麻麻再也不用担心你的递归了

今日目标：

1：能说出递归的概念和特点

2：能说出递归和循环的异同

3：能写出递归的模板代码

4：完成递归实战题目

5：能说出在递归过程中会出现的问及解决方案

1、递归概述

递归(Recursion)是一种非常广泛的算法，与其说递归是一种算法不如说递归是一种编程技巧。在后续数据结构和算法的编码实现过程中我们都要用到递归，比如DFS深度优先搜索，前中后序二叉树的遍历等都需要用到递归的知识，因此搞懂递归非常的重要，否则后续的学习会非常的吃力。

1.1、递归概念

- 例子1：

> 1：从前有个山
>
> 2：山里有个庙
>
> 3：庙里有个和尚讲故事
>
> 4：回到1

例子2：

> 天下有奇族人姓计，长生不老。一日其孙问其父：吾之18代祖名何?
> 其父不明，父问其父
> 其父不明，父问其父
> 其父不明，父问其父
> 其父不明，父问其父
> …
> 晌后，其18代祖回其子：你猜
> 然其回其子：你猜
> 然其回其子：你猜
> 然其回其子：你猜
> 然其回其子：你猜
> ……
> 终，计姓末代孙知其18代祖名“你猜”

例3：

假设你在一个电影院，你想知道自己坐在哪一排，但是前面人很多，你懒得去数了，于是你问前一排的人"你坐在哪一排?"，这样前面的人 (代号 A) 回答你以后，你就知道自己在哪一排了,你只要把 A 的答案加一，就是自己所在的排了。不料 A 比你还懒，他也不想数，于是他也问他前面的人 B “你坐在哪一排?"，这样 A 可以用和你一模一样的步骤知道自己所在的排。然后 B 也如法炮制。直到他们这一串人问到了最前面的一排，第一排的人告诉问问题的人"我在第一排"。最后大家就都知道自己在哪一排了。

例4：一部电影《盗梦空间》

故事主线情节是从飞机开始，去到一层一层的梦境，最后在一个雪山的屋子里，每一层梦境都互不干扰，主角可以在每一层梦境中做一些事情，当然主角要回到现实世界也需要一层一层的返回；主角每进入到下一层会发生一些改变，回来的时候也会把这些变化带回来。

综上：

递归在维基百科的官方解释为：**递归(Recursion),又名递回。**

在数学与计算机科学中，是指**在函数的定义中使用函数自身的方法。**

英文Recursion也就是重复发生，再次重现的意思。 而对应的中文翻译 ”递归“ 却表达了两个意思：”递“ + ”归“。 这两个意思，正是递归思想的精华所在，去的过程叫做递，回来的过程叫做归。

递归视频

在编程语言中对递归可以简单理解为：方法自己调用自己，只不过每次调用时参数不同而已。

```
int f(int n) {
    if (n == 1){
        return 1;
    }
    return f(n-1) + 1;
}
```

1.2、递归特点

与其说是递归的特点，不如说什么样的问题适合用递归来解决，即满足递归的条件

1、可拆解成可重复的子问题（重复子问题）

如果一个问题的解能够拆分成多个子问题的解，拆分之后，子问题和该问题在求解上除了数据规模不一样之外，求解的思路和该问题的求解思路完全相同，也就是说能够找到一种规律，那么这个问题就可以使用递归来求解。

比如电影院中你要知道“自己在哪一排”的问题，可以分解为“前一排的人在哪一排”这样一个子问题，你求解“自己在哪一排”的思路，和前面一排人求解“自己在哪一排”的思路，是一模一样的。

这种最近重复子问题的求解思路会形成一种规律，如果将这个规律用数学公式表达出来就是我们所谓的**递推公式**，

基本上所有的递归问题都可以使用递推公式来表示，比如说对于刚刚电影院的例子，我们可以使用递推公式表示出来是这个样子的：

```
f(n) = f(n-1)+1
```

2、递归要有出口（终止条件）

把一个问题的解分解为多个子问题的解，把子问题再分解为子子问题，一层一层分解下去，不能存在无限递归，这就需要有终止条件。就比如电影院的例子，第一排的人不需要再继续询问任何人，就知道自己在哪一排，也就是 f(1)=1，这就是递归的终止条件，所以对于电影院的例子，完整的递推公式应该是：

```
f(n) = f(n-1)+1, **其中f(1)=1**
```

综上所述：写递归代码的关键就是找到如何将一个问题拆分成多个小问题的规律，并且基于此写出递推公式，然后再找到递归终止条件，最后将递推公式和终止条件翻译成代码即可

1.3、递归和循环

从本质上来说：递归类似于循环，只不过是通过函数体调用自己来进行所谓的循环。

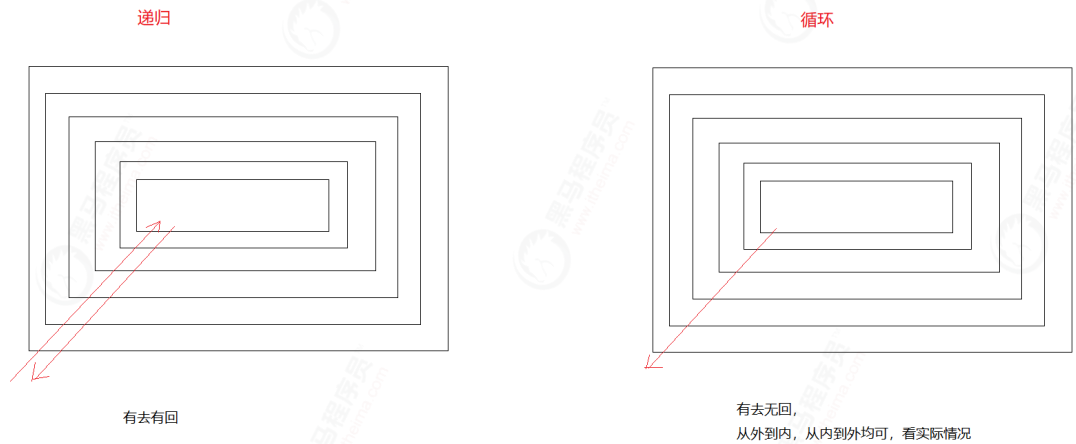
> 计算机语言在刚开始的时候本质上是汇编，它没有所谓的循环嵌套，它更多的是之前有一个函数或者指令在什么地方，程序不断的跳到这个地方去执行，这就是所谓的递归。
>
> 对于循环本身和递归有异曲同工之妙，这点可以从循环的汇编代码结构上可以看出来，因此递归和循环没有明显的边界。

但是从编程角度上来说递归和循环还是略有区别，递归是有去有回，循环是有去无回。

举个例子，给你一把钥匙，你站在门前面，问你用这把钥匙能打开几扇门。

递归：你打开面前这扇门，看到屋里面还有一扇门（这门可能跟前面打开的门一样大小，也可能门小了些），你走过去，发现手中的钥匙还可以打开它，你推开门，发现里面还有一扇门，你继续打开，。。。，若干次之后，你打开面前一扇门，发现只有一间屋子，没有门了。你开始原路返回，每走回一间屋子，你数一次，走到入口的时候，你可以回答出你到底用这钥匙开了几扇门。

循环：你打开面前这扇门，看到屋里面还有一扇门，（这门可能跟前面打开的门一样大小，也可能门小了些），你走过去，发现手中的钥匙还可以打开它，你推开门，发现里面还有一扇门，（前面门如果一样，这门也是一样，第二扇门如果相比第一扇门变小了，这扇门也比第二扇门变小了，你继续打开这扇门，。。。，一直这样走下去。入口处的人始终等不到你回去告诉他答案。



但同时递归和循环也是非常像的，**循环可以改写成递归，递归未必能改写成循环**，

有时候我们可以**使用迭代循环的方式将递归代码改写为非递归代码。**

2、递归的应用

2.1、代码模板

对于递归代码的编写，很多人很容易看懂，但是很难写出来，在此给出递归代码的参考模板

```
//递归代码参考模板
public void recur(int level,int param) { //参数随着实际的应用场景有所不同

    //part1：先编写递归终止条件（没有终止条件的递归就是无限递归，类似死循环）
    if (level > MAX_LEVEL) {
        //处理结果数据
        .......
        return;
    }

    //part2：处理当前层逻辑（有时需要先处理当前层逻辑再递归到下一层,有时候需要递归到下一层根据返回的结果处理当前层,所以需要灵活运用）
    process(level,param);
    .....

    //part3：进入到下一层(即开始递归)
    recur(level+1,param);

    //part4：当前层资源清理,比如全局的参数等
    ......

}
```

2.2、实战题目

2.2.1、n的阶乘

计算n的阶乘，n!= n* (n-1) * (n-2) * (n-3) *..... * 3 * 2 * 1;

递推公式：`f(n) = n * f(n-1),其中n>0且f(1)=1`

创建：`com.itheima.recur.NFactorial`，编写方法`public int fac(int n){}`

```
package com.itheima.recur;

import org.junit.jupiter.api.Test;

/**
 * Created by 传智播客*黑马程序员.
 */
public class NFactorial {


    /**
```

```
     * 计算n的阶乘
     * @param n
     * @return
     */
    public int fac(int n){
        //终止条件
        if ( n <= 1) {
            return 1;
        }

        return n* fac(n-1);
    }

    //递归改写成循环
    public int fac_loop(int n){
        int res = 1;

        for (int i=2;i<=n;i++) {
            res = res * i;
        }

        return res;
    }

    @Test
    public void test(){
        System.out.println(fac(6)+"----"+fac_loop(6));
    }
}

```

> 递归写法：从n一直下探到下一层最后走到1，然后依次返回
>
> 循环写法：从1开始按照规律(递推公式)循环到n，每次循环都进行计算
>
> 问题优化：如果输入的n过大，会导致计算结果超出int的存储范围，怎么办?

2.2.2、剑指 Offer 10- I. 斐波那契数列

字节跳动，亚马逊最近面试题，剑指 Offer 10- I. 斐波那契数列

1，题设条件已给出递推公式，按照递归代码的模板编写

```java
class Solution {
    public int fib(int n) {
        //终止条件
        if (n < 2 ) {
            return n ;
        }else {
            int x = fib(n-1) % 1000000007;
            int y = fib(n-2) % 1000000007;
            return  (x+y) %  1000000007;
        }
    }
}
```

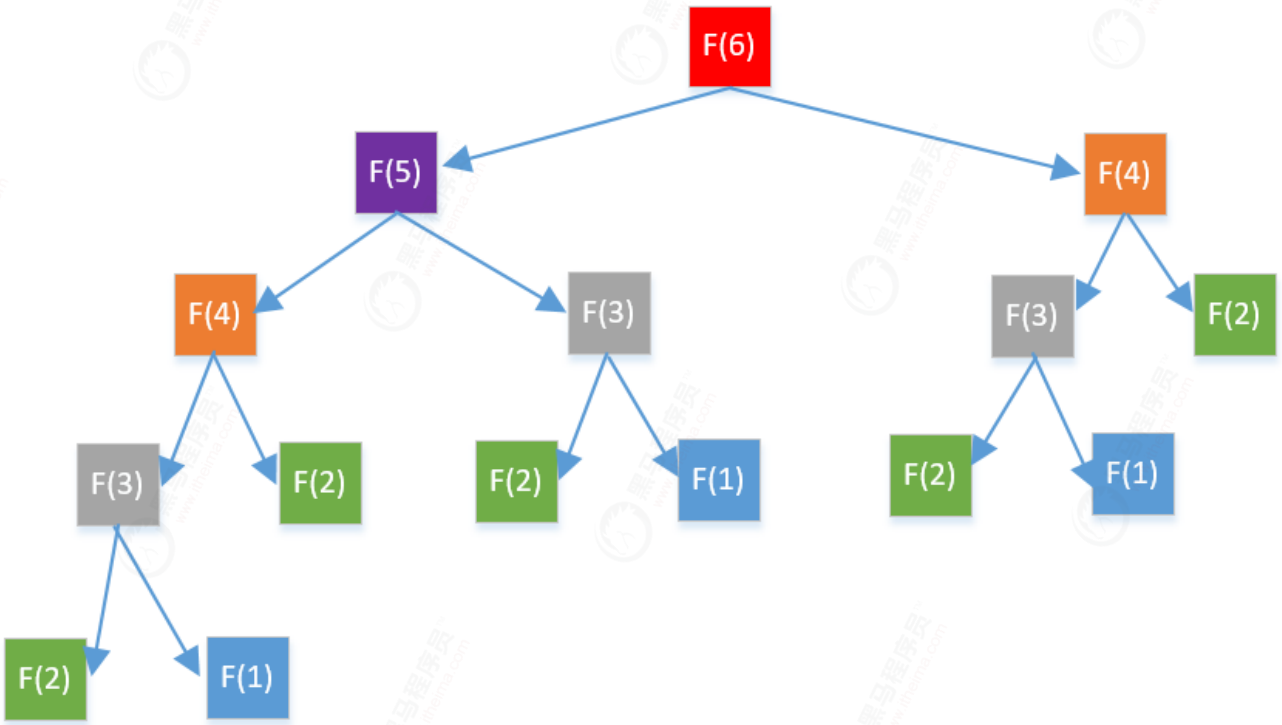
简单的测试 用例能通过 ， 但是有一些大的测试用力没通过，最终结果未AC，超时

原因是什么呢?

2.2.3、递归中出现的问题

1、重复计算

对于刚刚的斐波那契数列，我们可以简要的画一下递归整个过程的相关状态节点，以fib(6)为例



这个过程我们把它叫做**画递归的状态树（把递归过程的中间状态画出来）**，从这个图中我们可以发现，相同颜色的都是重复计算的数，当题目参数n越大，重复的越多，程序执行耗时更长！

> 这种**傻递归**的方式，**时间复杂度是O(2^n)**
>
> 递归代码的时间复杂度要根据具体的语境来判断，在能画出递归状态树的情况下比较好判断

如何优化?

使用缓存（哈希）将中间结果缓存起来，避免多余的重复计算

写法1：使用全局参数

> 就像你在不同梦境中穿越时，总有一双俯视全局的眼镜能看到你在干嘛

```
class Solution {

    //要缓存每一层的中间计算状态,哈希定义为全局的
    Map<Integer,Integer> hash = new HashMap();

    public int fib(int n) {
        //终止条件
        if (n < 2 ) {
            hash.put(n,n);
            return n ;
        }else {
            if (hash.containsKey(n)) {
                return hash.get(n);
            }
            int x = fib(n-1) % 1000000007;

            hash.put(n-1,x);
```

```
            int y = fib(n-2) % 1000000007;
            hash.put(n-2,y);
            int z = (x+y) % 1000000007;
            hash.put(n,z);
            return  z;
        }
    }
}
```

写法2：缓存对象随着递归依次进入到下一层，然后返回的时候可以得到在下一层的改变（类似盗梦空间主角进入下一层时可以携带一些东西，这些东西在下一层发生了改变，返回的时候可以将这种改变带回来），体现在编码上就是函数的参数

```
class Solution {

    public int fib(int n) {
        return fib(n,new HashMap());
    }

    public int fib(int n,Map<Integer,Integer> hash){
        //终止条件
        if (n < 2 ) {
            hash.put(n,n);
            return n ;
        }else {
            if (hash.containsKey(n)) {
                return hash.get(n);
            }
            int x = fib(n-1,hash) % 1000000007;
            hash.put(n-1,x);
            int y = fib(n-2,hash) % 1000000007;
            hash.put(n-2,y);
            int z = (x+y) % 1000000007;
            hash.put(n,z);
            return  z;
        }
    }
}
```

> 这种递归方式我们成为**记忆化递归，时间复杂度O(n)，空间复杂度O(n)**

非递归优化版：

参考`剑指 Offer 10- I. 斐波那契数列.png`

```
class Solution {
    public int fib(int n) {

        if ( n < 2) {

            return n;
```

```
        }
        //定义两个数
        int first = 0;
        int second = 1;

        int temp;
        for (int i = 2;i<=n;i++) {
            temp = second;
            second = (second + first) % 1000000007;
            first = temp;
        }
        return second;
    }
}
```

> **时间复杂度：O(N),空间复杂度O(1)**
>
> 这种解法也是后续要讲解的动态规划的算法思想

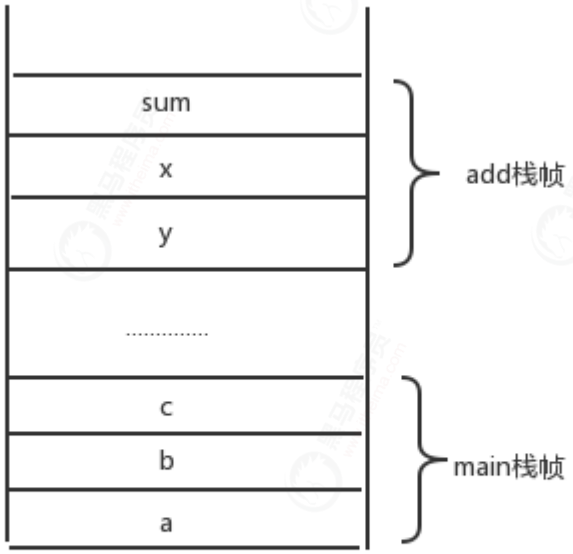
2、堆栈溢出

递归最常见的问题就是堆栈溢出，从而造成应用崩溃，那为什么递归代码会很容易造成堆栈溢出呢?

接下来我们看一段代码：

```
int main() {
   int a = 1;
   int b = 0;
   int c = 0;
   b = add(3, 5);
   c = a +b;
   return 0;
}
/*
* 求和
*/
int add(int x, int y) {
   int sum = 0;
   sum = x + y;
   return sum;
}
```

操作系统给每个线程分配了一块独立的内存空间，这块内存被组织成“栈”这种结构,称为函数调用栈。 用来存储函数调用时的临时变量。每进入一个函数，就会将临时变量作为一个栈帧入栈，当被调用函数执行完成，返回之后，将这个函数对应的栈帧出栈。从代码中我们可以看出，main() 函数调用了 add() 函数，获取计算结果，并且与临时变量 a 相加。为了让大家清晰地看到这个过程对应的函数栈里出栈、入栈的操作，我画了一张图。图中显示的是，在执行到 add() 函数时，函数调用栈的情况。



通过图我们发现递归代码在执行的过程中，每一次递归的调用都会向函数调用栈中压入临时变量，直到满足递归终止条件在回归的过程中才会依次的将临时变量压出栈，如果递归调用的层次很深，一直在压入栈，我们知道系统栈或者虚拟机栈的空间一般都不大，所以如果一直入栈就会出现堆栈溢出的风险。

> 递归代码隐含的使用了系统的“栈”

```
public int fib(int n) {
    //终止条件
    if (n < 2 ) {
        return n ;
    }else {
        int x = fib(n-1) % 1000000007;
        int y = fib(n-2) % 1000000007;
        return  (x+y) %  1000000007;
    }
}
```

> **重要：**通过这个我们能推断出递归的空间复杂度，如果递归过程中不额外开辟存储空间的话，**递归的空间复杂度就跟递归的深度有关系**，每次递归是O(1)，如果递归的深度跟数据规模n有关系，那空间复杂度就有可能是O(N)
>
> 而递归的时间复杂度则需要看递归的场景。大部分都是O(n)的。

那我们在编码的过程中如何避免堆栈溢出呢?

我们可以在代码中限制递归调用的最大深度，比如递归调用超过10000次后就不在继续递归调用了，直接返回或者抛出异常。但是其实这样去做并不能完全的解决问题，因为当前线程能允许的最大递归深度其实跟当前剩余的栈空间大小有关系，事先是不知道有多大的，如果想知道就得实时的去计算剩余的栈空间大小，这样也会影响我们的性能，所以说如果递归的深度不是特别大的话是可以使用这种方式来防止堆栈溢出的，否则的话这种办法也不合适。

2.2.4、70. 爬楼梯

字节，阿里，华为最近面试题，70. 爬楼梯

如果观察数学规律，可知本题是斐波那契数列，所以题解就是斐波拉契数列的题解

```
class Solution {
    //递推公式 f(n) = f(n-1) + f(n-2),其中f(1) = 1,f(2) = 2
    public int climbStairs(int n) {
        return recur(n,new HashMap());
    }

    public int recur(int n,Map<Integer,Integer> cache) {
        if ( n <= 2 ) {
            cache.put(n,n);
            return n;
        }
        if (cache.containsKey(n)) {
            return cache.get(n);
        }
        int a = recur(n-1,cache);
        cache.put(n-1,a);
        int b = recur(n-2,cache);
        cache.put(n-2,b);
        int res = (a + b);
        cache.put(n,res);
        return res;
    }
}
```

当然对于斐波拉契数列也有对应的数学解法，需要找到斐波拉契数列的通项公式：

思路

之前的方法我们已经讨论了 $f(n)$ 是齐次线性递推，根据递推方程 $f(n)=f(n-1)+f(n-2)$，我们可以写出这样的特征方程：

$$x^2=x+1$$

求得 $x_1=\frac{1+\sqrt{5}}{2}$，$x_2=\frac{1-\sqrt{5}}{2}$，设通解为 $f(n)=c_1x_1^n+c_2x_2^n$，代入初始条件 $f(1)=1$，$f(2)=1$，得 $c_1=\frac{1}{\sqrt{5}}$，$c_2=-\frac{1}{\sqrt{5}}$，我们得到了这个递推数列的通项公式：

$$f(n)=\frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n-\left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

接着我们就可以通过这个公式直接求第 n 项了。

```
public class Solution {
    public int climbStairs(int n) {
        double sqrt5 = Math.sqrt(5);
        double fibn = Math.pow((1 + sqrt5) / 2, n + 1) - Math.pow((1 - sqrt5) / 2, n + 1);
        return (int)(fibn / sqrt5);
    }
}
```

2.2.5、22. 括号生成

滴滴，网易最近面试题，22. 括号生成

1：先考虑生成所有组合的括号，

```
package com.itheima.leetcode;

import org.junit.jupiter.api.Test;

import java.util.ArrayList;
import java.util.List;

/**
 * Created by 传智播客*黑马程序员.
 */
public class NO22 {

    public List<String> generateParenthesis(int n) {
        List<String> res = new ArrayList<>();
        recur(1,2*n,"",res);
        return res;
    }
```

```
18
19      public void recur (int level,int max_level,String s,List<String> res) {
20          if (level > max_level) {
21              res.add(s);
22              System.out.println(s);
23              return;
24          }
25          String s1 = s +"(";
26          String s2 = s + ")";
27
28          //下探到下一层
29          recur(level+1,max_level,s1,res);
30          recur(level+1,max_level,s2,res);
31      }
32
33      @Test
34      public void test(){
35          generateParenthesis(3);
36      }
37  }
```

2：然后考虑如何保证是有效的括号

拆分为左括号和右括号+剪枝

```
1   package com.itheima.leetcode;
2
3   import org.junit.jupiter.api.Test;
4
5   import java.util.ArrayList;
6   import java.util.List;
7
8   /**
9    * Created by 传智播客*黑马程序员.
10   */
11  public class NO22 {
12
13      public List<String> generateParenthesis(int n) {
14          List<String> res = new ArrayList<>();
15          recur(0,0,n,"",res);
16          return res;
17      }
18
19      public void recur (int left,int right,int n,String s,List<String> res) {
20          if ( left == n && right == n) {
21              res.add(s);
22              System.out.println(s);
23              return;
24          }
25
26          if (left < n ) {
```

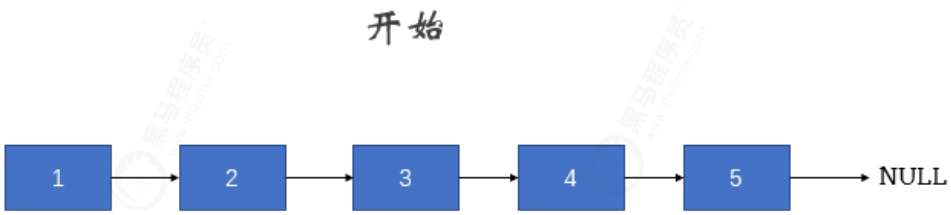
```
            String s1 = s +"(";
            //下探到下一层
            recur(left+1,right,n,s1,res);
        }
        if (right < left) {
            String s2 = s + ")";
            recur(left,right+1,n,s2,res);
        }
    }

    @Test
    public void test(){
        generateParenthesis(3);
    }
}
```

2.2.6、206. 反转链表

腾讯，美团，快手最近面试题，206. 反转链表

使用递归完成链表反转



> 1：使用递归函数，一直递归到链表的最后一个结点，该结点就是反转后的头结点，记作 ret
>
> 2：此后，每次函数在返回的过程中，让当前结点的下一个结点的 next 指针指向当前节点。
>
> 3：同时让当前结点的 next指针指向 NULL ，从而实现从链表尾部开始的局部反转
>
> 4：当递归函数全部出栈后，链表反转完成。

```java
class Solution {
    public ListNode reverseList(ListNode head) {
        //递归终止条件
        if (head == null || head.next == null) {
            return head;
        }
        //处理当前层逻辑
        ListNode node = reverseList(head.next);//假设后面都已经反转好了,注意返回的是反转好的链表头节点
        //如何反转当前节点，让当前节点下一个节点的next指针指向自己，自己的next指针为null
        head.next.next = head;
        head.next = null;
        return node;
    }
}
```

> 至此：我们发现一个有意思的现象
>
> **递归代码的编写就像是在证明一个命题的过程中将这个命题的结果当作已知条件来用**

2.2.7、2. 两数相加

[美团点评，华为最近面试题，2. 两数相加]

> 找重复子问题：每一位都是 l1.val + l2.val +carry；具备重复子问题特征，可以用递归
>
> 根据代码模板编写
>
> 1：终止条件：l1 == null && l2 == null && carry==0
>
> 2：处理当前层逻辑 l1.val + l2.val +carry，构造结果的当前节点
>
> 3：进入到下一层，并把下一层的返回接到当前节点的next指针上
>
> 4：返回当前节点
>
> 整体来看：递归去的时候计算每层结果值和进位，将进位带到下一层，直到最后；返回的时候依次把各节点连接上

```java
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        return recur(l1,l2,0);
    }

    public ListNode recur(ListNode l1, ListNode l2,int carry) {
        if ( l1 == null && l2 == null && carry ==0) {
            return null;
        }
        int a = l1 == null ? 0:l1.val;
        int b  = l2 == null ? 0:l2.val;
        int v = (a + b + carry ) % 10;
```

```
        carry = (a+b+carry) /10;
        ListNode node = new ListNode(v);
        node.next = recur(l1 == null? null:l1.next,l2==null?null:l2.next,carry);
        return node;
    }
}
```

2.2.8、其他题目

题目一：24. 两两交换链表中的节点

题目二：21. 合并两个有序链表

题目三：25. K 个一组翻转链表

2.3、递归问题的思考要点

要点1：不要人肉递归

要点2：一定要找最近重复子问题（重复子问题决定了可以使用递归，如果没有重复性，那复杂度就是客观存在的）

要点3：学会使用数学归纳法（可以使用数学方法来解决问题）

> 一个算法所需时间由下述递归方程表示， n = 1时 T(n) = 1 , n > 1时 T(n) = 2T(n/2) + n 该算法的时间复杂度是()
>
> A: O(n * log(n))
>
> B: O(n ^ 2)
>
> C: O(n)
>
> D: O(log(n))
>
> 题解：

来再给大家推一遍：

$$T(n)=\begin{cases}1, & n=1\\ 2T(\frac{n}{2})+n, & n>1\end{cases}$$

则迭代推 $T(n)=2T(\frac{n}{2})+n$ ……第1次

$=2[2T(\frac{n}{2^2})+\frac{n}{2}]+n$

$=2^2T(\frac{n}{2^2})+2n$ ……第2次

$=2^2[2T(\frac{n}{2^3})+\frac{n}{2^2}]+2n$

$=2^3T(\frac{n}{2^3})+3n$ ……第3次

故数学归纳法可得：第k次：$T(n)=2^kT(\frac{n}{2^k})+kn$ (*)

当 $\frac{n}{2^k}=1 \Rightarrow 2^k=n$ 时

$\begin{cases}k=\log_2 n & ①\\ T(1)=1 & ②\end{cases}$

①②代入(*)式可得：$T(n)=nT(1)+\log_2 n\cdot n$

$=n+n\log_2 n$

故：$T(n)=O(n\log_2 n)$

要点4：做题时按照模板思路来编写代码，减少对递归状态树的依赖，甚至不要（学习初期可以使用）