14 两数相加

更新时间：2019-08-22 11:51:09

> 立志是事业的大门，工作是登堂入室的旅程。
>
> ——巴斯德

刷题内容

> 难度: Medium
>
> 原题连接：https://leetcode-cn.com/problems/add-two-numbers/

内容描述

给出两个 非空 的链表用来表示两个非负的整数。其中，它们各自的位数是按照 逆序 的方式存储的，并且它们的每个节点只能存储一位数字。

如果，我们将这两个数相加起来，则会返回一个新的链表来表示它们的和。

您可以假设除了数字 0 之外，这两个数都不会以 0 开头。

示例：

输入：(2 -> 4 -> 3) + (5 -> 6 -> 4)
输出：7 -> 0 -> 8
原因：342 + 465 = 807

解题方案

思路 1：时间复杂度: O(N) 空间复杂度: O(N)

从题目中可以得知：

- 两个链表都是非空的，也就是至少拥有一个节点；
- 链表存储的是非负整数，且其位数是按照逆序的方式存储的 例如：342 存为 2 -> 4 -> 3；
- 非负整数不会以 0 开头，因此我们不需要考虑链表末尾有无数个 0 的情况；
- 最后两数相加的结果也要存为链表返回，并且是逆序表示的。

别笑，你第一想法是不是将 l1 和 l2 全部变成数字做加法再换回去？这是我们最直接的想法了，那好，我们就来实现一下。

下面来看具体代码：

Python beats 54.35%

```python
class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        # 获得 l1 和 l2 的字符串表示, 因为题目说了l1和l2均非空，这里可以直接取val不会有问题
        num1Str, num2Str = str(l1.val), str(l2.val)

        while l1.next:
            num1Str += str(l1.next.val)
            l1 = l1.next
        while l2.next:
            num2Str += str(l2.next.val)
            l2 = l2.next

        # 得到 l1 和 l2 相加之和，因为在链表中数字是逆序存储，所以要反转一下
        sums = int(num1Str[::-1]) + int(num2Str[::-1])

        # 将 sums 转成题目中 linkedlist 所对应的表示形式
        sums = str(sums)[::-1]

        # dummy.next 作为返回结果
        dummy = head = ListNode(None)
        for i in range(len(sums)):
            head.next = ListNode(int(sums[i]))
            head = head.next
        return dummy.next
```


Java beats 52.43%

```java
import java.math.BigInteger;
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        String numStr1 = String.valueOf(l1.val);
        String numStr2 = String.valueOf(l2.val);

        // 拼接 l1 的值
        while (l1.next != null){
            numStr1 += String.valueOf(l1.next.val);
            l1 = l1.next;
        }

        // 拼接 l2 的值
        while (l2.next != null) {
            numStr2 += String.valueOf(l2.next.val);
            l2 = l2.next;
        }

        // 使用 BigInteger 是为了防止大数的溢出
        BigInteger num1 = new BigInteger(new StringBuffer(numStr1).reverse().toString());
        BigInteger num2 = new BigInteger(new StringBuffer(numStr2).reverse().toString());

        BigInteger sum = num1.add(num2);

        String sumStr = new StringBuffer(String.valueOf(sum)).reverse().toString();

        ListNode head = new ListNode(0);
        ListNode dummy = head;
        // 将字符串 sum 转化为链表形式的 sum
        for (int i = 0; i < sumStr.length(); i++) {
            head.next = new ListNode(Integer.parseInt(String.valueOf(sumStr.charAt(i))));
            head = head.next;
        }
        return dummy.next;
    }
}
```


Go beats 89.43%

```go
// 反转字符串的辅助函数
func reverse(s string) string {
    runes := []rune(s)
    for i, j := 0, len(runes)-1; i < j; i, j = i+1, j-1 {
        runes[i], runes[j] = runes[j], runes[i]
    }
    return string(runes)
}

func addTwoNumbers(l1 *ListNode, l2 *ListNode) *ListNode {
    // 获得 l1 和 l2 的 字符串表示, 因为题目说了l1和l2均非空，这里可以直接取val不会有问题
    num1Str := strconv.Itoa(l1.Val)
    for l1.Next != nil {
        num1Str += strconv.Itoa(l1.Next.Val)
        l1 = l1.Next
    }

    num2Str := strconv.Itoa(l2.Val)
    for l2.Next != nil {
        num2Str += strconv.Itoa(l2.Next.Val)
        l2 = l2.Next
    }

    // 反转两个字符串，因为我们的链表是逆序的
    num1Str = reverse(num1Str)
    num2Str = reverse(num2Str)

    // 求出 l1 代表的数字，注意这里可能会有大数溢出
    num1 := new(big.Int)
    num1, _ = num1.SetString(num1Str, 10)

    // 求出 l2 代表的数字，注意这里可能会有大数溢出
    num2 := new(big.Int)
    num2, _ = num2.SetString(num2Str, 10)

    // 得到 l1 和 l2 相加之和, 注意这里可能会有大数溢出
    sums := new(big.Int)
    sums = sums.Add(sums, num1)
    sums = sums.Add(sums, num2)
    sumsStr := sums.String()
    sumsRune := make([]rune, 0)
    for i, _ := range sumsStr {
        sumsRune = append(sumsRune, rune(sumsStr[len(sumsStr)-1-i]))
    }

    // 将 sums 转成题目中 linkedlist 所对应的表示形式
    dummy := &ListNode{Val:0}
    head := dummy
    for _, digit := range sumsRune {
        head.Next = &ListNode{Val:int(digit - '0')}
        head = head.Next
    }

    // dummy.Next 作为返回结果
    return dummy.Next
}
```


C++

> 注意：本思路只适用于大数基础数据结构的语言，例如 java 的 BigInteger 和 go 的 bigint，c++需要自己实现大数加法，详见思路2。

思路1的话，我们先把相关信息全部存成我们适应处理的结构，然后才去处理，这样的话会浪费一定的空间，我们能做一些优化吗？不存下来可以吗？

思路 2：时间复杂度：O(N) 空间复杂度：O(N)

因为我们一定得遍历完 `l1` 和 `l2` 的每一位才能得到最终结果，所以时间复杂度为 `O(N)` 没得商量。

虽然时间复杂度无法减小，但是我们可以考虑减小我们的空间复杂度啊，刚才我们是将 `l1` 和 `l2` 全部转回数字，然后用两个列表将它们的数字形式存了下来，这消耗了 `O(N)` 的空间。

实际上我们完全可以模拟真正的加法操作，即从个位数开始相加，如果有进位就记录一下，等到十位数相加的时候记得加上那个进位 `1` 就可以了，这是我们小学就学过的知识。

那么我们就先处理个位数的相加。然后我们发现处理十位数、百位数和后面的位数都和个位数相加的操作是一个样子的，只不过后面计算的结果乘上 `10` 再加上个位数相加的结果，这才是最终的结果。

于是我们就想到了用递归的方法，即一步一步将大问题转化为更小的问题，直到遇到基础情况（这里指的是个位数相加）返回即可。

下面我们来看代码：

Python beats 83.40%

```python
class Solution:
    def addTwoNumbers(self, l1, l2):
        """
        :type l1: ListNode
        :type l2: ListNode
        :rtype: ListNode
        """
        # 因为处理到最后的时候，可能输入的 l1 和 l2 都不是一个 ListNode 而是 None 了
        if not l1 and not l2:
            return
        elif not (l1 and l2): # l1 和 l2 其中一个是 None
            return l1 or l2
        else: # l1 和 l2 都不是 None
            if l1.val + l2.val < 10: # 个位数相加没有进位
                l3 = ListNode(l1.val+l2.val)
                l3.next = self.addTwoNumbers(l1.next, l2.next) # 递归调用
            else: # 个位数相加有进位
                l3 = ListNode(l1.val+l2.val-10)
                # 递归调用，记得加上进位
                l3.next = self.addTwoNumbers(l1.next, self.addTwoNumbers(l2.next, ListNode(1)))
        return l3
```


Java beats 94.96%

```java
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode l3 = null;
        if (l1 == null && l2 == null) {
            // l1 与 l2 都为 null 的情况
            return null;
        } else if (l1 == null && l2 != null) {
            // l1 为 null 但是 l2 不为 null 的情况
            return l2;
        } else if (l1 != null && l2 == null) {
            // l1 不为 null 但是 l2 为 null 的情况
            return l1;
        } else {
            // l1 l2 都不为 null 的情况
            if (l1.val + l2.val < 10) {
                // 不需要进位的情况
                l3 = new ListNode(l1.val + l2.val);
                l3.next = addTwoNumbers(l1.next, l2.next);
            }else {
                // 需要进位的情况
                l3 = new ListNode(l1.val + l2.val - 10);
                l3.next = addTwoNumbers(l1.next, addTwoNumbers(l2.next, new ListNode(1)));
            }
        }
        return l3;
    }
}
```

Go beats 93.98%

```go
func addTwoNumbers(l1 *ListNode, l2 *ListNode) *ListNode {
    // 因为处理到最后的时候，可能输入的 l1 和 l2 都不是一个 ListNode 而是 nil 了
    if l1 == nil && l2 == nil {
        return nil
    } else if l1 == nil || l2 == nil { // l1 和 l2 其中一个是 nil
        if l1 == nil {
            return l2
        }
        if l2 == nil {
            return l1
        }
    } else {
        if l1.Val + l2.Val < 10 { // 个位数相加没有进位
            l3 := ListNode{Val: l1.Val + l2.Val}
            l3.Next = addTwoNumbers(l1.Next, l2.Next) // 递归调用
            return &l3
        } else {
            l3 := ListNode{Val: l1.Val + l2.Val - 10}
            // 递归调用，记得加上进位
            l3.Next = addTwoNumbers(l1.Next, addTwoNumbers(l2.Next, &ListNode{Val: 1}))
            return &l3
        }
    }
    return nil
}
```


c++ beats: 95.51%

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode* l3 = NULL;
        if (l1 == NULL && l2 == NULL) {
            // l1 与 l2 都为 null 的情况
            return NULL;
        } else if (l1 == NULL && l2 != NULL) {
            // l1 为 null 但是 l2 不为 null 的情况
            return l2;
        } else if (l1 != NULL && l2 == NULL) {
            // l1 不为 null 但是 l2 为 null 的情况
            return l1;
        } else {
            // l1 l2 都不为 null 的情况
            if (l1->val + l2->val < 10) {
                // 不需要进位的情况
                l3 = new ListNode(l1->val + l2->val);
                l3->next = addTwoNumbers(l1->next, l2->next);
            }else {
                // 需要进位的情况
                l3 = new ListNode(l1->val + l2->val - 10);
                l3->next = addTwoNumbers(l1->next, addTwoNumbers(l2->next, new ListNode(1)));
            }
        }
        return l3;
    }
};
```

尽管这次我们没有先存储再处理，为什么递归还是是O(N)的空间复杂度呢？因为我们虽然没有手动存储，但是计算机内部还是帮我们存储了一个函数调用栈的，又因为每调一次我们栈的高度都会增加1，而这里显然我们需要调用N次，N指的是较长的那个链表的长度。

现在我们既不想手动存储，也不想计算机帮我们存储，我们就不想用额外空间，可以吗？

思路 3 时间复杂度: O(N) 空间复杂度: O(1)

好的，我们不用递归了，但是我们可以用迭代的方式来模拟这个过程，这样我们的空间可算是节省下来了。

Python beats 86.80%

```python
class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        head, tail = None, None
        one = False # one 为True表示进位
        while l1 or l2:
            val = 0
            if l1:
                val += l1.val
                l1 = l1.next
            if l2:
                val += l2.val
                l2 = l2.next
            if one:
                val += 1
            if val >= 10: # 判断是否进位
                val -= 10
                one = True
            else:
                one = False
            if not head:
                head = tail = ListNode(val)
            else:
                tail.next = ListNode(val)
                tail = tail.next
        if one:
            tail.next = ListNode(1)
        return head
```

Java beats 87.41%

```java
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode head = null;
        ListNode tail = null;
        // one 用来判断是否应该进位
        boolean one = false;
        while (l1 != null || l2 != null) {
            int val = 0;
            if (l1 != null) {
                val += l1.val;
                l1 = l1.next;
            }
            if (l2 != null) {
                val += l2.val;
                l2 = l2.next;
            }
            if (one) {
                val += 1;
            }
            // 判断是否进位
            if (val >= 10) {
                val -= 10;
                one = true;
            }else {
                one = false;
            }
            if (head == null) {
                head = tail = new ListNode(val);
            }else {
                tail.next = new ListNode(val);
                tail = tail.next;
            }
        }
        if (one) {
            tail.next = new ListNode(1);
        }
        return head;
    }
}
```


Go beats 96.36%

```go
func addTwoNumbers(l1 *ListNode, l2 *ListNode) *ListNode {
    var head, tail *ListNode
    var one bool
    for l1 != nil || l2 != nil {
        val := 0
        if l1 != nil {
            val += l1.Val
            l1 = l1.Next
        }
        if l2 != nil {
            val += l2.Val
            l2 = l2.Next
        }
        if one {
            val += 1
        }
        if val >= 10 {
            val -= 10
            one = true
        } else {
            one = false
        }
        if head == nil {
            head = &ListNode{Val: val}
            tail = head
        } else {
            tail.Next = &ListNode{Val: val}
            tail = tail.Next
        }
    }
    if one {
        tail.Next = &ListNode{Val: 1}
    }
    return head
}
```


C++ beats 99.22%

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode *head = NULL, *tail = NULL;
        bool one = false; //是否进位
        while (l1 != NULL || l2 != NULL) {
            int val = 0;
            if (l1) val += l1->val;
            if (l2) val += l2->val;
            if (one) val++;
            //判断是否进位
            if (val >= 10) {
                val -= 10;
                one = true;
            } else {
                one = false;
            }
            if (!head) {
                head = tail = new ListNode(val);
            } else {
                tail->next = new ListNode(val);
                tail = tail->next;
            }
            if (l1) l1 = l1->next;
            if (l2) l2 = l2->next;
        }
        if (one) {
            tail->next = new ListNode(1);
        }
        return head;
    }
};
```


好的，我们现在没用额外空间来存储了，而是直接在链表上面操作，这样更节省时间。

总结

- 看题目的时候要注意非空、逆序、整数这样的敏感字眼；
- 要注意有的时候大数溢出是很容易被忽略的点；
- 我们知道了递归会涉及到电脑里的调用栈，所以递归还是会消耗额外空间的；
- 有的时候用迭代模拟递归的过程可以节省空间。

学习更多

- 链表的应用场景
- 大整数类的实现

}

13 实现 strStr() 函数

15 无重复字符的最长子串