A.6 变参模板

变参模板：就是可以使用不定数量的参数进行特化的模板。就像你接触到的变参函数一样，printf 就接受可变参数。现在，就可以给你的模板指定不定数量的参数了。变参模板在整个 C++ 线程库中都有使用，例如： std::thread 的构造函数就是一个变参类模板。从使用者的角度看，仅知道模板可以接受无限个参数就够了，不过当要写这么一个模板或对其工作原理很感兴趣时，就需要了解一些细节。

和变参函数一样，变参部分可以在参数列表章使用省略号 ... 代表，变参模板需要在参数列表中使用省略号：

```
template<typename ... ParameterPack>
class my_template
{};
```

即使主模板不是变参模板，模板进行部分特化的类中，也可以使用可变参数模板。例如， std::packaged_task<> (见4.2.1节)的主模板就是一个简单的模板，这个简单的模板只有一个参数：

```
template<typename FunctionType>
class packaged_task;
```

不过，并不是所有地方都这样定义；对于部分特化模板来说，其就像是一个“占位符”：

```
template<typename ReturnType,typename ... Args>
class packaged_task<ReturnType(Args...)>;
```

部分特化的类就包含实际定义的类；在第4章，可以写一个 std::packaged_task<int(std::string,double)> 来声明一个以 std::string 和double作为参数的任务，当执行这个任务后结果会由 std::future<int> 进行保存。

声明展示了两个变参模板的附加特性。第一个比较简单：普通模板参数(例如ReturnType)和可变模板参数(Args)可以同时声明。第二个特性，展示了 Args... 特化类的模板参数列表中如何使用，为了展示实例化模板中的Args的组成类型。实际上，因为这是部分特化，所以其作为一种模式进

行匹配；在列表中出现的类型(被Args捕获)都会进行实例化。参数包(parameter pack)调用可变参数Args，并且使用 `Args...` 作为包的扩展。

和可变参函数一样，变参部分可能什么都没有，也可能有很多类型项。例如，`std::packaged_task<my_class()>` 中ReturnType参数就是my_class，并且Args参数包是空的，不过 `std::packaged_task<void(int,double,my_class&,std::string*)>` 中，ReturnType为void，并且Args列表中的类型就有：int, double, my_class&和std::string*。

A.6.1 扩展参数包

变参模板主要依靠包括扩展功能，因为不能限制有更多的类型添加到模板参数中。首先，列表中的参数类型使用到的时候，可以使用包扩展，比如：需要给其他模板提供类型参数。

```
template<typename ... Params>
struct dummy
{
  std::tuple<Params...> data;
};
```

成员变量data是一个 `std::tuple<>` 实例，包含所有指定类型，所以dummy的成员变量就为 `std::tuple<int, double, char>` 。

可以将包扩展和普通类型相结合：

```
template<typename ... Params>
struct dummy2
{
  std::tuple<std::string,Params...> data;
};
```

这次，元组中添加了额外的(第一个)成员类型 `std::string` 。其优雅指出在于，可以通过包扩展的方式创建一种模式，这种模式会在之后将每个元素拷贝到扩展之中，可以使用 `...` 来表示扩展模式的结束。

例如，创建使用参数包来创建元组中所有的元素，不如在元组中创建指针，或使用 `std::unique_ptr<>` 指针，指向对应元素：

```
template<typename ... Params>
struct dummy3
{
  std::tuple<Params* ...> pointers;
  std::tuple<std::unique_ptr<Params> ...> unique_pointers;
};
```

类型表达式会比较复杂，提供的参数包是在类型表达式中产生，并且表达式中使用 `...` 作为扩展。当参数包已经扩展，包中的每一项都会代替对应的类型表达式，在结果列表中产生相应的数据项。因此，当参数包Params包含int，int，char类型，那么
`std::tuple<std::pair<std::unique_ptr<Params>,double> ... >` 将扩展为
`std::tuple<std::pair<std::unique_ptr<int>,double>` ，
`std::pair<std::unique_ptr<int>,double>` , `std::pair<std::unique_ptr<char>, double> >` 。如果包扩展被当做模板参数列表使用，那么模板就不需要变长的参数了；如果不需要了，参数包就要对模板参数的要求进行准确的匹配：

```
template<typename ... Types>
struct dummy4
{
  std::pair<Types...> data;
};
dummy4<int,char> a;  // 1 ok，为std::pair<int, char>
dummy4<int> b;  // 2 错误，无第二个类型
dummy4<int,int,int> c;  // 3 错误，类型太多
```

可以使用包扩展的方式，对函数的参数进行声明：

```
template<typename ... Args>
void foo(Args ... args);
```

这将会创建一个新参数包args，其是一组函数参数，而非一组类型，并且这里 `...` 也能像之前一样进行扩展。例如，可以在 `std::thread` 的构造函数中使用，使用右值引用的方式获取函数所有的参数(见A.1节)：

```
template<typename CallableType,typename ... Args>
thread::thread(CallableType&& func,Args&& ... args);
```

函数参数包也可以用来调用其他函数，将制定包扩展成参数列表，匹配调用的函数。如同类型扩展一样，也可以使用某种模式对参数列表进行扩展。

例如，使用 `std::forward()` 以右值引用的方式来保存提供给函数的参数：

```
template<typename ... ArgTypes>
void bar(ArgTypes&& ... args)
{
  foo(std::forward<ArgTypes>(args)...);
}
```

注意一下这个例子，包扩展包括对类型包ArgTypes和函数参数包args的扩展，并且省略了其余的表达式。

当这样调用bar函数：

```
int i;
bar(i,3.141,std::string("hello "));
```

将会扩展为

```
template<>
void bar<int&,double,std::string>(
         int& args_1,
         double&& args_2,
         std::string&& args_3)
{
  foo(std::forward<int&>(args_1),
      std::forward<double>(args_2),
      std::forward<std::string>(args_3));
}
```

这样就将第一个参数以左值引用的形式，正确的传递给了foo函数，其他两个函数都是以右值引用的方式传入的。

最后一件事，参数包中使用 `sizeof...` 操作可以获取类型参数类型的大小， `sizeof...(p)` 就是p参数包中所包含元素的个数。不管是类型参数包或函数参数包，结果都是一样的。这可能是唯一一次在使用参数包的时候，没有加省略号；这里的省略号是作为 `sizeof...` 操作的一部分，所以不算是用到省略号。

下面的函数会返回参数的数量：

```
template<typename ... Args>
unsigned count_args(Args ... args)
{
  return sizeof... (Args);
}
```

就像普通的sizeof操作一样， `sizeof...` 的结果为常量表达式，所以其可以用来指定定义数组长度，等等。