第三章：Yak语言中的语句、变量和表达式

从本章节开始，笔者将带领大家正式开始Yak语言的学习。本章内容包含Yak语言中语句的类型、变量的定义与使用、基本数据类型的定义与使用，复合类型以及Yak语言中出现的运算符和表达式。

3.1 语句类型概览

要了解一个Yak语言基础程序，需要从基本结构开始了解：

1. 文件格式：一个标准的Yak语言程序或者脚本文件，扩展名应该为“.yak”。
2. Yak文件中的代码：Yak代码由一个或者多个语句构成。其基本的语句之间可以通过“回车”来进行分隔，同时也支持“;”符号分隔。
3. 语句类型：目前Yak语言的语句主要有十三种类型，每种类型都具有不同的功能：

语句类型	目的	案例描述
注释语句	提供按行的注释或整块儿注释	# 号注释 # Comment 普通注释 // Comment 多行注释 /* Hello YakComment */
变量声明语句	自动或强制创建一个新的变量，这个变量会对应 YakVM 编译中的一个新符号	Golang 风格 var abc = 123 强制创建变量 abc := 123 自动创建 abc = 123
表达式语句	执行一个表达式，例如函数调用，数值运算，字符串运算等	1+1 "abc".HasPrefix("ab")
赋值表达式运算	赋值+表达式的简易写法	a += 1
代码块	主动创建一个新的定义域，执行若干行语句	a=1; {a++; a += 12}
IF 控制流	支持 if / elif / else if / else 风格的 IF 语句编写	if a>1 {println("Hello V1ll4n")}
Switch 控制流	支持 Case 多值短路特性的 Switch 语句，与 break / fallthrough 配套	switch a {case 1,2,3: println("Hello")}
FOR IN 循环语句	Python 风格的 For In 语句技术实现	for a in [1,2,3] {println(a)}
FOR RANGE 循环语句		for _, a = range [1,2,3] {println(a)}

	Golang 风格的 For Range 语句技术实现	
FOR 循环控制	经典的 C 风格三语句 FOR 循环	for i = 1; i < 10; i ++ {println(i)} for {println("无限循环")}
Defer 延迟执行语句	Golang 风格的在函数或执行体结尾执行的语句块儿或者函数调用	defer func{ if recover() != nil { println("Catched") }}
Go 并发语句	Golang 风格的并发语句	go server.Start()
ASSERT 断言语句	用以快速检查程序中是否有失败的问题，定义为 assert <expr>, expr1	assert 1+1 == 2, "计算失败"

笔者在后续的章节中会为大家详细介绍这些语句的使用，因此读者并不需要在这一小节完全理解各个语句的实例，有一个大概的印象即可。

3.2 变量与基本数据类型

变量和基本数据类型是编程语言的核心概念，它们对于一门编程语言的功能和表达能力至关重要。

3.2.1 变量的定义和使用

在编程中，变量是一个相当重要的概念，是存储和引用数据的标识符。它允许编程者在程序中存储值，并根据需要对这些值进行操作和更改，同时在程序中进一步跟踪和操作数据。在Yak语言中，变量的定义和使用非常简单和直观。本章将详细介绍如何在Yak语言中定义、声明和使用变量。

变量的定义

在Yak语言中，要定义一个变量，编程者可以使用 `var` 作为关键词， `var` 后使用空格作为分隔符，再写入变量名即可完成变量的生命。变量名可以是数字，字母和下划线组合，但是必须以字母或下划线作为开头。以下是一些具体示例：

```
1 var a             // 声明变量a
2 var b, c          // 声明变量b和c
3 var d, e = 1, 2   // 声明变量d和e，并分别赋值为1和2
```

根据上述示例：代码中声明了四个变量：a、b、c、d和e；用户可以选择只声明变量而不赋初值，也可以在声明时直接赋初值；根据第三行的案例，用户也可以使用一个关键字 `var` 后跟两个变量（使用“,”）作为分隔，可以同时声明两个变量，并同时为其赋值。

变量的赋值

在声明变量并赋值的过程中，用户使用到了本书第一个操作符“赋值操作符”，写作一个“=”符号。这个符号的意义就是把右边的“值”传递给左边的“标识符”中。

> 注意：这个“=”符号并不是数学中的“等于”，在计算机科学中，一般使用“==”作为“等于”使用。

在没有 `var` 修饰的时候，赋值符“=”会为左边的“标识符”自动创建一个变量。这是赋值符号一个非常好用的特性。因此以下示例在Yak语言中是合理的：

```
a = 10           // 将变量a赋值为10
b, c = 20, 30    // 将变量b赋值为20，变量c赋值为30
```

您可以单独给每个变量赋值，也可以同时给多个变量赋值。赋值操作符将右侧的值分配给左侧的变量。

除了“=”作为赋值符号的情况，在Yak语言中，`:=` 符号组合也可以作为“赋值”功能使用，因此上述的案例可改写成新的形式：

```
a := 10           // 将变量a赋值为10
b, c := 20, 30    // 将变量b赋值为20，变量c赋值为30
```

> 注意：“:=”的赋值和“=”自动赋值在某些情况下并不完全等价，我们在“定义域章节”中将会为大家详细介绍他们的区别。

变量的使用

变量的使用非常简单，只需在需要的地方使用变量名即可。例如：

```
a = 42           // 给变量a赋值为42
result = a + 2  // 使用变量a进行计算，并将结果存储在result变量中
```

在上述示例中，笔者使用变量 `a` 进行了计算： `+ 2` ，并将结果存储在了 `result` 变量中。

综合讲到的完整的创建变量并赋值、使用变量的行为，我们可以再用一个案例来向大家展示变量的完整用法：

```
var x, y = 5, 10  // 定义变量x和y，并分别赋值为5和10
sum = x + y       // 使用x和y进行计算，并将结果存储在sum变量中
dump(sum)         // 打印sum的值
```

运行上述代码将会在屏幕中输出

```
1 (int) 15
```

这个示例演示了如何在Yak语言中定义、赋值和使用变量，以及如何进行基本的数学计算，希望这能帮助读者更好地理解Yak语言中变量的使用方式。

3.2.2 基础数据类型

计算机在其最基础的层面上处理的数据都是由比特构成的。但为了更高效和直观地表示和处理数据，高级编程语言提供了一系列数据类型。这些数据类型可以理解为对底层比特数据的高级抽象，可以用整数、字符串等直观的方式表示数据，而不仅仅是一堆比特。

Yak语言提供了一系列内置的数据类型，这些数据类型意味着数据的灵活性和多样性，使得编程既能利用硬件的特性，又能便捷地表达多样的数据结构。在Yak语言中，数据类型分为两类：基础数据类型和复合数据类型。

Yak语言的基本数据类型如下：

- `int`：表示可以带正负号的整数数据类型（在Yak语言中占用的大小为64位）；
- `string`：用于表示一系列字符数据的，例如：`"Hello World"`就是一个字符串；
- `float`：用于表示浮点数；
- `byte`：等同于“无符号8位整数”，通常用来表示一个“字节”；
- `nil`与`undefined`一般用于表示一个未定义的变量或者空值；
- `bool`：表示“布尔值”，其值只有两种情况，`true`或`false`；

为了方便用户更直观地理解Yak语言中的基本数据类型，笔者创建了一个表格来对比编程语言中常见的基本数据类型：

对比类型	Yak	Python	Golang
字符串	string	string	string
二进制字符串	[]byte	b-string	[]byte
整数	int	int	int8, int16, int32, int64
			uint8, uint16, uint32, uint64
浮点	float	float	float32, float64 (double)
空值	undefined/nil	不支持	nil

布尔值	bool	bool	bool

通过以上对比，您可以更轻松地理解和掌握 Yak 的数据类型，并与其他语言进行比较。Yak 通过其丰富的数据类型，为开发者提供了便捷的方式来表达和处理各种数据。无论你需要表示一个简单的数字，还是一个复杂的数据结构，Yak 都能为你提供相应的工具和支持。在下面的章节中，我们将对数据类型进行详细的讲解。

整数与浮点数

Yak语言为开发者提供了简洁而强大的数字类型：整数 (int) 和浮点数 (float)。在这章里，我们将详细地探讨这些数字类型以及如何在 Yaklang 中使用它们。

> 为了让开发者专注于“表达逻辑”，Yak语言在设计中有意避开了“整数和浮点数”的“位数”的概念，这种屏蔽的层实现的设计可以有效避免新手用户被复杂的计算机底层原理干扰。

整数声明

在Yak语言中，声明一个整数十分简单：

```
var a = 10
```

实际的编程中，除了这种基础声明之外，用户往往会遇到其他的需求，例如声明一个“二进制”、“八进制”或“十六进制”的整数；在Yak语言中，用户可以直接声明一个非十进制的整数，可以参考如下案例：

```
// 二进制声明
a = 0b10
println(a) // 输出: 2

// 八进制声明
b = 0100
println(b) // 输出: 64

// 普通整数声明（十进制）
c = 100
println(c) // 输出: 100

// 十六进制声明
d = 0x10
println(d) // 输出: 16
```

在声明一个非十进制整数的时候，用户只需要记住几个前缀即可：

- `0b` 意味着声明二进制整数
- `0x` 意味着声明一个十六进制整数
- 单独一个 `0` 意味着声明一个八进制整数

浮点数声明

与整数相似，浮点数的声明也非常直观和简单。

```
a = 1.5
println(a) // 输出: 1.5

b = a / 0.5
println(b) // 输出: 3.0
```

数学运算

Yak语言提供了一整套基础的数学运算，用户可以以此对数字进行加、减、乘、除和取余等操作：

```
println(2 + 2)    // 输出: 4
println(50 - 5*6) // 输出: 20
println(8 / 5)    // 输出: 1
println(17 % 3)   // 输出: 2
```

整数与浮点的互操作

在Yak语言中，当整数和浮点数一起运算时，整数会被先转换为浮点数，在进行运算，这就意味着运算的结果将是一个浮点数：

```
 a = 5 / 2.0
println(a)      // 输出: 2.5
printf("%T", a) // 输出: float64
```

这种设计选择的好处是保证了数值计算的准确性和一致性，无论操作数是整数还是浮点数。

布尔值类型

在Yak语言中，布尔值只有两种可能的常量：`true` 和 `false`。这些值通常用于表示逻辑条件的真或假。以下是一个最基本的使用：

```
a = true
```

```
b = false
if a && b {
    println("won't go here")
} else if a || b {
    println("true || false == true")
}
```

虽然至此读者并没有正式学习 `if` 语句，但是上述代码案例并不像影响读者理解 `true` 和 `false` 的含义。需要用户注意的是：与某些语言不同，Yak语言中的布尔值并不能直接与数值进行算术运算，因此 `true + 1` 在Yak语言中一般被视为“非法”。

空值：nil 与 undefined

Yak语言中引入“空值”的概念，一般来说，用户在遇到 `nil` 和 `undefined` 的时候，它们之间并没有区别，被视为等价即可，用户可以使用两个词来表示同一个概念：“这个变量没有值”。以下是一个典型案例：

```
a = nil
println(a == undefined) // 输出: true
println(b == nil)       // 输出: true
```

如上例所示，尝试访问一个未声明的变量将返回 `nil` （或 `undefined` ），这为开发者提供了一个便捷的方法来检查一个变量是否被赋值。

字符声明

Yak语言可使用单引号来声明一个字符：使用单引号声明字符，双引号声明字符串，这样可以直观地区分字符和字符串，提高代码的可读性：

```
c = 'c'
println(c)   // 输出: 99 (ASCII 值)
```

> 注意：本质上单个字符的底层类型是 `uint8` 。

字符串

在编程语言中，字符串的处理是核心部分。Yak语言中的字符串处理吸收了众多语言的最佳时间，同时加入了一些独一无二的特性，读者可以在本节中深入理解Yak语言中的字符串处理的强大能力：

经典字符串声明

与大多数编程语言的行为一致，Yak可以使用双引号来声明字符串，这种声明方式简单直观，初学者可以轻易上手：

```
println("Hello World")
```

当涉及到换行、制表符或其他特殊字符时，可以使用反斜杠 `\` 进行转义。例如 `\n` 表示换行，而 `\t` 表示制表符。此外，也支持直接输入字符的ASCII码，提供了另一种方式来插入特殊字符：

```
println("Hello \nWorld")

/*
Output:

Hello
World
*/
```

文本块声明

当处理多行字符串时，经典的转义方式可能会显得冗长。为了解决这一问题，Yak语言引用反引号“`` ` ``”，作为文本块的界定符。读者可以观察如下案例：

```
abc = `Hello World
Hello Yak`
println(abc)

/*
Output:

Hello World
Hello Yak
*/
```

在这个案例中，用户没有输入“\n”的转义符号也可以成功换行。不仅可以轻松处理多行字符串，还省去了每行的转义工作，大大增强了代码的可读性。

字节序列

除了传统的字符串处理，Yak语言还十分注重字节数据的处理。在声明一个字符串之前使用 `b` 前缀修饰，可以创建一个字节序列：

```
name = b"Hello World\r\nHello Yak"
dump(name)

/*
OUTPUT:

([]uint8) (len=22 cap=24) {
 00000000  48 65 6c 6c 6f 20 57 6f  72 6c 64 0d 0a 48 65 6c  |Hello World..Hel|
 00000010  6c 6f 20 59 61 6b                                 |lo Yak|
}
*/
```

这种声明方式本质上是把字符串当做一个字符数组来对待，读者从 `dump` 的输出结果中就可以看出，这个字符串的原始字符编码也会被展示出来。这种声明方式非常适用于处理网络数据包，文件I/O等场景中的“原始数据”。

字符串格式化

Yak语言使用 `%` 进行基本的字符串格式化。 `%` 是一种传统的格式化字符串的方法，称为字符串插值（string interpolation）。在Yak语言之外的许多编程语言中都有使用，例如 C、C++、Python 等。 `%` 格式化语法允许你在字符串中插入变量的值，从而创建动态字符串。这使得字符串的格式化变得既直观又灵活，同时，还支持数组和其他数据结构的直接输入。以下是使用 `%` 进行格式化的基本语法：

1. 在字符串中，使用 `%` 符号作为占位符，后跟一个或多个格式说明符，例如 `%d` （整数）、 `%f` （浮点数）或 `%s` （字符串）。
2. 在字符串之后，使用 `%` 符号和括号（可选，用于多个变量）包含要插入的变量。

以下是一些使用 `%` 进行格式化的示例：

```
printf("Hello I am %d years old\n", 18)
println("Hello %v + %05d" % ["World", 4])

/*
OUTPUT:

Hello I am 18 years old
Hello World + 00004
*/
```

根据案例，Yak语言触发代码格式化的写法主要有两种：

1. 使用传统的 `printf` 函数进行触发，第一个参数为需要格式化的字符串模版，其余参数为可变参数，是格式化字符串的“材料”；

2. 使用 `%` 格式化操作符来操作： `%` 左边为需要格式化的模版，右边为一个格式化字符串的“材料”，例如 `"Hello %v" % "World"`；如果有多个需要格式化的点，那么需要使用 `[]` 来包裹，并用逗号分隔元素，例如： `"My name is %v, I am %d years old" % ["John", 18]` 。

读者可能注意到了，Yak语言在字符串模版中可以使用 `%v` 之类的组合来标记需要字符串格式化的位置和格式，以下是在编程中常用的一些案例，用户可随时查阅并动手实践：

项	解释	代码示例
%v	根据变量的类型自动选择格式。	`printf("Default format: %v\n", p)`
%T	输出变量的类型。	`printf("Type of variable: %T\n", p)`
%d	十进制整数。	`printf("Decimal integer: %d\n", 42)`
%b	二进制整数。	`printf("Binary integer: %b\n", 42)`
%o	八进制整数。	`printf("Octal integer: %o\n", 42)`
%x	十六进制整数，使用小写字母。	`printf("Hexadecimal integer (lowercase): %x\n", 42)`
%X	十六进制整数，使用大写字母。	`printf("Hexadecimal integer (uppercase): %X\n", 42)`
%f	浮点数，不带指数部分。	`printf("Floating-point number: %f\n", 3.141592)`
%c	ASCII码对应的字符。	`printf("Character: %c\n", 65)`
%q	带引号的字符或字符串。	`printf("Quoted character: %q\n", 65)`
%s	字符串。	`printf("String: %s\n", "Hello, world!")`
%p	输出十六进制表示的内存地址或引用。	`printf("Pointer: %p\n", &p)`

字符串模版字面量： `f-string`

Yak 语言中的 f-string（格式化字符串字面量）是一种方便的字符串插值方法，允许在字符串中直接使用表达式。这种方法可以让你轻松地将变量和表达式的值嵌入到字符串中，而无需使用复杂的字符串拼接或格式化函数。以下是一个简化的解释和示例：

1. 定义变量：
   在示例中，我们定义了两个变量， `a` 和 `name` 。 `a` 的值为 "World"，而 `name` 的值为 "Yak"。

```
a = "World"
name = "Yak"
```

2. 使用 `f-string`：
   要使用 `f-string`，需要在字符串的前面加上一个小写的 `f`，然后在字符串内部用 `${}` 包裹需要插入的表达式。在这个例子中，我们将在字符串中插入变量 `a` 和 `name` 的值。

```
println(f`Hello ${a}, Hello ${name}`)
```

3. 输出结果：
   这段代码将输出 "Hello World, Hello Yak"。这是因为 f-string 会将 `${a}` 替换为变量 `a` 的值，将 `${name}` 替换为变量 `name` 的值。

```
OUTPUT:

Hello World, Hello Yak
```

Yak语言中的 `f-string` 提供了一种简单直观的字符串插值方法，使得在字符串中嵌入变量和表达式的值变得非常简单。只需在字符串前加上一个小写的 `f`，并用 `${}` 包裹需要插入的表达式即可。

Fuzztag快速执行：x-string

> 注意：本小块内容在 `fuzztag` 的专门章节中会详细介绍，此处只做简略叙述。

为了更好地支持模糊测试，Yak 语言引入了 x-string（fuzztag扩展语法）。这种独特的字符串处理方式能够快速生成一系列基于模板的字符串，大大加速了模糊测试的流程：

在这个例子中，我们使用 x-string 创建一个模板，该模板将生成一个包含 1 到 10 之间的整数的字符串：

```
a = x"Fuzztag int(1-10): {{int(1-10)}}"
dump(a)
```

Yak 语言的字符串处理功能既丰富又灵活，同时具有高效性。无论是进行基本的字符串操作，还是处理复杂的二进制数据和模糊测试，Yak 语言都能轻松应对，成为该语言的一大亮点。

字符串运算

与许多编程语言相似，Yak 语言也采用加号 `+` 来进行字符串的连接。以下是一个简单的示例：

```
a = "Hello, "
b = "Yak"
println(a + b)  // 输出: Hello, Yak
```

受 Python 语言的启发，Yak 语言引入了星号 `*` 操作，允许将字符串重复 N 次。以下是一个示例：

```
a = "powerful "
println(a * 5 + "yak")  // 输出: powerful powerful powerful powerful powerful yak
```

Yak 语言使用索引和切片操作来创建字符串“切片”。通过方括号 `[]` 和下标，你可以轻松地获取子字符串或子元素。以下是一些示例：

```
a = "Hello, Yak"
println(a[0])      // 输出: H
println(a[1:5])    // 输出: ello
println(a[3:0:-1]) // 输出: lle
```

现在，扩展这个示例，包括 `a[1:]` 和 `a[:3]` 这样的用法，以详细介绍索引操作在 Yak 语言中的应用。

- 省略结束索引，表示从开始索引一直到字符串末尾：

```
a = "Hello, Yak"
println(a[1:]) // 输出: ello, Yak
```

- 省略开始索引，表示从字符串开头到结束索引（不包括结束索引）：

```
a = "Hello, Yak"
println(a[:3]) // 输出: Hel
```

- 使用负数索引，表示从字符串末尾开始计算：

```
a = "Hello, Yak"
println(a[-3:]) // 输出: Yak
```

在这些示例中，我们展示了如何在 Yak 语言中连接字符串、重复字符串以及使用索引和切片来截取子字符串。这些功能使得处理字符串变得简单且直观，为学习和使用 Yak 语言的用户提供了便利。

字符串内置方法

为了使字符串处理更加高效，Yak语言引入了一系列内置方法。这些方法类似于Python，但进行了必要的优化和扩展，使其更符合Yak语言的设计哲学。

在这里，我们将介绍一些关于字符串类型的常用内置方法及示例。在这些示例中，我们将使用 `assert` 语句来确保示例代码的正确性。 `assert` 语句用于测试表达式的值，如果表达式为真，则程序继续执行；如果为假，则程序抛出异常并终止。这是一种简便的检查代码正确性的方法。

- 反转字符串：将字符串进行反序。

```
1 assert "abcdefg".Reverse() == "gfedcba"
```

- 是否包含：判断字符串是否包含子字符串。

```
1 assert "abcabc".Contains("abc") == true
2 assert "abcabc".Contains("qwe") == false
```

- 替代：替代字符串中的子字符串。

```
1 assert "abcabc".ReplaceN("abc", "123", 1) == "123abc"
2 assert "abcabc".Replace("abc", "123") == "123123"
```

- 分割：将字符串根据子串进行分割，得到数组。

```
1 assert "abc1abc".Split("1") == ["abc", "abc"]
```

- 连接：使用特定的字符串连接数组。

```
1 assert "1".Join(["abc", "abc"]) == "abc1abc"
```

- 移除前后特定字符：

```
assert "pabcp".Trim("p") == "abc"
```

- 转换为大写：

```
assert "hello".Upper() == "HELLO"
```

- 转换为小写：

```
assert "HELLO".Lower() == "hello"
```

- 计算子字符串出现的次数：

```
assert "abcabc".Count("abc") == 2
```

- 查找子字符串首次出现的位置：

```
assert "abcabc".Find("abc") == 0
assert "abcabc".Find("qwe") == -1
```

通过这些示例，读者应该大致了解Yak语言中字符串处理的大部分常用功能。这些内置方法简化了字符串操作，使其更加直观和易于理解。

包含上述描述的例子，笔者把Yak语言的字符串处理方法整理成表格，以便读者随时查阅：

方法名称	代码案例	简要描述
First	`assert "hello".First() == 'h'`	获取字符串第一个字符
Reverse	`assert "hello".Reverse() == "olleh"`	倒序字符串
Shuffle	`newStr = "hello".Shuffle()`	随机打乱字符串
Fuzz	`results = "hello".Fuzz({"params": "value"})`	对字符串进行模糊处理
Contains	`assert "hello".Contains("ell") == true`	判断字符串是否包含子串
IContains	`assert "Hello".IContains("ell") == true`	判断字符串是否包含子串（忽略大小写）

ReplaceN	`assert "hello".ReplaceN("l", "x", 1) == "hexlo"`	替换字符串中的子串（指定替换次数）
ReplaceAll	`assert "hello".ReplaceAll("l", "x") == "hexxo"`	替换字符串中所有的子串
Split	`assert "hello world".Split(" ") == ["hello", "world"]`	分割字符串
Join	`assert " ".Join(["hello", "world"]) == "hello world"`	连接字符串
Trim	`assert " hello ".Trim(" ") == "hello"`	去除字符串两端的cutset
TrimLeft	`assert " hello ".TrimLeft(" ") == "hello "`	去除字符串左端的cutset
TrimRight	`assert " hello ".TrimRight(" ") == " hello"`	去除字符串右端的cutset
HasPrefix	`assert "hello".HasPrefix("he") == true`	判断字符串是否以prefix开头
RemovePrefix	`assert "hello".RemovePrefix("he") == "llo"`	移除前缀
HasSuffix	`assert "hello".HasSuffix("lo") == true`	判断字符串是否以suffix结尾
RemoveSuffix	`assert "hello".RemoveSuffix("lo") == "hel"`	移除后缀
Zfill	`assert "42".Zfill(5) == "00042"`	字符串左侧填充0
Rzfill	`assert "42".Rzfill(5) == "42000"`	字符串右侧填充0
Ljust	`assert "hello".Ljust(7) == "hello  "`	字符串左侧填充空格
Rjust	`assert "hello".Rjust(7) == "  hello"`	字符串右侧填充空格
Count	`assert "hello".Count("l") == 2`	统计字符串中substr出现的次数
Find	`assert "hello".Find("l") == 2`	查找字符串中substr第一次出现的位置
Rfind	`assert "hello".Rfind("l") == 3`	查找字符串中substr最后一次出现的位置
Lower	`assert "HELLO".Lower() == "hello"`	将字符串转换为小写
Upper	`assert "hello".Upper() == "HELLO"`	将字符串转换为大写
Title	`assert "hello world".Title() == "Hello World"`	将字符串转换为Title格式

IsLower	`assert "hello".IsLower() == true`	判断字符串是否为小写
IsUpper	`assert "HELLO".IsUpper() == true`	判断字符串是否为大写
IsTitle	`assert "Hello World".IsTitle() == true`	判断字符串是否为Title格式
IsAlpha	`assert "hello".IsAlpha() == true`	判断字符串是否为字母
IsDigit	`assert "123".IsDigit() == true`	判断字符串是否为数字
IsAlnum	`assert "hello123".IsAlnum() == true`	判断字符串是否为字母或数字
IsPrintable	`assert "hello".IsPrintable() == true`	判断字符串是否为可打印字符

3.3 复合数据类型

Yak语言除了基本数据类型之外，还支持一些符合类型，它们极大地丰富了Yak语言的表现力：

- `list`：列表（也可以叫数组，切片），与 Python 的 list 类似；
- `map`：一个键-值对的集合，与 Python 的 dict 类型相似；
- `channel`：用于在协程之间通信的数据通道；
- `var`：用于表示任意类型，与 Golang 的 interface{} 类型相似；

同样为了方便用户直观地理解Yak语言中的复合数据类型（高级类型），笔者可以把它们放在一起与Python和Golang进行对比：

对比类型	Yak	Python	Golang
键值组 (map/dict)	map	dict	map
数组/切片/列表	list	list	slice/array
结构体/类/接口	不支持	class	struct/interface 体系
数据通道	channel	不支持	channel
任意类型	var	object	any(interface{})

在后续的内容会详细为大家介绍这几种非常重要的复合类型，它们是Yak语言强大的灵活性和表现力的关键。

3.3.1 列表类型：list

在 Yak 语言中，List 是一种动态数组，它可以存储和管理相同类型的元素。Yak 语言支持字面量声明和 `make` 语法来创建 List。接下来，笔者将详细介绍如何声明 List 以及如何对 List 进行操作。

创建列表

字面量声明

使用 `[var1, var2, var3...]` 形式快速声明一个 List。Yak 语言会根据列表内的元素类型自动推断合适的 List 类型。例如：

```
a = [1, 2, 3]
println(typeof(a)) // 输出: []int

b = ["qwe", "asd"]
println(typeof(b)) // 输出: []string

c = [1, 2, "3"]
println(typeof(c)) // 输出: []interface {}
```

列表类型的自动推断规则为：

- 如果列表内混合了不同的数据类型，那么这个列表的类型为 any。
- 如果列表内只有整数，类型为 int。
- 如果列表内既有整数又有浮点数，类型为 float。

按类型构建（make）

Yak 语言的 List 支持使用 `make` 语法创建。例如：

```
// 创建一个空的 []int 类型列表
a = make([]int)
println(typeof(a)) // 输出: []int

// 创建一个带有2个元素的 []int 类型列表
b = make([]int, 2)
println(len(b)) // 输出: 2
```

列表操作

Yak 语言的 List 支持一系列操作和内置函数。例如：

```
a = [1, 2]
b = [4, 5, 6]

// 列表追加
a = append(a, 3) // a 的值变为: [1, 2, 3]

// 列表合并
a = a + b // a 的值变为: [1, 2, 3, 4, 5, 6]

// 访问列表元素
println(a[0]) // 输出: 1
println(a[:2]) // 输出: [1, 2]
println(a[::-1]) // 输出: [6, 5, 4, 3, 2, 1]
```

通过以上示例，读者应该已经对 Yak 语言中的 List 类型有了基本的了解。List 提供了丰富的操作和内置函数，使得处理数组变得简单直观。

列表的内置方法

除了上述的基本操作之外，Yak语言还提供了一套列表的“内置方法”方便用户直接对列表进行增删改查，读者可以根据下面的代码实例尝试使用Yak语言的内置方法：

- **创建和添加元素**：我们首先创建一个数组 `a = [1, 2, 3]`。然后，我们可以使用 `Append` 方法在数组的末尾添加一个元素，如：`a.Append(4)`。此时，`a` 的值应该为 `[1, 2, 3, 4]`。

```
a = [1, 2, 3]
a.Append(4)
println(a)  // 输出: [1, 2, 3, 4]
```

- **获取长度和容量**：我们可以使用 `Length` 方法获取数组的长度，如：`a.Length()`。此外，可以使用 `Capability` 方法获取数组的容量，如：`a.Capability()`。

```
println(a.Length())  // 输出: 4
println(a.Capability())  // 输出: 4
```

- **扩展数组**：我们可以使用 `Extend` 方法将新的数组 `[5, 6]` 添加到原数组的末尾，如：`a.Extend([5, 6])`。此时，`a` 的值应该为 `[1, 2, 3, 4, 5, 6]`。

```
a.Extend([5, 6])
println(a)  // 输出: [1, 2, 3, 4, 5, 6]
```

- **删除元素**：我们可以使用 `Pop` 方法删除数组的最后一个元素，如：`a.Pop()`。此外，我们还可以指定索引来删除数组中的特定元素，如：`a.Pop(1)`。

```
a = [1, 2, 3, 4]
v = a.Pop()
println(a)  // 输出：[1, 2, 3]
println(v)  // 输出：4
```

- **插入元素**：我们可以使用 `Insert` 方法在数组的特定位置插入一个元素，如：`a.Insert(1, 2)`。此时，`a` 的值应该为 `[1, 2, 3, 4]`。

```
a = [1, 3, 4]
a.Insert(1, 2)
println(a)  // 输出：[1, 2, 3, 4]
```

- **移除元素**：我们可以使用 `Remove` 方法移除数组中的一个元素，如：`a.Remove(1)`。此时，`a` 的值应该为 `[2, 1]`。

```
a = [1, 2, 1]
a.Remove(1)
println(a)  // 输出：[2, 1]
```

- **反转数组**：我们可以使用 `Reverse` 方法将数组的内容反转，如：`a.Reverse()`。此时，`a` 的值应该为 `[4, 3, 2, 1]`。

```
a = [1, 2, 3, 4]
a.Reverse()
println(a)  // 输出：[4, 3, 2, 1]
```

- **排序数组**：我们可以使用 `Sort` 方法对数组进行排序，如：`a.Sort()`。此时，`a` 的值应该为 `[1, 2, 3, 4]`。我们还可以通过传递 `true` 参数进行降序排序，如：`a.Sort(true)`。此时，`a` 的值应该为 `[4, 3, 2, 1]`。

```
a = [4, 1, 3, 2]
a.Sort()
```

```
3 println(a)  // 输出: [1, 2, 3, 4]
```

- **映射数组**：我们可以使用 `Map` 方法对数组中的每一个元素进行函数操作，如：`a.Map(func (v) {return v + 1})`。此时，`a` 的值应该为 `[2, 3, 4, 5]`。

```
1 a = [1, 2, 3, 4]
2 a = a.Map(func (v) {return v + 1})
3 println(a)  // 输出: [2, 3, 4, 5]
```

- **过滤数组**：我们可以使用 `Filter` 方法对数组中的每一个元素进行过滤，如：`a.Filter(func (v) {return v > 2})`。此时，`a` 的值应该为 `[3, 4]`。

```
1 a = [1, 2, 3, 4]
2 a = a.Filter(func (v) {return v > 2})
3 println(a)  // 输出: [3, 4]
```

- **清空数组**：我们可以使用 `Clear` 方法移除所有元素，如：`a.Clear()`。此时，`a` 的值应该为 `[]`。

```
1 a = [1, 2, 3]
2 a.Clear()
3 println(a)  // 输出: []
```

笔者将常见的内置方法展示在下面表格中，读者可以自行查阅并尝试构建测试案例学习使用：

方法名	参数	描述	示例
Append/Push	element	往数组/切片最后追加元素	`a.Append(1)` 或 `a.Push(1)`
Pop	(可选) index	弹出数组/切片的一个元素，默认为最后一个	`a.Pop()` 或 `a.Pop(1)`
Extend/Merge	newSlice	用一个新的数组/切片扩展原数组/切片	`a.Extend(b)` 或 `a.Merge(b)`
Length/Len	无	获取数组/切片的长度	`a.Length()` 或 `a.Len()`
Capability/Cap	无	获取数组/切片的容量	`a.Capability()` 或 `a.Cap()`

StringSlice	无	将数组/切片转换成[]string	`a.StringSlice()`
GeneralSlice	无	将数组/切片转换成最泛化的 Slice 类型[]any ([]interface{})	`a.GeneralSlice()`
Shift	无	从数据开头移除一个元素	`a.Shift()`
Unshift	element	从数据开头增加一个元素	`a.Unshift(1)`
Map	mapFunc	对数组/切片中的每个元素进行指定的函数运算，返回结果	`a.Map(func(i) { return i * 2 })`
Filter	filterFunc	根据指定的函数过滤数组/切片中的元素，返回结果	`a.Filter(func(i) { return i > 3 })`
Insert	index, element	在指定位置插入元素	`a.Insert(1, 2)`
Remove	element	移除数组/切片的第一次出现的元素	`a.Remove(1)`
Reverse	无	反转数组/切片	`a.Reverse()`
Sort	(可选) reverse	对数组/切片进行排序，可选参数 reverse 决定是否反向排序	`a.Sort()` 或 `a.Sort(true)`
Clear	无	清空数组/切片	`a.Clear()`
Count	element	计算数组/切片中元素数量	`a.Count(1)`
Index	indexInt	返回数组/切片中第n个元素	`a.Index(1)`

3.3.2 字典类型：map

在编程世界中，字典是我们经常使用的一种数据结构，它允许我们存储键值对。在Yak语言中，你会发现字典非常灵活且强大，支持多种创建、访问和操作方法。这一章节，笔者将详细解析Yak语言中的字典类型，帮助你在Yak程序中高效地使用字典。

创建字典

在Yak语言中，你可以直接使用字面量形式创建字典。例如，你可以创建一个基于字符串键和整数值的字典，如下所示：

```
m = {"a": 1, "b": 2}
println(typeof(m)) // 输出: map[string]int
```

你也可以创建一个基于字符串键和字符串值的字典，或者使用混合类型的键和值：

```
m2 = {"a":"b", "c":"d"}
println(typeof(m2)) // 输出: map[string]string

m3 = {"a": 1, "b": 1.5, "c": "d"}
println(typeof(m3)) // 输出: map[string]interface{}

m4 = {1: 2, "3":"4", "5": 6.0}
println(typeof(m4)) // 输出: map[interface{}]interface{}
```

除了字面量创建法，你还可以使用 `make` 函数来创建和初始化字典。这提供了一种更加直观的方式。以下是创建一个空字典和指定字典初始容量的例子：

```
a = make(map[string]int)
a["a"] = 1
println(a) // 输出: map[a:1]

b = make(map[string]var, 2)
b["x"] = true
b["y"] = "yak"
println(b) // 输出: map[x:true y:yak]
```

字典的基础操作

在Yak语言中，你会发现字典的使用方式与Go语言非常相似。以下是一些常用的操作示例：

首先，我们创建一个字典：

```
a = {"a": 234, "b": "sasdfasdf", "ccc": "13"}
```

- 获取字典的长度：

```
println("len(a): ", len(a)) // 输出: len(a):  3
```

- 获取字典中的值：

```
println(`a["b"]: `, a["b"]) // 输出: a["b"]:  sasdfasdf
println(`a["b"]: `, a.b) // 输出: a["b"]:  sasdfasdf
v = "b"
println(`a.$v: `, a.$v) // 输出: a.$v:  sasdfasdf
```

> 这行代码可能对新手来说比较难理解：
>
> ```
> println(`a.$v: `, a.$v) // 输出: a.$v:  sasdfasdf
> ```
>
> 在这行代码中，`println`是一个打印函数，用于输出信息到控制台。`a.$v`是一个特殊的语法，用于在字典`a`中查找键为`v`的值。在这个例子中，`v`的值为"b"，所以`a.$v`就等同于`a["b"]`，结果是"sasdfasdf"。
>
> 这种使用变量名作为键来访问字典值的方式在许多编程语言中都有应用，例如JavaScript。这种方式非常灵活，可以在运行时动态地决定要访问的键。

如果尝试获取字典中不存在的值，Yak语言会让取值变为`undefined`，但是如果你想使用默认值来代替`undefined`，这仍然是可以做到的：

```
a = {"a": 234, "b": "sasdfasdf", "ccc": "13"}
assert a["non-existed-key"] == undefined

f = "f" in a ? a["f"] : "fffff" // f 的值为 (string) (len=5) "fffff"
g = get(a, "g", "ggggg")        // g 的值为 (string) (len=5) "ggggg"
```

- 添加或修改字典中的值：

```
a = {}
// 为字典添加三个键值对
a["e"], a["f"], a["g"] = 4, 5, 6

/*
a:

(map[string]interface {}) (len=3) {
```

```
 (string) (len=1) "e": (int) 4,
 (string) (len=1) "f": (int) 5,
 (string) (len=1) "g": (int) 6
}
*/
```

除了使用索引调用的方式之外，Yak语言还支持直接使用“成员字段” 的方式使用，因此上述代码等价于：

```
a = {}
// 为字典添加三个键值对
a.e, a.f, a.g = 4, 5, 6

/*
a:

(map[string]interface {}) (len=3) {
 (string) (len=1) "e": (int) 4,
 (string) (len=1) "f": (int) 5,
 (string) (len=1) "g": (int) 6
}
*/
```

- 删除字典中的键：

```
delete(a, "b")
```

- 判断键是否存在于字典中：

```
if a["b"] != nil {
    println("key b in a")
}
if "b" in a {
    println("key b in a")
}
```

- 获取字典中所有的键：

```
v = a.Keys()
```

```
2 v.Sort()
```

- 获取字典中所有的值：

```
1 v = a.Values()
2 v.Sort()
```

- 遍历字典中所有的键值对：

```
1 for k, v in a.Items() {
2     assert k in ["a","b"]
3     assert v in [1,2]
4 }
```

- 使用函数遍历字典中所有的键值：

```
1 a.ForEach(func(k,v){
2     assert k in ["a","b"]
3     assert v in [1,2]
4 })
```

字典的内置方法

和列表类似，在Yak语言中，字典也可以使用内置方法来实现

方法名	参数	描述	代码实例
Keys	无	获取所有元素的key	`keys = myMap.Keys()`
Values	无	获取所有元素的value	`values = myMap.Values()`
Entries / Items	无	获取所有元素的entity	`entries = myMap.Entries()` 或 `items = myMap.Items()`
ForEach	handler	遍历元素	`myMap.ForEach(func(k, v) { println(k, v) })`
Set	key, value	设置元素的值，如果key不存在则添加	`myMap.Set("newKey", "newValue")`
	key	移除一个值	

Remove / Delete			`myMap.Remove("keyToRemove")` 或 `myMap.Delete("keyToRemove")`
Has / IsExisted	key	判断map元素中是否包含key	`exists = myMap.Has("keyToCheck")` 或 `exists = myMap.IsExisted("keyToCheck")`
Length / Len	无	获取元素长度	`length = myMap.Length()` 或 `length = myMap.Len()`

这些方法都是通过 `myMap` 这个字典实例进行调用的，你需要将 `myMap` 替换为你自己的字典实例名。另外，这些方法的具体实现可能会因为Yak语言的版本或实现差异而有所不同。如果在实际使用中遇到问题，建议查阅相关的Yak语言文档或者参考源代码。

3.3.3 通道类型：channel

在并发编程中，多任务间如何安全、高效地交换数据是一个重要的问题。Yak语言引入了一种特殊的数据类型 - channel，它就像是一个邮局，可以帮助不同任务（也叫协程）之间轻松地发送和接收数据。本章节将探讨Yak中的channel，帮助读者更好地理解和使用这个强大的工具。

> 注意：Yak语言中的并发笔者在后续的章节中会为读者详细介绍，本小节内容更深入的channel的使用，读者可以在后续的章节中找到更多案例；

创建与声明

在Yak中，可以使用make函数来创建一个新的channel。这就像是开设一个新的邮局，用于发送和接收包裹。

```
ch := make(chan int)      // 创建一个没有存储空间的int类型的channel
ch2 := make(chan var, 2)  // 创建一个有2个存储空间的var类型的channel
```

Channel 的基本操作：

读写数据

将数据写入channel就像是将包裹寄出，非常简单。

```
ch <- 1      // 将一个整数1寄出到ch，但是因为 ch 在上文创建时没有声明存储空间，这里会阻塞
```

上述案例一般会阻塞（或者描述为“卡住”），这是因为 ch 并没有存储空间，如果读者使用下面的代码，将不会阻塞，因为 ch2 中包含两个存储空间，在存储空间填满之前，写入数据将不会是阻塞的。

```
ch2 <- "a"    // 将一个字符串寄出到ch2
ch2 <- 0      // 将一个整数0寄出到ch2
```

从channel中读取数据就像是从邮局取走包裹。

```
v := <-ch     // 从ch取走一个包裹
```

同时，Yak还支持检查取走包裹是否成功的特性，这是通过返回两个值来实现的。

```
v, ok := <-ch
if ok {
    println("取走成功，值为:", v)
} else {
    println("邮局已关闭")
}
```

Channel的属性

Yak提供了一些内置的函数来检查channel的当前状态。

```
len(ch2)    // 查看ch2中还有多少个包裹
cap(ch2)    // 查看ch2最多能存放多少个包裹
```

关闭Channel

当笔者不再需要发送更多的数据时，应该关闭channel。

```
close(ch2)
```

关闭后的channel不能再寄出数据，但仍然可以从中取走数据。

遍历Channel

与其他数据结构一样，读者可以使用for循环来取走channel中的所有包裹。

```
for v = range ch2 {
    println("从ch2取走:", v)
```

```
3 }
```

3.4 类型转换

在了解完Yak语言中强大的数据类型系统之后，读者应该发现在数据类型或复合类型中，或多或少都存在一些“类型转换”问题，例如：

- 整数遭遇浮点数的时候，会被当成浮点数一样处理；
- 一个列表在创建时：Yak语言会根据传入的类型自动选择合适的复合类型来构建列表；同样的事情也发生在“字典”中；

在编程的过程中，读者可能会遇到需要将一种类型的数据转换为另一种类型的情况。为了解决这种需求，Yak语言提供了一套方便、灵活且强大的类型转换机制。在这一章，笔者将详细介绍如何在Yak语言中进行类型转换，以及这些转换的内部细节和注意事项。

3.4.1 隐式类型转换

在某些操作中，Yak会自动进行类型转换，就像一个聪明的翻译员，自动帮你把不同的语言翻译成你想要的语言。这种情况通常发生在调用某些内置函数时，Yak会检查变量类型并进行隐式类型转换。转换的优先级为：byte(uint8) < int < float。

一般作为用户来说，不需要额外关注“隐式类型转换”，但是Yak语言有一些特殊的隐式类型转换需要用户有一个基本印象：在Yak语言中，**字符串一般来说和字符序列基本可以互相隐式转换**，在需要的时候他会作为最合适的类型出现。这屏蔽掉了很多因为字符序列和字符串混用导致的编程Bug。

3.4.2 显式类型转换

除了隐式类型转换，Yak还支持显式类型转换，就像是你直接告诉翻译员你想要的语言一样。这是通过所谓的“伪函数”来实现的。例如：

```
a = "123"
aInt = int(a)    // 显式类型转换
```

在上面的例子中，字符串"123"被转换为整数123。

类型转换方案

下表列出了Yak语言支持的各种类型转换，以及这些转换的具体细节：

原类型\目标类型	int	bool	float	byte	string	var

int	-	非0为真	直接转，无信息丢失	可能会有信息丢失	等同于sprintf("%d",num)	-
bool	真为1，假为0	-	真为1，假为0	真为1，假为0	"true"和"false"	-
float	只保留整数部分	非0为真	-	只保留整数部分	等同于sprintf("%f",num)	-
byte	直接转，无信息丢失	非0为真	直接转，无信息丢失	-	等同于sprintf("%d",num)	-
string	解析字符串内的数据，失败则会抛出错误	同左	同左	同左	-	-
var	var类型变量可以储存任意类型数据，可以通过强制类型转换转回原类型	同左	同左	同左	同左	-

下面的教程将以表格中的类型转换为例，展示如何进行各种类型的转换：

1. **1. int转其他类型**

```
num = 10
b = bool(num)                    // 非0为真
dump(b)                          // 输出: (bool) true

f = float(num)                   // 直接转，无信息丢失
println("%T %.1f" % [f, f])      // 输出: float64 10.0

s = string(num)                  // 等同于 sprintf("%d",num)
dump(s)                          // 输出: (string) (len=2) "10"
```

注意：int转byte可能会有信息丢失。

2. **2. bool转其他类型**

```
b = true
n = int(b)        // 真为1，假为0
println(n)        // 输出: 1
```

```
4
5 f = float(b)      // 真为1，假为0
6 println(f)        // 输出: 1.0
7
8 s = string(b)     // "true"和"false"
9 println(s)        // 输出: "true"
```

注意：bool转byte，真为1，假为0。

3. 3. float转其他类型

```
1 f = 10.5
2 n = int(f)        // 只保留整数部分
3 println(n)        // 输出: 10
4
5 b = bool(f)       // 非0为真
6 println(b)        // 输出: true
7
8 s = string(f)     // 等同于 sprintf("%f",num)
9 println(s)        // 输出: "10.5"
```

注意：float转byte只保留整数部分。

4. 4. byte转其他类型

```
1 b = byte(10)
2 n = int(b)        // 直接转，无信息丢失
3 println(n)        // 输出: 10
4
5 f = float(b)      // 直接转，无信息丢失
6 dump(f)           // 输出: (float64) 10
7
8 s = string(b)     // 等同于 sprintf("%d",num)
9 println(s)        // 输出: "10"
```

注意：byte转bool，非0为真。

5. 5. string转其他类型

```
1 s = "123"
2 n = int(s)        // 解析字符串内的数据，失败则会抛出错误
3 println(n)        // 输出: 123
4
```

```
b = bool(s)       // 解析字符串内的数据，失败则会抛出错误
println(b)        // 输出: true

f = float(s)      // 解析字符串内的数据，失败则会抛出错误
dump(f)           // 输出: (float64) 123
```

注意：string转byte也需要解析字符串内的数据。

6. **6. var转其他类型**

var类型变量可以储存任意类型数据，可以通过强制类型转换转回原类型。

```
v = var(123)
n = int(v)        // 强制类型转换
dump(n)           // 输出: (int) 123
```

注意：var转其他类型都需要通过强制类型转换。

希望这个教程能帮助读者更好地理解和使用Yak语言中的类型转换。

3.5 表达式与运算符

3.5.1 基本概念

在编程的世界中，读者会频繁地遇到各种计算和操作，这就需要理解运算符和表达式的使用。在 Yak 语言中，运算符是一种特殊的符号，被用来执行各种计算和操作，例如加法、减法、乘法、除法、比较、逻辑操作等。表达式则是由一个或多个运算符和操作数（如变量或字面量）组成的代码片段，它能够计算出一个值。

以 `a + b` 为例，这就是一个表达式，其中 `+` 是运算符，`a` 和 `b` 是操作数。这个表达式的值就是 `a` 和 `b` 的和。

理解运算符和表达式是学习任何编程语言的基础，因为它们是构成编程语言的基本元素。在 Yak 语言中，读者会遇到各种不同类型的运算符，包括算术运算符、比较运算符、逻辑运算符、位运算符、赋值运算符等。这些运算符的优先级和结合性决定了表达式的计算顺序。

对于编程新手来说，理解表达式的关键在于理解它是如何计算出一个值的。可以将表达式视为一个简单的数学公式，它由运算符和操作数组成，运算符定义了操作数如何组合以计算出一个值。例如，在表达式 `3 + 4 * 2` 中，由于乘法运算符的优先级高于加法运算符，所以首先计算 `4 * 2` 得到 `8`，然后再与 `3` 相加，得到最终结果 `11`。

在实际编程中，读者可能会遇到更复杂的表达式，例如包含函数调用或条件运算符的表达式。但是，无论表达式多么复杂，其核心都是由运算符和操作数组成，并按照一定的优先级和结合性规则进行计算。

理解和掌握运算符和表达式是编程的基础，它能够帮助读者更好地理解和编写代码，解决实际问题。笔者在这个章节中将详细介绍 Yak 语言中的运算符和表达式。

3.5.2 运算符

基础运算符

在Yak语言中，基础的数学运算运算符包括加('+')、减('-')、乘('*')、除('/')，这些运算符的使用方法与我们在数学中的使用方法相同。例如，你可以写出如下的表达式：

```
1 result = 1 + 4 * 5
```

此外，Yak语言也支持取余数('%')操作，这个操作在处理诸如“每隔一定数量的循环”等问题时非常有用。

赋值运算符

在Yak语言中，赋值运算符有两种形式：'='和':='。它们的作用是一样的，都是将右侧的值赋给左侧的变量。例如：

```
1 a = 1
2 b := 2
```

这两行代码都是有效的，它们分别将1和2赋值给了变量a和b。

位运算

Yak语言支持一系列的位运算，包括按位与('&')、按位或('|')、按位异或('^')、按位清零('&^')、左移('<<')、右移('>>')。这些运算符在处理二进制数据时非常有用。

赋值

在Yak语言中，你可以使用特殊的运算赋值运算符，如'+='、'-='、'*='、'/='、'%='，它们的作用是将左侧的变量与右侧的值进行相应的运算，然后将结果赋值给左侧的变量。例如：

```
1 a += 1
```

这行代码的效果等同于 `a = a + 1`。

此外，Yak语言还提供了'++'和'--'运算符，它们分别表示将变量的值增加1和减少1。

关系运算符

在Yak语言中，关系运算符用于比较两个值的关系。这些运算符包括大于('>')、小于('<')、等于('==')、不等于('!=')、大于等于('>=')和小于等于('<=')。

当你需要判断一个数是否大于、小于、等于、不等于、大于等于或小于等于另一个数时，你可以使用这些运算符。以下是一些使用示例：

```
a = 5
b = 3

println(a > b)  // 输出: true
println(a < b)  // 输出: false
println(a == b) // 输出: false
println(a != b) // 输出: true
println(a >= b) // 输出: true
println(a <= b) // 输出: false
```

在这些例子中，你可以看到各种关系运算符的使用方法。请注意，这些运算符只能用于可以比较的类型，如数字和字符串，不能用于不能比较的类型，如数组和字典。

逻辑运算符

逻辑运算符在Yak语言中用于进行逻辑操作，包括逻辑与('&&')和逻辑或('||')。逻辑与运算符会在两个操作数都为真时返回真，否则返回假。逻辑或运算符会在至少有一个操作数为真时返回真，否则返回假。这两个运算符都具有短路特性，即如果左侧的操作数已经能确定整个表达式的值，那么就不会再计算右侧的操作数。以下是一些使用示例：

```
a = true
b = false

println(a && b) // 输出: false
println(a || b) // 输出: true
```

在这个例子中，对于逻辑与运算，由于b为假，所以不论a的值是什么，整个表达式的值都为假；对于逻辑或运算，由于a为真，所以不论b的值是什么，整个表达式的值都为真。

这两个运算符都具有短路特性，即如果左侧的操作数已经能确定整个表达式的值，那么就不会再计算右侧的操作数。

让我们来看一些具体的例子：

```
a = true
```

```
b = false
c = a && println("Hello")

/*
OUTPUT:

Hello
*/
```

在这个例子中，`println("Hello")`是一个函数调用表达式，会打印"Hello"。但是，由于a为真，所以`a && println("Hello")`的值取决于b。因此，程序会打印出"Hello"。

然而，如果我们将a改为false：

```
a = false
b = false
c = a && println("Hello") // 无输出
```

在这种情况下，由于a为假，所以不论`println("Hello")`的值是什么，`a && println("Hello")`的值都为假。因此，程序不会打印"Hello"。

同样的，对于逻辑或运算符：

```
a = true
b = true
c = a || println("Hello")
```

在这种情况下，由于a为真，所以不论`println("Hello")`的值是什么，`a || println("Hello")`的值都为真。因此，程序不会打印"Hello"。

以上就是逻辑运算符的短路特性的一些详细示例，读者可以自行编写代码检验这个有趣的特性；

三元逻辑运算符

三元运算符在Yak语言中的形式为`condition ? value1 : value2`。如果condition为真，则表达式的结果为value1，否则为value2。这个运算符也具有短路特性，即如果条件已经能确定整个表达式的值，那么就不会再计算其他的值。以下是一些使用示例：

```
a = 5
b = 3

result = a > b ? a : b
```

```
println(result) // 输出：5
```

在这个例子中，由于a大于b，所以 `a > b` 为真，因此整个表达式的值为a。

值得一提的是，三元逻辑运算符仍然有短路特性，请读者再次看以下这些案例：

```
a = 5
b = 3

result = a > b ? println("Hello") : println("World")
// 输出：Hello
```

在这个例子中， `a > b` 为真，因此整个表达式的值为 `println("Hello")` 。这个表达式打印"Hello"。请注意， `println("World")` 并没有被执行，这就是短路特性的表现。

然而，如果我们将a和b的值交换：

```
a = 3
b = 5

result = a > b ? println("Hello") : println("World")
// 输出：World
```

在这个例子中， `a > b` 为假，因此整个表达式的值为 `println("World")` 。这个表达式打印"World"。请注意， `println("Hello")` 并没有被执行，这同样是短路特性的表现。

所有支持的运算符列表

运算符	说明	代码示例
*	乘法	`a = 5 * 3; // a = 15`
/	除法	`a = 15 / 3; // a = 5`
%	取余	`a = 10 % 3; // a = 1`
<<	左移	`a = 1 << 2; // a = 4`
<	小于	`result = 5 < 3; // result = false`
>>	右移	`a = 4 >> 2; // a = 1`
>	大于	`result = 5 > 3; // result = true`

&	按位与	`a = 5 & 3; // a = 1`
&^	位清零 (AND NOT)	`a = 5 &^ 3; // a = 4`
+	加法	`a = 5 + 3; // a = 8`
-	减法	`a = 5 - 3; // a = 2`
^	按位异或	`a = 5 ^ 3; // a = 6`
|	按位或	`a = 5 | 3; // a = 7`
==	等于	`result = 5 == 3; // result = false`
<=	小于或等于	`result = 5 <= 3; // result = false`
>=	大于或等于	`result = 5 >= 3; // result = true`
!=	不等于	`result = 5 != 3; // result = true`
<>	不等于 (等价于!=)	`result = 5 <> 3; // result = true`
<-	通道操作符	`value = <-channel; // 接收通道中的值`
&&	逻辑与	`result = true && false; // result = false`
||	逻辑或	`result = true || false; // result = true`
?:	三元操作符（条件）	`result = 5 > 3 ? 5 : 3; // result = 5`
=	赋值	`a = 5;`
~	函数调用时处理错误，如果出现错误则直接崩溃	`result = someFunction()~; // 函数出错则崩溃`
:=	强制赋值	`a := 5; // 强制赋值`
++	自增	`a = 5; a++; // a = 6`
--	自减	`a = 5; a--; // a = 4`
+=	加赋值	`a = 5; a += 3; // a = 8`
-=	减赋值	`a = 5; a -= 3; // a = 2`
*=	乘赋值	`a = 5; a *= 3; // a = 15`
/=	除赋值	`a = 15; a /= 3; // a = 5`

%=	取余赋值	`a = 10; a %= 3; // a = 1`
^=	异或赋值	`a = 5; a ^= 3; // a = 6`
<<=	左移赋值	`a = 1; a <<= 2; // a = 4`
>>=	右移赋值	`a = 4; a >>= 2; // a = 1`
&=	与赋值	`a = 5; a &= 3; // a = 1`
|=	或赋值	`a = 5; a |= 3; // a = 7`
&^=	位清零赋值	`a = 5; a &^= 3; // a = 4`
!	逻辑非	`result = !true; // result = false`
.	访问内部成员	`a = {"foo": "bar"}; println(a.foo); // 输出: "bar"`
() => {}	闭包箭头函数	`f = () => { println("Hello, World!"); }; f(); // 输出: Hello, World!`
in	包含关系操作符	`a = "abcd"; "abc" in a; // true`

3.5.3 运算符与表达式优先级

在Yak语言中，理解运算符的优先级和执行顺序是至关重要的，因为这将影响到表达式的计算结果。运算符的优先级从高到低可以概括如下：

1. **单目运算**：包括类型字面量、字面量、匿名函数声明、Panic 和 Recover 函数、标识符、成员调用、切片调用、函数调用、括号表达式、闭包实例代码、make 表达式以及一元运算符表达式。
2. **二元位运算**：包括位二元位运算。在 Yak 中，位运算优先级高于数学运算，例如 `<<`，`>>`，`&`，`&^` 等就是位运算。
3. **数学运算**：包括乘性运算和加性运算。乘性运算的优先级高于加性运算：`*` 和 `/` 我们一般视为乘性运算。
4. **比较运算**：包括各种比较运算符。
5. **包含运算**：包括 'in' 运算符，它是初级逻辑运算，并不具备短路特征。
6. **高级逻辑运算**：包括 '&&' 和 '||' 逻辑运算，具备短路特征。
7. **三元运算**：包括 '?' 和 ':' 运算符。
8. **管道操作符**：包括 '<-' 运算符。

> 值得一提的是在Yak中“<-”也是一种一元运算符，当其以 `<- channel` 的形式出现时，优先级等同单目运算

理解这些运算符的优先级对于正确理解和编写 Yak 语言程序是非常重要的。在编写复杂的表达式时，笔者建议使用括号 () 来明确运算顺序，以避免可能的混淆和错误。

以下是一些代码案例，可以帮助读者理解运算符的优先级：

1. **乘性运算优先级高于加性运算**：

```
a = 2 + 3 * 4; // 结果是 14，而不是 20
```

在这个例子中，由于乘法运算的优先级高于加法运算，所以首先执行 3 * 4 得到 12，然后再加上 2，得到结果 14。

1. **位运算优先级高于乘性运算**：

```
b = 4 * 2 & 3; // 结果是 8，而不是 0
b = (4 * 2) & 3; // 结果是 0，而不是 8
```

在这个例子中，由于位运算的优先级高于乘法运算，所以首先执行 2 & 3 得到 2，然后再乘以 4，得到结果 8。

使用括号先计算乘法的时候，`2 * 4` 得到 `8`，二进制写 `0b1000` 和 `0b11` 进行按位与计算，结果为0；

1. **括号可以改变优先级**：

```
c = (2 + 3) * 4; // 结果是 20，而不是 14
```

在这个例子中，由于括号的存在，首先执行括号内的加法运算 2 + 3 得到 5，然后再乘以 4，得到结果 20。

这些例子说明了在 Yak 语言中，运算符的优先级和执行顺序对于正确解析和计算表达式是非常重要的。读者在编写代码时，需要特别注意这一点。