GO 语言开发 2048

一、实验介绍

1.1 实验内容

2048的游戏相信大家都玩过啦。这次带来的是 Go 语言版本的，我们来看看用 Go 语言写出来的2048和别的语言写出来的有什么不一样?

1.2 实验知识点

- Go 语言的基本语法
- Go 语言中 package 的创建与引用

1.3 实验环境

- Go 1.2.1
- Xfce终端

1.4 适合人群

本课程难度为一般，属于中级级别课程，适合具有 Go 基础的用户，熟悉 Go 基础知识加深巩固。

1.5 代码获取

本课程中的所有源码可以通过以下方式下载:

```
$ git clone https://github.com/shiyanlou/golang2048_game
```

1.6 参考文档

本实验课程的代码部分参考下列资料：

- https://github.com/wangwenbin/2048-go (https://github.com/wangwenbin/2048-go)

二. 2048 游戏设计

实验楼课程Go语言编程 (http://www.shiyanlou.com/courses/11)是本课程的基础课，建议完整学习Go语言编程 (http://www.shiyanlou.com/courses/11)后再学习本课程。

《2048》由19岁的意大利人Gabriele Cirulli于2014年3月开发。游戏任务是在一个网格上滑动小方块来进行组合，直到形成一个带有有数字2048的方块。作者开发这个游戏是为了测试自己是否有能力从零开始创造一款游戏，但游戏飙升的人气（不到1周内有400万访客）完全出乎他的预料。《2048》使用方向键让方块上下左右移动。如果两个带有相同数字的方块在移动中碰撞，则它们会合并为一个方块，且所带数字变为两者之和。每次移动时，会有一个值为2或者4的新方块出现。当值为2048的方块出现时，游戏即胜利。

本课程中我们将学习使用Go语言开发一个console版本的2048游戏，游戏的操作通过方向键盘来进行。先让我们来看下游戏的界面:

2.1 游戏逻辑设计

2048游戏使用4x4的格子来表示需要移动的数字，这不难想到可以使用一个矩阵来表示这些数字，我们使用 `type G2048 [4][4]int` 来表示。每一次使用方向键来移动数字时，对应方向上的数字需要进行移动和合并，也就是移动和合并 `G2048` 矩阵中的非零值。当按下不同的方向键时，移动的数字也不同。我们一共会向上、向下、向左、向右四个方向移动数字，但是我们可以通过旋转 `G2048` 矩阵将向下、向左、向右的移动都转换为向上的移动，这样能一定程度上简化游戏逻辑。

先让我们分析下，当向上移动的时候怎么样合并数据呢？假如我们有一列数据（这里为了演示方便我们将一列数据转换为了一行数据，同时向上的移动改为向左的移动，但是原理都是一样的）

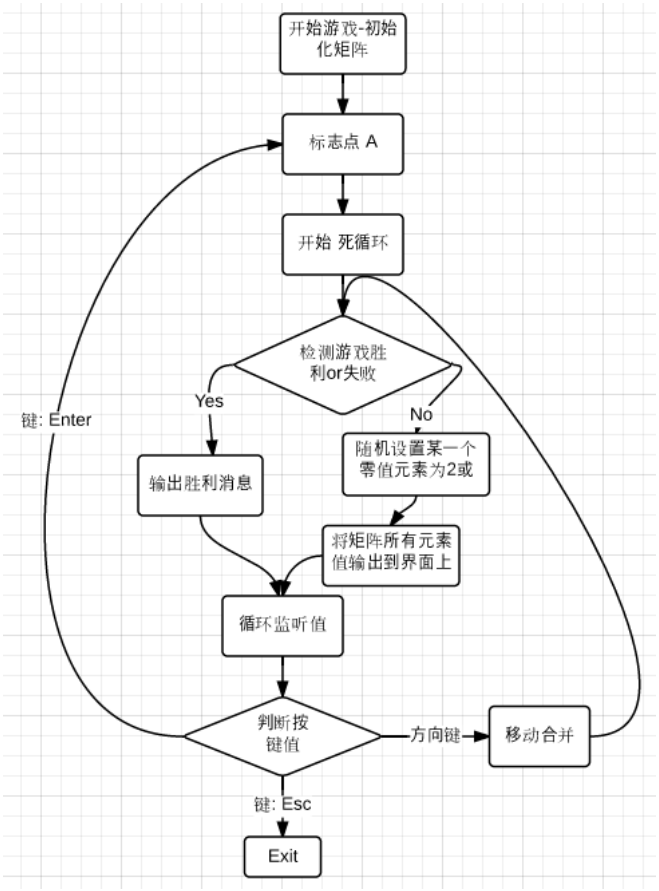
`[2, 2, 0, 2]` ，当我们向左移动合并的时候，我们首先会移动数据，然后进行合并，然后再次进行移动，这个过程产生的数字序列如下:

`[2, 2, 0, 2]` 向左移动后变为 `[2, 2, 2, 0]` ，然后我们进行合并操作后变为 `[4, 0, 2, 0]` ，然后我们再次进行移动操作，则变为 `[4, 2, 0, 0]` ，也就是我们向左移动合并后的最终结果

```
[2, 2, 0, 2] -> [2, 2, 2, 0] -> [4, 0, 2, 0] -> [4, 2, 0, 0]
```

我们可以对所有的行（列）执行以上操作完成整个矩阵的移动合并。

那整个游戏的逻辑是怎么样的呢？2048游戏逻辑如下：首先进入无限循环，检查矩阵中是否有元素的值大于等于2048，如果有，则游戏胜利，如果没有则游戏会随机将矩阵中值为零的元素的值设置为2或者4然后并监听用户键盘操作，然后将矩阵的所有元素的值打印到中4x4的表格中， 接着开始监听键盘事件，针对不同的按键做出不同的响应，比如当按下方向键时，游戏将执行矩阵相应方向的移动合并操作，然后再次开始循环。如果按下的是 `Esc` 键，则退出游戏。其流程图如下：



到目前为止整个游戏的逻辑一目了然。

2.2 界面设计

我们开发的2048游戏将运行在console（字符界面控制台，终端，控制台等）下。在console中，我们可以控制每一个字符单元的背景色，以及显示的字符。我们可以根据这一点，在console中绘制中图形，也就是2048游戏的框架：4x4的空白格子，然后每一个格子是4个字符单元，也就是最多能显示四位数字。我们将使用包 `github.com/shiyanlou/termbox-go` 进行界面的绘制， `termbox-go` 能很方便的设置字符单元的属性。我们首先需要配置好Go的环境变量

```
$ cd ~
$ git clone https://github.com/shiyanlou/golang2048_game
$ cd golang2048_game
$ export GOPATH=/home/shiyanlou/golang2048_game
$ go get github.com/shiyanlou/termbox-go
```

`termhox-go` 官方说明如下:

> Termbox is a library that provides a minimalistic API which allows the programmer to write text-based user interfaces. The library is crossplatform and has both terminal-based implementations on *nix operating systems and a winapi console based implementation for windows operating systems. The basic idea is an abstraction of the greatest common subset of features available on all major terminals and other terminal-like APIs in a minimalistic fashion. Small API means it is easy to implement, test, maintain and learn it, that's what makes the termbox a distinct library in its area

下面我们简单介绍下 `termbox` 中比较重要的函数:

- `func Iint() error`

  在使用termbox进行程序开发的时候，我们需要先使用termbox.Init方法来初始化，当不再使用termbox任何功能时候，使用termbox.Close来关闭对termbox的引入。

- `func Flush() error`

  同步后台的缓存。Flush方法一般用于将后台的处理输出到界面中。比如重新绘制一个界面。

- `func SetCell(x, y int, ch rune, fg, bg Attribute)`

  用于设置字符单元的属性，其中x表示所在行，y表示所在列，ch是需要设置的字符，fg，和bg分布表示前景色和背景色。

- `func PollEvent() Event`

用于等待键盘事件的触发，并返回事件，无事件发生时则无限等待。

- `func Size() (int, int)`

  获取**console**的尺寸。

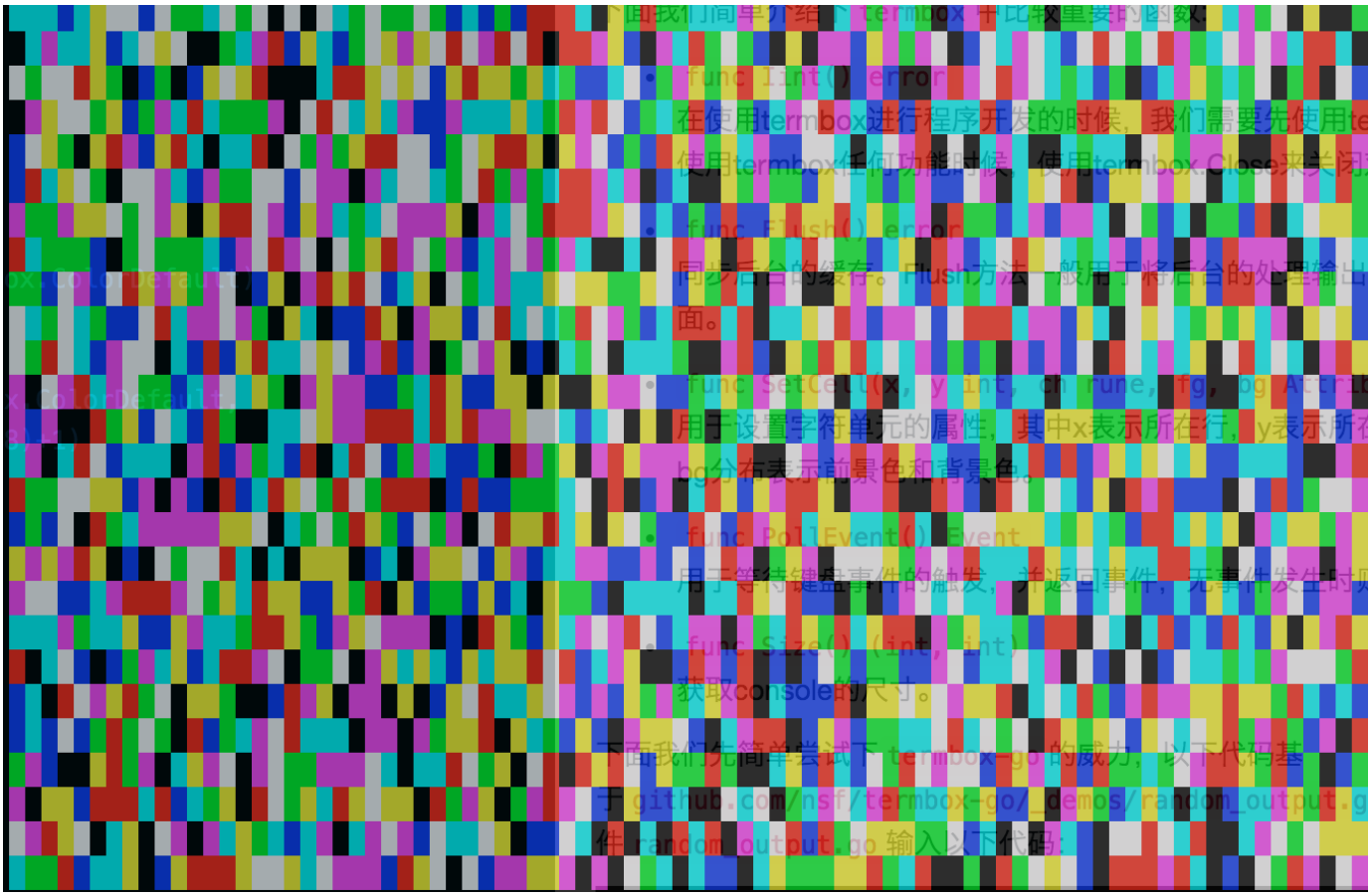
下面我们先简单尝试下 `termbox-go` 的威力，以下代码基于 `github.com/shiyanlou/termbox-go/_demos/random_output.go` ，创建源文件 `random_output.go` 输入以下代码：

```
package main

import "github.com/shiyanlou/termbox-go"
import "math/rand"
import "time"

func draw() { // 随机设置字符单元的属性
    w, h := termbox.Size()
    termbox.Clear(termbox.ColorDefault, termbox.ColorDefault)
    for y := 0; y < h; y++ {
        for x := 0; x < w; x++ {
            termbox.SetCell(x, y, ' ', termbox.ColorDefault,
                termbox.Attribute(rand.Int()%8)+1)
        }
    }
    termbox.Flush() // 刷新后台缓存到界面中
}

func main() {
    err := termbox.Init() // 初始化termbox包
    if err != nil {
        panic(err)
    }
    defer termbox.Close()

    event_queue := make(chan termbox.Event)
    go func() {
        for {
            event_queue <- termbox.PollEvent() // 开始监听键盘事件
        }
    }()

    draw()
    for {
        select {
        case ev := <-event_queue: // 获取键盘事件
            if ev.Type == termbox.EventKey && ev.Key == termbox.KeyEsc {
                return //如果是 Esc 键，则退出程序
            }
        default:
            draw()
            time.Sleep(10 * time.Millisecond)
        }
    }
}
```

下面让我们执行以上代码，就可以在**console**中看到五彩缤纷的数据流啦，如果想退出程序需按下 `Esc` 键。

```
$ go run random_output.go
```

效果图如下:



三. 2048游戏的实现

2048游戏中的难点有两个地方，一个是矩阵中数字的移动合并，另一个则是矩阵的变换，之所以需要对矩阵进行变换，是为了将2048游戏中向下的移动，向左的移动和向右的移动都转换成向上的移动操作。关于移动和合并操作已经在前文中分析过，这里只讨论矩阵的旋转。

3.1 矩阵的旋转

矩阵的旋转操作是为了将其他三个方向的移动都转换为向上的移动操作。向下(↓)、向左(←)、向右(→)转换为向上(↑)的操作时，数组需要进行的翻转操作如下所示：

- ↓ → ↑

  此类转换可以有多种方法做到：

  - 上下翻转矩阵，然后向上移动合并，再次上下翻转矩阵 上下翻转后： `martix_new[n-1-x][y] = martix_old[x][y]`
  - 顺时针翻转180度矩阵，然后向上移动合并，接着逆时针旋转180度 此时 `martix_new[n-1-x]n-1-y] = martix_old[x][y]`

- ← → ↑

  此类转换可以将矩阵向右旋转90度后，向上移动合并，接着向左旋转90度完成

  向右旋转90度后： `martix_new[y][n-x-1] = martix_old[x][y]`

  向左旋转90度后： `martix_new[n-y-1][x] = martix_old[x][y]`

- → → ↑

  此类转换可以将矩阵向左旋转90度后，向上移动合并，接着向右旋转90度完成

经过以上的分析中我们已经很清楚矩阵的翻转操作了，下面我们练习下，创建源文件 `martix_rorate.go` ，输入以下代码：

```
package main

import "fmt"

type g2048 [4][4]int

func (t *g2048) MirrorV() {
    tn := new(g2048)
    for i, line := range t {
        for j, num := range line {
            tn[len(t)-i-1][j] = num
        }
    }
    *t = *tn
}

func (t *g2048) Right90() {
    tn := new(g2048)
    for i, line := range t {
        for j, num := range line {
            tn[j][len(t)-i-1] = num
        }
    }
    *t = *tn
}

func (t *g2048) Left90() {
    tn := new(g2048)
    for i, line := range t {
        for j, num := range line {
            tn[len(line)-j-1][i] = num
        }
    }
    *t = *tn
}

func (g *g2048) R90() {
    tn := new(g2048)
    for x, line := range g {
        for y, _ := range line {
            tn[x][y] = g[len(line)-1-y][x]
        }
    }
    *g = *tn

}

func (t *g2048) Right180() {
    tn := new(g2048)
    for i, line := range t {
        for j, num := range line {
            tn[len(line)-i-1][len(line)-j-1] = num
        }
    }
    *t = *tn
}

func (t *g2048) Print() {
    for _, line := range t {
        for _, number := range line {
            fmt.Printf("%2d ", number)
        }
        fmt.Println()
    }
    fmt.Println()
    tn := g2048{{1, 2, 3, 4}, {5, 8}, {9, 10, 11}, {13, 14, 16}}
    *t = tn

}

func main() {
    fmt.Println("origin")
    t := g2048{{1, 2, 3, 4}, {5, 8}, {9, 10, 11}, {13, 14, 16}}
    t.Print()
    fmt.Println("mirror")
    t.MirrorV()
    t.Print()
    fmt.Println("Left90")
    t.Left90()
    t.Print()
```

```
    fmt.Println("Right90")
    t.R90()
    t.Print()
    fmt.Println("Right180")
    t.Right180()
    t.Print()
}
```

执行后，输出如下：

```
$ go run martix_rorate.go
origin
 1  2  3  4
 5  8  0  0
 9 10 11  0
13 14 16  0

mirror
13 14 16  0
 9 10 11  0
 5  8  0  0
 1  2  3  4

Left90
 4  0  0  0
 3  0 11 16
 2  8 10 14
 1  5  9 13

Right90
13  9  5  1
14 10  8  2
16 11  0  3
 0  0  0  4

Right180
 0 16 14 13
 0 11 10  9
 0  0  8  5
 4  3  2  1
```

3.2 2048的实现

到这里，整个2048的游戏逻辑已经非常清楚了，现在我们来实现这个游戏吧。首先我们来创建 package g2048。在 $GOPATH/src/ 目录中创建目录 g2048 ， 然后在 $GOPATH/src/g2048 目录中创建源文件 2048.go ， 输入以下内容:

```
/*
作者: www.shiyanlou.com
说明: 2048 游戏 Go语言版本
*/

package g2048

import (
    "fmt"
    "github.com/shiyanlou/termbox-go"
    "math/rand"
    "time"
)

var Score int
var step int

// 输出字符串
func coverPrintStr(x, y int, str string, fg, bg termbox.Attribute) error {

    xx := x
    for n, c := range str {
        if c == '\n' {
            y++
            xx = x - n - 1
        }
        termbox.SetCell(xx+n, y, c, fg, bg)
    }
    termbox.Flush()
    return nil
}

// 游戏状态
type Status uint

const (
    Win Status = iota
    Lose
    Add
    Max = 2048
)

// 2048游戏中的16个格子使用4x4二维数组表示
type G2048 [4][4]int

// 检查游戏是否已经胜利，没有胜利的情况下随机将值为0的元素
// 随机设置为2或者4
func (t *G2048) checkWinOrAdd() Status {
    // 判断4x4中是否有元素的值大于(等于)2048，有则获胜利
    for _, x := range t {
        for _, y := range x {
            if y >= Max {
                return Win
            }
        }
    }
    // 开始随机设置零值元素为2或者4
    i := rand.Intn(len(t))
    j := rand.Intn(len(t))
    for x := 0; x < len(t); x++ {
        for y := 0; y < len(t); y++ {
            if t[i%len(t)][j%len(t)] == 0 {
                t[i%len(t)][j%len(t)] = 2 << (rand.Uint32() % 2)
                return Add
            }
            j++
        }
        i++
    }

    // 全部元素都不为零（表示已满），则失败
    return Lose
}

// 初始化游戏界面
func (t G2048) initialize(ox, oy int) {
    fg := termbox.ColorYellow
```

```
    bg := termbox.ColorBlack
    termbox.Clear(fg, bg)
    str := "       SCORE: " + fmt.Sprint(Score)
    for n, c := range str {
        termbox.SetCell(ox+n, oy-1, c, fg, bg)
    }
    str = "ESC:exit " + "Enter:replay"
    for n, c := range str {
        termbox.SetCell(ox+n, oy-2, c, fg, bg)
    }
    str = " PLAY with ARROW KEY"
    for n, c := range str {
        termbox.SetCell(ox+n, oy-3, c, fg, bg)
    }
    fg = termbox.ColorBlack
    bg = termbox.ColorGreen
    for i := 0; i <= len(t); i++ {
        for x := 0; x < 5*len(t); x++ {
            termbox.SetCell(ox+x, oy+i*2, '-', fg, bg)
        }
        for x := 0; x <= 2*len(t); x++ {
            if x%2 == 0 {
                termbox.SetCell(ox+i*5, oy+x, '+', fg, bg)
            } else {
                termbox.SetCell(ox+i*5, oy+x, '|', fg, bg)
            }
        }
    }
    fg = termbox.ColorYellow
    bg = termbox.ColorBlack
    for i := range t {
        for j := range t[i] {
            if t[i][j] > 0 {
                str := fmt.Sprint(t[i][j])
                for n, char := range str {
                    termbox.SetCell(ox+j*5+1+n, oy+i*2+1, char, fg, bg)
                }
            }
        }
    }
    return termbox.Flush()
}

// 翻转二维切片
func (t *G2048) mirrorV() {
    tn := new(G2048)
    for i, line := range t {
        for j, num := range line {
            tn[len(t)-i-1][j] = num
        }
    }
    *t = *tn
}

// 向右旋转90度
func (t *G2048) right90() {
    tn := new(G2048)
    for i, line := range t {
        for j, num := range line {
            tn[j][len(t)-i-1] = num
        }
    }
    *t = *tn
}

// 向左旋转90度
func (t *G2048) left90() {
    tn := new(G2048)
    for i, line := range t {
        for j, num := range line {
            tn[len(line)-j-1][i] = num
        }
    }
    *t = *tn
}

func (t *G2048) right180() {
    tn := new(G2048)
    for i, line := range t {
        for j, num := range line {
```

```
            tn[len(line)-i-1][len(line)-j-1] = num
        }
    }
    *t = *tn
}

// 向上移动并合并
func (t *G2048) mergeUp() bool {
    tl := len(t)
    changed := false
    notfull := false
    for i := 0; i < tl; i++ {

        np := tl
        n := 0 // 统计每一列中非零值的个数

        // 向上移动非零值，如果有零值元素，则用非零元素进行覆盖
        for x := 0; x < np; x++ {
            if t[x][i] != 0 {
                t[n][i] = t[x][i]
                if n != x {
                    changed = true // 标示数组的元素是否有变化
                }
                n++
            }
        }
        if n < tl {
            notfull = true
        }
        np = n
        // 向上合并所有相同的元素
        for x := 0; x < np-1; x++ {
            if t[x][i] == t[x+1][i] {
                t[x][i] *= 2
                t[x+1][i] = 0
                Score += t[x][i] * step // 计算游戏分数
                x++
                changed = true
            }
        }
        // 合并完相同元素以后，再次向上移动非零元素
        n = 0
        for x := 0; x < np; x++ {
            if t[x][i] != 0 {
                t[n][i] = t[x][i]
                n++
            }
        }
        for x := n; x < tl; x++ {
            t[x][i] = 0
        }
    }
    return changed || !notfull
}

// 向下移动合并的操作可以转换向上移动合并:
// 1. 向右旋转180度矩阵
// 2. 向上合并
// 3. 再次向右旋转180度矩阵
func (t *G2048) mergeDwon() bool {
    //t.mirrorV()
    t.right180()
    changed := t.mergeUp()
    //t.mirrorV()
    t.right180()
    return changed
}

// 向左移动合并转换为向上移动合并
func (t *G2048) mergeLeft() bool {
    t.right90()
    changed := t.mergeUp()
    t.left90()
    return changed
}

/// 向右移动合并转换为向上移动合并
func (t *G2048) mergeRight() bool {
    t.left90()
    changed := t.mergeUp()
```

```
    t.right90()
    return changed
}

// 检查按键，做出不同的移动动作或者退出程序
func (t *G2048) mrgeAndReturnKey() termbox.Key {
    var changed bool
Lable:
    changed = false
    //ev := termbox.PollEvent()
    event_queue := make(chan termbox.Event)
    go func() { // 在其他goroutine中开始监听
        for {
            event_queue <- termbox.PollEvent() // 开始监听键盘事件
        }
    }()

    ev := <-event_queue

    switch ev.Type {
    case termbox.EventKey:
        switch ev.Key {
        case termbox.KeyArrowUp:
            changed = t.mergeUp()
        case termbox.KeyArrowDown:
            changed = t.mergeDwon()
        case termbox.KeyArrowLeft:
            changed = t.mergeLeft()
        case termbox.KeyArrowRight:
            changed = t.mergeRight()
        case termbox.KeyEsc, termbox.KeyEnter:
            changed = true
        default:
            changed = false
        }

        // 如果元素的值没有任何更改，则从新开始循环
        if !changed {
            goto Lable
        }

    case termbox.EventResize:
        x, y := termbox.Size()
        t.initialize(x/2-10, y/2-4)
        goto Lable
    case termbox.EventError:
        panic(ev.Err)
    }
    step++ // 计算游戏操作数
    return ev.Key
}

// 重置
func (b *G2048) clear() {
    next := new(G2048)
    Score = 0
    step = 0
    *b = *next

}

// 开始游戏
func (b *G2048) Run() {
    err := termbox.Init()
    if err != nil {
        panic(err)
    }
    defer termbox.Close()

    rand.Seed(time.Now().UnixNano())

A:

    b.clear()
    for { // 进入无限循环
        st := b.checkWinOrAdd()
        x, y := termbox.Size()
        b.initialize(x/2-10, y/2-4) // 初始化游戏界面
        switch st {
        case Win:
```

```
            str := "Win!!"
            strl := len(str)
            coverPrintStr(x/2-strl/2, y/2, str, termbox.ColorMagenta, termbox.ColorYellow)
        case Lose:
            str := "Lose!!"
            strl := len(str)
            coverPrintStr(x/2-strl/2, y/2, str, termbox.ColorBlack, termbox.ColorRed)
        case Add:
        default:
            fmt.Print("Err")
        }
        // 检查用户按键
        key := b.mrgeAndReturnKey()
        // 如果按键是 Esc 则退出游戏
        if key == termbox.KeyEsc {
            return
        }
        // 如果按键是 Enter 则从新开始游戏
        if key == termbox.KeyEnter {
            goto A
        }
    }
}
```

整个游戏代码相对简单，第二节中的流程图清晰的描述了2048游戏的执行逻辑。下面让我们创建真正开始游戏的源文件。在 $GOPATH 目录中，创建 g2048.go 源文件，输入以下代码:

```
package main

import "g2048"

func main() {
    var game g2048.G2048
    game.Run()
}
```

接着执行代码:

```
$ go run g2048.go
```

开始尽情游戏吧。

课程教师

Edward

共发布过**15**门课程

资深程序员，5年Linux运维、企业级开发经验及数据库实战和教学经验。

查看老师的所有课程 > (/teacher/20406)

前置课程

Go语言编程 (/courses/11)

进阶课程

制作Markdown预览器 (/courses/56)

GO语言FTP客户端的实现 (/courses/66)