第四章：控制流程

在前三章中，读者已经学习了Yak语言的基础数据类型、复合类型以及如何使用表达式和操作符来执行简单的运算和数据操作。掌握了这些知识，读者可以编写一些基本的程序，但这些程序通常是直线式的，即代码从上到下依次执行，没有任何的分支和循环。

为了编写更加复杂和有用的程序，我们需要引入控制流程的概念。控制流程是程序设计中的一个核心概念，它允许程序根据不同的条件执行不同的代码路径，或者重复执行某段代码直到满足特定条件。简而言之，控制流程给予了程序决策能力和重复执行的能力。

在本章中，我们将介绍Yak语言中实现控制流程的结构和语法，包括：

- 条件语句：如 `if`、`else` 和 `switch`，它们让程序能够根据条件的不同选择执行不同的代码块。
- 循环语句：如 `for` 和 `while`，它们能够重复执行一段代码直到指定的条件不再满足。
- 跳转语句：如 `break` 和 `continue`，它们用于在循环中提前跳出当前循环或跳过当前循环的剩余部分。

通过学习这些控制流程结构，读者将能够编写出更加动态和响应不同情况的程序。本章的目标是让读者理解和掌握如何根据不同的运行时情况，控制程序的执行流程。这不仅是编程的基础，也是编写高效、可读性强和可维护程序的关键。

4.1 条件分支语句

在编程中，条件分支语句是构建程序逻辑的基石，它们使得程序能够根据不同的条件执行不同的操作。在Yak语言中，条件分支可以通过两种主要的结构来实现：`if` 语句和 `switch` 语句。

4.1.1 IF 语句

`if` 语句是最基本的条件分支结构。它允许程序在满足特定条件时执行代码块，在不满足时则跳过该代码块或执行另一个代码块。Yak语言中的 `if` 语句非常灵活，支持多种条件和嵌套结构。通过 `else` 关键字，程序可以在 `if` 条件不满足时执行备选的代码路径。此外，`else if` 结构允许在多个不同条件之间进行选择。

简单的条件判断结构

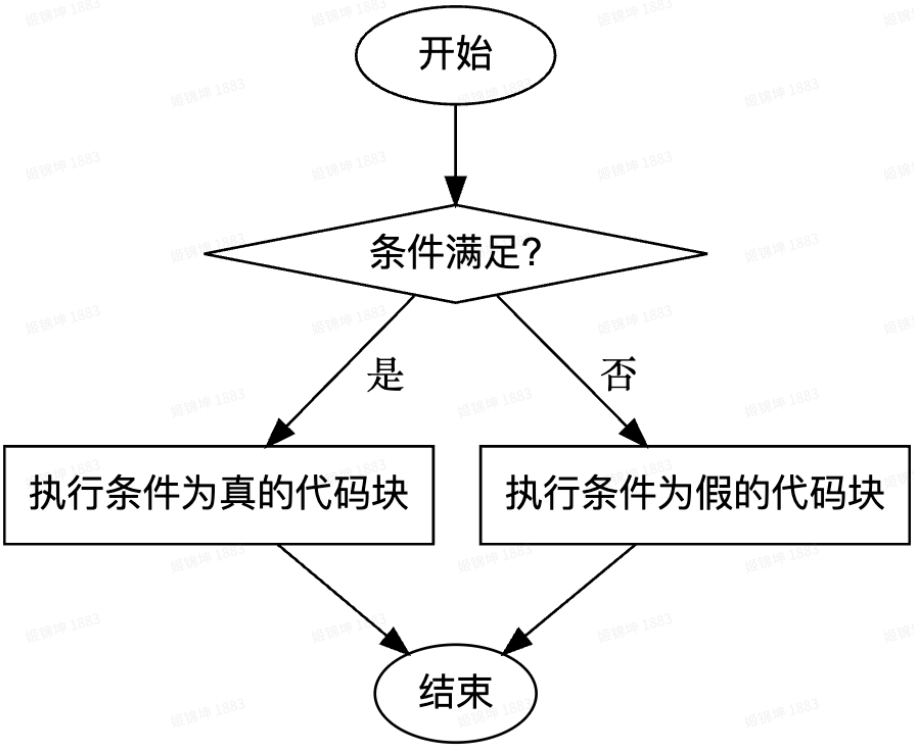
一个基础的 `if` 语句在Yak语言中的结构如下：

```
1 if condition {
2     // 条件为真时执行的代码块
3 } else {
```

```
    // 条件为假时执行的代码块
}
```

- `condition` 是一个布尔表达式，它的结果只能是 `true` 或 `false`。
- 如果 `condition` 为 `true`，则执行 `if` 后面的花括号 `{}` 中的代码。
- 如果 `condition` 为 `false`，则执行 `else` 后面的花括号 `{}` 中的代码。

读者可以参考这个流程图来理解 IF 语句的基本使用



为了方便用户理解，可以参考这个案例：假设你正在编写一个程序，根据天气情况来决定穿什么衣服。如果天气冷，你应该穿上外套；如果天气不冷，就不需要穿外套。

```
isCold = true; // 假设今天天气冷
if isCold {
    print("穿上外套");
} else {
    print("不需要穿外套");
}
```

在这个例子中，`isCold` 是一个布尔变量，它被设置为 `true`，表示天气冷。因此，`if` 语句中的条件判断为真，程序会执行 `print("穿上外套")`。

如果你只想处理某个特殊情况，并不想处理条件为假的情况，Yak语言可以直接忽略 else 语句，此时只会对条件为真时运行语句，条件为假时跳过条件之后的代码块，继续运行主程序：

```
if 布尔表达式 {
    // 当条件为真时执行的语句
}
```

嵌套的条件判断结构

在很多情况下，在使用IF语句时各种情况的判断会出现优先级问题。

比如以下的问题：*对于学科 A 的成绩，我们定义90即以上为非常优秀， 80-90 (包括80，下同)分为优，80-70 位良，70-60 为普通，60 以下为不及格，请判断学生的成绩 x = 88 分为什么范围?*

在此问题中，如果通过普通判断的话，将需要写一大片x >=70 && x<80这样的语句，但是如果我们引入条件的优先级，只需要先判断x >= 90，x >= 80即可。

在编程中处理这种情况时，通常会使用一系列的 `if-else if-else` 结构，来确保按照特定的顺序评估每个条件。在Yak语言中，我们也可以采用这种结构，来按照优先级判断学生成绩的范围。

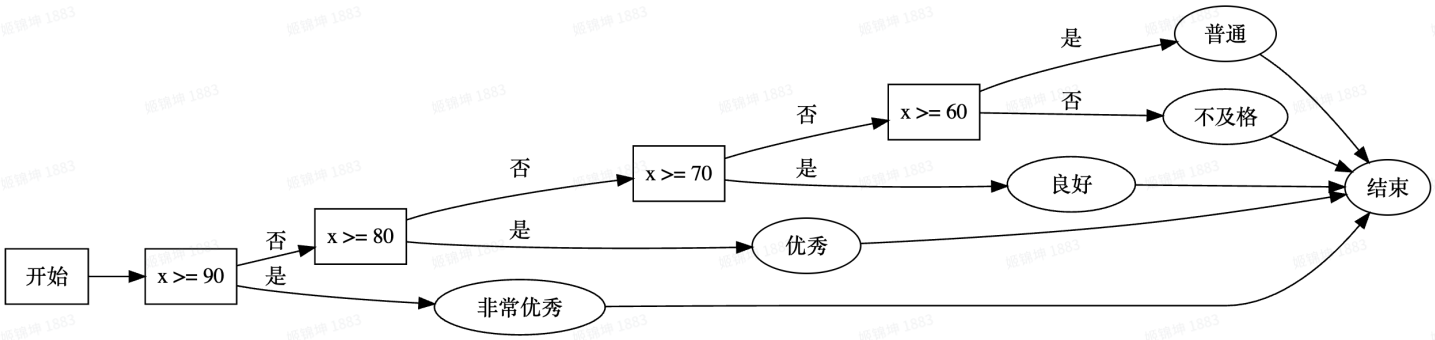
以下是如何使用嵌套的条件判断结构来解决上述问题：

```
x = 88; // 学生的成绩
if x >= 90 {
    print("非常优秀");
} else if x >= 80 {
    print("优秀");
} else if x >= 70 {
    print("良好");
} else if x >= 60 {
    print("普通");
} else {
    print("不及格");
}
```

在这个结构中，程序将从上到下依次检查每个条件：

1. 首先检查 `x` 是否大于等于90。如果是，打印 `"非常优秀"` ，然后跳过剩余的所有条件。
2. 如果 `x` 小于90，程序将继续检查 `x` 是否大于等于80。如果是，打印 `"优秀"` 。
3. 这个过程会继续，直到找到符合条件的范围，或者所有条件都不满足，最后打印 `"不及格"` 。

这种方法的优点是逻辑清晰，且不需要复杂的逻辑表达式来处理每个范围。每个 `else if` 块只会在前面的条件不满足时才会被评估，这样就保证了评估的顺序和优先级。我们把上面的逻辑绘制一个流程图，可以帮助大家很容易理解嵌套条件判断结构。



> 注意：在Yak语言中，`else if`和`elif`是完全等价的，读者可以挑选自己的编程习惯进行编程。

简化的条件表达式：三元运算符

三元运算符是一种常见的语法糖，它允许程序员在一行内进行简单的条件赋值。这种运算符通常用于替代简单的`if-else`语句，使代码更加简洁明了。基本语法结构是：

```
变量 = 条件 ? 真值表达式 : 假值表达式;
```

如果`条件`为真（即`true`），则整个表达式的值会是`真值表达式`的结果；如果条件为假（即`false`），则表达式的值会是`假值表达式`的结果。这种结构在许多编程语言中都非常相似，包括JavaScript、C、C++、Java和Python（通过使用不同的语法）。

这里是`Yak`语言示例，它演示了如何使用三元运算符：

```
condition = true
value = condition ? 1 : 0
println(value)
```

在这个例子中，变量`value`将会被赋值为`1`，因为`condition`是`true`。如果`condition`是`false`，那么`value`会被赋值为`0`。最后，`value`的值被打印出来。

三元运算符非常适合于简单的条件赋值，但如果涉及到更复杂的逻辑或多个条件，通常建议使用完整的`if-else`结构，因为这样的代码更容易阅读和维护。

4.1.2 SWITCH 语句

在上一节中，笔者详细介绍了IF语句的使用，它是实现条件分支的一种基本方式，适合处理较为简单或条件数目不多的场景。随着读者对条件逻辑的掌握日渐深入，接下来我们将转向另一种用于控制程序流程的重要结构——SWITCH语句。SWITCH语句在处理多条件分支时更为直观和便捷，尤其是当涉及到多个离散值需要被单独处理时。接下来，笔者将引导读者一探SWITCH语句的奥妙，看看它如何简化复杂的决策逻辑，并让代码更加清晰易读。

在Yak语言中，`SWITCH`语句提供了一种高效的方法来执行多路分支选择。这种语句允许程序根据一个表达式的值来选择不同的代码执行路径。相比于多个`if-else`语句，`SWITCH`语句在处理多个固定选项时更为清晰和直接。

基础概念

下面是Yak语言中`SWITCH`语句的语法定义：

```
switchStmt: 'switch' expression? '{' (ws* 'case' expressionList ':'
statementList?)* ( ws* 'default' ':' statementList?)? ws* '}';
```

根据这个定义，`SWITCH`语句的结构包括以下几个关键部分：

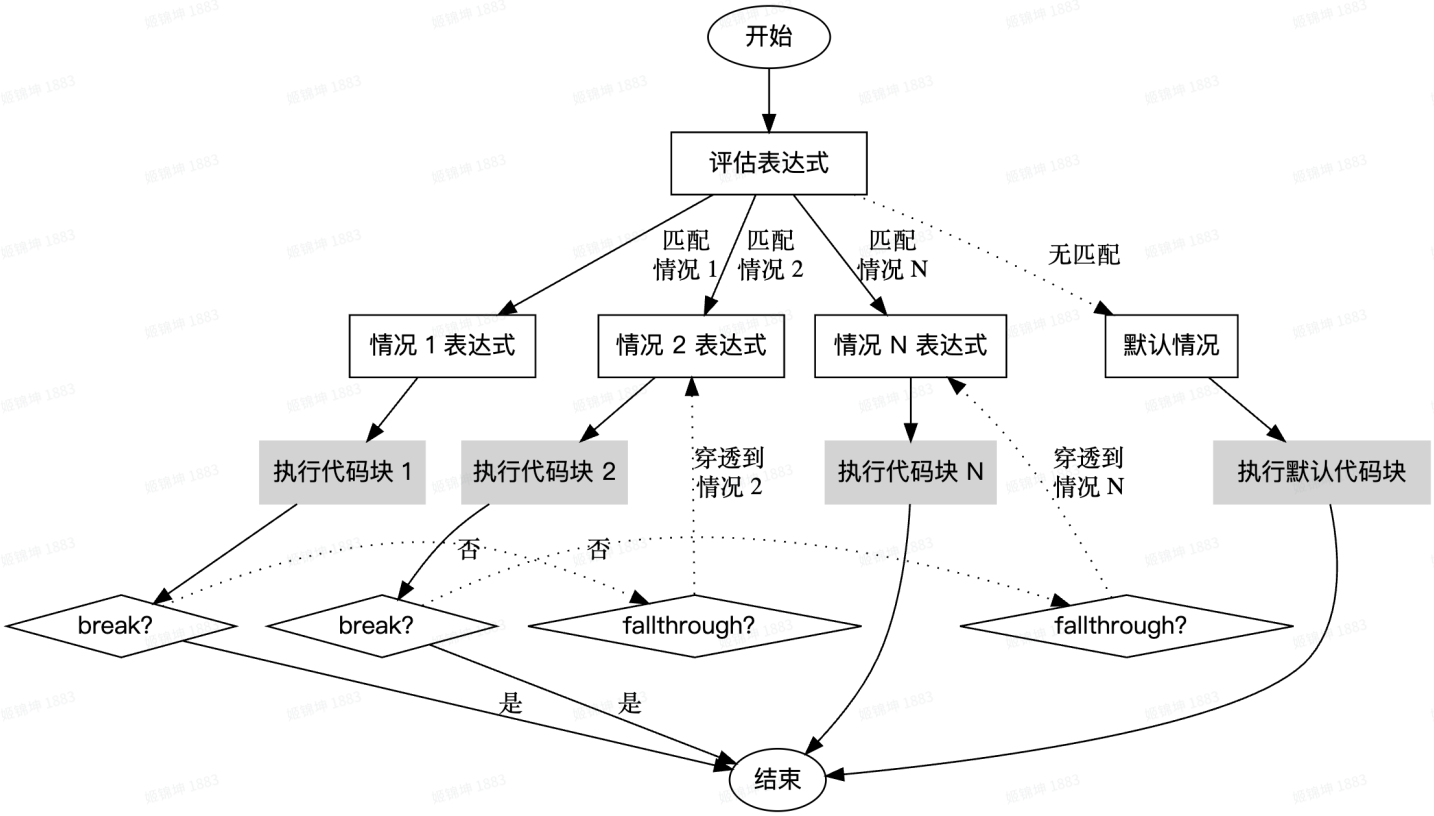
1. `switch`关键字：标记一个`SWITCH`语句的开始。
2. `expression`：这是一个可选项，其值用于决定哪一个`case`分支将被执行。
3. `case`关键字：后面跟随一个`expressionList`，表示当`switch`的表达式值与`case`后的表达式列表中的某个值匹配时，应执行该`case`所关联的`statementList`。
4. `default`关键字：这是一个可选项，其后的`statementList`在没有任何`case`匹配时执行。
5. 大括号`{}`：包围整个`SWITCH`语句的主体。

在一个`SWITCH`语句中，程序会评估`switch`后的表达式，并将结果与每个`case`后的表达式列表进行比较。一旦找到匹配项，相应的`statementList`将被执行。如果没有任何`case`匹配，且存在`default`语句，那么`default`后的`statementList`将被执行。

在一些语言中`switch`运行一个情况内的代码结束以后将会继续向下运行，此时需要`break`语句来跳出`switch`语句。在Yak中则是运行结束直接跳出`switch`语句，如果需要继续运行下一个情况的代码则需要使用`fallthrough`，当然如果希望在代码中某个情况下直接跳出`switch`，Yak也是支持`break`语句的，这会让代码更简洁。把上面的描述总结一下，switch 的语法如下：

```
switch 表达式 {
    case 数值1 :
        // 代码1
        break // 可选
    case 数值2 :
        // 代码2
        fallthrough // 可选
    // 在此处可以写任意数量的case语句
    default:  // 可选
        // 代码default
}
```

当然用户可以结合下面的流程图例来理解 Switch 语句



基础使用案例

```
grade = 'B'
switch (grade) {
  case 'A':
    println("优秀");
  case 'B':
    println("良好");
  case 'C':
    println("合格");
  case 'D':
    println("需要努力");
  default:
    println("无效的成绩");
}
```

在这个例子中， `grade` 是一个变量，其值用于与 `case` 语句中列出的等级进行比较。每个 `case` 块中的代码对应于不同的成绩评价。如果 `grade` 变量的值没有在任何 `case` 中列出，则执行 `default` 块中的代码，打印出"无效的成绩"。

分支多值匹配

除此之外，读者还可以阅读以下案例来进一步理解 switch 的其他用法：

```
switch a {
case 1, 2:
    println("a == 1 || a == 2")
default:
    println("default")
}
```

Yak语言的 `switch` 允许一个 `case` 分支匹配多个值。这段代码的意思是：

1. `switch` 关键字开始一个 `switch` 语句，它是一种多路分支结构。
2. `a` 是这个 `switch` 语句要检查的变量。
3. `case 1, 2:` 表示如果变量 `a` 的值等于1或者2，那么就执行冒号后面的代码块。在这个例子中，如果 `a` 等于1或2，程序将执行 `println("a == 1 || a == 2")`，打印出 `a == 1 || a == 2`。
4. `default:` 关键字用于定义一个默认的代码块，它将在没有任何其他 `case` 匹配时执行。在这个例子中，如果 `a` 的值既不是1也不是2，那么程序将执行 `println("default")`，打印出 `default`。

这个 `switch` 语句没有包含 `break` 语句，因为在Yak语言中，每个 `case` 块在执行完毕后会自动退出 `switch` 语句，不会发生C语言中那样的"fallthrough"现象（除非显式地使用了特定的 `fallthrough` 关键字）。

表达式匹配

```
switch {
case 1==2:
    println("1 == 2")
case 2 == 2:
    println("2 == 2")
default:
    println("default")
}
```

Yak语言中 `switch` 语句允许没有指明要检查的特定变量的情况，而是直接对表达式进行检查。这种类型的 `switch` 通常被称作"表达式匹配"。对上述代码的解释：

1. `switch` 关键字开始一个没有显式检查变量的 `switch` 语句。
2. `case 1==2:` 检查表达式 `1==2` 是否为真。这个表达式的结果显然是 `false`，因为1不等于2，所以这个 `case` 分支不会被执行。
3. `case 2==2:` 检查表达式 `2==2` 是否为真。这个表达式的结果是 `true`，因为2等于2，所以这个 `case` 分支会被执行，程序将打印 `"2 == 2"`。

4. `default:` 关键字定义了一个默认的代码块，如果前面的所有 `case` 都不匹配时执行。但在这个例子中，由于 `2==2` 是真的，所以 `default` 分支不会被执行。

由于 `switch` 语句通常在匹配到第一个 `true` 的 `case` 之后就结束，不会继续检查后面的 `case`，因此在这个例子中，只有 `"2 == 2"` 会被打印， `switch` 语句在执行完 `case 2 == 2:` 分支之后就结束了。

综上所述，这段代码的流程是：

- 检查表达式 `1==2` 是否为真，如果为真则执行相关代码块（在这个例子中不会发生）。
- 检查表达式 `2==2` 是否为真，由于为真，所以打印 `"2 == 2"`。
- 由于已经有一个 `case` 匹配成功， `switch` 语句结束，不执行 `default` 分支。

通过上述介绍和例子，读者应该能够理解Yak语言中 `SWITCH` 语句的基本构造和用法。在实际编程中， `SWITCH` 语句是控制复杂条件逻辑的有力工具，能够使代码组织得更为条理清晰。

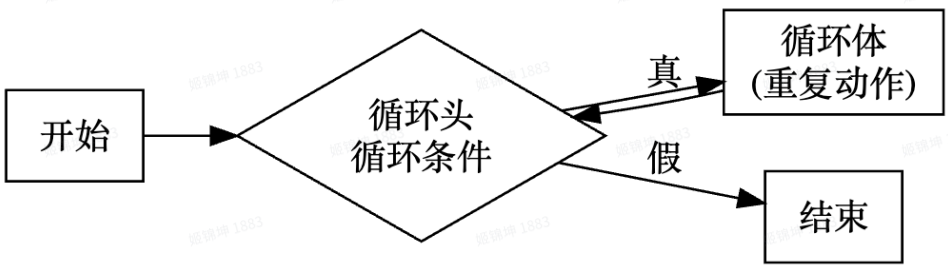
4.2 循环语句

在编程的世界里，读者已经掌握了条件分支语句的知识，这使得读者能够根据不同的条件执行不同的代码路径。但是，当读者面临需要重复执行某些操作直到满足特定条件时，仅仅使用条件分支是不够的。此时，读者需要了解并使用编程中的另一个核心概念——循环语句。

循环语句赋予了代码重复执行的能力，是实现自动化和重复任务的基础工具。如果将条件分支视为程序的决策点，那么循环则可以被看作是程序的脉搏，它保持着代码的动态运行，直到达成既定的目标。无论是遍历数据结构中的每个元素，还是等待用户输入，循环都是实现这些功能的关键。

笔者接下来将引导读者深入了解循环语句的各种形式和它们的应用，从基本的 `for` 循环使用到更复杂的 `for` 循环特性。笔者将一步一脚印地阐述如何在程序中有效利用循环语句。请读者准备好，一同深入循环语句的世界，让代码跳起精确而优雅的循环之舞。

一般情况下，循环的流程图表示如下：



几乎所有的循环都会遵循上图中的流程，但是针对不同的场景将会出现几种不同的 For 循环的语法，其中主要的区别在于循环判断语句的设置。

4.2.1 经典的 FOR 循环

最经典的 for 循环语法，定义三表达式：

```
for 表达式1 ; 表达式2 ; 表达式3 {
    循环体
}
```

Yak语言中的三表达式循环，首先运行表达式1，判断表达式2，如果成立则运行循环体中的代码，循环体运行结束执行表达式3，再次进行表达式2的判断并循环执行，直到表达式2判断为假，则结束整个函数。

`经典for循环表达式` 显式的声明了循环的起始语句、判断条件、迭代语句，避免了在循环体中混合函数迭代指令，代码更加清晰，而且通过 for 语句的编写即可表达整个 for 循环执行的次数判断是否出现无限循环，但是缺点就在于过于繁琐。

很多循环可能只了解他的判断条件而且无法设置初始化和迭代语句，此时的循环可以写为 `for ; condition ; {}`，对于这种写法 yak 提供了更加简单且具有可读性的语法：

```
for 布尔表达式 {
    循环体
}
```

这样的语法类似于while循环，只在布尔表达式为假的时候退出循环。

4.2.2 使用循环遍历对象

在前文中我们已经讲过复杂的数据类型：列表和字典，除了对于某些逻辑处理以外，循环另一个常见场景是遍历这种数据对象。

在Yak语言中，有两种循环语法用于遍历对象：

- `FOR-RANGE` 语法：`for key, value = range 遍历对象 {}`；
- `FOR-IN` 语法：`for value in 遍历对象`

两种语法的作用是一致的，主要为了降低拥有其他语言基础的读者的上手门槛，读者可以自行按照习惯选择。同时，两种语法在每次迭代时得到的数据会有些许的差别，接下来会详细的讲述。

Yak语言中可以进行遍历的对象一共有三种，接下来我们将会通过例子详细的说明对应的用法。

遍历列表

```
for i, v = range a {
    println(i, v)
}

/*
OUTPUT:
```

```
7
8 0 a
9 1 b
10 2 c
11 3 d
12 */
```

这段代码的解释如下：

- `for` 关键字启动一个循环。
- `i, v` 是我们在每次迭代中定义的两个变量，其中 `i` 将存储当前的索引，而 `v` 将存储与该索引对应的值。
- `range a` 是一个表达式，它创建了一个从集合 `a` 中提取索引和值的范围。
- 在 `for` 循环的大括号 `{}` 内，我们有一个循环体，其中包含了一个 `println` 函数调用，该函数将在每次迭代时执行。

当这段代码执行时，它会按顺序输出集合 `a` 中的每个元素及其索引。如果我们假设集合 `a` 包含元素 `['a', 'b', 'c', 'd']`，则输出将是：

```
1 0 a
2 1 b
3 2 c
4 3 d
```

输出解释：

- 在第一次迭代中，`i` 的值是 `0`，`v` 的值是 `'a'`，因此打印出 `0 a`。
- 在第二次迭代中，`i` 的值是 `1`，`v` 的值是 `'b'`，因此打印出 `1 b`。
- 在第三次迭代中，`i` 的值是 `2`，`v` 的值是 `'c'`，因此打印出 `2 c`。
- 在第四次迭代中，`i` 的值是 `3`，`v` 的值是 `'d'`，因此打印出 `3 d`。

这种循环结构非常有用，因为它允许程序员以一种简洁和直观的方式遍历数据结构中的所有元素。但是使用 `for-range` 循环有多种迭代方式。如果我们采用 `for i = range a {...}` 的形式，通常意味着我们只对集合 `a` 的索引感兴趣，而不关心对应的值。在这种情况下，循环将仅提供索引，而不会提供值。让我们来扩展这个教程，以说明这种情况。

在前面的例子中，我们使用了 `for` 循环和 `range` 关键字来遍历集合 `a` 的索引和值。然而，如果我们只需要索引，可以使用一种更简洁的形式。考虑以下代码：

```
1 for i = range a {
2     println(i)
```

```
3 }
```

这段代码的解释如下：

- `for` 关键字仍然表示我们将要开始一个循环。
- `i` 是我们在每次迭代中定义的变量，它将存储当前的索引。
- `range a` 是一个表达式，但这次我们没有提供一个用于存储值的变量，我们只获取索引。
- 循环体内只有一个 `println` 函数调用，它将在每次迭代时打印出索引 `i` 。

当这段代码执行时，它会按顺序输出集合 `a` 中元素的索引。如果集合 `a` 包含相同的元素 `['a', 'b', 'c', 'd']` ，输出将不再包含元素值，只有索引：

```
1 0
2 1
3 2
4 3
```

输出解释：

- 在第一次迭代中，我们得到索引 `0` ，打印出 `0` 。
- 在第二次迭代中，我们得到索引 `1` ，打印出 `1` 。
- 在第三次迭代中，我们得到索引 `2` ，打印出 `2` 。
- 在第四次迭代中，我们得到索引 `3` ，打印出 `3` 。

这种形式的 `for` 循环是非常有用的，特别是当你需要迭代的次数，或者当你只关心索引时。它简化了代码，并且在某些情况下可以提高代码的清晰度和执行效率。

除了 `for-range` 的格式之外，用户可以通过 `for-in` 得到几乎一样的效果：

```
1  a = ["a", "b", "c", "d"]
2  for v in a {
3      println(v)
4  }
5
6  /*
7  OUTPUT:
8
9  a
10 b
11 c
12 d
13 */
```

略有区别的是，如果用户使用 `for-in` 循环，他们将无法直接访问到当前的索引，因为这种循环结构仅关注于元素本身。如果您需要同时访问索引和元素，您应该使用 `for-range` 循环。然而，如果索引不重要或者暂时不想处理， `for-in` 循环通常是一个更简洁和更直接的选择。

遍历字典

Yak语言中字典是一种关联数组，每个元素由一个键（key）和一个值（value）组成。遍历字典时，我们可以使用不同的方法来获取我们需要的信息。下面，我们将探讨如何使用不同的循环结构来遍历字典。

假设我们有一个字典 `b` ，其中包含如下键值对：

```
b = {"a": 1, "b": 2, "c": 3}
```

我们可以使用 `range` 关键字遍历字典，这样可以同时获取键和值：

```
for k, v = range b {
    printf("%s:%d, ", k, v)
}
```

在这个循环中， `k` 和 `v` 分别在每次迭代时被赋予字典中的键和对应的值。输出结果将是：

```
a:1, b:2, c:3,
```

与 `range` 类似， `in` 关键字也允许我们在遍历时获取键和值：

```
for k, v in b {
    printf("%s:%d, ", k, v)
}
println()
```

这种方式同样会输出：

```
a:1, b:2, c:3,
```

如果我们只对键感兴趣，我们可以省略值的部分：

```
for k in b {
    printf("%s:%d, ", k, b[k])
}
```

这个循环只会迭代键，但我们仍然可以通过字典的键来获取值。输出也是：

```
a:1, b:2, c:3,
```

我们也可以使用 `range` 关键字与省略值的方式来只获取键：

```
for k = range b {
    printf("%s:%d, ", k, b[k])
}
```

这种方式获取的结果与前面的方法相同：

```
a:1, b:2, c:3,
```

使用FOR循环操作通道

根据第三章的内容，我们知道Yak语言提供了一种特殊的数据类型——通道（channel）。通道可以被想象为一种先进先出（FIFO）的队列结构，它允许数据从一个方向写入，并从另一个方向被读取。在本教程中，我们将学习如何创建通道，并使用 `for-range` 和 `for-in` 语句来遍历通道中的数据。

创建通道

首先，让我们创建一个通道。使用 `make` 函数可以创建一个指定大小的通道，如下所示：

```
ch := make(chan var, 2) // 创建一个可以存储两个元素的通道
```

在这个例子中，我们创建了一个名为 `ch` 的通道，它可以存储两个 `var` 类型的元素。

向通道写入数据

写入通道的操作很简单，只需要使用 `<-` 运算符即可：

```
1 ch <- 1 // 向通道写入数据1
2 ch <- 2 // 向通道写入数据2
```

这里，我们向 `ch` 通道中写入了两个数据： `1` 和 `2` 。

关闭通道

在完成数据的写入后，我们通常需要关闭通道，以表明没有更多的数据将被发送到通道中。关闭通道的操作如下：

```
1 close(ch) // 关闭通道
```

关闭通道是一个好习惯，可以防止在通道上发送更多数据，这对于避免程序中的死锁是非常重要的。

遍历通道

现在我们来遍历通道中的数据。我们可以使用 `for-range` 语句来实现这一点：

```
1 for result = range ch { // 遍历通道内的数据
2     println("fetch chan var [ch] element: ", result)
3 }
```

在这个 `for-range` 循环中，变量 `result` 会依次被赋予通道 `ch` 中的每个元素的值。每次迭代将打印出当前从通道中取出的元素。

与 `for-range` 类似，我们也可以使用 `for-in` 语句来遍历通道：

```
1 for result in ch { // 遍历通道内的数据
2     println("fetch chan var [ch] element: ", result)
3 }
```

使用 `for-in` 语句的效果与 `for-range` 相同，它也会逐个访问通道中的元素。需要注意的是， `for-in` 与 `for-range` 语句遍历通道时，只有通道被显式使用 `close()` 关闭并且通道内已经没有元素时，循环才会结束。

运行结果

无论是使用 `for-range` 还是 `for-in` 语句，上述例子的运行结果将是：

```
fetch chan var [ch] element:  1
fetch chan var [ch] element:  2
```

这表明我们成功地从通道中取出了先前写入的两个元素。

通过使用 `for-range` 或 `for-in` 语句，我们可以轻松地遍历通道中的数据。这些概念将在第六章中进行更详细的探讨，但现在您已经有了一个关于通道如何工作的基本理解，并且知道了如何通过遍历来处理通道中的数据。

4.2.3 FOR-NUMBER：简化循环次数的语法糖

在编程实践中，我们经常遇到需要重复执行某段代码多次的需求。传统的方法是使用 `for` 循环，指定起始条件、结束条件以及迭代步进，如下所示：

```
for i := 0; i < n; i++ {
    // 执行代码
}
```

为了简化这种常见的循环结构，Yak语言引入了一种简洁的写法，即 `FOR-NUMBER` 语法糖。这种语法让我们能够直接指定循环次数，而不需要编写完整的循环控制语句。下面是 `FOR-NUMBER` 的基本语法：

```
for in n {
    // 循环体将执行n次
}
```

此外，如果你需要在循环体内部访问当前的索引，Yak语言允许你这样写：

```
for i in n {
    // 可以使用变量i，它从0开始，直到n-1
}
```

你也可以使用 `range` 关键字，这与 `in` 的用法类似：

```
for range n {
    // 循环体将执行n次
}
```

如果需要索引，可以将 `i` 和 `range` 一起使用（注意：range 前有一个 `=`）：

```
for i = range n {
    // 可以使用变量i，它从0开始，直到n-1
}
```

在所有这些形式中， `i` 是可选的。如果你不需要在循环体内部使用索引，可以省略它。

示例

假设你想要打印出"Hello, Yak!"这个字符串5次，使用 `FOR-NUMBER` 语法糖，你可以这样写：

```
for in 5 {
    println("Hello, Yak!")
}
```

如果你需要在每次打印时显示迭代的次数，可以包含索引：

```
for i in 5 {
    println("Iteration", i, ": Hello, Yak!")
}
```

这将输出：

```
Iteration 0: Hello, Yak!
Iteration 1: Hello, Yak!
Iteration 2: Hello, Yak!
Iteration 3: Hello, Yak!
Iteration 4: Hello, Yak!
```

`FOR-NUMBER` 语法糖是Yak语言中的一项便捷功能，它允许开发者以更直观、更简洁的方式编写有限次数的循环。这种语法结构不仅减少了代码的冗余，而且使得代码的意图更加清晰。无论是简单重复任务还是需要索引的迭代， `FOR-NUMBER` 都提供了一个优雅的解决方案。

4.2.4 使用 Break 和 Continue 控制循环流程

在编程中，我们通常需要更细粒度的控制循环的执行流程。Yak语言，与许多其他编程语言一样，提供了 `break` 和 `continue` 这两个控制语句，让我们可以在循环中进行更复杂的操作。 `break` 用于完全终止循环，而 `continue` 用于跳过当前迭代，直接进入下一次迭代。

Break 语句

使用 `break` 可以立即退出循环，不再执行剩余的迭代。这在你已经找到所需结果或者需要提前终止循环时非常有用。下面是一个使用 `break` 的例子：

```
for i = range 4 {
    println(i)
    if i == 2 {
        break // 当i等于2时，退出循环
    }
}
println("Loop ended with break.")

/*
OUTPUT:

0
1
2
Loop ended with break.
*/
```

在上述代码中，当变量 `i` 等于2时， `break` 语句会导致循环立即终止，因此只会打印出0、1和2。

Continue 语句

与 `break` 不同， `continue` 并不会退出整个循环，而是结束当前的迭代，并继续执行下一个迭代。这在你想要跳过某些特定条件的迭代时非常有用。下面是一个使用 `continue` 的例子：

```
for i in 4 {
    if i == 2 {
        continue // 当i等于2时，跳过当前迭代
    }
    println(i)
}
println("Loop ended with continue.")

/*
OUTPUT:

0
1
3
Loop ended with continue.
```

```
16 */
```

在这个例子中，当变量 `i` 等于2时， `continue` 语句会跳过当前迭代，因此不会打印2，只会打印0、1和3。

`break` 和 `continue` 是控制循环流程的强大工具。 `break` 用于提前退出循环，而 `continue` 用于忽略某些迭代条件。合理使用这两个控制语句可以让我们的循环逻辑更加灵活和强大。在Yak语言中，它们的使用与其他主流编程语言保持一致，这有助于降低学习成本，同时提高代码的可读性和可维护性。