外部模板

传统 C++ 中，模板只有在使用时才会被编译器实例化。换句话说，只要在每个编译单元（文件）中编译的代码中遇到了被完整定义的模板，都会实例化。这就产生了重复实例化而导致的编译时间的增加。并且，我们没有办法通知编译器不要触发模板实例化。

C++11 引入了外部模板，扩充了原来的强制编译器在特定位置实例化模板的语法，使得能够显式的告诉编译器何时进行模板的实例化：

```
template class std::vector<bool>; // 强行实例化
extern template class std::vector<double>; // 不在该编译文件中实例化模板
```

尖括号 ">"

在传统 C++ 的编译器中，>>一律被当做右移运算符来进行处理。但实际上我们很容易就写出了嵌套模板的代码：

```
std::vector<std::vector<int>> wow;
```

这在传统C++编译器下是不能够被编译的，而 C++11 开始，连续的右尖括号将变得合法，并且能够顺利通过编译。

类型别名模板

在了解类型别名模板之前，需要理解『模板』和『类型』之间的不同。仔细体会这句话：模板是用来产生类型的。在传统 C++中，typedef 可以为类型定义一个新的名称，但是却没有办法为模板定义一个新的名称。因为，模板不是类型。例如：

```
template< typename T, typename U, int value>
class SuckType {
public:
 T a;
 U b;
 SuckType():a(value),b(value){}
};
template< typename U>
typedef SuckType<std::vector<int>, U, 1> NewType; // 不合法
```

C++11 使用 using 引入了下面这种形式的写法，并且同时支持对传统 typedef 相同的功效：

> 通常我们使用 typedef 定义别名的语法是：typedef 原名称 新名称;，但是对函数指针等别名的定义语法却不相同，这通常给直接阅读造成了一定程度的困难。

```
typedef int (*process)(void *); // 定义了一个返回类型为 int，参数为 void* 的函数指针类型，名字叫做 process
using process = int(*)(void *); // 同上，更加直观

template <typename T>
using NewType = SuckType<int, T, 1>; // 合法
```

默认模板参数

我们可能定义了一个加法函数：

```
template<typename T, typename U>
auto add(T x, U y) -> decltype(x+y) {
 return x+y
}
```

但在使用时发现，要使用 add，就必须每次都指定其模板参数的类型。

在 C++11 中提供了一种便利，可以指定模板的默认参数：

```
template<typename T = int, typename U = int>
auto add(T x, U y) -> decltype(x+y) {
 return x+y;
}
```

变长参数模板

模板一直是 C++ 所独有的黑魔法（一起念：Dark Magic）之一。在 C++11 之前，无论是类模板还是函数模板，都只能按其指定的样子，接受一组固定数量的模板参数；而 C++11 加入了新的表示方法，允许任意个数、任意类别的模板参数，同时也不需要在定义时将参数的个数固定。

```
template<typename... Ts> class Magic;
```

模板类 Magic 的对象，能够接受不受限制个数的 typename 作为模板的形式参数，例如下面的定义：

```
class Magic<int,
 std::vector<int>,
 std::map<std::string,
 std::vector<int>>> darkMagic;
```

既然是任意形式，所以个数为0的模板参数也是可以的：`class Magic<> nothing;`。

如果不希望产生的模板参数个数为0，可以手动的定义至少一个模板参数：

```
template<typename Require, typename... Args> class Magic;
```

变长参数模板也能被直接调整到到模板函数上。传统 C 中的 printf 函数，虽然也能达成不定个数的形参的调用，但其并非类别安全。而 C++11 除了能定义类

别安全的变长参数函数外，还可以使类似 printf 的函数能自然地处理非自带类别的对象。除了在模板参数中能使用 `...` 表示不定长模板参数外，函数参数也使用同样的表示法代表不定长参数，这也就为我们简单编写变长参数函数提供了便捷的手段，例如：

```
template<typename... Args> void printf(const std::string &str, Args... args);
```

那么我们定义了变长的模板参数，如何对参数进行解包呢？

首先，我们可以使用 `sizeof...` 来计算参数的个数，：

```
template<typename... Args>
void magic(Args... args) {
 std::cout << sizeof...(args) << std::endl;
}
```

我们可以传递任意个参数给 `magic` 函数：

```
magic(); // 输出0
magic(1); // 输出1
magic(1, ""); // 输出2
```

其次，对参数进行解包，到目前为止还没有一种简单的方法能够处理参数包，但有两种经典的处理手法：

1. 递归模板函数

递归是非常容易想到的一种手段，也是最经典的处理方法。这种方法不断递归的向函数传递模板参数，进而达到递归遍历所有模板参数的目的：

```
#include <iostream>
template<typename T>
void printf(T value) {
 std::cout << value << std::endl;
}
template<typename T, typename... Args>
void printf(T value, Args... args) {
 std::cout << value << std::endl;
 printf(args...);
}
int main() {
 printf(1, 2, "123", 1.1);
 return 0;
}
```

2. 初始化列表展开

> 这个方法需要之后介绍的知识，读者可以简单阅读以下，将这个代码段保存，在后面的内容了解过了之后再回过头来阅读此处方法会大有收获。

递归模板函数是一种标准的做法，但缺点显而易见的在于必须定义一个终止递归的函数。

这里介绍一种使用初始化列表展开的黑魔法：

```
// 编译这个代码需要开启 -std=c++14
// 因为版本原因，实验环境中的 g++ 尚不支持此特性，此处可以使用 clang++ 替代 g++
template<typename T, typename... Args>
auto print(T value, Args... args) {
 std::cout << value << std::endl;
 return std::initializer_list<T>{([&] {
 std::cout << args << std::endl;
 }(), value)...};
}
int main() {
 print(1, 2.1, "123");
 return 0;
}
```

在这个代码中，额外使用了 C++11 中提供的初始化列表以及 Lambda 表达式的特性（下一节中将提到），而 std::initializer_list 也是 C++11 新引入的容器（以后会介绍到）。

通过初始化列表，`(lambda 表达式, value)...` 将会被展开。由于逗号表达式的出现，首先会执行前面的 lambda 表达式，完成参数的输出。唯一不美观的地方在于如果不使用 `return` 编译器会给出未使用的变量作为警告。

> 事实上，有时候我们虽然使用了变参模板，却不一定需要对参数做逐个遍历，我们可以利用 `std::bind` 及完美转发等特性实现对函数和参数的绑定，从而达到成功调用的目的。