Functional Programming

Weng Kai

本周内容基于

- Functional Thinking, Neal Ford, 2015

-

我们用几分钟来想象一下自己是一名伐木工人，手里有林场里最好的斧子，因此你是工作效率最高的。突然有一天场里来了个推销的，他把一种新的砍树工具——链锯——给夸到了天上去。这人很有说服力，所以你也买了一把，不过你不懂得怎么用。你估摸着按照自己原来擅长的砍树方法，把链锯大力地挥向树干——不知道要先发动它。“链锯不过是时髦的样子货罢了”，没砍几下你就得出了这样的结论，于是把它丢到一边重新捡起用惯了的斧子。就在这个时候，有人在你面前把链锯给发动了.....

- 学习一种全新的编程范式，困难并不在于掌握新的语言
- 真正考验人的，是怎么学会用另一种方式去思考

范式发展

- 计算机科学的进步经常是间歇式的，好思路有时搁置数十年后才突然间变成主流
- Simula 1967—>C++ 1983
- Python 1986—>今天成为最流行的语言
- LISP 1952—>函数式编程范式

词频统计

```
while (m.find()) {
    String word = m.group().toLowerCase();
    if (!NON_ WORDS.contains(word) ) {
        if ( wordMap.get(word) == null) {
            wordMap.put(word, 1);
        } else {
            wordMap.put(word, wordMap.get(word)+1);
        }
    }
}
```

```
list.stream()
    .map(w -> w.toLowerCase())
    .filter(w -> !NON_ WORDS.contains(w))
    .forEach(w ->
        wordMap.put(
            w,
            wordMap.getOrDefault(w, 0) + 1
        )
    );
```

交织

- complect： 穿插缠绕地合为一体，使错综复杂

- 命令式编程风格常常迫使我们出于性能考虑，把不同的任务交织起来，以便能够用一次循环来完成多个任务

- 而函数式编程用map()、filter()这些高阶函数把我们解放出来，让我们站在更高的抽象层次上去考虑问题，把问题看得更清楚

谁支持函数式

- Clojure、Scala
- Groovy：支持记忆，运行时自动缓存函数返回值
- Java 8：高阶函数（Lambda）
- JavaScript：原本就有
- C++ 11：Lambda块
- Python

4GL（第四代语言）

- 1990年代出现大量新语言：dBASE、Clipper、FoxPro、Paradox
- 比3GL（C、Pascal）具有更高的抽象层次
- 用一句命令就可以实现丰富的功能

函数式语言

- 人生苦短，远离malloc
- 对于应用程序开发，失去对内存的直接控制没有什么可惋惜的
- Java接管内存分配减轻了程序员的负担
- 函数式语言用高阶抽象从容取代基本控制结构也有同样的意义
- 将繁琐的细节交给run-time

简洁

- 面向对象编程通过封装不确定因素来使代码能被人理解
  - 开发者针对具体问题建立专门的数据结构，相关的专门操作以“方法”的形式附加在数据结构上
- 函数式编程通过尽量减少不确定因素来使代码能被人理解
  - 在有限的几种关键数据结构（如list、set、map）上运用针对这些数据结构高度优化过的操作，以此构成基本的运转机构。开发者再根据具体用途，插入自己的数据结构和高阶函数去调整机构的运转方式

这是不同层面上的重用

- 面向对象编程范式就是建立新的类和类间的消息。把所有的数据结构都封装成类，一方面压制了方法层面的重用，另一方面鼓励了大粒度的框架式的重用

- 函数式编程的程序构造更方便我们在比较细小的层面上重用代码

字符串处理的例子

- 有一个名字列表，其中一些条目是单字符的，要求去掉这些单字符的条目，将剩下的每个条目的首字母改成大写，最后产生一个逗号分隔的字符串

命令式

- 程序是一系列改变状态的命令

```
public class TheCompanyProcess {
    public String cleanNames(List<String> listOfNames) {
        StringBuilder result = new StringBuilder();
        for(int i = 0; i < listOfNames.size();  i++) {
            if (listOfNames.get(i). length()  > 1) {
                result.append(capitalizeString(listOfNames.get(i))). append
",");
            }
        }
        return result.substring(0, result.length()  - 1). toString();
    }

    public String capitalizeString(String s) {
        return s.substring(0, 1). toUpperCase()  + s.substring(1, s.length()
    }
}
```

命令式

- 命令式编程鼓励程序员将操作安排在循环内部去执行。本例中做了三件事：

  - filter，筛选列表，去除单字符条目
  - transform，变换列表，使名字的首字母变成大写
  - convert，转换列表，得到单个字符串
  - 这三种操作可以说是我们在列表上施展的“ 三板斧”。在命令式语言里，这三种操作都必须依赖于相同的低层次机制（对列表进行迭代）

函数式

- 将程序描述为表达式和变换，以数学方程的形式建立模型，并且尽量避免可变的状态。函数式编程语言对问题的归类不同于命令式语言。 如上页所列的几种操作（filter、transform、convert），每一种都作为一个逻辑分类由不同的函数所代表，这些函数实现了低层次的变换，但依赖于开发者定义的高阶函数作为参数来调整其低层次运转机构的运作

Java 8的实现

```
public String cleanNames(List<String> names) {
    if (names == null) return "";
    return names
            .stream()
            .filter(name -> name.length()  > 1)
            .map(name -> capitalize(name))
            .collect(Collectors.joining(","));
}

private String capitalize(String e) {
    return e.substring(0, 1). toUpperCase()  + e.substrin
g(1, e.length());
}
```

并行要怎么做?

```
public class TheCompanyProcess {
    public String cleanNames(List<String> listOfNames) {
        StringBuilder result = new StringBuilder();
        for(int i = 0; i < listOfNames.size();  i++) {
            if (listOfNames.get(i). length()  > 1) {
                result.append(capitalizeString(listOfNames.get(i))). append
",");
            }
        }
        return result.substrin
    }

    public String capitalizeSt
        return s.substring(0,                                    th()
    }
}
```

```
public String cleanNamesP(List<String> names) {
    if (names == null) return "";
    return names
            .parallelStream()
            .filter(n -> n.length()  > 1)
            .map(e -> capitalize(e))
            .collect(Collectors.joining(","));
}
```

完美数分类

- 完美数的真约数之和正好等于自身
  - 6=1+2+3

完美数	真约数之和 = 数本身
过剩数	真约数之和 > 数本身
不足数	真约数之和 < 数本身

命令式

- ImpNumberClassifierSimple.java

没有副作用的版本

- NumberClassifier.java

函数式的版本

- number_classifier8/NumberClassifier.java

什么是函数式编程思维

- 函数式编程关心数据的映射，命令式编程关心解决问题的步骤

- 函数式编程关心类型（代数结构）之间的关系，命令式编程关心解决问题的步骤

- 函数式编程中的lambda可以看成是两个类型之间的关系，一个输入类型和一个输出类型。lambda演算就是给lambda表达式一个输入类型的值，则可以得到一个输出类型的值，这是一个计算，计算过程满足α-等价和 β-规约。

- 函数式编程的思维就是如何将这个关系组合起来，用数学的构造主义将其构造出你设计的程序。

Lambda演算

- Lambda演算是一套用于研究函数定义、函数应用和递归的形式系统。它由 Alonzo Church 和 Stephen Cole Kleene 在 20 世纪三十年代引入，Church 运用 lambda 演算在 1936 年给出 判定性问题 (Entscheidungs problem) 的一个否定的答案。这种演算可以用来清晰地定义什么是一个可计算函数。关于两个 lambda 演算表达式是否等价的命题无法通过一个通用的算法来解决，这是不可判定性能够证明的头一个问题，甚至还在停机问题之先

lambda函数

- Lambda演算中的函数是一个表达式，写成
  - lambda x . body
- 表示“一个参数参数为x的函数，它的返回值为body的计算结果。”这时我们说：Lambda表达式绑定了参数x
  - x -> body

术语

- 标识符引用（Identifier reference）： 标识符引用就是一个名字，这个名字用于匹配函数表达式中的某个参数名
- 函数应用（Function application）： 函数应用写成把函数值放到它的参数前面的形式，如(lambda x . plus x x) y
  - // js
  - const lambda = x => plus(x, x); lambda(y);

标识符

- 如果一个标识符是一个闭合Lambda表达式的参数，我们则称这个标识符是绑定标识符；如果一个标识符在任何封闭上下文中都没有绑定，那么它被称为自由标识符
    - lambda x . plus x y：在这个表达式中，y和plus是自由的，因为他们不是任何闭合的Lambda表达式的参数；而x是绑定的，因为它是函数定义的闭合表达式plus x y的参数。
    - lambda y . (lambda x . plus x y)：在内层演算lambda x . plus x y中，y和plus是自由的，x是绑定的。在完整表达中，x和y是绑定的：x受内层绑定，而y由剩下的演算绑定。plus仍然是自由的
- // js描述
    - x => plus(x, y); // x是绑定标识符，y和plus函数是自由标识符
    - y => x => plus(x, y); // 在内层函数中x是绑定标识符，y和plus函数是自由标识符。在整个函数中x和y是绑定标识符，plus函数是自由标识符。

Lambda演算运算法则

- Alpha转换

- Beta规约

Alpha α转换

- Alpha是一个重命名操作; 基本上就是说，变量的名称是不重要的：给定Lambda演算中的任意表达式，我们可以修改函数参数的名称，只要我们同时修改函数体内所有对它的自由引用

- x => x + 1;

- // 我们可以将x换成y，并不会更改表达式的含义

- y => y + 1;

Beta β规约

- 我们在应用lambda表达式时，在不引起绑定标识符和自由标识符之间的任何冲突的情况下，可以用参数值对lambda演算中对应的标识符相关的部分做替换，替换方法是把标识符用参数值替换

    - (lambda x . x + 1) 3: 所谓Beta规约就是，我们可以通过替换函数体（即“x + 1”）来实现函数应用，用数值“3”取代引用的参数“x”。于是Beta规约的结果就是“3 + 1”。

    - (lambda x y. x y) (lambda z . z * z) 3: 这是一个有两个参数的函数，它(的功能是)把第一个参数应用到第二个参数上。当我们运算时，我们替换第一个函数体中的参数“x”为lambda z . z * z；然后我们用“3”替换参数“y”，得到：(lambda z . z * z) 3。 再执行Beta规约，有“3 * 3”。

- 遇到标识符冲突时，我们可以使用Alpha转换：

    - (lambda z . (lambda x . x + z)) (x + 2) 3 // 我们先执行Alpha转换上面的表达式等价于下面的： (lambda z . (lambda y . y + z)) (x + 2) // 然后执行Beta规约 ==> (lambda y . y + (x + 2))

eta η规约

- 两个lambda算子若对于相同的输入产生相同的输出，则认为它们外延相等，可以互相规约。

- lambda x . f x <==执行eta规约，等价于==> f

- x => f(x); // 等价于f(x);

函数式编程

- 与面向对象编程（Object-oriented programming）和过程式编程（Procedural programming）并列的编程范式。

- 最主要的特征是，函数是第一等公民

- 强调将计算过程分解成可复用的函数，典型例子就是map方法和reduce方法组合而成 MapReduce 算法

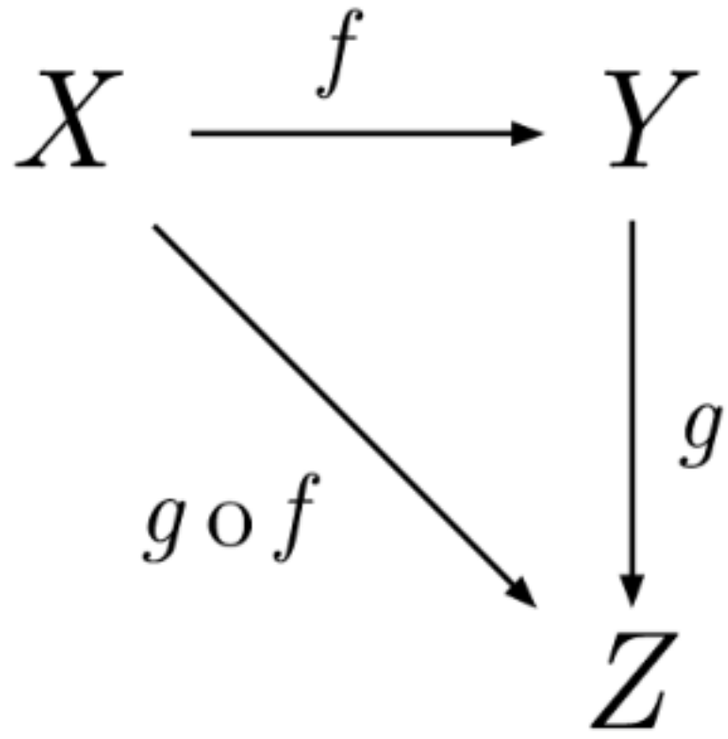
- 只有纯的、没有副作用的函数，才是合格的函数

- 相同的输入永远获得相同的输出（因此可以记忆）

- 合成与柯里化是最主要的运算

函数的合成

- 如果一个值要经过多个函数，才能变成另外一个值，就可以把所有中间步骤合并成一个函数，这叫做"函数的合成"（compose）




```
const compose = function (f, g) {
  return function (x) {
    return f(g(x));
  };
}
```

合成的结合律

```
compose(f, compose(g, h))
// 等同于
compose(compose(f, g), h)
// 等同于
compose(f, g, h)
```

- 函数就像数据的管道（pipe）。那么，函数合成就是将这些管道连了起来，让数据一口气从多个管道中穿过

柯里化

- f(x)和g(x)合成为f(g(x))，有一个隐藏的前提，就是f和g都只能接受一个参数。如果可以接受多个参数，比如f(x, y)和g(a, b, c)，函数合成就非常麻烦

- 这时就需要函数柯里化了。所谓”柯里化"，就是把一个多参数的函数，转化为单参数函数

- 有了柯里化以后，我们就能做到，所有函数只接受一个参数

柯里化

```
// 柯里化之前
function add(x, y) {
  return x + y;
}

add(1, 2) // 3
```

```
// 柯里化之后
function addX(y) {
  return function (x) {
    return x + y;
  };
}

addX(2)(1) // 3
```

函数式编程的优势

- 纯函数式语言中的变量是值的名字，而不是存储状态的单元
- 变量的值是不可变的
- 所有的运算都是产生新的值，而不会修改原来的值（也不影响外界环境）
- 函数式程序表明计算结构但不表明计算过程，因此适合惰性计算
- 函数式编程的函数都是可重入函数，因此适合并行计算

不可变的计算

```
class Point(x: Int, y: Int) {
    override def toString() = "Point (" +x + ", "
+ y +")"

    def moveBy(deltaX: Int, deltaY: Int) = {
        new Point(x + deltaX, y_ deltay)
    }
}
```

- 由于变量不可变，因此无法实现循环，只能用递归

函数式语言的其他特性

- 高阶函数（High-Order Function）
- 偏应用函数（Partially Applied Function）
- 柯里化（Currying）
- 闭包（Closure）

权责让渡

- 随着硬件能力的提高，我们将 越来越多的任务转嫁给语言和运行时。开发者曾经因为速度太慢而排斥解释型语言，现在它们已经随处可见

- 抽象的目的总是一样的：让开发者从繁琐的运作细节里解脱出来，去解答问题中非重复性的那些方面

迭代—>高阶函数

- 如果能够用高阶函数把希望执行的操作表达出来，语言将会把操作安排得更高效

高阶函数

- 参数或返回值为函数的函数

```
make "sum [
    [term a next b]
    [
        if gt :a :b
            [return 0]
            [return add term :a sum :term next :a :next :b]
    ]
]
print sum [[x] [return add :x :x]] 1 [[x] [return add :x 1]] 10
```

- 如何利用sum求[a,b]的整数的平方和?

闭包

- 所谓闭包，实际上是一种特殊的函数，它在暗地里绑定了函数内部引用的所有变量。换句话说，这种函数（或方法）把它引用的所有东西都放在一个上下文里“包”了起来

- A “lambda expression” is a block of code that you can pass around so it can be executed later, once or multiple times.

闭包

- 如果一个函数用到了它作用域外面的变量，那么这个变量和这个函数之间的环境就叫闭包

```
make "f [
    [x]
    [
        make "x add :x :x
        make "fun [
            [k]
            [
                return add :x :k
            ]
        ]
        export "fun
        make "x add :x :x
    ]
]
```

- MUA没有实现闭包
- 左侧的fun里的x并不会表达为f里的x
- 闭包需要在定义函数的地方解析函数
- 闭包需要实现变量引用访问
- f函数结束后，AR不能回收

Java的闭包

- 在Java中，闭包是通过“接口+内部类”实现，JAVA的内部类也可以有匿名内部类
- 在Java中，内部类可以访问到外围类的变量、方法或者其它内部类等所有成员，即使它被定义成private了，但是外部类不能访问内部类中的变量。这样通过内部类就可以提供一种代码隐藏和代码组织的机制，并且这些被组织的代码段还可以自由地访问到包含该内部类的外围上下文环境
- ShareClosure.java

Event Handling

```
btn.addActionListener(new ActionListener() {
   @Override
   public void actionPerformed(ActionEvent e) {
      System.out.println("OK");
   }
});
```

- Code as data — The object passed to the btn is code.

```
btn.addActionListener(event -> System.out.println("OK"));
```

Lambda Expression

```
Runnable run = ()->System.out.println("running");
new Thread(run).start();
```

```
Runnable run2 = ()->{
   System.out.println("running");
   System.out.println("end");
};
new Thread(run2).start();
```

流计算——函数式编程典型应用

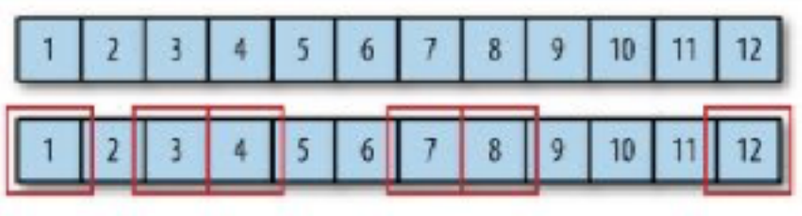
- 流计算：将容器遍历过程中要做的计算植入流，在遍历中计算，而不是把元素拿出来计算
  - lazy way vs eager way

基本构造单元

- 筛选filter
- 映射map
- 折叠/化约fold/reduce

筛选

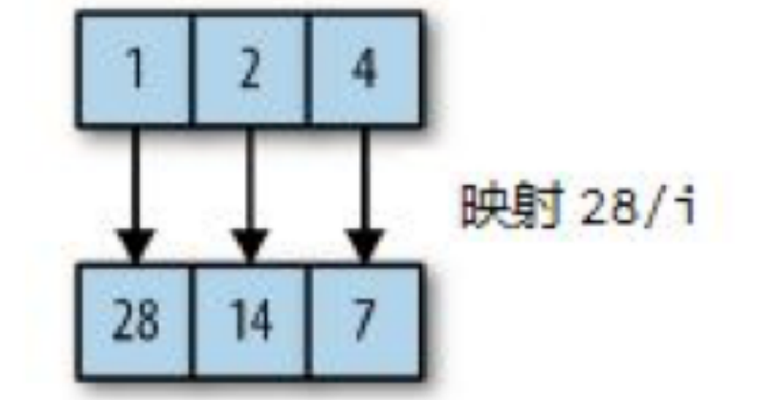
- 根据用户的定义来筛选列表中的条目



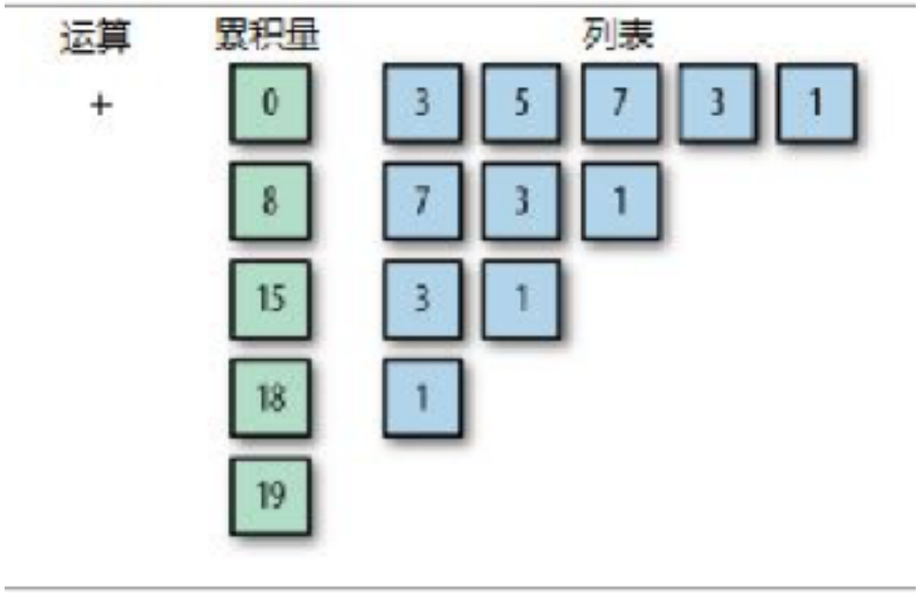

```
public static IntStream factorsOf(int number) {
    return range(1, number + 1)
            .filter(potential -> number % potential == 0);
}
```

映射

- 对容器中的每一个元素执行给定的函数从而变成一个新的容器



折叠/化约



- 用一个二元函数或运算符来结合列表的首元素和累积量的初始值（如果累积量有初始值的话）

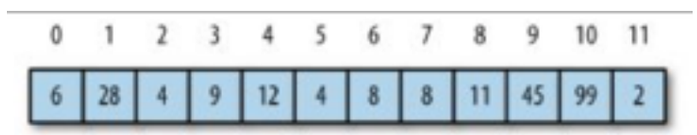
- 重复上一步直到列表耗尽，此时累积量的取值即为折叠运算的结果

- 学习函数式编程，或者任何一种新范式都有一个很大的挑战，那就是在掌握新的构造单元之后，还要善于从问题里“发现”它们的身影，从而抓住解答的脉络。函数式编程不会用很多抽象，但每个抽象的泛化程度都很高（特化的方面通过高阶函数注入）。函数式编程以参数传递和函数的复合作为主要的表现手段，我们不需要掌握太多作为“不确定因素” 存在的其他语言构造之间的交互规则，这一点对于我们的学习是有利的

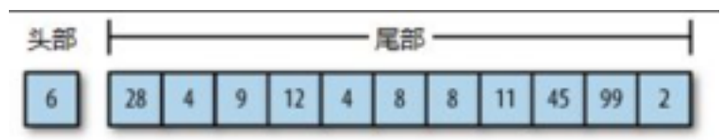
递归

- 让容器反复对自身调用相同的方法，使得容器随着每次迭代不断缩小，直到处理完成

- 命令式语言眼中的数组：



- 函数式语言眼中的数组：



递归

- 递归往往受制于实现而备受质疑
- 但是技术限制总会随着时间而减少或消失
- 尾递归优化
    - 当递归调用是函数执行的最后一个调用的时候，运行时可以在栈里就地更新

Python的函数式编程

- First-class and higher-order functions, which are sometimes known as pure functions.
- Immutable data.
- Strict and non-strict evaluation.
- Recursion instead of an explicit loop state.
- Functional type systems

纯函数

- 普通函数不使用全局变量的话可以视作纯函数
- mersenne = lambda x: 2 ** x -1
- mersenne(17)

高阶函数

- lst = map(int, input().split())

C++的lambda

```
#include<iostream>
#include<map>
using namespace std;
int main(){
    int a,b,c;
    cin>>a>>b>>c;
    using pf=int(*)(int,int);//f是该函数指针类型的一个别名
    map<int,pf>m={
        {0,[](int a,int b){return a+b;}},
        {1,[](int a,int b){return a-b;}},
        {2,[](int a,int b){return a*b;}},
        {3,[](int a,int b){return a/b;}}
    };
    m.insert({4,[](int a,int b){return a%b;}});//假设增加了一个求余运算
    cout<<m[c](a,b)<<endl;
    return 0;
}
```

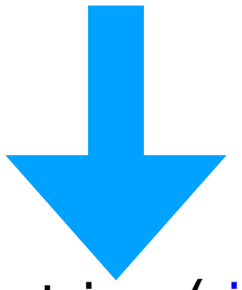
function<T>

```
#include<iostream>
#include<map>
#include<functional>
using namespace std;
int main(){
    int a,b,c;
    cin>>a>>b>>c;
    using pf=int(int,int);//注意没有*
    map<int,function<pf>>m={//注意变为了function模板
        {0,[](int a,int b){return a+b;}},
        {1,[](int a,int b){return a-b;}},
        {2,[](int a,int b){return a*b;}},
        {3,[](int a,int b){return a/b;}}
    };
    m.insert({4,[](int a,int b){return a%b;}});//假设增加了一个求余运算
    cout<<m[c](a,b)<<endl;
    return 0;
}
```

柯里化

```
int test(int a,int b,int c){
    return (a+b)*c;
}

int testForApplication(int c){
    return test(1,2,c);//假设该应用的a、b 参数分别为1、2
}
```



```
function<int(int)>testForApplication(int a,int b){
    return [a,b](int c){return test(a,b,c);};
}

auto i=testForApplication(1,2);//获取参数a、b是1、2的应用对应的只需一个参数的函数
cout<<i(3);//获取该应用下参数c为3的最终结果
cout<<i(4);//获取该应用下参数c为4的最终结果
```