Templates – Part 2/2

Proto Team 技术培训系列

上海合见工业软件集团有限公司
Shanghai UniVista Industrial Software Group Co.,Ltd.

Meta Function

模板元编程 – Template Meta Programming

- 为什么要学习TMP
  - STL的新版本越来越多使用 TMP
  - 许多第三方库(不仅仅是Boost)也在用
  - 越来越规范，不再那么黑魔法
  - 普通 C++ 程序员应该理解其基础部分
  - 库开发者需要熟练掌握其用法
- TMP为什么看上去很难
  - 逻辑通常并不复杂
  - 语法很陌生，实际上并非真正的C++语法
  - 程序员擅长处理变量，不擅长处理类型，不了解工具箱中有什么工具
    - 比如:如何判断一个 给定类型是否有拷贝构造函数？

元函数 – Metafunctions

- 元函数 (Metafunction) 不是函数，是class/struct
  - 但是它的行为像函数
  - 它的参数是类型/常量，通过模板的类型参数和变量参数传入
  - 返回值是一种约定，在获取返回值的时候，元函数才被“调用”
- 本身并非C++语言的一部分

- C++社区创造了一种“标准”的方式

Value & Type metafunctions

- Value Metafunction
  - “返回值”是一个可以在 编译期确定的常量
  - 约定使用 value

- Type Metafunction
  - “返回值”是一个 类型
  - 约定使用type

Metafunction的返回值

- 定义一个public的value值

```
template <typename T>
struct get_value {
    static constexpr int value = 42;
};
```

- 定义一个public的type类型

```
template <typename T>
struct get_type {
    using type = T;
};
```

https://godbolt.org/z/35E7ofcWh

约定的快捷方式

- Value metafunctions ending with “_v”
- Type metafunctions ending with “_t”

```
template <typename T>
constexpr int get_value_v = get_value<T>::value;

template <typename T>
using get_type_t = typename get_type<T>::type;
```

Hello, TMP

- is_void
  - 输入：给定Type
  - 输出：Bool 值，
    - True 表示是void，False表示不是void

- 元函数实际上是模板类
- 参数是模板的类型
- 返回值是value成员
- 用特例化实现条件判断

```
template <typename T>
struct is_void {
  static constexpr bool value = false;
};

template <>
struct is_void<void> {
  static constexpr bool value = true;
};

static_assert(is_void<void>::value == true);
static_assert(is_void<int>::value == false);
```

std::integral_constant

- 一个非常基础的metafunction

```
template <typename T, T v>
struct integral_constant {
    static constexpr T value = v;
    using value_type = T;
    using type = integral_constant;

    constexpr operator value_type() const noexcept {
        return value;
    }

    constexpr value_type operator()() const noexcept {
        return value;
    }
};
```

```
template <bool B>
using bool_constant = integral_constant<bool, B>;

using true_type = bool_constant<true>;
using false_type = bool_constant<false>;
```

https://godbolt.org/z/ajEnMTW3W

Hello, TMP (revised version)

```
2  #include <type_traits>
3  #include <iostream>
4
5  template <typename T>
6  struct is_void : std::false_type {};
7
8  template <>
9  struct is_void<void> : std::true_type {};
10
11 template<typename T>
12 constexpr bool is_void_v = is_void<T>::value;
13
14 int main()
15 {
16     static_assert(is_void_v<void> == true);
17     static_assert(is_void_v<int> == false);
18
19     return 0;
20 }
```

https://godbolt.org/z/joEennYzc

is_void – const & volatile

```
static_assert(is_void_v<void>);

static_assert(is_void_v<void const>);
static_assert(is_void_v<void volatile>);
static_assert(is_void_v<void const volatile>);
```

这些都成立么？

```
template <>
struct is_void<void>: std::true_type{};

template <>
struct is_void<void const>: std::true_type{};

template <>
struct is_void<void volatile>: std::true_type{};

template <>
struct is_void<void const volatile>: std::true_type{};
```

如果是判断整型呢？(int, unsigned int, short, unsigned short, …)

https://godbolt.org/z/vseEqffMK

Type transformation – remove const

```cpp
template <typename T>
struct remove_const {
    using type = T;
};

template <typename T>
struct remove_const<T const> {
    using type = T;
};
```

```cpp
template <typename T>
struct remove_const : std::type_identity<T> {};

template <typename T>
struct remove_const<T const> : std::type_identity<T> {};

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

std::type_identity (c++ 20)

```cpp
template<class T>
struct type_identity { using type = T; };
```

remove_volatile & remove_cv

```cpp
template <typename T>
struct remove_volatile : std::type_identity<T> {};

template <typename T>
struct remove_volatile<T volatile>: std::type_identity<T> {};

template <typename T>
using remove_volatile_t = remove_volatile<T>::type;
```

```cpp
template <typename T>
struct remove_cv: remove_const<typename remove_volatile<T>::type> {};
```

有语法错误，remove_const 的特例化找不到正确的类型

```cpp
template <typename T>
using remove_cv = remove_const<remove_volatile_t<T>>;

template <typename T>
using remove_cv_t = remove_cv<T>::type;
```

https://godbolt.org/z/9WeP8z555

is_same & is_void (ver. 3)

```
template<typename T, typename U>
struct is_same : std::false_type {};

template<typename T>
struct is_same<T, T> : std::true_type {};
```

```
template<typename T>
using is_void = is_same<remove_cv_t<T>, void>;

template<typename T>
static constexpr bool is_void_v = is_void<T>::value;
```

```
static_assert(is_void_v<void>);
static_assert(is_void_v<void const>);
static_assert(is_void_v<void volatile>);
static_assert(is_void_v<void const volatile>);
```

https://godbolt.org/z/vno1P7xer

is_float

```cpp
template<typename T>
struct is_float {
    static constexpr bool value = is_same_v<remove_cv_t<T>, float>
                                || is_same_v<remove_cv_t<T>, double>
                                || is_same_v<remove_cv_t<T>, long double>;
};

template<typename T>
static constexpr bool is_float_v = is_float<T>::value;
```

Duplication!

```cpp
static_assert(is_float_v<float>);
static_assert(is_float_v<double>);
static_assert(is_float_v<const double>);

static_assert(not is_float_v<int>);
```

is_same_raw & is_float (ver. 2)

```
template<typename U, typename V>
using is_same_raw = is_same<remove_cv_t<U>, remove_cv_t<V>>;

template<typename U, typename V>
static constexpr bool is_same_raw_v = is_same_raw<U, V>::value;
```

```
template<typename T>
using is_floating_point = std::bool_constant<
                              is_same_raw_v<float, T>
                           || is_same_raw_v<double, T>
                           || is_same_raw_v<long double, T>
                          >;
template<typename T>
static constexpr bool is_floating_point_v = is_floating_point<T>::value;
```

https://godbolt.org/z/zGn8Tx9PW

Takeaways

- Metafunction 是通过struct/class来定义的
- 对Metafunction的“调用”通过::value或::type来实现
  - 或者通过_v, _t后缀的方式来实现
- Metafunction跟普通的function一样，可以通过组合来实现复杂的功能

SFINAE

SFINAE

Substitution Failure Is Not An Error

在**函数模板**的**重载**的类型解析中，当用显式指定或推导的**类型替换**模板参数失败时，该特例化会从重载集中被舍弃，而不是导致编译错误

decltype & std::declval

- decltype
  - 用于在**编译时**推断表达式的类型
  - 任何传递给decltype的表达式都**不会**被执行

```
int add(int a, int b) {
    int sum = a + b;
    std::cout << "sum = " << sum << std::endl;
    return sum;
}
decltype(add(1, 2)) result; // int
```

- std::declval
  - 在不创建类型实例的情况下获取对任何类型对象的引用

```
decltype(add(std::declval<int>(),
             std::declval<int>()))
  result;
```

has_serialize

任务:判断给定的类型是否包含成员方法 std::string serialize()

```cpp
template<typename T, typename = void>
struct has_serialize: std::false_type{};
```

General case, return false

```cpp
template<typename T>
struct has_serialize<T,
    decltype(std::declval<T>().serialize())>: std::true_type{};
```

Specialization，优先匹配

```cpp
struct A {
    void serialize(){
        std::cout << "serialize" << std::endl;
    }
};

static_assert(has_serialize<A>::value);
static_assert(not has_serialize<int>::value);
```

但是，返回值不是string

实例化<int>模板类

1. 先从特例化模板推导
2. int没有serialize方法
3. 无法推导出模板的第二个类型
4. 失败，SFINAE忽略该替换
5. 继续尝试使用general case进行替换

std::enable_if

```
template< bool B, class T = void >
struct enable_if;
```

如果 B 是true, std::enable_if会有成员 type 类型 T; 否则，没有成员 type

```
template<bool B, typename T = void>
struct enable_if {};

template<typename T>
struct enable_if<true, T> { using type = T; };
```

```
template< bool B, typename T = void >
using enable_if_t = enable_if<B,T>::type;
```

has_serialize (revised.)

```cpp
template<typename T, typename = void>
struct has_serialize: std::false_type{};

template<typename T>
struct has_serialize<T,
    std::enable_if_t<
        std::is_same_v<
            std::string,
            decltype(std::declval<T>().serialize())
        >
    >
>: std::true_type{};


template<typename T>
constexpr bool has_serialize_v = has_serialize<T>::value;


struct A {
    std::string serialize(){
        return "serialize";
    }
};

static_assert(has_serialize_v<A>);
static_assert(not has_serialize_v<int>);
```

任务:判断给定的类型是否包含成员方法

std::string serialize()

https://godbolt.org/z/zzqjz615M

std::void_t

通常用于检测某种结果是否合法，而不在意其 结果为何

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

任务：检查给定的类型中是否定义了type，比如：

```
struct A {using type = int;};
```

```
template <typename T, typename = void>
struct has_member_type : std::false_type {};

template <typename T>
struct has_member_type<T, std::void_t<typename T::type>> : std::true_type {};

template <typename T>
static constexpr bool has_member_type_v = has_member_type<T>::value;

static_assert(has_member_type_v<A>);
```

Assert Failed!

模板中的类型有默认值时（这里是void），特例化的相应类型必须与主模板一致

has_serialize (rev. 2)

```cpp
template<typename T, typename = void>
struct has_serialize: std::false_type{};

template<typename T>
struct has_serialize<T,
          std::void_t<
              decltype(std::declval<std::string&>() = std::declval<T>().serialize())
          >
>: std::true_type{};

template<typename T>
constexpr bool has_serialize_v = has_serialize<T>::value;
```

```cpp
struct WithSerialize {
    std::string serialize(){
        return "serialize";
    }
};

struct WithWrongSerialize {
    void serialize();
};

struct WithoutSerialize {
    void foo();
};
```

```cpp
static_assert(has_serialize_v<WithSerialize>);
static_assert(not has_serialize_v<int>);
static_assert(not has_serialize_v<WithWrongSerialize>);
static_assert(not has_serialize_v<WithoutSerialize>);
```

https://godbolt.org/z/Ko1vf8418

快速总结

1. 通用的带有默认类型(惯例为void)的meta function，该默认类型只是用于占位，并不真正用于实际传递参数
2. 实际判断逻辑的特例化的meta function，在判断有效时对默认类型占位部分返回void（通常通过void_t），无效时会替换失败，SFINAE会让编译器选择1所定义的通用meta function
3. void_t可以接受任意数量的参数，所以可以进行多个条件的组合判断，是非常强大的meta function工具

Type Traits

Type categories

is_void (C++11)
is_null_pointer (C++14)
is_array (C++11)
is_pointer (C++11)
is_enum (C++11)
is_union (C++11)
is_class (C++11)
is_function (C++11)
is_reference (C++11)
is_lvalue_reference (C++11)
is_rvalue_reference (C++11)
is_member_pointer (C++11)
is_member_object_pointer (C++11)
is_member_function_pointer (C++11)
is_object (C++11)
is_scalar (C++11)
is_compound (C++11)
is_integral (C++11)
is_floating_point (C++11)
is_fundamental (C++11)
is_arithmetic (C++11)

Type properties

is_const (C++11)
is_volatile (C++11)
is_empty (C++11)
is_polymorphic (C++11)
is_final (C++14)
is_abstract (C++11)
is_aggregate (C++17)
is_implicit_lifetime (C++23)
is_trivial (C++11)
is_trivially_copyable (C++11)
is_standard_layout (C++11)
is_literal_type (C++11)(until C++20*)
is_pod (C++11)(deprecated in C++20)
is_signed (C++11)
is_unsigned (C++11)
is_bounded_array (C++20)
is_unbounded_array (C++20)
is_scoped_enum (C++23)
has_unique_object_representations (C++17)

Type trait constants

integral_constant (C++11)
bool_constant (C++17)
true_type (C++11)
false_type (C++11)

Metafunctions

conjunction (C++17)
disjunction (C++17)
negation (C++17)

Supported operations

is_constructible (C++11)
is_trivially_constructible (C++11)
is_nothrow_constructible (C++11)
is_default_constructible (C++11)
is_trivially_default_constructible (C++11)
is_nothrow_default_constructible (C++11)
is_copy_constructible (C++11)
is_trivially_copy_constructible (C++11)
is_nothrow_copy_constructible (C++11)
is_move_constructible (C++11)
is_trivially_move_constructible (C++11)
is_nothrow_move_constructible (C++11)
is_assignable (C++11)
is_trivially_assignable (C++11)
is_nothrow_assignable (C++11)
is_copy_assignable (C++11)
is_trivially_copy_assignable (C++11)
is_nothrow_copy_assignable (C++11)
is_move_assignable (C++11)
is_trivially_move_assignable (C++11)
is_nothrow_move_assignable (C++11)
is_destructible (C++11)
is_trivially_destructible (C++11)
is_nothrow_destructible (C++11)
has_virtual_destructor (C++11)
is_swappable_with (C++17)
is_swappable (C++17)
is_nothrow_swappable_with (C++17)
is_nothrow_swappable (C++17)

Relationships and property queries

is_same (C++11)
is_base_of (C++11)
is_convertible (C++11)
is_nothrow_convertible (C++20)
is_layout_compatible (C++20)
is_pointer_interconvertible_base_of (C++20)
is_pointer_interconvertible_with_class (C++20)
is_corresponding_member (C++20)
alignment_of (C++11)
rank (C++11)
extent (C++11)
is_invocable (C++17)
is_invocable_r (C++17)
is_nothrow_invocable (C++17)
is_nothrow_invocable_r (C++17)
reference_constructs_from_temporary (C++23)
reference_converts_from_temporary (C++23)

Type modifications

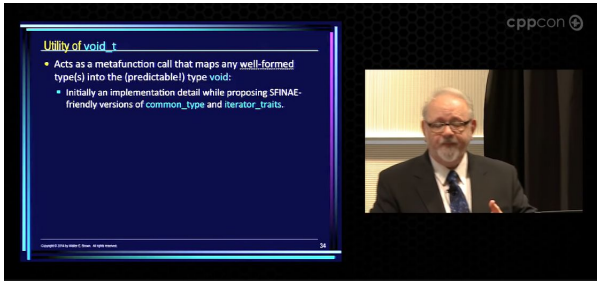
remove_cv (C++11)
remove_const (C++11)
remove_volatile (C++11)
add_cv (C++11)
add_const (C++11)
add_volatile (C++11)
make_signed (C++11)
make_unsigned (C++11)
remove_reference (C++11)
add_lvalue_reference (C++11)
add_rvalue_reference (C++11)
remove_pointer (C++11)
add_pointer (C++11)
remove_extent (C++11)
remove_all_extents (C++11)

Type transformations

aligned_storage (C++11)(deprecated in C++23)
aligned_union (C++11)(deprecated in C++23)
decay (C++11)
remove_cvref (C++20)
enable_if (C++11)
void_t (C++17)
conditional (C++11)
common_type (C++11)
common_reference (C++20)
underlying_type (C++11)
result_of (C++11)(until C++20*)
invoke_result (C++17)
type_identity (C++20)

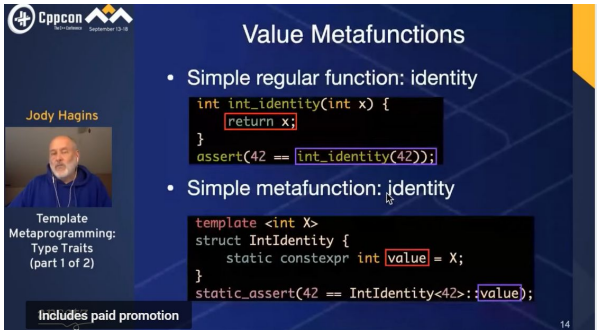
References

[CppCon 2014: Walter E. Brown "Modern Template Metaprogramming: A Compendium, Part II"]



[Template Metaprogramming: Type Traits (part 1 of 2) - Jody Hagins - CppCon 2020]

[Notes on C++ SFINAE, Modern C++ and C++20 Concepts - C++ Stories]



[Jean Guegant's Blog – An introduction to C++'s SFINAE concept: compile-time introspection of a class member]

THANKS

上海合见工业软件集团有限公司

Shanghai UniVista Industrial Software Group Co.,Ltd.