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**

字节教育

目录：

第一季 Go 语言基础、进阶、提高课

第六节　Go 语言结构体、切片、map,游戏用户留存

1、结构体

结构体用来定义复杂的数据结构，存储很多相同的字段属性；go 语言中的 struct 成员可以是任何类型，如普通类型、复合类型、函数、struct、interface 等

结构体的定义以及简单实用：

```
package main

import (
    "fmt"
)

func main() {
    type Student struct { //定义结构体
        name string
        age  int
    }
    s1 := new(Student) // 定义指向结构体的指针
    s1.name = "xiaomu"
    s1.age = 10
    fmt.Printf("name:%s\nage:%d\n", s1.name, s1.age)
}
```

结构体定义的三种方式，例如上面的 Student 类型，有如下方式定义

①var s1 Student　　在内存中直接定义一个结构体变量

②s1 := new(Student)　在内存中定义一个指向结构体的指针

③s1 := &Student{}　　同上

通过以下方式获取存储的值

①s1.name

②s1.name 或者(*s1).name

③同上

字节教育

struct 中的“构造函数”，称之为工厂模式，见代码

```
package main

import (
    "fmt"
)

type Student struct { //声明结构体
    Name string
    Age  int
}

func NewStudent(name string, age int) *Student { // 返回值指向 Student 结构体的指针
    return &Student{
        Name: name,
        Age:  age,
    }
}

func main() {
    s1 := NewStudent("xiaomu", 123) // 声明并且赋值指向 Student 结构体的指针
    fmt.Printf("name: %s\nage: %d", s1.Name, s1.Age)
}
```

特意声明注意事项!!!

结构体是值类型，需要使用 new 分配内存

匿名字段：

```
package main

import (
    "fmt"
)

func main() {
    type Class struct {
```

```
        ClassName string
    }
    type Student struct { //定义结构体
        name  string
        age   int
        Class // 定义匿名字段，继承了该结构体的所有字段
    }
    s1 := new(Student) // 定义指向结构体的指针
    s1.ClassName = "xiaomu"
    fmt.Printf("ClassName:%s\n", s1.ClassName)
}
```

结构体的方法：

结构体方法的使用：

```
package main

import (
    "fmt"
)

type Student struct { //定义结构体
    name string
    age  int
}

func (stu *Student) OutName() { // 定义 Student 方法
    fmt.Println(stu.name)
}

func main() {
    s1 := new(Student) // 定义指向结构体的指针
    s1.name = "xaiomu"
    s1.OutName()
}
```

结构体继承结构体，其中被继承结构体的方法全部为继承结构体吸收（吸星大法）

```
package main

import (
    "fmt"
```

```
)

type ClassName struct {
    className string
}

func (cla *ClassName) OutClassName() {
    fmt.Println(cla.className)
}

type Student struct { //定义结构体
    name      string
    age       int
    ClassName // 继承 ClassName 结构体的所有
}

func (stu *Student) OutName() { // 定义 Student 方法
    fmt.Println(stu.name)
}

func main() {
    s1 := new(Student) // 定义指向结构体的指针
    s1.className = "xiaomu"
    s1.OutClassName()
}
```

2、切片

数组虽然有适用它们的地方，但是数组不够灵活，因此在 Go 代码中数组使用的并不多。但是，切片则使用得相当广泛。切片基于数组构建，但是提供更强的功能和便利。**切片的类型是 `[]T`，`T` 是切片元素的类型**。和数组不同的是，切片没有固定的长度。切片的字面值和数组字面值很像，不过**切片没有指定元素个数**：

```
letters := []string{"a", "b", "c", "d"}
```

切片可以内置函数 `make` 创建，函数签名为：

```
func make([]T, len, cap) []T
```

`T` 代表被创建的切片元素的类型。函数 `make` 接受一个类型、一个长度和一个可选的容量参数。调用 `make` 时，内部会分配一个数组，然后返回数组对应的切片。

```
var s []byte s = make([]byte, 5, 5) // s == []byte{0, 0, 0, 0, 0}
```

当容量参数被忽略时，它默认为指定的长度。下面是简洁的写法：

```
s := make([]byte, 5)
```

可以使用内置函数 `len` 和 `cap` 获取切片的长度和容量信息。

```
len(s) == 5
cap(s) == 5
```

接下来的两个小节将讨论长度和容量之间的关系。

零值的切片类型变量为 `nil`。对于零值切片变量，`len` 和 `cap` 都将返回 `0`。

切片也可以基于现有的切片或数组生成。切分的范围由两个由冒号分割的索引对应的半开区间指定。

例如，表达式 `b[1:4]`创建的切片引用数组 `b` 的第 `1` 到 `3` 个元素空间（对应切片的索引为 0 到 2）。

```
b := []byte{'g', 'o', 'l', 'a', 'n', 'g'}
// b[1:4] == []byte{'o', 'l', 'a'}, sharing the same storage as b
```

切片的开始和结束的索引都是可选的；它们分别默认为零和数组的长度。

```
// b[:2] == []byte{'g', 'o'}
// b[2:] == []byte{'l', 'a', 'n', 'g'}
// b[:] == b
```

下面语法也是基于数组创建一个切片：

```
x := [3]string{"Лайка", "Белка", "Стрелка"}
s := x[:] // a slice referencing the storage of x
```

切片本质：

一个切片是一个数组切割区间的描述。它包含了指向数组的指针，切割区间的长度，和容量（切割区间的最大长度）。

前面使用 `make([]byte, 5)` 创建的切片变量 s 的结构如下：

长度是切片引用的元素数目。容量是底层数组的元素数目（从切片指针开始）。关于长度和容量和区域将在下一个例子说明。

我们继续对 `s` 进行切分，观察切片的数据结构和它引用的底层数组：

```
s = s[2:4]
```

切片并不复制整个切片元素。它创建一个新的切片执行同样的底层数组。这使得切片操作和数组索引一样高效。因此，通过一个新切片修改元素同样会影响到原始的切片。

```
d := []byte{'r', 'o', 'a', 'd'}
e := d[2:] // e == []byte{'a', 'd'}
e[1] = 'm' // e == []byte{'a', 'm'}
// d == []byte{'r', 'o', 'a', 'm'}
```

前面创建的切片 `s` 长度小于它的容量。我们可以增长切片的长度为它的容量：

```
s = s[:cap(s)]
```

切片生长（复制和最加）：

要增加切片的容量必须创建一个新的、更大容量的切片，然后将原有切片的内容复制到新的切片。整个技术是一些支持动态数组语言的常见实现。下面的例子将切片 s 容量翻倍，先创建一个 2 倍容量的新切片 t，复制 s 的元素到 t，然后将 t 赋值给 s：

```
t := make([]byte, len(s), (cap(s)+1)*2) // +1 in case cap(s) == 0
for i := range s {
    t[i] = s[i]
}
s = t
```

循环中复制的操作可以由 copy 内置函数替代。copy 函数将源切片的元素复制到目的切片。它返回复制元素的数目。

```
func copy(dst, src []T) int
```

3、map（映射）结构

map 基本操作：

```
// 1. 声明
var m map[string]int

// 2. 初始化，声明之后必须初始化才能使用
// 向未初始化的 map 赋值引起 panic: assign to entry in nil map.
m = make(map[string]int)
m = map[string]int{}

// 1&2. 声明并初始化
m := make(map[string]int)
m := map[string]int{}

// 3. 增删改查
m["route"] = 66
delete(m, "route") // 如果 key 不存在什么都不做
i := m["route"] // 三种查询方式，如果 key 不存在返回 value 类型的零值
i, ok := m["route"]
_, ok := m["route"]

// 4. 迭代（顺序不确定）
for k, v := range m {
    use(k, v)
}

// 5. 有序迭代
```

```
import "sort"
var keys []string
for k, _ := range m {
    keys = append(keys, k)
}
sort.Strings(keys)
for _, k := range keys {
    use(k, m[k]
}
```

map 键类型

支持 == 操作符的类型有：

- **boolean**,
- **numeric**,
- **string**,
- **pointer**,
- **channel**,
- **interface**(as long as dynamic type supports equality),
- *以及只包含上述类型*的 **array** 和 **struct**

不支持 == 操作符的类型有：

- **slice**,
- **map**,
- **func**,

补充

1. 不像 Java 可以为 class 自定义 hashcode 方法，以及 C++可以重载==操作符，golang map**不支持**==重载或者使用自定义的 hash 方法。因此，如果想要把 struct 用作 map 的 key，就必须保证 struct 不包含 slice, map, func
2. golang 为 uint32、uint64、string 提供了 fast access，使用这些类型作为 key 可以提高 map 访问速度，详见 hashmap_fast.go

map 并发

map 不是并发安全的，通常使用 sync.RWMutex 保护并发 map：

```
// 声明&初始化
var counter = struct {
    sync.RWMutex // gard m
    m map[string]int
}{m:make(map[string]int)}
// 读锁
counter.RLock()
counter.m["route"]
counter.RUnlock()
// 写锁
counter.Lock()
counter.m["route"]++
counter.Unlock()
```

map 小技巧：

利用 value 类型的零值

```
visited := map[*Node]bool
if visited[node] { // bool 类型 0 值为 false，所以不需要检查 ok
    return
}

likes := make(map[string][]*Person)
for _, p range people {
    for _, l range p.Likes {
        // 向一个 nil 的 slice 增加值，会自动 allocate 一个 slice
        likes[l] = append(likes[l], p)
    }
```

map[k1]map[k2]v 对比 map[struct{k1, k2}]v

```
// map[k1]map[k2]v
hits := make(map[string]map[string]int)
func add(m map[string]map[string]int, path, country string) {
    mm, ok := m[path]
    if !ok {
```

```
        mm = make(map[string]int) // 需要检查、创建子 map
        m[path] = mm
    }
    mm[country]++
}
add(hits, "/", "cn")
n := hits["/"]["cn"]

// map[struct{k1, k2}]v
type Key struct {
    Path, Country string
}
hits := make(map[Key]int)
hits[Key{"/", "cn"}]++
n := hits[Key{"/", "cn"}]
}
```

map 实现细节浅析：

如何计算 hash 值

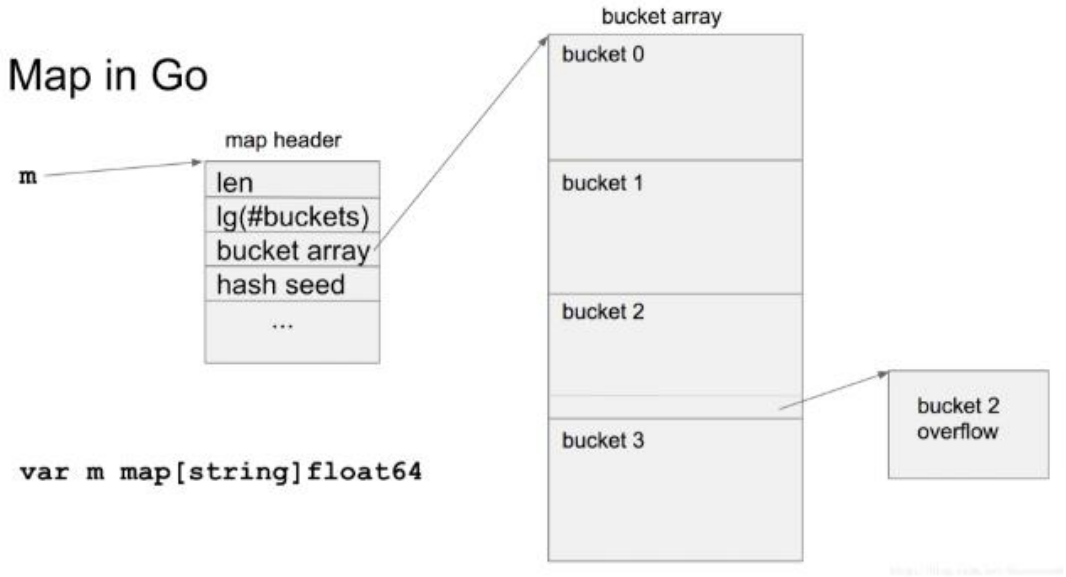
golang 为每个类型定义了类型描述器`_type`，并实现了 **hashable** 类型的`_type.alg.hash`和`_type.alg.equal`。

```
type typeAlg struct {
    // function for hashing objects of this type
    // (ptr to object, seed) -> hash
    hash func(unsafe.Pointer, uintptr) uintptr
    // function for comparing objects of this type
    // (ptr to object A, ptr to object B) -> ==?
    equal func(unsafe.Pointer, unsafe.Pointer) bool
}
```

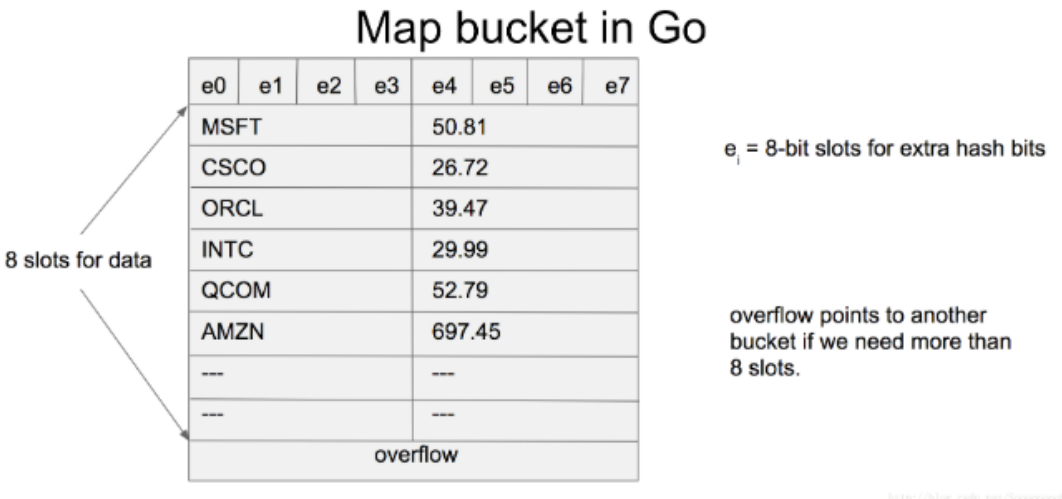
map 实现结构

map 的实现主要有三个 struct，

1. maptype 用来保存 map 的类型信息，包括 key、elem(value)的类型描述器，keysize，valuesize，bucketsize 等；
2. hmap - A header for a Go map. hmap 保存了 map 的实例信息，包括 count，buckets，oldbuckets 等；buckets 是 bucket 的首地址，用 hash 值的低 h.B 位`hash & (uintptr(1)<<h.B - 1)`计算出 key 所在 bucket 的 index；



3. bmap - A bucket for a go map. bmap 只有一个域 `tophash [bucketCnt]uint8`，它保存了 key 的 hash 值的高 8 位 `uint8(hash >> (sys.PtrSize*8 - 8))`；**一个 bucket 包括一个 bmap(toph ash 数组)，紧跟的 bucketCnt 个 keys 和 bucketCnt 个 values，以及一个 overfolw 指针。**



makemap 根据 maptype 中的信息初始化 hmap

```
func makemap(t *maptype, hint int64, h *hmap, bucket unsafe.Pointer) *hmap {
    ...
    // initialize Hmap
```

```
    if h == nil {
        h = (*hmap)(newobject(t.hmap))
    }
    h.count = 0
    h.B = B
    h.flags = 0
    h.hash0 = fastrand()
    h.buckets = buckets
    h.oldbuckets = nil
    h.nevacuate = 0
    h.noverflow = 0
    return h
}
```

如何访问 map

golang 的 maptype 保存了 key 的类型描述器，以供访问 map 时调用 key.alg.hash, key.alg.equal。

```
type maptype struct {
    key           *_type
    elem          *_type
    ...
}
func mapaccess1(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer {
    ...
    // 并发访问检查
    if h.flags&hashWriting != 0 {
        throw("concurrent map read and map write")
    }
    // 计算 key 的 hash 值
    alg := t.key.alg
    hash := alg.hash(key, uintptr(h.hash0)) // alg.hash

    // 计算 key 所在的 bucket 的 index
    m := uintptr(1)<<h.B - 1
    b := (*bmap)(add(h.buckets, (hash&m)*uintptr(t.bucketsize)))

    // 计算 tophash
    top := uint8(hash >> (sys.PtrSize*8 - 8))
    ...
    for {
        for i := uintptr(0); i < bucketCnt; i++ {
            // 检查 top 值
```

```
            if b.tophash[i] != top {
                continue
            }
            // 取 key 的地址
            k := add(unsafe.Pointer(b), dataOffset+i*uintptr(t.keysize))
            if alg.equal(key, k) { // alg.equal
                // 取 value 得地址
                v := add(unsafe.Pointer(b),
dataOffset+bucketCnt*uintptr(t.keysize)+i*uintptr(t.valuesize))
            }
        }
        ...
        if b == nil {
            // 返回零值
            return unsafe.Pointer(&zeroVal[0])
        }
    }
}
```

map 建议

- 如果知道 size，预先分配资源 `make(map[int]int, 1000)`
- uint32, uint64, string 作为键，非常快
- 清理 map：`for k:= range m { delete(m, k) }`
- key 和 value 中没有指针可以使 GC scanning 更快

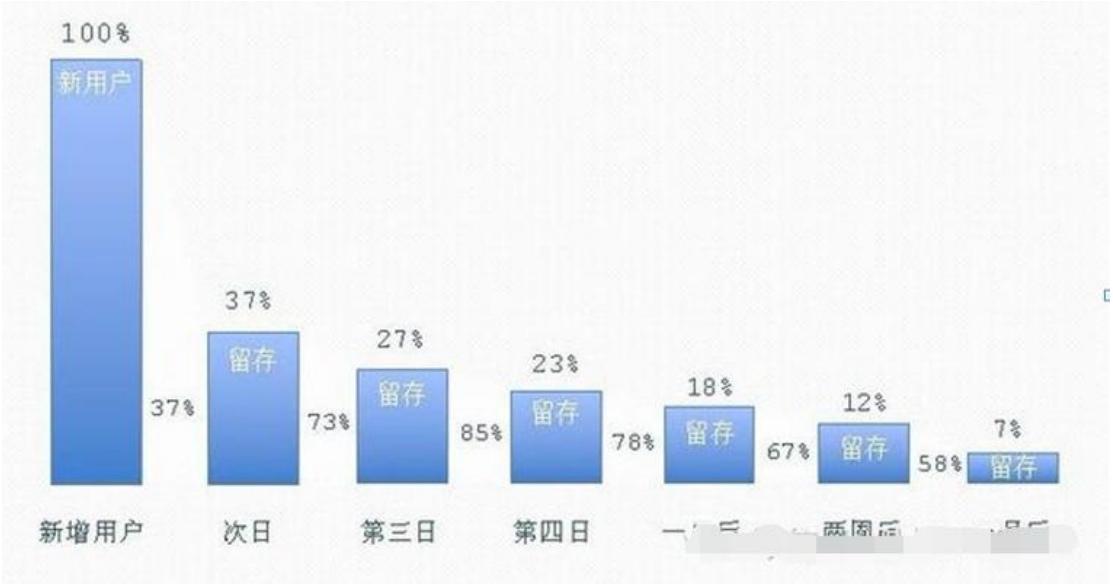
4、课后作业—难度：★ ☆ ☆ ☆ ☆

实现一个游戏并发安全 map：
<1> 使用标准库 sync 包。
<2> 实现并发安全 map 的写，读，delete、range 方法

5、如何提升用户留存

留存率的“40–20–10”规则

Facebook 平台流传出留存率“40–20–10”规则，规则中的数字表示的是次日留存率、第 7 日留存率和第 30 日留存率。规则所传达的信息如下：如果你想让游戏的 DAU 超过 100 万，那么新用户次日留存率应该大于 40%，7 天留存率和 30 天留存率分别大于 20%和 10%。



这里要注意的是，7 日留存率的概念，很多人容易误解为周留存率，在知乎上，也看到有人说，7 日留存率的计算，就是第 1 天注册，后推 1 周，也就是后推的 7 天，只要有 1 天登录，就算留存。这是一种就算方法，但不是上文“40–20–10”法则中的 7 日留存率。

第 7 日留存率和周留存率是不同的概念和计算方法。下面我们分类看看几种留存率的定义和计算公式。

新增用户留存率

谈用户留存率，必须先搞清楚新增用户，用户活跃。通常我们说的用户留存率，一般是指新增用户的留存率。

新增用户：通常指第一次使用该产品的用户。例如 QQ 的新用户，通常指当天注册的 QQ 帐号用户，这里的新，是专指这个 QQ 号，是新号；或许该号的主人并不是第一次用 QQ，但他今天新注册了一个 QQ 号，那么，我们就说这个 QQ 号是当天新增用户。

据传，现在每天仍有大量的新 QQ 号诞生，但绝大部分都只是新帐号而已，背后的自然人，大部分都是 QQ 的老用户。

一些没有帐号的产品怎么计算呢？一般是采用机器 ID 的识别，例如基于网卡 MAC 地址创造出来的一套新用户算法。

用户活跃

活跃用户：每个产品活跃的定义千差万别，如果是有帐号的客户端产品，例如 IM、端游等，通常以帐号登录作为活跃标识。

以 QQ 的活跃为例，腾讯 2013 年第 3 季度财报显示该季度月活跃账户数达到 8.156 亿，可以理解为月登录 QQ 用户数为 8.156 亿。

如果是某些工具软件，有的以启动作为活跃，例如看天气的。有些需要进行一些核心操作，例如拍照软件，至少是完成一张照片拍摄，才能算活跃吧。

有效用户

达到某一个指标（一般是在线时长）的日登录用户数量，例如登录 5 分钟以上的用户；或者，定义为完成某个核心操作的用户，例如 YY 语音产品，可以将有过语音交流的用户定义为有效活跃用户。这里的语音交流，又可以分为单向和双向，这就是不同程度的用户活跃。

如果是新增用户，可以衍生出有效新增的概念，就是新增和活跃的交集。例如当天新注册 100 个账号，其中 60 个登录超过 5 分钟，我们可以认为有效新增为 60。

以此类推，可以计算新增留存，有效新增留存。

按天留存率计算

就是指用户在首日新增后，在接下来的后推第 N 天活跃情况，用后推第 N 天活跃的用户除以首日新增用户，就得到后推第 N 天的新增用户留存率。

留存率计算案例

1 月 1 日，新增用户 200 人；

次日留存：第 2 天，1 月 2 日，这 200 人里面有 100 人活跃，则次日留存率为：

100 / 200 = 50% ；

2 日留存：第 3 天，1 月 3 日；这 200 名新增用户里面有 80 人活跃，

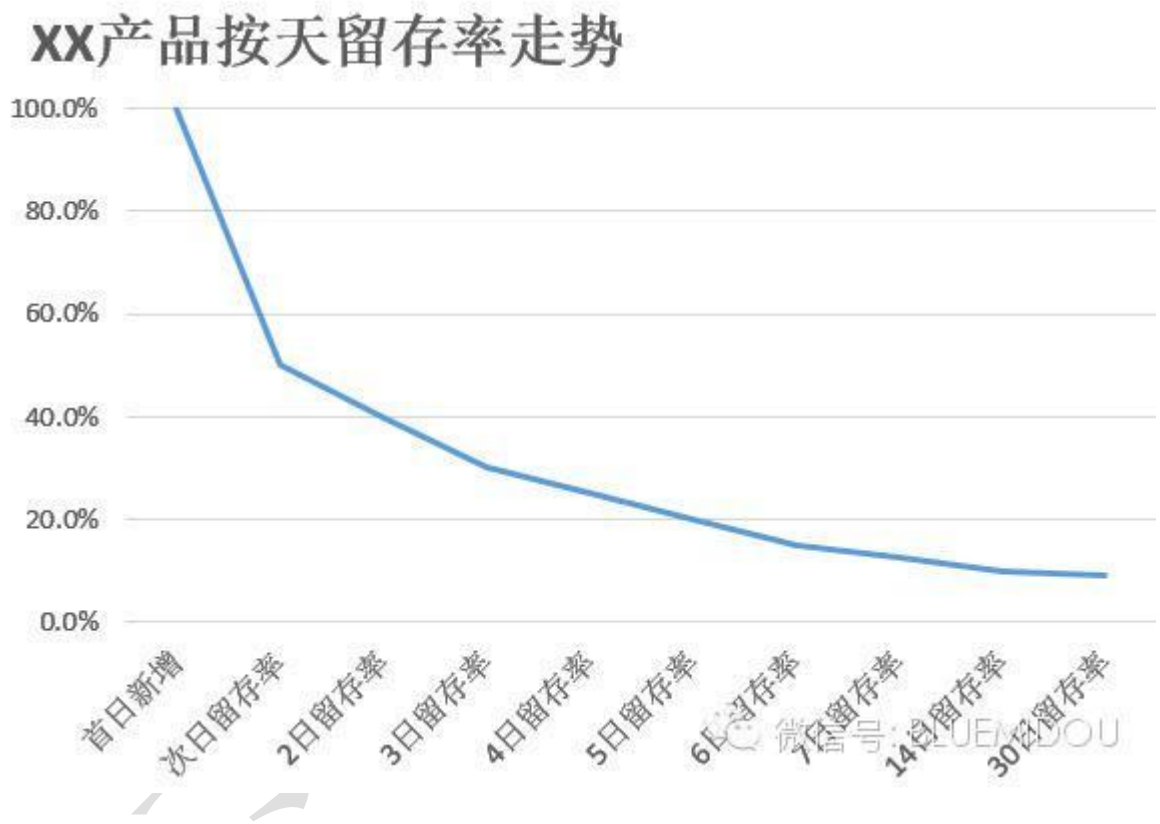
第 3 日新增留存率为: 80/200 = 40%；

7 日留存：第 8 天，1 月 8 日，这 200 名新增用户里面有 25 人活跃，

第 7 日新增留存率为: 25/200 = 12.5%；

整理为表格如下：

日期	1月1日	1月2日	1月3日	1月4日	1月5日	1月6日	1月7日	1月8日	1月15日	1月31日
用户数	200	100	80	60	50	40	30	25	20	18
第N日留存	首日新增	次日留存率	2日留存率	3日留存率	4日留存率	5日留存率	6日留存率	7日留存率	14日留存率	30日留存率
留存率	100.0%	50.0%	40.0%	30.0%	25.0%	20.0%	15.0%	12.5%	10.0%	9.0%



留存率反映的实际上是一种转化率，即由初期的不稳定的用户转化为活跃用户、稳定用户、忠诚用户的过程。

新增用户的第一天尤为重要，上文的 421 规则也显示多数产品的新增次日留存达到 40%已经不易，根据以往做产品的经验，又有 50%的用户在第一个 30 分钟就决定了他是否会继续使用产品，因此，产品经理们好好思考如何做好新用户在第一个小时，第一个 10 分钟的用户留存吧。

Talkingdata：APP 产品次日留存率

来自 Talkingdata.com.cn 网站的 APP 产品次日留存率数据，供大家参考。

6、微信公众平台及服务号

Golang 语言社区

Golang 技术社区