# 第 3 章

# 结构化编程——条件结构与循环结构

结构化编程是面向过程编程的重要范式，结构化编程最为重要的贡献是提出了 goto 语句是有害的，开发者仅通过顺序结构、条件结构和循环结构就能够控制整个程序的流程。所谓顺序结构，是上文所介绍的程序的语句逐条执行，在源代码中从上到下依次执行语句。

本章介绍结构化编程中另外两种重要的结构：条件结构和循环结构。在介绍这两种结构之前，需要先介绍另外一种数据类型——布尔类型，以及逻辑运算、关系运算的基本用法。

本章的核心知识点如下：

（1）布尔类型和枚举类型。

（2）逻辑运算、关系运算和复合赋值运算。

（3）条件结构：if 表达式、match 表达式。

（4）循环结构：for in 表达式、while 表达式和 do…while 表达式。

（5）死循环和循环嵌套的用法。

## 3.1 逻辑运算与关系运算

除了算术运算以外，判断数据和程序的状态、进行逻辑推理等也是程序需要解决的重要问题之一。例如，判断当前系统内存是否充足、判断用户当前是否已登录、提交登录表单前判断用户是否全部完成了表单的填写、用户输入的短信验证码是否正确等都需要逻辑运算和关系运算参与。

逻辑运算和关系运算都无法离开布尔类型。布尔类型是仅包含真和假两种值的数据类型，本节先介绍布尔类型的概念和用法，然后介绍逻辑运算和关系运算的用法。

### 3.1.1 布尔类型

布尔类型用 Bool 关键字声明，是仓颉语言中的逻辑类型，其值仅包括 true（真）和 false（假）两种，因此，布尔类型的字面量仅包括 true 和 false。

例如，定义布尔类型变量 flag1 并初始化为 true，定义布尔类型变量 flag2 并初始化为 false，代码如下：

```
var flag1 : Bool = true
var flag2 : Bool = false
```

布尔类型和二进制码实际上有很强的密切联系。计算机的运算和存储是通过二进制码的形式完成的。二进制码是计算机数据存储和运算的最小单位，称为 1 位（bit），仅包括 0 和 1 这两种状态值。由于布尔类型仅包含真和假两种状态，因此仅用 1 位二进制码即可表示一个布尔类型字面量，即用 0 表示假，用 1 表示真。

实际上，仓颉语言中布尔类型是通过 1 位二进制码存储的。只不过，在仓颉语言中不能通过强制类型转化的方式将布尔类型的值转化为数值，例如 Int8(true)的语法是错误的。

**注意** 整型和浮点型变量的值实际上也是以 0 和 1 组成的二进制码的形式存储的。例如，Int32 变量是用 32 位二进制码存储一个整数数值。

通过 print 函数可以输出布尔类型的变量和字面量。

**【实例 3-1】** 输出布尔类型的变量和字面量，代码如下：

```
//code/chapter03/example3_1.cj
func main(){
    print("true is ${true}\n")   //输出布尔类型字面量 true
    print("false is ${false}\n") //输出布尔类型字面量 false
    let flag = false
    println("flag: ${flag}")     //输出布尔类型变量 flag
}
```

编译并运行程序，输出结果如下：

```
true is true
false is false
flag: false
```

逻辑运算和关系运算的结果都是布尔类型的值。

### 3.1.2 逻辑运算

逻辑运算也被称为布尔运算，是针对布尔类型数据的逻辑变换，包括逻辑非、逻辑与、逻辑或等，其操作符和用法如表 3-1 所示。

逻辑非操作符是单目操作符，其操作数应当位于操作符的右侧，用于逻辑取反。逻辑与和逻辑或操作符是双目操作符，用于分析左右两侧操作符之间的逻辑关系。

**注意** 仓颉语言不支持原生的逻辑异或运算，开发者可以使用(a||b)&&(!a||!b)组合实现逻辑异或（其中，a 和 b 为布尔类型的变量、字面量或表达式）。

**表 3-1　逻辑操作符及其用法**

| 操　作　符 | 描　　述 | 用　　法 |
| --- | --- | --- |
| ! | 逻辑非（表达式逻辑取反） | ! true（结果为 false）<br>! false（结果为 true） |
| && | 逻辑与（两侧表达式均为 true 时为 true，否则为 false） | true && false（结果为 false）<br>false && false（结果为 false）<br>true && true （结果为 true） |
| \|\| | 逻辑或（两侧表达式逻辑值均为 false 时为 false，否则为 true） | true \|\| false （结果为 true）<br>false \|\| false （结果为 false） |

接下来，通过实例演示这些操作符的用法。

**【实例 3-2】** 定义布尔类型变量 ate，用于表征是否吃饭；定义 slept 变量，用于表征是否已睡觉。通过逻辑非、逻辑与和逻辑或运算回答一些问题，代码如下：

```
//code/chapter03/example3_2.cj
func main(){
    var ate : Bool = true        //吃饭：true 代表吃完了
    var slept : Bool = false     //睡觉：false 代表没有睡觉
    print("吃完饭了吗? ${ate}\n")
    print("没吃完饭吗? ${!ate}\n")
    print("吃完饭且睡完觉了吗? ${ate && slept}\n")
    print("吃完饭或睡完觉了吗? ${ate || slept}\n")
}
```

ate 变量为 true，所以!ate 的逻辑值和 ate 相反，为 false。表达式 ate && slept 使用逻辑与操作符，表示吃完饭且睡完觉的逻辑值。表达式 ate || slept 使用逻辑或操作符，表示吃完饭或睡完觉的逻辑值。编译并运行上述代码，输出如下：

```
吃完饭了吗? true
没吃完饭吗? false
吃完饭且睡完觉了吗? false
吃完饭或睡完觉了吗? true
```

逻辑异或的用法与逻辑与、逻辑或类似，但是相对来讲使用频率较低，读者可自行尝试。

与算术运算一样，多个逻辑运算也可以形成复合表达式，并且其运算顺序也遵循相同的规则。逻辑操作符的优先级如下：逻辑非（!）＞逻辑与（&&）＞逻辑或（||）。如果有括号操作符，则括号操作符的优先级最高。

**【实例 3-3】** 计算几个逻辑运算复合表达式的值，代码如下：

```
//code/chapter03/example3_3.cj
func main(){
    var a = true     //变量 a 为 true
    var b = false    //变量 b 为 false
```

```
    var c = true                          //变量 c 为 true
    print("test1: ${a && b && c}\n")      //从左到右依次逐个展开运算
    print("test2: ${a && !b && c}\n")     //先计算!b，然后从左到右依次逐个展开运算
    print("test3: ${a || b || c}\n")      //从左到右依次逐个展开运算
    //先计算 b && c 得到结果 false，再计算 a || false
    print("test4: ${a || b && c}\n")
    //逻辑异或的实现
    //先计算!a 和!b，然后计算(a || b)和(!a || !b)，最后计算整体表达式
    print("test5: ${(a || b) && (!a || !b)}\n")
}
```

main 函数中最后一个语句实现了变量 a 和 b 的逻辑异或运算。编译并运行代码，结果如下：

```
test1: false
test2: true
test3: true
test4: true
test5: true
```

逻辑运算是计算机最为基础的运算类型之一，上面这些逻辑运算在计算机中都用相应的门电路实现。

### 3.1.3 关系运算

关系运算用于比较大小、比较相等或不相等的关系，包括等于（==）、不等于（!=）、大于（>）、小于（<）、大于或等于（>=）、小于或等于（<=）等关系操作符，如表 3-2 所示。

**表 3-2 关系操作符及其用法**

| 操作符 | 描　　述 | 用法（当 value 变量值为 1 时） |
|---|---|---|
| == | 等于 | value == 1（结果为 true） |
| != | 不等于 | value != 1（结果为 false） |
| > | 大于 | value > 1（结果为 false） |
| < | 小于 | value < 1（结果为 false） |
| >= | 大于或等于 | value >= 1（结果为 true） |
| <= | 小于或等于 | value <= 1（结果为 true） |

关系运算支持整型、浮点型的数值比较，以及字符、字符串的比较。对于布尔类型，仅支持等于和不等于的关系运算。关系运算的结果为布尔值（true 或者 false），因此关系运算表达式也可以作为逻辑运算的操作数。当关系运算和逻辑运算组合复合表达式时，其相关操作符的优先级如下：逻辑非（!）＞关系运算操作符＞逻辑与（&&）＞逻辑或（||）。

字符和字符串的关系运算详见第 5 章的相关内容。下面介绍整型、浮点型和布尔类型的

关系运算。

### 1. 整型的关系运算

整型的关系运算方法可参考表 3-2 中的用法一栏，这里不再赘述。

**【实例 3-4】** 计算整型关系运算的复合表达式，代码如下：

```
//code/chapter03/example3_4.cj
func main(){
    var a = 1
    var b = 3
    var flag : Bool          //将运算结果赋值给 flag 变量
    flag = a == 1            //关系运算:判断 a 是否等于 1
    print("a == 1: ${flag}\n")
    flag = a == 2            //关系运算:判断 a 是否等于 2
    print("a == 2: ${flag}\n")
    flag = a == 1 || b == 2 //关系运算和逻辑运算结合:判断 a 是否等于 1 或 b 是否等于 2
    print("a == 1 || b == 2: ${flag}\n")
    flag = a == 1 && b == 2 //关系运算和逻辑运算结合:判断是否 a 等于 1 且 b 等于 2
    print("a == 1 && b == 2: ${flag}\n")
}
```

由于变量 a 的值为 1，所以 a==1 的结果为 true，a==2 的结果为 false，并且由于 b 的值为 3，所以 b == 2 的结果为 false。对于表达式 a == 1 || b == 2，先计算关系表达式得到 true || false，其结果为 true。对于表达式 a == 1 && b == 2，先计算关系表达式得到 true && false，其结果为 false。编译并运行程序，上述代码的输出结果如下：

```
a == 1: true
a == 2: false
a == 1 || b == 2: true
a == 1 && b == 2: false
```

### 2. 浮点型的关系运算

浮点型数据的关系运算方法与整型类似。

**【实例 3-5】** 计算浮点型关系运算的复合表达式，代码如下：

```
//code/chapter03/example3_5.cj
func main(){
    let PI = 3.1415926 //圆周率 PI
    print("PI < 3.2: ${PI < 3.2}\n") //判断 PI 是否小于 3.2
    print("PI != 3: ${PI != 3.0}\n") //判断 PI 是否不等于 3.0
//判断 PI 的值是否处于 3 和 4 之间
    print("PI > 3 && PI < 4 : ${PI > 3.0 && PI < 4.0}\n")
}
```

对于表达式 PI > 3.0 && PI < 4.0 来讲，先计算关系表达式 PI> 3.0 和 PI< 4.0，其值均为

true，因此将复合表达式转换为 true && true，其结果为 true。编译并运行程序，上述代码的输出结果如下：

```
PI <3.2: true
PI != 3: true
```

### 3. 布尔类型的关系运算

布尔类型仅支持等于（==）和不等于（!=）两个操作符。例如，将 flag 变量定义为 true，然后计算表达式 flag == true、flag != true 和 flag == false，代码如下：

```
var flag : Bool = true
print("flag 为 true 吗？${flag == true}\n")
print("flag 不为 true 吗？${flag != true}\n")
print("flag 为 false 吗？${flag == false}\n")
```

上述代码的运行结果如下：

```
flag 为 true 吗？ true
flag 不为 true 吗？ false
flag 为 false 吗？ false
```

布尔型的关系运算可以直接通过逻辑运算实现，并且更加简洁。例如 flag == true、flag !=false 与 flag 的结果相同，flag != true、flag == false 和!flag 的结果相同，如图 3-1 所示。

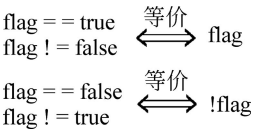


图 3-1　布尔类型关系运算和逻辑运算的等价关系（flag 为布尔型变量）

所以，上述代码也可以精简，精简后的代码如下：

```
var flag : Bool = true
print("flag 为 true 吗？${flag}\n")
print("flag 不为 true 吗？${!flag}\n")
print("flag 为 false 吗？${!flag}\n")
```

## 3.2　if 表达式

条件结构（也称为选择结构或分支结构）通过检查一系列的条件，当条件满足或者不满足时执行某些特定的程序。这里的条件是通过逻辑运算或关系运算完成的。在仓颉语言中，条件结构可以使用 if 表达式或 match 表达式完成。本节介绍 if 表达式的各种基本结构及其用法，3.3 节将介绍 match 表达式的基本用法。

**注意**　在绝大多数传统编程语言中，上述这些包含 if、else 关键字的结构通常不属于表

达式，所以通常称为 if 语句。在仓颉语言中，这些结构属于 if 表达式，所以是有值的，其功能更加强大。

根据判断条件的复杂程度，if 表达式可以使用 if 和 else 关键字，可以分为以下 4 种基本结构类型：

（1）if 结构，包含 1 个语句块。

（2）if-else 结构，包含 2 个语句块。

（3）包含 else if 结构的 if 结构，包含 2 个以上语句块。

（4）包含 else if 结构的 if-else 结构，包含 3 个以上语句块。

if 结构、if-else 结构都属于 if 表达式，if 表达式也称为 if 语句。下面分别介绍这几种结构类型的用法。

### 3.2.1　if 结构

if 结构是最简单的 if 表达式，通过 if 关键字定义，主要包括条件测试表达式和语句块（也称为代码块）两部分，其基本形式如下：

```
if (条件测试表达式) {语句块}
```

当条件测试表达式的结果为 true 时，执行语句块中的语句；反之，如果条件测试表达式的结果为 false，则不执行语句块中的语句。语句块通过花括号{}包裹，可以将多条语句进行组合，形成具有特定功能的整体。

**注意**　if 表达式中的条件测试表达式也可以是独立的布尔类型变量或字面量。

if 结构的执行流程图如图 3-2 所示。

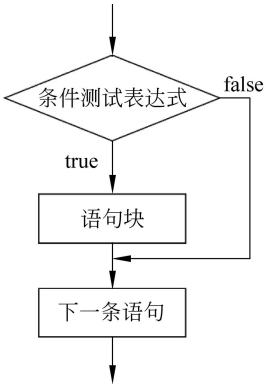


图 3-2　if 结构的执行流程图

**注意**　if 结构必须使用花括号包裹的语句块，不支持单独的语句，即“if (条件测试表达式) 执行语句”这样的形式是非法的。如果仅需要执行单独的语句，则要放到语句块中。

为了使程序结构更加清晰，通常使 if 结构以类似函数的方式占据源代码的多行，并采用 4 个空格（或 1 个 Tab 制表符）缩进语句块，代码如下：

```
if (条件测试表达式) {
    语句块
}
```

语句块中可以包含 1 条或者多条执行语句，代码如下：

```
if (条件测试表达式) {
    执行语句1
    执行语句2
```

```
    ……
}
```

**注意** 对于函数来讲，函数体花括号{}及其内部的代码本质上也是语句块。

接下来，通过一个简单的例子学习一下 if 结构的基本用法。

**【实例 3-6】** 通过 if 结构根据条件实现是否打印某个字符串，代码如下：

```
//code/chapter03/example3_6.cj
func main(){
    //定义 isEating 变量，并且初始化为 true
    var isEating : Bool = true
    //if 语句
    if (isEating) //括号内为条件测试表达式，当 isEating 为 true 时执行下方的语句块
    {
        print("我正在吃饭!\n") //当 isEating 为 true 时，打印“我正在吃饭!”文本
    }
    print("程序运行结束。\n")  //程序结束时，打印“程序运行结束。”文本
}
```

首先，定义了一个布尔类型的变量 isEating，并且初始化为 true，用于表征是否吃饭，然后紧跟着是一个 if 结构，其中的条件测试表达式为 isEating 本身。当 isEating 为 true 时，执行下方的打印“我正在吃饭!”文本的语句，否则则不执行任何操作。当程序结束时，打印“程序运行结束。”文本。编译并运行程序，输出结果如下：

```
我正在吃饭!
程序运行结束。
```

修改实例 3-6 代码的第 3 行，将 isEating 变量的初始化值修改为 false，代码如下：

```
var isEating : Bool = false
```

重新编译并运行程序，输出结果如下：

```
程序运行结束。
```

这时由于 isEating 表达式的值为 false，无法执行 if 结构后方的语句块，因此也无法打印“我正在吃饭!”的文本内容了。

### 3.2.2 if–else 结构

if 结构可以实现当条件测试表达式为 true 时需要执行的语句块，但没有定义当条件测试表达式为 false 时需要执行的语句块。通过 else 关键字可将 if 结构进行扩充，实现 if-else 结构。if-else 结构包含了 if 结构，并且可以通过 else 关键字加入另一个语句块，用于实现当条件测试表达式为 false 时需要执行的代码，其基本形式如下：

```
if (条件测试表达式) {
```

```
    语句块 1 //当条件测试表达式为 true 时执行
} else {
    语句块 2 //当条件测试表达式为 false 时执行
}
```

语句块 1 也称为 if 语句块，语句块 2 也称为 else 语句块。

**注意**　else 关键字不能单独使用，一定要和 if 结构配合使用。

与 if 结构类似，if-else 结构也属于 if 表达式，并且 if-else 结构“考虑事情”更加全面，能够根据条件测试表达式值的不同，执行不同的语句块。if-else 结构的执行流程图如图 3-3 所示。

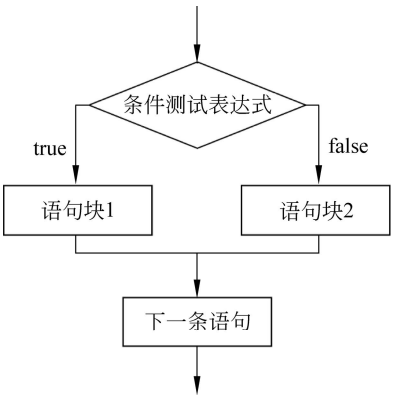


图 3-3　if-else 结构的执行流程图

下面对实例 3-6 的功能进行扩展。

**【实例 3-7】** 创建一个 isEating 变量，并且初始化为 true，然后实现当 isEating 为 true 时打印“我正在吃饭!”文本，当 isEating 为 false 时打印“我没有吃饭!”文本，代码如下：

```
//code/chapter03/example3_7.cj
func main(){
   var isEating : Bool = true //定义 isEating 变量，并且初始化为 true
    if (isEating)
    {
        print("我正在吃饭!\n") //当 isEating 为 true 时，打印“我正在吃饭！”文本
    } else {
        print("我没有吃饭!\n") //当 isEating 为 false 时，打印“我没有吃饭！”文本
    }
    print("程序运行结束。\n") //程序结束时，打印“程序运行结束。”文本
}
```

编译并运行程序，输出结果如下：

```
我正在吃饭!
```

```
程序运行结束。
```

修改上述程序的第 2 行，将 isEating 变量的初始化值改为 false，代码如下：

```
var isEating : Bool = false
```

重新编译并运行程序，输出结果如下：

```
我没有吃饭!
程序运行结束。
```

if-else 结构可以创建程序的分支，根据不同的状况选择运行不同的语句块。

### 3.2.3 if 表达式的嵌套和 else if 结构

if 语句和 if-else 结构仅通过 1 个条件测试表达式将程序最多分为两个分支，但在实际情况中，往往会更加复杂。例如，某项考试为百分制，分数大于或等于 90 分（含 90）为优秀，分数为 80～90（含 80）分为良好，分数为 60～80（含 60）分为合格，分数低于 60 分为不合格。现在需要设计一个程序，输入一个分数，并打印其成绩分类。这个程序存在 4 个分支，如图 3-4 所示，使用独立的 if-else 结构只能创建两个程序分支，无法完成这样的程序。

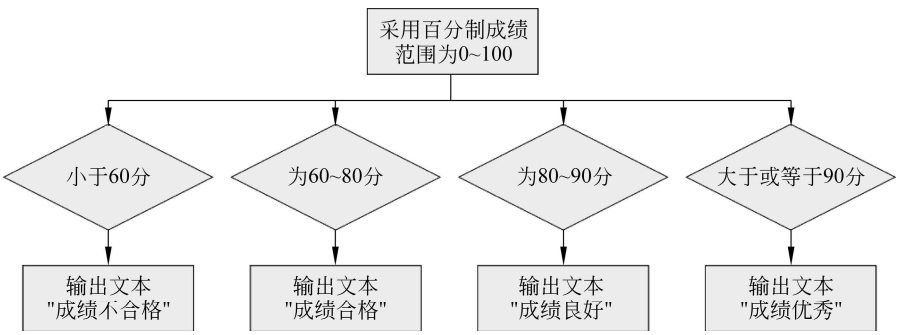


图 3-4　输出百分制成绩分类

本节介绍两种方法实现这一功能，分别是 if 表达式嵌套的“笨办法”和使用 else if 的“新办法”。

**注意**　除了本节所介绍的方法外，还可通过 4 个 if 结构实现这一功能，读者可以自行尝试实现。

#### 1. if 表达式的嵌套

if 表达式的嵌套是指在一个 if 表达式语句块（可以是 if 语句块，也可以是 else 语句块）中包含另外一个 if 表达式。被包含的 if 表达式称为内层 if 表达式，包含 if 表达式的 if 表达式称为外层 if 表达式。

**【实例 3-8】** 通过 if 表达式的嵌套实现百分制成绩分类的输出，代码如下：

```
//code/chapter03/example3_8.cj
```

```
func main(){
    var score : Int32 = 77 //定义 score 变量，用于存储分数，初始化为 77 分
    if (score >= 80){
        //当分数大于或等于 80 分时，执行该语句块
        if (score >= 90){
            print("成绩优秀\n")    //当分数大于或等于 90 分时
        } else {
            print("成绩良好\n")    //当分数小于 90 分且大于 80 分时
        }
    } else {
        //当分数小于 80 分时，执行该语句块
        if (score < 60){
            print("成绩不合格\n")  //当分数小于 60 分时
        } else {
            print("成绩合格\n")    //当分数大于或等于 60 分且小于 80 分时
        }
    }
}
```

这里将两个 if-else 结构分别嵌套到另一个 if-else 结构中的 if 语句块和 else 语句块中。在代码开头，通过 score 变量将成绩分数定义为 77 分。首先，判断 score>=80 表达式为 false，所以进入外层 if 表达式的 else 语句块中，然后，在该 else 语句块中进入内层 if 表达式，在这个 if-else 结构中判断 score<60 表达式为 false，因此进入相应的 else 语句块，最后打印“成绩合格”文本。编译并运行程序，输出结果如下：

```
成绩合格
```

读者可以尝试通过修改 score 变量改变分数，并且尝试打印不同的输出内容。

**注意**　if 表达式支持多级嵌套，即被 if 表达式嵌套的 if 表达式可以嵌套另外一个 if 表达式。例如，将 if 表达式 B 嵌套到 if 表达式 A 中，还可以将 if 表达式 C 嵌套到 if 表达式 B 中，并且可以以此类推再将其他的 if 表达式嵌套到 if 表达式 C 中。

if 表达式的嵌套虽然容易理解和设计，但是会增加程序的层级。层级变多的代码可能会导致难以阅读和扩展。

### 2. else if 结构

使用 else if 也可以实现类似 if 表达式嵌套的多条件结构，并且属于更加推荐的用法。else if 是将 else 和 if 关键字结合在一起，在原本的 if 结构或 if-else 结构中增加一个条件测试表达式和一个语句块，其基本形式如下：

```
if (条件测试表达式 1) {
    语句块 1 //当条件测试表达式 1 为 true 时执行
} else if(条件测试表达式 2) {
    语句块 2 //当条件测试表达式 2 为 true 时执行
```

```
} else if(条件测试表达式 3) {
    语句块 3 //当条件测试表达式 3 为 true 时执行
}
...
else if (条件测试表达式 n) {
    语句块 n //当条件测试表达式 n 为 true 时执行
} else {
    语句块 t //当上述条件测试表达式均为 false 时执行
}
```

黑体部分是新增加的代码部分，包含了多个 else if 结构。else if 结构是指 else if 及其后方相应条件测试表达式和语句块的部分。else if 结构中的语句块也称为 else if 语句块。else if 结构必须和 if 结构、if-else 结构配合使用，不可独立使用。

**注意** 上述语法结构中的 else 部分是可选的。

上述语法形式相当于在一个 if-else 结构（或 if 结构）中添加了一个或者多个 else if 结构。其中，每个 else if 结构都必须执行一次条件测试表达式的判断，如果成立，则进入相应的语句块中执行代码。如果 if 结构和 else if 结构中的条件测试表达式均不成立，则执行最后的 else 语句块。上述结构的执行流程图如图 3-5 所示。

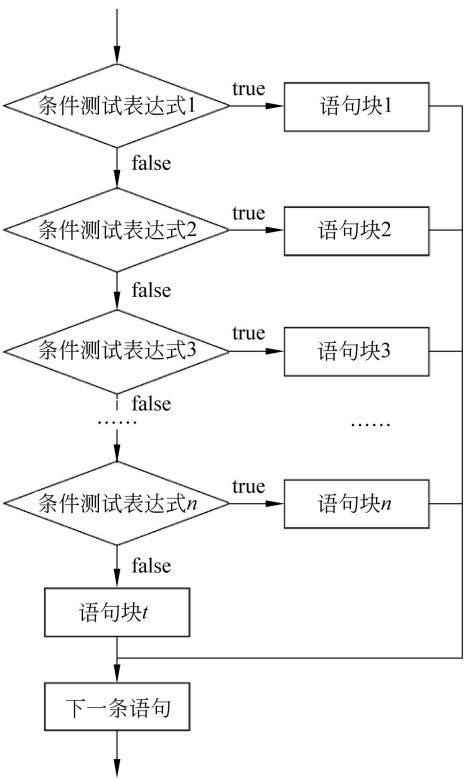


图 3-5 else if 语句的执行流程图

**注意**　一定要留心每个语句块前后花括号{}的匹配关系，不要重复或者丢掉，否则会导致编译错误。

**【实例 3-9】** 通过 if-else 结构和 else if 结构实现成绩的分级打印，代码如下：

```
//code/chapter03/example3_9.cj
func main(){
    var score : Int32 = 77  //score 变量存储了分数，初始化为 77 分
    if (score >= 90){       //条件测试表达式 1
        print("成绩优秀\n")   //语句块 1
    } else if (score >= 80) { //条件测试表达式 2
        print("成绩良好\n")   //语句块 2
    } else if (score >= 60) {   //条件测试表达式 3
        print("成绩合格\n")       //语句块 3
    } else {
        print("成绩不合格\n")     //语句块 4
    }
}
```

上述代码包括了 3 个条件测试表达式和 4 个语句块。其中，if 表达式的程序的执行过程如下：首先，测试表达式 score >= 90 的结果为 false，不执行语句块 1；然后，测试表达式 score>= 80 的结果为 false，不执行语句块 2；再后，测试表达式 score>= 60 的结果为 true，执行语句块 3，打印“成绩合格”文本。由于条件测试表达式 3 成立，所以后面的 else 语句块不会被执行。如果 score 变量的值小于 60，if 表达式则会一直执行到最后一个 else 语句块并打印“成绩不合格”文本。编译并运行上述程序，输出结果如下：

```
成绩合格
```

else 关键字及其语句块是一个不折不扣的“接盘侠”。只要 if 表达式中所有的条件测试表达式均不满足测试条件，就会进入 else 语句块，所以使用 else 关键字时一定要小心，仔细检查 else 语句块是否涵盖了程序分支的所有可能性，否则很容易使程序出现逻辑错误。

实际上，最后的 else 部分是非必需的。如果整个代码很长或者业务逻辑比较复杂，则将 else 关键字改成 else if 可能会使程序具有更好的可读性和健壮性。例如，修改实例 3-9 的代码，将 else 关键字替换为 else if，代码如下：

```
func main(){
    var score : Int32 = 77
    if (score >= 90){
        print("成绩优秀\n")
    } else if (score >= 80) {
        print("成绩良好\n")
    } else if (score >= 60) {
        print("成绩合格\n")
    } else if (score < 60) {
```

```
        print("成绩不合格\n")
    }
}
```

该程序代码也可以正常运行。读者可以尝试改变 score 变量的值从而使程序进入不同的语句块，并输出不同的结果。

以上已经介绍了所有的 if 表达式结构。可以发现，无论是何种结构的 if 表达式，最多只能运行其中的一个语句块。对于包含 else 关键字的 if 表达式，一定会运行其中的一个语句块。

### 3.2.4 if 表达式的值

与算术表达式等其他表达式一样，if 表达式也有值。由于 if 表达式的值依赖于语句块的值，所以本节先介绍语句块的值，然后介绍 if 表达式的值。

#### 1. 语句块的值

如果语句块的最后一个语句是表达式、变量或字面量，则语句块的类型是该表达式、变量或字面量的类型，语句块的值也是表达式、变量或字面量的值。如果语句块最后一个语句不是表达式，或者语句块为空，则语句块的类型为 Unit，其值为()。关于 Unit 类型和 Unit 类型的值详见 5.5.1 节的相关内容。

下面介绍几个简单的例子。

（1）语句块{ 2 }中仅包含一个语句，即字面量 2，所以该语句块的类型和值与字面量 2 相同，类型为 Int64，其值为 2。

（2）语句块{var a = 0.2; a + 1.0}包含两个语句，最后一个语句为 a + 1.0，所以该语句块的类型和值与表达式 a + 1.0 相同，类型为 Float64，其值为 1.2。

（3）空语句块{ }的类型为 Unit，其值为()。

（4）语句块{ let a = 2 }仅包含一个赋值表达式语句。由于该赋值表达式的类型为 Unit，其值为 2，所以该语句块的类型也是 Unit，其值为 2。

#### 2. if 表达式的值

如果 if 表达式仅包含一个语句块（if 结构），则 if 表达式的值是语句块的值。如果 if 表达式包含多个语句块，则其各个语句块的类型必须相同。如果类型不相同，则 if 表达式的类型是所有语句块的除了 Any 以外的最小公共父类型，其值为 if 表达式所执行语句块的值。

**注意** Any 类型是所有类型的父类型，详情可参见 7.3.2 节的相关内容。

表 3-3 列举了一些 if 表达式的类型和值。

**表 3-3 几个 if 表达式的类型和值的例子**

| 表达式 | 表达式的类型 | 表达式的值 |
|---|---|---|
| if (true) { 2 } | Int64 | 2 |
| if (true) {var a = 2.0; a + 0.1 } | Float64 | 2.1 |

续表

| 表达式 | 表达式的类型 | 表达式的值 |
| --- | --- | --- |
| if (true) {var a = 6; a ++ } | Unit | () |
| if(true) { 2 } else { 3 } | Int64 | 2 |
| if(true) { } else { 3 } | Unit | () |

如果 if 表达式的语句块没有除了 Any 以外的公共父类型，则会导致编译报错。例如，if (true) { 2 } else { 1.0 }及 if (true) { false } else { 2.1 }都是错误的表达式，会导致编译错误。

**【实例 3-10】** 通过 if 表达式判断某个变量是否小于 0，并打印最终的结果，代码如下：

```
//code/chapter03/example3_10.cj
func main() {
    let value = -3
    let r = if (value < 0) { true } else { false }
    print("value 值是负值? ${r}\n")
}
```

加粗的部分是 if 表达式，其类型为布尔型。该表达式的结果会赋值给变量 r，此时变量 r 也为布尔类型。编译并运行程序，运行结果如下：

```
value 值是负值? true
```

这个 if 表达式从效果上等同于 value < 0 表达式，所以没有实际意义。

**注意**　在仓颉语言中，可以通过 if 表达式的形式实现其他语言中三元表达式的效果。可见仓颉语言中 if 表达式的强大之处。

下面举一个更具有意义的例子。

**【实例 3-11】** 通过 if 表达式将成绩分数转换为成绩分级，如图 3-4 所示，代码如下：

```
//code/chapter03/example3_11.cj
func main(){
    var score : Int32 = 77
    let res = if (score >= 90){
        "成绩优秀\n"
    } else if (score >= 80) {
        "成绩良好\n"
    } else if (score >= 60) {
        "成绩合格\n"
    } else {
       "成绩不合格\n"
    }
    print(res)
}
```

上述 if 表达式根据 score 值的不同返回不同的字符串文本。将 if 表达式的值赋值给 res

变量，然后输出 res 的字符串文本。编译并运行程序，输出结果如下：

```
成绩合格
```

相对于实例 3-9，实例 3-11 充分利用 if 表达式具有值的特性，使程序更加易读。

if 表达式已经具备了实现程序分支的能力，但是 if 表达式存在一些缺点：

（1）在程序分支较多的应用场景下，if 表达式会使程序显得比较臃肿。

（2）无法通过 if 表达式遍历枚举类型。

此时，开发者可以使用另外一类条件结构 match 表达式来解决这些问题。

## 3.3 match 表达式与枚举类型

match 表达式是仓颉语言的另外一种条件结构，用于弥补 if 表达式的不足。特别是当程序分支较多的时候，match 表达式会使代码更加清晰。

**注意** match 表达式可以代替许多传统编程语言中的 switch 语句。

本节介绍 match 表达式和枚举类型的基本用法。

### 3.3.1 match 表达式

match 表达式需要用到 match 和 case 关键字，其中 match 关键字用于声明一个 match 表达式，而每个 case 关键字的出现意味着一个程序分支。

match 表达式有两类，分别为有匹配值的 match 表达式和没有匹配值的 match 表达式。

1. 有匹配值的 match 表达式和模式匹配

有匹配值的 match 表达式通过匹配一个表达式（或变量、字面量）的值，从而使程序进入不同的 case 分支，并执行相应的语句块，其基本形式如下：

```
match (待匹配的表达式) {
    case 模式 1 =>语句块 1
    case 模式 2 =>语句块 2
    ...
    case 模式 n =>语句块 n
}
```

match 关键字后的小括号()包含了待匹配的表达式，然后通过一个花括号{}包含了数个 case 分支。每个 case 分支都由 case 关键字、模式、双线箭头=>符号和语句块构成。其中模式是需要和待匹配的表达式的值进行比对的匹配值，这个比对过程是模式匹配。

这里的模式通常是待匹配表达式类型的字面量。

**注意** 每个 case 分支中的语句块不需要（也不能）使用花括号{}包裹。

执行 match 表达式，会先计算待匹配表达式的结果，然后依次（从上到下）和每个 case 关键字后的模式进行匹配。一旦表达式的值和模式匹配成功，就会执行双线箭头=>后的语句

块，如图 3-6 所示。

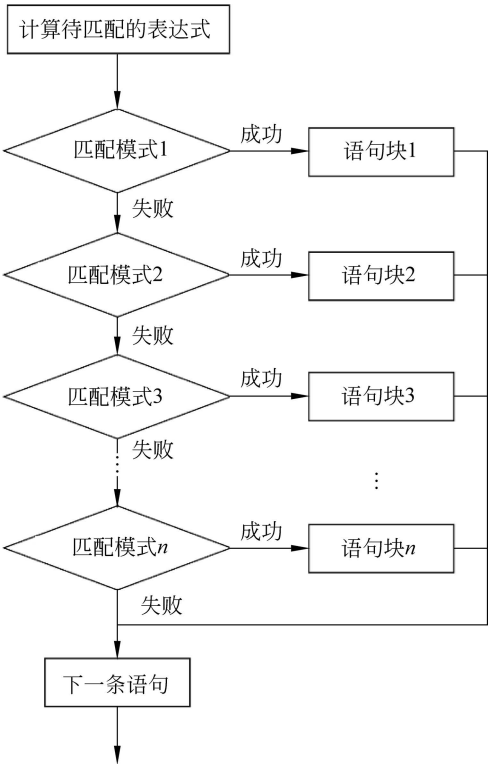


图 3-6　有匹配值的 match 表达式的执行流程图

match 表达式中的模式必须涵盖待匹配的表达式的所有可能性，否则会导致编译错误。如果 match 表达式中不能通过字面量一一列举所有的可能性，则最后一个 case 分支必须使用通配符模式或变量模式，用于匹配任意值，因此，match 表达式必然会执行某个（且只能是一个）语句块。

**1）通配符模式及 match 表达式的一般用法**

通配符模式为_，用于匹配任意值。使用通配符模式的分支必然是 match 表达式的最后一个分支。无论待匹配的表达式为何值，一旦进入通配符模式的分支，一定会匹配成功并执行相应的语句块，因此，使用通配符模式的分支用于“兜底”，当前面所有的分支模式匹配失败后就会执行通配符模式的分支语句块。

**【实例 3-12】** 使用 match 表达式对 value 变量进行模式匹配，并输出相应的文本信息：当 value 为 0 时输出“the value is 0”文本；当 value 为 1 时输出“the value is 1”文本；当 value 既不为 0 也不为 1 时输出“the value is neither 0 nor 1”文本，代码如下：

```
//code/chapter03/example3_12.cj
func main() {
```

```
    var value = 1 //将 value 变量的值定义为 1
    match (value) {
        case 0 =>  //当 value 为 0 时
            print("the value is 0\n")
        case 1 =>  //当 value 为 1 时
            print("the value is 1\n")
        case _ =>  //当 value 既不为 0 也不为 1 时
            print("the value is neither 0 nor 1\n")
    }
}
```

上述代码将语句块中的语句独立放置在代码的新行中，使程序更加清晰。

当程序运行到上述 match 表达式时，首先会计算待匹配表达式 value 的值，其值为 1，然后将这个值和第 1 个分支后的模式 0 进行匹配，由于 value 的值不为 0，所以匹配失败。之后，将这个值和第 2 个分支后的模式 1 进行匹配，此时 value 的值和 1 相等所以匹配成功，因此进入相应的语句块，输出“the value is 1”文本。编译并运行程序，输出结果如下：

```
the value is 1
```

双线箭头=>后方为语句块，所以可以执行多行语句组成的代码。例如，下面的代码也是合法的：

```
func main() {
   var value = 1
   match (value) {
      case 0 =>
         print("the value is 0\n")
       case 1 =>//该分支的语句块包含 3 行语句
         print("the value is 1\n")
         let v = value + 2
         print("the value plus 2 is ${v}\n")
      case _ =>
         print("the value is neither 0 nor 1\n")
   }
}
```

在上述代码 match 表达式的第 2 个分支中，定义并输出了变量 v 的值，并且变量 v 是通过 value + 2 表达式进行赋值的，即变量 value 和 2 的和。编译并运行程序，输出结果如下：

```
the value is 1
the value plus 2 is 3
```

通配符模式分支中的语句块并不能获得待匹配表达式的值。如果该语句块需要对待匹配表达式的值进行进一步处理，则需要使用变量模式。

**2）变量模式**

和通配符模式一样，变量模式同样可以匹配表达式的任意值，但是，变量模式需要通过标识符声明一个变量的名称，当程序进入分支语句块时，该变量的值是待匹配表达式的值。

**【实例 3-13】** 在实例 3-12 的基础上，实现当 value 的值不为 0 且不为 1 时，通过变量模式获取 value 的值，并在相应的语句块中输出 value 值的大小，代码如下：

```
//code/chapter03/example3_13.cj
func main() {
    var value = 88
    match (value) {
        case 0 =>
            print("the value is 0\n")
        case 1 =>
            print("the value is 1\n")
        case n =>
            print("the value is ${n}\n")
    }
}
```

在上述 match 表达式的最后一个分支中，模式 n 即为变量模式。这个变量名称可以由开发者自行定义，只要是合法的标识符即可。当程序执行到该分支时，n 可以匹配 value 表达式的任意值，并将 value 值赋值给变量 n。此时，即可在该分支的语句块中输出变量 n 的值了。

**注意**　变量模式中的变量为不可变变量（相当于 let 关键字声明的变量），所以不能在语句块中修改这个变量的值。

编译并运行程序，输出结果如下：

```
the value is 88
```

**3）同时匹配多个模式**

在 match 表达式的模式中，可以通过符号|将多个模式隔开，同时匹配多个模式。只要有一个模式匹配成功，就可以进入该分支相应的语句块中。

**【实例 3-14】** 通过 match 表达式判断 value 值是否为 10 以内的质数，代码如下：

```
//code/chapter03/example3_14.cj
func main() {
    let value = 5 //将变量 value 的值定义为 5
    match (value) {
        case 2 | 3 | 5 | 7 => print("value 是 10 以内的质数!\n")
        case _ => print("value 不是 10 以内的质数!\n")
    }
}
```

在上述 match 表达式中，一旦 value 值匹配到 2、3、5、7 中的任何一个值，即可进入该分支的语句块中。编译并运行程序，输出结果如下：

```
value 是 10 以内的质数!
```

上述有匹配值的 match 表达式用于比对变量和模式是否相同，从而进入不同的程序分支中，显然这种用法的局限性较强，然而，没有匹配值的 match 表达式就更加灵活了。

**4）模式守卫**

模式守卫（Pattern Guard）是对模式增加一个条件测试表达式。模式守卫处于模式的后方，其基本形式如下：

```
模式 if（条件测试表达式）
```

其中，加粗部分为模式守卫。如果模式守卫中条件测试表达式的结果为 false，则该模式无法匹配，反之模式可以正常匹配。

**【实例 3-15】** 通过天气状况和是否忙于工作的两种情况来判断是否可以出去游玩，代码如下：

```
//code/chapter03/example3_15.cj
func main() {
    var isGoodWeather = true //是否为好天气
    var isBusy = true //是否忙于工作
    match (isGoodWeather) {
        case true if (isBusy)=>
            print("忙于工作，不能出去玩\n")
        case true =>
            print("可以出去玩\n")
        case false =>
            print("天气不好，不能出去玩\n")
        case _ =>() //()为 Unit 类型的值
    }
}
```

在第 1 个分支中，当变量 isGoodWeather 和 isBusy 同时为 true 时才能匹配模式“true if (isBusy)”，即该模式相当于 isGoodWeather == true && isBusy == true。编译并运行程序，输出结果如下：

```
忙于工作，不能出去玩
```

实际上，上述代码永远不会进入最后的通配符分支。这里的通配符分支只是为了满足语法要求，并没有实际的意义。

**注意** 由于仓颉语言编译器难以在编译时判断 match 表达式分支是否覆盖了所有的情况，所以此时如果不使用通配符分支会导致编译错误。