

07 | 链表（下）：如何轻松写出正确的链表代码？

2018-10-05 王争



07 | 链表（下）：如何轻松写出正确的链表代码？

朗读人：修阳 12'30'' | 5.73M

上一节我讲了链表相关的基础知识。学完之后，我看到有人留言说，基础知识我都掌握了，但是写链表代码还是很费劲。哈哈，的确是这样的！

想要写好链表代码并不是容易的事儿，尤其是那些复杂的链表操作，比如链表反转、有序链表合并等，写的时候非常容易出错。从我上百场面试的经验来看，能把“链表反转”这几行代码写对的人不足 10%。

为什么链表代码这么难写？究竟怎样才能比较轻松地写出正确的链表代码呢？

只要愿意投入时间，我觉得大多数人都是可以学会的。比如说，如果你真的能花上一个周末或者一整天的时间，就去写链表反转这一个代码，多写几遍，一直练到能毫不费力地写出 **Bug free** 的代码。这个坎还会很难跨吗？

当然，自己有决心并且付出精力是成功的先决条件，除此之外，我们还需要一些方法和技巧。我根据自己的学习经历和工作经验，总结了几个写链表代码技巧。如果你能熟练掌握这几个技巧，加上你的主动和坚持，轻松拿下链表代码完全没有问题。

技巧一：理解指针或引用的含义

事实上，看懂链表的结构并不是很难，但是一旦把它和指针混在一起，就很容易让人摸不着头脑。所以，要想写对链表代码，首先就要理解好指针。

我们知道，有些语言有“指针”的概念，比如 C 语言；有些语言没有指针，取而代之的是“引用”，比

如 **Java**、**Python**。不管是“指针”还是“引用”，实际上，它们的意思都是一样的，都是存储所指对象的内存地址。

接下来，我会拿 **C** 语言中的“指针”来讲解，如果你用的是 **Java** 或者其他没有指针的语言也没关系，你把它理解成“引用”就可以了。

实际上，对于指针的理解，你只需要记住下面这句话就可以了：

将某个变量赋值给指针，实际上就是将这个变量的地址赋值给指针，或者反过来说，指针中存储了这个变量的内存地址，指向了这个变量，通过指针就能找到这个变量。

这句话听起来还挺拗口的，你可以先记住。我们回到链表代码的编写过程中，我来慢慢给你解释。

在编写链表代码的时候，我们经常会有这样的代码：**p->next=q**。这行代码是说，**p** 结点中的 **next** 指针存储了 **q** 结点的内存地址。

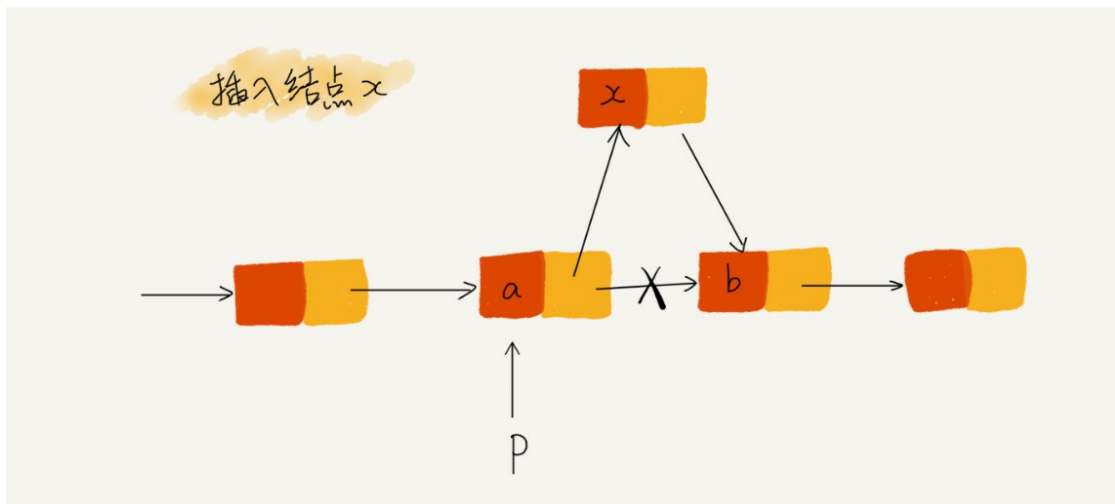
还有一个更复杂的，也是我们写链表代码经常会用到的：**p->next=p->next->next**。这行代码表示，**p** 结点的 **next** 指针存储了 **p** 结点的下下一个结点的内存地址。

掌握了指针或引用的概念，你应该可以很轻松地看懂链表代码。恭喜你，已经离写出链表代码近了一步！

技巧二：警惕指针丢失和内存泄漏

不知道你有没有这样的感觉，写链表代码的时候，指针指来指去，一会儿就不知道指到哪里了。所以，我们在写的时候，一定注意不要弄丢了指针。

指针往往都是怎么弄丢的呢？我拿单链表的插入操作为例来给你分析一下。



如图所示，我们希望在结点 **a** 和相邻的结点 **b** 之间插入结点 **x**，假设当前指针 **p** 指向结点 **a**。如果我们将代码实现变成下面这个样子，就会发生指针丢失和内存泄露。

```
1 p->next = x; // 将 p 的 next 指针指向 x 结点；
2 x->next = p->next; // 将 x 的结点的 next 指针指向 b 结点；
```

 复制代码

初学者经常会在这一儿犯错。**p->next** 指针在完成第一步操作之后，已经不再指向结点 **b** 了，而是指向结点 **x**。第 2 行代码相当于将 **x** 赋值给 **x->next**，自己指向自己。因此，整个链表也就断成了两

半，从结点 **b** 往后的所有结点都无法访问到了。

对于有些语言来说，比如 **C** 语言，内存管理是由程序员负责的，如果没有手动释放结点对应的内存空间，就会产生内存泄露。所以，我们插入结点时，一定要注意操作的顺序，要先将结点 **x** 的 **next** 指针指向结点 **b**，再把结点 **a** 的 **next** 指针指向结点 **x**，这样才不会丢失指针，导致内存泄漏。所以，对于刚刚的插入代码，我们只需要把第 1 行和第 2 行代码的顺序颠倒一下就可以了。

同理，删除链表结点时，也一定要记得手动释放内存空间，否则，也会出现内存泄漏的问题。当然，对于像 **Java** 这种虚拟机自动管理内存的编程语言来说，就不需要考虑这么多了。

技巧三：利用哨兵简化实现难度

首先，我们先来回顾一下单链表的插入和删除操作。如果我们在结点 **p** 后面插入一个新的结点，只需要下面两行代码就可以搞定。

```
1 new_node->next = p->next;
2 p->next = new_node;
```

 复制代码

但是，当我们要向一个空链表中插入第一个结点，刚刚的逻辑就不能用了。我们需要进行下面这样的特殊处理，其中 **head** 表示链表的头结点。所以，从这段代码，我们可以发现，对于单链表的插入操作，第一个结点和其他结点的插入逻辑是不一样的。

```
1 if (head == null) {
2     head = new_node;
3 }
```

 复制代码

我们再来看单链表结点删除操作。如果要删除结点 **p** 的后继结点，我们只需要一行代码就可以搞定。

```
1 p->next = p->next->next;
```

 复制代码

但是，如果我们要删除链表中的最后一个结点，前面的删除代码就不 **work** 了。跟插入类似，我们也需要对于这种情况特殊处理。写成代码是这样子的：

```
1 if (head->next == null) {
2     head = null;
3 }
```

 复制代码

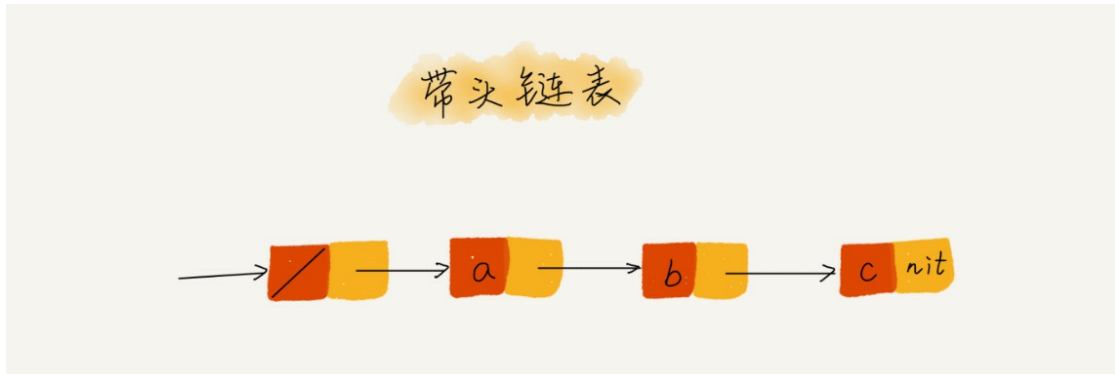
从前面的一步一步分析，我们可以看出，针对链表的插入、删除操作，需要对插入第一个结点和删除最后一个结点的情况进行特殊处理。这样代码实现起来就会很繁琐，不简洁，而且也容易因为考虑不全而出错。如何解决这个问题呢？

技巧三中提到的哨兵就要登场了。哨兵，解决的是国家之间的边界问题。同理，这里说的哨兵也是解决“边界问题”的，不直接参与业务逻辑。

还记得如何表示一个空链表吗？**head=null** 表示链表中没有结点了。其中 **head** 表示头结点指针，指向链表中的第一个结点。

如果我们引入哨兵结点，在任何时候，不管链表是不是空，**head** 指针都会一直指向这个哨兵结点。我们也把这种有哨兵结点的链表叫带头链表。相反，没有哨兵结点的链表就叫作不带头链表。

我画了一个带头链表，你可以发现，哨兵结点是不存储数据的。因为哨兵结点一直存在，所以插入第一个结点和插入其他结点，删除最后一个结点和删除其他结点，都可以统一为相同的代码实现逻辑了。



实际上，这种利用哨兵简化编程难度的技巧，在很多代码实现中都有用到，比如插入排序、归并排序、动态规划等。这些内容我们后面才会讲，现在为了让你感受更深，我再举一个非常简单的例子。代码我是用 **C** 语言实现的，不涉及语言方面的高级语法，很容易看懂，你可以类比到你熟悉的语言。

代码一：

```
1 // 在数组 a 中，查找 key，返回 key 所在的位置
2 // 其中，n 表示数组 a 的长度
3 int find(char* a, int n, char key) {
4     // 边界条件处理，如果 a 为空，或者 n<=0，说明数组中没有数据，就不用 while 循环比较了
5     if(a == null || n <= 0) {
6         return -1;
7     }
8
9     int i = 0;
10    // 这里有两个比较操作：i<n 和 a[i]==key.
11    while (i < n) {
12        if (a[i] == key) {
13            return i;
14        }
15        ++i;
16    }
17
18    return -1;
19 }
```

复制代码

代码二：

```
1 // 在数组 a 中，查找 key，返回 key 所在的位置
2 // 其中，n 表示数组 a 的长度
3 // 我举 2 个例子，你可以拿例子走一下代码
4 // a = {4, 2, 3, 5, 9, 6}  n=6 key = 7
5 // a = {4, 2, 3, 5, 9, 6}  n=6 key = 6
6 int find(char* a, int n, char key) {
7     if(a == null || n <= 0) {
8         return -1;
9     }
10
11     // 这里因为要将 a[n-1] 的值替换成 key，所以要特殊处理这个值
12     if (a[n-1] == key) {
13         return n-1;
14     }
15
16     // 把 a[n-1] 的值临时保存在变量 tmp 中，以便之后恢复。tmp=6。
17     // 之所以这样做的目的是：希望 find() 代码不要改变 a 数组中的内容
18     char tmp = a[n-1];
19     // 把 key 的值放到 a[n-1] 中，此时 a = {4, 2, 3, 5, 9, 7}
20     a[n-1] = key;
21
22     int i = 0;
23     // while 循环比起代码一，少了 i<n 这个比较操作
24     while (a[i] != key) {
25         ++i;
26     }
27
28     // 恢复 a[n-1] 原来的值，此时 a= {4, 2, 3, 5, 9, 6}
29     a[n-1] = tmp;
30
31     if (i == n-1) {
32         // 如果 i == n-1 说明，在 0...n-2 之间都没有 key，所以返回 -1
33         return -1;
34     } else {
35         // 否则，返回 i，就是等于 key 值的元素的下标
36         return i;
37     }
38 }
```

对比两段代码，在字符串 **a** 很长的时候，比如几万、几十万，你觉得哪段代码运行得更快点呢？答案是代码二，因为两段代码中执行次数最多就是 **while** 循环那一部分。第二段代码中，我们通过一个哨兵 **a[n-1] = key**，成功省掉了一个比较语句 **i<n**，不要小看这一条语句，当累积执行万次、几十万次时，累积的时间就很明显了。

当然，这只是为了举例说明哨兵的作用，你写代码的时候千万不要写第二段那样的代码，因为可读性太差了。大部分情况下，我们并不需要如此追求极致的性能。

技巧四：重点留意边界条件处理

软件开发中，代码在一些边界或者异常情况下，最容易产生 **Bug**。链表代码也不例外。要实现没有 **Bug** 的链表代码，一定要在编写的过程中以及编写完成之后，检查边界条件是否考虑全面，以及代码在边界条件下是否能正确运行。

我经常用来检查链表代码是否正确的边界条件有这样几个：

- 如果链表为空时，代码是否能正常工作？

- 如果链表只包含一个结点时，代码是否能正常工作？
- 如果链表只包含两个结点时，代码是否能正常工作？
- 代码逻辑在处理头结点和尾结点的时候，是否能正常工作？

当你写完链表代码之后，除了看下你写的代码在正常的情况下能否工作，还要看下在上面我列举的几个边界条件下，代码仍然能否正确工作。如果这些边界条件下都没有问题，那基本上可以认为没有问题了。

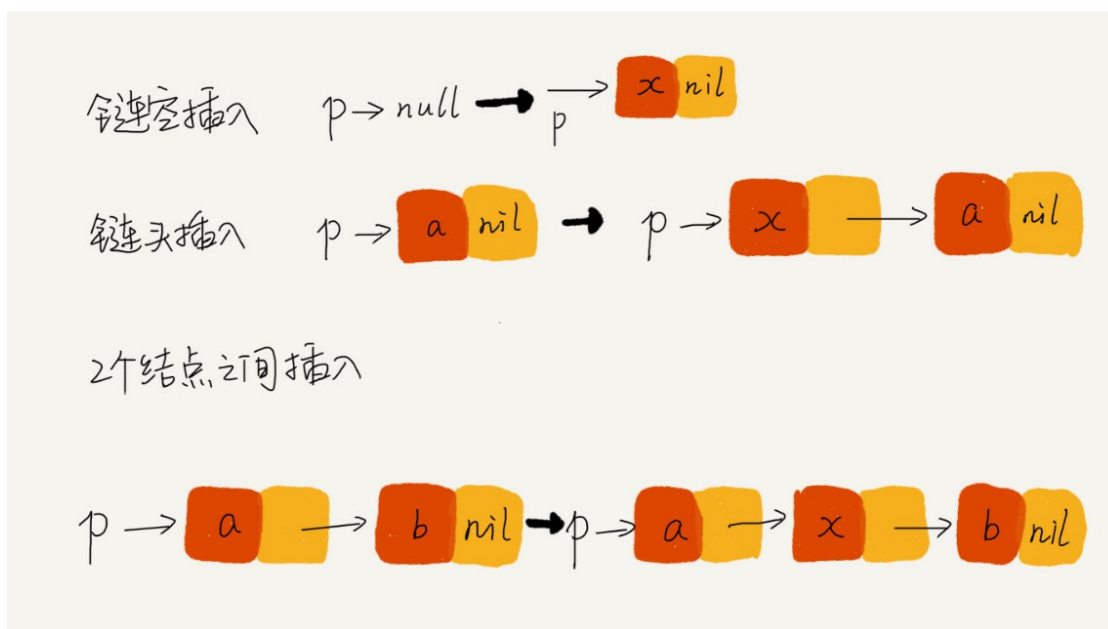
当然，边界条件不止我列举的那些。针对不同的场景，可能还有特定的边界条件，这个需要你自己去思考，不过套路都是一样的。

实际上，不光是写链表代码，你在写任何代码时，也千万不要只是实现业务正常情况下的功能就好了，一定要多想想，你的代码在运行的时候，可能会遇到哪些边界情况或者异常情况。遇到了应该如何应对，这样写出来的代码才够健壮！

技巧五：举例画图，辅助思考

对于稍微复杂的链表操作，比如前面我们提到的单链表反转，指针一会儿指这，一会儿指那，一会儿就被绕晕了。总感觉脑容量不够，想不清楚。所以这个时候就要使用大招了，举例法和画图法。

你可以找一个具体的例子，把它画在纸上，释放一些脑容量，留更多的给逻辑思考，这样就会感觉到思路清晰很多。比如往单链表中插入一个数据这样一个操作，我一般都是把各种情况都举一个例子，画出插入前和插入后的链表变化，如图所示：



看图写代码，是不是就简单多啦？而且，当我们写完代码之后，也可以举几个例子，画在纸上，照着代码走一遍，很容易就能发现代码中的 Bug。

技巧六：多写多练，没有捷径

如果你已经理解并掌握了我前面所讲的方法，但是手写链表代码还是会出现各种各样的错误，也不要着急。因为我最开始学的时候，这种状况也持续了一段时间。

现在我写这些代码，简直就和“玩儿”一样，其实也没有什么技巧，就是把常见的链表操作都自己多

写几遍，出问题就一点一点调试，熟能生巧！

所以，我精选了 5 个常见的链表操作。你只要把这几个操作都能写熟练，不熟就多写几遍，我保证你之后再也不会害怕写链表代码。

- 单链表反转
- 链表中环的检测
- 两个有序的链表合并
- 删除链表倒数第 n 个结点
- 求链表的中间结点

内容小结

这节我主要和你讲了写出正确链表代码的六个技巧。分别是理解指针或引用的含义、警惕指针丢失和内存泄漏、利用哨兵简化实现难度、重点留意边界条件处理，以及举例画图、辅助思考，还有多写多练。

我觉得，写链表代码是最考验逻辑思维能力的。因为，链表代码到处都是指针的操作、边界条件的处理，稍有不慎就容易产生 **Bug**。链表代码写得好坏，可以看出一个人写代码是否够细心，考虑问题是否全面，思维是否缜密。所以，这也是很多面试官喜欢让人手写链表代码的原因。所以，这一节讲到的东西，你一定要自己写代码实现一下，才有效果。

课后思考

今天我们讲到用哨兵来简化编码实现，你是否还能够想到其他场景，利用哨兵可以大大地简化编码难度？

欢迎留言和我分享，我会第一时间给你反馈。

[戳此查看本节内容相关的详细代码](#)



The banner features a dark grey background. On the left, the 'Geek Time' logo is at the top, followed by the title '数据结构与算法之美' in large, bold, dark red characters. Below the title is the subtitle '为工程师量身打造的数据结构与算法私教课'. At the bottom left, the name '王争' and his title '前 Google 工程师' are listed. On the right side of the banner is a portrait of a man with glasses and a dark t-shirt, standing with his arms crossed.



0xFFFFFFF

👍 49

练习题LeetCode对应编号：206，141，21，19，876。大家可以去练习，另外建议作者兄每章直接给出LC的题目编号或链接方便大家练习。

2018-10-06

| 作者回复

我可以集中写一篇练习题的。现在这种思考题的方式是早就定好的了。不好改了。

2018-10-06



姜威

👍 28

总结：如何优雅的写出链表代码？6大学习技巧

一、理解指针或引用的含义

1.含义：将某个变量（对象）赋值给指针（引用），实际上就是就是将这个变量（对象）的地址赋值给指针（引用）。

2.示例：

`p->next = q;` 表示p节点的后继指针存储了q节点的内存地址。

`p->next = p->next->next;` 表示p节点的后继指针存储了p节点的下下个节点的内存地址。

二、警惕指针丢失和内存泄漏（单链表）

1.插入节点

在节点a和节点b之间插入节点x，b是a的下一节点，p指针指向节点a，则造成指针丢失和内存泄漏的代码：`p->next = x;x->next = p->next;` 显然这会导致x节点的后继指针指向自身。

正确的写法是2句代码交换顺序，即：`x->next = p->next; p->next = x;`

2.删除节点

在节点a和节点b之间删除节点b，b是a的下一节点，p指针指向节点a：`p->next = p->next->next;`

三、利用“哨兵”简化实现难度

1.什么是“哨兵”？

链表中的“哨兵”节点是解决边界问题的，不参与业务逻辑。如果我们引入“哨兵”节点，则不管链表是否为空，head指针都会指向这个“哨兵”节点。我们把这种有“哨兵”节点的链表称为带头链表，相反，没有“哨兵”节点的链表就称为不带头链表。

2.未引入“哨兵”的情况

如果在p节点后插入一个节点，只需2行代码即可搞定：

```
new_node->next = p->next;
```

```
p->next = new_node;
```

但，若向空链表中插入一个节点，则代码如下：

```
if(head == null){
    head = new_node;
}
```

如果要删除节点p的后继节点，只需1行代码即可搞定：

```
p->next = p->next->next;
```

但，若是删除链表的最有一个节点（链表中只剩下这个节点），则代码如下：

```
if(head->next == null){
    head = null;
}
```

从上面的情况可以看出，针对链表的插入、删除操作，需要对插入第一个节点和删除最后一个节点的情况进行特殊处理。这样代码就会显得很繁琐，所以引入“哨兵”节点来解决这个问题。

3.引入“哨兵”的情况

“哨兵”节点不存储数据，无论链表是否为空，**head**指针都会指向它，作为链表的头结点始终存在。这样，插入第一个节点和插入其他节点，删除最后一个节点和删除其他节点都可以统一为相同的代码实现逻辑了。

4.“哨兵”还有哪些应用场景？

这个知识有限，暂时想不出来呀！但总结起来，哨兵最大的作用就是简化边界条件的处理。

四、重点留意边界条件处理

经常用来检查链表是否正确的边界4个边界条件：

- 1.如果链表为空时，代码是否能正常工作？
- 2.如果链表只包含一个节点时，代码是否能正常工作？
- 3.如果链表只包含两个节点时，代码是否能正常工作？
- 4.代码逻辑在处理头尾节点时是否能正常工作？

五、举例画图，辅助思考

核心思想：释放脑容量，留更多的给逻辑思考，这样就会感觉到思路清晰很多。

六、多写多练，没有捷径

5个常见的链表操作：

- 1.单链表反转
- 2.链表中环的检测
- 3.两个有序链表合并
- 4.删除链表倒数第n个节点
- 5.求链表的中间节点

2018-10-05



zeta

👍 17

建议大家在实现之前的思考时间不要太长。一是先用自己能想到的暴力方法实现试试。另外就是在一定时间内(比如半个到一个小时)实在想不到就要在网上搜搜答案。有的算法，比如链表中环的检测，的最优解法还是挺巧妙的，一般来说不是生想就能想到的

2018-10-06

作者回复

👏，高手！实际上，写链表代码还是主要为了锻炼写代码的能力，倒不是思考解决办法。像环的检测这种解决办法我也想不出来，都是看了答案之后恍然大悟。

2018-10-06



Rain

👍 12

谢谢老师，这节课又学到了，写完留言我要去思考那几个问题了，一个都不会。。

文中提到，

但是，如果我们要删除链表中的最后一个结点，前面的删除代码就不 **work** 了。跟插入类似，我们也需要对于这种情况特殊处理。写成代码是这样子的：

```
if (head->next == null) {  
    head = null;  
}
```

感觉此处代码处理的是当链表中只有表头一个节点的删除情况，而不是"要删除链表中的最后一个结点"的情况。是不是head应该改成p?

2018-10-05



zyzheng

👍 10

一直对手写链表代码有恐惧心理，这次硬着头皮也要迈过这个坎

2018-10-05



来自地狱的勇士

👍 7

问题一：文中提到，指针丢失会导致内存泄露，老师能解释下如何导致的内存泄露吗？

问题二：讲哨兵那块的内容时，说代码二比代码一成功省掉了一次比较 $i < n$ ，这句不大理解，代码二中，while的条件 $a[i] != key$ 也是在比较吧？

2018-10-05



广兴

👍 6

c语言不熟悉 看起来有点吃力

2018-10-05

作者回复

不好意思 我尽量写简单点 多加点注释

2018-10-08



Smallfly

👍 5

如何写好链表代码？

1. 理解指针或引用的含义

什么是指针？指针是一个变量，该变量中存的是其它变量的地址。将普通变量赋值给指针变量，其实是把它的地址赋值给指针变量。

2. 警惕指针丢失和内存泄漏

在插入和删除结点时，要注意先持有后面的结点再操作，否则一旦后面结点的前继指针被断开，就无法再访问，导致内存泄漏。

3. 利用哨兵简化难度

链表的插入、删除操作，需要对插入第一个结点和删除最后一个节点做特殊处理。利用哨兵对象可以不用边界判断，链表的哨兵对象是只存指针不存数据的头结点。

4. 重点留意边界条件处理

操作链表时要考虑链表为空、一个结点、两个结点、头结点、尾结点的情况。学习数据结构和算法主要是掌握一系列思想，能在其它的编码中也养成考虑边界的习惯。

5. 举例画图，辅助思考

对于比较复杂的操作，可以用纸笔画一画，释放脑容量来做逻辑处理（时间换空间思想），也便于完成后的检查。

6. 多写多练，没有捷径

孰能生巧，不管是什么算法，只有经过反复的练习，才能信手拈来。

哨兵对象思想，在 iOS AutoreleasePool 中有用到，在 AutoreleasePoolPush 时添加一个哨兵对象，Pop 时将到哨兵对象之间的所有 Autorelease 对象发送 release 消息。

2018-10-05



五岳寻仙

👍 4

老师您好！请教您一个问题。在学习了数组和链表之后，想知道在现实应用中有没有将二者结合过来的情况。

比如，我想用数组存储数据，但数组大小提前无法知道，如果使用动态数组的话，中间涉及到数组拷贝；如果使用链表的话，每增加一个元素都要`malloc`一次（频繁的`malloc`会不会影响效率并且导致内存碎片？）。

可不可以用链表将数组链接起来？也就是说链表里每个`node`存储了数组指针，这样每增加一个节点就可以多存放很多元素。如果可以的话，与直接使用动态数组或者直接使用链表比有没有什么优缺点，为何在网上搜索几乎找不到人这样用？

2018-10-07

作者回复

思考的深入 你说的这个很像内存池 你可以百度一下看看是不是你想要的

2018-10-08



Miletos

👍 4

C语言，二级指针可以绕过不带头结点链表删除操作的边界检查。

2018-10-05



hope

👍 3

看完了，打卡，稍后手写作业，去GitHub上看了下，希望老师把c的代码也添加上，谢谢

2018-10-05

作者回复

要不你写下 提个pull request?

2018-10-05



失火的夏天

👍 3

1.三个节点`p.pre`, `p`, `p.next`, 将`p`的`next`指针指向`p.pre`, 然后`p.pre=p`, `p=p.next`, `p.next=p.next.next`移动指针，就可以实现单链表反转。

2.最简单就是一个节点在头，一个节点一直遍历，地址相等就是环，不过好像还有一种简单的办法，快慢前进，一次就能搞定。这个老师能不能说下自己的思路，我有点想不明白。

3.建立第三个链表，每次比较`a`链表当前节点和`b`链表当前节点的大小。如果`a`比`b`小，则`c`的`next`指针指向`a`当前节点，`c=c.next`, 然后`a`指针后移。接着继续比较`a.b`当前节点大小，反之则把`a`换成`b`就行了。

4.一个`p`节点，然后找到距离`p`有`n`个`next`节点的点，一起往后遍历，到`pn.next`为空的时候，`p`就是我们要求的那个地址。

5.快慢指针，一个每次前进2个节点一个每次前进1节点。前进两个节点到表尾的时候，前进一个的就是中间点。

2018-10-05



过些天再换个名字 现在想不出来...

👍 2

代码二示例返回值`int`是不是写成`inf`了哈哈哈

算法设计思路应该是

// 用来找出给定`key`在数组中的下标，找不到则返回-1

`a`是被遍历的数组

`n`是数组长度

`key`是要寻找的值

- 1, 判断尾节点是不是要寻找的值, 是的话返回 $n-1$, 因为数组下标从0开始所以要长度-1才是下标
- 2, 使用哨兵变量保存尾节点
- 3, 把key放到尾节点, 让key成为数组中最后一个值, 这样做是为了下一步的遍历
- 4, 开始从头开始遍历数组, i 为数组下标, 如果找到与key相等的元素则退出遍历, 否则遍历整个数组
- 5, 如果 i 是尾节点下标, 说明没有找到key, 如果不是则 i 为寻找的节点下标, 返回 i
- 6, 把哨兵变量还原赋值到数组尾节点, 也就是还原数组

也就是说平时用的临时变量就是哨兵变量

2018-10-06



广进

👍 2

作为一个小白, 每节课都有看不懂的, 这次又来了, 那个代码二, 从while往下就不懂了, 怎么感觉和一的功能不一样了。求指导。

还有您都觉得二可读性差了, 加点注释照顾照顾我们这些小白呀。]]

2018-10-05

作者回复

不好意思 我以后多加点注释 不过两段代码的功能是一样的

2018-10-08



Richard Zhong

👍 2

深入浅出, 太赞了

2018-10-05



optvxq

👍 1

哨兵可以理解为其可以减少特殊情况的判断, 比如判空, 比如判越界, 比如减少链表插入删除中对空链表的判断, 比如例子中对越界的判断。

空与越界可以认为是小概率情况, 所以代码每一次操作都走一遍判断, 在大部分情况下都会是多余的。

哨兵的巧妙就是提前将这种情况去除, 比如给一个哨兵结点, 以及将key赋值给数组末元素, 让数组遍历不用判断越界也可以因为相等停下来。

使用哨兵的指导思想应该是将小概率需要的判断先提前扼杀, 比如提前给他一个值让他不为null, 或者提前预设值, 或者多态的时候提前给个空实现, 然后在每一次操作中不必再判断以增加效率。

2018-10-10



鲫鱼

👍 1

快哭了, 跨专业学习, 就自学了一点python。都不知道要怎么去理解了]]
但是还是能理解一点的, 慢慢坑了

2018-10-09

作者回复

买本大话数据结构或者算法图解结合着看吧 这门课本身就比较难学 只能多花点时间了呢

2018-10-09

詩揚

👍 1

/**

```
public class Node {  
    public char c;  
    public Node next;
```

```
    public Node(char c) {  
        this.c = c;  
    }  
}  
**/
```

```
public static Node reverse(Node head) {  
    if(head == null || head.next == null) {  
        return head;  
    }
```

```
    Node prev = null;  
    Node cur = head;  
    Node next = head.next;
```

```
    while(next != null) {  
        cur.next = prev;  
        prev = cur;  
        cur = next;  
        next = cur.next;  
    }  
    cur.next = prev;  
    return cur;  
}
```

```
public static boolean existsCircle(Node head) {  
    Node slow = head;  
    Node fast = head;  
    while(fast != null && fast.next != null) {  
        slow = slow.next;  
        fast = fast.next.next;  
        if(slow == fast) {  
            return true;  
        }  
    }  
    return false;  
}
```

```
public static Node merge(Node head1, Node head2) {
```

```
    Node guard = new Node('/');  
    Node cur = guard;
```

```
    while(head1 != null && head2 != null) {
```

```

if(head1.c <= head2.c) {
while(head1 != null && head1.c <= head2.c) {
cur.next = head1;
cur = cur.next;
head1 = head1.next;

}
} else {
while(head2 != null && head1.c > head2.c) {
cur.next = head2;
cur = cur.next;
head2 = head2.next;

}
}

if(head1 != null) {
cur.next = head1;
}
if(head2 != null) {
cur.next = head2;
}

return guard.next;

}

public static Node deleteLastN(Node head, int n) {
if(n < 1 || head == null) {
return head;
}
Node guard = new Node("/");
guard.next = head;

Node slow = guard;
Node fast = guard;

for(int i = 0; i < n; i++) {
if(fast != null) {
fast = fast.next;
}
}
while(fast != null && fast.next != null) {
slow = slow.next;
fast = fast.next;
}
slow.next = slow.next.next;
return guard.next;
}

```

```
public static Node getMiddle(Node head, int n) {  
    Node slow = head;  
    Node fast = head;  
  
    while(fast.next != null && fast.next.next != null) {  
        slow = slow.next;  
        fast = fast.next.next;  
    }  
  
    return slow;  
}
```

2018-10-09



观弈道人

👍 1

接上个反应的测试问题：我的测试代码是这样的。

```
Node node4 = new Node(4, null);  
Node node3 = new Node(3, node4);  
Node node2 = new Node(2, node3);  
Node node1 = new Node(1, node2);
```

```
deleteLastKth(node1, 4);  
测试结果为： 1、2、3、4
```

2018-10-06

作者回复

你怎么打印的结果？删除之后会返回新的链表的头指针，你要用那个头指针遍历打印的！

2018-10-07



观弈道人

👍 1

测试提供的源码，删除倒数第k个节点，感觉有点问题：测试数据为，1->2->3->4,删除0, 1, 2, 3, 4的结果分别为：1,2,3;1,2,3;1,2,4;1,3,4;1,2,3,4

2018-10-06

作者回复

k的边界条件忘了写，如果k<=0 直接return。你说的k=4的时候删除的结果不对，这个我去验证了下，跟你的结果不一样啊，我测出来是正确的呢。

2018-10-06