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）
C++模板元编程详细教程（之五）

类型处理

模板元编程的另一要素便是类型处理，英文叫type traits，所以也被翻译为「类型萃取」。其实这里的「萃取」并没有多么高大上的意思，类比前面章节介绍的「数值计算」，数值计算的结果应该是一个数值，而类型处理的结果就应该是个类型。

条件类型

最简单的类型处理就是进行条件选择，类似于if-else，如果条件为真则返回类型1，为假则返回类型2。STL中提供了`std::conditional`，其实现如下：

```
template <bool cond, typename T1, typename T2>
struct conditional {
  using type = T1;
};

template <typename T1, typename T2>
struct conditional<false, T1, T2> {
  using type = T2;
};

// 以下是使用的Demo
template <typename T>
void f(typename conditional<std::is_fundamental<T>::value, T, const T
&>::type t) {
}

void Demo() {
  int a = 0;
  std::string str = "abc";

  f(a); // f<int>(int)
```

```
  f(str); // f<std::string>(const std::string &)
}
```

上面的例子不难，但是值得解释的东西还是蛮多的，我们一个一个来。conditional是一个用于类型处理的辅助类，它拥有3个参数，第一个参数是一个静态布尔值，用于表示判断条件；后两个参数是用于选择的类型。当条件为真时，type成员会定义为前一个类型；当条件为假时，type成员会定义为后一个类型。

与前面介绍的value同理，这里的type也是STL中约定的命名方式，原则上可以不遵守，但倡导大家来遵守。type表示的就是这个辅助类的输出，既然这个辅助类的作用是「类型处理」，那么自然要输出一个类型。

在使用的Demo中我们可以看到，conditional<xxx, T, const T &>就是这里的辅助类型，而里面的xxx就需要一个静态布尔值。我们在这里用std::is_fundamental来判断一个类型是不是基本数据类型。取std::is_fundamental<T>::value获取这个判断的结果。再取conditional<xxx, T, const T &>::type用来获取类型处理的结果。

值得注意的是，conditional<xxx, T1, T2>本身是一种类型，但这个是**辅助类本身的类型**，而要**通过辅助类拿到类型处理的结果类型**则是要取一次type，也就是typename conditional<xxx, T1, T2>::type。（如果你不清楚为什么这里要加一个typename的话，可以参考C++的缺陷和思考（六）中“模板中类型定义和静态变量二义性”的章节，里面有详细解释）

因此，上面就是一个最简单的类型处理的模板元和它的使用方式。

辅助工具

相信大家应该已经发现了，使用这些辅助类型，再取成员（value或者type）会让代码迅速变长，尤其是取type的时候，还要加上typename，这玩意要是有嵌套的话，可读性会直接炸掉。因此，推荐的做法是针对这些辅助类搭配一个辅助工具，让代码变短，增强可读性。例如：

```
// 针对is_fundamental写的辅助工具
template <typename T>
constexpr inline bool is_fundamental_v = is_fundamental<T>::value;

// 针对conditional写的辅助工具
template <bool cond, typename T1, typename T2>
using conditional_t = typename conditional<cond, T1, T2>::type;
```

有了这两个工具，上一节的Demo代码就可以改写成：

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

在STL中，C++17标准下所有的模板元都配置了对应的辅助工具，用于数值计算的配备了对应_v结尾的工具，用于获取value；用于类型处理的配备了对应_t结尾的工具，用于获取type。所以上面的Demo如果完全使用STL则应该是：

```
template <typename T>
void f(std::conditional_t<std::is_fundamental_v<T>, T, const T &> t) {
}
```

在后面的教程中，我们使用STL工具的时候，如果要使用type或者value，我们都会优先使用辅助工具。而如果我们自己来编写模板元的话，也会按照这种方式来定义一个辅助工具去使用。这里也倡导大家如果要自行编写模板元，那么也应当按照这种方式提供对应的辅助工具。

变换型的类型处理

我们再来看一些其他的类型处理模板元。

我希望提供一个用于**去掉const**的工具，也就是说，如果一个类型有const修饰，那么返回去掉const后的类型，如果没有的话就原样返回。这个工具应该怎么做？

这里要注意的是，「一个类型有const修饰」是指这个类型本身，而不包括它内部含有的类型。举例来说，const int *并没有用const修饰，因为这个类型本身是指针类型，而const修饰的是它的解指针类型。因此它去掉const后应该还是它本身。而int *const类型才是用const修饰的，应当变换为int *。同理，const int *const应当变为const int *。再同理，const int &也应当是原样输出。

STL中也提供这样的工具，叫做remove_const，实现如下：

```
template <typename T>
struct remove_const {
  using type = T;
};

template <typename T>
struct remove_const<const T> {
  using type = T;
};

// 辅助工具
template <typename T>
using remove_const_t = typename remove_const<T>::type;
```

这时，当类型本身被const修饰时，会命中偏特化，T会识别为去掉const后的类型，因此type也就去掉了const。简单测试一下效果如下，读者也可以自行验证：

```
void Demo() {
  std::remove_const_t<int> a1; // int
  std::remove_const_t<const int> a2; // int
  std::remove_const_t<const int *> a3; // const int *
  std::remove_const_t<const int (*const)[5]> a4; // const int (*)[5]
  std::remove_const_t<void *const *> a6; // void *const *
  std::remove_const_t<void ** const> a7; // void **
  std::remove_const_t<void *const &> a8; // void *const &
}
```

完全退化

按照上一节的方法，我们自然也可以写出去掉指针符的，去掉引用符的，去掉右值引用符的，去掉`volatile`的，甚至可以组合起来，这些笔者在这里就不多介绍了，感兴趣的读者可以参考官方参考手册或阅读源码查看其实现。

不过有一种特殊的需求值得单拎出来讨论一下的。首先我们先来看一个例子：

```
template <typename T>
struct Test {
  Test(const T &t) {}
};


void Demo() {
  Test t{"abc"}; // 推导出Test<char[4]>类型
}
```

如果你是从头开始读本教程的，那么这个例子你应该会非常熟悉，这是前面「模板参数自动推导」章节中的一个例子。由于C++的特殊性，一些类型在函数传参的时候会进行隐式转换，例如数组类型会转换成首元素指针。另外就是各种形式的引用，在模板类型传递的时候也会很让人头大（这个在后面章节会有详细的例子）。因此，STL提供了一个模板，用于「完全退化」，它会去掉各种乱七八糟的修饰符，保留类型本身，并且遇到数组类型时会转换为首元素指针类型（注意这里仍然是类型本身的修饰符，或数组，内部嵌套的类型是不会改变的）。它就是`std::decay`，实现如下：

```
template <typename T>
struct decay {
 private:
  using U = std::remove_reference_t<T>;
 public:
  using type = std::conditional_t<
        std::is_array_v<U>,
        std::remove_extent_t<U> *, // 如果是数组就转指针
```

```
        std::conditional_t<
            std::is_function_t<U>,
            std::add_pointer_t<U>, // 如果是函数就转函数指针
            std::remove_cv_t<U>  // 其他情况则去掉const和volatile
  >>;
};

template <typename T>
using decay_t = typename decay<T>::type;
```


如果读者对于其中`is_array`、`remove_extent`、`is_function`、`add_pointer`、`remove_cv`的实现有疑问的话，可以参阅官方参考手册或阅读源码查看其实现。

有了`decay`以后，很多事情就好办了，它也是非常常用的一个工具。刚才那个例子就可以改写成：

```
template <typename T>
struct Test {
  Test(std::decay_t<T> const &t) {}
};


void Demo() {
  Test t{"abc"}; // 推导出Test<char[4]>类型，但传参时通过decay，实际调用的构造函数是Test(const char *const &t)
}
```

这样就避免了实例化生成`Test(const char (&t)[4])`类型的构造函数了。

小结

这一篇我们介绍了模板元编程的第二个要素，也就是类型处理。不过，这一篇介绍的都是非常基础的内容，模板元编程的威力还远远远远不止如此！目前我们介绍的都是单一类型的处理，然而对于复合类型（例如结构体，类类型）还可以继续深入到内部来做处理的。

下一篇开始将会是又一进阶型内容，我们将介绍更加复杂的类型处理。
C++模板元编程详细教程（之七）