第六章：高级编程技术

在学习完Yak语言的基础概念和语法之后，笔者将会带领大家开始Yak语言的高级编程技术，本节包含并发异步功能的使用；延迟执行，同步控制，错误处理等重要概念。

6.1 协程与异步执行

6.1.1 什么是同步执行和异步执行

一般而言，计算机程序按照代码的执行顺序分为同步执行以及异步执行。

首先，在同步异步的主要区别就是是否等待程序操作完成，程序操作是指程序中的一段完成某个功能的代码，可以包括一到多行代码，比如输出信息、读写文件、网络操作等，这都是一个程序操作。

A. 同步执行：

在程序运行的过程中，程序将会等待操作的完成。程序内的所有操作将会从上到下一步步执行。比如定义一段程序如下：

```
1 println("hello")
2 for i=0; i<10;i++{
3     println("in loop:")
4 }
5 println("end loop ")
```

在Yak代码执行的过程中程序将会按照代码内定义操作的顺序依次执行，只有前一条操作执行完成再继续执行后续的操作。比如在示例代码中，有三个操作：打印hello，循环打印十次”in loop“， 打印”end loop“。 在顺序执行的程序中将会首先打印hello，然后循环打印“in loop”， 在循环结束以后打印“end loop"。

同步执行好处是顺序简单直观。

缺点是每一条代码都需要等待前面的代码执行完毕才可以执行，假如在程序中间存在一些需要时间才能完成的操作（比如文件的读取写入、进行网络请求等），那么只有在该操作完成后才可以继续执行后续的代码，整个程序会停住等待该操作的完成。这个整个程序停止等待操作完成的行为称为程序的阻塞。

B. 异步执行：

在异步执行中，程序执行一个操作以后，并不会等待该操作完成，而是继续执行后续的代码。当操作完成的时候将会通过各种手段通知程序该操作运行结束。

异步执行的好处在于可以有效地避免程序的阻塞，缺点是程序需要进行对操作的处理、错误处理、多操作的状态同步，将会比较复杂。

C. 同步执行和异步执行的使用

同步执行和异步执行都是为了完成程序的任务，只是执行顺序不同，各有优劣需要按需使用。

比如，当程序需要大量的数据读写操作，并且这些数据读写互相无关，那么使用异步执行进行读写可以使程序运行的更有效率，异步执行将会同时启动多个数据的读写操作，而不是等待一个数据读写操作完成在进行下一个。

相反，如果程序的操作之间有依赖关系，比如需要先读取配置文件，然后根据配置文件内容启动其他的操作，那么其他所有操作都需要等待配置文件读取这个操作执行，这时候需要使用同步执行。

6.1.2 异步执行的方式

计算机操作系统中的程序默认的执行顺序是同步执行的，因此一般情况下一般是在程序的某些互不相关的操作上使用异步执行，使得这些操作可以同时执行，并在一个合适的位置等待所有异步执行的操作执行结束，继续同步执行。

A.进程

在计算机操作系统中，每一个程序都是一个进程，进程之间是互相不影响的，比如浏览器和文件管理器就是两个不同的进程，他们互相毫无关系，关闭其中的一个也不会对另一个产生影响，而且两个进程都是同时在运行的。

计算机领域最早的异步编程就是使用的多进程的方式进行，也就是程序需要异步执行的操作单独开一个进程来运行，这样就可以原进程可以继续执行而不需要等待该异步操作，而异步操作也如同预期一样和原进程同时运行，操作结束以后新的进程也结束，原进程通过进程之间通讯来获得该操作的结果。这是异步执行最早的执行方案，通过这样一个效率不高的方案读者可以大致理解异步执行的程序行为。

但进程的创建、销毁是需要非常多计算机内存、运行时间的，在小型操作时使用进程实现的异步执行剩下的运行时间还不如进程的创建销毁浪费的时间多，另一方面，进程间的通讯也并不够好用。

多进程的系统中，程序也并非真正的同时运行，在单CPU计算机上，其实是通过进程的快速切换达到“在同一时间段内多个进程同时运行”的效果，这称为进程的并发，多CPU计算机可以将不同进程分配到不同CPU来达到“同一个时间点上多个进程在同时执行”的效果，也就是进程的并行。

需要注意的是，多核心单CPU的计算机其实不一定可以实现并行，如果所有的核心都使用同一套内存管理单元(MMU)和缓存机制，也只能在同一时间执行一个进程，这取决于多核CPU的硬件设计。早期的多核心CPU一般不能支持进程并发。

多进程程序是程序并行的一种形式，但是计算机操作系统中实现多进程更关注的在于进程之间的独立性，每个进程独立运行互不干扰。然而在代码编写的过程中所需要的异步执行只是避免阻塞等待提高程序效率，程序大部分时候只是需要异步计算一些数据或只是等待网络或文件读写响应，对于每一次

异步执行所要求的独立性更加低，每段异步执行的代码并不需要拥有太多数据。这一需求有两种解决方案。

B. 线程与协程

首先在操作系统层面提供了线程，线程存在于进程内，一个进程可以启动多个线程同步执行程序，同一个进程内的多个线程之间共享地址空间，因此在使用上多线程的切换效率和通讯要比多进程更加方便。

操作系统提供的线程会在程序运行的用户态切换到操作系统内核中完成线程的切换，因此，操作系统进一步提供了用户态线程，用户态线程不需要经过操作系统内核就可以进行上下文切换。

同时在用户态也出现了协程的实现，协程与用户态线程非常类似，他们的切换都是在用户态进行的，线程是系统提供的，但是协程是用户态的代码提供的，并且协程可以由程序编写人员控制协程间的切换时机。现在活跃语言都拥有协程的实现，其中有些是通过第三方库实现有些是语言原生支持的。在Yak中提供了原生支持的协程。

6.1.3 如何在Yak中使用异步编程

在Yak中， 协程运行的基本单位是一个函数，创建协程异步执行的语法和普通的函数调用类似，只需要在开头加上"go"关键字即可。以下是语法示例：

```
go 函数名(参数列表)
```

以下是一个简单的代码案例：

```
func count() {
    for i := 1; i <= 5; i++ {
        println("count function:\t", i)
        sleep(1)
    }
}

count()
for i=1; i<=5; i++ {
    println("Main function:\t", i)
    sleep(1)
}
sleep(1)
```

在这个例子中， `count` 是一个函数，他的作用是循环5次打印 `count function: i` ， 在程序运行的时候，将会直接调用该函数，函数执行结束后将会继续执行后续代码，仍然循环5次 打印 `Main`

`function : i`， 程序将会产生以下输出：

```
count function:  1
count function:  2
count function:  3
count function:  4
count function:  5
Main function:   1
Main function:   2
Main function:   3
Main function:   4
Main function:   5
```

可以看到，一直等待到`count`函数执行结束才继续向后运行后续的循环，这就是一个同步执行的示例。

接下来，在函数调用的时候加入"go"关键字，将会使`count`函数异步执行, 异步执行代码示例如下：

```
func count() {
    for i := 1; i <= 5; i++ {
        println("count function:\t", i)
        sleep(1)
    }
}

go count()
for i=1; i<=5; i++ {
    println("Main function:\t", i)
    sleep(1)
}
sleep(1)
```

这一示例执行结果如下：

```
Main function:   1
count function:  1
Main function:   2
count function:  2
count function:  3
Main function:   3
count function:  4
Main function:   4
```

```
Main function:  5
count function:  5
```

可以看到程序将不会等待 `count` 的执行结束直接开始执行后续代码，而 `count` 函数也同样在执行，两段循环在同时执行，这就是 `count` 函数在异步执行的效果。

6.2 延迟运行函数：Defer

在编程中，有时程序希望在函数执行完成后执行一些清理操作或释放资源的操作。例如，可能需要在打开文件后关闭文件，或者在数据库操作后关闭数据库连接。延迟执行机制提供了一种方便的方式来处理这些情况。

延迟执行的基本单位也是函数，通过在函数调用前增加"defer"关键字，可以指定某个函数调用延迟执行，使这些函数调用将会在当前函数返回时自动执行。

这意味着无论函数中的控制流如何，这些延迟语句都会在函数返回之前被执行。这种机制可以确保无论函数是正常返回还是发生了异常，在该函数执行完毕后都一定会进行设置好的必要清理工作。

> 需要注意的是，在Yak中执行编写代码，默认是写入在主函数内的。因此在此时也可以直接使用“defer”延迟执行，在主函数内所有代码(也就是编写的代码)全部执行结束后，将会自动调用设置的延迟执行函数。

以下为延迟执行的关键字"defer"语法。

```
defer 函数名(参数列表)
```

6.2.1 创建延迟函数

以下是一个简单的代码样例：

```
println("statement 1")
defer println("statement 2")
println("statement 3")

subFunc1 = func(msg) {
    println("in sub function 1: ", msg)
}
subFunc2  = func() {
    defer subFunc1("call from subFunc2 defer")
    subFunc1("call from subFunc2")
}
subFunc2()
```

在这个示例中，在主函数，程序在defer关键字进行延迟执行 `println("statement 2")` 这次函数调用，定义 `subFunc1` 和 `subFunc2` 两个函数， 主函数将会调用 `subFunc2` 函数，并在 `subFunc2` 中，通过普通调用和延迟调用来拿各种功能方式调用 `subFunc1` 函数。当运行此程序的时候，将会产生以下输出：

```
1 statement 1
2 statement 3
3 in sub function 1:  call from subFunc2
4 in sub function 1:  call from subFunc2 defer
5 statement 2
```

可以看到使用"defer"关键字进行延迟执行的函数在整个函数执行结束以后才运行， 对于主函数来说，"statement 2"是在所有代码执行完毕后运行的； 对于 `subFunc2` 来说，带有"defer"的调用是在该函数运行结束返回的时候运行的。其他的普通函数调用将会按照代码顺序执行。

6.2.2 多个延迟函数

程序可以设置多个延迟函数，这些延迟函数将会被保存在一个先入后出的栈结构内，程序结束以后，将会依次从栈中弹出执行，也就是多个函数将会优先执行后定义的函数，从后向前执行。接下来的例子将会展示这一特性：

```
1 println("statement 1")
2 defer println("statement 2")
3 defer println("statement 3")
4 defer println("statement 4")
5 println("statement 5")
```

程序运行结束以后，将会从后向前执行定义好的延迟函数，该代码示例运行结果如下：

```
1 statement 1
2 statement 5
3 statement 4
4 statement 3
5 statement 2
```

6.2.3 程序出错时也会运行延迟函数

程序出错时也会执行其中的延迟函数，接下来的示例将会展示这一特性：

```
defer println("defer statement1 ")
a = 1 / 0
defer println("defer statement2 ")
```

该程序运行到 `1/0` 的时候，将会触发错误，程序将崩溃，而此时已经通过defer关键字设置了"defer statement1"的延迟函数调用，于是，该代码示例运行结果如下：

```
defer statement1
Panic Stack:
File "/var/folders/8f/m14c7x3x1c55rzvk5qvvb1w00000gn/T/yaki-code-287898179.yak", in __yak_main__
--> 2 a = 1 / 0

YakVM Panic: runtime error: integer divide by zero
```

值的注意的是，由于该代码在第二行 `1/0` 崩溃，因此第三行中的defer并没有被执行，也就不会被调用，仅有第一句中设置的延迟函数被调用了。

6.2.4 小结

函数延迟执行是一个简单但是实用的机制，通过延迟执行可以保证数据清理和资源释放操作。在本章(第六章)后续的错误处理和并发控制两个小节将会介绍更加具体两种的使用。

6.3 函数的直接调用

在第五章中已经详细讲解了函数的创建方式和直接的调用，一个代码的样例如下：

```
func a() {
    println("in sub function 1")
}
a()
```

在很多时候，临时的函数不一定需要被定名，可以直接定义函数并调用。比如如下的代码：

```
func() {
    println("in sub function 2")
}()
```

在Yak中进行协程创建和延迟运行的时候都需要编写一个函数调用，很多时候会创建一个简单的临时函数，并不给他定名然后调用，将会编写类似上述示例的代码，Yak对这种情况提供了更加简单的方案：

```
func {
    println("in sub function 3")
}
```

这样的函数等同于上述的两个函数调用。

在go关键字和defer关键字后，也可以编写这样的代码：

```
defer func {
    println("in defer")
}
go func {
    println("in go")
}
println("sleep 1")
sleep(1)
```

这样的程序似的程序编写的更加简洁，他的运行和定义函数进行调用是等效的，运行结果如下：

```
sleep 1
in go
in defer
```

6.4 并发控制：sync

在本章中已经提到协程的创建和使用，创建一个协程的开销很低，远远低于线程，是进行异步编程的重要基础。但在Yak的异步编程中，需要考虑两个问题：

- 异步编程处理数据的时候，需要有手段可以等待所有期望的协程结束，然后收集资源。
- 创建协程是有开销的，无限制地去创建协程只会让资源被白白浪费掉。需要有手段可以控制某个功能创建协程的数量上限。

对于协程的并发控制，在Yak提供了许多并发控制的支持。

6.4.1 等待异步执行：`WaitGroup`

下面请看这样一段代码样例：

```
for i in 16 {
    num = i
    go func{
        sleep(1)
        println(num)
    }
}
println("for statement done!")
```

在这段代码样例中，首先运行循环，在循环内创建协程打印数据，并且每个协程都会调用sleep函数等待1秒，最后打印循环结束的字符串。

但是这段程序的运行结果如下：

```
for statement done!
```

可以看到只有程序最后的输出。

出现这一情况的原因是，协程都是互相独立运行的，主程序也是一个单独的协程，程序在循环中创建了16个协程，算上主协程一共有17个协程，在循环内创建的协程都会等待1秒然后在打印数据，而主协程将会继续执行，主协程执行结束之后整个程序将会停止运行，所有的其他协程都会被销毁。

为了解决这样一个问题，Yak提供了等待协程的工具：WaitGroup，以下的代码展示了WaitGroup的使用，并通过这一工具解决了前一个代码示例中存在的问题。

```
wg = sync.NewWaitGroup()
for i in 16 {
    num = i
    wg.Add()
    go func{
        defer wg.Done()
        println(num)
    }
}
wg.Wait()
println("for statement done!")
```

首先创建一个WaitGroup实例。

每次创建协程的时候，调用 `wg.Add` 方法，表示增加一个需要等待的协程，在协程内，使用defer延迟函数的形式保证调用 `wg.Done` 方法，表示一个需要等待的协程已经结束。

最后使用 `wg.Wait` 等待所有注册的协程结束。协程运行到该函数的时候将会阻塞等待，直到所有需要等待的协程都结束，才会继续向后执行。

这段代码运行结果如下：

```
0
2
7
13
11
8
1
15
3
6
9
5
14
12
10
4
for statement done!
```

6.4.2 控制协程数量： `SizedWaitGroup`

`sync.NewSizedWaitGroup` 是Yak并发控制中一个重要库函数，接受一个字符作为参数表示协程的容量上限，返回一个 `SizedWaitGroup` 对象。可以简单地认为 `SizedWaitGroup` 是一个计数器，计数器的值就是协程的数量。程序可以通过 `Add` 方法使计数器的值增加，使用 `Done` 方法使计数器的值减少。如果计数器的值增加到了设置的容量上限，那么 `Add` 函数就会堵塞到计数器的值减少为止。

以下的例子演示了 `SideWaitGroup` 的简单使用：

```
swg = sync.NewSizedWaitGroup(1)
for i in 16 {
    num = i
    swg.Add(1)
    go func{
        defer swg.Done()
        println(num)
    }
}
swg.Wait()
```

在上述示例中，首先创建了一个SizedWaitGroup，容量上限为1。程序运行循环16次，并在每次循环中执行swg.Add(1)，然后创建一个协程打印当时的循环计数器，在协程函数内，使用defer进行延迟运行在协程退出的时候执行swg.Done()表达异步执行结束。最后在循环外使用swg.Wait()等待这个计数器归零表示所有协程都运行结束。

同时因为协程都是独立执行，将会分别打印数据，原本应该乱序输出0到15这几个字符，但是现在SizedWaitGroup对象上限为1， 也就是允许并发执行的协程最多为1， 当第一个协程调用 `swg.Add(1)` 的时候， `SizedWaitGroup` 到达上限，下次循环运行的时候将会在 `swg.Add(1)` 阻塞，等待第一个协程运行结束调用 `swg.Done()` 后继续运行，也就导致每一个循环都需要等待前一个循环内的协程运行结束才会运行。这段程序虽然使用了协程运行，但是会表现出同步运行的特性，得到的结果将会是顺序打印0到15。

sync库还提供了很多其他函数可以帮助我们完成并发控制，详情可以查看官方文档。

6.5 通道类型与并发编程：channel

在并发编程中，通信和数据共享是一个核心的问题。Yak语言引入了一种特殊的数据类型 - channel，它就像是一个邮局，可以帮助不同协程之间轻松地发送和接收数据。

本书第三章3.3.3已经简单介绍了通道类型的简单使用，本章节将继续深入探讨Yak中的channel，让读者更好地理解和使用这个强大的工具。

6.5.1 缓冲区和阻塞

可以将 Channel 理解为一个先入先出的管道，同时可以从一侧放入数据另一侧拿出数据，缓冲区表示在这个管道内保存的数据可以有多少。

我们使用一个程序示例讲解这个特性：

```
ch = make(chan int, 2) // 创建Channel，缓存区为2

ch <- 1 // 写入数据 此时缓存区[1]
ch <- 2 // 写入数据 此时缓存区[1, 2]
// ch <- 3 // 写入数据 此时缓存区已经满 将会阻塞等待有数据取出才能写入
println(<- ch) // 取出数据 1 此时缓存区[2]
println(<- ch) // 取出数据 2 此时缓存区[]
// println(<- ch) // 缓存区为空 将会阻塞等待数据写入
```

当缓存区满时，需要等待取出数据才可以继续向 channel 写入数据，当缓存区空时需要等待写入数据才可以从 channel 取出数据。

同样，如果没有设置缓冲区，无缓存区，表示缓存区大小默认为 0，此时只有两端同时读写才不会出现阻塞等待，否则无论是读还是写都会出现等待。

另一个需要注意的点是，当缓存区空以后继续尝试读取数据，如果是未关闭的 Channel 会导致阻塞等待，关闭的 Channel 则会直接返回nil, false`，当使用for-range或for-in进行数据遍历的时候，当缓存区为空时，未关闭 Channel 一样会等待，已关闭的 Channel 则会跳出循环。在不需要再数据写入的时候，应该关闭 Channel。

6.5.2与协程一起工作

单独使用 Channel 的阻塞特性可能让人奇怪，但是如果和协程一起工作，则会形成非常高效的并发通讯。

```
ch1 = make(chan int)
ch2 = make(chan int)
go fn{
    for i=0; i<100; i++ {
        ch1 <- i // 在协程中生成0-100写入Channel中
    }
    close(ch1) // 第一阶段数据写入结束 关闭ch1
}
go fn{
    for {
        i, ok := <- ch1 // 获取数据
        if !ok {
            break // 当 close(ch1)以后 ok=false
        }
        ch2 <- i + 2 // 从ch1中获取到的数据运算继续写入ch2
    }
    close(ch2) // 第二阶段数据写入结束 关闭ch2
}
for i = range ch2 { // 通过for-range读取ch2中的数据
    println(i)
}
```

以上的示例中，展示了协程之间数据传输的方案。首先创建两个channel并创建两个协程，第一个协程向 `ch1` 中写入0到100，第二个协程从 `ch1` 中读取数据，运算并写入 `ch2` ， 最后在

数据写入通过ch <- 1进行，数据写入结束以后通过 `close(ch)` 关闭channel。

数据读取在代码示例中使用了两种方法：

- 循环使用v, ok := <- ch并判断!ok的方案，可以读取 ch 内写入的所有数据，直到 Channel 关闭；
- 另一种方案使用for v = range ch或for v in ch通过循环遍历获取数据，同样是获取 Channel 内写入的所有数据，直到 Channel 关闭。

该代码样例将会打印从2到101的数据，并且由于使用通道进行数据传输，数据保持先入先出的原则，即使是在不同线程数据将会按照顺序打印。

6.6 错误处理

在程序运行中，可能会出现很多的错误以及崩溃。

比如以下这个代码样例：

```
a = 1/0
println("after a = ", a)
```

程序运行时将会产生以下的信息：

```
Panic Stack:
File "/var/folders/8f/m14c7x3x1c55rzvk5qvvb1w00000gn/T/yaki-code-3822814179.yak", in __yak_main__
--> 1 a = 1/0

YakVM Panic: runtime error: integer divide by zero
```

这就是在提示程序发生了崩溃。当崩溃发生时，程序将会直接退出，不会执行后续的代码，在这个示例中可以看到后续的 `println("after a = ", a)` 并没有被执行。

在大多数时候程序的程序的崩溃是应当被恰当处理的，当崩溃被处理之后，程序将不会直接退出，而是继续执行崩溃处理代码之后的代码，这就保证了代码的健壮。

崩溃可能由各种原因引发，包括运行时崩溃（例如，尝试从长度为零的列表中取值）和手动触发的崩溃（例如，调用 panic 或 die 函数）。

另一方面，还有许多 Yak 的库函数是可能执行失败的，这些标准库函数不能产生崩溃，而是使用返回值的方式表达执行的失败，这些函数通常会在返回值的最后有一个 error 类型的值。如果函数调用没有出现错误，此返回值为 nil；如果函数发生错误，将返回一个包含错误信息的 error 对象。调用该函数的时候，只需要通过判断其最后一个返回值是否为nil即可判断该函数执行是否成功。

6.6.1 处理错误

当调用可能返回错误的函数时，Yak 的常见处理方式如下：

```
ret, err = func(arg)
if err != nil {
    // 处理错误，通常是返回或触发崩溃
}
```

如果函数返回错误，则该函数调用的其它返回值通常不可信。在大多数情况下，发生错误后应终止当前函数的后续执行，以防止错误的返回值引发后续代码的问题。
Yak 中常用 panic 函数将错误转换为崩溃。函数 panic 接受一个参数，该参数就是崩溃信息。函数 die 也接受一个参数，但它会检查该参数是否为 nil。如果参数不为 nil，则直接调用 panic；否则，不执行任何操作。

Yak 还支持一种简化的错误处理语法。在函数调用后使用 `~`，表示此函数可能返回错误，并检查其最后一个参数（即返回的错误）是否为 nil。如果不为 nil，则调用 panic 函数。

以下是三种错误处理方式的等价代码：

```
// 使用 panic 进行错误处理
ret, err = func(arg)
if err != nil {
    panic(err)
}
// 使用 die 进行错误处理
ret, err = func(arg)
die(err)
// 使用 ~ 进行错误处理
ret = func(arg)~
```

注意，不仅在调用可能返回错误的函数时可以使用 panic，而且在代码的任何位置都可以使用 panic 来表示错误或崩溃。

6.6.2 使用Panic和Recover进行错误处理

如前文所述，panic 函数会触发一个崩溃。当崩溃发生时，它会立即终止当前函数并返回到上层函数，一层层向上返回。当返回到最外层时，程序会直接崩溃。
recover 是一个函数，当调用它时，它会立即检查是否存在崩溃的向上传递。如果存在，recover 就会捕获这个崩溃，并停止向上的传递，同时返回这个崩溃的信息。
一般来说，发生时，将停止执行任何后续语句，并向上层函数传递。当函数退出时，无论是否存在崩溃，都会运行函数的 defer 语句。因此，通常会使用 defer 和 recover 配合进行错误处理。
以下是一个代码示例：

```
defer fn{
    println(recover()) // 设置 main 函数的错误处理
}
a = () =>{
    panic("panic in a") // 触发错误
}
b = () => {
    defer fn{
```

```
        println("defer in b") // b 函数的延迟函数
    }
    a()
    println("after a function call ") // 当 a 函数调用结束
}
b()
```

在这段代码中，首先在 main 函数中设置了一个 defer 函数来处理错误。然后调用函数 b，函数 b 再调用函数 a，函数 a 调用 panic 触发崩溃。这会立即退出函数 a 并返回到函数 b，然后运行函数 b 的 defer 函数打印信息并立即退出函数 b，返回到 main 函数。最后，在运行 main 函数的 defer 函数时，recover被调用，捕获并打印错误。
这段代码的运行结果如下：

```
defer in b
panic in a
```

6.6.3 使用try-catch处理崩溃

Yak 将错误通过 panic 和相关语法转换为崩溃。对于崩溃的处理，Yak 提供了两种方式。前面一小节已经介绍了recover的崩溃处理方案，这一小节将介绍第二种方案，即 try-catch 模式。语法如下：

```
try {
    // 代码
    // 如果此处出现崩溃，则直接跳到 catch 代码块执行
} catch err {
    // 崩溃处理代码
    // 崩溃信息即为 panic 的参数，可以通过 err 变量获取
} finally {
    // 清理代码
    // 无论 try 或 catch 代码块执行结束后，都会执行 finally 代码块
}
```

在 catch 语句中，崩溃信息存储在名为 err 的变量中。如果不需要获取该信息，可以省略 err。
以下是一个代码示例：

```
try {
    println("We are in Trying")
    panic("panic in try!")
} catch err {
    println("Fetch Error" + f": ${err}")
```

```
} finally {
    println("working in finally")
}
```

这段代码首先在 try 语句中主动调用 panic 触发崩溃，因为在 try 语句内，所以会跳转到 catch 代码块执行，其中的错误信息存储在 err 变量中。最后，无论是否发生崩溃，都会执行 finally 代码块。代码示例的输出如下：

```
We are in Trying
Fetch Error: panic in try!
working in finally
```

6.7 作用域

作用域（也被称为冲突域）是一个非常重要的概念，它在编程语言中起到了关键的作用。一般来说，作用域是一个区域，它规定了在该区域内定义的变量、函数、和对象的可见性和生命周期。换句话说，作用域定义了在哪里和在何时可以访问一个变量或者一个实体。

为了更好地理解作用域的概念，以下是一个代码示例：

```
globalVar = "I'm global"; // 全局作用域

{
    localVar := "I'm local"; // 局部作用域

    println(globalVar); // 输出 "I'm global"
    println(localVar); // 输出 "I'm local"
}

println(globalVar); // 输出 "I'm global"
println(localVar); // 报错，localVar 在此作用域内未定义
```

在以上代码中，`globalVar` 是在全局作用域中定义的，所以它可以在代码的任何地方被访问。而 `localVar` 是在 `{}` 包裹的代码块的作用域内定义的，所以它只能在代码块内部被访问。当试图在代码块外部访问 `localVar` 时，程序会报错，因为 `localVar` 在外部的作用域内未定义。

这个例子展示了作用域如何控制变量的可见性和生命周期。在作用域之后，局部变量 `localVar` 就会被销毁，因为它的生命周期限制在了代码块作用域内。此外，由于 `localVar` 只在代码块作用域内可见，所以它不会影响到函数外部的任何代码，这就避免了可能的名称冲突。

在编程时，理解和正确使用作用域是至关重要的，因为它能够帮助编程者编写出结构清晰、易于维护的代码。通过有效地利用作用域，编程者可以控制变量、函数和对象的可见性和生命周期，从而提高代码的可读性和可维护性。

6.7.1 作用域的嵌套关系

Yak中很多语句都会创建自身的作用域，一般来说可以认为每一对 `{}` 内包裹的代码块都是一个新的作用域，同时Yak中的作用域使用嵌套关系组织，如果某个标识符在当前作用域找不到，则会在父作用域中查找。

以下的代码示例简单讲述了作用域的嵌套以及从父作用域中获取变量，两个函数都创建了自己的新作用域，并且使用了无法找到的标识符，则在父作用域查找，得到对应的值。

```
a = 1
f = () => {
    println("a:", a)
    b = 2
    f2 = () => {
        println("b:", b)
    }
    f2()
}
f()
```

代码运行结果如下：

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

因为Yak所有标识符都是变量，因此除了父作用域标识符的使用，在嵌套的作用域内，还可以进行父作用域标识符数据的修改。

```
a = 1
println("a = ", a)
{
    a = 2
    println("in block a = ", a)
}
println("after block a = ", a)
```

比如以上的代码示例，其中在 block 内是一个新的作用域，使用变量 a 的时候将会直接修改外部的 a 变量。此段代码运行结果如下：

```
a =  1
in block a =  2
after block a =  2
```

6.7.2 强制创建局部变量

Yak的标识符可向上查找和修改的特定使得闭包函数等操作实现较为简单，但是在一些情况下，我们希望内部变量和外部变量进行区分，也就是避免名称冲突的问题。

Yak提供 `:=` 赋值语句，表示在当前作用域强制创建一个全新的标识符，并忽略上层作用域是否存在同名的标识符。

在下面的代码示例中可以清晰的了解到强制赋值的特性。

```
a = 1
println("a = ", a)
{
    a := 2 // 这里使用强制赋值
    println("in block a = ", a)
}
println("after block a = ", a)
```

和之前修改父作用域变量的代码一致，区别只是将原本的赋值修改为了强制赋值。则{}内部的 a 和外部的 a 并不是同一个变量，对内部 a 标识符的任何操作也不会修改外部的 a。

运行结果如下：

```
a =  1
in block a =  2
after block a =  1
```

6.8 模块化和多文件编程

在考虑模块化和多文件编程时，我们经常需要根据位置定位文件和资源目录。Yak提供三个全局变量用于支持此功能。

- `YAK_MAIN`：bool 类型数据，只有当文件主动调用运行时会设置为 true，其他文件导入本文件时被设置为 false。

- `YAK_FILENAME`：当前执行脚本文件的具体文件名。
- `YAK_DIR`：当前执行脚本文件所在路径的位置。

6.8.1 导入变量：import函数

函数定义如下：

```
1 func import(file, exportsName) (var, error)
```

当我们调用此函数的时候，将会把对应的文件载入到Yak代码中，并把变量名为 `exportsName` 的变量导出，如果执行失败，返回值将会返回 `(nil, error)` ，

一个例子如下, 首先创建 `lib.yak` 脚本：

```
1 func callee(caller) {
2     println("callee is called by", caller)
3 }
```

创建main.yak脚本：

```
1 res, err = import("lib", "callee")
2 die(err)
3
4 res("main.yak")
```

执行文件main.yak时，将会从当前目录下找到lib.yak文件，并引入其中名为callee的变量，然后当作函数调用，将会打印如下内容。

```
1 callee is called by main.yak
```

6.8.2 导入另一个脚本：include

include 只有脚本执行前执行，一定位于代码的最前面，include 相当于把目标文件直接复制到当前脚本中，一起执行。

一个例子如下, 首先创建lib.yak脚本：

```
func callee(caller) {
    println("callee is called by", caller)
}
```

创建main.yak脚本：

```
include "lib.yak"
callee("main.yak")
```

执行文件main.yak时，将会从当前目录下找到lib.yak文件，并将文件内容替换掉 include 语句，然后后续代码可直接使用在此文件中的所有内容，上述的两个文件将会形成如下的代码

```
func callee(caller) {
    println("callee is called by", caller)
}
callee("main.yak")
```

以上代码运行将会打印如下内容。

```
callee is called by main.yak
```

6.8.3 判断是否被导入：YAK_MAIN

当调用运行一个 Yak 脚本时，此脚本内的YAK_MAIN全局变量则会设置为 true。如果使用被其他的包导入时则会被设置为 false。我们仍然使用前面所述的例子，但是对两个函数都加入 YAK_MAIN 的判断。

首先创建lib.yak脚本如下：

```
func callee(caller) {
    println("callee is called by", caller)
}
if YAK_MAIN {
    println("i am in lib block")
}
```

使用 include 语法的main.yak脚本如下：

```
include "lib.yak"
callee("main.yak")
if YAK_MAIN {
    println("i am in main block")
}
```

执行文件main.yak时， include 将会将lib.yak脚本内内容完全复制，因此lib.yak内的判断YAM_MAIN代码实际上是在main.yak内运行的，因此也为 true。执行结果如下：

```
i am in lib block
callee is called by main.yak
i am in main block
```

使用 import 语法的main.yak脚本如下：

```
res, err = import("lib", "callee")
die(err)

res("main.yak")
if YAK_MAIN {
    println("i am in main block")
}
```

此时使用 import 语法，则在lib.yak中的YAK_MAIN为 false，不会运行对应判断内代码。输出如下：

```
callee is called by main.yak
i am in main block
```

6.9 模糊文本渲染: FuzzTag

6.9.1 什么是FuzzTag

FuzzTag是yaklang内置的一种基于模糊文本生成引擎实现的tag语法，能够灵活地嵌入数据中，实现数据的模糊生成和数据加工。该特性可广泛应用于渗透测试中的fuzz测试过程。yaklang的模板字符串支持fuzztag语法的使用，可以方便的生成测试数据。

6.9.2 语法规则

一个合法的FuzzTag由标签边界、标签名、标签数据组成。如 `{{int(1-10)}}` ，其中 `{{` 和 `}}` 标志着标签的开始和结束，int是标签名，1-10是标签参数，在引擎工作时会将标签参数作为参数传递给标签函数，生成数据。int标签用以生成指定范围内的数字。

可以在Yak语言中使用模板字符串观察FuzzTag行为，如：

```
dump(x"{{int(1-10)}}")
```

输出：

```
([]string) (len=10 cap=10) {
 (string) (len=1) "1",
 (string) (len=1) "2",
 (string) (len=1) "3",
 (string) (len=1) "4",
 (string) (len=1) "5",
 (string) (len=1) "6",
 (string) (len=1) "7",
 (string) (len=1) "8",
 (string) (len=1) "9",
 (string) (len=2) "10"
}
```

示例中使用int标签，传递参数为1-10，生成结果为长度为10的string列表，其元素为string类型的1到10数字。示例中使用1-10作为标签参数，参数由标签函数自主解析，所以每个标签对标签数据格式有着不同的规范，但在设计上是易用性优先如 `1-10` 比 `1,10` 更直观。总体上标签参数遵循通用规范，对于多个参数通常是通过|对多个参数拼接构成标签参数。如int标签支持1到3个参数，三个参数含义分别是数字范围，数字长度,步长，其中数字范围是必选参数，数字长度默认为自动，步长默认为1。

测试代码： `fuzztag_test2.yak`

```
dump(x"{{int(7-13|2)}}")
dump(x"{{int(7-13|2|2)}}")
```

输出：

```
([]string) (len=7 cap=7) {
 (string) (len=2) "07",
 (string) (len=2) "08",
 (string) (len=2) "09",
```

```
  (string) (len=2) "10",
  (string) (len=2) "11",
  (string) (len=2) "12",
  (string) (len=2) "13"
}
([]string) (len=4 cap=4) {
  (string) (len=2) "07",
  (string) (len=2) "09",
  (string) (len=2) "11",
  (string) (len=2) "13"
}
```

6.9.4 数据嵌入

FuzzTag支持在数据中嵌入，在FuzzTag生成多个字符串时，默认会将每个字符串在原FuzzTag位置做替换，生成多条数据。

测试案例:fuzztag_test3.yak

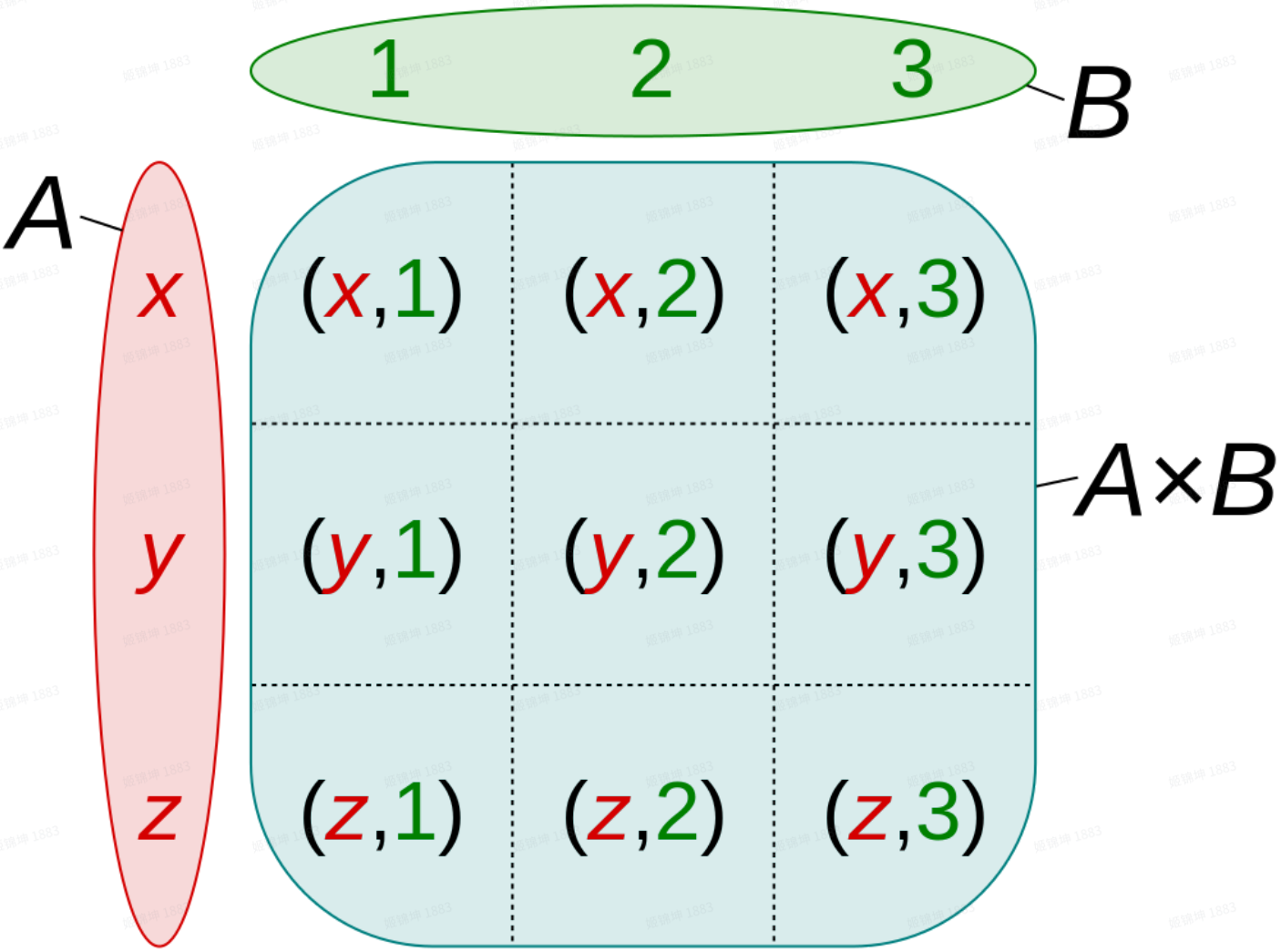
```
dump(x"id={{int(1-5)}}")
```

输出：

```
([]string) (len=5 cap=5) {
  (string) (len=4) "id=1",
  (string) (len=4) "id=2",
  (string) (len=4) "id=3",
  (string) (len=4) "id=4",
  (string) (len=4) "id=5"
}
```

6.9.5 多标签渲染

如果一段数据中存在了多个FuzzTag，默认渲染行为是将多个FuzzTag的渲染结果进行笛卡尔乘积后嵌入数据中。笛卡尔乘积原理如图：



测试案例:fuzztag_test3.yak

```
dump(x"id1={{int(1-2)}}&id2={{int(1-2)}}")
```

输出:

```
([]string) (len=4 cap=4) {
 (string) (len=11) "id1=1&id2=1",
 (string) (len=11) "id1=1&id2=2",
 (string) (len=11) "id1=2&id2=1",
 (string) (len=11) "id1=2&id2=2"
}
```

6.9.6 同步渲染

有些场景下多个标签之间存在对应关系，如在爆破账号时，字典的用户名列表为root、admin，密码列表为: root_123456、admin_000000，即用户名与密码存在一一对应的关系。这种场景下适合使用

同步渲染语法：在两个需要一一对应的标签名后加上相同的label名，如{{int::number(1-3)}} {{int::number(1-3)}}，执行结果为: 11、22、33。

以用户名密码爆破为例（array标签用于将多个参数生成列表），fuzztag_test4.yak：

```
dump(x"user={{array::user_password(root|admin)}}&password=
{{array::user_password(root_123456|admin_000000)}}")
```

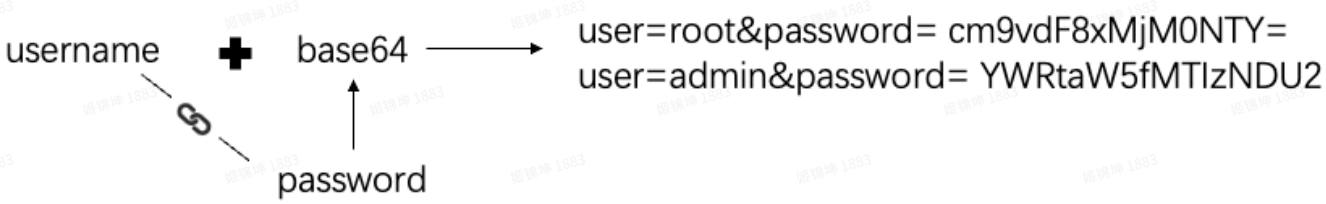
输出：

```
([]string) (len=2 cap=2) {
 (string) (len=30) "user=root&password=root_123456",
 (string) (len=32) "user=admin&password=admin_000000"
}
```

在一些特殊场景下，如密码需要使用base64编码，可能fuzz脚本为:

```
user={{array::user_password(root|admin)}}&password=
{{base64({{array::user_password(root_123456|admin_000000)}})}}
```

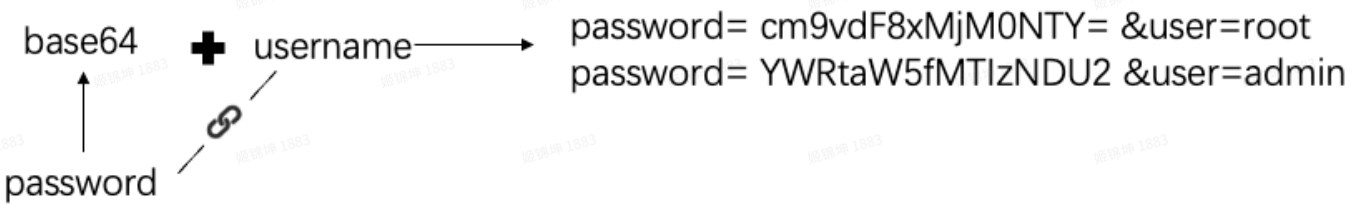
案例中的用户名和密码标签进行了同步，如图



在实际渲染过程中是按照从左向右的顺序执行标签，在执行用户名标签后会检查与之同步的标签，案例中会检查到password标签，再对password标签进行执行，password标签执行后会将执行结果抛给外层base64标签继续执行，最后对所有生成结果按照文本顺序进行拼接，得到渲染结果。

如果将两个标签调换顺序，如：

```
password={{base64({{array::user_password(root_123456|admin_000000)}})}}&user=
{{array::user_password(root|admin)}}
```



那么生成流程将变为base64标签先执行->调用子标签password->调用同步标签username,最后拼接为渲染结果。

上面两个案例看起来很合理，生成数据符合预期，但如果在更复杂场景，例如：

```
password={{array(aaa|
{{array::user_password(root_123456|admin_000000)}})}}&user=
{{array::user_password(root|admin)}}
```

按照执行顺序，第一个array标签在执行时，会调用子标签生成root_123456、admin_000000，子标签生成参数aaa|root_123456、aaa|admin_000000，最后array标签生成数据为aaa、root_123456、aaa、admin_000000。所以与user标签同步渲染后的结果为

```
password=aaa&user=root
password=root_123456&user=root
password=aaa&user=root
password=admin_000000&user=root
password=aaa&user=admin
password=admin_000000&user=admin
password=aaa&user=admin
password=admin_000000&user=admin
```

如果再调换顺序为：

```
user={{array::user_password(root|admin)}}&password={{array(aaa|
{{array::user_password(root_123456|admin_000000)}})}}
```

则生成结果变为了

```
user=root&password=aaa
user=admin&password=aaa
user=root&password=aaa
user=admin&password=aaa
```

```
5 user=root&password=admin_000000
6 user=admin&password=admin_000000
```

原因是第一次user标签执行时，通过同步调用了password标签，然后password标签将执行结果抛给外层array，生成了aaa、root_123456，但是user标签只与第一个aaa拼接，生成了`user=root&password=aaa`，第二次执行同理，直到第四次执行，password标签执行结束，但外层array标签缓存了上一次的执行结果`admin_000000`，最后拼接为`user=root&password=admin_000000`、`user=admin&password=admin_000000`。

6.9.7 常用标签与列表

前面案例介绍了int、array标签，它们都是用于数据生成，除了这类标签还有一类可以用于数据加工，如base64标签会对标签数据进行base64编码。如：{{base64(yaklang)}}，输出为：eWFrbGFuZw==。以下是全部标签介绍列表：

标签名	标签别名	标签描述
array	list	设置一个数组，使用	分割，例如：{{array(1|2 [1,2,3]，
base64dec	base64decode, base64d, b64d	进行 base64 解码，{{base64dec(YWJj)}} => abc
base64enc	base64encode, base64e, base64, b64	进行 base64 编码，{{base64enc(abc)}} => YWJj
base64tohex	b642h, base642hex	把 Base64 字符串转换为 HEX 编码，{{base64tohex(YWJj)}}
bmp		生成一个 bmp 文件头，例如 {{bmp}}
char	c, ch	生成一个字符，例如：{{char(a-z)}}, 结果为 [a b c ... x y z]
codec		调用 Yakit Codec 插件
codec:line		调用 Yakit Codec 插件，把结果解析成行
date		生成一个时间，格式为YYYY-MM-dd，如果指定了格式，将按
datetime	time	生成一个时间，格式为YYYY-MM-dd HH:mm:ss，如果指定了 式生成时间
doubleurldec	doubleurldecode, durldec, durldecode	双重URL解码，{{doubleurldec(%2561%2562%2563)}} => a
doubleurlenc	doubleurlencode, durlenc, durl	双重URL编码，{{doubleurlenc(abc)}} => %2561%2562%25
file		读取文件内容，可以支持多个文件，用竖线分割，{{file(/tmp tmp/1.txt|/tmp/test.txt)}}
file:dir	filedir	解析文件夹，把文件夹中文件的内容读取出来，读取成数组返 tmp/test)}} 或 {{file:dir(/tmp/test|/tmp/1)}}
file:line	fileline, file:lines	解析文件名（可以用	分割），把文件中的内容按行返 {{file:line(/tmp/test.txt)}} 或 {{file:line(/tmp/test.txt|/
fuzz:password	fuzz:pass	根据所输入的操作随机生成可能的密码（默认为 root/admin

fuzz:username	fuzz:user	根据所输入的操作随机生成可能的用户名（默认为 root/adm
gif		生成 gif 文件头
headerauth		用于java web回显payload执行时寻找特征请求
hexdec	hexd, hexdec, hexdecode	HEX 解码，{{hexdec(616263)}} => abc
hexenc	hex, hexencode	HEX 编码，{{hexenc(abc)}} => 616263
hextobase64	h2b64, hex2base64	把 HEX 字符串转换为 base64 编码，{{hextobase64(616263)
htmldec	htmldecode, htmlunescape	HTML 解码，{{htmldec(abc)}} => abc
htmlenc	htmlencode, html, htmle, htmlescape	HTML 实体编码，{{htmlenc(abc)}} => abc
htmlhexenc	htmlhex, htmlhexencode, htmlhexescape	HTML 十六进制实体编码，{{htmlhexenc(abc)}} => abc
ico		生成一个 ico 文件头，例如 {{ico}}
int	port, ports, integer, i, p	生成一个整数以及范围，例如 {{int(1,2,3,4,5)}} 生成 1,2,3,4,5 以使用 {{int(1-5)}} 生成 1-5 的整数，也可以使用 {{int(1-5&#1 数，但是每个整数都是 4 位数，例如 0001, 0002, 0003, 0004
jpg	jpeg	生成 jpeg / jpg 文件头
lower		把传入的内容都设置成小写 {{lower(Abc)}} => abc
md5		进行 md5 编码，{{md5(abc)}} => 900150983cd24fb0d6963
network	host, hosts, cidr, ip, net	生成一个网络地址，例如 {{network(192.168.1.1/24)}} 对应 c 有地址，可以逗号分隔，例如 {{network(8.8.8.8,192.168.1.1
null	nullbyte	生成一个空字节，如果指定了数量，将生成指定数量的空字节 个空字节
padding:null	nullpadding, np	使用 \x00 来填充补偿字符串长度不足的问题，{{nullpadding 填充到长度为 5 的字符串（\x00\x00abc），{{nullpadding(a 充到长度为 5 的字符串，并且在右边填充 (abc\x00\x00)
padding:zero	zeropadding, zp	使用0来填充补偿字符串长度不足的问题，{{zeropadding(ab 到长度为 5 的字符串（00abc），{{zeropadding(abc|-5)}} 表 为 5 的字符串，并且在右边填充 (abc00)
payload	x	从数据库加载 Payload, {{payload(pass_top25)}}
png		生成 PNG 文件头
punctuation	punc	生成所有标点符号
quote		strconv.Quote 转化
randint	ri, rand:int, randi	随机生成整数，定义为 {{randint(10)}} 生成0-10中任意一个随 {{randint(1,50)}} 生成 1-50 任意一个随机数，{{randint(1,50, 个随机数，重复 10 次
randomupper	random:upper, random:lower	随机大小写，{{randomupper(abc)}} => aBc
randstr	rand:str, rs, rands	随机生成个字符串，定义为 {{randstr(10)}} 生成长度为 10 的 {{randstr(1,30)}} 生成长度为 1-30 为随机字符串，{{randstr(机字符串，长度为 1-30
rangechar	range:char, range	按顺序生成一个 range 字符集，例如 {{rangechar(20,7e)}} 生 集

regen	re	使用正则生成所有可能的字符
repeat		重复一个字符串，例如：{{repeat(abc|3)}}，结果为：
repeat:range		重复一个字符串，并把重复步骤全都输出出来，例如：{{repe 果为：['' abc abcabc abcabcabc]
repeatstr	repeat:str	重复字符串，{{repeatstr(abc|3)}} => abcabcabc
sha1		进行 sha1 编码，{{sha1(abc)}} => a9993e364706816aba3e25717850c26c9cd0d89d
sha224		进行 sha224 编码，{{sha224(abc)}} => 23097d223405d8228642a477bda255b32aadbce4bda0b3f
sha256		进行 sha256 编码，{{sha256(abc)}} => ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb
sha384		进行 sha384 编码，{{sha384(abc)}} => cb00753f45a35e8bb5a03d699ac65007272c32ab0eded163 86072ba1e7cc2358baeca134c825a7
sha512		进行 sha512 编码，{{sha512(abc)}} => ddaf35a193617abacc417349ae20413112e6fa4e89a97ea20 92992a274fc1a836ba3c23a3feebbd454d4423643ce80e2a9
sm3		计算 sm3 哈希值，{{sm3(abc)}} => 66c7f0f462eeedd9d1f2d46bdc10e4e24167c4875cf2f7a3f0
tiff		生成一个 tiff 文件头，例如 {{tiff}}
timestamp		生成一个时间戳，默认单位为秒，可指定单位：s, ms, ns: {{t
trim		去除字符串两边的空格，一般配合其他 tag 使用，如：{{trim
unquote		把内容进行 strconv.Unquote 转化
upper		把传入的内容变成大写 {{upper(abc)}} => ABC
urldec	urldecode, urld	URL 强制解码，{{urldec(%61%62%63)}} => abc
urlenc	urlencode, url	URL 强制编码，{{urlenc(abc)}} => %61%62%63
urlescape	urlesc	url 编码(只编码特殊字符)，{{urlescape(abc=)}} => abc%3d
uuid		生成一个随机的uuid，如果指定了数量，将生成指定数量的u
yso:bodyexec		尽力使用 class body exec 的方式生成多个链
yso:dnslog		生成多个可以触发dnslog的payload，一般用于爆破利用链
yso:exec		生成所有命令执行的payload
yso:find_gadget_by_bomb		使用一个复杂的序列化对象作为payload，由于反序列化耗时 序列化漏洞，可以用于寻找gadget
yso:find_gadget_by_dns		使用dnslog外带目标环境的class信息
yso:headerecho		尽力使用 header echo 生成多个链
yso:urldns		使用transform触发指定域名的dns查询，用于验证反序列化