第七章：标准库函数

欢迎各位读者来到本教程的第7章，本章我们即将一起探索这门新语言中不可或缺的一环——基础内置函数。

在之前的六章中，各位读者已经与笔者一同建立了坚实的编程基础，学习了如何声明变量、理解数据类型、控制程序流程、以及编写基础的程序代码。编程，正如各位已经体会到的，本质上是一种与计算机的对话，而内置函数则赋予了我们与计算机对话的更多词汇和表达力。现在，我们将一起拓展我们的编程视野，深入学习那些能够使我们的编程既高效又充满表现力的强大工具。

内置函数是编程语言的核心，它们是预先定义好的、随时可供调用的功能单元，为我们执行常见任务提供了便捷的途径。笔者在这里要强调的是，正确地利用内置函数，能够显著提高我们的编程效率，简化代码结构，并减少不必要的重复性工作。

在本章中，笔者将与各位读者一起，深入探讨以下几个关键议题：

1. **全局函数**：这些函数无处不在，它们不需要特别的引用或声明就能够被调用。它们为我们的编程提供了基本但强大的能力，比如进行输出打印、数学计算和数据类型之间的转换。
2. **内置库函数**：我们的语言自带了一系列强大的标准库，这些库覆盖了广泛的功能，从文本处理到数学运算，再到数据结构的操作。通过这些库，各位读者可以轻松访问到高级功能，同时保持代码的整洁和可读性。
3. **函数的重要性**：内置函数的价值远不止于便利性，它们是我们编程语言哲学的体现——遵循“不要重复自己”（DRY）的原则。借助这些经过严格测试和优化的代码，我们能够减少错误发生的机会，提升编程效率，并将注意力集中在程序的核心逻辑上。
4. **实践案例**：笔者将提供具体的代码示例，指导各位读者如何在实际的编程中应用这些内置函数。这些示例将帮助大家理解每个函数的具体用途，并学会如何在自己的代码中高效地运用它们。

通过本章的学习，各位读者将能够熟练掌握这门语言的基础内置函数，这些函数将成为各位编程工具箱中的重要组成部分。你们将学会如何借助这些内置功能来简化编程任务，解决复杂问题，并最终编写出既强大又优雅的代码。

现在，就让我们一起开启这一章的学习之旅，深入理解这些强大的内置函数，使你们的编程之路更加平顺。

7.1 常用全局函数

在Yak语言中，我们提供了一个丰富的全局函数库，供各位读者即刻使用。除了前几章提到的输入输出操作、类型转换、错误处理等基础功能外，这里还包含了许多实用的工具函数。接下来，笔者将按照功能类别，逐一向各位展示这些全局函数的用途和威力。

7.1.1 输出函数

函数名	说明	参数	返回值	案例
`print`	输出参数到屏幕，无换行	可变参数列表	无	`print("Hello, ", "World!")`
`println`	输出参数到屏幕，并换行	可变参数列表	无	`println("Hello, World!")`
`printf`	格式化输出到屏幕，并换行	格式化字符串，可变参数	无	`printf("Hello, %s!", "World")`
`sprint`	格式化并返回一个字符串	可变参数列表	字符串	`sprint("Hello, ", "World!")`
`sprintln`	格式化并返回一个字符串，并换行	可变参数列表	字符串	`sprintln("Hello, ", "World!")`
`sprintf`	根据格式化字符串格式化并返回字符串	格式化字符串，可变参数	字符串	`sprintf("Hello, %s!", "World")`
`dump`	输出变量的详细信息	可变参数列表	无	`dump(variable)`
`sdump`	返回变量的详细信息的字符串	可变参数列表	字符串	`sdump(variable)`
`desc`	描述一个“结构体”可用的方法和信息，输出到屏幕	想要描述的结构	无	`desc(time.now())`

dump

说明： `dump` 函数用于输出变量的详细信息，便于调试。它会直接打印信息到屏幕。一般来说 `dump` 输出的函数会尽量携带人类可读的内容并附上变量类型，非常便于调试代码。

参数： 可以接受任意数量的参数。

返回值： 该函数没有返回值。

案例：

```
1 a = 1 + 1.0
2 dump(a) // (float64) 2
```

sdump

说明： `sdump` 函数与 `dump` 类似，但是它返回的是变量详细信息的字符串，而不是直接输出到屏幕。

参数： 可以接受任意数量的参数。

返回值： 返回变量详细信息的字符串。

案例：

```
a = 1 + 1.0
verboseInfo = sdump(a)
println(verboseInfo)

/*
OUTPUT:

(float64) 2
*/
```

`sprintf` 系列函数通常用于需要将多个值格式化为一个字符串时，比如构建消息、生成日志等；`print` 系列函数通常用于直接输出到屏幕，用于调试或者直接输出给用户屏幕信息；`dump` 和 `sdump` 函数则主要用于开发过程中的调试，可以帮助开发者快速查看变量的状态。

desc

说明： `desc` 一般用来描述一个用户有疑问或者“未知”的结构体，编程时调试使用，它的输出非常适合人类阅读。

使用案例：

```
timeInstance = now() // 使用 now() 函数构建一个当前时间的时间对象
desc(timeInstance)   // 使用 desc 描述这个对象可用的结构

/*
OUTPUT:

type time.(Time) struct {
  Fields(可用字段):
  StructMethods(结构方法/函数):
      func Add(v1: time.Duration) return(time.Time)
      func AddDate(v1: int, v2: int, v3: int) return(time.Time)
      func After(v1: time.Time) return(bool)
      func AppendFormat(v1: []uint8, v2: string) return([]uint8)
```

```
14        func Before(v1: time.Time) return(bool)
15        func Clock() return(int, int, int)
16        ...
17        ...
18        func Weekday() return(time.Weekday)
19        func Year() return(int)
20        func YearDay() return(int)
21        func Zone() return(string, int)
22        func ZoneBounds() return(time.Time, time.Time)
23    PtrStructMethods(指针结构方法/函数):
24 }
25 */
```

7.1.2 时间处理

Yak语言将常用的部分时间处理函数置于全局函数库中，包括时间戳、时间字符串、时间对象获取、时间戳、时间字符串、时间对象转化，等待一段时间。

函数名	说明	参数	返回值	案例
`timestamp`	获取当前时间戳	-	时间戳	`timestamp() // (int) 1699837389`
`nanotimestamp`	获取当前纳秒级时间戳	-	时间戳	`nanotimestamp() // (int) 1699837447787737000`
`datetime`	获取当前秒的时间字符串	-	年-月-日 时:分:秒格式的时间字符串	`datetime() // (string) (len=19) "2023-11-13 09:04:29"`
`date`	获取当前日的时间字符串	-	年-月-日格式的时间字符串	`date() // (string) (len=10) "2023-11-13"`
`now`	获取当前时间的时间对象	-	当前时间的时间对象	`now() // (time.Time) 2023-11-13 09:05:41.925419 +0800 CST m=+0.178098418`
`timestampToTime`	时间戳转化为时间对象	时间戳	时间戳对应的时间对象	`timestampToTime(1630425600) //(time.Time) 2021-09-01 00:00:00 +0800 CST`

`datetimeToTimestamp`	时间字符串转化为时间对象	时间字符串	时间戳，可能存在的错误	`ret, err = datetimeToTimestamp("2023-09-01 00:00:00") // (int) 1693526400, (interface {}) <nil>`
`timestampToDatetime`	时间戳转化为时间字符串	时间戳	年-月-日 时:分:秒格式的时间字符串	`timestampToDatetime(1630425600) //(string) (len=19) "2021-09-01 00:00:00"`
`sleep`	延时指定的时间	延时的秒数	-	`sleep(2.5)`
`wait`	延时指定的时间或等待上下文取消	延时时间指标或上下文	-	`wait(5)` 或 `wait(context.Background())`

读者通过阅读上述表格，应该对Yak中的“时间”处理有一个初步的认知，那么我们实际上在处理的过程中往往还会遇到其他的情况，需要大家手动把时间戳（或时间对象）按自己的要求转换格式，除了本篇讲解的基础函数，我们将在后面深入讲解 `now()` 和 `timestampToTime()` 得到的 `time.Time` 对象的使用。

7.1.3 类型转换函数

除了在“类型与变量”章节中我们掌握的类型转换之外，Yak语言还在全局变量中提供了一些便捷的类型转换函数，用户可以直接参考使用：

函数名	说明	参数	返回值	案例
`chr`	将整数转换为其对应的ASCII字符	整数 `i`	字符串	`chr(65)` 返回 `"A"`
`ord`	将字符转换为其对应的ASCII码整数	字符 `c`	整数	`ord("A")` 返回 `65`
`typeof`	返回传入参数的类型	任意 `i`	类型	`typeof("Hello")` 返回 `string`
`parseInt`	将字符串解析为十进制整数	字符串 `s`	整数	`parseInt("123")` 返回 `123`
`parseStr`	将参数转换为字符串，等价于 `sprint`	任意 `i`	字符串	`parseStr(123)` 返回 `"123"`
`parseString`	将参数转换为字符串，等价于 `sprint` ，是 `parseStr` 别名	任意 `i`	字符串	`parseString(123)` 返回 `"123"`

`parseBool`	将字符串解析为布尔值	字符串 `i`	布尔值	`parseBool("true")` 返回 `true`
`parseBoolean`	将字符串解析为布尔值，是 `parseBoolean` 别名	字符串 `i`	布尔值	`parseBoolean("F")` 返回 `false`
`parseFloat`	将字符串解析为浮点数	字符串 `i`	浮点数	`parseFloat("123.45")` 返回 `123.45`

7.1.4 其他辅助函数

函数名	说明	参数	返回值	使用案例
append	在一个列表后追加新元素，形成新列表	要追加的列表和元素	[]var: 追加后的新列表	`a = [1,2,3]; b = append(a, 4)`
len	用于获取一个对象的长度 (chan/slice/map)	i: 通道 / 列表 / 字典	int: 对象的长度	`len([1, 2, 3]); len({1: 2, 3: 4})`
die	当传入的 error 不为 nil 时会 panic	err: 错误对象	无	`err = someFunction(); die(err)`
close	关闭 chan	i: chan	无	`close(channel)`
min	取所有参数的最小值，支持 int / float	i: int / float	var: 最小值	`minValue = min(5, 3, 8)`
max	取所有参数的最大值，支持 int / float	i: int / float	var: 最大值	`maxValue = max(5, 3, 8)`
loglevel	根据传入的字符串 level 来设置日志等级	level: 日志等级字符串	无	`loglevel("debug")`
logquiet	禁用日志输出，相当于 loglevel("disable")	无	无	`logquiet()`
logdiscard	禁用日志输出，相当于 loglevel("disable")	无	无	`logdiscard()`
logrecover	设置日志输出为 os.Stdout	无	无	`logrecover()`
uuid		无		`uuidString = uuid()`

	生成一个唯一的 UUID 字符串		string: UUID 字符串	
randn	生成在 [min, max) 区间的随机整数	min: 最小值；max: 最大值	int: 随机整数	`randomInt = randn(1, 10)`
randstr	生成指定长度的随机字母字符串	len: 字符串长度	string: 随机字符串	`randomString = randstr(8)`

7.2 字符串处理库：str

在网络安全领域中有着大量的重复的性质的特定字符串处理工作，比如从指定的一段字符中提取URL字符串片段、通过无类别域间路由（Classless Inter-Domain Routing，简称CIDR）生成其表示的所有IP地址等。

常见的基础编程语言当然可以解决这些工作，但是无论是每次都重写代码还是自我创立工具库都犹如隔靴搔痒，不能达到开箱即用，快速高效的效果。而作为为网络安全而生的领域编程语言，Yak的str字符串工具库除了常规的字符串处理操作外，理所当然的内置了能快速解决这部分工作的函数。

笔者将从两个部分介绍这部分内容：“通用的字符串处理函数”和“安全领域的字符串工具函数”

7.2.1 通用的字符串处理函数

函数名（别名）	说明	参数	返回值	使用案例
`Index`	查找子串第一次出现的索引位置	`s, substr`	索引（整数）	`str.Index("hello", "e")` 返回 `1`
`StartsWith`（`HasPrefix`）	检查字符串是否以指定前缀开始	`s, prefix`	布尔值	`str.StartsWith("hello", "he")` 返回 `true`
`EndsWith`（`HasSuffix`）	检查字符串是否以指定后缀结束	`s, suffix`	布尔值	`str.EndsWith("hello", "lo")` 返回 `true`
`Contains`	检查字符串是否包含指定子串	`s, substr`	布尔值	`str.Contains("hello", "ll")` 返回 `true`
`ToLower`	将字符串中的所有字符转换为小写	`s`	字符串	`str.ToLower("Hello")` 返回 `"hello"`

`ToUpper`	将字符串中的所有字符转换为大写	`s`	字符串	`str.ToUpper("hello")` 返回 `"HELLO"`
`Trim`	从字符串两端删除指定的字符集	`s, cutset`	字符串	`str.Trim("!!hello!!", "!")` 返回 `"hello"`
`TrimSpace`	从字符串两端删除空白符	`s`	字符串	`str.TrimSpace(" hello ")` 返回 `"hello"`
`Split`	根据分隔符拆分字符串	`s, sep`	字符串数组	`str.Split("a,b,c", ",")` 返回 `["a", "b", "c"]`
`Join`	将字符串数组元素连接成一个字符串	`elems, sep`	字符串	`str.Join(["a", "b", "c"], ",")` 返回 `"a,b,c"`
`Replace`	替换字符串中的子串（替换次数）	`s, old, new, n`	字符串	`str.Replace("hello", "l", "x", 2)` 返回 `"hexxo"`
`ReplaceAll`	替换字符串中的子串	`s, old, new`	字符串	`str.Replace("Hello", "l", "x")` 返回 `"hexxlo"`
`Count`	计算子串在字符串中出现的次数	`s, substr`	整数	`str.Count("hello", "l")` 返回 `2`
`Compare`	比较两个字符串	`a, b`	整数	`str.Compare("hello", "hello")` 返回 `0`
`EqualFold`	检查两个字符串是否相等（不区分大小写）	`s, t`	布尔值	`str.EqualFold("Go", "go")` 返回 `true`
`Fields`	按空白符拆分字符串为一个字段的切片	`s`	字符串数组	`str.Fields("  foo bar baz   ")` 返回 `["foo", "bar", "baz"]`
`Repeat`		`s, count`	字符串	

	重复字符串n次			`str.Repeat("na", 2)` 返回 `"nana"`

在字符串处理的基础功能之外，我们还提供了一系列更高级的匹配函数，这些函数允许更灵活的字符串检查，包括大小写不敏感的子串匹配、通配符（glob）模式匹配，以及正则表达式匹配。这些函数能够处理各种复杂的匹配需求，并且易于使用，极大地增强了字符串处理的能力。

函数名	说明	参数	返回值	使用案例
`MatchAnyOfSubString`	判断是否有任意子串（不区分大小写）存在于输入中	`i, subStr...`	布尔值	`str.MatchAnyOfSubString("abc", "a", "z", "x")` 返回 `true`
`MatchAllOfSubString`	判断所有子串（不区分大小写）是否都存在于输入中	`i, subStr...`	布尔值	`str.MatchAllOfSubString("abc", "a", "b", "c")` 返回 `true`
`MatchAnyOfGlob`	使用glob模式匹配，判断是否有匹配成功的模式	`i, re...`	布尔值	`str.MatchAnyOfGlob("abc", "a*", "??b", "[^a-z]?c")` 返回 `true`
`MatchAllOfGlob`	使用glob模式匹配，判断是否所有模式都匹配成功	`i, re...`	布尔值	`str.MatchAllOfGlob("abc", "a*", "?b?", "[a-z]?c")` 返回 `true`
`MatchAnyOfRegexp`	使用正则表达式匹配，判断是否有匹配成功的正则	`i, re...`	布尔值	`str.MatchAnyOfRegexp("abc", "a.+", "Ab.?", ".?bC")` 返回 `true`
`MatchAllOfRegexp`	使用正则表达式匹配，判断是否所有正则都匹配成功	`i, re...`	布尔值	`str.MatchAllOfRegexp("abc", "a.+", ".?b.?", "\\w{2}c")` 返回 `true`
`RegexpMatch`	使用正则尝试匹配字符串，返回匹配结果	`pattern, s`	布尔值	`str.RegexpMatch("^[a-z]+$", "abc")` 返回 `true`

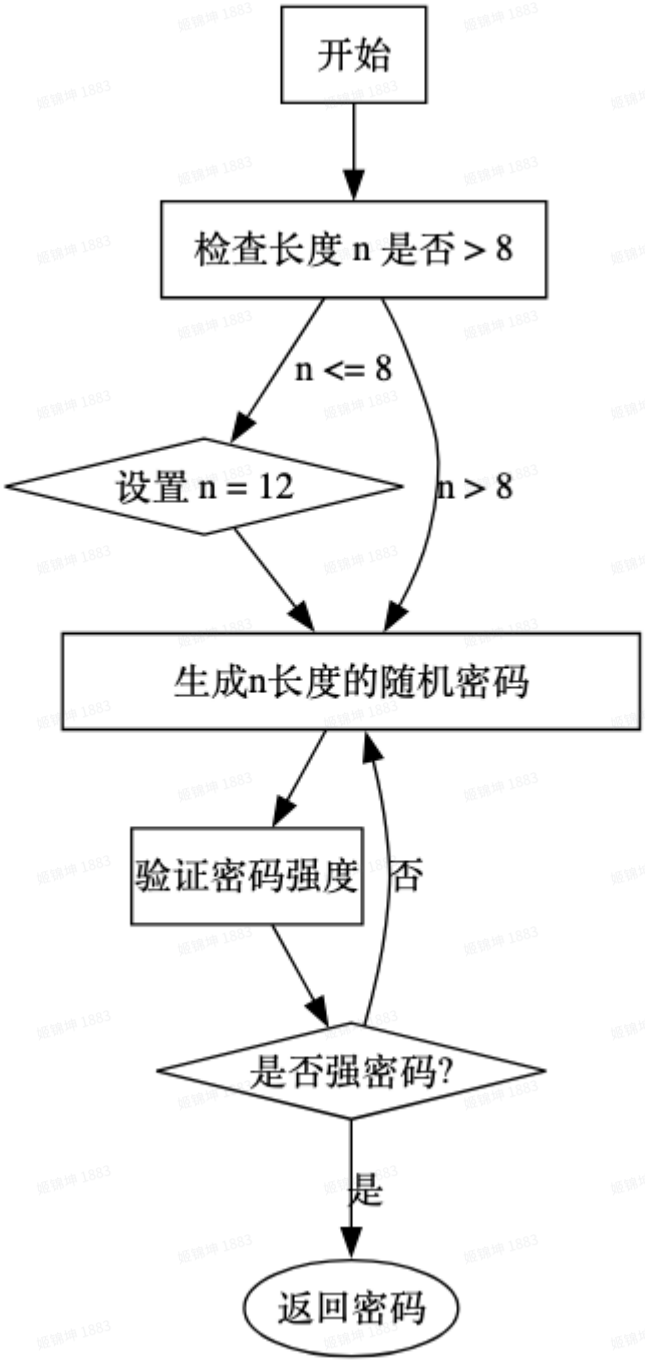
7.2.2 安全领域字符串工具函数

Yak语言中提供的“安全领域的字符串工具函数”覆盖了字符串的基本操作，还包括了安全领域所需的特定功能，如随机数生成、密码强度判断、文本相似度计算、网络数据解析、HTTP请求与响应处理、URL和路径操作、数据结构转换、版本控制以及集合操作等。这些功能的组合使得这套工具函数成为安全领域DSL开发的强有力工具。接下来，我们将深入探讨每个类别的具体函数及其应用。

随机密码与安全性

生成强密码 (`RandSecret`):

1. 确定密码长度。如果输入长度小于或等于8，将长度设置为12，因为强密码需要长度大于8。
2. 生成一个随机密码。在所有可见的ASCII字符中随机挑选指定数量的字符组成密码字符串。
3. 验证密码强度。使用 `IsStrongPassword` 函数检查生成的密码是否符合强密码的要求。
4. 如果密码不符合要求，重复步骤2和3，直到生成一个强密码。
5. 如果密码符合要求，返回这个密码字符串。



判断强密码 (`IsStrongPassword`):

1. 检查密码长度。如果长度小于或等于8，立即返回 `false` 。
2. 检查密码中是否包含至少一个特殊字符、小写字母、大写字母和数字。
3. 如果密码包含所有这些类型的字符，则返回 `true` ，表示这是一个强密码。
4. 如果密码缺少任何类型的字符，则返回 `false` ，表示这不是一个强密码。

总结上述的内容，用户可以参考如下代码进行验证：

```
// 假设 RandSecret 和 IsStrongPassword 函数已经定义好了
strongPassword = str.RandSecret(12) // 生成一个长度为12的强密码
dump(strongPassword)

```

```
isStrong = str.IsStrongPassword("YourP@ssw0rd!") // 检查 "YourP@ssw0rd!" 是否为强密码
dump(isStrong)

/*
OUTPUT:

(string) (len=12) "W)^H(={xg4i"
(bool) true
*/
```

网络与扫描目标解析与处理

函数名	说明	输入	输出	案例
ParseStringToHostPort	解析字符串为主机名和端口号	字符串，格式为"host:port"，也可以支持URL输入	主机名和端口号的元组	`host, port, err = str.ParseStringToHostPort("example.com:80")` 执行结果 `host -> example.com; port -> 80`;
ParseStringToPorts	把字符串解析为端口数组	字符串，支持逗号分隔和“-”表示范围	端口数组	`str.ParseStringToPorts("80,443,8080-8083")`
ParseStringToHosts	把字符串解析为主机字符串数组	字符串，支持逗号分隔，CIDR和域名格式	按主机分割的字符串数组	`str.ParseStringToHosts("www.example.com,192.168.1.1/24")`
IsIPv6	判断字符串是否为有效的IPv6地址	字符串	布尔值，表示是否为IPv6地址	`str.IsIPv6("2001:0db8:85a3:0000:0000:8a2e:0370:7334")` 返回 `true`
IsIPv4	判断字符串是否为有效的IPv4地址	字符串	布尔值，表示是否为IPv4地址	`str.IsIPv4("192.168.1.1")` 返回 `true`
ExtractHost	从URL或者主机端口格式中提取主机名	URL字符串	主机名字符串	`str.ExtractHost("http://www.example.com/path")` 返回 `"www.example.com"`

ExtractRootDomain	从域名中提取根域名	域名字符串	根域名字符串	`str.ExtractRootDomain("subdomain.example.com")` 返回 `["example.com"]`
SplitHostsToPrivateAndPublic	将主机名列表分为私有和公有主机名列表	需要分隔的内容（使用逗号分隔）	两个列表：私有主机名列表和公有主机名列表	`str.SplitHostsToPrivateAndPublic(`127.0.0.1,example.com`) // ["127.0.0.1"] ["example.com"]`
IPv4ToCClassNetwork	将IPv4地址转换为C类网络地址-CIDR	IPv4地址字符串	C类网络地址字符串	`str.IPv4ToCClassNetwork("192.168.1.1")`

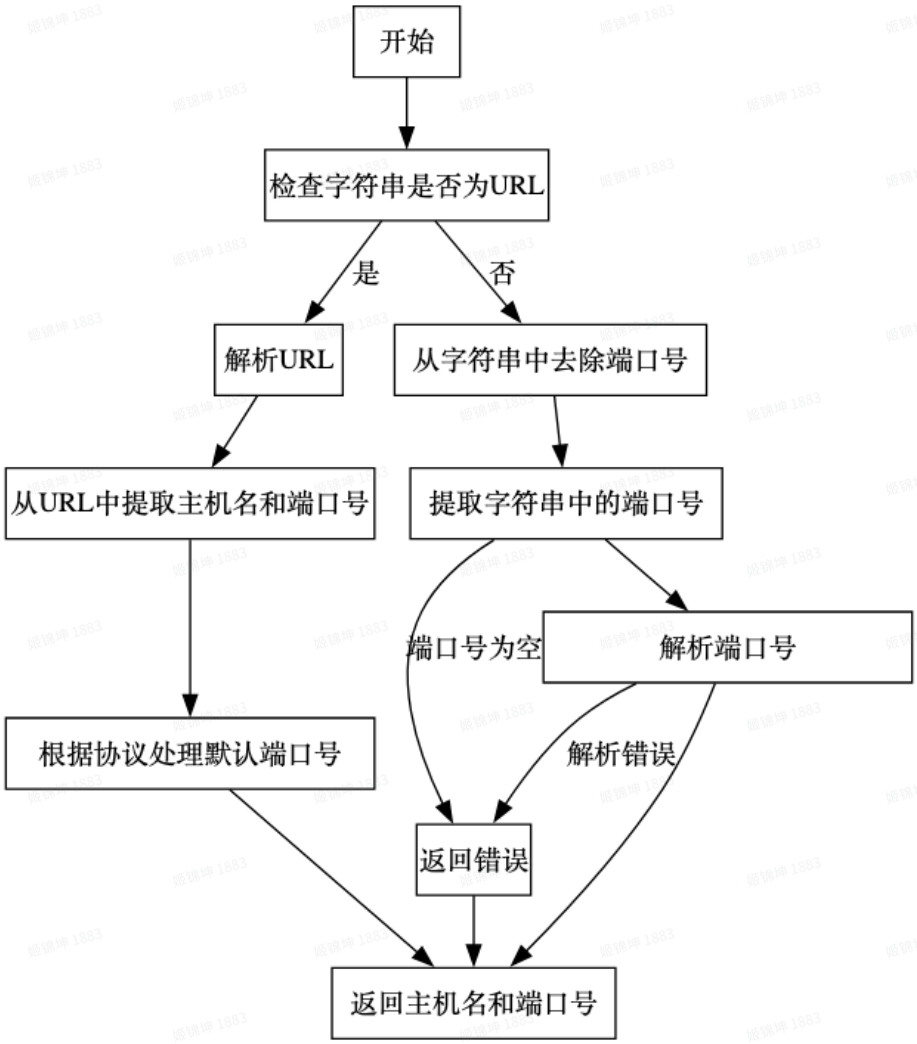
从单个目标中提取主机和端口

`ParseStringToHostPort` 函数的功能是从字符串中解析出主机名和端口号，并将其与可能的错误一起返回。根据输入的字符串，函数会尝试从中提取主机名和端口号，并根据不同的情况进行处理。

函数的描述如下：

- 如果输入字符串包含 "://"，则将其视为 URL，并解析出主机名和端口号。如果端口号无法解析或小于等于0，则根据 URL 的 scheme 设置默认端口号。
- 如果输入字符串不包含 "://"，则将其视为主机名和端口号的组合。函数会提取主机名和端口号，并对端口号进行解析。

函数的流程图如下所示



函数的示例：

```
host, port, err = str.ParseStringToHostPort("example.com:80")
// host: example.com
// port: 80
// err: nil

host, port, err = str.ParseStringToHostPort("example.com")
// host: example.com
// port: 0
// err: (*errors.fundamental)(0xc002431458)(unknown port for [example.com])

host, port, err = str.ParseStringToHostPort("https://example.com")
// host: example.com
// port: 443
// err: nil
```

解析为端口号列表

`ParseStringToPorts` 函数的功能是将字符串解析为端口号列表。该函数可以处理逗号分隔的端口号，并解析以连字符分隔的范围。

函数的描述如下：

- 函数接受一个字符串作为输入，该字符串表示端口号。端口号可以使用逗号分隔，并且可以使用连字符表示范围。
- 函数会解析输入字符串，并将解析后的端口号存储在一个整数列表中。
- 函数会根据需要进行排序，并返回最终的端口号列表。

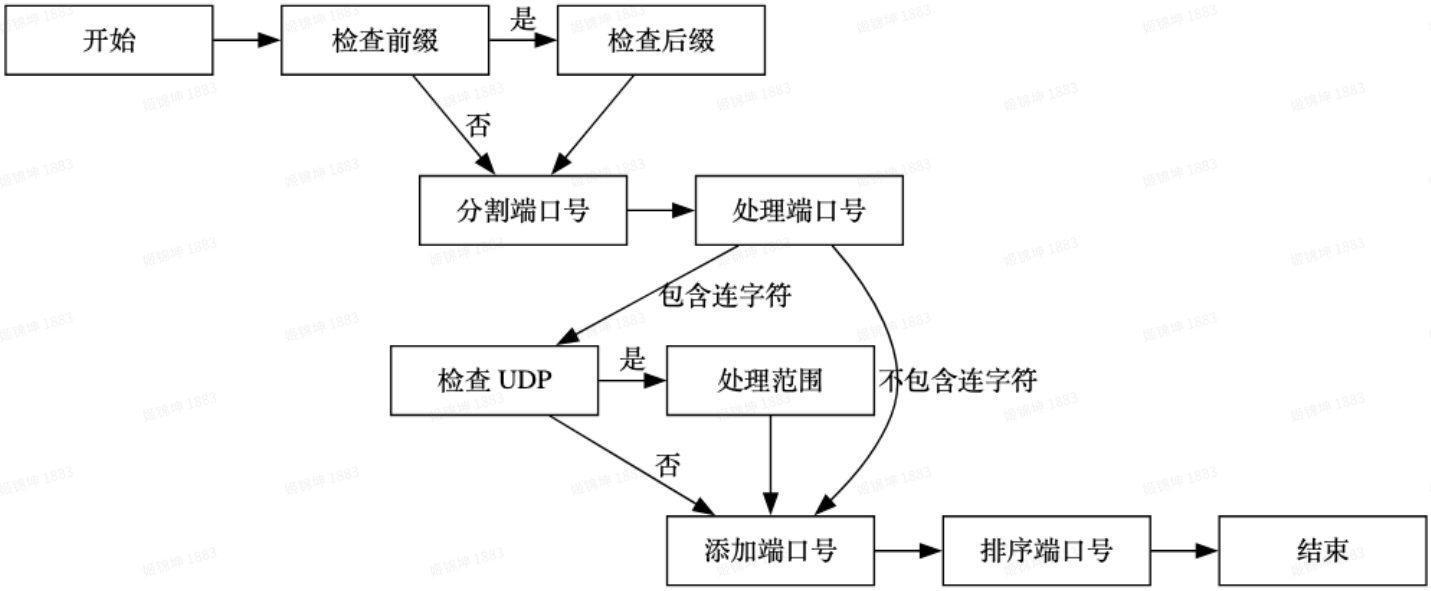
函数的示例：

- `ParseStringToPorts("10086-10088,23333")` 返回 `[10086, 10087, 10088, 23333]`。

函数的流程如下：

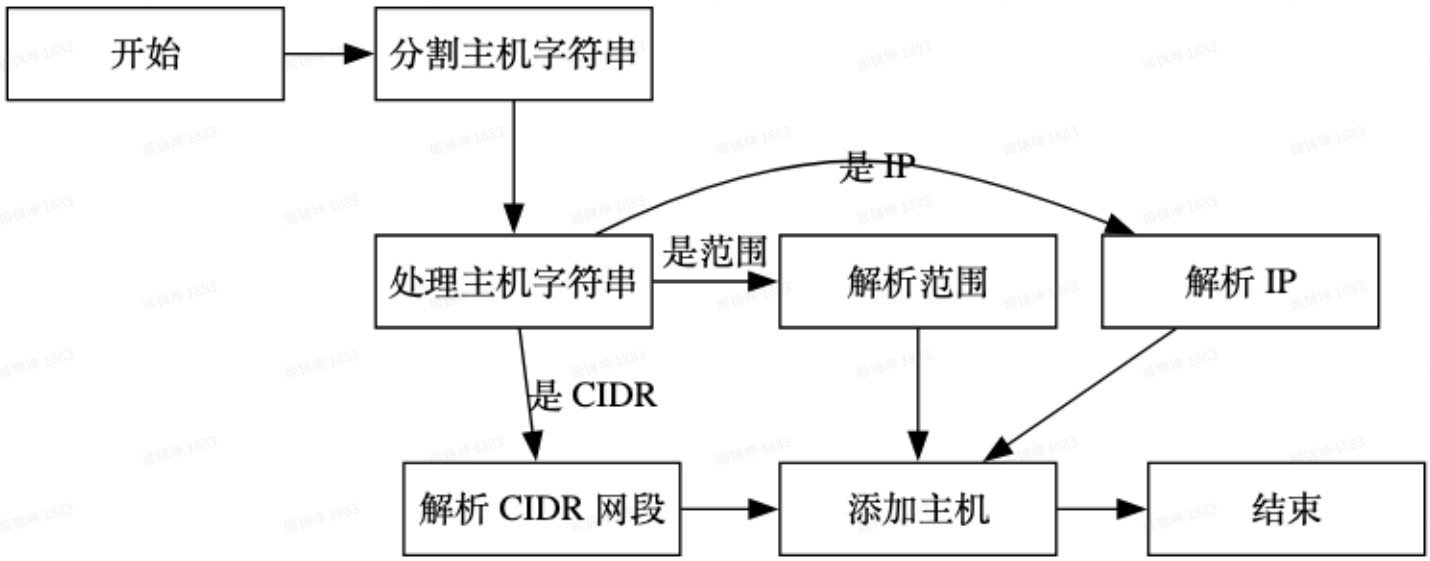
1. 首先，函数会检查输入字符串是否以连字符开头，如果是，则在字符串前面添加 "1"，以处理范围的起始端口号。
2. 接下来，函数会检查输入字符串是否以连字符结尾，如果是，则在字符串末尾添加 "65535"，以处理范围的结束端口号。
3. 然后，函数会按逗号分隔输入字符串，得到一个字符串数组，每个字符串表示一个端口号或范围。
4. 对于每个字符串，函数会进行如下处理：
    - 去除首尾的空格。
    - 检查是否包含 "U:"，如果包含，则表示该端口号使用 UDP 协议，需要将其从字符串中移除，并将协议设置为 "udp"。
    - 检查字符串是否包含连字符，如果包含，则表示该端口号为范围。函数会解析范围的起始端口号和结束端口号，并将它们之间的所有端口号添加到列表中。如果范围起始端口号大于结束端口号，则忽略该范围。
    - 如果字符串不包含连字符，则将其解析为单个端口号，并添加到列表中。
5. 最后，函数会对端口号列表进行排序，并返回最终的结果。

使用流程图表示这个解析过程如下：



解析主机与CIDR拆分

函数 `ParseStringToHosts` 用于将字符串解析为主机列表。主机字符串可以使用逗号分隔，并且可以解析CIDR网段。函数首先将原始字符串使用逗号分割成多个主机字符串。然后，对每个主机字符串进行处理。如果主机字符串可以解析为IP，则将其解析为IP并添加到主机列表中。如果主机字符串是CIDR网段，则将其解析为网段，并将网段中的所有IP添加到主机列表中。如果主机字符串是范围（如1.1.1.1-3），则解析范围并将范围内的所有IP添加到主机列表中。最后，返回过滤掉空字符串的主机列表。上述描述过程的流程图如下：



读者可以从这个示例快速学习这个重要函数的使用：

```
1 str.ParseStringToHosts("192.168.0.1/32,127.0.0.1") // 返回 ["192.168.0.1",
  "127.0.0.1"]
```

综合数据提取

函数名	说明	输入	输出	案例
ExtractHost	从URL中提取主机名	URL字符串	主机名字符串	`str.ExtractHost("http://www.example.com/path")` 返回 `"www.example.com"`
ExtractDomain	从URL或电子邮件地址中提取域名	URL或电子邮件地址字符串	域名字符串	`str.ExtractDomain("user@example.com")` 返回 `"example.com"`
ExtractRootDomain	从域名中提取根域名	域名字符串	根域名字符串	`str.ExtractRootDomain("subdomain.example.com")` 返回 `["example.com"]`
ExtractJson	解析和修复嵌套的 JSON 结构	数据源	被提取出的JSON字符串列表	`str.ExtractJson("abc{\"cc\":111}\"aaaaa")`
ExtractJsonWithRaw	解析和修复嵌套的 JSON 结构（原始数据，可能不符合标准）	数据源	被提取出的JSON字符串列表	`str.ExtractJsonWithRaw("abc{\"cc\":111}\"aaaaa")`

JSON提取技术

`str.ExtractJson` 和 `str.ExtractJsonWithRaw` 提取技术结合了堆栈驱动的状态机来解析和修复嵌套的 JSON 结构，通过逐字节扫描和字符状态跟踪，能够从杂乱的数据流中提取有效的 JSON 对象，并处理字符串转义序列，确保提取的 JSON 符合标准格式。

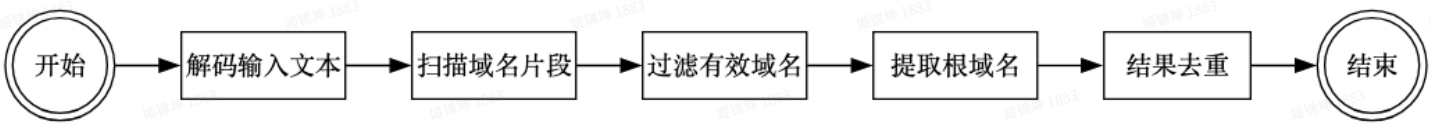
以下是一个经典案例：

```
data = `<html>

aasdfasd
df
{
  "code" : "0",
  "message" : "success",
  "responseTime" : 2,
  "traceId" : "a469b12c7d7aaca5",
  "returnCode" : null,
  "result" : {
    "total" : 0,
    "navigatePages" : 8,
    "navigatepageNums" : [ ],
    "navigateFirstPage" : 0,
```

```
    "navigateLastPage" : 0
  }
}

</html>`
for result in str.ExtractJson(data) {
    dump([]byte(result))
}

/*
OUT:

([]uint8) (len=270 cap=288) {
 00000000  7b 0a 20 20 22 63 6f 64  65 22 20 3a 20 22 30 22  |{.  "code" : "0"|
 00000010  2c 0a 20 20 22 6d 65 73  73 61 67 65 22 20 3a 20  |,.  "message" : |
 00000020  22 73 75 63 63 65 73 73  22 2c 0a 20 20 22 72 65  |"success",.  "re|
 00000030  73 70 6f 6e 73 65 54 69  6d 65 22 20 3a 20 32 2c  |sponseTime" : 2,|
 00000040  0a 20 20 22 74 72 61 63  65 49 64 22 20 3a 20 22  |.  "traceId" : "|
 00000050  61 34 36 39 62 31 32 63  37 64 37 61 61 63 61 35  |a469b12c7d7aaca5|
 00000060  22 2c 0a 20 20 22 72 65  74 75 72 6e 43 6f 64 65  |",.  "returnCode|
 00000070  22 20 3a 20 6e 75 6c 6c  2c 0a 20 20 22 72 65 73  |" : null,.  "res|
 00000080  75 6c 74 22 20 3a 20 7b  0a 20 20 20 20 22 74 6f  |ult" : {.    "to|
 00000090  74 61 6c 22 20 3a 20 30  2c 0a 20 20 20 20 22 6e  |tal" : 0,.    "n|
 000000a0  61 76 69 67 61 74 65 50  61 67 65 73 22 20 3a 20  |avigatePages" : |
 000000b0  38 2c 0a 20 20 20 20 22  6e 61 76 69 67 61 74 65  |8,.    "navigate|
 000000c0  70 61 67 65 4e 75 6d 73  22 20 3a 20 5b 20 5d 2c  |pageNums" : [ ],|
 000000d0  0a 20 20 20 20 22 6e 61  76 69 67 61 74 65 46 69  |.    "navigateFi|
 000000e0  72 73 74 50 61 67 65 22  20 3a 20 30 2c 0a 20 20  |rstPage" : 0,.  |
 000000f0  20 20 22 6e 61 76 69 67  61 74 65 4c 61 73 74 50  |  "navigateLastP|
 00000100  61 67 65 22 20 3a 20 30  0a 20 20 7d 0a 7d        |age" : 0.  }.}|
}
*/
```

域名提取技术

`str.ExtractDomain` 域名提取技术的核心原理是通过一系列正则表达式和过滤器来扫描和识别文本中的有效域名。它处理多种编码形式，包括百分比编码、Unicode、HTML实体等，首先对文本进行解码。然后逐字节扫描文本，收集可能的域名片段。这些片段基于字符有效性（字母、数字、连字符）和点号分隔进行组合。通过与预定义的顶级域名列表和黑名单词汇进行匹配，确定哪些片段构成有效的域名。最后，它提取和返回根域名，并去除重复项。



我们用一个简单的案例来解释这个过程，以下是一段从“HTTP响应数据包”中提取域名的代码：

```
domains = str.ExtractDomain(`HTTP/1.1 200 OK
Accept-Ranges: bytes
Cache-Control: max-age=604800
Content-Type: text/html; charset=utf-8
Date: Tue, 14 Nov 2023 03:00:22 GMT
Etag: "3147526947"
Expires: Tue, 21 Nov 2023 03:00:22 GMT
Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT
Server: EOS (vny/044F)
Content-Length: 1256

<!doctype html>
<html>
<head>
...
<body>
<div>
    <h1>Example Domain</h1>
    <p>This domain is for use in illustrative examples in documents. You may use this
    domain in literature without prior coordination or asking for permission.</p>
    <p><a href="https://www.iana.org/domains/example">More information...</a></p>
</div>
</body>
</html>
`)
dump(domains)

/*
OUTPUT:

([]string) (len=2 cap=2) {
 (string) (len=8) "iana.org",
 (string) (len=12) "www.iana.org"
}
*/
```

这个示例代码展示了如何使用 `str.ExtractDomain` 函数从字符串中提取域名。实际应用中，您可以根据需要将其集成到您的代码中，并根据具体的字符串格式和提取规则进行相应的调整。

版本比较函数

安全领域中版本合规中进行版本比较非常重要，它可以用于判断软件、库或系统的版本是否符合要求，从而进行相应的安全性评估和决策。在Yak语言的字符串处理函数中，提供了一系列工具可以帮助读者进行版本比较：

函数名	说明	输入输出描述	使用案例
`VersionGreater`	比较版本号v1是否大于v2	输入两个版本号字符串v1和v2，返回一个布尔值表示v1是否大于v2	`str.VersionGreater("1.0.0", "0.9.9")` 返回 `true`
`VersionGreaterEqual`	比较版本号v1是否大于等于v2	输入两个版本号字符串v1和v2，返回一个布尔值表示v1是否大于等于v2	`str.VersionGreaterEqual("3.0", "3.0")` 返回 `true`
`VersionEqual`	比较版本号v1是否等于v2	输入两个版本号字符串v1和v2，返回一个布尔值表示v1是否等于v2	`str.VersionEqual("3.0", "3.0")` 返回 `true`
`VersionLessEqual`	比较版本号v1是否小于等于v2	输入两个版本号字符串v1和v2，返回一个布尔值表示v1是否小于等于v2	`str.VersionLessEqual("0.9.9", "1.0.0")` 返回 `true`
`VersionLess`	比较版本号v1是否小于v2	输入两个版本号字符串v1和v2，返回一个布尔值表示v1是否小于v2	`str.VersionLess("0.9.9", "1.0.0")` 返回 `true`

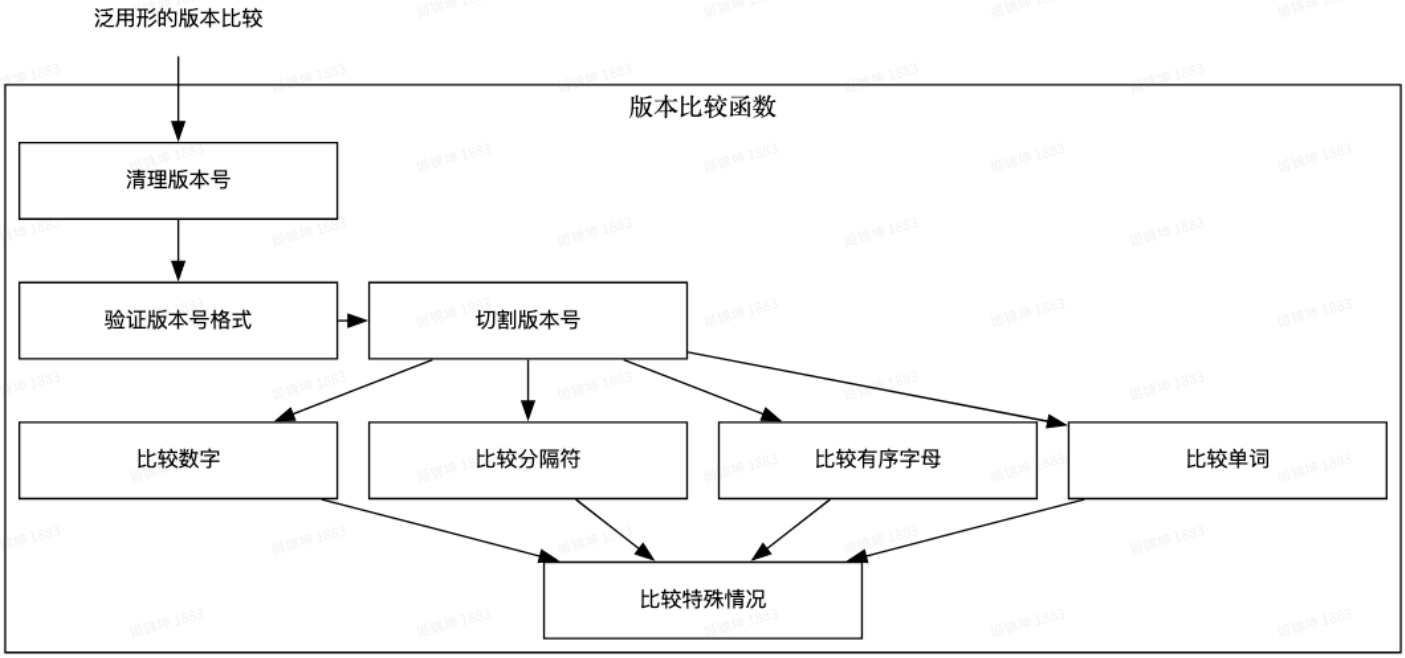
版本比较是一个非常复杂的过程，上述工具函数的底层是一个 `str.CompareVersion` 函数，这个函数被用于进行通用形式的版本比较。它接受两个字符串类型的参数 `v1` 和 `v2` ，并返回比较结果和可能的错误。

该函数的执行过程如下：

1. 清理版本号字符串，去除多余的空格和特殊字符。
2. 验证版本号是否符合标准，主要检查是否有空格或其他非法字符。
3. 切割版本号字符串，将其拆分为多个部分，例如数字、分隔符、字母等。
4. 比较两个版本号的部分，按照一定的规则进行比较。根据部分的类型，可能会调用不同的比较函数。
5. 如果部分类型不匹配，或者无法比较，返回错误信息。
6. 如果所有部分都相等，则返回0，表示两个版本号相等。
7. 如果有部分不相等，根据比较结果返回1或-1，表示其中一个版本号大于或小于另一个版本号。

8. 如果比较过程中出现错误，返回-2和相应的错误信息。

笔者把上述函数执行过程描述成流程图，读者可直接根据图示理解这个过程：



笔者在上述的小节中展示了部分常用的Yak语言中的库函数，实际Yak语言还有很多很方便的处理函数，用户可以在官方网站中找到这部分内容。

7.3 文件与操作系统工具函数

在本章节中，我们将深入探讨文件和操作系统工具函数的重要性和使用方法。文件和操作系统工具函数是编程中不可或缺的一部分，它们提供了处理文件和操作系统相关任务的功能模块和函数。

首先，我们将介绍"file"模块。该模块提供了一系列用于文件操作的函数，包括文件的创建、打开、读取、写入和关闭等。我们将学习如何使用这些函数来处理文件，从而实现文件的读取和写入，以及对文件进行其他常见操作。

接下来，我们将探讨"os"模块。该模块提供了与操作系统交互的函数，使读者能够执行与操作系统相关的任务，如获取当前工作目录、执行系统命令、创建和删除目录等。我们将学习如何使用"os"模块来管理操作系统级别的任务，并了解如何与操作系统进行交互。

通过学习文件和操作系统工具函数，读者将能够更好地理解和掌握文件和操作系统相关的编程任务。这将使读者能够更高效地处理文件操作、系统管理和与操作系统的交互，提高编程能力和效率。

在本章节中，我们将提供详细的示例代码和实践，以帮助读者深入理解和应用文件和操作系统工具函数。无论读者是初学者还是有一定经验的开发者，本章节都将为其提供有关文件和操作系统工具函数的全面指导，让读者能够在实际项目中灵活运用它们。

7.3.1 文件操作

基础概念

在开始学习文件操作之前，了解一些基础概念是非常重要的。这些概念将帮助您理解如何在计算机上存储和访问数据。

文件和文件系统的基本概念

文件是存储在计算机存储设备上的数据的集合。这些数据可以是文本、图片、音频、视频等。文件被组织在文件系统中，文件系统是操作系统用来控制如何在存储设备上存储数据和访问数据的一种方法。文件系统管理着文件的创建、删除、读取、写入等操作。

文件通常有两个关键属性：文件名和路径。文件名是指定给文件的标识符，而路径描述了在文件系统中找到该文件的位置。

文件路径的理解（绝对路径与相对路径）

- **绝对路径**：是从文件系统的根目录（在 Unix-like 系统中是 `/`，在 Windows 中是 `C:\` 或其他驱动器字母）开始的完整路径。它指向文件系统中的一个具体位置，不管当前工作目录是什么。例如，`/home/user/documents/report.txt` 或 `C:\Users\user\documents\report.txt`。
- **相对路径**：相对于当前工作目录的路径。它不是从根目录开始的，而是从当前所在位置开始。例如，如果当前工作目录是 `/home/user`，那么相对路径 `documents/report.txt` 指向的是 `/home/user/documents/report.txt`。

了解这两种路径的区别对于正确地访问文件非常重要。

文件类型（文本文件和二进制文件）

- **文本文件**：存储的是可以用标准文本编辑器阅读的字符（如字母、数字和符号）。文本文件通常存储编程代码或者是普通文档，并且它们的内容是人类可读的。例如，`.txt`、`.py`、`.html` 等文件扩展名通常表示文本文件。
- **二进制文件**：包含了编码后的数据，只能通过特定的程序或编辑器来解释和读取。它们不是为了人类直接阅读而设计的。图像、音频、视频文件以及可执行程序都是二进制文件的例子，如 `.jpg`、`.mp3`、`.exe` 等。

理解这两种文件类型有助于确定如何使用工具和程序来处理不同的数据。例如，文本编辑器可能无法正确显示二进制文件的内容，而图像查看器则不能用来打开文本文件。

通过掌握这些基础知识，您将能够更好地理解接下来的章节，其中将介绍如何使用编程语言来执行文件操作。

基本文件操作工具函数

快速文件操作

在本章节中，笔者将引导读者通过几个简单的例子，学习如何在Yak语言中进行文件的基本操作。这些操作包括保存文本到文件、读取文件内容、创建目录（文件夹）、以JSON格式保存数据，以及处理错

误。

- **保存文本到文件**

想要保存一段文本到文件中，可以使用 `file.Save` 函数。例如，要保存 `Hello World` 到 `test.txt` 文件中，可以这样做：

```
err = file.Save("test.txt",
Hello World
)
if err != nil { die(err) }
```

如果操作成功， `test.txt` 文件将包含文本 `Hello World` 。如果操作失败，比如因为磁盘空间不足或没有写权限， `die` 函数将被调用，程序将终止并报告错误。

- **读取文件内容**

读取文件内容同样简单。笔者使用 `file.ReadFile` 函数来读取 `test.txt` 文件的内容：

```
data = file.ReadFile("test.txt")~
dump(data)
```

`~` 符号是Yak语言中的错误传播操作符，它会在发生错误时自动终止当前操作。 `dump` 函数将打印出读取到的数据，如下所示：

```
([]uint8) (len=11 cap=512) {
 00000000  48 65 6c 6c 6f 20 57 6f  72 6c 64                 |Hello World|
}
```

这显示了文件内容以及其在内存中的字节表示。

- **创建文件目录与保存JSON数据**

创建目录（文件夹）并保存JSON格式的数据也是一件轻而易举的事。首先，使用 `file.MkdirAll` 函数来创建一个目录（文件夹）路径：

```
file.MkdirAll("yak/file/op")~
```

然后，可以使用 `file.SaveJson` 函数来保存JSON数据到文件中：

```
file.SaveJson("yak/file/op/test.txt", {"Hello": 1, "World": 2})~
```

接下来，再次使用 `file.ReadFile` 来验证数据是否正确保存：

```
dump(file.ReadFile(yak/file/op/test.txt)~)
```

输出将展示保存的JSON数据：

```
([]uint8) (len=21 cap=512) {
 00000000  7b 22 48 65 6c 6c 6f 22  3a 31 2c 22 57 6f 72 6c  |{"Hello":1,"Worl|
 00000010  64 22 3a 32 7d                                    |d":2}|
}
```

- **处理文件操作错误**

在文件操作中，错误处理是不可或缺的。例如，尝试从一个不存在的文件中读取内容将导致错误：

```
data, err = file.ReadFile("no-existed-file.txt")
if err != nil { die(err) }
```

如果文件不存在， `die` 函数将报告错误并停止程序执行。

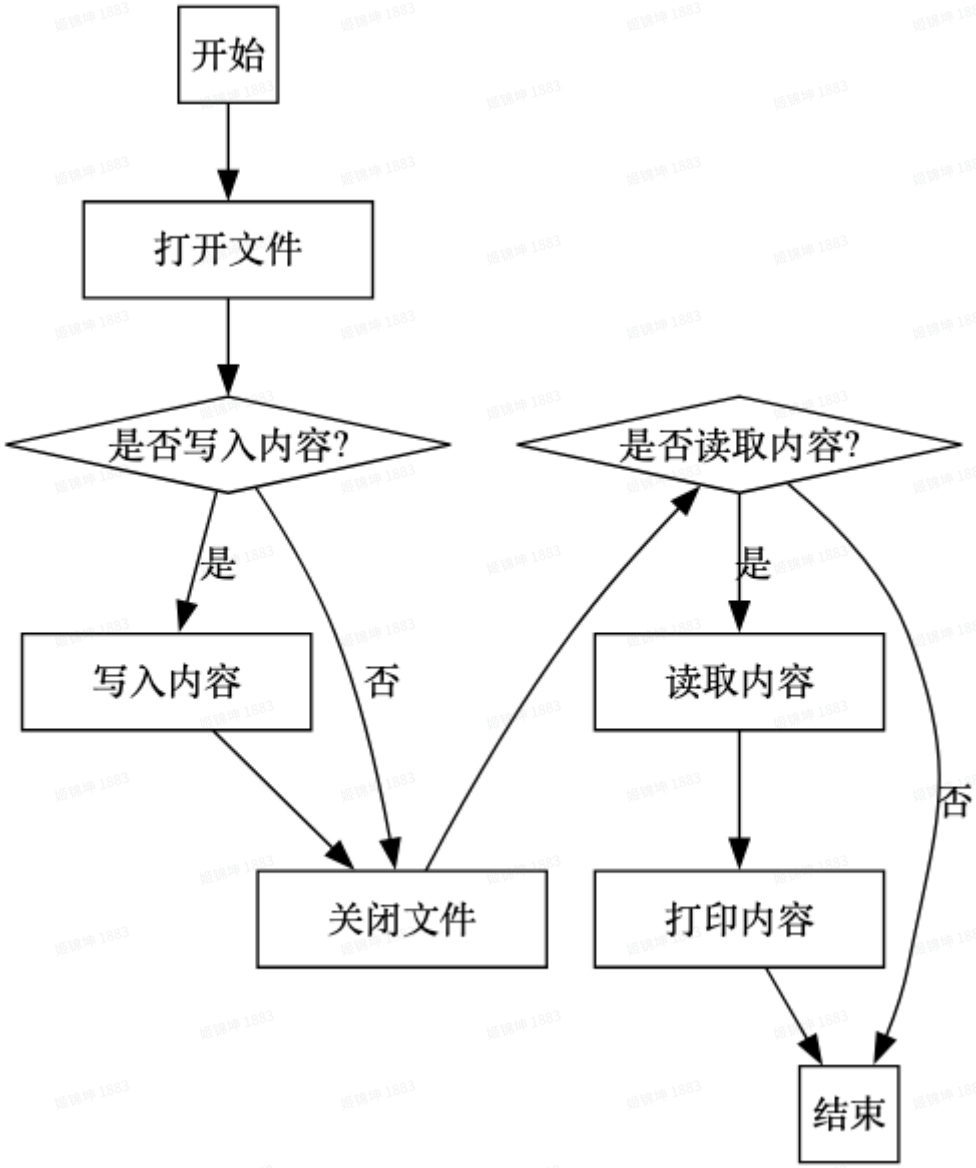
```
ERROR: open no-existed-file.txt: no such file or directory
```

通过上述示例，读者应该能够掌握Yak语言中的基本文件操作。记住，正确的错误处理能够帮助您的程序更加健壮和可靠。

标准文件操作

- **标准文件操作**

在本章节，笔者将展示如何在Yak语言中使用标准文件操作。这些操作包括打开文件、写入内容以及确保文件在操作完成后能够正确关闭。按照以下步骤，读者可以轻松地进行文件的基本读写操作。用户可以根据下面流程图了解标准文件操作的基本流程。



- **打开文件并写入内容**

要将内容写入文件，首先需要打开文件。在Yak语言中，`file.OpenFile`函数用于这个目的。以下是如何使用该函数打开文件并设置相应的权限：

```
1 f = file.OpenFile("/tmp/test.txt", file.O_CREATE|file.O_RDWR, 0o777)~
```

在这里，`file.O_CREATE`标志指示如果文件不存在则创建文件，`file.O_RDWR`标志允许读写文件。`0o777`则设置了文件的权限，使得所有用户都可以读写和执行文件。

成功打开文件后，可以使用`WriteLine`方法来写入一行文本：

```
1 f.WriteLine("Hello World")
```

写入完成后，不要忘记关闭文件。这可以通过`Close`方法来完成：

```
f.Close()
```

关闭文件是一个好习惯，它可以释放系统资源，并确保所有的数据都被正确写入磁盘。

- **快捷打开文件**

除了使用上述代码打开文件描述符之外，我们还提供了一个省略Flags和权限控制参数的快捷打开函数：

```
fp = file.Open("pathtofile.txt")
defer fp.Close()

/*
fp = os.OpenFile("pathtofile.txt", os.O_CREATE|os.O_RDWR, 0o777)
defer fp.Close()
*/
```

`file.Open` 函数在内部是设置了 `os.O_CREATE` 和 `os.O_RDWR` 标志，并且指定了默认的文件权限为 `0o777`。这意味着每次调用 `file.Open` 函数时，如果文件不存在，则会创建该文件，并且文件是以读写模式打开的。同时，由于指定了 `0o777` 权限，创建的文件将对所有用户开放所有权限（读、写、执行）。

> 需要注意的是，将文件权限设置为 `0o777` 相对来说比较宽泛，它允许任何用户对文件进行读写操作。通常来说，如果你希望更保守一些，可以设置一些如 `0o644`（所有用户可读，只有所有者可写）或 `0o600`（只有所有者可读写），会减少一些“风险”。

此外，`defer fp.Close()` 确保了文件在函数结束时会被关闭，这是一个防止资源泄露的好习惯。在并发环境或者有异常处理需求的场景下，确保打开的文件被正确关闭是非常重要的。

- **读取文件内容**

要验证写入操作，可以读取并打印文件内容：

```
content = file.ReadFile("/tmp/test.txt")~
dump(content)
```

`ReadFile` 函数读取指定文件的全部内容，`dump` 函数则用来打印这些内容，让读者可以看到文件中实际存储的数据。

- **使用defer确保文件关闭**

在某些情况下，可能会有多个退出点，这时候确保文件在函数结束时被关闭就显得尤为重要。Yak语言提供了 `defer` 关键字，可以用来保证在函数返回前执行某个操作，例如关闭文件：

```
f = file.OpenFile("/tmp/test2.txt", file.O_CREATE|file.O_RDWR, 0o777)~
defer f.Close()
```

使用 `defer` 关键字后，无论函数是正常结束还是由于错误而提前返回，都会执行 `f.Close()` 来关闭文件。这是一种优雅且有效的资源管理方式。

通过实验上述步骤，读者应该能够理解并执行Yak语言中的标准文件操作。正确地管理文件资源并确保操作的正确性，是编写高质量代码的关键所在。

- **文件打开的选项**

文件操作的函数通常提供了一组标志（flags）来指定在打开文件时期望的行为。这些标志可以通过位运算符（如 `|`，在Yak语言中表示按位或操作）组合使用。下面是Yak语言提供的标志列表中每个标志的含义：

1. `O_RDWR`：读写模式打开文件。
2. `O_CREATE`：如果文件不存在，则创建文件。
3. `O_APPEND`：写操作会将数据追加到文件末尾。
4. `O_EXCL`：与 `O_CREATE` 一起使用时，如果文件已存在，则会导致打开文件失败。
5. `O_RDONLY`：只读模式打开文件。
6. `O_SYNC`：使每次写入等到物理I/O操作完成，包括由该写入操作引起的文件属性更新。
7. `O_TRUNC`：如果文件已存在并且为写操作打开，则将其长度截断为0。
8. `O_WRONLY`：只写模式打开文件。

当你打开一个文件时，可以根据需要选择适当的标志。例如：

- 如果你想要打开一个文件用于读写，并且如果该文件不存在则创建它，你可以使用 `O_RDWR | O_CREATE`。
- 如果你想要打开一个文件用于追加内容，不管它是否存在，你可以使用 `O_APPEND`。
- 如果你想确保在创建文件时不会覆盖已有的文件，你可以使用 `O_CREATE | O_EXCL`。

在上面提供的代码示例中：

```
f = file.OpenFile("/tmp/test2.txt", file.O_CREATE|file.O_RDWR, 0o777)
defer f.Close()
```

这个调用将尝试以读写模式打开 `/tmp/test2.txt` 文件，如果该文件不存在则创建它，文件权限设置为 777（在Unix系统中意味着任何用户都有读、写和执行权限）。使用 `defer` 关键字来确保文件最终会被关闭，这是Go语言中的一种惯用法，用于确保资源的清理。

文件系统目录操作

- **使用 `file.MkdirAll` 和 `file.Mkdir` 来创建文件目录**

虽然读者在前文中已经见过这两个函数了，但是我们还是要在这里简单介绍一下。在文件系统中，目录（或文件夹）是用来组织文件的一种方式。创建目录是一个基本操作，它允许您在文件系统中构建一个结构化的存储模型。

使用 `file.MkdirAll` 函数可以创建一个新的目录。如果目录的上级目录还不存在，`MkdirAll` 也会创建必要的上级目录。这是一个递归创建的过程，确保了指定路径的所有组成部分都将被创建。

```
file.MkdirAll(pathName)
```

在上述代码中，`pathName` 表示要创建的目录的路径。这个函数会检查路径是否已经存在，如果不存在，它会创建路径中的所有目录。与之相对的是 `file.Mkdir(pathName)`，如果创建路径的时候，上级目录中有不存在的内容，他就不会创建成功。

读者可以跟随这一段代码来学习本小节的操作：

```
for fileName in [
    "/tmp/yak/1.txt",
    "/tmp/yak/a/2.txt",
    "/tmp/yak/other.txt",
    "/tmp/yak/aaa.txt",
] {
    pathName, name = file.Split(fileName)
    file.MkdirAll(pathName)

    file.Save(fileName, "Hello Yak File Operator")
}

for element in file.Dir("/tmp/yak/a") {
    println(element.Path)
}

/*
OUTPUT:

/tmp/yak/a/2.txt
*/

file.Walk("/tmp/yak", info => {println(f`${info.IsDir ? "dir ": "file"}\t${info.Path}`); return true})~
/*
OUTPUT:

```

```
file    /tmp/yak/1.txt
file    /tmp/yak/a/2.txt
dir     /tmp/yak/a
file    /tmp/yak/aaa.txt
file    /tmp/yak/other.txt
*/
```

- **目录遍历**

目录遍历是指检查目录中的所有文件和子目录。这在需要处理目录中的所有元素时非常有用。

```
for element in file.Dir("/tmp/yak/a") {
    println(element.Path)
}
```

`file.Dir` 函数返回指定目录下的所有文件和子目录。在循环中，`element.Path` 会输出每个元素的路径。在上述代码中，`element` 是一个内置结构体，读者可以阅读以下说明来了解这个结构体中可用的内容：

字段/方法名	类型	使用说明
BuildIn	`fs.FileInfo`	内置的 `fs.FileInfo` 结构体实例，包含文件的基础信息。
Path	`string`	文件的完整路径。
Name	`string`	文件的名称。
IsDir	`bool`	表示该文件信息是否代表一个目录。
IsDir()	`func() bool`	方法，返回一个布尔值，指示文件是否为目录。
ModTime()	`func() time.Time`	方法，返回文件的修改时间。
Mode()	`func() fs.FileMode`	方法，返回文件的模式和权限。
Name()	`func() string`	方法，返回文件的名称（不包括路径）。
Size()	`func() int64`	方法，返回文件的大小，单位为字节。
Sys()	`func() interface{}`	方法，返回底层数据源（如，文件系统信息）的接口值。

目录和文件遍历

更复杂的操作是递归遍历一个目录及其所有子目录中的文件。这可以用 `file.Walk` 函数实现。

```
file.Walk("/tmp/yak", info => {
    println(f`${info.IsDir ? "dir ": "file"}\t${info.Path}`)
    return true
})
```

`file.Walk` 函数接受一个目录路径和一个回调函数。对于目录中的每个文件和子目录，回调函数都会被调用一次。回调函数的参数 `info` 包含了当前遍历到的文件或目录的信息，例如是否是目录（`info.IsDir`）和路径（`info.Path`）。回调函数返回 `true` 来继续遍历，或者返回 `false` 来停止。这个函数中 `info` 是之前我们提到的结构体，各种操作都是通用的。

通过这些操作，可以在文件系统中创建目录结构，写入文件，并遍历目录中的内容。掌握这些基础技能对于进行更复杂的文件操作任务至关重要。

文件路径操作

除了提供文件的基础操作外，用户还可以在Yak语言中“操作”路径字符串。这个特性十分有用：

它允许程序员在不实际访问文件系统的情况下处理文件和目录的路径。这种操作通常包括：

- 合并路径：将多个路径片段组合成一个完整的文件路径。
- 分割路径：将文件路径分割成目录路径和文件名。
- 提取路径组成部分：如目录名、文件名、扩展名。

在Yak语言中，路径操作可能包含以下功能：

1. **路径合并**（`file.Join` 或类似函数）：将多个字符串参数合并成一个路径，确保正确使用目录分隔符。
2. **路径分割**（`file.Split` 或类似函数）：将路径字符串分割成目录部分和文件部分。
3. **路径规范化**（`file.Clean` 或类似函数）：简化路径，解析路径中的 `.`、`..` 和多余的分隔符等。
4. **获取文件名**（`file.GetBase` 或类似函数）：从路径中提取文件名。
5. **获取目录名**（`file.GetDirPath` 或类似函数）：从路径中提取目录路径。
6. **检查路径是否为绝对路径**（`file.IsAbs` 或类似函数）：判断给定的路径字符串是否是一个绝对路径。
7. **提取文件扩展名**（`file.GetExt` 或类似函数）：从文件名中提取扩展名。

这些操作通常不需要访问实际的文件系统，但它们对于路径字符串的处理至关重要，可以帮助避免许多常见的错误，如路径格式错误或不正确的路径分隔符使用。通过这些工具，用户可以更安全、更有效地编写代码来处理文件和目录路径。用户可以跟随下面实例代码进行操作：

```
```

```
// 获取文件扩展名
println(file.GetExt("file.txt"))           // 输出: .txt
println(file.GetExt("/tmp/a.txt"))         // 输出: .txt

// 获取文件所在目录路径
println(file.GetDirPath("file/aaa"))       // 输出: file/
println(file.GetDirPath("/tmp/a.txt"))     // 输出: /tmp/

// 获取文件基础名
println(file.GetBase("tmp/1.txt"))         // 输出: 1.txt
println(file.GetBase("/tmp/1.txt"))        // 输出: 1.txt

// 路径规范化
println(file.Clean("/tmp/1.txt"))          // 输出: /tmp/1.txt
println(file.Clean("tmp/../tmp/1.txt"))    // 输出: tmp/1.txt

// 路径分割
dir, filename = file.Split("tmp/1.txt")
println(dir, filename)                     // 输出: tmp/ 1.txt

dir, filename = file.Split("1.txt")
println(dir, filename)                     // 输出:  1.txt

dir, filename = file.Split("/tmp/1.txt")
println(dir, filename)                     // 输出: /tmp/ 1.txt

// 路径合并
println(file.Join("tmp", "1.txt"))         // 输出: tmp/1.txt
println(file.Join("/tmp", "1.txt"))        // 输出: /tmp/1.txt
println(file.Join("tmp", "a", "1.txt"))    // 输出: tmp/a/1.txt
```

7.3.2 系统操作

系统信息与基础操作

函数名/变量名	使用说明	代码案例
Remove	用于删除指定的文件或目录。如果删除的是目录，将递归删除目录及其下的所有内容。	`os.Remove("file.txt")`
RemoveAll	用于删除指定的目录及其下的所有内容。	`os.RemoveAll("dir")`

Rename	用于将文件或目录重命名。	`os.Rename("old.txt", "new.txt")`
Getwd	获取当前工作目录，返回值为当前工作目录和错误，如果获取失败，则错误不为空	`os.Getwd()`
Chdir	用于改变当前工作目录。	`os.Chdir("dir")`
Chmod	用于修改指定文件的权限模式（类Unix系统适用）。	`os.Chmod("file.txt", 0644)`
Chown	用于修改指定文件的所有者和所属组（类Unix系统适用）。	`os.Chown("file.txt", uid, gid)`
OS	返回当前操作系统的名称。跟随用户实际系统，macOS为 `darwin`，Windows 为 `windows`，Linux为 `linux`	`println(os.OS)`
ARCH	返回当前系统架构的名称。常见的为 `amd64` 和 `arm64` 的值	`println(os.ARCH)`
Executable	返回当前可执行文件的路径（可能会返回错误）。	`execPath, err = os.Executable()`
Getpid	返回当前进程的进程ID。	`os.Getpid()`
Getppid	返回当前进程的父进程ID（类Unix系统适用）。	`os.Getppid()`
Getuid	返回当前用户的用户ID（类Unix系统生效）。	`os.Getuid()`
Geteuid	返回当前用户的有效用户ID（类Unix系统适用）。	`os.Geteuid()`
Getgid	返回当前用户的组ID（类Unix系统适用）。	`os.Getgid()`
Getegid	返回当前用户的有效组ID（类Unix系统适用）。	`os.Getegid()`
Environ	返回当前环境变量的键值对。	`os.Environ()`
Hostname	返回当前主机的主机名。	`os.Hostname()`

环境变量操作函数

函数名/变量名	使用说明	代码案例
Unsetenv	用于删除指定的环境变量。	`os.Unsetenv("KEY")`
LookupEnv	用于获取指定的环境变量的值。	`os.LookupEnv("KEY")`
Clearenv	用于清空当前的环境变量。	`os.Clearenv()`
Setenv	用于设置指定的环境变量的值。	`os.Setenv("KEY", "VALUE")`
Getenv	用于获取指定的环境变量的值。	`os.Getenv("KEY")`

进程控制与输入输出

函数名/变量名	使用说明	代码案例
Exit	终止当前进程并返回指定的退出码。	`os.Exit(0)`
Args	返回当前程序的命令行参数。	`args = os.Args`
Stdout	标准输出的文件对象（Writer）。	`Stdout`
Stdin	标准输入的文件对象（Reader）。	`Stdin`
Stderr	标准错误输出的文件对象（Writer）。	`Stderr`

7.4 网络通信库函数

网络通信是计算机设备之间交换数据的过程，它依赖于一系列标准化的规则和协议来确保信息能够从一个地方顺利传输到另一个地方。在这个过程中，最基本的要素包括数据包、IP地址、端口和协议。

数据包是网络中传输信息的基本单位，它包含了要传输的数据以及发送和接收数据所需的地址信息。IP地址是分配给每个连接到网络设备的唯一数字标识，用于确保数据能够准确送达目的地。端口号则像是设备内部的地址，指导数据包到达正确的应用程序或服务。

协议定义了数据传输的规则和格式，其中TCP（传输控制协议）和UDP（用户数据报协议）是最为常见的两种。TCP提供可靠的、有序的和错误检测机制的数据传输方式，适用于需要准确数据传输的应用，如网页浏览和电子邮件。UDP则提供一种较为快速但不保证数据包送达顺序或完整性的传输方式，常用于流媒体和在线游戏。

网络通信还涉及到一些其他重要的概念，比如DNS（域名系统）将易于记忆的域名转换为IP地址，路由器帮助数据在不同网络间正确传输，防火墙保护网络不受未授权访问，而NAT（网络地址转换）允许多个设备共享同一个公共IP地址进行互联网访问。

这些元素共同构成了网络通信的基础框架，使得我们能够在全球范围内进行数据交换和通信。在实际应用中，如何有效地利用这些概念是网络编程的关键所在。Yak语言的网络通信库函数对TCP和UDP提供了简单易用的封装，用户可以通过基础API的学习快速掌握网络基础API操作。

7.4.1 TCP协议通信

TCP (传输控制协议) 是一种面向连接的、可靠的、基于字节流的传输层通信协议。它确保数据准确无误地从源传输到目的地。在TCP连接中，数据是按顺序发送的，所以接收方会按发送的顺序接收数据。TCP的特点:

- **面向连接**: 在数据传输之前，必须先建立连接。
- **可靠性**: 确保数据包准确无误地到达目的地，如果有丢失，发送方会重新发送。
- **数据顺序**: 数据包到达接收方时，能够重组为其原始发送顺序。
- **流量控制**: 控制数据的发送速率，以避免网络拥塞。
- **拥塞控制**: 避免过多的数据同时传输导致网络拥塞。

快速开始

Yak语言提供了一种新的TCP通信的方案（相对于 `socket` 来说），这种方式提供了一个比直接使用socket更简单、更高级的方法来进行TCP通信，非常适合初学者学习网络编程的基础，并且使得代码更易于理解和维护，我们用一个简单的案例来为大家展示这种通信方式：

```
go func{
    tcp.Serve("127.0.0.1", 8085 /*type: int*/, tcp.serverCallback(conn => {
        conn.Write("Hello I am server")
        conn.Close()
    }))
}
os.WaitConnect("127.0.0.1:8085", 4) // 等待服务器完全启动

conn = tcp.Connect("127.0.0.1", 8085)~
data = conn.ReadFast()~
dump(data)
/*
OUTPUT:

([]uint8) (len=17 cap=64) {
 00000000  48 65 6c 6c 6f 20 49 20  61 6d 20 73 65 72 76 65  |Hello I am serve|
 00000010  72                                                |r|
}
```

```
19 */
20 conn.Close()
```

- **创建TCP服务器**

首先，我们创建一个TCP服务器，它将在本地环回地址（`127.0.0.1`）上的`8085`端口监听传入的连接。

```
1 go func{
2     tcp.Serve("127.0.0.1", 8085, tcp.serverCallback(conn => {
3         conn.Write("Hello I am server")
4         conn.Close()
5     }))
6 }
```

这里的`tcp.Serve`函数启动了一个服务器，并指定了监听地址和端口。`tcp.serverCallback`是当新的连接建立时会调用的回调函数。在这个函数内，我们向客户端发送一条消息`"Hello I am server"`，然后关闭连接。注意：在上述代码中，我们把`tcp.Serve`的代码放在异步启动的过程中，用户需要根据实际情况选择到底是同步启动还是异步启动。

- **等待服务器启动**

```
1 os.WaitConnect("127.0.0.1:8085", 4)
```

`os.WaitConnect`是一个同步操作，确保服务器在继续之前已经开始监听端口。这是防止客户端在服务器准备好之前尝试连接。

- **创建TCP客户端并连接**

```
1 conn = tcp.Connect("127.0.0.1", 8085)
```

客户端使用`tcp.Connect`函数发起到服务器的连接。

- **接收数据**

```
1 data = conn.ReadFast()
```

连接建立后，客户端使用`conn.ReadFast()`方法快速读取服务器发送的数据。

- **输出数据**

```
1 dump(data)
```

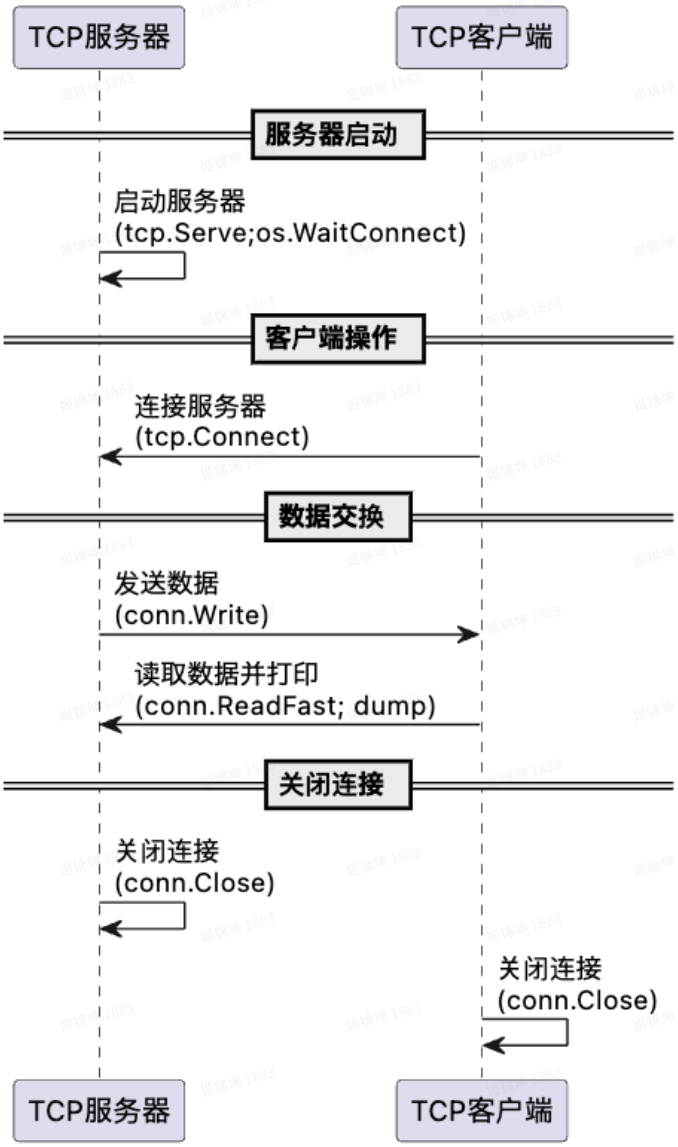
`dump` 函数用于打印接收到的数据，这里是服务器发送的消息。

- **关闭连接**

```
1 conn.Close()
```

数据传输完成后，客户端关闭连接。

这段完整的通信过程描述了服务器的建立到通信的完整生命周期，可以用如下时序图来表示：



使用连接对象读写信息

在上述的案例中，读者在客户端和服务端都用到了`conn`这个变量，这个变量指的是Yak语言内置结构中的`tcpConnection`对象。这个对象把常见的用户操作浓缩成了若干简单易用的接口，覆盖了各种常见的读写场景。

方法名	使用说明	使用案例
Close() error	关闭连接	`err = conn.Close()`
LocalAddr() addr	获取本地网络地址	`addr = conn.LocalAddr()`
Read([]byte)	经典的Read底层接口：从连接中读取数据到字符串	`buffer = make([]byte, 1024); n, err = conn.Read(buffer);`
RemoteAddr()	获取远程网络地址	`addr = conn.RemoteAddr()`
Write(data)	向连接写入数据，接受字符序列或字符串	`n, err = conn.Write(data)`
GetTimeout	获取当前设置的超时时间	`timeout = conn.GetTimeout()`
ReadFast()	快速读取数据：在超时时间内读取数据，如果数据在一段时间内没有额外返回了，就立即返回数据。	`data, err = conn.ReadFast()`
ReadFastUntilByte(sep)	快速读取数据：在超时时间内读取数据，如果数据在一段时间内没有额外返回了或读到了某一个特定字节，就立即返回数据；或者	`data, err = conn.ReadFastUntilByte('\n')`
Recv()	接收数据到字符串（受超时影响）	`data, err = conn.Recv()`
RecvLen(length)	接收指定长度的数据到字符串	`data, err = conn.RecvLen(1024)`
RecvString()	接收字符串数据	`str, err = conn.RecvString()`
RecvStringTimeout(du)	在超时时间内接收字符串数据	`str, err = conn.RecvStringTimeout(5)`
RecvTimeout	在超时时间内接收数据到字符串	`data, err = conn.RecvTimeout(4)`
Send	发送数据，参数可以是多种类型	`err = conn.Send(message)`
SetTimeout	设置超时时间	`conn.SetTimeout(timeout)`

7.4.2 UDP协议通信

UDP（用户数据报协议）是一种无连接的网络协议，提供了一种快速但不可靠的数据传输服务。与TCP不同，UDP不保证数据包的顺序、完整性或可靠性。但它的优势在于低延迟和较小的协议开销，非常适合对实时性要求高的应用，如视频流、在线游戏等。

快速开始

类似TCP通信中的接口，Yak语言也提供了一套UDP协议通信的接口，这些接口的表现形式和`socket`接口有一些区别，但是非常适合初学者学习和使用，用这些接口编写的代码更易于理解和维护，我们用一个简单的案例来为大家展示这种通信方式：

```
// 获取一个随机可用的 UDP 端口
host, port = "127.0.0.1", os.GetRandomAvailableUDPPort()

// 异步启动一个服务器
go func {
    udp.Serve(host, port, udp.serverCallback((conn, data) => {
        println(f`message from client: ${string(data)}`)
        conn.Write("UDP Message From Server")~
    }))
}

sleep(1) // 等待一秒，确保服务器完全启动

// 连接服务器的地址，并发送一个字符串
conn = udp.Connect(host, port)~
conn.Send("UDP Message From Client")~

// 设置超时，接收服务器的信息
conn.SetTimeout(2)
data = conn.Recv()~
conn.Close()
println(f"message from server: ${string(data)}")

```

创建UDP服务器

```
host, port = "127.0.0.1", os.GetRandomAvailableUDPPort()

go func {
    udp.Serve(host, port, udp.serverCallback((conn, data) => {
        println(f`message from client: ${string(data)}`)
```

```
        conn.Write("UDP Message From Server")~
    }))
}
```

在上述代码中，我们首先定义了服务器监听的地址和端口。`os.GetRandomAvailableUDPPort()` 函数用于获取一个随机可用的UDP端口。然后我们启动了一个 `goroutine` 来运行 `udp.Serve` 函数，它会监听指定的地址和端口，并在接收到数据时调用提供的回调函数。在回调函数中，我们打印出客户端发送的消息，并向客户端发送一条回复。注意：由于服务器是异步启动的，关于异步编程的内容，用户可随时返回第六章学习。

创建客户端

```
sleep(1)                                    // 等待一秒，确保服务器完全启动
conn = udp.Connect(host, port)~             // 连接服务器
conn.Send("UDP Message From Client")~       // 给服务器发送一个字符串

conn.SetTimeout(2)                          // 设置超时
data = conn.Recv()~                         // 接受服务器返回的内容
conn.Close()                                // 关闭连接
println(f"message from server: ${string(data)}")

```

在客户端代码中，我们首先等待一秒钟以确保服务器已经启动。然后使用 `udp.Connect` 函数连接到服务器，并发送一条消息。之后，我们设置了一个2秒的超时时间来等待服务器的响应。一旦接收到响应，我们就关闭连接，并打印出服务器发送的消息。

使用连接对象读写信息

在上述案例中，我们使用到了 `udpConnection` 作为返回的对象进行数据交换处理，在实际的处理中，连接大致分为读和写两大类，我们把常用的定义写在下面列表中

方法名	使用说明	使用案例
`Close()`	关闭 UDP 连接。	`conn.Close()`
`GetTimeout()`	获取当前设置的超时时间。	`timeout = conn.GetTimeout()`
`Recv()`	接收 UDP 数据包，返回数据和错误。	`data, err = conn.Recv()`
`RecvLen(lengthInt)`	接收指定长度的 UDP 数据包。	`data, err = conn.RecvLen(1024)`

`RecvString()`	接收 UDP 数据包并作为字符串返回。	`message, err = conn.RecvString()`
`RecvStringTimeout(seconds)`	设置超时并尝试接收字符串。	`message, err = conn.RecvStringTimeout(5.0)`
`RecvTimeout(seconds)`	设置超时并尝试接收数据。	`data, err = conn.RecvTimeout(5.0)`
`RemoteAddr()`	获取远程地址信息。	`addr = conn.RemoteAddr()`
`Send(data)`	发送数据，可以是任意类型，内部会转换为字节流发送。	`err = conn.Send("Hello World")`
`Write(data)`	发送数据（和 `Send` 基本相同），但是返回值不同，第一个返回值为字节数。	`n, err = conn.Write([]byte("Hello World"))`
`SetTimeout(seconds)`	设置连接的超时时间。	`conn.SetTimeout(10.0)`

这个表格可以帮助用户理解每个方法的功能和如何在代码中使用它们，如果需要更多的接口使用，用户可以直接在Yak语言的开源仓库中找到所有的实现。

7.5 正则表达式库函数：re

正则表达式（Regular Expression），也被称为“Regex”或“RegExp”，是一种用于描述字符串模式的文本模式。正则表达式通常由一系列字符和特殊字符组成，用于匹配或查找其他字符串中的模式。

正则表达式可以在多种编程语言和工具中使用。在使用正则表达式时，需要仔细考虑模式的要求，以确保正则表达式可以准确匹配所需的模式。

Yak的基础正则表达式模块，除了支持基础的正则表达式，还提供了网络安全领域中常用正则的预设，帮助用户更快的完成正则处理。

笔者本节将从“通用的正则表达式函数”和“安全领域的正则工具函数”两个方面介绍Yak的正则表达式库函数。

7.5.1 通用正则表达式函数

Yak 的正则表达式库支持对正则表达式的基础编译执行功能，支持将正则表达式编译成对应的正则表达式对象，并且支持 POSIX 和 Grok

函数名	说明	参数	返回值	使用案例
QuoteMeta	转义字符串中所有正则表达式的元字符。	s string	string	`re.QuoteMeta("1.5-2.0?")` 返回 `"1\\.5\\-2\\.0\\?"`
Compile	将正则表达式字符串编译成正则表达式对象。	pattern string	(*Regexp, error)	`re.Compile("^[a-z]+$")` 编译正则表达式
CompilePOSIX	将POSIX(ERE)正则表达式字符串编译成正则表达式对象。	pattern string	(*Regexp, error)	`re.CompilePOSIX("^[a-z]+$")` 编译POSIX正则表达式
MustCompile	类似于compile，但如果正则表达式无法编译，则会引发崩溃。	pattern string	*Regexp	`re.MustCompile("^[a-z]+$")` 强制编译正则表达式
MustCompilePOSIX	类似于CompilePOSIX，但如果正则表达式无法编译，则会引发崩溃。	pattern string	*Regexp	`re.MustCompilePOSIX("^[a-z]+$")` 强制编译POSIX正则表达式
Match	判断字符串是否与正则表达式匹配。	pattern string, s string	bool	`re.Match("^[a-z]+$", "test")` 检查字符串是否匹配
Grok	将字符串 line 使用 Grok 以规则 rule 进行解析，并返回解析结果(map)	line string, rule string	map[string][]string	`re.Grok("04/18-00:59:45.385191", "%{MONTHNUM:month}/%{MONTHDAY:day}-%{TIME:time}")` 使用Grok模式解析文本，返回 `map[HOUR:[00] MINUTE:[59] SECOND:[45.385191] day:[18] month:[04] time:[00:59:45.385191]]`

除去上述的编译方式的正则执行，Yak还支持无需编译、直接执行正则表达式操作匹配字符串的方式。

函数名	说明	参数	返回值	使用案例
`Find`	查找字符串中正则表达式的第一个匹配项。	s string, pattern string	string	`Find("abc", "a(b)")` 返回 `"ab"`
`FindIndex`	查找字符串中正则表达式的第一个匹配项的索引。	s string, pattern string	[]int	`FindIndex("abc", "a")` 返回 `[0,1]`
`FindAll`	查找字符串中所有正则表达式的匹配项。	s string, pattern string	[]string	`FindAll("abc acd adb", "a.", -1)` 返回 `["ab", "ac", "ad"]`
`FindAllIndex`	查找字符串中所有正则表达式匹配项的索引。	s string, pattern string	[][]int	`FindAllIndex("abc acd adb", "a", -1)` 返回 `[[0, 1], [4, 5], [8, 9]]`
`FindSubmatch`	查找字符串中正则表达式的第一个匹配项及其分组。	s string, pattern string	[][]string	`FindSubmatch("abc", "a(b)")` 返回 `[["ab", "b"]]`
`FindSubmatchIndex`	查找字符串中正则表达式的第一个匹配项及其分组的索引。	s string, pattern string	[]int	`FindSubmatchIndex("abc", "a(b)")` 返回 `[0, 2, 1, 2]`
`FindSubmatchAll`	查找字符串中所有正则表达式的匹配项及其分组。	s string, pattern string	[][]string	`FindSubmatchAll("abc abd", "a(b)")` 返回 `[["ab", "b"], ["ab", "b"]]`
`FindSubmatchAllIndex`	查找字符串中所有正则表达式匹配项及其分组的索引。	s string, pattern string	[][]int	`FindSubmatchAllIndex("abc abd", "a(b)")` 返回 `[[0, 2, 1, 2], [4, 6, 5, 6]]`
`FindGroup`	提取字符串中符合正则表达式的分组数据。	s string, pattern string	map[string]string	`FindGroup("ab", "(?P<key>a)(?P<value>b)")` 返回 `{"0":"ab","key": "a", "value": "b"}`
`FindGroupAll`	提取字符串中所有符合正则表达式的分组数据。	s string, pattern string	[]map[string]string	`FindGroupAll("ab ac ad", "(?P<key>a)(?P<value>b)")`

				返回 `[{"0":"ab","key": "a", "value": "b"}]`
`ReplaceAll`	使用正则表达式替换字符串。	s string, pattern string, newstring string	string	`ReplaceAll("abc cba", "abc", "ok")` 返回 `"ok cba"`
`ReplaceAllWithFunc`	使用函数处理字符串中的正则表达式匹配项，并替换它们。	s string, pattern string, repl func(string) string	string	`ReplaceAllWithFunc("axay", "a(x)", str.ToUpper)` 返回 `"AX ay"`

7.5.2 安全领域的正则工具函数

在网络安全领域，数据提取和模式识别是至关重要的任务。为了有效地处理和分析日志文件、网络流量数据以及其他安全相关的信息，我们常常需要从大量的数据中快速提取出有意义的信息。这就是为什么在我们的正则表达式匹配库中，除了上述介绍的通用正则表达式功能外，还特别包含了一系列预置的正则提取函数。这些函数专门用于识别和提取网络安全日志中的特定数据，如IP地址、域名、邮箱地址等。

使用这些预置的正则提取函数，安全分析师可以更加轻松地从复杂的数据中提取关键信息，从而快速响应安全事件，进行威胁狩猎和事件调查。下面，我们将介绍这些专为网络安全设计的正则提取函数，它们能够让复杂的数据提取任务变得简单高效。

在网络安全的正则表达式匹配库中，特定的预置函数能够帮助用户快速提取出日志或文本中的关键网络信息。下面是一些预置正则提取函数的简要说明

函数名	说明	输出	输出	使用案例
`ExtractIPv4`	提取有效的IPv4地址	文本字符串	IPv4地址字符串	`ExtractIPv4("Access from 192.168.1.1")` 返回 `["192.168.1.1"]`
`ExtractIPv6`	提取有效的IPv6地址	文本字符串	IPv6地址字符串	`ExtractIPv6("Request from 2001:0db8:85a3:0000:0000:8a2e:0370:7334")` 返回 `["2001:0db8:85a3:0000:0000:8a2e:0370:7334"]`
`ExtractIP`	提取IPv4或IPv6地址	文本字符串	IP地址字符串列表	`ExtractIP("Server IP: 192.168.1.1 or 2001:0db8:85a3::8a2e:0370:7334")` 返回 `["192.168.1.1",`

				`"2001:0db8:85a3::8a2e:0370:7334"]`
`ExtractEmail`	提取电子邮箱地址	文本字符串	邮箱地址字符串	`ExtractEmail("Contact: admin@example.com")` 返回 "admin@example.com"
`ExtractPath`	提取文件路径或URL路径	文本字符串	路径字符串	`ExtractPath("/var/log/syslog")` 返回 `["/var/log/syslog"]`
`ExtractTTY`	提取TTY设备信息	文本字符串	TTY设备信息字符串	`ExtractTTY("User logged in via pts/1")` 返回 `["pts/1"]`
`ExtractURL`	提取URL	文本字符串	URL字符串	`ExtractURL("Visit https://www.example.com for more info")` 返回 `["https://www.example.com"]`
`ExtractHostPort`	提取主机名和端口号	文本字符串	主机名和端口号字符串	`ExtractHostPort("Connect to localhost:8080")` 返回 `["localhost:8080"]`
`ExtractMac`	提取MAC地址。	文本字符串	MAC地址字符串	`ExtractMac("MAC: 00:1A:2B:3C:4D:5E")` 返回 `["00:1A:2B:3C:4D:5E"]`

这些函数是正则表达式库的扩展，专门用于匹配和提取网络安全领域中的常见数据格式。通过这些函数，用户可以快速地从文本中提取出有用的信息，这在进行网络监控、事件响应和日志分析时尤为重要。

7.6 JSON工具函数

JSON是一种轻量级的数据交换格式。它使用文本格式来传输结构化数据，包括数组、对象、字符串、数字、布尔值和null。JSON格式被广泛用于Web应用程序和API中，作为一种数据格式，以实现不同应用程序之间的数据交换。JSON是一种平台无关的格式，可以使用许多编程语言进行解析和生成，包括JavaScript、Python、Java等。JSON的语法简单、易于理解和阅读，与XML和HTML相比，它更轻量级和灵活，因此在数据传输和存储方面更加高效。

Yak的json库中不仅仅具备对json数据处理的基础支持，还有支持了更加优雅的JsonPath机制。

JSON基础处理

Yak对于json的处理支持，有两套API

函数名称	描述	使用示例
`json.New`	把字符串或者一个对象，变成 json 序列化之后的内容	`json.New(`{"test": 123}`)`
`json.Marshal`	把任意一个实例解析成 JSON 字符串()。	`json.Marshal(myInstan)`

下面是一个简单的案例：

```
myInstan = {"test":321}
println(json.New(`{"test": 123}`))
println(json.Marshal(myInstan))
```
```
OUTPUT:
[0xc002767240 <nil>] yakJson对象
[[123 34 116 101 115 116 34 58 51 50 49 125] <nil>]

```

上述的一套API中`json.New`返回的序列化内容是Yak内部定义的一个类型，此类型内置一些成员方法可用于辅助完成一些分析工作。

```
type palm/common/yak/yaklib.(yakJson) struct {
PtrStructMethods(指针结构方法/函数):
    // 判断解析出的对象是否是数组 []
    func IsArray() return(bool)
    func IsSlice() return(bool)

    // 判断解析出的对象是否是 Object / map
    func IsObject() return(bool)
    func IsMap() return(bool)

    // 判断是否是空
    func IsNil() return(bool)
    func IsNull() return(bool)

    // 判断解析出的是否是数字
    func IsNumber() return(bool)

    // 判断是否是字符串
    func IsString() return(bool)
```

```
20
21      // 获得解析出来的具体的值
22      func Value() return(interface {})
23  }
```

除了上述的一套API以外，Yak还有另一套API：`dumps/loads`

函数名称	描述
`json.dumps`	类似于 `json.Marshal` 的别名或自定义函数，可能具有特定于库的额外功能或配置。
`json.loads`	解析 JSON 字符串，将其解码为 Yak对象。

下面是一个例子，可以直观地看到这对函数的作用：

```
1  jsonRaw = `[1,23,4,"abc",true,false, {"abc": 123123, "dddd":"123"}]`
2  a = ["123", true, false, "123123", 123, {"abc": 123},nil]
3  dump(json.loads(jsonRaw))
4  println(json.dumps(a))
5  /*
6  OUTPUT:
7  ([]interface {}) (len=7 cap=8) {
8   (float64) 1,
9   (float64) 23,
10  (float64) 4,
11  (string) (len=3) "abc",
12  (bool) true,
13  (bool) false,
14  (map[string]interface {}) (len=2) {
15   (string) (len=3) "abc": (float64) 123123,
16   (string) (len=4) "dddd": (string) (len=3) "123"
17  }
18 }
19 ["123",true,false,"123123",123,{"abc":123},null]
20 */
```

JsonPath

JSONPath是一种用于从JSON格式的数据结构中提取特定数据的查询语言，类似于XPath。它提供了一种通用的方式来访问和操作JSON数据，可以用于编程语言或命令行中，以实现复杂的JSON数据处理和分析。

提取数据是JsonPath的重要用途。Yak的json库的json.find函数是对JsonPath的良好封装。

函数名称	描述
json.Find	使用JsonPath提取JSON数据

下面使用三个处理下述JSON数据的案例，来展示JsonPath优雅的提取数据能力。

```
jsonRaw=`{
    "name": "YaklangUser",
    "criticalList": [
        {"key": "a1", "name": "b1"},
        {"key": "a1-3", "name": "b4"},
        {"key": "a2", "value": "c3"},
        {"key": "a2-3", "value": "c6", "age": 12},
        {"key": "a5", "anothorList": [
            {"key": "in", "age": 30},
            {"key": "in3", "age": 88}
        ], "age": 14},
        {"key": "a6", "age": 19}
    ]
}`
```

1. 提取根节点的name字段

```
rootName = json.Find(jsonRaw, "$.name")
printf("Fetch `name` in root node: %v\n", rootName)

/*
OUTPUT:
    Fetch `name` in root node: YaklangUser
*/
```

2. 提取所有对象中的name字段

```
results = json.Find(jsonRaw, "$..name")
dump(results)
/*
OUTPUT:
([]interface {}) (len=3 cap=4) {
 (string) (len=11) "YaklangUser",
```

```
  (string) (len=2) "b1",
  (string) (len=2) "b4"
}
*/
```

3. 提取数组数据

```
results = json.Find(jsonRaw, "$.criticalList[1]")
dump(results)
/*
OUTPUT:
(map[string]interface {}) (len=2) {
 (string) (len=3) "key": (string) (len=4) "a1-3",
 (string) (len=4) "name": (string) (len=2) "b4"
}
*/
```

当然，还有更高级的用法，不过在这里就不多做赘述。

7.7 编解码与加解密库函数

在计算机领域中，编解码是将数据从一种形式转换为另一种形式的过程，以便在不同的系统之间传输或存储。它们的重要性在于确保数据能够在不同系统之间正确传递和处理，而不会受到编码格式的限制。举例来说，在 Web 开发中，浏览器和服务器之间使用 UTF-8 编码的 HTTP 协议进行通信，以确保数据能够准确传输。因此，编码和解码在保证数据传输和处理的正确性方面起着关键作用。

与编解码不同，加解密是保护敏感数据和信息安全的重要手段。它们的意义包括：保护隐私和机密性、防止数据被篡改、防止数据被窃听以及认证和授权。通过加密，只有授权人员可以访问加密的数据，确保隐私和机密性的保护。通过数字签名和加密技术，数据在传输和存储过程中不被篡改，保护数据的完整性。加密技术可以防止未经授权的人窃听数据传输过程，保护数据的机密性和安全性。通过数字证书和数字签名技术，可以验证数据的来源和真实性，确保数据的认证和授权，防止非法访问和恶意攻击。

Yak的codec库理所当然的支持大部分的编解码方式与加解密算法。

编解码

Yak支持丰富的编解码方式，包括Base64、URL编码等各种常见的编解码方式，具体使用可见下表：

函数名	输入说明	返回值说明	函数说明	使用示例

codec.EncodeToHex	待编码的文本字符串	编码后的十六进制字符串	将文本转换为十六进制表示。	`codec.EncodeToHex("Yak")// 59616b`
codec.DecodeHex	待解码的十六进制字符串	解码后的文本字符串，可能的错误	将十六进制字符串解码为文本，可能会遇到错误。	`codec.DecodeHex("686578")// hex`
codec.EncodeBase64	待编码的文本字符串	编码后的Base64字符串	将文本转换为Base64编码。	`codec.EncodeBase64("Yak")// WWFr`
codec.DecodeBase64	待解码的Base64字符串	解码后的文本字符串，可能的错误	将Base64字符串解码为文本，可能会遇到错误。	`codec.DecodeBase64("YmFzZTY0")// base64`
codec.EncodeBase32	待编码的文本字符串	编码后的Base32字符串	将文本转换为Base32编码。	`codec.EncodeBase32("Yak")// LFQWW===`
codec.DecodeBase32	待解码的Base32字符串	解码后的文本字符串，可能的错误	将Base32字符串解码为文本，可能会遇到错误。	`codec.DecodeBase32("LFQWW===")// Yak`
codec.EncodeBase64Url	待编码的文本字符串	URL安全的Base64编码字符串	进行URL安全的Base64编码	`codec.EncodeBase64Url("\xFB\xFF")// -_8`
codec.DecodeBase64Url	待解码的URL安全的Base6字符串	解码后的文本字符串，可能的错误	进行URL安全的Base64解码	`codec.DecodeBase64Url("-_8")// \xFB\xFF`
codec.EncodeUrl	待编码的URL字符串	编码后的URL字符串	将URL文本转换为编码后的URL格式。	`codec.EncodeUrl("http://example.com/test")// %68%74%74%70%3a%2f%2f%65%78%61%6d%70%6c%65%2e%63%6f%6d%2f%74%65%73%74`
codec.DecodeUrl	待解码的URL字符串	解码后的URL字符串，可能的错误	将编码后的URL字符串解码为原始URL格式，可能会遇到错误。	`decodedUrl, err = codec.DecodeUrl("%68%74%74%70%3a%2f%2f%65%78%61%6d%70%6c%65%2e%63%6f%6d%2f%74%65%73%74")`

				`//http://example.com/test`
codec.EscapePathUrl	待转义的URL字符串	转义后的URL字符串	转义URL字符串。	`codec.EscapePathUrl("http://example.com/test")//http:%2F%2Fexample.com%2Ftest`
codec.UnescapePathUrl	待还原的URL字符串	还原后的URL字符串，可能的错误	将转义后的URL字符串还原为原始格式，可能会遇到错误。	`codec.UnescapePathUrl("http:%2F%2Fexample.com%2Ftest")//http://example.com/test`
codec.EscapeQueryUrl	待转义的URL查询字符串	转义后的URL查询字符串	对URL查询部分进行转义。	`codec.EscapeQueryUrl("a=b&c=d")//a%3Db%26c%3Dd`
codec.UnescapeQueryUrl	待还原的URL查询字符串	还原后的URL查询字符串，可能的错误	将转义后的URL查询字符串还原为原始格式，可能会遇到错误。	`codec.UnescapeQueryUrl("a%3Db%26c%3Dd")//a=b&c=d`
codec.DoubleEncodeUrl	待双重编码的URL字符串	双重编码后的URL字符串	对URL文本进行双重编码。	`codec.DoubleEncodeUrl("http://example.com/test")//%2568%2574%2574%2570%253a%252f%252f%2565%2578%2561%256d%2570%256c%2565%252e%2563%256f%256d%252f%2574%2565%2573%2574`
codec.DoubleDecodeUrl	待双重解码的URL字符串	双重解码后的URL字符串，可能的错误	将双重编码后的URL字符串解码为原始格式，可能会遇到错误。	`codec.DoubleDecodeUrl("%2568%2574%2574%2570%253a%252f%252f%2565%2578%2561%256d%2570%256c%2565%252e%2563%256f%256d%252f%2574%2565%2573%2574")//http://example.com/test`
codec.EncodeHtml	待编码的HTML文本字符串	编码后的HTML字符串	将HTML文本转换为实体编码。	`codec.EncodeHtml("<div>yak</div>")//&#60;&#100;&#105;&#118`

				`;&#62;&#121;&#97;&#107;&#60;&#47;&#100;&#105;&#118;&#62;`
codec.EncodeHtmlHex	待编码的HTML文本字符串	编码后的十六进制HTML字符串	将HTML文本转换为十六进制实体编码。	`codec.EncodeHtmlHex("<div>yal</div>")// &#x3c;&#x64;&#x69;&#x76;&#x3e;&#x79;&#x61;&#x6b;&#x3c;&#x2f;&#x64;&#x69;&#x76;&#x3e;`
codec.EscapeHtml	待转义的HTML文本字符串	转义后的HTML字符串	对HTML文本进行转义。	`codec.EscapeHtml("<div>yak</div>")// <div>yak</div>`
codec.DecodeHtml	待解码的HTML实体字符串	解码后的HTML文本字符串，可能的错误	将HTML实体编码字符串解码为原始HTML文本，可能会遇到错误。	`codec.DecodeHtml("&#60;&#100;&#105;&#118;&#62;&#121;&#97;&#107;&#60;&#47;&#100;&#105;&#118;&#62;")// <div>yak</div>`
codec.EncodeToPrintable	待编码的文本（任意字符串）	编码后的可打印文本	将文本编码为可打印格式	`codec.EncodeToPrintable("yak\n")// "yak\x0a"`
codec.EncodeASCII	待编码的ASCII文本	编码后的ASCII文本	将文本编码为ASCII格式	`codec.EncodeASCII("yak\n")// "yak\x0a"`
codec.DecodeASCII	待解码的ASCII文本	解码后的文本，错误信息	将ASCII文本解码为原始文本	`codec.DecodeASCII(`"yak\x0a"`)// yak\n`
codec.EncodeChunked	待编码的文本（任意字符串）	分块编码后的文本	将文本进行HTTP分块编码	`codec.EncodeChunked(`yaklang`)// 3\r\nyak\r\n4\r\nlang\r\n0\r\n\r\n`
codec.DecodeChunked	待解码的分块编码文本	解码后的文本，错误信息	将分块编码的文本解码	`codec.DecodeChunked("3\r\nyak\r\n4\r\nlang\r\n0\r\n\r\n")// yaklang`
codec.StrconvQuote	待编码的文本（任意字符串）	转义编码后的文本	将文本用引号包围，并转义符号	`codec.StrconvQuote("Yak!")// "Yak\x21"`

codec.StrconvUnquote	待解码的引号包围的文本	解码后的文本，错误信息	反转义文本并移除引号包围	`codec.StrconvUnquote(`"Yak\x21"`)// Yak!`
codec.UTF8ToGBK	待编码的UTF-8文本	编码后的GBK文本	将UTF-8文本编码为GBK格式	`codec.UTF8ToGBK("转码测试")`
codec.UTF8ToGB18030	待编码的UTF-8文本	编码后的GB18030文本	将UTF-8文本编码为GB18030格式	`codec.UTF8ToGB18030("转码测试")`
codec.UTF8ToHZGB2312	待编码的UTF-8文本	编码后的HZGB2312文本	将UTF-8文本编码为HZGB2312格式	`codec.UTF8ToHZGB2312("转码测试")`
codec.GBKToUTF8	待编码的GBK文本	编码后的UTF-8文本	将GBK文本编码为UTF-8格式	`codec.GBKToUTF8("转码测试")`
codec.GB18030ToUTF8	待编码的GB18030文本	编码后的UTF-8文本	将GB18030文本编码为UTF-8格式	`codec.GB18030ToUTF8("转码测试")`
codec.HZGB2312ToUTF8	待编码的HZGB2312文本	编码后的UTF-8文本	将HZGB2312文本编码为UTF-8格式	`codec.HZGB2312ToUTF8("转码测试")`
codec.GBKSafe	待编码的GBK文本	安全的GBK编码文本	生成一个安全的GBK编码字符串	`codec.GBKSafe("安全编码")`
codec.FixUTF8	待修复的UTF-8文本	修复后的UTF-8文本，错误信息	修复无效的UTF-8编码字节	`codec.FixUTF8("example")`
codec.HTMLChardet	待检测编码的HTML文本	推测的编码格式，错误信息	检测HTML文本的字符编码	`codec.HTMLChardet("<html>...</html>")`
codec.HTMLChardetBest	待检测编码的HTML文本	最佳推测的编码格式，错误信息	检测并返回最佳猜测的HTML字符编码	`codec.HTMLChardetBest("<html>...</html>")`
codec.UnicodeEncode	待编码的文本（任意字符串）	编码后的Unicode文本	将文本编码为Unicode转义序列	`codec.UnicodeEncode("☺")// \u263a`
codec.UnicodeDecode	待解码的Unicode文本	解码后的文本，错误信息	将Unicode转义序列解码为原始文本	`codec.UnicodeDecode("\\u263A")// ☺`

加解密

在实际环境中，数据加密是保护敏感信息免受未经授权访问的重要手段。Yak 内置加密库提供了多种加密算法，以满足不同的安全需求。以下是支持的一些常见的商用加解密分类：

对称加密算法

对称加密算法使用相同的密钥进行加密和解密。这种加密方式因其算法的效率而广泛用于需要加密大量数据的场景，如文件存储、数据库加密和网络通信等。Yak对常见的几种对称加密方式都有支持。

- **AES (Advanced Encryption Standard)** AES是一种广泛使用的加密标准，可以使用128、192或256位的密钥长度。它被认为是非常安全的，并且是许多政府和企业的首选。

函数名	输入说明	返回值说明	函数说明	使用示例
AESEncrypt	密钥、明文、IV	密文、错误	使用AES算法进行加密	`codec.AESEncrypt(key, plaintext, iv)`
AESDecrypt	密钥、密文、IV	原文、错误	使用AES算法进行解密	`codec.AESDecrypt(key, ciphertext, iv)`
AESCBCEncrypt	密钥、明文、IV	密文、错误	使用AES的CBC模式进行加密	`codec.AESCBCEncrypt(key, plaintext, iv)`
AESCBCDecrypt	密钥、密文、IV	原文、错误	使用AES的CBC模式进行解密	`codec.AESCBCDecrypt(key, ciphertext, iv)`
AESCBCEncryptWithZeroPadding	密钥、明文、IV	密文、错误	AES CBC模式加密，使用零填充	`codec.AESCBCEncryptWithZeroPadding(key, plaintext, iv)`
AESCBCDecryptWithZeroPadding	密钥、密文、IV	原文、错误	AES CBC模式解密，使用零填充	`codec.AESCBCDecryptWithZeroPadding(key, ciphertext, iv)`
AESCBCEncryptWithPKCS7Padding	密钥、明文、IV	密文、错误	AES CBC模式加密，使用PKCS7填充	`codec.AESCBCEncryptWithPKCS7Padding(key, plaintext, iv)`
	密钥、密文、IV	原文、错误		

AESCBCDecryptWithPKCS7Padding			AES CBC模式解密，使用PKCS7填充	`codec.AESCBCDecryptWithPKCS7Padding(key, ciphertext, iv)`
AESECBEncrypt	密钥、明文、IV	密文、错误	使用AES的ECB模式进行加密	`codec.AESECBEncrypt(key, plaintext, iv)`
AESECBDecrypt	密钥、密文、IV	原文、错误	使用AES的ECB模式进行解密	`codec.AESECBDecrypt(key, ciphertext, iv)`
AESECBEncryptWithZeroPadding	密钥、明文、IV	密文、错误	AES ECB模式加密，使用零填充	`codec.AESECBEncryptWithZeroPadding(key, plaintext, iv)`
AESECBDecryptWithZeroPadding	密钥、密文、IV	原文、错误	AES ECB模式解密，使用零填充	`codec.AESECBDecryptWithZeroPadding(key, ciphertext, iv)`
AESECBEncryptWithPKCS7Padding	密钥、明文、IV	密文、错误	AES ECB模式加密，使用PKCS7填充	`codec.AESECBEncryptWithPKCS7Padding(key, plaintext, iv)`
AESECBDecryptWithPKCS7Padding	密钥、密文、IV	原文、错误	AES ECB模式解密，使用PKCS7填充	`codec.AESECBDecryptWithPKCS7Padding(key, ciphertext, iv)`
AESGCMEncrypt	密钥、明文、IV	密文、错误	使用AES的GCM模式进行加密	`codec.AESGCMEncrypt(key, plaintext, iv)`
AESGCMDecrypt	密钥、密文、IV	原文、错误	使用AES的GCM模式进行解密	`codec.AESGCMDecrypt(key, ciphertext, iv)`
AESGCMEncryptWithNonceSize16	密钥、明文、IV	密文、错误	AES GCM模式加密，Nonce大小为16	`codec.AESGCMEncryptWithNonceSize16(key, plaintext, iv)`
AESGCMDecryptWithNonceSize16	密钥、密文、IV	原文、错误	AES GCM模式解密，Nonce大小为	`codec.AESGCMDecryptWithNonceSize1`

			16	`6(key, ciphertext, iv)`
AESGCMEncryptWithNonceSize12	密钥、明文、IV	密文、错误	AES GCM模式加密，Nonce大小为12	`codec.AESGCMEncryptWithNonceSize12(key, plaintext, iv)`
AESGCMDecryptWithNonceSize12	密钥、密文、IV	原文、错误	AES GCM模式解密，Nonce大小为12	`codec.AESGCMDecryptWithNonceSize12(key, ciphertext, iv)`

- **DES (Data Encryption Standard)** DES曾是一种流行的加密算法，但由于其56位密钥长度被认为不再安全，现在已经被AES所取代。

函数名	输入说明	返回值说明	函数说明	使用示例
DESEncrypt	密钥、明文、IV	密文、可能的错误	DES算法的CBC模式加密	`ciphertext, err = codec.DESEncrypt(key, plaintext, iv)`
DESDecrypt	密钥、密文、IV	原文、可能的错误	DES算法的CBC模式解密	`plaintext, err = codec.DESDecrypt(key, ciphertext, iv)`
DESCBCEncrypt	密钥、明文、IV	密文、可能的错误	DES算法的CBC模式加密	`ciphertext, err = codec.DESCBCEncrypt(key, plaintext, iv)`
DESCBCDecrypt	密钥、密文、IV	原文、可能的错误	DES算法的CBC模式解密	`plaintext, err = codec.DESCBCDecrypt(key, ciphertext, iv)`
DESECBEncrypt	密钥、明文	密文、可能的错误		

			DES算法的ECB模式加密	`ciphertext, err = codec.DESECBEncrypt(key, plaintext)`
DESECBDecrypt	密钥、密文	原文、可能的错误	DES算法的ECB模式解密	`plaintext, err = codec.DESECBDecrypt(key, ciphertext)`

- **3DES (Triple Data Encryption Standard)** 3DES是DES的一个改进版本，它通过三次重复加密过程提供了更强的安全性。

函数名	输入说明	返回值说明	函数说明	使用示例
TripleDESEncrypt	密钥、明文、IV	密文、可能的错误	3DES算法的CBC模式加密	`ciphertext, err = codec.TripleDESEncrypt(key, plaintext, iv)`
TripleDESDecrypt	密钥、密文、IV	原文、可能的错误	3DES算法的CBC模式解密	`plaintext, err = codec.TripleDESDecrypt(key, ciphertext, iv)`
TripleDESCBCEncrypt	密钥、明文、IV	密文、可能的错误	3DES算法的CBC模式加密	`ciphertext, err = codec.TripleDESCBCEncrypt(key, plaintext, iv)`
TripleDESCBCDecrypt	密钥、密文、IV	原文、可能的错误	3DES算法的CBC模式解密	`plaintext, err = codec.TripleDESCBCDecrypt(key, ciphertext, iv)`
	密钥、明文	密文、可能的错误		

TripleDESECBEncrypt			3DES算法的ECB模式加密	`ciphertext, err = codec.TripleDESECBEncrypt(key, plaintext)`
TripleDESECBDecrypt	密钥、密文	原文、可能的错误	3DES算法的ECB模式解密	`plaintext, err = codec.TripleDESECBDecrypt(key, ciphertext)`

- **RC4(Rivest Cipher 4)** RC4是一种流加密算法。C4特别著名的是它的简单性和速度，在软件中实现可以非常高效。

函数名	输入说明	返回值说明	函数说明	使用示例
RC4Encrypt	密钥、明文	密文、可能的错误	RC4算法加密	`ciphertext, err = codec.RC4Encrypt(key, plaintext)`
RC4Decrypt	密钥、密文	原文、可能的错误	RC4算法解密	`plaintext, err = codec.RC4Decrypt(key, ciphertext)`

- **SM4** 是中国无线局域网标准WAPI（WLAN Authentication and Privacy Infrastructure）的一部分。是一种对称加密算法，是由中国国家密码管理局设计，并且在2006年被采纳为国家标准。

函数名	输入说明	返回值说明	函数说明	使用示例
Sm4CBCEncrypt	密钥、原文、iv	密文、错误	使用CBC模式的SM4加密函数	`encrypted, err = codec.Sm4CBCEncrypt(key, plaintext, iv)`
Sm4CBCDecrypt	密钥、密文、iv	原文、错误	使用CBC模式的SM4解密函数	`decrypted, err = codec.Sm4CBCDec`

				`rypt(key, ciphertext, iv)`
Sm4CFBEncrypt	密钥、原文、iv	密文、错误	使用CFB模式的SM4加密函数	`encrypted, err = codec.Sm4CFBEncrypt(key, plaintext, iv)`
Sm4CFBDecrypt	密钥、密文、iv	原文、错误	使用CFB模式的SM4解密函数	`decrypted, err = codec.Sm4CFBDecrypt(key, ciphertext, iv)`
Sm4ECBEncrypt	密钥、原文	密文、错误	使用ECB模式的SM4加密函数	`encrypted, err = codec.Sm4ECBEncrypt(key, plaintext)`
Sm4ECBDecrypt	密钥、密文	原文、错误	使用ECB模式的SM4解密函数	`decrypted, err = codec.Sm4ECBDecrypt(key, ciphertext)`
Sm4OFBEncrypt	密钥、原文、iv	密文、错误	使用OFB模式的SM4加密函数	`encrypted, err = codec.Sm4OFBEncrypt(key, plaintext, iv)`
Sm4OFBDecrypt	密钥、密文、iv	原文、错误	使用OFB模式的SM4解密函数	`decrypted, err = codec.Sm4OFBDecrypt(key, ciphertext, iv)`
Sm4GCMEncrypt	密钥、原文、iv	密文、错误	使用GCM模式的SM4加密函数	`encrypted, err = codec.Sm4GCMEncrypt(key, plaintext, iv)`

Sm4GCMDecrypt	密钥、密文、iv	原文、错误	使用GCM模式的SM4解密函数	`decrypted, err = codec.Sm4GCMDecrypt(key, ciphertext, iv)`

- **填充生成函数** codec库中还有一些用于填充（Padding）和取消填充（UnPadding）的函数，用在块加密算法中（一种对称加密算法类型）。块加密算法要求输入数据的长度必须是特定块大小的整数倍。当数据长度不满足这个要求时，就需要用到填充算法来扩展数据长度至块大小的整数倍。填充的内容在解密时需要被去除，这就是取消填充函数的作用

函数名	输入说明	返回值说明	函数说明	使用示例
codec.PKCS5Padding	数据（待填充的原始数据），块大小（填充到的块的大小）	填充后的数据	将数据按PKCS#5标准进行填充，以适应特定的块大小	`paddedData = codec.PKCS5Padding(data, blockSize)`
codec.PKCS5UnPadding	数据（待去除填充的数据）	去除填充后的数据	将使用PKCS#5标准填充的数据去除填充	`unpaddedData = codec.PKCS5UnPadding(paddedData)`
codec.PKCS7Padding	数据（待填充的原始数据）	填充后的数据	将数据按PKCS#7标准进行填充，块大小固定为16	`paddedData = codec.PKCS7Padding(data)`
codec.PKCS7UnPadding	数据（待去除填充的数据）	去除填充后的数据	将使用PKCS#7标准填充的数据去除填充	`unpaddedData = codec.PKCS7UnPadding(paddedData)`
codec.ZeroPadding	数据（待填充的原始数据），块大小（填充到的块的大小）	填充后的数据	使用零字节进行填充，直到满足块大小的要求	`paddedData = codec.ZeroPadding(data, blockSize)`
codec.ZeroUnPadding	数据（待去除填充的数据）	去除填充后的数据	去除数据末尾的零字节填充	`unpaddedData = codec.ZeroUnPadding(paddedData)`

非对称加密算法

非对称加密算法使用一对密钥：一个公钥用于加密数据，一个私钥用于解密。这些算法适合于数字签名和加密小量数据。

- **RSA (Rivest-Shamir-Adleman)** RSA是一种非常流行的非对称加密算法，R广泛用于网上银行、数字证书、安全网页浏览（HTTPS）以及许多其他安全通信协议。虽然非对称加密算法通常比对称加密算法慢，但它们在密钥交换和数字签名方面提供了极大的便利和安全性。

函数名	输入说明	返回值说明	函数说明	使用示例
RSAEncryptWithPKCS1v15	公钥、原文	密文和可能遇到的错误	使用PKCS#1 v1.5填充标准进行RSA加密	`cipherText, err = codec.RSAEncryptWithPKCS1v15(publicKey, plainText)`
RSADecryptWithPKCS1v15	私钥、密文	原文和可能遇到的错误	使用PKCS#1 v1.5填充标准进行RSA解密	`plainText, err = codec.RSADecryptWithPKCS1v15(privateKey, cipherText)`
RSAEncryptWithOAEP	公钥、原文	密文和可能遇到的错误	使用OAEP填充标准进行RSA加密	`cipherText, err = codec.RSAEncryptWithOAEP(publicKey, plainText)`
RSADecryptWithOAEP	私钥、密文	原文和可能遇到的错误	使用OAEP填充标准进行RSA解密	`plainText, err = codec.RSADecryptWithOAEP(privateKey, cipherText)`

- **SM2** SM2是一种公钥加密算法，是中国国家密码管理局在2010年发布的商用密码算法标准之一，用于替代RSA等国际算法。SM2算法基于椭圆曲线密码学（Elliptic Curve Cryptography，ECC），其安全性依赖于椭圆曲线离散对数问题（ECDLP）的难解性。与RSA相比，SM2算法在相同安全级别下可以使用更短的密钥长度，从而减少计算量，提高效率。

函数名	输入说明	返回值说明	函数说明	使用示例
codec.Sm2EncryptC1C2C3	公钥 (PEM格式)、原文	密文、可能遇到的错误	使用C1C2C3模式进行SM2加密	`cipherText, err = codec.Sm2EncryptC1C2C3(publicKey, plainText)`
codec.Sm2DecryptC1C2C3	私钥 (PEM格式)、密文	原文、可能遇到的错误	使用C1C2C3模式进行SM2解密	`plainText, err = codec.Sm2DecryptC1C2C3(privateKey, cipherText)`
codec.Sm2DecryptC1C2C3WithPassword	PEM格式私钥、密文、密码	原文、可能遇到的错误	使用密码对PEM格式私钥解密后，再用C1C2C3模式解密	`plainText, err = codec.Sm2DecryptC1C2C3WithPassword(privateKeyPem, cipherText, password)`
codec.Sm2EncryptC1C3C2	公钥 (PEM格式)、原文	密文、可能遇到的错误	使用C1C3C2模式进行SM2加密	`cipherText, err = codec.Sm2EncryptC1C3C2(publicKey, plainText)`
codec.Sm2DecryptC1C3C2	私钥 (PEM格式)、密文	原文、可能遇到的错误	使用C1C3C2模式进行SM2解密	`plainText, err = codec.Sm2DecryptC1C3C2(privateKey, cipherText)`
codec.Sm2DecryptC1C3C2WithPassword	PEM格式私钥、密文、密码	原文、可能遇到的错误	使用密码对PEM格式私钥解密后，再用C1C3C2模式解密	`plainText, err = codec.Sm2DecryptC1C3C2WithPassword(privateKeyPem, cipherText, password)`

codec.Sm2EncryptAsn1	公钥 (PEM格式)、原文	密文、可能遇到的错误	使用ASN.1编码格式进行SM2加密	`cipherText, err = codec.Sm2EncryptAsn1(publicKey, plainText)`
codec.Sm2DecryptAsn1WithPassword	PEM格式私钥、密文、密码	原文、可能遇到的错误	使用密码对PEM格式私钥解密后，再用ASN.1格式解密	`plainText, err = codec.Sm2DecryptAsn1WithPassword(privateKeyPem, cipherText, password)`
codec.Sm2DecryptAsn1	私钥 (PEM格式)、密文	原文、可能遇到的错误	使用ASN.1编码格式进行SM2解密	`plainText, err = codec.Sm2DecryptAsn1(privateKey, cipherText)`

哈希函数 （散列）

哈希函数用于创建数据的固定大小的唯一指纹。它们在存储密码、数据完整性验证和其他安全应用中非常有用。

- **MD5 (Message-Digest Algorithm 5)** 是一种广泛使用的密码哈希函数，由罗纳德·李维斯特（Ronald Rivest）于1991年设计，可以产生一个128位（16字节）的哈希值（hash value），通常用一个32字符的十六进制数表示。MD5已经在互联网上成为一种标准的哈希算法，用于确保信息传输的完整性。

函数名	输入说明	返回值说明	函数说明	使用示例
codec.Md5	数据	哈希值 (32 bytes hash)	计算并返回数据的Md5哈希值	`hash = codec.Md5(data)`

- **SHA (Secure Hash Algorithm) 系列** 包括SHA-1, SHA-256, SHA-384, 和SHA-512，SHA-3等，它们生成不同长度的哈希。

函数名	输入说明	返回值说明	函数说明	使用示例
codec.Sha1	数据	哈希值 (20 bytes hash)	计算并返回数据的SHA-1哈希值	`hash = codec.Sha1(data)`

codec.Sha224	数据	哈希值 (28 bytes hash)	计算并返回数据的SHA-224哈希值	`hash = codec.Sha224(data)`
codec.Sha256	数据	哈希值 (32 bytes hash)	计算并返回数据的SHA-256哈希值	`hash = codec.Sha256(data)`
codec.Sha384	数据	哈希值(48 bytes hash)	计算并返回数据的SHA-384哈希值	`hash = codec.Sha384(data)`
codec.Sha512	数据	哈希值 (64 bytes hash)	计算并返回数据的SHA-512哈希值	`hash = codec.Sha512(data)`

- **SM3** sm3是一种密码哈希函数，由中国国家密码管理局发布为国家标准GB/T 32905-2016。SM3算法被设计用来提供一个安全的哈希机制，可以将任意长度的消息压缩成一个固定长度（256位）的哈希值。

函数名	输入说明	返回值说明	函数说明	使用示例
codec.Sm3	数据	哈希值 (32 bytes hash)	计算并返回数据的SHA-1哈希值	`hash = codec.Sm3(data)`

- **MurmurHash** MurmurHash 是一种非加密型哈希函数，适用于一般的哈希检索操作。

函数名	输入说明	返回值说明	函数说明	使用示例
codec.MMH3Hash32	数据	32位整数形式的哈希值	使用MurmurHash3算法生成32位哈希值	`hash = codec.MMH3Hash32(data)`
codec.MMH3Hash128	数据	128位十六进制形式的哈希值（32 bytes）	使用MurmurHash3算法生成128位哈希值	`hash = codec.MMH3Hash128(data)`
codec.MMH3Hash128x64	数据	同上	使用MurmurHash3算法针对64位平台	`hash = codec.MMH3Hash128x64(data)`

			优化生成128位哈希值	

- **HMAC（Hash-based Message Authentication Code）** HMAC是一种通过特定算法，基于密钥和消息计算出的消息摘要（哈希值），用于验证消息的完整性和真实性的技术。HMAC可以用于任何哈希函数（如MD5, SHA-1, SHA-256等），结合了哈希函数和加密技术的优点。广泛应用于各种安全通信协议。

函数名	输入说明	返回值说明	函数说明	使用示例
codec.HmacSha1	密钥、需要加密的消息	计算摘要后的HMAC SHA1散列值	使用HMAC SHA1算法对给定消息进行消息摘要	`codec.HmacSha1("mysecretkey", "Message to hash")`
codec.HmacSha256	密钥、需要加密的消息	计算摘要后的HMAC SHA256散列值	使用HMAC SHA256算法对给定消息进行消息摘要	`codec.HmacSha256("mysecretkey", "Message to hash")`
codec.HmacSha512	密钥、需要加密的消息	计算摘要后的HMAC SHA512散列值	使用HMAC SHA512算法对给定消息进行消息摘要	`codec.HmacSha512("mysecretkey", "Message to hash")`
codec.HmacMD5	密钥、需要加密的消息	计算摘要后的HMAC MD5散列值	使用HMAC MD5算法对给定消息进行消息摘要	`codec.HmacMD5("mysecretkey", "Message to hash")`
codec.HmacSM3	密钥、需要加密的消息	计算摘要后的HMAC SM3散列值	使用HMAC SM3算法对给定消息进行消息摘要	`codec.HmacSM3("mysecretkey", "Message to hash")`

7.8 HTTP 协议基础库

HTTP（Hypertext Transfer Protocol）是互联网上最常用的应用层协议之一，负责在客户端和服务器之间传输数据。

HTTP 协议是 Web 应用程序的基础，许多应用程序、框架和库都使用 HTTP 协议进行通信。HTTP承载着在客户端和服务器之间传输数据的作用，是网络安全的一大重要关注点。Yak语言作为网络安全领域

的DSL，对HTTP的协议支持非常完善：除了本章节中介绍的“HTTP协议基础库”之外，在第八章中将会介绍一些非常高级的HTTP协议测试的库和用法，拥有一个高效易用的HTTP基础库是刚需。

本节将从“发送HTTP请求”、“控制HTTP请求配置”以及“处理HTTP响应”三个方面介绍Yak的HTTP协议基础库。

发送HTTP请求

HTTP协议的基本概念

- 请求与响应模型：HTTP采用请求-响应模型。客户端（如浏览器）发送请求到服务器，服务器处理该请求并返回响应。请求和响应都由消息构成，消息包含请求方法、URL、HTTP版本、头部信息和可选的消息体。
- 无状态性：HTTP是一个无状态协议，这意味着每个请求都是独立的，服务器不会记住之前的请求状态。这种设计提高了协议的可扩展性，但在需要保持会话状态的应用中，通常会使用Cookies等技术来实现状态管理。
- 可扩展性：HTTP协议通过头部字段的扩展，允许客户端和服务器之间进行功能扩展。开发者可以根据需要增加自定义头部，以便传递额外的信息。
- 版本演进：HTTP经历了多个版本的演变，从最初的HTTP/0.9到HTTP/1.0、HTTP/1.1，再到现代的HTTP/2和HTTP/3。每个版本都引入了新的特性和改进，例如HTTP/1.1支持持久连接，而HTTP/2则引入了二进制分帧和多路复用等技术，以提高性能。

快速开始

在Yak里发送一个简单快速的HTTP请求是很容易的，只需要一行代码。

```
rsp = http.Get("http://example.com")~
http.show(rsp)
```

这段代码将会向目标网址发送一个GET请求。并且在接收到响应之后通过 `http.show` 来展示完整的响应内容，完整相应内容如下：

```
HTTP/1.1 200 OK
Server: ECAcc (lac/55B8)
Accept-Ranges: bytes
Date: Sat, 24 Aug 2024 09:40:13 GMT
Etag: "3147526947"
X-Cache: HIT
Age: 303902
Content-Type: text/html; charset=UTF-8
Expires: Sat, 31 Aug 2024 09:40:13 GMT
```

```
Cache-Control: max-age=604800
Vary: Accept-Encoding
Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT
Content-Length: 1256

<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <style type="text/css">
    body {
        background-color: #f0f0f2;
        margin: 0;
        padding: 0;
        font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI",
"Open Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;

    }
    div {
        width: 600px;
        margin: 5em auto;
        padding: 2em;
        background-color: #fdfdff;
        border-radius: 0.5em;
        box-shadow: 2px 3px 7px 2px rgba(0,0,0,0.02);
    }
    a:link, a:visited {
        color: #38488f;
        text-decoration: none;
    }
    @media (max-width: 700px) {
        div {
            margin: 0 auto;
            width: auto;
        }
    }
    </style>
</head>

<body>
<div>
    <h1>Example Domain</h1>
```

```
55      <p>This domain is for use in illustrative examples in documents. You may
    use this
56      domain in literature without prior coordination or asking for permission.
    </p>
57      <p><a href="https://www.iana.org/domains/example">More information...</a>
    </p>
58  </div>
59  </body>
60  </html>
61
```

同样的还有发送POST请求的函数

函数名	函数说明	使用案例
`http.Get`	发送一个GET请求	`rsp = http.Get("http://example.com")~` 发送一个对 `http://example.com` 的Get请求
`http.Post`	发送一个Post请求	`rsp = http.Post("http://example.com")~` 发送一个对 `http://example.com` 的POST请求

这样简单的请求发送可能不易于解决一些复杂的场景，所以Yak还支持客户端式的HTTP请求发送。

```
1  req = http.NewRequest("HEAD", "http://example.com")~
2  rsp = http.Do(req)~
```

使用http.NewRequest函数建立一个请求，并使用http.Do发送出去。

函数名	函数说明	使用案例
`http.NewRequest`	新建Request对象	`req = http.NewRequest("HEAD", "http://example.com")~` 新建 `http://example.com` 的一个HEAD Request请求对象
`http.Do`	执行一次Request	`rsp = http.Do(req)~` 执行一个 Request请求对象

控制HTTP请求配置

`NewRequest` 函数在建立请求对象的时候可以指定请求方法，有了更好的HTTP请求控制。不过这还不够，HTTP请求中还有诸如请求头、请求体、代理等配置需要控制。所以Yak的HTTP库中还提供一些接口函数，来帮助用户可以快速进行HTTP请求配置，这些配置函数通常是“小写”的（区别于 `http.Get` 或 `http.Post` 等函数）。

函数名	描述	使用案例
`http.ua`	设置HTTP请求的 `UserAgent` 。	`http.Get('http://example.com', http.ua('Mozilla/5.0'))`
`http.useragent`	设置 `UserAgent` ，与 `http.ua` 功能相同。	`http.Get('http://example.com', http.useragent('Mozilla/5.0'))`
`http.fakeua`	设置一个伪造的 `UserAgent` 。	`http.Get('http://example.com', http.fakeua())`
`http.header`	配置HTTP请求的 `Header` 。	`http.Get('http://example.com', http.header({'Accept': 'application/json'}))`
`http.cookie`	为HTTP请求附加 `Cookie` 。	`http.Get('http://example.com', http.cookie({'session': 'abcd1234'}))`
`http.body`	设置HTTP请求的 `Body` （请求正文），用于POST请求传输数据。	`http.Post('http://example.com', http.body('key=value'))`
`http.json`	发送一个 `JsonBody` ，用于需要JSON格式数据的API请求。	`http.Post('http://example.com', http.json({'key': 'value'}))`
`http.params`	为URL添加 `GetParams` ，用于GET请求的查询字符串参数。	`http.Get('http://example.com', http.params({'key': 'value'}))`
`http.postparams`	为POST请求编码并设置 `PostParams` 。	`http.Post('http://example.com', http.postparams({'key': 'value'}))`
`http.proxy`	为HTTP请求设置代理服务器配置 `Proxy` 。	`http.Get('http://example.com', http.proxy('http://proxyserver:port'))`
`http.timeout`		

	为HTTP请求设置超时时间 `timeout` 。	`http.Get('http://example.com', http.timeout(30))`
`http.redirect`	配置重定向处理器 `RedirectHandler` ，用于处理HTTP请求的重定向。	`http.Get('http://example.com', http.redirect(True))`
`http.noredirect`	禁用自动重定向，允许手动处理HTTP重定向。	`http.Get('http://example.com', http.noredirect())`
`http.session`	维护跨多个HTTP请求的 `Session` 。	`http.Get('http://example.com', http.session())`

除了标准全面的HTTP请求相关函数外，YAK还提供了一些工具函数来帮助用户更快更舒适地处理HTTP相关内容。

函数名	描述
GetAllBody	获取响应数据包的响应体内容
dump	获取http数据包内容，类似 `sprint`
Show	输出http数据包内容，类似 `print`
dumphead	获取http数据包头部内容
showhead	输出http数据包头部内容

案例：使用代理访问某个网站

在本教程中，我们将学习如何使用 Yaklang 的 `http` 模块通过代理访问一个网站。我们将以访问 `http://www.example.com` 为例，展示如何设置代理并处理响应。

步骤 1：设置代理

首先，我们需要定义代理的地址。在本例中，我们将使用本地代理服务器 `http://127.0.0.1:7890` 。确保你的代理服务器已经启动并可以正常工作。

步骤 2：发送 GET 请求

使用 `http.Get` 函数发送 GET 请求，并通过代理进行访问。以下是具体的代码示例：

```
1 rsp = http.Get(
```

```
    "http://www.example.com",
    http.proxy("http://127.0.0.1:7890")
)
```

在这段代码中，我们调用 `http.Get` 方法，传入目标 URL 和代理地址。`rsp` 将保存服务器的响应。

步骤 3：处理响应

一旦我们获得了响应，接下来需要处理它。通常，我们会检查响应的状态码，并读取响应的内容。以下是如何显示响应的状态和内容的示例：

```
http.show(rsp)
```

`http.show(rsp)` 将输出响应的状态和内容，帮助我们了解请求的结果。

处理HTTP响应

HTTP 响应是与服务器进行通信的重要组成部分。通过处理 HTTP 响应，我们可以获取服务器返回的数据、状态信息以及其他有用的信息。当你向服务器发送请求时，服务器会返回一个响应。这个响应通常包含状态码、响应头和响应体。了解如何提取这些信息对于调试和数据处理非常重要。

在本小节中，我们将访问一个示例网站，获取其响应头和响应体，并展示如何输出这些信息。

步骤 1：发送 GET 请求

首先，我们需要发送一个 GET 请求到目标网址。使用 `http.Get` 函数可以轻松实现这一点。以下是代码示例：

```
rsp = http.Get("http://www.example.com")
```

这行代码将向 `http://www.example.com` 发送一个 GET 请求，并将响应保存到变量 `rsp` 中。

步骤 2：获取响应头

HTTP 响应的头部包含了关于响应的元数据，例如服务器类型、内容类型等。我们可以使用 `rsp.GetHeader` 方法来获取特定的响应头信息。在下面的代码中，我们获取了 "Server" 头信息：

```
serverHeader = rsp.GetHeader("Server")
println(serverHeader)
```

执行这段代码后，你将看到类似以下的输出：

```
ECAcc (lac/5586)
```

这表明服务器使用了 ECAcc 作为其处理请求的程序。

步骤 3：获取响应体

响应体包含了服务器返回的实际内容。在这个例子中，我们将使用 `rsp.Data()` 方法获取响应体，并输出其内容：

```
data = rsp.Data()
println(data)
```

这段代码将输出服务器返回的完整 HTML 内容，例如：

```
<!doctype html>
<html>
<head>
    <title>Example Domain</title>
    ...
    <h1>Example Domain</h1>
    <p>This domain is for use in illustrative examples in documents. You may use this
    domain in literature without prior coordination or asking for permission.</p>
    <p><a href="https://www.iana.org/domains/example">More information...</a></p>
</div>
</body>
</html>
```

将上述步骤结合起来，完整的Yak代码如下：

```
rsp = http.Get("http://www.example.com")

serverHeader = rsp.GetHeader("Server")
println(serverHeader)

data = rsp.Data()
```

```
println(data)
```

更多操作与接口列表

为了方便用户操作，我们将会制作一个表格，在表格中详细列出了 YakHttpResponse 接口中的所有成员，包括字段和方法。每个成员都有清晰的类型标注、功能说明和简单的使用示例。通过这个表格，你可以快速了解如何使用这个接口操作 HTTP 响应：

成员名	成员类型	成员解释说明	小案例
Response	普通成员	原始的 http.Response 对象	rsp.Response.StatusCode
Cookies	方法	返回响应中的 cookies	rsp.Cookies()
Location	方法	返回响应中的重定向地址	loc, _ := rsp.Location()
Data	方法	返回响应体的字符串形式	body := rsp.Data()
GetHeader	方法	获取指定的响应头	server := rsp.GetHeader("Server
Json	方法	将响应体解析为 JSON 对象	obj := rsp.Json()
Raw	方法	返回响应体的原始字节数组形式	bodyBytes := rsp.Raw()

参考资料：原始结构描述

```
type YakHttpResponse struct {
  Fields(可用字段):
      Response: *http.Response
  StructMethods(结构方法/函数):
      func Cookies() return([]*http.Cookie)
      func Location() return(*url.URL, error)
      func ProtoAtLeast(v1: int, v2: int) return(bool)
      func Write(v1: io.Writer) return(error)
  PtrStructMethods(指针结构方法/函数):
      func Cookies() return([]*http.Cookie)
      func Data() return(string)
      func GetHeader(v1: string) return(string)
      func Json() return(interface {})
      func Location() return(*url.URL, error)
      func ProtoAtLeast(v1: int, v2: int) return(bool)
      func Raw() return([]uint8)
      func Write(v1: io.Writer) return(error)
}
```