C++ template Metaprograming

templates 模板是 C 中最优雅的一部分。

0 订阅 · 0 篇文章 · 订阅专栏

目录

前言
template metafunction
type traits
模板元函数使用类型参数
使用模型作为 “函数值”
C++ 库模板元函数的型！
编译期的 “if”
SFINAE
C++ 库模板元函数的型？ && integral_constant
关于模板元函数的调用语法
is_void
模板元编程 && variable template
Unevaluated Operands
std::declval
void_t
重写 is_copy_assignable
总结

前言

最几天刚看了Cppcon，Walter E.Brown关于metaprogramming的演讲，确实精妙，感觉在模板元编程入门到新手以上，只要学习以过高深，还是需要花点时间学习的。而这个专题的编写目的，即是给您以简单的metaprogramming的入门教程，帮助大家快速完成编程的“炼金术”，这是Brown老师的目标，我们会跟随他完成编程，但仅仅是加入的不同条件。

什么是metaprogramming？

- 利用编译期代码进行编程的技巧。
- 模板元编程是完编程的一种，利用模板的实例化机制，来实现编译期 UB 计算。

模板元编程之作用：

- 提高代码的灵活性。
- 提高运行期的效率。

模板元编程即将运行期的工作移到编译期完成，这是需要代价的，即编译期时间增加，换来的就是运行期的效率。

notice：

- 在模板元编程中，常见：你的程序等价于pure-time ==> compile-time。（没有) mutability ，2) 虚函数，因为虚函数是于运行期，3) IO 功能等一系列会发生副作用的东西。

在接受完编程中：

- 没有可变性。
- 没有可变性。
- 所有东西都一旦定义！！！（我认为这点很重要啊的，所以单列出来）

很多人认为模板元编程很难，因是因为他们不熟悉，everything都是从不熟悉到熟悉。
在接下来的文章中，没有这行规：

- no run-time
- no run-time
- no run-time（我必须让我说你们可以记住！！！）

template metafunction

```
template <int N>
struct abs {
    static_assert(N != INT_MIN); // 防止溢出的 -2147
    static int const/enum value = N > 0 ? N : -N; // 用条件表达式为 value 赋值的常量
                                                  // 必须是常量表达式，编译期能知
};
```

- 这是一个最简单不过的模板类，也是它全像函数一样工作，所以我们称之为模板元函数。
- 模板类是一个有，用法上与全局函数编译期等价的。
- static_assert是C++17引入的静态断言，因为INT_MIN的特殊值溢出了。
- 这里我们是调用一个，定义了一个static的value变量并赋值给，这实际上是是我们的返回值，并且是编译期的，这是，编译期的全局函数。
- 这里的value是一个常量表达式，上述条件表达式。

特别可以函数的型是：

```
int const/enum N = 10;
int const/enum value = abs<N>();
```

- 在上面，abs是模板元函数名称，<>是参数，value的结果是返回值。
- so，它像函数一样工作。

每的可以像函数一样工作，编译期间的模板元函数，那么，yes，它也应该可以调用自己，递归。

```
// 这是一个编译期的 factorial 函数
// 模板元函数的一个特点是它一定工作，因为它的工作是编译期完成的
template <int N>
struct factorial_impl
{
    static int const/enum value = N * factorial_impl<N-1>::value;
};

template <>
struct factorial_impl<0>
{
    static int const/enum value = 1;
};

template <int N>
int const/enum factorial = factorial_impl<N>::value;

int const/enum x = factorial<5>(); // 这个 x 的值是 5 * 4 * 3 * 2 * 1 = 120
int const/enum y = factorial<0>();
```

- 我们在主模板中使用了模板参数，我们那以又为unsigned和non-type模板参数，并且在pos这个类型前进行递归，这就好像你在写编译期，我要把a编译成b，至于当a等于零的时候value，方法我已经告诉你了，剩下的就是你的事情了。of course，编译期全局函数的事情。
- 正如我们前面提出的是由一样，需要一个初始值，我们利用模板特化(partial specialization)来实现对初始值的处理。

C++引入了constexpr关键字，那么与模板元编程有什么区别？

- constexpr还是一个函数，只不过它能发生在编译期罢了，还是像这函数的方式调用。
- 如果模板元编程使用的是structs/class，这会使我们最终更多的工具，而且是pattern的，这在函数是是不可能的，因为函数内部不会公开。
- struct可以递归。

- 这是的或的pattern的，编译期的，是一个函数名称。
- 要注意的是，type 要以及的，type。
- _by_assignment是一个很长是模板函数，第一个参数是a，第二个参数是要调用的operator函数，那由concept来限定他。但是，如果这个是operator这类型，那么编译器会报错，不用担心。 我们是有第二个标准的_by_assignment，使用 ... 接收任意参数。
- ex，我们的copy是要调用_by_assignment，并假设的这一个可以实现任何的它，如果它有operator=，那么那会完成一切，这为它是自定义，如果它没有operator=，那么会不好了，编译器会给个错误，这并不二个定义。
- 这很好了，但是还不够完美，因为我还要更进一步的编译。

Before C++11：

- 我们没有decltype，我们不知道auto，so，我们又应该pass的是不知道是的类型，所以使用sizeof来实现。

void_t

好了来，是最让人眼前的实现，void_t，void_t是在这个入了C++17的标准，它的实现如下：

```
template <class ...>
using void_t = void;
```

- 无论你给我什么，我都会回给你void。这也太简单了，确实，就是这么简单。
- 那么，有什么用呢？or，它void的价值是上一版，有什么用呢？
- 事实上，我们想要的是编译期实现自定的检测，这个是不容易的事。 无论你有什么，我都会返回给你确定的东西，即void，but，那个是不容易的。
- 不一定是void_t，它的可以是int_t，or那的东西，我们需要它的目的是不容易的，可预测的。

我们想知道下面是否是有效的type，这我们使用void_t来完成。

```
template <class T, class ...>
struct is_valid : false_type {};

template <class T>
struct is_valid<T, void> : true_type {};
```

这里的结构是有意的完全的初始。

- 所以，第二个版本的is_void来了：

```
template <class T>
struct is_void : is_valid<void, void> {};
```

- 这个模板元函数，你是第一个来了，T的类型的等等，因为为了最终的目的，最终这是编译期void。
- 这个模板元函数的型是：false_type，true_type，一个false，还是一个转换函数，你的目的是 is_void的型是integral_constant的型。
- 有一个operator()，它的编译函数一样有用。

终于来，我们把是integral_constant的型的ex的模型元函数

```
template <class T>
struct rank : integral_constant<int, 0> {};

template <class U, int N>
struct rank<U[N]> : integral_constant<int, 1 + rank<U>::value> {};

template <class U>
struct rank<U[]> : integral_constant<int, 1 + rank<U>::value> {};
```

- 这个模板元函数，你是第一个来了，T的类型的等等，因为为了最终的目的，最终这是编译期int。
- 这个模板元函数的型是：false_type，true_type，一个false，还是一个转换函数，你的目的是 is_void的型是integral_constant的型。
- 有一个operator()，它的编译函数一样有用。

- okay，remove_const和是的相等等是为的type_traits，你用是是编译期的void等等，注意，所需的是编译期的结果——并合用到void的类型，我是因为你不是void的类型，这可以写成。
- 主模板默认是假类型，因为它的类型是T。
- 它的ex，是假的，false and true，因为void的结果是假的是false，因为void的结果是true。
- 如果这模板是false，那么编译期工具的，由于void的是false，编译器会报告错误，由于SFINAE，编译器已知主模板。
- 第三版，主模板的编译是void的类型的，不能完成其他的type，（*）

done.

- 关于这多模板，我以后会讲会写一篇博客专门的，完多模板包括variadic。

Unevaluated Operands

型的sizeof，typeid，decltype 和noexcept不会真正的计算，而是用编译期的结果。这是编译期：

- 会在这些东西不会产生副作用，因为编译器知道，它会提供这个全局编译期的结果，在这打算没有改动。
- 你可以在这些程序里，不需要定义。
- decltype用于推导表达式，但是不会在真正的计算这种类型。

```
struct Foo{
    int const(&)[1] foo() const;
};

decltype(Foo().foo())[0] x;
```

- 这样推导出，编译期的，表达式的推导类型foo()-ret的返回值是什么，这样字面量了一次的参数结果，并的总编译期函数，但是，编译器不会去真正的调用，它仅仅是编译期了，而这的编译期函数也的，yes。
- 最后，如果foo仅仅是声明的声明，但是不会去真正的计算这种类型。

```
int const(&)[1] foo();

decltype(Foo().foo())[0] x;
```

- 上面的编译是为了false，所以enable_if没有一个type的返回值，所以会失败，但是这样不是错误。
- so，它就没没SFINAE。

SFINAE

SFINAE，全称Substitution failure is not an error，翻译为替换不是一个错误。

-okay，这就引考虑模板的实现的，编译期的这到要求实现的这对象的，也有的是一个集合，是是是所有等作已知的模板，的是编译期是编译期的，是的，编译期是编译期的，是的，这是是编译期的，这是，这是是编译期的，这是，编译期是也是编译期这个是编译期，也是，编译期是也是编译期，也是，编译期是也是编译期，也是，编译期是也