课程安排，请关注**微信公众平台**或者**官方微博**

编程语言： Golang 与 html5

编程工具： Goland 和 HBuilder

预计平均一周左右更新一或二节课程

授人以鱼，不如授人以渔业。

大家好，
欢迎来到 **字节教育** 课程的学习
字节教育官网：**www.ByteEdu.Com**
腾讯课堂地址：**Gopher.ke.qq.Com**
技术交流群 ： 221 273 219
微信公众号 ：**Golang 语言社区**
微信服务号 ：**Golang 技术社区**

目录：

第一季 Go 语言基础、进阶、提高课

第四节　Go 语言函数

Go 语言的函数的基本组成包括：关键字 func，函数名，参数列表，返回值，函数体和返回语句。

1、函数的定义

这里举个简单的例子来说明一下 Go 语言中函数的定义问题：:

```
func SumFunc(a int, b int) int{
    return a + b
}
```

当形参的类型一样的时候可以简化为：

```
func SumFunc(a, b int) int {
    return a + b
}
```

对于函数的定义，除了多了个关键字 func 之外，其它的和其它语言也没什么大的区别。这里需要注意一下一个概念**函数的标识符**。函数的类型被称为函数的标识符。如果两个函数形式参数列表和返回值列表中的变量类型一一对应，那么这两个函数被认为有相同的类型和标识符。你可以使用以下代码打印上面函数的类型

```
fmt.Printf("%T\n", SumFunc)  //func(int, int) int
```

你可能会偶尔遇到没有函数体的函数声明，这表示该函数不是以 Go 实现的。这样的声明定义了函数标识符。

```
package math
func Sin(x float64) float //implemented in assembly language
```

多返回数值

在 Go 语言中，可以一次性返回多个函数值，前面的例子已经讲过，比如看下面的一个从网上下载图片的代码：

```
// 文件的复制操作
func CopyFile(src, dst string) (w int64, err error) {
        srcFile, err := os.Open(src)
        defer srcFile.Close()
        if err != nil {
                fmt.Println(err.Error())
                return
        }

        dstFile, err := os.Create(dst)
        defer dstFile.Close()
        if err != nil {
                fmt.Println(err.Error())
                return
        }
        return io.Copy(dstFile, srcFile)
}
```

如果一个函数将所有的返回值都显示的变量名，那么该函数的 return 语句可以省略操作数。这称之为 bare return。像上面的代码就可以写成 ： return

函数值

Go 语言提供了函数值的功能，可以将函数作为值使用：

```
func FuncValue(a, b int) int {
    return a + b
}

funcvalue := FuncValue

funcvalue2 := func(a, b int) int {
    return a * b
}

fmt.Println(funcvalue(1, 2))
fmt.Println(funcvalue2(3, 4))
```

匿名函数

在 Go 里面，匿名函数与 C 语言的回调函数比较类似。但 GO 语言可以在函数内部随时定义匿名函数。**这种方式定义的函数可以访问完整的词法环境（lexical environment），这意味着在函数中定义的内部函数可以引用该函数的变量.**

```
//直接定义匿名函数
funcvalue2 := func(a, b int) int {
    return a * b
}
funcvalue2(1,2)

//在函数内部返回匿名函数
func AnonymousFunc() func() {
    var sum int
    childFunc := func() {
        sum++
        fmt.Println("The sum is", sum)
    }
    return childFunc

}
```

如果上述的匿名函数采用下面的调用方式:

```
func1 := AnonymousFunc()
func1()
func1()
func1()
```

结果将会是 func1 := AnonymousFunc()

```
The sum is 1
The sum is 2
The sum is 3
```

这里说明 sum 变量和匿名函数是同时存在的，当你执行匿名函数 func1 的时候，sum 的值是一直保存的。这里牵扯到另外一个概念**闭包**。闭包个人理解就是内层函数可以访问外层的变量，但外层的没法访问内层的变量。

可变参数

参数可变的函数称为可变参数函数，像我们的 fmt 的 Println 这种包就是典型的可变参数函数。

```
//可变参数
func UnstableParam(vals ...int) {
    for i, val := range vals {
```

```
        fmt.Println(i, val)
    }
}
```

```
UnstableParam(1, 2, 3, 4, 5)
UnstableParam(1, 2)
```

如果用 slice 作为参数的话，可以使用下面代码：

```
myslice := []int{10, 11, 12, 13, 14}
UnstableParam(myslice...)
```

这个和 append 函数使用一样，原理应该是将 myslice 打散成一个个的元素，然后传递给函数。

Panic()和 recover()

Go 语言引入了这两个函数用来处理程序运行过程中的错误。

Go 的类型系统会在编译时捕获很多错误，但有些错误只能在运行时检查，如数组访问越界、空指针引用等。这些运行时错误会引起 painc 异常。当然也可以通过 panic 函数直接触发 panic 错误。**一般而言，当 panic 异常发生时，程序会中断运行，并立即执行在该 goroutine（可以先理解成线程，在第 8 章会详细介绍）中被延迟的函数（defer 机制）**

看看例子：

```
func ErrProcess() {
    panic("wwx")

}

func main(){
    ErrProcess()
    fmt.println("hello")
}
```

这样 hello 是不会输出的，因为 panic 会把所在的 goroutine（可以理解为线程先）终止掉。

但有时候有些异常我们不想直接把整个线程给终止掉。这时候就需要使用 recover 函数了。recover 函数可以直接捕获掉这个异常，不再上报给上一层。什么意思那？

假设函数 funa 调用了函数 funb,funb 又调用函数 func，如果 func 里面产生了 panic 异常，如果没有 recover 函数，那么该异常会上报给 funb，如果 funb 不存在 recover，将上报给 funa，一直到整个 goroutine 的开始。所以如果想打印出下面的 hello，那么需要在 ErrProcess 里面用 recover 函数来捕获这个异常：

```
func ErrProcess() {
```

```
    defer func() {
        if p := recover(); p != nil {
            fmt.Println("internal error: %v", p)
        }
    }()
    panic("wwx")

}
```

2、Go 方法

Go 语言中同时有函数和方法。一个方法就是一个包含了接受者的函数，接受者可以是命名类型或者结构体类型的一个值或者是一个指针。所有给定类型的方法属于该类型的方法集。

下面定义一个结构体类型和该类型的一个方法：

```
package main

import (
    "fmt"
)

/* 定义结构体 */
type Circle struct {
  radius float64
}

func main() {
  var c1 Circle
  c1.radius = 10.00
  fmt.Println("Area of Circle(c1) = ", c1.getArea())
}

//该 method 属于 Circle 类型对象中的方法
func (c Circle) getArea() float64 {
  //c.radius 即为 Circle 类型对象中的属性
  return 3.14 * c.radius * c.radius
}
```

普通函数与方法的区别

```
package structTest

    //普通函数与方法的区别（在接收者分别为值类型和指针类型的时候）

    import (
        "fmt"
    )

    func StructTest06Base() {
        structTest0601()
        structTest0602()
    }

    //1.普通函数
    //接收值类型参数的函数
    func valueIntTest(a int) int {
        return a + 10
    }

    //接收指针类型参数的函数
    func pointerIntTest(a *int) int {
        return *a + 10
    }

    func structTest0601() {
        a := 2
        fmt.Println("valueIntTest:", valueIntTest(a))
        //函数的参数为值类型，则不能直接将指针作为参数传递
        //fmt.Println("valueIntTest:", valueIntTest(&a))
        //compile error: cannot use &a (type *int) as type int in function
argument

        b := 5
        fmt.Println("pointerIntTest:", pointerIntTest(&b))
        //同样，当函数的参数为指针类型时，也不能直接将值类型作为参数传
递
        //fmt.Println("pointerIntTest:", pointerIntTest(b))
        //compile error:cannot use b (type int) as type *int in function
argument
    }
```

```
    //2.方法
    type PersonD struct {
        id   int
        name string
    }

    //接收者为值类型
    func (p PersonD) valueShowName() {
        fmt.Println(p.name)
    }

    //接收者为指针类型
    func (p *PersonD) pointShowName() {
        fmt.Println(p.name)
    }

    func structTest0602() {
        //值类型调用方法
        personValue := PersonD{101, "Will Smith"}
        personValue.valueShowName()
        personValue.pointShowName()

        //指针类型调用方法
        personPointer := &PersonD{102, "Paul Tony"}
        personPointer.valueShowName()
        personPointer.pointShowName()

        //与普通函数不同，接收者为指针类型和值类型的方法，指针类型和值类型的变量均可相互调用
    }
End
```

3、实现游戏服务器日志函数

```
// 日志函数
func Log(data string, data1 ...string) {
        var datatmp string

        datatmp = data
        // 循环取值
        for _, data1 := range data1 {
                datatmp = datatmp + data1
        }
        //=================================================
```

```
        var path string
        if os.IsPathSeparator('\\') { //前边的判断是否是系统的分隔符
                path = "\\"
        } else {
                path = "/"
        }
        dir, _ := os.Getwd()                        // 获取当前的程序路径
        os.MkdirAll(dir+path+"log", os.ModePerm) //生成多级目录
        //=================================================
        //创建日志文件
        t := time.Now()
        filepath := "./log/access_run_" + FilePort + "_" +
t.Format("2006-01-02") + ".txt"
        _, err := os.Stat(filepath)
        var file *os.File
        var sTmp string
        if err != nil {
                file, err = os.Create(filepath)
                defer file.Close()
        } else {
                file, err = os.OpenFile(filepath,
os.O_WRONLY|os.O_APPEND|os.O_CREATE, 0666)
                defer file.Close()
        }
        //sTmp = strings.Replace(t.String()[:19], ":", "_", 3) + ":  "
+ datatmp + "\n"
        sTmp = strings.Replace(t.String()[:19], ":", ":", 3) + ":  " +
datatmp + "\r\n"
        file.WriteString(sTmp)
        //file.Close()
}
```

4、课后作业—难度：★ ★ ☆ ☆ ☆

实现一个游戏消息缓存数据机构：

<1> 结合上节问题实现基础上，使用 map 机制作为缓存结构。

<2> 实现缓存模块的存、取、删除等方法

<3> 实现主动出现 panic 异常，清除 map 结构函数

5、深度分析塔防游戏关卡设计

隐约还记得最开始接触的塔防游戏是在《魔兽争霸 3》里的一个塔防地图，一个田字型的地图，四周一波一波的来怪物，而玩家需要在路边建塔消灭进攻的怪物，阻止它们到达终点。我想这应该是最基本的塔防了，路线、塔、怪物、刷怪点、守护点等几个简单的元素拼凑成一个塔防游戏。而如今塔防类游戏发展迅速，越来越多的新元素加入使得塔防游戏变得玩法多样，趣味十足。

工欲善其事必先利其器，我们先了解一下塔防类游戏的各类基本元素，才能利用好这些元素搭建一个充满策略性的塔防关卡。下面是我大体对塔防类游戏核心玩法的各种元素分析图，如有漏掉之处，希望大家积极指出交流：

上图左侧为关卡所需配置的一些元素，右侧为玩家防守所需控制的一些元素。下面我就对这些元素进行一一解析，希望对大家有所帮助

路线

1.1 一维路线

所谓的一维路线就是指怪物行进的道路只有前后，没有左右。简单的说怪物就是一条路走到底，很多年前的 WAR3 上 TD 关卡大多为这类路线，而玩家就在道路两侧修建防御塔射杀这些在路上傻傻的怪物。

当然如果你以为一维路线就如此简单那就大错特错了，那只是最开始的塔防。现在的塔防已经发展到各种路线，虽然怪物还是在指定路线上傻傻的走着，但是多种多样的进攻路线，为玩家建造

防御塔的投入选择上增加了很大的策略性。

上图只是众多路线中比较简单的一种——双路线，双刷怪点。这种关卡布局的好处有二。一是玩家需要对路线进行猜测，然后决定那一条为主进攻路线，方便投入更多资源来防守；二是多路线容易形成交汇点，这种战略意义很高的点，至于具体什么意义，有什么效果，我会在下面塔位置中详细分析。

《植物大战僵尸》在路线上也有很大的创新，它设定为 5 路并行，让玩家在资源分配上有了更多的选择策略。当让路线上只不过是它众多创新中的一小点。

1.2 二维路线

虽然看上去二维路线只比一维路线中怪物行进方向多了个左右移动的维度，但是实际效果上却大大增加了策略意义，单同时对程序大兄弟的技术考验也增加了很多（寻路、碰撞什么的，跟策划无关，我就不说了）

这里说的左右可不是指路线宽一点，怪物可以略微摆动，而是没有路线。怪物可以从四面八方来进攻，玩家需要考虑到各种情况。当然我这么说你可能很陌生，WAR3 的最新这一代塔防地图也有很多类似的，但是我还是选一个大家更为熟悉的例子吧——《部落冲突》COC，这款游戏的影响我就不多说了，是有目共睹的大成功，说是塔防类游戏的改革都不夸张，这款游戏给塔防

类带进了新的时代。

有兴趣的玩家可以去查一下 COC 的布局文章，相信网上有很多。这种开放式路径的塔防游戏（COC 主要还是 PVP，相当于攻防结合），在防守策略上有了很大的扩展，调动了玩家的积极性（针对策略型玩家），玩家们在攻防间的争夺使其逐渐升温。同时舍弃了固定的刷怪点和防守点，玩家决定出怪点，然后把防守点分配到每个建筑上，让防守玩家有所取舍。

场景

2.1 场景物品

在塔防游戏发展上，游戏设计者逐渐考虑塔防类游戏的每一寸空间，从最开始的 WAR3 上大片空地，到逐渐加一些小的陪衬景物，到现在的连景物都不放过…为了给游戏增加策略性和趣味性，原来只起到陪衬作用的景物也开始被配置了各种奖励，被攻击。这个要数《保卫萝卜》做的最好

了，当然《保卫萝卜》的成功不仅仅是因为这个，还有它的关卡产出消耗配置，这个详细的解释我会在怪物那里单独分析。

《保卫萝卜》利用塔防类游戏共有的一点——下一波等待期间和空闲防御塔等待时间，充分利用每一秒，前期让玩家有充分的操作来探索场景物品带来的奖励，外加破坏场景来增添塔位置。

2.2 场景建筑

当然也有一些和平的设计者，选择把场景物品加入到玩家的战斗中去。比如最开始给玩家设定一些破损的建筑，玩家需要用资源修复该建筑，然后这个场景建筑会给玩家提供一定程度上的帮助。《KINGDOM RUSH》中就有运用到这一方法。

上图是一个猎手大厅，玩家修复后可以召唤弓手来在路线上进行防御。KR 是 TD 类游戏一个重要的代表作，很多地方值得我们去学习，下面我还会用到这个游戏当例子。

2.3 场景限制

并不是所有场景都是一帆风顺的建造，有时候还会给玩家出一下建造限制，这个做的比较经典的就是《植物大战僵尸》了。相信很多人都玩过，没玩过的人极力推荐去尝试，这里我就不放图片了。

其主要限制分为两类。一是建筑限制，泳池关卡和房顶关卡相信大家还记忆犹新，都是需要先建造荷叶或者花盆才能在放置植物。不过这种限制一定要让玩家觉得自然，比如植物确实需要花盆才能放屋顶，因为植物不能长在瓦片上（＝＝！现实中瓦片中的泥土另算）。二是视野限定，大雾关卡大家应该也有印象。就是大雾遮罩下，导致僵尸到了很近的位置才会被发现，这种限制对玩家有很强的应变性要求。玩家需要预测或者贮存资源作为应急手段。

塔位置

3.1 固定位置

固定位置是指防御塔被要求只能建造在路线旁的指定位置上，但是这个位置并不是随意设定了。如果玩家发现一个重要的策略点上没有那个建塔的位置，玩家会觉得很不舒服，就好比掰开花生壳却发现里边没有花生一样郁闷。

固定位置需要注意几点。一是数量上要足够，不要出现玩家把所有位置都建造满了还是无法通关的死局；二是位置上要精巧，重要的交汇点一定不能落下，否则玩家绝对会吐槽。这方面《KINGDOM RUSH》上做的非常好。

当然不可能仅仅通以上几点就能布置出，还要结合数值，比如最后一波怪物的战斗力是否小于防御塔战斗力总和，战斗力什么的就让数值童鞋帮你弄吧，因为不同防御塔的各类 DEBUFF 和攻速、射程一大堆参数决定的。不过也不用过于精确，因为即使 KR 中，也没有把所有空位建满，有些空位只是为了布局美观和多选择点而已。

3.2 任意位置

任意位置的话就简单多了。一种是 TD 游戏初期，WAR3 路边的那种乱建乱造的防御塔，另一种则是 COC 中的高策略性、玩家自行布阵的模式。当然任意位置在增添策略性的同时也有很大弊端，那就是计算困难。策略性高就代表着可控性低，策划在计算的时候容易出现很大偏差，只能给出一个合理的范围。而期间策划需要把握的度就需要很多的经验和尝试了。

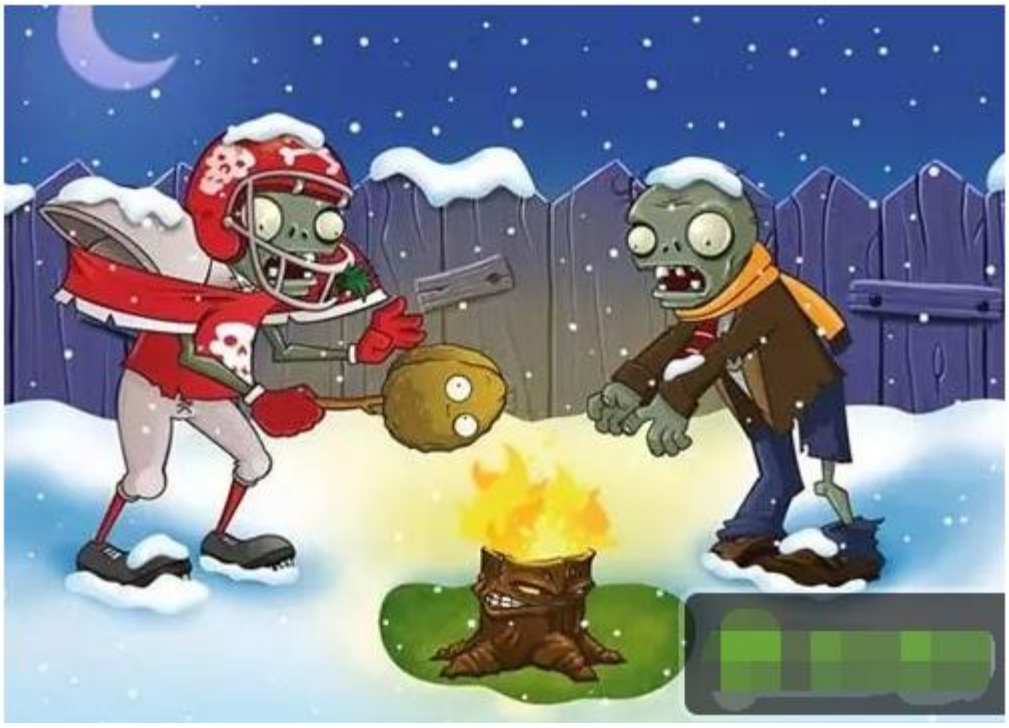
怪物

4.1 种类

怪物肯定不能是千律一变，要有各种特点，才能产生不同的策略需求。怪物主要分为以下几类：均衡类，肉盾类，强攻类，急速类，密集类等。分别对应的防御塔为：初级塔，穿透塔，肉盾塔，射速塔，区域塔。

所谓的均衡类就是最初接触 TD 时出现的怪物，所有属性都不高，没什么特色。接下里就会出现肉盾类怪物，厚血量高防御注定只能用穿透塔来消灭。强攻类怪物只有在对抗塔防类游戏中才会出现，比如《植物大战僵尸》中的橄榄球员僵尸，这类怪物攻高血厚，需要肉盾塔在前面拖延，由后面的防御塔消灭。急速类就是指快速移动的怪物，不过血量很少，需要高攻速的防御塔来消灭。密集类则是一小队数量多但是属性低的怪物，需要 AOE 攻击的防御塔来消灭。

实际战斗中多种情况结合，战况十分复杂，以上几类只是简单的概括基本的要素。



4.2 波次

每波怪物总战斗力的配置要小于已通过波数总资源加初始资源转化为防御塔的战斗力。简单说就是之前打怪给的钱买的塔，能消灭这一波怪，没通过的话那就是玩家策略上资源分配的失误。这个说起来比较难，有兴趣的人可以去尝试一下 KINGDOM RUSH 这款 TD 游戏，这款游戏的波次配置相当完美，玩家从开始到结束都在思考资源分配的问题，基本不会出现前期紧张的要死，后期相当于挂机的配置失误。每一波对战斗力的把握十分精确。

这里也顺便说一下最初的教学关卡配置问题吧。最开始玩家接触 TD 游戏时，要单独配置几波特色怪物来体现防御塔的特点，比如提示玩家建造穿透塔，然后上几个肉盾类怪物，让玩家爽的同时充分表达出防御塔对怪物种类的克制关系，这会让玩家在后面布置上更容易。

4.3 特点

4.1 中所说的怪物只是最基本的分类，当然还需要一些特殊的怪物来给玩家一些意想不到的惊喜，或者说给一些特殊的怪物配一些独有的技能。这一点在《植物大战僵尸中》做的非常好，有各种各样的僵尸，玩家需要根据不同的僵尸种植不同的植物，以及及时替换植物来进行应对。

塔

从这节开始下面就是玩家考虑的元素了，不过设计关卡的话还是要是要掌握熟悉这些元素，才能更好的设计出玩家体验感更好的关卡。

5.1 分类

首先防御塔的分类是与怪物基本对应的。这里的对应是指该类防御塔针对某类怪物的效果最强。关卡配置这种怪物时，玩家第一反应是建造该类防御塔去应对，让玩家形成最基本的策略反应，这样才有利于后期发展不同的策略意义。

同时根据防守路线或位置的不同也要防止不同类型的防御塔，让防御塔的效果达到最大化。比如KR 中某个角落的防御塔范围与进攻路线交集较小，这种情况下就更适合建造兵营，因为兵营跟射程没有太大关系，它只是负责在路线上放置士兵。

5.2 消耗

玩家在资源分配上会考虑两种情况：一是升级原有防御塔，二是建造新的防御塔。一般玩家都会优先去选择建造新的防御塔，因为建造新塔反馈给玩家的视觉效果更直接，而升级原有防御塔只有在低级防御塔效果太差或者位置不够的情况下才会去做。

关卡在配置关卡产出时要首先考虑到期望玩家该关最终效果。比如该关的总产出足够建造几个几级塔，是什么样的战斗效果。这一点在《保卫萝卜》上做的十分出色。他们关卡设定的最终效果是场景中有很多高级塔，战斗效果各种塔的攻击特效展现，十分酷炫。而为了达到这种效果，他们关卡的资源投放也很多。不过要考虑一点就是一定要有相同数量和质量的怪物来让玩家消灭。

5.3 性价比

这个就是指玩家建造或者升级防御塔时，针对不同防御塔进行的选择，而这种选择的依据一般是根据下一波怪物的提示进行的。这个信息要通过关卡 UI 来显示出来。不过现在很多游戏的怪物单关卡类型变化没有那么明显，估计是考虑到玩家的策略能力问题。所以现在的塔防仅剩的一点策略就是建造顺序了，主要还是让玩家很轻松的就能体验到 TD 类游戏的快感。

魔法

现在由于玩家在策略能力上的退化，游戏设计者们不得不设计一些额外的系统来帮助游戏玩家缓冲，比如消灭怪物或者召唤防御者。当然如果设计得当还会起到一定的策略意义，比如由于魔法 CD 的存在，施放时机和位置也是三思而后行。

6.1 进攻类

此类魔法多为 AOE，主要用来消灭汇聚在交叉路口的怪物群，每局有 CD 或者次数限制。

6.2 防御类

此类魔法多为召唤类，召唤临时工在道路上阻碍怪物进攻，让怪物停滞在释放的点上，用于让防御塔输出最大化。

英雄

7.1 作用

英雄系统在 TD 类游戏中出现的越来越多了，从策划角度来看，这无疑是一块大蛋糕，可以在英雄的成长系统中给玩家挖很大的坑（付费点），而玩家也乐此不彼的使用着英雄这个帮助效果很大的系统。当然这个系统的存在并不是单纯的为了挖坑，同时也增加了玩家的操作内容，根据该系统在 TD 游戏中所占位置重要性的不同，操作量也会有所改变。

就《KINGDOM RUSH》和《兽人必须死》这两款在英雄系统上极端表现来看，KR 主要还是在塔防系统上，英雄只不过相当于一个可以移动的、高属性的防御兵而已。而《兽人必须死》则是完全不同，玩家主要以操作英雄为主，进行设计，放置陷阱等操作，基本上都属于射击类游戏了。当然陷阱的放置起到的作用还是不容忽视。

7.2 培养

英雄系统谈到培养，基本就已经开始走远了。KR 中的英雄系统就是简单的解锁，英雄在本关升级有效而已。而《兽人必须死》则是各种内容各解锁提升。这里就不赘述了，市面上 RPG 类或者卡牌类游戏的培养系统做的已经十分完善了，想做这方面的人可以去借鉴一下，毕竟现在游戏慢慢的都走向综合型了嘛。

7.3 表现

既然挖坑了就要有一定的诱饵给玩家看，一系列美术资源是最基本的了。策划层面上还要给英雄在战斗中各种操作互动，比如最基本的移动、技能（参考卡牌类游戏）等。还可以在英雄系统上做消耗，比如关卡中死亡后，玩家需要花费资源复活或者花很长时间等之类的设定。

还有一点需要注意的是，如果你的英雄操作量不够多，那么最好就用数量来弥补，毕竟还是一个很大的系统，如果玩家没有充分利用到的话，不仅策划浪费，玩家也会觉得可玩性少，适得其反。

6、微信公众平台及服务号

Golang 语言社区

Golang 技术社区