questions

951. 翻转等价二叉树

```
func flipEquiv(root1 *TreeNode, root2 *TreeNode) bool {
    // dfs递归
    // 递归边界
    // 1.当两个节点为nil的时候返回true
    if root1 == nil && root2 == nil {
        return true
    }
    // 2.当两个节点不相等时返回false
    //   因为上面已经排除了root1,root2同时为nil的情况，所以这里是判断其中一个为nil时的情况，返回false
    if root1 == nil || root2 == nil {
        return false
    }
    //   两个节点val不相等时返回false
    if root1.Val != root2.Val {
        return false
    }
    // 递归体
    // 1.root1的左子树节点与root2的左子树节点比较，root1的右子树节点与root2的右子树节点比较，都相同则tmp1=true，否则为false
    tmp1 := flipEquiv(root1.Left, root2.Left) && flipEquiv(root1.Right, root2.Right)
    // 2.root1的左子树节点与root2的右子树节点比较，root1的右子树节点与root2的左子树节点比较，都相同则tmp2=true，否则为false
    tmp2 := flipEquiv(root1.Left, root2.Right) && flipEquiv(root1.Right, root2.Left)
    return tmp1 || tmp2
}
```

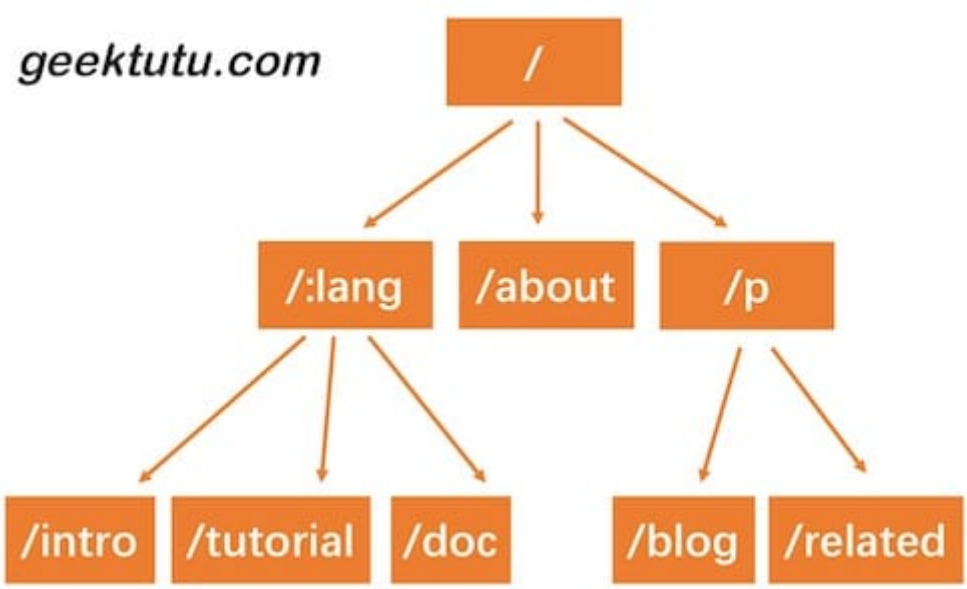
208. 实现 Trie (前缀树)

http动态路由请求匹配

实现动态路由最常用的数据结构，被称为前缀树(Trie树)。每一个节点的所有的子节点都拥有相同的前缀。这种结构非常适用于路由匹配，比如我们定义了如下路由规则：

- /:lang/doc
- /:lang/tutorial
- /:lang/intro
- /about
- /p/blog
- /p/related

用前缀树表示：



> HTTP请求的路径恰好是由`/`分隔的多段构成的，因此，每一段可以作为前缀树的一个节点。通过树结构查询，如果中间某一层的节点都不满足条件，那么就说明没有匹配到的路由，查询结束。

```
// Trie 前缀树结构体
type Trie struct {
    // 每个节点有26个子节点空间用于存储子结点
    children [26]*Trie
    // 判断当前节点是否为叶子结点
    isEnd bool
}

// Constructor 前缀树构造器,返回一个Trie实例
func Constructor() Trie {
    return Trie{}
}

// Insert 向前缀树插入数据
func (this *Trie) Insert(word string) {
    node := this
    for _, ch := range word {
        ch -= 'a'
        // 判断该节点是否在当前节点的子结点,不是则初始化该子节点
        if node.children[ch] == nil {
            node.children[ch] = &Trie{}
        }
        // 将当前节点指针指向该子节点
        node = node.children[ch]
    }
    node.isEnd = true
}

// Search 在前缀树中搜索数据
func (this *Trie) Search(word string) bool {
    node := this
    for _, ch := range word {
        ch -= 'a'
        // 将当前节点指针指向该子节点,直到word遍历完
        node = node.children[ch]
```

```
        // 若该子结点不存在则返回false
        if node == nil {
            return false
        }
    }
    // 若遍历得的最后一个节点是end节点，则返回true，否则返回false
    return node.isEnd
}

// StartsWith 在前缀树中进行前缀匹配
func (this *Trie) StartsWith(prefix string) bool {
    node := this
    for _, ch := range prefix {
        ch -= 'a'
        node = node.children[ch]
        if node == nil {
            return false
        }
    }
    // 与Search唯一不同的是不需要判断最后一个节点是否为end节点，直接返回true
    return true
}
```

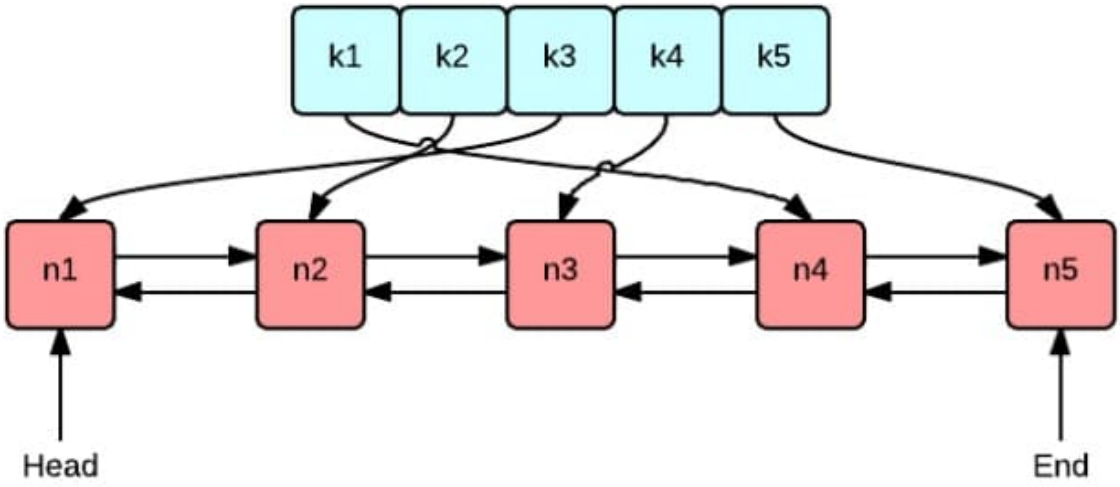
617. 合并二叉树

```
// dfs
func mergeTrees(root1 *TreeNode, root2 *TreeNode) *TreeNode {
    // 1.递归边界
    //   当root1节点为空时，返回root2,
    if root1 == nil {
        return root2
    }
    //   当root2节点为空时，返回root1
    if root2 == nil {
        return root1
    }
    //   当root1,root2节点都为空时,返回nil,上面以包含
    // 2.递归体
    //   当root1,2都不为空,则合并到root1
    root1.Val += root2.Val
    root1.Left = mergeTrees(root1.Left, root2.Left)
    root1.Right = mergeTrees(root1.Right, root2.Right)

    return root1
}
```

146. LRU 缓存

分布式存储系统缓存策略LRU



> 最近最少使用(LRU)，相对于仅考虑时间因素的 FIFO 和仅考虑访问频率的 LFU，LRU 算法可以认为是相对平衡的一种淘汰算法。LRU 认为，如果数据最近被访问过，那么将来被访问的概率也会更高。LRU 算法的实现非常简单，维护一个队列，如果某条记录被访问了，则移动到队尾，那么队首则是最近最少访问的数据，淘汰该条记录即可。

```
// 最近最少使用缓存
type LRUCache struct {
    size     int // 双向链表上的节点个数
    capacity int // 链表上节点的最大容量
    //ll       *list.List             // go语言标准库中的双向链表
    //cache    map[int]*list.Element // 值*list.Element是双向链表中对应节点的指针
    head, tail *DLinkedNode          //双向链表头节点和尾节点
    cache      map[int]*DLinkedNode //k-v map字典,v指向双向链表中的节点
}

// 自己实现的双向链表
type DLinkedNode struct {
    key, value int
    prev, next *DLinkedNode
}

// 初始化双向链表节点
func initDLinkedNode(key, value int) *DLinkedNode {
    return &DLinkedNode{
        key:   key,
        value: value,
    }
}

// 构建LRU缓存
func Constructor(capacity int) LRUCache {
    lc := LRUCache{ // 初始化LRC缓存
        capacity: capacity,
        head:     initDLinkedNode(0, 0),
        tail:     initDLinkedNode(0, 0),
        cache:    map[int]*DLinkedNode{},
    }
    // 连接头尾
    lc.head.next = lc.tail
    lc.tail.prev = lc.head
```

```
    return lc
}

// Get 获取LRU缓存中key-value值
func (this *LRUCache) Get(key int) int {
    // 查看缓存中是否有key对应的值
    if _, ok := this.cache[key]; !ok {
        return -1
    }
    // 存在key对应的值,获取该节点
    node := this.cache[key]
    // 将此节点移动到双链表头部（双链表没有明确的头尾，这里只是象征意义上的）
    this.MoveToFront(node)
    // 并返回该key对应的value值
    return node.value
}

// Put 存放node节点到双链表中
func (this *LRUCache) Put(key int, value int) {
    // 判断该链表中是否有该节点
    if _, ok := this.cache[key]; !ok {
        // 不存在，初始化一个双向链表节点，并在map上建立链接
        node := initDLinkedNode(key, value)
        this.cache[key] = node
        // 将node加至链表头，size++
        this.AddToFront(node)
        this.size++
        // 判断是否超过该cache的容量
        if this.size > this.capacity {
            removed := this.ReMoveTail()
            delete(this.cache, removed.key)
            this.size--
        }
    } else {
        // 存在key,将key对应的value重新赋值，并移动至链头
        node := this.cache[key]
        node.value = value
        this.MoveToFront(node)
    }
}

// MoveToFront 移动双链表中的节点到链表头,分为两步：先删除该节点，再添加该节点到链头
func (this *LRUCache) MoveToFront(node *DLinkedNode) {
    this.ReMoveNode(node)
    this.AddToFront(node)
}

// ReMoveTail 移除尾节点，分为两步：获取尾节点，移除节点
func (this *LRUCache) ReMoveTail() *DLinkedNode {
    node := this.tail.prev
    this.ReMoveNode(node)
    return node
}

// AddToFront head之后添加头节点
```

```
func (this *LRUCache) AddToFront(node *DLinkedNode) {
    node.prev = this.head
    node.next = this.head.next
    this.head.next.prev = node
    this.head.next = node
}

// ReMoveNode 删除某一节点
func (this *LRUCache) ReMoveNode(node *DLinkedNode) {
    node.prev.next = node.next
    node.next.prev = node.prev
}
```

29. 两数相除

```
func divide(dividend int, divisor int) int {
    // 边界处理
    if dividend == 0 {
        return 0
    }
    if divisor == 1 {
        return dividend
    }
    if divisor == -1 {
        if dividend > math.MinInt32 {
            return -dividend
        }
        return math.MaxInt32
    }
    // -+判断
    var a = dividend
    var b = divisor
    var sign = 1
    if (a > 0 && b < 0) || (a < 0 && b > 0) {
        sign = -1
    }
    if a < 0 {
        a = -a
    }
    if b < 0 {
        b = -b
    }

    //div
    res := div(a, b)
    if sign == 1 {
        return res
    }
    return -res
}

func div(a int, b int) int {
    // 递归边界
```

```
    if a < b {
        return 0
    }
    var count int = 1
    var tb = b
    // 当a至少是b的两倍时
    for (tb + tb) <= a {
        count = count + count
        tb = tb + tb
    }
    // 递归体 用余数进行递归
    return count + div(a-tb, b)
}
```

196. 删除重复的电子邮箱

```
-- 从p1中删除p2里满足where条件的数据
delete p1 from Person p1,Person p2
where p1.Email = p2.Email and p1.Id > p2.Id
```

184. 部门工资最高的员工

```
-- 先从Employee表中按照departmentId分组，并且选出每组的最高工资
-- 使用默认的join(inner join)连接两个表，筛选出符合where的记录
select Department.name as 'Department', Employee.name as 'Employee', salary
from Employee join Department on Department.id = Employee.id
where (Employee.departmentId, salary) in (
    select departmentId, max(salary)
    from Employee
    group by departmentId
)
```

607. 销售员

```
-- 以orders表进行左连接，筛选出连接表 公司名为RED的销售id
-- 利用not in进行排除
select s.name
from salesperson s
where s.sales_id not in (
    select orders.sales_id
    from orders left join company on orders.com_id = company.com_id
    where company.name='RED'
)
```