链表: 最简单的递归结构

递归

递归，本质上就是将当前的问题转化成更小的同一个问题

阶乘

n! = n × (n-1) × (n-2) × ... × 1

n! = n × (n-1)!
(n-1)! = (n-1) × (n-2)!　(n-1)! 就是更小的问题

2! = 2 × (2-1)!
1! = 1　1! = 1 就是最基本的问题

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n-1)
```

递归的关键就在于如何将原任务分解成更小的相同任务，以及找到最基本的问题的解

动态规划问题

有n阶台阶，小青蛙每次能跳1阶或者2阶，求小青蛙跳上该台阶共有多少种跳法

最基本的问题是什么？

所谓最基本的问题，就是能够直接给出答案的情况

假设只有1个台阶，那么小青蛙只能跳1次，所以总计只有1种跳法
假设只有2个台阶，那么小青蛙可以连续跳2次，或者一次跳2个台阶，共2种跳法

```
def jump(n):
    if n <= 2:
        return 2
```

更小的问题是什么？

答案隐藏在了题干中: 小青蛙每次能跳1阶或者2阶

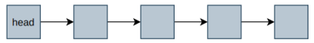
如果小青蛙第一次跳了1阶，那么还剩下n-1阶
如果小青蛙第一次跳了2阶，那么还剩下n-2阶

f(n) = f(n-1) + f(n-2)

实现

```
def jump(n):
    if n <= 2:
        return 2
    return jump(n-1) + jump(n-2)
```

链表天然的递归结构



对于一个链表来说，我们可以将其拆分成两部分: 头结点+少了头结点的链表

而少了头结点的链表又可以继续地拆分: 头结点+少了头结点的链表。最终，就拆分成了一个头节点+NULL

使用递归的方式删除链表特定元素

需求　就地删除链表中值为value的全部元素，并保持链表中的顺序

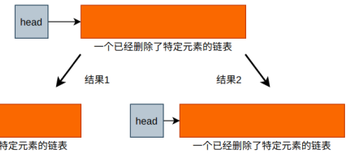
分析

如果利用递归的思想，首先对链表进行拆分，能够将其拆分成头结点和一个更短的链表，这里称之为子链表

假设子链表已经是删除了特定元素的链表，那么现在要做的事情就是将子链表和头结点合并

头结点的值不是value　头结点+子链表就是最终结果，返回原有头结点

头结点的值是value　那么子链表就是最终的结果，将子链表的头结点返回即可



实现

```
def delete(self, head, value):
    if head is None:
        return None

    head.next = self.delete(head.next, value)

    return head.next if head.data == value else head
```

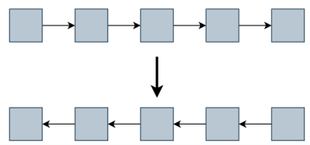
使用递归的方式会存在最大调用栈深度的问题，一个解决的方法是使用尾递归进行优化，但是尾递归的实现在不同场景中的编码复杂程度也不尽相同

反转链表

反转二叉树，Max Howell的一生之敌 ：）

链表的反转和上面删除链表特定元素的思路基本上是一样的，首先构建出子问题，构建子问题之后在解决最基本问题，最基本问题解决之后再对结果进行组装

需求



将一个链表完全倒过来　不是把白板倒过来 ：）

分析

最基本问题是什么？

链表本身就是空链表，或者是只剩下一个节点的时候。这时候得把该节点返回

```
def revert(self, head):
    if not (head and head.next):
        return head
```

子问题是什么？

现在链表有两部分: 原始的头结点和已经反转了的链表，那么只需要将原始头结点给处理掉，整个链表就是反转的

```
tail = head.next
tail.next = head
head.next = None
```

实现

```
def revert(self, head):
    if not (head and head.next):
        return head
    new_head = self._revert(head.next)
    head.next.next = head
    head.next = None
    return new_head
```

编写递归函数的第一个步骤就是明确递归函数的用途，也就是这函数到底要做什么。另外就是不要带太多脑子，避免思维陷入递归深度中