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

模板特化

有了前两篇的基础，相信大家对模板编程已经有一点初步的感觉了。趁热打铁，这一篇我们主要来介绍一下模板特化。

首先来看一下下面的例子：

```
template <typename T>
void add(T &t1, const T &t2) {
  t1 += t2;
}
```

上面是一个简单的模板函数，用于把第二个参数的值加到第一个参数中去。这个模板函数对于基本数据类型的实例化都是没什么问题的，但是如果是字符串的话，那将会有问题：

```
void Demo() {
  int a = 1, b = 3;
  add(a, b); // add<int>，调用符合预期

  char c1[16] = "abc";
  char c2[] = "123";

  add(c1, c2); // add<char *>， 调用不符合预期
}
```

这里的问题就在于，对于字符串类型（这里指原始C字符串，而不是`std::string`）来说，「相加」并不是简单的`+=`，因为字符串主要是用字符指针来承载的，指针相加是不合预期的。我们希望的是字符串拼接。

因此，我们希望，单独针对于`char *`的实例化能够拥有不同的行为，而不遵从「通用模板」中的定义。这种语法支持就叫做「特化」，或「特例」。可以理解为，针对于模板参数是某种特殊情况下进行的特殊实现。

因此，我们在通用模板的定义基础上，再针对`char *`类型定义一个特化：

```
#include <cstring>

template <typename T>
void add(T &t1, const T &t2) {
  t1 += t2;
}

template <> // 模板特化也要用模板前缀，但由于已经特化了，所以参数为空
void add<char *>(char *&t1, char *const &t2) { // 特化要指定模板参数，模板体中也要使用具体的类型
  std::strcat(t1, t2);
}

void Demo() {
  int a = 1, b = 3;
  add(a, b); // add<int>是通过通用模板生成的，因此本质是a += b，符合预期

  char c1[16] = "abc";
  char c2[] = "123";

  add(c1, c2); // add<char *>有定义特化，所以直接调用特化函数，因此本质是strcat(c1, c2)，符合预期
}
```


上例简单展示了一下模板特化目标解决的问题，和其基本的语法。但其实模板特化远不止如此，它有着巨大的潜力。

模板的特化分两种情况，一种是全特化（有的地方也叫特例），一种是偏特化（有的地方也叫部分特化）。全特化相对简单一些，笔者会先来介绍。而偏特化会伴随SFINAE理论，它将会成为模板元编程最核心的理论基础。

全特化与模板的链接方式

首先复习一下我们在开篇时候所提到的一个非常重要的概念。**模板本身不是可使用的代码，而是一种代码的升成方法。需要经过实例化后才能成为实际可用的代码。**比如说模板函数需要指定模板参数（可以是显式指定，也可以是编译器自动推导）实例化后，才能成为函数，同理，模板类也需要实例化后才能成为类。

然而「全特化」就是说，当**所有模板参数都指定了**的时候，才叫「全」。那么上一节中`add`的示例就是一个全特化，因为它原本只有一个模板参数，把它特化了自然是「完全」特化的。

而要谈到全特化，就不得不谈到一个非常容易踩坑的点，那就是模板的链接方式。在一个单独的.cpp文件中使用模板并不会有什么链接性问题，但如果在多个文件中都要使用呢？自然要通过「头文件声明+链接」的方式来完成了。

但模板本身又很特殊，它本身不是可用的代码，而是代码生成器，因此编译器会在编译期用模板来生成代码，注意，这个时候还没有开始链接！所以问题就产生了，假如我们按照直觉和习惯，把模板的声明和定义分文件来写，会怎么样呢？请看下面示例：

tmp.h

```
#pragma once

template <typename T>
void f(const T &t); // 声明
```

tmp.cpp

```
#include "tmp.h"

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

main.cpp

```
#include "tmp.h"

int main() {
  f(1); // f<int>
  f(1.0); // f<double>
  return 0;
}
```

如果我们真的这么做了，你会发现链接时会报错。原因是这样的，我们在tmp.h中的这种写法，并不是「声明了一个模板函数」，模板函数本不是函数，是不需要声明的，大家记住模板永远是生成代码的工具。所以tmp.h中的写法是「声明了一组函数」，包括我们在`main`函数中使用的`f<int>`和`f<double>`，之所以能通过编译，就是因为tmp.h中存在它们的声明。换句话说，`template <typename T> void f(const T &);`相当于`void f<int>(const int &);`,`void f<double>(const double &);`,`void f<char *>(char *const &);`.....这一系列的「函数声明」。

所以，编译是没问题的，但是链接的时候会报找不到`f<int>`和`f<double>`的实现。这是为什么呢？明明我在tmp.cpp中实现了呀！那我们来「换位思考一下」，假如你是编译器，我们知道「编译」过程是单文件行为，那么你现在来编译main.cpp，首先进行预处理，把`#include`替换成对应头文件内容，那么main.cpp就变成了:

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

```
int main() {
  f(1); // f<int>
  f(1.0); // f<double>
  return 0;
}
```

上面的f是函数声明，下面编译主函数的时候，根据参数推导出了f<int>和f<double>，于是，通过上面的「模板函数声明」生成了2条实际的「函数声明」语句，也就是：

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

调用都是符合声明的，OK，结束编译，我们得到了main.o。

好了，下面我们来编译tmp.cpp。同理，先做预处理，得到了：

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

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

这时，**问题的关键点来了！**，这个模板函数f在当前这个编译单元中，并没有任何实例化，那么你自然就不知道应当按这个模板来生成哪些实例。所以，你**只能什么都不做**，很无奈地生成了一个空白的tmp.o。

最后，main.o和tmp.o链接，main.o中的f<int>和f<double>都找不到实现，所以链接报错。

这就是模板的链接方式问题，由于模板都是编译期进行实例化，因此，必须在编译期就得知道需要哪些实例化，然后把这些实例化后的代码编译出来，再去参与链接，才能保证结果正确。

所以，要保证编译期能知道所有需要的实例，我们只能**把模板实现放在头文件里**。这样，每一个编译单元都能根据自己需要的实例来生成代码。也就是说，上面的代码应该改造成：

tmp.h

```
#pragma once
template <typename T>
void f(const T &t);

template <typename T>
void f(const T &t) {} // 当然，文件内部没有声明依赖关系的时候，声明和实现可以合并
```

main.cpp

```
#include "tmp.h"

int main() {
  f(1);
  f(1.0);

  return 0;
}
```

这时，在编译main.cpp时，就会把`f<int>`和`f<double>`的实例都编译出来，这样就不会链接报错了。

但这样会引入另一个问题，如果多个.cpp引入同一个含有模板的.h文件，并做了相同的实例化，会不会生成多份函数实现呢？这样链接的时候不是也会报错吗？

设计编译器的大佬们自然也想到这个问题了，那么解决方法就是，通过模板实例出的内容，会打上一个全局标记，最终链接时只使用一份（毕竟是从同一份模板生成出来的，每一份自然是相同的）。再换句更通俗易懂的说法就是**模板实例一定是`inline`的**，编译器会给每个模板实现自动打上`inline`标记，确保链接时全局唯一。

现在我们再回头看一下全特化模板，全特化模板已经是**实例化过**的了，因此并不会出现编译期不知道要怎么实例化的问题。如果这时我们还把实现放在头文件中会怎么样？

tmp.h

```
#pragma once
template <typename T>
void f(T t) {} // 通用模板

template <>
void f<int>(int t) {} // 针对int的全特化
```

t1.cpp

```
#include "tmp.h"

void Demo1() {
  f(1); // f<int>
}
```

t2.cpp

```
#include "tmp.h"

void Demo2() {
```

```
  f(1); // f<int>
}
```

我们再来当一次编译期。首先编译t1.cpp，预处理展开，得到了f<int>的实现，所以把f<int>编译过来，输出t1.o。同理，编译t2.cpp后，也会有一份f<int>的实现在t2.o中。最后链接的时候，发现f<int>重定义了！

因此我们发现，**全特化的模板其实已经不是模板了**，在这里f<int>会按照普通函数一样来进行编译和链接。所以直接把实现放在头文件中，就有可能在链接时重定义。解决方法有两种，第一种就是我们手动补上inline关键字，提示编译期要打标全局唯一。

tmp.h

```
#pragma once
template <typename T>
void f(T t) {} // 通用模板，编译器用通用模板生成的实例会自动打上inline

template <>
inline void f<int>(int t) {} // 针对int的全特化，必须手动用inline修饰后才能在编译期打标保证链接全局唯一
```

第二种方法就是，当做普通函数处理，我们把实现单独抽到一个编译单元中独立编译，最后在链接时才能保证唯一：

tmp.h

```
#pragma once
template <typename T>
void f(T t) {} // 通用模板

template <>
void f<int>(int t); // 针对int的全特化声明（函数声明）
```

tmp.cpp

```
#include "tmp.h"

template <>
void f<int>(int t) {} // 函数实现
```

之后，f<int>会随着tmp.cpp的编译，单独存在在tmp.o中，最后链接时就是唯一的了。

另外，对于特化的模板函数来说，参数必须是按照通用模板的定义来写的（包括个数、类型和顺序），但对于模板类来说，则没有任何要求，我们可以写一个跟通用模板压根没什么关系的一种特化，比如说：

```
template <typename T>
struct Test { // 通用模板中有2个成员变量，1个成员函数
  T a, b;
  void f();
};

template <>
struct Test<int> { // 特化的内部定义可以跟通用模板完全不同
  double m;
  static int ff();
}
```

偏特化

偏特化又叫部分特化，既然是「部分」的，那么就不会像全特化那样直接实例化了。偏特化的模板本质上还是模板，它仍然需要编译期来根据需要进行实例化的，所以，在链接方式上来说，全特化要按普通函数/类/变量来处理，而偏特化模板要按模板来处理。

先明确一个点：**模板函数不支持偏特化**，因此偏特化讨论的主要是模板类。

我们先来看一个最简单的偏特化的例子：

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

template <typename T>
struct Test<int, T> {};
```

上面代码就是针对`Test`模板类，第一个参数为`int`时的「偏特化」，那么只要是第一个参数为`int`的时候，就会按照偏特化模板来进行实例化，否则会按照通用模板进行实例化。为了方便说明，我们在通用模板和偏特化模板中加一些用于验证性的代码：

```
#include <iostream>

template <typename T1, typename T2>
struct Test {
};

template <typename T>
struct Test<int, T> {
  static void f();
```

```
};

template <typename T>
void Test<int, T>::f() {
  std::cout << "part specialization" << std::endl;
}

void Demo() {
  Test<int, int>::f(); // 按照偏特化实例化，有f函数
  Test<int, double>::f(); // 按照偏特化实例化，有f函数
  Test<double, int>::f(); // 按照通用模板实例化，不存在f函数，编译报错
}
```


偏特化模板本身仍然是模板，仍然需要经历实例化。但偏特化模板可以指定当一些参数满足条件时，应当按照指定方式进行实例化而不是通用模板定义的方式来实例化。

那如果偏特化和全特化同时存在呢？比如下面的情况：

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

template <typename T>
struct Test<int, T> {}; // 【1】偏特化模板

template <>
struct Test<int, int> {}; // 【2】全特化模板

void Demo() {
  Test<int, int> t; // 按照哪个实例化？
}
```

先说答案，上面的实例会按照【2】的方式，也就是直接调用全特化。大致上来说，全特化优先级高于偏特化，偏特化高于通用模板。

对于函数来说，**模板函数不支持偏特化**，但支持重载，并且重载的优先级高于全特化。比如说：

```
void f(int a, int b) {} // 重载函数

template <typename T1, typename T2>
void f(T1 a, T2 b) {} // 通用模板

template <>
void f<int, int>(int a, int b) {} // 全特化
```

```
void Demo() {
  f(1, 2); // 会调用重载函数
  f<>(1, 2); // 会调用全特化函数f<int, int>
  f(2.5, 2.6); // 会用通用模板生成f<double, double>
}
```

回到模板类的偏特化上，除了上面那种制定某些参数的偏特化以外，还有一种相对复杂的偏特化，请看示例：

```
template <typename T>
struct Tool {}; // 这是另一个普通的模板类

template <typename T>
struct Test {}; // 【0】通用模板

template <typename T>
struct Test<Tool<T>> {}; // 【1】偏特化

void Demo() {
  Test<int> t1; // 使用【0】实例化Test<int>
  Test<Tool<int>>; // 使用【1】实例化Test<Tool<int>>
  Test<Tool<double>>; // 使用【1】实例化Test<Tool<double>>
}
```

有的资料会管上面这种特化叫做「模式特化」，用于区分普通的「部分特化」。但它们其实都属于偏特化的一种，因为偏特化都是相当于特化了参数的范围。在上面的例子中，我们是针对于「参数是Tool的实例类型」这种情况进行了特化。

所以，偏特化并不一定意味着模板参数数量变小，它有可能不变，甚至有可能是增加的，比如说：

```
template <typename T1, typename T2>
struct Tool {}; // 这是另一个普通的模板类

template <typename T>
struct Test {}; // 【0】通用模板

template <typename T1, typename T2>
struct Test<Tool<T1, T2>> {}; // 【1】偏特化模板

template <typename T>
struct Test<Tool<int, T>> {}; // 【2】偏特化模板

void Demo() {
```

```
  Test<int> t1; // 【0】
  Test<Tool<int, double>> t2; // 【2】
  Test<Tool<double, int>> t3; // 【1】
}
```

所以偏特化的引入，让模板编程这件事有了爆炸性的颠覆，因为其中的组合可以随意发挥想象。但这里就引入了另一个问题，就比如上例中，【1】和【2】都是偏特化的一种，但为什么`Test<Tool<int, double>>`选择了【2】而不是【1】呢？这么说，看来不仅仅是跟全特化和通用模板存在优先级问题，多种偏特化之间也仍然存在优先级问题，那么编译器究竟是按照什么方式来进行偏特化匹配的呢？这就是我们下一篇要着重研究的问题了。

小结

这一篇我们主要介绍了模板的链接方式和模板的特化，重点希望读者理解和掌握的是模板类的偏特化，因为C++的模板元编程其实就是一系列模板的偏特化来实现各种静态功能的。

下一篇会介绍偏特化的优先级匹配法则和SFINAE特性。
C++模板元编程详细教程（之四）