第二章、蹒跚学步：进阶操作

0·总结

1. 类(*class*)与结构体(*struct*)，与隐藏参数(*this*)

二者没有实质性的区别，使用格式为：

```
class Name
   var data = 0
    #数据成员的定义
   function func()
        #语句块
   end
    #成员函数的定义
end
```

会导入一个名为this的参数，可以使用访问符`.`访问该类的成员，如`this.data`，`this.func()`。

2. 全面做好商品类：类的继承(*extend*)、重载(*override*)与隐藏参数(*parent*)

可以使用继承声明派生类与基类间的 *is a* 关系，使用格式为：

```
class A
    var data_A = 0
    function func_A()
        #语句块
    end
end

class B extends A
    var data_B = 0
    function func_B()
        #语句块
    end
end
```

派生类可以使用基类所有数据成员及成员函数，也可以使用重载覆盖基类的同名成员函数，如：

```
class B extends A
    function func_A() override
        #语句块
    end
end
```

派生类会加入一个隐藏的参数*parent*，可以用该参数调用基类成员，如：

```
    this.parent.func_A()
```

3·进货不简单——完善商品类：作用域(Scope)、结构化绑定(structured binding)

进入新的语句块，如`if`、`while`等，会进入新的作用域。更上层作用域的同名变量会隐藏下层作用域的变量，从而无法直接访问。

可以使用`global.varName`访问全局变量，使用`local.varName`访问当前作用域变量。

结构化绑定指的是将数组中的值按位置绑定至小括号括起的逗号表达式列表中，格式为：(表达式1, 表达式2...) = 表达式3，其中，等号右侧的表达式的形式一定为数组，如：`(a,b,c) = {1,3,10}`
结构化绑定是可以嵌套的，相应的，嵌套的另一层需要按照相同的结构嵌套，如：`(a,(b,c)) = {1,{3,10}}`

4·为批量管理做准备：数据结构(list, pair, hash_map)、new、gcnew、构造函数(initialize)与析构函数(finalize)

list是双向链表，使用方式与array类似，具体使用方法参考***API***
pair是由冒号对应的一个键值对，如： 2:3
hash_map为散列表，即映射表，是由映射组成的容器。要求其存储的映射的键的类型必须支持生成哈希值，可使用下标运算符访问散列表中的键对应的值，若键不存在，将自动建立键与 0 组成 的映射。也可以使用点运算符访问散列表中的字符串键对应的值，如与散列表扩展函数冲突则被忽略，若键不存在将触发运行时错误。

new为新建栈对象，遵循*RAII原则*，离开作用域时被摧毁。gcnew为新建堆对象，由垃圾回收器进行回收。在对象被新建时将调用构造函数，被摧毁时调用析构函数。构造函数与析构函数的定义方式如下：

```
function initialize()
    #语句块
end

function finalize()
    #语句块
end
```

5·超市管理系统的诞生——合并收银、进货两部分：引入(import)包(package)与命名空间(namespace)

可以将代码模块化，负责一个模块的源代码进行打包，在文件头加入`package 包名`。在另外的文件头加入`import 包名`，即可导入该模块。

在使用导入模块的功能时，需要使用访问符`.`访问内部的信息，如`包名.data`。也可以使用`using 包名`将内部的信息暴露出来，直接使用。 导入模块时，会自动将模块内信息封装在与包名相同的命名空间内，只能通过访问符`.`访问内部的信息，如`包名.data`。

我们也可以使用命名空间显式进行数据的隐藏，使用方法如下：

```
namespace 空间名
    #语句块
end
```

在命名空间内部，名称空间中只允许引入其他名称空间，变量定义，函数定义，类型定义以及名称空间的定义。

可以使用`using 包名`引入一个名称空间，可以直接访问该空间内部的信息。

6·售卖超市管理系统——把程序打包：编译(compile)与解释(interpretcomp)的细节

除了编译器(由CovScript GUI提供)外，*CovScript*也提供了另外一种运行代码的方式，为解释器(*interpreter*)，即*CovScript REPL*。

1·初步构思——进货商品怎么办：类(class)和结构体(struct)

当我们想要制作一个简单的商品管理系统，前述的所有操作都会难以有效地去按照现实生活的逻辑去进行编程。此时需要一个比较简单的方式来处理一种属于商品的类别，而类/结构体(*class/struct*)刚好能够胜任这份任务。

类(*class*)与结构体(*struct*)是 *CovScript* 用于支持面向对象编程的设计，二者在使用时并无实质性差别，属于用户自定义类型，使用数据与函数来描述一种对象所拥有的数据以及其行为（如处理数据的操作）。一个类的数据称为这个类的数据成员，一个类的函数称为这个类的成员函数。类中只允许变量的定义与函数的定义。

当需要描述一个商品的时候，一般使用商品名、生产地、生产日期、保质期与价格等数据去完整地描述一个商品。此处为了更简便地使用，仅使用商品名(*Name*)与价格(*Price*)作为商品类(*Good*)的数据成员。

```
class Good
    var Name = ""
    var Cost = 0  #成本
    var Price = 0 #售价

    function show_data()
        system.out.println(this.Name + "的成本为" + this.Cost + "元,售价为" +
this.Price + "元")
    end
end
```

其中,*Name* 与 *Price* 是其数据成员，将其初始化为空字符串与零。*show_data* 为其成员函数，可以使用他来输出独属于这个对象的数据。再函数中，我们使用了 *this* 参数与`.`访问符，*this* 参数是派生类中自动插入的，能够调用成员函数的结构实例本身。当我们实例化一个对象，通过 *this* 访问到的成员表示的便是这个对象的成员。

于是便成功定义了一个名为 *Good* 的类型，并可以像 *number*，*char* 等数据类型一样声明之。不同的是，可以通过访问符`.`来直接访问并修改这个对象内部的数据。

例：我们想要定义一件名为“小瓶百事可乐”的商品，其成本为2.3元，售价为3元，并且定义一个成员函数，能够输出相应的数据，则需要进行以下操作：

```
var good = new Good
good.Name = "小瓶百事可乐"
good.Cost = 2.3
good.Price = 3
good.show_data()
```

(因为字符串包括了中文,不要忘记加入预处理语句 *@charset:gbk*)

至此,我们已经学会了如何自定义一个数据类型,并且调用其成员函数,完成一定的操作。

值得注意的是，虽然类看上去只是普通的变量与普通的函数的集合，但编译器会为成员函数添加一个隐式的 *this* 参数。关于这个参数的使用方法及技巧，我们会在之后的小节(2-2)进行详述。

小技巧： 我们常定义 *get* 函数访问一个自定义对象的值，用 *set* 函数修改自定义对象的值。这样不仅能使代码有一定的可读性，也能保证数据的封装性。

2·全面做好商品类：类的继承(extend)、重载(override)与隐藏参数(this，parent)

我们能用商品名来描述商品，但仅有"商品"显然不能够有所分别的商品全部分离开来，针对不同类别的商品，我们需要添加额外的数据来描述。而如果我们对于每个商品都单独声明一个类，会显得十分冗余，且难以进行系统性的修改，缺少代码的复用性。因此我们可以使用继承(extend)，来描述一种`is a`关系，让一个类基于另一个类来定义，从而形成一种树状的关系网。已有的类被称为基类，新定义的类被称为派生类。

派生类是在基类的基础上拥有额外数据成员或成员函数，例如若是饮品(*Drink*),则会有标注体积,故我们可以新建一个饮品类，并额外添加一个数据成员 *Volume*。

```
@charset:gbk

class Good
    var Name = ""
    var Cost = 0
    var Price = 0

    function show_data_a()
        system.out.println(this.Name + "的成本为" + this.Cost + "元,售价为" +
this.Price + "元")
    end

    function show_data_b()
        system.out.println(this.Name + "的成本为" + this.Cost + "元,售价为" +
this.Price + "元")
    end

end

class Drink extends Good
    var Volumn = 0
```

```
    function show_data_b() override
        this.parent.show_data()
        system.out.println("体积为" + Volumn + "ml")
    end

end

var drink = new Drink
drink.Name = "苏打水"
drink.Cost = 4
drink.Price = 7
drink.Volumn = 450
drink.show_data_a()
drink.show_data_b()
```

与第一小节的例子一样,定义一个商品(*Good*)类,拥有数据成员商品名(*Name*)与价格(*Price*),并且拥有一个成员函数 show_data 。饮品(Drink)类是商品(Good)类的派生类，他不仅包括商品(Good)类的数据成员，还额外含有数据成员体积(Volume)。

倘若需要对所有的商品类进行修改，增加一个数据成员为利润(*Profit*)，利用继承关系进行定义的方法则能够直接在基类添加一个数据成员 *Profit* 即可。

派生类可以同样使用基类的成员函数，如上例 *drink.show_data_a()* ，则是可以输出：

苏打水的成本为4元，售价为7元

但若想要将信息全部输出，包括派生类独有的数据 *Volume* ，显然原有的成员函数是不够的。此时我们可以重新定义一个函数，如 *show_data_c()* ，但这个方式会造成大量无用的函数存在。我们可以用重载(*Override*)来使派生类继承得到的成员函数保持相同的函数名，却可以执行独有的函数体。具体代码格式如下：

```
    function func(args) override
       语句块
    end
```

使用重载(*override*)方法可以使派生类与基类拥有同名函数，具体实现形式却有所差别，提高代码重用性，也能使代码可读性更高。

每个类会自动一人一个名为 *this* 的成员，可以用该变量访问该类的所有数据成员或成员变量，如 `this.data`。通常，在成员函数中使用this参数来访问该类的成员数据，以与函数的参数加以区分。也会在派生类的成员函数中利用this指针访问本身的成员函数，来与基类的成员函数加以区分。

派生类会自动引入一个名为 *parent* 的成员，可以使用该变量来调用基类的成员函数。如在上例中的 `this.parent.show_data_b()` ，便可以复用基类的函数，输出基类的信息后，再输出派生类的信息。

至此，我们已经能够利用继承设计一个比较良好的类关系，根据这个模式，我们可以从商品(*Good*)开始创建基类，并包括多个成员数据，如商品名，价格等。再以商品(*Good*)为基类创建派生类，如饮品，用品，食品等。如若需要更详细的分类，我们还可以以某个派生类为基类继续创建他的派生类。

3·进货不简单——完善商品类：作用域(Scope)、结构化绑定(structured binding)

我们已经将商品类用以下代码实现:

```
@charset:gbk
class Good
    var Name = ""
    var Cost = 0
    var Price = 0

    function show_data()
        system.out.println(this.Name + "的成本为" + this.Cost + "元,售价为" +
this.Price + "元")
    end
end

class Drink extends Good
    var Volumn = 0

    function show_data() override
        this.parent.show_data()
        system.out.println("体积为" + this.Volumn + "ml")
    end
end

class Food extends Good
    var Weight = 0

    function show_data() override
        this.parent.show_data()
        system.out.println("质量为" + this.Weight + "g")
    end
end
```

其中,商品(*Good*)类为基类，饮品(*Drink*)类与食品(*Food*)类为派生类，各自增加了一个独有的数据成员，且均重载了基类的成员函数 *show_data()*。这样，我们不需要关心哪些类究竟包括哪些函数，只需要记住如果我想获得一个类的所有信息，只需要使用 *xxx.get_data()*(*xxx*为这三个类的任意一种)即可。

为了使代码可读性更高，我们可以使用一个函数 `show_data(good)`，直接负责该任务。

```
function show_data(good)
    good.show_data()
end
```

该函数名与三个商品类的显示信息的函数名相同，在使用的时候却不会混淆，原因是我们正确地利用了作用域:

作用域能够限定我们寻找并使用变量或函数的范围，编译器会在当前作用域搜索尝试访问的变量，如果没有，将会进入下一层查找，直至搜索到最外层(全局)的变量。

```
var a = 3
var b = 4
while a > 0
    var b = 1
    system.out.println("b的值为" + b)
end
```

在进入 *while* 语句块之前，声明了一个值为4的变量 *b* ，进入 *while* 语句块之后，声明了值为1的变量 *b*， 因此在循环体内尝试访问变量 b 时，会从该层作用域内往外寻找，即会输出该层 *b* 的值1。

一般，我们将一个语句块划为一层，如 *if*、*while* 等。特别地，我们可以使用 `global` 直接访问最外层的变量，使用 `local`访问当前作用域的变量，如 `local.num1`、`global.character`。类(class)与结构体(struct)通过访问符`.`与该方法类似。

但对于中间的作用域，甚至是整个文件的作用域，暂时无法高效且标准地进行访问，我们后面会进行命名空间(*namespace*)的讲解，可以有效解决这个问题。

商品的信息获得已经变得如此简便，针对商品信息的定义却还是一个一个地赋值，显得十分冗余，我们可以使用结构化绑定(*structured binding*)来简化这个过程。

结构化绑定指的是将数组中的值按位置绑定至小括号括起的逗号表达式列表中，格式为：(表达式1, 表达式2...) = 表达式3，其中，等号右侧的表达式的形式一定为数组，如：

```
var food = new Food
(food.Name, food.Cost, food.Price, food.Weight) = {"奥利奥巧克力", 3, 5, 100}
get_data(food)
```

便可以将多个赋值语句整合在同一条语句中。结构化绑定可以嵌套，如 `(food.Name, food.Cost, (drink.Name, drink.Price)) = {"面包", 3, {"奶茶", 5}}`。有了这个功能，我们能够更有效地进行商品对象的定义与使用。

4·为批量管理做准备：数据结构(list, pair, hash_map)、new、gcnew、构造函数(initialize)与析构函数(finalize)

前三小节将商品类做了完整的定义，现需要制作一个管理系统将商品进行有效的管理，包括商品的定义，商品进货及卖出操作的定义等，后面我们将依次讲解。

为将多个商品进行储存，我们需要选择一种容器。之前学习的 *array* 是一个不错的选择，此处也可以选择 *list* (*list* 为双向链表， *array* 为双向队列)。

list 与 *array* 的操作十分相似，均包括插入操作(*push_back*)与删除操作(*pop_back*)。**(详见API)**

商场管理系统主要由两个部分组成，一个是商品的管理，这个将由一个或若干个 list 负责。另一个是金钱的管理，这个直接使用一个变量进行储存即可。

```
class Supermarket
    var goods = new list
    var money = 0
end
```

为提高封装性，为商品(*Good*)类添加以下成员函数：

```
    function get_Name()
        return this.Name
    end

    function get_Cost()
        return this.Cost
    end

    function get_Price()
        return this.Price
    end
```

以下定义*Supermarket*类的成员函数:
定义初始化:

```
    function init(Money)
        this.money = Money
    end
```

定义查看收银台金额：

```
    function get_money()
        return this.money
    end
```

定义进货：

```
    function add_good(good)
        if this.money < good.get_Cost()
            system.out.println("余额不足,进货失败")
            return                                  #跳出函数
        end
        goods.push_back(good)                       #添加商品
        money -= good.get_Cost()                   #扣去成本
    end
```

进货时要注意余额是否足够达到商品的进货价，否则进货失败。此处将*goods*(*list*)作为容器储存商品，进货时，使用 `list.push_back()` 即可。要访问所有商品时，遍历一遍 *goods* 即可。

定义展示商品：

```
function show_goods()
    var num = 1
    foreach it in goods
        system.out.println("第" + num + "件商品的信息为：")
        it.show_data()
    end
end
```

利用 *foreach* 遍历 *goods* 列表，并显示所有的信息。

定义出售商品(以名字选择)：

```
function sell_good(name)
    var it = goods.begin                                #迭代器
    while it != goods.end                               #终止条件
        if name == it.data.get_Name()
            money += it.data.get_Price()
            it = goods.erase(it)                        #从容器中移除
            system.out.println("出售成功")
            return
        end
        it.next()
    end
    system.out.println("未找到商品")
    return
end
```

it 是迭代器，所有的容器都有迭代器，如 *arr.begin* ，迭代器是遍历容器最简单且最有效的方式，常用的方式如下：

```
var it = container.begin
while it != container.end #it.end为容器最后一个元素的下一个位置
    语句块
    it.next()访问下一个元素
end
```

我们通过遍历的方式在容器中定位商品，并做出相应的操作。**(list 的更多操作请查阅API)**

list 可以作为储存商品的容器的选择之一。若是一个更简单的对象，仅考虑键与值，我们可以考虑使用 hash_map 作为容器进行储存。

散射表(*hash_map*)是由映射(*pair*)构成的，映射的形式是用冒号分割的键值对，如 `2:3`、`'a':1`等。而散列表则是可以直接通过下标访问，获取键所对应的值，如使用`map['a']`获得值1。

现在我们已经建立了管理系统的几个操作，接下来可以进行功能的测试，我们需要实例化一个对象进行相应的操作，和成员数据 *goods* 一样，需要用 *new* 实例化一个对象。实例化对象的方式有两种，一种是常见的 *new* ，如`var foo = new Foo`，在栈上新建对象，离开作用域时会自动回收。另一种是 *gcnew*，在堆上建立对象，返回指向该对象的指针，如`var foo = gcnew Foo`(*foo*是指向*Foo*对象的指针)。

使用*new*/*gcnew*时，会调用对象的构造函数，利用这个特点，我们可以在声明对象的时候便为数据成员赋值，即初始化，一般赋值为0。声明及使用方法如下：

```
class Foo
    var data = 0
    function  initialize()   #构造函数的声明
        system.out.println("使用了构造函数")
    end

    function init(Data) #初始化
        data = Data
    end
end

var foo = new Foo              #实例化对象
foo.init(2)
system.out.println("a.data的值为" + a.data)
```

此时会输出：
使用了构造函数
*a.data*的值为2

对象在被实例化的时候会调用其构造函数，即 *initializer*。类似的，在被摧毁时，也会自动调用其析构函数，即 *finalizer*，使用方法如下：

```
    function finalize()
        语句块
    end
```

在没有人为声明构造函数与析构函数时，编译器会自动添加语句块为空的构造函数与析构函数。

栈对象，即通过*new*产生的对象，会在离开作用域的时候被销毁，调用析构函数。如在*if*、*while*语句块内声明的对象，会在离开语句块时调用析构函数。堆对象，即通过*gcnew*产生的对象，会由垃圾回收器自动回收，在被回收的时候被摧毁，调用析构函数。

利用构造函数与析构函数，我们能在需要的时候对对象内部的数据进行更严谨的控制。

在商品管理系统中，暂不需要进行构造函数或析构函数的使用，但我们需要知道，在*new*的与离开作用域的过程中，对象会自动调用构造与析构函数，我们可以利用这个特点进行数据的维护。

```
var supermarket = new Supermarket
supermarket.init(100)             #初始化拥有金额
var food = new Food
(food.Name, food.Cost, food.Price, food.Weight) = {"苹果", 10, 15, 500} #初始化一个
食品类
supermarket.add_good(food)        #进货
var drink = new Drink
(drink.Name, drink.Cost, drink.Price, drink.Volumn) = {"苏打水", 4, 7, 450} #初始化
一个饮品类
supermarket.add_good(drink)       #进货
supermarket.show_goods()          #查看信息
supermarket.sell_good("苹果")      #卖出商品
supermarket.show_goods()          #查看信息
```

到这里，我们已经实现了商品管理的所有基础功能，能够实现进货与卖出商品，但我们还需要良好的交互模式，使程序的实用性更高，将在下一小节进行讲解。

5·超市管理系统的诞生——合并收银、进货两部分：引入(import)包(package)与初识项目(program)

管理系统需要一直运行，故将程序主体放在一个死循环内，直至选择推出或强制退出。提供的操作选择有： 1.进货
2.出售商品 3.查看商品信息 4.查看余额信息 5.退出系统
因进货的操作相对冗杂，影响主体代码的直观性，故可以提前定义进货商品的定义函数。

```
function new_good()
    system.out.println("请输入 商品类别(food/drink) 商品名 商品成本 商品售价 商品体积/
质量")
    var arr = string.split(system.in.getline(), {' '})  #将输入信息以空格分割开,以数
组形式储存在arr中,具体使用方法详见string API
    var flag = true                                     #定义标志位,读取错误信息后利
用标志位给予警告
    var i = 0
    if array.size(arr) != 5                             #读取数据格式错误
        flag = false
    end
    switch arr[0]
        case "food"
            var food = new Food
            (food.Name, food.Cost, food.Price, food.Weight) = {arr[1],
string.to_number(arr[2]), string.to_number(arr[3]), string.to_number(arr[4])}
            return food               #跳出函数,返回值为 Food 类型的对象
        end
        case "drink"
            var drink = new Drink
            (drink.Name, drink.Cost, food.Price, food.Weight) = {arr[1],
string.to_number(arr[2]), string.to_number(arr[3]), string.to_number(arr[4])}
            return drink              #跳出函数,返回值为 Drink 类型的对象
        end
```

```
        default
            flag = false
        end
    end
    if flag == false
        system.out.println("输入数据有误")  #发出警告
        return false
    end
end

function main()
    var sup = new Supermarket
    system.out.println("欢迎使用商品管理系统,请输入收银台的初始金额：（单位：元）")
    sup.init(string.to_number(system.in.getline()))                    #初始化，输
入收银台金额
    system.console.clrscr()                                            #清除屏幕
    while true
        var flag = true
        system.out.println("1.进货\n2.出售商品\n3.查看商品信息\n4.查看余额信息\n5.退
出系统")
        var input = system.in.getline()                                #读取输入
        switch input[0]                                                #获取的输入
默认是字符(串),只读取第一位
            case '1'
                var good = new_good()
                if good != false
                    sup.add_good(good)
                    system.out.println("进货成功,购入商品信息:")
                    good.show_data()
                    system.out.println("收银台当前金额:" + sup.get_money() + "元")
                end
            end
            case '2'
                sup.sell_good(system.in.getline())
                system.out.println("出售商品成功")
                system.out.println("收银台当前金额:" + sup.get_money() + "元")
            end
            case '3'
                sup.show_goods()
            end
            case '4'
                system.out.println("收银台的剩余金额为：" + sup.get_money() + "元")
            end
            case '5'
                flag = false
            end
            default
                system.out.println("请输出1-5中的数字")
            end
        end
        if flag == false
            break
        end
        system.out.println("\n\n请点击回车键确认")
```

```
        system.in.getline()                                    #读取
输入
        system.console.clrscr()                                #清除
屏幕
    end
end

main()
```

其中，`system.console.clrscr()` 为清除控制台信息，使交互界面更有序。`system.in.getline()`获得的输入是以字符串的形式储存在变量中，而`string.to_number(str)`能够将类型变为number，以方便数学中的加减。
一个能够正常运行的收银管理系统程序已经编写完成，但这个程序在功能上还只是个雏形，但可以从以下方面进行改进：
1.每次程序的数据能够保存，需要利用文件流，使用文件进行数据的储存。
2.商品的定义不完善，可以更详细地定义商品类，或对食品(*Food*)类与饮品(*Drink*)类继承细化获得新的派生类。
3.无法持续录入商品信息(进货)，在 *while* 语句块中进行修改即可。

我们可以为刚刚诞生的商品管理系统欢呼，因为这是我们使用*CovScript*编写的第一个有一定规模，有特定功能的程序。而这已经写完的代码，也就是我们定义完毕的几个类，也可以在以后的程序中再次使用，只需要我们将这个项目进行“打包”：

我们可以将定义的商品类单独放在一个文件，并在代码首行写上一行： `package good`，代表着为这份代码打了个包，并命名为 *good*，为保持良好的习惯，我们将文件名也取名为 *good*，全称为 `good.csp`。注意，我们平时的代码文件均为 *csc* 后缀，而打包的文件为 *csp* 文件，旨在告诉编译器这份文件是头文件，可以使用 *import* 导入从而复用代码。

在导入包时，我们需要使用命名空间(*namespace*)规则来使用对应的函数或类，如`var food = new good.Food`，其中，`good`是包的命名空间，只有通过访问符`.`才能够访问包内部的函数或数据。这样能够将不同包内部相同名称的函数区别开来，若同时导入两个包 *testA*， *testB*，且均包含*test*函数，则可以利用如*testA.test()*的方式访问 *testA* 包中的 *test* 函数。

与包的作用域类似，我们可以在任意地方定义命名空间，如：

```
namespace test
    语句块
end
```

在该命名空间内部只能引入其他命名空间、完成函数，变量的定义、类型定义及名称空间定义，若要访问该命名空间内部的对象，需要使用访问符来访问之。

特别地，我们可以在任意处使用 *using* 导入某个命名空间，如 `using good`， 则 *good* 包中的所有信息都直接暴露出来，可以不通过访问符进行访问。命名空间也是如此，在定义了命名空间后，可以使用 *using* 暴露该空间，并直接访问内部对象。但该方式容易导致冲突，不建议一个程序同时导入两个及以上的命名空间，除非能够保证一定不会有冲突产生。

新创建一个文件，在首行写上`import good`，并把 *good.csp* 文件中 *main* 函数的定义与调用 *main* 函数的语句剪切到该文件中，保存为 *good.csc*，运行。可以看见程序正常运行，那么打包与导入包的操作都已成功。如果

想在其他的程序导入 *good* 包，只需要在同一目录，并 `import good` 即可。一个程序可以任意次导入(import)包，但每个程序只能打包(*package*)一次。

我们往往会将实现一类功能或定义一种或一类对象放在同一个文件中，并称之为一个模块。如我们可以针对字符串编写相关的操作(函数)，放在一个文件里，并打包为 *str_tools*，在我们需要利用字符串的相关功能时，则加入以下代码： `import str_tools`即可。有了命名空间与包的定义，我们能够将程序进行模块化，不同的模块执行不同的功能，使代码更具有层次性，可读性更高，同时也会更加符合现实生活的思维逻辑。在文件结构上，可以将相同层次或相似功能的模块(源文件)放在同一个文件夹内，从而使程序更有层次感，可维护性更高。

6·售卖超市管理系统——把程序打包：编译(compile)与解释(interpretcomp)的细节

我们所编写的代码被称为源代码，是便于人类阅读的而遵循高级语言(如我们正在学习的*CovScript*)的语法规则所编写出来的文字集合。我们需要通过**编译**(*compile)来将程序转化成机器语言，从而使电脑运行我们所编写的程序。而目前我们编写完代码并保存后，点击的"运行"按钮则是执行了两个步骤：第一步是将你编写的源代码进行编译，生成机器码。第二步是使电脑运行该机器码。

除了编译器(由CovScript GUI提供)外，*CovScript*也提供了另外一种运行代码的方式，为解释器(*interpreter*)，即*CovScript REPL*。
解释器会将源代码翻译成更高效率的中间代码，并立刻执行。可以将解释器比作黑盒子，每当我们提供源码的输入，便会得到相应的结果。