14讲SFINAE：不是错误的替换失败是怎么回事

你好，我是吴咏炜。

我们已经连续讲了两讲模板和编译期编程了。今天我们还是继续这个话题，讲的内容是模板里的一个特殊概念——替换失败非错（substituion failure is not an error），英文简称为 SFINAE。

函数模板的重载决议

我们之前已经讨论了不少模板特化。我们今天来着重看一个函数模板的情况。当一个函数名称和某个函数模板名称匹配时，重载决议过程大致如下：

- 根据名称找出所有适用的函数和函数模板
- 对于适用的函数模板，要根据实际情况对模板形参进行替换；替换过程中如果发生错误，这个模板会被丢弃
- 在上面两步生成的可行函数集合中，编译器会寻找一个最佳匹配，产生对该函数的调用
- 如果没有找到最佳匹配，或者找到多个匹配程度相当的函数，则编译器需要报错

我们还是来看一个具体的例子（改编自参考资料 [1]）。虽然这例子不那么实用，但还是比较简单，能够初步说明一下。

```
#include <stdio.h>

struct Test {
  typedef int foo;
};

template <typename T>
void f(typename T::foo)
{
  puts("1");
}

template <typename T>
void f(T)
{
  puts("2");
}

int main()
{
  f<Test>(10);
  f<int>(10);
}
```

输出为：

```
1
2
```

我们来分析一下。首先看 `f<Test>(10);` 的情况：

- 我们有两个模板符合名字 `f`
- 替换结果为 `f(Test::foo)` 和 `f(Test)`
- 使用参数 `10` 去匹配，只有前者参数可以匹配，因而第一个模板被选择

再看一下 `f<int>(10)` 的情况：

- 还是两个模板符合名字 `f`
- 替换结果为 `f(int::foo)` 和 `f(int)`；显然前者不是个合法的类型，被抛弃
- 使用参数 `10` 去匹配 `f(int)`，没有问题，那就使用这个模板实例了

在这儿，体现的是 SFINAE 设计的最初用法：如果模板实例化中发生了失败，没有理由编译就此出错终止，因为还是可能有其他可用的函数重载的。

这儿的失败仅指函数模板的原型声明，即参数和返回值。函数体内的失败不考虑在内。如果重载决议选择了某个函数模板，而函数体在实例化的过程中出错，那我们仍然会得到一个编译错误。

编译期成员检测

不过，很快人们就发现 SFINAE 可以用于其他用途。比如，根据某个实例化的成功或失败来在编译期检测类的特性。下面这个模板，就可以检测一个类是否有一个名叫 `reserve`、参数类型为 `size_t` 的成员函数：

```
template <typename T>
struct has_reserve {
  struct good { char dummy; };
  struct bad { char dummy[2]; };
  template <class U,
            void (U::*)(size_t)>
  struct SFINAE {};
  template <class U>
  static good
  reserve(SFINAE<U, &U::reserve>*);
  template <class U>
  static bad reserve(...);
  static const bool value =
    sizeof(reserve<T>(nullptr))
    == sizeof(good);
};
```

在这个模板里：

- 我们首先定义了两个结构 `good` 和 `bad`；它们的内容不重要，我们只关心它们的大小必须不一样。
- 然后我们定义了一个 `SFINAE` 模板，内容也同样不重要，但模板的第二个参数需要是第一个参数的成员函数指针，并且参数类型是 `size_t`，返回值是 `void`。
- 随后，我们定义了一个要求 `SFINAE*` 类型的 `reserve` 成员函数模板，返回值是 `good`；再定义了一个对参数类型无要求的 `reserve` 成员函数模板（不熟悉 `...` 语法的，可以看参考资料 [2]），返回值是 `bad`。
- 最后，我们定义常整型布尔值 `value`，结果是 `true` 还是 `false`，取决于 `nullptr` 能不能和 `SFINAE*` 匹配成功，而这又取决于模板参数 `T` 有没有返回类型是 `void`、接受一个参数并且类型为 `size_t` 的成员函数 `reserve`。

那这样的模板有什么用处呢？我们继续往下看。

SFINAE 模板技巧

enable_if

C++11 开始，标准库里有了一个叫 `enable_if` 的模板（定义在 <type_traits> 里），可以用它来选择性地启用某个函数的重载。

假设我们有一个函数，用来往一个容器尾部追加元素。我们希望原型是这个样子的：

```
template <typename C, typename T>
void append(C& container, T* ptr,
            size_t size);
```

显然，container 有没有 reserve 成员函数，是对性能有影响的——如果有的话，我们通常应该预留好内存空间，以免产生不必要的对象移动甚至拷贝操作。利用 enable_if 和上面的 has_reserve 模板，我们就可以这么写：

```
template <typename C, typename T>
enable_if_t<has_reserve<C>::value,
            void>
append(C& container, T* ptr,
       size_t size)
{
  container.reserve(
    container.size() + size);
  for (size_t i = 0; i < size;
       ++i) {
    container.push_back(ptr[i]);
  }
}

template <typename C, typename T>
enable_if_t<!has_reserve<C>::value,
            void>
append(C& container, T* ptr,
       size_t size)
{
  for (size_t i = 0; i < size;
       ++i) {
    container.push_back(ptr[i]);
  }
}
```

要记得之前我说过，对于某个 type trait，添加 _t 的后缀等价于其 type 成员类型。因而，我们可以用 enable_if_t 来取到结果的类型。enable_if_t<has_reserve<C>::value, void> 的意思可以理解成：如果类型 C 有 reserve 成员的话，那我们启用下面的成员函数，它的返回类型为 void。

enable_if 的定义（其实非常简单）和它的进一步说明，请查看参考资料 [3]。参考资料里同时展示了一个通用技巧，可以用在构造函数（无返回值）或不想手写返回值类型的情况下。但那个写法更绕一些，不是必需要用的话，就采用上面那个写出返回值类型的写法吧。

decltype 返回值

如果只需要在某个操作有效的情况下启用某个函数，而不需要考虑相反的情况的话，有另外一个技巧可以用。对于上面的 append 的情况，如果我们想限制只有具有 reserve 成员函数的类可以使用这个重载，我们可以把代码简化成：

```
template <typename C, typename T>
auto append(C& container, T* ptr,
            size_t size)
  -> decltype(
    declval<C&>().reserve(1U),
    void())
{
  container.reserve(
    container.size() + size);
  for (size_t i = 0; i < size;
       ++i) {
    container.push_back(ptr[i]);
  }
}
```

这是我们第一次用到 `declval` [4]，需要简单介绍一下。这个模板用来声明一个某个类型的参数，但这个参数只是用来参加模板的匹配，不允许实际使用。使用这个模板，我们可以在某类型没有默认构造函数的情况下，假想出一个该类的对象来进行类型推导。`declval<C&>().reserve(1U)` 用来测试 `C&` 类型的对象是不是可以拿 `1U` 作为参数来调用 `reserve` 成员函数。此外，我们需要记得，C++ 里的逗号表达式的意思是按顺序逐个估值，并返回最后一项。所以，上面这个函数的返回值类型是 `void`。

这个方式和 `enable_if` 不同，很难表示否定的条件。如果要提供一个专门给**没有** `reserve` 成员函数的 `C` 类型的 `append` 重载，这种方式就不太方便了。因而，这种方式的主要用途是避免错误的重载。

void_t

`void_t` 是 C++17 新引入的一个模板 [5]。它的定义简单得令人吃惊：

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

换句话说，这个类型模板会把任意类型映射到 `void`。它的特殊性在于，在这个看似无聊的过程中，编译器会检查那个“任意类型”的有效性。利用 `decltype`、`declval` 和模板特化，我们可以把 `has_reserve` 的定义大大简化：

```
template <typename T,
          typename = void_t<>>
struct has_reserve : false_type {};

template <typename T>
struct has_reserve<
  T, void_t<decltype(
       declval<T&>().reserve(1U))>>
  : true_type {};
```

这里第二个 has_reserve 模板的定义实际上是一个偏特化 [6]。偏特化是类模板的特有功能，跟函数重载有些相似。编译器会找出所有的可用模板，然后选择其中最“特别”的一个。像上面的例子，所有类型都能满足第一个模板，但不是所有的类型都能满足第二个模板，所以第二个更特别。当第二个模板能被满足时，编译器就会选择第二个特化的模板；而只有第二个模板不能被满足时，才会回到第一个模板的通用情况。

有了这个 has_reserve 模板，我们就可以继续使用其他的技巧，如 enable_if 和下面的标签分发，来对重载进行限制。

标签分发

在上一讲，我们提到了用 true_type 和 false_type 来选择合适的重载。这种技巧有个专门的名字，叫标签分发（tag dispatch）。我们的 append 也可以用标签分发来实现：

```
template <typename C, typename T>
void _append(C& container, T* ptr,
             size_t size,
             true_type)
{
  container.reserve(
    container.size() + size);
  for (size_t i = 0; i < size;
       ++i) {
    container.push_back(ptr[i]);
  }
}

template <typename C, typename T>
void _append(C& container, T* ptr,
             size_t size,
             false_type)
{
  for (size_t i = 0; i < size;
       ++i) {
    container.push_back(ptr[i]);
  }
}

template <typename C, typename T>
void append(C& container, T* ptr,
            size_t size)
{
  _append(
    container, ptr, size,
    integral_constant<
      bool,
      has_reserve<C>::value>{});
}
```

回想起上一讲里 `true_type` 和 `false_type` 的定义，你应该很容易看出这个代码跟使用 `enable_if` 是等价的。当然，在这个例子，标签分发并没有使用 `enable_if` 显得方便。作为一种可以替代 `enable_if` 的通用惯用法，你还是需要了解一下。

另外，如果我们用 `void_t` 那个版本的 `has_reserve` 模板的话，由于模板的实例会继承 `false_type` 或 `true_type` 之一，代码可以进一步简化为：

```
template <typename C, typename T>
void append(C& container, T* ptr,
            size_t size)
{
  _append(
    container, ptr, size,
    has_reserve<C>{});
}
```

静态多态的限制?

看到这儿，你可能会怀疑，为什么我们不能像在 Python 之类的语言里一样，直接写下面这样的代码呢?

```
template <typename C, typename T>
void append(C& container, T* ptr,
            size_t size)
{
  if (has_reserve<C>::value) {
    container.reserve(
      container.size() + size);
  }
  for (size_t i = 0; i < size;
       ++i) {
    container.push_back(ptr[i]);
  }
}
```

如果你试验一下，就会发现，在 C 类型没有 `reserve` 成员函数的情况下，编译是不能通过的，会报错。这是因为 C++ 是静态类型的语言，所有的函数、名字必须在编译时被成功解析、确定。在动态类型的语言里，只要语法没问题，缺成员函数要执行到那一行上才会被发现。这赋予了动态类型语言相当大的灵活性；只不过，不能在编译时检查错误，同样也是很多人对动态类型语言的抱怨所在……

那在 C++ 里，我们有没有更好的办法呢？实际上是有的。具体方法，下回分解。

内容小结

今天我们介绍了 SFINAE 和它的一些主要惯用法。虽然随着 C++ 的演化，SFINAE 的重要性有降低的趋势，但我们仍需掌握其基本概念，才能理解使用了这一技巧的模板代码。

课后思考

这一讲的内容应该仍然是很烧脑的。请你务必试验一下文中的代码，加深对这些概念的理解。同样，有任何问题和想法，可以留言与我交流。

参考资料

[1] Wikipedia, “Substitution failure is not an error”. https://en.wikipedia.org/wiki/Substitution_failure_is_not_an_error

[2] cppreference.com, “Variadic functions”. https://en.cppreference.com/w/c/variadic

[2a] cppreference.com, “变参数函数”. https://zh.cppreference.com/w/c/variadic

[3] cppreference.com, “std::enable_if”. https://en.cppreference.com/w/cpp/types/enable_if

[3a] cppreference.com, “std::enable_if”. https://zh.cppreference.com/w/cpp/types/enable_if

[4] cppreference.com, “std::declval”. https://en.cppreference.com/w/cpp/utility/declval

[4a] cppreference.com, “std::declval”. https://zh.cppreference.com/w/cpp/utility/declval

[5] cppreference.com, “std::void_t”. https://en.cppreference.com/w/cpp/types/void_t

[5a] cppreference.com, “std::void_t”. https://zh.cppreference.com/w/cpp/types/void_t

[6] cppreference.com, “Partial template specialization”. https://en.cppreference.com/w/cpp/language/partial_specialization

[6a] cppreference.com, “部分模板特化”. https://zh.cppreference.com/w/cpp/language/partial_specialization

精选留言

三味

emmmm....

这一节内容如果是半年前看到，应该能节省我好多时间去写序列化，真是我实实在在的需求啊！

我自己在写数据序列化为json文本的时候，就遇到了这样头疼的问题：如何根据类型，去调用对应的函数。

如果是简单的int，bool，float，直接特化就好了。

如果是自定义的结构体呢？我的做法就是判断自定义结构体中是否有serializable和deserializable函数，就用到了文中最开始的方法判断。

然而那会儿我写得还是太简单粗暴，在代码中用的是if去判断，对于不支持的类型，直接报错，并不能做到忽略。

看了本文之后，真是受益颇多啊！留言于此，告诉大家，别以为用不到这些内容，都是实实在在的干货！

2019-12-30 16:06

作者回复

我喜欢这样的留言。哈哈，写专栏就是希望能给大家帮助的。

2019-12-31 21:58

禾桃

请问有编译器本身什么工具或者日志模式，可以显示模版实例化的过程?

2019-12-27 08:29

作者回复

新发现一个工具，可以展示实例化的过程。你可以去看一下：

https://cppinsights.io/

2019-12-27 08:51

禾桃

https://en.cppreference.com/w/cpp/types/enable_if

"

Possible implementation

template<bool B, class T = void>

struct enable_if {};

template<class T>

struct enable_if<true, T> { typedef T type; };

"

下列的测试代码可以编译，运行
"
#include <type_traits>
#include <iostream>

template <typename X, typename T, typename Y>
typename std::enable_if<std::is_same<T, int>::value>::type update(X x, T t, Y y)
{
std::cout << "int" << std::endl;
}

int main () {
float a = 1.4;
int b = 1;
double c =2.2;
update(a, b, c);
}
"

其中我用的是std::enable_if<std::is_same<T, int>::value>， 并没有提供参数来推导T，
这是不是意味着对于true这种情况的偏特化implementation是
template<class T = void>
struct enable_if<true, T> { typedef T type; };

而不是

template<class T>
struct enable_if<true, T> { typedef T type; };

谢谢！
2020-02-16 22:44
作者回复
我觉得应该这么理解：

你给出的表达式是

typename std::enable_if<std::is_same<T, int>::value>::type

估值后得到

typename std::enable_if<true>::type

实例化时，编译器看到没有提供第二个参数，加上缺省参数按 std::enable_if<true, void> 实例化，再偏特化得到

template<>
struct enable_if<true, void>
{
typedef void type;

};

2020-02-17 13:47

贾陆华

吴老师，写的满满干货，之前模板并没有深入学习，才知道模板可以这么花，用途可以这么多

2020-02-14 21:47

作者回复

模板绝对是 C++ 的一大特点。虽说有些技巧会过时（SFINAE 到 C++20 也会变得有点过时；看完全部你就知道了），但模板和静态鸭子类型这样的用法在目前的其他主流语言里是没有的。

2020-02-15 10:18

曹哲铭

表示真的太烧脑了

2020-01-17 07:53

作者回复

这个网站也许可以帮你：

https://cppinsights.io/

2020-01-17 13:20

光城~兴

假设U是一个Class,内部有foo方法与bar方法，那么U::*表示的是所有成员函数的指针吧，而U::foo*是foo成员函数指针，还请老师指点。

2020-01-07 10:48

作者回复

你的问题不那么清楚啊。

U::* 不是完整类型，void (U::*)() 才是一个类型。U::foo 是一个具体的函数，&U::foo 是函数指针。

2020-01-07 18:07

木瓜777

没看这几篇文章前，以为理解模板了，现在才知道模板博大精深

2020-01-02 07:48

作者回复

写这一讲时，我自己也觉得很舒心——正好把相关的技巧整理一遍。这部分还是挺复杂的。

2020-01-02 18:57

李亮亮

template <typename T,
typename = void_t<>>
struct has_reserve : false_type {};
这里的冒号是什么语法?

2019-12-30 10:23

作者回复

继承啊。

2019-12-31 19:21

总统老唐

吴老师，关于这一课，有 3 个问题
1，在最开始定义 has_reserve 类时，两个 reserve 模板函数实际上只是声明了，但是并没有真正的函数体，而最后的 value 成员实际上是用 nullptr 调用了 reserve 函数，这就相当于调用一个没有只有声明没有定义的函数，为什么没有报错?
2，关于模板函数的调用
假设有如下模板
template <typename T1, typename T2>
int add(T1 a, T2 b);
既可以add<int, double>(1, 2.5)调用，也可以add(1, 2.5)调用，两者的差别是不是第一种方式相当于先声明了一个特化版本，在用这个特化版本来调用，后一种方式是编译器自行推断？但若是没有定义对应的特化版本，第一种方式和第二种方式是不是

完全没有区别?

3，在 void_t 的部分，模板定义时，第二个参数是这样写的：typename = void_t<>， 我试了一下，直接写成 typename = void ，也是可以的，你采用这种写法是有什么特殊考虑吗?

2019-12-29 19:29

作者回复

1. 一个函数没有真正被调用，代码里就不会产生对它的引用，链接没有也就不会出问题。

2. 不是特化，而是自动推断后进行自动实例化。特化是需要有能看得到的特化定义的。

3. 主要是和下面的定义对称。因为这儿的类型不实际使用，写任何的合法类型都是可以的。

2019-12-30 10:55

禾桃

“

template <typename T, typename = void_t<>>

struct has_reserve : false_type {};

template <typename T>

struct has_reserve<T, void_t<decltype(declval<T&>().reserve(1U))>> : true_type {};

declval().reserve(1U) 用来测试 C& 类型的对象是不是可以拿 1U 作为参数来调用 reserve 成员函数

“

请问

- 如果是, decltype(declval<T&>().reserve(1U))> 返回的是void，这个好理解，因为void_t会把任何数目（包括零个）的类型转换为类型void

- 如果不是, 编译器看到decltype(declval<T&>().reserve(1U))> 会做什么?

然后编译器看到void_t<decltype(declval<T&>().reserve(1U))> 又会做什么?

2019-12-27 15:03

作者回复

不是说了吗，把任意类型映射到void。任意类型哦……只要表达式合法就行。

2019-12-27 21:47