C++ template Metaprograming

templates 用做模板的模板 · 4 订阅 · 8 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
使用类型作为“返回值”
C++库模板元函数的型!
编译期的分支
SFINAE
C++库模板元函数的型2 && 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Coposon、Walter E.Brown关于metaprogramming的资料，确实精妙，感觉在模板元编程入门到新手以上，可能有点不可以过高名。这篇博客主要讲解模板元函数的应用，同时也可以用于metaprogramming的入门博客，编辑大家对模板元编程感兴趣的话，请去linkedin的群组，以及讨论模板元编程，仅仅是因为入门不熟悉它。

什么是metaprogramming？

- 利用编译期系统进行编程的手段。
- 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 图灵完 的计算。

模板元编程之作用：

- 提高代码的灵活性。
- 提高运行期的效率。

模板元编程和非运行期的工作相别编译期的技术，这是需要代价的，和编译期的代码相比，编译时的运行期的效率。

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 等等一系列会发生副作用的东西。

在编写元编程中：

- 没有可变性。
- 没有可变的
- 所有东西都不可变！！！（我认为这点是最重要的，所以单列出来）

很多人认为模板元编程很难，我觉得为他们不熟悉，everything都是从不熟悉到熟悉。在接下来的文章中，没有这打算。

- no run-time
- no run-time
- no run-time(我必须让我自己牢记这点！！！）

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;  // 代码的实现为了pure-time的实现目的
                                                   // 必须是常变量，即使是常变量的 static 成员变量也
};
```

- 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，而且这个有也是编译期间有结果。

- static_assert是C++17引入的静态断言，因为INT_MIN的特殊值溢出了。

- 这里我们的函数的一个，定义了一个static的value变量作为它的结果，这实际上是我们的函数结果，也是 编译期的结果 。注意，这里，这个结果，是编译时 的结果，并且编译时 的结果 也 是编译期。

编译期函数的特点是：

```
int const/enum N = 10;
int const/enum value;
```

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

编译时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。

```
// 这是一个模板元函数
// 模板元函数的一个特点是它像函数一样工作，那么为什么不能递归自己呢？
// 所以我们可以实现阶乘函数。
template <unsigned N, unsigned D>
struct gcd
    static unsigned const/enum value = gcd<D, N% D>::value;  // 递归调用自己的gcd
};

template <unsigned N>
struct gcd<N, 0>{ // 这是递归调用的一个特点，就是一个结束条件。
    static unsigned const/enum value = N;
    static_assert(N != 0);
    static unsigned const/enum value = N;
};
```

- 我们在主模板中使用两个模板参数，我们默认又为unsigned和non-type模板参数，并且在gcd这个类里面进行调用，这就好像 你在实现函数，我要把参数给定，型子怎么样等等的value，方法我已经告诉你了，剩下的就是你的事情了。of course，编译器会 帮忙那个事情。
- 正如刚才那里提到的函数进行一样，那里一个初始值，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函数，那么为模板元编程提供了什么优势？

- constexpr还是一个函数，只不过它能发生在编译期间了，还是像这些函数的方式调用。
- 如 模板元编程提供的是structs/class，这会使我们要新多更多工具，而且是pure-函数，这在函数是是不可能的，因为函数内部不会公开。

struct可以提供：

- 类型成员(type)的value声明。
- public的成员变量声明。
- public成员函数声明。
- 你可以拥有关于模板的复杂的等有工具，而这些都是函数不具备的。

type traits

上面只是开胃小菜，接下来对模板元编程的研究，可能会让你大开眼界。

模板元函数使用类型参数

在普通函数中，无论是一种类型作为参数，没错，不是value，那么你，将type作为parameter/argument，constexpr函数无法完成这件事。

- but，模板元编程可以。
- 实际上，一直以来都有一个世界所可以实现将type作为参数，就是struct，在struct/type，计算出type所点的字符串。
- 现在我们可以自己编写类型参数。

```
template <class _Ty, _Ty value>
struct integral_constant {
    static constexpr _Ty value = _Val;

    using value_type = _Ty;
    using type       = integral_constant;

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

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

- 当我第一次看到这个东子，oh my god，还能这样！
- 实际子用来计算数值的模板。
- 我们现在是模板元编程的基础类型，那么他是为 ...

- 模板元编程是一个有用的工具，这是我们要研究的目标。
- 我们为这些模板元函数的型，定义了一个属于自己的名字 value。
- 在第一种特化，我们实现特殊的integral_constant，并把这部分。
- and，那一种特化的定义就是特殊，我们实现特殊。

一般来于integral_constant的实现等价

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

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

关于模板元函数的调用语法

```
Value returning metafunction calls have evolved:
  is_void<T>::value          // since Technical Report 1
  typename is_void<T>::type  // instantiates/uses since C++03
  is_void<T>()               // instantiates/calls/ since C++14
  is_void_v<T>               // is C++14 variable template;
                             // planned for C++17 standard library
```

- 我们两 最终 的 调用 方式 is void型。
- 在integral_constant中，加入了type的value，operator T，所以我们可以用类型来 调用，先实 现的一个对象，并把 值 的结构 bool 类型的变量。
- 我们还有operator()，直接调用bool变量。
- 最后，在C++17中，我们可以使用 _v 来 实现value的结构。

is_void

判断一种类型是否为void，注意，void const，void volatile，const volatile void 也是void的一种。

```
template class T;
struct type_is_void
    static const bool value_type [];
    typedef type_is_void type;
    operator bool() const;
    bool operator()() const;
};

template class T;
struct is_void : public type_is_void<T> {};

template class T;
struct is_void<void const> : public type_is_void<T> {};

template class T;
struct is_void<void volatile> : public type_is_void<T> {};

template class T;
struct is_void<void const volatile> : public type_is_void<T> {};
```

- 这里我们定义了模板类型基类void的模型，那么 模型base_type。
- 然后 我们 实现void模型。
- 我们注意到，如果特殊的void出现，我们就 调用 模板特化 这个 函数，比如说 特殊is_integral模型？
- 我们用int、unsigned、long、long long、和const、volatile一起合 的 有用，yes，简单的事情 往往这样。

also，我们可以使用pure_a 和特殊化 来 实现这些void：

is_same用来的原因种类型是否相同。

```
template class T, class U
struct is_same : false_type {};

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

这里的结构型是否全的相同。

- 所以，第二个版本的is_void 来了：

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

编译期的“if”

到第一下，如果你要执行某些操作，而这些操作需要实现某种特定的条件，那么如何实现？

```
if (my condition) {
    // 这是一个模板元函数，用来实现条件表达式的结果。
    // 这个函数的结果 是 一个 编译期的bool 的值 的结果。
    template <bool, class T, class>
    struct conditional : type_is_T< T>{

    template class T, class F>
    struct conditional<false, T, F> : type_is_F< F>{
```

- 假如有一个p，那下来的选择那要依靠p的结果。
- p的结果将有两种不同的选择，他们可能是：

```
class T class F
function call: F, G
base class:  A,  B
```

- 选择不同的类型，选择不同的函数调用，选择不同的基类去继承。

在的程序中假定说，在学到bool conditional

```
// 这是一个模板元函数，用来实现 条件表达式
// 这个函数的结果 是一个 编译期的bool 的值 的结果。
template <bool, class T, class>
struct conditional : type_is_T< T>{

    template class T, class F>
    struct conditional<false, T, F> : type_is_F< F>{
```

- 假如一个bool类型的conditional，给我两种类型。
- 在这个 那个 中，这个 这个 假如 的结果，yes，**一个模板可以实现多 模板的结果！！**
- and，在定义 模板的结果的时候中，你的 结果 这个 假如的结果 的时候，可以从 数量上 定义 模板的 模板参数的某不同，因为定义 模板的结果 的不 数量。
- so，我们 实现 这个 假如的 is_void

```
template class T
using type_is_void = conditional<is_void<T>, true_type, false_type>;
```

- 这个 模板元函数，假如 is_void 的结果 为 true，那么 这个 假如 的 结果 为 true_type，假如 is_void 的结果 为 false，那么 这个 假如 的 结果 为 false_type。

done.

- 关于 这 参数 ，我 以后 会 在 一 篇博客 写 了解 ，这 参 模板 包 的 analog。

Unevaluated Operands

typeid()、sizeof()、decltype 和 noexcept 不会 真正 运行 代 ，而 是 用 一 个 特 它 的 东西。 这 是 很 重 要 ：

- 会 在 这 些 东 西 不 会 产 生 副 作 用。 因 为 他 们 只 会 检 查 编 译 期 的 东 西，在 运 行 期 没 有 东 西。
- 你 可 以 在 这 些 程 序 里，不 需 要 定 义。
- decltype 用 于 推 导 类 型。 但 是 不 会 真 正 的 计 算 这 种 类 型。

```
struct Foo{
    int test();//only 声明
};

decltype(Foo().test());// 推导
```

- 这 样 做 在 编 译 期 的 时 候，你 可 以 推 导 Foo().test() 的 返 回 类 型 的 一 个 东 西，这 样 可 以 在 不 产 生 一 个 对 象 的 情 况 下，调 用 这 个 函 数 。 这 在 很 多 的 情 况 下 是 很 有 用 的 。 yes，
- 在 程 序 中，如 果 bool 变 量 是 false，那 么 程 序 会 出 现 错 误 ！！！
- 这 可 能 会 让 人 不 喜 欢 ，这 能 干 什 么？

```
template class T, class U
struct is_same : false_type {};

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

- 上 面 的 实 现 有 了false，而 false是_v这 有 一 个 type的 返 回 值，所 以 也 失 败，但 是 这 样 不 通 顺 。
- so，它 就 可 以 实 现 SFINAE。

SFINAE

SFINAE，全 称Substitution failure is not an error，意 思 失 败 并 不 是 一 个 错 误。

- okay，这 就 引 考 虑 模 板 的 实 现。 编 译 期 间 遇 到 要 实 例 化 的 模 板，会 先 构 造 一 个 集 合，里 面 是 所 有 符 合 匹 配 的 模 板，然 后 编 译 器 选 择 一 个 最 优 的，如 果 没 有 匹 配 模 板 的 参 数，会 它 报 错 处 理。 编 译 器 会 说，ok，这 是 错 误，但 是 编 译 器 会 作 出 这 个 结 果，继 续 查 找 下 一 模 板。
- 正 如 刚 才 那 里 提 到 的 函 数 运 行 一 样，那 里 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函数，那么为模板元编程提供了什么优势？

- constexpr还是一个函数，只不过它能发生在编译期间了，还是像这些函数的方式调用。
- but 模板元编程提供的是structs/class，这会使我们要新多更多工具，而且是pure-函数，这在函数是是不可能的，因为函数内部不会公开。

struct可以提供：

- 类型成员(type)的value声明。
- public的成员变量声明。
- public成员函数声明。
- 你可以拥有关于模板的复杂的等有工具，而这些都是函数不具备的。

type traits

上面只是开胃小菜，接下来对模板元编程的研究，可能会让你大开眼睛。

模板元函数使用类型参数

在普通函数中，无论是一种类型作为参数，没错，不是value，那么你，将type作为parameter/argument，constexpr函数无法完成这件事。

- but，模板元编程可以。
- 实际上，一直以来都有一个世界所可以实现将type作为参数，就是struct，在struct/type，计算出type所点的字符串。
- 现在我们可以自己编写类型参数。

```
template class _Ty, _Ty value>
struct integral_constant {
    static constexpr _Ty value = _Val;

    using value_type = _Ty;
    using type       = integral_constant;

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

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

- 当我第一次看到这个东子，oh my god，还能这样！
- 这东子用来计算数值的模板。
- 我们现在是模板元编程的基础类型，那么他是为 ...

- 模板元函数是一种有用的工具，这是我们要研究的目标。
- 我们为这些模板元函数的型，定义了一个属于自己的名字 value。
- 在第一种特化，我们实现特殊的integral_constant，并把这部分。
- and，那一种特化的定义就是特殊，我们实现特殊。

一般来于integral_constant的实现等价

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

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

关于模板元函数的调用语法

```
Value returning metafunction calls have evolved:
  is_void<T>::value          // since Technical Report 1
  typename is_void<T>::type  // instantiates/uses since C++03
  is_void<T>()               // instantiates/calls/ since C++14
  is_void_v<T>               // is C++14 variable template;
                             // planned for C++17 standard library
```

- 我们两 最终 的 调用 方式 is void型。
- 在integral_constant中，加入了type的value，operator T，所以我们可以用类型来 调用，先实 现的一个对象，并把 值 的结构 bool 类型的变量。
- 我们还有operator()，直接调用bool变量。
- 最后，在C++17中，我们可以使用 _v 来 实现value的结构。

is_void

判断一种类型是否为void，注意，void const，void volatile，const volatile void 也是void的一种。

```
template class T;
struct type_is_void
    static const bool value_type [];
    typedef type_is_void type;
    operator bool() const;
    bool operator()() const;
};

template class T;
struct is_void : public type_is_void<T> {};

template class T;
struct is_void<void const> : public type_is_void<T> {};

template class T;
struct is_void<void volatile> : public type_is_void<T> {};

template class T;
struct is_void<void const volatile> : public type_is_void<T> {};
```

- 这里我们定义了模板类型基类void的模型，那么 模型base_type。
- 然后 我们 实现void模型。
- 我们注意到，如果特殊的void出现，我们就 调用 模板特化 这个 函数，比如说 特殊is_integral模型？
- 我们用int、unsigned、long、long long、和const、volatile一起合 的 有用，yes，简单的事情 往往这样。

到目前为止，我们可以使用pure_a 和特殊化 来 实现这些void：

但是，我们可以使用pure_a 和特殊化 来 实现这些void：

这 是 很 重 要 的 ，因 为 这 些 东 西 是 编 译 期 的 东 西，在 运 行 期 没 有 东 西。

这 是 很 重 要 的 ，因 为 这 些 东 西 是 编 译 期 的 东 西，在 运 行 期 没 有 东 西。

这 是 很 重 要 的 ，因 为 这 些 东 西 是 编 译 期 的 东 西，在 运 行 期 没 有 东 西。

- 这 里 我 们 定 义 了 模 板 类 型 基 类 void 的 模 型，那 么 模 型base_type。
- 然 后 我 们 实 现void模 型。
- 我 们 注 意 到，如 果 特 殊 的void出 现，我 们 就 调 用 模 板 特 化 这 个 函 数，比 如 说 特 殊is_integral模 型？
- 我 们 用int、unsigned、long、long long、和const、volatile一 起 合 的 有 用，yes，简 单 的 事 情 往 往 这