boost::lambda 的应用与实现

软件学院 孙子平 2015013249

1 引言

在2011年，`lambda`表达式正式进入C++11标准，成为了语法级别的组件。而本文所涉及的C++函数编程，是指在C++11标准前，借助函数调用运算符重载实现的一种编程范式。

`boost`库里的函数式编程库颇为丰富，其中尤以`lambda`和`Phoenix`两个库著名。在此，我们就以`boost::lambda`作为典型案例，分析C++的函数式编程。在接下来的文章里，我将首先概述函数式编程（第2章），接着分析`boost::lambda`的应用（第3章）和实现（第4章），最后给出自己改进版的实现（第5章），分析其安全、性能和灵活性。在本文的最后，我将对C++的函数式编程进行一个总结（第6章）。

在这里，我假定读者拥有一定的模板编程技能。对于函数对象、匿名函数等相关概念已经有很多了解的读者可以跳过第2章。

声明：所有的代码都通过了**gcc 6.1.1**和**clang 3.8.0**的测试，**visual studio 2015**的编译器对于部分代码可能会出现错误。

2 C++的函数式编程

2.1 函数对象

借助运算符重载，我们可以构建出函数对象。该技术很早就出现在了C++的标准库里，标准库`functional`集中地提供了各种各样的函数对象。接下来我以LLVM头文件（这里我对LLVM的代码进行了简化，以增加可读性）作为示例。标准库在头文件`functional`里定义了`std::less`，其可能的实现如下：

```
template <class T>
struct less {
    bool operator()(const T &lhs, const T &rhs) const {
        return lhs < rhs;
    }
};
```

因而标准库`algorithm`里面只带2个参数的`std::sort`就可以采用如下的实现：

```
template <class RandomAccessIterator>
inline void sort(RandomAccessIterator first, RandomAccessIterator last) {
    sort(first, last, less<typename iterator_traits<RandomAccessIterator>::value_type>());
}
```

在这里我们通过标准库`iterator`中的`std::iterator_traits`进行类型推断得到其迭代器所指的值类型，进而得到可以用于小于比较的函数对象，将之传递给了拥有3个参数的`std::sort`函数（第3个参数是一个用于比较的函数对象）。

当然，我们惊喜的发现函数对象的语法是和函数指针相互兼容的，考虑带3个参数的`std::sort`函数的声明，大致是这个样子的：

```
template <class RandomAccessIterator, class Compare>
inline void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

假设`RandomAccessIterator`迭代器其所指的类型叫`T`，而我们定义了函数`bool compareFunc(const T &lhs, const T &rhs)`用于比较，当我们调用`sort(first, last, compareFunc)`时，模板形参`Compare`的值为`bool (const T &lhs, const T &rhs)`是一种函数类型，又由于当函数类型作为函数形参的类型时会自动转换为函数指针类型，所以函数形参`comp`的真实类型为`bool (*)(const T &lhs, const T &rhs)`，而函数形参`comp`的值为`&compareFun`。这里，我们看到C++对于旧有的C语言的相关概念惊人的抽象和兼容能力，数组抽象为了容器（序列容器），指针抽象为了迭代器，而函数指针抽象为了函数对象。

接下来我们分析一下性能，先丢结论，函数对象的性能可能高于函数指针。虽然`sort`函数多传进去一个参数，但由于该参数实际不占任何空间，实际编译后不会存在一个空对象被压入栈的指令。又由于类模板成员函数的内联展开，原来定义对于`std::less`对象的函数调用运算符会在`sort`函数内展开成一个小于比较的表达式，相较于函数指针，减少了一次函数调用的开销。所以函数对象在空间开销上不差于函数指针，在性能开销上可能优于函数指针。

2.2 匿名函数

脱离C++，所谓匿名函数，就是一个不具有名字的函数。这种函数通常很小，其定义处就是其使用处。考虑C++98，就其语法层面，匿名函数是不直接支持的，但其实通过函数对象，我们可以构建出类似的语法。考虑上面第3行代码，我们希望最后的代码长成这个样子：

```
1 sort(first, last, _1 < _2);
```

形式上，`_1 < _2`就像是定义了一个匿名函数，并将这个函数作为`sort`的第3个参数传了进去。实际上，`_1 < _2`就是所谓的lambda表达式，该表达式产生一个类似`std::less`的函数对象，`_1`代表该函数对象的第一个参数，同理`_2`代表该函数对象的第二个参数。

这里我们可以看出实现这样一个lambda表达式有两个疑问：

1. `_1`和`_2`改如何实现？就上面的代码，我们可以看出来那必须是某一个全局对象。
2. `<`该如何实现？似乎就上面的代码，我们可以假象有这么一个全局作用域的`operator < ()`，当其左右操作数分别是`_1`和`_2`时会返回一个类似`std::less`的对象。

不论如何，这至少证明了借助模板、函数对象、运算符重载和几个预定义的全局变量，C++的匿名函数是可以实现的。当然，`boost::lambda`实现了。

3 boost::lambda的应用

3.1 占位符

诸如`_1`、`_2`和`_3`等的被称之占位符(placeholder)。它们本身形成了一个原子lambda表达式，并分别对应于lambda表达式定义的函数的第1个、第2个和第3个参数。使用的占位符排位最高的那个决定了函数是几元的：

```
1 _1 + 5               // 一元
2 _1 * _1 + _1         // 一元
3 _1 + _2              // 二元
4 bind(f, _1, _2, _3) // 三元
5 _3 + 10              // 三元
```

注意最后一个例子产生的函数忽略了前两个参数。不过，对于lambda表达式产生的函数，永远可以提供给超过其元数的实参，多余的实参会被忽略：

```
1 int i, j, k;
2 _1(i, j, k)        // 返回 i，忽略 j and k
3 (_2 + _2)(i, j, k) // 返回 j+j，忽略 i and k
```

所有的lambda表达式产生的函数均为传引用调用，因而作用在占位符上的副作用将作用在实参上：

```
1 int i = 1;
2 (_1 += 2)(i);          // i 现在是 3
3 (++_1, cout << _1)(i) // i 现在是 4，输出 4
```

3.2 运算符表达式

一条基本的规则是任何一个以lambda表达式作为操作数（子lambda表达式）的运算都是lambda表达式。

不被支持的运算符

一些运算符不被支持，包括`::`、`.`、`.*`、`->`、`->.`、`new`、`new[]`、`delete`、`delete[]`、`?:`、左操作数不是lambda表达式的`=`和左操作数不是lambda表达式的`[]`：

```
int i;
_1 = i;     // 可以
i = _1;     // 不可以。 i 不是lambda表达式
```

逻辑运算符符合短径求值

```
bool flag = true; int i = 0;
(_1 || ++_2)(flag, i);          // i 不会自增
```

成员指针运算符

成员指针运算符`->*`可被自由地重载。有两种使用情况：

- 右操作数是成员变量：返回该成员数据的引用

```
struct A { int d; };
A* a = new A();
    ...
(a ->* &A::d);     // 返回 a->d 的引用
(_1 ->* &A::d)(a); // 类似
```

- 有操作数是成员函数：返回该成员函数的延迟调用

```
struct B { int foo(int); };
B* b = new B();
    ...
(b ->* &B::foo)          // 返回 b->foo 的延迟调用
                         // 接下来必须跟随函数参数列表
(b ->* &B::foo)(1)       // 可以，调用 b->foo(1)
(_1 ->* &B::foo)(b);     // 返回 b->foo 的延迟调用
                         // 并没有任何作用
(_1 ->* &B::foo)(b)(1); // 调用 b->foo(1)
```

3.3 绑定表达式

绑定表达式可以包含以下两种形式：

```
bind(target-function, bind-argument-list)
bind(target-member-function, object-argument, bind-argument-list)
```

绑定表达式延迟对于函数的调用。函数的参数个数与之后给的绑定列表必须一一对应。若返回值推导失败，可以显示指定返回值类型`bind<RET>(target-function, bind-argument-list)`。

函数指针或引用作为目标

```
X foo(A, B, C); A a; B b; C c;
bind(foo, _1, _2, c)(a, b);
bind(&foo, _1, _2, c)(a, b);
bind(_1, a, b, c)(foo);
```

对于上述情况，返回值推到总能成功。对于有歧义的函数，应当通过强制类型转换确定函数。

成员函数作为目标

对象参数既可以是对象的引用也可以是对象的指针：

```
bool A::foo(int) const;
A a;
vector<int> ints;
```

```
    ...
find_if(ints.begin(), ints.end(), bind(&A::foo, a, _1));
find_if(ints.begin(), ints.end(), bind(&A::foo, &a, _1));
```

成员变量作为目标

成员目标虽然不是函数，但也可以使用绑定表达式。

3.4 延迟常量和变量

通过`constant`和`var`可以将其参数转换为返回对应的引用类型的lambda表达式：

```
for_each(a.begin(), a.end(), cout << constant(' ') << _1);
int index = 0;
for_each(a.begin(), a.end(), cout << ++var(index) << ':' << _1 << '\n');
```

这可用于将非lambda表达式转换为lambda表达式，通常这是因为结合性和优先级不满足lambda表达式形成的规则的辅助手段。此外，对于`=`和`[]`这两个运算，其左操作数必须是lambda表达式。

4 boost::lambda的实现

在这里首先感谢shfzhzhr在iteye.com上发表的帖子“C++ Template Metaprogramming——一个小型lambda库的实作”，其中给的减缩版`boost::lambda`给了我很大的帮助。

整个减缩版约莫300多行，可在附件`src/minimal-boostlambda/lambda.h`里看到完整的源代码。注意，简化代码只包含占位符和运算符表达式的实现，不包括绑定表达式之类的实现。

4.1 概要

需要注意，整个**lambda**表达式的实现其实有两套参数，一套是构建**lambda**表达式时，各个操作数组成的函数参数，另一套是调用**lambda**表达式时的函数参数。具体实现上，针对前者为了惰性求值需要捕获其值（或引用），对于后者并不需要存储，但需要递归地传递给子lambda表达式，更具体一些，后者的传递包含了类型的传递和值（或引用）的传递，类型的传递是为了进行返回值类型推导，用的是成员类型`sig::type`，而值的传递属于简单的递归，用的是成员函数`call()`。类型推导使用的是元编程（**metaprogramming**）的技巧。

构建lambda表达式其实是借助运算符的优先级结合性自然地生成一个嵌套的对象。而调用lambda表达式实际上经历了以下4个步骤：

1. 类型传递
2. 类型推导
3. 值（或引用）传递
4. 求值

接下来，我简要介绍一下`boost::lambda`的大致实现。`lambda_functor`是所有lambda表达式得到的最后对象，它为lambda的调用实现一层简要的封装，主要可以用来简化代码，也为后期的类型推断做出一些帮助，其被封装的对象必须实现用于传递参数的类型并最终推导返回值类型的成员类型`sig::type`，以及用于传递参数的值（或引用）并最终求值的成员函数`call()`。

`placeholder`是占位符的类型，它的成员函数`call()`只会返回某一位置的参数。而`lambda_functor_base`是所有运算的共同类型，根据其模板参数的不同会有不同的行为（比如加减乘除），它会借助全局的运算符重载被构建出来，并用引用记录下构建它时的所有操作数，而它的成员函数`call()`会将其调用的所有参数传递给所有操作数，并将得到的所有值进行一次操作后（比如加减乘除）返回。传递是借助一个叫`select()`的全局函数，如果其第一个操作数是`lambda_functor`类型，则调用其被封装的对象的名为`call()`的成员函数，传递给它所有的参数，如果其第一个参数是其他类型，则直接返回该参数。

通俗一点，`select()`全局函数是递归调用子**lambda**表达式的中间层，`placeholder`和延迟常量变量它们本身就是一个原子lambda表达式，不存在子表达式所以无需调用`select()`，而对于其他的诸如运算符表达式、绑定表达式之类的语法，它们可能存在子lambda表达式，因而也就需要调用`select()`。

4.2 `lambda_functor` —— lambda表达式的封装类型

`lambda_functor`是所有lambda表达式最后得到结构的最外层派生得到的函数对，简化版的`lambda_functor`的定义大致长成这个样子：

```
template<class T>
class lambda_functor: public T {
public:
    typedef T inherited;

    lambda_functor() {}
    lambda_functor(const lambda_functor& l) : inherited(l) {}
    lambda_functor(const T& t) : inherited(t) {}
        ...
    template<class A, class B, class C>
    typename inherited::template sig<tuple<A, B, C> >::type
    operator()(A& a, B& b, C& c) const
    {
        return inherited::template call<
            typename inherited::template sig<tuple<A, B, C> >::type
        >(a, b, c, cnull_type());
    }
    template<class A, class B, class C>
    typename inherited::template sig<tuple<A, B, C> >::type
    operator()(A const& a, B const& b, C const& c) const
    {
        return inherited::template call<
            typename inherited::template sig<tuple<A , B , C> >::type
        >(a, b, c, cnull_type());
    }
};
    ...
class plus_action {};
LAMBDA_BINARY_ACTION(+,plus_action)
```

其中`cnull_type()`时一个空类的对象，长这个样子，没有实际的用途：

```
struct null_type {};
static const null_type constant_null_type = null_type();
#define cnull_type() constant_null_type
```

总的而言`lambda_functor`会对其被继承的类做一个封装，做到可复制。总的而言，它具有以下3个功能：

1. 将各个不同元的`operator()()`的参数类型，用`boost::tuple`封装后通过模板参数传递给基类的`sig`结构体。并将得到的结果`sig::type`作为返回值类型。注意：`boost::tuple`元素的个数就是函数调用参数的个数，它是变长的。
2. 将各种不同元的`operator()()`函数调用不改变参数`const`属性传递地给基类指定元数的名为`call()`的成员函数调用（缩减版的代码里面转为了三元），不足的用一个空类`null_type`的实例对象填充。不改变参数`const`属性借助的是函数的`const`重载。上面的代码省略了其他零元、一元、二元的`operator()()`的定义。
3. 唯一地标识了lambda表达式，能被更容易地识别出这是lambda表达式。这对于lambda表达式的构建和`select()`函数的实现是非常有必要的。

这里所涉及的函数参数都属于上文说的调用lambda表达式时的函数参数，上述功能的前两个分别就是类型的传递和值（或引用）的传递。

4.3 `placeholder` —— 占位符

`placeholder`的组件如下，这里只选取了`_2`作为例子：

```
enum { NONE = 0x00, FIRST = 0x01, SECOND = 0x02, THIRD = 0x04, EXCEPTION = 0x08, RETHROW = 0x10 };

template<int N, class T> struct tuple_element_as_reference {
    typedef typename boost::tuples::access_traits<
            typename boost::tuples::element<N, T>::type
    >::non_const_type type;
};
template<int N, class Tuple> struct get_element_or_null_type {
    typedef typename tuple_element_as_reference<N, Tuple>::type type;
};
template<int N> struct get_element_or_null_type<N, null_type> {
```

```
    typedef null_type type;
};

template<int I> struct placeholder;
    ...
template<> struct placeholder<SECOND> {
    template<class SigArgs> struct sig {
        typedef typename get_element_or_null_type<1, SigArgs>::type type;
    };
    template<class RET, class A, class B, class C, class Env>
    RET call(A& a, B& b, C& c, Env& env) const { return b;}
};
    ...
const lambda_functor<placeholder<SECOND> >& _2 = lambda_functor<placeholder<SECOND> >();
```

可以看出`placeholder`其`call()`成员函数的实现真的非常简单，即返回指定位置的参数，对于`_1`就是`return a`、对于`_2`就是`return b`，以此类推。

而其用于类型推到的`sig`则稍微复杂。这里再次提一下`sig`的模板参数`SigArgs`实际上就是一个`boost::tuple<A, B, ...>`，其中`A`、`B`等等对应的就是`call`的模板参数，`boost::tuple`的模板参数是变长的，具体依赖于调用参数的个数，而`sig::type`对应的就是`call()`的模板参数`RET`。

首先`boost::tuples::element<N, SigArgs>::type`会返回`SigArgs`第N个类型，接着`tuple_element_as_reference<N>::type`会在其上增加一个引用，最后`get_element_or_null_type<N, SigArgs>::type`实际上是防御性编程，针对`SigArgs`不为`boost::tuple`而是`null_type`做了特殊处理。总的而言，`sig<SigArgs>::type`的类型便是与之对应的`call()`的参数的引用类型

让我们想一下下面这段代码会有什么结果：

```
int i = 0;
_2(i);
```

实际上单看成员函数`call()`似乎并不产生问题，因为`lambda_functor`会用`cnull_type()`填充这些确实的参数，最后`_2`返回的便是`cnull_type()`。然而实际上错误会发生在类型推导时，在传给`placeholder<SECOND>::sig`的模板形参`SigArgs`的值实际上是`boost::tuple<int>`，而`boost::tuples::element<1, SigArgs>::type`便会报错，无法找到对应的参数，实际通常编译器报的错是无法推导出改模板的类型参数。

4.4 `lambda_functor_base` —— 运算符表达式

相对于`placeholder`，`lambda_functor_base`不可不说是简明扼要：

```
template<class Act, class Args>
class lambda_functor_base;

#define LAMBDA_BINARY_ACTION(SYMBOL, ACTION_CLASS)                          \
template<class Args> class lambda_functor_base<ACTION_CLASS, Args> {        \
public:                                                                     \
  Args args;                                                                \
public:                                                                     \
  explicit lambda_functor_base(const Args& a) : args(a) {}                  \
  template<class RET, class A, class B, class C, class Env>                 \
  RET call(A& a, B& b, C& c, Env& env) const {                              \
    return select(boost::tuples::get<0>(args), a, b, c, env)                \
           SYMBOL                                                           \
           select(boost::tuples::get<1>(args), a, b, c, env);               \
  }                                                                         \
  template<class SigArgs> struct sig {                                      \
    typedef typename                                                        \
        boost::tuples::element<0,SigArgs>::type type;                       \
  };                                                                        \
};
```

其中构造函数的参数`args`是一个`boost::tuple`的对象，不同于上面的`boost::tuple`这次它不仅是用来存储类型，还是用来真正存储值（或引用）。实际上它被用于存储运算的所有操作数，对于一个二元运算总共会有两个操作数，因此实际上`boost::tuple`在上面的例子中是个二元组。

先看`call()`，其中的`select()`我们将在之后具体讨论，实际上它是将所有的参数再一次传递给了子lambda表达式，最后将得到的结果进行一次运算返回。至于`sig`的实现不可不说是简单粗暴，直接返回其左操作数的类型。由于这是减缩版，并不具参考性。个人认为这会是一个bug。

通过下面的这段代码，可以看出如何通过全局作用域重载运算符实现代运算符的lambda表达式的合成。

```
template<bool If, class Then, class Else> struct IF { typedef Then RET; };
template<class Then, class Else> struct IF<false, Then, Else> { typedef Else RET;};

template<class T> struct const_copy_argument {
    typedef typename typename IF<boost::is_function<T>::value, T&, const T>::RET  type;
};

#define LAMBDA_BE(OPER_NAME, ACTION)                                                \
template<class Arg, class B>                                                        \
inline const lambda_functor<                                                        \
  lambda_functor_base<                                                              \
    ACTION, tuple<lambda_functor<Arg>, typename const_copy_argument <const B>::type>    \
  >                                                                                 \
>                                                                                   \
OPER_NAME (const lambda_functor<Arg>& a, const B & b) {                             \
  return  lambda_functor_base<                                                      \
      ACTION, tuple<lambda_functor<Arg>, typename const_copy_argument <const B>::type>  \
    > (tuple<lambda_functor<Arg>, typename const_copy_argument <const B>::type>(a, b)); \
}
    ...
LAMBDA_BE(operator+, plus_action)
```

这里我们可以看出实际上对于函数类型作为操作数的情况，`boost::lambda`对此加了引用，这是元编程的手法。限于知识水平，我没有悟出这么做的意义何在。通过const引用，可以成功地捕获住操作数。但需要注意，这里由于采用了const引用捕获其值，因而对于可能改变其值得运算，会有编译不通过的问题。实际的实现应当采用const重载。再次说一下，由于这是`boost::lambda`的减缩版，并不完全具有参考性，作者只实现了加减乘除4个运算，对于作者的情况这是可以的。

4.5 `select` —— 传递子lambda表达式

对于包含子lambda表达式的情况，`select()`是一个举足轻重的函数，实际上它的实现很简单，即当其第一个参数为`lambda_functor`类型的时候便会调用它，递归计算子表达式。如果其第一个参数是普通类型，即为一个被捕获的值，则直接返回它。

```
template<class Any, class A, class B, class C, class Env>
inline Any& select(Any& any, A& a, B& b, C& c, Env& env) { return any; }

template<class Arg, class A, class B, class C, class Env>
inline typename Arg::template sig<tuple<A, B, C, Env&> >::type
select ( const lambda_functor<Arg>& op, A& a, B& b, C& c, Env& env ) {
    return op.template call<
        typename Arg::template sig<tuple<A, B, C, Env&> >::type
    > (a, b, c, env);
}
    ...
```

4.6 局限

类型推导

由于没有采用C++11，`boost::lambda`借助了traits技术，并通过`boost::tuple`这一类型作为模板参数在各个函数调用中传递，从而解析出返回类型。其返回类型的推导是较为主观的。例如其加减乘除会认为在其左右操作数都是同一类型时，其返回类型也是该类型。而当遇到其左右操作数不是同一类型的时候，这将出现函数推导错误。当然`boost::lambda`提供了一套比较复杂的机制，用户可使用模板特化帮助这种情况下的类型推导。这是`boost::lambda`的一大局限。

参数个数

由于没有变长模板参数，实际上`boost::lambda`处理的参数个数是有最高限制的，虽然通常情况下这种限制并不容易达到，但无疑，通过重复的把一元、二元一直到九元的情形都写一遍，会造成代码膨胀。对于函数参数个数处理不够灵活又是`boost::lambda`的一大局限。

5 基于C++11的lambda实现

C++11对于lambda表达式的实现无疑是一大福音，我认为主要体现在下面3个地方：

1. decltype带来的类型推导：无疑`boost::lambda`最令人头大的代码集中在类型推导上。借助`decltype`，我们可以删去所有的基于`sig`的类型推导，不仅简化代码，还可以更加合理的进行类型推导。
2. 变长模板参数：通过变长模板参数，我们可以用少得多的代码涵盖更多的参数。
3. `std::forward`、右值引用和引用折叠：这些语法避免了const重载，也对右值有了更好的支持。代码更短，更安全且适用范围可以更广。

于是，基于C++11，我重新实现了一个精简版的lambda库。我实现了占位符、除了`->*`之外的所有可重载运算符（包括需要特殊处理的`=`和`[]`）以及延迟变量的相关语法。其他语法也可以非常简单的扩展。限于篇幅，我就只展现一部分代码。完整的代码可参见附件`src/mylambda/lambda.h`。

5.1 `placeholder` 变长参数模板的递归实例化

运用变长模板参数可以递归实例化`placeholder`，在对`placeholder<0>`写一个模板特例，即可实现占位符的功能，具体代码如下：

```
template<int N> struct placeholder {
    template<class Arg, class... Args>
    auto call(Arg &&arg, Args &&... args) const -> decltype(placeholder<N - 1>().call(std::forward<Args>(args)...)) {
        do_nothing(arg);
        return placeholder<N - 1>().call(std::forward<Args>(args)...);
    }
};
template<> struct placeholder<0> {
    template<class Arg, class... Args>
    auto call(Arg &&arg, Args &&... args) const -> decltype(arg) {
        do_nothing(args...);
        return std::forward<Arg>(arg);
    }
};

#define LAMBDA_PLACEHOLDER(POSITION, NAME) const auto NAME = lambda_functor<placeholder<POSITION> >();
LAMBDA_PLACEHOLDER(0, _1)
LAMBDA_PLACEHOLDER(1, _2)
LAMBDA_PLACEHOLDER(2, _3)
    ...
```

举个例子`placeholder<2>().call(a, b, c, d)`，实际会递归调用`placeholder<1>().call(b, c, d)`，再调用`placeholder<0>().call(c, d)`，这时，匹配上了特化的模板，最后返回的值为`c`。由于采用了一种叫做perfect forwarding的技巧，这种占位符能够完整地保留下参数类型。同时在这里，可以看到后缀返回值声明配上`decltype`是如何解决返回值推导的问题的。在C++14的帮助下，我们甚至可以完全省略后缀的返回值声明，代码将更为精简。

5.2 细化`lambda_functor_base`

在我写的库中，为了避免使用`boost::tuple`存储下捕获的值，我取消掉了原来的`lambda_functor_base`，转而以多个更加具体的类替代，包括：

- `lambda_binary_functor`：所有的二元运算符
- `lambda_prefix_unary_functor`：所有的一元前缀运算符
- `lambda_postfix_unary_functor`：所有的一元后缀运算符
- `lambda_ref_functor`：延迟变量`var()`

这里就以最为简单的延迟变量为例：

```
template<class Arg> struct lambda_ref_functor {
    Arg &arg;
    lambda_ref_functor(Arg &arg): arg(arg) {}
    template<class... Args>
    auto call(Args &&... args) const -> decltype(select(arg, std::forward<Args>(args)...)) {
        return select(arg, std::forward<Args>(args)...);
    }
};
template<class A> inline lambda_functor<lambda_ref_functor<A> > var(A &a) {
    return lambda_ref_functor<A>(a);
}
template<class A> inline lambda_functor<lambda_ref_functor<const A> > var(const A &a) {
    return lambda_ref_functor<const A>(a);
}
```

这是一个很典型的例子，我直接用模板参数`Arg`直接存储需要捕获的变量，同时借助const重载保留被捕获变量的const属性，这比用`boost::tuple`更为直接了当。

6 结束语

早期C++的面向对象采用了颇为烧脑的泛型编程和函数对象的技术，甚至不惜动用元编程进行一些编译期的逻辑。然而随着时代的进步，我们可以看到，C++11就像是一门全新的语言，它摧枯拉朽般地将旧有的拗口的技术全部推翻，代之以简洁明了的语义。然而另一方面，新的lambda表达式的语法，似乎也算是给上文所讨论的一切判了死刑。如今C++已然从语法的层面上支持lambda表达式，但这不代表我们应当忘记模板赋予这门语言的自由。