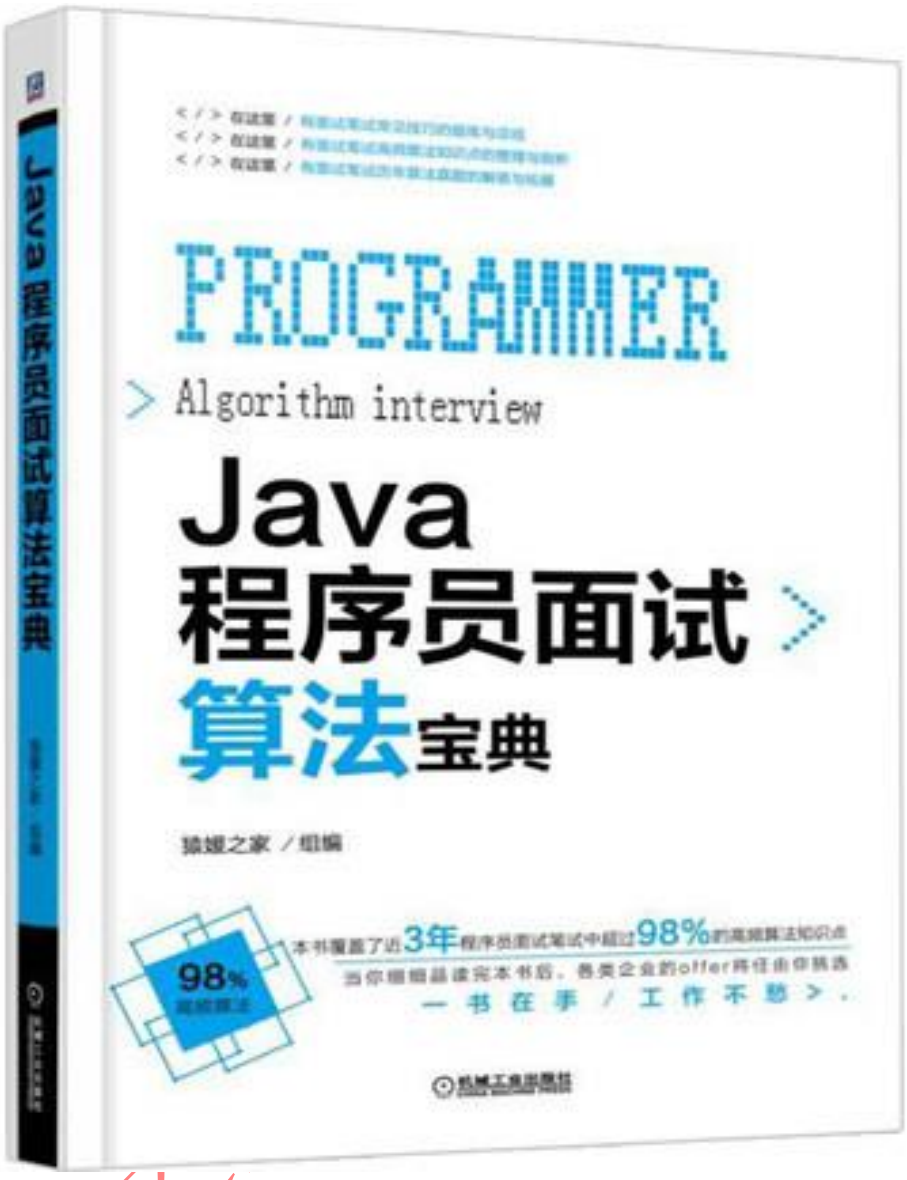


本书已在京东、淘宝、当当各大电商平台有售，支持正版，支持作者劳动成功。

一书在手，工作不愁。

程序员最可信赖的求职帮手

Java 程序员面试算法宝典

猿媛之家　组编

机 械 工 业 出 版 社

本书是一本讲解 Java 程序员面试算法的书籍，在写法上，除了讲解如何解答算法问题外，还引入了实例辅以说明，让读者能够更好地理解本书内容。

本书将 Java 程序员面试、笔试过程中各类算法类真题一网打尽。在题目的广度上，本书收集了近三年来几乎所有 IT 企业面试、笔试算法高频题目，所选择题目均为企业招聘使用题目。在题目的深度上，本书由浅入深，庖丁解牛式地分析每一个题目，并提炼归纳。同时，引入实例与源代码、时间复杂度与空间复杂度的分析，而这些内容是其他同类书籍所没有的。本书根据真题所属知识点进行分门别类，力图做到结构合理、条理清晰，对于读者进行学习与检索意义重大。

本书是一本计算机相关专业毕业生面试、笔试的求职用书，也可以作为本科生、研究生学习数据结构与算法的辅导书，同时也适合期望在计算机软、硬件行业大显身手的计算机爱好者阅读。

图书在版编目（CIP）数据

Java 程序员面试算法宝典 / 猿媛之家组编. —北京：机械工业出版社，2018.7
ISBN 978-7-111-60395-5

Ⅰ. ①J… Ⅱ. ①猿… Ⅲ. ①JAVA 语言—程序设计 Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字（2018）第 149932 号

机械工业出版社（北京市百万庄大街 22 号　邮政编码 100037）
策划编辑：时　静　责任编辑：时　静　王　荣
责任校对：张艳霞　责任印制：

印刷（　　装订）

2018 年 7 月第 1 版 • 第 1 次印刷
184mm×260mm • 18.75 印张 • 456 千字
0001—　册
标准书号：ISBN 978-7-111-60395-5
定价：　元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务　　网络服务
服务咨询热线：010-88361066　机 工 官 网：www.cmpbook.com
读者购书热线：010-68326294　机 工 官 博：weibo.com/cmp1952
　　010-88379203　金　书　网：www.golden-book.com
封面无防伪标均为盗版　　教育服务网：www.cmpedu.com

前　言

程序员，一类听起来神秘看上去却普普通通的人群，写着那些密密麻麻的常人看不懂的代码，领着一份让很多人都羡慕的薪水。也许每个人心中对程序员的理解都不尽相同，也许每个程序员的生活状态各有千秋，但毋庸置疑，在程序员是否是一个高大上的职业上，他们还是惊人的一致：是。

在信息技术高度发达的今天，很难想象，离开了程序员，这个社会会是什么样子。从手机、计算机、电视机、洗衣机等日常生活用品到宇宙飞船、空间站、飞机、坦克、大炮，都离不开计算机程序。计算机程序已然渗透到生活中的方方面面。虽然计算机程序是一个软实体，看不见摸不着，人们也感知不到它的存在，但是你们是否知道，没有了操作系统与应用程序，手机就是一个废物；没有了计算机程序，大炮、飞机、航空母舰就是一个铁疙瘩；没有了计算机程序，洗衣机、空调就是一个摆设。

计算机程序有多么重要，也许上面说的还不清楚，在此举两个简单例子就可以说明。当前，智能手机高度发展，但是手机的 Android 操作系统是由软件巨头谷歌（Google）研发的，所有使用 Android 系统的手机厂商都受制于 Google。诺基亚（Nokia）手机曾经是世界上销量非常好的手机，由于没能跟上 Android 的步伐，最终走向了覆灭，被另一家软件巨头微软（Microsoft）公司收购了。苹果（Apple）智能设备之所以很受欢迎，除了其良好的外形设计以外，另一个重要原因就是其苹果操作系统（iOS）的独一无二。这些都是软件打败硬件的例子。

计算机技术博大精深，而且日新月异，Hadoop、GPU 计算、移动互联网、模式匹配、图像识别、神经网络、蚁群算法、大数据、机器学习、人工智能、深度学习等新技术让人眼花缭乱，稍有不慎，就会被时代所抛弃。于是，很多 IT 从业者就开始困惑了，不知道从何学起，到底什么才是计算机技术的基石。其实，究其本质还是最基础的数据结构与算法知识：Hash、动态规划、分治、排序、查找等，所以无论是世界级的大型 IT 企业还是小公司，在面试求职者时，往往都会考察这些最基础的知识，无论研究方向是什么，这些基础知识都是必须熟练掌握的。

本书在写作风格上推陈出新，对于算法的讲解，不仅文字描述，更以示例佐证，让读者能够更好地读懂本书内容。为了能够写出精品书籍，我们对每一个技术问题，都反复推敲，与算法精英一起反复论证可行性；对文字，我们咬文嚼字、字斟句酌，所有这些付出，只为让读者能够对书中技术点放心，文字描述舒心。

市面上同类型书籍很多，但是，我们相信，我们能够写出更适合读者的高质量精品书籍。为了能够在有限的篇幅里面尽可能地全是“干货”，作者在选择题目上下了很大的工夫。首先，我们通过搜集近三年来几乎所有 IT 企业的面试、笔试算法真题，包括已经出版的其他著作、技术博客、在线编码平台、刷题网站等，保证所选样本足够全面。其次，在选择题目时，尽可能不选择那种一眼就能知道结果的简单题，也没有选择那种怪题、偏题、难题。我们的原则是选择那些难度适中或者看上去简单但实际容易中陷阱的题目。我们力求遴选出来的算法真题能够最大限度地帮助读者。在真题的解析上，采用层层递进的写法，先易后难，将问题抽丝剥茧，使得读者能够跟随作者的思路，一步步找到问题的最优解。

写作的过程是一个自我提高、自我认识的过程，很多知识，只有深入理解与剖析后，才能领悟其中的精髓，掌握其中的技巧，程序员求职算法也不例外。本书不仅具备了其他书籍分析透彻、代码清晰合理等优点，还具备以下几个方面独有的优势。

第一，本书分多种语言版本实现：C/C++、Java、C#等，不管读者侧重于哪一种语言，都能够

从书中找到适合自己的内容。后续可能还有 PHP 等其他语言描述的图书出现。**本书中如果没有特别强调，则代码实现均默认使用 Java 语言。**

第二，每个题目除了有循序渐进的分析以外，还对方法进行了详细阐述，针对不同方法的时间复杂度与空间复杂度都进行了详细的分析。除此之外，为了更具说服力，每一种方法几乎都对应有示例讲解辅以说明。

第三，代码较为规范，完全参照华为编程规范、Google 编程规范和编码规范。程序员要想在团队中大展拳脚，就离不开合作，而合作的基础就是共同遵循统一的编码规范。不仅如此，规范化的编码往往有助于读者理解代码。

第四，除了题目讲解，还有部分触类旁通的题目供读者练习。本书不可能将所有的程序员求职类的数据结构与算法类题目囊括，但是会尽可能地将一些常见的求知类算法题、具有代表性的算法题重点讲解，将其他一些题目以练习题的形式展现在读者面前，以供读者思考与学习。

数据结构与算法知识博大精深，非一本或是几本著作就能将其讲解透彻的。尽管我们力求将所有程序员求职过程中出现的面试、笔试题一网打尽，试图做到知识覆盖面广，内容知识全，但仍然无法做到面面俱到，百分之百的读者满意率是本书以及后续改版奋斗与追求的目标，希望读者能够体谅。有兴趣的读者可以参阅《算法导论》《编程珠玑》等国外知名专家编写的专著进行知识的扩展与延伸。

其实，本书不仅可以作为程序员求职的应试类书籍，还可以作为数据结构与算法的教辅书籍。书中的很多思想和方法对于提高对数据结构与算法的理解是大有裨益的，不管是本科生还是研究生，不管是低年级学生还是高年级学生，不管对计算机底层知识或是当前的计算机前沿知识是否了解，都不影响你学好本书。

本书是作者历经四年时间打造的一本技术类精品图书，尽管我们尽最大努力希望做到百分之百的准确性，但百密一疏，书中不足之处在所难免，在恳请读者原谅的同时，也希望读者们能够将这些问题进行反馈，以便于未来继续改进与提高，为读者提供更加优秀的作品。

本书中引用的部分内容来源于网络上的无名英雄，无法追踪到最原始的出处，在此对这些幕后英雄致以最崇高的敬意。没有学不好的学生，只有教不好的老师，我们希望无论是什么层次的学生，都能毫无障碍地看懂书中所讲内容。如果读者存在求职困惑或是对本书中的内容存在异议，都可以通过 yuancoder@foxmail.com 联系作者。

猿媛之家

目　　录

面试、笔试经验技巧篇

想找到一份程序员的工作，一点技术都没有显然是不行的，但是只有技术也是不够的。面试、笔试经验技巧篇主要针对程序员面试、笔试中遇到的 13 个常见问题进行深度解析，并且结合实际情景，给出了一个较为合理的参考答案以供读者学习与应用，掌握这 13 个问题的解答精髓，对于求职者大有裨益。

程序员最可信赖的求职帮手

经验技巧 1　如何巧妙地回答面试官的问题

所谓“来者不善，善者不来”，程序员面试中，求职者不可避免地需要回答面试官各种“刁钻”、犀利的问题，回答面试官的问题千万不能简单地回答“是”或者“不是”，而应该具体分析“是”或者“不是”的理由。

回答面试官的问题是一门很深的学问。那么，面对面试官提出的各类问题，如何才能条理清晰地回答呢？如何才能让自己的回答不至于撞上枪口呢？如何才能让自己的答案令面试官满意呢？

谈话是一门艺术，回答问题也是一门艺术。同样的话，不同的回答方式，往往也会产生出不同的效果，甚至是截然不同的效果。在此，编者提出以下几点建议，供读者参考。

首先，回答问题务必谦虚谨慎。既不能让面试官觉得自己很自卑、唯唯诺诺，也不能让面试官觉得自己清高自负，而应该通过问题的回答表现出自己自信从容、不卑不亢的一面。例如，当面试官提出“你在项目中起到了什么作用”的问题时，如果求职者回答：我完成了团队中最难的工作，此时就会给面试官一种居功自傲的感觉，而如果回答：我完成了文件系统的构建工作，这个工作被认为是整个项目中最具有挑战性的一部分内容，因为它几乎无法重用以前的框架，需要重新设计。这种回答不仅不傲慢，反而有理有据，更能打动面试官。

其次，回答面试官的问题时，不要什么都说，要适当地留有悬念。人一般都有猎奇的心理，面试官自然也不例外。而且，人们往往对好奇的事情更有兴趣、更加偏爱，也更加记忆深刻。所以，在回答面试官问题时，切记说关键点而非细节，说重点而非和盘托出，通过关键点，吸引面试官的注意力，等待他们继续“刨根问底”。例如，当面试官对你的简历中一个算法问题有兴趣，希望了解时，可以如下回答：我设计的这种查找算法，对于 80%或以上的情况，都可以将时间复杂度从 O(n)降低到 O(log n)，如果您有兴趣，我可以详细给您分析具体的细节。

最后，回答问题要条理清晰、简单明了，最好使用“三段式”方式。所谓“三段式”，有点类似于中学作文中的写作风格，包括“场景/任务”“行动”和“结果”三部分内容。以面试官提的问题“你在团队建设中，遇到的最大挑战是什么”为例，第一步，分析场景/任务：在我参与的一个 ERP 项目中，我们团队一共四个人，除了我以外的其他三个人中，两个人能力很强，人也比较好相处，但有一个人却不太好相处，每次我们小组讨论问题时，他都不太爱说话，分配给他的任务也很难完成。第二步，分析行动：为了提高团队的综合实力，我决定找个时间和他单独谈一谈。于是我利用周末时间，约他一起吃饭，吃饭的时候顺便讨论了一下我们的项目，我询问了一些项目中他遇到的问题，通过他的回答，我发现他并不懒，也不糊涂，只是对项目不太了解，缺乏经验，缺乏自信而已，所以越来越孤立，越来越不愿意讨论问题。为了解决这个问题，我尝试着把问题细化到他可以完成的程度，从而建立起他的自信心。第三步，分析结果：他是小组中水平最弱的人，但是，慢慢地，他的技术变得越来越厉害了，也能够按时完成安排给他的工作了，人也越来越自信了，也越来越喜欢参与我们的讨论，并发表自己的看法，我们也都愿意与他一起合作了。“三段式”回答的一个最明显的好处就是条理清晰，既有描述，也有结果，有根有据，让面试官一目了然。

回答问题的技巧，是一门大学问。求职者可以在平时的生活中加以练习，提高自己与人沟通的技能，等到面试时，自然就得心应手了。

经验技巧 2　如何回答技术性的问题

程序员面试中，面试官会经常询问一些技术性的问题，有的问题可能比较简单，都是历年的面试、笔试真题，求职者在平时的复习中会经常遇到。但有的题目可能比较难，来源于 Google、Microsoft 等大企业的题库或是企业自己为了招聘需要设计的题库，求职者可能从来没见过或者不能完整地、独立地想到解决方案，而这些题目往往又是企业比较关注的。

如何能够回答好这些技术性的问题呢？编者建议：会做的一定要拿满分，不会做的一定要拿部分分。即对于简单的题目，求职者要努力做到完全正确，毕竟这些题目，只要复习得当，完全回答正确一点问题都没有（编者认识的一个朋友曾把《编程之美》《编程珠玑》《程序员面试笔试宝典》上面的技术性题目与答案全都背熟，找工作时遇到该类问题解决得非常轻松）；对于难度比较大的题目，不要惊慌，也不要害怕，即使无法完全做出来，也要努力思考问题，哪怕是半成品也要写出来，至少要把自己的思路表达给面试官，让面试官知道你的想法，而不是完全回答不会或者放弃，因为面试官很多时候除了关注求职者的独立思考问题的能力以外，还会关注求职者技术能力的可塑性，观察求职者是否能够在别人的引导下去正确地解决问题。所以，对于不会的问题，面试官很有可能会循序渐进地启发求职者去思考，通过这个过程，让面试官更加了解求职者。

一般而言，在回答技术性问题时，求职者大可不必胆战心惊，除非是没学过的新知识，否则，一般都可以采用以下六个步骤来分析解决。

（1）勇于提问

面试官提出的问题，有时候可能过于抽象，让求职者不知所措，或者无从下手，因此，对于面试中的疑惑，求职者要勇敢地提出来，多向面试官提问，把不明确或二义性的情况都问清楚。不用担心你的问题会让面试官烦恼，影响面试成绩，相反还对面试结果产生积极的影响：一方面，提问可以让面试官知道求职者在思考，也可以给面试官一个心思缜密的好印象；另一方面，方便后续自己对问题的解答。

例如，面试官提出一个问题：设计一个高效的排序算法。求职者可能没有头绪，排序对象是链表还是数组？数据类型是整型、浮点型、字符型还是结构体类型？数据基本有序还是杂乱无序？数据量有多大，1000 以内还是百万以上？此时，求职者大可以将自己的疑问提出来，问题清楚了，解决方案也自然就出来了。

（2）高效设计

对于技术性问题，如何才能打动面试官？完成基本功能是必需的，仅此而已吗？显然不是，完成基本功能最多只能算及格水平，要想达到优秀水平，至少还应该考虑更多的内容，以排序算法为例：时间是否高效？空间是否高效？数据量不大时也许没有问题，如果是海量数据呢？是否考虑了相关环节，如数据的“增删改查”？是否考虑了代码的可扩展性、安全性、完整性以及鲁棒性。如果是网站设计，是否考虑了大规模数据访问的情况？是否需要考虑分布式系统架构？是否考虑了开源框架的使用？

（3）伪代码先行

有时候实际代码会比较复杂，上手就写很有可能会漏洞百出、条理混乱，所以求职者可以首先征求面试官的同意，在编写实际代码前，写一个伪代码或者画好流程图，这样做往往会让思路更加清晰明了。

（4）控制节奏

如果是算法设计题，面试官都会给求职者一个时间限制用以完成设计，一般为 20min。完成得太慢，会给面试官留下能力不行的印象，但完成得太快，如果不能保证百分百正确，也会给面试官留下毛手毛脚的印象。速度快当然是好事情，但只有速度，没有质量，速度快根本

不会给面试加分。所以，编者建议，回答问题的节奏最好不要太慢，也不要太快，如果实在是完成得比较快，也不要急于提交给面试官，最好能够利用剩余的时间，认真检查一些边界情况、异常情况及极性情况等，看是否也能满足要求。

（5）规范编码

回答技术性问题时，多数都是纸上写代码，离开了编译器的帮助，求职者要想让面试官对自己的代码一看即懂，除了字迹要工整外，最好是能够严格遵循编码规范：函数变量命名、换行缩进、语句嵌套和代码布局等。同时，代码设计应该具有完整性，保证代码能够完成基本功能、输入边界值能够得到正确的输出、对各种不合规范的非法输入能够做出合理的错误处理，否则写出的代码即使无比高效，面试官也不一定看得懂或者看起来非常费劲，这些对面试成功都是非常不利的。

（6）精心测试

任何软件都有 bug，但不能因为如此就纵容自己的代码，允许错误百出。尤其是在面试过程中，实现功能也许并不十分困难，困难的是在有限的时间内设计出的算法，各种异常是否都得到了有效的处理，各种边界值是否都在算法设计的范围内。

测试代码是让代码变得完备的高效方式之一，也是一名优秀程序员必备的素质之一。所以，在编写代码前，求职者最好能够了解一些基本的测试知识，做一些基本的单元测试、功能测试、边界测试以及异常测试。

在回答技术性问题时，千万别一句话都不说，面试官面试的时间是有限的，他们希望在有限的时间内尽可能地多了解求职者，如果求职者坐在那里一句话不说，不仅会让面试官觉得求职者技术水平不行，思考问题能力以及沟通能力可能都存在问题。

其实，在面试时，求职者往往会存在一种思想误区，把技术性面试的结果看得太重要了。面试过程中的技术性问题，结果固然重要，但也并非最重要的内容，因为面试官看重的不仅仅是最终的结果，还包括求职者在解决问题的过程中体现出来的逻辑思维能力以及分析问题的能力。所以，求职者在与面试官的“博弈”中，要适当地提问，通过提问获取面试官的反馈信息，并抓住这些有用的信息进行辅助思考，进而提高面试的成功率。

经验技巧 3　如何回答非技术性问题

评价一个人的能力，除了专业能力，还有一些非专业能力，如智力、沟通能力和反应能力等，所以在 IT 企业招聘过程的笔试、面试环节中，并非所有的内容都是 C/C++/Java、数据结构与算法及操作系统等专业知识，也包括其他一些非技术类的知识，如智力题、推理题和作文题等。技术水平测试可以考查一个求职者的专业素养，而非技术类测试则更强调求职者的综合素质，包括数学分析能力、反应能力、临场应变能力、思维灵活性、文字表达能力和性格特征等内容。考查的形式多种多样，部分与公务员考查相似，主要包括行政职业能力测验（简称“行测”）（占大多数）、性格测试（大部分都有）、应用文和开放问题等内容。

每个人都有自己的答题技巧，答题方式也各不相同，以下是一些相对比较好的答题技巧（以行测为例）：

1）合理有效的时间管理。由于题目的难易不同，答题要分清轻重缓急，最好的做法是不按顺序答题。“行测”中有各种题型，如数量关系、图形推理、应用题、资料分析和文字逻辑等，不同的人擅长的题型是不一样的，因此应该首先回答自己最擅长的问题。例如，如果对数字比较敏感，那么就先答数量关系。

2）注意时间的把握。由于题量一般都比较大，可以先按照总时间/题数来计算每道题的平均答题时间，如 10s，如果看到某一道题 5s 后还没思路，则马上做后面的题。在做行测题目

时，以在最短的时间内拿到最多分为目标。
3）平时多关注图表类题目，培养迅速抓住图表中各个数字要素间相互逻辑关系的能力。
4）做题要集中精力、全神贯注，才能将自己的水平最大限度地发挥出来。
5）学会关键字查找，通过关键字查找，能够提高做题效率。
6）提高估算能力，有很多时候，估算能够极大地提高做题速度，同时保证正确率。
除了行测以外，一些企业非常相信个人性格对入职匹配的影响，所以都会引入相关的性格测试题用于测试求职者的性格特性，看其是否适合所投递的职位。大多数情况下，只要按照自己的真实想法选择就行了，因为测试是为了得出正确的结果，所以大多测试题前后都有相互验证的题目。如果求职者自作聪明，则很可能导致测试前后不符，这样很容易让企业发现求职者是个不诚实的人，从而首先予以筛除。

经验技巧 4　如何回答快速估算类问题

有些大企业的面试官，总喜欢出一些快速估算类问题，对他们而言，这些问题只是手段，不是目的，能够得到一个满意的结果固然是他们所需要的，但更重要的是通过这些题目可以考查求职者的快速反应能力以及逻辑思维能力。由于求职者平时准备的时候可能对此类问题有所遗漏，一时很难想到解决的方案。而且，这些题目乍一看确实是毫无头绪，无从下手，其实求职者只要冷静下来，稍加分析，就能找到答案。因为此类题目比较灵活，属于开放性试题，一般没有标准答案，只要弄清楚回答要点，分析合理到位，具有说服力，能够自圆其说，就是正确答案。
例如，面试官可能会问这样一个问题：“请估算一下一家商场在促销时一天的营业额?”求职者又不是统计局官员，如何能够得出一个准确的数据呢？求职者又不是商场负责人，如何能够得出一个准确的数据呢？即使求职者是商场的负责人，也不可能弄得清清楚楚明明白白吧？
难道此题就无解了吗？其实不然，本题只要能够分析出一个概数就行了，不一定要精确数据，而分析概数的前提就是做出各种假设。以该问题为例，可以尝试从以下思路入手：从商场规模、商铺规模入手，通过每平方米的租金，估算出商场的日租金，再根据商铺的成本构成，得到全商场日均交易额，再考虑促销时的销售额与平时销售额的倍数关系，乘以倍数，即可得到促销时一天的营业额。具体而言，包括以下估计数值：
1）以一家较大规模商场为例，商场一般按 6 层计算，每层长约 100m，宽约 100m，合计 60000m^2 的面积。
2）商铺规模占商场规模的一半左右，合计 30000m^2。
3）商铺租金约为 40 元/ m^2，估算出年租金为 40×30000×365 元=4.38 亿元。
4）对商户而言，租金一般占销售额 20%，则年销售额为 4.38 亿元×5=21.9 亿元。计算平均日销售额为 21.9 亿元/365=600 万元。
5）促销时的日销售额一般是平时的 10 倍，所以约为 600 万元×10=6000 万元。
此类题目涉及面比较广，如估算一下北京小吃店的数量？估算一下中国在过去一年方便面的市场销售额是多少？估算一下长江的水的质量？估算一下一个行进在小雨中的人 5 分钟内身上淋到的雨的质量？估算一下东方明珠电视塔的质量？估算一下中国一年一共用掉了多少块尿布？估算一下杭州的轮胎数量？但一般都是即兴发挥，不是哪道题记住答案就可以应付得了的。遇到此类问题，一步步抽丝剥茧，才是解决之道。

经验技巧 5　如何回答算法设计问题

程序员面试中的很多算法设计问题，都是历年来各家企业的“炒现饭”，不管求职者以前对算法知识掌握得是否扎实，理解得是否深入，只要面试前买本《程序员面试笔试宝典》，应付此类题目完全没有问题。但遗憾的是，很多世界级知名企业也深知这一点，如果纯粹是出一些毫无技术含量的题目，对于考前“突击手”而言，可能会占尽便宜，但对于那些技术好的人而言是非常不公平的。所以，为了把优秀的求职者与一般的求职者更好地区分开来，面试题会年年推陈出新，越来越倾向于出一些有技术含量的“新”题，这些题目以及答案，不再是以前的问题了，而是经过精心设计的好题。

在程序员面试中，算法的地位就如同是 GRE 或托福考试在出国留学中的地位一样，必须但不是最重要的，它只是众多考核方面中的一个方面而已。虽然如此，但并非说就不用去准备算法知识了，因为算法知识回答得好，必然会成为面试的加分项，对于求职成功，有百利而无一害。那么如何应对此类题目呢？很显然，编者不可能将此类题目都在《程序员面试笔试宝典》中一一解答，一是由于内容过多，篇幅有限，二是也没必要，今年考过了，以后一般就不会再考了，不然还是没有区分度。编者认为，靠死记硬背肯定是行不通的，解答此类算法设计问题，需要求职者具有扎实的基本功和良好的运用能力，因为这些能力需要求职者“十年磨一剑”，但“授之以鱼不如授之以渔”编者可以提供一些比较好的答题方法和解题思路，以供求职者在面试时应对此类算法设计问题。

（1）归纳法

此方法通过写出问题的一些特定的例子，分析总结其中的规律。具体而言，就是通过列举少量的特殊情况，经过分析，最后找出一般的关系。例如，某人有一对兔子饲养在围墙中，如果它们每个月生一对兔子，且新生的兔子在第二个月后也是每个月生一对兔子，问一年后围墙中共有多少对兔子。

使用归纳法解答此题，首先想到的就是第一个月有多少对兔子。第一个月最初的一对兔子生下一对兔子，此时围墙内共有两对兔子。第二个月仍是最初的一对兔子生下一对兔子，共有 3 对兔子。到第三个月除最初的兔子新生一对兔子外，第一个月生的兔子也开始生兔子，因此共有 5 对兔子。通过举例，可以看出，从第二个月开始，每一个月兔子总数都是前两个月兔子总数之和，Un+1=Un+Un−1。一年后，围墙中的兔子总数为 377 对。

此种方法比较抽象，也不可能对所有的情况进行列举，所以得出的结论只是一种猜测，还需要进行证明。

（2）相似法

如果面试官提出的问题与求职者以前用某个算法解决过的问题相似，此时就可以触类旁通，尝试改进原有算法来解决这个新问题。而通常情况下，此种方法都会比较奏效。

例如，实现字符串的逆序打印，也许求职者从来就没遇到过此问题，但将字符串逆序肯定在求职准备的过程中是见过的。将字符串逆序的算法稍加处理，即可实现字符串的逆序打印。

（3）简化法

此方法首先将问题简单化，如改变数据类型、空间大小等，然后尝试着将简化后的问题解决，一旦有了一个算法或者思路可以解决这个问题，再将问题还原，尝试着用此类方法解决原有问题。

例如，在海量日志数据中提取出某日访问×××网站次数最多的那个 IP。由于数据量巨大，直接进行排序显然不可行，但如果数据规模不大时，采用直接排序是一种好的解决方法。那

么如何将问题规模缩小呢？这时可以使用 Hash 法，Hash 往往可以缩小问题规模，然后在简化过的数据里面使用常规排序算法即可找出此问题的答案。

（4）递归法

为了降低问题的复杂度，很多时候都会将问题逐层分解，最后归结为一些最简单的问题，这就是递归。此种方法，首先要能够解决最基本的情况，然后以此为基础，解决接下来的问题。

例如，在寻求全排列时，可能会感觉无从下手，但仔细推敲，会发现后一种排列组合往往是在前一种排列组合的基础上进行的重新排列。只要知道了前一种排列组合的各类组合情况，只需将最后一个元素插入到前面各种组合的排列里面，就实现了目标：即先截去字符串 s[1…n]中的最后一个字母，生成所有 s[1…n−1]的全排列，然后再将最后一个字母插入到每一个可插入的位置。

（5）分治法

任何一个可以用计算机求解的问题所需的计算时间都与其规模有关。问题的规模越小，越容易直接求解，解题所需的计算时间也越少。而分治法正是充分考虑到这一内容，将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。分治法一般包含以下三个步骤：

1）将问题的实例划分为几个较小的实例，最好具有相等的规模。

2）对这些较小的实例求解，而最常见的方法一般是递归。

3）如果有必要，合并这些较小问题的解，以得到原始问题的解。

分治法是程序员面试常考的算法之一，一般适用于二分查找、大整数相乘、求最大子数组和、找出伪币、金块问题、矩阵乘法、残缺棋盘、归并排序、快速排序、距离最近的点对、导线与开关等。

（6）Hash 法

很多面试、笔试题目，都要求求职者给出的算法尽可能高效。什么样的算法是高效的？一般而言，时间复杂度越低的算法越高效。而要想达到时间复杂度的高效，很多时候就必须在空间上有所牺牲，用空间来换时间。而用空间换时间最有效的方式就是 Hash 法、大数组和位图法。当然，有时，面试官也会对空间大小进行限制，那么此时求职者只能再去思考其他的方法了。

其实，凡是涉及大规模数据处理的算法设计中，Hash 法就是最好的方法之一。

（7）轮询法

在设计每道面试、笔试题时，往往会有一个载体，这个载体便是数据结构，如数组、链表、二叉树或图等，当载体确定后，可用的算法自然而然地就会显现出来。可问题是很多时候并不确定这个载体是什么，当无法确定这个载体时，一般也就很难想到合适的方法了。

编者建议，此时，求职者可以采用最原始的思考问题的方法——轮询法。常考的数据结构与算法一共就几种（见表 0-1），即使不完全一样，也是由此衍生出来的或者相似的。

表 0-1　最常考的数据结构与算法知识点

数据结构	算　　法	概　　念
链表	广度（深度）优先搜索	位操作
数组	递归	设计模式
二叉树	二分查找	内存管理（堆、栈等）
树	排序（归并排序、快速排序等）	—

堆（大顶堆、小顶堆）	树的插入/删除/查找/遍历等	—
栈	图论	—
队列	Hash 法	—
向量	分治法	—
Hash 表	动态规划	—

此种方法看似笨拙，却很实用，只要求职者对常见的数据结构与算法烂熟于心，一点都没有问题。

为了更好地理解这些方法，求职者可以在平时的准备过程中，应用此类方法去答题，做题多了，自然对各种方法也就熟能生巧了，面试时再遇到此类问题，也就能够得心应手了。当然，千万不要相信能够在一夜之间练成“绝世神功”。算法设计的功底就是平时一点一滴的付出和思维的磨炼。方法与技巧只能锦上添花，却不会让自己变得从容自信，真正的功力还是需要一个长期的积累过程的。

经验技巧 6　如何回答系统设计题

应届生在面试时，偶尔也会遇到一些系统设计题，而这些题目往往只是测试求职者的知识面，或者测试求职者对系统架构方面的了解，一般不会涉及具体的编码工作。虽然如此，对于此类问题，很多人还是感觉难以应对，也不知道从何处答题。

如何应对此类题目呢？在正式介绍基础知识之前，首先列举几个常见的系统设计相关的面试、笔试题。

题目 1：设计一个 DNS 的 Cache 结构，要求能够满足 5000 次/s 以上的查询，满足 IP 数据的快速插入，查询的速度要快（题目还给出了一系列的数据，比如站点数总共为 5000 万、IP 地址有 1000 万等）。

题目 2：有 N 台机器，M 个文件，文件可以以任意方式存放到任意机器上，文件可任意分割成若干块。假设这 N 台机器的宕机率小于 33%，要想在宕机时可以从其他未宕机的机器中完整导出这 M 个文件，求最好的存放与分割策略。

题目 3：假设有 30 台服务器，每台服务器上面都存有上百亿条数据（有可能重复），如何找出这 30 台机器中，根据某关键字重复出现次数最多的前 100 条？要求使用 Hadoop 来实现。

题目 4：设计一个系统，要求写速度尽可能快，并说明设计原理。

题目 5：设计一个高并发系统，说明架构和关键技术要点。

题目 6：有 25TB 的 log(query->queryinfo)，log 在不断地增长，设计一个方案，给出一个 query 能快速返回 queryinfo。

以上所有问题中凡是不涉及高并发的，基本可以采用 Google 的三个技术解决，即 GFS、MapReduce 和 Bigtable，这三个技术被称为“Google 三驾马车”。Google 只公开了论文而未开源代码，开源界对此非常有兴趣，仿照这三篇论文实现了一系列软件，如 Hadoop、HBase、HDFS 及 Cassandra 等。

在 Google 这些技术还未出现之前，企业界在设计大规模分布式系统时，采用的架构往往是 DataBase+Sharding+Cache，现在很多网站（比如淘宝网、新浪微博）仍采用这种架构。在这种架构中，仍有很多问题值得去探讨，如采用哪种数据库，是 SQL 界的 MySQL 还是 NoSQL 界的 Redis/TFS，两者有何优劣？采用什么方式 sharding（数据分片），是水平分片还是垂直

分片？据网上资料显示，淘宝网、新浪微博图片存储中曾采用的架构是 Redis/MySQL/TFS+Sharding+Cache，该架构解释如下：前端 Cache 是为了提高响应速度，后端数据库则用于数据永久存储，防止数据丢失，而 Sharding 是为了在多台机器间分摊负载。最前端由大块的 Cache 组成，要保证至少 99%（淘宝网图片存储模块是真实的）的访问数据落在 Cache 中，这样可以保证用户访问速度，减少后端数据库的压力。此外，为了保证前端 Cache 中的数据与后端数据库中的数据一致，需要有一个中间件异步更新（为什么使用异步？理由是，同步代价太高）数据。新浪有个开源软件叫 Memcachedb（整合了 Berkeley DB 和 Memcached），正是用于完成此功能。另外，为了分摊负载压力和海量数据，会将用户微博信息经过分片后存放到不同节点上（称为“Sharding”）。

这种架构优点非常明显——简单，在数据量和用户量较小时完全可以胜任。但缺点是扩展性和容错性太差，维护成本非常高，尤其是数据量和用户量暴增之后，系统不能通过简单地增加机器解决该问题。

鉴于此，新的架构应运而生。新的架构仍然采用 Google 公司的架构模式与设计思想，以下将分别就此内容进行分析。

GFS 是一个可扩展的分布式文件系统，用于大型的、分布式的、对大量数据进行访问的应用。它运行于廉价的普通硬件上，提供容错功能。现在开源界有 HDFS（Hadoop Distributed File System），该文件系统虽然弥补了数据库+Sharding 的很多缺点，但自身仍存在一些问题，比如由于采用 master/slave 架构，因此存在单点故障问题；元数据信息全部存放在 master 端的内存中，因而不适合存储小文件，或者说如果存储大量小文件，那么存储的总数据量不会太大。

MapReduce 是针对分布式并行计算的一套编程模型。其最大的优点是，编程接口简单，自动备份（数据默认情况下会自动备三份），自动容错和隐藏跨机器间的通信。在 Hadoop 中，MapReduce 作为分布计算框架，而 HDFS 作为底层的分布式存储系统，但 MapReduce 不是与 HDFS 耦合在一起的，完全可以使用自己的分布式文件系统替换 HDFS。当前 MapReduce 有很多开源实现，如 Java 实现 Hadoop MapReduce、C++实现 Sector/sphere 等，甚至有些数据库厂商将 MapReduce 集成到数据库中了。

BigTable 俗称“大表”，是用来存储结构化数据的。编者认为，BigTable 开源实现最多，包括 HBase、Cassandra 和 levelDB 等，使用也非常广泛。

除了 Google 的这“三驾马车”以外，还有其他一些技术可供学习与使用。

Dynamo 亚马逊的 key-value 模式的存储平台，可用性和扩展性都很好，采用 DHT（Distributed Hash Table）对数据分片，解决单点故障问题，在 Cassandra 中也借鉴了该技术，在 BT 和电驴这两种下载引擎中，也采用了类似算法。

虚拟节点技术 该技术常用于分布式数据分片中。具体应用场景：有一大块数据（可能 TB 级或者 PB 级），需按照某个字段（key）分片存储到几十（或者更多）台机器上，同时想尽量负载均衡且容易扩展。传统做法是：Hash(key) mod N，这种方法最大的缺点是不容易扩展，即增加或者减少机器均会导致数据全部重分布，代价太大。于是新技术诞生了，其中一种是上面提到的 DHT，现在已经被很多大型系统采用，还有一种是对“Hash(key) mod N”的改进：假设要将数据分布到 20 台机器上，传统做法是 Hash(key) mod 20，而改进后，N 取值要远大于 20，比如是 20000000，然后采用额外一张表记录每个节点存储的 key 的模值，比如：

node1：0～1000000

node2：1000001～2000000

……

这样，当添加一个新的节点时，只需将每个节点上部分数据移动给新节点，同时修改一下该

表即可。

Thrift Thrift 是一个跨语言的 RPC 框架，分别解释“RPC”和“跨语言”如下：RPC 是远程过程调用，其使用方式与调用一个普通函数一样，但执行体发生在远程机器上；跨语言是指不同语言之间进行通信，比如 C/S 架构中，Server 端采用 C++编写，Client 端采用 PHP 编写，怎样让两者之间通信，Thrift 是一种很好的方式。

本篇最前面的几道题均可以映射到以上几个系统的某个模块中。

1）关于高并发系统设计，主要有以下几个关键技术点：缓存、索引、数据分片及锁粒度尽可能小。

2）题目 2 涉及现在通用的分布式文件系统的副本存放策略。一般是将大文件切分成小的 block（如 64MB）后，以 block 为单位存放三份到不同的节点上，这三份数据的位置需根据网络拓扑结构配置，一般而言，如果不考虑跨数据中心，可以这样存放：两个副本存放在同一个机架的不同节点上，而另外一个副本存放在另一个机架上，这样从效率和可靠性上，都是最优的（这个 Google 公布的文档中有专门的证明，有兴趣的读者可参阅一下）。如果考虑跨数据中心，可将两份存在一个数据中心的不同机架上，另一份放到另一个数据中心。

3）题目 4 涉及 BigTable 的模型。主要思想：将随机写转化为顺序写，进而大大提高写速度。具体方法：由于磁盘物理结构的独特设计，其并发的随机写（主要是因为磁盘寻道时间长）非常慢，考虑到这一点，在 BigTable 模型中，首先会将并发写的大批数据放到一个内存表（称为“memtable”）中，当该表大到一定程度后，会顺序写到一个磁盘表（称为“SSTable”）中，这种写是顺序写，效率极高。此时，随机读可不可以这样优化？答案是：看情况。通常而言，如果读并发度不高，则不可以这么做，因为如果将多个读重新排列组合后再执行，系统的响应时间太慢，用户可能接受不了，而如果读并发度极高，也许可以采用类似机制。

经验技巧 7　如何解决求职中的时间冲突问题

对求职者而言，求职季就是一个赶场季，一天少则几家、十几家企业入校招聘，多则几十家、上百家企业招兵买马。企业多，选择项自然也多，这固然是一件好事情，但由于招聘企业实在是太多，自然而然会导致另外一个问题的发生：同一天企业扎堆，且都是自己心仪或欣赏的大企业。如果不能够提前掌握企业的宣讲时间、地点，是很容易迟到或错过的。但有时候即使掌握了宣讲时间、笔试和面试时间，还是会因为时间冲突而必须有所取舍。

到底该如何取舍呢？该如何应对这种时间冲突的问题呢？在此，编者将自己的一些想法和经验分享出来，供读者参考。

1）如果多家心仪企业的校园宣讲时间发生冲突（前提是只宣讲、不笔试，否则请看后面的建议），此时最好的解决方法是和同学或朋友商量好，各去一家，然后大家进行信息共享。

2）如果多家心仪企业的笔试时间发生冲突，此时只能选择其一，毕竟企业的笔试时间都是考虑到了成百上千人的安排，需要提前安排考场、考务人员和阅卷人员等，不可能为了某一个人而轻易改变。所以，最好选择自己更有兴趣的企业参加笔试。

3）如果多家心仪企业的面试时间发生冲突，不要轻易放弃。对面试官而言，面试任何人都是一样的，因为面试官谁都不认识，而面试时间也是灵活性比较大的，一般可以通过电话协商。求职者可以与相关工作人员（一般是企业的 HR）进行沟通，以正当理由（如学校的事宜、导师的事宜或家庭的事宜等，前提是必须能够说服人，不要给出的理由连自己都说服不了）让其调整时间，一般都能协调下来。但为了保证协调的成功率，一般要接到面试通知后第一时间联系相关工作人员变更时间，这样他们协调起来也更方便。

以上这些建议在应用时，很多情况下也做不到全盘兼顾，当必须进行多选一的时候，求职者就要对此进行评估了，评估的项目包括对企业的中意程度、获得录取的概率及去工作的可能性等。评估的结果往往具有很强的参考性，求职者依据评估结果做出的选择一般也会比较合理。

经验技巧 8　如果面试问题曾经遇见过，是否要告知面试官

面试中，大多数题目都不是凭空想象出来的，而是有章可循，只要求职者肯花时间，耐得住寂寞，复习得当，基本上在面试前都会见过相同的或者类似的问题（当然，很多知名企业每年都会推陈出新，这些题目是很难完全复习到位的）。所以，在面试中，求职者曾经遇见过面试官提出的问题也就不足为奇了。那么，一旦出现这种情况，求职者是否要如实告诉面试官呢？

选择不告诉面试官的理由比较充分：首先，面试的题目 60%～70%都是已见题型，见过或者见过类似的不足为奇，难道要一一告知面试官吗？如果那样，估计就没有几个题不用告知面试官了。其次，即使曾经见过该问题了，也是自己辛勤耕耘、努力奋斗的结果，很多人复习不用功或者方法不到位，也许从来就没见过，而这些题也许正好是拉开求职者差距的分水岭，是面试官用来区分求职者实力的内容。最后，一旦告知面试官，面试官很有可能会不断地加大面试题的难度，对求职者的面试可能没有好处。

同样，选择告诉面试官的理由也比较充分：第一，如实告诉面试官，不仅可以彰显出求职者个人的诚实品德，还可以给面试官留下良好的印象，能够在面试中加分。第二，有些问题，即使求职者曾经复习过，但也无法保证完全回答正确，如果向面试官如实相告，没准还可以规避这一问题，避免错误的发生。第三，求职者如果见过该问题，也能轻松应答，题目简单倒也无所谓，一旦题目难度比较大，求职者却对面试官有所隐瞒，就极有可能给面试官造成一种求职者水平很强的假象，进而导致面试官的判断出现偏差，后续的面试有可能向着不利于求职者的方向发展。

其实，仁者见仁，智者见智，这个问题并没有固定的答案，需要根据实际情况来决定。针对此问题，一般而言，如果面试官不主动询问求职者，求职者也不用主动告知面试官真相。但如果求职者觉得告知面试官真相对自己更有利的时候，也可以主动告知。

经验技巧 9　在被企业拒绝后是否可以再申请

很多企业为了能够在一年一度的招聘季节中，提前将优秀的程序员锁定到自己的麾下，往往会先下手为强。他们通常采取的措施有两种：一是招聘实习生；二是多轮招聘。很多人可能会担心，万一面试时发挥不好，没被企业选中，会不会被企业接入黑名单，从此与这家企业无缘了。

一般而言，企业是不会“记仇”的，尤其是知名的大企业，对此都会有明确的表示。如果在企业的实习生招聘或在企业以前的招聘中未被录取，一般是不会被拉入企业的“黑名单”。在下一次招聘中，和其他求职者，具有相同的竞争机会（有些企业可能会要求求职者等待半年到一年时间才能应聘该企业，但上一次求职的不好表现不会被计入此次招聘中）。

录取被拒绝了，也许是在考验，也许是在等待，也许真的是拒绝。但无论出于什么原因，此时此刻都不要对自己丧失信心。所以，即使被企业拒绝了也不是什么大事，以后还是有机会的，有志者自有千计万计，无志者只感千难万难，关键是看求职者愿意成为什么样的人。

经验技巧 10　如何应对自己不会回答的问题

在面试的过程中，对面试官提出的问题求职者并不是都能回答出来，计算机技术博大精深，

很少有人能对计算机技术的各个分支学科了如指掌。而且抛开技术层面的问题，在面试那种紧张的环境中，回答不上来的情况也容易出现。面试过程中遇到自己不会回答的问题时，错误的做法是保持沉默或者支支吾吾、不懂装懂，硬着头皮胡乱说一通，这样会使面试气氛很尴尬，很难再往下继续进行。

其实面试遇到不会的问题是一件很正常的事情，没有人是万事通，即使对自己的专业有相当的研究与认识，也可能会在面试中遇到感觉没有任何印象、不知道如何回答的问题。在面试中遇到实在不懂或不会回答的问题，正确的做法是本着实事求是的原则，态度诚恳，告诉面试官不知道答案。例如，“对不起，不好意思，这个问题我回答不出来，我能向您请教吗?”征求面试官的意见时可以说说自己的个人想法，如果面试官同意听了，就将自己的想法说出来，回答时要谦逊有礼，切不可说起没完。然后应该虚心地向面试官请教，表现出强烈的学习欲望。

所以，遇到自己不会的问题时，正确的做法是，“知之为知之，不知为不知”，不懂就是不懂，不会就是不会，一定要实事求是，坦然面对。最后也能给面试官留下诚实、坦率的好印象。

经验技巧 11　如何应对面试官的“激将法”语言

“激将法”是面试官用以淘汰求职者的一种惯用方法，它是指面试官采用怀疑、尖锐或咄咄逼人的交流方式来对求职者进行提问的方法。例如，“我觉得你比较缺乏工作经验”“我们需要活泼开朗的人，你恐怕不合适”“你的教育背景与我们的需求不太适合”“你的成绩太差”“你的英语没过六级”“你的专业和我们不对口”“为什么你还没找到工作”或“你竟然有好多门课不及格”等。很多求职者遇到这样的问题，会很快产生是来面试而不是来受侮辱的想法，往往会被“激怒”，于是奋起反抗。千万要记住，面试的目的是要获得工作，而不是要与面试官争高低，也许争辩取胜了，却失去了一次工作机会。所以对于此类问题求职者应该巧妙地去回答，一方面化解不友好的气氛，另一方面得到面试官的认可。

具体而言，受到这种“激将法”时，求职者首先应该保持清醒的头脑，企业让求职者来参加面试，说明已经通过了他们第一轮的筛选，至少从简历上看，已经表明求职者符合求职岗位的需要，企业对求职者还是感兴趣的。其次，做到不卑不亢，不要被面试官的思路带走，要时刻保持自己的思路和步调。此时可以换一种方式，如介绍自己的经历、工作和优势，来表现自己的抗压能力。

针对面试官提出的非名校毕业的问题，比较巧妙的回答是：比尔·盖茨也并非毕业于哈佛大学，但他一样成了世界首富，成为举世瞩目的人物。针对缺乏工作经验的问题，可以回答：每个人都是从没经验变为有经验的，如果有幸最终能够成为贵公司的一员，我将很快成为一个经验丰富的人。针对专业不对口的问题，可以回答：专业人才难得，复合型人才更难得，在某些方面，外行的灵感往往超过内行，他们一般没有思维定式，没有条条框框。面试官还可能提问：你的学历对我们来讲太高了。此时也可以很巧妙地回答：今天我带来的三张学历证书，您可以从中挑选一张您认为合适的，其他两张，您就不用管了。针对性格内向的问题，可以回答：内向的人往往具有专心致志、锲而不舍的品质，而且我善于倾听，我觉得应该把发言机会更多地留给别人。

面对面试官的“挑衅”行为，如果求职者回答得结结巴巴，或者无言以对，抑或怒形于色、据理力争，那就掉进了对方所设的陷阱。所以当求职者碰到此种情况时，最重要的一点就是保持头脑冷静，不要过分较真，以一颗平常心对待。

经验技巧 12　如何处理与面试官持不同观点这个问题

在面试的过程中，求职者所持有的观点不可能与面试官一模一样，在对某个问题的看法上，很有可能两个人相去甚远。当与面试官持不同观点时，有的求职者自作聪明，立马就反驳面试官，例如，“不见得吧！”“我看未必”“不会”“完全不是这么回事！”或“这样的说法未必全对”等，其实，虽然也许确实不像面试官所说的，但是太过直接的反驳往往会导致面试官心理的不悦，最终的结果很可能是“逞一时之快，失一份工作”。

就算与面试官持不一样的观点，也应该委婉地表达自己的真实想法，因为我们不清楚面试官的度量，碰到心胸宽广的面试官还好，万一碰到了“小心眼”的面试官，他和你较真起来，吃亏的还是自己。

所以回答此类问题的最好方法往往是应该先赞同面试官的观点，给对方一个台阶下，然后再说明自己的观点，用“同时”“而且”过渡，千万不要说“但是”，一旦说了“但是”“却”就容易把自己放在面试官的对立面去。

经验技巧 13　什么是职场暗语

随着求职大势的变迁发展，以往常规的面试套路因为过于单调、简明，已经被众多“面试达人”们挖掘出了各种“破解秘诀”，形成了类似“求职宝典”的各类“面经”。面试官们也纷纷升级面试模式，为求职者们制作了更为隐蔽、间接、含混的面试题目，让那些早已流传开来的“面试攻略”毫无用武之地，一些蕴涵丰富信息但以更新面目出现的问话屡屡“秒杀”求职者，让求职者一头雾水，掉进了陷阱里面还以为“吃到肉”了。例如，“面试官从头到尾都表现出对我很感兴趣的样子，营造出马上就要录用我的氛围，为什么我最后还是落选？”“为什么 HR 会问我一些与专业、能力根本无关的奇怪问题，我感觉回答得也还行，为什么最后还是被拒绝了？”其实，这都是没有听懂面试“暗语”，没有听出面试官“弦外之音”的表现。“暗语”已经成为一种测试求职者心理素质、挖掘求职者内心真实想法的有效手段。理解这些面试中的暗语，对于求职者而言，不可或缺。

以下是一些常见的面试暗语，求职者一定要弄清楚其中蕴含的深意，不然可能“躺着也中枪”，最后只能铩羽而归。

（1）请把简历先放在这，有消息我们会通知你的

面试官说出这句话，则表明他对你已经“兴趣不大”，为什么一定要等到有消息了再通知呢？难道现在不可以吗？所以，作为求职者，此时一定不要自作聪明、一厢情愿地等待着他们有消息通知，因为他们一般不会有消息了。

（2）我不是人力资源的，你别拘束，咱们就当是聊天，随便聊聊

一般来说，能当面试官的人都是久经沙场的老将，都不太好对付。表面上彬彬有礼，看上去很和气的样子，说起话来可能偶尔还带点小结巴，但没准儿巴不得下个套把面试者套进去。所以，作为求职者，千万不能被眼前的这种“假象”所迷惑，而应该时刻保持高度警觉，面试官不经意间问出来的问题，看似随意，很可能是他最想知道的。所以千万不要把面试过程当作聊天，当作朋友之间的侃大山，不要把面试官提出的问题当作是普通问题，而应该对每一个问题都仔细思考，认真回答，切忌不经过大脑的随意接话和回答。

（3）是否可以谈谈你的要求和打算

面试官在翻阅了求职者的简历后，说出这句话，很有可能是对求职者有兴趣，此时求职者应该尽量全方位地表现个人水平与才能，但也不能引起对方的反感。

（4）面试时只是“例行公事”式的问答

如果面试时只是“例行公事”式的问答，没有什么激情或者主观性的赞许，此时希望就很渺茫了。但如果面试官对你的专长问得很细，而且表现出一种极大的关注与热情，那么此时希望会很大。作为求职者，一定要抓住机会，将自己最好的一面展示在面试官面前。

（5）你好，请坐

简单的一句话，从面试官口中说出来其含义就大不同了。一般而言，面试官说出此话，求职者回答“你好”或“您好”不重要，重要的是求职者是否“礼貌回应”和“坐不坐”。有的求职者的回应是“你好”或“您好”后直接落座，也有求职者回答“你好，谢谢”或“您好，谢谢”后落座，还有求职者一声不吭就坐下去，极个别求职者回答“谢谢”但不坐下来。前两种方法都可接受，后两者都不可接受。通过问候语，可以体现一个人的基本修养，直接影响在面试官心目中的第一印象。

（6）面试官向求职者探过身去

在面试的过程中，面试官会有一些肢体语言，了解这些肢体语言对于了解面试官的心理情况以及面试的进展情况非常重要。例如，当面试官向求职者探过身去时，一般表明面试官对求职者很感兴趣；当面试官打呵欠或者目光呆滞、游移不定，甚至打开手机看时间或打电话、接电话时，一般表明面试官此时有了厌烦的情绪；而当面试官收拾文件或从椅子上站起来，一般表明此时面试官打算结束面试。针对面试官的肢体语言，求职者也应该迎合他们：当面试官很感兴趣时，应该继续陈述自己的观点；当面试官厌烦时，此时最好停下来，询问面试官是否愿意再继续听下去；当面试官打算结束面试，领会其用意，并准备好收场白，尽快地结束面试。

（7）你从哪里知道我们的招聘信息的

面试官提出这种问题，一方面是在评估招聘渠道的有效性，另一方面是想知道求职者是否有熟人介绍。一般而言，熟人介绍总体上会有加分，但是也不全是如此。如果是一个在单位里表现不佳或者其推荐的历史记录不良的熟人介绍，则会起到相反的效果，而大多数面试官主要是为了评估自己企业发布招聘广告的有效性。

（8）你念书的时间还是比较富足的

表面上看，这是对他人的高学历表示赞赏，但同时也是一语双关，如果“高学历”的同时还搭配上一个“高年龄”，就一定要提防面试官的质疑：比如有些人因为上学晚或者工作了以后再回校读的研究生，毕业年龄明显高出平均年龄。此时一定要向面试官解释清楚，否则面试官如果自己揣摩，往往会向不利于求职者的方向思考。例如，求职者年龄大的原因是高考复读过、考研用了两年甚至更长时间或者是先工作后读研等，如果面试官有了这种想法，最终的求职结果也就很难说了。

（9）你有男/女朋友吗？对异地恋爱怎么看待

一般而言，面试官都会询问求职者的婚恋状况，一方面是对求职者个人问题的关心，另一方面，对于女性而言，绝大多数面试官不是看中求职者的美貌性感、温柔贤惠，很有可能是在试探求职者是否近期要结婚生子，将会给企业带来什么程度的负担。“能不能接受异地恋”，很有可能是考察求职者是否能够安心在一个地方工作，或者是暗示该岗位可能需要长期出差，试探求职者如何在感情和工作上做出抉择。与此类似的问题还有：如果求职者已婚，面试官会问是否生育，如果已育可能还会问小孩谁带。所以，如果面试官有这一层面的意思，尽量要当场表态，避免将来的麻烦。

（10）你还应聘过其他什么企业

面试官提出这种问题是在考核求职者的职业生涯规划，同时评估下被其他企业录用或淘汰的可能性。当面试官对求职者提出此种问题，表明面试官对求职者是基本肯定的，只是还不能下决定是否最终录用。如果求职者还应聘过其他企业，请最好选择相关联的岗位或行业回答。一般而言，如果应聘过其他企业，一定要说自己拿到了其他企业的录用通知，如果其他的行

业影响力高于现在面试的企业，无疑可以加大求职者自身的筹码，有时甚至可以因此拿到该企业的顶级录用通知，如果行业影响力低于现在面试的企业，如果回答没有拿到录用通知，则会给面试官一种误导：连这家企业都没有给录用通知，我们如果给录用通知了，岂不是说明不如这家企业。

（11）这是我的名片，你随时可以联系我

在面试结束时，面试官起身将求职者送到门口，并主动与求职者握手，提供给求职者名片或者自己的个人电话，希望日后多加联系。此时，求职者一定要明白，面试官已经对自己非常肯定了，这是被录用的信息，因为很少有面试官会放下身段，对一个已经没有录用可能的求职者还如此“厚爱”。很多面试官在整个面试过程中会一直塑造出一种即将录用求职者的假象。例如，“你如果来到我们公司，有可能会比较忙”等模棱两可的表述，但如果面试官亲手将名片呈交，言谈中也流露出兴奋、积极的意向和表情，一般是表明了一种接纳求职者的态度。

（12）你担任职务很多，时间安排得过来吗

对于有些职位，如销售岗位等，学校的积极分子往往更具优势，但在应聘研发类岗位时，却并不一定占优势。面试官提出此类问题，其实就是对一些在学校当“领导”的学生的一种反感，大量的社交活动很有可能占据学业时间，从而导致专业基础不牢固等。所以，针对上述问题，求职者在回答时，一定要告诉面试官，自己参与组织的“课外活动”并没有影响到自己的专业技能。

（13）面试结束后，面试官说“我们有消息会通知你的”

一般而言，面试官让求职者等通知，有多种可能性：①无录取意向；②面试官不是负责人，还需要请示领导；③公司对求职者不是特别满意，希望再多面试一些人，如果有比求职者更好的就不用求职者了，没有的话会录取；④公司需要对面试过并留下来的人进行重新选择，可能会安排二次面试。所以，当面试官说这话时，表明此时成功的可能性不大，至少这一次不能给予肯定的回复，相反如果对方热情地和求职者握手言别，再加一句“欢迎你应聘本公司”，此时一般就有录用的可能了。

（14）我们会在几天后联系你

一般而言，面试官说出这句话，表明了面试官对求职者还是很感兴趣的，尤其是当面试官仔细询问求职者所能接受的薪资情况等相关情况后，否则他们会尽快结束面谈，而不是多此一举。

（15）面试官认为该结束面试时的暗语

一般而言，求职者自我介绍之后，面试官会相应地提出各类问题，然后转向谈工作。面试官先会把工作内容和职责介绍一番，接着让求职者谈谈今后工作的打算和设想，然后双方会谈及福利待遇问题，这些都是高潮话题，谈完之后求职者就应该主动做出告辞的姿态，不要盲目拖延时间。

面试官认为该结束面试时，往往会说以下暗示的话语来提醒求职者：

1）我很感激你对我们公司这项工作的关注。

2）真难为你了，跑了这么多路，多谢了。

3）谢谢你对我们招聘工作的关心，我们一旦做出决定就会立即通知你。

4）你的情况我们已经了解。你知道，在做出最后决定之前我们还要面试几位申请人。

此时，求职者应该主动站起身来，露出微笑，和面试官握手告辞，并且谢谢他，然后有礼貌地退出面试室。适时离场还包括不要在面试官结束谈话之前表现出浮躁不安、急欲离去或另去赴约的样子，过早地想离场会使面试官认为求职者应聘没有诚意或做事情没有耐心。

（16）如果让你调到其他岗位，你愿意吗

有些企业招收岗位和人员较多，在面试中，当听到面试官说出此话时，言外之意是该岗位也

许已经“人满为患”或“名花有主”了，但企业对求职者兴趣不减，还是很希望求职者能成为企业的一员。面对这种提问，求职者应该迅速做出反应，如果认为对方是个不错的企业，求职者对新的岗位又有一定的把握，也可以先进单位再选岗位；如果对方企业情况一般，新岗位又不太适合自己，最好当面回答不行。

（17）你能来实习吗

对于实习这种敏感的问题，面试官一般是不会轻易提及的，除非是确实对求职者很感兴趣，相中求职者了。当求职者遇到这种情况时，一定要清楚面试官的意图，他希望求职者能够表态，如果确实可以去实习，一定及时地在面试官面前表达出来，这无疑可以给予自己更多的机会。

（18）你什么时候能到岗

当面试官问及到岗的时间时，表明面试官已经同意给录用通知了，此时只是为了确定求职者是否能够及时到岗并开始工作。如果确有难题千万不要遮遮掩掩，含糊其辞，说清楚情况，诚实守信。

针对面试中存在的这种暗语，求职者在面试过程中，一定不要“很傻很天真”，要多留心，多推敲面试官的深意，仔细想想其中的“潜台词”，从而将面试官的那点“小伎俩”看透。

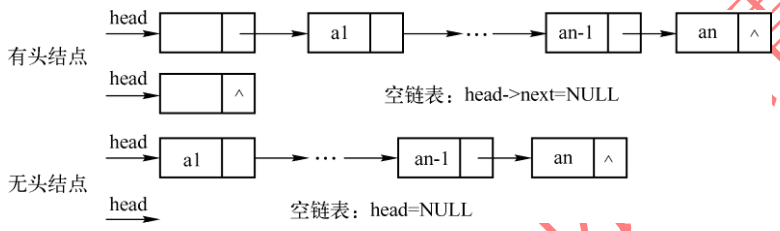
面试、笔试真题解析篇

面试、笔试真题解析篇主要针对近三年以来近百家顶级 IT 企业的面试、笔试算法真题而设计，这些企业涉及业务包括系统软件、搜索引擎、电子商务、手机 APP、安全关键软件等，面试、笔试真题难易适中，覆盖面广，非常具有代表性与参考性。本篇对这些真题进行了合理划分与归类（包括链表、栈、队列、二叉树、数组、字符串、大数据处理等内容），并且对其进行了庖丁解牛式的分析与讲解，针对真题中涉及的部分重难点问题，本篇都进行了适当扩展与延伸，力求对知识点的讲解清晰而不紊乱，全面而不啰唆，使得读者不仅能够通过本书获取到求职的知识，同时更有针对性地进行求职准备，最终能够收获一份满意的工作。

第 1 章　链　　表

链表作为最基本的数据结构，它不仅在实际应用中有着非常重要的作用，而且也是程序员面试、笔试中必考的内容。具体而言，它的存储特点为：可以用任意一组存储单元来存储单链表中的数据元素（存储单元可以是不连续的），而且除了存储每个数据元素 ai 外，还必须存储指示其直接后继元素的信息。这两部分信息组成的数据元素 ai 的存储映像称为结点。N 个结点链在一块被称为链表，当结点只包含其后继结点的信息的链表就被称为单链表，而链表的第一个结点通常被称为头结点。

对于单链表，又可以将其分为有头结点的单链表和无头结点的单链表，如下图所示。



在单链表的开始结点之前附设一个类型相同的结点，称之为头结点，头结点的数据域可以不存储任何信息（也可以存放如线性表的长度等附加信息），头结点的指针域存储指向开始结点的指针（即第一个元素结点的存储位置）。**需要注意的是，在 Java 中没有指针的概念，而类似指针的功能都是通过引用来实现的，为了便于理解，我们仍然使用指针（可以认为引用与指针是类似的）来进行描述，而在实现的代码中，都是通过引用来建立结点之间的关系。**

具体而言，头结点的作用主要有以下两点：

1）对于带头结点的链表，当在链表的任何结点之前插入新结点或删除链表中任何结点时，所要做的都是修改前一个结点的指针域，因为任何结点都有前驱结点。若链表没有头结点，则首元素结点没有前驱结点，在其前面插入结点或删除该结点时操作会复杂些，需要进行特殊的处理。

2）对于带头结点的链表，链表的头指针是指向头结点的非空指针，因此，对空链表与非空链表的处理是一样的。

由于头结点有诸多的优点，因此，本章中所介绍的算法都使用了带头结点的单链表。

如下是一个单链表数据结构的定义示例：

```
class LNode
{
    int   data;                          //数据域
    LNode   next;                        //下一个结点的引用
}
```

1.1 如何实现链表的逆序

【出自 TX 笔试题】

难度系数：★★★☆☆　　被考察系数：★★★★☆

题目描述：

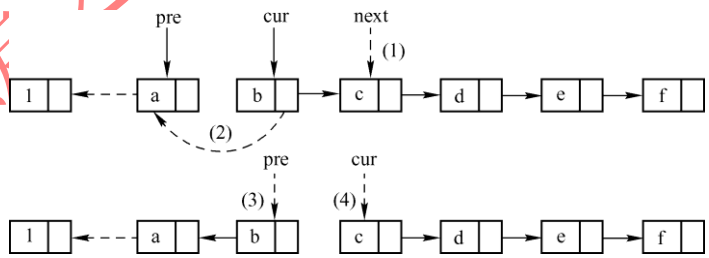
给定一个带头结点的单链表，请将其逆序。即如果单链表原来为 head→1→2→3→4→5→6→7，则逆序后变为 head→7→6→5→4→3→2→1。

分析与解答：

由于单链表与数组不同，单链表中每个结点的地址都存储在其前驱结点的指针域中，因此，对单链表中任何一个结点的访问只能从链表的头指针开始进行遍历。在对链表的操作过程中，需要特别注意在修改结点指针域的时候，记录下后继结点的地址，否则会丢失后继结点。

方法一：就地逆序

主要思路：在遍历链表时，修改当前结点的指针域的指向，让其指向它的前驱结点。为此，需要用一个指针变量来保存前驱结点的地址。此外，为了在调整当前结点指针域的指向后还能找到后继结点，还需要另外一个指针变量来保存后继结点的地址，在所有的结点都被保存好以后就可以直接完成指针的逆序了。除此之外，还需要特别注意对链表首尾结点的特殊处理。具体实现方式如下图所示。



在上图中，假设当前已经遍历到 cur 结点，由于它所有的前驱结点都已经完成了逆序操作，因此，只需要使 cur.next=pre 即可完成逆序操作。在此之前，为了能够记录当前结点的后继结点的地址，需要用一个额外的指针 next 来保存后继结点的信息，通过上图（1）～（4）四步把实线的指针调整为虚线的指针就可以完成当前结点的逆序；当前结点完成逆序后，通过向后移动指针来对后续的结点用同样的方法进行逆序操作。算法实现如下：

```
public class Test
{
    /*方法功能：对单链表进行逆序输入参数：head:链表头结点*/
    public static void Reverse(LNode head)
```

```
    {
        // 判断链表是否为空
        if (head == null || head.next == null)
            return;
        LNode pre = null;                           // 前驱结点
        LNode cur = null;                           // 当前结点

        LNode next = null;                          // 后继结点
        // 把链表首结点变为尾结点
        cur = head.next;
        next = cur.next;
        cur.next = null;
        pre = cur;
        cur = next;
        // 使当前遍历到的结点 cur 指向其前驱结点
        while (cur.next != null)
        {
            next = cur.next;
            cur.next = pre;
            pre = cur;
            cur = cur.next;
            cur = next;
        }
        // 结点最后一个结点指向倒数第二个结点
        cur.next = pre;
        // 链表的头结点指向原来链表的尾结点
        head.next = cur;
    }

    public static void main(String[] args)
    {
        // 链表头结点
        LNode head = new LNode();
        head.next = null;
        LNode tmp = null;
        LNode cur = head;
        // 构造单链表
        for (int i; i< 8; i++) {
            tmp = new LNode();
            tmp.data = i;
            tmp.next = null;
            cur.next = tmp;
            cur = tmp;
        }
        System.out.print("逆序前：");
        for (cur = head.next; cur != null; cur = cur.next)
            System.out.print(cur.data + " ");
        System.out.print("\n 逆序后：");
        Reverse(head);
        for (cur = head.next; cur != null; cur = cur.next)
            System.out.print(cur.data + " ");
    }
}
```

程序的运行结果为

```
逆序前：1  2  3  4  5  6  7
逆序后：7  6  5  4  3  2  1
```

算法性能分析：

以上这种方法只需要对链表进行一次遍历，因此，时间复杂度为 O(N)。其中，N 为链表的长度。但是需要常数个额外的变量来保存当前结点的前驱结点与后继结点，因此，空间复杂度为 O(1)。

方法二：递归法

假定原链表为 1→2→3→4→5→6→7，递归法的主要思路为：先逆序除第一个结点以外的子链表（将 1→**2→3→4→5→6→7** 变为 1→**7→6→5→4→3→2**），接着把结点 1 添加到逆序的子链表的后面（1→**2→3→4→5→6→7** 变为 7→**6→5→4→3→2→1**）。同理，在逆序链表 2→3→4→5→6→7 时，也是先逆序子链表 3→4→5→6→7（逆序为 2→**7→6→5→4→3**），接着实现链表的整体逆序（2→**7→6→5→4→3** 转换为 7→**6→5→4→3→2**）。实现代码如下：

```
/*
**方法功能：对不带头结点的单链表进行逆序
**输入参数：firstRef:链表头结点
*/
private static LNode RecursiveReverse(LNode head)
{
    //如果链表为空或者链表中只有一个元素
    if(head==null || head.next==null)
        return head;
    else
    {
        //反转后面的结点
        LNode newhead=RecursiveReverse(head.next);
        //把当前遍历的结点加到后面结点逆序后链表的尾部
        head.next.next=head;
        head.next=null;
        return newhead;
    }
}
/*
**方法功能：对带头结点的单链表进行逆序
**输入参数：head:链表头结点
*/
public static void Reverse(LNode head)
{
    if (head == null)
        return;
    //获取链表第一个结点
    LNode firstNode=head.next;
    //对链表进行逆序
    LNode newhead=RecursiveReverse(firstNode);
    //头结点指向逆序后链表的第一个结点
    head.next=newhead;
}
```

算法性能分析：

由于递归法也只需要对链表进行一次遍历，因此，算法的时间复杂度也为 O(N)。其中，N 为链表的长度。递归法的主要优点是：思路比较直观，容易理解，而且也不需要保存前驱结点的地址；缺点是：算法实现的难度较大。此外，由于递归法需要不断地调用自己，需要额外的压栈与弹栈操作，因此，与方法一相比性能会有所下降。

方法三：插入法

插入法的主要思路：从链表的第二个结点开始，把遍历到的结点插入到头结点的后面，直到遍历结束。假定原链表为 head→1→2→3→4→5→6→7，在遍历到 2 时，将其插入到头结点后，链表变为 head→2→1→3→4→5→6→7，同理将后序遍历到的所有结点都插入到头结点 head 后，就可以实现链表的逆序。实现代码如下：

```
public static void Reverse(LNode head)
{
    // 判断链表是否为空
    if (head == null || head.next == null)
        return;
    LNode cur = null;                    // 当前结点
    LNode next = null;                   // 后继结点
    cur = head.next.next;
    // 设置链表第一个结点为尾结点
    head.next.next = null;
    // 把遍历到结点插入到头结点的后面
    while (cur != null) {
        next = cur.next;
        cur.next = head.next;
        head.next = cur;
        cur = next;
    }
}
```


算法性能分析：

以上这种方法也只需要对单链表进行一次遍历，因此，时间复杂度为 O(N)。其中，N 为链表的长度。与方法一相比，这种方法不需要保存前驱结点的地址，与方法二相比，这种方法不需要递归地调用，效率更高。

引申：①对不带头结点的单链表进行逆序；

②从尾到头输出链表。

分析与解答：

对不带头结点的单链表的逆序，读者可以自己练习（方法二已经实现了递归的方法），这里主要介绍单链表逆向输出的方法。

方法一：就地逆序+顺序输出

首先对链表进行逆序，然后顺序输出逆序后的链表。这种方法的缺点是改变了链表原来的结构。

方法二：逆序+顺序输出

申请新的存储空间，对链表进行逆序，然后顺序输出逆序后的链表。逆序的主要思路为：每当遍历到一个结点的时候，申请一块新的存储空间来存储这个结点的数据域，同时把新结点插入到新的链表的头结点后。这种方法的缺点是需要申请额外的存储空间。

方法三：递归输出

递归输出的主要思路：先输出除当前结点外的后继子链表，然后输出当前结点，假如链表为 1→2→3→4→5→6→7，那么先输出 2→3→4→5→6→7，再输出 1。同理，对于链表 2→3→4→5→6→7，也是先输出 3→4→5→6→7，接着输出 2，直到遍历到链表的最后一个结点 7 的时候会输出结点 7，然后递归地输出 6，5，…，1。实现代码如下：

```
public void ReversePrint (LNode firstNode) {
    if(firstNode==null)
        return;
    ReversePrint (firstNode.next);
    System.out.print(firstNode.data+" ");
}
```

程序的运行结果为

```
顺序输出：0  1  2  3  4  5  6  7
逆序输出：7  6  5  4  3  2  1  0
```


算法性能分析：

以上这种方法只需要对链表进行一次遍历，因此，时间复杂度为 O(N)。其中，N 为链表的长

度。

1.2 如何从无序链表中移除重复项

【出自 GG 面试题】

难度系数：★★★☆☆　　被考察系数：★★★★☆

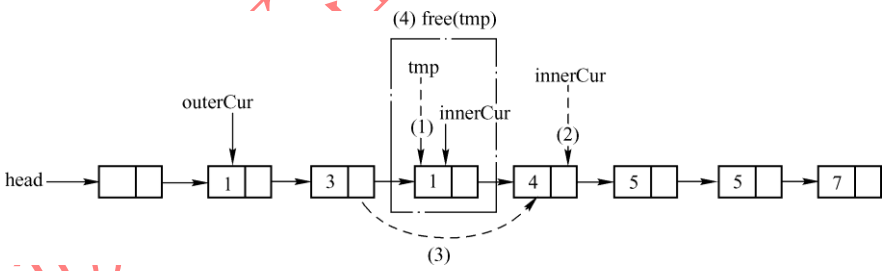
题目描述：

给定一个没有排序的链表，去掉其重复项，并保留原顺序，如链表 1→3→1→5→5→7，去掉重复项后变为 1→3→5→7。

分析与解答：

方法一：顺序删除

主要思路为：通过双重循环直接在链表上执行删除操作。外层循环用一个指针从第一个结点开始遍历整个链表，然后内层循环用另外一个指针遍历其余结点，将与外层循环遍历到的指针所指结点的数据域相同的结点删除，如下图所示。



假设外层循环从 outerCur 开始遍历，当内层循环指针 innerCur 遍历到上图实线所示的位置（outerCur.data==innerCur.data）时，此时需要把 innerCur 指向的结点删除。具体步骤如下：

1）用 tmp 记录待删除的结点的地址。

2）为了能够在删除 tmp 结点后继续遍历链表中其余的结点，使 innerCur 指针指向它的后继结点：innerCur=innerCur.next。

3）从链表中删除 tmp 结点。

实现代码如下：

```
public class Test
{
    /*
     **方法功能：对带头结点的无序单链表删除重复的结点
     **输入参数：head:链表头结点
    */
    public static void removeDup(LNode head)
```

```
    {
        if (head == null || head.next == null)
            return;
        LNode outerCur = head.next;          // 用于外层循环，指向链表第一个结点
        LNode innerCur = null;                    // 内层循环用来遍历 ourterCur 后面的结点
        LNode innerPre = null;               // innerCur 的前驱结点
        for (; outerCur != null; outerCur = outerCur.next)
        {
            for (innerCur = outerCur.next, innerPre = outerCur; innerCur != null;)
            {
                // 找到重复的结点并删除
                if (outerCur.data == innerCur.data) {
                    innerPre.next = innerCur.next;
                    innerCur = innerCur.next;
                } else {
                    innerPre = innerCur;
                    innerCur = innerCur.next;
                }
            }
        }
    }

    public static void main(String[] args)
    {
        int i = 1;
        LNode head = new LNode();
        head.next = null;
        LNode tmp = null;
        LNode cur = head;
        for (; i< 7; i++) {
            tmp = new LNode();
            if (i % 2 == 0)
                tmp.data = i + 1;
            else if (i % 3 == 0)
                tmp.data = i - 2;
            else
                tmp.data = i;
            tmp.next = null;
            cur.next = tmp;
            cur = tmp;
        }
        System.out.print("删除重复结点前：");
        for (cur = head.next; cur != null; cur = cur.next)

            System.out.print(cur.data + " ");
        removeDup(head);
        System.out.print("\n 删除重复结点后：");
        for (cur = head.next; cur != null; cur = cur.next)
            System.out.print(cur.data + " ");
    }
}
```

程序的运行结果为

```
删除重复结点前：1  3  1  5  5  7
删除重复结点后：1  3  5  7
```

算法性能分析：

由于这种方法采用双重循环对链表进行遍历，因此，时间复杂度为 O(N^2)。其中，N 为链表的长度。在遍历链表的过程中，使用了常量个额外的指针变量来保存当前遍历的结点、前驱结点和被删除的结点，因此，空间复杂度为 O(1)。

方法二：递归法

主要思路为：对于结点 cur，首先递归地删除以 cur.next 为首的子链表中重复的结点，接着从以 cur.next 为首的子链表中找出与 cur 有着相同数据域的结点并删除。实现代码如下：

```
private static LNode removeDupRecursion(LNode head)
{
    if (head.next == null)
        return head;
    LNode pointer = null;
    LNode cur = head;
    // 对以 head.next 为首的子链表删除重复的结点
    head.next = removeDupRecursion(head.next);
    pointer = head.next;
    // 找出以 head.next 为首的子链表中与 head 结点相同的结点并删除
    while (pointer != null)
    {
        if (head.data == pointer.data)
        {
            cur.next = pointer.next;
            pointer = cur.next;
        } else
        {
            pointer = pointer.next;
            cur = cur.next;
        }
    }
    return head;
}

/*
* 方法功能：对带头结点的单链表删除重复结点输入参数：head:链表头结点
*/
public static void removeDup(LNode head)
{
    if (head == null)

        return;
    head.next = removeDupRecursion(head.next);
}
```

算法性能分析：

这种方法与方法一类似，从本质上而言，由于这种方法需要对链表进行双重遍历，因此，时间复杂度为 O(N^2)。其中，N 为链表的长度。由于递归法会增加许多额外的函数调用，因此，

从理论上讲，该方法效率比方法一低。

方法三：空间换时间

通常情况下，为了降低时间复杂度，往往在条件允许的情况下，通过使用辅助空间实现。具体而言，主要思路如下。
1）建立一个 HashSet，HashSet 中的内容为已经遍历过的结点内容，并将其初始化为空。
2）从头开始遍历链表中的所以结点，存在以下两种可能性：
① 如果结点内容已经在 HashSet 中，则删除此结点，继续向后遍历。
② 如果结点内容不在 HashSet 中，则保留此结点，将此结点内容添加到 HashSet 中，继续向后遍历。

引申：如何从有序链表中移除重复项。

分析与解答：

上述介绍的方法也适用于链表有序的情况，但是由于以上方法没有充分利用到链表有序这个条件，因此，算法的性能肯定不是最优的。本题中，由于链表具有有序性，因此，不需要对链表进行两次遍历。所以，有如下思路：用 cur 指向链表第一个结点，此时需要分为以下两种情况讨论。
1）如果 cur.data==cur.next.data，那么删除 cur.next 结点。
2）如果 cur.data!= cur.next.data，那么 cur=cur.next，继续遍历其余结点。

1.2 如何计算两个单链表所代表的数之和

【出自 HW 笔试题】

难度系数：★★★☆☆　　被考察系数：★★★★☆

题目描述：

给定两个单链表，链表的每个结点代表一位数，计算两个数的和。例如，输入链表（3→1→5）和链表（5→9→2），输出：8→0→8，即 513+295=808，注意个位数在链表头。

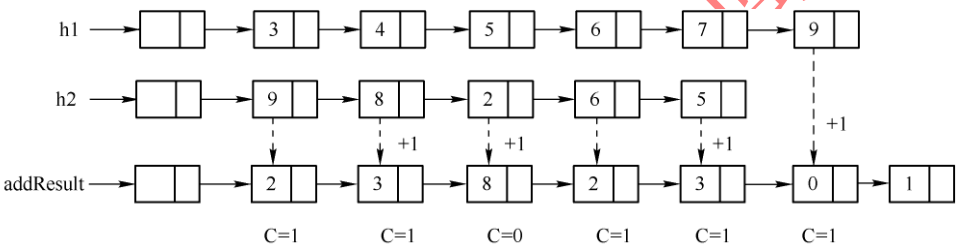
分析与解答：

方法一：整数相加法

主要思路为：分别遍历两个链表，求出两个链表所代表的整数的值，然后把这两个整数进行相加，最后把它们的和用链表的形式表示出来。这种方法的优点是计算简单，但是有个非常大的缺点：当链表所代表的数很大时（超出了 long int 的表示范围），就无法使用这种方法了。

方法二：链表相加法

主要思路为：对链表中的结点直接进行相加操作，把相加的和存储到新的链表中对应的结点中，同时还要记录结点相加后的进位，如下图所示。



使用这种方法需要注意如下几个问题：①每组结点进行相加后需要记录其是否有进位；②如果两个链表 H1 与 H2 的长度不同（长度分别为 L1 和 L2，且 L1<L2），当对链表的第 L1 位计算完成后，接下来只需要考虑链表 L2 剩余的结点的值（需要考虑进位）；③对链表所有结点都完成计算后，还需要考虑此时是否还有进位，如果有进位，则需要增加新的结点，此结点的数据域为 1。实现代码如下：

```
public class Test
{
    /*
    **方法功能：对两个带头结点的单链表所代表的数相加
    **输入参数：h1:第一个链表头结点；h2:第二个链表头结点
    **返回值：相加后链表的头结点
    */
    public static LNode add(LNode h1, LNode h2)
    {
        if(h1==null || h1.next==null)
            return h2;
        if(h2==null || h2.next==null)
            return h1;
        int c=0;                                    //用来记录进位
        int sum=0;                                  //用来记录两个结点相加的值
        LNode p1=h1.next;                           //用来遍历 h1
        LNode p2=h2.next;                           //用来遍历 h2
        LNode tmp=null;                             //用来指向新创建的存储相加和的结点
        LNode resultHead=new LNode();               //相加后链表头结点
        resultHead.next=null;
        LNode p=resultHead;                         //用来指向链表 resultHead 最后一个结点
        while(p1!=null && p2!=null)
```

```
        {
            tmp=new LNode();
            tmp.next=null;
            sum=p1.data+p2.data+c;
            tmp.data=sum%10;                    //两结点相加和
            c=sum/10;                           //进位
            p.next=tmp;
            p=tmp;
            p1=p1.next;
            p2=p2.next;
        }
        //链表 h2 比 h1 长，接下来只需要考虑 h2 剩余结点的值
        if(p1==null)
        {

            while( p2!=null)
            {
                tmp=new LNode();
                tmp.next=null;
                sum=p2.data+c;
                tmp.data=sum%10;
                c=sum/10;
                p.next=tmp;
                p=tmp;
                p2=p2.next;
            }
        }
        //链表 h1 比 h2 长，接下来只需要考虑 h1 剩余结点的值
        if(p2==null)
        {
            while( p1!=null)
            {
                tmp=new LNode();
                tmp.next=null;
                sum=p1.data+c;
                tmp.data=sum%10;
                c=sum/10;
                p.next=tmp;
                p=tmp;
                p1=p1.next;
            }
        }
        //如果计算完成后还有进位，则增加新的结点
        if(c==1)
        {
            tmp=new LNode();
            tmp.next=null;
            tmp.data=1;
            p.next=tmp;
        }
        return resultHead;
    }

    public static void main(String[] args)
    {
        int i=1;
        LNode head1=new LNode();
        head1.next=null;
```

```
            LNode head2=new LNode();
            head2.next=null;
            LNode tmp=null;
            LNode cur=head1;
            LNode addResult=null;
            //构造第一个链表
            for(;i<7;i++)
            {
                tmp=new LNode();
                tmp.data=i+2;
                tmp.next=null;
                cur.next=tmp;

                cur=tmp;
            }
            cur=head2;
            //构造第二个链表
            for(i=9;i>4;i--)
            {
                tmp=new LNode();
                tmp.data=i;
                tmp.next=null;
                cur.next=tmp;
                cur=tmp;
            }
            System.out.print("Head1：  ");
            for(cur=head1.next;cur!=null;cur=cur.next)
                System.out.print(cur.data+" ");
            System.out.print("\n Head2：  ");
            for(cur=head2.next;cur!=null;cur=cur.next)
                System.out.print(cur.data+" ");
            addResult=add(head1,head2);
            System.out.print("\n 相加后：");
            for(cur=addResult.next;cur!=null;cur=cur.next)
                System.out.print(cur.data+" ");
        }
    }
```

程序的运行结果为

```
Head1：  3  4  5  6  7  8
Head2：  9  8  7  6  5
相加后：2  3  3  3  3  9
```

运行结果分析：

前 5 位可以按照整数相加的方法依次从左到右进行计算，第 5 位 7+5+1（进位）的值为 3，进位为 1。此时，Head2 已经遍历结束，由于 Head1 还有结点没有被遍历，所以，依次接着遍历 Head1 剩余的结点：8+1(进位)=9，没有进位。因此，运行代码可以得到上述结果。

算法性能分析：

由于这种方法需要对两个链表都进行遍历，因此，时间复杂度为 O(N)。其中，N 为较长的链表的长度，由于计算结果保存在一个新的链表中，因此，空间复杂度也为 O(N)。

1.4 如何对链表进行重新排序

【出自 WR 笔试题】

难度系数：★★★☆☆　　被考察系数：★★★★☆

题目描述：

给定链表 L0→L1→L2→…→Ln-1→Ln，把链表重新排序为 L0→Ln→L1→Ln-1→L2→Ln-2→…。要求：①在原来链表的基础上进行排序，即不能申请新的结点；②只能修改结点的 next 域，不能修改数据域。

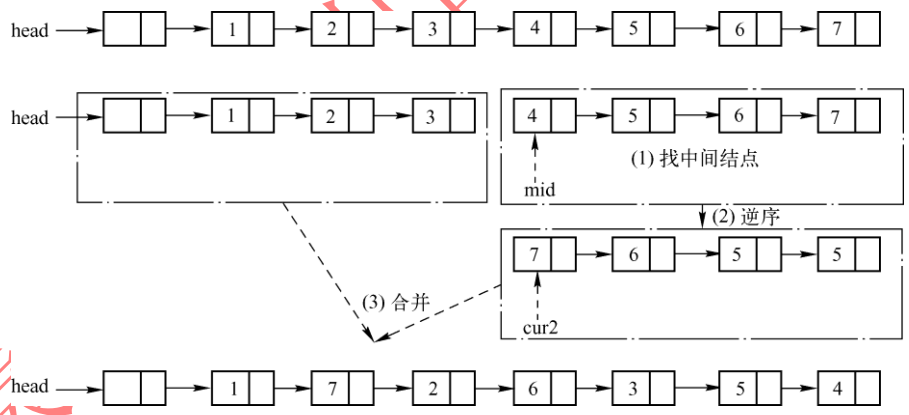
分析与解答：

主要思路为：

1）首先找到链表的中间结点。

2）对链表的后半部分子链表进行逆序。

3）把链表的前半部分子链表与逆序后的后半部分子链表进行合并，合并的思路为：分别从两个链表各取一个结点进行合并。实现方法如下图所示。



实现代码如下：

```
public class Test
{
    /*
    ** 方法功能：找出链表 head 的中间结点，把链表从中间断成两个子链表
    ** 输入参数：head:链表头结点
    ** 返回值：链表中间结点
    */
    private static LNode FindMiddleNode(LNode head)
    {
        if(head==null || head.next==null)
            return head;
        LNode fast = head;                          //遍历链表时每次向前走两步
```

```
        LNode slow = head;                          //遍历链表时每次向前走一步
        LNode slowPre=head;
        //当 fast 到链表尾时，slow 恰好指向链表的中间结点
        while(fast != null&&fast.next!=null){
            slowPre=slow;
        slow=slow.next;
        fast=fast.next.next;
    }
        //把链表断开成两个独立的子链表
        slowPre.next=null;
        return slow;
    }

    /*
    ** 方法功能：对不带头结点的单链表翻转
    ** 输入参数：head:链表头结点
    */
    private static LNode Reverse(LNode head)
    {
        if(head==null || head.next==null)

            return head;
        LNode pre=head;                             //前驱结点
        LNode cur=head.next;                        //当前结点
        LNode next;                                 //后继结点
        pre.next=null;

        //使当前遍历到的结点 cur 指向其前驱结点
        while(cur!=null)
        {
            next=cur.next;
            cur.next=pre;
            pre=cur;
            cur=cur.next;
            cur=next;
        }
        return pre;
    }

    /*
    ** 方法功能：对链表进行排序
    ** 输入参数：head:链表头结点
    */
    public static void Reorder(LNode head)
    {
        if(head==null || head.next==null)
            return;
        //前半部分链表第一个结点
        LNode cur1=head.next;
        LNode mid=FindMiddleNode(head.next);
        //后半部分链表逆序后的第一个结点
        LNode cur2=Reverse(mid);
        LNode tmp=null;
        //合并两个链表
        while(cur1.next!=null)
        {
            tmp=cur1.next;
            cur1.next=cur2;
```

```
                cur1=tmp;

                tmp=cur2.next;
                cur2.next=cur1;
                cur2=tmp;
            }
            cur1.next=cur2;
        }

        public static void main(String[] args)
        {
            int i=1;
            LNode head=new LNode();
            head.next=null;
            LNode tmp=null;
            LNode cur=head;
            //构造第一个链表
            for(;i<8;i++){

                tmp=new LNode();
                tmp.data=i;
                tmp.next=null;
                cur.next=tmp;
                cur=tmp;
            }
            System.out.print("排序前：  ");
            for(cur=head.next;cur!=null;cur=cur.next)
                System.out.print(cur.data+" ");
            Reorder(head);
            System.out.print("\n 排序后：  ");
            for(cur=head.next;cur!=null;cur=cur.next)
                System.out.print(cur.data+" ");
        }
    }
```

程序的运行结果为

```
    排序前：  1  2  3  4  5  6  7
    排序后：  1  7  2  6  3  5  4
```

算法性能分析：

查找链表的中间结点的方法的时间复杂度为 O(N)，逆序子链表的时间复杂度也为 O(N)，合并两个子链表的时间复杂度也为 O(N)。因此，整个方法的时间复杂度为 O(N)，其中，N 表示链表的长度。由于这种方法只用了常数个额外指针变量，因此，空间复杂度为 O(1)。

引申：如何查找链表的中间结点

分析与解答：

主要思路：用两个指针从链表的第一个结点开始同时遍历结点，一个快指针每次走两步，另

外一个慢指针每次走一步；当快指针先到链表尾部时，慢指针则恰好到达链表中部（快指针到链表尾部时，当链表长度为奇数时，慢指针指向的即是链表中间指针，当链表长度为偶数时，慢指针指向的结点和慢指针指向结点的下一个结点都是链表的中间结点）。上面的代码 FindMiddleNode 就是用来求链表的中间结点的。

1.5 如何找出单链表中的倒数第 k 个元素

【出自 WR 笔试题】

难度系数：★★★☆☆　　被考察系数：★★★★★

题目描述：

找出单链表中的倒数第 k 个元素，如给定单链表：1→2→3→4→5→6→7，则单链表的倒数第 k=3 个元素为 5。

分析与解答：

方法一：顺序遍历两遍法

主要思路：首先遍历一遍单链表，求出整个单链表的长度 n，然后把求倒数第 k 个元素转换为求顺数第 n - k 个元素，再去遍历一次单链表就可以得到结果。但是该方法需要对单链表进行两次遍历。

方法二：快慢指针法

由于单链表只能从头到尾依次访问链表的各个结点，因此如果要找单链表的倒数第 k 个元素，也只能从头到尾进行遍历查找。在查找过程中，设置两个指针，让其中一个指针比另外一个指针先前移 k 步，然后两个指针同时往前移动。循环直到先行的指针值为 null 时，另一个指针所指的位置就是所要找的位置。程序代码如下：

```
public class Test
{
    /* 构造一个单链表 */
    public static LNode ConstructList(){
        int i=1;
        LNode head=new LNode();
        head.next=null;
        LNode tmp=null;
        LNode cur=head;
        //构造第一个链表
        for(;i<8;i++){
```

```
                tmp=new LNode();
                tmp.data=i;
                tmp.next=null;
                cur.next=tmp;
                cur=tmp;
            }
            return head;
        }
        /* 顺序打印单链表结点的数据 */
        public static void PrintList(LNode head){
            for(LNode cur=head.next;cur!=null;cur=cur.next)
                System.out.print(cur.data+" ");
        }

        /*
        ** 方法功能：找出链表倒数第 k 个结点
        ** 输入参数：head:链表头结点
        ** 返回值：倒数第 k 个结点
        */
        public static LNode FindLastK(LNode head,int k){
            if(head==null || head.next==null)
                return head;
            LNode slow,fast;
            slow=fast=head.next;
            int i;
            for(i=0; i<k&&fast!=null ;++i){                    //前移 k 步
                fast=fast.next;
            }
            //判断 k 是否已超出链表长度
            if(i<k)
                return null;
            while(fast!=null){
                slow=slow.next;
                fast=fast.next;
            }
            return slow;

        }

        public static void main(String[] args)
        {
            LNode head=ConstructList();
            LNode result=null;
            System.out.print("链表：");
            PrintList(head);
            result=FindLastK(head,3);
            if (result!=null)
                System.out.print("\n 链表倒数第三个元素为："+result.data);
        }
    }
```

程序的运行结果为

```
链表：  1   2   3   4   5   6   7
链表倒数第三个元素为：5
```

算法性能分析：

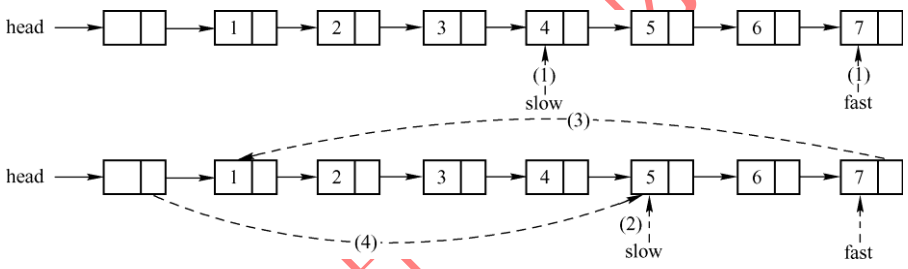
这种方法只需要对链表进行一次遍历，因此，时间复杂度为 O(N)。另外，由于只需要常量个指针变量来保存结点的地址信息，因此，空间复杂度为 O(1)。

引申：如何将单链表向右旋转 k 个位置。

题目描述：给定单链表 1→2→3→4→5→6→7，k=3，那么旋转后的单链表变为 5→6→7→1→2→3→4。

分析与解答：

主要思路为：①首先找到链表倒数第 k+1 个结点 slow 和尾结点 fast（如下图所示）；②把链表断开为两个子链表，其中后半部分子链表结点的个数为 k；③使原链表的尾结点指向链表的第一个结点；④使链表的头结点指向原链表倒数第 k 个结点。



实现代码如下：

```
public class Test
{
    /*      方法功能：把链表右旋 k 个位置  */
    public static void RotateK(LNode head,int k)
    {
        if(head==null || head.next==null)
            return ;
        //fast 指针先走 k 步，然后与 slow 指针同时向后走
        LNode slow,fast,tmp;
        slow=fast=head.next;

        int i;
        for(i=0; i<k&&fast!=null ;++i){ //前移 k 步
            fast=fast.next;
        }
        //判断 k 是否已超出链表长度
        if(i<k)
            return ;
        //循环结束后 slow 指向链表倒数第 k+1 个元素，fast 指向链表最后一个元素
        while(fast.next!=null){
            slow=slow.next;
            fast=fast.next;
        }
        tmp=slow;
```

```
            slow=slow.next;
            tmp.next=null;                          //如上图（2）
            fast.next=head.next;                    //如上图（3）
            head.next=slow;                         //如上图（4）
        }

        public static LNode ConstructList()
        {
            int i=1;
            LNode head=new LNode();
            head.next=null;
            LNode tmp=null;
            LNode cur=head;
            //构造第一个链表
            for(;i<8;i++)
            {
                tmp=new LNode();
                tmp.data=i;
                tmp.next=null;
                cur.next=tmp;
                cur=tmp;
            }
            return head;
        }

        /* 顺序打印单链表结点的数据 */
        public static void PrintList(LNode head)
        {
            for(LNode cur=head.next;cur!=null;cur=cur.next)
                System.out.print(cur.data+" ");
        }

        public static void main(String[] args)
        {
            LNode head=ConstructList();
            System.out.print("旋转前：  ");
            PrintList(head);
            RotateK(head,3);
            System.out.print("\n 旋转后:");
            PrintList(head);
        }
    }
```

程序的运行结果为

```
旋转前:  1  2  3  4  5  6  7
旋转后:  5  6  7  1  2  3  4
```

算法性能分析：

这种方法只需要对链表进行一次遍历，因此，时间复杂度为 O(n)。另外，由于只需要几个指针变量来保存结点的地址信息，因此，空间复杂度为 O(1)。

1.6 如何检测一个较大的单链表是否有环

【出自 ALBB 笔试题】

难度系数：★★★★☆　　被考察系数：★★★★★

题目描述：

单链表有环指的是单链表中某个结点的 next 域指向链表中在它之前的某一个结点，这样在链表的尾部形成一个环形结构。如何判断单链表是否有环存在？

分析与解答：

方法一：蛮力法

定义一个 HashSet 用来存放结点的引用，并将其初始化为空，从链表的头结点开始向后遍历，每遍历到一个结点就判断 HashSet 中是否有这个结点的引用。如果没有，说明这个结点是第一次访问，还没有形成环，那么将这个结点的引用添加到 HashSet 中去。如果在 HashSet 中找到了同样的结点，那么说明这个结点已经被访问过了，于是就形成了环。这种方法的时间复杂度为 O(N)，空间复杂度也为 O(N)。

方法二：快慢指针遍历法

定义两个指针 fast（快）与 slow（慢），两者的初始值都指向链表头，指针 slow 每次前进一步，指针 fast 每次前进两步。两个指针同时向前移动，快指针每移动一次都要跟慢指针比较，如果快指针等于慢指针，就证明这个链表是带环的单向链表，否则证明这个链表是不带环的循环链表。实现代码见后面引申部分。

引申：如果链表存在环，那么如何找出环的入口点？

分析与解答：

当链表有环时，如果知道环的入口点，那么在需要遍历链表或释放链表所占的空间时方法将会非常简单，下面主要介绍查找链表环入口点的思路。

如果单链表有环，那么按照上述方法二的思路，当走得快的指针 fast 与走得慢的指针 slow 相遇时，slow 指针肯定没有遍历完链表，而 fast 指针已经在环内循环了 n 圈（1≤n）。如果 slow 指针走了 s 步，则 fast 指针走了 2s 步（fast 步数还等于 s 加上在环上多转的 n 圈），假

设环长为 r，则满足如下关系表达式：

```
2s = s + nr
```

由此可以得到：s= nr
设整个链表长为 L，入口环与相遇点距离为 x，起点到环入口点的距离为 a。则满足如下关系表达式：

```
 a + x = nr
 a + x = (n – 1)r +r = (n-1)r + L – a
a = (n-1)r + (L – a – x)
```

(L – a – x)为相遇点到环入口点的距离，从链表头到环入口点的距离=(n-1)*环长+相遇点到环入口点的长度，于是从链表头与相遇点分别设一个指针，每次各走一步，两个指针必定相遇，且相遇的第一点为环入口点。实现代码如下：

```
public class Test
{
    /* 构造链表 */
    public static LNode constructList()
    {
        int i=1;
        LNode head=new LNode();
        head.next=null;
        LNode tmp=null;
        LNode cur=head;
        //构造第一个链表
        for(;i<8;i++)
        {
            tmp=new LNode();
            tmp.data=i;
            tmp.next=null;
            cur.next=tmp;
            cur=tmp;
        }
        cur.next=head.next.next.next;
        return head;
    }
    /*
    ** 方法功能：判断单链表是否有环
    ** 输入参数：head:链表头结点
    ** 返回值：null：无环，否则返回 slow 与 fast 相遇点的结点
    */
    public static LNode isLoop(LNode head)
    {
        if(head==null || head.next==null)
            return null;
        //初始 slow 与 fast 都指向链表第一个结点
        LNode slow=head.next;
        LNode fast=head.next;
        while(fast!=null&&fast.next!=null)
        {
            slow=slow.next;
            fast=fast.next.next;
            if(slow==fast)
                return slow;
        }
        return null;
```

```
        }

        /*
        ** 方法功能：找出环的入口点

        ** 输入参数：head: 头结点，meetNode: fast 与 slow 相遇点
        ** 返回值：  null:无环，否则返回 slow 与 fast 指针相遇点的结点
        */
        public static LNode findLoopNode(LNode head,LNode meetNode)
        {
            LNode first=head.next;
            LNode second=meetNode;
            while(first!=second)
            {
                first=first.next;
                second=second.next;
            }
            return first;
        }

        public static void main(String[] args)
        {
            LNode head=constructList();//头结点
            LNode meetNode=isLoop(head);
            LNode loopNode=null;
            if(meetNode!=null)
            {
                System.out.println("有环");
                loopNode=findLoopNode(head,meetNode);
                System.out.println("环的入口点为："+loopNode.data);
            }
            else
            {
                System.out.println("无环");
            }
        }
    }
```

程序的运行结果为

```
有环
环的入口点为：3
```

运行结果分析：

示例代码中给出的链表为 1→2→3→4→5→6→7→3（3 实际代表链表第三个结点）。因此，isLoop 函数返回的结果为两个指针相遇的结点，所以链表有环，通过函数 FindLoopNode 可以获取到环的入口点为 3。

算法性能分析：

这种方法只需要对链表进行一次遍历，因此，时间复杂度为 O(N)。另外，由于只需要几个指针变量来保存结点的地址信息，因此，空间复杂度为 O(1)。

1.7 如何把链表相邻元素翻转

【出自 TX 笔试题】

难度系数：★★★☆☆　　被考察系数：★★★★☆

题目描述：

把链表相邻元素翻转，如给定链表为 1→2→3→4→5→6→7，则翻转后的链表变为 2→1→4→3→6→5→7。

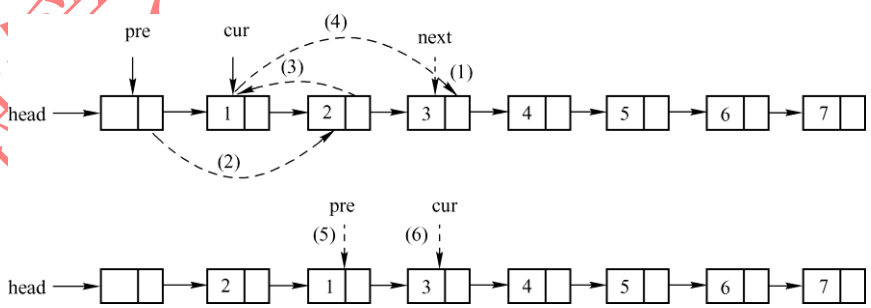
分析与解答：

方法一：交换值法

最容易想到的方法就是交换相邻两个结点的数据域，这种方法由于不需要重新调整链表的结构，因此比较容易实现，但是这种方法并不是考官所期望的解法。

方法二：就地逆序

主要思路：通过调整结点指针域的指向来直接调换相邻的两个结点。如果单链表恰好有偶数个结点，那么只需要将奇偶结点对调即可，如果链表有奇数个结点，那么只需要将除最后一个结点外的其他结点进行奇偶对调即可。为了便于理解，下图给出了其中第一对结点对调的方法。



在上图中，当前遍历到结点 cur，通过步骤（1）～（6）用虚线的指针来代替实线的指针实现相邻结点的逆序。其中，步骤（1）～（4）实现了前两个结点的逆序操作，步骤（5）和（6）向后移动指针，接着可以采用同样的方式实现后面两个相邻结点的逆序操作。实现代码如下：

```
public class Test
{
    /* 把链表相邻元素翻转   */
```

```
    public static void reverse2(LNode head)
    {
        // 判断链表是否为空
        if (head == null || head.next == null)
            return;
        LNode cur = head.next;                          // 当前遍历结点
        LNode pre = head;                               // 当前结点的前驱结点
        LNode next = null;                              // 当前结点后继结点的后继结点
        while (cur != null&&cur.next != null)
        {
            next = cur.next.next;                       // 见上图（1）
            pre.next = cur.next;                        // 见上图（2）
            cur.next.next = cur;                        // 见上图（3）
            cur.next = next;                            // 见上图（4）
            pre = cur;                                  // 见上图（5）
            cur = next;                                 // 见上图（6）
        }

    }

    public static void main(String[] args)
    {
        int i=1;
        LNode head = new LNode();
        head.next = null;
        LNode tmp = null;
        LNode cur = head;
        for (; i< 8; i++)
        {
            tmp = new LNode();
            tmp.data = i;
            tmp.next = null;
            cur.next = tmp;
            cur = tmp;
        }
        System.out.print("顺序输出：");
        for (cur = head.next; cur != null; cur = cur.next)
            System.out.print(cur.data + "");
        reverse(head);
        System.out.print("\n 逆序输出：");
        for (cur = head.next; cur != null; cur = cur.next)
            System.out.print(cur.data + "");
        for (cur = head.next; cur != null;)
        {
            cur = cur.next;
        }
    }
}
```

程序的运行结果为

```
顺序输出：1  2  3  4  5  6  7
逆序输出：2  1  4  3  6  5  7
```

上例中，由于链表有奇数个结点，因此，链表前三对结点相互交换，而最后一个结点保持在原来的位置。

算法性能分析：

这种方法只需要对链表进行一次遍历，因此，时间复杂度为 O(N)。另外，由于只需要几个指针变量来保存结点的地址信息，因此，空间复杂度为 O(1)。

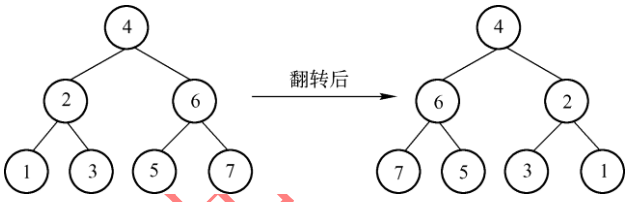
3.11 如何对二叉树进行镜像反转

【出自 TB 笔试题】

难度系数：★★★☆☆　　被考察系数：★★★☆☆

题目描述：

二叉树的镜像就是二叉树对称的二叉树，就是交换每一个非叶子结点的左子树指针和右子树指针，如下图所示，请写出能实现该功能的代码。注意：请勿对该树做任何假设，它不一定是平衡树，也不一定有序。



分析与解答：

从上图可以看出，要实现二叉树的镜像反转，只需交换二叉树中所有结点的左右孩子即可。由于对所有的结点都做了同样的操作，因此，可以用递归的方法来实现。由于需要调用 printTreeLayer 层序打印二叉树，这种方法中使用了队列来实现，实现代码如下：

```
import java.util.LinkedList;
import java.util.Queue;

public class Test
{
    /* 对二叉树进行镜像反转 */
    public static void reverseTree(BiTNode root)
    {
        if(root==null)
            return;
        reverseTree(root.lchild);
        reverseTree(root.rchild);
        BiTNode tmp=root.lchild;
        root.lchild=root.rchild;
        root.rchild=tmp;
    }
```

```
    public static BiTNode    arraytotree(int[] arr, int start, int end)
    {
        BiTNode root=null;
        if(end>=start)
        {
            root = new BiTNode();
            int mid=(start+end+1)/2;
            //树的根结点为数组中间的元素
            root.data = arr[mid];
            //递归的用左半部分数组构造 root 的左子树
            root.lchild=arraytotree(arr,start,mid-1);
            //递归的用右半部分数组构造 root 的右子树
            root.rchild=arraytotree(arr, mid+1, end);
        }
        else
        {
            root = null;
        }
        return root;
    }
    publics tatic void printTreeLayer(BiTNode root)
    {
        if(root==null) return;
        BiTNode p;
        Queue<BiTNode>queue= new LinkedList<BiTNode>();
        //树根结点进队列
        queue.offer(root);
        while(queue.size()>0)
        {
            p=queue.poll();
            //访问当前结点
            System.out.print(p.data+" ");
            //如果这个结点的左孩子不为空则入队列
            if(p.lchild!=null)
                queue.offer(p.lchild);
            //如果这个结点的右孩子不为空则入队列
            if(p.rchild!=null)
                queue.offer(p.rchild);
        }
    }

    public static void main(String[] args)
    {
        int arr[] = {1,2,3,4,5,6,7};
        BiTNode root;
        root=arraytotree(arr, 0,arr.length-1);
        System.out.print("二叉树层序遍历结果为：");
        printTreeLayer(root);
        System.out.println();
        reverseTree(root);
        System.out.print("反转后的二叉树层序遍历结果为：");
        printTreeLayer (root);
    }
}
```

程序的运行结果为

```
二叉树层序遍历结果为：4 2 6 1 3 5 7
反转后的二叉树层序遍历结果为：4 6 2 7 5 3 1
```

算法性能分析：

由于对给定的二叉树进行了一次遍历，因此，时间复杂度为 ~~O(n)~~O(N)。

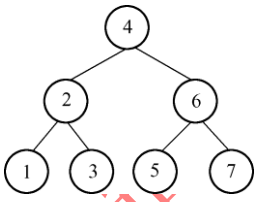
3.12 如何在二叉排序树中找出第一个大于中间值的结点

【出自 HW 面试题】

难度系数：★★★★☆　　被考察系数：★★★☆☆

题目描述：

对于一棵二叉排序树，令 f=(最大值+最小值)/2，设计一个算法，找出距离 f 值最近、大于 f 值的结点。例如，下图所给定的二叉排序树中，最大值为 7，最小值为 1，因此，f=(1+7)/2=4，那么在这棵二叉树中，距离结点 4 最近并且大于 4 的结点为 5。



分析与解答：

首先需要找出二叉排序树中的最大值与最小值。由于二叉排序树的特点是，对于任意一个结点，它的左子树上所有结点的值都小于这个结点的值，它的右子树上所有结点的值都大于这个结点的值。因此，在二叉排序树中，最小值一定是最左下的结点，最大值一定是最右下的结点。根据最大值与最小值很容易就可以求出 f 的值。接下来对二叉树进行中序遍历。如果当前结点的值小于 f，那么在这个结点的右子树中接着遍历，否则遍历这个结点的左子树。实现代码如下：

```
public class Test
{
    /*
    ** 方法功能：查找值最小的结点
    ** 输入参数：root:根结点
    ** 返回值：值最小的结点
    */
    private static BiTNode    getMinNode(BiTNode    root)
    {
            if(root==null)
            return root;
        while(root.lchild!=null)
            root=root.lchild;
        return root;
    }
```

```
/*
** 方法功能：查找值最大的结点
** 输入参数：root:根结点
** 返回值：值最大的结点
*/
private static BiTNode getMaxNode(BiTNode    root)
{
        if(root==null)
        return root;
    while(root.rchild!=null)
        root=root.rchild;
    return root;
}

public static BiTNode getNode(BiTNode    root)
{
    BiTNode maxNode=getMaxNode(root);
    BiTNode minNode=getMinNode(root);
    intmid=(maxNode.data+minNode.data)/2;
    BiTNode    result=null;
    while(root!=null)
    {
        //当前结点的值不大于 f，则在右子树上找
        if(root.data<=mid)
        {
        root=root.rchild;
        }
        //否则在左子树上找
else
        {
           result=root;
           root=root.lchild;
        }
    }
    return result;
}
public static BiTNode    arraytotree(int[] arr, intstart, int end)
{
    BiTNode root=null;
    if(end>=start)
    {
        root = new BiTNode();
        int mid=(start+end+1)/2;
        //树的根结点为数组中间的元素
        root.data = arr[mid];
        //递归的用左半部分数组构造 root 的左子树
        root.lchild=arraytotree(arr,start,mid-1);
        //递归的用右半部分数组构造 root 的右子树
        root.rchild=arraytotree(arr, mid+1, end);
    }
    else
    {
        root = null;
    }
    return root;
}
public static void main(String[] args)
{
```

```
            int arr[] = {1,2,3,4,5,6,7};
            BiTNode root;
            root=arraytotree(arr,0,arr.length-1);   //3.2 节
            System.out.println(getNode(root).data);
        }
    }
```

程序的运行结果为

```
    5
```

算法性能分析：

这种方法在查找最大结点与最小结点时的时间复杂度为 O(h)，h 为二叉树的高度，对于有 N 个结点的二叉排序树，最大的高度为 ~~O(n)~~O(N)，最小的高度为 O(~~logn~~logN)。同理，在查找满足条件的结点时，时间复杂度也是 O(h)。综上所述，这种方法的时间复杂度在最好的情况下是 O(~~logn~~logN)，最坏的情况下为 ~~O(n)~~O(N)。

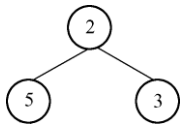
3.13 如何在二叉树中找出路径最大的和

【出自 HW 面试题】

难度系数：★★★★☆　　被考察系数：★★★★☆

题目描述：

给定一棵二叉树，求各个路径的最大和，路径可以以任意结点作为起点和终点。比如，给定以下二叉树：



返回 10。

```
    class TreeNode
    {
        int val;
        TreeNode left;
        TreeNode right;
        public TreeNode(intval)
        {
            this.val = val;
        }
    }
    int maxPathSum(TreeNode root)
```

分析与解答：

本题可以通过对二叉树进行后序遍历来解决。
对于当前遍历到的结点 root，假设已经求出在遍历 root 结点前最大的路径和为 max：
1）求出以 root.left 为起始结点，叶子结点为终结点的最大路径和为 maxLeft。
2）同理求出以 root.right 为起始结点，叶子结点为终结点的最大路径和 maxRight。
包含 root 结点的最长路径可能包含如下三种情况：
1）leftMax=root.val+maxLeft（右子树最大路径和可能为负）。
2）rightMax=root.val+maxRight（左子树最大路径和可能为负）。
3）allMax=root.val+maxLeft+maxRight（左右子树的最大路径和都不为负）。
因此，包含 root 结点的最大路径和为 tmpMax=max(leftMax,rightMax,allMax)。
在求出包含 root 结点的最大路径后，如果 tmpMax>max，那么更新最大路径和为 tmpMax。
实现代码如下：

```
class TreeNode
{
    int val;
    TreeNode left;
    TreeNode right;
    public TreeNode(int val)
    {
        this.val = val;
    }
}

class IntRef {
    public int val;
}

public class Test
{
    /*求 a，b，c 的最大值*/
    int Max(int a, int b, int c)
    {
        int max = a>b ? a : b;
        max = max>c ? max : c;
        return max;
    }
    /*寻找最长路径*/
    public int findMaxPathrecursive(TreeNode root, IntRef max)
    {
        if (null == root)
        {
            return 0;
        }
        else
        {
            //求左子树以 root.left 为起始结点的最大路径和
            int sumLeft = findMaxPathrecursive(root.left,max);
            //求右子树以 root.right 为起始结点的最大路径和
            int sumRight = findMaxPathrecursive(root.right,max);

            //求以 root 为起始结点，叶子结点为结束结点的最大路径和
```

```
                int allMax = root.val + sumLeft + sumRight;
                int leftMax = root.val + sumLeft;
                int rightMax = root.val + sumRight;
                int tmpMax = Max(allMax, leftMax, rightMax);
                if (tmpMax>max.val)
                    max.val = tmpMax;
                int subMax = sumLeft > sumRight ? sumLeft : sumRight;
                //返回以 root 为起始结点，叶子结点为结束结点的最大路径和
                return root.val + subMax;
            }
        }
        public int findMaxPath(TreeNode root)
        {
            IntRef max=new IntRef();
            max.val = Integer.MIN_VALUE;
            findMaxPathrecursive(root,max);
            return max.val;
        }
        public static void main(String[] args)
        {
            TreeNode root = new TreeNode(2);
            TreeNode left = new TreeNode(3);
            TreeNode right = new TreeNode(5);
            root.left = left;
            root.right = right;
            Test t = new Test();
            System.out.println( t.findMaxPath(root) );
        }
    }
```

程序的运行结果为

```
10
```

算法性能分析：

二叉树后序遍历的时间复杂度为 O(N)，因此，这种方法的时间复杂度也为 O(N)。

3.14 如何实现反向 DNS 查找缓存

【出自 BD 面试题】

难度系数：★★★★☆　　被考察系数：★★★★☆

题目描述：

反向 DNS 查找指的是使用 Internet IP 地址查找域名。例如，如果在浏览器中输入 74.125.200.106，它会自动重定向到 google.in。
如何实现反向 DNS 查找缓存？

分析与解答：

要想实现反向 DNS 查找缓存，主要需要完成如下功能：
1）将 IP 地址添加到缓存中的 URL 映射。
2）根据给定 IP 地址查找对应的 URL。
对于本题的这种问题，常见的一种解决方案是使用哈希法（使用 Hashmap 来存储 IP 地址与 URL 之间的映射关系），由于这种方法相对比较简单，这里就不做详细的介绍了。下面重点介绍另外一种方法：Trie 树。这种方法的主要优点如下：
1）使用 Trie 树，在最坏的情况下的时间复杂度为 O(1) ，而哈希方法在平均情况下的时间复杂度为 O(1)；
2）Trie 树可以实现前缀搜索（对于有相同前缀的 IP 地址，可以寻找所有的 URL）。
当然，由于树这种数据结构本身的特性，所以使用树结构的一个最大的缺点就是需要耗费更多的内存，但是对于本题而言，这却不是一个问题，因为 Internet IP 地址只包含有 11 个字母（0～9 和.）。所以，本题实现的主要思路：在 Trie 树中存储 IP 地址，而在最后一个结点中存储对应的域名。实现代码如下：

```
class DNSCache
{
    /* IP 地址最多有 11 个不同的字符 */
    private final intCHAR_COUNT=11;

    /* IP 地址最大的长度 */
    private TrieNode root = new TrieNode();
    /* Trie 树的结点 */
    public class TrieNode
    {
        boolean isLeaf;
        String url;
        TrieNode[] child; //CHAR_COUNT

    public TrieNode()
    {
        this.isLeaf=false;
        this.url=null;
        this.child=new TrieNode[CHAR_COUNT];
        for (int i=0; i<CHAR_COUNT; i++)
        {
            child[i]=null;
        }
    }
}

    public int getIndexFromChar(char c)
    {
        return (c == '.')? 10: (c - '0');
    }

    public char getCharFromIndex(int i)
    {
        return (i== 10)? '.' : (char)('0' + i);
    }
```

```
    /* 把一个 IP 地址和相应的 URL 添加到 Trie 树中，最后一个结点是 URL */
    public void insert( String ip, String url)
    {
        /* IP 地址的长度 */
        int len = ip.length();
        TrieNode pCrawl = root;

        int level;
        for (level=0; level<len; level++)
        {
            /* 根据当前遍历到的 ip 中的字符，找出子结点的索引 */
            int index = getIndexFromChar(ip.charAt(level));

            /* 如果子结点不存在，则创建一个 */
            if (pCrawl.child[index]==null)
            pCrawl.child[index] = new TrieNode();

            /* 移动到子结点 */
            pCrawl = pCrawl.child[index];
        }

        /* 在叶子结点中存储 IP 对应的 URL */
        pCrawl.isLeaf = true;
        pCrawl.url = url;
    }

    /* 通过 IP 地址找到对应的 URL */
    public String searchDNSCache(String ip)
    {
        TrieNode pCrawl = root;
        int len = ip.length();

        int level;
        // 遍历 IP 地址中所有的字符.
        for (level=0; level<len; level++)
        {
            int index = getIndexFromChar(ip.charAt(level));
            if (pCrawl.child[index]==null)
            return null;
            pCrawl = pCrawl.child[index];
        }

        /* 返回找到的 URL */
        if (pCrawl!=null&&pCrawl.isLeaf)
        return pCrawl.url;

        return null;
    }
}

public class Test
{
    public static void main(String[] args)
    {
        String[] ipAdds= {"10.57.11.127", "121.57.61.129", "66.125.100.103"};
        String[] url = {"www.samsung.com", "www.samsung.net", "www.google.in"};
        int n = ipAdds.length;
        DNSCache cache=new DNSCache();
```

```
            /* 把 IP 地址和对应的 URL 插入到 Trie 树中 */
            for (int i=0; i<n; i++)
            cache.insert(ipAdds[i],url[i]);

            String ip = "121.57.61.129";
            String res_url = cache.searchDNSCache(ip);
            if (res_url != null)
                System.out.println("找到了 IP 对应的 URL:\n"+ ip+"--->"+ res_url);
            else
              System.out.println("没有找到对应的 URL\n");
        }
    }
```

程序的运行结果为

```
    找到了 IP 对应的 URL:
    121.57.61.129 --> www.samsung.net
```

显然，由于上述算法中涉及的 IP 地址只包含特定的 11 个字符（0～9 和.），所以，该算法也有一些异常情况未处理。例如，不能处理用户输入的不合理的 IP 地址的情况，有兴趣的读者可以继续朝着这个思路完善后面的算法。