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）
C++模板元编程详细教程（之四）

STL中提供的工具

从这一篇开始，我们将正式介绍模板元编程。在STL中已经提供了很多静态的工具可供使用，模板元编程过程中自然少不了使用这些工具。当然，由于我们是模板元编程的详细教程，因此我也会带着大家来手写这些工具的源码。

需要注意的是，并不是所有的工具都是能手撸出来的，有极个别工具的底层实现依赖于编译器的“魔法操作”，真的遇到了那我们也没办法，其余能够手撸的，笔者都会介绍其手动实现方法。

STL中提供的模板元主要收纳在`type_traits`头文件中，有兴趣的读者可以查看官方的API参考文档。

模板元编程的两个要素

在上一篇，我们引出了`std::enable_if`的用法，开启了模板编程的新纪元。STL中的`std::enable_if`跟我们上一节写的demo还是稍微有一点区别的，下面是它的实现：

```
template <bool cond, typename T = void>
struct enable_if {};

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

从这个模板元的元老中可以看出，它有2个要素，一个是用于静态判断的`cond`，另一个是用于类型处理的`T`。所以，模板元编程无非就是做两件事，一个是静态数学计算（包括了布尔类型的逻辑运算和整数类型的数值运算。这里的「静态」是指编译期行为。）；另一个是类型处理（type traits，也被翻译为「类型萃取」）。

所以，**静态计算**和**类型处理**的编写过程，就称为「模板元编程」。把这两个要素的结果放到`enable_if`（或类型行为的模板）中，再通过SFINAE特性生成需要的代码再去参与编译。

强调一下，模板元编程是完完全全的编译期行为，任何设计运行期才能确定的数据都不可用，因此我们的编程思维跟普通代码编写的思维也是完全不同的，请读者一定要注意。

静态计算

静态计算主要有整数运算（C++20起也支持浮点数运算了）和逻辑运算。其中逻辑运算是重点，也是我们在模板元编程中最常用到的静态计算。下面分别来介绍。

整数运算

静态的整数运算在模板元编程中并不是特别常用，类似于数组元素个数这样的静态参数，往往在编写的时候都是确定好的，并不需要做额外的运算，即便是做了，可能也就是`+1`这种非常简单的运算，笔者在这里就不着重来介绍了。

我们看一个真正意义上使用静态整数运算的例子。假如我要在程序中用到斐波那契数列的第30项。注意！我只用它的第30项，30这个数是静态确定的，并不是程序的输入。我应该怎么做？

相信有读者会说，「嗨！这还不简单！写个计算的函数不就好了嘛！」

```
uint64_t Fib(uint16_t n) {
  if (n == 0 || n == 1) {
    return 1;
  }
  return Fib(n - 2) + Fib(n - 1);
}

void Demo() {
  std::cout << Fib(30) << std::endl;
}
```

这样做肯定是正确的，但仔细想想，`Fib`这个函数会跟随程序编译成一段指令，每次程序运行的时候，都会执行这段指令，现场算出`Fib(30)`，再打印出来。但斐波那契数列第30项它就是一个固定值呀！何必每次都要现算呢？

给出这样的提示，相信又会有读者会说，「哦，那我知道了！我先手算一遍，第30项应该是1346269，所以直接用这个值去替换」。

```
void Demo() {
  std::cout << 1346269 << std::endl;
}
```

非常好！如果你能想到这里，那离成功已经近一大步了！这里的`1346269`就是个固定值，斐波那契数列第30项无论什么时候都是这个值，不需要再去计算。可直接这样写上去一个魔数会让程序可读性变差，所以我们应该搞一个名字给它：

```
uint64_t fib_30 = 1346269;
void Demo() {
```

```
  std::cout << fib_30 << std::endl;
}
```

可读性是有了，但这样会引入另一个问题，`fib_30`是一个全局变量，既然是变量，它就是占内存空间的，并且会在主函数执行之前对其进行初始化。换句话说，上面的程序变成了，一开始先分配一片内存空间，把`1346269`这个值写进去，等后面需要用的时候，读取这片内存空间。

看得出，这片内存空间也是多余的，还是那句话，`1346269`这个值永久不变，编译期就应该确定，因此我们要再优化一下，让编译器用常量的方式来处理它：

```
constexpr uint64_t fib_30 = 1346269; // 这里换成宏定义也是一样的效果
void Demo() {
  std::cout << fib_30 << std::endl;
}
```

好，上面的需求我们解决了。那假如，这时我除了要用第30项以外，还要用第18、21、24和28项，怎么办？嗯，都算出来写上去肯定是一种方法：

```
constexpr uint64_t fib_18 = 4181;
constexpr uint64_t fib_21 = 17711;
constexpr uint64_t fib_24 = 75025;
constexpr uint64_t fib_30 = 1346269;

void Demo() {
  std::cout << 18 << "->" << fib_18 << std::endl;
  std::cout << 21 << "->" << fib_21 << std::endl;
  std::cout << 24 << "->" << fib_24 << std::endl;
  std::cout << 30 << "->" << fib_30 << std::endl;
}
```

但这时我们就发现了，这不是长久之计，毕竟咱不可能在程序里穷举出所有的数列的项，更何况像数值计算这种事怎么能是人工手算呢？万一算错一个数字都会有问题。我们要的效果是，**在程序运行之前，就把需要的数列项的值计算好**，而在程序运行的时候，这些值就是常数值了。因此，这就需要用到静态的数值计算。

编写的思路同样是递归，只不过要用模板元编程的方式，让所有的数值计算变为静态期。请看例程：

```
template <uint16_t N>
struct Fib {
  constexpr static uint64_t value = Fib<N - 2>::value + Fib<N - 1>::value;
};

template <>
```

```
struct Fib<0> {
  constexpr static uint64_t value = 1;
};

template <>
struct Fib<1> {
  constexpr static uint64_t value = 1;
};

void Demo() {
  std::cout << 18 << "->" << Fib<18> << std::endl;
  std::cout << 21 << "->" << Fib<21> << std::endl;
  std::cout << 24 << "->" << Fib<24> << std::endl;
  std::cout << 30 << "->" << Fib<30> << std::endl;
}
```


这里把`Fib`定义为一个模板类，其成员`value`是一个静态量，表示对应的数列项的值。通用模板中定义其为前两项的和，然后把第0项和第1项单独特化出来。这样一来，我们就实现了完全编译期计算的目的。

这样做的好处不仅仅是提升程序的运行性能，还可以把计算的值用做其他模板元。比如说，我希望定义一个数组，它的元素个数是斐波那契数列的第10项，那么就可以写成：

```
void Demo() {
  std::array<int, Fib(10)> arr; // Fib(10)是编译期可确定的值，因此可以传递
}
```

而这种情况如果是动态计算出的值，则无法作为函数参数传递，也就是说，下面这种写法是非法的：

```
uint64_t fib(uint16_t n) {
// 省略实现
}

void Demo() {
  std::array<int, fib(10)> arr; // 非法，因为fib(10)不是编译期可确定的值
  int arr2[fib(10)]; // 同理非法（有的编译器可能会优化容错使这种写法通过编译，但按语言标准来说，这种写法是非法的）
}
```

然而，在程序中用到数列的某一项这种需求其实几乎不存在，我们也仅仅是在讲解知识点的时候用做示例。真正在模板元编程中起作用的静态数值计算，其实是用于生成序列的功能。笔者将会在后续章节详细介绍生成序列的方法以及其用途。

逻辑运算

静态逻辑运算不单单是对数值做逻辑判断（比如说N>0这种），在模板元编程中，更多的是对「类型」进行是非判断。下面举一些例子可能更容易说明。

简单的静态判断

假如我希望判断当前类型是否是某一类型，如何来做？这里我们仍然是利用偏特化，下面例程用于展示，判断T1和T2是否是同一类型：

```
template <typename T1, typename T2>
struct is_same {
  constexpr static bool value = false;
};

template <typename T>
struct is_same<T, T> {
  constexpr static bool value = true;
};

// 下面是一个demo，利用enable_if，当参数为int时才生效
template <typename T>
typename std::enable_if<is_same<T, int>::value, void>::type
f(T t) {
}
```

上面例子本身很好理解，就不过多啰嗦了。不过不知道大家是否能发现，在处理这些静态逻辑判断时，使用的这些工具类，都会有一个静态的成员常量，我们这里叫value。当然了，实际上你把它叫什么名字都不影响使用，只不过在模板元编程中，我们倾向于符合STL当中的规范，使用value这个名字。

另一点就是，像上面这样把value展开来写会比较长，容易出错，也降低了一些可读性。在STL当中，把计算属性的部分抽象出了一个基类，同时，又给布尔类型的实例起了别名，就像下面这样：

```
template <typename T, T val> // 注意这里的写法
struct integer_constant {
  constexpr static T value = val;
};

// 对布尔类型的实例单独起名
template <bool val>
using bool_constant = integer_constant<bool, val>;
```

那么上一节讲到的斐波那契数列前两项的特化就可以改写成：

```
template <>
struct Fib<0> : std::integer_constant<uint64_t, 1> {};

template <>
struct Fib<1> : std::integer_constant<uint64_t, 1> {};
```

就避免了展开去写容易出错且可读性低的问题。

同时，由于布尔类型只有`true`和`false`两个取值，因此`bool_constant<true>`和`bool_constant<false>`也被单独定义了出来：

```
using true_value = bool_constant<true>;
using false_value = bool_constant<false>;
```

这样一来，我们改写一下前面的`is_same`：

```
template <typename T1, typename T2>
struct is_same : false_value {};

template <typename T>
struct is_same<T, T> : true_value {};
```

后续的示例中，我们都会采用这种方式而不是自行展开，也倡导读者在进行模板元编程时，按照这样隐形的规范来编写，这样别人在使用时会更方便。

由`is_same`还派生出了很多其他的，比如说`is_void`，`is_null_pointer`，`is_integral`等等，这里就不展开介绍了，感兴趣的读者可以去看STL源码，或者到参考手册中查看其实现方式。

模板元的与或非运算

逻辑运算既然有了，自然少不了与或非这3个基本布尔运算。这里的难点在于，如果我们拿到的是值，那直接用`&&`、`||`、`!`即可对值进行运算。但现在我们拿到的是逻辑模板元，也就是`true_value`或`false_value`的派生类。多个模板元之间如何进行布尔运算，得到一个新的模板元呢？

这里要先扯个题外话，「与」「或」「非」是工科范畴的叫法，通俗来说就是「俗称」，它们在离散数学领域是有一个「学名」的，分别叫做「合取(conjunction)」「析取(disjunction)」「取反(negation)」运算，STL中也是按照这个来命名的。

模板元编程里，思路都是一样的，就是偏特化，我们先来实现一下取反运算，其实就是对`true`和`false`的实例，写一个相反的特化即可：

```
template <typename T>
struct negation {};
```

```
template <>
struct negation<true_value> : false_value {};

template <>
struct negation<false_value> : true_value {};
```

这样一来，negation<true_value>会得到false_value，negation<false_value>会得到true_value，而对于其他类型的negation<T>则会命中通用模板，并不含有value成员，用于表示此项不合法。

那么合取和析取运算怎么写呢？我们知道，合取和析取运算是可以联立的，也就是多个值进行合取或析取。不过首先，我们先看看仅两个项的情况：

```
template <typename T1, typename T2>
struct conjunction : false_value {};

template <typename T2>
struct conjunction<true_value, T2> : T2 {};
```

根据逻辑短路原则，如果T1是false_value，那么直接返回false_value；如果T1是true_value，那么结果跟T2保持一致。

同理，析取也可以写出来：

```
template <typename T1, typename T2>
struct disjunction : true_value {};

template <typename T2>
struct disjunction<false, T2> : T2 {};
```

那么，多项的怎么处理呢？这时候就是递归大法好了，我们在两项相同的思路上来进行扩展：

```
template <typename... Args>
struct conjunction : false_value {};

// 单独考虑仅一个true_value的情况
template <>
struct conjunction<true_value> : true_value {};

// 多项的情况就是拆开，把第一项单独拿出来，如果第一项是true_value，那么就按后面的走，如果是false，此偏特化不命中，走向通用模板，而通用模板就是false_value
template <typename... Args>
struct conjunction<true_value, Args...> : conjunction<Args...> {};
```

```
// 用于验证的demo
void Demo() {
         std::cout << conjunction<true_value, true_value,
false_value>::value << std::endl; // 打印0
         std::cout << conjunction<true_value, true_value,
true_value>::value << std::endl; // 打印1
         std::cout << conjunction<false_value, false_value,
false_value>::value << std::endl; // 打印0
}
```

请读者着重去理解上面例程中的那几行注释。如果合取的没问题了，那么析取的也自然没问题：

```
template <typename... Args>
struct disjunction : true_value {};

template <>
struct disjunction<false_value> : false_value {};

template <typename... Args>
struct disjunction<false_value, Args...> : disjunction<Args...> {};
```

趁热打铁，来尝试这样一个需求：实现一个模板函数，包含两个参数，当这两个参数都是有符号的浮点数(`float`或`double`)时才生效。利用合取、析取以及`enable_if`来完成，不使用其他工具，那么效果如下：

```
template <typename T1, typename T2>
typename std::enable_if<
  std::conjunction<std::disjunction<
                                          std::is_same<T1, float>,
                       std::is_same<T1, double>
                      >, std::disjunction<
                       std::is_same<T2, float>,
                       std::is_same<T2, double>
                      >
                     >::value,
  void>::type
f() {}
```

小结

本篇我们主要介绍了模板元编程的两要素，然后详细介绍了其中的第一个，也就是静态数值计算。

下一篇将会开始介绍另一个要素，也就是类型处理的相关内容。

C++模板元编程详细教程（之六）