24讲Boost：你需要的“瑞士军刀”

你好，我是吴咏炜。

我们已经零零碎碎提到过几次 Boost 了。作为 C++ 世界里标准库之外最知名的开放源码程序库，我们值得专门用一讲来讨论一下 Boost。

更多课程微信 laoxu_110

Boost 概览

Boost 的网站把 Boost 描述成为经过同行评审的、可移植的 C++ 源码库（peer-reviewed portable C++ source libraries）[1]。换句话说，它跟很多个人开源库不一样的地方在于，它的代码是经过评审的。事实上，Boost 项目的背后有很多 C++ 专家，比如发起人之一的 Dave Abarahams 是 C++ 标准委员会的成员，也是《C++ 模板元编程》一书 [2] 的作者。这也就使得 Boost 有了很不一样的特殊地位：它既是 C++ 标准库的灵感来源之一，也是 C++ 标准库的试验田。下面这些 C++ 标准库就源自 Boost：

- 智能指针
- thread
- regex
- random
- array
- bind
- tuple
- optional
- variant
- any
- string_view
- filesystem
- 等等

当然，将来还会有新的库从 Boost 进入 C++ 标准，如网络库的标准化就是基于 Boost.Asio 进行的。因此，即使相关的功能没有被标准化，我们也可能可以从 Boost 里看到某个功能可能会被标准化的样子——当然，最终标准化之后的样子还是经常有所变化的。

我们也可以在我们的编译器落后于标准、不能提供标准库的某个功能时使用 Boost 里的替代品。比如，我之前提到过老版本的 macOS 上苹果的编译器不支持 optional 和 variant。除了我描述的不正规做法，改用 Boost 也是方法之一。比如，对于 variant，所需的改动只是：

- 把包含 <variant> 改成包含 <boost/variant.hpp>
- 把代码中的 `std::variant` 改成 `boost::variant`

这样，就基本大功告成了。

作为一个准标准的库，很多环境里缺省会提供 Boost。这种情况下，在程序里使用 Boost 不会额外增加编译或运行时的依赖，减少了可能的麻烦。如果我需要某个功能，在标准库里没有，在 Boost 里有，我会很乐意直接使用 Boost 里的方案，而非另外去查找。如果我要使用非 Boost 的第三方库的话，那一般要么是 Boost 里没有，要么就是那个库比 Boost 里的要好用很多了。

鉴于 Boost 是一个库集合，当前版本（1.72）有 160 个独立库，即使写本书也不可能完整地讨论所有的库。这一讲里，我们也就管中窥豹式地浏览几个 Boost 库。具体你需要什么，还是得你自己回头去细细品味。

Boost 的安装

在主要的开发平台上，现在你都可以直接安装 Boost，而不需要自己从源代码编译了：

- 在 Windows 下使用 MSVC，我们可以使用 NuGet 安装（按需逐个安装）
- 在 Linux 下，我们可以使用系统的包管理器（如 apt 和 yum）安装（按需逐个安装，或一次性安装所有的开发需要的包）
- 在 macOS 下，我们可以使用 Homebrew 安装（一次性安装完整的 Boost）

如果你在某个平台上使用非缺省的编译器，如在 Windows 上或 macOS 上使用 GCC，一般就需要自己编译了，具体步骤请参见 Boost 的文档。不过，很多 Boost 库是完全不需要编译的，只需要把头文件加到编译器能找到的路径里就可以——如我们上一讲讨论的 Boost.Multiprecision 就是这样。我们讨论 Boost 库的时候，也会提一下使用这个库是否需要链接某个 Boost 库——需要的话，也就意味着需要编译和安装这个 Boost 库。

Boost.TypeIndex

TypeIndex 是一个很轻量级的库，它不需要链接，解决的也是使用模板时的一个常见问题，如何精确地知道一个表达式或变量的类型。我们还是看一个例子：

```
#include <iostream>
#include <typeinfo>
#include <utility>
#include <vector>
#include <boost/type_index.hpp>

using namespace std;
using boost::typeindex::type_id;
using boost::typeindex::
```

```
  type_id_with_cvr;

int main()
{
  vector<int> v;
  auto it = v.cbegin();

  cout << "*** Using typeid\n";
  cout << typeid(const int).name()
       << endl;
  cout << typeid(v).name() << endl;
  cout << typeid(it).name() << endl;

  cout << "*** Using type_id\n";
  cout << type_id<const int>() << endl;
  cout << type_id<decltype(v)>()
       << endl;
  cout << type_id<decltype(it)>()
       << endl;

  cout << "*** Using "
          "type_id_with_cvr\n";
  cout
    << type_id_with_cvr<const int>()
    << endl;
  cout << type_id_with_cvr<decltype(
            (v))>()
       << endl;
  cout << type_id_with_cvr<decltype(
            move((v)))>()
       << endl;
  cout << type_id_with_cvr<decltype(
            (it))>()
       << endl;
}
```

上面的代码里，展示了标准的 `typeid` 和 Boost 的 `type_id` 和 `type_id_with_cvr` 的使用。它们的区别是：

- `typeid` 是标准 C++ 的关键字，可以应用到变量或类型上，返回一个 `std::type_info`。我们可以用它的 `name` 成员函数把结果转换成一个字符串，但标准不保证这个字符串的可读性和唯一性。
- `type_id` 是 Boost 提供的函数模板，必须提供类型作为模板参数——所以对于表达式和变量我们需要使用 `decltype`。结果可以直接输出到 IO 流上。
- `type_id_with_cvr` 和 `type_id` 相似，但它获得的结果会包含 const/volatile 状态及引用类型。

上面程序在 MSVC 下的输出为：

```
*** Using typeid
int
class std::vector<int,class std::allocator<int> >
class std::_Vector_const_iterator<class std::_Vector_val<struct std::_Simple_types<int> >
>
*** Using type_id
int
class std::vector<int,class std::allocator<int> >
class std::_Vector_const_iterator<class std::_Vector_val<struct std::_Simple_types<int> >
>
*** Using type_id_with_cvr
int const
class std::vector<int,class std::allocator<int> > &
class std::vector<int,class std::allocator<int> > &&
class std::_Vector_const_iterator<class std::_Vector_val<struct std::_Simple_types<int> >
> &
```

在 GCC 下的输出为：

```
*** Using typeid
i
St6vectorIiSaIiEE
N9__gnu_cxx17__normal_iteratorIPKiSt6vectorIiSaIiEEEE
*** Using type_id
int
std::vector<int, std::allocator<int> >
__gnu_cxx::__normal_iterator<int const*, std::vector<int, std::allocator<int> > >
*** Using type_id_with_cvr
int const
std::vector<int, std::allocator<int> >&
std::vector<int, std::allocator<int> >&&
__gnu_cxx::__normal_iterator<int const*, std::vector<int, std::allocator<int> > >&
```

我们可以看到 MSVC 下 `typeid` 直接输出了比较友好的类型名称，但 GCC 下没有。此外，我们可以注意到：

- `typeid` 的输出忽略了 const 修饰，也不能输出变量的引用类型。
- `type_id` 保证可以输出友好的类型名称，输出时也不需要调用成员函数，但例子里它忽略了 `int` 的 const 修饰，也和 `typeid` 一样不能输出表达式的引用类型。
- `type_id_with_cvr` 可以输出 const/volatile 状态和引用类型，注意这种情况下模板参数必须包含引用类型，所以我用了 `decltype((v))` 这种写法，而不是 `decltype(v)`。如果你忘了这两者的区别，请复习一下 [第 8 讲] 的 `decltype`。

显然，除非你正在使用 MSVC，否则调试期 `typeid` 的用法完全应该用 Boost 的 `type_id` 来替代。另外，如果你的开发环境要求禁用 RTTI（运行时类型识别），那 `typeid` 在 Clang 和 GCC 下根本不能使用，而使用 Boost.TypeIndex 库仍然没有问题。

当然，上面说的前提都是你在调试中试图获得变量的类型，而不是要获得一个多态对象的运行时类型。后者还是离不开 RTTI 的——虽然你也可以用一些其他方式来模拟 RTTI，但我个人觉得一般的项目不太有必要这样做。下面的代码展示了 `typeid` 和 `type_id` 在获取对象类型上的差异：

```
#include <iostream>
#include <typeinfo>
#include <boost/type_index.hpp>

using namespace std;
using boost::typeindex::type_id;

class shape {
public:
  virtual ~shape() {}
};

class circle : public shape {};

#define CHECK_TYPEID(object, type) \
  cout << "typeid(" #object << ")" \
       << (typeid(object) ==       \
               typeid(type)        \
             ? " is "              \
             : " is NOT ")         \
       << #type << endl

#define CHECK_TYPE_ID(object,      \
                      type)        \
  cout << "type_id(" #object       \
       << ")"                      \
       << (type_id<decltype(       \
                 object)>() ==     \
               type_id<type>()     \
             ? " is "              \
             : " is NOT ")         \
       << #type << endl

int main()
{
  shape* ptr = new circle();
  CHECK_TYPEID(*ptr, shape);
  CHECK_TYPEID(*ptr, circle);
  CHECK_TYPE_ID(*ptr, shape);
  CHECK_TYPE_ID(*ptr, circle);
  delete ptr;
}
```

输出为：

```
typeid(*ptr) is NOT shape
typeid(*ptr) is circle
type_id(*ptr) is shape
type_id(*ptr) is NOT circle
```

Boost.Core

Core 里面提供了一些通用的工具，这些工具常常被 Boost 的其他库用到，而我们也可以使用，不需要链接任何库。在这些工具里，有些已经（可能经过一些变化后）进入了 C++ 标准，如：

- `addressof`，在即使用户定义了 `operator&` 时也能获得对象的地址
- `enable_if`，这个我们已经深入讨论过了（[第 14 讲]）
- `is_same`，判断两个类型是否相同，C++11 开始在 <type_traits> 中定义
- `ref`，和标准库的相同，我们在 [第 19 讲] 讨论线程时用过

我们在剩下的里面来挑几个讲讲。

boost::core::demangle

`boost::core::demangle` 能够用来把 `typeid` 返回的内部名称“反粉碎”（demangle）成可读的形式，看代码和输出应该就非常清楚了：

```
#include <iostream>
#include <typeinfo>
#include <vector>
#include <boost/core/demangle.hpp>

using namespace std;
using boost::core::demangle;

int main()
{
  vector<int> v;
  auto it = v.cbegin();

  cout << "*** Using typeid\n";
  cout << typeid(const int).name()
       << endl;
  cout << typeid(v).name() << endl;
  cout << typeid(it).name() << endl;

  cout << "*** Demangled\n";
  cout << demangle(typeid(const int)
                     .name())
       << endl;
  cout << demangle(typeid(v).name())
       << endl;
  cout << demangle(
            typeid(it).name())
       << endl;
}
```

GCC 下的输出为：

```
*** Using typeid
i
St6vectorIiSaIiEE
N9__gnu_cxx17__normal_iteratorIPKiSt6vectorIiSaIiEEEE
*** Demangled
int
std::vector<int, std::allocator<int> >
__gnu_cxx::__normal_iterator<int const*, std::vector<int, std::allocator<int> > >
```

如果你不使用 RTTI 的话，那直接使用 TypeIndex 应该就可以。如果你需要使用 RTTI、又不是（只）使用 MSVC 的话，`demangle` 就会给你不少帮助。

boost::noncopyable

`boost::noncopyable` 提供了一种非常简单也很直白的把类声明成不可拷贝的方式。比如，我们 [第 1 讲] 里的 `shape_wrapper`，用下面的写法就明确表示了它不允许被拷贝：

```
#include <boost/core/noncopyable.hpp>

class shape_wrapper
  : private boost::noncopyable {
  …
};
```

你当然也可以自己把拷贝构造和拷贝赋值函数声明成 = delete，不过，上面的写法是不是可读性更佳?

boost::swap

你有没有印象在通用的代码如何对一个不知道类型的对象执行交换操作？不记得的话，标准做法是这样的：

```
{
  using std::swap;
  swap(lhs, rhs);
}
```

即，我们需要（在某个小作用域里）引入 `std::swap`，然后让编译器在“看得到” `std::swap` 的情况下去编译 `swap` 指令。根据 ADL，如果在被交换的对象所属类型的名空间下有 `swap` 函数，那个函数会被优先使用，否则，编译器会选择通用的 `std::swap`。

似乎有点小啰嗦。使用 Boost 的话，你可以一行搞定：

```
boost::swap(lhs, rhs);
```

当然，你需要包含头文件 <boost/core/swap.hpp>。

Boost.Conversion

Conversion 同样是一个不需要链接的轻量级的库。它解决了标准 C++ 里的另一个问题，标准类型之间的转换不够方便。在 C++11 之前，这个问题尤为严重。在 C++11 里，标准引入了一系列的函数，已经可以满足常用类型之间的转换。但使用 Boost.Conversion 里的 `lexical_cast` 更不需要去查阅方法名称或动脑子去努力记忆。

下面是一个例子：

```
#include <iostream>
#include <stdexcept>
#include <string>
#include <boost/lexical_cast.hpp>

using namespace std;
using boost::bad_lexical_cast;
using boost::lexical_cast;

int main()
{
  // 整数到字符串的转换
  int d = 42;
  auto d_str =
    lexical_cast<string>(d);
  cout << d_str << endl;

  // 字符串到浮点数的转换
  auto f =
    lexical_cast<float>(d_str) /
    4.0;
  cout << f << endl;

  // 测试 lexical_cast 的转换异常
  try {
    int t = lexical_cast<int>("x");
    cout << t << endl;
  }
  catch (bad_lexical_cast& e) {
    cout << e.what() << endl;
  }

  // 测试标准库 stoi 的转换异常
  try {
    int t = std::stoi("x");
    cout << t << endl;
  }
  catch (invalid_argument& e) {
    cout << e.what() << endl;
  }
}
```

GCC 下的输出为：

```
42
10.5
bad lexical cast: source type value could not be interpreted as target
stoi
```

我觉得 GCC 里 `stoi` 的异常输出有点太言简意赅了……而 `lexical_cast` 的异常输出在不同的平台上有很好的一致性。

Boost.ScopeExit

我们说过 RAII 是推荐的 C++ 里管理资源的方式。不过，作为 C++ 程序员，跟 C 函数打交道也很正常。每次都写个新的 RAII 封装也有点浪费。Boost 里提供了一个简单的封装，你可以从下面的示例代码里看到它是如何使用的：

```
#include <stdio.h>
#include <boost/scope_exit.hpp>

void test()
{
  FILE* fp = fopen("test.cpp", "r");
  if (fp == NULL) {
    perror("Cannot open file");
  }
  BOOST_SCOPE_EXIT(&fp) {
    if (fp) {
      fclose(fp);
      puts("File is closed");
    }
  } BOOST_SCOPE_EXIT_END
  puts("Faking an exception");
  throw 42;
}

int main()
{
  try {
    test();
  }
  catch (int) {
    puts("Exception received");
  }
}
```

唯一需要说明的可能就是 `BOOST_SCOPE_EXIT` 里的那个 `&` 符号了——把它理解成 lambda 表达式的按引用捕获就对了（虽然 `BOOST_SCOPE_EXIT` 可以支持 C++98 的代码）。如果不需要捕获任何变量，`BOOST_SCOPE_EXIT` 的参数必须填为 `void`。

输出为（假设 test.cpp 存在）：

```
Faking an exception
File is closed
Exception received
```

使用这个库也只需要头文件。注意实现类似的功能在 C++11 里相当容易，但由于 ScopeExit 可以支持 C++98 的代码，因而它的实现还是相当复杂的。

Boost.Program_options

传统上 C 代码里处理命令行参数会使用 `getopt`。我也用过，比如在下面的代码中：

https://github.com/adah1972/breaktext/blob/master/breaktext.c

这种方式有不少缺陷：

- 一个选项通常要在三个地方重复：说明文本里，`getopt` 的参数里，以及对 `getopt` 的返回结果进行处理时。不知道你觉得怎样，我反正发生过改了一处、漏改其他的错误。
- 对选项的附加参数需要手工写代码处理，因而常常不够严格（C 的类型转换不够方便，尤其是检查错误）。

Program_options 正是解决这个问题的。这个代码有点老了，不过还挺实用；懒得去找特别的处理库时，至少这个伸手可用。使用这个库需要链接 boost_program_options 库。

下面的代码展示了代替上面的 `getopt` 用法的代码：

```
#include <iostream>
#include <string>
#include <stdlib.h>
#include <boost/program_options.hpp>

namespace po = boost::program_options;
using std::cout;
using std::endl;
using std::string;

string locale;
string lang;
int width = 72;
bool keep_indent = false;
bool verbose = false;

int main(int argc, char* argv[])
{
  po::options_description desc(
    "Usage: breaktext [OPTION]... "
    "<Input File> [Output File]\n"
    "\n"
    "Available options");
```

```
    "Available options");
  desc.add_options()
    ("locale,L",
     po::value<string>(&locale),
     "Locale of the console (system locale by default)")
    ("lang,l",
     po::value<string>(&lang),
     "Language of input (asssume no language by default)")
    ("width,w",
     po::value<int>(&width),
     "Width of output text (72 by default)")
    ("help,h", "Show this help message and exit")
    (",i",
     po::bool_switch(&keep_indent),
     "Keep space indentation")
    (",v",
     po::bool_switch(&verbose),
     "Be verbose");

  po::variables_map vm;
  try {
    po::store(
      po::parse_command_line(
        argc, argv, desc),
      vm);
  }
  catch (po::error& e) {
    cout << e.what() << endl;
    exit(1);
  }
  vm.notify();

  if (vm.count("help")) {
    cout << desc << "\n";
    exit(1);
  }
}
```

略加说明一下：

- `options_description` 是基本的选项描述对象的类型，构造时我们给出对选项的基本描述。
- `options_description` 对象的 `add_options` 成员函数会返回一个函数对象，然后我们直接用括号就可以添加一系列的选项。
- 每个选项初始化时可以有两个或三个参数，第一项是选项的形式，使用长短选项用逗号隔开的字符串（可以只提供一

种），最后一项是选项的文字描述，中间如果还有一项的话，就是选项的值描述。
- 选项的值描述可以用 `value`、`bool_switch` 等方法，参数是输出变量的指针。
- `variables_map`，变量映射表，用来存储对命令行的扫描结果；它继承了标准的 `std::map`。
- `notify` 成员函数用来把变量映射表的内容实际传送到选项值描述里提供的那些变量里去。
- `count` 成员函数继承自 `std::map`，只能得到 0 或 1 的结果。

这样，我们的程序就能处理上面的那些选项了。如果运行时在命令行加上 `-h` 或 `--help` 选项，程序就会输出跟原来类似的帮助输出——额外的好处是选项的描述信息较长时还能自动帮你折行，不需要手工排版了。建议你自己尝试一下，提供各种正确或错误的选项，来检查一下运行的结果。

当然现在有些更新的选项处理库，但它们应该都和 Program_options 更接近，而不是和 `getopt` 更接近。如果你感觉 Program_options 功能不足了，换一个其他库不会是件麻烦事。

Boost.Hana

Boost 里自然也有模板元编程相关的东西。但我不打算介绍 MPL、Fusion 和 Phoenix 那些，因为有些技巧，在 C++11 和 Lambda 表达式到来之后，已经略显得有点过时了。Hana 则不同，它是一个使用了 C++11/14 实现技巧和惯用法的新库，也和一般的模板库一样，只要有头文件就能使用。

Hana 里定义了一整套供**编译期**使用的数据类型和函数。我们现在看一下它提供的部分类型：

- `type`：把类型转化成对象（我们在 [第 13 讲] 曾经示例过相反的动作，把数值转化成对象），来方便后续处理。
- `integral_constant`：跟 `std::integral_constant` 相似，但定义了更多的运算符和语法糖。特别的，你可以用字面量来生成一个 `long long` 类型的 `integral_constant`，如 `1_c`。
- `string`：一个编译期使用的字符串类型。
- `tuple`：跟 `std::tuple` 类似，意图是当作编译期的 `vector` 来使用。
- `map`：编译期使用的关联数组。
- `set`：编译期使用的集合。

Hana 里的算法的名称跟标准库的类似，我就不一一列举了。下面的例子展示了一个基本用法：

```
#include <boost/hana.hpp>
namespace hana = boost::hana;

class shape {};
class circle {};
class triangle {};

int main()
{
  using namespace hana::literals;

  constexpr auto tup =
    hana::make_tuple(
      hana::type_c<shape*>,
      hana::type_c<circle>,
      hana::type_c<triangle>);

  constexpr auto no_pointers =
    hana::remove_if(
      tup, [](auto a) {
        return hana::traits::
          is_pointer(a);
      });

  static_assert(
    no_pointers ==
    hana::make_tuple(
      hana::type_c<circle>,
      hana::type_c<triangle>));
  static_assert(
    hana::reverse(no_pointers) ==
    hana::make_tuple(
      hana::type_c<triangle>,
      hana::type_c<circle>));
  static_assert(
    tup[1_c] == hana::type_c<circle>);
}
```

这个程序可以编译，但没有任何运行输出。在这个程序里，我们做了下面这几件事：

- 使用 `type_c` 把类型转化成 `type` 对象，并构造了类型对象的 `tuple`
- 使用 `remove_if` 算法移除了 `tup` 中的指针类型
- 使用静态断言确认了结果是我们想要的

- 使用静态断言确认了可以用 `reverse` 把 `tup` 反转一下
- 使用静态断言确认了可以用方括号运算符来获取 `tup` 中的某一项

可以看到，Hana 本质上以类似普通的运行期编程的写法，来做编译期的计算。上面展示的只是一些最基本的用法，而 Hana 的文档里展示了很多有趣的用法。尤其值得一看的是，文档中展示了如何利用 Hana 提供的机制，来自己定义 `switch_`、`case_`、`default_`，使得下面的代码可以通过编译：

```
boost::any a = 'x';
std::string r =
  switch_(a)(
    case_<int>([](auto i) {
      return "int: "s +
             std::to_string(i);
    }),
    case_<char>([](auto c) {
      return "char: "s +
             std::string{c};
    }),
    default_(
      [] { return "unknown"s; }));
assert(r == "char: x"s);
```

我个人认为很有意思。

内容小结

本讲我们对 Boost 的意义做了概要介绍，并蜻蜓点水地简单描述了若干 Boost 库的功能。如果你想进一步了解 Boost 的细节的话，就得自行查看文档了。

课后思考

请你考虑一下，我今天描述的 Boost 库里的功能是如何实现的。然后自己去看一下源代码（开源真是件大好事！），检查一下跟自己想象的是不是有出入。

参考资料

[1] Boost C++ Libraries. https://www.boost.org/

[2] David Abarahams and Aleksey Gurtovoy, C++ Template Metaprogramming. Addison-Wesley, 2004. 有中文版（荣耀译，机械工业出版社，2010 年）

精选留言

风羽星泉

老师的文章里提到：“实现类似（Boost.ScopeExit）的功能在 C++11 里相当容易“。能教我怎么实现吗?

2020-01-20 08:43

作者回复

定义一个RAII类模板，构造时接受一个lambda表达式，把它存到成员里。析构时调用这个成员就行啦。

2020-01-21 18:28

tt

老师，通过看某些源代码来学习现代C++，看Boost库的，是一个好的开始不?

2020-01-20 08:44

作者回复

可以看自己感兴趣的实现。都看（太多了）没必要。另外，Boost很多库都是前C++11的，有些技巧现在已经不必要了，这点留意一下。

2020-01-21 18:33

Sochooligan

来了，来了。

2020-01-20 08:19