C++模板之SFINAE和enable_if分析

CPP开发者 2020-12-12 04:50

> 来源：xdesk
> https://blog.csdn.net/xiangbaohui/article/details/106575509

【导读】：本文主要介绍C++如何实现类型萃取以及如何禁止函数编译期间根据特定的条件来选择启用或禁用特定的重载，感兴趣的同学一起和小编学习吧。

以下是正文

在《C++ type traits分析》 这篇文章中，我们讲述了type traits的实现用法和基本原理。

本文我们讨论一下两个问题：

如果实现is_class的type traits。

怎么样禁止函数编译期间根据特定的条件来选择启用或禁用特定的重载。

1. SFINAE

SFINAE是"Substitution failure is not an error"(替换失败不是错误)，官方给的解释为：在函数模板的重载决议中：为模板形参替换推导类型失败时，从重载集抛弃特化，而非导致编译错误。

用一句简单的话来说，C++在编译的时候，模板实例化的时候就算实例化失败也不会出现错误。

例如我们用一个简单的例子来描述：

```
template<typename T>
void test(typename T::noexist t)
{

}

```

```
void test(int t)
{

}

int main(int args, char* argv[])
{
  test(100.1);
  return 0;
}
```

我们在使用test(100.1)实例化模板test的时候，并不会编译错误，只是会实例化失败，使用void test(int t)这个函数。

所以SFINAE 的含义是：编译器在将函数模板形参替换成实参的过程中，如果针对某个函数模板产生了不合法的代码，其不会直接抛出错误信息，而是继续尝试去匹配其他可能的重载函数。

What… 这是什么鬼？不是本来C++特性就是这样嘛？有必要讲的那么高大上嘛？不急，我们使用这个特性看一下SFINAE可以做什么事情。

2. isClass的实现

```
namespace SFINAE
{
  template<typename T>
  class IsClass
  {
  private:
    template<typename U>
    static constexpr bool is_class(int U::*)
{
      return true;
    }
    template<typename U>
    static constexpr bool is_class(...)
{
      return false;
```

```
    }

  public:
    static constexpr bool value = is_class<T>(nullptr);
  };
}

class A
{

};

class B
{
  int a;
  int b;
};
int main(int args, char* argv[])
{
  std::cout << "A is class: " << SFINAE::IsClass<A>::value << std::endl;
  std::cout << "B is class: " << SFINAE::IsClass<B>::value << std::endl;
  std::cout << "int is class: " << SFINAE::IsClass<int>::value << std::en
  std::cout << "double is class: " << SFINAE::IsClass<double>::value << s
  return 0;
}
```

这里有两个技巧：

int U::* 使用这个来模板实例化，如果是类，可以定义类的成员指针，这个最佳匹配。

is_class(...) 如果不是类，那么这个函数适配所有的类型。

这个的运行结构为：

```
A is class: 1
B is class: 1
int is class: 0
```

```
4  double is class: 0
```

这个就是利用模板实例化的时候，如果第一个实例化不了，会继续实例化其他而不会出错的特性。

3. enable_if

enable_if解决的一个问题是：如何在函数编译期间根据特定的条件来选择启用或禁用模板实例化。加入我们存在一个Add加法操作，但是我们限定一个东西，就是Add只能使用整数，像std::string不能适配，我们可以使用如下：

```
1  //整数
2  template<typename T,
3    typename std::enable_if<std::is_integral<T>::value>::type* = nullptr>
4  auto Add(T a, T b) -> T
5  {
6    return a + b;
7  }
8
9  int main(int args, char* argv[])
10 {
11   std::string str = "abc";
12   Add(100, 200);
13   str + "abc";
14   Add(str, str);
15   return 0;
16 }
```

此时编译的时候就会出错：

```
1  : error C2672: 'Add': no matching overloaded function found
2  : error C2783: 'T Add(T,T)': could not deduce template argument for '__fo
3  : note: see declaration of 'Add'
```

也就是说在模板实例化的时候并没有成功。enable_if 的主要作用就是当某个 condition 成立时，enable_if可以提供某种类型。

我们看下enable_if的实现原理。

```
template<bool _Test,
  class _Ty = void>
  struct enable_if
  {  // type is undefined for assumed !_Test
  };

template<class _Ty>
  struct enable_if<true, _Ty>
  {  // type is _Ty for _Test
  using type = _Ty;
  };
```

也就是说，默认情况下enable_if是一个空的类定义，如果bool _Test 参数为TRUE的话，使用片特例模板struct enable_if<true, _Ty>，这个模板可以定义type类型。例如：

```
class A
{
public:
  int a;
};
int main(int args, char* argv[])
{
  std::enable_if<std::is_class<A>::value, A>::type a;
  a.a = 100;
  return 0;
}
```

C++库还定义了如下类型：

```
template<bool _Test,
  class _Ty = void>
  using enable_if_t = typename enable_if<_Test, _Ty>::type;
```

类似的东西在C++标准库中使用很多，例如std::vector，如下：

```
template<class _Iter,
    class = enable_if_t<_Is_iterator_v<_Iter>>>
```

```
    vector(_Iter _First, _Iter _Last, const _Alloc& _Al = _Alloc())
    : _Mybase(_Al)
    {  // construct from [_First, _Last) with optional allocator
    _Adl_verify_range(_First, _Last);
    _Range_construct_or_tidy(_Get_unwrapped(_First), _Get_unwrapped(_Last
    }
```

- EOF -

推荐阅读 — 点击标题可跳转

1、据说程序员等电梯的时候都想过调度算法，网友：还真是

2、C++ 智能指针用法详解

3、C++ type traits分析

关于 C++模板之SFINAE和enable_if，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！