函数

递归函数

简单来说，递归就是函数自己调用自己。有2种实现方式，一种是直接在自己函数中调用自己，一种是间接在自己函数中调用的其他函数中调用了自己。

- 递归函数需要有边界条件、递归前进段、递归返回段
- 递归一定要有边界条件
- 当边界条件不满足时，递归前进
- 当边界条件满足时，递归返回

斐波那契数列递归

斐波那契数列Fibonacci number：1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

如果设F(n）为该数列的第n项（n∈N*），那么这句话可以写成如下形式：F(n)=F(n-1)+F(n-2)

有F(0)=0，F(1)=1, F(n)=F(n-1)+F(n-2)


```go
package main

import "fmt"

// 非递归版，循环版
func fib(n int) int {
    switch {
    case n < 0:
        panic("n is not negative")
    case n == 0:
        return 0
    case n == 1 || n == 2:
        return 1
    }
    a, b := 1, 1
    for i := 0; i < n-2; i++ {
        a, b = b, a+b
    }
    return b
}

func main() {
    for i := 1; i < 10; i++ {
        fmt.Println(fib(i))
    }
}
```

使用递归实现，需要使用递归公式F(n)=F(n-1)+F(n-2) 。

递归有2种形式实现

1. 采用递推公式

```
func fib(n int) int {
    // 递归公式版本。为了让同学们更加清晰的看清主干，这里简化switch
    if n < 3 {
        return 1
    }
    return fib(n-1) + fib(n-2)
}

fib(45)
```

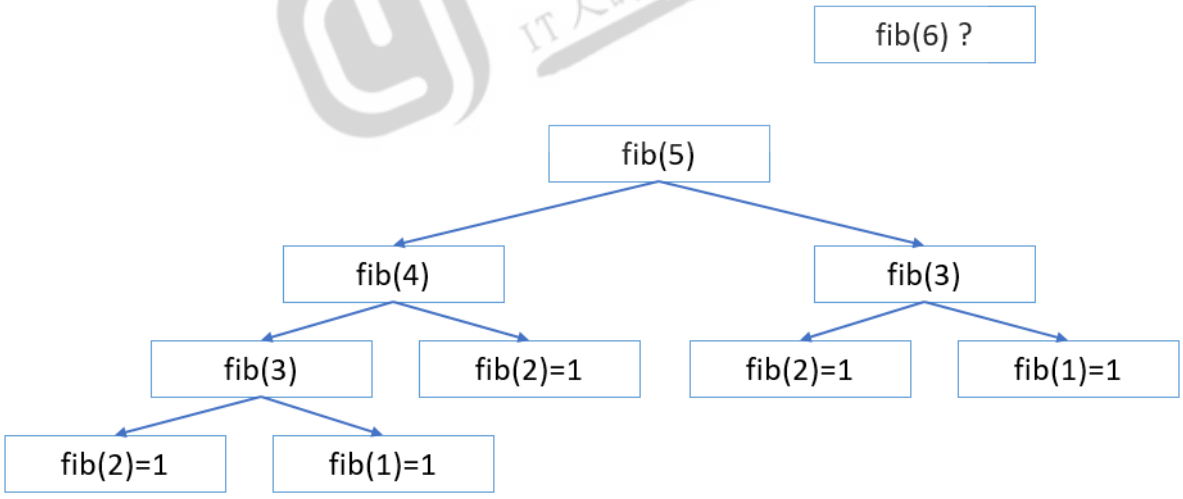
2. 循环层次变成递归函数层次

```
func fib(n, a, b int) int {
    if n < 3 {
        return b
    }
    return fib(n-1, b, a+b)
}

fib(45, 1, 1)
```

第一种使用那么美的递归公式为什么慢?

以fib(5)为例。看了下图后，fib(6)是怎样计算的呢?



这个函数进行了大量的重复计算，所以慢。

递归要求

- 递归一定要有退出条件，递归调用一定要执行到这个退出条件。没有退出条件的递归调用，就是无限调用
- 递归调用的深度不宜过深
- Go语言不可能让函数无限调用，栈空间终会耗尽
    - goroutine stack exceeds 1000000000-byte limit

递归效率

以上3个斐波那契数列实现，请问那个效率高？递归效率一定低吗？哪个版本好？

递归版本1效率极低，是因为有大量重复计算。

递归版本2采用了递归函数调用层次代替循环层次，效率还不错，和循环版效率差不多。

那么递归版2和循环版谁好？

循环版好些，因为递归有深度限制，再一个函数调用开销较大。

间接递归

```
func foo() {
    bar()
}

func bar() {
    foo()
}

foo()
```

间接递归调用，是函数通过别的函数调用了自己，这一样是递归。

只要是递归调用，不管是直接还是间接，都要注意边界返回问题。但是间接递归调用有时候是非常不明显，代码调用复杂时，很难发现出现了递归调用，这是非常危险的。

所有，使用良好的代码规范来避免这种递归的发生。

总结

- 递归是一种很自然的表达，符合逻辑思维
- 递归相对运行效率低，每一次调用函数都要开辟栈帧
- 递归有深度限制，如果递归层次太深，函数连续压栈，栈内存就溢出了
- 如果是有限次数的递归，可以使用递归调用，或者使用循环代替，循环代码稍微复杂一些，但是只要不是死循环，可以多次迭代直至算出结果
- 绝大多数递归，都可以使用循环实现
- 即使递归代码很简洁，但是能不用则不用递归

匿名函数

顾名思义，就是没有名字的函数。在函数定义中，把名字去掉就可以了。

```go
func(x, y int) int {
    result := x + y
    fmt.Println(result)
    return result
}(4, 5) // 定义后立即调用

add := func(x, y int) int {
    return x + y
} // 使用标识符指向一个匿名函数
fmt.Println(add(4, 5))
```

匿名函数主要作用是用作高阶函数中，是传入的逻辑，函数允许传入参数，就是把逻辑外置。

例如，给定2个整数，请给定计算函数，得到结果

```go
package main

import "fmt"

func calc(a, b int, fn func(int, int) int) int {
    return fn(a, b)
}

func main() {
    fmt.Println(calc(4, 5, func(a, b int) int { return a + b })) // 加法
    fmt.Println(calc(4, 5, func(a, b int) int { return a * b })) // 乘法
}
```

但是Go语言没有lambda表达式，也没有类似JavaScript的箭头函数，匿名函数写起来还是较为繁琐，只能使用类型别名简化，但是并没有什么太大的作用。

```go
package main

import "fmt"

type MyFunc = func(int, int) int

func calc(a, b int, fn MyFunc) int {
    return fn(a, b)
}

func main() {
    fmt.Println(calc(4, 5, func(a, b int) int { return a + b })) // 加法
    fmt.Println(calc(4, 5, func(a, b int) int { return a * b })) // 乘法
}
```

函数嵌套

```go
package main

import "fmt"

func outer() {
    c := 99
    var inner = func() {
        fmt.Println("1 inner", c)
    }
    inner()
    fmt.Println("2 outer", c)
}

func main() {
    outer()
}
```

可以看到outer中定义了另外一个函数inner，并且调用了inner。outer是包级变量，main可见，可以调用。而inner是outer中的局部变量，outer中可见。

嵌套作用域

```go
package main

import "fmt"

func outer() {
    c := 99
    var inner = func() {
        c = 100
        fmt.Println("1 inner", c) // 请问c是多少
    }
    inner()
    fmt.Println("2 outer", c) // 请问c是多少
}

func main() {
    outer()
}
```

上例分析

- 第9、12行都输出100
- 说明内外用的同一个c声明，用的同一个标识符，也就是c是outer的局部变量，而不是inner的局部变量

```go
package main

import "fmt"

```

```go
func outer() {
    c := 99
    var inner = func() {
        c = 100
        fmt.Println("1 inner", c) // 请问c是多少
        c := 101 // var c = 101
        fmt.Println("3 inner", c) // 请问c是多少
    }
    inner()
    fmt.Println("2 outer", c) // 请问c是多少
}

func main() {
    outer()
}
```

上例分析

- 第9、14行都输出100，第11行输出101
- 输出结果说明第9、14行是同一个c，都是outer的c；而第10行的c是inner的c，因为这是定义，即在当前作用域中定义新的局部变量，而这个局部变量只能影响当前作用域，不能影响其外部作用域，对外不可见

注：这个代码在不同语言中，几处c输出结果和Go语言不一定相同

闭包

自由变量：未在本地作用域中定义的变量。例如定义在内层函数外的外层函数的作用域中的变量。

闭包：就是一个概念，出现在嵌套函数中，指的是**内层函数引用到了外层函数的自由变量**，就形成了闭包。很多语言都有这个概念，最熟悉就是JavaScript。闭包是运行期动态的概念。

- 函数有嵌套，函数内定义了其它函数
- 内部函数使用了外部函数的局部变量
- 内部函数被返回（非必须）

```go
package main

import "fmt"

func outer() func() {
    c := 99
    fmt.Printf("outer %d %p\n", c, &c)
    var inner = func() {
        fmt.Printf("inner %d %p\n", c, &c)
    }
    return inner
}

func main() {
    var fn = outer()
    fn()
```

```
17 }
```

上例有闭包吗？为什么？

- 首先有嵌套函数，也就是有嵌套作用域
- inner函数中用到了c，但是它没有定义c，而外部的outer有局部变量c

代码分析

- 第15行调用outer函数并返回**inner函数对象**，并使用标识符fn记住了它。outer函数执行完了，在其内部定义的局部变量应该释放，这个内部定义的inner函数也是局部的，但是并不能释放，因为fn要用它
- 在某个时刻，fn函数调用时，需要用到c，但是其内部没有定义c，它是outer的局部变量，如果这个c早已随着outer的调用而释放，那么fn函数调用一定出现错误，所以，这个outer的c不能释放，但是outer已经调用完成了，怎么办？闭包，让inner函数记住自由变量c（内存地址）