

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。

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](http://www.ByteEdu.Com)

腾讯课堂地址：[Gopher.ke.qq.Com](http://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 .....	2
第五节 Go 语言指针 .....	2
1、指针相关 .....	2
2、对象与指针 .....	10
3、使用 reflect 探究 struct 成员方法接收者指针 .....	11
4、课后作业—难度：★ ★ ★ ★ ☆ .....	14
5、设计出优秀游戏关卡的六大步骤 .....	15
6、微信公众平台及服务号 .....	19

# 第一季 Go 语言基础、进阶、提高课

## 第五节 Go 语言指针

### 1、指针相关

指针的概念：

概念	说明
变量	是一种占位符，用于引用计算机的内存地址。可理解为内存地址的标签
指针	表示内存地址，表示地址的指向。指针是一个指向另一个变量内存地址的值
&	取地址符，例如：{指针}:=&{变量}
*	取值符，例如：{变量}:=*{指针}

### 内存地址说明

计算机的内存 RAM 可以把它想象成一些有序的盒子，一个接一个的排成一排，每一个盒子或者单元格都被一个唯一的数字标记依次递增，这个数字就是该单元格的地址，也就是内存的地址。

内存定义：



**硬件角度：**内存是 CPU 沟通的桥梁，程序运行在内存中。

**逻辑角度：**内存是一块具备随机访问能力，支持读写操作，用来存放程序及程序运行中产生的数据的区域。

概念	比喻
----	----

概念	比喻
内存	一层楼层
内存块	楼层中的一个房间
变量名	房间的标签，例如：总经理室
指针	房间的具体地址（门牌号），例如：总经理室地址是 2 楼 201 室
变量值	房间里的具体存储物
指针地址	指针的地址：存储指针内存块的地址

## 内存单位和编址

### 内存单位：

单位	说明
位（bit）	计算机中最小的数据单位，每一位的状态只能是 0 或 1
字节（Byte）	1Byte=8bit，是内存基本的计量单位
字	“字”由若干个字节构成，字的位数叫字长，不同档次的机器有不同的字长
KB	1KB=1024Byte，即 1024 个字节
MB	1MB=1024KB
GB	1GB=1024MB

### 内存编码：

计算机中的内存按字节编址，每个地址的存储单元可以存放一个字节的数据，CPU 通过内存地址获取指令和数据，并不关心这个地址所代表的空间在什么位置，内存地址和地址指向的空间共同构成了一个内存单元。

内存地址：

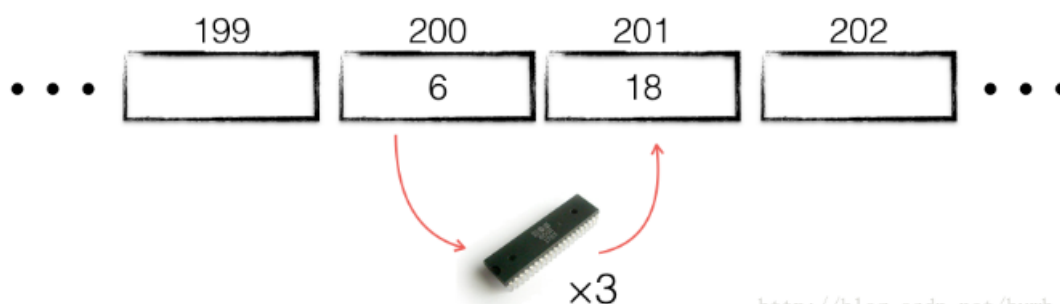
内存地址通常用 16 进制的数据表示，例如 0x0ffc1。

## 变量与指针运算理解

编写一段程序，检索出值并存储在地址为 200 的一个块内存中，将其乘以 3，并将结果存储在地址为 201 的另一块内存中

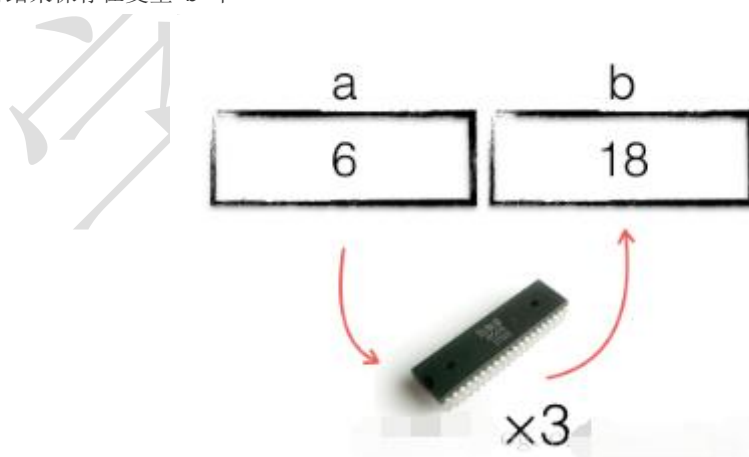
本质

1. 检索出内存地址为 200 的值，并将其存储在 CPU 中
2. 将存储在 CPU 中的值乘以 3
3. 将 CPU 中存储的结果，写入地址为 201 的内存块中



## 基于变量的理解

1. 获取变量 a 中存储的值，并将其存储在 CPU 中
2. 将其乘以 3
3. 将结果保存在变量 b 中



```
var a = 6  
var b = a * 3
```

## 基于指针的理解

```
func main() {  
    a := 200  
    b := &a  
    *b++  
    fmt.Println(a)  
}
```

以上函数对 a 进行+1 操作，具体理解如下：

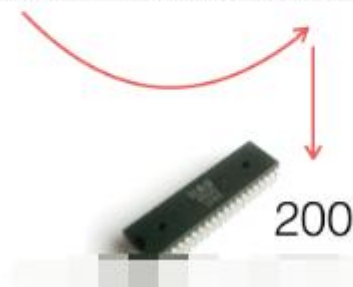
1. a:=200

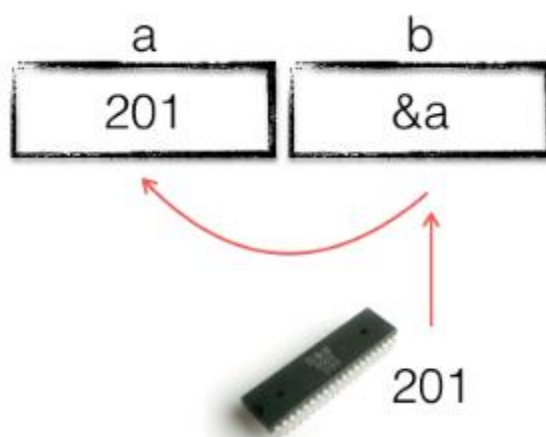


2. b := &a



3. \*b++





### 一般的指针情况:

```
var pointer *int;  
pointer = new(int);  
*pointer = 3;  
fmt.Println(*pointer);
```

### 多重指针情况:

```
var outer **int;  
var inter *int;  
inter = new(int);  
*inter = 3;  
outer = &inter;  
// 地址一样  
fmt.Println(inter);  
fmt.Println(*outer);  
// 值一样  
fmt.Println(*inter);  
fmt.Println(**outer);
```

### 另外指针的情况:

```
var inter *int;  
var outer **int;  
inter = new(int);  
*inter = 3;  
outer = new(*int);
```

```
*outer = inter;
// 地址一样
fmt.Println(inter);
fmt.Println(*outer);
// 值一样
fmt.Println(*inter);
fmt.Println(**outer);
```

## 函数值上面都是在玩指针，下面看看基本的数据结构。

基本的数据结构有：数组与结构体（map 和树之类的不在讨论范围）

golang 中的数组与 C 中的数组有很大的差别

golang 中的数组是这样说的：Arrays are values, not implicit pointers as in C.

1. 数组做参数时，需要被检查长度。
2. 变量名不等于数组开始指针！
3. 不支持 C 中的  $*(ar + \text{sizeof(int)})$  方式的指针移动。需要使用到 **unsafe** 包
4. 如果 p2array 为指向数组的指针，`*p2array` 不等于 `p2array[0]`

例子 1 数组做参数时，需要被检查长度。

```
func use_array(args [4]int) {
    args[1] = 100
}

func main() {
    var args = [5]int{1, 2, 3, 4, 5}
    use_array(args)
    fmt.Println(args)
}
```

编译出错: cannot use args (type [5]int) as type [4]int in function argument, 需要有长度上的检查

例子 2 变量名不等于数组开始指针！

```
func use_array(args [4]int) {
    args[1] = 100
}

func main() {
    var args = [5]int{1, 2, 3, 4, 5}
    use_array(args)
    fmt.Println(args)
}
```

输出结果是 [1 2 3 4]，没有保存结果，数组名的用法与 C 的不一样。在 golang 里是这样的：

```
// 又长度检查, 也为地址传参
func use_array(args *[4]int) {
    args[1] = 100 //但是使用还是和 C 一致, 不需要别加"*"操作符
}

func main() {
    var args = [4]int{1, 2, 3, 4}
    use_array(&args) //数组名已经不是表示地址了, 需要使用"&"得到地址
    fmt.Println(args)
}
```

例子 2 如果 p2array 为指向数组的指针, \*p2array 不等于 p2array[0]

对比一下 C 和 golang 在这方面的差别:

```
void main(int argc, char *argv[]) {
    int *p2array;
    p2array = (int *) malloc(sizeof(int) * 3);
    //等于 p2array[0]
    *p2array = 0;
    printf("%d\n", *p2array + 1);
}
```

\* 输出为 1

```
func main() {
    var p2array *[3]int ;
    p2array = new([3]int);
    fmt.Printf("%x\n", *p2array + 1); //不管 p2array 是指针变量还是数组变量, 都只能使用"[]"方式使用
}
```

\* 报错.

// 翻墙

在 [http://golang.org/doc/go\\_spec.html#Selectors](http://golang.org/doc/go_spec.html#Selectors) 一节中有描述。

## 指针的使用

### 方法中的指针

方法即为有接受者的函数, 接受者可以是类型的实例变量或者是类型的实例指针变量。但两种效果不同。



## 1、类型的实例变量

```
func main(){
    person := Person{"vanyar", 21}
    fmt.Printf("person<%s:%d>\n", person.name, person.age)
    person.sayHi()
    person.ModifyAge(210)
    person.sayHi()
}

type Person struct {
    name string
    age int
}

func (p Person) sayHi() {
    fmt.Printf("SayHi -- This is %s, my age is %d\n", p.name, p.age)
}

func (p Person) ModifyAge(age int) {
    fmt.Printf("ModifyAge")
    p.age = age
}

//输出结果
person<vanyar:21>
SayHi -- This is vanyar, my age is 21
ModifyAgeSayHi -- This is vanyar, my age is 21
```

尽管 `ModifyAge` 方法修改了其 `age` 字段，可是方法里的 `p` 是 `person` 变量的一个副本，修改的只是副本的值。下一次调用 `sayHi` 方法的时候，还是 `person` 的副本，因此修改方法并不会生效。

即实例变量的方式并不会改变接受者本身的值。

## 2、类型的实例指针变量

```
func (p *Person) ChangeAge(age int) {
    fmt.Printf("ModifyAge")
    p.age = age
}
```

Go 会根据 `Person` 的示例类型，转换成指针类型再拷贝，即 `person.ChangeAge` 会变成 `(&person).ChangeAge`。

指针类型的接受者，如果实例对象是值，那么 go 会转换成指针，然后再拷贝，如果本身就是指针对象，那么就直接拷贝指针实例。因为指针都指向一处值，就能修改对象了。

## 2、对象与指针

```
package main

import (
    "fmt"
)

type Person struct {
    age int
}

func Create(a int) (p Person) {
    return Person{age: a}
}

func Add(p Person) {
    p.age += 10
}

func Add1(p *Person) {
    p.age += 10
}

func main() {
    p := Create(10)
    fmt.Println(p)

    Add(p)
    fmt.Println(p)

    Add1(&p)
    fmt.Println(p)
}
```

运行结果如下：

```
{10}
```

```
{10}
```

```
{20}
```

可以看到对象是没有改变的，指针是改变的

### 3、使用 reflect 探究 struct 成员方法接收者指针

#### 问题背景

Go 语言的面向对象在概念上与 java 大同小异，但是由于 Go 语言在给 struct 添加 method 的时候需有一个显示的接收者（receiver），receiver 可以是指针类型或是 struct 的形参，二者到底有啥区别是在学习 Go 面向对象内容时最容易糊涂的地方。

#### 问题描述

为了更好的阐述问题，首先撸上一段小代码：

```
//定义一个名为 Controller 的类
type Controller struct {
    domain string
    count int
}

//给 Controller 添加第一个成员方法 firstFunc
func (c *Controller) firstFunc(domain string, count int) {
    c.domain = domain
    c.count = count
    fmt.Println("firstFunc's domain is "+c.domain+" count is ", c.count)
}

//给 Controller 添加第二个成员方法 secondFunc
func (c Controller) secondFunc(domain string, count int) {
    c.domain = domain
    c.count = count
    fmt.Println("secondFunc's domain is "+c.domain+" count is ", c.count)
}
```

如上，在 Go 中定义类用 struct 类型，在 Controller 类有两个 field，分别为 domain 和 count，还添加了两个方法，分别是 firstFunc(domain string, count int) 和 secondFunc(domain string, count int)，不同的是 firstFunc 的 receiver 是 Controller 的指针而 secondFunc 传入的 receiver 是 Controller 的形参。那么问题来了不同类型的 receiver 具体会有哪些区别呢？下面将以笔者拙见进行总结归纳，分享之余也用于给自己记录。

## 问题解答

对类字段值修改结果不同

这个问题根据指针的意义就能理解，receiver 为指针时相当于类的引用，`*Controller.domain=domain` 修改了 Controller 中 domain 字段的值，receiver 为形参时，相当于 Controller 类的一个 copy 值，因此 `Controller.domain=domain` 只是修改了 Controller 的一个副本的字段值，并非原 Controller 本身字段值。代码验证如下：

```
func main() {  
  
    //创建一个名为 controller 的 Controller 实体指针并初始化字段值  
    controller := &Controller{  
        domain: "www.baidu.com",  
        count: 1000,  
    }  
  
    //输出 controller 原始字段值  
    fmt.Println("controller's original domain is "<"+controller.domain+"> count  
is ", controller.count)  
  
    //调用 firstFunc 并查看 controller 字段值  
    controller.firstFunc("www.sohu.com", 500)  
    fmt.Println("firstFunc>controller's domain is "<"+controller.domain+"> count  
is ", controller.count)  
  
    //调用 secondFunc 并查看 controller 字段值  
    controller.secondFunc("www.qq.com", 800)  
    fmt.Println("secondFunc>controller's domain is "<"+controller.domain+"> count  
is ", controller.count)  
}
```

运行结果如下：

---

Controller 实例与 Controller 实例指针包含方法不同

简单概括来讲，类的实例指针（也就是上一段代码中的 controller）包含所有方法，即在以上代码中 controller 包含 firstFunc 和 secondFunc 两个方法，而实例只包含 receiver 为形参的方法，即若 controller 不是指针则 firstFunc 是不能够被成功调用的，因为其 receiver 为指针。为了能够更加直观的进行说明，下面用 reflect 反射机制进行验证：

```
func main() {  
  
    //初始化一个类指针 controller1  
    controller1 := &Controller{  
        domain: "www.baidu.com",  
        count: 1000,  
    }  
  
    //初始化一个类 controller2  
    controller2 := Controller{  
        domain: "www.sohu.com",  
        count: 800,  
    }  
  
    t1 := reflect.TypeOf(controller1)  
  
    //查看 controller1 的类型与包含的方法个数  
    fmt.Println("controller1's type is", t1.Kind(), "include", t1.NumMethod(),  
"methods")  
  
    //查看 controller1 的所有方法名称  
    for j := 0; j < t1.NumMethod(); j++ {  
        fmt.Println(++j, t1.Method(j).Name)  
    }  
  
    //查看 controller2 的类型与包含的方法个数  
    fmt.Println("controller2's type is", t2.Kind(), "include", t2.NumMethod(),  
"methods")  
  
    //查看 controller2 的所有方法名称  
    for j := 0; j < t2.NumMethod(); j++ {  
        fmt.Println(t2.Method(j).Name)  
    }  
}
```

执行结果如下:

---

### 对接口实现的区别

接口通常会规定一些方法,实现了这些方法的类就相当于实现了相应接口,类方法的 receiver 是否为指针对接口实现的影响主要还在于类是否包含接口规定的所有方法。现定义一个接口:

```
type ControllerInter interface {
```

```

    firstFunc(domain string, count int)
    secondFunc(domain string, count int)
}

```

在上一问题验证代码中 controller1 因为包含了 ControllerInter 规定的所有方法, 因此实现了该接口, 而 controller 则没有实现。因此在判断类是否实现了某一接口时要特别注意其成员方法的 receiver 类型以及类的类型, 尽管结果已经很清晰, 但同样还是用代码加以验证:

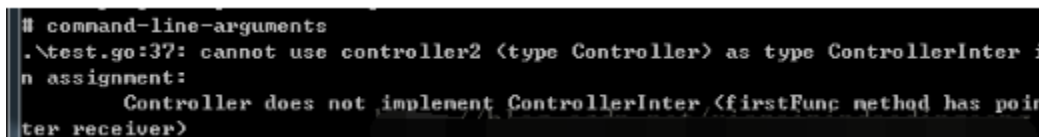
```

func main() {
    controller1 := &Controller{
        domain: "www.baidu.com",
        count: 1000,
    }
    controller2 := Controller{
        domain: "www.sohu.com",
        count: 800,
    }

    var controllerInter ControllerInter
    controllerInter = controller1
    controllerInter = controller2
}

```

执行结果如下:



```

# command-line-arguments
./test.go:37: cannot use controller2 (type Controller) as type ControllerInter in assignment:
    Controller does not implement ControllerInter (firstFunc method has pointer receiver)

```

声明一个名为 controllerInter 的接口变量, 并将 controller1 和 controller2 依次复制给它, 实现了接口的将顺利执行, 没有实现的将会报错, 如上图中 controller2 does not implement ControllerInter 接口。

#### 4、课后作业—难度: ★ ★ ★ ★ ☆

实现一个游戏网络 net 的链接存储机构:

- <1> 指针保存链接信息, 链接信息目前可以用“NULL”字符串代替。
- <2> 检测链接数据信息的内存地址, 在每次 update<1>步骤重复操作是否变化
- <3> 实现主动出现 panic 异常, 清除 map 结构函数

## 5、设计出优秀游戏关卡的六大步骤

关卡设计是关于发挥创造性，吸引玩家并引领着玩家走遍你的游戏。基于不同游戏类型，你设计关卡的方法以及使用的工具也会有所不同。

当独立游戏设计师需要使用《Super Mario Maker》去创建一个关卡时会发生什么？他们能带给其他新人设计师什么样的教训？这些问题便是我们在今天将要探索的内容。

几周前的一次经历鼓舞了我们写下这篇文章。我们受邀在 EHX 展上展示《皮影木偶》。而在出发前几周，我们遇到了一个很有趣的挑战：即我们是否能够使用全新游戏《Super Mario Maker》去创建一个关卡并将其带到 EGX 上。而这一挑战就好像在生活中有人问我们“你想要来块美味的巧克力蛋糕吗？”一样，我们的回答总是“天哪，我要！”

在《Super Mario Maker》这款游戏中，你能够创造属于自己的超级马里奥关卡，在网上分享它们，并与其他创造者一起体验这些关卡。



SuperMarioMaker(from gamasutra)

这是我们第一次尝试《Super Mario Maker》，我们也不知道结果会是怎样。我们发现这种编辑程序是进入关卡设计过程的一种好方法，所以我们想要把握住这次机会向你们分享一些有关该主题的建议。我们将以我们在《Super Mario Maker》上创造的关卡“Super Shadow Puppeteer”为例去分享关卡设计中真正有效的步骤。

免责声明：根据个人喜好与经历，这些内容也会有所变动。

### 关卡设计

任何人都可能成为一名“关卡设计师”，并且许多人想要成为一名“关卡设计师”。但是想要成为一名优秀的关卡设计师却并不容易。大多数游戏都有属于自己的完全不同的关注点，目标与风格，但其实从核心看来很多步骤都是通用的。

## 1.了解可以使用怎样的工具和机制

这需要花费一些时间，但当你完全熟悉这些可能性时，你便能够创造出更棒且更具创造性的关卡。

在《SMM》中，你需要逐渐呈现出游戏风格，道具和效果。也许这么做会有点麻烦，但是游戏希望提供给你机会去逐渐熟悉自己的选择。通过尝试每种道具，你将能够更快打开这些内容。

建议：尽可能随性点，放置各种道具，在这一阶段你不用担心内容是否“完美”。如果你在这个关卡中放置更多道具，你便能够更快打开这个过程。

## 2.决定你的目标用户并调整难度

你要考虑什么类型的玩家会玩你的游戏/关卡。他们的年龄？他们在游戏中的目标？他们是否有游戏的经验？他们是否想要探索，挑战，或者收集东西？

这是一个巨大的过程，即包括平台和类型等选择。而在《SMM》中事情变得不那么复杂，但却也未因此变得不重要。

你需要根据玩家类型去决定《SMM》的关卡类型。你是想瞄准那些想要迎接更多挑战的硬核玩家还是那些更休闲的“马里奥”粉丝？难度并不总是等同于“乐趣”。创造者非常乐于看到玩家在尝试关卡时遭遇失败，但是玩家却也可能因此而受挫。所以你应该慎重考虑玩家类型与难度级别。

你应该大胆地去挑战玩家，但也需要把握好挑战与挫折间的度。《SMM》宣称将在 11 月 4 日发行一个免费更新内容，即将在游戏中添加一些检查点。

你们需要牢记，较长的关卡不应该太难，而超级复杂的关卡最好保持较短或者设置一些有效的检查点，提供一些能够激励玩家继续前进的内容。

来自 IGN 的建议：“你也可以创造一些捷径，即让玩家在到达中间点后能够发现这些捷径。例如安放于关卡中途的管道能将玩家带回开始处——反之亦然。”

对于我们创造的关卡，我们知道主要受众应该是那些参加 EGX 的人。而为了拥有属于自己的粉丝，我们决定遵循《皮影木偶》的原则：不要太复杂，比起让玩家受挫，更应该提供给他们更多乐趣，让玩家觉得自己其实很聪明。即不需要经历太多次死亡便能够感受到乐趣。比起让玩家不断避免死亡，让他们能在移动与解决谜题中获得挑战会更有意思。

## 3.决定主要游戏玩法——核心

选择一种风格以及一些游戏机制——基于此去创造所有内容。你的关卡的主要动作是什么？跳跃，滑动，射击，闪避，探索或者解决谜题？

很多人会担心游戏中缺少多样性。而在此真正重要的东西是深度，而非复杂性。即使只专注于一个机制，你也可以创造出一款优秀的游戏。你应该确保自己能够尽可能多地探索一个机制，然后再考虑是否添加其它机制。

在《SMM》，关于你能够专注于什么内容的选择非常有限，但是如果你具有绝对的创造性，你便会惊讶于自己能够创造的内容。你甚至能够创造出一些别人认为不可能的东西。



对于我们的关卡，我们希望玩家能够从中回想起《皮影木偶》。毕竟我们还要前往 EGX 去展示这款游戏。

《皮影木偶》的核心是关于两个玩家之间的同步平台游戏和谜题解决机制。游戏使用了灯光和重力改变作为主要的多元化元素。而因为《SMM》不具有两个玩家的模式，所以灯光机制或动力改变机制会有点受限。

但是《超级马里奥》系列与《皮影木偶》在游戏玩法中也拥有一些共同点，我们也发现带有一个游戏世界的马里奥风格游戏总是能够为我们呈现出华丽的视觉效果。

#### 4.概述游戏玩家的“故事”

就像交响乐拥有基于不同感觉和速度的不同动作，你也想要创造动态化的关卡“故事”。也就是说你要考虑玩家在整个关卡中的体验。他们在一开始的感受是怎样？在之后的感受又是怎样？你应该完善他们的体验和情感变化，并提供给他们更具奖励性和满足感的体验。

这是关卡设计中一种典型的模式：首先你需要引进一个游戏玩法元素，然后你将重复行动，让玩家习惯游戏机制，随后你将扩展其使用与多样性。在经过多次引进后，你便能将玩家带到最终测试了。

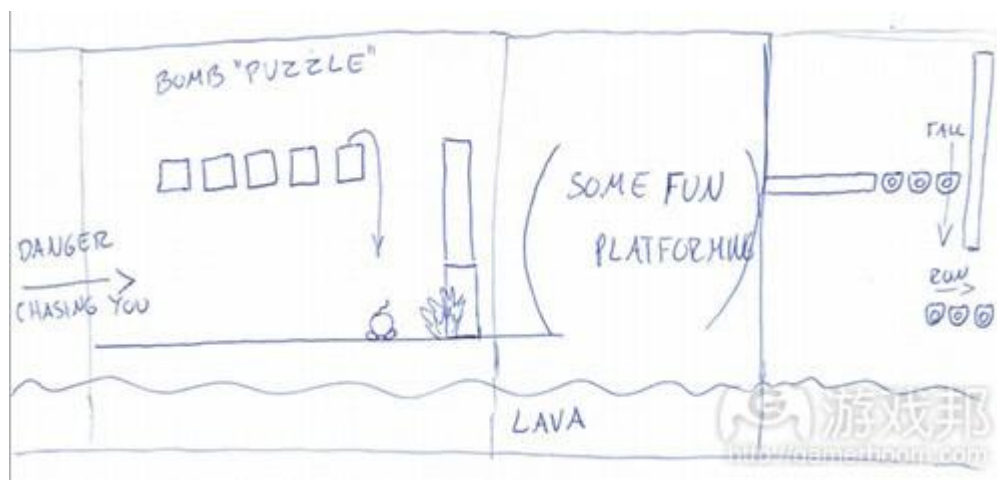
对于我们的关卡，我们希望游戏体验和难度都带有《皮影木偶》的影子。从一些特定关卡中获取了灵感，我们创造了一个关于休闲探索的主要关卡以及模仿两个可怕的追逐序列的次关卡。

而我们有意将次关卡设置为与主关卡完全不同的内容，不过其难度却是相似的。并且玩家能够在此感知到的危险和威胁也远远大于实际。

#### 5.草拟出主要元素，为创造性留下足够空间

经历了第 3 和第 4 个步骤，你可能对游戏玩法以及整体的游戏流程已经有了一些不错的想法。在纸上勾勒出这一想法是帮助你规划游戏关卡的一种快速且简单的方法。

你必须清楚自己是不可能事先规划好每一个绝妙的想法。所以你最好能在规划好的游戏玩法之间留下足够的空间。之后你可能还会因为所使用的工具而获得灵感。当你在关卡中设置好那些已经规划好的游戏元素时，你将能够获得一些不错的理念。如果你要进行适当的规划，你可以记录下怎样的中间性游戏玩法更适合自己的“故事”。就像如果你只是提供给玩家一个挑战，那么在中间提供给他们一个奖励或一些有趣但却简单的内容将能让他们在迎接下一个挑战前先喘口气。



drawing(from gamasutra)

我们为自己的主关卡和次关卡选择了一些独特的风格。主关卡拥有一些有机平台，有些是静态的，有些是动态的。它突出了一些能够晃荡的葡萄藤以及会让玩家感到紧张的甜甜圈形状的组块，同时还有一些小小的贝壳形状的敌人。该区域拥有几条可选择的路线以及许多不错的奖励。



Main\_Sub\_Level(from gamasutra)

而我们的次关卡主要是由一些“人造”元素所构成：如砖块和管道，墙壁和窗户。我们添加了熔岩和炸弹作为核心元素为玩家制造压力与挑战。在这里我们需要确保带给玩家压力，但却不会让他们轻易死亡；就像我们提供了一些会让他们遭受到来自 Twompers 的攻击的蘑菇。

## 6.通过测试去保证游戏流程与“叙述”的合理运行

确保你能多次尝试游戏关卡并判断游戏流程是否有效运行以及难度设置是否合理。邀请一些之前从未尝试过你的关卡的人帮你进行测试。你将会从他们的游戏进程以及失败中学到许多。

《SMM》拥有非常出色的设置，即每次当你死亡时你便能够进入关卡编辑器并从上次离开的位置重新开始游戏。如此一来调整与优化过程也就变得更加简单且更加有趣了。

在这里添加像货币等额外元素既能引导玩家也能装饰游戏，从而让关卡显得更加华丽。同时添加一些额外奖励也能够放慢玩家的速度并提供给他们一些具有选择性的挑战。

## 结论

设计一个游戏关卡不只是在编辑器上乱忙活，你还能够独自决定该如何做这件事。关键是你需要考虑你想提供给玩家怎样的体验，你还要了解他们的喜好，并为此不断进行迭代。

我们非常喜欢我们的《Super Mario Maker》关卡，我们也从制作过程中获得了许多乐趣。真心希望这个例子能够带给你们帮助和灵感。

## 6、微信公众平台及服务号



Golang 语言社区



Golang 技术社区