七. 模块化程序设计

1. 临时作用域

有的时候我们不希望一个变量污染作用域，用完之后想删除它，或是原有作用域中变量已经太多，词穷了不知怎样起名，这时就可以使用临时作用域。

思考：使用现有知识，如何建立一个临时作用域？

我们知道 if、while等都拥有其独立作用域，而while等循环结构会执行代码多次，我们只希望执行一次，所以if是最佳选择：

```
if true
    语句块
end
```

将if的条件设为true将保证if中的内容一定执行，又因为if有独立作用域，所以其中的变量在执行完毕后会被销毁。

提示：当我们需要让一部分代码不再执行时，也可以利用if false将代码"短路"。

实际上 CovScript 直接提供了临时作用域语法：

block

    语句块

end.

这与 if true 实际上是等价的。

2. 名称空间

当我们想表示归属于一个种类的变量时，我们可能这样写：

var type1_a = ...

var type1_b = ...

var type2_a = ...

var type2_b = ...

CovScript 提供了名称空间将各个变量分开：

namespace 名称空间名

    语句块

end.

这里需要注意名称空间中只允许变量定义、函数定义、类型定义才是被容许的名称空间。

#例 7.2.1.

```
namespace type1
    var a=1
    var b=2.
end.
namespace type2.
    var a=3.
    var b=4.
end.
system.out.println(type1.a+type2.a+type1.b+type2.b).
```

我们会发现，a这个变量出现了两次，但由于作用域不同，两者并不冲突。这里需要注意访问名称空间中的变量要用到"."运算符。

名称空间与临时作用域不同，其中的变量直到名称空间销毁时才会销毁。

3.函数

函数我们在第四章已经介绍过了，但我们在这里要介绍的是使用函数搭建模块化的架构。

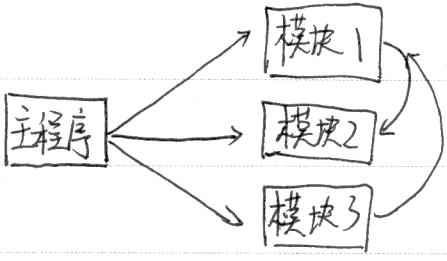
①为什么要模块化？

到目前为止我们写出的程序功能都十分简单，逻辑也相对比较简单，所以有时除非要用到递归，一般大家都不写函数，因为麻烦。

但对于稍大一点的程序，一旦出现问题将很难排查，因为逻辑都揉在一起，各个功能之间的耦合度非常高；而且稍大的程序都不可避免的重复使用一部分代码，我们往往会选择复制一粘贴。但我们有时会修改这部分代码，每次修改都要修改所有添入到的位置，非常麻烦。

更不要说上千行、上万行的大型程序，若不用模块化设计，那真能把人给逼疯了。比如 CorScript 解释器，其这样代码量大概是两万行左右，而且还是高度压缩的 ctt 代码。若 CorScript 自举，估计至少要多一倍。

而模块化能很好的解决问题。



模块之间可相互调用，每个模块可重复调用，调整模块不会影响主程序。

②如何进行模块化

函数是最小单位的模块。所以我们先从函数开始。

○ 将重复次数多的代码放进为函数，

○ 将过程相对独立的逻辑放进为函数

如，一个程序要完成“输入”→“数据处理”→“输出”三个过程，我们可以将数据处理部分放进为函数。

○ 将一个大的过程分成几个阶段放进为函数，

如，数据处理一般包括“资源初始化”→“预处理”→“格式化”→“解析”→“运算”五个阶段，具体来说，给出CorScript解释器的几大阶段：

I：初始化符号表、词法规则、语法规则等资源（init 函数）.

II：运行预处理器将输入按行分割并剔除注释（preprocess 函数）.

III：运行词法分析器将输入格式化为可识别的词元（lexer 函数）.

IV：运行语法分析器将词元的析为高级语法树（parser 函数）.

V：解释高级语法树（exec 函数）.

当然，这只是大体上的五个阶段，实际上CorScript解释器远没有这么简单，每一个阶段里面又分好多个小阶段。

使用函数包装之后，可以进一步用名称空间分类再次包装。

4. 包(Package).

到现在为止我们只是在单一文件中编程，模块化的另一个方式是多文件编程。比如 CorScript 的解释器，近两万行代码分了几十个文件。一个文件写太多代码看起来很费劲，所以合适情况下要多文件编程。

在 CorScript 中可用 import 语句引入其他文件：

```
import 包名
```

CorScript 要求引入的文件只能是：

① CorScript 扩展：使用 C++ 编写，我们暂时不介绍

② CorScript 包：使用 CorScript 编写，我们今天介绍如何写包

之前我们没强调 CorScript 源程序的文件后缀名，因为这个的确没什么影响。我们规定，CorScript 源代码的后缀名为 csc（CorScript Code），CorScript 包的后缀名为 csp（CorScript Package），CorScript 扩展的后缀名为 cse（CorScript Extension）。

为什么现在要强调后缀名？因为 CorScript 会根据后缀名查找包。比如：

```
import abc
```

CorScript 会在 Import 路径*中首先查找 abc.csp 文件，若未找到则查找 abc.cse 文件，若两者都不存在则报错。

*：Import 路径跟据你的环境不同而不同。对于Windows平台上的CorScript GUI，可以在"选项"中设置。对于cs解释器，无平台差异，一般是当前目录。具体请阅读各个软件的文档。

然而包并不只是换个文件名这么简单。包中要求用声明"我是一个包"：

package 包名.

原则上包名应与文件名相同。

包在声明后，在当前作用域中就可以使用包名.名称访问本包内的变量。

提示：为什么不能直接访问或用global？

CorScript有一个特性叫做"动态作用域"。也就是说，一个变量名字究竟是哪一个变量，与上下文环境有关。如：

```
function dynamic_scope().
    system.out.println(a).
end.
```

函数dynamic_scope会访问变量a，但很明显在这里a并未定义。若这时调用此函数，将提示变量a未定义。

但要是这样写：

var a=10.

dynamic_scope().

将正常输出"10"。因为上层作用域中存在变量a。

同理，在包中访问的变量，若引入此包的程序中，可能是另外一种变量。所以为避免不必要的错误，请尽量使用"包名.变量名"来访问包内变量。

#例 7.4.1.

```
package test.
function foo(a,b).
    system.out.println(a+b).
end.
```

这是一个包，我们应将其保存为"test.csp"文件

#例 7.4.2

```
import test.
test.foo(2,3).
```

此程序将输出"5"

作业：尝试多文件编写程序

八、for循环与错误处理

1. for循环

我们在编写程序时经常用到类似于这样的循环：

```
var i=1
while i<=n.
    #TO DO
    i=i+1
end.
```

CorScript为这种循环提供了简化版——for循环。

语法：

```
for 迭代器=初始值 to 迭代终点
    #TODO
end,
```

for循环一般是用于遍历一个区间，或说范围(range)。在这里，如要定义变量i从1为始遍历区间[b,e]，可以这样写：

```
for i=b to e
    #TO DO
end.
```

for循环默认步长为1，但也可以指定步长：

for 迭代器 = 初始值 to 迭代终点 step 步长

    #TO DO

end.

注意：① for循环仅支持数值类型的区间

② 步长不能为小于或等于零的数，因为CorScript的判定条件为迭代器小于等于迭代终点。

作业：① 使用for循环重写以前的作业。

② 尝试将for循环与while循环之间相互转换

2. 范围for循环

我们在第六章介绍了迭代器。迭代器在本质上也是提供了元素的一个范围，所以对于支持迭代器的容器，我们提供了遍历容器的更加简便的方法。

语法：

for 迭代器 iterate 容器

    #TO DO

end.

这个循环等价于：

```
var 迭代器 = 容器.begin().
while 迭代器 != 容器.term().
    #TO DO.
    迭代器.forward().
end.
```

但也不完全相同。为什么？因为在范围for中迭代器并不是真正的迭代器，是一种特殊的迭代器，不需要调用.data()其本身就代表着容器中的元素，不能够调用.forward()和.backward()前后移动，也不能用于修改容器。

也就是说范围for仅能用于遍历容器。

~~#8.2.1.~~ #例 8.2.1.

```
var cor = {2,4,6,8}
for it iterate cor
    system.out.println(it).
    it=it+1.
end.
```

3. 错误处理

程序总是不可避免的会出现错误。但有一些错误允许用户对其进行处理，这类可恢复的错误被称为异常(exception).

异常处理分三步：~~抛出(throw)~~

尝试(try) -> 抛出(throw) -> 抓取(catch).

我们首先假设一段代码会抛出异常，并尝试运行它。

一旦这段程序抛出异常，我们立刻抓取它，并进行处理。

注意，若不抓取异常，程序就会终止运行。

首先我们来看如何抛出异常。

throw 表达式

在这里表达式的值只能是 runtime.exception 函数的返回值：

runtime.exception(异常语句(字符串)).

此函数将建立一个异常，然后我们可以使用 throw 语句抛出它。

然后我们看如何抓取异常，这要用到 try-catch 结构：

try

{ 语句块.1

catch 异常.

语句块.2.

end.

try-catch结构会尝试运行语块1，若抛出异常则抓取并运行语块2。

样例8.3.1.

```
function test_exception().
    throw runtime.exception("Hello!").
end.
try
  test_exception().
catch e
  system.out.println(e.what()).
end.
```

注意：异常可调用.what()方法获取异常详情，其类型为字符串。

此程序会输出"Hello!"

作业：尝试捕获用户的非法输入，如超出范围