第5章 模块编程技术

Wolfram 语言是函数式的语言，模块编程是自定义函数的主要方法。模块编程通过内置函数 Module、Block、With 或 Compile 实现，这些内置函数起到“容器”的作用，将其中的语句或语句组组织在一起，实现特定的自定义功能。除此之外，Module 函数可定义局部变量，Module 函数是定义自定义函数的常用“容器”，每次调用 Module 函数都将生成不同的局部变量；Block 函数可以使全局变量的值局部化，即在 Block 函数中出现的全局变量的赋值操作不影响全局变量的值，例如，在笔记本中定义“x＝5”，然后，执行 Block 函数“Block［{x，y}，y＝x；x＝10；{y，x}］”将得到“{10，10}”，而在 Block 函数外部访问 x 的值，仍然为 5。With 函数在执行前用常量替换掉函数中相应的符号，例如，“With［{x＝3}，x＋1］”将返回 4。一般地，由于 With 函数单纯地执行常量替换符号操作，所以 With 函数的执行速度最快；Block 函数直接使用全局变量（或局部定义的变量）的名称，执行速度比 Module 函数快；Module 函数每次执行都将创建新的局部变量，其执行速度最慢。但是，由于 Module 模块具有将变量局部化的特征，其应用最广泛。Compile 函数将其中的语句组编译为可执行的机器代码，以提高函数的运行效率。

5.1 Module 模块

Module 模块由 Module 函数实现。在 Wolfram 语言中，任一笔记本中定义的变量均为全局变量，在所有其他的笔记本中均可访问。为了使变量的作用域局部化，最常用的方法是借助于 Module 函数。在 Module 函数中，定义的变量均为局部变量，其作用域为整个 Module 函数。

5.1.1 Module 函数

Module 函数的语法如下：

（1）Module［{x，y，z，…}，语句组］，这里的“x，y，z，…”为定义的局部变量名，“语句组”为由分号“；”分隔的任意数量的语句，“语句组”的最后一条语句的执行结果为 Module 函数的返回值。

（2）Module［{x，y＝y0，z，…}，语句组］，该语法与上述语法相似，该语法说明自定义的局部变量可以在定义时赋初始值。

一般地，使用 Module 函数实现自定义函数，例如计算一个数的绝对值，如图 5-1 所示。

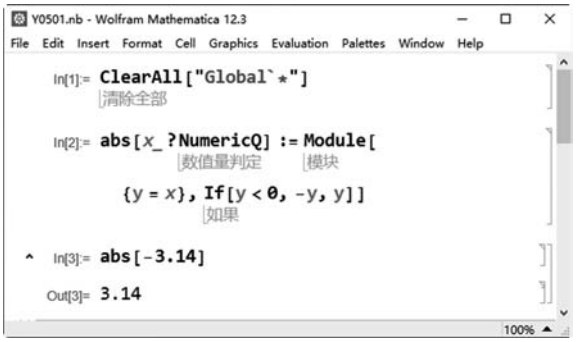

图 5-1　使用 Module 函数自定义函数

使用 Trace 函数可以跟踪自定义函数的全部执行过程，如图 5-2 所示。

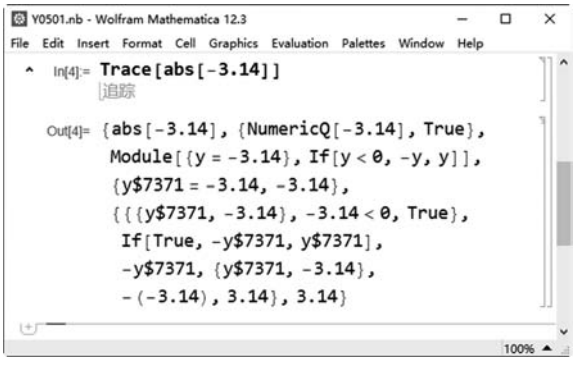

图 5-2　自定义函数 abs 的执行过程

在图 5-2 中，“In[4]”中的“Trace[abs[－3.14]]”将展示自定义函数的执行过程，如“Out[4]”所示。由“Out[4]”可知，“abs[－3.14]”的执行步骤如下：

第一步：“abs[－3.14]”，将函数调用发送到 Mathematica 内核。

第二步：“{NumericQ[－3.14],True}”，判断参数 x(这里为－3.14)是否为数值，返回“True”。

第三步：“Module[{y＝－3.14},If[y＜0,－y,y]]”，解析函数体。

第四步：“{y＄7371＝－3.14,－3.14}”，在 Module 函数中重命名局部变量 y 为“y＄7371”，这里的后缀“＄7371”称为模块编号，该编号保存在一个系统全局变量“＄ModuleNumber”中，随着模块的创建和调用其数值自动增加。Wolfram 语言使用这种方法为 Module 函数分配专用的临时局部变量。这一步执行了自定义变量并初始化的语句“{y＝x}”。

第五步：“{{{y＄7371,－3.14}，－3.14＜0，True}，If[True，－y＄7371，y＄7371]，－y＄7371，{y＄7371，－3.14}，－(－3.14)，3.14}”，这一步首先解析语句“If[y＜0,－y,y]”中的条件表达式，即将所有的局部变量和常数代入该条件表达式，判断结果为“True”；然后，解析“If”函数，得到“－y＄7371”；最后，解析“－y＄7371”得到结果“3.14”。

第六步：返回结果“3.14”。

通过在上述第五步的解析过程可知，Wolfram 语言使用回溯的方式，解析各个符号（变量），直到发现它们的数值。例如，解析“－y＄7371”时，将再次解析“y＄7371”。

Module 函数中可以包括内置函数和自定义函数，如图 5-3 所示。

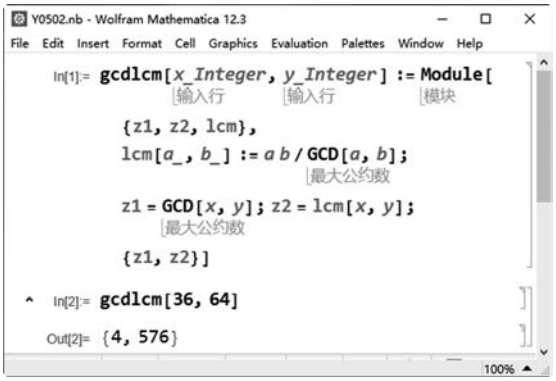

图 5-3　Module 函数中可内嵌自定义函数

在图 5-3 中，“In[1]”使用 Module 函数定义了函数 gcdlcm，该函数具有两个整型参数 x 和 y，返回这两个参数的最大公约数和最小公倍数。在 Module 函数中，定义了局部变量 z1、z2 和符号 lcm，其中，z1 用于保存最大公约数，z2 用于保存最小公倍数，符号 lcm 为自定义函数“lcm[a_,b_]:＝a b/GCD[a,b]”，用于计算两个整数的最小公倍数，这里的“GCD”为计算最大公约数的内置函数。然后，语句“z1＝GCD[x,y]”得到 x 和 y 的最大公约数；语句“z2＝lcm[x,y]”得到 x 和 y 的最小公倍数。最后，“{z1,z2}”作为 Module 函数的返回值，即为包含最大公约数 z1 和最小公倍数 z2 的列表。

在“In[2]”中调用“gcdlcm[36,64]”计算 36 和 64 的最大公约数和最小公倍数，其结果为“{4,576}”，如“Out[2]”所示。

Module 函数中的局部变量可以赋初始值，如图 5-4 所示。

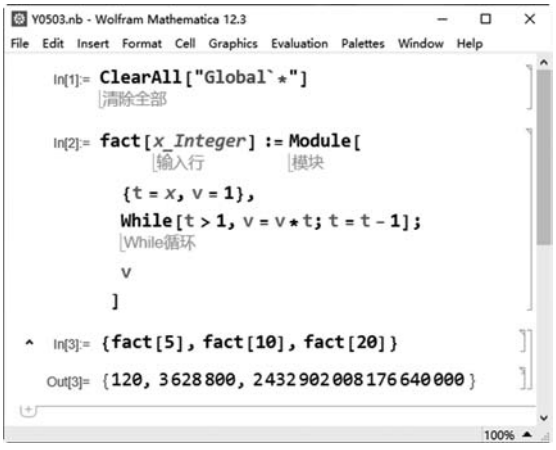

图 5-4　Module 函数局部变量初始化

在图 5-4 中，“In[2]”使用 Module 函数定义了函数 fact，具有一个整型参数 x，计算 x 的阶乘。在 Module 函数中，定义了局部变量 t 和 v，t 用 x 初始化，v 的初始值为 1。由于参数

x 在自定义函数 fact 调用时将被赋予数值，所以，x 在 Module 函数中以常数的形式存在，不能作为变量赋值，一般地，将参数 x 赋给一个局部变量，例如这里的“t”。在“While[t > 1, v=v * t;t=t-1]”循环体中，计算 t 的阶乘，结果保存在局部变量 v 中。最后，“v”作为 Module 函数的输出。

在“In[3]”中，“{fact[5], fact[10], fact[20]}”依次计算了 5、10 和 20 的阶乘，如“Out[3]”所示。

5.1.2 Module 模块实例

这里借助 Module 函数解决一个实际问题，即房贷计算问题。假设某人为买一套住房向银行贷款 100 万元，年利率为 4.5%，按月计算复利(月利率为 0.045/12)，计划 15 年还清全部贷款，且每月还款金额相同(按月等额还款方式)，编程计算每月应还款多少元。这里使用循环搜索解的方式(不套用公式计算)，开始时设置较大的步长，然后，设置小步长，精确度至少为 0.01 元。

这里设每月应还款 x 元，月利率为 rate，设从银行发放贷款的下一个月开始还款，设剩余还款金额保存在变量 pt 中，于是，第 0 个月 pt 初始化为总贷款额度；第 1 月后，pt = pt * (1+rate)-x；第二个月后，pt = pt * (1+rate)-x(此处 pt 为第一个月后的 pt)，以此类推，直到 pt 为 0，表示还款完成。按这种思路编写的程序如图 5-5 所示。

在图 5-5 中，定义了 house 函数，具有 money、years 和 rateofyear 三个参数，分别表示贷款总额、还款年限和年利率。在 Module 模块中，定义了局部变量“{pt, y=years, rate, x, n=108}”，这里 pt 表示剩余还款额度，y 初始化为还款年限数，rate 用于保存月利率，x 为每月还款额，n 设置为 10^8，用作 Do 循环的最大循环次数。然后，执行语句“rate=rateofyear/12; pt=money; x=0;”设置月利率 rate，设置剩余还款额 pt 为总贷款额，设置每月还款额 x 为 0 元。接着进入如下的 Do 循环：

```
"Do[Table[pt = pt * (1 + rate) - x,{i,1,y * 12}]; If[pt > 0,pt = money;x = x + 100,Break[]], {j,1,n}];"
```

在上述 Do 循环中，首先使用给定的月还款额 x，进行还款操作“Table[pt=pt * (1+rate)-x, {i,1,y * 12}]”，如果执行后，“pt > 0”，说明每月还款额 x 过少，于是“pt=money; x=x+100”，即令 pt 为总还款额，每月还款额 x 增加 100 元。在 Do 循环中重复这一过程，直到 pt 小于或等于 0，说明每月还款额 x 已大于需要的真实每月还款额。由于步长为 100 元，所以，真实的每月还款额在区间[x-100,x]内。

在下一个 Do 循环前，先执行“x=x-100; pt=money;”，即将每月还款额 x 设置为刚好不够还款的额度(在步长为 100 元的情况下)，并将待还款额 pt 设为总还款额 money，再执行下面的 Do 循环：

```
"Do[Table[pt = pt * (1 + rate) - x,{i,1,y * 12}]; If[pt > 0,pt = money;x = x + 10,Break[]],{j,1,n}];"
```

上述 Do 循环与前一个 Do 循环实现的功能类似，只是将每月还款额 x 的步长设为 10，经过上述 Do 循环，真实的每月还款额将落在区间[x-10,x]内。

回到图 5-5 中，上述 Do 循环后面的 Do 循环操作与此类似，每月还款额 x 的步长取为 1 元，如下所示：

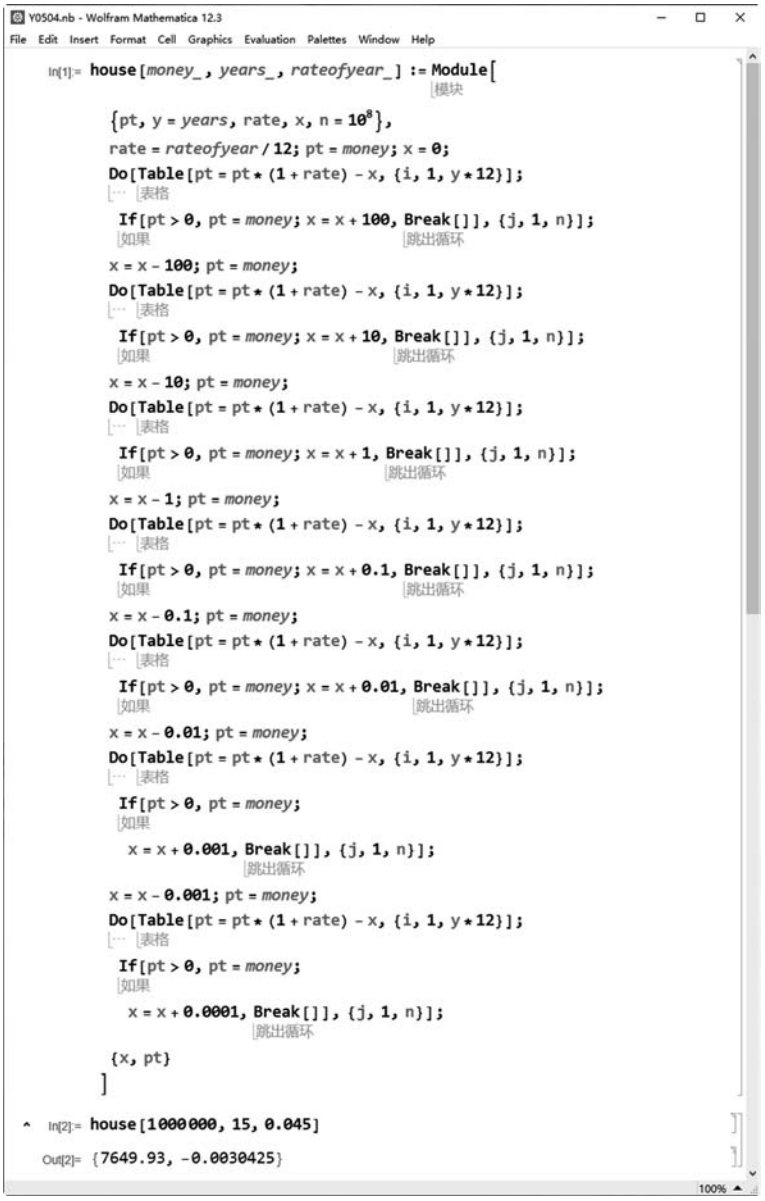

图 5-5　房贷计算问题

```
"x = x - 10;pt = money;
Do[Table[pt = pt * (1 + rate) - x,{i,1,y * 12}]; If[pt > 0,pt = money;x = x + 1,Break[]],{j,1,n}];"
```

这里，首先将每月还款额 x 设置为刚好不够还款的额度(在步长为 10 元的情况下)，并将待还款额 pt 设为总还款额 money，接着执行 Do 循环直到 pt 小于或等于 0，此时，真实的每月还款额将落在区间[x－1，x]内。

然后，进一步缩小步长至 0.1 元，执行如下代码：

```
"x = x - 1;pt = money;
Do[Table[pt = pt * (1 + rate) - x,{i,1,y * 12}]; If[pt > 0,pt = money;x = x + 0.1,Break[]],{j,1,n}];"
```

上述代码的分析与前述 Do 循环类似，执行完该 Do 循环后，真实的每月还款额将落在区间[x－0.1，x]内。

之后，进一步缩小步长至 0.01 元，执行如下代码：

```
"x = x - 0.1;pt = money;
Do[Table[pt = pt * (1 + rate) - x,{i,1,y * 12}]; If[pt > 0,pt = money;x = x + 0.01,Break[]],{j,1,n}];"
```

上述代码执行完成后，真实的每月还款额将落在区间[x－0.01，x]内。

接着，再次缩小步长至 0.001 元，执行如下代码：

```
"x = x - 0.01;pt = money;
Do[Table[pt = pt * (1 + rate) - x,{i,1,y * 12}]; If[pt > 0,pt = money;x = x + 0.001,Break[]],{j,
1,n}];"
```

上述代码执行完成后，真实的每月还款额将落在区间[x－0.001，x]内。

最后，将每月还款额的步长设为 0.0001 元，执行如下代码：

```
"x = x - 0.001;pt = money;
Do[Table[pt = pt * (1 + rate) - x,{i,1,y * 12}]; If[pt > 0,pt = money;x = x + 0.0001,Break[]],{j,
1,n}];"
```

执行完上述 Do 循环后，最后得到的每月真实还款额将位于区间[x－0.0001，x]内，而题目要求精确度至少为 0.01 元，因此，这时可取 x 为每月还款额。

在“In[2]”中调用“house[1000000,15,0.045]”，得到结果为“{7649.93，－0.0030425}”，即每月还款额应为 7649.93 元，如“Out[2]”所示。

为了方便介绍算法的实现过程，将图 5-5 中的自定义函数 house 代码写得比较冗长，图 5-6 中将 house 函数的代码作了简化整理。

对比图 5-5，在图 5-6 自定义函数 house 的 Module 函数中，多定义了一个局部变量 dx，用于保存每月还款额的步长，初始值为 100 元。在 Module 函数中，令“x＝money/(12y)”，设置 x 的初始值为零利率的每月还款额。然后，使用两重 Do 循环，内层 Do 循环与图 5-5 中的每个 Do 循环类似，如下：

```
"Do[pt = money; Table[pt = pt * (1 + rate) - x,{i,1,y * 12}]; If[pt > 0,x = x + dx, Break[]], {j,1,n}];"
```

在这个 Do 循环中，首先将剩余还款额 pt 赋为总还款额 money；然后，使用当前的每月还款额 x 进行还款；如果 pt 大于 0，说明每月还款额 x 小于实际的每月应还款额，将 x 增加到 x＋dx；循环执行这个操作，直到 pt 小于或等于 0，说明实际的每月应还款额在区间[x－dx，x]上。

在外层的 Do 循环中，将 x 设为 x－dx，即在步长为 dx 的情况下，刚好不够还款的每月还款额度；将 dx 缩小 10 倍，即“dx＝dx/10.0;”。然后，回到循环体的开头继续执行，直到剩余还款额 pt 的绝对值小于 0.01 元为止。最后，将 x＝x＋dx 作为实际的每月还款额。

在图 5-6 的“In[3]”中，调用“house[1000000,15,0.045]”，得到实际的每月还款额为 7649.93 元，如“Out[3]”所示。

图 5-5 和图 5-6 的程序具有通用性，输入任意的贷款额、贷款年限和年利率均可以计算出每月应还款额度。在图 5-6 的“In[4]”中，调用“house[2000000,20,0.050]”计算了贷款额 200

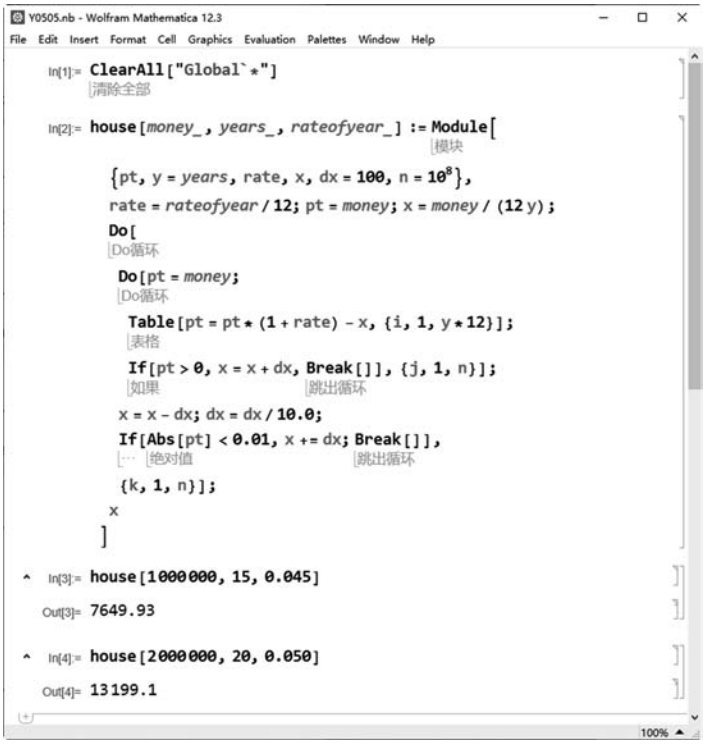

图 5-6 简化后的 house 函数

万元、贷款 20 年和年利率为 5%时每月还款额，结果为 13199.1 元，如“Out[4]”所示。

图 5-6 的程序可以使用 While 循环实现，代码更加简洁，如图 5-7 所示。

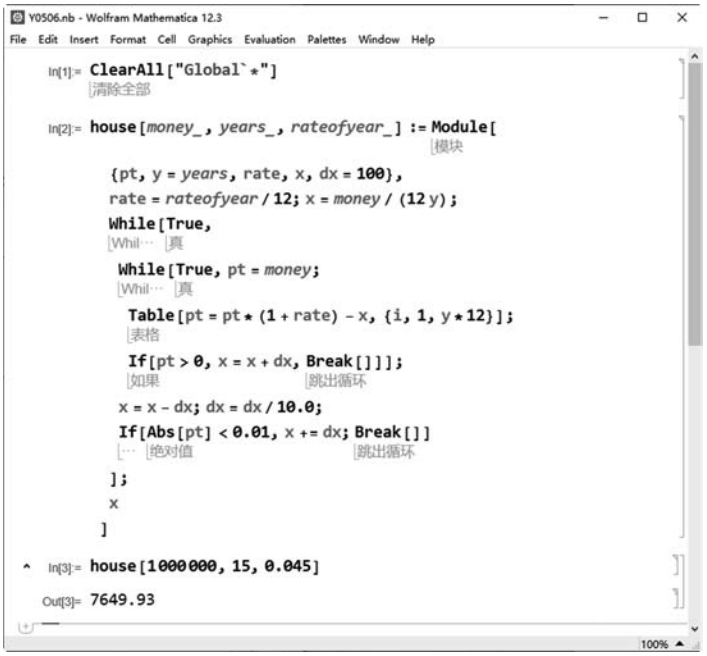

图 5-7 使用 While 实现的 house 函数

对比图 5-6 可知，在图 5-7 中，使用 While 循环替代了 Do 循环，改进后无须关心 Do 循环的执行次数。

5.2 Block 模块

Block 模块由 Block 函数实现。与 Module 函数类似，Block 函数是变量局部化的重要方法。与 Module 函数不同之处在于，Block 函数为每个变量（或符号）在 Block 函数内部分配临时的值，这些临时的值不影响每个变量（或符号）在 Block 函数外的取值；而不是像 Module 函数那样创建新的局部变量。一定意义上，Block 函数可以完全替代 Module 函数。Block 函数的执行速度比 Module 函数更快，Block 函数可用于所有使用 Module 函数的情况下；此外，Block 函数可以临时地调整系统全局变量的值，以达到这些全局变量作用局部化的效果。

5.2.1 Block 函数

Block 函数的语法如下所示：

（1）Block[{x，y，z，…}，语句组]，其中，“{x，y，z，…}”为局部变量列表，这些局部变量可以与笔记本中的全局变量同名；“语句组”为由分号“；”分隔的任意多条语句。

（2）Block[{x，y＝y0，z，…}，语句组]，该语法说明局部变量列表中的各个局部变量可以赋初始值。

下面通过图 5-8 所示程序说明 Block 函数与 Module 函数的异同点。

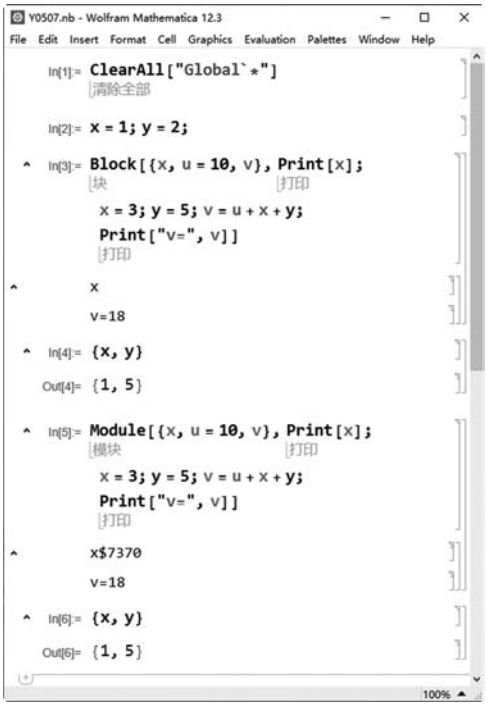

图 5-8　Block 函数和 Module 函数的异同点

图 5-8 中，在“In[2]”中定义了全局变量 x 和 y，分别赋值为 1 和 2。“In[3]”为 Block 函数，定义了局部变量 x、u 和 v，其中，u 初始化为 10；然后，调用“Print[x]”打印 x，将得到“x”；接着，对局部变量 x 赋值 3，这不影响全局变量 x；之后，对全局变量 y 赋值 5，令“v=u+x+y”，此时，v=10+3+5=18，最后“Print["v=",v]”打印 v 的值为“v=18”。在“In[4]”中显示“{x,y}”的值为“{1,5}”，即全局变量 x 不受同名的局部变量的影响，全局变量 y 在 Block 函数中重新赋值为 5。

在“In[5]”中使用 Module 函数实现相同的功能，这里“Print[x]”得到了“x$7370”，即一个新的局部变量，命名规则为“局部变量名$模块编号”。在“In[6]”中显示“{x,y}”的值为“{1,5}”，即全局变量 x 不受 Module 函数中“同名”局部变量的影响，而全局变量 y 在 Module 函数中被赋值，这与 Block 函数实现的功能相同。

图 5-9 中的程序只能使用 Block 函数。

图 5-9　Block 函数使系统变量作用域局部化

在图 5-9 中，“In[2]”使用 Block 函数计算精度为 32 的浮点数运算。这里只能使用 Block 函数，因为 Module 函数为每个局部变量创建新的变量形式。在 Block 函数中，设定了系统全局变量$MaxPrecision 和$MinPrecision 的新值，均为 32，表示使用精度为 32 的浮点数运算，这并不影响系统全局变量$MaxPrecision 和$MinPrecision 的全局设定值。然后，计算了“Exp[0.84'32]”，这里的“'32”表示浮点数的精度为 32，计算结果如“Out[2]”所示。在“In[3]”中函数“RealDigits”将“Out[2]”中的浮点数的各位提取出来，以列表的形式保存，如“Out[3]”所示，列表中的第 1 个元素“{2，3，1，6，3，6，6，9，7，6，7，8，1，0，9，1，7，3，3，5，0，0，2，4，7，1，9，2，8，6，5，5}”为“Out[2]”各位上的数字，第 2 个元素“1”表示小数点的位置在第 1 个元素之后。在“In[4]”中“Length[First[%]]”显示了“Out[2]”中数位的个数为 32，如“Out[4]”所示。

由图 5-8 和图 5-9 可知，Block 函数可以取代 Module 函数，实现符号（或变量）的作用域局部化，只是 Block 函数为符号分配新的存储空间（而不改变符号的名称），而 Module 函数

将为符号创建新的副本(用“符号名 $ 模块编号”命名该副本),由于每次运行 Module 函数,其“模块编号”均增加,故每次调用 Module 函数为局部符号(或变量)创建的副本都不相同。但是 Module 函数却不能完全替代 Block 模块,特别是在如图 5-9 所示的情况下,只能使用 Block 函数,而不能使用 Module 函数。由于 Block 函数的运算速度快于 Module 函数,所以,应尽可能使用 Block 函数进行模块化编程。

5.2.2 Block 模块实例

假设在图 5-10 所示的边长为 2 的正方形(正方形的中心位于坐标原点)中随机撒入细沙粒,若沙粒在正方形中服从均匀分布,则位于单位圆中的沙粒数与落入正方形中的总沙粒数之比应为单位圆的面积与正方形的面积之比。通过这种方式可以近似计算圆周率,如图 5-11 所示。

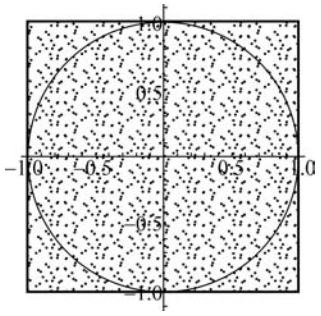

图 5-10 蒙特卡实验

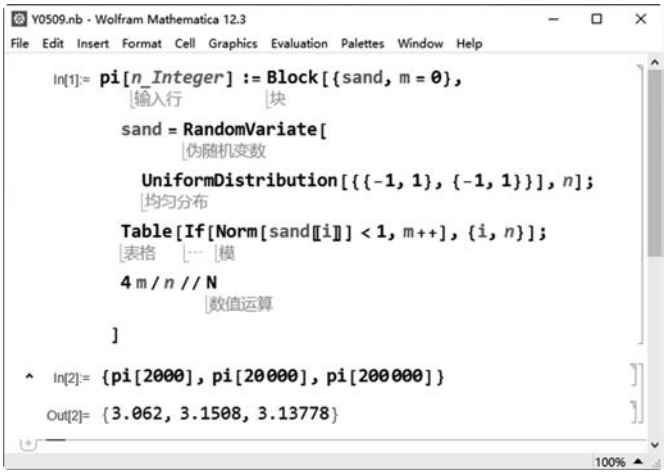

图 5-11 计算圆周率

在图 5-11 的“In[1]”中定义了函数 pi,具有一个整型参数 n,表示使用的沙粒的总数量。在 Block 函数中定义了两个局部变量 sand 和 m,m 初始化为 0,用于记录落在单位圆内的沙粒个数。然后,调用 RandomVariate 内置函数生成在图 5-10 所示正方形内均匀分布的 n 个随机变量,赋给 sand。接着,“Table[If[Norm[sand[[i]]]< 1,m++],{i,n}]”统计落入单位圆中的沙粒数,赋给变量 m。最后,计算“4m/n//N”得到圆周率的近似值。

在“In[2]”中,调用“{pi[2000], pi[20000], pi[200000]}”计算了当沙粒个数为 2000、20000 和 200000 时的圆周率的值,如“Out[2]”所示。可见,沙粒数越多,圆周率的计算结果越准确。

现在回到图 5-7 中,添加一条语句统计使用 Module 函数的 house 函数的执行时间,如图 5-12 所示。

然后,将图 5-7 中的“Module”改为“Block”,即使用 Block 函数实现 house 函数,其余内容保持不变,如图 5-13 所示。

在图 5-13 中,使用 Block 函数实现了自定义函数 house,house 函数的其余内容与图 5-7 中的内容相同。对于图 5-12 中“In[4]”和图 5-13 中“In[3]”的执行结果,可知,Block 函数实

图 5-12 Module 函数实现的 house 函数执行时间

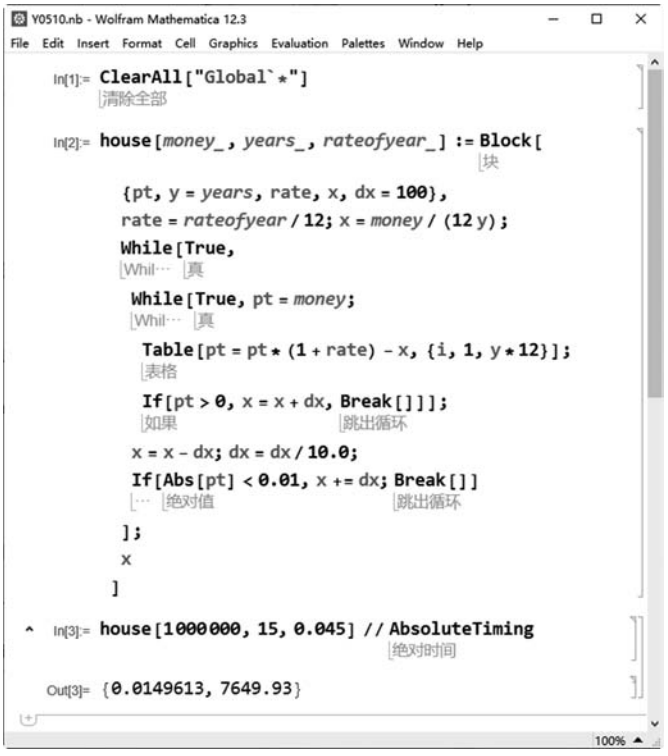

图 5-13 Block 函数的 house 函数执行时间

现的 house 函数的执行时间为 0.0149613 秒，比 Module 函数实现的 house 函数的执行时间(0.0153025 秒)短。当程序更加复杂时，Block 函数将明显地快于 Module 函数。对比图 5-7 和图 5-13 可知，任意的 Module 函数均可直接将“Module”改为“Block”函数，程序仍然工作正常。

5.3 With 模块

With 模块由 With 函数实现，With 函数通过给局部变量赋初始值的方式使局部变量成为 With 函数中的“局部常数”。With 函数体中首先执行这些局部变量的初始值替换，然后，对替换后的表达式进行计算，不需要创建新的局部变量存储空间，因此，With 函数的执行效

率比 Block 和 Module 函数更高。

5.3.1 With 函数

With 函数的语法如下：

```
With[{x = x0, y = y0, z = z0, …}, 语句组]
```

在 With 函数中，“{x＝x0，y＝y0，z＝z0，…}”声明局部“变量”的同时必须赋初始值，而且，这里的局部“变量”被其赋予的值替代，而不再是通常意义上的“变量”，即不能再对它进行赋值操作。

With 函数的基本用法如图 5-14 所示。

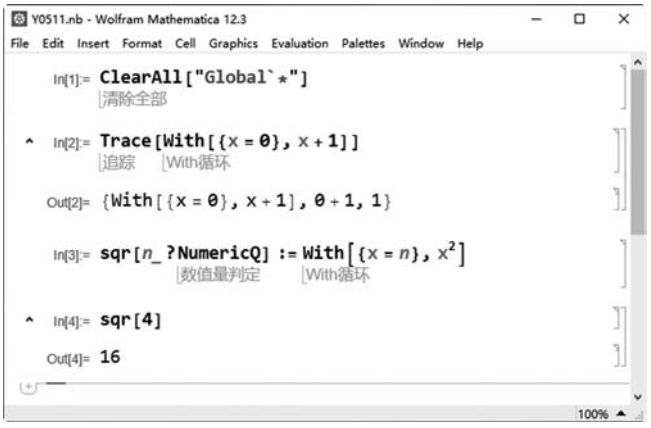

图 5-14　With 函数基本用法

在图 5-14 中，“In[2]”用 Trace 函数跟踪了 With 函数的执行。由“Out[2]”可知，在解析了“With[{x＝0},x＋1]”之后，直接将 x 替换为常数 0，最后返回 1。因此，无法在 With 函数中对 x 进行赋值操作，因为 With 函数体中没有 x 了。可以将 With 函数理解为将“常量”局部化。

在“In[3]”中，使用 With 函数定义了函数 sqr，这里的 With 函数直接将 x 替换为 n，返回 n2。在“In[4]”中调用“sqr[4]”，得到 16，如“Out[4]”所示。

5.3.2 With 模块实例

With 函数的语法为“With[{x＝x0，y＝y0，z＝z0，…}，语句组]”，其实现的功能为用局部“变量”列表中的初始值“常量”替换“语句组”中的局部“变量”，有些类似于“ReplaceAll”函数实现的功能。With 函数的这种替换有两个主要的应用场合，一是生成复杂的纯函数，二是函数的替换。

图 5-15 展示了 With 函数的两个典型应用实例。

在图 5-15 中，“In[2]”定义了函数 f，具有 a、b 和 c 三个参数，在 With 函数中，令“y＝a x^2＋b x＋c”，代换纯函数中的 y。在“In[3]”中，“{f[3,4,5]，f[3,4,5][1]，f[1,2,1][1]}”依次得到参数为 3、4 和 5 的纯函数“Function[x,5＋4 x＋3 x^2]”和当这个纯函数作用在 1 上

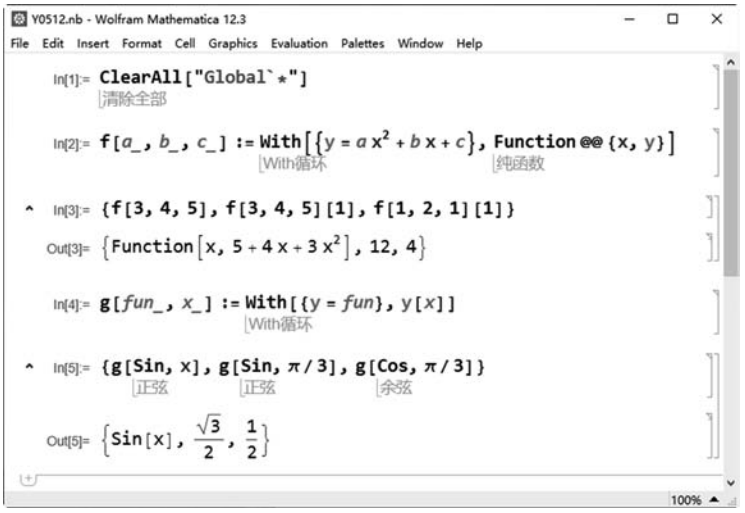

图 5-15　With 函数的典型应用实例

的函数值 12 以及参数为 1、2 和 1 的纯函数作用于 1 上的函数值 4，如“Out[3]”所示。

在“In[4]”中，用 With 函数定义了函数 g，具有两个参数 fun 和 x，其中 fun 表示函数名，x 表示“fun”的参数。在“In[5]”中调用了“{g[Sin,x], g[Sin,π/3], g[Cos,π/3]}”，结果如“Out[5]”所示，为“$\left\{\text{Sin}[x],\frac{\sqrt{3}}{2},\frac{1}{2}\right\}$”。

5.4　Compile 模块

Wolfram 语言的内置函数均作了优化处理，具有与机器代码相当的执行效率。对于自定义函数，应尽可能调用内置函数实现所定义的功能，这样保证自定义函数具有较高的执行效率。对于执行数值计算的自定义函数，可以借助于 Compile 模块将其编译为机器代码执行。Wolfram 语言提供了两种编译目标：①编译为运行于 Wolfram 虚拟机上的机器代码；②编译为 C 语言机器代码。Compile 模块对自定义函数执行了编译处理，这种编译仅对整型、浮点型、复数类型和逻辑型(True 和 False)的数据有效，因此，仅能对一小部分内置函数进行编译，也就是说，只有使用整型、浮点型、复数类型和逻辑型的数据且包含了可以被编译的内置函数的自定义函数，才能被 Compile 模块编译。可以通过指令“Compile`CompilerFunctions[]”查看可以被编译的部分内置函数列表(一些内置函数可以被编译但是没有列于其中)。

5.4.1　配置 MinGW64 编译器

这里使用 MinGW64 编译器实现自定义函数的代码编译。

登录网址 http://mingw-w64.org/doku.php/download，如图 5-16 所示。

在图 5-16 中，单击 MingW-W64-builds，进入图 5-17 所示界面。

在图 5-17 中单击“Sourceforge”，下载 MinGW-W64 安装文件。下载后的文件名为

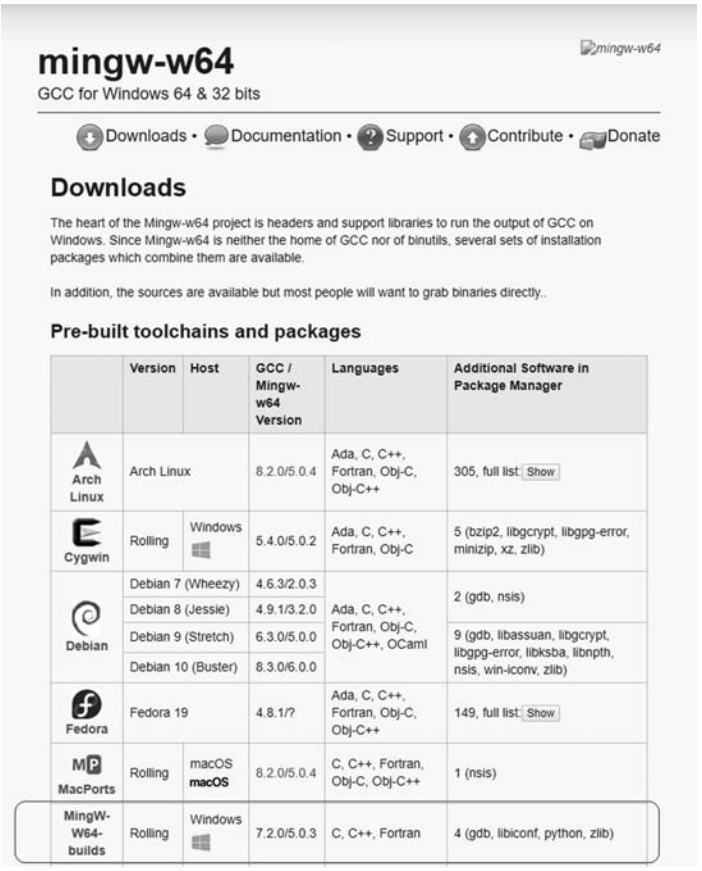

图 5-16　下载 MinGW-W64 安装程序

图 5-17　MinGW-W64 下载链接

mingw-w64-install.exe，文件大小约为 938KB。在 Windows 10（64 位）环境下运行文件 mingw-w64-install.exe，进入图 5-18 所示安装界面，其中，架构“Architecture”中选择“x86_64”，版本号为“8.1.0”。

在图 5-18 中单击“Next”按钮，进入图 5-19 所示窗口。

图 5-19 中默认安装目录为 C:\Program Files\mingw-w64\x86_64-8.1.0-posix-seh-rt_v6-rev0。然后，单击“Next”进行安装（需联网）。安装完成后，MinGW-W64 所在的目录为“C:\Program Files\mingw64\mingw64”，目录和文件结构如图 5-20 所示。

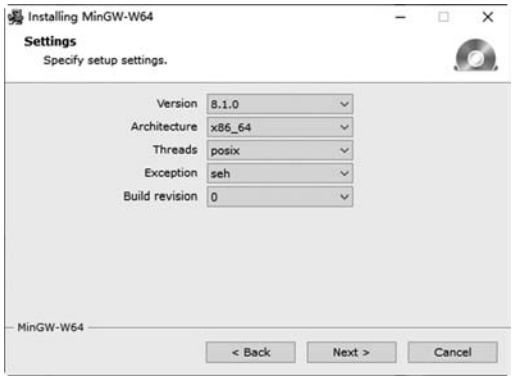

图 5-18　安装 MinGW-W64

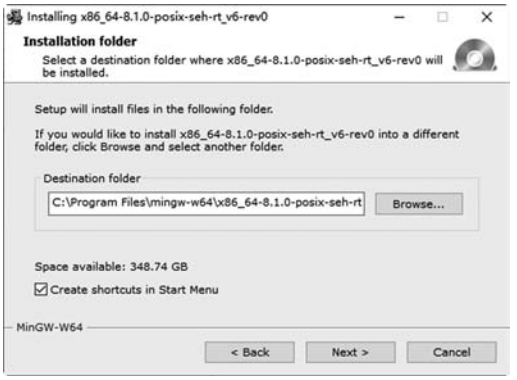

图 5-19　安装目录设置

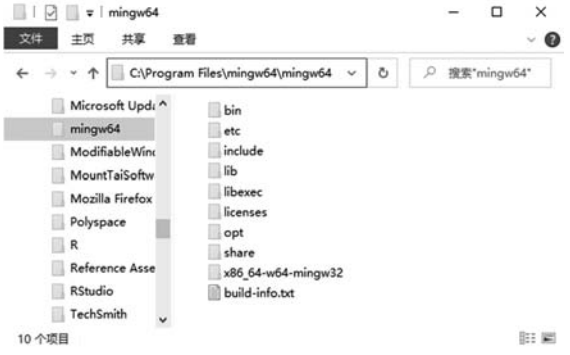

图 5-20　MinGW-W64 目录结构

右击“我的电脑”,在弹出菜单中选择“属性”；然后,进入“高级系统设置”；在“高级”选项卡中,单击“环境变量(N)…”,在弹出的“系统变量(S)”界面“编辑”路径“Path”,在其列表的最后一行添加路径“C:\Program Files\mingw64\mingw64\bin”。

现在,在目录 E:\ZYMaths\YongZhang\ZYCPrj 中编写 myhello.c 文件,如图 5-21 所示。

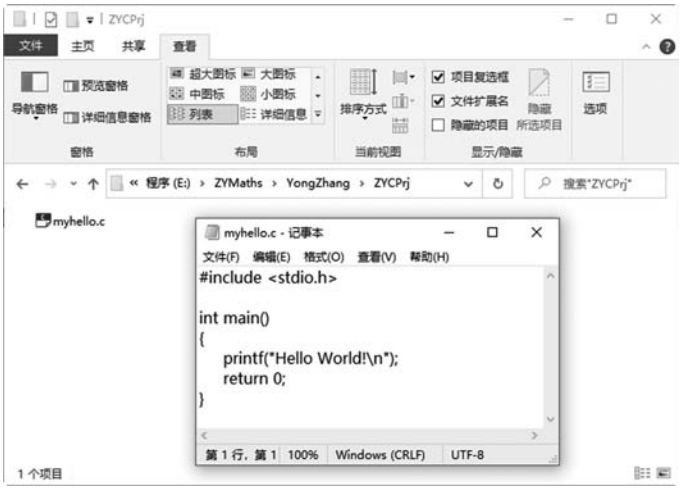

图 5-21　myhello.c 文件

打开“命令提示符”工作窗口，工作路径设为 E:\ZYMaths\YongZhang\ZYCPrj，如图 5-22 所示。

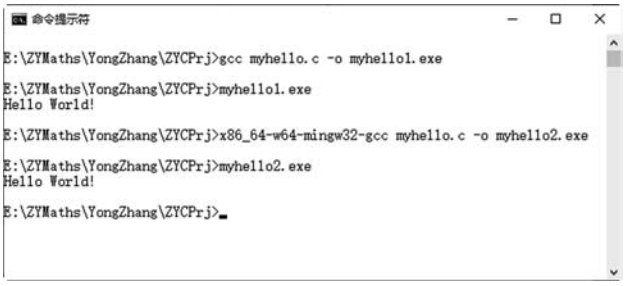

图 5-22　编译 myhello.c 并执行

在图 5-22 中，调用了 gcc 和 x86_64-w64-mingw32-gcc 编译了 myhello.c 文件，并分别生成了 myhello1.exe 和 myhello2.exe 文件。执行这两个可执行文件均显示“Hello World!”，说明 MinGW-W64 安装成功。

现在，在计算机 C 盘根目录下建立子目录 MinGW-w64，然后，将图 5-20 中所示内容复制到目录 C:\MinGW-w64 中。之后，编辑 C:\ProgramData\Mathematica\Kernel 目录下的 init.m 文件，设定其内容如下：

```
Needs["CCompilerDriver'GenericCCompiler'"];
$CCompiler = {"Compiler" -> GenericCCompiler, "CompilerInstallation" ->"C:/MinGW - w64", "
CompilerName" ->"x86_64 - w64 - mingw32 - gcc.exe"};
```

文件 init.m 在 Mathematica 启动时自动被调用，将编译器配置为 MinGW-w64。

在 Mathematica 软件中，新建笔记本 Y0513.nb，输入如图 5-23 所示的代码，在 Block 函数中指定编译目标为 C 语言可执行代码，即“$CompilationTarget＝"C"”，可实现对 Compile 函数代码的编译。

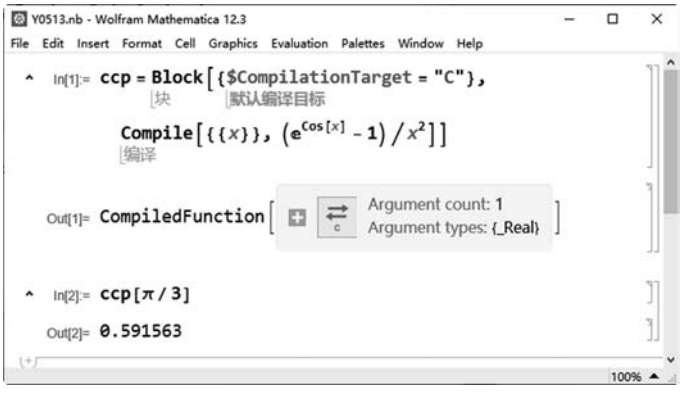

图 5-23　使用 MinGW-W64 编译 Block 模块

在图 5-23 中，Compile 函数有一个参数 x，该参数将用作自定义函数 ccp 的参数，如“In[2]”所示，执行结果如“Out[2]”所示。此外，在 Compile 模块中使用选项“CompilationTarget －> "C"”，可将 Wolfram 代码编译为 C 语言可执行代码，将在下一节中详细介绍 Compile 函数。

当编译包含了已编译模块的代码时，需要配置编译选项为“CompilationOptions －> {"InlineExternalDefinitions" －> True}”，如图 5-24 所示。

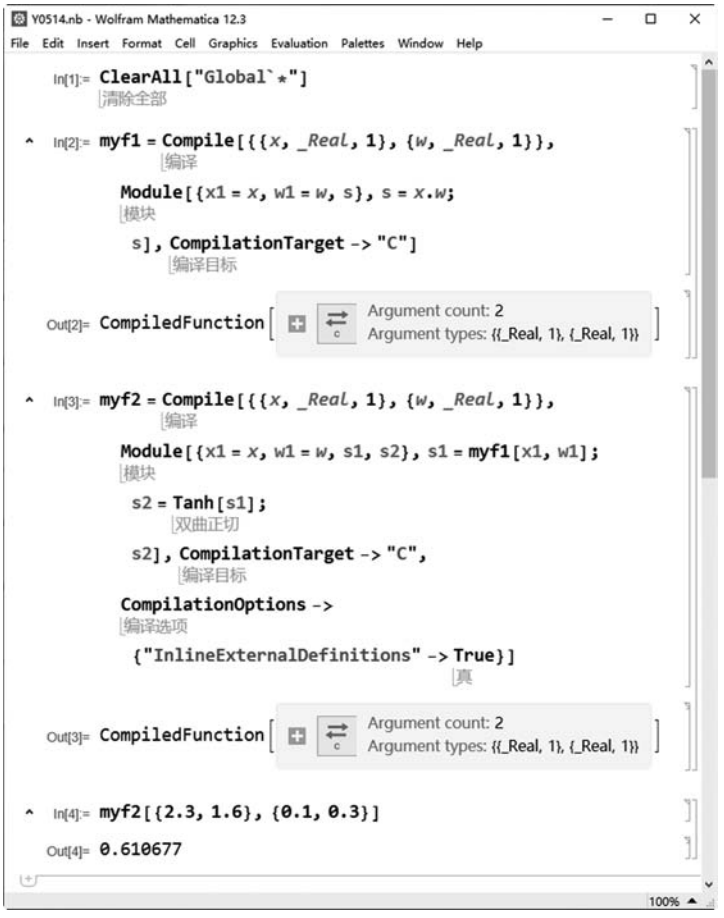

图 5-24 Compile 编译配置

在图 5-24 中“In[2]”所示的 myf1 函数中，使用选项“CompilationTarget－>"C"”将其编译为 C 语言可执行代码；在“In[3]”所示的 myf2 函数中，调用了 myf1 函数，故使用了选项“CompilationOptions －> {"InlineExternalDefinitions" －> True}”。在“In[4]”中调用“myf2[{2.3,1.6},{0.1,0.3}]”得到执行结果 0.610677，如“Out[4]”所示。下一节将深入介绍 Compile 函数。

5.4.2 Compile 函数

Compile 函数用于生成编译执行的代码模块，以提高代码的执行效率。目前 Wolfram 语言仅支持整型、实型、复数类型和逻辑型的数据处理方式的代码的编译。由于 Wolfram 语言是函数式的语言，而自定义函数必将调用内置函数和其他的自定义函数，所以，自定义函数的代码编译实质上是将其调用的内置函数和其他的自定义函数在可执行代码的级别上建立了接口。通过这种方式提高编译后的自定义函数(即 Compile 函数)的代码执行效率。

Compile 函数的调用语法如下：

(1) Compile[{x1,x2,…}, 语句组]

这里的“{x1,x2,…}”为 Compile 函数定义的函数的参数，默认为实数类型，“语句组”可以为 Module 函数或 Block 函数，也可以为由分号“;”分隔的大量语句组成，如图 5-25 所示。

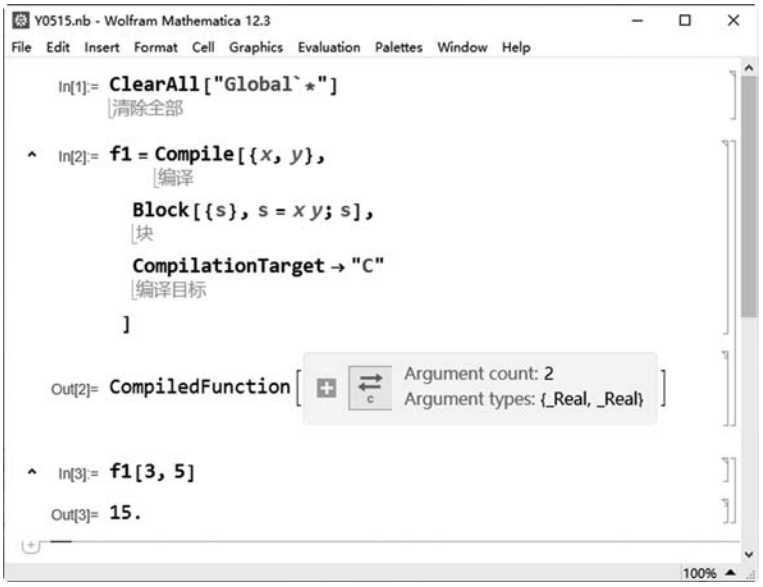

图 5-25　Compile 函数第一种语法实例

在图 5-25 的“In[2]”中，使用 Compile 函数定义了函数 f1，具有 x 和 y 两个参数，默认为实型参数。Compile 函数体由 Block 函数组成，这里 Block 函数实现了参数 x 与 y 相乘，积 s 作为返回值，也是 Compile 函数的返回值。而“CompilationTarget->"C""为编译选项，如果省略该选项，相当于“CompilationTarget->"WVM""，即编译为运行于 Wolfram 虚拟机上的机器代码。编译成功的函数如“Out[2]”所示，为“CompiledFunction”函数，显示的“Argument count: 2”表示具有 2 个参数，“Argument types: {_Real, _Real}”表示 2 个参数均为实数类型，单击“ ”将显示更多的编译后的函数信息。

(2) Compile[{{x1, 类型}, {x2, 类型}, …}, 语句组]

这里的“{{x1, 类型}, {x2, 类型}, …}”为 Compile 函数定义的函数的参数及其类型声明，注意：这里的“类型”仅支持整型、实型、复数类型和逻辑型(True 或 False，逻辑型用“True|False”显式指定)。当这里的“类型”为整数、浮点数类型和复数类型时，分别用“_Integer”“_Real”和“_Complex”指定，表示为单个的数值参数。这里的“语句组”可以为 Module 函数或 Block 函数，也可以为由分号“;”分隔的任意多条语句。

在 Compile 函数中可以使用笔记本中定义的全局变量，但是一般情况下，不应在 Compile 函数中使用全局变量，而应全部使用局部变量，因此，Compile 函数体常由 Module 或 Block 函数组成。Compile 函数中也可以使用 With 函数，在使用 With 函数时需注意，With 函数并不能定义局部变量，只能其将定义的局部“变量”替换为“常量”或“函数”符号进行后续计算，With 函数用于 Compile 函数中，一般只用作函数替换使用。

Compile 函数的这种语法实例如图 5-26 所示。

图 5-26　Compile 函数的第二种语法实例

在图 5-26 中，“In[4]”使用 Compile 函数定义了一个函数 f2，具有 x、y 和 z 三个参数，依次为整型、实型和复数类型的参数。在 Module 函数内部，使用了自定义函数 f1，然后，添加了选项“CompilationOptions->{"InlineExternalDefinitions" -> True}”，表示本函数使用了编译了的自定义函数。自定义函数 f2 实现了 x * y+|z|的计算。在“In[5]”中调用了“f2[3,5,2+3I]”，得到结果“18.6056”，如“Out[5]”所示。

(3) Compile[{{x1，类型，维度}，{x2，类型，维度}，…}，语句组]

Compile 函数的这种语法比第(2)种语法多了一个参数的“维度”，如图 5-27 所示。

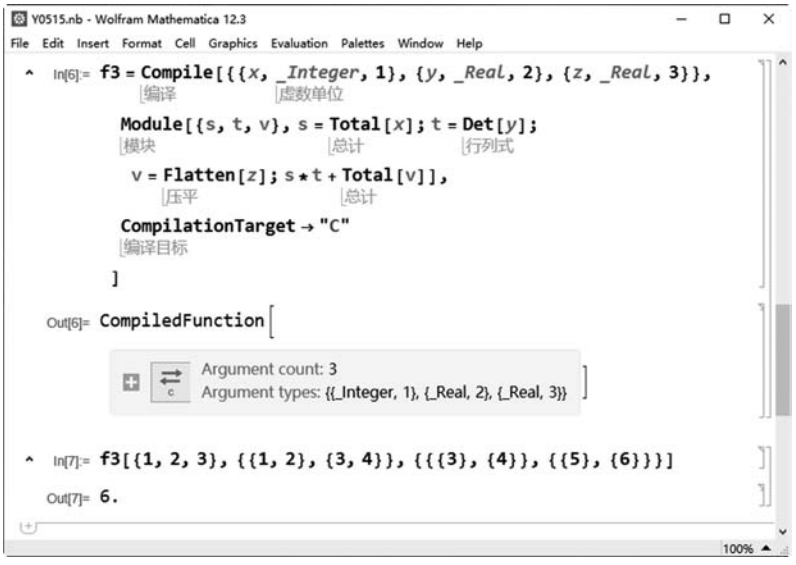

图 5-27　Compile 函数的第三种用法

在图 5-27 中，“In[6]”使用 Compile 函数的第(3)种语法定义了函数 f3，具有 x、y 和 z 三个参数，这里的参数 x 为一维列表，参数 y 为二维列表，参数 z 为三维列表。在 Module 函数内，“s＝Total[x]”将列表 x 的元素之和赋给 s，“t＝Det[y]”将二维列表 y(必须输入方阵)的行列式赋给 t，“v＝Flatten[z]”将三维列表 z 压平为一维列表，赋给 v。“s＊t＋Total[v]”作为 Module 函数的返回结果，也是 Compile 函数的返回结果。在“In[7]”中，执行调用“f3[{1,2,3}, {{1,2},{3,4}}, {{{3},{4}},{{5},{6}}}]”，这里，第一个参数“{1,2,3}”为一维列表，第二个参数“{{1,2}, {3,4}}”为一个二维列表，第三个参数“{{{3},{4}},{{5},{6}}}”为一个三维列表，计算结果为“6.”，如“Out[7]”所示。

5.4.3 Compile 模块实例

本节将介绍两个使用 Compile 函数实现的实例，其一为生成 Logistic 混沌序列，其二为使用 RC4 进行数据流加密与解密。

实例一　Logistic 混沌序列发生器

这里，使用的离散 Logistic 映射的形式为 xn＋1＝1－2，状态的取值范围为区间(－1,1)，将状态序列转化为 0～255 的整数序列，程序如图 5-28 所示。

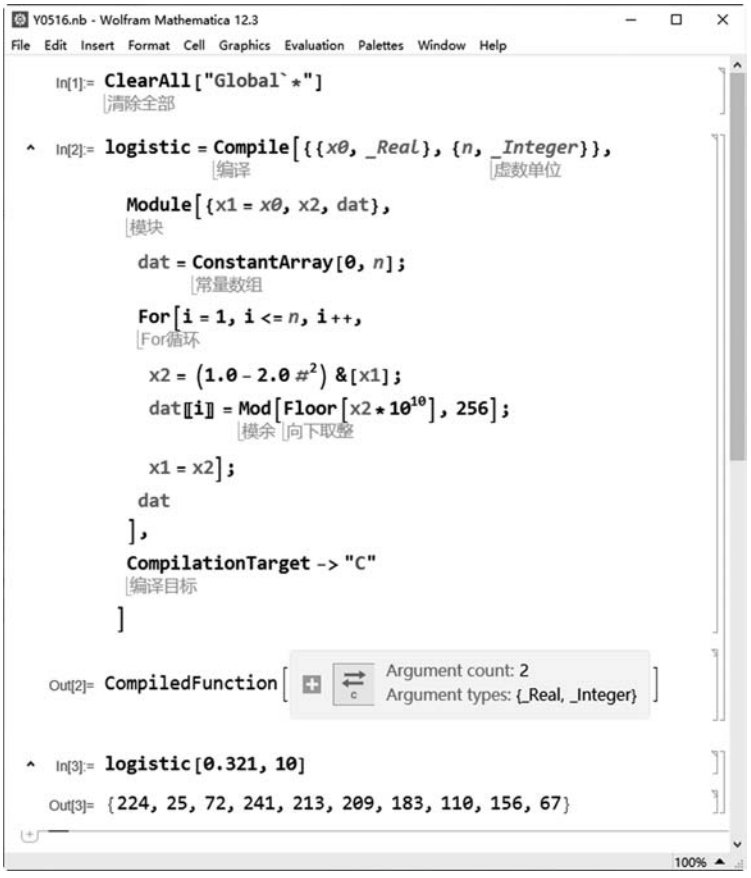

图 5-28　Logistic 混沌序列发生器

在图 5-28 中，“In[2]”使用 Compile 函数定义了函数 logistic，具有两个参数 x0 和 n，分别表示 Logistic 映射的初值和生成的序列的长度，这里的“x0”和“n”在函数内部应视为常数，不能作为变量使用。在 Module 模块中，定义了局部变量 x1(赋初值 x0)、x2 和 dat，这里的“x1”和“x2”用于表示 Logistic 映射的两个状态，“dat”用于存储生成的伪随机序列。然后，“dat＝ConstantArray[0，n]”将 dat 变量初始化为长度为 n 且元素值为 0 的一维列表。接着，在 For 循环中，循环变量 i 从 1 至 n，循环执行语句组“x2＝(1.0－2.0＃^{2})&[x1]；dat[[i]]＝Mod[Floor[x2＊10^{10}]，256]；x1＝x2”n 次，每次执行先由状态 x1 迭代得到状态 x2，然后，由状态 x2 得到序列的第 i 个值 dat[[i]]，再将 x2 赋给 x1 进行下一次迭代。这里的 Logistic 映射使用了纯函数的形式“(1.0－2.0＃2)&”。

在“In[3]”中，调用了 logistic 自定义函数“logistic[0.321，10]”，设置初始值参数为 0.321，序列长度参数为 10，得到结果“{224，25，72，241，213，209，183，110，156，67}”，如“Out[3]”所示。

实例二　RC4 加密与解密数据流

RC4 密码，全称为“Rivest Cipher 4”，是一种典型的分组密钥，习惯上称之为流密码，因为 RC4 可用于互联网中的实时数据流传输。RC4 的密钥长度可为 1～256B，建议实际保密通信应用中使用 128 字节以上的密钥。

这里，设 ***p*** 表示明文，***k*** 表示密钥，c 表示密文，均为基于字节的向量。RC4 加密过程如图 5-29 所示。

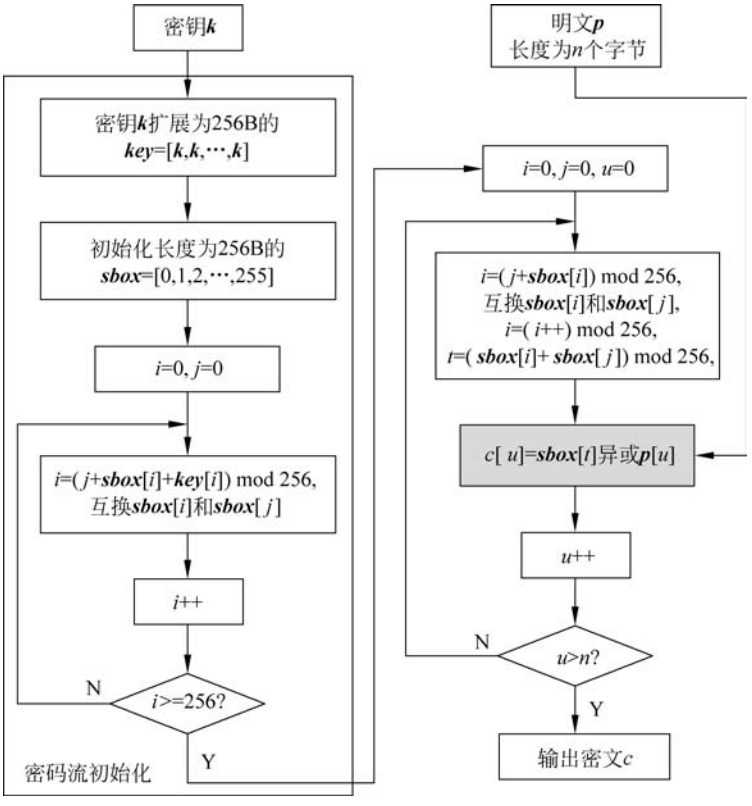

图 5-29　RC4 加密过程

结合图 5-29 可知，对于 RC4 加密过程，输入为密钥 $\boldsymbol{k}$ 和长度为 n 个字节的明文 $\boldsymbol{p}$，输出为长度为 n 个字节的密文 c。具体的加密步骤如下：

1. 密码流初始化

第 1 步：将密钥 $\boldsymbol{k}$ 扩展为长度为 256 字节的 $\boldsymbol{key}$。设密钥 $\boldsymbol{k}$ 的长度为 m 个字节，则

$$\boldsymbol{key}[i++]=\boldsymbol{k}[(i++) \bmod m],\ i=0,1,2,\cdots,255$$

第 2 步：初始化长度为 256 字节的数组 $\boldsymbol{sbox}$，即 $\boldsymbol{sbox}=[0,1,2,\cdots,255]$。

第 3 步：循环变量 i 从 0 至 255，循环执行以下两条语句：

① $j=(j+\boldsymbol{sbox}[i]+\boldsymbol{key}[i]) \bmod 256$；

② 互换 $\boldsymbol{sbox}[i]$与 $\boldsymbol{sbox}[j]$的值。

经过上述 3 步得到的 $\boldsymbol{sbox}$ 称为初始密码流。

2. 加密算法

已知明文 $\boldsymbol{p}$ 的长度为 n。初始化变量 $i=0$、$j=0$。变量 u 为 $0\sim n-1$，循环执行以下语句：

① $j=(j+\boldsymbol{sbox}[i]) \bmod 256$；

② 互换 $\boldsymbol{sbox}[i]$与 $\boldsymbol{sbox}[j]$的值；

③ $t=(\boldsymbol{sbox}[i]+\boldsymbol{sbox}[j]) \bmod 256$；

④ $i=(i++) \bmod 256$；

⑤ $\boldsymbol{c}[u]=\boldsymbol{sbox}[t]$ 异或 $\boldsymbol{p}[u]$。

最后得到的 $\boldsymbol{c}$ 即为密文。

需要注意的是，RC4 密码不是一次一密算法，使用 RC4 密码的通信双方在“密码流初始化”之后，将随着图 5-29 中循环变量 u 的增加持续加密过程。RC4 可能的不安全性在于密码流的重复(或循环再现)。因此，RC4 密码不宜长期使用，在使用一段时间(加密了足够长的数据)后，应借助于公钥技术替换 RC4 密码的密钥 $\boldsymbol{k}$。此外，RC4 不宜加密大量的重复性内容，这种情况下即使密码流是变化的，仍然有信息泄露的危险。

RC4 密码的解密过程与加密过程相似，但有两点不同：①输入为密钥 $\boldsymbol{k}$ 和密文 $\boldsymbol{c}$，输出为还原后的明文 $\boldsymbol{p}$；②图 5-29 中有灰色填充的方框中的内容由原来的“$\boldsymbol{c}[u]=\boldsymbol{sbox}[t]$ 异或 $\boldsymbol{p}[u]$”变为“$\boldsymbol{p}[u]=\boldsymbol{sbox}[t]$ 异或 $\boldsymbol{c}[u]$”。

借助于 Compile 函数实现的 RC4 密码(密钥为 $\boldsymbol{k}$、明文为 $\boldsymbol{p}$、密文为 $\boldsymbol{c}$)自定义函数如图 5-30 和图 5-31 所示。

在图 5-30 中，“In[2]”用 Compile 函数定义了函数 stream，该函数具有一个输入参数 k，表示密钥，为在 0～255 取值的整数序列(一维列表)，如“In[3]”所示。在“In[3]”使用了 20 个字节表示的整数序列作为密钥 k，即密钥 k 的长度为 160 比特。

回到图 5-30 的“In[2]”，在 Compile 函数内部的 Module 函数中，定义了局部变量 key 保存密钥 k 扩展至 256 字节后的密钥，用语句“key＝PadRight[k,256,k]”实现，这里的“PadRight”函数将密钥 k 向右填充为长度为 256 的列表，使用密钥 k 填充，相当于把密钥 k 循环扩展为长度为 256 的序列。Module 函数还定义了局变量 j，并初始化为 0；定义了局部变量 sbox，在“sbox＝Range[0,255]”，将 sbox 初始为列表{0，1，…，255}。在 For 循环

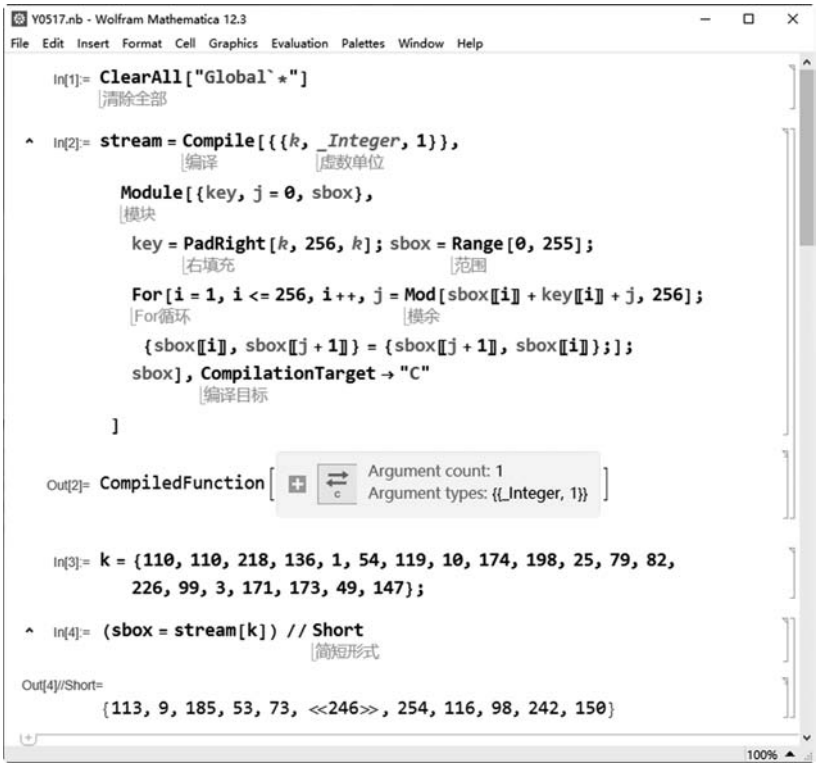

图 5-30 密码流初始化

“For[i=1, i<=256, i++, j=Mod[sbox[[i]] + key[[i]] + j, 256]; {sbox[[i]], sbox[[j+1]]}={sbox[[j+1]], sbox[[i]]};]”中，根据密钥 key 的值，打乱 sbox 中各个元素的顺序。然后，sbox 作为 stream 函数的输出。

在图 5-30 中的“In[4]”中，输入密钥 k，由“(sbox=stream[k])//Short”得到加密用的 sbox，这里“Short”函数表示仅显示 sbox 的部分数据，sbox 为一个长度为 256 的列表，元素为 0～255 共 256 个整数的特定的乱序排列。

现在，由图 5-30 过渡到图 5-31。在图 5-31 中，由 Compile 函数定义了函数 rc4，具有两个参数，一个为表示明文序列的 p，另一个为表示密钥的 k。在 Module 函数内部，定义了 6 个局部变量，其中，i 和 j 均初始化为 0；n 存储明文序列的长度，即“n=Length[p]”；sbox 由密钥 k 经图 5-30 的函数 stream 得到，即“sbox=stream[k]”；c 保存密文，初始化为元素均为 0 的列表，即“c=ConstantArray[0,n]”，密文 c 的长度与明文 p 的长度相同。在 Table 函数中实现对明文序列 p 的加密，其中，u 为循环变量(1～n)，循环执行以下操作：

(1) j=Mod[j+sbox[[i+1]], 256]，根据 i 的值借助于 sbox 更新 j 的值；

(2) {sbox[[i+1]], sbox[[j+1]]}={sbox[[j+1]], sbox[[i+1]]}，交换 sbox 的第 i+1 个和第 j+1 个元素；

(3) t=Mod[sbox[[i+1]]+sbox[[j+1]], 256]，由 sbox 的第 i+1 个和第 j+1 个元素之和得到临时变量 t 的值；

(4) i=Mod[i+1, 256]，更新 i 的值；

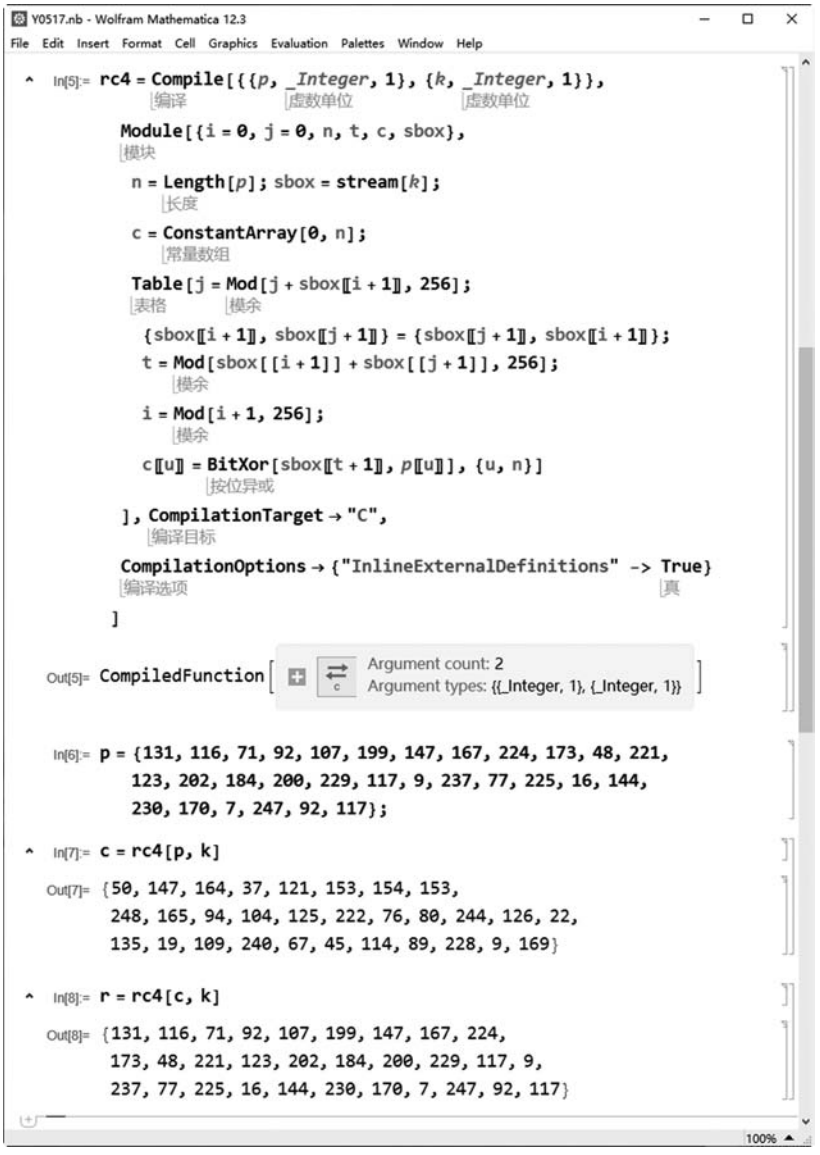

图 5-31　RC4 密码

(5) c[[u]]=BitXor[sbox[[t+1]], p[[u]]],由 sbox 的第 t+1 个元素与明文 p 的第 u 个元素相异或得到密文 c 的第 u 个元素。这些操作对应着 RC4 密码的加密过程。

最后,Table 函数的输出 c 为 Compile 函数的输出,c 即为加密后的密文。

回到图 5-30,密钥 k 为“{110, 110, 218, 136, 1, 54, 119, 10, 174, 198, 25, 79, 82, 226, 99, 3, 171, 173, 49, 147}”,在图 5-31 中,明文序列 p 为“{131, 116, 71, 92, 107, 199, 147, 167, 224, 173, 48, 221, 123, 202, 184, 200, 229, 117, 9, 237, 77, 225, 16, 144, 230, 170, 7, 247, 92, 117}”,如“In[6]”所示。在“In[7]”中调用“c=rc4[p,k]”,使用密钥 k 对明文 p 进行加密,得到密文 c 为“{50, 147, 164, 37, 121, 153, 154, 153, 248, 165, 94, 104, 125, 222, 76, 80, 244, 126, 22, 135, 19, 109, 240, 67, 45, 114, 89, 228,

9，169}”，如“Out[7]”所示。

在图 5-31 的“In[8]”中，调用“r＝rc4[c,k]”，使用密钥 k 对刚生成的密文序列 c 进行解密，解密后的文本 r 为“{131，116，71，92，107，199，147，167，224，173，48，221，123，202，184，200，229，117，9，237，77，225，16，144，230，170，7，247，92，117}”，如“Out[8]”所示，与原始的明文序列 p 完全相同。这是因为，上述 RC4 密码的加密函数和解密函数是相同的。

由于在图 5-31 的 rc4 函数中调用了 stream 函数，故使用了选项“CompilationOptions－>{"InlineExternalDefinitions" －> True}”。

5.5 并行编程

对于复杂的算法，设计其并行计算处理算法是件非常困难的事情，需要考虑算法执行过程中可能出现的诸多因素。在 Wolfram 语言中，并行计算由 Wolfram 内置函数 Parallelize 管理，这是一个为并行计算高效分配算法资源的函数。内置函数放在 Parallelize 函数中，Wolfram 语言将自动进行并行化处理，以尽可能多的并行指令执行这些内置函数。

在 Mathematica 软件的笔记本中，选择菜单“Edit | Preferences…”，在弹出的窗口“Preferences”中选择选项卡“Parallel”，并在该页面选择“Local Kernels”页面，在这个页面可以设定执行并行计算的内核数。可设定的内核数受计算机的 CPU 内核总数以及 Mathematica 的版权限制，最多可支持 16 个并行内核。设定了并行内核个数后，下面介绍并行编程相关的内置函数，首先介绍并行计算函数 Parallelize 函数，然后介绍并行处理函数 ParallelTable、ParallelMap、ParallelDo、ParallellSum、ParallelProduct 和 ParallelArray 函数等。

5.5.1 并行计算函数

Parallelize 函数的语法为“Parallelize[语句组]”，自动使用并行计算方式计算“语句组”，Parallelize 有一个 Method 选项，常用的选项为“Method－>"FinestGrained"”和“Method－>"CoarsestGrained"”，分别表示将计算任务分成尽可能小的计算子单元（细粒度划分）和将计算任务按计算机并行内核的数量进行分隔（粗粒度划分）。

这里用 Parallelize 函数计算乘方 x^y，使用“平方—乘算法”。“平方—乘算法”实现乘方 x^y 的方法为：将 y 转化为二进制形式，例如 y 为 10101110b，则 x^y 为 $x^{10101110b}$。令 s＝1，从 y 的二进制序列的最左边为 1 的位开始向右遍历，当某位为 1 时，s 的平方乘以 x 赋给 s，即 $s=s^2 \cdot x$；当某位为 0 时，s 的平方赋给 s，即 $s=s^2$。这一过程本质上由 y 的二进制序列构造 y 的十进制数的值，因此，“平方—乘算法”是正确的。

Parallelize 函数的典型实例如图 5-32 所示。

图 5-32 中，“In[2]”为实现“平方—乘算法”的自定义函数 sqmul，具有两个整型参数 x 和 y，计算 xy。在 Module 模块中，定义了两个局部变量 h 和 s，h 用于保存 y 的二进制数的各个数位，“h＝IntegerDigits[y，2]”；s 用于保存最后的结果，初始化为 1。Table 函数实现“平方—乘算法”，“Table[s＝s2；If[t＝＝1，s＝s＊x]，{t，h}]”，循环变量 t 在列表 h 中取

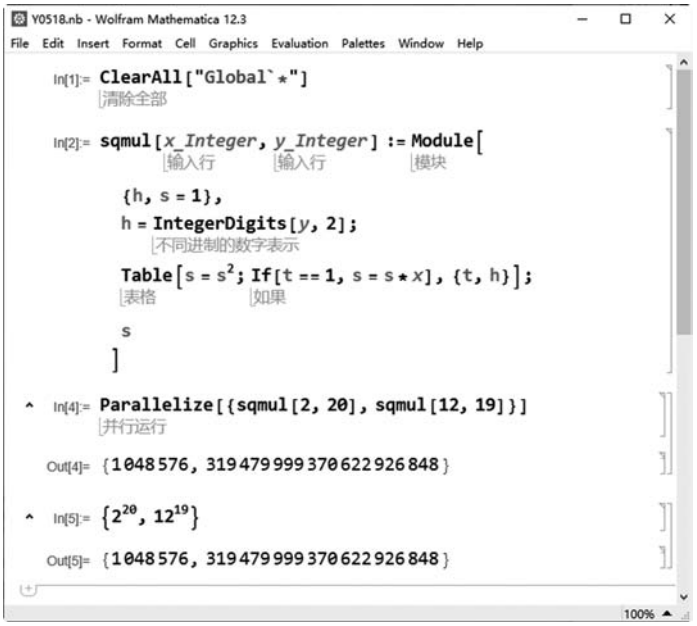

图 5-32　Parallelize 函数典型实例

值，当 t 为 0 时，s 的平方赋给 s；当 t 为 1 时，s 的平方乘以 x 赋给 s。最后，s 作为函数 sqmul 的返回值。在“In[4]”中将“{sqmul[2,20]，sqmul[12,19]}”作为函数 Parallelize 的参数，执行结果如“Out[4]”所示。“In[5]”和“Out[5]”为 Wolfram 语言计算 2^{20} 和 12^{19} 的结果，以供参考对比。

需要注意的是，并非使用了并行计算函数后，计算任务的执行时间一定会大幅度减少。并行计算需要将原来的计算任务划分为可并行执行的计算子单元，并需要考虑各个计算子单元间的数据通信，对于并不复杂的运算任务而言，这些并行预处理工作花费的时间可能比计算任务本身的执行时间更长。并行计算主要应用于非常复杂和耗时的计算任务中。

5.5.2　并行处理函数

并行处理函数都有对应的单线程函数，例如，ParallelTable 函数与单线程函数 Table 函数对应，其语法也类似；ParallelMap 函数与单线程函数 Map 对应，其语法也类似。但是并行处理函数将其中的变量局部化，如果读取其中的变量，必须使用 SetSharedVariable 函数使这些变量全局化。

这里主要介绍常用的并行处理函数，其中，图 5-33 演示了 ParallelTable 和 ParallelDo 函数的典型用法。

在图 5-33 中，“In[2]”调用 Table 函数“Table[Pause[1];i^2,{i,4}]//AbsoluteTiming”并统计其执行的时间，这里“Pause[1]”为延时 1 秒，由于 Table 函数是单线程顺序执行语句组“Pause[1]；i^2”4 次（循环变量 i 为 1～4），故运行时间至少为 4 秒，结果“Out[2]”显示运行时间为 4.06139 秒。而“In[4]”调用并行处理函数 ParallelTable 函数“ParallelTable[Pause[1]；

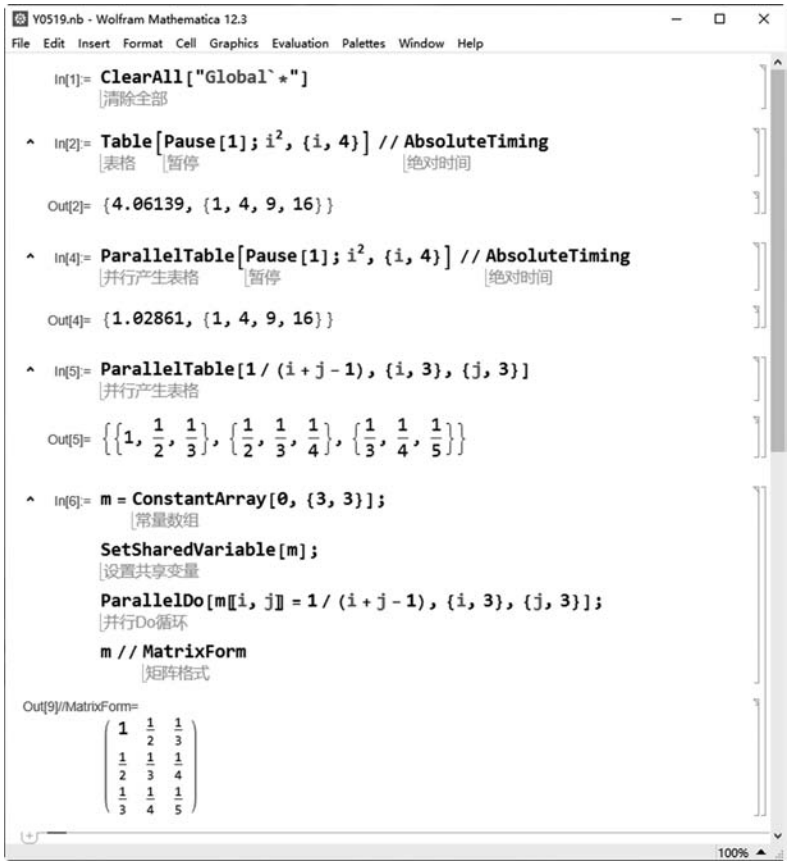

图 5-33 并行处理函数 ParallelTable 和 ParallelDo

i^2, {i,4}]//AbsoluteTiming",这里并行执行语句组"Pause[1]; i^2",执行时间如"Out[4]"所示,为 1.02861 秒。除了将变量局部化外,ParallelTable 与 Table 的作用相同,在"In[5]"中,"ParallelTable[1/(i+j-1),{i,3},{j,3}]"计算了 3 阶 Hilbert 矩阵,如"Out[5]"所示。

在图 5-33 中,"In[6]"定义了全局变量 m,为 3×3 的全 0 矩阵;然后,调用函数"SetSharedVariable[m]"将 m 设为并行处理共享变量,接着,调用并行处理函数 ParallelDo 函数"ParallelDo[m[[i,j]]=1/(i+j-1),{i,3},{j,3}]"计算 3 阶 Hilbert 矩阵,最后,以矩阵形式输出 m,如"Out[9]"所示。

使用并行处理函数时需要注意,如果程序本身是顺序执行的方式设计的,即前面语句组的执行结果将影响其后的语句组,此时使用并行处理函数可能得不到正确的结果,如图 5-34 所示。

在图 5-34 中,"In[6]"拟使用并行 ParallelTable 函数计算 $1^2+2^2+\cdots+10^2$,但是这个函数对于每个循环变量 i 的取值作并行处理,最后得到的结果 s 为 i=10 和 s 初始值为 0 的情况下的值,即 100,如"Out[9]"所示。这不是设计的算法的正确结果。这时应使用"In[10]"的并行求和函数 ParallelSum,即"ParallelSum[i^2,{i,10}]",计算结果为 385,如"Out[10]"所示。此外,Wolfram 语言还有并行乘法函数 ParallelProduct,在"In[11]"中计算了 10 的阶乘,即"ParallelProduct[i,{i,10}]",结果为 3628800,如"Out[11]"所示。

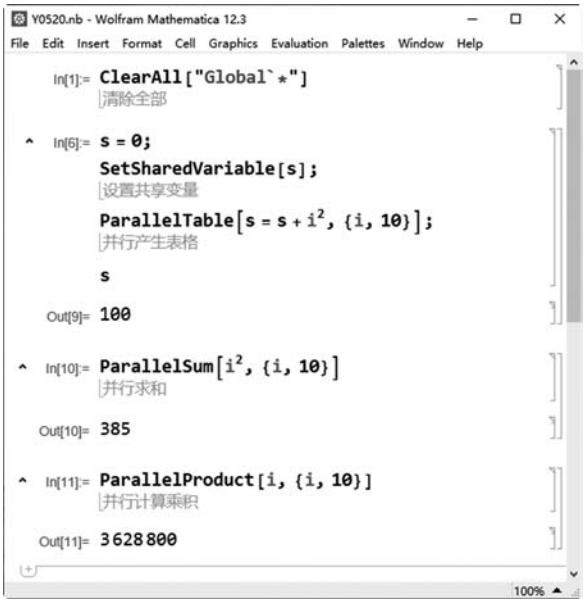

图 5-34　并行处理函数的注意事项

最后需要介绍的三个常用并行处理函数为 ParallelMap、ParallelArray 和 ParallelEvaluate，如图 5-35 所示。

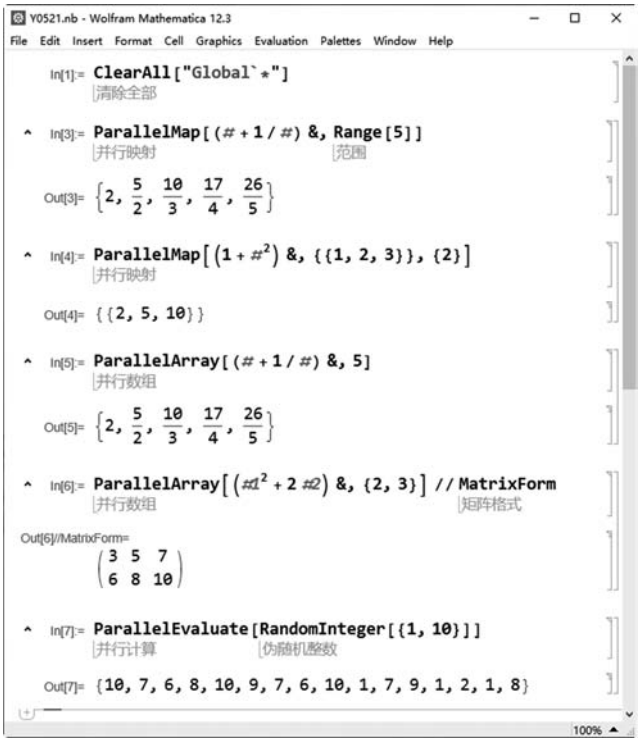

图 5-35　并行处理函数 ParallelMap、ParallelArray 和 ParallelEvaluate

在图 5-35 中，“In[3]”使用平行处理函数 ParallelMap 将纯函数“(＃＋1/＃)&”作用于列表“Range[5]”(即“{1,2,3,4,5}”上，得到结果如“Out[3]”所示。ParallelMap 函数是 Map 函数的并行版本，同样可以作用于列表的不同层上，例如“In[4]”中，ParallelMap 函数将纯函数“(1＋＃^{2})&”作用于列表“{{1,2,3}}”第 2 层上的各个元素上，得到结果如“Out[4]”所示。

ParallelArray 函数是 Array 函数的并行版本，图 5-35 中的“In[5]”使用 ParallelArray 函数将纯函数“(＃＋1/＃)&”作用于{1,2,3,4,5}上，输出结果如“Out[5]”所示，与“In[3]”的计算结果相同。“In[6]”使用 ParallelArray 函数将纯函数“(＃1^2＋2＃2)&”作用于{i,j}上，这里 i={1,2},j={1,2,3}，得到二维列表如“Out[6]”所示。

ParallelEvaluate 函数是单线程 Evaluate 函数的并行版本，“ParallelEvaluate[表达式]”将使用所有的并行内核计算“表达式”的值，而且各个内核之间互不相关。在图 5-35 的“In[7]”中，“ParallelEvaluate[RandomInteger[{1,10}]]”使用所有的并行内核执行函数“RandomInteger[1, 10]”得到 1～10 的伪随机整数，如“Out[7]”所示，这里共 16 个内核，得到一个长度为 16 的伪随机整数列表。

本章小结

Wolfram 语言有 6000 多个内置函数，涵盖了数学、物理、化学、生物和计算机科学等众多领域的常用计算方法，熟练掌握并灵活应用这些内置函数是精通 Mathematica 软件的关键，而有效地组织内置函数的方法是借助于模块编程技术。本章详细介绍了 Wolfram 语言的四种模块化编程方法，即 Module 模块、Block 模块、With 模块和 Compile 模块。Module 模块可以定义局部变量并为局部变量赋初值，并可以组织任意多的内置函数共同完成特定的计算功能，是最常用的模块化编程手段。与 Module 模块类似，Block 模块也可以定义局部变量并为局部变量赋初值，但是 Block 模块还可以将全局变量的值局部化，即给全局变量赋予新的值，并使这个值的作用域为整个 Block 模块，而 Block 模块外部，全局变量的值不受影响。一般地，认为 Block 模块可以替代 Module 模块，也就是说，所有的 Module 函数均可以直接变换为 Block 函数，其计算结果仍然正确且执行效率更高。Compile 函数是针对数值计算的情况下，将 Wolfram 语言函数编译为机器代码以提高算法的执行效率。这种优化本质上为 Compile 函数中使用的内置函数提供了机器代码级别的接口，可以使用 Wolfram 语言自带的编译器生成 Wolfram 语言虚拟机上执行的机器代码，也可以借助于外部编译器(如 Visual Studio 或 MinGW64)等对 Compile 函数进行编译优化。有时需要对比两个算法的运算速度，此时，借助于单线程的 Compile 模块可以相对公平地评测算法速度性能。