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）
C++模板元编程详细教程（之六）

前面我们介绍了一些基础的静态数值计算和类型处理，这一篇开始将会介绍一些进阶型的内容，做更加复杂的逻辑判断和类型处理。

函数类型的处理

如果我希望处理出一个函数类型的返回值，要怎么办呢？请看例程：

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

template <typename R, typename... Args>
struct GetRet<R(Args...)> {
  using type = R;
};

template <typename T>
using GetRet_t = typename GetRet<T>::type;

// 以下是示例
int f() {return 0;}
void Demo() {
  GetRet_t<decltype(f)> a;
  std::cout << std::is_same_v<std::decay_t<decltype(a)>, int>; // true
}
```

我们注意到，此时的偏特化，用到了前面章节模板基础知识中的「函数类型」，也就是说，只有符合`R(Args...)`形式的参数才会入到这个偏特化中。那么在这种情况下，「函数类型」和「函数指针类型」就是截然不同的。也就是说下面的代码不能正确解析：

```
void Demo() {
  GetRet_t<decltype(f)> a; // f是函数类型，可以正确推导
```

```
  GetRet_t<decltype(&f)> b; // &f是函数指针类型，不能命中偏特化，而是会用通用模板，又因为通用模板不含type成员，因此这里报错
}
```

同样，仿函数类型、lambda类型、函数对象类型、成员函数类型都无法命中，进而无法取出返回值。因此，我们如果希望支持所有的情况，那就还要考虑支持其他的类型。对于仿函数和lambda类型来说，我们就要去取出它的operator ()方法的返回值类型，对于函数对象类型来说也是一样的。所以我们代码可以改造成：

```
template <typename T>
struct GetRet {
 private:
  using DT = std::decay_t<T>;
 public:
  // 如果内部含有operator()就取它的类型
  using type = typename GetRet<decltype(&DT::operator())>::type;
};

// 对于函数类型
template <typename R, typename... Args>
struct GetRet<R(Args...)> {
  using type = R;
};

// 对于函数指针类型
template <typename R, typename... Args>
struct GetRet<R(*)(Args...)> {
  using type = R;
};

// 对于非静态成员函数类型
template <typename T, typename R, typename... Args>
struct GetRet<R(T::*)(Args...)> {
  using type = R;
};
template <typename T, typename R, typename... Args>
struct GetRet<R(T::*)(Args...) const> {
  using type = R;
};

template <typename T>
using GetRet_t = typename GetRet<T>::type;

// 测试用例
```

```
int f() {return 0;}
struct T1 {
  int m();
};

struct T2 {
  int operator()();
};

void Demo() {
// 函数类型
  GetRet_t<decltype(f)> a;
  // 函数指针类型
  GetRet_t<decltype(&f)> b;
  // 仿函数类型
  GetRet_t<T2> c;
  // lambda类型
  GetRet_t<decltype([]()->int{return 0;})> d;
  // 非静态成员函数类型
  GetRet_t<decltype(&T1::m)> e;
  // 函数对象类型
  GetRet_t<std::function<int()>> g;

  std::cout << std::is_same_v<std::decay_t<decltype(a)>, int>; // true
  std::cout << std::is_same_v<std::decay_t<decltype(b)>, int>; // true
  std::cout << std::is_same_v<std::decay_t<decltype(c)>, int>; // true
  std::cout << std::is_same_v<std::decay_t<decltype(d)>, int>; // true
  std::cout << std::is_same_v<std::decay_t<decltype(e)>, int>; // true
  std::cout << std::is_same_v<std::decay_t<decltype(g)>, int>; // true
}
```

这样做确实可以解决问题，但是有点太「老实巴交」了，踏踏实实去适配所有情况肯定是比较保守的做法，只不过对于当前这个需求，我们还有一种更简单的做法：

```
template <typename T>
struct GetRet {
  // 直接invoke这个类型的成员，推导返回值
  using type = decltype(std::declval<std::decay_t<T>>()());
};

// 对于非静态成员函数类型（这个还是要单独适配）
template <typename T, typename R, typename... Args>
struct GetRet<R(T::*)(Args...)> {
```

```
  using type = R;
};
template <typename T, typename R, typename... Args>
struct GetRet<R(T::*)(Args...) const> {
  using type = R;
};
```

这里需要解释一下declval的功能。在类型变换中，我们是没有实际数据的，换句话说，不会真的去定义一个T类型的变量，而仅仅是需要它来参与类型的变换。但假如这时T不含有无参构造的话，就会失败：

```
struct Test {
  Test(int); // 没有无参构造
  void method();
};

template <typename T>
struct XXX {
  using type = decltype(T{}.method()); // 这里构造T{}的时候会失败
};
```

这里我们为了调用非静态成员函数method，就不得不构造一个T类型的对象，但这个对象只在静态分析的时候才有，并不需要真的构造，所以它的构造函数是怎么样的不重要，但上面这种情况又会让编译器去匹配构造函数的问题，从而产生错误。

为了解决这个「仅需要类型，而不想做构造检测」的问题，我们只能想其他的方法，例如这样：

```
template <typename T>
struct XXX {
  // 通过指针转换，避开构造检测机制
  using type = decltype((static_cast<T *>(nullptr))->method());
};
```

而STL中就提供了declval工具，用来避开构造检测而生成一个纯类型的对象，实现如下：

```
template <typename T>
T declval() {
// 不需要实现，因为它不会真正被调用，仅仅用于静态处理获取类型
}
```

所以，刚才的例子我们就可以改写成：

```
struct Test {
  Test(int); // 没有无参构造
```

```
  void method();
};

template <typename T>
struct XXX {
  using type = decltype(declval<T>().method()); // 这样就不会构造失败了
};
```

再回头看一开始的例子：

```
template <typename T>
struct GetRet {
  // 直接invoke这个类型的成员，推导返回值
  using type = decltype(std::declval<std::decay_t<T>>()());
};
```

前面`std::declval<std::decay_t<T>>()`用于生成这个纯类型的对象，再后面一个`()`就表示invoke行为（对于函数、函数指针会进行调用；对于仿函数、lambda、函数对象类型则会调用其`operator()`函数），再对调用行为进行一次`decltype`即可获取到返回值类型。

那，如果我们想获取参数类型呢？比如说，我想获取函数的第二个参数的类型，要怎么做？这时，由于不像返回值那样可以直接invoke来判断，参数类型的获取就只能用传统的方法了，效果如下：

```
template <typename T>
struct Get2ndArg {
 private:
  using DT = std::decay_t<T>;
 public:
  // 如果内部含有operator()就取它的类型
  using type = typename Get2ndArg<decltype(&DT::operator())>::type;
};

template <typename R, typename Arg1, typename Arg2, typename... Args>
struct Get2ndArg<R(Arg1, Arg2, Args...)> {
  using type = Arg2;
};

template <typename R, typename Arg1, typename Arg2, typename... Args>
struct Get2ndArg<R(*)(Arg1, Arg2, Args...)> {
  using type = Arg2;
};

template <typename T, typename R, typename Arg1, typename Arg2, typename...
Args>
```

```
struct Get2ndArg<R(T::*)(Arg1, Arg2, Args...)> {
  using type = Arg2;
};

template <typename T, typename R, typename Arg1, typename Arg2, typename...
Args>
struct Get2ndArg<R(T::*)(Arg1, Arg2, Args...) const> {
  using type = Arg2;
};

template <typename T>
using Get2ndArg_t = typename Get2ndArg<T>::type;
```


如果要获取第一个参数，或者第N个参数的类型，那么都是相同的道理，这里就不再啰嗦了。

自定义类型的处理

单一的类型处理我们已经体验过了，那对于复杂类型呢？比如说，判断类型`T`中是否存在一个名为`f`的成员？

这个需求里，最主要的思路就是，我们要「尝试」来推导一下`T::f`的类型，如果这个东西存在，那么就能够推导出来（尽管推导出来的类型是什么我们并不关心）；而如果不存在`T::f`这个东西，那么就会实例化失败，从而触发SFINAE继续匹配通用模板。

判断T是否含有一个类型为int的成员f

首先我们先来简化一下问题，这里我们要求`T::f`必须是`int`类型的静态成员，那么代码如下：

```
template <typename T, typename V = int> // V是辅助参数
struct HasMemberF : std::false_type {}; // 判断T中是否函数f成员

template <typename T>
struct HasMemberF<T, decltype(T::f)> : std::true_type {};

template <typename T>
constexpr inline bool HasMemberF_v = HasMemberF<T>::value;

// 以下是Demo
struct T1 {
  static int f; // 符合条件
};

struct T2 {
```

```
 static double f; // 含有f但类型不匹配
};

struct T3 {}; // 不含f

void Demo() {
  std::cout << HasMemberF_v<T1> << std::endl; // 1
  std::cout << HasMemberF_v<T2> << std::endl; // 0
  std::cout << HasMemberF_v<T3> << std::endl; // 0
  std::cout << HasMemberF_v<int> << std::endl; // 0
}
```


着重解释一下这里的写法，首先，通用模板中含有2个参数，但是第二个参数V是用做辅助参数的，我们给它默认参数为int。之后，其实我们是要给HasMemberF<T, int>来绑定true_type的。但仅当T::f的类型是int时才生效。

例如，当传入的参数是T1时，decltype(T1::f)是int，因此，下面的偏特化便存在，也就是模板会变成：

```
template <typename T, typename V = int>
struct HasMemberF : std::false_type {};

template <typename T>
struct HasMemberF<T, int> : std::true_type {}; // 当T是T1时，decltype(T::f)变成int，所以偏特化存在
```

而又因为通用模板中，V默认就是int，所以HasMemberF<T1>就是HasMemberF<T1, int>，显然是命中了偏特化的，因此value为true。

而当用```T2``来实例化时，下面的偏特化也存在，但是变成了：

```
template <typename T, typename V = int>
struct HasMemberF : std::false_type {};

template <typename T>
struct HasMemberF<T, double> : std::true_type {}; // 当T是T2时，
decltype(T::f)变成double，所以偏特化存在
```

而又因为默认参数的存在，HasMemberF<T2>就是HasMemberF<T2, int>，并没有命中偏特化，所以value为false。

再来看T3的情况，由于T3不存在成员f，所以decltype(T3::f)就成为了不合法语句，根据SFINAE原则，这里的偏特化也就不会生成代码。所以仅仅存在一个通用模板，那么HasMemberF<T3>自然会命中通用模板，其value是false。HasMemberF<int>跟HasMemberF<T3>是相同的道理。

那如果不要求T::f是静态的，只要是int类型成员就符合呢？道理是相同的，只不过由于非静态成员不能直接通过类型来取出（T::f语句不合法），因此只能换一种方式，通过declval来取出（std::declval<T>().f）。下面是代码：

```
template <typename T, typename V = int>
struct HasMemberF : std::false_type {};

template <typename T>
struct HasMemberF<T, decltype(std::declval<T>().f)> : std::true_type {}; // 这里用std::declval<T>().f代替T::f

template <typename T>
constexpr inline bool HasMemberF_v = HasMemberF<T>::value;

// 以下是Demo
struct T1 {
  static int f; // 符合条件，静态int成员
};

struct T2 {
  int f; // 符合条件，非静态int成员
};

struct T3 {}; // 不含f

void Demo() {
  std::cout << HasMemberF_v<T1> << std::endl; // 1
  std::cout << HasMemberF_v<T2> << std::endl; // 1
  std::cout << HasMemberF_v<T3> << std::endl; // 0
}
```

这样，无论静态还是非静态，只要是int类型就符合。

但如果我要求只想筛选出非静态的怎么办？那就只能通过取地址的方式拿出成员变量了（&T::f取出成员的指针类型），代码如下：

```
template <typename T, typename V = int T::*> // 注意这里改成成员指针类型
struct HasMemberF : std::false_type {};
```

```
template <typename T>
struct HasMemberF<T, decltype(&T::f)> : std::true_type {};

template <typename T>
constexpr inline bool HasMemberF_v = HasMemberF<T>::value;

// 以下是Demo
struct T1 {
  static int f; // 不符合条件，静态int成员
};

struct T2 {
  int f; // 符合条件，非静态int成员
};

struct T3 {}; // 不含f

void Demo() {
  std::cout << HasMemberF_v<T1> << std::endl; // 0
  std::cout << HasMemberF_v<T2> << std::endl; // 1
  std::cout << HasMemberF_v<T3> << std::endl; // 0
}
```


判断T中是否含有成员f

确定类型的成员判断我们已经会了，那此时如果需求变为「判断类型T中是否含有成员f」呢？换句话说，现在我们不关心f是什么类型了，变量也好，函数也好，静态也好，非静态也好，只要它里面有一个叫f的成员就算数。这种的怎么做呢？

其实思路还是没变的，只不过我们要对推导出的类型来做多一步处理了。上一节中，通用模板的辅助参数的默认值要符合偏特化当中生成的，才能让使用时命中偏特化。但现在既然对类型不关心了，那么也就是说，**无论decltype出什么类型，我们都要给它转换成一个相同的类型X**，然后让通用模板的第二个参数默认值设定为X即可。示例如下：

```
struct X {}; // 辅助类型，仅用作静态推导，无运行期意义

// 辅助工具，用于把任意类型转为X
template <typename T>
using ToX = X;

template <typename T, typename V = X> // 注意这里改成X
struct HasMemberF : std::false_type {};
```

```
template <typename T>
struct HasMemberF<T, ToX<decltype(&T::f)>> : std::true_type {};

template <typename T>
constexpr inline bool HasMemberF_v = HasMemberF<T>::value;

// 以下是Demo
struct T1 {
  static int f; // 符合条件，含有f
};

struct T2 {
  double f; // 符合条件，含有f
};

struct T3 {}; // 不含f

struct T4 {
  int f() const; // 符合条件，含有f
  void f2();
};

void Demo() {
  std::cout << HasMemberF_v<T1> << std::endl; // 1
  std::cout << HasMemberF_v<T2> << std::endl; // 1
  std::cout << HasMemberF_v<T3> << std::endl; // 0
  std::cout << HasMemberF_v<T4> << std::endl; // 0
}
```

到这里大家应该能发现，其实这个辅助类型X是什么并不重要，只要**通用模板的第二个参数的默认值**和**偏特化中映射出的类型**相匹配即可完成功能。比如说我们可以把X改成int：

```
// 辅助工具，用于把任意类型转为int
template <typename T>
using ToInt = int;

template <typename T, typename V = int> // 注意这里改成int
struct HasMemberF : std::false_type {};

template <typename T>
struct HasMemberF<T, ToInt<decltype(&T::f)>> : std::true_type {};
```

```
template <typename T>
constexpr inline bool HasMemberF_v = HasMemberF<T>::value;
```

或者改成void也是一样的效果：

```
// 辅助工具，用于把任意类型转为void
template <typename T>
using ToVoid = void;

template <typename T, typename V = void> // 注意这里改成void
struct HasMemberF : std::false_type {};

template <typename T>
struct HasMemberF<T, ToVoid<decltype(&T::f)>> : std::true_type {};

template <typename T>
constexpr inline bool HasMemberF_v = HasMemberF<T>::value;
```

而STL中，提供了一个工具叫void_t，用于把任意类型（任意个数的任意类型）转换为void类型，实现如下：

```
template <typename... Args>
using void_t = void;
```

因此，我们这里就可以直接用void_t来代替自定义繁琐的工具，所以这个功能的最终版本是这样的：

```
template <typename T, typename V = void>
struct HasMemberF : std::false_type {};

template <typename T>
struct HasMemberF<T, std::void_t<decltype(&T::f)>> : std::true_type {};

template <typename T>
constexpr inline bool HasMemberF_v = HasMemberF<T>::value;
```

这就是用于判断某个类型中是否函数某个成员的模板元的编写方法。

小结

这一篇我们介绍了一些更加高级的模板元编程技巧，后面还引出了void_t工具的使用方法。下一篇将会继续介绍其他的模板元编程技巧。