14 | SFINAE：不是错误的替换失败是怎么回事？

2019-12-27 吴咏炜

讲述：吴咏炜
时长 10:30　大小 9.63M

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。

复制代码

```
#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：

复制代码

```
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：

复制代码

```
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：

复制代码

```
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：

复制代码

```
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：

复制代码

```
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：

复制代码

```
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：

复制代码

```
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：

复制代码

```
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

上一篇　13 | 编译期能做些什么？一个完整的计算世界

下一篇　15 | constexpr：一个常态的世界

精选留言 (5)

写留言

三味
2019-12-30

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。...

展开

李亮亮
2019-12-30

template <typename T,

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

展开

1

总统老唐
2019-12-29

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...

展开

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。

禾桃
2019-12-27

“

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

template <typename T>...

展开

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。

2

禾桃
2019-12-27

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？

> 作者回复: 这倒不知道有。如果失败了，输出错误信息里可以找到提示的。成功了的话，只能靠往代码里插调试语句了，可以是运行时日志或static_assert。