C++ type traits分析

CPP开发者 2020-12-08 19:20

(给CPP开发者加星标，提升C/C++技能)

> 来源：xdesk
> https://blog.csdn.net/xiangbaohui/article/details/106571580

C++ type traits分析

我们在平时常常会听到有人说traits/萃取等高大上的东西，有时候可能也会对此产生很大的疑问，觉得type tratis很高大上，高深莫测；其实说到底这个东西很简单，总结为一句话就是在运行的时候识别类型（即类型萃取）。

本文我们大致看一下type traits的基本实现技术。

1. integral_constant

了解萃取机之前，我们先了解一下integral_constant, 这个在C++库中定义为一个常量的整数，定义如下：

```
template<class _Ty,
  _Ty _Val>
  struct integral_constant
  {  // convenient template for integral constant types
  static constexpr _Ty value = _Val;

  using value_type = _Ty;
  using type = integral_constant;

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

  _NODISCARD constexpr value_type operator()() const noexcept
```

```
{   // return stored value
      return (value);
      }
    };
```

这个的主要核心是定义了一个静态常量值：

```
static constexpr _Ty value = _Val;
```

为什么需要定义这样一个东西呢？我们不直接使用_Ty value = _Val定义一个全局的变量不是挺好的嘛，为啥需要搞的那么麻烦呢？

主要原因是：为了C++编译的时候能够使用模板初编译来确定其中的值。

从integral_constant引申出来了两个东西：

true_type

false_type

这两个东西分别代表TRUE 和 FALSE，如下：

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

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

2. C++库的type traits

2.1 Primary type categories

Primary type categories

is_array	Is array (class template)
is_class	Is non-union class (class template)
is_enum	Is enum (class template)
is_floating_point	Is floating point (class template)
is_function	Is function (class template)
is_integral	Is integral (class template)
is_lvalue_reference	Is lvalue reference (class template)
is_member_function_pointer	Is member function pointer (class template)
is_member_object_pointer	Is member object pointer (class template)
is_pointer	Is pointer (class template)
is_rvalue_reference	Is rvalue reference (class template)
is_union	Is union (class template)
is_void	Is void (class template)

2.2 Composite type categories

is_arithmetic	Is arithmetic type (class template)
is_compound	Is compound type (class template)
is_fundamental	Is fundamental type (class template)
is_member_pointer	Is member pointer type (class template)
is_object	Is object type (class template)
is_reference	Is reference type (class template)
is_scalar	Is scalar type (class template)

2.3 Type properties

is_abstract	Is abstract class (class template)
is_const	Is const-qualified (class template)
is_empty	Is empty class (class template)
is_literal_type	Is literal type (class template)
is_pod	Is POD type (class template)
is_polymorphic	Is polymorphic (class template)
is_signed	Is signed type (class template)
is_standard_layout	Is standard-layout type (class template)
is_trivial	Is trivial type (class template)
is_trivially_copyable	Is trivially copyable (class template)
is_unsigned	Is unsigned type (class template)
is_volatile	Is volatile-qualified (class template)

2.4 Type features

has_virtual_destructor	Has virtual destructor (class template)
is_assignable	Is assignable (class template)
is_constructible	Is constructible (class template)
is_copy_assignable	Is copy assignable (class template)
is_copy_constructible	Is copy constructible (class template)
is_destructible	Is destructible (class template)
is_default_constructible	Is default constructible (class template)
is_move_assignable	Is move assignable (class template)
is_move_constructible	Is move constructible (class template)
is_trivially_assignable	Is trivially assignable (class template)
is_trivially_constructible	Is trivially constructible (class template)
is_trivially_copy_assignable	Is trivially copy assignable (class template)
is_trivially_copy_constructible	Is trivially copy constructible (class template)
is_trivially_destructible	Is trivially destructible (class template)
is_trivially_default_constructible	Is trivially default constructible (class template)
is_trivially_move_assignable	Is trivially move assignable (class template)
is_trivially_move_constructible	Is trivially move constructible (class template)
is_nothrow_assignable	Is assignable throwing no exceptions (class template)
is_nothrow_constructible	Is constructible throwing no exceptions (class template)
is_nothrow_copy_assignable	Is copy assignable throwing no exceptions (class template)
is_nothrow_copy_constructible	Is copy constructible throwing no exceptions (class template)
is_nothrow_destructible	Is nothrow destructible (class template)
is_nothrow_default_constructible	Is default constructible throwing no exceptions (class template)
is_nothrow_move_assignable	Is move assignable throwing no exception (class template)
is_nothrow_move_constructible	Is move constructible throwing no exceptions (class template)

2.5 Type relationships

is_base_of	Is base class of (class template)
is_convertible	Is convertible (class template)
is_same	Is same type (class template)

2.6 Property queries

alignment_of	Alignment of (class template)
extent	Array dimension extent (class template)
rank	Array rank (class template)

2.7 Type transformations

Const-volatile qualifications

add_const	Add const qualification (class template)
add_cv	Add const volatile qualification (class template)
add_volatile	Add volatile qualification (class template)
remove_const	Remove const qualification (class template)
remove_cv	Remove cv qualification (class template)
remove_volatile	Remove volatile qualification (class template)

Compound type alterations

add_pointer	Add pointer (class template)
add_lvalue_reference	Add lvalue reference (class template)
add_rvalue_reference	Add rvalue reference (class template)
decay	Decay type (class template)
make_signed	Make signed (class template)
make_unsigned	Make unsigned (class template)
remove_all_extents	Remove all array extents (class template)
remove_extent	Remove array extent (class template)
remove_pointer	Remove pointer (class template)
remove_reference	Remove reference (class template)
underlying_type	Underlying type of enum (class template)

Other type generators

aligned_storage	Aligned storage (class template)
aligned_union	Aligned union (class template)
common_type	Common type (class template)
conditional	Conditional type (class template)
enable_if	Enable type if condition is met (class template)
result_of	Result of call (class template)

3. type traits的例子

```
class CData1
{
public:
  CData1() {}
  virtual ~CData1() {}
};

class CData2
{
public:
  CData2() {}
```

```
  ~CData2() {}
};

class CData3
{
public:
  int a;
  int b;
  int c;
};
int main(int args, char* argv[])
{
  std::cout << "CData1 has_virtual_destructor : " << std::has_virtual_des
  std::cout << "CData2 has_virtual_destructor : " << std::has_virtual_des
  std::cout << "CData3 has_virtual_destructor : " << std::has_virtual_des
  std::cout << "CData1 is_pod : " << std::is_pod<CData1>::value << std::e
  std::cout << "CData2 is_pod : " << std::is_pod<CData2>::value << std::e
  std::cout << "CData3 is_pod : " << std::is_pod<CData3>::value << std::e
  return 0;
}
```

输出结果如下：

```
CData1 has_virtual_destructor : 1
CData2 has_virtual_destructor : 0
CData3 has_virtual_destructor : 0
CData1 is_pod : 0
CData2 is_pod : 0
CData3 is_pod : 1
```

从上面我们可以看到type traits是非常厉害的，他能够在编译器的时候知道C++定义类型的所有属性。

4. type tratis的实现

我们看几个例子来大致看一下type traits的实现原理.

4.1 std::is_integral

std::is_integral用来判断一个类型是否是整数，这个的实现原理如下：

```
// STRUCT TEMPLATE _Is_integral
template<class _Ty>
  struct _Is_integral
    : false_type
  {  // determine whether _Ty is integral
  };

template<>
  struct _Is_integral<bool>
    : true_type
  {  // determine whether _Ty is integral
  };

template<>
  struct _Is_integral<char>
    : true_type
  {  // determine whether _Ty is integral
  };

template<>
  struct _Is_integral<unsigned char>
    : true_type
  {  // determine whether _Ty is integral
  };

template<>
  struct _Is_integral<signed char>
    : true_type
  {  // determine whether _Ty is integral
  };

 #ifdef _NATIVE_WCHAR_T_DEFINED
template<>
  struct _Is_integral<wchar_t>
```

```
    : true_type
  {  // determine whether _Ty is integral
  };
 #endif /* _NATIVE_WCHAR_T_DEFINED */

template<>
  struct _Is_integral<char16_t>
    : true_type
  {  // determine whether _Ty is integral
  };

template<>
  struct _Is_integral<char32_t>
    : true_type
  {  // determine whether _Ty is integral
  };

template<>
  struct _Is_integral<unsigned short>
    : true_type
  {  // determine whether _Ty is integral
  };

template<>
  struct _Is_integral<short>
    : true_type
  {  // determine whether _Ty is integral
  };

template<>
  struct _Is_integral<unsigned int>
    : true_type
  {  // determine whether _Ty is integral
  };

template<>
  struct _Is_integral<int>
```

```
      : true_type
    {   // determine whether _Ty is integral
    };

template<>
    struct _Is_integral<unsigned long>
      : true_type
    {   // determine whether _Ty is integral
    };

template<>
    struct _Is_integral<long>
      : true_type
    {   // determine whether _Ty is integral
    };

template<>
    struct _Is_integral<unsigned long long>
      : true_type
    {   // determine whether _Ty is integral
    };

template<>
    struct _Is_integral<long long>
      : true_type
    {   // determine whether _Ty is integral
    };

    // STRUCT TEMPLATE is_integral
template<class _Ty>
    struct is_integral
      : _Is_integral<remove_cv_t<_Ty>>::type
    {   // determine whether _Ty is integral
    };
```

首先定义了一个template<class _Ty> struct _Is_integral : false_type 通用的模板，这个模板中有一个bool value = false的静态成员。

然后就是真的所有的整数类型，创建特化模块，例如如下：

```
template<>
  struct _Is_integral<int>
    : true_type
  {  // determine whether _Ty is integral
  };
```

这个模板中有一个bool value = true的静态成员。

从这里大致我们可以看出type traits是使用特化来确定特定的情况。

4.2 std::is_pod

对于简单类型的判断比较容易，我们实现所有类型的模板特化即可，但是对于类复杂类型的判断，就比较麻烦了，C++标准库的实现如下：

```
template<class _Ty>
  struct is_pod
    : bool_constant<__is_pod(_Ty)>
  {  // determine whether _Ty is a POD type
  };

template<class _Ty>
  _INLINE_VAR constexpr bool is_pod_v = __is_pod(_Ty);

  // STRUCT TEMPLATE is_empty
template<class _Ty>
  struct is_empty
    : bool_constant<__is_empty(_Ty)>
  {  // determine whether _Ty is an empty class
  };

template<class _Ty>
  _INLINE_VAR constexpr bool is_empty_v = __is_empty(_Ty);

```

```
  // STRUCT TEMPLATE is_polymorphic
template<class _Ty>
  struct is_polymorphic
    : bool_constant<__is_polymorphic(_Ty)>
  {  // determine whether _Ty is a polymorphic type
  };
```

对于__is_podC++标准库并没有公开的代码，这里也不知道具体如何实现，跟编译器的底层实现细节有关，但是从我们所有的type traits来说，这个功能还是十分强大的。

5. iterator_traits

在萃取中，存在一个比较重要的萃取，如果上面的is_class, is_pod都没有用过的话，那么iterator_traits这个萃取机肯定是用过的，例如：

```
 template<typename _InputIterator, typename _Size, typename _ForwardItera
   inline _ForwardIterator
   uninitialized_copy_n(_InputIterator __first, _Size __n,
                        _ForwardIterator __result)
{ return std::__uninitialized_copy_n(__first, __n, __result,
                                     std::__iterator_category(__first
```

其中std::__iterator_category(__first))这个就是类型萃取机，这个实现如下：

```
template<typename _Iter>
   inline _GLIBCXX_CONSTEXPR
   typename iterator_traits<_Iter>::iterator_category
   __iterator_category(const _Iter&)
   { return typename iterator_traits<_Iter>::iterator_category(); }
```

iterator_traits 这个就是迭代器的萃取机，这个萃取机可以做如下事情：

萃取迭代器的类型。

萃取迭代器代表的值的类型。

萃取迭代器使用值的引用指针等类型。

这个迭代器实现如下：

```
template<typename _Iterator>
  struct iterator_traits
  {
    typedef typename _Iterator::iterator_category iterator_category;
    typedef typename _Iterator::value_type        value_type;
    typedef typename _Iterator::difference_type   difference_type;
    typedef typename _Iterator::pointer           pointer;
    typedef typename _Iterator::reference         reference;
  };

 /// Partial specialization for pointer types.
 template<typename _Tp>
  struct iterator_traits<_Tp*>
  {
    typedef random_access_iterator_tag iterator_category;
    typedef _Tp                         value_type;
    typedef ptrdiff_t                   difference_type;
    typedef _Tp*                        pointer;
    typedef _Tp&                        reference;
  };

 /// Partial specialization for const pointer types.
 template<typename _Tp>
  struct iterator_traits<const _Tp*>
  {
    typedef random_access_iterator_tag iterator_category;
    typedef _Tp                         value_type;
    typedef ptrdiff_t                   difference_type;
    typedef const _Tp*                  pointer;
    typedef ptrdiff_t                   difference_type;
    typedef const _Tp*                  pointer;
    typedef const _Tp&                  reference;
  };
```

对于我们STL的迭代器，都需要定义这些类型：

Iterator::iterator_category 迭代器类型。

Iterator::value_type : 迭代器的值类型。

Iterator::difference_type : 迭代器的距离信息。

Iterator::pointer ： 迭代器指针。

Iterator::reference : 迭代器的引用。

STL的迭代器其实就是模拟指针来实现的，所以指针，应该天生适合最合适的迭代器，因此给_Tp* 和 const _Tp* 定义了特殊的萃取类型。

6. 总结

从上面分析，对于C++库，萃取的实现一般都是定义模板来实现，对于普通的类型，匹配这个模板的定义；然后针对特殊类型实现特化模板支持。

- EOF -

推荐阅读 点击标题可跳转

1、C++ STL萃取机制

2、C++ 模板特化与偏特化

3、如何减少 vector 的内存占用

关于 C++ type traits分析，欢迎在评论中和我探讨。觉得文章不错，请点赞和在看支持我继续分享好文。谢谢！