



链滴

链表（下）：如何轻松写出正确的链表代码？

作者：[someone26671](#)

原文链接：<https://ld246.com/article/1547479918580>

来源网站：[链滴](#)

许可协议：[署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)

<p>本文章是从极客时间抄写的，仅仅是看不懂，抄了一遍，分享给大家。
© 版权归极客邦科技所有</p>

<hr>

<p>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，但是链表代码还是很费劲的。哈哈，的确是这样的！</p>

<p>想要写好链表代码并不是很容易的事儿，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”这几行写对的人不足10%。</p>

<p>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？</p>

<p>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整天时间，就去写链表反转这一个代码，多写即便，一直练到毫不费力地写出 Bug free 的代码。这次还会很难跨吗？</p>

<p>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。我据自己的学习经历和工作经验，总结出了几个写链表代码技巧。如果你能熟练掌握着几个技巧，加上你的主动和坚持，轻松拿下链表代码完全没有问题。</p>

<h2 id="技巧一-理解指针或引用的含义">技巧一：理解指针或引用的含义</h2>

<p>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。</p>

<p>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。不管事“指针”还是“引用”，实际上，它们的意思都是一样的，都是储所指对象的内存地址。</p>

<p>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关系，你把它理解成“引用”就可以了。</p>

<p>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：</p>

<p>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，指向了这个变变量，通过指针就能找到这个变量。</p>

<p>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</p>

<p>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 结点的 next 指针存储了 q 结点的内存地址。</p>

<p>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。</p>

<p>掌握了指针或者引用的概念，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了一步！</p>

<h2 id="技巧二-警惕指针丢失和内存泄露">技巧二：警惕指针丢失和内存泄露</h2>

<p>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。以，我们在写的时候，一定注意不要弄丢了指针。</p>

<p>指针旺旺都是怎么弄丢的呢？我拿单链表的插入操作为例来给你分析一下。</p>

<p></p>

<p>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。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。</p>

```
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight-cl"> 1 p-&gt;next = x; //将p的next指针指向x结点;</span></span><span class="highlight-line"><span class="highlight-cl"> 2 x-&gt;next = p &gt;next; //将x的结点的next指针指向b结点;</span></span></code></pre>
```

<p>初学者经常会在哪儿犯错。p->next 指针在完成第一步操作之后，已经不再指向结点 b 了，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，整个链表也就成了半，从结点 b 往后的所有结点都无法访问到了。</p>

<p>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

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，致内存泄露。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。

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，也会出内存泄露的问题。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。如果我们在结合 p 后面插入一个新的结点，需要下面两行代码就可以搞定。

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

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 }
```

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。

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

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 }
```


从前面一步一步分析，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，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。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

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。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。 </p>

<p>代码一: </p>

```
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight-cl"> 1 //在数组 a 中，查找key，返回key所在位置
</span></span><span class="highlight-line"><span class="highlight-cl"> 2 //其中， n表示
组 a 的长度
</span></span><span class="highlight-line"><span class="highlight-cl"> 3 int find(char* a,
nt n,char key){
</span></span><span class="highlight-line"><span class="highlight-cl"> 4 //边界条件处
， 如果a为空， 或者n<&lt;=0,说明数组中没有数据， 就不同while循环比较了
</span></span><span class="highlight-line"><span class="highlight-cl"> 5 if(a == null ||
&&lt;= 0){
</span></span><span class="highlight-line"><span class="highlight-cl"> 6     return -1;
</span></span><span class="highlight-line"><span class="highlight-cl"> 7 }
</span></span><span class="highlight-line"><span class="highlight-cl"> 8
</span></span><span class="highlight-line"><span class="highlight-cl"> 9 int i = 0;
</span></span><span class="highlight-line"><span class="highlight-cl">10 //这里有两个
较操作: i&&lt;n 和a[i]==key.
</span></span><span class="highlight-line"><span class="highlight-cl">11 while(i &&lt; n)

</span></span><span class="highlight-line"><span class="highlight-cl">12     if(a[i] == k
y){
</span></span><span class="highlight-line"><span class="highlight-cl">13         return i;
</span></span><span class="highlight-line"><span class="highlight-cl">14     }
</span></span><span class="highlight-line"><span class="highlight-cl">15     ++i;
</span></span><span class="highlight-line"><span class="highlight-cl">16 }
</span></span><span class="highlight-line"><span class="highlight-cl">17
</span></span><span class="highlight-line"><span class="highlight-cl">18 return -1;
</span></span><span class="highlight-line"><span class="highlight-cl">19 }
</span></span></code></pre>
```

<p>代码二: </p>

```
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight-cl"> 1 //在数组a中， 查找key， 返回key所在的位置
</span></span><span class="highlight-line"><span class="highlight-cl"> 2 //其中， n表示
组a的长度
</span></span><span class="highlight-line"><span class="highlight-cl"> 3 //我举2个例子
你可以拿例子走一下代码
</span></span><span class="highlight-line"><span class="highlight-cl"> 4 // a = {4,2,3,5,9
6} n=6 key = 7
</span></span><span class="highlight-line"><span class="highlight-cl"> 5 // a = {4,2,4,5,9
6} n=6 key = 6
</span></span><span class="highlight-line"><span class="highlight-cl"> 6 int find(char* a,
nt n,char key){
</span></span><span class="highlight-line"><span class="highlight-cl"> 7     if(a == null ||
n&&lt;=0){
</span></span><span class="highlight-line"><span class="highlight-cl"> 8         return -1;
</span></span><span class="highlight-line"><span class="highlight-cl"> 9     }
</span></span><span class="highlight-line"><span class="highlight-cl">10
</span></span><span class="highlight-line"><span class="highlight-cl">11 //这里因为
将 a[n-1]的值替换成key， 所以要特殊处理这个值
</span></span><span class="highlight-line"><span class="highlight-cl">12     if(a[n-1] ==
key){
```

```

</span></span><span class="highlight-line"><span class="highlight-cl">13      return n-1
</span></span><span class="highlight-line"><span class="highlight-cl">14    }
</span></span><span class="highlight-line"><span class="highlight-cl">15
</span></span><span class="highlight-line"><span class="highlight-cl">16    //把a[n-1]
值临时保存在变量tmp中，以便之后恢复。tmp=6。
</span></span><span class="highlight-line"><span class="highlight-cl">17    //之所以这
做的目的是：希望find()代码不要改变 a 数组中的内容
</span></span><span class="highlight-line"><span class="highlight-cl">18    char tmp =
[n-1];
</span></span><span class="highlight-line"><span class="highlight-cl">19    //把key的
放到a[n-1]中，此时a={4,2,3,5,9,7}
</span></span><span class="highlight-line"><span class="highlight-cl">20    a[n-1] = key
</span></span><span class="highlight-line"><span class="highlight-cl">21
</span></span><span class="highlight-line"><span class="highlight-cl">22    int i = 0;
</span></span><span class="highlight-line"><span class="highlight-cl">23    // while循
比起代码一，少了 i < n 这个比较操作
</span></span><span class="highlight-line"><span class="highlight-cl">24    while(a[i] !=
key){
</span></span><span class="highlight-line"><span class="highlight-cl">25        ++i;
</span></span><span class="highlight-line"><span class="highlight-cl">26    }
</span></span><span class="highlight-line"><span class="highlight-cl">27
</span></span><span class="highlight-line"><span class="highlight-cl">28    //恢复a[n-1
原来的值，此时a={4,2,3,5,9,6}
</span></span><span class="highlight-line"><span class="highlight-cl">29    a[n-1] = tmp
</span></span><span class="highlight-line"><span class="highlight-cl">30
</span></span><span class="highlight-line"><span class="highlight-cl">31    if(i == n-1){
</span></span><span class="highlight-line"><span class="highlight-cl">32        //如果 i=
n-1说明，在0...n-2之间都没有key，所以返回-1
</span></span><span class="highlight-line"><span class="highlight-cl">33        return -1;
</span></span><span class="highlight-line"><span class="highlight-cl">34    }else{
</span></span><span class="highlight-line"><span class="highlight-cl">35        //否则，
回i，就等于key的值的元素的下标
</span></span><span class="highlight-line"><span class="highlight-cl">36        return i;
</span></span><span class="highlight-line"><span class="highlight-cl">37    }
</span></span><span class="highlight-line"><span class="highlight-cl">38 }
</span></span></code></pre>

```

对于两段代码，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、几十万时，累积的时间就很明显了。

当然，这知识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 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？
- 如果链表只含一个结点是，代码是否能正常工作？

- 如果链表只包含两个结点是，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？

<p>当你写完链表代码之后，除了看下你写的代码在正常情况下能否工作，还要看下载上面我列举的个边界条件下，代码仍然能否正常工作。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了。 </p>

<p>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。 </p>

<p>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你或者异常情况。遇到了如何应对，这样写出来的代码才够健壮！ </p>

技巧五：举例画图，辅助思考</h2>

<p>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。 </p>

<p>你可以找一个具体的例子，把它画在纸上，释放一些脑容量，留更多给逻辑思考，这样就会感觉思路清晰很多。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，出插入前和插入后的链表变化，如图所示： </p>

<p></p>

<p>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，着代码走一遍，很容易就能发现代码中的 Bug。 </p>

技巧六：多写多练，没有捷径</h2>

<p>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，也要着急。因为我最开始学的时候，这种状况也持续了一段内装饰件。 </p>

<p>现在我写这些代码，简直就和“玩儿”一样，其实也没什么技巧，就是把常见的链表操作都自多写即便，出问题就一点一点调试，熟能生巧！ </p>

<p>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，我保你之后在也不会害怕写链表代码。 </p>

- 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</h2>

<p>这节我主要和你讲了写出正确链表代码的六个技巧。本别是理解指针或引用的含义、警惕指针丢和内存泄露、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，还有多写练。 </p>

<p>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。链表代码写的好坏，可以看出一个人写代码否够细心，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。所以，这一节讲到的东西，你一定要写代码实现一下，才有效果。 </p>

课后思考</h2>

<p>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难度？ </p>

<p>欢迎留言和我分享，我会第一时间给你反馈。 </p>

<hr>

<p>我已将本节内容相关的详细代码更新到 GitHub，戳此即可查看。</p>

<p></p>

abf14d3jpg" data-src="https://b3logfile.com/file/2019/01/8e603e3d795fc0ab2698f6f5eabf143-d81ebf76.jpg?imageView2/2/interlace/1/format/jpg"></p>