第一章 开端：语法与基础

0·总结

1·造一个简单计算器：数值(number)的四则运算(运算符)

没有了计算功能，计算机就会失去灵魂。我们不难想象，大型程序是由一个个计算语句组成。所以，我们探索 CovScript 的第一步将从简单的四则运算开始。

运行以下代码，它作用是：将a的值打印到屏幕。

```
using system
var a = 16
out.print(a)
```

在解读这段代码前，我们先了解一些知识：

```
var a = 16
```

var 是 variable（变量）的缩写，`var`关键字表示声明变量，变量的类型由`=`后的内容决定。
变量在所有编程语言中都占有举足轻重的地位，几乎所有数据在编程语言中都需要由变量来保存。我们可以将变量理解为“杯子”，而内容（数据）被“装”在杯子里。

CovScript 对这段代码的执行流程是：

> 我们声明了变量`a`，并向变量`a`中“装入”了一个值为 16 的数据，CovScript 自动将 16 识别为数值类型。
> 至此，我们得到了一个数值类变量`a`，**目前**它的值为 16。

```
out.print(a)
```

而在之前的 hello word 演示代码中，我们知道`out.print()`是可以向屏幕打印内容的。只是这里打印的是变量的内容，而变量是不需要加双引号的。

不同的变量对应了不同的用途，虽然它们都是用`var`关键字进行创建，但是在具体操作中还是会因为变量类型不同而产生不同的情况。以下是CovScript常见类型变量表：

变量类型	含义	取值
number（数值类型）	表达数值	实数域－∞ ~ ＋∞
boolean（逻辑类型）	表达二元逻辑	true或false
char（字符类型）	表达单个字符	ASCII全体字符，如：'A'，'a'，'?'，'\n'
string（字符串类型）	表达多个字符	由字符构成的序列，如："CovScript"

变量类型	含义	取值
...	...	...

在代码运行过程中可以通过以下方法来确定某个变量的类型

```
out.print(type(a)) # 通过这种方式来得到某个变量的类型
```

对于以上的代码，会输出`cs::number`，代表变量a是一个number类型的变量。可以自己尝试对于变量的不同赋值，会使变量的属性变为什么类型。

让我们回到刚才的程序

```
using system
var a = 16
out.print(a)
```

细心的读者可能会发现：之所以说目前`a`的值为 16，是因为变量还可以进行更改。
现在，我们尝试用赋值语句来修改`a`的值。向之前的代码添加一句：

```
using system
var a = 16
a = a + 4
out.print(a)
```

显而易见，代码的运行结果为 20。
我们再聊聊赋值语句：

```
a = a + 4
```

对于此处的加法，此处理解为数学意义上的加法。
几乎所有编程语言的**赋值语句**都设计得大同小异：我们以人类阅读从左至右的习惯，读出这行话为“a 等于 a 加 4”。不难猜出，赋值语句是要让程序将`a`的值变为`a + 4`。
进一步解释，赋值语句是要将 = 右侧内容，写入 = 左侧内容，从而改变左侧变量的值。
CovScript 对这段代码的执行流程是：

> CovScript 先计算 = 右侧的值，再将a的值修改为右侧计算出的值

值得注意的是：这里的 4 可不是变量。4 就是常理中认为的 4，好比一升水就是一升水，我们并没有说这是“某瓶里的一升水”。因此这里的“水”可以没有“瓶子”，也就不是变量——但“不是瓶子里的水”并不会影响“装瓶”，不是变量的 4 也不会影响计算加法与赋值。

有了 var 关键字和赋值语句，我们可以完成最简单的加法计算器了！虽然它只能计算固定数值（以下是计算 16 + 3 的和）。

```
using system
var a = 16, b = 3
var c = a + b
out.print(c)
```

如果想要加入输入的功能，只需使用 **in.input()** 的语句。此代码会从键盘读入两个数，输出它们的和。这样，加法计算器就完成了！同理你可以完成减法，乘除法的计算器。

```
using system
var a = new number, b = new number, c = new number
a = in.input(); b = in.input();
c = a + b
out.print(c)
```

new number意为：当前并不想为此变量赋值（因为要在之后取输入值），但又必须指明该变量是何种类型。此时可以使用new关键字指明该变量类型为number。试想，声明的变量只是较为简单的 number，可以使用诸如 var a = 123 等来表示 a 的类型，所幸 123 的内容并不多。如若声明更为复杂的类型，不使用 new 关键字来表示声明变量的类型，将会写出很冗长的代码。

in.input()会将从键盘读入的内容返回给 CovScript，你只需要用赋值语句，也就是 = 号，将输入的内容"送"给你的变量。

值得注意的是： 第 3 行代码中，每句句尾添加了引号;，它的作用和中文中的句号类似，表示一句话的结束。这是因为 CovScript 会默认将一行代码识别为一条语句，如果一定要将多句写入一行，你需要手动为它们标记语句的结束。

聪明的读者会发现两个细节：第 2 行的连续声明，和第 3 行的声明。也许会因此将赋值和变量声明联想在一起（毕竟它们都有 = 号）——它们的确有关系，但现在并不适合阐述，你只要会用就行了。但你仍可以在wiki上发现关于变量初始化的细节，和赋值的定义。

2·计算器进一步优化：判断(if, ?:)和字符串(string)

让我们再仔细想想，如果仅用上一节提及的知识制作除法计算器的不完美之处：它在运算除法时并不会检测除数是否为 0，但合格的计算器需要这个功能。

本节我们来为除法计算器添加此功能。

```
using system
var a = new number, b = new number, c = new number
a = in.input(); b = in.input();
if b==0
    out.print("error: can not divide 0.")
    exit(0)
else
```

```
    c = a / b
    out.print(c)
end
```

我们使用 **if else** 来实现判断功能。
if 判断的格式为：

```
if 条件表达式
    代码块 1
else
    代码块 2
end
```

if 与其后的判断条件使用空格隔开。如果条件表达式为真，则执行 if 后的代码块1；反之执行 else 后的代码块2，并以 end 作为整个条件判断的结尾。
第 6 行的 **exit(0)** 表示执行到此时，结束整个程序。

一个简易的除法计算器完成了！但有挑剔的读者可能会觉得上面的错误信息不够详细。是的，我们可以做得更详细，但取而代之的是更长的字符串。比如除以 0 的时候，我们希望告诉用户，到底哪个数不能除以 0。
为了更美观地使用字符串，我们可以定义一个名叫“错误信息”字符串，以增加代码的复用程度。
部分代码如下：

```
...
var err_info = new string
err_info = "error, the zero can not divided by "
...
if b == 0
    out.print(err_info + a)
end
```

在解释以上代码片段前，先介绍基础的字符串相关知识：
字符串的类型名为**string**，支持使用加号 **+** 拼接前后两个字符串。
比如，我们可以令`var str = "hello" + " " + "world"`，那么 str 的值将是 "hello world"。
对以上代码片段，我们仅对 out.print() 里的内容做解释。

> 在`out.print()`中`err_info`被视为字符串，而本该为`number`类型的`a`也被视为字符串。实际上，`a`在这里被转换了成了 `string` 类型。

如果你这些解释存在疑虑，也许后面的内容能立刻解答你的疑惑。不妨抱着疑虑继续往下读。

如果要把除法功能做的更严谨，我们知道除以 0 时是可以产生无穷大的。进一步，我们默认 0 / 0 中，除数与被除数同阶，那么 0 / 0 将为 1 。
实现这种做法，需要使用嵌套的循环，以达到分类讨论的效果。

```
using system
var a = new number, b = new number, c = new number
a = in.input(); b = in.input();
if b == 0
    if a > 0
        c = "+inf"
    end
    if a  == 0
        c = 1
    end
    if a < 0
        c = "-inf"
    end
else
    c = a / b
end
out.print(c)
```

像如上的结构，称之为 if 的嵌套结构。你可以在任何位置嵌套使用 if。显然，if 判断可以没有 else（即条件为假则不执行任何内容）。
第 4 行中的==符号意为判断两个值是否相等（因为 = 号意为赋值，故设计语言时采用 == 号）。
也许有读者对第 6 行有所疑问：为什么一个 number 类型的变量，却能等于一个字符串呢？这是因为 CovScript 是动态强类型语言，变量的类型在初次声明之后是可以被更改的。

对于上面的代码，我们还能简化

```
using system
var a = new number, b = new number, c = new number
a = in.input(); b = in.input();
if b == 0
    if a  == 0
        c = 1
    else
        c = a>0 ? "+inf" : "-inf"
    end
else
    c = a / b
end
out.print(c)
```

解释代码前，我们先介绍一中功能和 if 类似的语句：**三目表达式。**
三目表达式的格式为：

```
条件表达式 : 表达式1 ? 表达式2
```

它的值取决于条件表达式。条件表达式为真，则三目表达式的值为 表达式1，否则为 表达式2。

让我们再回到以上代码的逻辑：

> 在 b 为 0 的情况下，如果 a 为 0，则 c = 1，如果 a 不为 0，则为 c 根据情况进行赋值：当 a > 0 时，c 为 "+inf"，否则为 "-inf"。

涉及到以条件取值时，可以使用三目表达式，这样可以极大地简化代码量，而不必重复地写 if 语句。

3·增加数列求和计算、科学计算等功能：循环(for, while, loop)与常量(constant)

到目前为止，我们的计算器也只能每次进行一次运算，如果我们想进行批量的运算呢？比如说数列的求和。
尽管许多数列有各自的求和公式，但为了计算器的普遍适应，这里我们不采用求和公式，而是将数列中的每一个元素逐次相加求和。

如果你没有学过数列，也没关系，下面是数列的简单定义：

> 数列（sequence）是一列有序的数。数列中的每一个数都叫做这个数列的项。第 0 个数称为第 0 项，第 1 个数称为第 1 项，第 n 个数称为第 n 项。通常第 0 项又称首项。

最朴素的思路是：利用循环，逐次将每一项计算一次加法，将总和记录下来。代码如下。

```
using system

var sum = 0, n = new number
n = in.input()

var i = 0
while i < n
    var a_i = in.input()
    sum += a_i
    i++
end

out.print("the sum of sequence = "+sum)
```

我们使用了 **while 循环** 来达到循环执行代码的目的。
while 循环格式为：

```
while 条件表达式
    语句块
end
```

> 当条件表达式为真时，便执行一次语句块。每当执行完一次语句块，便会判断条件表达式是否仍然成立。直至条件表达式为假，从 end 处退出 while 循环。

我们来看代码，首先我们读入了一个变量 n，代表此数列由 n 个元素构成。接下来每次循环中执行相应的操作。
控制循环次数最常用的手段之一，便是使用循环控制变量。此处我们用 i（index的缩写） 来表示循环变量。

每执行一次循环，便在第 10 行使 i++。 `i++` 是 `i = i + 1` 的简便写法。

而对于每次循环，我们都将读入 a_i，然后将 sum 累加上 a_i。这样，经过循环累加的 sum 便是此次数列的总和。`sum += a_i` 是 `sum = sum + a_i` 的简便写法。

最后，我们使用 + 号将字符串 "the sum of sequence = " 与刚才求到的总和 sum 拼接在一起。

注意： 这里的 + 号不是加法意义上的加号，而是字符串的拼接。还记得上一节我们提到的动态类型吗？实际上在 out.print() 里，所有类将变成字符串类，你可以按需求将它们拼接。（事实上这是我第二次强调字符串拼接了）

当然，类似的写法还有 loop until 循环。

```
var sum = 0, n = new number
n = in.input()
var i = 0
loop
    var a_i = in.input()
    sum += a_i
    i++
until i >= n
end
```

loop until 循环格式为：

```
loop
    语句块
until 条件表达式
```

> 执行完语句块后，如果不满足条件表达式，则回到 loop，继续执行语句块。

注意： while 循环和 loop 循环的区别不仅在于执行取决于条件的真假，而且一个是先判断后执行，另一个是执行完后判断。 你甚至可以去掉 until 一行，转而用 end 关键字替代，这样就变成了 loop end 循环，需要手动设计代码跳出。

既然使用循环变量来控制循环次数是如此常见的套路，有没有专门为循环变量设计的循环语句呢？就像备胎是那么常见，乃至产生了许多为放下备胎而设计的汽车。 for 循环也许能让您满意。

```
using system

var sum = 0, n = new number
n = in.input()

for i = 0, i < n , i++
    var a_i = in.input()
    sum += a_i
end

out.print("the sum of sequence = "+sum)
```

for 循环的格式为：

```
for 循环变量 = 初始值，条件表达式， 后处理表达式
    语句块
end
```

不难发现，for 循环的设计刚好满足了普通计数循环的需求：一个循环变量，每次执行完循环对循环变量的更改，以及所有循环都有的判断条件。

> 首次执行 for 循环时，for 会帮你创建一个**仅用于当前 for 循环**的循环变量，不管是否会进入循环，都会为其赋初始值；满足条件表达式则进入循环体，执行一次循环；在执行完语句块所有代码后，执行后处理表达式，并进入下一次循环的条件判断。

注意： 除了赋初值的行为只会在首次运行 for 循环时发生，其余两种行为则会在每次循环的始末执行。

也许至此，有读者认为我们还是不能彻底掌控循环（尤其是会使用汇编语言的读者）。CovScript 还提供了控制语句，以便随心所欲地掌控你的循环。
break 语句： 立即跳出当前循环。break 语句格式为：

```
任意循环语句
    break
end
```

continue 语句： 立即结束此轮循环，进而执行下一轮循环。continue 语句格式为：

```
任意循环语句
    continue
end
```

注意： continue 和 break 只允许在循环体中的语句块内编写。

也许以上的数列计算太过娱乐了，那么我们再来做一些科学计算会使用到的功能。比如，一些计算器会有 exp(x) 功能，它表示计算 e^x 。下面是自然常数 e 的介绍。

> e，自然常数，为数学中一个常数，是一个无限不循环小数，且为超越数，其值约为 2.718281828459045。

让我们写一个 exp(x) 的功能。

```
using system
constant e = 2.71
var x = in.input()
out.print(e^x)
```

其中，constant是对常量的声明。你可以理解为与变量相对应的概念：不可更改的量。使用常量，可以防止你的代码更改常用的定值。当然，这个习惯的好处绝不仅仅如此。

同理，你现在能写出拥有圆的周长、面积计算功能的计算器了。

4·功能整合——完善你的多功能计算器：分支(switch)与函数(function)

至此，我们介绍并实现了许多不同功能的计算器，但它们都是单独的程序。有没有什么办法能够将这些功能整合为一个程序呢？ 比如，我们将四则运算整合在一起，很容易想到利用 if 和 loop 来实现此目的。

```
using system

constant SELECT_MESSAGE = "+: addition\n"
                        + "-: subtraction\n"
                        + "*: Multiplication\n"
                        + "/: division\n"
                        + "exit: exit calculator\n";

var a = new number, b = new number ,c = new number

loop
    out.println(SELECT_MESSAGE)
    var opt = in.input()
    boolean flag = false

    if opt == "+"
        ...
        flag = true
    end

    if opt == "-"
        ...
        flag = true
    end

    ...

    if opt == "exit"
        break;
    end

    if flag == false
        out.println("please enter correct key.")
    end
end
out.println()
```

> 第 12 行中println()意为 print line（打印一行），其功能仅比 print 在打印内容末尾多添加了一个'\n'（换行符）。

当然，这很简单。唯一值得一提的是，我们用到了一个开关变量 flag，表示用户是否进入了正确的选择分支，如果以上的选择都未选择，那么只好输出一句 "please enter correct key." 。

但我们在本节要介绍的是，对于处理多个单选分支更为“专业”的语句：switch。

switch 分支格式为：

```
switch 变量
    case 值1
        语句块1
    end
    case 值2
        语句块2
    end
    ...
    default
        语句块n
    end
end
```

> 进入 switch 分支语句后，检查变量当前值，并进入相应的 case 块中，执行相应的代码。若无匹配 case 块，则进入 default 块。

值得一提的是： switch 也是可以使用 beak 跳出的。但由于 switch 只会进入对应的 case 或 defaul t块中，所以你也只能将 break 写进某个 case 或 default 块中。

所以，有了 switch 语句，将所有功能整合进一个计算器程序也不是问题了。为了篇幅简洁，仅放上代码片段供读者参考。

```
switch opt
    case "+"
        ...
    end
    case "-"
        ...
    end

    ...

    default
        out.println("please enter correct key.")
    end
```

我们发现，switch 还能帮助代码节省一个“开关量”，也就是之前的 boolean 型变量 flag。这在编码上是比之前的 if 方案优美简洁的。 聊到编码的简洁，也许即使是初学者都可能听过一个说法：“代码复用”。通俗地讲就是“一次编码，处处使用”。其实函数（function）就具备这样的功能。

函数的定义：

```
function 函数名(参数1, 参数2, 参数3, ...)
    语句块
    return 返回值
end
```

函数的调用（使用）：

```
函数名(参数1, 参数2, 参数3, ...)
```

> 函数需要先定义，再使用。 **定义：** 需指明**函数名**，函数名后需写明参数列表。整个参数列表使用括号()扩起，其中的参数使用逗号,隔开，表示调用该函数时必须传入的参数。函数的定义中可以有返回值，其表示向外部返回一个值，返回语句用 return 值 来表示。
> **调用：** 函数的使用被称之为**调用**，调用时需指明**函数名称**与填写**参数列表**。若函数有返回值，则该函数执行完后，向调用处返回一个值。
> **执行：** 在触发函数调用处，CovScript 会根据函数名称与参数来识别你所调用的函数，然后将参数传入函数，进入函数体，执行语句块，并在执行到 return 语句时跳出函数，将 return 的返回值返回给调用处。

也许有读者第一次接触函数，认为有所不适，需要大量练习才能熟练掌握函数的用途，但其实大可不必。函数最早出现于十七世纪，用来表示数学上量之间变化的关系，而通常我们最早接触的函数是初中的数学课，诸如 $y = x$ 类的正比函数也许是我们所知最简单的函数之一。而在高中的数学课，你会知道形如 $f(x) = y$ 是函数的一种更普遍的表示方法，也会知道这里自变量与应变量用何种字母代替并无差别。

计算机学科与自然科学的差异在于它是人造学科，目的是为自然科学与实践服务，对数学也不例外。所以从符号语言的形式上讲，编程语言中的函数在很多场合与数学中的表现形式也大相径庭。比方要在 CovScript 中实现一个一次函数（$f(x) = 2x + 1$）,我们可以这样定义并使用：

```
using system
function f(x)
    return 2*x + 1
end

var y = f(3)
out.println(y)
```

你可以动手试一试以上代码，它和你所想的结果应该一致，会输出结果 7。 而没有返回值的函数，我们称之为"无返回函数"。在 Pascal 中，这种无返回值的函数被称之为"过程"；而在 Java 中，无返回值函数被定义为 void 类型。光从别的语言对无返回值函数定义，我们不难体会到无返回值函数被设计的目的：完成一组操作。下面是无返回值函数的演示实例。

```
using system
function my_fun()
    out.print("this is my function")
end
my_fun()
```

当然，这里的`my_fun()`之所以没有参数，是因为函数里的所有操作都不需要参数。既然不需要，何必写出来呢？因此，**有无返回值，有无参数，编写人员要依据代码的需求而决定。**

我们之前使用过很多次的`in.input()`，`out.print()`，`out.println()`实际上都是函数。若作者不将这些基础功能写成一个个可调用的函数，那么遭罪的可是语言的使用者。假如一个功能完整的`in.input()`内容有一百行，如果没有函数，我们读入三个数据可就不是三个 input 可以搞定的了，那需要三百行！读者可以仔细品味。

有了函数，我们就可以将计算器的代码整合地更为简洁了,部分代码展示如下。

```
function add()
    var a = in.input(), b = in.input()
    out.println(a+b)
end

...

switch opt
    case "+"
        add()
    end

    ...

end
```

函数还有一种用途很广的用法，叫做**递归**，这里只花小篇幅作简单介绍。

> 递归（Recursion），又译为递回，在数学与计算机科学中，是指在函数的定义中使用函数自身的方法。
> 递归一词还较常用于描述以自相似方法重复事物的过程。——wikipedia

在算法中，深度优先搜索（dfs）是递归最广泛地应用。如果想要彻底理解并对递归运用自如，建议学会 dfs。由于 dfs 属于算法内容，本书不以予介绍。算法，被称为编程的灵魂绝不过分。没有算法的代码，充其量只能是玩具。　下面展示一种求斐波那契数列任意项的递归实现。**注意：斐波那契数列求法有很多种，该算法效率奇低，耗费极高，故经常被用作许多硬件与软件的性能测试。**

```
# 默认输入值x非负，定义首项为0，次项为1
function fibonacci(x)
    return x>1 ? f(x-1) + f(x-2) : x;
end
```

5·矩阵计算器——解放线性代数作业：数组(array)

本节仍处于讨论中

6·来数钱！超市收银台：综合运用

经过了一整章的学习，相信应该对于基本的变量和语法已经有所了解，那么本章最后一节，将带领读者完成一个简易的**超市收银台**应用，综合性理解，我们是如何用程序解决生活中的问题的。

你即将作为程序员，解决**用户**的需求。所有的程序，都是为了解决种种实际问题而设计的。

6·1 目标

对于输入的包括商品名、单价和数量的多个商品信息，经过计算，打印购物清单（类似收银条，样例如下），要求有完整的命令行式交互菜单界面。

> **对于输入的信息：** 苹果 3元/个 3个 香蕉 4元/个 10个 蛋糕 13元/个 2个

> **得到的清单如下:**

```
----------------------购物清单-----------------------
------------------------------------------------------------
商品：苹果      单价：3         数量：3         共计：9
商品：香蕉      单价：4         数量：10        共计：40
商品：蛋糕      单价：13        数量：2         共计：26
------------------------------------------------------------
总计：75元
------------------------------------------------------------
```

6.2 设计程序

首先，要考虑该如何记录我们输入的信息，可能输入的商品信息条目是不定的，我们可以通过构建一个由switch语句控制的**菜单**来帮我们控制程序是要“继续输入信息”，还是要“打印购物清单”了，相应的，我们应当创建一个变量来控制switch语句的条件表达式。 想要用户了解如何输入的信息，可以通过out.print输出相关情况以及需要的信息，让用户知晓当前程序的状态。也可以输出一些用于分隔，装饰，缩进等等的字符串。

有了这个思路，我们就可以构建一个菜单的框架了：

```
loop
    out.println("\n---欢迎使用超市收银台---")
    out.println("请选择功能：")
    out.println("1.添加商品和价格\n2.结账\n3.退出")    #状态的列表
    out.println("----------------------")
    out.print("请输入你的选择：")
    var choice =in.input()      #用choice来承载当前状态，接受用户输入
    switch choice
        case 1
            ################################
            添加商品价格相关的代码！！！
            ################################
        end
        case 2
            ################################
```

```
            添加结账并输出购物清单相关的代码！！！
            ##################################
            break
        end
        case 3
            out.println("感谢您的使用!")     # 退出程序
            break
        end
        default
            out.println("****输入非法数据！请重新输入****")
            continue
        end
    end
end
```

细心的同学可能发现了，在整段代码的最外层有一个`loop-end`语句，这是考虑到可能需要不断地**添加商品**，并且不知道可能执行多少次，那么就需要用`loop-end`语句配合循环控制关键字`break`,`contine`实现不断地进入该菜单，执行用户需要的语句，并在合适的时候跳出循环，结束程序。

上面的代码中，到第5行之前都是向用户介绍菜单的交互信息，输出一些装饰用的分割线，而第六行创建的`choice`变量则是用于控制`switch`语句的变量，需要读取用户的输入。

如果用户输入的是1，即`case 1`会执行与添加商品价格相关的代码。

而`case 2`则会执行输出购物清单相关的代码。注意！因为清单输出后已经结束程序的任务，所以在`case 2`的最后有一行`break`帮助我们跳出循环，结束程序。

`case 3`则代表直接退出程序，同样用了`break`。

`default`则代表用户输入了数字`1 2 3`以外的字符，则要求用户重新输入，这里通过`continue`就实现了从该处中断最外层的`loop-end`循环，然后重新进入该循环，重新进入菜单，重新读入用户输入的状态。

上方代码中，菜单部分的输出如下：

```
---欢迎使用超市收银台---
请选择功能:
1.添加商品和价格
2.结账
3.退出
------------------------
请输入你的选择:
```

6·3补充知识：数组（这块要是讨论好了，可以移到前面去）

在程序执行的过程中，为了保证最后清单中可以输出所有的商品信息，势必要将所有的商品名称、价格和数量都保存下来，对于这种批量保存**相同类型的数据**，又需要对其进行访问的情况，一般采用**数组**结构进行处理。

> **数组(Array)**，顾名思义，是很多“数”构成的**序列**，然而一般来说编程语言中的数组是广义的数组，即序列中的每一个元素可以是其他的的 ***变量类型***，比如可以将`Bob,Mary,Jack,Mike`这样一个四个字符串的序列，放入一个名为`name`的数组中，它就是一个由字符串组成的数组。 **Covscript中的数组**更加完备，

> 可以存放不同的数据类型，当然除非在你可以确保不混淆自己视听的特殊情况，不建议在同一个数组中混合存储不同类型的数据。

当你想要存放不定数量的多个数据时，很直观的就会想到要利用数组进行存储，声明并创建一个数组的方法是：

```
var myarray = new array
```

上面的代码，创建了一个名为myarray的数组变量，它是空的。

同时如果想创建一个“已知内容”的数组，可以通过利用**大括号**的方式来直接创建数组,以上文的name数组为例：

```
var name = {"Bob","Mary","Jack","Mike"}
```

想要直接访问该数组的内容，可以很简单的利用[ ]（一般称作下标运算符）进行访问，通过下面的例子可以很简单的理解该用法

```
out.println(name[0])
out.println(name[1])
out.println(name[2])
out.println(name[3])
```

以上代码的输出结果是，即name[0]完全等价于字符串"Bob"

```
Bob
Mary
Jack
Mike
```

可以看到，声明一个数组的方法，与声明一个变量非常相似，作为一种**数据结构**，数组具备**“增删查改”**的功能，即添加数据，删除数据，查找数据，修改数据四个功能（这也是各种数据结构都共有的功能），这里仅简单介绍一下我们这个程序需要的功能。

当你创建了空数组以后，第一步就是向其中添加数据，CovScript一般采用如下的方式向数组中添加数据，以myarray为例

```
# 下方的push_back()函数向数组的最后添加括号内的元素
myarray.push_back("Bob")
myarray.push_back("Jack")
myarray.push_back("Mike)
```

经过这样的处理，得到的数组等价于用如下方式创建的数组

```
var myarray = {"Bob","jack","Mike"}`
```

如果想要得到与name相同的数组，需要在Bob和jack中间插入一个Mary,可以通过insert()函数实现

```
myarray.insert(1,"Mary")
```

此处因为重载有问题未完成！

6·4完善程序

首先是我们存储信息需要的数据结构，对于每个商品，需要保存**商品名**、**商品单价**，**购买数量**三个变量，那么我们应用3个数组结构来分别存储这些信息，用以下的方式初始化

```
var name = new array     #存放商品名的数组
var money = new array    #存放单价的数组
var num = new array      #存放商品个数数组，不用numbers是因为会和系统内置的numbers关键字重复
```

这样，我们就拥有了可以保存数据的容器。所有输入的数据都可以存放在这里，那么接下来，我们就可以完成前面**菜单部分**中未完成的两段代码了。

为了让程序可以更加直观，提高可读性，让其他人在阅读本例的代码时，在只看到上文**菜单部分**的switch语句的情况下，能很直观的理解到程序各个部分的作用，于是我们考虑利用之前学到的**函数**来封装该部分的代码。

于是我们有两个函数等待完成，首先是负责读取**每一个**商品信息的输入部分，根据其功能我们将其命名为addProduct函数，考虑到该函数一定会向我们之前构建的三个数组存入数据，那么在该函数的参数列表中，我们要添加三个参数，分别向函数中输入我们之前的三个数组作为参数，这样，函数就取得了对于三个数组的控制能力，便于在之后的过程中，实现对数据的存储。

同理，对于switch的case 2情况，也要构建一个**输出函数**，我们可以将其命名为printSheet，它要从之前创建的三个数组中读取信息，所以同样的，它的参数列表也是由那三个数组构成。

此时，我们距离完成程序的步骤，只剩下实现刚才未定义的两个函数addproduct和printSheet了，总结现在已有的框架如下。

```
using system



var name = new array
var money = new array
```

```
var num = new array

#########################
要在这里完成下文中的两个函数!
#########################

loop
    out.println("\n---欢迎使用超市收银台---")
    out.println("请选择功能: ")
    out.println("1.添加商品和价格\n2.结账\n3.退出")
    out.println("----------------------")
    out.print("请输入你的选择: ")
    var choice =in.input()
    switch choice
        case 1
            #添加的函数! 有三个参数, 分别对应上方三个数组
            addProduct(name,money,num)
        end
        case 2
            #添加的函数! 有三个参数, 分别对应上方三个数组
            printSheet(name,money,num)
        end
        case 3
            out.println("感谢您的使用!")
            break
        end
        default
            out.println("****输入非法数据! 请重新输入****")
            continue
        end
    end
end
```

首先来完成输入函数`addProduct(name,money,num)`，对函数定义如下:

```
function addProduct(Name,Money,Num)
    #即将添加的内容
end
```

函数的形参并不要求和输入的参数同名。相反，不同名但是意思相近更有利于区分函数内的参数，与输入的变量。当然，在你能够明晰谁是谁的情况下，用同样的名字也是可以的。

该函数有3个任务:

- 告知用户该输入什么信息
- 读取用户输入
- 将用户输入的信息利用参数中的数组进行保存

分别用这三种方式实现:

- out.print()函数输出
- in.input()函数输入
- 数组的push_back()方法

具体代码如下：

```
function addProduct(Name,Money,Num)
    out.print("请输入商品名字：")
    Name.push_back(in.input())
    out.print("请输入商品价格：")
    Money.push_back(in.input() )
    out.print("请输入商品个数：")
    Num.push_back(in.input() )
end
```

这样一来，我们就实现了读取用户输入并存储的任务。

可以看到在3,5,7行，我们通过将数组的push_back()函数和in.input()函数组组合在一起的方式，在一行之内实现了输出，这样潜在的技巧在编写代码过程中是值得学习借鉴的。

接下来我们就要实现输出的函数，总体来说就是for循环语句、数组内容的访问与out.print()语句的结合，其中的sum_money是利用循环来将价格加和处理的，具体实现如下，请读者自行理解：

```
function printSheet(Name,Money,Num)
    var sum_money = 0
    out.println("\n-------------------------购物清单-------------------------")
    out.println("----------------------------------------------------------")
    for i = 0, i < Money.size, i++
        sum_money += Money[i] * Num[i];
        out.println("商品："+ Name[i] + " \t单价：" + Money[i] +" \t数量："+Num[i]
+" \t共计：" +Money[i] * Num[i])
    end
    out.println("----------------------------------------------------------")
    out.println("总计："+ sum_money + "元")
    out.println("----------------------------------------------------------")
end
```

由此，我们就实现了整个的超市收银台功能，全部的代码如下：

```
@charset:gbk
using system

function addProduct(Name,Money,Num)
    out.print("请输入商品名字：")
    Name.push_back(in.input())
    out.print("请输入商品价格：")
    Money.push_back(in.input() )
    out.print("请输入商品个数：")
```

```
    Num.push_back(in.input() )
end

function printSheet(Name,Money,Num)
    var sum_money = 0
    out.println("\n-------------------------购物清单-------------------------")
    out.println("----------------------------------------------------------")
    for i = 0, i < Money.size, i++
        sum_money += Money[i] * Num[i];
        out.println("商品: "+ Name[i] + " \t单价: " + Money[i] +" \t数量: "+Num[i]
+" \t共计: " +Money[i] * Num[i])
    end
    out.println("----------------------------------------------------------")
    out.println("总计: "+ sum_money + "元")
    out.println("----------------------------------------------------------")
end

var name = new array
var money = new array
var num = new array

loop
    out.println("\n---欢迎使用超市收银台---")
    out.println("请选择功能: ")
    out.println("1.添加商品和价格\n2.结账\n3.退出")
    out.println("----------------------")
    out.print("请输入你的选择: ")
    var choice =in.input()
    switch choice
        case 1
            addProduct(name,money,num)
        end
        case 2
            printSheet(name,money,num)
        end
        case 3
            out.println("感谢您的使用!")
            break
        end
        default
            out.println("****输入非法数据! 请重新输入****")
            continue
        end
    end
end
```

这里要强调一下第一行的`@charset:gbk`的功能，因为我们日常使用的汉字，在计算机中可能会以不同的编码形式存在，常见的中文编码有`GB2312`编码，`GBK`编码，`Unicode`也就是`UTF`编码等等，因为本程序输出了中文字符，所以要用这种方式事先告诉解释器：“我要用中文啦！”，才能保证我们的中文被正常输出。切记，所有输出中文的程序都要预先用`@charset:XXXXXX`声明为需要的编码类型。

这样当我们向程序输入信息时，就可以得到对应的价钱，以及总价了！

样例输入：

```
1
苹果
3
3
1
香蕉
4
10
1
蛋糕
2
13
2
```

样例输出：

```
-------------------------购物清单-------------------------
------------------------------------------------------------
商品：苹果      单价：3         数量：3         共计：9
商品：香蕉      单价：4         数量：10        共计：40
商品：蛋糕      单价：2         数量：13        共计：26
------------------------------------------------------------
总计：75元
------------------------------------------------------------
```

如果有针式打印机的话，就可以打印一张像模像样的购物小票了。

第一章完