模块化、语境和包

In [ ]:

```
y = 3;
f[x_] := Module[{y}, y = x + 1; x/y];
f[10]
y
```

Module 的作用是定义一个封闭的环境，其中存在一些局部变量（由第一个参数（必须是一个表）的元素组成）。

在 Module 中，局部变量 y 被替换为带有内部编号的一种变量 y num，其中num是一个自然数，取自$Mathematica$内部的一个计数变量ModuleNumber，于是 y$ 后面的数字永远不相同，所以也就不会有重名问题。

In [5]:

```
print := Module[{y}, y]
print
```

Out[6]:

```
y$3017
```

Module的这种特性有的时候我们在其它程序中也能使用

In [7]:

```
y = Unique[x]
y = Unique[x]
y = Unique[x]
```

Out[7]:

```
x$3411
x$3414
x$3417
```

带 $ 和编号的变量具有 Temporary属性:

In [10]:

```
Module[{x}, Print[x, " has attributes ", Attributes[x]]]
```

```
x$3746 has attributes {Temporary}
```

带有临时属性的变量在生存期结束时就会被系统 Remove，除非它们在生存期内被显式地返回给外部的全局环境

In [14]:

```
(* With[{n}, Print[n]]:局部变量指定 {n}  *)
With[{n = 5}, Print[n]]
With[{n = a}, Print[n]]
```

```
5
a
```

In [17]:

```
With[{n = 5}, n = 4; Print[n]]
(n = 4; Print[n]) /. {n -> 5}
ReleaseHold[Hold[(n = 4; Print[n])] /. {n -> 5}]
```

```
Set::setraw: Cannot assign to raw object 5.
```

```
5
4
5
```

In [31]:

```
With[{n = 6}, Print[n]]
Print[n]
```

```
6
5
```

With 的作用是定义一个封闭的环境，其中存在一些局部常量。在 With 的第二个参数表达式中所有这些常量都会直接替换成第一个参数中赋值语句右边的值。在这个环境中，局部常量的值是不变的。

In [34]:

```
a = x;
With[{n = a}, n = b; Print[n]]
a
x
```

```
b
```

Out[36]:

```
b
b
```

注意，在这个例子中，a 是一个外部环境中的变量，Mathematica 的变量实际上都是指针，所以，在 With 内部，n 是与 a 相等的指针，当我们写 n=b 时，实际上是在写 a=b，它的作用是把 a 的值换成 b 的值，这个新的值可能还在 a 的值原来所在的地址，也可能换成了新地址，但是不管是哪种情况，n 和 a 始终是指向同一个地方的，所以 n 仍是"常量"。

Module 和 With 的内部环境叫做相应的局部变量或局部常数的作用域。

- 几种隐式出现的作用域。

In [ ]:

```
f = Function[x, Function[y, x + y]];
f[1]
```

f 的函数体部分因为某种外部原因（x->1）改变了，这时 Mathematica 就会把内层的变量 y 自动命名为 y$，这是一个临时变量，它会在函数定义结束后被销毁。

In [38]:

```
With[{w = x}, h[x_] :> w + x]
```

Out[38]:

```
h[x$_] :> b + x$
```

Function 定义的是纯函数，或者叫匿名函数、λ-表达式，它们的参数是所谓的哑变量
其原因是为了避免与可能存在的全局变量重名

除了Module、With、 Function 以外，各种赋值、规则中的模式名称（x_、x__、x:pattern）也都具有自动改名的功能，这些赋值、规则语句就成为这些改名后的临时变量的作用域。
但是要注意，常量是不能自动改名的。

In [39]:

```
x = 1;  y = a;  f[x_] := (x = 2);  g[y_] := (y = 2);
{f[x], g[y], x, y}
```

```
Set::setraw: Cannot assign to raw object 1.
```

```
Set::setraw: Cannot assign to raw object 1.
```

Out[40]:

```
{2, 2, 1, 1}
```

- 一些哑变量的例子

In [ ]:

```
i = 50; x = a;
(* 此例中 x 必须是变量，不能是常数，否则不能做积分变量。 *)
{Sum[i, {i, 1, 100}], Integrate[x, {x, 0, 1}]}
{i, x}
```

In [ ]:

```
p[n_] := Integrate[f[x] x^n, {x, 0, 1}]
p[x]
q[n_] := Module[{x}, Integrate[f[x] x^n, {x, 0, 1}]]
q[x] (* 此例中 Integrate 换成 Sum 后效果相同。 *)
```

Module 和 With 都是通过改名的办法将作用域内的变量变为与外部全局变量不同的局部变量。

有的时候我们需要另一种隔离手段，我们想要构造一个作用域，其中的局部变量与全局变量具有相同的"名字"和不同的"值"，这时候就要用 Block 来做隔离。

In [45]:

```
Module[{x = a + 1}, x^2 + 3]
Block[{x = a + 1}, x^2 + 3]
```

Out[45]:

7

7

In [ ]:

```
y = x^2 + 3;
Module[{x = a + 1}, y]
Block[{x = a + 1}, y]
```

在上面的例子中，Module 里的局部变量只生存在 Module 内，它不会跟踪 y 的定义跑到 Module 外面去；另一方面，Block 里的局部变量 x 会跟踪出去，找出 y 中出现的所有 x，然后替换，所以它的作用域可以从 Block 延伸出去，侵入到 y 的作用域中。

Module 和 Block 的不同的作用域规则来源于人们对"函数的内部"这一概念的两种截然不同的理解：

- 第一种看法认为，一个函数的内部是源程序中这个函数从开始定义到定义结束的文本段落。这是从词法分析角度出发的定义，所以叫词法作用域（lexical scope）；

- 第二种看法认为，一个函数的内部是这个函数从开始执行到执行结束的这个时间段。这是从时间角度出发的定义，所以叫动态作用域（dynamic scope）。

一般认为词法作用域适于做静态分析，所以各种编译型语言都遵循词法作用域规则。而动态作用域则比较复杂，无法编译成静态代码，只能动态地解释。另外，动态作用域规则下很容易出现难于发现的问题，不易调试。

所以大部分现代语言都使用词法作用域，例如 C 家族、Pascal、Scheme、ML、Haskell 等；一些比较古老的语言，如 APL、TeX、早期 Lisp 则使用动态作用域规则。有的语言允许用户自己指定作用域规则，如 Common Lisp、Perl，以及 Mathematica。

因为词法作用域比动态作用域要安全，所以如无特别地必要，尽量使用 Module，避免 Block。
但是在Mathematica中，Block要比Module快一点，所以对性能有比较高的要求时可以适量地使用Block。

In [51]:

```
(* 闭包的例子：计数器产生器 *)
CounterCreator[first_ : 1, delta_ : 1] :=
 Module[{i = first - delta}, (i += delta) &]

counter1 = CounterCreator[];
counter2 = CounterCreator[];
counter3 = CounterCreator[19, 2];
counter1[]
counter2[]
counter3[]
```

Out[56]:

```
1
1
19
```

In [59]:

```
(* 另一个闭包的例子：状态机 *)
StateMachine =
  Module[{i},
   Function[func, i = 0;
    Switch[func, 0, i &, 1, ++i &, 2, --i &, 3, (i = 0) &, _,
     Print["Illegal function!"]]]];

read = StateMachine[0];
up = StateMachine[1];
down = StateMachine[2];
reset = StateMachine[3];
read[]
up[]
down[]
reset[]
```

Out[65]:

```
0
1
0
0
```

在各种程序语言中，还有一种显式指定作用域的方法。典型的例子就是 C++ 语言中的命名空间（namespace）。
在Mathematica中也有一种类似的机制，叫做语境（Context）。

In [ ]:

```
x = 1;
?x
```

这里的"Global`"是 Mathematica 的默认语境，所有用户自己定义的符号、函数都存在于这个语境中。 用户也可以自己定义语境：

In [ ]:

```
foo`x = 2;
?x
?foo`x
```

In [69]:

```
(* 语境还可以嵌套： *)
foo`bar`x = 2;
?foo`bar`x
```

无论何时，我们在 Mathematica 中总存在于一个语境中，称为当前语境，它可以通过变量 $Context 来查询。

当 Mathematica 遇到一个符号，它首先会在当前语境中查找这个符号的意义。
如果找不到，则会在 $ContextPath 所指定的语境路径中按顺序查找这个符号。

我们可以通过赋值改变当前语境，也可以向语境路径中添加我们的语境：

In [ ]:

```
$Context = "foo`"
$ContextPath = PrependTo[$ContextPath, "bar`"]
```

函数 Context[] 可以查找当前语境或者某个符号所属的语境：

In [ ]:

```
Context[]
Context[x]
```

函数 Contexts[] 可以列出所有语境，或者按通配符列出相应的语境：

In [ ]:

```
Contexts["System`*"]
```

语境可以通过 Begin["Content`"] 和 End[ ] 进出：

In [ ]:

```
Begin["新语境`"]
Print[x = 1];
End[]
```

包（package）都自带一个语境。如果导入了某个程序包，那么它的语境会自动添加为语境路径中的第一个语境。

In [ ]:

```
Needs["Quaternions`"]
$ContextPath
```