{};

    auto locals = py::dict("name"_a="World", "number"_a=42);
    py::exec(R"(
        message = "Hello, {name}! The answer is {number}".format(**locals())
    )", py::globals(), locals);

    auto message = locals["message"].cast<std::string>();
    std::cout << message;
}
```

12.3 导入模块

使用 `module_::import()` 可以导入Python模块。

```
py::module_ sys = py::module_::import("sys");
py::print(sys.attr("path"));
```

为方便起见，内嵌解释器时，会将当前工作路径包含到 `sys.path` 中。这样我们可以方便地导入本地Python文件。

```
"""calc.py located in the working directory"""


def add(i, j):
    return i + j
```

```
py::module_ calc = py::module_::import("calc");
py::object result = calc.attr("add")(1, 2);
int n = result.cast<int>();
assert(n == 3);
```

如果运行时源文件被修改（如被外部进程修改），可以使用 `module_::reload()` 重新导入模块。这在下面的场景中十分有用：有个应用程序要导入用户定义数据处理脚本，该脚本需要在用户修改后更新时。注意，这个函数不会递归地重新加载模块。

12.4 添加内嵌模块

使用宏 `PYBIND11_EMBEDDED_MODULE` 可以添加内嵌的二进制模块。这个定义需要放在全局作用域中。定义后，他们可以向其他模块一样导入。

```cpp
#include <pybind11/embed.h>
namespace py = pybind11;

PYBIND11_EMBEDDED_MODULE(fast_calc, m) {
    // `m` is a `py::module_` which is used to bind functions and classes
    m.def("add", [](int i, int j) {
        return i + j;
    });
}

int main() {
    py::scoped_interpreter guard{};

    auto fast_calc = py::module_::import("fast_calc");
    auto result = fast_calc.attr("add")(1, 2).cast<int>();
    assert(result == 3);
}
```

Unlike extension modules where only a single binary module can be created, on the embedded side an unlimited number of modules can be added using multiple `PYBIND11_EMBEDDED_MODULE` definitions (as long as they have unique names).

These modules are added to Python’s list of builtins, so they can also be imported in pure Python files loaded by the interpreter. Everything interacts naturally:

```
"""py_module.py located in the working directory"""
import cpp_module

a = cpp_module.a
b = a + 1
#include <pybind11/embed.h>
namespace py = pybind11;

PYBIND11_EMBEDDED_MODULE(cpp_module, m) {
    m.attr("a") = 1;
}

int main() {
    py::scoped_interpreter guard{};

    auto py_module = py::module_::import("py_module");

    auto locals = py::dict("fmt"_a="{} + {} = {}",
**py_module.attr("__dict__"));
    assert(locals["a"].cast<int>() == 1);
    assert(locals["b"].cast<int>() == 2);

    py::exec(R"(
        c = a + b
        message = fmt.format(a, b, c)
    )", py::globals(), locals);

    assert(locals["c"].cast<int>() == 3);
    assert(locals["message"].cast<std::string>() == "1 + 2 = 3");
}
```

12.5 解释器的生命周期

当 `scoped_interpreter` 销毁时，程序会自动关闭Python解释器。后面再创建一个新的示例会重启解释器。或者，我们也可以使用 `initialize_interpreter` / `finalize_interpreter` 这组函数在任意时刻直接设置解释器状态。

解释器重启后，pybind11创建的模块可以安全地重新初始化，但第三方扩展模块可能会有些问题。问题在于Python本身不能完全卸载扩展模块，并且会有一些解释器重启的警告。简而言之，由于Python引用循环或用户创建的全局数据，并非所有内存都可能被释放。具体细节可以查看CPython文档。

Warning

Creating two concurrent `scoped_interpreter` guards is a fatal error. So is calling `initialize_interpreter` for a second time after the interpreter has already been initialized.

Do not use the raw CPython API functions `Py_Initialize` and `Py_Finalize` as these do not properly handle the lifetime of pybind11’s internal data.

14. 杂项

14.1 关于便利宏的说明

pybind11提供了一些便利宏如 `PYBIND11_DECLARE_HOLDER_TYPE()` 和 `PYBIND11_OVERRIDE_*` 。由于这些宏只是在预处理中计算(预处理程序没有类型的概念)，它们会被模板参数中的逗号搞混。如：

```
PYBIND11_OVERRIDE(MyReturnType<T1, T2>, Class<T3, T4>, func)
```

预处理器会将其解释为5个参数（逗号分隔），而不是3个。有两种方法可以处理这个问题：使用类型别名，或者使用 `PYBIND11_TYPE` 包裹类型。

```
// Version 1: using a type alias
using ReturnType = MyReturnType<T1, T2>;
using ClassType = Class<T3, T4>;
PYBIND11_OVERRIDE(ReturnType, ClassType, func);

// Version 2: using the PYBIND11_TYPE macro:
PYBIND11_OVERRIDE(PYBIND11_TYPE(MyReturnType<T1, T2>),
                  PYBIND11_TYPE(Class<T3, T4>), func)
```

`PYBIND11_MAKE_OPAQUE` 宏不需要上述解决方案。

14.2 全局解释器锁（GIL）

在Python中调用C++函数时，默认会持有GIL。`gil_scoped_release` 和 `gil_scoped_acquire` 可以方便地在函数体中释放和获取GIL。这样长时间运行的C++代码可以通过Python线程实现并行化。示例如下：

```
class PyAnimal : public Animal {
public:
    /* Inherit the constructors */
    using Animal::Animal;

    /* Trampoline (need one for each virtual function) */
    std::string go(int n_times) {
        /* Acquire GIL before calling Python code */
        py::gil_scoped_acquire acquire;

        PYBIND11_OVERRIDE_PURE(
            std::string, /* Return type */
            Animal,      /* Parent class */
            go,          /* Name of function */
            n_times      /* Argument(s) */
        );
    }
};

PYBIND11_MODULE(example, m) {
    py::class_<Animal, PyAnimal> animal(m, "Animal");
    animal
        .def(py::init<>())
        .def("go", &Animal::go);

    py::class_<Dog>(m, "Dog", animal)
        .def(py::init<>());

    m.def("call_go", [](Animal *animal) -> std::string {
        /* Release GIL before calling into (potentially long-running) C++
code */
        py::gil_scoped_release release;
        return call_go(animal);
    });
}
```

我们可以使用 `call_guard` 策略来简化 `call_go` 的封装：

```
m.def("call_go", &call_go, py::call_guard<py::gil_scoped_release>());
```

14.3 通过多个模块来划分代码

通常我们可以直接将绑定代码分隔到多个模块中，即便模块引用的类型在其他模块中定义。有个例外场景，就是当前扩展的类型定义在其他模块中，参见下面的例子：

```
py::class_<Pet> pet(m, "Pet");
pet.def(py::init<const std::string &>())
   .def_readwrite("name", &Pet::name);

py::class_<Dog>(m, "Dog", pet /* <- specify parent */)
    .def(py::init<const std::string &>())
    .def("bark", &Dog::bark);
```

假设 `Pet` 类的绑定定义在 `basic` 模块中，而 `Dog` 绑定定义在其他模块。在 `class_<Dog>` 中明确与 `Pet` 类的继承关系时需要知道 `Pet` ，问题是在其他模块定义的 `Pet` 不再对 `Dog` 可见。我们可以这样处理：

```
py::object pet = (py::object) py::module_::import("basic").attr("Pet");

py::class_<Dog>(m, "Dog", pet)
    .def(py::init<const std::string &>())
    .def("bark", &Dog::bark);
```

或者，你可以将基类作为模板参数给 `class_` ，让pybind11自动查找到相应的Python类型。但也需要调用一次 `import` 函数，确保 `basic` 模块的绑定代码已经执行。

```
py::module_::import("basic");

py::class_<Dog, Pet>(m, "Dog")
    .def(py::init<const std::string &>())
    .def("bark", &Dog::bark);
```

如果存在循环依赖时，上述两种方法都将失效。

注意，pybind11代码在编译时会默认隐藏符号的可见性（如通过GCC/Clang的 `-fvisibility=hidden` 标识），这会干扰访问在其他模块定义的类型的能力。这需要通过手动导出需要被其他模块访问的类型，像这样：

```
class PYBIND11_EXPORT Dog : public Animal {
    ...
};
```

在运行时也可以共享任意的C++对象，尽管很少用到该特性。使用capsule机制在模块间共享内部库数据，可以用来存储、修改、访问用户自定义数据。注意，一个模块能够看到其他模块的数据，仅在他们使用相同的pybind11版本编译时才能实现。参考下面的例子：

```
auto data = reinterpret_cast<MyData *>(py::get_shared_data("mydata"));
if (!data)
    data = static_cast<MyData *>(py::set_shared_data("mydata", new
MyData(42)));
```

如果在几个单独编译的扩展模块中使用了上述代码段，第一个导入的模块将创建 `MyData` 实例，并和指针联系起来。后续导入的模块就可以访问该指针指向的数据了。

14.4 模块析构

pybind11没有提供明确的机制在模块析构时调用清理代码。在少数需要该功能的场景下，可以使用Python capsules或析构回调函数的弱引用来模仿它。

```
auto cleanup_callback = []() {
    // perform cleanup here -- this function is called with the GIL held
};

m.add_object("_cleanup", py::capsule(cleanup_callback));
```

该方法一个潜在地缺陷是，在cleanup函数调用时，模块公开的类实例可能仍存活着（这是否可以接受通常取决于应用程序）。

或者，我们可以将capsule存储在类型对象中，确保它不会在回收该类型的所有实例之前被调用：

```
auto cleanup_callback = []() { /* ... */ };
m.attr("BaseClass").attr("_cleanup") = py::capsule(cleanup_callback);
```

上面的方法都在Python中暴露了一个 `_cleanup` 的危险属性，从API的角度来看，这种做法并不受欢迎（Python过早的显式调用它可能会导致未定义行为）。这可以通过使用cleanup函数回调的弱引用来规避。

```
// Register a callback function that is invoked when the BaseClass object is
collected
py::cpp_function cleanup_callback(
    [](py::handle weakref) {
        // perform cleanup here -- this function is called with the GIL held

        weakref.dec_ref(); // release weak reference
    }
);

// Create a weak reference with a cleanup callback and initially leak it
(void) py::weakref(m.attr("BaseClass"), cleanup_callback).release();
```

15. [illegible]

15.1 “ImportError: dynamic module does not define init function”

1. [illegible]`PYBIND11_MODULE`[illegible].so[illegible]
2. [illegible]`Python`[illegible]`Python2`[illegible]`Python3`[illegible]

15.2 “Symbol not found: __Py_ZeroStruct _PyInstanceMethod_Type”

[illegible]15.1

15.3 “SystemError: dynamic module not initialized properly”

[illegible]15.1

15.4 [illegible]Python[illegible]

[illegible]15.1

15.5[illegible]

[illegible]`C++`[illegible]

```
void increment(int &i)
{
    i++;
}
void increment_ptr(int *i)
{
    (*i)++;
}
```

在 `Python` 中，所有参数都是通过引用传递的，因此从Python绑定此类代码时一般不会有问题。 但是，某些基本 `Python` 类型（如 `str` ， `int` ， `bool` ， `float` 等）是不可变的。这意味着尝试将函数移植到 `Python` 后，对调用者提供的值不会产生相同的影响，事实上，它啥都没干。

```
def increment(i):
    i += 1  # nope..
```

`pybind11` 也受到此类语言级约定的影响，这意味着绑定 `increment` 或者 `increment_ptr` 还将创建不修改其参数的 `Python` 函数。虽然不方便，但一种解决方法是将不可变类型封装在允许修改的自定义类型中影响。另一种选择涉及绑定一个小包装 `lambda` 函数，该函数返回一个包含所有输出参数的元组（有关绑定 `lambda` 函数的示例，请参阅文档的其余部分）。例如：

```
int foo(int &i)
{
    i++;
    return 123;
}
```

绑定代码为：

```
m.def("foo",
    [](int i) {
        int rv = foo(i);
        return std::make_tuple(rv, i);
    });
```

15.6 如何减少编译时间？

在多个文件上拆分绑定代码是一种很好的做法，如下例所示： `example.cpp`

```
void init_ex1(py::module_ &);
void init_ex2(py::module_ &);
/* ... */
PYBIND11_MODULE(example, m) {
    init_ex1(m);
    init_ex2(m);
    /* ... */
}
```

`ex1.cpp`：

```
void init_ex1(py::module_ &m) {
    m.def("add",
        [](int a, int b) {
            return a + b;
        });
}
```

`ex2.cpp`：

```
void init_ex2(py::module_ &m) {
    m.def("sub",
    [](int a, int b) {
        return a - b;
    });
}
```

`python` 调用：

```
import example

example.add(1, 2)  # 3
example.sub(1, 1)  # 0
```

[illegible]`init_ex`[illegible]

1. [illegible]
2. [illegible]
3. [illegible]

15.7 “recursive template instantiation exceeded maximum depth of 256”

[illegible]`GCC/Clang`[illegible]`-ftemplate-depth=1024`[illegible]C++14[illegible]

15.8 “‘SomeClass’ declared with greater visibility than the type of its field ‘SomeClass::member’ [-Wattributes]”

[illegible]`-fvisibility`[illegible].pybind11[illegible] pybind[illegible]`py::object`[illegible]`py::list`[illegible]pybind[illegible]`-fvisibility=hidden`[illegible] [illegible]`-fvisibility=hidden`[illegible]`pybind`[illegible]`pybind`[illegible] [illegible]`Python`[illegible]`POSIX`[illegible]`RTLD_local`[illegible]`dlopen`[illegible]`Python`[illegible] [illegible]`-fvisibility=hidden`[illegible],[illegible]. [illegible]`-fvisibility=hidden`[illegible].)

15.9 [illegible]?

[illegible]`pybind11` [illegible]

```
__ZN8pybind1112cpp_functionC1Iv8Example2JRNSt3__16vectorINS3_12basic_stringIw
NS3_
11char_traitsIwEENS3_9allocatorIwEEEENS8_ISA_EEEEEJNS_4nameENS_7siblingENS_9i
s_
methodEA28_cEEEMT0_FT_DpT1_EDpRKT2_
```

这是以下函数类型的展开形式：

```
pybind11::cpp_function::cpp_function<void, Example2,
std::__1::vector<std::__1::basic_
string<wchar_t, std::__1::char_traits<wchar_t>, std::__1::allocator<wchar_t>
>,
std::__1::allocator<std::__1::basic_string<wchar_t,
std::__1::char_traits<wchar_t>,
std::__1::allocator<wchar_t> > > >&, pybind11::name, pybind11::sibling,
pybind11::is_method, char [28]>(void (Example2::*)
(std::__1::vector<std::__1::basic_
string<wchar_t, std::__1::char_traits<wchar_t>, std::__1::allocator<wchar_t>
>,
std::__1::allocator<std::__1::basic_string<wchar_t,
std::__1::char_traits<wchar_t>,
std::__1::allocator<wchar_t> > > >&), pybind11::name const&,
pybind11::sibling
const&, pybind11::is_method const&, char const (&) [28])
```

仅存储此函数的错位名称（196 字节）所需的内存大于它所代表的实际代码段（111 字节）！另一方面，甚至给这个函数起个名字都是愚蠢的———毕竟，它只是一个更大的机器中的一个小齿轮，不暴露于外界。因此，我数 `-fvisibility=hidden` 来实现，它将默认符号可见性设置为隐藏，这对生成的扩展库的最终二进制大小有巨大影响。（在 `Visual Studio` 上，默认情况下符号已隐藏，因此无需在此处进行任何操作。）除了减小二进制大小之外， `-fvisibility=hidden` 还可以避免在加载多个模块时出现潜在的严重问题，并且是正确 pybind 们通常只想为那些实际从外部调用的函数导出符号。这可以通过为 `GCC` 和 `Clang` 指定参 操作所必需的。有关更多详细信息，请参阅之前的常见问题解答条目。

15.10 使用古老的基于Windows的Visual Studio 2008

`Python` 的官方 `Windows` 发行版是使用缺乏良好 `C++11` 支持的真正古老版本的 `Visual Studio` 编译的。一些用户隐含地假设不可能将使用 `Visual Studio 2015` 构建的插件加载到使用 `Visual Studio 2008` 编译的 `Python` 发行版中。但

[illegible] `DLL` [illegible]/[illegible] `C` [illegible] `malloc()` [illegible] `free()` [illegible] `ABI` [illegible] `pybind11` [illegible]

15.11 [illegible]Ctrl-C[illegible]

`Ctrl-C` [illegible] `Python` [illegible] `GIL` [illegible] PyErr_CheckSignals() [illegible] Python [illegible] py::error_already_set [illegible] KeyboardInterrupt[illegible]

```
PYBIND11_MODULE(example, m){
    m.def("long running_func",
        [](){
            for (;;)
            {
                if (PyErr_CheckSignals() != 0)
                throw py::error_already_set();
                // Long running iteration
            }
            });
}
```

15.12 CMake[illegible]Python[illegible]

[illegible] `CMake` [illegible] `Python` [illegible] `Python` [illegible] `CMakeCache.txt` [illegible] `-DPYTHON_EXECUTABLE=$(which python)` [illegible] `CMake` [illegible] `$(which python)` [illegible] `python` [illegible] `-DPYBIND11_FINDPYTHON=ON` [illegible] `CMake FindPython` [illegible] `pybind11` [illegible] `CMake 3.12+[illegible]3.15+` [illegible] `3.18.2+` [illegible] `pybind11` [illegible] `CMakeLists.txt` [illegible]

15.13 CMake和pybind11中Python版本检测不一致

`CMake` 提供的用于 `Python` 版本检测的函数 `find_package(PythonInterp)` 和 `find_package(PythonLibs)` 被 `pybind11` 修改，原因是它们不适合 `pybind11` 的需要。相反， `pybind11` 提供了自己的、更可靠的 `Python` 检测 `CMake` 代码。但是，当在安装了多个 `Python` 版本的系统中使用 `CMake Python` 检测的项目中使用 `pybind11` 时，可能会出现冲突。 如果在同一个项目中使用这两种机制，这种差异可能会导致不一致和错误。考虑在安装了 `Python2.7` 和 `3.x` 的系统中执行的以下 `CMake` 代码：

```
find_package(PythonInterp)
find_package(PythonLibs)
find_package(pybind11)
```

它将检测 `Python2.7` ， `pybind11` 也会选择它。 相比之下，这段代码：

```
find_package(pybind11)
find_package(PythonInterp)
find_package(PythonLibs)
```

将为 `pybind11` 检测 `Python3.x` ，之后可能会在 `find_package(PythonLibs)` 上崩溃。 有三种可能的解决方案：

1. 避免使用 `CMake` 中的 `find_package(PythonInterp)` 和 `find_package(PythonLibs)` 并依赖 `pybind11` 检测 `Python` 版本。如果这不可能，则应在包含 `pybind11` 之前调用 `CMake` 机器。
2. 将 `PYBIND11_FINDPYTHON` 设置为 `True` 或在现代 `CMake` 上使用 `find_package(Python COMPONENTS Interpreter Development)` （3.12+，3.15+更好，3.18.2+ 最好）。在这些情况下， `Pybind11` 使用新的 `CMake FindPython` 而不是旧的、已弃用的搜索工具，并且这些模块在查找正确的 `Python` 方面要好得多。
3. 将 `PYBIND11_NOPYTHON` 设置为 `TRUE` ° `Pybind11` 不会搜索 `Python` 。但是，您将不得不使用基于目标的系统，并自己进行更多设置，因为它不知道或不包含依赖于 `Python` 的东西，例如 `pybind11_add_module` 。这可能非常适合集成到现有系统中，例如 `scikit-build` 的 `Python` 助手。

15.14 如何引用这个项目?

我们建议使用以下 BibTeX 模板在科学话语中引用 pybind11：

```
@misc{pybind11,
author = {Wenzel Jakob and Jason Rhinelander and Dean Moldovan},
year = {2017},
note = {https://github.com/pybind/pybind11},
title = {pybind11 -- Seamless operability between C++11 and Python} }
```

16. 案例

16.1 c/c++基本类型传递

- c基本类型

```
char ca;
unsigned char uca;
```

- c++基本类型和模板

```
vector
stl
```

pybind11使用指南

1. 基础用法

1.1 安装与编译

在安装python3-dev和下载了pybind11源码的前提下，可以直接include pybind11头文件目录和python3头文件目录即可。cmake示例如下:

```
set(PYTHON_TARGET_VER 3.6)
find_package(PythonInterp ${PYTHON_TARGET_VER} EXACT)
find_package(PythonLibs ${PYTHON_TARGET_VER} EXACT REQUIRED)

include_directories(pybind11_include_path)
include_directories(${PYTHON_INCLUDE_DIRS})
```

1.2 绑定函数

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

int add(int i, int j) {
    return i + j;
}

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

宏 `PYBIND11_MODULE` 会创建模块初始化函数，它在Python中 `import` 模块时被调用。其参数分别是模块名，类型为 `py::module_` 的变量（m），是创建绑定的主要接口。 `module_::def()` 方法，可以生成函数的绑定。

1.2.1 关键字参数

使用 `py::arg` 可以指定函数的参数名，Python侧调用函数时可以使用关键字参数，以增加代码的可读性。

```
m.def("add", &add, "A function which adds two numbers",
      py::arg("i"), py::arg("j"));
```

更简洁的写法：

```
// regular notation
m.def("add1", &add, py::arg("i"), py::arg("j"));
// shorthand
using namespace pybind11::literals;
m.def("add2", &add, "i"_a, "j"_a);
```

Python使用示例：

```
import example
example.add(i=1, j=2)  #3L
```

1.2.2 参数默认值

pybind11不能自动地提取默认参数，因为它不属于函数类型信息的一部分。我们需要借助 `arg` 在绑定时指定参数默认值：

```
m.def("add", &add, "A function which adds two numbers",
      py::arg("i") = 1, py::arg("j") = 2);
```

更简短的声明方式：

```
// regular notation
m.def("add1", &add, py::arg("i") = 1, py::arg("j") = 2);
// shorthand
m.def("add2", &add, "i"_a=1, "j"_a=2);
```

1.2.3 重载函数

重载方法即拥有相同的函数名，但入参不一样的函数。

在绑定重载函数时，我们需要增加函数签名相关的信息以消除歧义。绑定多个函数到同一个Python名称，将会自动创建函数重载链。Python将会依次匹配，找到最合适的重载函数。

```
m.def("add", static_cast<int(*)(int, int)>(&add), "A function which adds two int numbers");
m.def("add", static_cast<double(*)(double, double)>(&add), "A function which adds two double numbers");
```

如果你的编译器支持C++14，也可以使用下面的语法来转换重载函数：

```
m.def("add", py::overload_cast<int, int>(&add), "A function which adds two int numbers");
m.def("add", py::overload_cast<double, double>(&add), "A function which adds two double numbers");
```

这里，`py::overload_cast`仅需指定函数类型，不用给出返回值类型，以避免原语法带来的不必要的干扰(`void (Pet::*)`)。如果是基于const的重载，需要使用`py::const`标识。

1.3 导出变量

我们可以使用`attr`函数来注册需要导出到Python模块中的C++变量。内建类型和常规对象会在指定attriutes时自动转换，也可以使用`py::cast`来显式转换。

~~~
PYBIND11_MODULE(example, m) {
    m.attr("the_answer") = 42;
    py::object world = py::cast("World");
    m.attr("what") = world;
}
``


Python⊠⊠⊠⊠⊠⊠
```pyhton
>>> import example
>>> example.the_answer
42
>>> example.what
'World'
~~~

1.4 绑定类或结构体

现在我们来绑定一个C++自定义数据结构`Pet`。定义如下：

```
struct Pet {
    Pet(const std::string &name) : name(name) { }
    void setName(const std::string &name_) { name = name_; }
    const std::string &getName() const { return name; }

    std::string name;
};
```

绑定代码如下所示：

```
#include <pybind11/pybind11.h>
namespace py = pybind11;

PYBIND11_MODULE(example, m) {
    py::class_<Pet>(m, "Pet")
        .def(py::init<const std::string &>())
        .def("setName", &Pet::setName)
        .def("getName", &Pet::getName)
        .def("__repr__",
            [](const Pet &a) {
                return "<example.Pet named '" + a.name + "'>";
            });
}
```

`class_` 会创建C++ class或 struct的绑定。`init()` 方法用于创建绑定类的构造函数，它使用类构造函数的参数类型作为模板参数，并包装相应的构造函数。

使用 `print(p)` 打印对象信息时，默认会得到一些没用的信息。我们可以绑定一个工具函数到 `__repr__` 方法，来返回可读性好的摘要信息。在不改变Pet类的基础上，使用一个匿名函数来完成这个功能是一个不错的选择。

Python使用示例如下；

```
>>> import example
>>> p = example.Pet("Molly")
>>> print(p)
<example.Pet named 'Molly'>
>>> p.getName()
u'Molly'
>>> p.setName("Charly")
>>> p.getName()
u'Charly'
```

静态成员函数需要使用 `class_::def_static` 来绑定。

1.4.1 成员函数

使用 `class_::def_readwrite` 方法可以导出公有成员变量，使用 `class_::def_readonly` 方法则可以导出只读成员。

```
py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string &>())
    .def_readwrite("name", &Pet::name)
    // ... remainder ...
```

Python中使用示例如下：

```
>>> p = example.Pet("Molly")
>>> p.name
u'Molly'
>>> p.name = "Charly"
>>> p.name
u'Charly'
```

假设 `Pet::name` 是一个私有成员变量，向外提供setter和getters方法。

```
class Pet {
public:
    Pet(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;
};
```

可以使用 `class_::def_property()`（只读成员使用 `class_::def_property_readonly()`）来定义并私有成员，并生成相应的setter和geter方法：

```
py::class_<Pet>(m, "Pet")
    .def(py::init<const std::string &>())
    .def_property("name", &Pet::getName, &Pet::setName)
    // ... remainder ...
```

只写属性通过将read函数定义为nullptr来实现。

相似的方法 `class_::def_readwrite_static()`，`class_::def_readonly_static()` `class_::def_property_static()`，`class_::def_property_readonly_static()` 用于绑定静态变量和属性。

1.4.2 动态属性

原生的Pyhton类可以动态地获取新属性：

```
>>> class Pet:
...     name = "Molly"
...
>>> p = Pet()
>>> p.name = "Charly"  # overwrite existing
>>> p.age = 2  # dynamically add a new attribute
```

默认情况下，从C++导出的类不支持动态属性，其可写属性必须是通过 `class_::def_readwrite` 或 `class_::def_property` 定义的。试图设置其他属性将产生错误：

```
>>> p = example.Pet()
>>> p.name = "Charly"  # OK, attribute defined in C++
>>> p.age = 2  # fail
AttributeError: 'Pet' object has no attribute 'age'
```

要让C++类也支持动态属性，我们需要在 `py::class_` 的构造函数添加 `py::dynamic_attr` 标识：

```
py::class_<Pet>(m, "Pet", py::dynamic_attr())
    .def(py::init<>())
    .def_readwrite("name", &Pet::name);
```

这样，之前报错的代码就能够正常运行了。

```
>>> p = example.Pet()
>>> p.name = "Charly"  # OK, overwrite value in C++
>>> p.age = 2  # OK, dynamically add a new attribute
>>> p.__dict__  # just like a native Python class
{'age': 2}
```

需要提醒一下，支持动态属性会带来小小的运行时开销。不仅仅因为增加了额外的 `__dict__` 属性，还因为处理循环引用时需要花费更多的垃圾收集跟踪花销。但是不必担心这个问题，因为原生Python类也有同样的开销。默认情况下，pybind11导出的类比原生Python类效率更高，使能动态属性也只是让它们处于同等水平而已。

1.4.3 重载方法

重载类的方法同上一节的普通函数重载，这里举个实例仅供参考：

```
struct Pet {
    Pet(const std::string &name, int age) : name(name), age(age) { }

    void set(int age_) { age = age_; }
    void set(const std::string &name_) { name = name_; }

    std::string name;
    int age;
};

// method 1
py::class_<Pet>(m, "Pet")
   .def(py::init<const std::string &, int>())
   .def("set", static_cast<void (Pet::*)(int)>(&Pet::set), "Set the pet's
age")
   .def("set", static_cast<void (Pet::*)(const std::string &)>(&Pet::set),
"Set the pet's name");

// method 2
py::class_<Pet>(m, "Pet")
    .def("set", py::overload_cast<int>(&Pet::set), "Set the pet's age")
    .def("set", py::overload_cast<const std::string &>(&Pet::set), "Set the
pet's name");
```

1.5 绑定枚举类型

对于C风格的枚举类型，绑定示例如下：

```
enum Flags {
    Read = 4,
    Write = 2,
    Execute = 1
};

py::enum_<Flags>(m, "Flags", py::arithmetic())
    .value("Read", Flags::Read)
    .value("Write", Flags::Write)
    .value("Execute", Flags::Execute)
    .export_values();
```

`enum_::export_values()` 用来导出枚举项到父作用域，C++11的强枚举类型需要跳过这点。

枚举类型的枚举项会被导出到类 `__members__` 属性中，`name` 属性可以返回枚举值的名称的unicode字符串，`str(enum)` 也可以做到，但两者的实现目标不同。

1.6 接收*args和**kwargs参数

Python的函数可以接收任意数量的参数和关键字参数：

```
def generic(*args, **kwargs):
    ...  # do something with args and kwargs
```

我们也可以通过pybind11来创建这样的函数：

```
void generic(py::args args, const py::kwargs& kwargs) {
    /// .. do something with args
    if (kwargs)
        /// .. do something with kwargs
}

/// Binding code
m.def("generic", &generic);
```

`py::args` 继承自 `py::tuple`，`py::kwargs` 继承自 `py::dict`。

2. [illegible]

2.1 [illegible]

Python[illegible]C++[illegible]no-trivial[illegible]Python[illegible]C++[illegible]pybind11[illegible]`model::def()`[illegible]`class_def()`[illegible]`return_value_policy::automatic`[illegible]

[illegible]

```
/* Function declaration */
Data *get_data() { return _data; /* (pointer to a static data structure) */ }
...

/* Binding code */
m.def("get_data", &get_data); // <-- KABOOM, will cause crash when called
from Python
```

[illegible]Python[illegible]`get_data()`[illegible]C++[illegible]Python[illegible]`return_value_policy::automatic`[illegible]pybind11[illegible]`_data`[illegible]

[illegible]Python[illegible]`_data`[illegible]Python[illegible]pybind11[illegible]C++[illegible]operator delete()[illegible]

[illegible]`return_value_policy::reference`[illegible]

```
m.def("get_data", &get_data, py::return_value_policy::reference);
```

[illegible]pybind11[illegible]

[illegible]	[illegible]
`return_value_policy::take_ownership`	[illegible]0[illegible]Pyhton[illegible]delete[illegible]
`return_value_policy::copy`	[illegible]Python[illegible]
`return_value_policy::move`	[illegible]`std::move`[illegible]Python[illegible]
`return_value_policy::reference`	[illegible]C++[illegible]Pyth [illegible]C++[illegible]
`return_value_policy::reference_internal`	[illegible]`this`[illegible]`self`[illegible]referenc [illegible]`keep_alive<0, 1>`[illegible]Python[illegible]`def_property`[illegible]`def_readwrite`[illegible]getter[illegible]
`return_value_policy::automatic`	[illegible] `return_value_policy::take_owner` [illegible] `return_value_policy::copy` [illegible] `py::cla` [illegible]
`return_value_policy::automatic_reference`	[illegible] `return_value_policy::reference` [illegible]C++[illegible]Python[illegible] `pybind11/stl.h` [illegible]casters[illegible]

[illegible]

```
class_<MyClass>(m, "MyClass")
    .def_property("data", &MyClass::getData, &MyClass::setData,
                  py::return_value_policy::copy);
```

在技术层面，上述代码会将策略同时应用于getter和setter函数，但是setter函数并不关心返回值策略，这样做仅仅出于语法简洁的考虑。或者，你可以通过 `cpp_function` 构造函数来传递目标参数：

```
class_<MyClass>(m, "MyClass")
    .def_property("data"
        py::cpp_function(&MyClass::getData, py::return_value_policy::copy),
        py::cpp_function(&MyClass::setData)
    );
```

注意：代码使用无效的返回值策略将导致未初始化内存或多次free数据结构，这将导致难以调试的、不确定的问题和段错误。因此，花点时间来理解上面表格的各个选项是值得的。

提示：

1. 上述策略的另一个重点是，他们仅可以应用于pybind11还不知晓的实例，这时策略将澄清返回值的生命周期和所有权问题。当pybind11已经知晓参数（通过其在内存中的类型和地址来识别），它将返回已存在的Python对象封装，而不是创建一份拷贝。
2. 下一节将讨论上面表格之外的调用策略，他涉及到返回值和函数参数的引用关系。
3. 可以考虑使用智能指针来代替复杂的调用策略和生命周期管理逻辑。智能指针会告诉你一个对象是否仍被C++或Python引用，这样就可以消除各种可能引发crash或未定义行为的矛盾。对于返回智能指针的函数，没必要指定返回值策略。

2.2 附加的调用策略

除了以上的返回值策略外，进一步指定调用策略可以表明参数间的依赖关系，确保函数调用的稳定性。

保活（keep alive）

当一个C++容器对象包含另一个C++对象时，我们需要使用该策略。`keep_alive<Nurse, Patient>` 表明至少在索引Nurse被回收前，索引Patient应该被保活。0表示返回值，1及以上表示参数索引。1表示隐含的参数this指针，而常规参数索引从2开始。当Nurse的值在运行前被检测到为None时，调用策略将什么都不做。

当nurse不是一个pybind11注册类型时，实现依赖于创建对nurse对象弱引用的能力。如果nurse对象不是pybind11注册类型，也不支持弱引用，程序将会抛出异常。

如果你使用一个错误的参数索引，程序将会抛出“Could not cativate keep_alive!“警告的运行时异常。这时，你应该review你代码中使用的索引。

参见下面的例子：一个list append操作，将新添加元素的生命周期绑定到添加的容器对象上：

```
py::class_<List>(m, "List").def("append", &List::append, py::keep_alive<1, 2>
());
```

为了一致性，构造函数的实参索引也是相同的。索引1仍表示this指针，索引0表示返回值（构造函数的返回值被认为是void）。下面的示例将构造函数入参的生命周期绑定到被构造对象上。

```
py::class_<Nurse>(m, "Nurse").def(py::init<Patient &>(), py::keep_alive<1, 2>
());
```

> Note: `keep_alive` 与Boost.Python中的 `with_custodian_and_ward` 和 `with_custodian_and_ward_postcall` 相似。

Call guard

`call_guard<T>` 策略允许任意T类型的scope guard应用于整个函数调用。示例如下：

```
m.def("foo", foo, py::call_guard<T>());
```

上面的代码等价于：

```
m.def("foo", [](args...) {
    T scope_guard;
    return foo(args...); // forwarded arguments
});
```

仅要求模板参数T是可构造的，如 `gil_scoped_release` 就是一个非常有用的类型。

`call_guard` 支持同时制定多个模板参数，`call_guard<T1, T2, T3 ...>` 。构造顺序是从左至右，析构顺序则相反。

> See also: `test/test_call_policies.cpp` 含有更丰富的示例来展示 `keep_alive` 和 `call_guard` 的用法。

2.3 Keyword-only参数

Python3提供了keyword-only参数（在函数定义中使用 `*` 作为匿名参数）：

```
def f(a, *, b):  # a can be positional or via keyword; b must be via keyword
    pass

f(a=1, b=2)  # good
f(b=2, a=1)  # good
f(1, b=2)  # good
f(1, 2)  # TypeError: f() takes 1 positional argument but 2 were given
```

pybind11提供了 `py::kw_only` 对象来实现相同的功能：

```
m.def("f", [](int a, int b) { /* ... */ },
      py::arg("a"), py::kw_only(), py::arg("b"));
```

注意，该特性不能与 `py::args` 一起使用。

2.4 Positional-only参数

python3.8引入了Positional-only参数语法，pybind11通过 `py::pos_only()` 来提供相同的功能：

```
m.def("f", [](int a, int b) { /* ... */ },
       py::arg("a"), py::pos_only(), py::arg("b"));
```

现在，你不能通过关键字来给定 `a` 参数。该特性可以和keyword-only参数一起使用。

2.5 Non-converting参数

有些参数可能支持类型转换，如:

- 通过 `py::implicitly_convertible<A,B>()` 进行隐式转换
- 将整形变量传给入参为浮点类型的函数
- 将非复数类型（如float）传给入参为 `std::complex<float>` 类型的函数
- Calling a function taking an Eigen matrix reference with a numpy array of the wrong type or of an incompatible data layout.

有时这种转换并不是我们期望的，我们可能更希望绑定代码抛出错误，而不是转换参数。通过 `py::arg` 来调用 `.noconvert()` 方法可以实现这个事情。

```
m.def("floats_only", [](double f) { return 0.5 * f; },
py::arg("f").noconvert());
m.def("floats_preferred", [](double f) { return 0.5 * f; }, py::arg("f"));
```

尝试进行转换时，将抛出 `TypeError` 异常:

```
>>> floats_preferred(4)
2.0
>>> floats_only(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: floats_only(): incompatible function arguments. The following
argument types are supported:
    1. (f: float) -> float

Invoked with: 4
```

该方法可以与缩写符号 `_a` 和默认参数配合使用，像这样 `py::arg().noconvert()` 。

3. 类绑定进阶

3.1 继承与多态

现在有两个具有继承关系的类：

```
struct Pet {
    Pet(const std::string &name) : name(name) { }
    std::string name;
};

struct Dog : Pet {
    Dog(const std::string &name) : Pet(name) { }
    std::string bark() const { return "woof!"; }
};
```

pybind11提供了两种方法来指明继承关系：1）将C++基类作为派生类 `class_` 的模板参数；2）将基类名作为 `class_` 的参数绑定到派生类。两种方法是等效的。

```
py::class_<Pet>(m, "Pet")
   .def(py::init<const std::string &>())
   .def_readwrite("name", &Pet::name);

// Method 1: template parameter:
py::class_<Dog, Pet /* <- specify C++ parent type */>(m, "Dog")
    .def(py::init<const std::string &>())
    .def("bark", &Dog::bark);

// Method 2: pass parent class_ object:
py::class_<Dog>(m, "Dog", pet /* <- specify Python parent type */)
    .def(py::init<const std::string &>())
    .def("bark", &Dog::bark);
```

指明继承关系后，派生类实例将获得两者的字段和方法：

```
>>> p = example.Dog("Molly")
>>> p.name
u'Molly'
>>> p.bark()
u'woof!'
```

上面的例子是一个常规非多态的继承关系，表现在Python就是:

```
// ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
m.def("pet_store", []() { return std::unique_ptr<Pet>(new Dog("Molly")); });
```

```
>>> p = example.pet_store()
>>> type(p)  # `Dog` instance behind `Pet` pointer
Pet          # no pointer downcasting for regular non-polymorphic types
>>> p.bark()
AttributeError: 'Pet' object has no attribute 'bark'
```

`pet_store`函数返回了一个Dog实例，但由于基类并非多态类型，Python只识别到了Pet。在C++中，一个类至少有一个虚函数才会被视为多态类型。pybind11会自动识别这种多态机制。

```
struct PolymorphicPet {
    virtual ~PolymorphicPet() = default;
};

struct PolymorphicDog : PolymorphicPet {
    std::string bark() const { return "woof!"; }
};

// Same binding code
py::class_<PolymorphicPet>(m, "PolymorphicPet");
py::class_<PolymorphicDog, PolymorphicPet>(m, "PolymorphicDog")
    .def(py::init<>())
    .def("bark", &PolymorphicDog::bark);

// Again, return a base pointer to a derived instance
m.def("pet_store2", []() { return std::unique_ptr<PolymorphicPet>(new
PolymorphicDog); });
```

```
>>> p = example.pet_store2()
>>> type(p)
PolymorphicDog  # automatically downcast
>>> p.bark()
u'woof!'
```

pybind11[illegible]C++[illegible]

3.2 Python[illegible]C++[illegible]

[illegible]Python[illegible]

```
class Animal {
public:
    virtual ~Animal() { }
    virtual std::string go(int n_times) = 0;
};

class Dog : public Animal {
public:
    std::string go(int n_times) override {
        std::string result;
        for (int i=0; i<n_times; ++i)
            result += "woof! ";
        return result;
    }
};

std::string call_go(Animal *animal) {
    return animal->go(3);
}

PYBIND11_MODULE(example, m) {
    py::class_<Animal>(m, "Animal")
        .def("go", &Animal::go);

    py::class_<Dog, Animal>(m, "Dog")
        .def(py::init<>());

    m.def("call_go", &call_go);
}
```

这样绑定之后，用户在Python中继承实现Animal会报错，提示“No constructor defined!“。

这时，我们需要定义一个新的Animal类作为辅助跳板：

```
class PyAnimal : public Animal {
public:
    /* Inherit the constructors */
    using Animal::Animal;

    /* Trampoline (need one for each virtual function) */
    std::string go(int n_times) override {
        PYBIND11_OVERRIDE_PURE(
            std::string, /* Return type */
            Animal,      /* Parent class */
            go,          /* Name of function in C++ (must match Python name)
*/
            n_times      /* Argument(s) */
        );
    }
};
```

定义纯虚函数时需要使用`PYBIND11_OVERRIDE_PURE`宏，而有默认实现的虚函数则使用`PYBIND11_OVERRIDE`。`PYBIND11_OVERRIDE_PURE_NAME` 和`PYBIND11_OVERRIDE_NAME`宏的功能类似，主要用于C函数名和Python函数名不一致的时候。以`__str__`为例：

```
std::string toString() override {
  PYBIND11_OVERRIDE_NAME(
      std::string, // Return type (ret_type)
      Animal,      // Parent class (cname)
      "__str__",   // Name of method in Python (name)
      toString,    // Name of function in C++ (fn)
  );
}
```

Animal类的绑定代码也需要一些微调：

```
PYBIND11_MODULE(example, m) {
    py::class_<Animal, PyAnimal /* <--- trampoline*/>(m, "Animal")
        .def(py::init<>())
        .def("go", &Animal::go);

    py::class_<Dog, Animal>(m, "Dog")
        .def(py::init<>());

    m.def("call_go", &call_go);
}
```

pybind11通过向 `class_` 指定额外的模板参数PyAnimal，让我们可以在Python中继承Animal类。

接下来，我们可以像往常一样定义构造函数。绑定时我们需要使用真实类，而不是辅助类。

```
py::class_<Animal, PyAnimal /* <--- trampoline*/>(m, "Animal");
    .def(py::init<>())
    .def("go", &PyAnimal::go); /* <--- THIS IS WRONG, use &Animal::go */
```

下面的Python代码展示了我们继承并重载了 `Animal::go` 方法，并通过虚函数来调用它：

```
from example import *
d = Dog()
call_go(d)     # u'woof! woof! woof! '
class Cat(Animal):
    def go(self, n_times):
        return "meow! " * n_times

c = Cat()
call_go(c)   # u'meow! meow! meow! '
```

- 如果你在派生的Python类中自定义了一个构造函数，你必须保证显示调用C++构造函数(通过 `__init__`)，不管它是否为默认构造函数。否则，实例属于C++那部分的内存就未初始化，可能导致未定义行为。在pybind11 2.6版本中，这种错误将会抛出 `TypeError` 异常。

```
class Dachshund(Dog):
    def __init__(self, name):
        Dog.__init__(self)  # Without this, a TypeError is raised.
        self.name = name

    def bark(self):
        return "yap!"
```

注意必须显式地调用 `__init__`，而不应该使用 `supper()`。在一些简单的线性继承中， `supper()` 或许可以正常工作；一旦你混合Python和C++类使用多重继承，由于Python MRO和C++的机制，一切都将崩溃。

3.3 虚函数与继承

综合考虑虚函数与继承时，你需要为每个你允许在Python派生类中重载的方法提供重载方式。下面我们扩展Animal和Dog来举例：

```
class Animal {
public:
    virtual std::string go(int n_times) = 0;
    virtual std::string name() { return "unknown"; }
};
class Dog : public Animal {
public:
    std::string go(int n_times) override {
        std::string result;
        for (int i=0; i<n_times; ++i)
            result += bark() + " ";
        return result;
    }
    virtual std::string bark() { return "woof!"; }
};
```

上节涉及到的Animal辅助类仍是必须的，为了让Python代码能够继承 `Dog` 类，我们也需要为 `Dog` 类增加一个跳板类，来实现 `bark()` 和继承自Animal的 `go()`、 `name()` 等重载方法（即便Dog类并不直接重载name方法）。

```
class PyAnimal : public Animal {
public:
    using Animal::Animal; // Inherit constructors
    std::string go(int n_times) override {
PYBIND11_OVERRIDE_PURE(std::string, Animal, go, n_times); }
    std::string name() override { PYBIND11_OVERRIDE(std::string, Animal,
name, ); }
};
class PyDog : public Dog {
public:
    using Dog::Dog; // Inherit constructors
    std::string go(int n_times) override { PYBIND11_OVERRIDE(std::string,
Dog, go, n_times); }
    std::string name() override { PYBIND11_OVERRIDE(std::string, Dog, name,
); }
    std::string bark() override { PYBIND11_OVERRIDE(std::string, Dog, bark,
); }
};
```

> 注意到 `name()` 和 `bark()` 尾部的逗号，这用来说明辅助类的函数不带任何参数。当函数至少有一个参数时，应该省略尾部的逗号。

注册一个继承已经在pybind11中注册的带虚函数的类，同样需要为其添加辅助类，即便它没有定义或重载任何虚函数：

```
class Husky : public Dog {};
class PyHusky : public Husky {
public:
    using Husky::Husky; // Inherit constructors
    std::string go(int n_times) override {
PYBIND11_OVERRIDE_PURE(std::string, Husky, go, n_times); }
    std::string name() override { PYBIND11_OVERRIDE(std::string, Husky, name,
); }
    std::string bark() override { PYBIND11_OVERRIDE(std::string, Husky, bark,
); }
};
```

我们可以使用模板辅助类将简化这类重复的绑定工作，这对有多个虚函数的基类尤其有用：

```
template <class AnimalBase = Animal> class PyAnimal : public AnimalBase {
public:
    using AnimalBase::AnimalBase; // Inherit constructors
    std::string go(int n_times) override {
PYBIND11_OVERRIDE_PURE(std::string, AnimalBase, go, n_times); }
    std::string name() override { PYBIND11_OVERRIDE(std::string, AnimalBase,
name, ); }
};
template <class DogBase = Dog> class PyDog : public PyAnimal<DogBase> {
public:
    using PyAnimal<DogBase>::PyAnimal; // Inherit constructors
    // Override PyAnimal's pure virtual go() with a non-pure one:
    std::string go(int n_times) override { PYBIND11_OVERRIDE(std::string,
DogBase, go, n_times); }
    std::string bark() override { PYBIND11_OVERRIDE(std::string, DogBase,
bark, ); }
};
```

这样，我们只需要一个辅助方法来定义虚函数和纯虚函数的重载了。只是这样编译器就需要生成许多额外的方法和类。

下面我们在pybind11中注册这些类：

```
py::class_<Animal, PyAnimal<>> animal(m, "Animal");
py::class_<Dog, Animal, PyDog<>> dog(m, "Dog");
py::class_<Husky, Dog, PyDog<Husky>> husky(m, "Husky");
// ... add animal, dog, husky definitions
```

注意，Husky不需要一个专门的辅助类，因为它没定义任何新的虚函数和纯虚函数的重载。

Python中的使用示例：

```
class ShihTzu(Dog):
    def bark(self):
        return "yip!"
```

3.4 非公有析构函数

如果一个类拥有私有或保护的析构函数（例如单例类），通过pybind11绑定类时编译器将会报错。本质的问题是 `std::unique_ptr` 智能指针负责管理实例的生命周期需要引用析构函数，即便没有资源需要回收。Pybind11提供了辅助类 `py::nodelete` 来禁止对析构函数的调用。这种情况下，C++侧负责析构对象避免内存泄漏就十分重要。

```
/* ... definition ... */

class MyClass {
private:
    ~MyClass() { }
};

/* ... binding code ... */

py::class_<MyClass, std::unique_ptr<MyClass, py::nodelete>>(m, "MyClass")
    .def(py::init<>())
```

3.5 隐式转换

假设项目中有A和B两个类型，A可以直接转换为B。

```
py::class_<A>(m, "A")
    /// ... members ...

py::class_<B>(m, "B")
    .def(py::init<A>())
    /// ... members ...

m.def("func",
    [](const B &) { /* .... */ }
);
```

如果想func函数传入A类型的参数a，Pyhton侧需要这样写 `func(B(a))`，而C++则可以直接使用 `func(a)`，自动将A类型转换为B类型。

这种情形下（B有一个接受A类型参数的构造函数），我们可以使用如下声明来让Python侧也支持类似的隐式转换：

```
py::implicitly_convertible<A, B>();
```

3.6 重载操作符

假设有这样一个类 `Vector2`，它通过重载操作符实现了向量加法和标量乘法。

```
class Vector2 {
public:
    Vector2(float x, float y) : x(x), y(y) { }

    Vector2 operator+(const Vector2 &v) const { return Vector2(x + v.x, y +
v.y); }
    Vector2 operator*(float value) const { return Vector2(x * value, y *
value); }
    Vector2& operator+=(const Vector2 &v) { x += v.x; y += v.y; return *this;
}
    Vector2& operator*=(float v) { x *= v; y *= v; return *this; }

    friend Vector2 operator*(float f, const Vector2 &v) {
        return Vector2(f * v.x, f * v.y);
    }

    std::string toString() const {
        return "[" + std::to_string(x) + ", " + std::to_string(y) + "]";
    }
private:
    float x, y;
};
```

操作符绑定代码如下：

```
#include <pybind11/operators.h>

PYBIND11_MODULE(example, m) {
    py::class_<Vector2>(m, "Vector2")
        .def(py::init<float, float>())
        .def(py::self + py::self)
        .def(py::self += py::self)
        .def(py::self *= float())
        .def(float() * py::self)
        .def(py::self * float())
        .def(-py::self)
        .def("__repr__", &Vector2::toString);
}
```

`.def(py::self * float())` 是如下代码的简短标记：

```
.def("__mul__", [](const Vector2 &a, float b) {
    return a * b;
}, py::is_operator())
```

3.7 深拷贝支持

Python通常在赋值中使用引用。有时需要一个真正的拷贝，以防止修改所有的拷贝实例。 `copy` 模块提供了这样的拷贝能力。

在Python3中，带pickle支持的类自带深拷贝能力。但是，自定义 `__copy__` 和 `__deepcopy__` 方法能够提高拷贝的性能。在Python2.7中，由于pybind11只支持cPickle，要想实现深拷贝，自定义这两个方法必须实现。

对于一些简单的类，可以使用拷贝构造函数来实现深拷贝。如下所示：

```
py::class_<Copyable>(m, "Copyable")
    .def("__copy__",  [](const Copyable &self) {
        return Copyable(self);
    })
    .def("__deepcopy__", [](const Copyable &self, py::dict) {
        return Copyable(self);
    }, "memo"_a);
```

> Note: 本例中不会复制动态属性。

3.8 多重继承

pybind11支持绑定多重继承的类，只需在将所有基类作为 `class_` 的模板参数即可：

```
py::class_<MyType, BaseType1, BaseType2, BaseType3>(m, "MyType")
   ...
```

基类间的顺序任意，甚至可以穿插使用别名或者holder类型，pybind11能够自动识别它们。唯一的要求就是第一个模板参数必须是类型本身。

允许Python中定义的类继承多个C++类，也允许混合继承C++类和Python类。

有一个关于该特性实现的警告：当仅指定一个基类，实际上有多个基类时，pybind11会认为它并没有使用多重继承，这将导致未定义行为。对于这个问题，我们可以在类构造函数中添加 `multiple_inheritance` 的标识。

```
py::class_<MyType, BaseType2>(m, "MyType", py::multiple_inheritance());
```

当模板参数列出了多个基类时，无需使用该标识。

3.9 绑定protected成员函数

通常不可能向Python公开protected 成员函数：

```
class A {
protected:
    int foo() const { return 42; }
};

py::class_<A>(m, "A")
    .def("foo", &A::foo); // error: 'foo' is a protected member of 'A'
```

[illegible]Python[illegible]protected [illegible]

```
class A {
protected:
    int foo() const { return 42; }
};

class Publicist : public A { // helper type for exposing protected functions
public:
    using A::foo; // inherited with different access modifier
};

py::class_<A>(m, "A") // bind the primary class
    .def("foo", &Publicist::foo); // expose protected methods via the
publicist
```

[illegible] `&Publicist::foo` [illegible] `&A::foo` [illegible] `Publicist` [illegible] `public` [illegible]

[illegible]Python[illegible] `protected` [illegible]publicist pattern[illegible]trampoline[illegible]

```
class A {
public:
    virtual ~A() = default;

protected:
    virtual int foo() const { return 42; }
};

class Trampoline : public A {
public:
    int foo() const override { PYBIND11_OVERRIDE(int, A, foo, ); }
};

class Publicist : public A {
public:
    using A::foo;
};

py::class_<A, Trampoline>(m, "A") // <-- `Trampoline` here
    .def("foo", &Publicist::foo); // <-- `Publicist` here, not `Trampoline`!
```

3.10 绑定final类

在C++11中，我们可以使用 `findal` 关键字来确保一个类不被继承。 `py::is_final` 属性则可以用来确保一个类在Python中不被继承。底层的C++类型不需要定义为final。

```
class IsFinal final {};

py::class_<IsFinal>(m, "IsFinal", py::is_final());
```

在Python中试图继承这个类，将导致错误：

```
class PyFinalChild(IsFinal):
    pass

TypeError: type 'IsFinal' is not an acceptable base type
```

4. 异常处理

4.1 C++内置异常到Python异常的转换

当Python通过pybind11调用C++代码时，pybind11将捕获C++异常，并将其翻译为对应的Python异常后抛出。这样Python代码就能够处理它们。

pybind11定义了`std::exception`及其标准子类，和一些特殊异常到Python异常的翻译。由于它们不是真正的Python异常，所以不能使用Python C API来检查。相反，它们是纯C++异常，当它们到达异常处理器时，pybind11将其翻译为对应的Python异常。

Exception thrown by C++	Translated to Python exception type
`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::key_error`	`KeyError` (used to indicate out of bounds access in `__getitem__`, `__setitem__` in dict-like objects, etc.)
`pybind11::value_error`	`ValueError` (used to indicate wrong value passed in `container.remove(...)`)
`pybind11::type_error`	`TypeError`

Exception thrown by C++	Translated to Python exception type
`pybind11::buffer_error`	`BufferError`
`pybind11::import_error`	`ImportError`
`pybind11::attribute_error`	`AttributeError`
Any other exception	`RuntimeError`

异常翻译不是双向的。即上述异常不会捕获源自Python的异常。Python的异常，需要捕获 `pybind11::error_already_set` 。

这里有个特殊的异常，当入参不能转化为Python对象时， `handle::call()` 将抛出 `cast_error` 异常。

4.2 注册定制异常翻译

如果上述默认异常转换策略不够用，pybind11也提供了注册自定义异常翻译的支持。类似于pybind11 class，异常翻译也可以定义在模块内或global。要注册一个使用C++异常的 `what()` 方法将C++到Python的异常转换，可以使用下面的方法：

```
py::register_exception<CppExp>(module, "PyExp");
```

这个调用在指定模块创建了一个名称为PyExp的Python异常，并自动将CppExp相关的异常转换为PyExp异常。

相似的函数可以注册模块内的异常翻译：

```
py::register_local_exception<CppExp>(module, "PyExp");
```

方法的第三个参数handle可以指定异常的基类：

```
py::register_exception<CppExp>(module, "PyExp", PyExc_RuntimeError);
py::register_local_exception<CppExp>(module, "PyExp", PyExc_RuntimeError);
```

这样，PyExp异常可以捕获PyExp和RuntimeError。

Python内置的异常类型可以参考Python文档Standard Exceptions，默认的基类为 `PyExc_Exception` 。

`py::register_exception_translator(translator)` 和 `py::register_local_exception_translator(translator)` 提供了更高级的异常翻译功能，它可以注册任意的异常类型。函数接受一个无状态的回调函数 `void(std::exception_ptr)` 。

9.3 在C++中处理Python异常

当C++调用Python函数时（回调函数或者操作Python对象），若Python有异常抛出，pybind11会将Python异常转化为 `pybind11::error_already_set` 类型的异常，它包含了一个C++字符串描述和实际的Python异常。`error_already_set` 用于将Python异常传回Python（或者在C++侧处理）。

Exception raised in Python	Thrown as C++ exception type
Any Python `Exception`	`pybind11::error_already_set`

举个例子：

```
try {
    // open("missing.txt", "r")
    auto file = py::module_::import("io").attr("open")("missing.txt", "r");
    auto text = file.attr("read")();
    file.attr("close")();
} catch (py::error_already_set &e) {
    if (e.matches(PyExc_FileNotFoundError)) {
        py::print("missing.txt not found");
    } else if (e.matches(PyExc_PermissionError)) {
        py::print("missing.txt found but not accessible");
    } else {
        throw;
    }
}
```

该方法并不适用与C++到Python的翻译，Python侧抛出的异常总是被翻译为 `error_already_set`.

```
try {
    py::eval("raise ValueError('The Ring')");
} catch (py::value_error &boromir) {
    // Boromir never gets the ring
    assert(false);
} catch (py::error_already_set &frodo) {
    // Frodo gets the ring
    py::print("I will take the ring");
}

try {
    // py::value_error is a request for pybind11 to raise a Python exception
    throw py::value_error("The ball");
} catch (py::error_already_set &cat) {
    // cat won't catch the ball since
    // py::value_error is not a Python exception
    assert(false);
} catch (py::value_error &dog) {
    // dog will catch the ball
    py::print("Run Spot run");
    throw;  // Throw it again (pybind11 will raise ValueError)
}
```

9.4 处理Python C API的错误

尽可能地使用pybind11 wrappers代替直接调用Python C API。如果确实需要直接使用Python C API，除了需要手动管理引用计数外，还必须遵守pybind11的错误处理协议。

在调用Python C API后，如果Python返回错误，需要调用 `throw py::error_already_set();` 语句，让pybind11来处理异常并传递给Python解释器。这包括对错误设置函数的调用，如 `PyErr_SetString` 。

```
PyErr_SetString(PyExc_TypeError, "C API type error demo");
throw py::error_already_set();

// But it would be easier to simply...
throw py::type_error("pybind11 wrapper type error");
```

也可以调用 `PyErr_Clear` 来忽略错误。

任何Python错误必须被抛出或清除，否则Python/pybind11将处于无效的状态。

9.5 处理unraiseable异常

如果Python调用的C++析构函数或任何标记为`noexcept(true)`的函数抛出了异常，该异常不会传播出去。如果它们在调用图中抛出或捕捉不到任何异常，c++运行时将调用std::terminate()立即终止程序。在C++析构函数中调用Python尤其需要注意异常的捕获，必须捕获所有`error_already_set`类型的异常，并使用`error_already_set::discard_as_unraisable()`来抛弃Python异常。

类似的，在类`__del__`方法引发的Python异常也不会传播，但被Python作为unraisable错误记录下来。在Python 3.8+中，将触发system hook，并记录auditing event日志。

任何noexcept函数应该使用try-catch代码块来捕获`error_already_set`（或其他可能出现的异常）。pybind11包装的Python异常并非真正的Python异常，它是pybind11捕获并转化的C++异常。noexcept函数不能传播这些异常。我们可以将它们转换为Python异常，然后丢弃`discard_as_unraisable`，如下所示。

```
void nonthrowing_func() noexcept(true) {
    try {
        // ...
    } catch (py::error_already_set &eas) {
        // Discard the Python error using Python APIs, using the C++ magic
        // variable __func__. Python already knows the type and value and of
the
        // exception object.
        eas.discard_as_unraisable(__func__);
    } catch (const std::exception &e) {
        // Log and discard C++ exceptions.
        third_party::log(e);
    }
}
```

5. 类型转换

6. python C++接口

7. 杂项

7.1 关于便利宏的说明

pybind11提供了一些便利宏如 `PYBIND11_DECLARE_HOLDER_TYPE()` 和 `PYBIND11_OVERRIDE_*` 。由于这些宏只是在预处理中计算(预处理程序没有类型的概念)，它们会被模板参数中的逗号搞混。如：

```
PYBIND11_OVERRIDE(MyReturnType<T1, T2>, Class<T3, T4>, func)
```

预处理器会将其解释为5个参数（逗号分隔），而不是3个。有两种方法可以处理这个问题：使用类型别名，或者使用 `PYBIND11_TYPE` 包裹类型。

```
// Version 1: using a type alias
using ReturnType = MyReturnType<T1, T2>;
using ClassType = Class<T3, T4>;
PYBIND11_OVERRIDE(ReturnType, ClassType, func);

// Version 2: using the PYBIND11_TYPE macro:
PYBIND11_OVERRIDE(PYBIND11_TYPE(MyReturnType<T1, T2>),
                  PYBIND11_TYPE(Class<T3, T4>), func)
```

`PYBIND11_MAKE_OPAQUE` 宏不需要上述解决方案。

7.2 全局解释器锁（GIL）

在Python中调用C++函数时，默认会持有GIL。`gil_scoped_release` 和 `gil_scoped_acquire` 可以方便地在函数体中释放和获取GIL。这样长时间运行的C++代码可以通过Python线程实现并行化。示例如下：

```
class PyAnimal : public Animal {
public:
    /* Inherit the constructors */
    using Animal::Animal;

    /* Trampoline (need one for each virtual function) */
    std::string go(int n_times) {
        /* Acquire GIL before calling Python code */
        py::gil_scoped_acquire acquire;

        PYBIND11_OVERRIDE_PURE(
            std::string, /* Return type */
            Animal,      /* Parent class */
            go,          /* Name of function */
            n_times      /* Argument(s) */
        );
    }
};

PYBIND11_MODULE(example, m) {
    py::class_<Animal, PyAnimal> animal(m, "Animal");
    animal
        .def(py::init<>())
        .def("go", &Animal::go);

    py::class_<Dog>(m, "Dog", animal)
        .def(py::init<>());

    m.def("call_go", [](Animal *animal) -> std::string {
        /* Release GIL before calling into (potentially long-running) C++
code */
        py::gil_scoped_release release;
        return call_go(animal);
    });
}
```

我们可以使用 `call_guard` 策略来简化 `call_go` 的封装：

```
m.def("call_go", &call_go, py::call_guard<py::gil_scoped_release>());
```

[illegible]Mermaid[illegible]

[illegible] `Mermaid` [illegible],[illegible],[illegible],[illegible],[illegible],[illegible] `markdown` [illegible].

[illegible]:

- [illegible]`Mermaid`[illegible];
- [illegible]`Mermaid`[illegible];
- [illegible] `Gitbook` [illegible].

[illegible]Mermaid[illegible]

[illegible]

```
- [illegible]
- [illegible]
- [illegible]
```

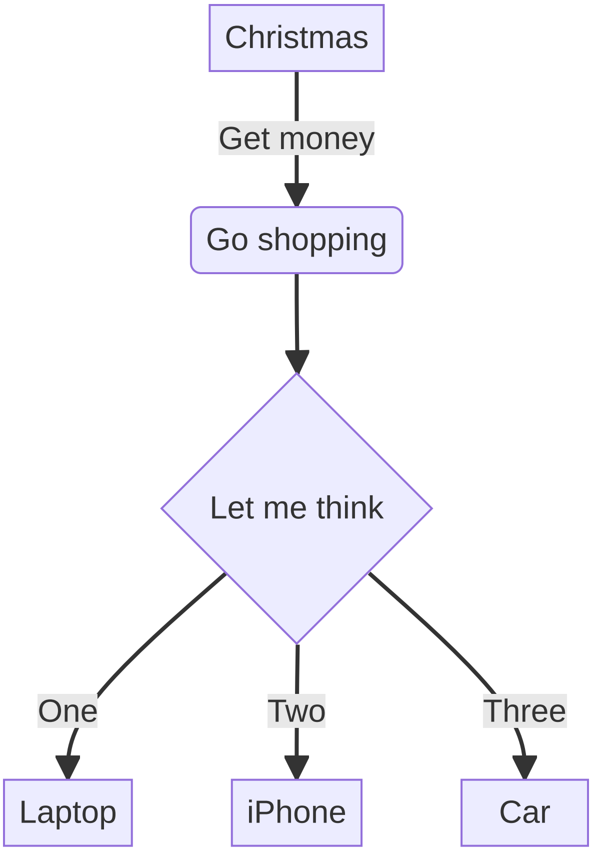
[illegible],[illegible].

`Mermaid`[illegible] `Javascript` [illegible].[illegible] `markdown` [illegible],[illegible].

[illegible]

```
graph TD
  A[Christmas] -->|Get money| B(Go shopping)
  B --> C{Let me think}
  C -->|One| D[Laptop]
  C -->|Two| E[iPhone]
  C -->|Three| F[fa:fa-car Car]
```

[illegible]


Christmas
Get money
Go shopping
Let me think
One
Two
Three
Laptop
iPhone
Car

- 项目地址：https://github.com/mermaid-js/mermaid
- 在线编辑：https://mermaidjs.github.io/mermaid-live-editor/
- 官方文档：https://mermaid-js.github.io/mermaid/#/flowchart

Mermaid流程图快速入门

布局方向

关键词

```
+ TB
+ BT
+ LR
+ RL
```

Syntax error in graph
mermaid version 9.2.2

[illegible],[illegible],[illegible]: `top`([illegible]), `bottom`([illegible]), `left`([illegible])[illegible] `right`([illegible]).[illegible]: `TB` ([illegible]), `BT` ([illegible]), `LR` ([illegible])[illegible] `RL` ([illegible])[illegible].

> [illegible]: [illegible],[illegible].

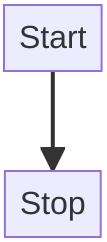
- TB

> [illegible]: from **T**op to **B**ottom

[illegible]

```
graph TB
    Start --> Stop
```

[illegible]



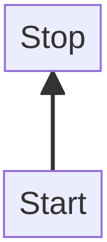
- BT

> 从下到上: from **B**ottom to **T**op

源码

```
graph BT
    Start --> Stop
```

效果



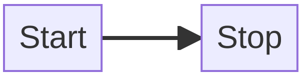
- LR

> 从左往右: from **L**eft to **R**ight

源码

```
graph LR
    Start --> Stop
```

效果



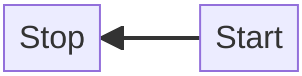
- RL

> 从右往左: from **R**ight to **L**eft

源码

```
graph RL
    Start --> Stop
```

效果



形状

关键词

```
- ⊠⊠⊠⊠
    + [⊠⊠]
        - [[⊠⊠⊠⊠]]
        - [(⊠⊠)]
        - [{⊠⊠⊠⊠}]
        - [/⊠⊠⊠⊠⊠/]
        - [\⊠⊠⊠⊠⊠\]
        - [/⊠⊠\]
        - [\⊠⊠/]
    + (⊠⊠⊠⊠)
        - ((⊠⊠))
        - ([⊠⊠⊠])
        - ({⊠⊠⊠⊠})
    + {⊠⊠}
        - {{⊠⊠⊠}}
        - {[⊠⊠⊠⊠]}
        - {(⊠⊠⊠⊠)}
    + >⊠⊠⊠⊠⊠]
```

Syntax error in graph

mermaid version 9.2.2

流程图节点形状，默认支持矩形和圆两种基本形状，包括基本形状的简单变体，支持嵌套组合形式，其中 `[]` 表示矩形，`()` 表示圆弧，`{}` 表示尖角（窃以为 `<>` 更适合）等等。

> 核心：最外层代表主形状，内层辅助修饰。

一次性节点

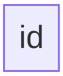
一次性节点，默认表现为矩形节点，其文本内容直接显示 `id` 的值，适合后续不会出现多次引用的情况。

> `id` 建议直接写成有意义的文本描述而不是当成唯一标识。

源码

```
graph TD
    id
```

效果



可重复节点

可重复节点，指定节点形状，其文本内容不再是 `id` 的值而是 `<node shape>` 的值，适合后续出现多次引用相同节点的情况。

> `id` 代表节点的唯一标识,当前节点的文本描述由 `<node shape>` 的值指定,建议 `id` 写成有意义的唯一标识.

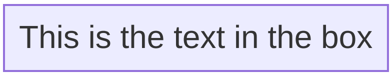
- 矩形

> 一般格式: `[node description]` , `[]` 中括号表示节点是矩形形状, `node description` 是节点的描述文本.

源码

```
graph LR
    id1[This is the text in the box]
```

效果


This is the text in the box

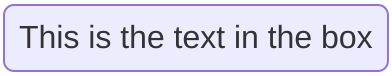
- 圆角矩形

> 一般格式: `(node description)` , `()` 小括号表示节点是圆角矩形形状, `node description` 是节点的描述文本.

源码

```
graph LR
    id1(This is the text in the box)
```

效果


This is the text in the box

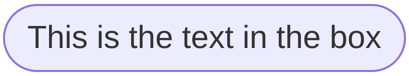
- 体育场

> 一般格式：`([node description])` , `()` 小括号嵌套 `[]` 中括号表示节点是大弧度的圆角矩形形状,也就是体育场形状, `node description` 是节点的描述文本.

源码

```
graph LR
    id1([This is the text in the box])
```

效果


This is the text in the box

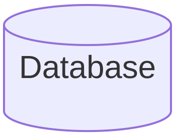
- 圆柱

> 一般格式：`[(node description)]` , `[]` 中括号嵌套 `()` 小括号表示节点是圆柱形状, `node description` 是节点的描述文本.

源码

```
graph LR
    id1[(Database)]
```

效果



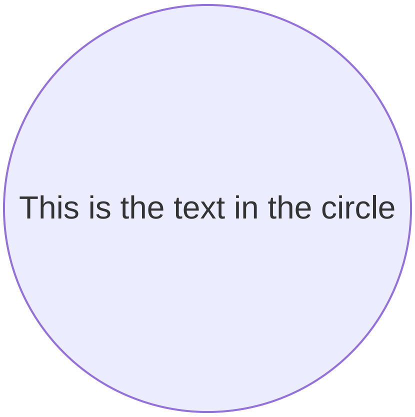
- 圆形

> 一般格式：`((node description))`，`()` 小括号嵌套 `()` 小括号表示节点是圆形形状，`node description` 是节点的描述文本。

源码

```
graph LR
    id1((This is the text in the circle))
```

效果



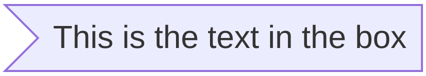
- 不对称矩形

> 一般格式：`>node description]`，左边是右尖括号 `>`，右边是右中括号 `]` 表示不对称矩形形状，`node description` 是节点的描述文本。

源码

```
graph LR
    id1>This is the text in the box]
```

效果


This is the text in the box

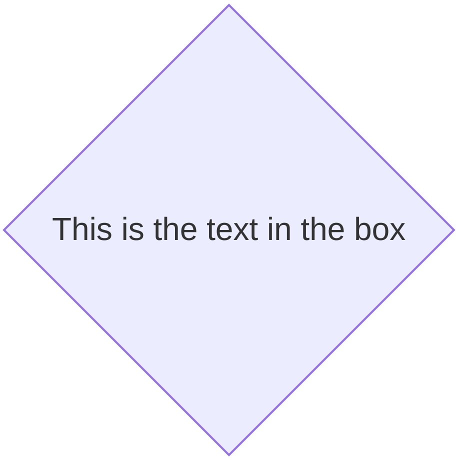
- [illegible]

> [illegible]: `{node description}` , `{}` [illegible], `node description` [illegible].

[illegible]

```
graph LR
    id1{This is the text in the box}
```

[illegible]


This is the text in the box

- [illegible]

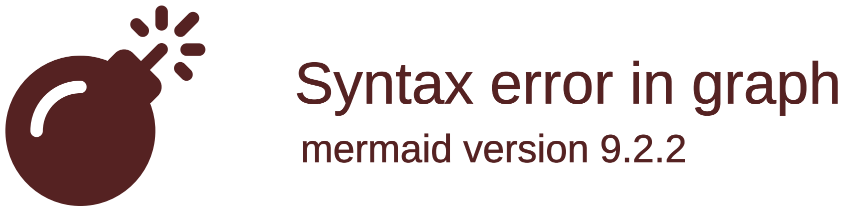
> [illegible]: `{{node description}}` , `{}` [illegible] `{}` [illegible], `node description` [illegible].

[illegible]

```
graph LR
    id1\{\{This is the text in the box\}\}
```

> `Gitbook` 语法中双大括号 `{}` 表示特殊意义,上述源码只能转义处理,实际上并不需要 `\` 进行转义.

效果


Syntax error in graph
mermaid version 9.2.2

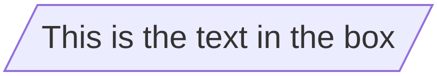
- 平行四边形

> 一般格式: `[/node description/]` , `[]` 中括号嵌套 `//` 左斜杠表示左斜平行四边形形状, `node description` 是节点的描述文本.

源码

```
graph TD
    id1[/This is the text in the box/]
```

效果


This is the text in the box

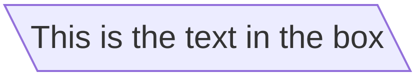
- 平行四边形

> 一般格式: `[\node description\]` , `[]` 中括号嵌套 `\\` 右斜杠表示右斜平行四边形形状, `node description` 是节点的描述文本.

源码

```
graph TD
    id1[\This is the text in the box\]
```

效果


This is the text in the box

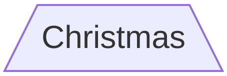
- 梯形

> 一般格式：`[/node description\]`，`[]` 中括号嵌套 `/\` 左右斜杠表示上短下长梯形形状，`node description` 是节点的描述文本。

源码

```
graph TD
    A[/Christmas\]
```

效果



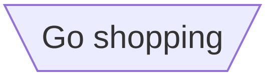
- 另一种梯形

> 一般格式：`[\node description/]`，`[]` 中括号嵌套 `\/` 右左斜杠表示上长下短梯形形状，`node description` 是节点的描述文本。

源码

```
graph TD
    B[\Go shopping/]
```

效果


Go shopping

连接线

关键词

```
+ [illegible]/[illegible]
    - --
    - -.
+ [illegible]/[illegible]
    - >
    - -
+ [illegible]/[illegible]
    - [illegible]
        + --[illegible]
        + |[illegible]|
    - [illegible]
        + -.[illegible]
        + |[illegible]|
+ [illegible]
    - ==
+ [illegible]
    - -->
    - ---
    - -.->
    - -.-
    - [illegible]
        + --[illegible]-->
        + -->|[illegible]|
    - [illegible]
        + --[illegible]---
        + ---|[illegible]|
    - [illegible]
        + -.[illegible]-.->
        + -.->|[illegible]|
    - [illegible]
        + -.[illegible]-.-
        + -.-|[illegible]|
    - ==>
    - ===
    - [illegible](2)
        + ==[illegible]==>
        + ==>|[illegible]|
    - [illegible](2)
        + ==[illegible]===
        + ===|[illegible]|
```

Syntax error in graph

mermaid version 9.2.2

流程图连接线样式,支持实线和虚线以及有箭头样式和无箭头样式,除此之外还支持添加连接线描述文字,其中 `--` 代表实线,实线中间多一点 `-.-` 代表虚线,添加箭头用右尖括号 `>`,没有箭头继续用短横线 `-`.

> 核心: 先实线再虚线,先有箭头再去箭头,左边位置添加描述文字需要区分实现还是虚线,右边位置添加描述文字格式一致.

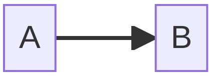
- 有箭头无描述实线

> 一般格式: `-->`,其中 `--` 表示实线, `>` 表示有箭头.

源码

```
graph LR
    A-->B
```

效果



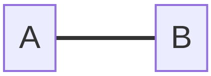
- 无箭头实线

> 一般格式: `---`,其中 `--` 表示实线, `-` 表示无箭头.

源码

```
graph LR
    A --- B
```

[illegible]



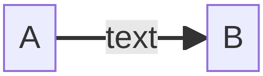
- [illegible]

> [illegible]: `--connection line description-->` ,[illegible] `--` [illegible],[illegible] `-->` [illegible].

[illegible]

```
graph LR
    A-- text -->B
```

[illegible]

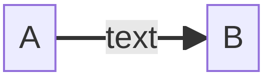


> [illegible]: `|connection line description|` ,[illegible] `||` [illegible].

[illegible]

```
graph LR
    A-->|text|B
```

[illegible]



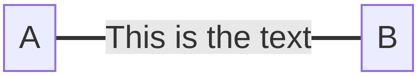
- 带描述的无箭头实线

> 一般格式：`--connection line description`，其中左边的 `--` 添加到实线左边位置，右边的 `---` 表示不带箭头的实线。

源码

```
graph LR
    A-- This is the text ---B
```

效果

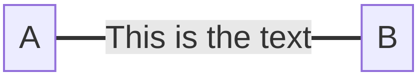

A
This is the text
B

> 一般格式：`|connection line description|`，其中 `||` 添加到连接线右边位置。

源码

```
graph LR
    A---|This is the text|B
```

效果


A
This is the text
B

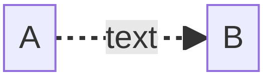
- 有箭头虚线

> 一般格式：`-.connection line description.->`，其中左边的 `-.` 添加到虚线左边位置，右边的 `.->` 表示带箭头的虚线。

源码

```
graph LR
   A-. text .-> B
```

效果



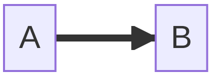
- 有箭头加粗实线

> 一般格式：`==>`，表示加粗实线。

源码

```
graph LR
   A ==> B
```

效果



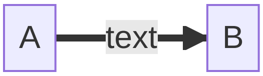
- 带描述的有箭头加粗实线

> 一般格式：`==connection line description`，左边的 `==` 添加到加粗实现左边，右边的 `==>` 代表加粗实线。

源码

```
graph LR
   A == text ==> B
```

效果



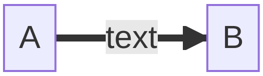
- 带描述的有箭头加粗实线

> 一般格式：`|connection line description|`，其中 `||` 添加到连接线右边位置.

源码

```
graph LR
   A ==>|text| B
```

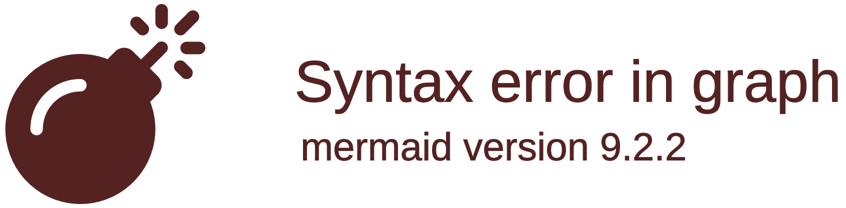
效果



高级用法

关键词

```
+ -->-->
+ &
+ ""
+ %%
+ subgraph
```



Syntax error in graph

mermaid version 9.2.2

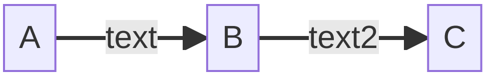
- 多节点链式连接

源码

支持链式连接方式，`A-->B-->C` 等价于 `A-->B` 和 `B-->C` 形式。

```
graph LR
   A -- text --> B -- text2 --> C
```

效果



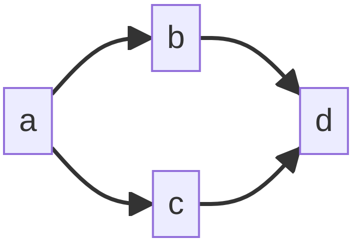
- 多节点共同连接

支持共同连接方式，`A-->B & C` 等价于 `A-->B` 和 `A-->C` 形式。

源码

```
graph LR
   a --> b & c--> d
```

效果



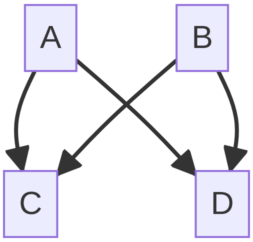
- 多节点相互连接

多节点共同连接的变体形式，`A & B --> C & D` 等价于 `A-->C`，`A-->D`，`B-->C` 和 `B-->D` 四种组合形式。

源码

```
graph TB
    A & B--> C & D
```

效果



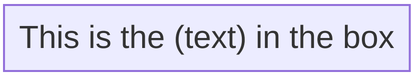
- 双引号包裹特殊字符

连接线描述文字存在特殊字符使用双引号 `""` 包裹处理，如遇到 `[]` 和 `()` 以及 `{}` 等特殊字符情况。

源码

```
graph LR
    id1["This is the (text) in the box"]
```

效果


This is the (text) in the box

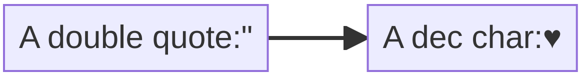
- 双引号包裹转义字符

支持 `Html` 转移字符

源码

```
graph LR
    A["A double quote:#quot;"] -->B["A dec char:#9829;"]
```

效果



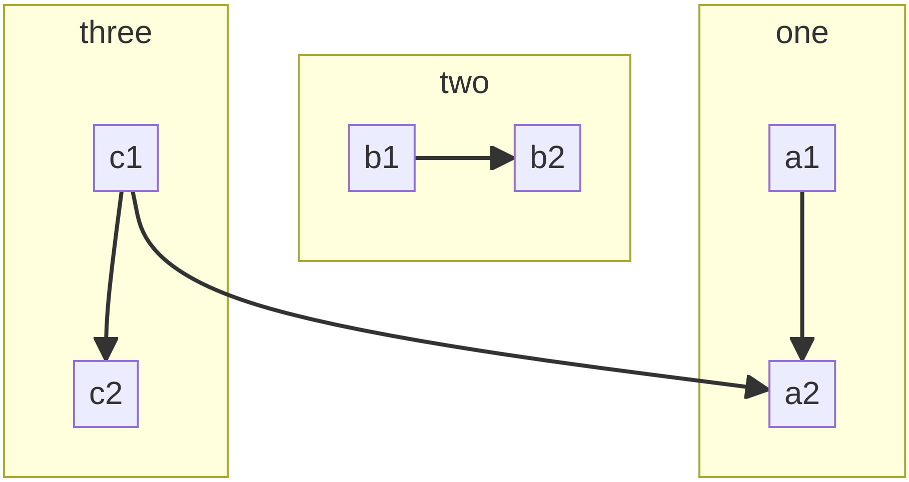
- 嵌套子流程图

定义

```
subgraph title
    graph definition
end
```

示例

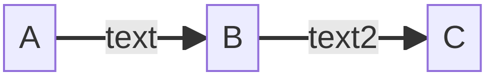
```
graph TB
    c1-->a2
    subgraph one
    a1-->a2
    end
    subgraph two
    b1-->b2
    end
    subgraph three
    c1-->c2
    end
```


three
two
one
c1
b1
b2
a1
c2
a2

- 注释语法

注释以 `%%` 开头并且独占一行.

```
graph LR
%% this is a comment A -- text --> B{node}
   A -- text --> B -- text2 --> C
```


A
text
B
text2
C

快速入门流程图回顾总结

关键词

```
- ⊠⊠⊠⊠⊠⊠
- ⊠⊠⊠⊠⊠
- ⊠⊠⊠⊠
```

mermaid-flow-chart-summary-simplemindmap.png

`Mermaid` 是一款开源的制图工具,可使用 `Markdown` 语法绘制流程图,支持更改流程图节点形状,添加描述文字以及更改连接线样式等.

英文单词缩写

> 四种布局方向的值是英文单词首字母大写缩写形式,默认仅支持垂直方向.

中文	英文	示例
图解	graph	`graph` 流程图类型标识
子图	subgraph	`subgraph` 嵌套子流程图标识
上	top	`TB` 或 `BT`,从上到下或从下到上的布局方向
下	bottom	`BT` 或 `TB`,从下到上或从上到下的布局方向
左	left	`LR` 或 `RL`,从左往右或从右往左的布局方向
右	right	`RL` 或 `LR`,从右往左或从左往右的布局方向

几何化形状

> 键盘符号形象化几何形状,组合形式表示形状的叠加,其中最外层符号是主形状,嵌套符号是辅助形状.

- 基本单元

表示法	含义	类型	备注
`[]`	矩形	节点形状	支持
`()`	圆角矩形	节点形状	支持
`{}`	菱形	节点形状	支持
`<>`	菱形	节点形状	不支持
`--`	实线	连接线样式	支持

表示法	含义	类型	备注
`-.`	虚线	连接线样式	支持
`==`	加粗实线	连接线样式	支持
`=:`	加粗虚线	连接线样式	不支持
`>`	有箭头	连接线样式	支持
`-`	无箭头	连接线样式	支持
`⊠⊠⊠`	右边连接线描述文字	连接线描述文字	支持
`--`	左边实线连接线描述文字	连接线描述文字	支持
`-.`	左边虚线连接线描述文字	连接线描述文字	支持
`==`	左边加粗实线连接线描述文字	连接线描述文字	支持
`=:`	左边加粗虚线连接线描述文字	连接线描述文字	不支持

- 组合单元

表示法	含义	类型	备注
`[[]]`	正方形	节点形状	不支持
`[()]`	圆柱体	节点形状	支持
`[{}]`	棱柱体	节点形状	不支持
`(())`	圆形	节点形状	支持
`([])`	体育场	节点形状	支持
`({})`	圆弧	节点形状	不支持
`⊠⊠⊠⊠`	六边形	节点形状	支持
`{[]}`	正多边形	节点形状	不支持
`{()}`	圆弧	节点形状	不支持
`-->`	实线带箭头	连接线样式	支持

表示法	含义	类型	备注
`---`	实线无箭头	连接线样式	支持
`-.>`	虚线带箭头	连接线样式	不支持
`-.->`	虚线带箭头	连接线样式	支持
`.->`	虚线带箭头	连接线样式	支持
`-.-`	虚线无箭头	连接线样式	支持
`.-`	虚线无箭头	连接线样式	支持
`==>`	加粗实线带箭头	连接线样式	支持
`===`	加粗实线无箭头	连接线样式	支持
`=:>`	加粗虚线带箭头	连接线样式	不支持
`=:=>`	加粗虚线带箭头	连接线样式	不支持
`=:=`	加粗虚线无箭头	连接线样式	不支持
`:=`	加粗虚线无箭头	连接线样式	不支持
`☒☒☒`	右边连接线描述文字	连接线描述文字	支持
`--connection line description-->`	左边实线带箭头连接线描述文字	连接线描述文字	支持
`-.connection line description-.->`	左边虚线带箭头连接线描述文字	连接线描述文字	支持
`--connection line description---`	左边实线无箭头连接线描述文字	连接线描述文字	支持
`-.connection line description-.-`	左边虚线无箭头连接线描述文字	连接线描述文字	支持
`==connection line description==>`	左边加粗实线带箭头连接线描述文字	连接线描述文字	支持

表示法	含义	类型	备注
`=:connection line description=:=>`	左边加粗虚线带箭头连接线描述文字	连接线描述文字	不支持
`==connection line description===`	左边加粗实线无箭头连接线描述文字	连接线描述文字	支持
`=:connection line description=:=`	左边加粗虚线无箭头连接线描述文字	连接线描述文字	不支持

有限语法

不论是节点形状还是连接线样式,语法支持是有限的,并不是随意组合的叠加状态,也可能随着后续更新会支持更多,一切以官方文档为主.

除了提供最基础的操作节点的能力之外,还可以根据 `JS` 和 `CSS` 相关知识高度自定义流程图行为表现,具体可参考官方文档.

> 官方文档: https://mermaid-js.github.io/mermaid/#/flowchart?id=styling-and-classes

- 交互能力 Interaction : https://mermaid-js.github.io/mermaid/#/flowchart?id=interaction
- 外观样式 Styling and classes : https://mermaid-js.github.io/mermaid/#/flowchart?id=interaction
- 字体支持 Basic support for fontawesome: https://mermaid-js.github.io/mermaid/#/flowchart?id=basic-support-for-fontawesome
- 空格分隔 https://mermaid-js.github.io/mermaid/#/flowchart?id=graph-declarations-with-spaces-between-vertices-and-link-and-without-semicolon

Reference

The following admonishments are implemented by the mdbook-admonish plugin and are automatically themed to match Catppuccin.

Directives

All supported directives are listed below.

`note`

Note

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

`abstract`, `summary`, `tldr`

Abstract

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

`info`, `todo`

Info

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

`tip`, `hint`, `important`

Tip

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

`success`, `check`, `done`

✓ Success

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

`question`, `help`, `faq`

Question

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

`warning`, `caution`, `attention`

⚠ Warning

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

`failure`, `fail`, `missing`

✕ Failure

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

`danger`, `error`

⚡ Danger

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

`bug`

Bug

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

`example`

Example

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

`quote`, `cite`

Quote

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency.

Bienvenue sur notre site de développement 3D !

Bienvenue sur notre site dédié au développement 3D. Ici, vous trouverez des ressources, des tutoriels et des informations utiles pour vous lancer dans le monde passionnant de la 3D.

Info

A beautifully styled message.

Un example

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla et euismod nulla. Curabitur feugiat, tortor non consequat finibus, justo purus auctor massa, nec semper lorem quam in massa.

Une note

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla et euismod nulla. Curabitur feugiat, tortor non consequat finibus, justo purus auctor massa, nec semper lorem quam in massa.

Un warning

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla et euismod nulla. Curabitur feugiat, tortor non consequat finibus, justo purus auctor massa, nec semper lorem quam in massa.

Collapsing note

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla et euismod nulla. Curabitur feugiat, tortor non consequat finibus, justo purus auctor massa, nec semper lorem quam in massa.

Le javascript c’est yolo préférez Typescript

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla et euismod nulla. Curabitur feugiat, tortor non consequat finibus, justo purus auctor massa, nec semper lorem quam in massa.

Referencing* and *dereferencing

The opposite of *referencing* by using `&` is *dereferencing*, which is accomplished with the dereference operator, `*`.

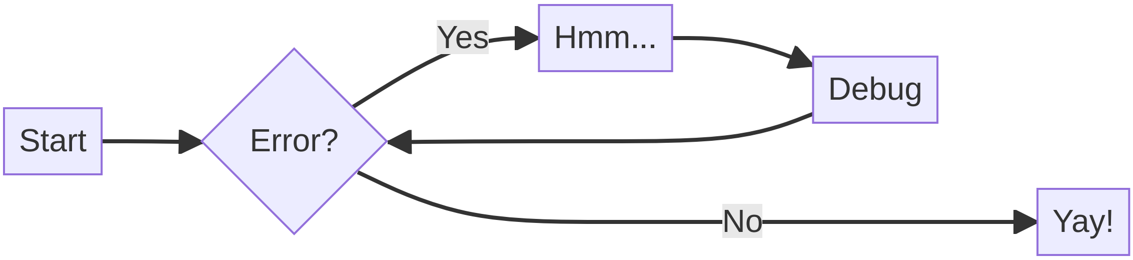
Bug

This syntax won’t work in Python 3:

```
print "Hello, world!"
```

À propos de nous

Nous sommes une équipe passionnée par la 3D et nous avons pour mission de partager nos connaissances avec la communauté. Vous trouverez ici des articles, des exemples de code et des démonstrations pour vous aider à démarrer votre voyage dans le développement 3D.



Pour commencer

Si vous êtes nouveau dans le domaine de la 3D, ne vous inquiétez pas ! Notre page “Getting Started” vous guidera à travers les étapes essentielles pour démarrer rapidement.

Restons en contact

N’hésitez pas à nous suivre sur les réseaux sociaux pour rester à jour avec nos dernières publications et annonces. Si vous avez des questions ou des commentaires, n’hésitez pas à nous contacter !

Mizux

kroki

```
graph TD
  A[ Anyone ] -->|Can help | B( Go to github.com/yuzutech/kroki )
  B --> C{ How to contribute? }
  C --> D[ Reporting bugs ]
  C --> E[ Sharing ideas ]
  C --> F[ Advocating ]
```

```
    0       3
     *-------*      +y
  1 /|    2 /|       ^
   *-------* |       |
   | |4    | |7      | ↰
   | *-----|-*     ⤹ +-----> +x
   |/      |/       / ⤴
   *-------*       v
  5       6      +z
```