13 | 编译期能做些什么？一个完整的计算世界

2019-12-25 吴咏炜

讲述：吴咏炜
时长 12:49　大小 8.81M

你好，我是吴咏炜。

上一讲我们简单介绍了模板的基本用法及其在泛型编程中的应用。这一讲我们来看一下模板的另外一种重要用途——编译期计算，也称作“模板元编程”。

编译期计算

首先，我们给出一个已经被证明的结论：C++ 模板是图灵完全的 [1]。这句话的意思是，使用 C++ 模板，你可以在编译期间模拟一个完整的图灵机，也就是说，可以完成任何的计算任务。

当然，这只是理论上的结论。从实际的角度，我们并不**想**、也不可能在编译期完成所有的计算，更不用说编译期的编程是很容易让人看不懂的——因为这并不是语言设计的初衷。即便如此，我们也还是需要了解一下模板元编程的基本概念：它仍然有一些实用的场景，并且在实际的工程中你也可能会遇到这样的代码。虽然我们在开篇就说过不要炫技，但使用模板元编程写出的代码仍然是可理解的，尤其是如果你对递归不发怵的话。

好，闲话少叙，我们仍然拿代码说话：

复制代码

```
template <int n>
struct factorial {
  static const int value =
    n * factorial<n - 1>::value;
};

template <>
struct factorial<0> {
  static const int value = 1;
};
```

上面定义了一个递归的阶乘函数。可以看出，它完全符合阶乘的递归定义：

$$
\begin{aligned}
0! &= 1 \\
n! &= n \times (n-1)!
\end{aligned}
$$

除了顺序有特定的要求——先定义，才能特化——再加语法有点特别，代码基本上就是这个数学定义的简单映射了。

那我们怎么知道这个计算是不是在编译时做的呢？我们可以直接看编译输出。下面直接贴出对上面这样的代码加输出（`printf("%d\n", factorial<10>::value);`）在 x86-64 下的编译结果：

复制代码

```
.LC0:
        .string "%d\n"
main:
        push    rbp
```

```
        mov     rbp, rsp
        mov     esi, 3628800
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        mov     eax, 0
        pop     rbp
        ret
```

我们可以明确看到，编译结果里明明白白直接出现了常量 3628800。上面那些递归什么的，完全都没有了踪影。

如果我们传递一个负数给 `factorial` 呢？这时的结果就应该是编译期间的递归溢出。如 GCC 会报告：

> fatal error: template instantiation depth exceeds maximum of 900 (use -ftemplate-depth= to increase the maximum)

如果把 `int` 改成 `unsigned`，不同的编译器和不同的标准选项会导致不同的结果。有些情况下错误信息完全不变，有些情况下则会报负数不能转换到 `unsigned`。通用的解决方案是使用 `static_assert`，确保参数永远不会是负数。

复制代码

```
template <int n>
struct factorial {
  static_assert(
    n >= 0,
    "Arg must be non-negative");
  static const int value =
    n * factorial<n - 1>::value;
};
```

这样，当 `factorial` 接收到一个负数作为参数时，就会得到一个干脆的错误信息：

> error: static assertion failed: Arg must be non-negative

下面我们看一些更复杂的例子。这些例子不是为了让你真的去写这样的代码，而是帮助你充分理解编译期编程的强大威力。如果这些例子你都完全掌握了，那以后碰到小的模板问题，你一定可以轻松解决，完全不在话下。

回想上面的例子，我们可以看到，要进行编译期编程，最主要的一点，是需要把计算转变成类型推导。比如，下面的模板可以代表条件语句：

复制代码

```
template <bool cond,
          typename Then,
          typename Else>
struct If;

template <typename Then,
          typename Else>
struct If<true, Then, Else> {
  typedef Then type;
};

template <typename Then,
          typename Else>
struct If<false, Then, Else> {
  typedef Else type;
};
```

`If` 模板有三个参数，第一个是布尔值，后面两个则是代表不同分支计算的类型，这个类型可以是我们上面定义的任何一个模板实例，包括 `If` 和 `factorial`。第一个 struct 声明规定了模板的形式，然后我们不提供通用定义，而是提供了两个特化。第一个特化是真的情况，定义结果 `type` 为 `Then` 分支；第二个特化是假的情况，定义结果 `type` 为 `Else` 分支。

我们一般也需要循环：

复制代码

```
template <bool condition,
          typename Body>
struct WhileLoop;

template <typename Body>
struct WhileLoop<true, Body> {
  typedef typename WhileLoop<
```

```
    Body::cond_value,
    typename Body::next_type>::type
    type;
};

template <typename Body>
struct WhileLoop<false, Body> {
  typedef
    typename Body::res_type type;
};

template <typename Body>
struct While {
  typedef typename WhileLoop<
    Body::cond_value, Body>::type
    type;
};
```

这个循环的模板定义稍复杂点。首先，我们对循环体类型有一个约定，它必须提供一个静态数据成员，`cond_value`，及两个子类型定义，`res_type` 和 `next_type`：

`cond_value` 代表循环的条件（真或假）

`res_type` 代表退出循环时的状态

`next_type` 代表下面循环执行一次时的状态

这里面比较绕的地方是用类型来代表执行状态。如果之前你没有接触过函数式编程的话，这个在初学时有困难是正常的。把例子多看两遍，自己编译、修改、把玩一下，就会渐渐理解的。

排除这个抽象性，模板的定义和 `If` 是类似的，虽然我们为方便使用，定义了两个模板。`WhileLoop` 模板有两个模板参数，同样用特化来决定走递归分支还是退出循环分支。`While` 模板则只需要循环体一个参数，方便使用。

如果你之前模板用得不多的话，还有一个需要了解的细节，就是用 `::` 取一个成员类型、并且 `::` 左边有模板参数的话，得额外加上 `typename` 关键字来标明结果是一个类型。上面循环模板的定义里就出现了多次这样的语法。MSVC 在这方面往往比较宽松，不写 `typename` 也不会报错，但这是不符合 C++ 标准的用法。

为了进行计算，我们还需要通用的代表数值的类型。下面这个模板可以通用地代表一个整数常数：

复制代码

```
template <class T, T v>
struct integral_constant {
  static const T value = v;
  typedef T value_type;
  typedef integral_constant type;
};
```

`integral_constant` 模板同时包含了整数的类型和数值，而通过这个类型的 `value` 成员我们又可以重新取回这个数值。有了这个模板的帮忙，我们就可以进行一些更通用的计算了。下面这个模板展示了如何使用循环模板来完成从 1 加到 n 的计算：

复制代码

```
template <int result, int n>
struct SumLoop {
  static const bool cond_value =
    n != 0;
  static const int res_value =
    result;
  typedef integral_constant<
    int, res_value>
    res_type;
  typedef SumLoop<result + n, n - 1>
    next_type;
};

template <int n>
struct Sum {
  typedef SumLoop<0, n> type;
};
```

然后你使用 `While<Sum<10>::type>::type::value` 就可以得到 1 加到 10 的结果。虽然有点绕，但代码实质就是在编译期间进行了以下的计算：

复制代码

```
int result = 0;
while (n != 0) {
  result = result + n;
```

```
  n = n - 1;
}
```

估计现在你的头已经很晕了。但我保证，这一讲最难的部分已经过去了。实际上，到现在为止，我们讲的东西还没有离开 C++98。而我们下面几讲里很快就会讲到，如何在现代 C++ 里不使用这种麻烦的方式也能达到同样的效果。

编译期类型推导

C++ 标准库在 <type_traits> 头文件里定义了很多工具类模板，用来提取某个类型（type）在某方面的特点（trait）[2]。和上一节给出的例子相似，这些特点既是类型，又是常值。

为了方便地在值和类型之间转换，标准库定义了一些经常需要用到的工具类。上面描述的 `integral_constant` 就是其中一个（我的定义有所简化）。为了方便使用，针对布尔值有两个额外的类型定义：

复制代码

```
typedef std::integral_constant<
  bool, true> true_type;
typedef std::integral_constant<
  bool, false> false_type;
```

这两个标准类型 `true_type` 和 `false_type` 经常可以在函数重载中见到。有一个工具函数常常会写成下面这个样子：

复制代码

```
template <typename T>
class SomeContainer {
public:
  …
  static void destroy(T* ptr)
  {
    _destroy(ptr,
      is_trivially_destructible<
        T>());
  }

private:
```

```
13    static void _destroy(T* ptr,
14                         true_type)
15    {}
16    static void _destroy(T* ptr,
17                         false_type)
18    {
19      ptr->~T();
20    }
21  };
```

类似上面，很多容器类里会有一个 `destroy` 函数，通过指针来析构某个对象。为了确保最大程度的优化，常用的一个技巧就是用 `is_trivially_destructible` 模板来判断类是否是可平凡析构的——也就是说，不调用析构函数，不会造成任何资源泄漏问题。模板返回的结果还是一个类，要么是 `true_type`，要么是 `false_type`。如果要得到布尔值的话，当然使用 `is_trivially_destructible<T>::value` 就可以，但此处不需要。我们需要的是，使用 `()` 调用该类型的构造函数，让编译器根据数值类型来选择合适的重载。这样，在优化编译的情况下，编译器可以把不需要的析构操作彻底全部删除。

像 `is_trivially_destructible` 这样的 trait 类有很多，可以用来在模板里决定所需的特殊行为：

- is_array
- is_enum
- is_function
- is_pointer
- is_reference
- is_const
- has_virtual_destructor
- ...

这些特殊行为判断可以是像上面这样用于决定不同的重载，也可以是直接用在模板参数甚至代码里（记得我们是可以直接得到布尔值的）。

除了得到布尔值和相对应的类型的 trait 模板，我们还有另外一些模板，可以用来做一些类型的转换。以一个常见的模板 `remove_const` 为例（用来去除类型里的 const 修饰），它的定义大致如下：

```
template <class T>
struct remove_const {
  typedef T type;
};
template <class T>
struct remove_const<const T> {
  typedef T type;
};
```

同样，它也是利用模板的特化，针对 const 类型去掉相应的修饰。比如，如果我们对 `const string&` 应用 `remove_const`，就会得到 `string&`，即，`remove_const<const string&>::type` 等价于 `string&`。

这里有一个细节你要注意一下，如果对 `const char*` 应用 `remove_const` 的话，结果还是 `const char*`。原因是，`const char*` 是指向 `const char` 的指针，而不是指向 `char` 的 const 指针。如果我们对 `char * const` 应用 `remove_const` 的话，还是可以得到 `char*` 的。

简易写法

如果你觉得写 `is_trivially_destructible<T>::value` 和 `remove_const<T>::type` 非常啰嗦的话，那你绝不是一个人。在当前的 C++ 标准里，前者有增加 `_v` 的编译时常量，后者有增加 `_t` 的类型别名：

```
template <class T>
inline constexpr bool
  is_trivially_destructible_v =
    is_trivially_destructible<
      T>::value;
```

```
template <class T>
using remove_const_t =
  typename remove_const<T>::type;
```

复制代码

至于什么是 `constexpr`，我们会单独讲。`using` 是现代 C++ 的新语法，功能大致与 `typedef` 相似，但 `typedef` 只能针对某个特定的类型，而 `using` 可以生成别名模板。目前我们只需要知道，在你需要 trait 模板的结果数值和类型时，使用带 `_v` 和 `_t` 后缀的模板可能会更方便，尤其是带 `_t` 后缀的类型转换模板。

通用的 fmap 函数模板

你应当多多少少听到过 map-reduce。抛开其目前在大数据应用中的具体方式不谈，从概念本源来看，map [3] 和 reduce [4] 都来自函数式编程。下面我们演示一个 map 函数（当然，在 C++ 里它的名字就不能叫 `map` 了），其中用到了目前为止我们学到的多个知识点：

复制代码

```
template <
  template <typename, typename>
  class OutContainer = vector,
  typename F, class R>
auto fmap(F&& f, R&& inputs)
{
  typedef decay_t<decltype(
    f(*inputs.begin()))>
    result_type;
  OutContainer<
    result_type,
    allocator<result_type>>
    result;
  for (auto&& item : inputs) {
    result.push_back(f(item));
  }
  return result;
}
```

我们：

用 `decltype` 来获得用 `f` 来调用 `inputs` 元素的类型（参考第 8 讲）；

用 `decay_t` 来把获得的类型变成一个普通的值类型；

缺省使用 `vector` 作为返回值的容器，但可以通过模板参数改为其他容器；

使用基于范围的 for 循环来遍历 `inputs`，对其类型不作其他要求（参考第 7 讲）；

存放结果的容器需要支持 `push_back` 成员函数（参考第 4 讲）。

下面的代码可以验证其功能：

复制代码

```
vector<int> v{1, 2, 3, 4, 5};
int add_1(int x)
{
    return x + 1;
}

auto result = fmap(add_1, v);
```

在 `fmap` 执行之后，我们会在 `result` 里得到一个新容器，其内容是 2, 3, 4, 5, 6。

内容小结

本讲我们介绍了模板元编程的基本概念和例子，其本质是**把计算过程用编译期的类型推导和类型匹配表达出来**；然后介绍 type traits 及其基本用法；最后我们演示了一个简单的高阶函数 map，其实现中用到了我们目前已经讨论过的一些知识点。

课后思考

这一讲的内容可能有点烧脑，请你自行实验一下例子，并找一两个简单的算法用模板元编程的方法实现一下，看看能不能写出来。

如果有什么特别想法的话，欢迎留言和我分享交流。

参考资料

[1] Todd L. Veldhuizen, “C++ templates are Turing complete”. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.3670

[2] cppreference.com, “Standard library header <type_traits>”.
https://en.cppreference.com/w/cpp/header/type_traits

[2a] cppreference.com, “标准库头文件 <type_traits>”.
https://zh.cppreference.com/w/cpp/header/type_traits

[3] Wikipedia, “Map (higher-order function)”.
https://en.wikipedia.org/wiki/Map_(higher-order_function)

[4] Wikipedia, “Fold (higher-order function)”.
https://en.wikipedia.org/wiki/Fold_(higher-order_function)

上一篇　12 | 编译期多态：泛型编程和模板入门

下一篇　14 | SFINAE：不是错误的替换失败是怎么回事?

精选留言 (11)

写留言

三味
2019-12-30

这个课后思考题真不是盖得。。。还看到了5分钟编译期堆排序的段子。。。看得我心惊胆战。。。
不过还是一堆百度写出了一个，针对int型，写出从[0, N)的编译期生成数组的例子。。。
兼容其他类型还要再额外写好多代码。。偷懒直接写了int型。。。
我在msvc下测试了，应该没问题。。。用到了模板不定参数，其他不知道什么技巧的方...
展开


总统老唐
2019-12-27

吴老师，关于最后这个例子，有两个小问题：
1，我们平时定义一个 vector 的时候，一般并不会写成 vector<int, allocator<int>> vec 这种形式，为什么模板函数里面定义返回值 result 时，需要多一个 allocator?
2，fmap函数的入参和for循环，全都用的右值引用，有什么特殊考量么?
展开

作者回复: 1. 因为模板的定义就是这个样子，虽然我们平时第二个参数用的都是默认模板参数。

2. 不是右值引用，是转发引用。复习一下第 3 讲结尾部分吧。


总统老唐
2019-12-27

吴老师，看了你的While模板，试着想把这个计算累加的功能扩展一下，输入任意两个数，可以求他们之间的数的累加和，代码如下：
template <int from, int to, int sum>
struct SumAnyTwo_A
{...
展开

作者回复: 我真是自作自受，得看这样的代码。😁

1. 我给的这些模板只是说明能力的，肯定不是让你真的这么写代码的。比如你这种累加，可以考虑这样写（只考虑了正向）：

template <int from, int to, int sum, bool ending>
struct sum_two_op;

```
template <int from, int to, int sum>
struct sum_two_op<from, to, sum, false> {
    static const int value = sum;
};

template <int from, int to, int sum>
struct sum_two_op<from, to, sum, true> {
    static const int value =
        sum_two_op<from + 1, to, sum + from, (from < to)>::value;
};

template <int from, int to>
struct sum_two {
    static const int value = sum_two_op<from, to, 0, (from <= to)>::value;
};
```

2. 你已经很靠近了，只犯了一个小错误，SumTwo 里的方向。应该是：

```
template <int from, int to>
struct SumTwo {
    typedef SumAnyTwo<(from < to), from, to> type;
};
```

另外，你目前的处理对于 from==to 的情况有点问题。可参考我上面的写法，那是可以正确处理的。

3

zhang

2019-12-26

您好，我想问一个mutex相关的问题，虽然这部分内容以后会讲，但我现在工作中有一个疑问，麻烦您看一下，谢谢。
代码简写如下：
class Mutex {
public:...

展开

作者回复: 对你的业务场景不熟悉，随便评论几句。

- C++11 里有现成的 mutex、condition_variable 和 unique_lock。
- Client::Lock 和 Client::Unlock 似乎没有用处。

・变量 pending 的命名让人困惑：收到数据了，“挂起”标志被设为真，然后发送数据就能继续往下执行了？连续两次 sendData 中间必须有一次 recvData 才行，而且 sendData 里的等待是在发送之后？这块感觉有问题。
・inter_mutex 和 inter_protect 本身目前没有看出问题。

2

安静的雨
2019-12-25

模版编程很有趣，期待老师的更新。

展开

作者回复: 觉得有趣就好，这个我们要讲上好几讲的。

总统老唐
2019-12-25

记得吴老师之前预告过，这一节可能会比较难，确实被难住了。在第一个 If 模板这里就被卡住了，老师能给个简单的例子来说明这个 If 模板该如何使用么？

展开

作者回复: 下面的函数和模板是基本等价的：

```
int foo(int n)
{
  if (n == 2 || n == 3 || n == 5) {
    return 1;
  } else {
    return 2;
  }
}

template <int n>
struct Foo {
  typedef typename If<
    (n == 2 || n == 3 || n == 5),
    integral_constant<int, 1>,
    integral_constant<int, 2>>::type
    type;
```

};

你可以输出 foo(3)，也可以输出 Foo<3>::type::value。


禾桃
2019-12-25

脑壳儿疼的兄弟姐妹们，我这有个小偏方， 哈哈

While< Sum<2>::type >::type::value 实例化(instantiation)过程
--> While< SumLoop<0, 2> >::type::value
--> WhileLoop<SumLoop<0, 2>::cond_value, SumLoop<0, 2>>::type::value...
展开

作者回复: 对，对于模板，就是要在脑子里或纸上、电脑上把它展开……☺


小一日一
2019-12-25

#include <iostream>
#include <vector>
#include <type_traits>

using namespace std;...
展开

作者回复: 试试 c++1y、c++14 等标准选项了。这个 GCC 太老了……我要求 C++17、GCC 7 的。


2


李义盛
2019-12-25

一到模板就处于看不懂状态了。
展开

作者回复: 拿纸笔来展开试试？实际上就是一种展开而已。

禾桃
2019-12-25

“常用的一个技巧就是用 is_trivially_destructible 模板来判断类是否是可平凡析构的——也就是说，不调用析构函数，不会造成任何资源泄漏问题。”

麻烦解释一下，
#1 这个类模版是如何识别 “，不调用析构函数，不会造成任何资源泄漏问题”？ 这的资...
展开

> 作者回复: 1 是有点编译器魔法的。如果你有析构函数，或者你没有析构函数但有个非 POD 数据成员，is_trivially_destructible 就不成立了。
>
> 2 trivial 是很常见的数学术语，没什么特别的。见：
>
> https://baike.baidu.com/item/%E5%B9%B3%E5%87%A1/16739977
> https://zh.wikipedia.org/wiki/%E5%B9%B3%E5%87%A1_(%E6%95%B8%E5%AD%B8)
> https://en.wikipedia.org/wiki/Triviality_(mathematics)

hello world
2019-12-25

一直对模板元编程感兴趣，但总是搞不明白，今天学习很有收获，特别是最后的fmap，感谢老师，记得模板编程还有policy之类的东西，老师之后在编译期这方面还会更详细讲解吗

> 作者回复: 编译期要连续讲到第 18 讲，甚至之后还会有提到的机会。你喜欢那是最好了。我是怕很多人会被编译期编程吓退呢。😅
>
> policy 这个概念不单独讲，但我觉得在讨论了使用常数来对模板进行特化之后，这个概念应该没有特别之处。我们的例子倒是会有标准库提供的 policy。😁