Lambda和高阶函数

◎ 文／刘新宇

杜甫有诗：窗含西岭千秋雪，门泊东吴万里船。编程领域有时也会有类似的境界，本文作者使用C++模版实现与LISP语言相近的功能是一个有趣的尝试，也可以看作泛型编程的一例吧。

“孔子登东山而晓鲁，登泰山而晓天下。”在城市水泥森林中，举目只能看到“咫尺蓝天”。编写程序也是如此，如果终日沉浸于代码，就难免“学而不思则罔”。本文将从常见的日常程序开始，通过不断抽象，尝试从代码的“森林”中走出来。

首先看三段比较常见的代码：

```
int sumInt(int a, int b){
    int res=0;
    for(int i= a; i<= b;  ++i)
        res+=i;
    return res;
}
```

```
int sumCube(int a, int b){
    int res=0;
    for(int i= a; i<= b;  ++i)
        res+=  i*i*i;
    return res;
}
```

```
double sumPi(int a, int b){
    double res=0;
    for(int i=1; i<= b; i+=4)
          res+=1/(static_cast< double>(i)
*static_cast< double>(i+2));
    return res;
}
```

这三个程序表面不同，但形式上却很相似。好像填词，只要词牌一样，就有一致的形式。“明月几时有，把酒问青天”是《水调歌头》，“才饮长沙水，又食武昌鱼”也是《水调歌头》。

这三段程序分别计算如下的序列的和 ：

$$1+2+3+\ldots=\sum_{i=1}^{n} i \tag{1}$$

$$1^3+2^3+3^3+\ldots=\sum_{i=1}^{n} i^3 \tag{2}$$

$$\frac{1}{1\cdot 3}+\frac{1}{5\cdot 7}+\frac{1}{9\cdot 11}+\ldots=\sum_{i=1}^{n}\frac{1}{(2i-1)(2i+1)} \tag{3}$$

其中(3)会收敛到π / 8。

对于计算机程序，这种类似意味着抽象，一旦加以归纳，就可以减少重复劳动，从而减小了重复中出现错误的风险。上述代码可以抽象为：

```
template <class  T, class F, class  G >
T  sumGeneric(T  a,  T  b, F func,  G  next){
    T  res(0);
   for(T  i= a; i<= b; next(i))
      res  +=  func(i);
   return res;
};
```

这段代码，允许用户将循环体内的过程func和next以参数的形式传入。只要它们能以函数的形式加以调用即可。

func和next可以采用非常轻量的方法实现：

```
template <class  T >
struct Self{
    T  operator()(T  x) { return x; }
};

template <class  T >
struct Cube{
    T  operator()(T  x) { return x*x*x; }
};

template <class  T >
struct MyFunc{
    T  operator()(T  x){
      return 1/(x*(x+2));
    }
};

template <class  T >
struct Inc{
   void operator()(T &  x){  ++ x;}
};

template <class  T >
struct Inc4{
   void operator()(T &  x){ x+=4;}
};
```

测试代码如下：

```
std::cout<<"sum int 1..100=" <<sumGeneric(1,
100, Self< int>(), Inc< int>()) <<"\n";
std::cout<<"sum cube 1..100=" <<sumGeneric
(1, 100, Cube < int>(), Inc< int>()) <<"\n";
std::cout<<"sum to PI 1..100="<<8.0*sumGeneric
(1.0, 100.0, MyFunc <double>(),Inc4< double>())
<<". \n";
```

程序输出 ：

```
sum int 1..100=5050
sum cube 1..100=25502500
```

```
sum to PI 1..100=3.12159
```

可见，通用的累加函数可以达到同样效果，但却更加灵活。用户可以自己定制func和next。在数学上，通常称这种“函数的函数”为高阶函数。

高阶函数

高阶函数，虽然进行了一定程度的抽象，但仍然存在一些问题。为了使用高阶函数，用户不得不提供自己的Inc，Inc4, Self, Cube和MyFunc，明显可以看出这样的functor数量众多、内容简单、分布于程序的各处、并且不易维护；有时甚至很难命名。这种functor爆炸的问题怎么解决？

由于高阶函数操作其他函数，因此作为高阶函数参数的各种普通函数会分布于程序各处。这些普通函数由用户传给高阶函数使用，他们通常小巧简单，但却数量众多。并且高阶函数越抽象，就越能覆盖更广的应用，也就是能够操作更多的普通函数。

这里所说的普通函数，并不一定是C++语法上的函数，而是某种程序单元，它们可以是普通函数，更多的情况下，以functor的形式出现。随着程序越来越复杂，各种普通函数会充斥在程序的各个角落，变得不容易维护。

Lambda

为了给出本质的解决方案，可以引入Lambda的概念。Lambda在最近数年中逐渐活跃并受到重视，一些流行的语言先后加入了对Lambda演算的支持，例如python，例如C#等等。实际上Lambda并不是什么新潮的东西，一些具有良好传统的语言如Lisp，很早就开始大规模使用Lambda。Lambda演算[4]由数理学家Alonzo Church在20世纪30年代引入。它对函数式编程，特别是Lisp语言有着巨大的影响。并且在后来被发现与图灵机等价，从而成为计算机科学的奠基石。

本文仅仅给出一个相对不严格的直观的解释。Lambda可以表示为如下形式：

(λ 〈format − paramcters〉 · 〈body〉)

由于计算机代码中，无法输入希腊字母以及排版下标，所以Lambda演算可以用普通英文字母表示为l：

(lambda (<formal-parameters>). <body>)

例如定义一个匿名函数，它接受一个自变量x，该函数将x + 2，可以用lambda演算表达为：

(lambda (x). (x+2))

而前面的MyFunc函数，可以直接用Lambda演算表达为：

(lambda (n). 1/((2*n-1)*(2*n+1)))

有了Lambda演算为什么就能够解决前面叙述的小函数爆炸的问题呢？因为可以不在程序的开始部分定义众多的小函数，而仅仅在使用高阶函数时，通过Lambda演算将匿名小函数传给高阶函数，从而实现原来的功能。例如下面的伪代码，使用Lambda演算，实现π / 8的计算：

```
sumGeneric(
    1, 100,
    lambda(n). 1/((2*n-1)*(2*n+1)),
    lambda(i). (i++)
    )
```

上述代码在使用高阶函数sumGeneric时，针对第三个参数，提供了一个匿名小函数，该函数将会针对自变量计算相邻两个奇数乘积的倒数；针对第四个参数，也提供了一个匿名小函数，该函数将会将自变量的值增加1。

而计算x3和sin(x)积分的例子则可以写为：

```
integral(lambda(x). x*x*x, 0, pi/2, 0.001);
integral(lambda(x). sin(x), 0, 4, 0.001);
```

实现

如何在C++中实现Lambda呢？如果编译器提供对Lambda关键字的支持，自然是“善之善者也。”但是目前没有任何C++编译器做到这点。

为了支持Lambda，一些具有创造性的工作采用了C++语言的特性，提供了一定程度实现。下面的代码使用了boost::lambda来实现一个匿名小函数lambda(x).(x-=4)：

```
# include " boost/lambda/lambda.hpp"

int main() {
  using namespace boost::lambda;

  std::vector< int>  vec(3);
  vec[0]=12;
  vec[1]=10;
  vec[2]=7;

 // Transform using a lambda expression
 std::transform(vec.begin(),vec.end(),vec.begin(),_1-=4);
}
```

boost::lambda中，不需要显示地写出lambda关键字，而是使用_1、_2这样的占位符来直接描述函数体。作者在这里给出一个独立思考出的实现。没有boost强大，但是简单直观。

实现Lambda要考虑以下几个问题：

- 执行如下lambda语句后，

lambda <arg> <body>

从概念上讲会得到一个函数。但C++中函数都是通过代码静态定义的，而< body>部分的描述是动态的。使用宏来实现lambda就有一定的困难。因为某些情况下，它不能作为一个值传入高阶函数，从而引发编译错误。

所以< body >部分只能是一个“行为看起来像函数”的东西，也就functor。而其本身又是由诸多元素组合而成（加减乘除等），由此推断，这些元素也必然是functor。

● <body>可以是含有变量的表达式，如

2*x+y-1

但是<body>中不能使用C++的变量，因为这与1中的结论矛盾。所以< body >中的变量也是一种functor，它们可以将求值时的代换操作传播到所有functor内。

● 使用

<body>(1,2)

求值时，必须保证1传播到所有x的地方，2传播到所有y的地方。这显然是一种递归的操作，所以模型是一种递归模型

根据上述分析，就可以从简单到复杂逐步实现Lambda。

● 首先实现< body >中的变量，它们在递归论中称为“投影函数”。可以通过vari的形式可以获取第i个变量，如下：

```
std::cout<<"var<1>()(1,2)=" << Var<1>()(1,2) <<std::endl;
std::cout<<"var<2>()(1,2)=" << Var<2>()(1,2) <<std::endl;
std::cout<<"var<1>()(1)=" << Var<1>()(1) <<std::endl;
```

Var可以利用一个接受整数参数的模板来实现2：

```
template <int n > struct Var;

template <> struct Var<1>{
   template <class T >
    T operator()(T a1, T a2){ return a1; }
    template <class T >
    T operator()(T a1){ return a1; }
};

template <> struct Var<2>{
   template <class T >
    T operator()(T a1, T a2){ return a2; }
};
# define _x Var<1>()
# define _y Var<2>()
```

● 对于二元运算，例如加法操作，可能的使用方式大致可以列举为：

```
std::cout<<"plus(_x, _y)(3, 4)= " <<plus(_x, _y)(3, 4)
<<'\n'; std::cout<<"plus(1,2)()="<<plus(1,2)()<<'\n';
std::cout<<"plus<_x, 3)(2)= " <<plus(_x, 3)(2) <<'\n';
std::cout<<"plus(_x, plus(_x, 2)(1)="
        <<plus(_x, plus(_x, 2))(1)<<'\n';
```

一个自然的想法是写一个加法模板，其接受两个functor，实现如下：

```
template <class Func1, class Func2 >
struct Plus{
   Plus(Func1 f1, Func2 f2):_f1(f1),_f2(f2){}

    // ex: f=plus(_x, _y), f(1, 2)
    template <class T >
     T operator()(T x1, T x2){
        T x = _f1(x1, x2);
        T y = _f2(x1, x2);
       return x + y;
     }

private:
    Func1 _f1;
    Func2 _f2;
};
```

模板在构造时把x和y对应的functor保存起来。在将来调用时，使用这两个functor传播x和y，然后相加即可。为了方便用户，可以提供一个辅助函数来简化构造，如下：

```
template <class A1, class A2 >
Plus< A1, A2 > plus(A1 a1, A2 a2){
   return Plus< A1, A2 >(a1, a2);
}
```

当加数或被加数不全为变量时，_f1或_f2可能是基本类型，也可能是functor。如果_f1是基本类型，计算结果是：

_f1+_f2(x)

反之为：

_f2(x)+_f1

如果它们都是一元函数，则结果是：

_f1(x)+_f2(x)

但是在C++中，不能写出这样的语句：

```
int a;
if(_f1 is preimers)
  a = _f1;
else
  a = _f1(x);
```

这是因为，如果_f1是基本类型，则else一句在编译时会错。这个问题可以通过模板偏特化解决。构造基本类型表：

```
template <class T, class U >
struct TList{
   typedef T First;
   typedef U Rest;
};

typedef TList< int,
   TList< long,
      TList< char,
         TList<float ..., Empty > > ... > Premiers;
```

然后编写一个编译期程序，用于判断某个类型是否是基本类型：

```
template <class TypeList, class T > struct Find{
   static const int value = 1 + Find< typename TypeList::
Rest, T>::value;
};

template <class T, class U > struct Find< TList< T,U >,T >
```

```
{
    static const int value = 0;
};

template <class  T >  struct Find< Empty,  T >{
    static const int value  =  -1000;
};

template <class Func >  struct IsFunctor{
    static const bool value  =  Find<Premiers, Func>::
value  <  0;
};
```

其中Find程序利用递归的方法在表中寻找某类型是否是存在，如果找到，则返回类型的下标，反之返回一个负数。使用上述辅助过程可以编写出一个混合求值器：

```
template < bool isFunctor>  struct Eval;

// f is a functor
template <>  struct Eval< true>{
    template <class F, class  T >
    static  T  apply(F f,  T  x){
        return f(x);
    }

    template <class F, class  T >
    static  T  apply(F f,  T  x1,  T  x2){
        return f(x1, x2);
    }
};

// f is not a functor
// ==>f is premier value;
template <>  struct Eval<false>{
    template <class F, class  T >
    static  T  apply(F f,  T  x){
        return f;
    }
};
```

这个模板使用apply将一个函数作用到其自变量上。对于基本类型则直接返回它的值。使用Eval后，Plus模板改写如下：

```
template <class Func1, class Func2 >
struct Plus{
   Plus(Func1 f1, Func2 f2):_f1(f1),_f2(f2){}

   // ex: f=plus(_x, _y), f(1, 2)
   template <class  T >
    T  operator()(T  x1,  T  x2){
       T  x = IsFunctor< Func1>::value _f1(x1, x2) : x1;
       T  y = IsFunctor< Func2>::value _f2(x1, x2) : x2;
       return x + y;
    }

   // ex: f=plus(_x, 1), f(2)
   template <class  T >
    T  operator()(T  x1){
       T  x = Eval<IsFunctor< Func1>::value>::apply(_f1, x1);
       T  y = Eval<IsFunctor< Func2>::value>::apply(_f2, x1);
       return x + y;
    }

   // ex: f=plus(1, 2), f()
   //means that _f1  &  f2 are premiers
   Func1 operator()(){
       return _f1+_f2;
    }

private:
    Func1 _f1;
    Func2 _f2;
};
```

前面的所有测试运行结果如下：

```
_x1,2)=1
_y(1,2)=2
_x(1)=1
plus(1, 2)()=3
plus(_x, _y)(3, 4)=7
plus<_x, 3)(2)=5
plus(_x, plus(_x, 2)(1)=4
```

仿照加法，可以写出减法、乘法和除法等二元操作，它们非常相似，可以采用Policy的方式，将运算符注入。同理可以实现一元函数。这样就可以利用它们组合出Lambda表达式。

最终的lambda，是前面内容的语法封装，以方便理解和使用：

```
template <class A1, class A2 >
int arg(A1 a1, A2 a2){ return 0; }

template <class A1 >
int arg(A1 a1){ return 0; }

template <class  T >
T  body(T  f){ return f; }

template <class  T >
T  lambda(int /*arg*/,  T  f){ return f;}
```

使用方式如下：

```
template <class F >
int sum(int a, int b, F f){
  int res(0);
  for(int i= a; i<= b;  ++i)
    res+=f(i);
  return res;
}

int main(int argc, char** argv){
  std::cout<< "[f(x)=2*(x+1)], f(1)+f(2)+...+f(100)= "
     << sum(1, 100, lambda(arg(_x), body(times(2, plus
(_x, 1))))) << "\n";
}
```

程序输出：

```
[f(x)=2*(x+1)], f(1)+f(2)+...+f(100)=10300
```

这个lambda的使用方式，更像Lisp——使用前缀运算。如果希望使用中缀表达式，可以通过重载全局运算符来实现。

虽然lambda的“体”部分功能受限，但由于其具有“值”的特性，因此可以直接嵌入到表达式中。■

■ 责任编辑：赵健平（zhaojp@csdn.net）