前序文章请看：
C++模板元编程详细教程（之一）
C++模板元编程详细教程（之二）
C++模板元编程详细教程（之三）

偏特化模板的匹配优先级

在前面的章节我们提到了多种偏特化的模板的匹配优先级问题，那么当遇到多种偏特化时到底以哪一个为准呢？

这里，匹配优先级的原则是：**优先匹配特化程度更高的**。那么，怎么理解这个特化程度呢？我们先来举个最简单例子：

```
template <typename T1, typename T2, typename T3>
struct Test {}; // 【0】

template <typename T1, typename T2>
struct Test<T1, T2, int> {}; // 【1】

template <typename T>
struct Test<T, int, int> {}; // 【2】

void Demo() {
  Test<double, int, int> t; // 匹配【2】而不是【1】
}
```

上面这个很好理解，因为`Test<double, int, int>`显然是【2】更加符合这种形式。

那是不是可以理解成，优先匹配参数少的那个呢？未必？请看下面的实例：

```
template <typename T1, typename T2, typename T3>
struct Test {}; // 【0】

template <typename T1, typename T2>
struct Test<T1, T2, int> {}; // 【1】

template <typename T>
struct Test<int, int, T> {}; // 【2】

void Demo() {
```

```
  Test<int, int, int> t; // 匹配哪个？
}
```

大家可以先猜猜，上面这个例子会匹配哪个。如果按照参数少的来匹配，那应该匹配【2】才对。但实际情况是，哪个都不会匹配，会直接报错。

```
Ambiguous partial specializations of 'Test<int, int, int>'
```

所以我们一定要理解那句「特化程度」，它并不以参数个数为评判标准。再来看一个例子：

```
template <typename T1, typename T2>
struct Test {}; //  【0】

template <typename T1, typename T2, typename T3>
struct Test<T1, T2(T3)> {}; //  【1】

template <typename T>
struct Test<int, T> {}; //  【2】

void Demo() {
  Test<int, int(int)> t; // 会匹配哪个？
}
```

会匹配哪个呢？答案和上一个例子相同，会报Ambiguous的错误。所以说匹配时更关注的是参数的「形式」，而非个数。拿上面的例子来说，【1】号特化表示的是「第二个参数是一个函数类型，并且函数只有一个参数」的情况；【2】号特化表示的是「第一个参数为int的情况」。所以Test<int, int(int)>也是同时符合了，并没有「程度上」更贴近哪个，所以也会报错。

例子还有很多，这里就不过多列举了，总之，偏特化模板匹配的原则就是「特化程度更高者优先」，如果遇到可以同时命中多种的时候，将会报错（除非有更加匹配的特化，或者有对应的全特化，那么它会优先）。请读者一定要理解这里的「程度」究竟是什么含义。（这里只能说，要去体会了，因为它并没有一个可以描述出来的规则，所以希望大家能去体会。）

SFINAE

终于，我们来到了模板元编程前的最后一个基础知识——SFINAE。直接解释SFINAE会有点困难，也会让读者看着一头雾水，所以笔者打算稍后再来解释概念，我们先来看个引子：假如我现在要写一个模板函数，这个函数接收一个参数。最直接的方法就是这样：

```
template <typename T>
void f(T t) {}
```

但这样的问题就在于，如果T比较大，或者是一个不可复制的类型，那这样传参就会有问题，这种情况下应当用引用来传参：

```
template <typename T>
void f(const T &t) {}
```

但如果改成这样，对于那些基本类型（比如说int、char之类的）来说，是徒增了它的开销（引用本质是指针，所以多了一个指针的空间开销，以及若干取值、解指针的操作开销）。那能不能想个办法，把这两种情况都支持？暂时我们先不考虑是否可拷贝的问题，我们把长度小于等于一个指针大小的类型，用复制的方式传参，大于一个指针大小的类型，用引用传参。怎么做呢？其实方法很多，我们这里介绍一种可以引出SFINAE的方法，先看一下我们现在的诉求：

```
template <typename T>
void f(T t) {}

template <typename T>
void f(const T &t) {}

// 这里给一个比指针长的类型用来测试
struct Test {
  int arr[16];
};
```

上面这个例子中，我们写了2个同名的模板函数f，但我们希望针对一个确定的T时，只会按照**其中的一个**进行实例化。比如说f<int>就用第一种实例化，f<Test>就用第二种实例化。

但假如不加任何限制的话，编译器就会把两个f都进行实例化，比如说f<int>就会同时生成一组重载函数：

```
void f<int>(int t);
void f<int>(const int &t);
```

那么这时候我们再调用比如说f<int>(1)的时候，就会报错，因为两种函数原型都可以命中。

那如何让它只选其中的一个来生成实例呢？这需要我们用一个辅助类来完成：

```
template <typename T, bool Cond>
struct EnableIf {};

template <typename T>
struct EnableIf<T, true> {
  using type = T;
};
```

```
template <typename T>
void f(typename EnableIf<T, sizeof(T) <= sizeof(void *)>::type t) {
  std::cout << 1 << std::endl;
}

template <typename T>
void f(typename EnableIf<T, (sizeof(T) > sizeof(void *))>::type const &t) {
  std::cout << 2 << std::endl;
}

void Demo() {
  f<int>(1); // 打印1

  Test t;
  f<Test>(t); // 打印2
}
```


上面例程中，我们使用了一个辅助类`EnableIf`，接收两个参数。大家注意看第二个参数，是一个布尔类型，我们针对于第二个参数为`true`的情况进行了偏特化，此时把`T`透传出来。那么也就是说，第二个参数为`true`的时候，才存在`type`这个成员，而`false`时会命中通用模板，此时是没有`type`这个成员的。

然后我们在两个模板函数`f`中都使用了`typename EnableIf::type`，只是第二参数传入的条件不同。那么这个时候编译器会怎么做呢？这就是模板实例化时的一个核心环节——匹配（Substitution）。

所谓的「匹配」就是指，**编译器会拿着实参去尝试实例化**，比如，实例化`f<int>`的时候，编译器会先尝试用第一个模板函数来实例化，也就是变成了：

```
void f(typename EnableIf<int, sizeof(int) <= sizeof(void *)>::type t);
```

- 1

然后这里的判断条件是符合的，所以替换为`true`：

```
void f(typename EnableIf<int, true>::type t);
```

这时会去实例化`EnableIf<int, true>`，命中了它的偏特化，里面是有`type`的，而`typename EnableIf<int, true>::type`就是`int`，所以这里就变成了：

```
void f(int t);
```

没什么问题，于是编译器就按照这个模板生成了`f<int>`函数，函数原型是`void f<int>(int)`。我们称这个过程为「匹配成功（Substitution Success）」。

那对于`f<Test>`呢？同理，也会发生类似的过程。首先，按照第一个模板参数来尝试实例化：

```
void f(typename EnableIf<Test, sizeof(Test) <= sizeof(void *)>::type t);
```

然后判断条件不符合，所以替换为`false`：

```
void f(typename EnableIf<Test, false>::type t);
```

接下来实例化`EnableIf<Test, false>`，没有命中偏特化，因此要用通用模板来实例化。但`EnableIf<Test, false>`中并没有`type`这个成员。因此，`typename EnableIf<Test, false>::type`这个表达就是个错误的表达，无法用它来实例化。我们把这个过程称之为「匹配失败（Substitution Failure）」。

注意，重点来了！！虽然匹配失败了，但这时编译器并不会立刻报错，而是会继续尝试匹配其他的模板，因为我们还有一个模板函数`f`还没尝试呢！于是，编译器会继续用第二个模板尝试实例化：

```
void f(typename EnableIf<Test, sizeof(Test) > sizeof(void *)>::type const
&t);
```

变成：

```
void f(typename EnableIf<Test, true>::type const &t);
```

同理，命中了偏特化，此时`typename EnableIf<Test, true>::type`就是`Test`，所以变成了：

```
void f(Test const &t);
```

没问题，于是按照这个模板进行实例化，函数原型是：`void f<Test>(Test const &t)`。

因此，编译器在实例化模板时，如果遇到多个同名模板，则会逐一「尝试」匹配，在这个过程中如果发生了失败，并不会马上报错，因此把这种特性称之为「匹配失败不是错误（Substitution Failure Is Not An Error，简称SFINAE）」。

这里我们写的`EnableIf`其实就是STL中提供的`std::enable_if`的一个简化版。SFINAE是模板元编程最重要的理论基础，整个静态推导都是基于「构造一种模式，让其匹配成功或者失败」的方式来进行的。

小结

这一篇相对来说比较短，但内容却非常核心，希望大家能够理解SFINAE，并且通过上面的例程，开始建立起对模板元编程的概念。当使用`std::enable_if`的时候，其实我们就已经在进行模板元编程了！

至此，理论基础部分已经介绍完毕，下一篇会开始正式介绍模板元编程。

第五篇已脱稿，请看C++模板元编程详细教程（之五）