**Boost::bind使用详解**

**阅读目录(Content)**

* [1.Boost::bind](https://www.cnblogs.com/blueoverflow/p/4740093.html#_label0)
* [2.bind的工作原理](https://www.cnblogs.com/blueoverflow/p/4740093.html#_label1)
* [3.常用的函数对象工具](https://www.cnblogs.com/blueoverflow/p/4740093.html#_label2)
* [4.bind与其他函数对象工具区别](https://www.cnblogs.com/blueoverflow/p/4740093.html#_label3)
  + [4.1 区别与mem\_fun和fun\_ptr](https://www.cnblogs.com/blueoverflow/p/4740093.html#_lab2_3_0)
  + [4.2 区别与bind1st和bind2st](https://www.cnblogs.com/blueoverflow/p/4740093.html#_lab2_3_1)
  + [4.3 区别传ref和传instance](https://www.cnblogs.com/blueoverflow/p/4740093.html#_lab2_3_2)
* [5.bind的应用场景](https://www.cnblogs.com/blueoverflow/p/4740093.html#_label4)
  + [5.1 绑定普通函数](https://www.cnblogs.com/blueoverflow/p/4740093.html#_lab2_4_0)
  + [5.2 bind绑定成员函数](https://www.cnblogs.com/blueoverflow/p/4740093.html#_lab2_4_1)
  + [5.3 bind绑定成员变量](https://www.cnblogs.com/blueoverflow/p/4740093.html#_lab2_4_2)
  + [5.4 绑定函数对象](https://www.cnblogs.com/blueoverflow/p/4740093.html#_lab2_4_3)
  + [5.5 使用ref库](https://www.cnblogs.com/blueoverflow/p/4740093.html#_lab2_4_4)
  + [5.6 存储bind表达式](https://www.cnblogs.com/blueoverflow/p/4740093.html#_lab2_4_5)
  + [5.7 绑定非标准函数](https://www.cnblogs.com/blueoverflow/p/4740093.html#_lab2_4_6)

**1.Boost::bind**

　　在STL中，我们经常需要使用bind1st，bind2st函数绑定器和fun\_ptr,mem\_fun等函数适配器，这些函数绑定器和函数适配器使用起来比较麻烦，需要根据是全局函数还是类的成员函数，是一个参数还是多个参数等做出不同的选择，而且有些情况使用STL提供的不能满足要求，所以如果可以我们最好使用boost提供的bind，它提供了统一的接口，提供了更多的支持，比如说它增加了shared\_ptr，虚函数，类成员的绑定。

**2.bind的工作原理**

　　bind并不是一个单独的类或函数，而是非常庞大的家族，依据绑定的参数的个数和要绑定的调用对象的类型，总共有数十种不同的形式，编译器会根据具体的绑定代码制动确定要使用的正确的形式，bind的基本形式如下：

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 template<class R,class F> bind(F f);

2 template<class R,class F,class A1> bind(F f,A1 a1);

3 namespace

4 {

5 boost::arg<1> \_1;

6 boost::arg<2> \_2;

7 boost::arg<3> \_3;

8 ….. //其他6个占位符

9 };

[复制代码](javascript:void(0);)

**bind**接收的**第一个参数**必须是一个**可调用的对象f**，包括**函数**、**函数指针**、**函数对象**、和**成员函数指针**，之后bind**最多接受9个参数**，**参数数量**必须与**f的参数数量相等**，这些参数被传递给f作为入参。 绑定完成后，**bind**会**返回**一个**函数对象**，它内部**保存了f的拷贝**，具有**operator()**，**返回值类型**被**自动推导**为**f的返回类型**。在发生调用时这个**函数对象**将把之前**存储的参数**转发给**f**完成调用。例如，有一个函数func，它的形式是：

1 func(a1,a2);

那么，他将等价于一个具有无参operator()的bind函数对象调用：

1 bind(func,a1,a2)();

　　这是bind最简单的形式，bind表达式存储了func和a1、a2的拷贝，产生了一个临时函数对象。因为func接收两个参数，而a1和a2的拷贝传递给func完成真正的函数调用。

**bind**的真正威力在于它的**占位符**，它们分别定义为**\_1,\_2,\_3**,**一直到 \_9**,位于一个匿名的名字空间。**占位符**可以取代bind**参数的位置**，在发生**调用时**才**接受真正的参数**。**占位符的名字**表示它在**调用式**中的**顺序**，而在绑定的表达式中没有没有顺序的要求，\_1不一定必须第一个出现，也不一定只出现一次，例如：

1 bind(func,\_2,\_1)(a1,a2);

　　返回一个具有两个参数的函数对象，第一个参数将放在func的第二个位置，而第二个参数则放在第一个位置，调用时等价于：

1 func(a2,a1);

**3.常用的函数对象工具**

(1)**bind1st**,**bind2st**函数绑定器，把**二元函数对象**变为**一元函数对象**。  
(2)**mem\_fun**,把**成员函数**变为**函数对象**。  
(3)**fun\_ptr**，把一般的**全局函数**变为**函数对象**。  
(4)**boost::bind()**,包含了**以上所有的功能**。

**4.bind与其他函数对象工具区别**

**4.1 区别与mem\_fun和fun\_ptr**

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 #include <functional>

2 #include <iostream>

3 #include <string>

4 #include "boost/bind.hpp"

5 class some\_class

6 {

7 public:

8 void print\_string(const std::string& s) const

9 {

10 std::cout << s << '\n';

11 }

12 void print\_classname()

13 {

14 std::cout << "some\_class" << std::endl;

15 }

16 };

17 void print\_string(const std::string s)

18 { std::cout << s << '\n';

19 }

20 void print\_functionname()

21 {

22 std::cout << "Print\_functionname" <<std::endl;

23 }

24 int main()

25 {

26 std::ptr\_fun(&print\_string)("hello1");

27 //std::ptr\_fun<void>(&print\_functionname);

28 some\_class sc0;

29 std::mem\_fun\_ref(&some\_class::print\_classname)(sc0);

30 std::mem\_fun\_ref<void,some\_class>(&some\_class::print\_classname)(sc0);

31 //std::mem\_fun1\_ref<void,some\_class,const std::stirng>(&some\_class::print\_string)(sc0,"hello2");

32

33 (boost::bind(&print\_string,\_1))("Hello func!");

34 boost::bind(&print\_functionname);

35 some\_class sc;

36 (boost::bind(&some\_class::print\_classname,\_1)(sc));

37 (boost::bind(&some\_class::print\_string,\_1,\_2))(sc,"Hello member!");

38 }

[复制代码](javascript:void(0);)

**4.2 区别与bind1st和bind2st**

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 #include <functional>

2 #include <iostream>

3 #include <string>

4 #include <vector>

5 #include <algorithm>

6 #include "boost/bind.hpp"

7 void main()

8 {

9 std::vector<int> ints;

10 ints.push\_back(7);

11 ints.push\_back(4);

12 ints.push\_back(12);

13 ints.push\_back(10);

14 int count=std::count\_if(ints.begin(),

15 ints.end(),

16 boost::bind(std::logical\_and<bool>(),boost::bind(std::greater<int>(),\_1,5),boost::bind(std::less\_equal<int>(),\_1,10))

17 );

18 std::cout << count << '\n';

19 std::vector<int>::iterator int\_it=std::find\_if(ints.begin(),

20 ints.end(),

21 boost::bind(std::logical\_and<bool>(),boost::bind(std::greater<int>(),\_1,5),boost::bind(std::less\_equal<int>(),\_1,10))

22 );

23 if (int\_it!=ints.end())

24 { std::cout << \*int\_it << '\n';}

25

26 }

[复制代码](javascript:void(0);)

**4.3 区别传ref和传instance**

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 // bind instance or reference

2 #include <functional>

3 #include <iostream>

4 #include <string>

5 #include <vector>

6 #include <algorithm>

7 #include "boost/bind.hpp"

8 class tracer

9 {

10 public:

11 tracer() { std::cout << "tracer::tracer()\n"; }

12 tracer(const tracer& other) { std::cout << "tracer::tracer(const tracer& other)\n"; }

13 tracer& operator=(const tracer& other)

14 { std::cout << "tracer& tracer::operator=(const tracer& other)\n"; return \*this; }

15 ~tracer() { std::cout << "tracer::~tracer()\n";

16 }

17 void print(const std::string& s) const

18 { std::cout << s << '\n'; }

19 };

20

21 void main()

22 {

23 tracer t;

24 boost::bind(&tracer::print,t,\_1)(std::string("I'm called on a copy of t\n"));

25 tracer t1;

26 boost::bind(&tracer::print,boost::ref(t1),\_1)( std::string("I'm called directly on t\n"));

27

28 }

[复制代码](javascript:void(0);)

**5.bind的应用场景**

**5.1 绑定普通函数**

**bind**可以绑定**普通函数**，包括**函数**、**函数指针**，假设我么有如下的函数定义：

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

1 int f(int a,int b){return a+b;} //二元函数

2 int g(int a,int b,int c) {return a+b+c;} //三元函数

3 typedef int (\*f\_type)(int,int); //函数指针定义

4 typedef int (\*g\_type)(int,int,int); //函数指针定义

　　那么，bind(f,1,2) 将返回一个无参调用函数对象，等价于f(1,2),bind(q,1,2,3)同样返回一个无参调用的函数对象，等价于 g(1,2,3)。这两个绑定表达式没有使用占位符，而是给出了全部的具体参数，代码：

1 cout<<bind(f,1,2)()<<endl;

2 cout<<bind(g,1,2,3)()<<endl;

　　相当于：

1 cout<<f(1,2)<<endl;

2 cout<<g(1,2,3)<<endl;

　　使用占位符bind可以有更多的变化，这才是它真正应该做的工作，下面列出了一些占位符的用法：

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

1 bind(f,\_1,9)(x); //f(x,9),相当于bind2nd(f,9)

2 bind(f,\_1,\_2)(x,y); //f(x,y)

3 bind(f,\_2,\_1)(x,y); //f(y,x)

4 bind(f,\_1,\_1)(x,y); //f(x,x),y参数被忽略

5 bind(g,\_1,8,\_2)(x,y) //g(x,8,y)

6 bind(g,\_3,\_2\_2)(x,y,z) //g(z,y,y),x参数被忽略

　　注意：**必须**在**绑定表达式**中提供**函数要求**的**所有参数**，无论是**真实参数**还是**占位符均可以**。**占位符**可以**出现**也可以**不出现**，出现的**顺序**和**数量没有限定**，但**不能**使用**超过函数参数数量**的**占位符**，比如在绑定f是不能用\_3,在绑定g时不能使用\_4，也不能写bind(f,\_1,\_2,\_2)，这样的形式会导致编译错误。bind完全可以代替标准库中的bind1st和bind2nd，使用bind(f,N,\_1)和bind(f,\_1,N)。要注意的是它们均使用了一个占位符，bind1st把第一个参数用固定值代替，bind2nd把第二个参数用固定值代替。**bind**也可以**绑定函数指针**，用法相同，例如：

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

1 f\_type pf = f;

2 g\_type pg = g;

3 int x =1,y=2,z=3;

4 cout<<bind(pf,\_1,9)(x)<<endl; //(\*pf(x,9))

5 cout<<bind(pg,\_3,\_2,\_2)(x,y,z)<<endl; //(\*pg)(z,y,y)

**5.2 bind绑定成员函数**

　　类的成员函数不同于普通的函数，因为**成员函数指针不能直接**调用**operator()**,它**必须**被**绑定到**一个**对象**或**指针**，然后才能得到**this指针**进而调用**成员函数**。因此bind需要**“牺牲”一个占位符**，要求提供一个**类的实例**、**引用**或者**指针**，通过**对象**作为**第一个参数**来调用**成员函数**，即：

1 bind(&X::func,x,\_1,\_2,…)

　　这意味着使用**成员函数**时只能**最多绑定8个参数**。例如，有一个类demo

1 struct demo

2 {

3 int f(int a,int b){return a+b;}

4 };

　　那么，下面的bind表达式都是成立的：

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

1 demo a,&ra = a; //类的实例对象和引用

2 demo \* p = & a; //指针

3 cout<<bind(&demo::f,a,\_1,20)(10)<<endl;

4 cout<<bind(&demo::f,ra,\_2,\_1)(10,20)<<endl;

5 cout<<bind(&demo::f,p,\_1,\_2)(10,20)<<endl;

　　注意：我们**必须**在**成员函数**前面加上**取地址的操作符&**，表明这是一个**成员函数指针**，否则会无法编译通过，这是与绑定函数的一个小小的不同。**bind**同样支持绑定**虚拟成员函数**，用法**与非虚函数相同**，虚函数的行为将由实际调用发生时的实例来决定。

**5.3 bind绑定成员变量**

**bind**的另一个对类的操作是它可以**绑定public成员变量**，用法与绑定成员函数类似，只需要把**成员变量名**像一个**成员函数**一样去使用。例如：

1 vector<point> v(10);

2 vector<int> v2(10);

3 transform(v.begin(),v.end(),v2.begin(),bind(&point::x,\_1));

4 BOOST\_FOREACH(int x,v2) cout<<x<<“,”;

　　代码中的bind(&point::x,\_1)取出point对象的成员变量x，transform算法调用bind表达式操作容器v,逐个把成员变量填入到v2中。看到这里感到有点困惑，有点难以理解：bind返回的是一个函数对象，该对象对“（）”进行了重载，在transform调用该重载函数应该是将v中的每一个成员变量作为参数传进去，从而取出每一个元素的x变量。

　　使用bind可以实现SGISTL/STLport中的非标准函数适配器select1st和select2nd的功能，直接选择出pair对象first和second成员，例如：

1 typedef pair<int,string> pair\_t;

2 pair\_t p(123,”string”);

3 cout<<bind(&pair\_t::first,p)()<<endl;

4 cout<<bind(&pair\_t::second,p)()<<endl;

**5.4 绑定函数对象**

**bind**不仅能够绑定函数和函数指针，也能够绑定**任意的函数对象**，包括标准库中预定义的函数对象。如果**函数对象**有内部类型定义**result\_type**，那么**bind**可以**自动推导**出**返回值类型**，用法与普通函数一样。但如果**函数对象没有定义result\_type**，则需要在绑定形式上做一点改动，用**模板参数**指明**返回类型**，像这样：

1 bind<result\_type>(Functor,…);

　　标准库和Boost库中的大部分函数都具有result\_type定义，因此不需要特别的形式就可以直接使用bind,例如：

1 bind(std::greater<int>(),\_1,10); //检查 x>10

2 bind(plus<int>(),\_1,\_2); //执行 x+y

3 bind(modulus<int>(),\_1,3), //执行 x%3

　　对于自定义的函数对象，如果没有result\_type类型定义，例如：

1 struct f

2 {

3 int operator() (int a,int b) {return a +b;}

4 };

　　那么我们必须指明

1 bind<int> (f(),\_1,\_2)(10,20)<<endl;

　　这种写法所烧会有些不方便，因此，在编写**自己的函数对象**时，**最好**遵循规范为它们**增加内部typedef result\_type**,这将使函数对象与其他的标准库和Boost库组件配合工作。

**5.5 使用ref库**

　　bind采用**拷贝的方式**保存**绑定对象**和**参数**，这意味着**绑定表达式**中的**每一个变量**都会有**一份拷贝**，如果函数对象或值参数很大、拷贝代价很高，或者无法拷贝，那么bind的使用就会受到限制。因此bind库可以搭配ref库使用，**ref库**包装了**对象的引用**，可以让**bind**存储**对象引用的拷贝**，从而降低了拷贝的代价。但这也带来了一个隐患，因为有时候bind的调用可能会延后很久，程序员必须保证bind被调用时引用是有效的。如果调用是引用的变量或者函数对象你被销毁了，那么将会发生未定义行为。ref配合bind用法的代码如下：

1 int x = 10;

2 cout<<bind(g,\_1,cref(x),ref(x))(10)<<endl;

3 f af;

4 cout<<bind<int>(ref(af),\_1,\_2)(10,20)<<endl;

　　下面的代码则因为引用失效，引发未定义行为：

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 int x = 10;

2 BOOST\_AUTO(r,ref(x));

3 {

4 int \* y = new int(20);

5 r = ref(\*y);

6 cout<<r<<endl;

7 cout<<bind(g,r,1,1)()<<endl;

8 delete y;

9 }

10 cout<<bind(g,r,1,1)()<<endl;

[复制代码](javascript:void(0);)

**5.6 存储bind表达式**

　　很多时候我们需要把写好的bind表达式存储起来，以便稍后再度使用，但bind表达式生成的对象类型声明非常复杂，通常无法写出正确的类型，因此可以使用typeof库的**BOOST\_AUTO宏**来辅助我们，例如：

1 BOOST\_AUTO(x,bind(greater<int>(),\_1,\_2));

2 cout<<x(10,20)<<endl;

**bind可以嵌套**，一个bind表达式生成的函数对象可以被另一个bind再绑定，从而实现类似**f(g(x))**的形式，如果我们有f(x)和g(x)两个函数，那么f(g(x))的bind表达式就是：

1 bind(f,bind(g,\_1))(x)

　　使用bind的嵌套用法必须小心，它不太容易一次写对，也不太容易理解，超过两个以上的bind表达式通常只能被编译器读懂，必须配合良好的注释才能够使用bind嵌套用法。

**5.7 绑定非标准函数**

　　bind可以适配任何一种C++中的函数，但标准形式bind(f,…)不是适用所用的情况，有些非标准函数无法制动推导出返回值类型，典型的就是C中的可变参数函数printf()。必须用bind<int>(printf,…)(…)，例如：

1 bind<int>(printf,”%d+%d=%d\n”,\_1,\_1,\_2)(6,7);

**bind的标准形式**也**不能支持使用了不同的调用方式**，如：**\_\_stdcall**、**\_\_fastcall**、**extern”C”**的函数，通常bind把他们看做函数对象，需要显示的指定bind的返回值类型才能绑定。