


qicosmos

一点梦想：尽自己一份力，让c++的世界变得更美好！顶级c++社区欢迎你：

<http://purecpp.org/> 社区公众号purecpp 欢迎关注学习最新的c++知识

随笔 - 95, 文章 - 0, 评论 - 531, 引用 - 0

导航

[博客园](#)[首页](#)[新随笔](#)[联系](#)[订阅](#) [管理](#)

< 2020年5月 >						
日	一	二	三	四	五	六
26	27	28	29	30	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6

公告

昵称： qicosmos(江南)

园龄： 7年6个月

粉丝： 456

关注： 0

+加关注

搜索

<input type="text"/>	<input type="button" value="找找看"/>
<input type="text"/>	<input type="button" value="谷歌搜索"/>

常用链接

[我的随笔](#)[我的评论](#)[我的参与](#)[最新评论](#)[我的标签](#)

积分与排名

积分 - 234526

排名 - 1994

随笔分类

[boost\(8\)](#)[C++ \(88\)](#)[c++11 使用c++11改进我们的程序系列\(43\)](#)[C++11工程实践\(27\)](#)

泛化之美--C++11可变模版参数的妙用

1概述

C++11的新特性--可变模版参数（variadic templates）是C++11新增的最强大的特性之一，它对参数进行了高度泛化，它能表示0到任意个数、任意类型的参数。相比C++98/03，类模版和函数模版中只能含固定数量的模版参数，可变模版参数无疑是一个巨大的改进。然而由于可变模版参数比较抽象，使用起来需要一定的技巧，所以它也是C++11中最难理解和掌握的特性之一。虽然掌握可变模版参数有一定难度，但是它却是C++11中最有意思的一个特性，本文希望带领读者由浅入深的认识和掌握这一特性，同时也会通过一些实例来展示可变参数模版的一些用法。

2可变模版参数的展开

可变参数模版和普通模版的语义是一样的，只是写法上稍有区别，声明可变参数模版时需要在typename或class后面带上省略号“...”。比如我们常常这样声明一个可变模版参数：template<typename...>或者template<class...>，一个典型的可变模版参数的定义是这样的：

```
template <class... T>
void f(T... args);
```

上面的可变模版参数的定义当中，省略号的作用有两个：

- 1.声明一个参数包T... args，这个参数包中可以包含0到任意个模版参数；
- 2.在模版定义的右边，可以将参数包展开成一个一个独立的参数。

上面的参数args前面有省略号，所以它就是一个可变模版参数，我们把带省略号的参数称为“参数包”，它里面包含了0到N（N>=0）个模版参数。我们无法直接获取参数包args中的每个参数的，只能通过展开参数包的方式来获取参数包中的每个参数，这是使用可变模版参数的一个主要特点，也是最大的难点，即如何展开可变模版参数。

可变模版参数和普通的模版参数语义是一致的，所以可以应用于函数和类，即可变模版参数函数和可变模版参数类，然

代码质量(4)
设计开发(5)
设计模式(10)

随笔档案

2019年1月(1)
2018年11月(1)
2016年8月(2)
2016年3月(1)
2015年12月(2)
2015年11月(4)
2015年10月(2)
2015年9月(1)
2015年8月(5)
2015年6月(2)
2015年5月(7)
2015年4月(1)
2015年3月(2)
2015年2月(2)
2014年11月(1)
2014年9月(1)
2014年8月(1)
2014年7月(3)
2014年6月(1)
2014年5月(3)
2014年4月(4)
2014年3月(3)
2014年2月(3)
2014年1月(4)
2013年12月(4)
2013年11月(3)
2013年10月(6)
2013年9月(7)
2013年8月(5)
2013年7月(1)
2013年6月(5)
2013年5月(2)
2013年4月(5)

最新评论

1. Re:C++11实现一个轻量级的AOP框架
请问, enum{ value = std::is_same<decltype(Check<T>)(0)), std::true_type>value }; 能设法用 std::integral_constant替代...

--Haining00

2. Re:thinking in object pool
你好! 后来花时间琢磨了一下, 对象池的代码, 需要在

而, 模版函数不支持偏特化, 所以可变模版参数函数和可变模版参数类展开可变模版参数的方法还不尽相同, 下面我们来分别看看他们展开可变模版参数的方法。

2.1可变模版参数函数

一个简单的可变模版参数函数:



```
template <class... T>
void f(T... args)
{
    cout << sizeof...(args) << endl; //打印变参的个数
}

f();           //0
f(1, 2);       //2
f(1, 2.5, ""); //3
```



上面的例子中, f()没有传入参数, 所以参数包为空, 输出的size为0, 后面两次调用分别传入两个和三个参数, 故输出的size分别为2和3。由于可变模版参数的类型和个数是不固定的, 所以我们可以传任意类型和个数的参数给函数f。这个例子只是简单的将可变模版参数的个数打印出来, 如果我们需要将参数包中的每个参数打印出来的话就需要通过一些方法了。展开可变模版参数函数的方法一般有两种: 一种是通过递归函数来展开参数包, 另外一种是通过逗号表达式来展开参数包。下面来看看如何用这两种方法来展开参数包。

2.1.1递归函数方式展开参数包

通过递归函数展开参数包, 需要提供一个参数包展开的函数和一个递归终止函数, 递归终止函数正是用来终止递归的, 来看看下面的例子。



```
#include <iostream>
using namespace std;
//递归终止函数
void print()
{
    cout << "empty" << endl;
}
//展开函数
template <class T, class ...Args>
void print(T head, Args... rest)
```

对象池的析构函数中，增加一个标志记录是否已经被析构，如果已经被析构，就不再增加到map中，另外，在程序运行的过程中，需要将map中的对象都Get一遍，...

--fan6662000

3. Re:thinking in object pool

你好！我购买了你的关于C11的书，深入应用C 11。关于书书中的对象池的代码，分别在VS2015和CENTOS7下，编译运行了一下。发现几个奇怪的现象。一，VS2015 1，VS2015下，...

--fan6662000

4. Re:新书《深入应用C++11：代码优化与工程级应用》出版，感谢支持
感谢C++学习路上的引领者！

--一花一世界，一叶一乾坤

5. Re: (原创) C++ 同步队列

你好 请教下文中的stop标志位是什么作用？取存线程的状态位吗？

--EI_北林

阅读排行榜

1. 泛化之美--C++11可变模版参数的妙用(57133)
2. (原创) 用C++11的std::async代替线程的创建(36016)
3. C++11模版元编程(33255)
4. C++ REST SDK的基本用法(32534)
5. 从4行代码看右值引用(30186)

评论排行榜

1. 新书《深入应用C++11：代码优化与工程级应用》出版，感谢支持(60)
2. 《深入应用C++11：代码优化与工程级应用》勘误表(33)
3. 一个更好的C++序列化/反序列化库Kapok(27)
4. 一点项目经验的总结(25)

```
{
    cout << "parameter " << head << endl;
    print(rest...);
}
```

```
int main(void)
{
    print(1,2,3,4);
    return 0;
}
```



上例会输出每一个参数，直到为空时输出empty。展开参数包的函数有两个，一个是递归函数，另外一个为递归终止函数，参数包Args...在展开的过程中递归调用自己，每调用一次参数包中的参数就会少一个，直到所有的参数都展开为止，当没有参数时，则调用非模板函数print终止递归过程。

递归调用的过程是这样的：

```
print(1,2,3,4);
print(2,3,4);
print(3,4);
print(4);
print();
```

上面的递归终止函数还可以写成这样：

```
template <class T>
void print(T t)
{
    cout << t << endl;
}
```

修改递归终止函数后，上例中的调用过程是这样的：

```
print(1,2,3,4);
print(2,3,4);
print(3,4);
print(4);
```

当参数包展开到最后一个参数时递归为止。再看一个通过可变模版参数求和的例子：



```
template<typename T>
T sum(T t)
{
    return t;
}

template<typename T, typename ... Types>
```

5. 泛化之美--C++11可变模版参数的妙用(20)

推荐排行榜

1. (原创) 用C++11的std::async代替线程的创建(14)
2. 从4行代码看右值引用(14)
3. (原创) 谈谈架构师的职责(一) (13)
4. 泛化之美--C++11可变模版参数的妙用(11)
5. 一点项目经验的总结(11)

```
T sum (T first, Types ... rest)
{
    return first + sum<T>(rest...);
}
```

```
sum(1,2,3,4); //10
```



sum在展开参数包的过程中将各个参数相加求和，参数的展开方式和前面的打印参数包的方式是一样的。

2.1.2逗号表达式展开参数包

递归函数展开参数包是一种标准做法，也比较好理解，但也有一个缺点,就是必须要一个重载的递归终止函数，即必须要有一个同名的终止函数来终止递归，这样可能会感觉稍有不便。有没有一种更简单的方式呢？其实还有一种方法可以不通过递归方式来展开参数包，这种方式需要借助逗号表达式和初始化列表。比如前面print的例子可以改成这样：



```
template <class T>
void printarg(T t)
{
    cout << t << endl;
}

template <class ...Args>
void expand(Args... args)
{
    int arr[] = {(printarg(args), 0)...};
}

expand(1,2,3,4);
```



这个例子将分别打印出1,2,3,4四个数字。这种展开参数包的方式，不需要通过递归终止函数，是直接在expand函数体中展开的, printarg不是一个递归终止函数，只是一个处理参数包中每一个参数的函数。这种就地展开参数包的方式实现的关键是逗号表达式。我们知道逗号表达式会按顺序执行逗号前面的表达式，比如：

```
d = (a = b, c);
```

这个表达式会按顺序执行：b会先赋值给a，接着括号中的逗号表达式返回c的值，因此d将等于c。expand函数中的逗号表达式：(printarg(args), 0)，也是按照这个执行顺序，先执行printarg(args)，再得到逗号表达式的结果0。同时还用到了C++11的另外一个特性——初始化列表，通过

初始化列表来初始化一个变长数组, {(printarg(args), 0)...}将会展开成

((printarg(arg1),0), (printarg(arg2),0), (printarg(arg3),0), etc ...), 最终会创建一个元素值都为0的数组int arr[sizeof...

(Args)]. 由于是逗号表达式, 在创建数组的过程中会先执行逗号表达式前面的部分printarg(args)打印出参数, 也就是说在构造int数组的过程中就将参数包展开了, 这个数组的目的纯粹是为了在数组构造的过程展开参数包。我们可以把上面的例子再进一步改进一下, 将函数作为参数, 就可以支持lambda表达式了, 从而可以少写一个递归终止函数了, 具体代码如下:



```
template<class F, class... Args>void expand(const
F& f, Args&&...args)
{
    //这里用到了完美转发, 关于完美转发, 读者可以参考笔者在上
    一期程序员中的文章《通过4行代码看右值引用》
    initializer_list<int>{(f(std::forward< Args>
    (args)),0)...};
}
expand([](int i){cout<<i<<endl;}, 1,2,3);
```



上面的例子将打印出每个参数, 这里如果再使用C++14的新特性泛型lambda表达式的话, 可以写更泛化的lambda表达式了:

```
expand([](auto i){cout<<i<<endl;}, 1,2.0,"test");
```

2.2可变模版参数类

可变参数模板类是一个带可变模板参数的模板类, 比如C++11中的元祖std::tuple就是一个可变模板类, 它的定义如下:

```
template< class... Types >
class tuple;
```

这个可变参数模板类可以携带任意类型任意个数的模板参数:

```
std::tuple<int> tp1 = std::make_tuple(1);
std::tuple<int, double> tp2 = std::make_tuple(1,
2.5);
std::tuple<int, double, string> tp3 =
std::make_tuple(1, 2.5, "");
```

可变参数模板的模板参数个数可以为0个, 所以下面的定义也是合法的:

```
std::tuple<> tp;
```

可变参数模板类的参数包展开的方式和可变参数模板函数的展开方式不同，可变参数模板类的参数包展开需要通过模板特化和继承方式去展开，展开方式比可变参数模板函数要复杂。下面我们来看一下展开可变模版参数类中的参数包的方法。

2.2.1模版偏特化和递归方式来展开参数包

可变参数模板类的展开一般需要定义两到三个类，包括类声明和偏特化的模板类。如下方式定义了一个基本的可变参数模板类：



```
//前向声明
template<typename... Args>
struct Sum;

//基本定义
template<typename First, typename... Rest>
struct Sum<First, Rest...>
{
    enum { value = Sum<First>::value +
Sum<Rest...>::value };
};

//递归终止
template<typename Last>
struct Sum<Last>
{
    enum { value = sizeof (Last) };
};
```



这个Sum类的作用是在编译期计算出参数包中参数类型的size之和，通过sum<int,double,short>::value就可以获取这3个类型的size之和为14。这是一个简单的通过可变参数模板类计算的例子，可以看到一个基本的可变参数模板应用类由三部分组成，第一部分是：

```
template<typename... Args> struct sum
```

它是前向声明，声明这个sum类是一个可变参数模板类；第二部分是类的定义：

```
template<typename First, typename... Rest>
struct Sum<First, Rest...>
{
    enum { value = Sum<First>::value +
```

```
Sum<Rest...>::value };
};
```

它定义了一个部分展开的可变模参数模板类，告诉编译器如何递归展开参数包。第三部分是特化的递归终止类：

```
template<typename Last> struct sum<last>
{
    enum { value = sizeof (First) };
}
```

通过这个特化的类来终止递归：

```
template<typename First, typename... Args>struct
sum;
```

这个前向声明要求sum的模板参数至少有一个，因为可变参数模板中的模板参数可以有0个，有时候0个模板参数没有意义，就可以通过上面的声明方式来限定模板参数不能为0个。上面的这种三段式的定义也可以改为两段式的，可以将前向声明去掉，这样定义：



```
template<typename First, typename... Rest>
struct Sum
{
    enum { value = Sum<First>::value +
Sum<Rest...>::value };
};

template<typename Last>
struct Sum<Last>
{
    enum{ value = sizeof(Last) };
};
```



上面的方式只要一个基本的模板类定义和一个特化的终止函数就行了，而且限定了模板参数至少有一个。

递归终止模板类可以有多种写法，比如上例的递归终止模板类还可以这样写：



```
template<typename... Args> struct sum;
template<typename First, typenameLast>
struct sum<First, Last>
```

```
{
    enum{ value = sizeof(First) +sizeof( Last) };
};
```



在展开到最后两个参数时终止。

还可以在展开到0个参数时终止：

```
template<>struct sum<> { enum{ value = 0 }; };
```

还可以使用std::integral_constant来消除枚举定义value。利用std::integral_constant可以获得编译期常量的特性，可以将前面的sum例子改为这样：



```
//前向声明
template<typename First, typename... Args>
struct Sum;

//基本定义
template<typename First, typename... Rest>
struct Sum<First, Rest...> :
    std::integral_constant<int, Sum<First>::value +
    Sum<Rest...>::value>
{
};

//递归终止
template<typename Last>
struct Sum<Last> : std::integral_constant<int,
sizeof( Last)>
{
};

sum<int, double, short>::value; //值为14
```



2.2.2继承方式展开参数包

还可以通过继承方式来展开参数包，比如下面的例子就是通过继承的方式去展开参数包：



```
// 整型序列的定义
template<int...>
struct IndexSeq{};

//继承方式，开始展开参数包
```



```

template<int N, int... Indexes>
struct MakeIndexes : MakeIndexes<N - 1, N - 1,
Indexes...> {};

// 模板特化, 终止展开参数包的条件
template<int... Indexes>
struct MakeIndexes<0, Indexes...>
{
    typedef IndexSeq<Indexes...> type;
};

int main()
{
    using T = MakeIndexes<3>::type;
    cout << typeid(T).name() << endl;
    return 0;
}

```

其中MakeIndexes的作用是为了生成一个可变参数模板类的整数序列, 最终输出的类型是: struct IndexSeq<0,1,2>。

MakeIndexes继承于自身的一个特化的模板类, 这个特化的模板类同时也在展开参数包, 这个展开过程是通过继承发起的, 直到遇到特化的终止条件展开过程才结束。

MakeIndexes<1,2,3>::type的展开过程是这样的:

```

MakeIndexes<3> : MakeIndexes<2, 2>{}
MakeIndexes<2, 2> : MakeIndexes<1, 1, 2>{}
MakeIndexes<1, 1, 2> : MakeIndexes<0, 0, 1, 2>
{
    typedef IndexSeq<0, 1, 2> type;
}

```

通过不断的继承递归调用, 最终得到整型序列 IndexSeq<0, 1, 2>。

如果不希望通过继承方式去生成整形序列, 则可以通过下面的方式生成。

```

template<int N, int... Indexes>
struct MakeIndexes3
{
    using type = typename MakeIndexes3<N - 1, N - 1, Indexes...>::type;
};

```

```
template<int... Indexes>
struct MakeIndexes3<0, Indexes...>
{
    typedef IndexSeq<Indexes...> type;
};
```



我们看到了如何利用递归以及偏特化等方法来展开可变模版参数，那么实际当中我们会怎么去使用它呢？我们可以用可变模版参数来消除一些重复的代码以及实现一些高级功能，下面我们来看看可变模版参数的一些应用。

3可变参数模版消除重复代码

C++11之前如果要写一个泛化的工厂函数，这个工厂函数能接受任意类型的入参，并且参数个数要能满足大部分的应用需求的话，我们不得不定义很多重复的模版定义，比如下面的代码：

[View Code](#)

可以看到这个泛型工厂函数存在大量的重复的模板定义，并且限定了模板参数。用可变模板参数可以消除重复，同时去掉参数个数的限制，代码很简洁，通过可变参数模版优化后的工厂函数如下：



```
template<typename... Args>
T* Instance(Args&&... args)
{
    return new T(std::forward<Args>(args)...);
}

A* pa = Instance<A>(1);
B* pb = Instance<B>(1, 2);
```



4可变参数模版实现泛化的delegate

C++中没有类似C#的委托，我们可以借助可变模版参数来实现一个。C#中的委托的基本用法是这样的：



```
delegate int AggregateDelegate(int x, int y); //声明委托类型

int Add(int x, int y){return x+y;}
int Sub(int x, int y){return x-y;}

AggregateDelegate add = Add;
```

```
add(1,2); //调用委托对象求和
AggregateDelegate sub = Sub;
sub(2,1); // 调用委托对象相减
```



C#中的委托的使用需要先定义一个委托类型，这个委托类型不能泛化，即委托类型一旦声明之后就不能再用来接受其它类型的函数了，比如这样用：

```
int Fun(int x, int y, int z){return x+y+z;}
int Fun1(string s, string r){return
s.Length+r.Length; }
AggregateDelegate fun = Fun; //编译报错，只能赋值相同类型的函数
AggregateDelegate fun1 = Fun1; //编译报错，参数类型不匹配
```

这里不能泛化的原因是声明委托类型的时候就限定了参数类型和个数，在C++11里不存在这个问题了，因为有了可变模版参数，它就代表了任意类型和个数的参数了，下面让我们来看一下如何实现一个功能更加泛化的C++版本的委托（这里为了简单起见只处理成员函数的情况，并且忽略const、volatile和const volatile成员函数的处理）。



```
template <class T, class R, typename... Args>
class MyDelegate
{
public:
    MyDelegate(T* t, R (T::*f) (Args...))
    :m_t(t),m_f(f) {}

    R operator() (Args&&... args)
    {
        return (m_t->*m_f) (std::forward<Args>
(args) ...);
    }

private:
    T* m_t;
    R (T::*m_f) (Args...);
};

template <class T, class R, typename... Args>
MyDelegate<T, R, Args...> CreateDelegate(T* t, R
(T::*f) (Args...))
{
    return MyDelegate<T, R, Args...>(t, f);
}
```

```
}

struct A
{
    void Fun(int i) {cout<<i<<endl;}
    void Fun1(int i, double j) {cout<<i+j<<endl;}
};

int main()
{
    A a;
    auto d = CreateDelegate(&a, &A::Fun); //创建委托
    d(1); //调用委托, 将输出1
    auto d1 = CreateDelegate(&a, &A::Fun1); //创建委托
    d1(1, 2.5); //调用委托, 将输出3.5
}
```



MyDelegate实现的关键是内部定义了一个能接受任意类型和个数参数的“万能函数”：R (T::*m_f)(Args...)，正是由于可变模版参数的特性，所以我们才能够让这个m_f接受任意参数。

5总结

使用可变模版参数的这些技巧相信读者看了会有耳目一新之感，使用可变模版参数的关键是如何展开参数包，展开参数包的过程是很精妙的，体现了泛化之美、递归之美，正是因为它具有神奇的“魔力”，所以我们可以更泛化的去处理问题，比如用它来消除重复的模版定义，用它来定义一个能接受任意参数的“万能函数”等。其实，可变模版参数的作用远不止文中列举的那些作用，它还可以和其它C++11特性结合起来，比如type_traits、std::tuple等特性，发挥更加强大的威力，将在后面模板元编程的应用中介绍。

本文曾发表于《程序员》2015年2月刊。转载请注明出处。

后记：本文的内容主要来自于我在公司内部培训的一次课程，因为很多人对C++11可变模板参数搞不清或者理解得不深入，所以我觉得有必要拿出来分享一下，让更多的人看到，就整理了一下发到程序员杂志了，我相信读者看完之后对可变模板参数会有全面深入的了解。

一点梦想：尽自己一份力，让c++的世界变得更美好！