前言

模板是C++体系中非常重要的一环，由其衍生的模板编程体系也算得上是C++特色，但也因为它学习门槛较高，语法很奇怪、反直觉让人望而却步。

笔者也经常碰到有人问我类似于「C++不写模板难道就不行吗？」「我们团队直接禁止写模板，我也没觉得对我的开发有什么影响啊。」「你这些特性我不用模板也能实现啊。」之类的问题。笔者仍然持有「困难的东西我们应当攻克下，再去客观评判其合理性。而不是为自己的逃避找开脱的理由」的态度，与此同时，笔者还认为「“能做”并不代表“适合”」，工具是用来帮助我们提升效率的，只有当我们充分了解了，才能判断何为「适用的场景」，从而真正让工具为我们的工作提供服务。因此感觉非常有必要出一篇完整且详细讲解C++模板元编程的教程。

本系列文章将会介绍如下的内容：

1. C++模板的基础用法
2. C++模板的编译方式和需要注意的点
3. SFINAE法则
4. 模板元编程的思路和体系结构
5. 模板范式和其他范式（例如OOP）的比较和转换
6. 更适合使用模板编程的场景极其理由
7. 一些个人的体会和思考

一些说明：

- 本文编写时C++20标准已经发行了一段时间了，但在业界还并不成熟，编译期对C++20标准的支持并不完善，并且C++20标准的权威文档、示例等也比较匮乏，目前很少有团队在生产环境推行C++20标准的。因此出于这样大环境的考虑，本文主要以C++17为默认的标准，重点介绍的也是C++17标准下的各种情况，对于C++17、C++14、C++11和C++98则不会做特别的标注和区分。而对于C++20上会有比较重大变化，或是其他涉及C++20标准时会特别标注出来。
- 「使用了模板」跟「模板元编程」是两回事，本文会先介绍模板的用法，后面会详细介绍模板元编程的内容。
- 本文**不适合完全没有模板编程基础的读者**，如果说你从来没写过模板，或者说压根连「模板」是什么都不知道的话，那建议先了解了解语法基础，有一些体会时再来阅读本文。本文更适合有「涉猎」过C++模板编程，有初步的体会，但希望自己能够更上一层楼的读者，或是已经有一定C++模板编程经历，但还没有系统化地穿线、回顾和来不及深入研究的读者。读完本文一定会让你对C++模板编程有一个更深层次的体会，让你的模板编程技术更上一层。
- 本文的目的旨在详细介绍C++模板元编程，因此读者如果发现一些示例和讲解上会出现「用其他语法或范式会更合适」的情况时，希望可以明了笔者在这里重点介绍的是如何用模板编程体系来解决问题，并不是倡导大家把所有的代码都改写过来，也**不是传达模**

板编程在任何情况下都比其他方式更好的意思，希望读者明了文章的目的，不要断章取义。
- 如果读者有任何问题，欢迎在评论区指正。笔者在后续发现文章问题的时候也会持续更正。

C++模板的基础用法

模板的概念

「模(mú)板」原本是工程上的概念，我们用「浇注」这个工序更好理解。首先你得有个模具，里面是空心的，空心的形状正好是最终工件的形状。然后我们把铁水浇注进去，等冷却后，打开模具，取出里面的工件。

从上面的例子我们可以总结出这么两个要点：

1. 模具本身不是工件，得用模具来生成工件
2. 最终的工件材料取决于浇注进去的材料（比如说浇注铁水出铁件，浇注铝水就出铝件），模具只决定工件的形状

类比到C++的模板也是一样的：

1. 模板本身不是实际代码，得用模板来生成代码（这个过程叫「实例化」）
2. 用模板实例化的代码取决于模板参数，模板本身只是一个「实例化的方式」（这一点能够解释很多时候为什么模板本身没有报错，但是实例化后报错了。）

那么这里我们能够看到模板拥有2个要点：「参数（类比工件材料）」和「实例化方式（类比工件形状）」

写一个hello world级别的模板

有了上一节的2个要点，我们自然而然就可以引出模板的基本语法，下面展示一个最简单的模板：

```
template <typename T>
void show(T t) {
  std::cout << t << std::endl;
}
```

`template`是模板关键字，表示后面即将定义一个模板。后面尖括号中的就是「模板参数」，这就是模板的第1个要点。之后的所有内容叫做「模板实现」，对应模板的第2个要点。

详细一点来解释就是，上面定义了一个「定义函数的方法」，参数中的`T`就是「材料」。把「材料」浇注到「模具」中就可以形成工件，也就是说，指定了「参数」的「模板函数」就可以形成「函数」。

那么使用模板的方法，就是给模板传递参数，使其实例化成为一个可实际调用的代码。以上面的例子来讲就是：

```
void Demo() {
  show<int>(5); // 首先实例化show<int>函数，然后再调用show<int>函数
}
```

这里请读者一定一定要建立一个概念，模板本身并不是可使用的代码，必须实例化以后才是。也就是说，这里的show**并不是函数**，而是「模板函数」。「模板函数」经过**实例化**后才能成为「函数」，而函数才能直接调用。

那么上面使用模板就分了2个步骤。首先，给模板函数show指定参数，将其实例化成show<int>，成为一个函数。那么，这个show<int>函数怎么定义呢？那就要看模板是怎么实现的了（这就是前一节说的，模板其实规定了实例化的方式）。因此，我们这里得到的函数是：

```
void show<int>(int t) {
  std::cout << t << std::endl;
}
```

第二步才是调用这个函数，所以这里尖括号中的int是模板参数，而小括号中的5则是函数实参。

请读者先熟悉并接受上面的理念，更加详细和深入的内容会在后面章节逐渐展开。

模板的分类

要问C++模板有哪些分类，倒不如问「哪些语法可以用模板生成」。按照这个维度，我们可以将模板分为：

- 模板类
- 模板函数
- 模板全局常量
- 模板类型重命名

接下来会简单展示一下各种类型的语法和大致用法。**注意：下面很多例程可能都不是非常准确，这里是为了方便读者理解才写的一个简化版本，不能够直接投入使用。**

模板类

顾名思义，模板类就是用于生成「类」的模板。不过这里的「类」并不是单指class，我们知道在C++中，class和struct的区别仅仅在默认权限上，因此几乎都可以互相代替，不过这仅仅实在「语法」上。在实际使用中，我们更在意它直观表达的「语义」，因此对于纯数据类型的组合，一般更习惯用struct，而对于带有操作的自定义类型（或者理解为OOP中的「抽象」），一般更习惯用class。比如我们可以写一个模板struct：

```
template <typename T1, typename T2>
struct Test {
  T1 t1;
```

```
  T2 t2;
};
```

只不过照着这种习惯来讲，「纯数据类型组合」似乎并没有太多写成模板的价值，所以对于模板类来说，class和struct的使用习惯还有那么一点不同。在模板类中，如果是用于「生成对象」的模板类，我们更习惯用class，而对于「模板元编程」的静态语言描述中，我们更习惯用struct。由于我们还没有详细介绍什么是「模板元编程」，因此读者暂且不必过多纠结，只需要知道struct关键字的模板也属于模板类就可以了。

下面例程就是一个简单的模板类和实例化调用的例子：

```
template <typename T, size_t size>
class Array {
 public:
  Array();
  T &at();
  size_t size() const;
 private:
  T data_[size];
};

void Demo() {
  Array<int, 5> arr; // 实例化，并创建对象
  arr.at(1) = 6;
}
```

关于实例化的步骤，读者可以参考「写一个hello world级别的模板」章节中的描述。

刚才我们说，class和struct都可以模板化，成为模板类。那union呢？其实union跟struct是基本类似的，唯一的区别在于成员共享首地址。因此从语法上来说，同样是支持union模板的:

```
template <typename T1, typename T2>
union Test {
  T1 m1_;
  T2 m2_;
};
```

只不过这种用法就跟「纯数据类型组合」的struct写成模板后类似，虽然语法上可行，但实际使用场景真的寥寥无几就是了（当然，一些炫技的骚操作除外，后面章节会详述）。

模板函数

模板函数就是用模板生成一个函数，主体是一个函数。当然，普通函数、成员函数、包括lambda表达式都是可以写成模板的。下面就一口气给出这几个语法的例程：

```
// 普通模板函数
template <typename T>
void show(const T &item) {
  std::cout << item << std::endl;
}

class Test {
 public:
  // 模板成员函数
  template <typename T>
  void mem_func(const T &item) {}
};

// 模板lambda表达式（只能是全局变量承载）
template <typename T>
auto f = [](const T &item) {}
```

实例化和使用的方式大致如下：

```
void Demo() {
  show<int>(5);

  Test t;
  t.mem_func<double>(5.1);

  f<char>('A');
}
```

模板全局常量

模板的全局常量一般是由一个模板类来做「引导」的，而且由于模板必须在编译期实例化，因此模板全局常量一定也会在编译期**有一个确定的值**，必须由constexpr修饰，而不可以是「变量」（不加constexpr的模板全局变量虽然语法允许，但并不符合模板定义的初衷，后续会有章节来专门介绍，暂时请读者建立「模板全局变量必须是常量表达式」的印象）。

下面给出一个例程：

```
// 用于引导模板全局常量的模板类(用于判断一个类型的长度是否大于指针)
template <typename T>
struct IsMoreThanPtr {
  static bool value = sizeof(T) > sizeof(void *);
```

```
};

// 全局模板常量
template <typename T>
constexpr inline bool IsMoreThanPtr_v = IsMoreThanPtr<T>::value;
```

这样的用法在后面章节所要介绍的「模板元编程」中非常重要的作用，后续会详细介绍。

模板类型重命名

C++中的类型重命名主要有两种语法，一种是`typedef`，另一种是`using`，它们都支持模板生成，效果是相同的。

模板类型重命名可以直接借用一个模板类来做「偏特化」，或者也可以像模板全局常量一样由一个模板类来引导，请看下面例程：

```
// 普通的模板类
template <typename T, size_t size>
class Array {};

// 偏特化作用的模板类型重命名
template <typename T>
using DefaultArray = Array<T, 16>;

// 也可以作用给typedef语法
template <typename T>
typedef DefaultArray<T *> DefaultPtrArray;

void Demo() {
  DefaultArray<int> arr1; // 相当于Array<int, 16> arr1;
  DefaultPtrArray<char> arr2; // 相当于Array<char *, 16> arr2;
}
```

再展示一个由模板类做引导的例子：

```
// 用于引导的模板类
template <typename T>
struct GetPtr {
  using type = T *;
};
// 用模板类引导的模板类型重命名
template <typename T>
using GetPtr_t = typename GetPtr<T>::type;
```

```
void Demo() {
  GetPtr_t<int> p; // 相当于typename GetPtr<int>::type p; 也相当于int *p;
}
```

上面这种写法同样在模板元编程中非常重要，后续章节会详细讲解。

模板参数

在上一章节中，我们展示了几种基本的模板语法，相信读者也注意到了在例程中展示的一些模板参数。那么这一节笔者将详细介绍模板参数。

C++的模板参数主要分为三种：

1. 类型
2. 整数（或整数的衍生类型）
3. 模板

类型好理解，就是这个参数要求传入某种「数据类型」。整数也好理解，就是字面意思。但这个「整数的衍生类型」还有「模板参数模板」就比较有趣且复杂了，下面我们来逐一攻克。

类型模板参数

模板参数中最常用的就是这里的类型了，用于传递一个类型来实例化模板。关键字是`typename`。先看一个简单的例子：

```
template <typename T>
void show(T t) {
  std::cout << t << std::endl;
}
```

`show`是一个模板函数，接受1个参数`T`，并且它是一个类型参数。实例化的时候，就需要在`T`的位置传入一个类型标识符（例如`int`、`void *`或者`std::string`）。

```
void Demo() {
  show<int>(5);
  show<void *>(nullptr);
  show<std::string>("abc");
}
```

这里值得一提的是，在早些版本的C++中，用于表示类型参数的关键字是`class`，但是这个关键字可能会产生歧义，让人觉得这里必须要传一个「类」类型。但其实并不是这样的，基本类型也是OK的。所以后续又支持了`typename`关键字来表示类型参数，更加贴近语义一些。不过`class`也保留下来了，现在用来表示「模板的类型参数」这里，两个关键字是相同的，没有区别。所以上面的例程也可以写作：

```
template <class T>
void show(const T &t) {
  std::cout << t << std::endl;
}
```

正如上面例程展示的这样，类型模板参数除了直接使用以外，还可以和其他符号（比如*、&、&&、const等）进行组合。

整数

我们在前面的章节也展示过整数作为模板参数的情况，当时的例子是：

```
template <typename T, size_t size>
class Array {
  // ...
 private:
  T data[size];
};
```

那么这里的size就是整数，当然不止size_t类型，一切整型都是支持的，比如说int、short、char，甚至bool都是可以。

但这里需要再次强调一点，我们多次强调的一个问题，模板是编译期语法，因此，这里的整型数据也必须是编译期能确定的，比如说常数、常量表达式等，而不可以是动态的数据。请看下面例程：

```
// 整数参数的模板
template <int N>
struct Test {};

void Demo() {
  Test<5> t1; // 常数OK

  constexpr int a = 5;
  Test<a> t2; // 常量表达式OK

  const int b = 6;
  Test<b> t3; // ERR，b是只读变量，不是常量

  Test<a * 3> t4; // 常数运算OK

  std::vector<int> ve {1, 2, 3};
  Test<ve.size()> t5; // ERR，size是运行时数据
  Test<ve[1]> t6; // ERR，ve的成员是运行时数据
```

```
  int arr1[] {1, 2, 3};
  Test<arr1[0]> t6; // ERR，arr1的成员是运行时数据

  constexpr int arr2[] {2, 4, 6};
  Test<arr2[1]> t7; // 常量表达式修饰的普通数组成员OK
}
```

所以请读者把握一个原则，就是只有编译期能够确定的量，才能用来实例化模板。

在C++20以前，只允许整数参数，但从C++20起，可以支持浮点数做参数，同样，只要是编译期能确定的量就OK：

```
// C++20标准：
template <double C>
struct Test {};
```

整数的衍生类型

说到整数的衍生类型，也就是说「可以用整数表示」，或者说「本质上是整数」的类型。

指针类型

这里面最经典的就是指针，请看下面例程：

```
template <int *p>
void f(int data;) {
  *p = data;
}
```

不过这里实例化时能够支持的数据就很有说头了。既然我们说指针是整数的衍生类型（指针本质就是地址，地址就是个整数），那么规则自然也是跟整数的规则是一样的，要「编译期能够确认的值」。

只不过，对于指针来说，「编译期能够确认的值」就应该指的是「编译期能够确定的地址」。但这显然是个伪命题，因为编译期根本不存在地址的概念，程序在只有在进程预加载的时候才会确定所有变量的地址。所以这个问题需要我们换一个角度来看。**如果说某一个变量，它的地址在程序运行期间能够一直不发生改变，或者说它不会中途被释放**的话，我们就认为这个变量的地址是「确定的」。或者说，这个地址一旦确定要给这个变量来使用，那么它就一直都会给它使用，而不会中途变成交给其他变量。

沿着这个思路，只要是「程序运行中确定的」地址，就可以用来实例化模板。下面给出一些实例：

```
int a = 1; // 全局变量

class Test {
 public:
```

```
  int m1; // 成员变量
  static int m2; // 静态成员变量
};

int Test::m2 = 4;

void Demo() {
  int b = 2; // 局部变量
  static int c = 3; // 静态局部变量

  f<&a>(0); // OK，a是全局变量，程序运行期间不会被释放
  f<&b>(0); // ERR，b是局部变量，在局部代码块运行完毕后会被释放，所以说b的地址也有可能不仅仅表示b，回收后可能会表示其他数据，所以不可以
  f<&c>(0); // OK，c是静态局部变量，不会随着代码块的结束而释放

  f<&Test::m1>(0); // ERR，Test::m1其实并不是变量，要指定了对象才能确定，因此是非确定值，所以不可以
  f<&Test::m2>(0); // OK，Test::m1本质就是一个全局变量，在程序运行期间不会被释放，所以OK
}
```

希望读者能够通过上面的例子来理解什么是「不变的地址」，其实并不说是说地址不可变，因为地址本身就是不可变的。这里说的是，这个地址一直只表示确定的数据，而不会改变（不会被释放后重新分配给其他数据）。

函数类型

既然讲到指针类型，那我们也逃不开一类特殊的指针——函数指针。函数指针其实本质上也是地址，所以同样属于整数的衍生类型。而分配给函数（指令段）的地址在程序执行过程中就是不会变的，但如果用的是变量的值那么同样会因为动态数据问题而报错。

文字解释不直观，我们还是来看例程：

```
// 函数指针类型模板参数
template <void (*func)()>
void f() {
  func();
}

// 普通函数
void f1() {}

class Test {
 public:
  void f2() {} // 成员函数
```

```
  static void f3() {} // 静态成员函数
};

void Demo() {
  void (*pf1)() = &f1; // 局部变量
  constexpr void (*pf2)() = &f1; // 常量表达式

  f<&f1>(); // OK，函数本身就是地址不可变的
  f<&Test::f2>(); // ERR，虽然成员函数也是地址不可变的，但&Test::f2的类型是void (Test::*)()，类型不匹配所以报错
  f<&Test::f3>(); // OK，静态成员函数是地址不可变的，类型也匹配
  f<pf1>(); // ERR，pf1是静态数据，编译期值不确定，所以不可以
  f<pf2>(); // OK，用常量表达式修饰的在编译期可以确定，所以可以
}
```

在C语言中，直接使用函数名其实就会转义为函数指针类型。但C++引入了「函数类型」的概念，它在很多时候也是可以转换为函数指针类型的。那么在模板参数中，同样也支持了「函数类型」，用法跟上述函数指针类型完全相同，并且还可以跟函数指针类型互用：

```
// 函数类型模板参数
template <void func()>
void f() {
  func();
}

// 普通函数
void f1() {}

class Test {
 public:
  void f2() {} // 成员函数
  static void f3() {} // 静态成员函数
};

void Demo() {
  void (*pf1)() = &f1; // 局部变量
  constexpr void (*pf2)() = &f1; // 常量表达式

  f<f1>(); // OK
  f<&f1>(); // OK
  f<Test::f3>(); // OK
  f<&Test::f3>(); // OK
  f<pf2>(); // OK
```

```
  // 与前一例程相同，不再赘述
}
```

同样地，「函数类型」本身就可以隐式转换为「函数指针类型」，因此使用上都是互通的，就不再赘述了。不过这里提前剧透一下，虽然在大多数情况下函数类型和函数指针类型都可以互转，没有明显区别，但在一种特殊场景下二者会有着天壤之别，详情会在后续介绍到模板偏特化的章节中提到。

模板类型

这个类型非常容易让人晕菜，所谓「模板类型的模板参数」，其实就是嵌套模板的意思，把「某一种类型的模板」作为一个参数传给另一个模板。请看示例：

```
// 模板类型的模板参数
template <template <typename, typename> typename Tem>
void f() {
  Tem<int, std::string> te;
  te.show();
}

// 符合条件的模板类
template <typename T1, typename T2>
struct Test1 {
  void show() {std::cout << 1 << std::endl;}
};

template <typename T1, typename T2>
struct Test2 {
  void show() {std::cout << 2 << std::endl;}
};

// 不符合条件的模板类
template <int N>
struct Test3 {
  void show() {std::cout << 3 << std::endl;}
};

void Demo() {
  f<Test1>(); // 注意这里，要传模板，而不是实例化后的模板
  f<Test<int, int>>(); // ERR，模板参数类型不匹配

  f<Test2>(); // OK
  f<Test3>(); // ERR，类型不匹配
}
```

上述例子中，模板函数`f`接受一个参数，这个参数需要是一个模板类型，并且是一个「含有2个类型参数的模板」类型。

在实例化的示例中，`Test1`和`Test2`都是`template <typename, typename>`类型，所以可以用来实例化`f`。

而实例化后的`Test<int, int>`已经是一个普通类型了，也就是说它是一个`typename`类型，而不是`template <typename, typename>`类型，所以不匹配。同理，`Test3`是`template <int>`类型，也不是`template <typename, typename>`类型，所以不匹配。

小结

本篇介绍了本系列文章，然后介绍了模板的概念，还有最基础的C++模板的用法。

关于更高级的用法请关注本系列后续文章。下一篇将重点介绍模板参数的自动推导。

C++模板元编程详细教程（之二）