**【王牌游戏】后端工程师-测试题**

**姓名**：胡超

**目的：**通过文档化信息来进行快速交流和了解，最终提高面试质量和速度，节省双方时间，实现高效共赢

**简要说明：**王牌游戏对组织和人才的要求非常高，认为他是公司中除了战略方向外的第二大模块，所以公司CEO会亲自带队参与和推进，以保证未来人才和薪酬都能达到北京游戏行业的TOP3，同时也用文化来鼓励人才去推动公司的持续成长。王牌希望聚集一批领先的人才，让所有人在自己的优势模块上能持续追求极致，深度方面不断成长和提升，并且在其他同事的身上也获得广度的增值。（2021年Q4启动，因为公司能达到财务充裕）**公司简介：**2019年12月，王牌游戏创立。1年多后，王牌游戏就已经年利润过1亿。首个自研项目《圣光与荣耀》投入3000万研发费，仅海外市场就年利润过亿，生命周期预计3-6亿利润，目前已盈利3个亿。目前还有4个项目在研发中。CEO背景：03年加入游戏行业，并于2012年自投50万元，且作为制作人创立了上游网络，2013年，上游网络以13亿的估值并购给了掌趣科技（300315）。2017年对赌结束，因竞业原因，创建了泛阿网络，做阿拉伯的互联网行业，投资1个亿额，但最终以失败告终。CEO复盘认为是失败核心是2个问题：1）过于自信，不够自知和自省，对自己能力模型的颗粒度，分析得不够细致和深入，导致很多战略和战术方向的事情上没能扬长避短，最终决策错误；2）因为自知不足，所以不够重视组织和人才。在搭建团队的时候，也未找能补强自己短板的高管团队，并且自身并没投入到人才的识别、吸引和公司核心文化建设方面。2021年Q4后，当资金充裕后，王牌会着重在团队自省和组织发展方面投入公司重要的资源，希望打造北京TOP3的游戏人才团队。我们的使命：用心 **惊艳**玩家，让 **中国游戏**闪耀于 **世界舞台**我们的愿景：到2025年，成为北京**TOP3影响力**的游戏公司价值观2.0： **使命必达   追求极致**   **直面用户   开放思辨**  **终身学习   诚信尽责**

-------------------------------------------

此题不限时，不限撰写格式

**一、基础信息题**

**根据题目要求，编写相应的代码程序**

**1、****编写一个程序，在给定的二叉树中查找所有节点对(A, B)，这些节点对的路径长度相同，且路径中节点的值之和相同，但路径中没有共同的节点。**

RE：

package main

import (

"fmt"

)

// TreeNode 表示二叉树的节点

type TreeNode struct {

Val int

Left \*TreeNode

Right \*TreeNode

}

// PathInfo 存储路径信息

type PathInfo struct {

length int // 路径长度

sum int // 路径节点值之和

nodes []\*TreeNode // 路径上的节点

}

// DFS 遍历树并收集路径信息

func DFS(node \*TreeNode, length int, sum int, path []\*TreeNode, paths \*[]PathInfo) {

if node == nil {

return

}

// 将当前节点加入路径

path = append(path, node)

length++

sum += node.Val

// 添加当前路径信息到 paths

pathCopy := make([]\*TreeNode, len(path))

copy(pathCopy, path)

\*paths = append(\*paths, PathInfo{length, sum, pathCopy})

// 继续遍历左右子树

DFS(node.Left, length, sum, path, paths)

DFS(node.Right, length, sum, path, paths)

}

// hasCommonNode 检查两个路径是否有共同节点

func hasCommonNode(path1, path2 []\*TreeNode) bool {

nodeSet := make(map[\*TreeNode]struct{})

for \_, node := range path1 {

nodeSet[node] = struct{}{}

}

for \_, node := range path2 {

if \_, exists := nodeSet[node]; exists {

return true

}

}

return false

}

// FindAllPairs 查找所有符合条件的节点对

func FindAllPairs(root \*TreeNode) [][2]\*TreeNode {

var paths []PathInfo

DFS(root, 0, 0, []\*TreeNode{}, &paths)

var result [][2]\*TreeNode

n := len(paths)

// 查找路径长度相同且路径和相同的节点对

for i := 0; i < n; i++ {

for j := i + 1; j < n; j++ {

if paths[i].length == paths[j].length && paths[i].sum == paths[j].sum {

if !hasCommonNode(paths[i].nodes, paths[j].nodes) {

result = append(result, [2]\*TreeNode{paths[i].nodes[0], paths[j].nodes[0]})

}

}

}

}

return result

}

func main() {

// 构建一个示例二叉树

// 1

// / \

// 2 3

// /| |\

// 4 5 6 7

root := &TreeNode{Val: 1}

root.Left = &TreeNode{Val: 2}

root.Right = &TreeNode{Val: 3}

root.Left.Left = &TreeNode{Val: 4}

root.Left.Right = &TreeNode{Val: 5}

root.Right.Left = &TreeNode{Val: 6}

root.Right.Right = &TreeNode{Val: 7}

// 查找符合条件的节点对

pairs := FindAllPairs(root)

for \_, pair := range pairs {

fmt.Printf("节点对: (%d, %d)\n", pair[0].Val, pair[1].Val)

}

}

**2、程序运行时候启动2个线程，2个线程都会访问相同的一串数据，数据的数量和类型是不确定的，请编写一段程序，可以让2个线程在无锁的情况下，访问修改添加删除这段数据和数据里的任意数据类型。**

RE：

package main

import (

"fmt"

"sync"

"sync/atomic"

"time"

)

func main() {

var data sync.Map

var counter int32

data.Store("key1", "value1")

data.Store("key2", "value2")

// first goroutine 用于添加新数据

go func() {

for i := 0; i < 10; i++ {

key := fmt.Sprintf("key%d", atomic.AddInt32(&counter, 1))

value := fmt.Sprintf("value%d", i)

data.Store(key, value)

time.Sleep(100 \* time.Millisecond)

}

}()

// second goroutine 用于读取和打印数据

go func() {

for i := 0; i < 10; i++ {

key := fmt.Sprintf("key%d", i)

if value, ok := data.Load(key); ok {

fmt.Printf("Key: %s, Value: %v\n", key, value)

} else {

fmt.Printf("Key: %s not found\n", key)

}

time.Sleep(150 \* time.Millisecond)

}

}()

time.Sleep(2 \* time.Second) // 确保 goroutine 有时间运行

data.Range(func(key, value interface{}) bool {

fmt.Printf("Final Key: %v, Final Value: %v\n", key, value)

return true }) }

**3、设计一个名为”abTestHash"的哈希函数，用于AB测试的分组。该函数接受玩家的UUID（int64）和AB测试的ID（int32）作为参数，并返回分组概率（int32）。要求分组均匀且无耦合。**

RE：

// abTestHash 接收玩家的UUID和AB测试的ID，并返回一个表示分组的int32值（0为A组，1为B组）

func abTestHash(uuid int64, testID int32) int32 {

//使用哈希函数对UUID和AB测试ID的组合进行哈希

// 创建一个FNV哈希器

hasher := fnv.New32a()

// 将UUID和测试ID写入哈希器（先写入UUID，再写入测试ID，以确保每个玩家在不同的测试中具有不同的哈希值

combined := fmt.Sprintf("%d%d", uuid, testID)

hasher.Write([]byte(combined))

// 使用哈希值的最低位来决定分组（0或1）

hashValue := hasher.Sum32()

return int32(hashValue & 1) // 如果最低位是0，返回0（A组）；如果是1，返回1（B组）

}

func main() {

uuid := int64(123456789012345678)

testID := int32(4)

group := abTestHash(uuid, testID)

fmt.Printf("UUID %d; Test ID %d; group %d\n", uuid, testID, group)

}

**4、计算给定正整数数组的所有非空子集的最大异或和。例如，对于输入数组[1,2,3,4]，最大异或和为7，因为子集[3,4]的异或和为7。**

RE：

// getMaxXorSubset 计算所有非空子集的最大异或和

func getMaxXorSubset(arr []int) int {

// 创建一个长度为64的数组basis，用于存储每个位上的基

basis := make([]int, 64)

// 遍历数组中的每一个数

for \_, num := range arr {

// 检查num的每一位，从高到低

for i := 63; i >= 0; i-- {

// 如果num在第i位上没有1，继续检查下一位

if (num>>i)&1 == 0 {

continue

}

// 如果basis[i]是0，说明该位置还没有基向量，把num放入basis[i]

if basis[i] == 0 {

basis[i] = num

break

}

// 如果basis[i]已经存在，则将num与basis[i]异或，继续检查下一位

num ^= basis[i]

}

}

// 初始化maxXor为0，用于存储最大异或和

maxXor := 0

// 遍历basis中的每一个基向量

for i := 63; i >= 0; i-- {

// 尝试将basis[i]加入到maxXor中，如果能得到更大的值则更新maxXor

if (maxXor ^ basis[i]) > maxXor {

maxXor ^= basis[i]

}

}

return maxXor

}

func main() {

// 示例数组

arr := []int{1, 2, 3, 4}

// 计算并打印所有非空子集的最大异或和

fmt.Println("所有非空子集的最大异或和是:", getMaxXorSubset(arr)) } // 输出: 7

**5、设计一个Memory Pool，并实现malloc和free方法，要求Memory Pool中的Memory Block有object，index，size的管理。**

RE：

package main

import (

"errors"

"fmt"

"sync"

)

// MemoryBlock 表示内存池中的一个内存块

type MemoryBlock struct {

Object interface{} // 理论上可以存储任何对象，但这里不直接使用

Index int // 内存块在内存池中的索引

Size int // 内存块的大小（在这个简单示例中，所有块大小相同）

Used bool // 标记该内存块是否已被使用

}

// MemoryPool 表示内存池

type MemoryPool struct {

Blocks []\*MemoryBlock // 内存块数组

PoolSize int // 内存池的大小（即包含的块数）

BlockSize int // 每个内存块的大小（这里仅用于演示，实际不分配具体内存）

mutex sync.Mutex // 锁，用于并发控制

}

// NewMemoryPool 创建一个新的内存池

func NewMemoryPool(poolSize, blockSize int) \*MemoryPool {

blocks := make([]\*MemoryBlock, poolSize)

for i := 0; i < poolSize; i++ {

blocks[i] = &MemoryBlock{

Index: i,

Size: blockSize,

Used: false,

}

}

return &MemoryPool{

Blocks: blocks,

PoolSize: poolSize,

BlockSize: blockSize,

}

}

// Malloc 从内存池中分配一个内存块，简化版的，不实际分配内存，仅标记为已使用

func (mp \*MemoryPool) Malloc() (\*MemoryBlock, error) {

mp.mutex.Lock()

defer mp.mutex.Unlock()

for \_, block := range mp.Blocks {

if !block.Used {

block.Used = true

return block, nil

}

}

return nil, errors.New("no available memory block")

}

// Free 释放一个内存块

func (mp \*MemoryPool) Free(block \*MemoryBlock) error {

mp.mutex.Lock()

defer mp.mutex.Unlock()

if block.Index < 0 || block.Index >= mp.PoolSize {

return errors.New("invalid memory block index")

}

if mp.Blocks[block.Index] != block {

return errors.New("memory block does not belong to this pool")

}

block.Used = false

return nil

}

func main() {

// 创建一个包含10个块，每个块“大小”为10的内存池

mp := NewMemoryPool(10, 10)

// 分配内存块

block, err := mp.Malloc()

if err != nil {

fmt.Println("Error allocating memory:", err)

return

}

fmt.Printf("Allocated block: Index=%d, Size=%d, Used=%t\n", block.Index, block.Size, block.Used)

// 释放内存块

err = mp.Free(block)

if err != nil {

fmt.Println("Error freeing memory:", err)

return

}

fmt.Printf("Freed block: Index=%d, Size=%d, Used=%t\n", block.Index, block.Size, block.Used)}

**6、设计一个好友管理系统，用户可以在系统中添加好友、发送消息、查看好友动态等。要求系统记录用户之间的交互，并提供查询功能。**

RE：

用户表（users）

● user\_id (INT, 主键, 自动增长)

● username (VARCHAR, 用户名)

● email (VARCHAR, 邮箱，唯一)

● created\_at (DATETIME, 创建时间)

好友关系表（friendships）

● friendship\_id (INT, 主键, 自动增长)

● user\_id (INT, 外键, 引用users表的user\_id)

● friend\_id (INT, 外键, 引用users表的user\_id)

● status (ENUM('pending', 'accepted'), 默认'pending', 表示好友请求的状态)

● created\_at (DATETIME, 请求时间)

消息表（messages）

● message\_id (INT, 主键, 自动增长)

● sender\_id (INT, 外键, 引用users表的user\_id)

● receiver\_id (INT, 外键, 引用users表的user\_id)

● content (TEXT, 消息内容)

● sent\_at (DATETIME, 发送时间)

动态表（dynamics）

● dynamic\_id (INT, 主键, 自动增长)

● user\_id (INT, 外键, 引用users表的user\_id)

● content (TEXT, 动态内容)

● posted\_at (DATETIME, 发布时间)

**查询用户信息**

SELECT \* FROM users WHERE user\_id = ?;

**查询好友关系**

-- 查询用户ID为?的所有已接受的好友ID

SELECT friend\_id FROM friendships WHERE user\_id = ? AND status = 'accepted';

-- 查询双向好友关系（即A是B的好友且B也是A的好友）

SELECT DISTINCT a.user\_id AS user1, b.friend\_id AS user2

FROM friendships a

JOIN friendships b ON a.user\_id = b.friend\_id AND a.friend\_id = b.user\_id

WHERE a.status = 'accepted' AND b.status = 'accepted';

-- 查询用户ID为?的好友请求（包括已接受和待处理的）

SELECT \* FROM friendships WHERE (user\_id = ? AND status = 'pending') OR (friend\_id = ? AND status IN ('pending', 'accepted'));

**查询消息**

-- 查询用户ID为?发送的所有消息

SELECT \* FROM messages WHERE sender\_id = ?;

-- 查询用户ID为?接收的所有消息

SELECT \* FROM messages WHERE receiver\_id = ?;

-- 查询用户ID为?与好友ID为?之间的消息记录

SELECT \* FROM messages WHERE (sender\_id = ? AND receiver\_id = ?) OR (sender\_id = ? AND receiver\_id = ?);

**查询动态**

-- 查询用户ID为?发布的所有动态

SELECT \* FROM dynamics WHERE user\_id = ?;

-- 查询指定时间范围内的动态

SELECT \* FROM dynamics WHERE user\_id = ? AND posted\_at >= ?;

-- 查询所有用户的最新动态（假设每条动态都有一个唯一的ID，且ID越大表示越新）

SELECT \* FROM dynamics ORDER BY dynamic\_id DESC LIMIT 10

**二、MBTI职业性格测试题：**

**请做93题版本，<http://www.apesk.com/mbti/dati.asp> （测试结果可截图附上）**

请做93题标准通用高频使用版本，http://www.apesk.com/mbti/dati.asp

说明：MBTI即迈尔斯-布里格斯类型指标是全球领先的职业性格测试题，他能快速的判断人的底层思维方式和性格等，通过此工具相对准确的判断每个人更适合做哪种职业，这样能让人更好的发挥自身天赋。即如果是个敏捷型的天赋玩家，让他从事力量型的职业，则会非常浪费天赋，成长也事倍功半！

王牌一直贯彻用户第一原则，对于公司HR来说，所有的入职员工都是我们最重要的用户，我们希望努力给更多人创造适合自己成长的方向，知识和机会，让有能力的人都会越来越好，为中国游戏闪耀于世界舞台上贡献一份力量。

测试结果可截图附上（带性格比例第一个页面完整截图，包含该图下方相关说明）



王牌游戏期待您的加入，共同创造梦想^-^\*