两数之和

```
func twoSum(nums []int, target int) []int {
    n := len(nums)
    //哈希表，key 是数本身，value 是下标
    hashTable := make(map[int]int, 0)
    for i := 0; i < n; i++ {
        hashTable[nums[i]] = i
    }
    for i := 0; i < n; i++ {
        if value , ok := hashTable[target-nums[i]]; ok {
            if value != i {
                return []int{i, value}
            }
        }
    }
    return []int{}
}
```

15. 三数之和

```
func threeSum(nums []int) [][]int {
    sort.Ints(nums)
    result := make([][]int, 0)
    n := len(nums)
    hashMap := make(map[int]int, 0)
    for i := 0; i < n; i++ {
        hashMap[nums[i]] = i
    }
    for i := 0; i < n; i++ {
        if i != 0 && nums[i] == nums[i-1] {continue} //避免 a 重复，1 1 3 ...
        for j := i+1; j < n; j++ {
            if j != i+1 && nums[j] == nums[j-1] {continue} //避免 b 重复 1 2 2
            target := -1*(nums[i]+nums[j])
            if _,ok := hashMap[target]; !ok {
                continue
            }
            k := hashMap[target]
            if k > j {
                resultItem := make([]int, 0)
                resultItem = append(resultItem, nums[i], nums[j], nums[k])
                result = append(result, resultItem)
            }
        }
    }
    return result
}
```

160. 相交链表

```
func getIntersectionNode(headA, headB *ListNode) *ListNode {
    //go 里没有 hashset，用 map 代替
    hashTable := make(map[*ListNode]bool)
    p := headA
    for p != nil {
        hashTable[p] = true
        p = p.Next
    }
```

```
    p = headB
    for p != nil {
        if hashTable[p] {
            return p
        }
        p = p.Next
    }
    return nil
}
```

141. 环形链表

```
func hasCycle(head *ListNode) bool {
    //go 里没有 hashset, 用 map 代替
    hashTable := make(map[*ListNode]bool)
    p := head
    for p != nil {
        if hashTable[p] {
            return true
        } else {
            hashTable[p] = true
        }
        p = p.Next
    }
    return false
}
```

面试题 02.01. 移除重复节点

```
func removeDuplicateNodes(head *ListNode) *ListNode {
    if head == nil {return head}
    set := make(map[int]bool)
    newHead := &ListNode{}
    tail := newHead
    p := head
    for p != nil {
        tmp := p.Next
        if !set[p.Val] {
            set[p.Val] = true
            tail.Next = p
            tail = p
            tail.Next = nil
        }
        p = tmp
    }
    return newHead.Next
}
```

面试题 16.02. 单词频率

```
type WordsFrequency struct {
    wordMap  map[string]int
}

func Constructor(book []string) WordsFrequency {
    wordMap := make(map[string]int, 0)
    for _, word := range book {
        count := 1
        if c , ok := wordMap[word]; ok {
```

```
            count += c
        }
        wordMap[word] = count
    }
    return  WordsFrequency{wordMap:wordMap}
}

func (this *WordsFrequency) Get(word string) int {
    if _, ok := this.wordMap[word]; !ok {
        return 0
    }
    return this.wordMap[word]
}
```

面试题 01.02. 判定是否互为字符重排

```
func CheckPermutation(s1 string, s2 string) bool {
    s1ht := make(map[byte]int, 0)
    for i := 0; i < len(s1); i++ {
        c := s1[i]
        count := 1
        if c2, ok := s1ht[c]; ok {
            count += c2
        }
        s1ht[c] = count
    }
    //s2 去跟 s1 匹配
    for i := 0; i < len(s2); i++ {
        c := s2[i]
        if _, ok := s1ht[c]; !ok {
            return false
        }
        count := s1ht[c]
        if count == 0 {return false}
        s1ht[c] = count-1
    }
    //检查 s1ht 是否为空
    for i := 0; i < len(s1); i++ {
        c := s1[i]
        if s1ht[c] != 0 {return false}
    }
    return true
}
```

242. 有效的字母异位词

```
func isAnagram(s string, t string) bool {
    if len(s) != len(t) {
        return false
    }
    nums1 := make([]int, 26)
    for i := 0; i < len(s); i++ {
        c := s[i]
        nums1[c-'a']++
    }
    nums2 := make([]int, 26)
    for i := 0; i < len(t); i++ {
        c := t[i]
        nums2[c-'a']++
```

```
    }
    for i := 0; i < 26; i++ {
        if nums1[i] != nums2[i] {
            return false
        }
    }
    return true
}
```

49. 字母异位词分组

```
func groupAnagrams(strs []string) [][]string {
    groupMap := make(map[string][]string, 0)
    for _, str := range strs {
        array := []byte(str)
        sort.Slice(array, func(i, j int) bool {
            return array[i] < array[j]
        })
        key := string(array)
        list := make([]string, 0)
        if _, ok := groupMap[key]; ok {
            list = groupMap[key]
        }
        list = append(list, str)
        groupMap[key] = list
    }
    //没有现成的取 map 里 values 的方法，这里一个一个获取
    result := make([][]string, 0)
    for _, value := range groupMap {
        result = append(result, value)
    }
    return result
}
```

剑指 Offer 03. 数组中重复的数字

```
func findRepeatNumber(nums []int) int {
    //go 标准库没有 set，用 map 实现
    set := make(map[int]bool, 0)
    for i := 0; i < len(nums); i++ {
        if set[nums[i]] {return nums[i]}
        set[nums[i]] = true
    }
    return -1
}
```

136. 只出现一次的数字

```
func singleNumber(nums []int) int {
    hashTable := make(map[int]int, 0)
    for i := 0; i < len(nums); i++ {
        count := 1
        if c, ok := hashTable[nums[i]]; ok {
            count += c
        }
        hashTable[nums[i]] = count
    }
    for i := 0; i < len(nums); i++ {
```

```
        count := hashTable[nums[i]]
        if count == 1 {return nums[i]}
    }
    return -1
}
```

349. 两个数组的交集

```
func intersection(nums1 []int, nums2 []int) []int {
    hashTable := make(map[int]bool, 0)
    for i := 0; i < len(nums1); i++ {
        hashTable[nums1[i]] = true
    }
    result := make([]int, 0)
    for i := 0; i < len(nums2); i++ {
        if hashTable[nums2[i]] {
            //delete 或者 hashTable[nums2[i]]=false
            delete(hashTable, nums2[i])
            result = append(result, nums2[i])
        }
    }
    return result
}
```

1122. 数组的相对排序

```
func relativeSortArray(arr1 []int, arr2 []int) []int {
    //arr2 中每个数字在 arr1 中出现的次数
    counts := make(map[int]int, 0)
    //先用 arr2 构建 hash 表
    for i := 0; i < len(arr2); i++ {
        counts[arr2[i]] = 0
    }
    //扫描 arr1 统计 arr2 中每个数字在 arr1 中出现的次数
    for i := 0; i < len(arr1); i++ {
        if oldCount, ok := counts[arr1[i]]; ok {
            counts[arr1[i]] = oldCount+1
        }
    }
    result := make([]int, len(arr1))
    k := 0
    //将 counts 的数据按照 arr2 的顺序输出
    for i := 0; i < len(arr2); i++ {
        count := counts[arr2[i]]
        for j := 0; j < count; j++ {
            result[k+j] = arr2[i]
        }
        k += count
    }
    //将 arr1 中未出现在 arr2 中的数据有序输出到 result
    sort.Ints(arr1)
    for i := 0; i < len(arr1); i++ {
        if _, ok := counts[arr1[i]]; !ok {
            result[k] = arr1[i]
            k++
        }
    }
```

```
    return result
}
```

面试题 16.21. 交换和

```
func findSwapValues(array1 []int, array2 []int) []int {
    n := len(array1)
    m := len(array2)
    //计算数组 1 的和
    sum1 := 0
    for i := 0; i < n; i++ {
        sum1 += array1[i]
    }
    //计算数组 2 的和，并且将元素放到哈希表中
    sum2 := 0
    hashTable := make(map[int]bool, 0)
    for i := 0; i < m; i++ {
        sum2 += array2[i]
        hashTable[array2[i]] = true
    }
    //sum1+sum2 是奇数，无解
    sum := sum1+sum2
    if sum % 2 == 1 {return []int{}}
    //遍历数组 1 中的每个元素，在哈希表中查找
    diff := sum/2 - sum1
    for i := 0; i < n; i++ {
        target := array1[i] + diff
        if hashTable[target] {
            return []int{array1[i], target}
        }
    }
    return []int{}
}
```

706. 设计哈希映射

```
type Pair struct {
    key   int
    value int
}
type MyHashMap struct {
    slots  [][]Pair
}

const SLOTS_COUNT = 3535

func Constructor() MyHashMap {
    return MyHashMap{slots: make([][]Pair, SLOTS_COUNT),}
}

func (this *MyHashMap) hash(key int) int{
    return key % SLOTS_COUNT
}

func (this *MyHashMap) Put(key int, value int)  {
    slot := this.slots[this.hash(key)]
    if slot == nil {
        this.slots[this.hash(key)] = make([]Pair, 0)
```

```
        slot = this.slots[this.hash(key)]
    }
    for i := 0; i < len(slot); i++ {
        pair := slot[i]
        if pair.key == key {
            slot = append(slot[:i], slot[i+1:]...) //remove i
            break
        }
    }
    slot = append(slot, Pair{key, value})
    //append 会返回新的 slice，因此这里还需要再放回去
    this.slots[this.hash(key)] = slot
}

func (this *MyHashMap) Get(key int) int {
    slot := this.slots[this.hash(key)]
    if slot == nil {
        return -1
    }
    for i := 0; i < len(slot); i++ {
        pair := slot[i]
        if pair.key == key {
            return pair.value
        }
    }
    return -1
}

func (this *MyHashMap) Remove(key int)  {
    slot := this.slots[this.hash(key)]
    if slot == nil {
        return
    }
    for i := 0; i < len(slot); i++ {
        pair := slot[i]
        if pair.key == key {
            slot = append(slot[:i], slot[i+1:]...) //remove i
            break
        }
    }
    this.slots[this.hash(key)] = slot
}
```

146. LRU 缓存机制

```
type DLinkedNode struct {
    key   int
    value int
    prev  *DLinkedNode
    next  *DLinkedNode
}

type LRUCache struct {
    hashtable map[int]*DLinkedNode
    size      int
    capacity  int
    head      *DLinkedNode
    tail      *DLinkedNode
}
```

```go
func Constructor(capacity int) LRUCache {
    lruCache := LRUCache{
        size:      0,
        capacity:  capacity,
        hashtable: make(map[int]*DLinkedNode, 0),
        head: &DLinkedNode{key: -1, value: -1},
        tail: &DLinkedNode{key: -1, value: -1},
    }
    lruCache.head.prev = nil
    lruCache.head.next = lruCache.tail
    lruCache.tail.prev = lruCache.head
    lruCache.tail.next = nil
    return lruCache
}

func (this *LRUCache) Get(key int) int {
    if this.size == 0 {
        return -1
    }
    node := this.hashtable[key]
    if node == nil {
        return -1
    }
    this.removeNode(node)
    this.addNodeAtHead(node)
    return node.value
}

func (this *LRUCache) Remove(key int) {
    node := this.hashtable[key]
    if node != nil {
        this.removeNode(node)
        delete(this.hashtable, key)
        this.size--
        return
    }
}

func (this *LRUCache) removeNode(node *DLinkedNode) {
    node.next.prev = node.prev
    node.prev.next = node.next
}

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

func (this *LRUCache) Put(key int, value int)  {
    node := this.hashtable[key]
    if node != nil {
        node.value = value
        this.removeNode(node)
        this.addNodeAtHead(node)
        return
    }
    if this.size == this.capacity {
        delete(this.hashtable, this.tail.prev.key)
```

```
        this.removeNode(this.tail.prev)
        this.size--
    }
    newNode := &DLinkedNode{key:key, value:value}
    this.addNodeAtHead(newNode)
    this.hashtable[key] = newNode
    this.size++
}
```