HZ BOOKS
华章科技

世界知名IT企业程序员技术面试经典问题解析
涵盖大数据、云计算、移动开发等热点领域
技术实力和软实力同步提升
轻松有趣的学习方式，助力程序员实现职业梦想

程序员生存手册

面试篇

Programmer's Survival Handbook

Ricky Sun（孙宇熙）编著

机械工业出版社
China Machine Press

程序员生存手册

面试篇

Programmer's Survival Handbook

Ricky Sun（孙宇熙）编著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

程序员生存手册：面试篇 / 孙宇熙编著 . —北京：机械工业出版社，2016.1

ISBN 978-7-111-52441-0

I. 程… II. 孙… III. 程序设计 – 技术手册 IV. TP311.1-62

中国版本图书馆 CIP 数据核字（2016）第 003443 号

程序员生存手册：面试篇

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：曲 熠　　责任校对：殷 虹

印　　刷：中国电影出版社印刷厂　　版　　次：2016 年 1 月第 1 版第 1 次印刷

开　　本：185mm × 260mm 1/16　　印　　张：18.5

书　　号：ISBN 978-7-111-52441-0　　定　　价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88379426 88361066　　投稿热线：（010）88379604

购书热线：（010）68326294 88379649 68995259　　读者信箱：hzit@hzbook.com

华章科技
HZBOOKS | Science & Technology

前 言

上个世纪末，硅谷的高科技产业（Hi-Tech Industry）发展得如火如荼，整个硅谷呈现出一派欣欣向荣的景象，每天都有新兴的公司涌现，各种各样的招聘活动风起云涌，每个创新型公司[⊖]在人力资源投资上的重头戏无一例外都是程序员（在“码农”或“DevOps”这些新新人类的词汇诞生之前，我们还保留严格意义上的程序员这一荣誉称号吧）。

说到程序员招聘，它和其他所有职位最大的不同就是技术面试，“会写代码才是王道”在相当长的时间里都是程序员这个行当里颠扑不破的真理。整个招聘流程中对编程、代码调试、编译、优化、架构理解与设计等技术能力的考查贯穿始终，甚至从 HR（或招聘经理）最初联系你时便已开始。记得当年在 Santa Clara Convention Center（会议中心）参加一次硅谷招聘大会时，逛到一个展台前，被一美国大姐问道：“What's your favorite language？”（你最喜欢什么语言？）我没过脑子就回答道：“C、C++”。在以硅谷为背景的对话中，语言 = 编程语言，俨然是个约定俗成的事儿。

程序员面试中，除了有那么几个著名的公司喜欢对大学毕业生们提一些脑筋急转弯类的问题（比如：读完“Stanford 大学”需要多久？答案：一秒足矣）以外，绝大多数还是相当简朴、实在的。面试的形式可能多种多样，包括电话约谈（phone screen）、面谈（这类最简单、最轻松）、笔试（过不了这关就甭想进入下一关）、上机（实战，这类面试虽然不多，但也是不得不防的）、做主题报告（这是相当学院派的做法，不过很多研究院、CTO 性质的机构会常用）等。本书的内容基本上覆盖了以上列出的所有面试形式中常见的问题并提供了较为详尽的解答，值得指出的是很多问题答案并不唯一，特别是设计类的问题，几乎没有标准、统一的正确答案。面试所要检验或者观察的是你在遇到问题时的第一反应、思维逻辑、思考方法、解决问题的方式、偏好等，面试官将综合这些因素来评估你在未来的团队中的位置和融入的可能性。

本书的内容可以说是我和很多志趣相投的同事、朋友对过去十几年（甚至几十年）中的亲身经历与收集的大量面试考题的整理、分类、提纯再细化。从地域而言，跨越了从美国到中国的大大小小上百家公司；从内容上看，既涵盖了数据结构、算法、操作系统、网络等基础类知识的面试题，也兼顾了最近几年关注度最高的移动编程（如 Android）、云计算与大数据等内容。与传统的教科书不同，本书的目的是通过真实面试题的 Q&A 分析，让读者重温在书本中可能漏掉的重点，换一个甚至多个角度去思考问题的解决之道。因为工业界的解决

⊖ 硅谷里面有不以创新为导向的公司吗？我的答案是：好像没有。那么创新型这个修饰词其实也显得冗余了。

方案与学术界往往大相径庭，但有时却又一脉相承。

要想了解从学校过渡到工业界的跨度，不妨来看两个简单的例子。

例子 1 在数据结构中，最美妙的大概就是递归算法的使用。使用递归的代码看起来非常简短（虽然逻辑解读上可能更复杂），而且应用的地方很多，比如著名的链表反转（reversing a link-list）。遥想当年大一的时候一群人在 DEC-8 终端机上奋笔疾书写出那些自以为惊世骇俗的递归程序时，没人思考过递归会占用更多的内存（context saving），其效率也相对更差（调用前的内存准备、占用和调用后的内存释放），在面对长链表的情况下堆栈溢出也是潜在问题。在业界的真实应用中，单纯的递归几乎是不存在的，取而代之的是循环（不占用额外的存储，堆栈溢出的危险有效降低，等等）或递归嵌套循环（并且时刻监视内存的使用及是否有内存泄露等风险）。

例子 2 再来看一个更接地气的问题。有一个大型的无线网络，其中几千台无线热点部署在城市的热点地区。管理员发现其中 80% 的同一型号的 AP 会不定期出现死机的现象，一旦发生这样的现象，远程无法访问，也无法自动重启，只能人工重新在本地断电重启。由于所有的 AP 均为企业级，有日志定期上传到管理服务器，因此分析得出此类 AP 会在频繁使用的 24 ~ 48 小时内出现上述死机现象。这个问题极度困扰网管部门，如何解决？

除了统一升级固件，还有其他办法吗？我们知道，提供服务的网络设备（如无线 AP）通常会提供远程访问接口，常见的有 SNMP、Telnet、TFTP，高级的有 SSH（更像一台 *nix 服务器了）。TFTP 有的会被用来上传固件；SNMP 多数只提供读，支持写的比较少，这一批 AP 也不支持 SNMP 写；最后就只剩下 Telnet 接口了。传统意义上远程 Telnet 登录并执行 reboot 操作即可，现在的问题是如何自动登录几千台 AP？经过上面的分析，这个问题已经简化为：写一个 UserAgent 来自动 Telnet 登录，把它放到一个大的循环当中访问几千台 AP，而自动登录的时间选择在凌晨 3 点到 4 点，因为几乎没有客户会在这个时间段登录。再完善一点的解决方案是先通过 SNMP 或 Telnet 判断是否有活跃链接用户，等待其下线再重启，这有点像 Apache Web 服务器的平滑重启（graceful restart）。再进一步考量的是发起大规模 Telnet 登录请求的主机用何种方法来完成工作：循环、多进程、多线程、非阻塞 I/O（Non-blocking I/O），以及其他考虑因素，如异常处理等。

通过上面两个简单的例子，希望大家可以感受真实环境下我们在面对问题、分析问题并寻找解决之道时是如何因地制宜的。最简单的方法不一定是最好的，但能解决客户问题的方法一定是好的（good enough solution that makes sense）。

本书并不是一本超级严肃、充满条条框框的教科书，你可以把它看作一本面试参考书，也可以闲来翻翻作为了解 IT 发展史的课外书。本书的内容中虽然有绝大部分是纯技术的，但是也有软技能相关的章节。总之，一个优秀的技术人员不仅要有过硬的技术背景，也要有一定的软技能（待人接物）；要有常识，要能融入团队，这样你才能走得更远、变得更强。

不知道有没有人统计过一个程序员在其技术生涯中要学习、掌握多少种技能（语言、系统、环境与流程）。我个人感觉，摩尔（Moore）定律也适用于这个问题，平均每 1.5 年就会有一整套新的颠覆性技术出现，如果不想被淘汰，就不得不迎头赶上。对于热爱编程的你而言，本书覆盖了 6 种典型的计算机语言。鉴于篇幅所限，我并没有刻意搜集更多的编程语言，因

为我相信触类旁通，掌握一门语言，熟悉它、灵活运用它，再去攻克另一门语言是一件很简单的事情（有时候甚至就是半个下午的事情）。以一个科班出身（CS 专业）IT 老兵的角度看，对于计算机语言的接触与掌握可经历如下阶段：C → C++/Java → Perl/PHP/SQL 或 Android/iOS。C 是基础，数据结构、操作系统、网络编程都是以 C 为范例讲解的；C++/Java 是 OO 的两大代表性语言，C++ 更硬核（hardcore）一些，而 Java 这几十年经久不衰自然有它的过人之处；Perl 是脚本语言中集大成者（PHP、Python、Ruby、JavaScript 这些统统可以看作 Perl 的“小弟”兼衍生品。严格意义上说，Perl 源自 Shell Scripting，像 sed、awk、sh、csh、bash 等不一而足，但是集大成者当属 Perl，由于篇幅及受众所限，本书没有为 Shell 编程单独开辟章节）；SQL 是一门很人性化的语言，在大数据备受关注的今天，熟练掌握 SQL 绝对不会让你落伍；Android/iOS 分别可以看作 Java 与 C++ 的变种……总而言之，从 C 开始熟悉基本概念、夯实基础，会让你走得更远、更好！

最近几年，随着云计算和大数据的风起云涌，很多问题已经不再是简单（或直接）的编码问题，而更侧重于系统的体系架构设计、逻辑分析、优劣选择或方案折中；同时随着移动互联网的飞速发展，新的编程语言不断涌现，本书也选取了一些具有代表性的 Q&A 与大家分享，希望对大家的工作与学习有所助益。

另外，需要说明的是，本书的每一个 Q&A 都可以独立成文，虽然前后的数个 Q&A 可能会有知识点上的关联性，但是从阅读、查询、恶补或是作为面试问题的角度上来看并不存在相互依赖性。书中一些重要的 Q&A 配有作者录制的讲解视频，读者可扫码进行观看。每章后还附有该章内容相关的基础知识的 MOOC 的二维码，需要的读者可扫码进入学习。

孙宇熙

2015 年 9 月

致 谢

从最早的Q&A收集整理到最终集结成书，时间跨度足足有十几年。最早的版本是笔者本人的日常英文笔记（都是在历次技术面试中凭记忆记录下来的经典问题），涵盖了100多道硅谷IT公司典型的程序员面试技术问题，后来尝试着分享给身边的三五好友，颇得好评。在实战中，这本小册子不断被验证，之后又有很多同事和朋友陆续补充了一些题目。一来二往，小册子变成了厚厚的大册子（超过500道问题，涉猎的技术领域、问题类型也更加广泛）。

在本书付梓之际，需要感谢的人很多。按照华人的传统，首先要感谢父母，感谢他们对我一如既往的关怀、支持与理解；其次要感谢我的家人，没有你们的理解与宽容，我不可能有大把的周末时光把自己关在书房中埋头码字（同时也有了合理的借口可以不参与周末家庭大扫除）。

感谢在过去多年内提供了很多Q&A来充实和丰富本书的朋友和同事们，包括：我的大学同学、PaloAlto Networks技术副总裁Wilson Xu，前Yahoo!的高级工程师Stephanie Xu，Facebook早期员工Eric Ji，我的大学同学、卡内基梅隆大学高材生张成，我的前微软同事周健博士、Neo Han Chen，我的前合伙人和老朋友、硅谷硬件孵化器创始人Hilbert Ming Guo，我的EMC同事、技术总监Michelle Lei女士，我在EMC研究院的大数据与云计算的同事和专家曹逾博士、李三平博士、董哲、郭晓燕、陈曦，还有那些曾经在面试中折磨过我也被我折磨过的人们，在此一并感谢。

特别需要感谢的还有机械工业出版社的朱劼女士和我的同事朱捷先生以及开课吧团队同仁，没有他们的积极策划与耐心帮助，本书不会成为现实。

孙宇熙

2015年2月28日深夜于硅谷米深堂

目 录

第四篇 软技能篇

第一篇 Part 1

基 础 篇

- 第 1 章　数据结构
- 第 2 章　算法与优化
- 第 3 章　操作系统
- 第 4 章　面向对象

第 1 章

数据结构

关于数据结构的重要性，Eric S. Raymond（ESR）在其所著的那本广为人知的软件开源运动圣经——《大教堂与集市》[⊖]一书中有精辟的阐述：

> Smart data structure and dumb code work a lot better than the other way around.（聪明的数据结构配上傻瓜代码要好过聪明的代码配傻瓜的数据结构。）

笔者以为这句话一语中的！无数的经验告诉我们：解决问题的不二法门是数据结构 + 算法，好的数据结构再匹配上高效的算法几乎可以无往而不利。

在本章中，我们会通过经典的面试题来回顾各类数据结构。

1.1 链表

如果有人问我面试中最常见的问题是哪一类，我一定会告诉你非链表莫属。下面这个问题是最能展示一个程序员的基本功和实力的。因为这类问题没有标准答案，而且链表这东西在逻辑上的确比较绕，但这恰恰是考察你学以致用的能力或真才实干最好的试金石。

如何翻转一个单向链表（头变尾、尾变头）？

笔者认为这道面试题是程序员面试中出现频度最高的问题，没有之一。解决方法绝对不止一种，不过大多数学院派的同学最熟悉的就是用当年在 C 语言和数据结构课上学到的递归算法来解决了。让我们重温一下经典的解决方案：

```
// Recursively, In case you forgot.(递归解法)
reverse_ll(struct node ** hashref) {
    struct node * first,  last;
    if(*hashref == NULL) return -1;
    first = *hashref;
    rest = first->next;
    if(rest == NULL) return;

    reverse_ll(&rest);

    first->next->next = first;  //reversion happens.(反转)
    first->next = NULL;
} //-END-
```

⊖ 该书已由机械工业出版社引进出版，书号为 978-7-111-45247-8。——编辑注

上面的递归代码中，核心部分只有几行，其他都是边界条件（boundry condition）检查的部分（潜台词：却也是最重要的，正所谓不怕一万就怕万一）。

但是，很重要的但是，递归程序虽然简短，但依然有代价。递归的效率并不高，每一次调用自己都产生大量额外内存占用（与清除）。对于长链表，发生堆栈溢出是绝对有可能的，更优质的解决方案（面试官想看到的）是采用非递归。通常如果面试者上来就使用递归方案，循循善诱的面试官会追问：还有没有不使用额外内存的解决方案？

这时，如果你能给出下面的方案，面试官一定会对你刮目相看。这个解决方案使用了一个相当简单的循环，只使用了一个临时指针在循环中调整指针的指向，将头变尾、尾变头。在工业界，大多数面试官期待的是这种基于循环的答案。

```
// Non-Recursively
Node *p  = head;          // h points to head of the linklist.(指向链表头)
If( p == NULL)
    return NULL;
link *h;
h = p;
if(p->next)
    p = p->next;
h->next = NULL;    // h is now an isolated node which will be the tail node
eventually.(h现在是孤立结点，最终会成为尾结点)
while( p != NULL) {
    link *t = p->next;    // tmp node.
    p->next = h;          // reverse the linkage
    h = p;                // moving one node forward
    p = t;                // moving forward
}
return h;  //h points to the new Head.(h指向新表头)
//-END-
```

思考：为什么递归有风险？

答：有可能出现堆栈溢出，特别是对于大数据集或过度使用嵌套递归的情况。

如何反向打印一个链表（从尾到头）？

和上一题异曲同工的就是这一道题了，硅谷历史上赫赫有名的公司在面试时都相当钟爱这个问题，比如 Yahoo! 和 Netscreen。后一个公司名字可能很多读者不太熟悉，但 Juniper Networks 肯定听说过吧？在网络公司里抗衡 Cisco，在高端路由器与防火墙领域所向披靡，当年 40 亿美金收购了 Netscreen，在那之前 Netscreen 在 Nasdaq 上市，表现相当耀眼。

还是先给出递归解决方案，超简单（不过效率就堪忧了）：

```
rev_ll (Node *h) {
    If(!h) return(-1) ;
    else { rev_ll(h->next); print(h->data);}
}
```

如果采用非递归的方法，类似于上题，先把链表反转，然后从新的头节点开始打印到尾部。从算法复杂度上看大约相当于 $O(2*N)$。

和本题类似的一个问题是：如何反转一个字符串（string）（要求复杂度为 $O(N/2)$）？

答案其实很简单。以字符串数组类型为例，将第一与倒数第一置换、第二与倒数第二置换，一直到自己（subscript= 脚注、下标）与自己相等则停止置换。

如何翻转双向链表？

双向链表的翻转其实比单链表的翻转简单，道理不用多解释，直接就用循环吧。期间只需要做两件事情：头尾置换、向前指针与向后指针的互换。

```
Node *PCurrent = pHead, *pTemp;
While(pCurrent) {
    pTemp = pCurrent->next;
    pCurrent->next = pCurrent->prev;
    pCurrent->prev = pTemp;
    pHead = pCurrent;
    pCurrent = pTemp;
}
// pHead will point to the newly reversed head by now.
// -END-
```

其实仔细想来，这道题有点无聊，双向链表翻转意义何在呢？关键在于面试官要通过这个题目来检验的是你的逻辑思维能力。重申一下，链表有些绕。

在一个按升序排列的双向量表中，如何插入一个节点？

好吧，我们再接再厉，看看怎样在一个链表当中插入一个节点。这道题考察的绝不是算法复杂性，而是对边界条件的检查，这是一个优秀的程序员必备的能力。

先想象一下可能在哪些地方插入一个节点：头部、中间、尾部。现在，让我们在代码中付诸实践吧！

```
//Assume we have a struct:
Struct Node {
    Int data;
    Node * prev;
    Node *next;
};

//pHead, pCur, nn (new node), ppCur (pre-pCur)
if(pHead == NULL) return(0);
pCur = pHead;
while(pCur) {
    if(pCur != pHead) {
        ppCur = pCur->prev;  //keep track of prev node.
    }
    if(pCur->data >= nn->data) {
```

```
        if(pCur == pHead) {  // insert at the head
            pHead = nn;
            nn->prev = NULL;
            nn->next = pCur;
            pCur->prev = nn;
        } else { // insert at non head.
            if(pCur->next == NULL) { // insert at the tail.
                nn->next = NULL;
                pCur->next = nn;
                nn->prev = pCur;
            } else { // insert somewhere in the middle.
                nn->next = pCur;
                pCur->prev = nn;
                ppCur->next = nn;
                nn->prev = ppCur;
            }
            return(pHead);  // return head of double-linklist.
        }
    } else { // keep going!
        pCur = pCur->next;
    }
} // end of while()
```

在双向链表中，如何删除一个节点？

既然学会了如何插入新节点，那么删除一个节点也不是什么大问题了吧？这个问题也是出自名门，比如 Frontier 的现场（onsite）上机面试题，给你一个键盘和一台 14 寸的 CRT 显示器，几分钟内写出可以正确编译执行的代码。Frontier 的团队相当一部分是从 NetScreen 出来的，包括它的创始人也是原 NetScreen 的联合创始人，所以面试虽然难了一点，不过只要准备充分，这道题不在话下。

```
// Need to consider all boundary conditions.
void del(node *n)
{
    if (!n) return;
    if (n->prev) {  // Treat one direction.
        n->prev->next = n->next;
    } else {
        n->next->prev = NULL; //head deletion
        head = n->next; //reassign list head!
    }

    if (n->next) {  // Treat the other direction.
        n->next->prev = n->prev;
    } else {
        n->prev->next = NULL; // delete at tail!
    }

    delete (n);
```

```
} //END of del()
```

在单链表中如何发现 loop（环）？

前面几个问题变着花样考查的是链表 manipulation 的问题。我们再来点 tricky 的，比如下面这个 Yahoo! 曾经很喜欢问的问题：如何在单链表中侦测到循环。

这里提供一种解决方案（当然不止一种）。发现是否有环的基本算法就是从表头出发，两个指针在循环中以不同的步幅前进，其中 p1 步幅为 1，p2 步幅为 2，如果在某次迭代中指向同一点，则说明出现环（loop）。

```
Node *p1, *p2, *head;
p1 = p2 = head;
do {
    p1 = p1->next;
    p2 = p2->next->next;
} while ( p1 != p2 && p1->next != NULL && p2->next != NULL && p2->next->next != NULL )

if ( p1->next == NULL || p2->next == NULL || p2->next->next == NULL) {
    printf( "None loop found!\n" ;
} else if ( p1 == p2 && p1 != NULL ) {
    printf( "Loop found!\n" );
} // -END-
```

小插曲：十余年前的那个夏天，Yahoo! 硅谷总部，笔者与搜索部门的技术总监遭遇，信心满满地写完以上的代码，总监又抛出了一个“连环炮”——指针 p1 与 p2 会陷入无限死循环的黑洞吗？在本人年轻气盛的时候，这种问题是立刻会反驳的（在经过逻辑分析后），单链表中两个不同步幅的指针会陷入死循环？不可能，绝对不可能！很显然总监没有料到有面试者会这么直截了当地否定他的命题。结果当然是笔者没有进入 Yahoo! 的搜索部门（笔者随后阴差阳错地加入了 Yahoo! SDS 战略数据服务部门，今天我们更愿意把类似的工作叫大数据处理与分析）。

本着发扬刨根问底精神的原则，我们用下面两幅图（见图 1-1、图 1-2）来说明为什么即使链表中有环存在，以上的代码也不会陷入死循环。

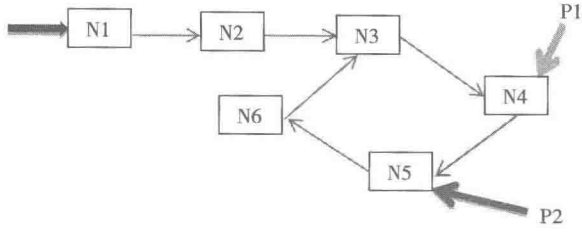

图 1-1　单链表中的环

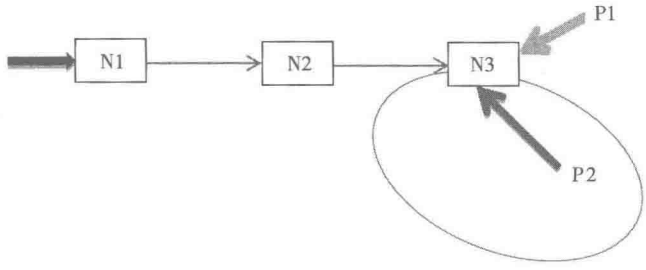

图 1-2　单链表中的环特例

- 图 1-1 中，N6 会指向 N3 从而形成环，但是 p1、p2 会在从 N3 开始的 4 次循环内汇合，从而做出有环的判断。
- 图 1-2 是一种特例，N3 自己指向自己，这种情况下，只要 1 次循环就会让 p1 和 p2 碰撞！
- 理论上只有一种可能在即便有环的情况下，让 p1、p2 一经出发就很难再相遇，那就是链表无限长，表尾指向表头。不过这种情况下，用链表作为数据存储结构，然后还要试图找到循环，本身就很无聊。通过一个哈希表来跟踪一个节点是否被多个同链表节点指向可能更容易发现问题、更便于规避风险。

如何在一次遍历中找到一个链表中从尾部算起的第 N 个节点？

这个问题可以说是一个智力测验，最关键是先想到一个简单的算法，能在一次遍历中就可以寻址到链表中的第 N 个元素。很明显，我们需要借助两个指针，所谓一个巴掌拍不响，至少要有两个指针才能帮助我们丈量这个距离。

```
// Utilize 2 pointers, keep them about N-node pace away.
if (head == NULL) return(0);
ptr2=ptr1 = head;
int i=0;
for(i=1; i<=n-1; i++)   // have ptr1 march (N-1) nodes
{
    if(ptr1->next == NULL) return(0);
    ptr1 = ptr1->next;
}
while (ptr1 != NULL)
{
    if(ptr1->next == null) { // found
        return(ptr2)
    }
    ptr1 = ptr1->next;
    ptr2 = ptr2->next;
}
```

如何打印二叉树？要求从树头开始，一层接一层地打印。

这一问题可以简单归纳为二叉树遍历，逐层（level-by-level）的遍历是典型的广度优先