18讲应用可变模板和tuple的编译期技巧

你好，我是吴咏炜。

今天我们讲一个特殊的专题，如何使用可变模板和 tuple 来完成一些常见的功能，尤其是编译期计算。

可变模板

可变模板 [1] 是 C++11 引入的一项新功能，使我们可以在模板参数里表达不定个数和类型的参数。从实际的角度，它有两个明显的用途：

- 用于在通用工具模板中转发参数到另外一个函数
- 用于在递归的模板中表达通用的情况（另外会有至少一个模板特化来表达边界情况）

我们下面就来分开讨论一下。

转发用法

以标准库里的 `make_unique` 为例，它的定义差不多是下面这个样子：

```
template <typename T,
          typename... Args>
inline unique_ptr<T>
make_unique(Args&&... args)
{
  return unique_ptr<T>(
    new T(forward<Args>(args)...));
}
```

这样，它就可以把传递给自己的全部参数转发到模板参数类的构造函数上去。注意，在这种情况下，我们通常会使用 `std::forward`，确保参数转发时仍然保持正确的左值或右值引用类型。

稍微解释一下上面三处出现的 `...`：

- `typename... Args` 声明了一系列的类型——`class...` 或 `typename...` 表示后面的标识符代表了一系列的类型。
- `Args&&... args` 声明了一系列的形参 `args`，其类型是 `Args&&`。
- `forward<Args>(args)...` 会在编译时实际逐项展开 `Args` 和 `args`，参数有多少项，展开后就是多少项。

举一个例子，如果我们需要在堆上传递一个 `vector<int>`，假设我们希望初始构造的大小为 100，每个元素都是 1，那我们可以这样写：

```
make_unique<vector<int>>(100, 1)
```

模板实例化之后，会得到相当于下面的代码：

```
template <>
inline unique_ptr<vector<int>>
make_unique(int&& arg1, int&& arg2)
{
  return unique_ptr<vector<int>>(
    new vector<int>(
      forward<int>(arg1),
      forward<int>(arg2)));
}
```

如前所述，`forward<Args>(args)...` 为每一项可变模板参数都以同样的形式展开。项数也允许为零，那样，我们在调用构造函数时也同样没有任何参数。

递归用法

我们也可以用可变模板来实现编译期递归。下面就是个小例子：

```
template <typename T>
constexpr auto sum(T x)
{
  return x;
}

template <typename T1, typename T2,
          typename... Targ>
constexpr auto sum(T1 x, T2 y,
                   Targ... args)
{
  return sum(x + y, args...);
}
```

在上面的定义里，如果 sum 得到的参数只有一个，会走到上面那个重载。如果有两个或更多参数，编译器就会选择下面那个重载，执行一次加法，随后你的参数数量就少了一个，因而递归总会终止到上面那个重载，结束计算。

要使用上面这个模板，我们就可以写出像下面这样的函数调用：

```
auto result = sum(1, 2, 3.5, x);
```

模板会这样依次展开：

```
sum(1 + 2, 3.5, x)
sum(3 + 3.5, x)
sum(6.5 + x)
6.5 + x
```

注意我们都不必使用相同的数据类型：只要这些数据之间可以应用 +，它们的类型无关紧要……

再看另一个复杂些的例子，函数的组合 [2]。如果我们有函数 f 和 函数 g，要得到函数的联用 $g \circ f$，其满足：

$$
(g \circ f)(x) = g(f(x))
$$

我们能不能用一种非常简单的方式，写不包含变量 x 的表达式来表示函数组合呢？答案是肯定的。

跟上面类似，我们需要写出递归的终结情况，单个函数的“组合”：

```
template <typename F>
auto compose(F f)
{
  return [f](auto&&... x) {
    return f(
      forward<decltype(x)>(x)...);
  };
}
```

上面我们仅返回一个泛型 lambda 表达式，保证参数可以转发到 `f`。记得我们在 [第 16 讲] 讲过泛型 lambda 表达式，本质上就是一个模板，所以我们按转发用法的可变模板来理解上面的 `...` 部分就对了。

下面是正常有组合的情况：

```
template <typename F,
          typename... Args>
auto compose(F f, Args... other)
{
  return [f,
          other...](auto&&... x) {
    return f(compose(other...)(
      forward<decltype(x)>(x)...));
  };
}
```

在这个模板里，我们返回一个 lambda 表达式，然后用 `f` 捕捉第一个函数对象，用 `args...` 捕捉后面的函数对象。我们用 `args...` 继续组合后面的部分，然后把结果传到 `f` 里面。

上面的模板定义我实际上已经有所简化，没有保持值类别。完整的包含完美转发的版本，请看参考资料 [3] 中的 functional.h 实现。

下面我们来试验一下使用这个 `compose` 函数。我们先写一个对输入范围中每一项都进行平方的函数对象：

```
auto square_list =
  [](auto&& container) {
    return fmap(
      [](int x) { return x * x; },
      container);
  };
```

我们使用了 [第 13 讲] 中给出的 `fmap`，而不是标准库里的 `transform`，是因为后者接口非函数式，无法组合——它要求参数给出输出位置的迭代器，会修改迭代器指向的内容，返回结果也只是单个的迭代器；函数式的接口则期望不修改参数的内容，结果完全在返回值中。

我们这儿用了泛型 lambda 表达式，是因为组合的时候不能使用模板，只能是函数对象或函数（指针）——如果我们定义一个 `square_list` 模板的话，组合时还得显式实例化才行（写成 `square_list<const vector<int>&>` 的样子），很不方便。

我们再写一个求和的函数对象：

```
auto sum_list =
  [](auto&& container) {
    return accumulate(
      container.begin(),
      container.end(), 0);
  };
```

那先平方再求和，就可以这样简单定义了：

```
auto squared_sum =
  compose(sum_list, square_list);
```

我们可以验证这个定义是可以工作的：

```
vector v{1, 2, 3, 4, 5};
cout << squared_sum(v) << endl;
```

我们会得到：

```
55
```

tuple

上面的写法虽然看起来还不错，但实际上有个缺陷：被 compose 的函数除了第一个（最右边的），其他的函数只能接收一个参数。要想进一步推进类似的技巧，我们得首先解决这个问题。

在 C++ 里，要通用地用一个变量来表达多个值，那就得看多元组——`tuple` 模板了 [4]。`tuple` 算是 C++98 里的 `pair` 类型的一般化，可以表达任意多个固定数量、固定类型的值的组合。下面这段代码约略地展示了其基本用法：

```
#include <algorithm>
#include <iostream>
#include <string>
#include <tuple>
#include <vector>

using namespace std;

// 整数、字符串、字符串的三元组
using num_tuple =
```

```
  tuple<int, string, string>;

ostream&
operator<<(ostream& os,
           const num_tuple& value)
{
  os << get<0>(value) << ','
     << get<1>(value) << ','
     << get<2>(value);
  return os;
}

int main()
{
  // 阿拉伯数字、英文、法文
  vector<num_tuple> vn{
    {1, "one",   "un"},
    {2, "two",   "deux"},
    {3, "three", "trois"},
    {4, "four",  "quatre"}};
  // 修改第 0 项的法文
  get<2>(vn[0]) = "une";
  // 按法文进行排序
  sort(vn.begin(), vn.end(),
       [](auto&& x, auto&& y) {
         return get<2>(x) <
                get<2>(y);
       });
  // 输出内容
  for (auto&& value : vn) {
    cout << value << endl;
  }
  // 输出多元组项数
  constexpr auto size = \
    tuple_size_v<num_tuple>;
  cout << "Tuple size is " << size << endl;
}
```

输出是：

```
2,two,deux
4,four,quatre
3,three,trois
1,one,une
```

> `Tuple size is 3`

我们可以看到：

- `tuple` 的成员数量由尖括号里写的类型数量决定。
- 可以使用 `get` 函数对 `tuple` 的内容进行读和写。（当一个类型在 `tuple` 中出现正好一次时，我们也可以传类型取内容，即，对我们上面的三元组，`get<int>` 是合法的，`get<string>` 则不是。）
- 可以用 `tuple_size_v` （在编译期）取得多元组里面的项数。

如果我们要用一个三项的 `tuple` 去调用一个函数，我们可以写类似这样的代码：

```
template <class F, class Tuple>
constexpr decltype(auto) apply(
  F&& f, Tuple&& t)
{
  return f(
    get<0>(forward<Tuple>(t)),
    get<1>(forward<Tuple>(t)),
    get<2>(forward<Tuple>(t)));
}
```

这似乎已经挺接近我们需要的形式了，但实际调用函数的参数项数会变啊……

我们已经有了参数的项数（使用 `tuple_size_v`），所以我们下面要做的是生成从 0 到项数减一之间的整数序列。标准库里已经定义了相关的工具，我们需要的就是其中的 `make_index_sequence` [5]，其简化实现如下所示：

```
template <class T, T... Ints>
struct integer_sequence {};

template <size_t... Ints>
using index_sequence =
  integer_sequence<size_t, Ints...>;

template <size_t N, size_t... Ints>
struct index_sequence_helper {
  typedef
    typename index_sequence_helper<
      N - 1, N - 1, Ints...>::type
      type;
};

template <size_t... Ints>
struct index_sequence_helper<
  0, Ints...> {
  typedef index_sequence<Ints...>
    type;
};

template <size_t N>
using make_index_sequence =
  typename index_sequence_helper<
    N>::type;
```

正如一般的模板代码，它看起来还是有点绕的。其要点是，如果我们给出 `make_index_sequence<N>`，则结果是 `integer_sequence<size_t, 0, 1, 2, …, N - 1>`（一下子想不清楚的话，可以拿纸笔来模拟一下模板的展开过程）。而有了这样一个模板的帮助之后，我们就可以写出下面这样的函数（同样，这是标准库里的 `apply` 函数模板 [6] 的简化版本）：

```
template <class F, class Tuple,
          size_t... I>
constexpr decltype(auto)
apply_impl(F&& f, Tuple&& t,
           index_sequence<I...>)
{
  return f(
    get<I>(forward<Tuple>(t))...);
}

template <class F, class Tuple>
constexpr decltype(auto)
apply(F&& f, Tuple&& t)
{
  return apply_impl(
    forward<F>(f),
    forward<Tuple>(t),
    make_index_sequence<
      tuple_size_v<
        remove_reference_t<
          Tuple>>>{});
}
```

我们如果有一个三元组 `t`，类型为 `tuple<int, string, string>`，去 `apply` 到一个函数 `f`，展开后我们得到 `apply_impl(f, t, index_sequence<0, 1, 2>{})`，再展开后我们就得到了上面那个有 `get<0>`、`get<1>`、`get<2>` 的函数调用形式。换句话说，我们利用一个计数序列的类型，可以在编译时展开 `tuple` 里的各个成员，并用来调用函数。

数值预算

上面的代码有点复杂，而且似乎并没有完成什么很重要的功能。我们下面看一个源自实际项目的例子。需求是，我们希望快速地计算一串二进制数中 1 比特的数量。举个例子，如果我们有十进制的 31 和 254，转换成二进制是 00011111 和 11111110，那我们应该得到 5 + 7 = 12。

显然，每个数字临时去数肯定会慢，我们应该预先把每个字节的 256 种情况记录下来。因而，如何得到这些计数值是个问题。在没有编译期编程时，我们似乎只能用另外一个程序先行计算，然后把结果填进去——这就很不方便很不灵活了。有了编译期编程，我们就不用写死，而让编译器在编译时帮我们计算数值。

利用 constexpr 函数，我们计算单个数值完全没有问题。快速定义如下：

```
constexpr int
count_bits(unsigned char value)
{
  if (value == 0) {
    return 0;
  } else {
    return (value & 1) +
           count_bits(value >> 1);
  }
}
```

可 256 个，总不见得把计算语句写上 256 遍吧？这就需要用到我们上面讲到的 index_sequence 了。我们定义一个模板，它的参数是一个序列，在初始化时这个模板会对参数里的每一项计算比特数，并放到数组成员里。

```
template <size_t... V>
struct bit_count_t {
  unsigned char
    count[sizeof...(V)] = {
      static_cast<unsigned char>(
        count_bits(V))...};
};
```

注意上面用 sizeof...(V) 可以获得参数的个数（在 tuple_size_v 的实现里实际也用到它了）。如果我们模板参数传 0, 1, 2, 3，结果里面就会有个含 4 项元素的数组，数值分别是对 0、1、2、3 的比特计数。

然后，我们当然就可以利用 make_index_sequence 来展开计算了，想产生几项就可以产生几项。不过，要注意到 make_index_sequence 的结果是个类型，不能直接用在 bit_count_t 的构造中。我们需要用模板匹配来中转一下：

```
template <size_t... V>
bit_count_t<V...> get_bit_count(
  index_sequence<V...>)
{
  return bit_count_t<V...>();
}

auto bit_count = get_bit_count(
  make_index_sequence<256>());
```

得到 bit_count 后，我们要计算一个序列里的比特数就只是轻松查表相加了，此处不再赘述。

内容小结

今天我们讨论了在编译期处理不确定数量的参数和类型的基本语言特性，可变模板，以及可以操控可变模板的重要工具——tuple 和 index_sequence。用好这些工具，可以让我们轻松地完成一些编译期计算的工作。

课后思考

请考虑一下：

1. 我展示了 compose 带一个或更多参数的情况。你觉得 compose 不带任何参数该如何定义？它有意义吗？
2. 有没有可能不用 index_sequence 来初始化 bit_count？如果行，应该如何实现？
3. 作为一个挑战，你能自行实现出 make_integer_sequence 吗？

期待你的答案。

参考资料

[1] cppreference.com, “Parameter pack”. https://en.cppreference.com/w/cpp/language/parameter_pack

[1a] cppreference.com, “形参包”. https://zh.cppreference.com/w/cpp/language/parameter_pack

[2] Wikipedia, “Function composition”. https://en.wikipedia.org/wiki/Function_composition

[2a] 维基百科, “复合函数”. https://zh.wikipedia.org/zh-cn/复合函数

[3] 吴咏炜, nvwa. https://github.com/adah1972/nvwa

[4] cppreference.com, “std::tuple”. https://en.cppreference.com/w/cpp/utility/tuple

[4a] cppreference.com, “std::tuple”. https://zh.cppreference.com/w/cpp/utility/tuple

[5] cppreference.com, “std::integer_sequence”. https://en.cppreference.com/w/cpp/utility/integer_sequence

[5a] cppreference.com, “std::integer_sequence”. https://zh.cppreference.com/w/cpp/utility/integer_sequence

[6] cppreference.com, “std::apply”. https://en.cppreference.com/w/cpp/utility/apply

[6a] cppreference.com, “std::apply”. https://zh.cppreference.com/w/cpp/utility/apply

精选留言

hello world

compose那是完全没看懂唉，还有sequence那...

2020-01-09 07:40

作者回复

给个提示，到下面这个网站上看看模板是如何展开的：

https://cppinsights.io/

2020-01-09 09:44

tt

回过头来看这一课，才发现在make_index_sequence的例子中，最终的目标竟然是实例化后的模板的非类型实参而不是实例化后的模板对象本身。然后把这些非类型实参包用get<>模板展开。

仔细想想，这当然是可以的，因为可变参数模板处理的就是参数包啊，而参数包可以是非类型相关的

2020-02-29 16:21

三味

17和18讲可以说，真是劝退不少非c++读者吧。。。

个人认为，如果不是需要写库，这两节的内容应该用得也不多吧。

作为一名图形工作者，我看这些东西，其实是因为好多图形库（说的就是你，CGAL）都是模板代码，看得眼疼。。。
实用更优先。当然，理解这些这几讲的内容还是很有帮助的。
顺便说一句，留言区贴代码实在是太费劲了。。。极客时间不好好搞搞Markdown回复格式么？好多问题还要贴代码的，有的就是挺长的。。
试着回答一下问题：
1. compose不带任何参数，

```
template<class... Args>
auto compose() {
return [](auto&&... args) { return compose<Args...>(); };
}
```

关于不带参数的意义，我理解的是，没有参数，那么就没有要执行的操作，那么就什么都不执行，返回个空。再深挖掘我就想不到了。。。这里我是为了形式上的统一，返回了一个依然什么都不做的自身。

2. 想不到其他方法。。要再预编译阶段就展开256个数值。。我还是等等答案吧。。

3. 自己实现是不可能的。这辈子都不可能。之前我看别人的代码，有通过借助写了一个辅助的PushBack操作来实现。这个自然不是我原创，拿来主义：

```
template <class T, T... Ints>
struct integer_sequence {};

template <size_t... Ints>
using index_sequence = integer_sequence<size_t, Ints...>;

template <size_t, typename T>
struct push_back{};

template <size_t N, size_t... Ints>
struct push_back<N, index_sequence<Ints...>> {
using type = index_sequence<Ints..., N>;
};

template <size_t N>
struct index_sequence_helper {
using type = typename push_back<N-1, typename index_sequence_helper<N-1>::type>::type;
};

template <>
struct index_sequence_helper<1> {
using type = index_sequence<0>;
};

template <size_t N>
using make_index_sequence = typename index_sequence_helper<N>::type;
```

2020-01-10 16:11

作者回复

如果极客时间要支持留言用 Markdown，恐怕也得是可选的，否则有些不用 Markdown 的就要晕菜了……倒是长度限制有点妨碍贴代码。

1. 代码对。

意义回头我再说。别人的代码我就不评价了。反正这个最后我会贴个参考答案。

2020-01-10 22:42

罗乾林

1、看了老师的nvwa：

```
inline auto compose()
{
return [](auto&& x) -> decltype(auto)
{
return std::forward<decltype(x)>(x);
};
}
```

感觉还是为了使用方便，真的需要还可以这样：compose(std::identity{})

第3题在网上查了些资料，发现很复杂。

本节的思考题都好难，求老师解答。

2020-01-07 11:16

作者回复

呃，compose() 返回的正是 identity（C++20 才有）啊。

这讲的题目都会有个参考答案的。我会最后公布。

2020-01-07 18:16

禾桃

```
template <typename F>
auto compose(F f)
{
return [f](auto&&... x) {
return f(x...);
};
}
```

貌似用compiler(gcc version 4.8.5 20150623) 就会遇到下面编译错误

```
// In function ‘auto compose(F)’:
// error: expansion pattern ‘auto&&’ contains no argument packs
// return [f](auto&&... x) {
```

用compiler(gcc version 8.3.1 20190311)就不会有问题。

如果公司目前只允许用(gcc version 4.8.5 20150623)，请问有什么workaround?

谢谢!

2020-01-07 11:07

作者回复

--- 更新 ---

我费了九牛二虎之力，终于把例子改得能在 gcc 4.8 下工作了。我觉得你不想维护这样的代码的。

这儿空间不够。我放在这里：

http://wyw.dcweb.cn/download.asp?path=&file=jike_18_gcc48.cpp

我觉得升级 GCC 绝对是更好的主意。

--- 原回复 ---
没啥好办法……泛型lambda至少要求gcc 4.9。只能不用这类功能了。

手写一个函数对象模板也许可以完成这个功能（让 f 成为其数据成员）。你可以试试看。我暂时没时间试验。
2020-01-07 18:36


禾桃
#1
template <typename F,
typename... Args>
auto compose(F f, Args... args)
{
return [f, args...]() {
return f(
compose(args...)());
};
}

template <typename F>
auto compose(F f)
{
return f;
}

如果这些函数都在函数体内操作一个公用的数据，而且这些函数依次执行的顺序反应了一定的工程需求，那么就是有意义的吧。
2020-01-07 08:05

作者回复
你这个compose带参数了呀……
2020-01-07 18:39


李亮亮
N-->(N-1, N-1)-->(N-2, N-2, N-1)-->(1, 1 , 2 ....N-1)-->(0, 0, 1, 2...N-1)
2020-01-06 20:57

作者回复
学得挺快。
2020-01-06 23:43