张京 胡凌云 编著

C/C++程序员生存手册

为自己赢得一份IT名企职位

Handbook For C/C++ Software Engineering Interview

全方位引导IT职业规划 解密名企选人标准 揭示IT职场潜规则

- 职业规划、简历撰写
- 面试笔试、思维拓展
- 踏上征途、渐入佳境
- 风雨江湖、更上层楼

张京 胡凌云 编著

C/C++程序员生存手册

为自己赢得一份IT名企职位

Handbook For C/C++ Software Engineering Interview

全方位引导IT职业规划 解密名企选人标准 揭示IT职场潜规则

- 职业规划、简历撰写
- 面试笔试、思维拓展
- 踏上征途、渐入佳境
- 风雨江湖、更上层楼

人 民 邮 电 出 版 社
北 京

图书在版编目（CIP）数据

C/C++程序员生存手册 ：为自己赢得一份IT名企职位 / 张京，胡凌云编著. -- 北京 ：人民邮电出版社，2010.7

ISBN 978-7-115-22719-5

Ⅰ. ①C… Ⅱ. ①张… ②胡… Ⅲ. ①C语言—程序设计—技术手册 Ⅳ. ①TP312-62

中国版本图书馆CIP数据核字(2010)第059278号

内 容 提 要

本书详细分析了软件工程师求职过程中的常见问题，深入解析了各大 IT 公司考查求职者的面试真题，告诉读者用人单位需要什么样的技术人才、考查什么样的技术知识以及如何甄别人才。全书分 4 篇，共 17 章。第 1 篇是求职过程，讲述了程序员求职的整个过程，包括职业规划、简历撰写、简历投递、笔试以及各类面试，并列出了最常用的英文面试词汇方便读者参考。第 2 篇是 C/C++面试题，作为本书的核心，主要讲述了 C/C++程序员需要掌握的各项技术，并结合各大公司实际的面试题进行讲解，对一些面试所考查的重点和难点进行了全面和深入的分析解答。读者可以通过阅读本部分全面了解 C/C++技术面试的各个方面，快速复习 C/C++编程的知识。第 3 篇是智力测试，囊括了面试中常见的智力面试题，读者可通过阅读本部分迅速提高分析和解答问题的能力。第 4 篇是职场生涯，读者可以全面了解和感悟办公室文化，从而提升自己的软实力。

本书适合应聘计算机软件开发领域职位的应届毕业生和其他求职者阅读，也适合作为软件开发从业人员和计算机爱好者的参考书。

C/C++程序员生存手册：为自己赢得一份 IT 名企职位

◆ 编　　著　张　京　胡凌云
　责任编辑　蒋　佳

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街 14 号
　邮编　100061　　电子函件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京艺辉印刷有限公司印刷

◆ 开本：800×1000　1/16
　印张：29.75
　字数：637 千字　　　　2010 年 7 月第 1 版
　印数：1 – 4 000 册　　2010 年 7 月北京第 1 次印刷

ISBN 978-7-115-22719-5

定价：55.00 元

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154

前 言

当求职者应聘一份程序设计、软件开发方面的工作时，招聘方一般都会安排一次面试来考察其程序设计能力。由于 C/C++语言具有很大的灵活性，被广泛运用于各个领域，很多公司将 C/C++编程能力作为考察程序员基本素质的主要方式。市场上虽然有许多 C/C++编程方面的书籍，但大多数都是讲解语言编程，很少针对 C/C++面试，读者很难迅速了解并掌握面试所需要的知识。

本书涵盖 C/C++面试中出现的各个知识点，包括 C 语言编程基础、C++面向对象、算法、数据结构等。所有知识点都结合例题进行说明，每道例题都取材于各大公司的实际招聘面试题，并在题后紧跟详细的分析步骤和解答。

本书面向正在找工作和将要找工作的程序员。相信本书一定能帮助读者快速复习有关的知识，并获得一份满意的工作。

本书的特点

本书全面讲解了 C/C++面试的各个知识点，并对于一些重点和难点进行了细致的分析。其特点主要体现在以下几个方面。

- 编排细致。

软件公司对于求职者的考查，看重基础知识的掌握，但是往往考点非常细。求职者必须具备扎实的编程基础和良好的编程习惯才能轻松应对。本书编排从 C/C++核心基础开始，由浅入深地逐渐转入到高级部分，强调了如何在实际工作中活用基础知识，进行高质量的程序开发。

- 内容最新。

本书中所有题目都来自于近两年各大 IT 公司的面试真题，经过汇总和按知识点归类，真正做到了结构设置科学、知识点全面。

- 实用性强。

技术面试题的全部意义在于检验求职者的编程能力，目的在于挑选能够迅速胜任工作岗位的求职者。本书中针对 C/C++的知识点，结合实际应用进行了讲解，对于工作中需要注意的重点和难点，做了着重介绍。

- 增加智力考题。

随着软件开发的全球化趋势，国内软件企业对求职者的综合素质要求越来越高，面试中也出现越来越多的智力测试部分。没有这方面经验的求职者，常常感觉“智力不够用”。本书对大部分常见的智力题进行了归类及分析解答，引导求职者把握思路线索。

本书的内容安排

本书共分为 4 篇，共 17 章。

第 1 篇（第 1 章～第 3 章）求职过程。讲述了程序员求职的整个过程，包括职业规划、简历撰写、简历投递、笔试以及各类面试试题，方便读者参考。

第 2 篇（第 4 章～第 12 章）C/C++面试题。这一部分是本书的核心，占全书篇幅的 85%，讲述了 C/C++程序员需要掌握的各项技术，并结合各大公司实际的面试题进行讲解，对一些面试所考察的重点和难点进行了全面和深入的分析解答。

第 3 篇（第 13 章）智力测试。这一部分囊括了面试中常见的智力面试题，方便读者迅速提高智力题的分析解答技巧。

第 4 篇（第 14 章～第 17 章）职场生涯。这部分包括踏上征途、渐入佳境、风雨江湖和更上层楼 4 个章节。读者可以全面了解和感悟办公室文化，从而提升自己的软实力。

适合阅读本书的读者

- 即将步入 IT 行业的应届毕业生。
- 有一定工作经验但 C/C++编程基础不好的程序员。
- C/C++编程爱好者。
- 公司管理人员或人力资源管理人员。

编者

PDG

目 录

第1篇 求职过程

第1章 职业规划……2

1.1 关于理想……2
1.2 职业方向……2
1.3 如何选择职业方向……3
1.3.1 兴趣……4
1.3.2 能力……4
1.3.3 经验……5
1.3.4 现实需求……5
1.4 IT人员的职业方向……6
1.5 大小公司……6
1.5.1 大公司……6
1.5.2 小公司……7
1.6 中企外企……8
1.6.1 语言差异……8
1.6.2 文化差异……8

第2章 简历撰写……9

2.1 基本要求……9
2.2 主要内容……10
2.3 目标型简历……11
2.4 英文简历……12
2.5 模板……13
2.5.1 中文简历示例……13
2.5.2 英文简历示例……15
2.6 求职信……17

2.7 推荐信……19
2.8 其他手段——博客、网站……20
2.9 简历投放……20

第3章 面试……22

3.1 准备面试……22
3.2 面试方式……23
3.2.1 电话面试……23
3.2.2 面对面面试……24
3.2.3 常见问题……26
3.2.4 感谢信……31
3.2.5 笔试……31
3.3 待遇谈判……32

第2篇 C/C++面试题

第4章 C/C++程序基础……36

4.1 变量赋值……36
4.1.1 一般赋值语句……36
4.1.2 i++与++i……39
4.2 编程规范……42
4.2.1 条件比较……42
4.2.2 命名规则……43
4.3 类型转换……44
4.4 数值交换……45
4.5 C和C++的联系与区别……47
4.6 main函数之后的调用……50

第5章 预处理、const、static与sizeof……52

5.1 预处理……52
5.1.1 #ifdef、#else、#endif指示符……52
5.1.2 宏定义……54
5.2 const（常量）……57

5.3 static 变量（静态变量）……59
5.4 sizeof 操作符……62
5.5 inline 与宏定义……75

第 6 章 引用和指针……78

6.1 引用……78
6.1.1 引用的基本问题……78
6.1.2 参数引用……81
6.1.3 常量引用……84
6.1.4 引用与指针的区别……85
6.2 指针基础……86
6.2.1 指针的声明……87
6.2.2 指针的运算……88
6.2.3 指针常量与常量指针……93
6.2.4 C++中 this 指针……95
6.3 指针数组与数组指针……97
6.4 函数指针与指针函数……100
6.5 野指针……105
6.6 动态内存的传递……111
6.7 指针与句柄的区别……118

第 7 章 字符串……120

7.1 数字与字符串的转化……120
7.1.1 数字转化为字符串……120
7.1.2 字符串转化为数字……123
7.2 字符串与数组……125
7.2.1 strcpy 函数与 memcpy 函数……126
7.2.2 数组越界……128
7.2.3 其他编程问题……131

第 8 章 位运算与嵌入式编程……165

8.1 位制转换与位运算……165
8.1.1 位制转换……165
8.1.2 位运算……166
8.2 嵌入式编程……171

第 9 章 C++面向对象 ······ 176

9.1 面向对象的基本概念 ······ 176
9.2 class 和 struct 的区别 ······ 178
9.3 成员变量 ······ 182
9.4 构造函数和析构函数 ······ 188
9.4.1 构造函数 ······ 188
9.4.2 析构函数 ······ 194
9.5 复制构造函数和赋值函数 ······ 197
9.5.1 复制构造函数 ······ 197
9.5.2 赋值函数 ······ 202
9.6 函数重载和运算符重载 ······ 214
9.6.1 函数重载 ······ 214
9.6.2 运算符重载 ······ 217

第 10 章 C++继承和多态 ······ 228

10.1 继承的概念 ······ 228
10.2 私有继承 ······ 233
10.3 多态的概念 ······ 237
10.4 多重继承和虚拟继承 ······ 246
10.5 纯虚函数和抽象基类 ······ 251
10.6 COM（组件对象模型） ······ 257

第 11 章 数据结构 ······ 266

11.1 单链表 ······ 266
11.2 循环链表 ······ 278
11.3 双向链表 ······ 280
11.4 双向循环链表 ······ 286
11.5 队列 ······ 293
11.6 栈 ······ 297
11.7 二叉树 ······ 303

第 12 章 排序 ······ 317

12.1 插入排序 ······ 317
12.1.1 直接插入排序 ······ 317

12.1.2 希尔（Shell）排序 ······319
12.2 交换排序······321
12.2.1 冒泡排序······321
12.2.2 快速排序······324
12.3 选择排序······325
12.3.1 直接选择排序······326
12.3.2 堆排序······327
12.4 归并排序······330
12.5 基数排序······333
12.6 各种排序方法比较 ······336

第3篇 智力测试

第 13 章 智力测试题······340

13.1 数学能力······340
13.2 推理能力······351
13.3 反应能力······368

第4篇 职场生涯

第 14 章 踏上征途 ······374

14.1 第一天上班 ······374
14.2 入职事项······375
14.3 最初几周······376
14.4 蘑菇管理定律 ······377

第 15 章 渐入佳境 ······379

15.1 从依赖走向独立 ······379
15.1.1 习惯一：积极主动······380
15.1.2 习惯二：以终为始······383
15.1.3 习惯三：要事第一······386
15.2 建立目标······389
15.3 评价工作表现 ······390

15.3.1 年终总结……391
15.3.2 绩效考评……391
15.4 开会及发言……392
15.4.1 会议主持人……392
15.4.2 会议参与者……395
15.4.3 PowerPoint 技巧……395
15.5 电子邮件……397
15.5.1 基本注意事项……397
15.5.2 电子邮件类型……398
15.6 电话沟通……399
15.6.1 打电话的技巧……400
15.6.2 接电话的技巧……401
15.7 面谈……402
15.8 培训……403
15.9 加班……403
15.10 请假……404
15.11 出差……405
15.12 报销……406

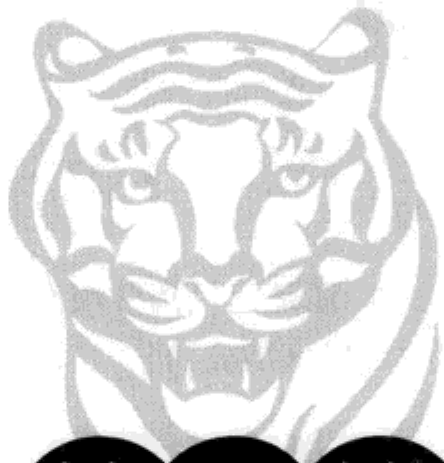
第 16 章 风雨江湖……407

16.1 从独立走向互赖……407
16.1.1 习惯四：双赢思维……408
16.1.2 习惯五：知彼解己……409
16.1.3 习惯六：统合综效……409
16.1.4 习惯七：不断更新……410
16.1.5 总图……410
16.2 和老板的关系……411
16.2.1 给你的老板分个类……411
16.2.2 老板的话必须要听……417
16.2.3 老板的话要听懂……418
16.2.4 老板说的总是对的……419
16.2.5 不要给老板惊奇……421
16.2.6 和老板建立良好关系……422
16.2.7 被老板表扬……424
16.2.8 被老板批评……426

16.2.9 意见和建议……427
16.2.10 尽快投靠新老板……428
16.2.11 外国老板……429
16.2.12 异地老板……432
16.3 和同事的关系……433
16.3.1 合作……433
16.3.2 处理争议……437
16.3.3 沟通……438
16.3.4 竞争……441
16.4 和客户的关系……443
16.4.1 态度决定成败……444
16.4.2 谈判风格……446
16.4.3 谈判技巧……449
16.5 和其他部门的关系……450
16.5.1 和业务部门之间的关系……450
16.5.2 和服务部门之间的关系……452

第 17 章 更上层楼……453

17.1 升职加薪……453
17.1.1 如何谋求升职……453
17.1.2 如何要求加薪……455
17.2 换部门……459
17.2.1 在现岗上做出成绩……460
17.2.2 学会谦虚与等待……460
17.2.3 如何提？把戏演好……460
17.3 辞职……461
17.3.1 决定是否要辞职……461
17.3.2 如果不能辞职……462
17.3.3 开始找新工作……463
17.3.4 提出辞职……463
17.3.5 离开工作岗位……464



第1篇

求职过程

第1章 职业规划

对于每一个准备求职的人来讲，不论是刚刚走出大学校门，还是已经离开现有职位，或者正在犹豫要不要换一份新的工作，最需要考虑清楚的是未来的职业规划，简而言之：你最终想要达到的目标是什么？你的理想是什么？

1.1 关于理想

从我们懂事的那一刻起，理想就已经开始伴随我们。我们小学一年级的课文里就有：你长大了准备干什么？

古语有云："有志者立长志，无志者常立志。"但是随着现代社会的迅速变化，立长志变得几乎不可能。更多的情况是，你的理想需要随着现实状况的变化而做适当的调整。美国有一个调查显示：80%的小学生相信自己将来长大了以后可以当总统，这个比例到了中学阶段就降低到不到 30%，到了大学以后还以当总统为理想的则连 1%也不到了。所以理想的调整随着我们年龄的增长几乎是一个不可避免的事情，随着我们经验的积累，阅历的增加以及现实的选择，每个人的理想都会或多或少地发生变化。

但即便如此，一个相对恒定的理想还是会使你的职业生涯获益良多。因为理想指明了你前进的方向，有了理想，你才能知道该往哪个方向努力，并探寻努力的方法并最终达到目标。

人生会有很多种理想，亲情理想、爱情理想、家庭理想，还有一个很重要的就是你的职业理想。

1.2 职业方向

在规划你未来的职业取向时首先要考虑的是职业方向。IT 行业的从业范围很广，从大的范围来分，可以分为技术人员、商务人员和管理人员几大类。在技术人员里又可以分为程序员、技术支持、测试人员等。

有的人认为程序员是 IT 从业人员不得不做的痛苦的选择，其实并不尽然。学过图论的人应

该知道图论中非常有名的迪杰斯特拉（Dijkstra）算法。迪杰斯特拉是荷兰著名的计算机科学家，曾获得过计算机界的最高奖 1972 年的图灵奖。他曾提到在荷兰的时候人们问他："你的头衔是什么？"他说："程序员。"他们说："不可能，那不算是一个头衔，程序员只不过是编码员，就跟中世纪的书记员差不多。"迪杰斯特拉说："不，我就是一个程序员，并且我为我是一个程序员而感到骄傲。"所以很多情况下，你到底选择做什么，是要根据你的兴趣爱好来决定，而不要以不得已为借口。

1.3 如何选择职业方向

在选择职业方向之前，最重要的是认识你自己，了解你自己是什么样的人。

爱因斯坦小时候是个十分贪玩的孩子，他的母亲常常为此忧心忡忡。母亲的再三告诫对他来说如同耳边风。直到 16 岁那年的秋天，一天上午，父亲将正要去河边钓鱼的爱因斯坦拦住，并给他讲了一个故事，正是这个故事改变了爱因斯坦的一生。

父亲说："昨天我和咱们的邻居杰克大叔去清扫南边的一个大烟囱，那烟囱只有踩着里面的钢筋踏梯才能上去。杰克大叔在前面，我在后面。我们抓着扶手一阶一阶地终于爬上去了，下来时，杰克大叔依旧走在前面，我还是跟在后面。后来，钻出烟囱，我们发现了一件奇怪的事情，杰克大叔的后背、脸上全被烟囱里的烟灰蹭黑了，而我身上竟连一点烟灰也没有。"

爱因斯坦的父亲继续微笑着说："我看见杰克大叔的模样，心想我一定和他一样，脸脏得像个小丑，于是我就到附近的小河里去洗了又洗。而杰克大叔呢，他看我钻出烟囱时干干净净的，就以为他也和我一样干干净净的，只草草地洗了洗手就上街了。结果，街上的人都笑破了肚子，还以为杰克大叔是个疯子呢。"

爱因斯坦听罢，忍不住和父亲一起大笑起来。父亲笑完后，郑重地对他说："其实别人谁也不能做你的镜子，只有自己才是自己的镜子。拿别人做镜子，白痴或许会把自己照成天才的。"

早在 2 000 多年前，古希腊人就把"认识你自己"作为铭文刻在阿波罗神庙的门柱上。在日常生活中，我们既不可能每时每刻去反省自己，也不可能总把自己放在局外人的地位来观察自己，于是只能借助外界信息来认识自己。正因如此，每个人在认识自我时很容易受外界信息的暗示，迷失在环境当中，受到周围信息的暗示，并把他人的言行作为自己行动的参照。人很容易受到来自外界信息的暗示，从而出现自我知觉的偏差，认为一种笼统的、一般性的人格描述十分准确地揭示了自己的特点。

实际上，决定一个人的职业方向大致有以下几个要素需要考量：你的兴趣、你的能力、你的经验和现实需求。这 4 个要素的重要程度并不相等，我们在这里把最重要的排在前面，次要的排在后面，所以当你在考虑问题的时候要按重要程度来作权衡。

1.3.1 兴趣

兴趣二字在维基百科里的定义如下："兴趣是一种人们在空闲时享受及乐于去做的活动，很多时候人们不是为赚钱而参与这些活动。"兴趣的产生首先是从好奇开始，从每个人呱呱坠地那一刻起，就对这个世界充满了好奇，可以说，正是由于对外部世界的强烈好奇心使我们人类有别于其他动物。当我们带着一颗好奇心观察外部世界时，由于每个人性格的不同，对特定刺激的敏感度不同，从而使我们认为某件事物更有吸引力，并愿意投入更多的精力去从事该项活动，逐渐地我们对于某一类特定的事物产生一种倾向性，最终这种倾向性发展成为兴趣。

所有职业规划师都承认，对于职业生涯来说，兴趣所占的重要性往往是第一位的，因为兴趣是发自内心的从事某种活动的最强大动力，外在因素可以作用于一时，但兴趣往往会伴随你一生。著名的华裔科学家丁肇中曾经说过："兴趣比天才重要。"充分说明了兴趣对于职业成功的重要性。

在考虑你未来的职业取向时，还要注意区分创造兴趣和享受兴趣。只有创造兴趣才能成为你职业选择的方向，比如有人说："我喜欢游戏，所以我想去一家游戏公司工作。"这里实际上有一个误区，你是喜欢玩游戏，还是喜欢创造游戏？这是两个很不相同的概念。如果我们问，谁喜欢吃饭？那么大约100%的人都喜欢吃饭。但你若再问，谁喜欢做饭？恐怕人数就没那么多了。同样，喜欢玩游戏不代表喜欢创造游戏。如果你真准备选择游戏行业，那你需要先了解清楚你自己是否愿意整天为了同一个游戏画面而烦恼，或者整个星期、整个月反复不停地测试同一个游戏功能。游戏之所以吸引人，是因为它反复变化的多样性，但当游戏成为工作时，它就不一定那么有趣了。

所谓创造兴趣是指你要问问你自己：如果你必须要创造一种东西的话，那你希望创造什么？从这个角度思考，可以帮助你找到自己真正的兴趣所在。

1.3.2 能力

能力实际上是可以在兴趣的基础上进行培养的。兴趣决定了你前进的方向，而能力会决定你能走多远。准确认识自身的能力有助于你取长补短，更有针对性地获得成功。

关于职业能力的分类，有很多种分法，比如5大职业核心能力分为：言语理解能力、判断推理能力、数量关系能力、资料分析能力和思维策略能力。对于IT界人士来讲，可以简单地把能力分为两类：硬实力和软实力。所谓硬实力就是指在学校里学到的以及在工作中积累的解决具体技术问题的能力；而软实力则是指处理人际关系，解决冲突，表达自我的能力。在实际工作中，二者缺一不可，但在具体工作岗位上，侧重点可以有所不同。如果你自认硬实力一流，软实力稍逊，可以考虑去应聘偏技术方面的工作，如果你认为自己硬实力不太强，但软实力很强，则应该考虑偏销售市场方面的工作。相对来说，硬实力的培养较软实力的培养有迹可循，通过系统的学习，从事特定工作，可以有效提升硬实力；而软实力的提高除了

有针对性的学习之外，还需要有较长时间的磨炼。

美国培训专家吉格·吉格勒说过："除了生命本身，没有任何才能不需要后天的锻炼。"才能的养成需要后天的努力，没有人能只依靠天分成功。上帝给予了天分，而勤奋将天分变为天才。

中国近代史上的风云人物曾国藩建立了自己的不朽功业，但他的天赋却不高。在取得功名之前，有一天曾国藩在家读书，一篇文章重复不知道多少遍了，还是背不下来。这时候他家来了一个小偷，潜伏在他家的屋檐下，希望等曾国藩睡觉之后再行动。可是等啊等，就是不见他睡觉，还是翻来覆去地读那篇文章。小偷大怒，跳下梁来说："这种水平还读什么书？"然后将那文章背诵一遍，扬长而去！

小偷是很聪明，至少比曾先生要聪明，但是他只能成为小偷，而曾国藩经过自己的勤奋苦读，成就了自己在中国历史上的丰功伟业。古语云：勤能补拙是良训，一分辛苦一分才。小偷的记忆力很好，听过几遍的文章都能背下来，而且很勇敢，见别人不睡觉居然可以跳出来发怒。可惜，他的天赋没有加上勤奋，变得不知所终。伟大的成功和辛勤的劳动是成正比的，有一分劳动就有一分收获，日积月累，从少到多，奇迹就可以创造出来。

1.3.3 经验

对于有一定职业经验的人来说，过往的经历会在很大程度上对今后的方向有决定权。从短期来看，以前从事过多年的工作会对谋求下一个类似职位有很大帮助；从长期来看，从事某一职位愈久，人生轨迹愈不可避免地将固定在这一岗位上，或者要变动的话需要付出很大代价。这一点也说明了在初入行时做好正确的职业选择的重要性。

1.3.4 现实需求

在某种特定情况下，比如长期找不到合适的工作，经济问题等，人是需要对现实作出暂时的妥协，这种妥协并不可耻，也不说明你是不够坚强的人。古语有云："识时务者为俊杰，时势造英雄而非英雄造时势。"如果的确为现实所迫必须尽快找到工作，则应尽全力去顺应形势，然后在形势中再寻找机会。你的人生轨迹也许会因此而发生改变，但人生恰恰因为这些不确定而美丽，并且塞翁失马，焉知非福，也许一个全新的机遇就在这样的妥协中诞生。苹果总裁乔布斯（Steve Jobs）17岁休学后，百无聊赖，于是跑去学书法，他学习了serif与san serif字体，学到在不同字母组合间变更字间距，学到活版印刷伟大的地方。在回忆这段经历时，他说："我没预期过学到的这些东西能在我生活中起些什么实际作用，不过10年后，当我在设计第一台麦金塔时，我想起了当时所学的东西，所以把这些东西都设计进了Mac电脑里，这是第一台能印刷出漂亮东西的计算机。如果我没沉溺于那样一门课里，Mac电脑可能就不会有多重字体跟变间距字体了。又因为Windows抄袭了Mac电脑的使用方式，如果当年我没这样做，大概世界上所有的个人计算机都不会有这些东西，印不出现在我们看到的漂亮的字来了。"所以你看，一个完全按部就班地规划好的人生并不那么有趣，不是吗？

1.4 IT人员的职业方向

了解清楚职业规划的几个要素之后，我们再来看一个IT业界人员，具体地说从程序员做起，可以有哪些选择。一般来讲，如果是从程序员做起的话，在未来的职业道路上，有两种选择：作程序员或者作经理。当然也不排除其他选择，我们这里只谈最基本的。

而作程序员和作经理是两条不同的发展方向。

1．技术方向

沿程序员这条路向上，可以从程序员做起，然后是高级程序员，系统架构师，直到总工程师。

2．管理方向

而如果是做技术经理的话，从一开始就要有意识地学习与管理有关的一切知识，掌握最基本的沟通和组织技巧，从程序员到项目经理，到部门经理，直到CTO。

这两条路是如此不同，所以从一开始就要考虑清楚你这一生到底喜欢做什么。尽管不论走哪一条，都需要有比较好的沟通技巧，但相对地来说，程序员这条路不需要处理太多的人际关系，而更多的偏重于学术性的研究。而走经理这条路，则需要非常多的沟通交际能力，能够组织很多比你的技术水平高很多的人来共同完成一个项目，这是对经理的最大的考验。

1.5 大小公司

在决定了职业方向之后，下一步就是看选择进大公司还是小公司。有一句俗话说：大公司看文化，中公司看待遇，小公司看发展，大致可以作为你选择的依据。

1.5.1 大公司

对于选择进大公司的人来说，更多看中的是稳定的工作机会，一般来说想谋求发展的人会选择进小公司，而进大公司的人一般希望能比较长久地在公司里待下去，这样的话能不能待得好很重要的一点取决于公司的文化。如果你喜欢这个公司的文化，那你就会待得很舒服并且还能有向上发展的空间，反之如果你并不认同公司的文化和价值观，那么你会待得很累而且也无法体现自身的价值。但是大公司也有它自身的缺陷，对于年轻人来说，进入大公司就必须把自己降格成一颗螺丝钉，只了解自己所做的那一小块事务，对于公司的全貌无法做深入的了解，就像盲人摸象，管中窥豹一样，时间长了一旦离开大公司，就会发现自己在社会上的实际生存能力很差。中国惠普前总裁孙振耀曾经谈到："外企员工的成功很大程度上是公司的成功，并非个人的成功，西门子的确比国美大，但并不代表西门子中国经理比国美的老板强，甚至可以说差得很远。而进入外企的人往往并不能很早理解这一点，把自己的成功90%归功于自己的能力，

Offer

实际上，外企公司随便换个中国区总经理并不会给业绩带来什么了不起的影响。”换总经理都不会有什么影响，更何况是普通一兵？所以，如果要进大公司，我们必须很清楚地了解：我们进大公司想谋求什么？有的人认为进了大公司就是图个稳定，就像进了保险箱一样，只要不犯错误，就可以安安稳稳到退休，但随着经济竞争的加剧，大公司事实上也并不比小公司稳定多少，从 2007 年年底开始的金融风暴就证明了这一点，很多大公司都开始选择裁员，所以仅图安稳是绝对不可取的。

综合来看，我认为大公司有以下几点好处。

1. 培训机会

大公司一般会提供各种各样的培训，有些是与具体工作直接相关的，有些是软技巧的培训，我认为后者的价值要远大于前者。大公司里政治斗争复杂激烈，如果不善于处理人际关系，不善于表达自己，是很难生存下去的，通过这些培训，新员工可以迅速成熟起来，并且在这些方面得到更多的锻炼。

2. 专业性

一般小公司里杂事比较多，甚至端茶倒水都要亲力亲为。而大公司往往有专人负责这些事务，员工可以专注于自己手头正在从事的工作。

3. 福利待遇

大公司普遍要优于小公司，不论是医疗保障，还是住房补贴或交通补助等都应有尽有。另外如果你希望出差时是坐飞机而不是坐火车，住五星级酒店的话，那么大公司是不二选择，尤其是全球 500 强的企业。这些是很现实很庸俗的考虑，但不得不说它也是大公司吸引人才的重要手段。

4. 人脉

由于大公司里吸收了很多同等学历同等背景的人才，很容易在这里找到和自己志同道合的朋友，不论是一起在公司里做事还是将来一起做自己的事业，都是很宝贵的财富。

但大公司的缺陷也是明显的，普遍来讲，大公司培养出来的员工都是专才，只懂某一项业务，甚至只会使用某一种特定工具，特别是由于长期在大公司里从事同样的重复劳动并且拿着较高的薪水，容易使人产生一种错觉，这样简单的工作就应该拿这么多钱，从而失去上进心。一旦失业，再想找类似的工作很难。

1.5.2 小公司

相对来讲，小公司里由于人手少，每个人都要身兼数职，每个员工的综合能力可以得到长足的发展。它的好处是，即使失业，也很快可以找到合适的工作。但在福利待遇等方面，无法和大公司相比。另外，小公司看发展，所以在选择小公司的时候，一定要了解清楚小公司的商业模式，当前的经营状况，未来发展前景如何，你自己在小公司当中所处的位置，等等。如果能找到一家正处于上升途中并且未来发展很有前景的小公司的话，未必比大公司要差，甚至要远好于大公司。

1.6 中企外企

随着中国加入WTO，外企在中国的“超国民待遇”逐渐被取消，并且中国的内资企业也逐渐开始走出国门，国企和外企的概念正在逐渐模糊。比如像华为、联想这样的企业，它们的薪资待遇并不比外企差，而在发展空间上对于国人反倒更有吸引力，所以单纯以资本构成来选择企业并不明智。

1.6.1 语言差异

但是外企和国企私企之间的差别并不是完全没有，并且这些差异在相当长一段时间内也不会完全消弭，所以这里还有必要分析一下。这些差别主要表现在文化及语言方面。对于外企员工来说，英语水平至关重要。如果想进外企的话，英语的听说读写必须要过关。如果英语水平不够，即使勉强能混进门，但语言上的劣势会导致你只能在低层次的岗位上徘徊。

外企里的另外一道看不见的障碍是现实存在但又讳莫如深的种族差异，对于职位较低的员工来说，这一点感觉不是很明显。但在做到一定层次之后，你会发现很难有向上发展的空间，因为外企公司里的高管基本上都是外国人，再其次是港台人，土生土长的大陆人极少有机会进入外企真正的高层。

1.6.2 文化差异

另外，外企的文化和国企不同。一般我们在国企，见到老总都需要叫某总，而在外企里，则不论是位置多高的官员，也都只以名字相称，以表示亲切。此外，外企在管理方面比国企更具人性化一些，如果是研究性岗位，基本不计考勤，来去自由，只需按时间把工作完成即可。还有开会以及决策方面，很少搞一言堂，基本都是鼓励大家积极发表意见，即使几百人以上的大会（Town Hall Meeting），老板讲完话之后也会给员工留出提问时间，并且鼓励员工提很多很尖锐的问题，比如，公司最近业绩不佳，你本人是否对此负有责任？你认为公司的股票价格什么时候会回到正常价位？等等。

另外，员工之间文化上的冲突也屡见不鲜。由于从小受教育方式的不同，使中外员工之间在共事时会有隔阂感。举例来说，在中国的课堂上，如果老师问：大家还有问题吗？一般是没有学生举手的，老师也就认为大家都听懂了；而在国外的课堂上，如果老师问大家有没有问题，竟然没有一个人举手，老师会认为自己的教学是失败的，连问题都问不出来，说明学生彻底没听懂。由于从小教育这种潜移默化的作用，外国员工普遍认为中国员工缺乏主见，不自信，关键问题上态度不够强硬等等，而中国员工会认为外国员工粗鲁，不体谅别人。

总体来讲，外企的文化更倾向于人性化及民主化，尽力使公司里每个员工都感觉很受尊重，以便最大程度地发挥员工的主观能动性，从而实现公司利益最大化。而国企更强调员工纪律和执行力，在决策上讲究高效率。所以到底选择国企还是外企，要根据各人不同的能力、性格和需求来决定，不能一概而论。

第2章 简历撰写

明确了你的职业方向之后，我们来看一下简历该如何写。在现代社会，几乎所有求职都要从简历开始，可以说，一份好的简历至少决定了一半以上的机会你能不能谋到一份你理想中的工作。

事实上，简历是你整个求职文档的核心，但在核心之外必然还应该有一个合适的包装。一般来讲，这个包装包括 2 部分：求职信和推荐信。我们会在后面着重介绍一下这两者。

2.1 基本要求

简历可分为目标型简历和资源型简历 2 种。所谓目标型简历是指你已经明确地预知你所要加入的公司，所要从事的职位，因而有针对性地制作的简历，它的主要投放对象是相关公司的人事部门负责人；而资源型简历则与之相反，你并不清楚你将来会进哪家公司，或者准备应聘哪个职位，所以只是提供你目前所拥有的知识、技能和经验背景作为资源以供对方挑选，它的主要投放对象是猎头。

一般建议是你最好先从资源型简历作为开始，先把资源型简历写好，然后以此为基础有针对性地修改它，使之变成目标型简历。

不论是资源型简历还是目标型简历，以下几个通用性原则不可违反。

1．真实性

真实性是简历的最基本原则。虽然说简历是推销自己的手段，有点类似于菜市场王婆卖瓜，自卖自夸，容易使人产生夸大的倾向。但求职毕竟不同于卖菜，卖菜是一锤子买卖，而求职是谋求一份长期合作的机会，如果简历中有很大的虚假成分，很可能会在面试时被识破或者作背景调查时被揭穿，并且会在以后的合作中造成自己都无法想象的严重后果。

所以，你应当尽全力保证你的简历里的所有内容都是真实的。真实是最强大的力量，只有真实才能带给你自信。当然，真实不意味着你必须自曝其短，因为简历是你自己的作品，不是别人制定的表格，所以你当然可以扬长避短，突出你的优点，忽视那些弱点。比如一个班里的优等生说他门门功课都是优，这是真实的；如果你成绩很差，也要说自己门门功课都是优，这就是欺诈了。但你完全可以选择在简历里不谈你的成绩，这并不构成欺骗。除非有

像 Google 这样的公司明确要求你提供上大学时的成绩单，即便如此，难道你能提供一张假成绩单吗？

2．简洁性

虽然说简历的长度并无一定之规，有人写几十页如论文一般的也有，有人写寥寥几句话的也有，但一般来说，简历的长度还是以一满页纸到两满页纸为宜。太短的容易使招聘人员认为你太不认真，太长的会使招聘人员找不到重点，失去耐心，毕竟他想找的只是一个合适这个岗位的人，而不想花那么长时间去读你的论文。

一般来讲，简历当中应当至少包含以下信息：教育经历，工作经历，谋求职位。

工作经历的顺序应当是倒序的，即以最近的经历靠前。工作经历要描述你的主要工作职责，工作成绩，并且最好有量化的指标，比如曾经带领过多少个人的团队，在多长时间内做出了什么数量的成绩，等等。

教育经历的排列顺序也是倒序，把最高学历排在最前面。

3．条理性

描述清楚你的工作职责范围，如果你不知道如何描述的话，找别人的简历参考一下。但是要切实理解别人简历当中的每一句话，不要生搬硬套。

同时，要注意排版时的字体及格式，尽量使用同一种字体，中文应选择宋体，英文应选择 Arial、Verdana 或者 Times New Roman。不要过度使用粗体、斜体或者带下划线的字体，字号的选择以英文 10 号或者中文五号为准。

2.2 主要内容

一份合格的简历至少应当包含以下内容。

① 姓名：包括中文姓名及英文姓名。

② 性别。

③ 联系方式：包括手机号码及 E-mail 地址。家庭住址可以不必包含，但建议说明一下现居住城市。

④ 工作经历：要写出你任职过的公司的全名，你所担任的职务，工作年限要具体到月，具体负责的事务，做出了哪些成绩等。如果经历较多的话，离现在最近的要多写，而比较远的只要提供一个简要信息就够了。

⑤ 教育经历：你就读的学校，获得的学位、在读年份、专业等。

⑥ 奖励或专利等：如果你在某单位曾获杰出员工奖或拥有某项与所应聘工作有关的专利的话，可以列在这里。在学校期间所获奖励不要写，因为你是在求职，所要表现的是你在职业方面的所长。

其他一些涉及个人隐私的部分不必包括。随着国外反歧视风潮的盛行，国外的员工开始注

重保障自己的权益不受侵犯，这些歧视包括种族歧视、性别歧视、地域歧视、年龄歧视甚至容貌歧视等。所以，凡是与职业无关的特质都不必包含在简历里。

简历不同于档案，所以档案里要求写的民族，年龄，籍贯，政治面貌，婚姻状况等一概可以不写。在国外因为有明确的反歧视立法，如果雇主要求你说明种族等情况的话，你甚至可以上法庭起诉对方。但在国内目前还没有这样的立法，所以如果对方询问这些细节的话，如果你不反对则可以告诉对方，如果你反对则可以礼貌地拒绝。这些信息可以在你的档案里体现，但简历里不必包含。

2.3 目标型简历

在有了最基本的资源型简历之后，下一步是考虑如何把它修改成符合你所要应聘岗位的目标型简历。

第一步，找到所应聘职位的职位描述（Job Description 或者缩写 JD）。

职位描述的来源一般有 2 个：一个是猎头主动寄给你的，另一个是你自己从求职网站或者大公司的招聘网站找到的。不管是哪种来源，形式都不会差太多，一般会包含 3 个部分：工作头衔、职责描述和基本要求。下面给出一个招聘高级软件开发工程师的例子。

Title: Senior Software Development Engineer

Location: Beijing

Position Overview:

Are you passionate about developing world class IT solutions for the best technology company? Are you interested in implementing capabilities for enterprise systems using many different technologies and third-party products? Are you able to solve technical challenges and business ambiguities and provide the best solutions?

Our team is a young and dynamic team that has grown tremendously since it started a year ago. We are facing exciting opportunities to provide growing business value with the best technology solutions. We are looking for an enthusiastic Senior SDE to join our Development Team in Beijing. You will demonstrate your exceptional technical and leadership skills, and have large impact on the future direction of the team.

Responsibilities

1. Drive the functional and technical design for the team, and contribute to the functional and technical roadmap of the application.
2. Develop functional and technology prototypes to prove out concepts and demonstrate to business partners.
3. Develop key, high impact modules for the applications you own.

4. Contribute to the technology and process standards of the team, and drive compliance through design and code reviews.

5. Bring leadership and extend influence on the team with a focus on long-term quality and short-term results.

Requirements and skills:

1. Outstanding track record of delivering products from requirements to production.

2. Strong experience in .NET, C#, ASP.NET, SQL Server, XML, Web Services, or equivalent technologies.

3. Strong SQL coding and database design skills.

4. Ability to work effectively with related teams, including Project Management, Test and Operations plus the on-shore Development team.

5. Excellent written and verbal communication skills, with outstanding fluency in the Chinese and English languages.

6. Demonstrated expertise in software architecture, object oriented design, design patterns, and data structures.

7. Good understanding of SDLC process and best engineering practices.

8. B.A. / B.S. in information systems, software engineering, computer science or related fields with 5+ years of hands-on experience in software development.

第二步，仔细研读职位描述及工作要求，找出其中的关键词。

首先，最关键的是工作头衔，如果你确实想要应聘这个职位的话，首先要做的就是把你的通用简历中的求职方向改为这个头衔。比如上面例子中的头衔是：高级软件研发工程师。

其次，阅读职责描述，看是否有和你过往经验相类似的地方，如果有的话，可以把你通用简历中的相关工作职责提到前面，但这里要注意，千万不要直接抄袭职责描述里的内容，而要用你自己的语言描述清楚。上例中关于职责部分的关键词如：functional and technical design，roadmap（功能和技术设计，路书）等。

最后，阅读基本要求。基本要求一般会有一些硬性要求和一些可选要求，硬性要求是必须要掌握的，如上例中 Requirements and skills 部分第 1,2,3,4,5,6,8 条；可选要求如果你也能具备的话则会是锦上添花，如上例第 7 条。如果你的简历里有和硬性要求相关的技能，则必须要把它排在前面，并且在过往职位描述里与之相关的部分加以特别说明，再次如果有与可选技能相关部分，也要突出显示。往往硬性要求比较宽泛，很多人可以达到，比如说要求熟练掌握 C 语言，而可选要求恰恰突出了差异，比如要求对某个音乐软件特别精通，如果你恰好对此软件很熟悉的话，则应该重点突出，这样可以有效地拉大你领先于竞争对手的差距；但切忌无中生有，指望看几本书就敢说自己已经精通某项技术，在面试的时候很容易露出破绽，反而让面试官对你其他所有经历产生怀疑。

2.4 英文简历

关于英文简历的特别注意事项。

1．单词拼写和语法一定要注意

这可以说是最基本的要求了。汉语中的一个词有时候既可以作名词也可以作动词，比如“开发”这个词，用于句头的时候可能是动词，开发过什么软件，用于宾语的时候可能是名词，从事过什么软件的开发。但在英语中这是两个不同的单词 develop 和 development，所以一定要注意词性。类似这样的例子还有很多，不一一列举了。另外要注意时态，一般来讲，简历中的工作经历部分都是过往事件，都应该用过去式。比如应该用 Developed，而不应该用 Develop 或者 Developing。最基本的检查方法是利用 Word 里的自动拼写检查，但并不能完全依赖 Word，还需要靠自己的能力检查，或者有条件的话，找英语好的老师或朋友帮忙修改。不管你的英语水平实际如何，最起码不要在简历里的书面语言上就给人落下一个英语不好的印象，不要让你的面试机会由于英语单词的拼写错误而白白失去。其实这一条对于中文也适用，所以也要反复检查你简历里的中文部分，不要有错别字，不要的、地、得乱用，有些面试官在这方面非常细心，哪怕只是一个错别字都有可能使你失去一个机会。

2．注意英文的习惯用法

不要把英文简历写成 Chinglish 简历。多读一些网上现成的英文简历原文，特别是和你的职业贴近的简历，确实搞懂每个单词、词组的用法。用词不要千篇一律，每个段落都是 Developed something 或者 Managed something 会给人落下一个你词汇量贫乏的感觉，同一种事情可以换一种说法，英文里有大量的同义词、近义词可供选择，最理想的情况是一篇简历当中一个关键单词只使用一次。

2.5 模　　板

简历的模板大约是 Microsoft Word 里历史最悠久的模板之一了，每个版本的 Word 里都有很多可供选择的简历模板，从最早期的脱机模板到现在的联机模板。你所要做的是反复试验，寻找一个真正适合你的模板。

下面给出一个中文样例和英文样例，供你参考：

2.5.1 中文简历示例

姓名：王刚
性别：男
手机：13912345678
现居住城市：北京
E-mail Address：　12345678@sohu.com

应聘职位：网络系统工程师

个人简介

具备丰富的工作经验，认真踏实负责，且具备优秀的表达能力，曾多次主持对用户和内部的技术讲解和培训，获得用户和公司的一致好评。

工作经历

*1998年5月至今××公司 网络系统工程师

- Cisco、IBM网络产品的技术支持
  网络系统方案（局域网和广域网）的设计和规划，解答用户的疑问，根据用户需求提出最佳解决方案；Cisco、IBM网络产品的现场调试和系统维护。
- 客户技术培训，及公司内部的技术交流与培训
  网络基本原理及技术：LAN、WAN、TCP/IP和ATM等，以及Cisco、IBM网络设备调试过程；讲解IBM AIX基本系统管理及高级系统管理、Netview、NFS、HACMP等。
- 曾经参与的项目
  设计、安装、调试"ABC工程"——某省电信综合管理系统ATM网络及主机系统，某卷烟厂、某银行及某出版社信息系统；设计某省邮电办公信息集成系统，某市广电ATM宽带综合业务网、某省有线电视宽带网。

*1996年9月至1998年3月××公司 系统管理员/工程师

- Internet网络信息中心的系统管理
  熟练掌握UNIX（Sun Solaris）操作系统、网络管理并参与组建了Internet网络中心（包括网络设计、安装系统、联调、维护、网络编程）。
- 某网6城市网络站点的建设
  参与各站点Internet网络中心的总体规划和建设及其与卫星主干网的连接，对主干卫星网、X.25分组交换、Frame Relay、ISDN和DDN及其相关设备有所了解。
- Internet网络中心Web系统管理员。

教育背景

北京大学1993.9至1997.7计算机专业，获学士学位。

另：其他培训情况

- 微软认证系统工程师MCSE培训
- CISCO认证网络工程师

2.5.2 英文简历示例

Joe Employee
555 Main Street
Sacramento, CA 95628
myname@myemail dot com
(555)555-1111

SUMMARY

A results-driven, customer-focused, articulate and analytical Senior Software Engineer who can think "out of the box". Strong in design and integration problem solving skills. Expert in Java, C#, .NET, and T-SQL with database analysis and design. Skilled in developing business plans, requirements specifications, user documentation, and architectural systems research. Strong written and verbal communications. Interested in a challenging technical track career in an application development environment.

Experienced in:

- Engineering web development, all layers, from database to services to user interfaces.
- Supporting legacy systems with backups of all cases to/from parallel systems.
- Analysis and design of databases and user interfaces.
- Managing requirements.
- Implementing software development life cycle policies and procedures.
- Managing and supporting multiple project.
- Highly adaptable in quickly changing technical environments with very strong organizational and analytical skills.

EMPLOYMENT

*E*Trade Financial, Sacramento, CA July* 2002～*Present*
Software Engineer (Customer Service Systems)

- Re-engineered customer account software systems used by brokerage teams. Web developer for user interfaces to trading inquiries, support parallel systems.
- Developed and implemented new feedback system for users concerns, bugs, and defect tracking regarding use and functionality of new interfaces.
- Coded web designed interfaces using Java, XML, XSL, AJAX, and JWS.

- Support system for existing intranet for employees, including designing and developing the Advantage@Work system company wide.
- Code and support provided through ASP.NET, T-SQL, Microsoft SQL Server, and Oracle 9i.
- Collaborated in the development of in-house development of new banking software interfaces. Supported existing legacy system to provide newly created cases and insured they were available in the systems in parallel until legacy systems were retired.

Intel Corporation, Folsom, CA Jan 2000～*Jul* 2002

Systems Programmer (Remote Servers and SSL Product Analyst)

- Deployed and tested Remote Installation Services(RIS)-Server Installs on Windows XP.
- Focused deployment of server builds and handled some client builds.
- Modified Visual Basic applications for use in post-server builds for customizing builds.
- Researched RIS and Active Directory for future deployment world-wide. Presented findings to both the Networking Operating System Network Technology Integration team and the Microsoft Joint Development Team (JDP) at Intel. Produced a document binder for RIS and Active Directory to follow the project to the next team representative.
- Wrote bi-monthly progress reports, participated in weekly staff meetings and JDP team meetings designed to develop white paper processing.
- Provided technical support to the SSL team, managing inventory.
- Participated in testing and use of new SAP system as it was integrated into Intel.
- Managed chipset products for IO Business Units.

CSU Chico, Chico, CA 2000～2002

Business Department (Visual Basic Teaching Assistant)

Computer Science Department (Supervisor MS Office Suite Teaching Assistant)

- Supervised all lab assistants, guiding them with student project development.
- Provided one-to-one guidance with Visual Basic programming instruction techniques.
- Wrote small program projects for assignments.
- Presented structured learning labs where students could ask questions regarding Visual Basic programming construct and syntax.
- Prepared structured teaching guides pertaining to chapter material that complimented the

lectures by the professor.

- Provided customized software for tracking student progress throughout the semester. It included reporting for the professor on assessments, projects, homework, lab work, attendance, and overall grades.

SOFTWARE SKILLS

Experience with:

- Databases: MySQL, Oracle, Access, SAP.
- Software: Microsoft Office, Remedy, Microsoft SQL Server, DB Artisan, Eclipse, Visual Studio.NET, FrontPage.
- Languages: C#, Java, Visual Basic, ASP, XML, XSL, JWS, SQL, and T-SQL.

EDUCATION

CALIFORINA STATE UNIVERSITY, Chico, CA

BS Computer Science/ Business Minor

4.0/4.0 GPA

COLLEGE OF THE SISKIYOUS, Weed, CA

AS Computer Science

2.6 求 职 信

除了简历本身之外，一封得体的求职信（Cover Letter）也会使你的求职获益良多。很多人只注意写简历，而完全忽视了求职信，在发给招聘人员的信件里只是简单的一句话：我希望应聘某职位，这是我的简历，请查阅。这样的信件一来是平淡无奇，无法给人留下印象，而更严重的是，这样的邮件隐含着不礼貌和傲慢在里面。实际上，虽然招聘人员每天要看大量的邮件，你每天可能也要发大量的简历，但一封亲切的有人情味的求职信依然会吸引招聘人员，表现你求职的诚意，从而增加招聘人员对你的好感，增加他阅读你的简历的机会。

仔细想一想，难道你的简历真的能百分之百说明你的经历，你的特长吗？简历由于其简，其关注点是在通用性地介绍你的经历，而不能完全反映你的性格，你的态度，所有这些最恰当表现方式恰恰是在求职信里。

一封好的求职信可以充分说明你目前的状况，你对于职业的考虑，对于行业的思考，对于你所要应聘的公司的认识，对你所要应聘的岗位的认识等等。从某种程度上来说，求职信有时会比简历还重要。

写求职信时有如下几个注意事项。

（1）长短合适，宜短不宜长。一般有两三段即可，大量的信息已经包含在你的简历当中，不必在这里再重复。

（2）描述一下你是如何知道这个职位的，以及你为什么适合这个职位。

（3）描述一下你对公司的看法，表达自己对于应聘公司的诚意。

下面给出一封求职信的范例以供参考。请注意：这不是模板，好的求职信必须融入自己真实的理解及感情，有时候还需要专门针对某公司做深入的调查研究，才能写出令人信服的求职信。

Dear Hiring Manager:

This letter is to express my interest in discussing the Senior Programmer Analyst position posted on the Company web site. The opportunity presented in this listing is very appealing, and I believe that my strong technical experience and education will make me a very competitive candidate for this position.

The key strengths that I possess for success in this position include, but are not limited to, the following:

* I have successfully designed, developed, and supported live use applications.
* I am a self-starter.
* Eager to learn new things.
* Strive for continued excellence.
* Provide exceptional contributions to customer service for all customers.

With a MS degree in Information Systems Management I have a full understanding of the full life cycle of a software development project. I also have experience in learning and excelling at new technologies as needed. My experience includes but is not limited to:

* Customer service and support.
* Programming both new applications and maintenance work.
* Problem isolation and analysis.
* Software quality testing.
* Application and requirement analysis.
* Process improvement and documentation.

Please see my resume for additional information on my experience.

ABC Company is a company that would provide me with the opportunity to put my personality, skills and successes to work. At a personal meeting I would like to discuss with you how I will contribute to the continued growth of your company.

I can be reached anytime via my cell phone, 555-555-5555. Thank you for your time and consideration. I look forward to speaking with you about this employment opportunity.

Sincerely,

FirstName LastName

中文如下。

尊敬的人事部主管：

您好！我是看到你们发布在 XX 网上的招聘启事后来应聘高级程序分析师职位的。这个工作机会对我来讲很有吸引力，而我相信我坚实的技术经验和教育背景使我非常适合这一职位。

我适合此职位的主要优势包括以下几点：

* 我曾经成功地设计，开发并支持过相关的应用程序。
* 工作主动积极。
* 渴望学习新知识。
* 刻苦努力。

* 为所有客户提供杰出的服务。

作为一名信息系统管理专业的硕士毕业生，我对于软件开发项目的全部生命周期有完整的理解。同时我在学习并掌握新技术方面有很强的实际经验。我的经验主要包括以下几点：

* 客户服务与支持。
* 编写全新软件及维护已有软件。
* 问题隔离及分析。
* 软件质量测试。
* 应用及需求分析。
* 过程改进及文档管理。

具体细节请参阅我简历中的相关部分。

ABC 公司是一家能够提供给我机会来发挥我的所长的公司，我相信我的技术和经验能给贵公司带来重要价值。我希望能和您面谈来详细讨论我可以怎样在公司里做出贡献以支持公司的长期持续发展。

您可以随时打我的电话和我联系，我的电话号码是 555-555-5555。感谢您的时间和考虑。期望能有机会和您面谈。

诚挚的，
姓名

2.7 推 荐 信

求职资料中另外一个关键资料是推荐信，如果运用得宜的话，可以起到事半功倍的效果。推荐信一般出自你过去的上司，在你离开上一个工作岗位之前可以礼貌地请他或她为你写推荐信，如果老板比较忙，也可以你先草拟一个底稿，经他修改之后再签字确认。

下面给出一封推荐信范例以供参考。

Company Name
Company Address
City, State, Zip
Phone number
Date

To Whom It May Concern: (or name of contact requesting reference)

Joe Employee worked for me as a Software Engineer from September 1,1997 until May 23, 2004. His responsibilities included requirements gathering, analysis and design of complex web applications using a variety of technologies.

During the course of his employment, Joe proved himself to be a dependable employee, and a hard worker, with solid problem solving and technical skills. I was always impressed by Joe's ability to complete the work assigned to him on time.

Overall, Joe is a talented, hard-working employee, and I am sad to see him leave. I strongly recommend Joe for any mid-level development position.

Sincerely,
Manager Name
Manager Title

中文翻译如下。

致有关人士：

王刚于1997年9月1日到2004年5月23日期间在我部门任软件工程师一职。他的工作职责包括：使用各种技术手段为一个复杂的网络应用程序进行需求收集，分析和设计工作。

工作期间，王刚踏实肯干，工作认真负责，有极强的解决问题能力以及出色的技术实力。每次交给他的任务都能按时完成，给我留下了很深的印象。

总的来说，王刚是一个头脑清醒，工作努力的同事，对于他的离职我感到非常遗憾。我认为他是软件项目中层开发职位的合适人选，特此推荐。

诚挚的，
经理姓名
经理职位

2.8 其他手段——博客、网站

随着 Internet 的飞速发展，各种新型的网站工具层出不穷，从最早的独立网站，到个人博客，再到现在的 SNS，Twitter，作为 IT 界一员的你如果对这些工具不熟悉的话，很难让人相信你是一个时刻跟随时代潮流的优秀技术人员。你的博客就像是你的一张活动名片，应该能够反映你所熟悉的技术领域，你所攻克的技术难题，一方面可以供别人参考了解你，另一方面也可以作为你自己的一个技术储备仓库供自己日后查阅。如果想应聘外企的话，不妨多写几篇英文博客。当然博客的内容可以引用别人的，但切忌照抄。如果是别人的某篇博客引发了你的某种思考，可以放一个链接指向他的原文，这样会使你的博客看起来非常专业。

如果想在外企求职的话，有一个社会化网站 Linkedin.com 不可不加入。Linkedin 是目前网络上最火的专业人士社交网络，它提供标准化的模板引导你逐步完成自己的简历，并且提供入口可以让你以前的主管或老板对你的工作作出评价。很多猎头公司以及大公司的人事部主管会在 Linkedin 里搜索他们满意的人选。

2.9 简 历 投 放

简历的投放途径有以下几种。

1．寄给猎头

猎头几乎是寻找中高级职位人士必经之路。猎头公司手里一般掌握着两方面信息，一方面是企业招聘职位的信息，另一方面是大量高级人才的信息。他们会根据职位在自己的库里搜寻合适的人才，或者根据人才经验背景及需求搜寻合适职位。

2. 直接寄给招人的公司

由于猎头公司希望增加推荐的成功率，有时候难免会夸大其词，而且即使经过猎头初步筛选的人才也还要经过用人单位进一步筛查，对于用人单位的人事部门和部门经理来说招聘工作量并没有减轻多少，所以现在很多公司干脆跳过猎头，直接面对求职者。比较而言，没有了猎头这一中间缓冲层，这样的招聘对于应聘者来说往往难度更大。

3. 寄给招聘网站

现在已经有很多招聘网站，国内比较著名的如招聘网 zhaopin.com、前程网 51jobs.com 等，他们都提供很方便的入口可以输入你的简历。但这些网站目前来讲，职位都偏中下，高级职位很少或者几乎没有。

4. 在招聘会上投放

招聘会这种形式曾经在 20 世纪 90 年代一度很火爆，目前则基本上是面对应届毕业生，或者提供一些初级、入门级职位。

第3章 面试

简历寄出之后，下一步要准备的就是面试了。如果你的简历得到了用人单位的认可，他们会主动联系你，和你安排时间面试。面试一般会分好几轮，形式有电话面试，面对面面试，团体面试等多种。

3.1 准备面试

不论接下来的是何种方式的面试，你都必须从现在开始准备了。

1．研究公司

在你参加面试之前，最重要的就是要对你所要应聘的公司有充分的了解。只有对公司有了解，你才能更有把握地回答问题并提出自己的问题，而且你还可以借此机会研究公司的文化，决定自己是否喜欢和适合这份工作。

你可以通过多种方式了解公司。

（1）访问该公司的网站，了解他们的经营范围、产品、服务、公司历史、管理团队以及公司文化。这些信息通常可以在关于我们栏目里找到。

（2）用搜索引擎 Google 或者百度搜索一下其他第三方关于该公司的评价以及该公司近期的新闻。

（3）看看朋友们是否有熟悉的人在公司里或者曾经参加过他们的面试，可以向他们了解一下。

2．准备面试

（1）熟悉你的简历。

把你的简历打印出来，反复检阅，特别是关于公司名称，公司名称的英文译名，你的职位，在职年限等，要确保烂熟于胸中。

（2）准备面试问题。

对一些常见的面试问题要准备好如何回答，具体问题请见后面章节。

（3）练习面试。

仅有理论知识是不够的，最好的老师是实践。如果有条件，找家人或同学模拟几次面试，

请他或她帮你挑出一些显而易见的毛病加以改正。即使没条件，也要自己一个人对着镜子大声说出拟想中面试时问题的答案，看看语气、语调和表情上有没有什么需要纠正的地方。

3．面试穿着

面试时最合适的穿着就是一身合体的西装，颜色以深色为主，黑色蓝色灰色均可，如果能有八成新的最好，实在没有，也可以穿新西装去。里面穿白色或浅蓝色衬衣，素雅的领带，黑色皮鞋，头不要太尖。切忌穿夹克、仔裤、拖鞋等非正式服装。如果没有合身的衣服的话，这时候需要去添置一套。但皮鞋尽量找舒适的旧鞋，新皮鞋硌脚，会让你很难受。

很多人认为面试时穿什么不重要，特别是技术人员，往往一厢情愿地认为：我是搞技术的，又不做销售，穿那么正式干什么？但事实上，穿着打扮会在相当程度上影响你面试的成败。第一，因为大家在技术层面上实力相较并不大，一个好的精神面貌会使你占据很大优势；第二，正式的服装会增强你的自信心；第三，使面试你的人觉得你很重视这次面试，体现你做事情很认真；第四，你今天来应聘的是技术人员，但未来的某一天也许你会带领一个团队甚至代表公司去谈判，你需要展示你有让公司向上培育你的潜力。

还有一个误区是觉得对方公司的文化很随意，特别是一些互联网或者游戏企业，比如Google这样的公司，大家都是穿T恤牛仔裤上班，所以以为自己去面试时穿着也可以随意。这种想法也是不对的，人家公司的文化随意，那是进了公司门之后的事情。在未进门之前，还是应该很郑重地参加面试，这是最基本的礼节。

3.2 面试方式

面试一般在简历初试合格之后，由单位人事部门或者由猎头通知你。面试的形式各公司会有所不同。大致上可分为电话面试、面对面面试和笔试 3 种。面对面面试又分为单独面试和群体面试 2 种，由于应答策略不同，我们下面分别叙述。

面试的次数根据公司文化以及职位的重要程度，各公司也不相同，少则二三轮，多则六七轮。有的时候会先安排电话面试，后安排面对面面试，但也有时由于老板在国外，会先安排本地员工作面对面面试，最后再和老板在电话里或者电视电话里作面试。不论哪种情况，听人事部门安排就行了。

3.2.1 电话面试

电话面试一般用于在正式面试之前作进一步筛选或为了节省差旅费用。我们先来看一下作为应试者电话面试所应注意的事项。

1．做好准备

和正式面试一样，在电话面试之前也要做好各项准备。

（1）把你寄给对方的简历打印好放在面前。由于你可能有多个不同版本的简历，注意不要

弄错了。

（2）准备一些常见问题的应对，具体内容可参看后面章节。如果有条件的话可以先请家人或朋友帮自己练习一下，录音记录，看一看自己是否经常结巴，是否经常说嗯、啊等口头语，并有针对性地改正。

（3）由于电话面试时间可能比较长，把你的手机呼叫转移到一部座机上会比较方便，如果条件不具备的话，至少也要确保手机处于充满电的状态。如果手机有呼叫等待功能的话最好关闭，以防止面试途中有人打电话进来。

（4）找一个安静的房间，确保没有外人的吵闹，关掉电视机或者收音机，关上房门。

（5）准备一杯水在旁边，万一谈话过程中口渴了可以喝两口。但别喝太多，我想你也不希望电话过程中突然内急吧。

（6）如果时间不合适，询问对方是否可以换个时间，并且提供几个时间项供对方选择。

2．面试进行中应当注意的问题

（1）保持微笑。尽管对方看不到你的表情，但他能感觉到你的语气和音调，并且你的微笑是对自己最大的鼓励。

（2）放慢语速，特别是在做英语面试的时候。如果你平常语速比较快的话更应特别注意，说得快并不意味着你英语熟练或者专业纯熟，关键是每个词说的都是都对的，这比说得快重要的多。

（3）称呼对方的时候要用您，不要用你。

（4）如果对方正在说话，一定要等到他说完再说，千万不要中途打断对方。

（5）如果遇到不容易回答的问题，可以请对方给自己点时间思考。如果确实有困难，则应礼貌地告知对方，不要一言不发。

（6）电话面试的目的是希望能获得下一个面对面面试的机会，所以在结束的时候一定要问是否可以有机会面谈。

3．面试之后

（1）把面试中你被问到的问题和你的回答尽可能记录下来，以便于将来总结经验教训。

（2）面试结束之前一定要说谢谢。同时还应写一封感谢信，感谢对方给你面试机会，并重申你对这个职位的兴趣。

3.2.2 面对面面试

如果电话面试没问题，接下来就是面对面的面试。面对面的面试有两种方式：一种是经理本人和你单独面谈，多个经理的话会安排多轮面谈，多见于美国公司；另一种是一群经理和你同时面谈，多见于欧洲公司。这里我们先谈一下单独面谈的要点。

1．准备面试

和电话面试一样，面对面面试也需要做准备工作。

（1）准备一个公文包或夹子，带上两份打印好的简历以及几张白纸，带上2支笔，检查一

下钢笔的出水是否流畅。

（2）面试前一天，去理个发，洗个澡，把指甲修剪整齐，把第二天要穿的衣服准备好，把皮鞋擦亮。香水的问题，男士女士都可以喷，但不可过浓，一两滴即可。对于男士来说，如果没有体味的话也可以不喷。

（3）再次复习一下之前研究的公司概况，准备一下常见问题的应对方式。

（4）准备好闹钟，把闹钟调到稍早的时间，因为路上的情况复杂，可能堵车，所以要提前一些，宁可早到，不可迟到。

（5）出行工具方面，如果不是自驾车的话，既然都穿西装了，就不要再骑自行车或者挤公交地铁了，直接打车过去，可以保持一个良好的心情。在车上嚼个口香糖，记得下车前把口香糖吐在纸里扔到垃圾桶，别见了面还在嚼。

（6）把手机彻底关机，连震动也不要，省得分心。

2．面试进行中

做完了以上那些准备工作，你已经武装到牙齿，可以上战场了。

（1）始终微笑，这是永远的秘诀。不论遇到多么困难尴尬的问题都应该保持笑容。经过了如此漫长的筛选，连面都见到了，还有什么值得忧虑的吗？你当然相信：你就是最好的，最适合的那个人。

（2）关于握手的问题，通常情况下，见面时握一次手，面试结束要离开时再握一次手。握手时不应过于虚握，也不应过于用力，手掌接触时间以一到两秒为宜。切记：你要主动伸出手来，不要等面试官伸手。

（3）保持冷静。面试过程中可能会遇到一些困难的问题，不要急于回答，想清楚了再开口。

（4）学会倾听，切不可打断面试官的谈话，一定要等他说完了你再说。同时听的时候要注意能够真正听懂他的话，做到尽量一次回答准确。比如他说请你介绍一下你的经验，他想听到的是与该职位相关的经验，所以你一定要贴近这个职位谈，而不要东拉西扯一些与此不相干的东西。

（5）知道什么该说，什么不该说。一切问题都要想清楚再开口，不论对方提的问题有如何的诱导性，切忌说任何人，包括以前的老板、同事、朋友或家人的任何坏话，以及对以往任职的公司的任何不满。很简单的一个原则：你看待这个世界的方式是积极乐观的，这就够了。

（6）结束面试时，再次表达自己对这个职位的兴趣，并感谢面试官提供给你这个面试机会，希望将来有机会能一起工作。别忘了主动握手。

3．面试后

（1）和电话面试一样，必须给面试官写感谢信，再次感谢他提供给你面试机会，并表达你对这个职位的强烈兴趣。

（2）回想一下面试中遇到的问题，自己评估一下哪个问题回答的好，哪个问题回答的不好，总结经验教训，以利于以后再战。

3.2.3 常见问题

面试时回答问题的基本原则还是：诚信。特别是在一些基本问题上，比如你离职前的职位是什么？薪水是多少？等等这样的问题，更是容不得半点虚假。现在很多公司会做背景调查，如果你回答问题不诚实，很容易被调查出来，从而对你的职业生涯产生负面影响。

1．请介绍一下你自己

这恐怕是面试中最常遇到的一个问题了，而且往往也是你参加面试之前准备过的一个问题。但在回答的时候要表现自然，不要像背书一样。回答的重点放在与你要应聘的职位有关的事项上，与此无关的一概不谈，除非被刻意问到。介绍自己的工作经历时顺序正好和简历里的相反，从你大学毕业后的第一份工作谈起，最后再谈到目前的工作，工作经历中与应聘职位有关的部分可谈得具体一些，其他无关的一带而过即可。

2．你为什么要换工作

这里要注意一定要对你工作过的所有公司、老板、同事以及环境都要始终保持正面评价，千万不要谈公司管理中的任何问题，否则你会吃大亏。回答这个问题的时候，只需要保持微笑，强调你离职的原因是因为有更好的机会，想尝试做一些特别的事情或者其他正面原因。

3．你在工作方面有什么经验

这是一个表现自己的机会，尽可以把以往和此工作相关的经验都充分完整地讲出来，最好有事实有数据。如果没有非常吻合的类似经验，则回想一下你以往经历中那些最可能接近应聘岗位的经验，把它表述清楚。

4．你觉得你自己事业成功吗

不用想，这道题只有一个标准答案是。然后再略加解释原因。一个比较好的解释是："我设定了我的人生目标，已经实现了其中的一些，现在正在努力实现另外一些。"

5．你的同事如何评价你

在参加面试之前，你要事先联系好一些关系比较好的同事，请他们对你的工作作出评价。这样有两个好处，一是以备日后背景调查时用，二是如果在面试时遇到了这样的问题，你可以马上说出你的同事的名字和他们对你的评价，比如，"我在某某公司的同事王刚经常说我是他见过的工作最刻苦的同事"。这样直接引述别人的话比诸如"他们都觉得我很努力"这样的泛泛而谈更有说服力。

6．你对我们公司有什么了解

看，这就是要你在面试前先研究公司的原因。你可以谈一谈你的调查结果，他们公司的主要业务、经营现状及未来方向等等，别忘了提一下公司老总的名字。也可以谈一下从网上或者报刊杂志上看来的对公司的评价，但不要谈负面的议论。毕竟你是准备谋求这里的一份工作，而不是来批评公司的记者。

7．你通过什么样的方法来提高你的专业技术和知识水平

尽可能说出一些与你应聘职位有关的活动，比如参加相关培训或者经常访问相关网站等，

能说出的方法越多越好，所以你在事先要准备一下。

8．你还在别的公司求职或应聘别的职位吗

在这个问题上你要诚实回答，有就是有，没有就是没有。但即使有，也不要在这个问题上花太多时间，比如去解释在哪个公司应聘什么职位等等。应该把注意力放在你目前应聘的这个职位上，强调你能为公司做出什么样的贡献，能给公司带来什么样的价值等。谈太多别的只能分散注意力，削弱你的应聘形象。

9．你为什么会想来我们公司工作

这个问题在你面试之前需要花时间想清楚，回答时要充分联系你所做的对于公司的研究，说明你的知识，技能和经验如何吻合公司的产品及未来发展方向。要充分表现你为公司工作的诚意，面试官能够感觉得出你是否真诚。同时别忘了联系你的长期职业规划，说明这一选择对你个人来讲也是意义非凡。

10．你认识我们公司的别人吗

有些公司的规定不允许亲戚朋友在一起任职，所以不要轻易回答是。但一般情况下他们问的是你的亲戚而不是普通朋友，所以如果只是普通朋友，你可以不必提起，除非他在公司里很有名。

11．你期望的薪水是多少

很关键的问题。如果你先回答就有可能输掉这场游戏。所以，最好的策略是不正面回答这个问题。你可以说："这个问题很难回答，你能不能告诉我这个职位的大致薪酬范围是多少？"通常情况下，如果前面谈得比较顺利，考官放松戒备的话会告诉你。如果他坚持不告诉你，你可以说："这个要看工作的具体要求，然后给一个很宽泛的范围。"

12．你的团队合作意识如何

你当然是一个好的团队成员。最好能准备一些例子，说明你总是把团队利益放在个人利益之上。注意你说话时的语气，不要用自我吹嘘式的态度，而尽量以陈述事实的方式讲述。在这里语气是关键点。

13．如果我们雇佣了你，你准备干多久再换工作

千万不要给定任何明确的答案，譬如说 3 年，5 年，哪怕是 10 年也不好。给一个模糊的答案，比如，"我希望能做非常长时间"或者"只要公司和我个人都觉得我做得非常好，我希望能一直做下去"。

14．你曾经解雇过别人吗？感觉如何

这个问题一般会问谋求管理职位的人，但要注意这是一个很严肃的问题。千万不要表现得很轻松或者好像你很喜欢解雇员工似的。你要指出如果解雇员工是唯一正确选择的话，你可以那么做。特别是如果员工的表现严重影响了公司利益的话，你必须保护公司利益。但你要搞清楚，解雇和裁员是两码事。

15．你的工作理念是什么

面试官想听到的不是多么花哨的表述。其实很简单，你对于把工作认真完成有强烈的感情

吗？是的，这就是最好的答案。简洁而正面，显示出对公司的价值就够了。

16. 如果你有足够多的钱可以退休养老了，你愿意退休吗

如果你愿意那就说愿意。但是因为现在还没那么多钱，所以还需要工作，而这份工作就是你最喜欢的。当然如果你是工作狂，那你就说不，我愿意奋斗一生。

17. 你曾经被解雇或裁员吗

如果没有，就说没有。如果有，就简单回答即可，不要谈论任何与此有关的个人或公司的负面情况，比如老板管理不善，公司业绩不良等。

18. 你能为公司带来什么样的价值

这应该是一个你期待了很久的问题。它给了你机会正面阐述你对于你所应聘的职位所具有的一切优势，同时你还可以考虑谈一些你在这个职位上未来的发展等。

19. 我们为什么应该雇用你

指出你的背景和经验如何吻合公司要求就可以了。不要把自己和别人比，证明自己比别人强。

20. 能不能讲一个你曾经做出过的工作建议

这个问题最好事先准备好答案。一定要讲一个已经被采纳并且证明取得很大成功的建议。如果这个建议和你应聘的职位有关那就更好。

21. 你的同事里哪一点最让你不满

这是一个带有陷阱的问题。你可以努力回想但最后找不出哪怕一件小事使你不满。好的答案是："我一直和同事相处得非常好。"

22. 你最大的强项是什么

有很多好答案，只要保持正面形象就可以。比如，解决问题的能力、安排时间的能力、在压力下工作的能力、专注于项目的能力、专业技能、领导能力和积极的态度等。

23. 你理想中的职业是什么

还记得我们第一章中讲的职业规划吗？那是给自己准备的，在这里可别直接回答，你不可能赢的。要避开一切具体的头衔，如果你说你现在应聘的这个职位就是你理想中的职业，会让人感觉你很虚伪。但如果你说你的理想是别的职位，又会让人觉得你对这个职位其实并不满意。所以最好的方法是笼统的回答，比如说，你理想中的职业就是能有一份你喜欢的工作，和周围的同事和睦相处，能贡献你的才华，让你迫不及待想要去工作的职业。

24. 你为什么认为你能胜任这份工作

给出几方面的原因，经验、技能和兴趣等。

25. 你不愿意和什么样的人共事

不要谈琐屑的性格。你可以说："我不愿意和对公司不忠实，有暴力或者违法倾向的人共事。"如果你说你不喜欢某种细微的性格，比如沉默寡言、太过张扬等，只会使人感觉你自己是个爱抱怨的人。

26. 对你来说什么更重要，是金钱还是工作

金钱当然很重要，但工作是最重要的。没有更好的答案了。

27．你前任老板认为你最大的强项在哪里

有很多种可选择的好答案：忠诚、热情、积极的态度、领导能力、合作精神、专业性、创新性、耐心、努力工作和能解决问题等。

28．有没有和老板发生不愉快的事情？你是如何解决的？举个例子

这个问题听上去很和善，但其实是最大的陷阱。这其实是在测试你是否会说你老板的坏话。如果你中计了，真的讲了和以前老板不愉快的事情，你的面试可能也就到此完结了。你可以冥思苦想，但似乎以前从没有发生过和老板不愉快这样的事情，就行了。

29．你在工作中感到过失望吗

不要牵涉到琐屑和负面的情绪中去。可以谈的方面不多，但你可以谈谈，比如挑战性不够，或者由于公司业务不好你被裁员了，否则你可以承担更重大的职责等。

30．请讲一下你在压力下工作的情况

你可以说你曾经在不同类型的压力下工作过。举例子时讲一个与你应聘职位有关的压力。

31．你的技能适合这个职位还是更适合其他职位

应该是这个职位。不要给对方对你的猜疑火上浇油，你除了这个职位别的什么职位也不想要。

32．什么样的激励会使你把工作做到最好

这也许是一个只有你自己最清楚答案的问题，但这里有一些好的例子，比如挑战性、成就感、老板和同事的认可等。

33．你愿意加班吗？晚上加班或者周末加班

这个要取决于你个人。应当完全诚实地回答，是就是“是”，不是就是“不是”。不要为了得到这份工作而勉强答应，否则到时会苦不堪言。

34．你如何判定你的事业是成功的

有几种不同的衡量标准，你给自己定了一个很高的标准并且完成了；你的工作成绩非常出色；你的老板表扬你，说你的工作很成功等。

35．如果公司需要的话，你愿意在另外一个城市或国家工作吗

像这样的问题应该在面试之前和家人取得一致。如果实际情况是不能，那你就别说愿意。否则会给你以后的职业生涯带来无数的麻烦。现在的诚实将避免你日后的痛苦。

36．你是否愿意把公司利益放在个人利益之上

这是最直接的关于忠诚与奉献的问题。不用去忧虑潜在的道德上的或者人生哲学上的可能影响，直接回答是就好了。

37．描述一下你的管理风格

要尽量避免谈一些过于明显的风格，比如进取心、感染力或者纪律性等，因为不同的人对这些词的理解不同。比较安全的回答是审时度势，根据当时的形势及时调整管理策略，而不是一种风格贯穿始终。

38．你以前在工作中犯过错误吗？你从错误中学到了什么

有谁从不犯错吗？所以对第一个问题必须回答是，否则你的诚信会受质疑。举例时不要胡编乱造，但要尽可能找那种小错误，最好是有良好动机并且造成的后果也基本是正面的错误。比如说，在参加的一个项目中，属于你自己的那部分进度太快，导致项目协调受影响等。

39．你在工作中有盲区吗

这是一个带点悖论的问题。如果你要是知道这是你的盲区，那它也就不再是盲区了。你不需要自己揭示你个人方面的任何短处，让他们自己找出你的弱点好了。

40．如果是你在招聘这个职位的人，你希望招什么样的人

你应该谈一些这个职位所需要的并且你所具备的特长。

41．你觉得相对于你的资历来说，这个职位太低了吗

这个问题一般会问那些资历比较高而又谋求较低职位的人，用人单位有担心怕你不安心工作。所以不管你以前的资历有多高，都要强调并说明你的资历非常适合这份工作，不存在过高的问题。

42．你准备如何弥补你在这个行业或领域经验上的不足

首先，如果你有面试官不知道的与此相关的经验，你要予以说明；其次，你要说明你是一个勤奋学习并能快速掌握相关知识的快手，最好能举一些实际例子。

43．你喜欢什么样的老板

说一些通用和正面的特点，比如知识丰富、有幽默感、公平、体谅下属和高标准严要求等。几乎所有老板都认为自己拥有这些品质。

44．你曾经解决过争执吗？如何解决的

举一个具体的例子。焦点放在你解决争议的方法，说明你解决问题的能力，而不要过多地谈论争议的细节。

45．你愿意在项目中担任哪方面的角色

如实而谈即可。如果你愿意在项目中担任各种不同角色，也可以说出来。

46．请谈一下你所喜欢的工作风格

要多强调维护公司利益的方面，比如全力以赴完成工作，努力工作并乐在其中等。

47．你工作中经历过的最令你失望的事情是什么

回答此问题的关键是要找一件超出你能力控制的事情，别弄成你应该为此负责的错误。但要表现出你虽然失望但理解并接受这个结局的态度，而不是怨声载道。

48．你工作中最开心的事情是什么

讲一件你为公司完成的事情，它的结果使你很开心就够了。

49．你有什么问题要问我吗

这通常是最后一个问题了，但你必须在事先准备好一些问题。不要问得太泛泛了，要把你自己和公司联系在一起。一些好问题的例子如，“我大概要多久能够熟悉业务”或“我会参加什么样的项目”等等。

3.2.4 感谢信

面试之后的感谢信非常关键，有时候一封小小的感谢信就能使你从竞争者中脱颖而出。一封好的感谢信可以充分表达你的热情与诚意，有时候这些素质比专业技术更能打动面试官。

我曾经面试过的一个工程师，他的专业技术能力在所有应聘者中并不是最强的，笔试成绩也不太好，当时面试完之后并没有强烈的想聘用他的想法。但一天之后，他给我写来了感谢信，坦承他自己在这方面经验有限，但他愿意刻苦学习补上所有不足，并且他把所有在面试和笔试中遇到的问题记录了下来，附上他面试后找到的正确答案，请我帮助订正。这样一种认真的态度当即打动了我，拍板把他定了下来。随后合作的两年时间证明，当初做的这个决定是正确的，他成了全团队里最认真负责的工程师。

下面给出一封感谢信的样例，供参考。

Dear Mr. Kramer:

I would like to take this opportunity to thank you for your time and consideration in interviewing me for the Software Engineer position this past Tuesday. The interview was very informative. I was especially impressed with the cohesiveness of the staff and their support of the company effort. I was also favorably impressed with the flexible management style.

The career opportunities available at RD Labs fit very well with my long term professional goals. I believe I could make a significant contribution to the company.

Thank you again. I look forward to speaking with you next week.

Sincerely,

Jack

中文如下。

尊敬的李先生：

我想借此机会感谢您百忙之中拨冗在上周二对我进行软件工程师职位的面试。这次面试给我提供了很多有用的信息，给我留下非常深刻印象的是贵公司员工的团结和他们对公司的支持与奉献，我同时也非常欣赏公司的灵活的管理风格。

贵公司研发实验室的工作机会非常符合我的长远发展规划。我深信我能够为公司做出杰出的贡献。

再次感谢您！期待着下周能与您再次会谈。

诚挚的，

王刚

3.2.5 笔试

对于应聘软件工程师职位的人来说，笔试并不新奇。本书后面将花大量篇幅讲解各种笔试例题。

这里讲述一下笔试的通常注意事项。

（1）事先了解清楚该公司该职位都大致会用到哪些编程语言。如果你真的很想进某企业的话，一两周突击该语言成功的可能性不大，建议你至少花半年时间专攻该语言。

（2）搞清楚企业的研发方向。即使用的是同一种语言，软件企业和硬件企业对于语言的要求也不同，不同的职位要求也不同。同样是 C++语言，硬件企业喜欢考一些指针引用方面的问题，而软件企业则更看重设计模式等。在这方面不妨找同学或朋友了解一下情况，然后有的放

矢地进行准备。

（3）纸上写程序恐怕是所有程序员最痛恨的一件事情，平常编软件都是在电脑里编，如果哪里错了，一编译就发现了，改正也很容易，在纸上写要求落笔之前脑子里就要有非常清晰的思路，这段程序要用到多少变量，每个变量是什么类型，如何命名等都要在脑子里已经形成，落笔只是把脑子里的程序搬出来而已。但是对付纸上写程序实在是没有什么捷径，唯一的方法就是平常多练，把基本的算法诸如排序算法、查找算法等一遍遍地写，直到能够一口气完全写正确为止。

3.3 待遇谈判

虽然我们开篇谈到职业发展远比金钱重要，但不可否认的是工资待遇会在很大程度上决定你是否会在以后的工作中感到满意，并进而影响到你未来职业的发展。所以谈到一个好的条件，对个人和企业都至关重要。

在讲工资谈判之前，我们先来看一个小故事。

有一天，一位禅师为了启发他的门徒，给他的徒弟一块石头，叫他去蔬菜市场，并且试着卖掉它，这块石头很大，很美丽。但是师父说："不要卖掉它，只是试着卖掉它。注意观察，多问一些人，然后只要告诉我在蔬菜市场它能卖多少钱。"

这个人去了。在菜市场，许多人看着石头想："它可作很好的小摆件，我们的孩子可以玩，或者我们可以把它当作称菜用的秤砣。"于是他们出了价，但只不过几个小硬币。那个人回来。他说："它最多只能卖几个硬币。"师父说："现在你去黄金市场，问问那儿的人。但是不要卖掉它，光问问价。"从黄金市场回来，这个门徒很高兴，说："这些人太棒了。他们乐意出到 1 000 块钱。"师父说："现在你去珠宝市场那儿，低于 50 万不要卖掉。"他去了珠宝商那儿。他简直不敢相信，他们竟然乐意出 5 万块钱，他不愿意卖，他们继续抬高价格，他们出到 10 万。但是这个门徒说："这个价钱我不打算卖掉它。"他们说："我们出 20 万、30 万！"这个门徒说："这样的价钱我还是不能卖，我只是问问价。"虽然他觉得不可思议："这些人疯了！"他自己觉得蔬菜市场的价已经足够了，但是没有表现出来。最后，他以 50 万的价格把这块石头卖掉了。

他回来之后，师父对他说："现在你应该明白，这个测试是要看你是不是有试金石、理解力。如果你不想要更高的价钱，你就永远不会得到更高的价钱。"

工资谈判怎么谈？有一个最基本的原则要掌握：千万不要主动谈。什么时候可以开始谈工资？经过了前面的面试，笔试，一切都合格之后，用人单位会给你出价（Offer），这时候才是谈工资的时候。

（1）事先准备。要从你的朋友同学那里了解你这个行业，这个地区，这个职位的平均工资大约是多少，做到心中有数。

（2）要有耐心。美国幽默作家比林说过：人一生中的麻烦有一半是由于太快说"是"，太慢

说“不”造成的。

（3）除非你志在必得，非此职位不可，否则不要怕拒绝别人。如果你真是合适人选，对方会考虑加薪。但你自己要考虑清楚，如果你拒绝对方的出价，有些公司是会转而直接考虑第二竞争人的。

（4）福利待遇是工资之外很重要的补偿，所以要和工资结合在一起来看。

（5）入职时间要在此时谈定，如果原单位要求一个月时间交接的话，要和新单位说明情况。

（6）签合同时一定要认真阅读，了解清楚合同年限是多长，有无试用期，试用期多长，以及有无违约金。这一点至关重要，如果你不想日后由于过早离职而给自己带来麻烦的话，对于有违约金的合同应当断然拒绝。

我记得2000年互联网泡沫刚破裂的时候，我们企业正准备招一些网站工程师。当时我面试的一个人有在一家大型网站工作的经验，但他大学本科毕业刚两年，还达不到高级程序员的标准。在面试的时候我问他，你期望的工资是多少？他毫不犹豫地说，12 000 元每月。我明确地告诉他，我们最多只能出到6 000元。一周之后，他打电话给我说，6 000元他也能接受。但这时我们已经肯定不会再要他了，如果你的期望薪水这么多，你勉强答应这份工作的话，第一，我会怀疑你是不是真心愿意好好工作；第二，如果我能提供的工资远低于你的要求的话，你肯定只是利用我们公司作为一个临时立足点，好去寻找下一个机会，而且这个时间会很短。所以我们没有聘用他。

关于更多的谈判技巧，可以参考本书第6章第4节关于客户关系部分，很多原理都是可以借鉴的。

[illegible]

第2篇 C/C++面试题

第4章 C/C++程序基础

招聘方很重视求职者的项目经验，关心求职者做过什么类型的项目，担任何种角色，以及在项目中如何与他人沟通等。除此之外，也非常重视求职者的编程能力，包括其编程风格，对赋值语句、递增语句、类型转换、数据交换等程序设计基本概念的理解。因此，求职者最好在面试之前复习一下程序设计的基本概念，并且要特别重视那些比较细致的考点。本章将讲解C/C++程序设计基本概念的考题。

4.1 变量赋值

对于一个变量进行赋值操作有两种方式，一种是用“=”操作符；另一种是用“++”、“--”操作符。

4.1.1 一般赋值语句

想要访问内存时，就需要相应的地址以表明访问哪块内存，而变量是内存的映射，因此变量名就相当于一个地址。对于内存的操作，在一般情况下只有读取内存中的数值和将数值写入内存。在 C++中，为了将一数值写入某变量对应的地址所标识的内存中，只需先书写变量名，后接“=”，再接要写入的数字，例如 a=1。

面试例题 1：分析代码写输出——一般赋值语句。

考点：一般赋值语句的概念和方法。

出现频率：★★★

```
1    #include <stdio.h>
2
3    int main(void)
4    {
5        int x = 3, y, z;
6
7        x *= (y = z = 4); printf("x = %d\n", x);
8
```

```
9        z = 2;
10       x = (y = z); printf("x = %d\n", x);
11       x = (y == z); printf("x = %d\n", x);
12       x = (y & z); printf("x = %d\n", x);
13       x = (y && z); printf("x = %d\n", x);
14
15       y = 4;
16       x = (y | z); printf("x = %d\n", x);
17       x = (y || z); printf("x = %d\n", x);
18
19       x = (y == z)? 4: 5;
20       printf("x = %d\n", x);
21
22       x = (y == z)? 1: (y < z)? 2: 3;
23       printf("x = %d\n", x);
24
25       return 0;
26   }
```

解析

程序的说明如下。

- 程序执行至第 8 行时，x 的值为 3，y 和 z 未被初始化。此行的执行顺序是首先执行 z=4，然后 y=z，最后是 x*=y。因此 x 的值为 3*4=12。
- 程序执行至第 10 行时，z 的值为 2。此行的执行顺序是首先执行 y=z，然后 x=y。因此 x 的值为 2。
- 程序执行至第 11 行时，y 和 z 的值都为 2。此行的执行顺序是首先执行 y==z 比较 y 和 z 的值是否相等，然后将比较的结果赋给 x。因此 x 的值为 1。
- 程序执行至第 12 行时，y 和 z 的值都为 2。此行把 y 和 z 做按位与（&）运算的结果赋给变量 x。y 和 z 的二进制都是 10，y & z 的结果为二进制 10。因此 x 的值为 2。
- 程序执行至第 13 行时，y 和 z 的值都为 2。此行把 y 和 z 做逻辑与（&&）运算的结果赋给变量 x。此时 y 和 z 的值都不是 0，y && z 的结果为 1。因此 x 的值为 1。
- 程序执行至第 16 行时，y 的值为 4，z 的值为 2。此行把 y 和 z 做按位或（|）运算的结果赋给变量 x。此时 y 和 z 的二进制表示分别为 100 和 010，因此 y|z 的结果为 110。因此 x 的值为 110，十进制表示为 6。
- 程序执行至第 17 行时，y 的值为 4，z 的值为 2。此行把 y 和 z 做逻辑或（||）运算的结果赋给变量 x。此时 y 和 z 的值都不是 0，y||z 的结果为 1。因此 x 的值为 1。
- 程序执行至第 19 行时，y 的值为 4，z 的值为 2。此行首先比较 y 和 z 的大小是否相等，如果相等，则将 x 取 4；否则 x 取 5。在这里 y 不等于 z，因此 x 的值为 5。
- 程序执行至第 22 行时，y 的值为 4，z 的值为 2。此行首先比较 y 和 z 大小是否相等，如果相等，x 取 1；否则判断 y 是否大于 z，如果是则 x 取 2，否则 x 取 3。在这里 y

的值大于 z 的值，因此 x 的值为 3。

总结：这个考题只是考察各种基本的赋值运算。要注意位运算与逻辑运算的区别，以及三元操作符的用法。通过程序代码 17 行以及 19 行的举例，可以发现三元操作符有时可以代替条件判断 if/else/else if 的组合。

答案

x = 12

x = 2

x = 1

x = 2

x = 1

x = 6

x = 1

x = 5

x = 3

面试例题 2：分析代码写运行结果——C++域操作符。

考点：C++域操作符的使用。

出现频率：★★★

请指出下面这个程序在 C 和 C++中的输出分别是什么？

```
1   #include <stdio.h>
2
3       int value = 0;
4
5   void printvalue()
6   {
7       printf("value = %d\n", value);
8   };
9
10  int main()
11  {
12      int value = 0;
13
14      value = 1;
15          printf("value = %d\n", value);
16
17      ::value = 2;
18      printvalue();
19
20      return 0;
21  }
```

解析

如果将文件保存为后缀名为.C 的文件，在 Visual C++ 6.0 中不能通过编译并且提示 17 行有语法错误。而如果文件保存为后缀名为.cpp 的文件，在 Visual C++中就能顺利通过编译并且运行。

这段程序有两个变量，其名字都是 value。不同的是其中一个是在 main 函数之前就声明的全局变量，而另外一个是在 main 函数内部声明的局部变量。这两个变量的作用域是不一样的。

注意：在函数 printvalue 里打印的是全局变量的值，在 main 函数的 15 行打印的是局部变量的值。这是因为在 main 函数里的局部变量 value 引用优先。在 C++中可以通过域操作符“::”来直接操作全局变量（即代码的 17 行操作的 value 是全局变量），但是在 C 中不支持这个操作符，因此会报错。在 C 中不推荐这种局部变量与全局变量同名的设计方式。

答案

在 C 中编译不能通过，并指示 17 行符号错误。

在 C++中的输出如下。

value=1（局部变量 value）

value=2（全局变量 value）

4.1.2 i++与++i

i++与++i 都是自增运算。i++先调用（未增值的）i 再对 i 增值，而++i 先对 i 加 1 再调用增值后的 i。为了便于记忆，可以把这个规则记成“++在前先加再用，++在后先用再加”。这个规则在 i--与--i 上同样适用。

面试例题 3：分析代码写输出——i++和++i 的区别。

考点：i++和++i 的区别。

出现频率：★★★★★

```
1    #include <stdio.h>
2
3    int main(void)
4    {
5        int i=8;
6
7        printf("%d\n",++i);
8        printf("%d\n",--i);
9        printf("%d\n",i++);
10       printf("%d\n",i--);
11       printf("%d\n",-i++);
12           printf("%d\n",-i--);
13       printf("------\n");
14
15       return 0;
16   }
```

解析

程序的说明如下。

- 程序第 7 行，此时 i 的值为 8。这里先 i 自增 1 再打印 i 的值。因此输出 9，并且 i 的值也变为 9。
- 程序第 8 行，此时 i 的值为 9。这里先 i 自减 1 再打印 i 的值。因此输出 8，并且 i 的值也变为 8。
- 程序第 9 行，此时 i 的值为 8。这里先打印 i 的值再 i 自增 1。因此输出 8，并且 i 的值变为 9。
- 程序第 10 行，此时 i 的值为 9。这里先打印 i 的值再 i 自减 1。因此输出 9，并且 i 的值变为 8。
- 程序第 11 行，此时 i 的值为 8。这里的“-”表示负号运算符。因此先打印-i 的值再 i 自增 1。因此输出-8，并且 i 的值变为 9。
- 程序第 12 行，此时 i 的值为 9。这里的第一个“-”表示负号运算符，后面的“--”表示自减运算符。因此先打印-i 的值再 i 自减 1。因此输出-9，并且 i 的值也变为 8。

答案

```
9
8
8
9
-8
-9
------
```

面试例题 4：i++与++i 哪个效率更高？

考点：i++和++i 的效率比较。

出现频率：★★★

解析

简单的比较前缀自增运算符和后缀自增运算符的效率是片面的，因为存在很多因素影响这个问题的答案。首先考虑内建数据类型的情况：如果自增运算表达式的结果没有被使用，而是仅仅简单地用于增加一元操作数，答案是明确的，前缀法和后缀法没有任何区别。编译器的处理都应该是相同的，很难想象得出有什么编译器实现可以别出心裁地在二者之间制造任何差异。示例程序如下。

```
1    #include <stdio.h>
2
3    int main()
4    {
```

```
5        int i = 0;
6        int x = 0;
7
8        i++;
9        ++i;
10       x = i++;
11       x = ++i;
12
13         return 0;
14     }
```

上面的代码在 Visual C++ 6.0 上编译得到的汇编如下：

```
; Line 5
    mov     DWORD PTR _i$[ebp], 0
; Line 6
    mov     DWORD PTR _x$[ebp], 0
; Line 8
    mov     eax, DWORD PTR _i$[ebp]
    add     eax, 1
    mov     DWORD PTR _i$[ebp], eax
; Line 9
    mov     ecx, DWORD PTR _i$[ebp]
    add     ecx, 1
    mov     DWORD PTR _i$[ebp], ecx
; Line 10
    mov     edx, DWORD PTR _i$[ebp]
    mov     DWORD PTR _x$[ebp], edx
    mov     eax, DWORD PTR _i$[ebp]
    add     eax, 1
    mov     DWORD PTR _i$[ebp], eax
; Line 11
    mov     ecx, DWORD PTR _i$[ebp]
    add     ecx, 1
    mov     DWORD PTR _i$[ebp], ecx
    mov     edx, DWORD PTR _i$[ebp]
    mov     DWORD PTR _x$[ebp], edx
```

- 代码段第 8 行和第 9 行生成的汇编代码分别对应 Line 8 和 Line 9 下面的汇编代码，可以看到 3 个步骤几乎完全一样。
- 代码段第 10 行和第 11 行生成的汇编代码分别对应 Line 10 和 Line 11 下面的汇编代码，可以看到都是 5 个步骤，只是在加 1 的先后顺序上有一些区别，效率也是完全一样的。

由此说明，考虑内建数据类型时，它们的效率差别不大（去除编译器优化的影响）。再考虑自定义数据类型（主要是指类）的情况。此时不需要再做很多汇编代码的分析，因为前缀式（++i）可以返回对象的引用，而后缀式（i++）必须返回对象的值，所以导致在大对象的时候产生了较大的复制开销，引起效率降低，因此使用自定义类型（注意不是指内建类型）的时候，应该尽

可能地使用前缀式递增或递减。

答案

在内建数据类型的情况下，效率没有区别。

在自定义数据类型的情况下，++i 效率较高。

4.2 编程规范

编程规范是指程序员编程时遵守的一种规范，它包括很多方面，如下所示。

- 程序的版式。版式虽然不会影响程序的功能，但会影响可读性。程序的版式追求清晰、美观。版式设计包括空行和空格的添加，代码行的对齐等方面。
- 表达式。合理地书写表达式能够让程序具有更好的可读性，减少错误的发生。
- 命名规则。顾名思义，也就是给变量、结构体或类取名，这样从名字中能推断出它们在程序中的用途。

每个公司都有一套自己的编程规范，但实际上都大同小异，都是为了让代码看着更加美观，程序更容易维护。

4.2.1 条件比较

条件比较是编程规范中经常要考察的地方，拥有良好编程规范的程序代码，可以提高程序的可读性。

面试例题 5：选择编程风格良好的条件比较语句。

考点：良好的编程风格。

出现频率：★★★★

（1）假设布尔变量名字为 flag，它与零值比较的标准 if 语句如下。

第 1 种：

```
1    if (flag == TRUE)
2    if (flag == FALSE)
```

第 2 种：

```
1    if (flag)
2    if (!flag)
```

（2）假设整型变量的名字为 value，它与零值比较的标准 if 语句如下。

第 1 种：

```
1    if (value == 0)
2    if (value != 0)
```

第 2 种：

```
1    if (value)
2    if (value)
```

（3）假设浮点变量的名字为 x，它与 0.0 的比较如下。

第 1 种：

```
1    if (x == 0.0)
2    if (x != 0.0)
```

第 2 种：

```
1    if ((x >= -EPSINON) && (X <= EPSINON))
2    if ((x < -EPSINON) || (X > EPSINON))
```

其中 EPSINON 是允许的误差（即精度）。

（4）指针变量 p 与 0 的比较如下。

第 1 种：

```
1  if (p == NULL)
2  if (p != NULL)
```

第 2 种：

```
1    if (p == 0)
2    if (p != 0)
```

解析

- （1）的第 2 种风格较好。根据布尔类型的语义，零值为“假”（记为 False），任何非零值都是“真”（记为 True）。True 的值究竟是什么并没有统一的标准。例如 Visual C++ 将 true 定义为 1，而 Visual Basic 则将 true 定义为-1。因此不可将布尔变量直接与 true、false 进行比较。
- （2）的第 1 种风格较好。第 2 种风格会让人误解 value 是布尔变量，应该将整型变量用“==”或“! =”直接与 0 比较。
- （3）的第 2 种风格较好。注意，无论是 float 还是 double 类型的变量，都有精度限制，所以一定要避免将浮点变量用“==”或“! =”与数字比较，应该设法转化成“>=”或“<=”的形式。
- （4）的第 1 种风格较好。指针变量的零值是“空”（记为 NULL）。尽管 NULL 的值与 0 相同，但是两者意义不同。用 p 与 NULL 显式比较，强调 p 是指针变量。如果用 p 与 0 比较，容易让人误解 p 是整型变量。

4.2.2 命名规则

命名规则是为了方便不同的开发者从名字推断变量、结构体或类在程序中的用途。比较著名的命名规则当推 Microsoft 公司的“匈牙利”法，该命名规则的主要思想是“在变量和函数名中加入前缀以增进人们对程序的理解”。“匈牙利”法最大的缺点就是烦琐。实际上很多公司在各自的编程规范里详细制定了命名规则。

由于命名规则牵涉到很多具体细节，在本节就不一一列举。读者可以参考 Microsoft 公司的

“匈牙利”法的实施细节。

4.3 类型转换

类型转换是将一种类型的值映射为另一种类型的值。类型转换包含自动隐含和强制的两种。C++语言编译系统提供的内部数据类型的自动隐式转换规则如下。

- 程序在执行算术运算时，低类型自动隐式转换为高类型。
  在赋值表达式中，右边表达式的值自动隐式转换为左边变量的类型，并赋值给左边变量。
- 在函数调用时，将实参值赋给形参，系统隐式地将实参转换为形参的类型，并赋值给形参。
- 函数有返回值时，系统自动地将返回表达式类型转换为函数类型，并赋值给调用函数。

当在程序中发现两个数据类型不相容时，又不能自动完成隐式转换，则将出现编译错误。例如：

```
int * p = 100;
```

在这种情况下，编译程序将报错，为了消除错误，可以进行如下所示的强制类型转换。

```
int * p = (int *)100;
```

将整型数 100 显式地转换成指针类型。

面试例题 6：分析代码写结果——有符号变量与无符号变量值的转换。

考点：有符号变量与无符号变量的区别和联系。

出现频率：★★★★

```
1   #include <stdio.h>
2
3   char getChar(int x, int y)
4   {
5       char c;
6       unsigned int a = x;
7
8       (a + y > 10)? (c = 1): (c = 2);
9       return c;
10  }
11
12  int main(void)
13  {
14      char c1 = getChar(7, 4);
15      char c2 = getChar(7, 3);
16      char c3 = getChar(7, -7);
```

```
17        char c4 = getChar(7, -8);
18
19        printf("c1 = %d\n", c1);
20        printf("c2 = %d\n", c2);
21        printf("c3 = %d\n", c3);
22        printf("c4 = %d\n", c4);
23
24        return 0;
25    }
```

解析

首先说明一下 getChar()函数的作用。getChar()有两个输入参数，分别是整型的 x 和 y。在函数体内，把参数 x 的值转换为无符号整型后再与 y 相加，其结果与 10 进行比较，如果大于 10 则函数返回 1，否则返回 2。

注意：当表达式中存在有符号类型和无符号类型时，所有的操作数都自动转换成无符号类型。

由于 a 是无符号数，代码第 8 行中 y 值会首先自动转换成无符号的整数然后再与 a 相加，最后再与 10 进行比较。以下是在 main 函数中各调用 getChar()函数的分析。

- 代码第 14 行，传入的参数分别为 7 和 4，两个数相加后为 11，因此 c1 返回 1。
- 代码第 15 行，传入的参数分别为 7 和 3，两个数相加后为 10，因此 c1 返回 2。
- 代码第 16 行，传入的参数分别为 7 和-7，-7 首先被转换成一个很大的数，然后与 7 相加后正好溢出，其值为 0，因此 c1 返回 2。
- 代码第 17 行，传入的参数分别为 7 和-8，-8 首先被转换成一个很大的数，然后与 7 相加。两个数相加后为很大的整数（差 1 就溢出了），因此 c1 返回 1。

可以看出，由于无符号整数的特性，当参数 x 为 7 时，如果 y 等于区间[-7,3]中的任何整数值，getChar()函数都将返回 2。当 y 的值在区间[-7,3]之外时，函数返回 1。

总之，使用表达式时要注意符号变量与无符号变量之间的转换，占用不同字节内存的变量之间的赋值等操作，否则可能会出现意想不到的运行结果。

答案

```
c1 = 1
c2 = 2
c3 = 2
c4 = 1
```

4.4 数值交换

如果一个函数要修改传入参数的值，直接传值是不行的。因为这样只是在函数的栈中复制

了参数的值，并没有修改实际的参数值。在C中可以通过将参数声明为指针类型达到目的。而在C++中，又多了一种选择，就是可以将参数声明为引用类型，实际上引用就是某种意义上的安全指针。

面试例题7：将数a、b的值进行交换，并且不使用任何中间变量。

考点：两个变量值的交换方法。

出现频率：★★★★

```
1   #include <stdio.h>
2
3   void swap1(int& a, int& b)
4   {
5       int temp = a;  //使用局部变量 temp 完成交换
6           a = b;
7       b = temp;
8   };
9
10  void swap2(int& a, int& b)
11  {
12      a=a+b;  //使用加减运算完成交换
13      b=a-b;
14      a=a-b;
15  };
16
17  void swap3(int& a, int& b)
18  {
19      a^=b;  //使用异或运算完成交换
20      b^=a;
21      a^=b;
22  };
23
24  int main(void)
25  {
26      int a1 = 1, b1 = 2;
27      int a2 = 3, b2 = 4;
28      int a3 = 5, b3 = 6;
29      int a = 2147483647, b = 1;
30
31      swap1(a1, b1);  //测试使用临时变量进行交换的版本
32      swap2(a2, b2);  //测试使用加减运算进行交换的版本
33      swap3(a3, b3);  //测试使用异或运算进行交换的版本
34
35      printf("after swap...\n");
36      printf("a1 = %d, b1 = %d\n", a1, b1);
37      printf("a2 = %d, b2 = %d\n", a2, b2);
```

```
38        printf("a3 = %d, b3 = %d\n", a3, b3);
39
40        swap2(a, b);
41        printf("a = %d, b = %d\n", a, b);
42
43        return 0;
44    }
```

解析:

以上的 C++程序中有 3 个 swap 函数，都是采用引用传参的方式。

- swap1()采用的是教科书里常见的方式，用一个局部变量 temp 保存其中一个值来达到交换目的。当然，这种方式不是本题要求的答案。
- swap2()采用的是一种简单的加减算法来达到交换 a、b 的目的。这种方式的缺点是做“a+b”和“a-b”运算时可能会导致数据溢出。
- swap3()采用了按位异或的方式交换 a、b。按位异或运算符“^”的功能是将参与运算的两数各对应的二进制位相异或，如果对应的二进制位相同，则结果为 0，否则结果为 1。这样运算 3 次即可交换 a、b 的值。

代码第 31 行～第 32 行调用 3 种 swap 函数。注意 40 行的调用，在 swap2 函数栈中的运算会有数据溢出发生。我们知道在 32 位平台下 int 占 4 个字节内存，其范围是-2 147 483 648～2 147 483 647，因此 2 147 483 647 加 1 就变成了-2 147 483 648。不过通过运行结果我们可以看到，虽然产生了溢出，但是交换操作依然是成功的。

答案

```
after swap...
a1 = 2, b1 = 1
a1 = 4, b1 = 3
a1 = 6, b1 = 5
a1 = 1, b1 = 2 147 483 647
```

注意：*采用程序代码中 swap2 和 swap3 的交换方式，swap2 有可能发生数据溢出的缺点。相比较 swap2，推荐 swap3 采用的按位异或的方式。*

4.5 C 和 C++的联系与区别

C++从 C 基础上发展而来，且大大扩充了 C 的内容和功能，提供了更多更全面的支持。只会 C 语言还远远不能说会了 C++。对于 C 语言程序员，完全可以把 C++作为一门新语言来学。

面试例题 8：C++与 C 有什么不同？

考点：C 和 C++的联系与区别。

出现频率：★★★★

解析

C是一个结构化语言，它的重点在于算法和数据结构。对语言本身而言，C是C++的子集。

C程序的设计首要考虑的是如何通过一个过程，对输入进行运算处理得到输出。

对于C++，首要考虑的是如何构造一个对象模型，让这个模型能够配合对应的问题，这样就可以通过获取对象的状态信息得到输出或实现过程控制。

因此C与C++的最大区别在于它们解决问题的思想方法不一样。

C实现了C++中过程化控制及其他相关功能。而在C++中的C，相对于原来的C还有所加强，引入了重载、内联函数、异常处理等，C++拓展了面向对象设计的内容，如类、继承、虚函数、模板和包容器类等。

在C++中，不仅需要考虑数据封装，还需要考虑对象粒度的选择、对象接口的设计和继承、组合与继承的使用等问题。

相对于C，C++包含了更丰富的设计概念。

面试例题9：C++是面向对象化的而C是面向过程化的？

考点：C++与C的区别。

出现频率：★★★

解析

C是面向过程化的，但是C++不是完全面向对象化的。在C++中也完全可以写出与C一样过程化的程序，所以只能说C++拥有面向对象的特性。

面试例题10：为什么标准头文件都有类似以下所示的结构？

考点：标准头文件中一些通用结构的理解。

出现频率：★★★★

```
1    #ifndef __INCvxWorksh
2    #define __INCvxWorksh
3    #ifdef __cplusplus
4    extern "C" {
5    #endif
6    /*...*/
7    #ifdef __cplusplus
8    }
9    #endif
10   #endif /* __INCvxWorksh */
```

解析

显而易见，代码1、2、10行的作用是防止该头文件被重复引用。代码第3行的作用是表示当前使用的是C++编译器。如果要表示当前使用的是C编译器，可以这样指定：

```
#ifdef __STDC__
```

那么代码第 4 行～第 8 行中的“extern "C"”有什么作用呢？

“extern "C"”包含双重含义。

（1）被它修饰的目标是“extern”的。也就是告诉编译器，其声明的函数和变量可以在本模块或其他模块中使用。通常，在模块的头文件中对本模块供给其他模块引用的函数和全局变量以关键字 extern 声明。例如，如果模块 B 打算引用该模块 A 中定义的全局变量和函数，只需包含模块 A 的头文件即可。这样，模块 B 调用模块 A 的函数时，在编译阶段，模块 B 即便找不到该函数，编译器也不会报错，它会在连接阶段中从模块 A 编译生成的目标代码中找到此函数。

（2）被它修饰的目标是“C”的。意思是其修饰的变量和函数是按照 C 语言方式编译和连接的。首先必须弄清 C++中对类似 C 的函数是怎样编译的。作为一种面向对象的语言，C++支持函数重载，而过程式语言 C 则不支持。函数被 C++编译后在符号库中的名字与 C 语言的不同。例如下面两个函数：

```
void foo( int x, int y );
void foo( int x, float y );
```

这两个函数编译生成的符号是不相同的，前者可能为_foo_int_int 类，而后者可能为_foo_int_float 类。可以发现，这样的名字包含了函数名、函数参数数量及类型信息，C++就是靠这种机制来实现函数重载的。这样，如果在 C 中连接 C++编译的符号，就会因找不到符号问题发生链接错误。

如果加“extern"C"”声明，模块编译生成 foo 的目标代码时，就不会对其名字进行特殊处理，而且采用 C 语言的方式，生成形如_foo 的类，不会加上函数参数数量及类型信息相关的一串字符。因此“extern "C"”是 C++编译器提供的与 C 连接交换指定的符号，用来解决名字匹配问题。

答案

代码第 1、2、10 行的作用是防止该头文件被重复引用。

代码第 13 行的作用是表示当前使用的是 C++编译器。

代码第 4 行～第 8 行中的 extern "C"是 C++编译器提供的与 C 连接交换指定的符号，用来解决名字匹配问题。

面试例题 11：#include <head.h>和#include "head.h" 有什么区别？

考点：头文件引用中〈〉与" "的区别。

出现频率：★★★★

解析

〈〉表明括号中的文件是一个工程或标准头文件。查找过程会首先检查预定义的目录，开发者可以通过设置搜索路径环境变量或命令行选项来修改这些目录。

如果文件名用一对引号括起来则表明该文件是用户提供的头文件，查找该文件时将从当前

文件目录或文件名指定的其他目录中寻找，然后再在标准位置寻找。

4.6 main函数之后的调用

main函数代表进程的主线程。程序开始执行时，系统为程序创建一个进程，main函数其实并不是首先被调用的函数，而是操作系统调用了C/C++运行期启动函数，该函数负责对C/C++运行期库进行初始化。它还能够保证已经声明了的任何全局对象和静态对象能够在代码执行之前正确的创建。

完成这些工作后，就调用进入点函数（控制台程序为main函数），并在main函数里面执行一系列操作。

在main执行完毕后，从main函数返回，启动函数调用C运行期的exit()函数，将返回值传递给它。其中在exit()会调用ExitProcess()函数，结束进程。

面试例题12：C++中main函数执行完后还执行其他语句吗？

考点：atexit()函数的使用。

出现频率：★★★★

解析

在程序退出的时候经常需要做一些诸如释放资源的操作，但程序退出的方式有很多种，例如main()函数运行结束、在程序的某个地方用exit()结束程序、用户通过快捷键Ctrl+C等操作发信号来终止程序等，因此需要一种与程序退出方式无关的方法来进行程序退出时的必要处理。atexit()函数用来注册程序正常终止时要被调用的函数。

atexit()函数的参数是一个函数指针，函数指针指向一个没有参数也没有返回值的函数。atexit()的函数原型如下：

```
int atexit (void (*)(void));
```

在一个程序中最多可以用atexit()注册32个处理函数，这些处理函数的调用顺序与其注册的顺序相反，即最先注册的最后调用，最后注册的最先调用。请看下面的程序代码。

```
1    #include<stdlib.h>     //使用 atexit()函数必须包含头文件 stdlib.h
2    #include<stdio.h>
3
4    void fn1(void);
5    void fn2(void);
6
7    int main(void)
8    {
9        atexit(fn1);              //使用 atexit 注册 fn1()函数
10       atexit(fn2);              //使用 atexit 注册 fn2()函数
11       printf("main exit...\n");
12       return 0;
```

```
13    }
14
15    void fn1()
16    {
17        printf("calling fn1()...\n");    //fn1()函数打印内容
18    }
19
20    void fn2()
21    {
22        printf("calling fn2()...\n");    //fn2()函数打印内容
23    }
```

上面的程序代码在 main 函数调用 atexit()函数时依次注册了 fn1()和 fn2()函数。运行这个程序可以得到下面输出：

```
main exit…
calling fn1()…
calling fn2()…
```

在这里 fn2()与 fn1()在 main()函数结束后被依次调用，并且它们被调用的顺序与它们在 main()函数被注册的顺序相反。

答案

可以用 atexit()函数来注册程序正常终止时要被调用的函数，并且在 main()函数结束时调用这些函数的顺序与注册它们的顺序相反。

第5章 预处理、const、static与sizeof

本章要讨论的问题是 C/C++设计语言中的难点，也是企业考查的重点，尤其是 static 和 sizeof，在许多公司的面试题中反复出现。

5.1 预处理

预处理是指通过预处理的内建功能对一个资源进行等价替换，最常见的预处理有：文件包含（#include）、条件编译（#ifdef、#else 等）、布局控制（#progma）以及宏替换（#define）。

5.1.1 #ifdef、#else、#endif 指示符

#ifdef 指示符常被用来判断一个预处理器常量是否已被定义，以便有条件地包含程序代码。#ifndef 与#ifdef 作用相同，只是判断方式相反。

面试例题 1：分析代码写结果——预处理的使用。

考点：#ifdef、#else、#endif 在程序中的使用。

出现频率：★★★

```
1    #include <stdio.h>
2    #include <stdlib.h>
3
4    #define DEBUG
5
6    int main()
7    {
8        int i = 0;
9        char c;
10
11       while(1)
```

```
12          { i++;
13          c = getchar();
14          if (c != '\n')
15              { getchar();
16          }
17          if (c == 'q' || c == 'Q')
18  { #ifdef DEBUG
19                  printf("we got:%c, about to exit.\n", c);
20  #endif
21              break;
22          } else
23              { printf("i = %d", i);
24  #ifdef DEBUG
25              printf(", we got:%c", c);
26  #endif
27              printf("\n");
28            }
29            }
30
31      return 0;
32  }
```

解析

在代码第 4 行，首先定义了名为 DEBUG 的预处理器常量，然后分别在第 18 行和第 24 行用#ifdef 判断 DEBUG 是否被定义了，如果被定义了，就进行 printf 输出信息。传给 main 的代码如下：

```
6   int main()
7   {
8       int i = 0;
9       char c;
10
11      while(1)
12          { i++;
13          c = getchar();
14          if (c != '\n')
15              { getchar();
16          }
17          if (c == 'q' || c == 'Q')
19              { printf("we got:%c, about to exit.\n", c);
21              break;
22          } else
23              { printf("i = %d", i);
25              printf(", we got:%c", c);
27              printf("\n");
28          }
29      }
```

```
30
31          return 0;
32      }
```

根据上面展开之后的代码段，容易看出这个程序就是从终端每次读一个字符，如果字符为 q 或者 Q，程序退出，否则打印收到的字符。执行结果如下。

```
输入：A
输出：i = 1, we got:A
输入：a
输出：i = 2, we got:a
输入：q
输出：we got:q, about to exit.
```

以上输出含有“we got”字符串，均为#ifdef 与#endif 之间的打印消息。

如果注释代码 4 行，即 DEBUG 没有被定义，那么#ifdef 与#endif 之间的打印消息将不会被输出。

5.1.2 宏定义

良好的宏定义对于写好 C 语言十分重要，使用宏定义可以防止出错，提高代码可移植性和可读性等。因此面试过程中，经常会通过考查宏定义来了解各个求职者 C 语言的基本功。

面试例题 2：用#define 实现宏，并求最大值和最小值。

考点：#define 宏定义使用。

出现频率：★★★★

解析

以下为实现代码：

```
1    #define   MAX( x, y ) ( ((x) > (y)) ? (x) : (y) )
2    #define   MIN( x, y ) ( ((x) < (y)) ? (x) : (y) )
```

在这里 MAX(x,y)表示取 x 和 y 的最大值，MIN(x,y)表示取 x 和 y 的最小值。此题有以下 3 个目的。

- 考查#define 在宏上应用的基本知识。
- 考查三重条件操作符的应用。这个操作符能产生比 if-else 更优化的代码，并且书写上更加简洁明了。
- 考查宏定义的写法。在宏中需要把参数小心地用括号括起来。因为宏只是简单的文本替换，如果不注意很容易引起歧义。

面试例题 3：分析代码写结果——宏定义的使用。

考点：#define 宏定义使用时需要注意的地方。

出现频率：★★★★

```
1    #include <stdio.h>
2    #define SQR(x) (x*x)
3
4    int main()
5    {
6        int a, b = 3;
7        a = SQR(b+2);
8        printf("a = %d\n", a);
9        return 0;
10   }
```

解析

这里定义的 SQR(x)函数显然是想要获得 x 的二次乘方，在第 7 行中调用的参数为 b+2，原本想将 a 赋为(b+2)*(b+2)也就是 5 的二次方，即 25。但是由于宏定义展开是在预处理时期，也就是在编译之前。此时 b 并没有被赋值，这时的 b 只是一个符号。因此在第 7 行被展开成：

```
a = (b+2*b+2);
```

于是程序执行后，可以看到 a 被赋成 11。

为了达到原来的目的，可以将 SQR(x)改成如下定义：

```
#define SQR(x) ((x)*(x))
```

这样在第 7 行会被展开成：

```
a = ((b+2)*(b+2));
```

程序执行后，a 被赋成 25。

答案

a = 11

面试例题 4：分析代码写结果——宏参数的连接。

考点：如何连接宏参数。

出现频率：★★★

```
1    #include <stdio.h>
2
3    #define STR(s)      #s
4    #define CONS(a,b)   (int)(a##e##b)
5
6    int main()
7    {
8        printf(STR(vck));
9        printf("\n");
10       printf("%d\n", CONS(2,3));
11
12       return 0;
13   }
```

解析

在本程序中，使用#把宏参数变为一个字符串，用##把两个宏参数贴合在一起。

- 代码第 3 行 STR(s)定义的是一个参数 s 表示的字符串，在第 8 行的调用中，STR(vck)实际表示的就是字符串“vck”。
- 代码第 4 行 CONS(a,b)定义的是一个将参数 a 与 b 按 aeb 连接起来的一个整型值，在第 10 行的调用中，CONS(2,3)实际表示就是整型值 2e3，也就是十进制数 2000。

答案

vck

2000

面试例题 5：用宏定义得到一个字的高位和低位字节。

考点：宏定义与位运算的使用。

出现频率：★★★★

解析

程序代码如下所示：

```
1    #define   WORD_LO(xxx)   ((byte) ((word)(xxx) & 255))
2    #define   WORD_HI(xxx)   ((byte) ((word)(xxx) >> 8))
```

一个字由 2 字节组成。因此 WORD_LO(xxx)取参数 xxx 的低 8 位，WORD_HI(xxx)取参数 xxx 的高 8 位。

答案

```
#define   WORD_LO(xxx)   ((byte) ((word)(xxx) & 255))
#define   WORD_HI(xxx)   ((byte) ((word)(xxx) >> 8))
```

面试例题 6：用宏定义得到一个数组所含的元素个数。

考点：宏定义与 sizeof 的使用。

出现频率：★★★★

解析

代码如下：

```
#define   ARR_SIZE(a)   (sizeof((a)) / sizeof((a[0])))
```

假设一个数组定义如下：

```
int array[100];
```

它含有 100 个 int 型的元素。如果 int 为 4 字节，那么这个数组总共有 400 字节。sizeof(array)为总大小，即 400 字节，sizeof(array[0])为一个 int 大小，即 4 字节。两个相除就是 100，也就是数组的元素个数。这里为了保证宏定义不会发生“二义性”，在 a 以及 a[0]上都加了括号。

答案

```
#define   ARR_SIZE(a)   (sizeof((a)) / sizeof((a[0])))
```

5.2 const（常量）

const 是 C 语言的一个关键字，它所限定变量不允许被改变。使用 const 可以在一定程度上增强程序的健壮型，减少程序出错。虽然 const 有诸多优势，但是 const 的使用却并不简单。

面试例题 7：找错——const 的使用。

考点：const 使用时的注意点。

出现频率：★★★★

```
1    #include <stdio.h>
2
3    int main()
4    {
5        const int x = 1;
6        int b = 10;
7        int c = 20;
8
9        const int* a1 = &b;
10       int* const a2 = &b;
11       const int* const a3 = &b;
12
13       x = 2;
14
15       a1 = &c;
16       *a1 = 1;
17
18       a2 = &c;
19       *a2 = 1;
20
21       a3 = &c;
22       *a3 = 1;
23
24       return 0;
25   }
```

解析

程序代码说明如下。

- 代码第 13 行，由于变量 x 为整型常量，因此不能改变 x 的值。在这里会出现编译错误，并提示“l-value specifies const object”，即等号左边的是常量对象。如果在代码第 5 行没有给 x 初始化，那么 x 就是一个随机数，并且以后也不能给它赋值。
- 代码第 15 行和代码第 16 行，a1 定义为 const int*类型，注意这里的 const 在 int*的左

侧，它是用修饰指针所指向的变量，即指针指向为常量。代码 15 行中把 a1 指向变量 c 是允许的，因为这修改的是指针 a1 本身，而不是指针 al 所指向的内容。但是代码 16 行改变 a1 指向的内容，是不允许的。编译器给出的错误同样是“l-value specifies const object”。

- 代码第 18 行和第 19 行，a2 定义为 int* const 类型，注意这里的 const 位于 int*的右侧，它修饰指针本身，即指针本身为常量。因此代码第 18 行中修改指针 a2 是不允许的，而代码第 19 行修改 a2 指向的内容是允许的。
- 代码第 21 行和代码第 22 行，a3 定义为 const int* const 类型，这里有两个 const，分别出现在 int*的左右两侧，因此它表示不仅指针本身不能修改，并且其指向的内容也不能修改。所以代码第 21 行和代码第 22 行都会出现编译错误。

注意：变量 x、a2 和 a3 在声明的同时一定要初始化，因为它们在后面都不能被赋值。变量 a1 可以在声明的时候不初始化。

答案

代码第 13、16、18、21 和 22 行会出现编译错误。

面试例题 8：请说明 const 与#define 的各自特点及区别。

考点：对 const 与#define 的特点及区别的理解。

出现频率：★★★

解析

#define 只是用来做文本替换的，例如：

```
1    #define PI 3.141 59 26
2    float angel;
3    angel = 30 * PI / 180;
```

那么，当程序进行编译的时候，编译器会首先将“#define PI 3.1 415 926”以后所有代码中的“PI”全部换成“3.1 415 926”，然后再进行编译。因此#define 常量的生命周期止于编译期，它存在于程序的代码段，在实际程序中它只是一个常数，一个命令中的参数，并没有实际的存在。

const 常量存在于程序的数据段，并在堆栈中分配了空间。const 常量是一个 Run-Time 的概念，它在程序中确确实实地存在着并可以被调用、传递。const 常量有数据类型，而宏常量没有数据类型。编译器可以对 const 常量进行类型安全检查。

面试例题 9：C++中 const 有什么作用？至少说明 3 种。

考点：对 C++中 const 作用的理解。

出现频率：★★★★★

解析

const 的作用说明如下。

- const 用于定义常量：const 定义的常量编译器可以对其进行数据静态类型安全检查。
- const 修饰函数形式参数：当输入参数为用户自定义类型和抽象数据类型时，将“值传递”改为“const &传递”可以提高效率。比较下面两段代码：

```
void fun(A a);
void fun(A const &a);
```

第 1 个函数效率低。函数体内产生 A 类型的临时对象用于“值传递”参数 a，临时对象的构造、复制、析构过程都将消耗时间。而第 2 个函数提高了效率。用“引用传递”不需要产生临时对象，省了临时对象的构造、复制、析构过程消耗的时间。但只用引用有可能改变 a，所以加 const。

- const 修饰函数的返回值：如果给“指针传递”的函数返回值加 const，则返回值不能被直接修改，且该返回值只能被赋值给 const 修饰的同类型指针。例如：

```
const char *GetChar(void){};
char *ch = GetChar(); // error
const char *ch = GetChar(); // correct
```

- const 修饰类的成员函数(函数定义体)：任何不需要修改数据成员的函数都应该使用 const 修饰，这样即使不小心修改了数据成员或调用了非 const 成员函数，编译器也会报错。const 修饰类的成员函数形式为：

```
    int GetCount(void)  const;
```

5.3 static 变量（静态变量）

static 关键字代表静态，它可以作用于变量以及函数。例如，在局部变量前加上 static 关键字后，就定义了静态局部变量，在函数的返回类型前加上 static 关键字后，就定义了静态函数。在 C++的面向对象编程中，static 还可以加在类的数据成员或成员函数之前，这样定义的数据成员或成员函数就被类所拥有，而不再属于类的对象。

面试例题 10：static 有什么作用？至少说明两种。

考点：对 C++中 static 作用的理解。

出现频率：★★★★★

解析

在 C++语言中，关键字 static 有 3 个明显的作用。

- 在函数体，一个被声明为静态的变量在函数被调用的过程中维持其值不变。
- 在模块内（但在函数体外），一个被声明为静态的变量可以被模块内所有函数访问，但不能被模块外其他函数访问。它是一个本地的全局变量。
- 在模块内，被声明为静态的函数只能被这一模块内的其他函数调用。即函数被限制在声明它的模块范围内。

大多数求职者能正确回答第一部分，而另一部分求职者能正确回答第2部分，只有很少的人能回答第3部分。作为一个合格的软件工程师，要理解本地化数据和代码的好处和重要性。

面试例题 11：static 全局变量与普通全局变量有什么区别？static 局部变量和普通局部变量有什么区别？static 函数与普通函数有什么区别？

考点：对 C++中 static 的作用的理解。

出现频率：★★★★

解析

全局变量的声明之前加上 static 就构成了静态的全局变量。全局变量本身就是静态存储变量，静态全局变量当然也是静态存储变量。这两者在存储方式上并无不同。这两者的区别在于非静态全局变量的作用域是整个源程序，当源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的，而静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其他源文件中不能使用它。由于静态全局变量的作用域局限于一个源文件内，只能为该源文件内的函数公用，因此可以避免在其他源文件中引起错误。

从以上分析可以看出，把局部变量改变为静态变量后是改变了它的存储方式，即改变了它的生存期。把全局变量改变为静态变量后是改变了它的作用域，即限制了它的使用范围。

static 函数与普通函数作用域不同，其作用域仅在本文件。只在当前源文件中使用的函数应该说明为内部函数(static)，内部函数应该在当前源文件中说明和定义。对于可在当前源文件以外使用的函数，应该在一个头文件中说明，要使用这些函数的源文件要包含这个头文件。

答案

static 全局变量与普通全局变量的区别是：static 全局变量只初始化一次，防止在其他文件单元中被引用。

static 局部变量和普通局部变量的区别是：static 局部变量只被初始化一次，下一次依据上一次结果值。

static 函数与普通函数的区别是：static 函数在内存中只有一份，普通函数在每个被调用中维持一份复制品。

面试例题 12：分析代码写结果——C++类的静态成员。

考点：对 C++中类的静态成员的理解。

出现频率：★★★★★

```
1   #include <iostream.h>
2   class widget
3   {
4   public:
```

```
5          widget()
6          {
7              count++;
8          }
9          ~ widget()
10         {
11             --count;
12         }
13         static int num()
14         {
15             return count;
16         }
17     private:
18         static int count;
19     };
20
21     int widget::coun t = 0;
22
23     int main()
24     {
25         widget x,y;
26         cout << "The Num.is" << widget::num() << endl;
27         if(widget::num()>1)
28         {
29             widget x, y, z;
30             cout << "The Num.is" << widget::num() << endl;
31         }
32         widget z;
33         cout << "The Num.is" << widget::num() << endl;
34
35         return 0;
36     }
```

解析

类 widget 有一个静态成员 count 和一个静态方法 num()。类中的静态成员或方法不属于类的实例，而属于类本身并在所有类的实例间共享。在调用它们时应该用类名加上操作赋“::”来引用。

在代码第 7 行类 widget 的构造方法里把静态成员 count 的值加 1，在代码第 11 行类 widget 的析构方法里把静态成员 count 的值减 1。也就是说静态成员 count 的值表示类 widget 实例的个数。

通过以上的分析，可以看到运行到代码第 26 行时，只有两个类 widget 的实例，运行到代码第 30 行时，又产生了 3 个实例，这 3 个实例在第 31 行结束后被销毁。最后运行到代码第 32 行又产生了 1 个实例。

答案

The Num.is2

The Num.is5
The Num.is3

5.4 sizeof 操作符

sizeof 是一种单目操作符，而不是函数。sizeof 操作符以字节形式给出了其操作数的存储空间。操作数可以是一个表达式或括在括号内的类型名。操作数的存储空间由操作数的类型决定。

面试例题 13：填空题——使用 sizeof 计算普通变量所占空间。

考点：sizeof 计算普通变量所占空间大小。

出现频率：★★★★

在 32 位 WinNT 操作系统环境下，有如下代码:

```
1    char str[] = "Hello";
2    char *p = str;
3    int n = 10 ;
4    sizeof(str) = ______;
5    sizeof(p) = ______;
6    sizeof(n) = _______;
7    void Func ( char str[100] )
8    {
9        sizeof(str) = ____;
10   }
11   void *p = malloc(100);
12   sizeof(p) =______;
```

解析

程序代码说明如下。

- ❑ 代码第 4 行，str 变量表示数组，对数组变量做 sizeof 运算得到的是数组占用内存的总空间。注意，数组最后有一个元素保存字符串结束符，所以 sizeof(str) 为 strlen("Hello")+1。
- ❑ 代码第 5 行和第 6 行中的 p 和 n 分别是指针型变量和 int 型变量。在 32 位 WinNT 平台下，两者都占 4 个字节的空间，所以结果都是 4。
- ❑ 代码第 9 行中的 str 是函数的参数，它在做 sizeof 运算时被认为是指针。这是因为调用函数 Func（str）时，数组是“传址”的，程序会在栈上分配 4 字节的指针来指向数组，因此结果也是 4。
- ❑ 代码第 11 行中的 p 首先指向 100 字节的堆内存。这里仍是对指针做 sizeof 运算，结果是 4。

总之，数组和指针的 sizeof 运算有着细微的区别。如果数组变量被传入函数中做 sizeof 运算，则和指针的运算没有区别，否则会得到整个数组占用内存的总大小。对于指针，无论是何种类型的指针，其大小都是固定的，在 32 位 WinNT 平台下结果都是 4。

面试例题 14：分析代码写结果——使用 sizeof 计算类对象所占空间大小。

考点：sizeof 计算类对象所占空间大小。

出现频率：★★★★

请指出下面程序在 32 位 WinNT 操作系统环境下的输出是什么。

```
1     #include <iostream.h>
2
3     class A
4     {
5     public:
6             int i;
7     };
8
9     class B
10    {
11    public:
12           char ch;
13    };
14
15    class C
16    {
17    public:
18           int i;
19           short j;
20     };
21
22    class D
23    {
24    public:
25           int i;
26           short j;
27           char ch;
28    };
29
30    class E
31    {
32    public:
33           int i;
34           int ii;
```

```
35        short j;
36        char ch;
37        char chr;
38    };
39
40    class F
41    {
42    public:
43        int i;
44        int ii;
45        int iii;
46        short j;
47        char ch;
48        char chr;
49    };
50
51    int main()
52    {
53        cout << "sizeof(int) = " << sizeof(int) << endl;
54        cout << "sizeof(short) = " << sizeof(short) << endl;
55        cout << "sizeof(char) = " << sizeof(char) << endl;
56        cout << endl;
57        cout << "sizeof(A) = " << sizeof(A) << endl;
58        cout << "sizeof(B) = " << sizeof(B) << endl;
59        cout << "sizeof(C) = " << sizeof(C) << endl;
60        cout << "sizeof(D) = " << sizeof(D) << endl;
61        cout << "sizeof(E) = " << sizeof(E) << endl;
62        cout << "sizeof(F) = " << sizeof(F) << endl;
63
64        return 0;
65    }
```

解析

这里定义了 6 个类，很多人遇到对结构体或类做 sizeof 运算时，只是简单地把各个成员所占的内存数量相加。在 32 位 WinNT 操作系统环境下，char 占 1 个字节，int 占 4 个字节，short 占 2 个字节。因此会很轻易地做出以下的计算：

```
sizeof(A) = sizeof(int) = 4
sizeof(B) = sizeof(char) = 1
sizeof(C) = sizeof(int) + sizeof(short) = 4 + 2 = 6
sizeof(D) = sizeof(int) + sizeof(short) + sizeof(char) = 4 + 2 + 1 = 7
sizeof(E) = 2 * sizeof(int) + 2 * sizeof(char) + sizeof(short) = 2*4 + 2*1 + 2 = 12
sizeof(F) = 3 * sizeof(int) + 2 * sizeof(char) + sizeof(short) = 3*4 + 2*1 + 2 = 15
```

但实际上程序的执行结果如下：

```
sizeof(A) = 4
sizeof(B) = 1
sizeof(C) = 8
```

```
sizeof(D) = 8
sizeof(E) = 12
sizeof(F) = 16
```

这个问题会使许多初学者疑惑不解，而这种情况是由于字节对齐引起的。由于各个硬件平台对存储空间的处理上有很大的不同。一些平台对某些特定类型的数据只能从某些特定地址开始存取，然而其他平台可能没有这种情况，为了提高效率，要求工作对数据进行对齐。例如有些平台每次都是从偶地址开始读入，如果一个 int 型（假设为 32 位系统）存放在偶地址开始的地方，那么一个读周期就可以读出，而存放在奇地址开始的地方，会需要 2 个读周期，并对两次读出的结果的高低字节进行拼凑才能得到该 int 数据。显然读取效率会低很多。

通常写程序的时候，不需要考虑对齐问题。编译器会替我们选择适合目标平台的对齐策略。当然，也可以通过给编译器传递预编译指令而改变对指定数据的对齐方法。

字节对齐的细节与编译器实现相关，一般而言，需要满足 3 个准则。

- 结构体变量的首地址能够被其最宽的基本类型成员的大小所整除。
- 结构体每个成员相对于结构体首地址的偏移量（offset）都是成员大小的整数倍，如有需要编译器会在成员之间加上填充字节（internal adding）。
- 结构体的总大小为结构体最宽的基本类型成员的整数倍，如有需要编译器会在最末一个成员之后加上填充字节（trailing padding）。

以下面的结构体为例：

```
1   struct S
2   {
3       char c1;
4       int i;
5       char c2;
6   };
```

c1 的偏移量为 0，i 的偏移量为 4，c1 与 i 之间便需要 3 个填充字节。c2 的偏移量为 8，加起来就是 1+3+4+1 等于 9 个字节。由于这里最宽的基本类型为 int，大小为 4 字节，再补 3 字节凑成 4 的倍数，一共是 12 字节。示例代码中各个类大小的计算过程如下：

```
sizeof(A) = 4;
sizeof(B) = 1
sizeof(C) = 4 + 1 + 3 (补齐) = 8
sizeof(D) = 4 + 2 + 1 + 1 (补齐) = 8
sizeof(E) = 4 + 4 + 2 + 1 + 1 = 12
sizeof(F) = 4 + 4 + 4 + 2 + 1 + 1 = 16
```

答案

```
sizeof(int) = 4
sizeof(short) = 2
sizeof(char) = 1
sizeof(A) = 4
sizeof(B) = 1
```

```
sizeof(C) = 8
sizeof(D) = 8
sizeof(E) = 12
sizeof(F) = 16
```

面试例题 15：分析代码写结果——使用 sizeof 计算含有虚函数的类对象的空间大小。

考点：sizeof 计算含有虚函数的类对象的空间大小。

出现频率：★★★★

请指出下面程序在 32 位 WinNT 操作系统环境下的运行结果是什么。

```
1    #include <iostream>
2    using namespace std;
3
4    class Base
5    {
6    public:
7        Base(int x) : a(x)
8        {
9        }
10
11       void print()
12       {
13           cout << "base" << endl;
14       }
15    private:
16        int a;
17    };
18
19    class Derived : public Base
20    {
21    public:
22       Derived(int x) : Base(x-1), b(x)
23       {
24       }
25
26       void print()
27       {
28           cout << "derived" << endl;
29       }
30    private:
31       int b;
32    };
33
```

```
34  class A
35  {
36  public:
37      A(int x) : a(x)
38      {
39      }
40      virtual void print()
41      {
42          cout << "A" << endl;
43      }
44  private:
45      int a;
46  };
47
48  class B : public A
49  {
50  public:
51      B(int x) : A(x-1), b(x)
52      {
53      }
54      virtual void print()
55      {
56          cout << "B" << endl;
57      }
58  private:
59      int b;
60  };
61
62  int main()
63  {
64      Base obj1(1);
65      cout << "size of Base obj is " << sizeof(obj1) << endl;
66      Derived obj2(2);
67      cout << "size of Derived obj is " << sizeof(obj2) << endl;
68
69      A a(1);
70      cout << "size of A obj is " << sizeof(a) << endl;
71      B b(2);
72      cout << "size of B obj is " << sizeof(b) << endl;
73
74      return 0;
75  }
```

解析

上面这个程序定义了 4 个类，分别是 Base、Derived、A 和 B。其中 Derived 类是 Base 的子

类，B 是 A 的子类。

程序代码说明如下。

- 对于 Base 类来说，sizeof(int)等于 4，print()函数不占内存。
- 对于 Derived 类来说，比 Base 类多一个整型成员，因而多 4 字节，一共是 12 字节。
- 对于 A 类来说，由于它含有虚函数，因此占用的内存除了一个整型变量之外，还包括一个隐含的虚表指针成员，一共是 8 字节。
- 对于 B 类来说，比 A 类多一个整型成员，因而多 4 字节，一共是 12 字节。

通过这个例子可以看出，普通函数不占用内存，只有虚函数会占用一个指针大小的内存，原因是系统用一个指针维护这个类的虚函数表，并且注意这个虚函数无论含有多少项（类中含有多少个虚函数）都不会影响类的大小。

答案

```
size of Base obj is 4
size of Derived obj is 8
size of A obj is 8
size of B obj is 12
```

面试例题 16：分析代码写结果——使用 sizeof 计算虚拟继承的类对象的空间大小。

考点：sizeof 计算虚拟继承的类对象的空间大小。

出现频率：★★★★

```
1    #include <iostream>
2    using namespace std;
3
4    class A
5    {
6    };
7
8    class B
9    {
10   };
11
12   class C:public A, public B
13   {
14   };
15
16   class D:virtual public A
17   {
18   };
19
```

```
20    class E:virtual public A, virtual public B
21    {
22    };
23
24    class F
25    {
26    public:
27        int a;
28        static int b;
29    }
30
31    int F::b = 10;
32
33    int main()
34    {
35        cout << "sizeof(A) = " << sizeof(A) << endl;
36        cout << "sizeof(B) = " << sizeof(B) << endl;
37        cout << "sizeof(C) = " << sizeof(C) << endl;
38        cout << "sizeof(D) = " << sizeof(D) << endl;
39        cout << "sizeof(E) = " << sizeof(E) << endl;
40        cout << "sizeof(F) = " << sizeof(F) << endl;
41
42        return 0;
43    }
```

解析

程序代码说明如下。

- 代码第 35 行，由于 A 是空类，编译器会安插一个 char 空类，标记它的每一个对象，因此其大小为 1 字节。
- 代码第 36 行，类 B 大小和 A 相同，都是 1 字节。
- 代码第 37 行，类 C 是多重继承自 A 和 B，其大小仍然为 1。
- 代码第 38 行，类 D 是虚继承自 A，编译器为该类安插一个指向父类的指针，指针大小为 4。由于此类有了指针，编译器不会安插一个 char 了，因此其大小是 4 字节。
- 代码第 39 行，类 E 虚继承自 A 并且也虚继承自 B，因此它有指向父类 A 的指针与父类 B 的指针，加起来大小为 8 字节。
- 代码第 40 行，类 F 含有一个静态成员变量，这个静态成员的空间不在类的实例中，而是像全局变量一样在静态存储区中，被类共享，因此其大小是 4 字节。

答案

```
sizeof(A) = 1
sizeof(B) = 1
```

```
sizeof(C) = 1
sizeof(D) = 4
sizeof(E) = 8
sizeof(F) = 4
```

面试例题 17：sizeof 与 strlen 有哪些区别？

考点：理解 sizeof 与 strlen 的区别。

出现频率：★★★

解析

它们的区别如下。

- sizeof 是算符，strlen 是函数。
- sizeof 操作符的结果类型是 size_t，它在头文件中 typedef 为 unsigned int 类型，该类型保证能容纳实现所建立的最大对象的字节大小。
- sizeof 可以用类型做参数，strlen 只能用 char*做参数，且必须是以'“\0”结尾。
- 数组做 sizeof 的参数不退化，但是传递给 strlen 就退化为指针了。
- 大部分编译程序在编译的时候 sizeof 就被计算过了，这就是 sizeof(x)可以用来定义数组维数的原因。strlen 的结果要在运行的时候才能计算出来，它用来计算字符串的长度，不是类型占内存的大小。
- sizeof 后如果是类型必须加括弧，如果是变量名可以不加括弧。这是因为 sizeof 是个操作符不是个函数。
- 在计算字符串数组的长度上有区别。例如：

```
char str[20]="0123456789";
int a=strlen(str);
int b=sizeof(str);
```

a 计算的是以 0x00 结束的字符串的长度（不包括 0x00 结束符），这里结果是 10。

b 计算的则是分配的数组 str[20]所占的内存空间的大小，不受里面存储内容的改变而改变，这里结果是 20。

- 如果要计算指针指向的字符串的长度，则一定要使用 strlen。例如：

```
char* ss = "0123456789";
int a = sizeof(ss);
int b = strlen(ss);
```

a 计算的是 ss 指针占用的内存空间大小，这里结果是 4。

b 计算的是 ss 指向的字符串的长度，这里结果是 10。

面试例题 18：sizeof 有哪些用途？

考点：理解 sizeof 的用途。

出现频率：★★★★

解析

sizeof 有以下用途。

- 与存储分配和 I/O 系统的例程进行通信。例如：
- 查看某个类型的对象在内存中所占的单元字节。

```
void* memset(void* s,int c, sizeof(s));
```

- 动态分配对象时，可以使系统知道要分配多少内存。
- 便于一些类型的扩充，在 Windows 中很多结构类型有一个专用的字段是用来存放该类型的字节大小。
- 由于操作数的字节数在实现时可能出现变化，建议在涉及操作数字节大小时用 sizeof 来代替常量计算。
- 如果操作数是函数中的数组形参或函数类型的形参，sizeof 给出其指针的大小。

面试例题 19：找错——使用 strlen()函数代替 sizeof 计算字符串长度。

考点：sizeof 不能用于计算字符串长度。

出现频率：★★★

```
1   #include <iostream.h>
2   #include <string.h>
3
4   void UpperCase(char str[])
5   {
6       int test = sizeof(str);
7       int test2 = sizeof(str[0]);
8
9       for(size_t i=0; i<sizeof(str)/sizeof(str[0]); ++i)
10          if('a'<=str[i] && str[i]<='z')
11              str[i] -= ('a'-'A');
12  }
13
14  int main()
15  {
16      char str[] = "aBcDe";
17      cout << "The length of str is " << sizeof(str)/sizeof(str[0]) << endl;
18      UpperCase( str );
19      cout << str << endl;
20      return 0;
21  }
```

解析

这个程序存在着 2 方面的问题。

- 函数 UpperCase(char str[])的意图是将 str 指向的字符串中小写字母换为大写字母。于是在代码第 9 行利用“sizeof(str)/sizeof(str[0])”获得数组中的元素个数以便做循环操作。然而 sizeof(str)得到的并不是数组占用内存的总大小，而是 1 个字符指针的大小，为 4 字节。

因此这里只能循环4次，在代码18行main()函数的调用中只能改变对数组的前4个字符进行转换。转换的结果为“ABCDe”。

- 代码第17行的意图是使用要打印字符串的长度。然而，“sizeof(str)/sizeof(str[0])”计算的是数组元素的个数，比字符串的长度大1，原因是数组的长度还包括字符串的结束符“\0”。

应该用strlen()函数来代替sizeof计算字符串长度。正确的代码如下：

```
1    #include <iostream.h>
2    #include <string.h>
3
4    void UpperCase(char str[])
5    {
6        int test = sizeof(str);
7        int test2 = sizeof(str[0]);
8
9        for(size_t i=0; i<strlen(str); ++i)   //计算字符串的长度
10           if('a'<=str[i] && str[i]<='z')
11               str[i] -= ('a'-'A');
12   }
13
14   int main()
15   {
16       char str[] = "aBcDe";
17
18       cout << "The length of str is " << strlen(str) << endl;   //计算字符串的长度
19       UpperCase( str );
20       cout << str << endl;
21       return 0;
22   }
```

答案

代码第9行和第17行中的“sizeof(str)/sizeof(str[0])”应该用strlen(str)替换。

面试例题20：分析代码写结果——使用sizeof计算联合体的大小。

考点：使用sizeof计算联合体的大小。

出现频率：★★★★

```
1    #include <iostream.h>
2
3    union u
4    {
5        double a;
6        int b;
7    };
8
9    union u2
```

```
10  {
11      char a[13];
12      int b;
13  };
14
15  union u3
16  {
17      char a[13];
18      char b;
19  };
20
21  int main()
22  {
23      cout<<sizeof(u)<<endl;
24      cout<<sizeof(u2)<<endl;
25      cout<<sizeof(u3)<<endl;
26
27      return 0;
28  }
```

解析

这个程序定义了 3 个联合体：u、u2 和 u3。联合体的大小取决于它所有的成员中占用空间最大的一个成员的大小，并且对于复合数据类型，例如 union、struct、class，对齐方式为成员中最大的成员对齐方式。

- 对于 u 来说，大小取决于最大的 double 类型成员 a，即 sizeof(u)=sizeof(double)=8。
- 对于 u2 来说，最大的空间是 char[13]类型的数组。这里要注意，由于它的另一个成员 int b 的存在，使 u2 的对齐方式变成 4，也就是说 u2 的大小必须在 4 倍数的对界上，所以占用的空间变为最接近 13 的对界，即 16。
- 对于 u3 来说，最大的空间是 char[13]类型的数组，即 sizeof(u3)=13。

这里又出现了 CPU 对齐的问题。编译器会尽量把成员对齐以提高内存的命中率。对齐是可以更改的，使用“#pragma pack(x)”可以改变编译器的对齐方式。C++固有类型的对界取编译器对齐方式与自身大小中较小的一个。例如，指定编译器按 2 对齐，int 类型的大小是 4，则 int 的对界为 2 和 4 中较小的 2。在默认的对界方式下，几乎所有的数据类型都不大于默认的对界方式 8（除了 long 和 double），因此所有的固有类型的对齐方式可以认为就是类型自身的大小。示例程序如下：

```
1  #include <iostream.h>
2
3  #pragma pack(2)
4
5  union u
6  {
7      char buf[9];
```

```
8          int a;
9      };
10
11     int main()
12     {
13         cout << sizeof(u) << endl;
14
15         return 0;
16     }
```

上面的程序中，由于使用手动更改对齐方式为 2，所以 int 的对齐也变成了 2（int 自身对齐为 4），u 的对齐取成员中最大的对齐，也是 2，所以此时 sizeof(u)=10。

如果注释上面的代码第 3 行，int 的对齐使用 4，取成员中最大的对齐，也是 4，所以此时 sizeof(u)=12。

答案

```
8
16
13
```

面试例题 21：选择题——#pragma pack 的作用。

考点：理解#pragma pack 指令的作用。

出现频率：★★★★

```
1      #include <iostream.h>
2
3      #pragma pack(1)
4
5          struct test {
6          char c;
7          short s1;
8          short s2;
9          int i;
10     };
11
12     int main()
13     {
14         cout<< sizeof(test) <<endl;
15
16         return 0;
17     }
```

A. 9　　B. 10　　C. 12　　D. 16

解析

代码第 3 行用“#pragma pack”将对齐设为 1。由于结构体 test 中的成员 s1、s2 和 i 的自身对齐为分别为 2、2 和 4，都大于 1。因此它们都是用 1 作为对齐，sizeof(test) = 1 + 2 + 2 + 4 = 9。

如果注释代码第 3 行，则编译器默认对齐为 8。各个成员自身的对齐都小于 8，因此它们使用自身的对齐，sizeof(test)=1+1(补齐)+2+2+2（补齐）+4=12。

答案

A

5.5 inline 与宏定义

inline（内联函数）是 C++引入的机制，其目的是解决使用宏定义的一些缺点。面试中求职者经常会被问及 inline 和宏定义之间的区别。显然，这类的题目需要求职者对于 C 的宏定义以及 C++的 inline 知识都十分清楚，掌握各自的特点以及区别。

面试例题 22：为什么要引入内联函数？

考点：理解内联函数的作用。

出现频率：★★★★

解析

引入内联函数的主要目的是用它替代 C 中表达式形式的宏定义，解决程序中函数调用的效率问题。在 C 语言里可以使用如下的宏定义：

```
#define ExpressionName(Var1,Var2) (Var1+Var2)*(Var1-Var2)
```

这种宏定义在形式及使用上像一个函数，但它使用预处理器实现，没有了参数压栈、代码生成等一系列的操作，因此效率很高。这种宏定义在形式上类似于一个函数，但在使用它时，只是做预处理器符号表中的简单替换，因此它不能进行参数有效性的检测，也就不能享受 C++编译器严格类型检查的好处，另外它的返回值也不能被强制转换为可转换的合适类型，这样，它的使用就存在着一系列的隐患和局限性。

另外，在 C++中引入了类及类的访问控制，这样，如果一个操作或者说一个表达式涉及类的保护成员或私有成员，你就不可能使用这种宏定义来实现（因为无法将 this 指针放在合适的位置）。

inline 推出的目的是为了取代这种表达式形式的宏定义，消除了宏定义的缺点，同时又很好地继承了它的优点。

面试例题 23：为什么 inline 能很好地取代表达式形式的预定义呢？

考点：理解内联函数相比宏定义的优越处。

出现频率：★★★

解析

原因如下。

- inline 定义类的内联函数，函数代码被放入符号表中，在使用时直接进行替换（像宏一样展开），没有了调用的开销，效率很高。
- 类的内联函数也是一个真正的函数。编译器在调用一个内联函数时，首先会检查它的参数的类型，保证调用正确，然后再进行一系列的相关检查，就像对待任何一个真正的函数一样。这样就消除了它的隐患和局限性。
- inline 可以作为某个类的成员函数，当然可以使用所在类的保护成员及私有成员。

面试例题 24：说明内联函数使用的场合。

考点：理解内联函数的作用场合。

出现频率：★★★

解析

首先使用 inline 函数可以完全取代表达式形式的宏定义。

内联函数在 C++类中应用最广的，是用来定义存取函数。定义的类中一般会把数据成员定义成私有的或者保护的，这样，外界就不能直接读写类成员的数据了。对于私有或者保护成员的读写就必须使用成员接口函数来进行。如果把这些读写成员函数定义成内联函数的话，将会获得比较好的效率。示例的代码如下。

```
1    Class A
2    {
3    Private:
4        int nTest;
5    Public:
6        int readTest()
7        {
8            return nTest;
9        }
10       void setTest(int i);
11   };
12
13   inline void A::setTest(int i)
14   {
15       nTest = i;
15   };
```

类 A 的成员函数 readTest()和 setTest()都是 inline 函数。readTest()函数的定义体被放在类声明之中，因而 readTest()自动转换成 inline 函数，setTest()函数的定义体在类声明之外，因此要加上 inline 关键字。

面试例题 25：为什么不把所有的函数都定义成内联函数？

考点：理解内联函数的缺点。

出现频率：★★★★

解析

内联是以代码膨胀（复制）为代价，仅仅省去了函数调用的开销，从而提高函数的执行效率。一方面，如果执行函数体内代码的时间相比于函数调用的开销较大，那么效率的收获会很少。另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。以下情况不宜使用内联。

- 如果函数体内的代码比较长，使用内联将导致内存消耗代价较高。
- 如果函数体内出现循环，那么执行函数体内代码的时间要比函数调用的开销大。

另外，类的构造函数和析构函数容易让人误解成使用内联更有效。要当心构造函数和析构函数可能会隐藏一些行为，如“偷偷地”执行了基类或成员对象的构造函数和析构函数。所以不要随便地将构造函数和析构函数的定义体放在类声明中。

一个好的编译器将会根据函数的定义体，自动地取消不值得的内联（这说明了 inline 不应该出现在函数的声明中）。

面试例题 26：内联函数与宏有什么区别？

考点：理解内联函数与宏定义的区别。

出现频率：★★★★

解析

二者区别如下。

- 内联函数在编译时展开，宏在预编译时展开。
- 在编译的时候内联函数可以直接被嵌入到目标代码中，而宏只是一个简单的文本替换。
- 内联函数可以完成诸如类型检测、语句是否正确等编译功能，宏就不具有这样的功能。
- 宏不是函数，inline 函数是函数。
- 宏在定义时要小心处理宏参数(一般情况是把参数用括号括起来)，否则容易出现二义性。而内联函数定义时不会出现二义性。

第6章 引用和指针

引用是C++常用的重要内容之一，正确、灵活地使用引用，可以使程序简洁、高效。

指针是C语言中广泛使用的一种数据类型。使用指针可以编写出精练而高效的程序。学习指针是学习C语言最重要的一环，同时，也是最为困难的一环。

指针问题是各公司针对求职者的重点考点，常见问题涉及数组指针、函数指针、常量指针、指针传值、多维指针等。

本章基于求职者实际会遇到的公司面试题，对指针的各个方面的重点难点进行全面且细致的分析。通过阅读本章，读者能解决在运用指针时可能面对的各个难点。

6.1 引　　用

引用（reference），又称为别名（alias），可以作为某一对象的另一个名字，是由C++引入的特性。通过引用可以间接地操纵对象，其使用方式类似于指针，但是不需要指针的语法。

6.1.1 引用的基本问题

引用可以看作是对象的一个别名，可以通过操作这个别名来操作实际对象。注意：引用在声明的同时必须被初始化。以下代码会出现编译错误：

```
int a = 1;
int &b;                //编译错误
```

上面代码的代码第2行如果改成：

```
int &b = a             //b 为 a 的引用
```

这时，b就被声明为a的引用，对b的赋值实际上也就是对a的赋值。

面试例题1：分析代码写结果——一般变量引用。

考点：一般变量引用。

出现频率：★★★★

```
1    #include <iostream>
2    #include <string>
```

```
3    using namespace std;
4
5    int main(int argc, char* argv[])
6    {
7        int a = 10;
8        int b = 20;
9        int &rn = a;
10       int equal;
11
12       rn=b;
13       cout << "a = " << a << endl;
14       cout << "b = " << b << endl;
15
16       rn = 100;
17
18       cout << "a = " << a << endl;
19       cout << "b = " << b << endl;
20
21       equal = (&a == &rn)? 1: 0;
22
23       cout << "equal = " << equal << endl;
24
25       return 0;
26   }
```

解析

- 代码第 7 行和第 8 行，整型变量 a 和整型变量 b 分别被初始化为 10 和 20。
- 代码第 9 行，声明 rn 为变量 a 的一个引用。
- 代码第 12 行，将 rn 的值赋值为 b 的值。此时 rn 其实就是 a 的一个别名，对 rn 的赋值其实就是对 a 的赋值。因此执行完赋值后，a 的值就是 b 的值，即都是 20。
- 代码第 16 行，将 rn 的值赋值为 100，于是 a 的值变成了 100。
- 代码第 21 行，将 a 的地址与 rn 的地址进行比较，如果相等，变量 equal 的值为 1，否则为 0。将 rn 声明为 a 的引用，不需要为 rn 另外开辟内存单元。rn 和 a 占内存中的同一个存储单元，它们具有同一地址。所以 equal 的值为 1。

答案

```
a = 20
b = 20
a = 100
b = 20
equal = 1
```

面试例题 2：分析代码写结果——指针变量引用。

考点：指针变量引用。

出现频率：★★★★

```
1    #include <iostream>
2    using namespace std;
```

```
3    int main(int argc, char* argv[])
4    {
5        int a = 1;
6        int b = 10;
7        int* p = &a;
8        int* &pa = p;
9
10       (*pa)++;
11       cout << "a = " << a << endl;
12       cout << "b = " << b <<endl;
13       cout << "*p = " << *p <<endl;
14
15       pa = &b;
16       (*pa)++;
17       cout << "a = " << a << endl;
18       cout << "b = " << b <<endl;
19       cout << "*p = " << *p <<endl;
20
21       return 0;
22   }
```

解析

- 代码第5行和第6行，整型变量a和整型变量b分别被初始化为1和10。
- 代码第7行，声明整型的指针变量p并初始指向a。
- 代码第8行，声明p的一个指针引用pa。
- 代码第10行，将pa指向的内容加1。由于pa是p的引用，所以此时实际上是对p指向的内容加1，也就是a加1，结果为a变成了2。
- 代码第15行，将pa指向变量b的地址，由于pa是p的引用，所以此时p也指向了b的地址。
- 代码第16行，将pa指向的内容加1。由于pa是p的引用，所以此时实际上是对p指向的内容加1，也就是b加1，结果为b变成了12。

答案

```
a = 2
b = 10
*p = 2
a = 2
b = 11
*p = 11
```

面试例题3：分析代码找错误——变量引用。

考点：一般变量引用。

出现频率：★★★

```
1    #include <iostream>
2    using namespace std;
```

```
3
4   int main(int argc, char* argv[])
5   {
6       int a = 1, b = 2;
7       int &c;
8       int &d = a;
9       &d = b;
10      int *p;
11
12      *p = 5;
13
14      return 0;
15  }
```

解析

- 代码第 6 行正确。声明并初始化整型变量 a 和 b。
- 代码第 7 行编译错误。声明一个引用类型的变量 c，但是没有初始化。引用类型的变量在声明的同时必须初始化。
- 代码第 8 行正确。声明了变量 a 的引用 d。
- 代码第 9 行编译错误。因为引用只能在声明的时候被赋值，以后都不能再把该引用名作为其他变量名的别名。
- 代码第 10 行正确。声明了一个整型的指针变量 p。
- 代码第 12 行运行错误。由于 p 没有被初始化，因此 p 是个野指针，对野指针赋值是非常危险的，会导致程序运行时崩溃。

6.1.2 参数引用

引用的一个重要作用就是作为函数的参数。如果 C 语言中有大块数据作参数传递，采用的方案往往是指针。C++增加了一种同样有效率的选择，这就是引用。

面试例题 4：交换两个字符串——参数引用。

考点：参数引用。

出现频率：★★★

解析

程序代码如下：

```
1   #include<iostream.h>
2   #include<string.h>
3   void swap(char *&x, char *&y)
4   {
5       char *temp;
6       temp=x;
7       x=y;
```

```
8          y=temp;
9      }
10
11     int main()
12     {
13         char *ap = "hello";
14         char *bp = "how are you?";
15
16         cout << "ap:" << ap << endl;
17         cout << "bp:" << bp << endl;
18
19         swap(ap, bp);
20
21         cout << "swap ap,bp" << endl;
22         cout << "ap:" << ap << endl;
23         cout << "bp:" << bp << endl;
24
25         return 0;
26     }
```

swap 函数利用指针引用实现字符串交换。由于 swap 函数是指针引用，因此传入函数的就是实参，而不是形参。

如果不用指针引用，那么指针交换只能在 swap 函数中有效，这是因为在函数体中，函数栈会分配两个临时的指针变量分别指向两个指针参数，对实际的 ap 和 bp 没有影响。

函数执行的结果如下：

```
ap:hello
bp:how are you?
swap ap,bp
ap:how are you?
bp:hello
```

从执行结果来看，swap 函数确实起到了交换两个字符串的目的。

如果不使用引用，可以用二维指针达到同样的目的。这时需要把 swap 函数的定义改为下面的形式：

```
1     void swap1(char **x, char **y)
2     {
3         char *temp;
4         temp = *x;
5         *x = *y;
6         *y = temp;
7     }
```

并且源代码第 19 行改成：

```
swap1(&ap, &bp);
```

用这种传指针地址的方式，同样可以交换两个字符串。

面试例题 5：程序查错——参数引用。

考点：参数引用。

出现频率：★★★

```
1    #include <iostream.h>
2
3    const float pi=3.14f;
4    float f;
5
6    float f1(float r)
7         { f = r*r*pi;
8
9           return f;
10   }
11   float& f2(float r)
12        { f = r*r*pi;
13
14          return f;
15   }
16
17   int  main(){
18        float f1(float=5);
19        float& f2(float=5);
20        float a=f1();
21        float& b=f1();
22        float c=f2();
23        float& d=f2();
24
25        d += 1.0f;
26
27        cout << "a = " << a << endl;
28        cout << "b = " << b << endl;
29        cout << "c = " << c << endl;
30        cout << "d = " << d << endl;
31        cout << "f = " << f << endl;
32
33        return 0;
34   }
```

解析

这里 f1()函数返回的是全局变量 f 的值，f2()函数返回的是全局变量 f 的引用。

- 代码第 18 行正确，声明函数 f1()的默认参数调用，其默认参数值为 5。
- 代码第 19 行正确，声明函数 f2()的默认参数调用，其默认参数值为 5。
- 代码第 20 行正确，将变量 a 赋为 f1()的返回值。
- 代码第 21 行错误，将变量 b 赋为 f1()的返回值。因为在 f1()函数里，全局变量 f 的值

78.5 赋给临时变量 temp，而 temp 变量由编译器隐式建立，然后再建立 temp 的引用 b。这里对临时变量 temp 进行引用会发生错误。

- 代码 22 行正确，f2()函数在返回值时并没有隐式地建立临时变量 temp，而是直接将全局变量 f 返回给主函数。
- 代码 23 行正确，主函数中不定义变量，而是直接使用全局变量的引用，这种是最节省内存空间的方式。但必须注意它所引用变量的有效期，此处全局变量 f 的有效期肯定长于引用 d，所以是安全的，否则，会出现错误。例如，将局部变量的引用返回，此时全局变量 f 的值为 78.5。
- 代码 25 行正确，将 d 的值加 1.0，此时 d 是全局变量 f 的引用，因此 f 的值变成 79.5。

答案

代码 21 行错误。注释 21 行与 28 行后，运行结果如下：

```
A = 78.5
C = 78.5
D = 79.5
F = 79.5
```

6.1.3 常量引用

常引用声明方式：const 类型标识符 &引用名=目标变量名。例如：

```
1   int a ;
2   const int &refa = a;
3   refa = 1; //错误
4   a = 1; //正确
```

因为上述程序不能通过引用对目标变量的值进行修改，从而使引用的目标成为常量，达到了引用安全的目的。

面试例题 6：程序查错——参数引用。

考点：参数引用。

出现频率：★★★

```
1   #include <iostream>
2   using namespace std;
3
4   class Test
5   {
6   public:
7       void f(const int& arg);
8   private:
9       int value;
10  };
11
12  void Test::f(const int& arg)
```

```
13          { arg = 10;
14          cout << "arg = " << arg << endl;
15          value = 20;
16      }
17
18      int main()
19      {
20          int a = 7;
21          const int b = 10;
22          int &c = b;
23          const int &d = a;
24
25          a++;
26          d++;
27
28          Test test;
29          test.f(a);
30          cout << "a = " << a << endl;
31
32          return 0;
33      }
```

解析

把 const 放在引用之前表示声明一个常量引用。不能使用常量引用修改变量的值。

- 代码第 13 行错误。因为参数 arg 是常量引用类型，arg 的值在函数体内不能被修改。
- 代码第 20 行和 21 行中 a 被声明为整型变量，b 被声明为整型常量。
- 代码第 22 行，错误。因为 b 为常量，而 c 不是常量引用，所以 C 不可以是 b 的引用。正确的方式应为：

```
const int &c = b;
```

- 代码第 23 行正确声明 d 为 a 的常量引用。
- 代码第 25 行正确变量 a 自增 1。
- 代码 26 行错误。d 是常量引用，不能对 d 是用赋值操作。

从上面的分析可以看出，对于常量类型的变量，其引用也必须是常量类型的。对于非常量类型的变量，其引用可以是非常量的也可以是常量的，但是，无论什么情况，都不能使用常量引用修改其引用的变量的值。

答案

代码第 13 行错误。原因是 arg 的值不能被修改。

代码第 22 行错误。原因是常量类型变量不能定义非常量的引用。

代码第 26 行错误。原因是不能使用常量引用修改变量的值。

6.1.4 引用与指针的区别

指针与引用编写方法完全不同（指针使用操作符*和->，引用使用操作符&），但是它们有

类似的功能。指针与引用都可以间接引用其他对象。下面通过例题说明什么时候使用指针，又在什么时候使用引用。

面试例题7：指针和引用有什么区别？

考点：引用和指针的区别。

出现频率：★★★★★

解析

区别如下所示。

- 初始化要求不同。引用在创建的同时必须初始化，而指针在定义的时候不必初始化，可以在定义后面的任何地方重新赋值。
- 可修改性不同。引用一旦被初始化，它就不能被另一个对象引用；而指针在任何时候都可以指向另一个对象。
- 不存在NULL引用。引用不能使用指向空值的引用，它必须指向某个对象；而指针则可以是 NULL，不需要总是指向某些对象，可以把指针指向任意对象，所以指针更加灵活，也容易出错。
- 测试时的区别。由于引用不会指向空值，这意味着使用引用之前不需要测试它的合法性；而指针则需要经常进行测试。因此使用引用的代码效率比使用指针的要高。
- 应用的区别。如果指向一个对象后就不会改变指向，那么应该使用引用。如果指向NULL（不指向任何对象）或在不同的时刻指向不同的对象，应该使用指针。

实际上，在二进制层面，引用一般都是通过指针来实现的，只不过编译器帮我们完成了转换。总的来说，引用既具有指针的效率，又具有变量使用的方便性和直观性。

面试例题8：为什么引用比指针安全？

考点：引用和指针的区别。

出现频率：★★★★

解析

由于不存在空引用，并且引用一旦被初始化指向一个对象，它就不能被改变为另一个对象的引用，因此引用很安全。

指针可以随时指向别的对象，并且可以不被初始化，或初始化为，所以不安全。

6.2 指针基础

C语言之所以功能强大，很大部分是因为它有灵活的指针运用。C++作为从C语言发展而来的一门高级语言，并没有摒弃指针，而是积极地采纳它，与此同时又注入了面向对象的高级

思想，这使得 C++成为了最优秀的语言之一。

指针的概念很简单，可以把它看做一种数据类型，定义指针变量就像定义 int、char 型变量一样，只不过 int 型变量存储整数，char 型变量存储字符，而指针存储的是内存地址。

6.2.1 指针的声明

指针的声明非常灵活，应聘中经常会考查求职者是否能熟练掌握各种复杂指针声明的方法。

面试例题 9：复杂指针的声明。

考点：复杂指针的声明。

出现频率：★★★★★

用变量 a 给出下面的定义：

（1）定义一个整型数。

（2）定义一个指向整型数的指针。

（3）定义一个指向指针的的指针，它指向的指针是一个指向整型数指针。

（4）定义一个有 10 个整型数的数组。

（5）定义一个有 10 个指针的数组，该指针是指向一个整型数的指针。

（6）定义一个指向有 10 个整型数数组的指针。

（7）定义一个指向函数的指针，该函数有一个整型参数并返回一个整型数。

（8）定义一个有 10 个指针的数组，该指针指向一个函数，该函数有一个整型参数并返回一个整型数。

答案

（1）int a;

（2）int *a;

（3）int **a;

（4）int a[10];

（5）int *a[10];

（6）int (*a)[10];

（7）int (*a)(int);

（8）int (*a[10])(int);

知识扩展：解读复杂指针声明

右左法则：首先从最里面的圆括号看起，然后往右看，再往左看。每当遇到圆括号时，就应该调转阅读方向。一旦解析完圆括号里面所有的东西，就跳出圆括号。重复这个过程直到整个声明解析完毕。

这里对这个法则进行一个小小的修正，应该是从未定义的标识符开始阅读，而不是从括号读起，这是因为一个声明里未定义的标识符只会有一个。

现在通过几个例子来讨论如何运用右左法则解读复杂指针声明。

```
int (*func)(int *p);
```

首先找到未定义的标识符 func，它的外面有一对圆括号，而且左边是一个*号，这说明 func 是一个指针，然后跳出这个圆括号，先看右边，也是一个圆括号，这说明(*func)是一个函数，而 func 是一个指向这类函数的指针，就是一个函数指针，这类函数具有 int*类型的形参，返回值类型是 int。

```
int (*func)(int *p, int (*f)(int*));
```

func 被一对括号包含，且左边有一个*号，说明 func 是一个指针，跳出括号，右边也有个括号，那么 func 是一个指向函数的指针，这类函数具有形参 int *和 int (*)(int*)，返回值为 int 类型。func 的形参 int (*f)(int*)类似前面的解释，f 也是一个函数指针，指向的函数具有 int*类型的形参，返回值为 int。

```
int (*func[5])(int *p);
```

func 右边是一个[]运算符，说明 func 是一个具有 5 个元素的数组，func 左边有一个*，说明 func 的元素是指针，要注意这里的*不是修饰 func 的，而是修饰 func[5]的，原因是[]运算符优先级比*高，func 先跟[]结合，因此*修饰的是 func[5]。跳出这个括号，看右边，也是一对圆括号，说明 func 数组的元素是函数类型的指针，它所指向的函数具有 int*类型的形参，返回值类型为 int。

```
int (*(*func)[5])(int *p);
```

func 被圆括号括起，左边有一个*，因此 func 是一个指针，跳出括号，右边是一个[]运算符号，说明 func 是一个指向数组的指针，向左看，左边是*号，说明这个数组的元素是指针，再跳出括号，右边又有一个括号，说明这个数组的元素是指向函数的指针。总结一下，即 func 是一个指向数组的指针，这个数组的元素是函数指针，这些指针指向具有 int*形参，返回值为 int 类型的函数。

```
int (*(*func)(int *p))[5];
```

func 是一个函数指针，这类函数具有 int*类型的形参，返回值是指向数组的指针，指向的数组的元素是具有 5 个 int 元素的数组。

6.2.2 指针的运算

指针的运算就是地址的运算，因此指针运算不同于普通变量，它只允许有限的几种运算。除了可把指针指向某一存储单元外，允许指针与整数相加或相减，用来移动指针；允许两个指针相减，可以得到两个地址之间的数据个数；还允许指针与指针或指针与地址之间进行比较，决定指针所指向的存储位置的先后。

面试例题 10：分析代码写结果——用指针赋值。

考点：用指针赋值。

出现频率：★★★★

```
1    #include <stdio.h>
2
3    int main(void)
```

```
4     {
5         char a[] = "hello, world";
6         char * ptr = a;
7
8         printf("%c\n", *(ptr+4));
9         printf("%c\n", ptr[4]);
10        printf("%c\n", a[4]);
11        printf("%c\n", *(a+4));
12
13        *(ptr+4) += 1;
14        printf("%s\n", a);
15
16        return 0;
17    }
```

解析

程序代码说明如下。

- 代码第 5 行，声明了字符数组 a，并初始化为“hello, world”，包括以“\0”结束字符。
- 代码第 6 行，声明字符指针 ptr，并初始化指向数组 a 首地址（a 的第一个元素地址）。
- 代码第 8 行，将 ptr 加 4，再输出地址的内容，即输出 a[4]的内容。
- 代码第 9 行，ptr[4]和*(ptr+4)一样，也是 a[4]的内容。
- 代码第 10 行，输出 a[4]的内容。
- 代码第 11 行，*(a+4)和 a[4]一样，也是 a[4]的内容。
- 代码第 13 行，使数组 a[4]的内容加 1。由于原来 a[4]的内容为“Hello, world”字符串的第 5 个字符“o”，加 1 后就是“p”。

答案

```
o
o
o
o
p
```

面试例题 11：分析代码写结果——指针加减操作。

考点：指针加减操作。

出现频率：★★★★

```
1     #include <stdio.h>
2
3     int main(void)
4     {
5         int a[5]={1,2,3,4,5};
6         int *ptr=(int *)(&a+1);
7
```

```
8        printf("%d\n", *(a+1));
9        printf("%d\n", *(ptr-1));
10
11       return 0;
12   }
```

解析

这里主要考查对指针加减操作的理解。

对指针进行加 1 操作，得到的是下一个元素的地址，而不是原有地址值直接加 1。所以，类型为 t 的指针的移动是以 sizeof(t)为移动单位。

- 代码第 5 行，声明一维数组 a，并且 a 有 5 个元素。
- 代码第 6 行，ptr 是一个 int 型的指针（&a + 1）。即先取 a 的地址，该地址的值加 sizeof(a) 的值，即&a + 5*sizeof(int)，也就是 a[5]的地址，显然当前指针已经越过了数组的界限。(int *)(&a+1)则是把上一步计算出来的地址，强制转换为 int*类型，并赋值给 ptr。
- 代码第 8 行，a 与&a 的地址是一样的，但含意不一样，a 是数组首地址，用户 a[0]的地址，&a 是对象（数组）首地址，a+1 是数组下一元素的地址，即 a[1]，而&a+1 是下一个对象的地址，即 a[5]。因此这里输出 2。
- 代码第 9 行，因为 ptr 指向 a[5]，并且 ptr 是 int *类型，所以*(ptr-1)指向 a[4]，输出 5。

答案

```
2
5
```

面试例题 12：分析代码写结果——指针比较。

考点：指针比较操作。

出现频率：★★★★

```
1    #include <iostream>
2    using namespace std;
3
4    int main(void)
5    {
6        char str1[]= "abc";
7        char str2[]= "abc";
8        const char str3[] = "abc";
9        const char str4[] = "abc";
10       const char* str5= "abc";
11       const char* str6= "abc";
12       char* str7= "abc";
13       char* str8= "abc";
14
15       cout << ( str1==str2 ) << endl;
16       cout << ( str3==str4 ) << endl;
17       cout << ( str5==str6 ) << endl;
```

```
18        cout << ( str6==str7 ) << endl;
19        cout << ( str7==str8 ) << endl;
20
21        return 0;
22    }
```

解析

程序考查有关是内存中各个数据的存储方式。

数组 str1、str2、str3 和 str4 都是在栈中分配的，内存中的内容都是“abc”加一个“\0”，但是它们的存储位置不同，因此代码 15 行和 16 行的输出都是 0。

指针 str5、str6、str7 和 str8 也是在栈中分配的，它们都指向“abc”字符串，注意“abc”存放在数据区，所以 str5、str6、str7 和 str8 指向同一块数据区的内存，因此 17、18 和 19 行的输出是 1。

答案

```
0
0
1
1
1
```

面试例题 13：分析代码找错误——内存访问违规。

考点：指针操作内存违规。

出现频率：★★★★

```
1     #include <iostream>
2     using namespace std;
3
4     int main()
5     {
6         char a;
7         char*str1 = &a;
8         char* str2 = "AAA";
9
10        strcpy(str1, "hello");
11        cout << str1 << endl;
12
13        str2[0]='B';
14        cout << str2 << endl;
15
16        return 0;
17    }
```

解析

- 代码第 10 行，str1 指向一个字节大小的内存区。由于复制“hello”字符串最少需要 6

字节，显然内存大小不够。因此会因为越界进行内存读写而导致程序崩溃。

- 代码第13行，str2指向“AAA”这个字符串常量。因为是常量，所以对str2[0]的赋值操作是不合法的，也会导致程序崩溃。

答案

代码第10行导致运行错误。

代码第13行导致运行错误。

面试例题14：分析代码找错误——指针的隐式转换。

考点：指针类型的隐式转换。

出现频率：★★★★

```
1    #include <stdio.h>
2
3    int main()
4    {
5        int ival = 1024;
6        int ival2 = 2048;
7        int *pi1 = &ival;
8        int *pi2 = &ival2;
9        int **pi3 = 0;
10
11       ival = *pi3;
12       *pi2 = *pi3;
13       ival = pi2;
14       pi2 = *pi1;
15       pi1 = *pi3;
16       ival = *pi1;
17       pi1 = ival;
18       pi3 = &pi2;
19
20       return 0;
21   }
```

解析

- 代码第5行正确，声明并初始化整型变量ival。
- 代码第6行正确，声明并初始化整型变量ival2。
- 代码第7行正确，声明整型指针变量pi1，初始化指向ival。
- 代码第8行正确，声明整型指针变量pi2，初始化指向ival2。
- 代码第9行正确，声明二维整型指针变量pi3，初始化为0。
- 代码第11行编译错误，*ival是int类型，pi3是int*类型，不能隐式转换。
- 代码第12行编译错误，*pi2是int类型，*pi3是int*类型，不能隐式转换。
- 代码第13行编译错误，ival是int类型，pi2是int*类型，不能隐式转换。

- 代码第 14 行编译错误，pi2 是 int *类型，*pi1 是 int 类型，不能隐式转换。
- 代码第 15 行运行错误，pi3 是 NULL 指针，试图得到*pi3 的值时，发生运行错误。
- 代码第 16 行正确，将 ival 的值赋为*pi1。
- 代码第 17 行编译错误，pi1 是 int *类型，ival 是 int 类型，不能隐式转换。
- 代码第 18 行正确，将 pi3 的值赋为&pi2，都是 int **类型。

本题中错误类型有两种，一种是编译错误，另一种是运行错误。导致编译错误是因为类型之间不能隐式转换，如 int 转换成 int *、int *转换成 int 等。导致运行错误是因为：在 Windows 平台，进程的内存空间有一块是专门用于 NULL 指针分配的分区，这个分区的地址空间是禁止进入的，因此就会发生内存访问违规现象，同时该进程将终止运行。

答案

代码 11、12、13、14、17 行编译错误。

15 行运行错误。

6.2.3 指针常量与常量指针

指针常量与常量指针是两个容易混淆的概念。许多公司在面试时通过这两个概念考查应聘者基本功。

面试例题 15：指针常量与常量指针的区别。

考点：指针常量与常量指针的区别。

出现频率：★★★★★

解析

像两个连着的词，通常前面的词是修饰部分，后面的词则是中心词。

- 常量指针，即指针是常量形成，它首先应该是一个指针。
- 指针常量，常量是指针形式，它首先应该是一个常量。

接下来进行详细分析。

常量指针，它是一个指向常量的指针。常量指针指向一个常量，是防止对指针误操作出现修改常量这样的错误。因此，常量指针是指向常量的指针，指针所指向的地址的内容是不可修改的。

指针常量，它首先是一个常量，然后才是一个指针。指针常量不能修改指针所指向的地址，一旦初始化，地址就固定了，不能对它进行移动操作。如果对指针常量进行自增操作，系统会提示出错。但是注意，指针常量的内容是可以替换的，这和常量指针是完全不同的概念。总之，指针常量是不可改变地址的指针，但是可以对它所指向的内容进行修改。

答案

常量指针是指向常量的指针，它所指向的地址的内容是不可修改的。

指针常量就是指针的常量，它是不可改变地址的指针，但是可以对它指向的内容进行修改。

面试例题 16：分析代码回答问题——指出下列几种指针的区别。

考点：const 关键字在指针声明时的作用。

出现频率：★★★★★

下述 4 个指针有什么区别？

```
1    char* const p1;
2    char const* p2;
3    const char* p3;
4    const char * const p4;
```

解析

const 位于*号的左侧，则 const 是用来修饰指针所指向的变量，即指针指向常量；如果 const 位于*号的右侧，const 就是修饰指针本身，即指针本身是常量。因此，p1 指针本身是常量，但它指向的内容可以被修改。p2 和 p3 都是指针所指向的内容为常量。p4 则表示它指针本身是常量，并且它指向的内容也不可被修改。

答案

p1 是指针常量，它本身不能被修改，但是指向的内容可以被修改。

p2 和 p3 是常量指针，它们本身可以被修改，但是指向的内容不可以被修改。

p4 是常量指针，并且指向的内容也不可以被修改。

面试例题 17：找错——常量指针和指针常量的作用。

考点：常量指针和指针常量的作用。

出现频率：★★★★

```
1    #include <stdio.h>
2
3    int main()
4    {
5        const char* node1 = "abc";
6        char* const node2 = "abc";
7
8        node1[2] = 'k';
9        *node1[2] = 'k';
10       *node1 = "xyz";
11       node1 = "xyz";
12
13       node2[2] = 'k';
14       *node2[2] = 'k';
15       *node2 = "xyz";
16       node2 = "xyz";
17
18           return 0;
19   }
```

解析

上面的代码中，node1 和 node2 分别是常量指针和指针常量，并且在初始化时都指向了常量字符串"abc"。因此，它们对指向内存的修改都是非法的，如果是 node1 操作，会出现编译错误，而 node2 会出现运行错误。

答案

代码第 8、9、10、14、16 行出现编译错误。

代码 11 行正确。

代码第 13、15 行出现运行时错误。

6.2.4 C++中 this 指针

在 C++中，对象的 this 指针并不是对象本身的一部分，不会影响 sizeof（对象）的结果。This 指针的作用域是类内部，在类的非静态成员函数中访问类的非静态成员时，编译器会自动将对象地址作为一个隐含参数传递给函数。也就是说，即使没有 this 指针，编译器在编译的时候也是加上 this 指针，它作为非静态成员函数的隐含形参，对各成员的访问均通过 this 进行。例如调用：

```
date.SetMonth(9);
```

this 帮助完成下面代码的转换：

```
SetMonth(&date, 9);
```

this 指针的使用情况说明如下。

一种情况是在类的非静态成员函数中返回类对象本身的时候，直接使用 return *this；另外一种情况是当参数与成员变量名相同时用来区分二者，如 this->n = n。

面试例题 18：this 指针的正确叙述。

考点：this 指针的基本概念。

出现频率：★★★

下列关于 this 指针的叙述中，正确的是：

A. 任何与类相关的函数都有 this 指针。

B. 类的成员函数都有 this 指针。

C. 类的友元函数都有 this 指针。

D. 类的非静态成员函数才有 this 指针。

解析

- A 错误。类的非静态成员函数属于类的对象，含有 this 指针。而类的 static 函数属于类本身，不含 this 指针。
- B 错误。类的非静态成员函数属于类的对象，含有 this 指针。而类的 static 函数属于类本身，不含 this 指针。
- C 错误。友元函数是非成员函数，所以它无法通过 this 指针进行复制。

- D 正确。

面试例题 19：分析代码写结果——this 指针。

考点：this 指针的使用。

出现频率：★★★

下面的代码输出结果是什么？如果取消代码 14 行的注释，输出又是什么？

```
1   #include <iostream>
2   using namespace std;
3
4   class MyClass
5   {
6   public:
7       int data;
8       MyClass(int data)
9       {
10          this->data = data;
11      }
12      void print()
13      {
14          //cout << data << endl;
15          cout << "hello!" << endl;
16      }
17  };
18
19
20  int main()
21  {
22      MyClass* pMyClass;
23      pMyClass = new MyClass(1);
24      pMyClass->print();
25      pMyClass[0].print();
26      pMyClass[1].print();
27      pMyClass[10000000].print();
28
29      return 0;
30  }
```

解析

对于类成员函数而言，并不是一个对象对应一个单独的成员函数体，而是类的所有对象共用一个成员函数体，程序被编译后，此成员函数地址即已确定。调用类成员函数时，会将当前对象的 this 指针传递给成员函数。一个类的成员函数体只有一个，而成员函数之所以能把属于此类的各个对象的数据区别开，是因为每次执行类成员函数时，都会把当前对象的 this 指针（也即对象首地址）传入成员函数，函数体内所有对类数据成员的访问，都会转化为“this->数据成

员”的方式。

当 print 函数里没有任何访问对象的数据成员时，此时传进来对象的 this 指针实际上是没有任何用处的，这样的函数，其特征与全局函数没有太大区别。如果取消 14 行的注释，print 函数要访问类的数据成员 data，而程序只构造了一个 MyClass，显然，下标“1”和下标“10000000”的 MyClass 对象根本不存在，那么对它们的数据成员访问也显然是非法的。

答案

注释代码第 14 行时的输出：

```
hello!
hello!
hello!
hello!
```

取消代码第 14 行注释后的输出：

```
1
hello!
1
hello!
-33686019
hello!
段错误
```

6.3 指针数组与数组指针

指针数组与数组指针是两个不同的概念，它们区别如下。

- 指针的数组，即数组的元素是指针形成。
- 数组的指针，即指向数组的指针。

面试例题 20：指针数组与数组指针的区别是什么？

考点：指针数组与数组指针的区别。

出现频率：★★★★★

解析

指针数组指一个数组里存放的都是同一个类型的指针，例如：

```
int * a[10];
```

数组 a 存放了 10 个 int *型变量，由于它是一个数组，已经在栈区分配了 10 个(int *)的空间，在 32 位机上是 40 字节，每个空间都可以存放一个 int 型变量的地址，这时候可以初始化这个数组的每一个元素。

数组指针是指向一维或者多维数组的指针，例如：

```
int* b=new int[10];
```

指针 b 指向含有 10 个整型数据的一维数组。

注意：这个时候释放空间一定要 delete []，否则会造成内存泄漏。

```
1    #include <iostream>
2    using namespace std;
3
4    int main()
5    {
6        int x1[4] = {1, 2, 3, 4};
7        int x2[2] = {5, 6};
8        int x3[3] = {7, 8, 9};
9        int*a[2];
10       int*b = x1;
11       int i = 0;
12
13       a[0] = x2;
14       a[1] = x3;
15
16       cout << "输出 a[0]: ";
17       for(i = 0; i < sizeof(x2) / sizeof(int); i++)
18       {
19           cout << a[0][i] << " ";
20       }
21       cout << endl;
22
23       cout << "输出 a[1]: ";
24       for(i = 0; i < sizeof(x3) / sizeof(int); i++)
25       {
26           cout << a[1][i] << " ";
27       }
28       cout << endl;
29
30       cout << "输出 b: ";
31       for (i = 0; i < sizeof(x1) / sizeof(int); i++)
32       {
33           cout << b[i] << " ";
34       }
35       cout << endl;
36
37       return 0;
38   }
```

程序中有指针数组 a 和数组指针 b。a 的两个指针元素分别指向数组 x2 和 x3，数组指针 b 指向数组 x1。输出如下：

```
输出 a[0]: 5 6
输出 a[1]: 7 8 9
输出 b: 1 2 3 4
```

答案

指针数组表示一个数组，并且数组中的每一个元素都是指针类型。

数组指针表示一个指针，并且是指向数组的指针。

面试例题 21：找错——指针数组和数组指针的使用。

考点：指针数组和数组指针的使用。

出现频率：★★★★

```
1    #include <stdio.h>
2
3    int main()
4    {
5        char *str[]={"Welcome","to","Fortemedia","Nanjing"};
6        char **p = str + 1;
7        str[0] = (*p++) + 2;
8        str[1] = *(p+1);
9        str[2] = p[1] + 3;
10       str[3] = p[0] + (str[2] - str[1]);
11       printf("%s\n", str[0]);
12       printf("%s\n", str[1]);
13       printf("%s\n", str[2]);
14       printf("%s\n", str[3]);
15
16       return 0;
17   }
```

解析

本题的每次执行结果都依赖于上一语句执行的情况，并出现了一个语句同时修改 str 和 p 的值的情况。

代码第 5 行结束时，str 是下列数组的第 1 个值。

- ❑ 第 1 个字符串的首地址的存放地址，标记为 A，其内容为“Welcome”。
- ❑ 第 2 个字符串的首地址的存放地址，标记为 B，其内容为“to”。
- ❑ 第 3 个字符串的首地址的存放地址，标记为 C，其内容为“Fortemedia”。
- ❑ 第 4 个字符串的首地址的存放地址，标记为 D，其内容为“Nanjing”。

代码第 6 行结束时，p 指向 B。

代码第 7 行结束时，p 指向 C。此时 str[0]指向第四个字符串“Nanjing”后面的元素，因此其内容为空。

代码第 8 行结束时，p 没有移动，str[1]指向 p 后一个元素的地址，即 D。

代码第 9 行，此时 p[1]指向 D。p[1] + 3 即指向字符串的元素的第 4 个元素，即“j”字符。此行执行之后，str[2]等于“j”的地址。

代码第 10 行，由第 8 行和第 9 行可知 str[2] – str[1]等于 3，而 p[0]指向“j”的地址。因此

str[4]指向“Nanjing”字符串中的最后一个字符“g”的地址。

答案

```
(空)
Nanjing
jing
g
```

6.4 函数指针与指针函数

函数指针与指针函数也是两个容易混淆的概念，尤其是当它们一起出现时候。因此，许多公司的笔试或者面试题中出现了这两个概念的辨析，以此考查应聘者是否有扎实的基础。大家只需要记住一点：函数指针是函数，指针函数是指针。

面试例题22：函数指针与指针函数的区别。

考点：函数指针与指针函数的区别。

出现频率：★★★★★

解析

指针函数是指带指针的函数，即本质是一个函数，并且返回的是某一类型的指针。其定义如下：

```
返回类型标识符 *返回名称（形式参数表）{ 函数体 }
```

事实上，每一个函数，即使它不带有返回类型的指针，它也有一个入口地址，而该地址相当于一个指针。比如函数返回一个整型值，实际上相当于返回一个指针变量值，不过这时的变量是函数本身而已，即整个函数相当于一个“变量”。

函数指针是指向函数的指针变量，因而它首先是指针变量，只不过该指针变量指向函数。有了指向函数的指针变量后，便以可以调用函数，如同用指针变量可以引用其他类型变量一样。

请看下面这个例子程序：

```
1    #include <iostream>
2    using namespace std;
3
4    int max(int x,int y)
5    {
6        return(x > y? x: y);
7    };
8
9    float* find(float* p, int x)
10   {
11       return(p+x);
12   };
13
```

```
14  int main()
15  {
16      float score[] = {10, 20, 30, 40};
17      int (*p)(int, int);
18      float* q = find(score+1, 1);
19      int a;
20
21      p = max;
22      a=(*p)(1, 2);
23
24      cout << "a = " << a << endl;
25      cout << "*q = " << *q << endl;
26
27      return 0;
28  }
```

函数 find()被定义为指针函数，指针 p 被定义为函数指针类型。main 函数调用 find()函数时，将数组中第 2 个元素的地址和偏移量 1 传入，返回的应该是数组中第 3 个元素的地址。对于指针 p，在代码第 21 行赋为 max()函数的地址，因此在代码第 22 行使用指针 p 就可以调用 max()函数。输出如下：

a = 2

*q = 30

答案

指针函数是返回指针类型的函数。

函数指针是指向函数地址的指针。

面试例题 23：数组指针与函数指针的定义。

考点：数组指针与函数指针的定义。

出现频率：★★★

定义下面的几种类型变量。

（1）含有十个元素的指针数组

（2）数组指针

（3）函数指针

（4）指向函数的指针数组

答案

（1）int*a[10];

（2）int*a = new int[10];

（3）void (*fn)(int, int);

（4）int (*fnArray[10])(int, int);

面试例题 24：各种指针的定义。

考点：各种指针的定义。

出现频率：★★★

写出函数指针、函数返回指针、const 指针、指向 const 的指针、指向 const 的 const 指针。

答案

函数指针：void (*f)(int, int)，其中 f 是指向 void max(int x, int y)类型的函数指针。

函数返回指针：int *fn()，其中 fn 是返回 int 指针类型的函数。

const 指针：const int*p，其中 p 是一个指向 const 的指针，指向一个常量。

指向 const 的指针：int* const q，其中 q 是一个 const 指针。

指向 const 的 const 指针：const int* const ptr，其中 ptr 是指向 const 的 const 指针。

面试例题 25：代码改错——函数指针的使用。

考点：函数指针的使用。

出现频率：★★★★

下面的程序要求打印出 3 个数的最大者，但是程序本身存在问题，请指出。

```
1   #include <iostream>
2   using namespace std;
3
4   int max(int x, int y)
5   {
6       return x > y? x:y;
7   }
8
9   int main()
10  {
11      int*p;
12      int a, b, c;
13      int result;
14      int max(x, y);
15
16      p = max;
17      cout << "Please input three integer " << endl;
18      cin >> a >> b >> c;
19      result = (*p)((*p)(a, b), c);
20      cout << "result = " << result << endl;
21
22      return 0;
23  }
```

解析

这道程序题中函数指针的使用存在错误，如下所示。

- 代码第 14 行，声明 max 函数的方法错误。
- 代码第 16 行，p 指向 max 函数地址，会出现指针不能转换的错误。p 被声明为一个 int *类型的指针，但是 max 地址却为(int *)(int, int)类型。

答案

正确的代码如下：

```
1   #include <iostream>
2   using namespace std;
3
4   int max(int x, int y)
5   {
6       return x > y? x:y;
7   }
8
9   int main()
10  {
11      int (*p)(int, int);               //改正
12      int a, b, c;
13      int result;
14      int max(int, int);                //改正
15
16      p = &max;
17      cout << "Please input three integer " << endl;
18      cin >> a >> b >> c;
19      result = (*p)((*p)(a, b), c);
20      cout << "result = " << result << endl;
21
22      return 0;
23  }
```

面试例题 26：分析代码写结果——函数指针的使用。

考点：函数指针的使用。

出现频率：★★★★

```
1   #include <stdio.h>
2   int add1(int a1,int b1);
3   int add2(int a2,int b2);
4   int main(int argc,char* argv[])
5   {
6       int numa1=1,numb1=2;
7       int numa2=2,numb2=3;
8       int (*op[2])(int a,int b);
9       op[0]=add1;
10      op[1]=add2;
11      printf("%d %d\n",op[0](numa1,numb1),op[1](numa2,numb2));
```

```
12        getchar();
13
14        return 0;
15    }
16
17    int add1(int a1,int b1)
18    {
19        return ( a1+b1 ) ;
20    }
21
22    int add2(int a2,int b2)
23    {
24        return ( a2+b2 ) ;
25    }
```

解析

代码第8行，定义了一个函数指针数组op，它含有两个指针元素，在代码第9行和代码第10行把这两个元素分别指向add1和add2两个函数地址。最后在代码11行打印出使用函数指针调用add1和add2这两个函数返回的结果。

答案

```
3 5
```

面试例题27：typedef用于函数指针定义。

考点：函数指针定义中typedef的作用。

出现频率：★★★

下面的定义有什么作用？

```
typedef int (*pfun)(int x,int y);
```

解析

pfun是一个使用typedef自定义的数据类型。它表示一个函数指针，其参数有两个，都是int类型，返回值也是int类型。可以按如下步骤使用：

```
1    typedef int (*pfun)(int x,int y);
2    int fun(int x, int y);
3    pfun p = fun;
4    int ret = p(2, 3);
```

简单说明如下。

- 代码第1行定义了pfun类型，表示一个函数指针类型。
- 代码第2行定义了一个函数。
- 代码第3行定义了一个pfun类型的函数指针p，并赋给它fun的地址。
- 代码第4行调用p(2, 3)，实现fun(2, 3)的调用功能。

答案

定义了一个函数指针类型，表示带2个int参数指向返回值为int的函数指针类型。可以用

这种类型来定义函数指针，从而调用相同类型的函数。

6.5 野 指 针

野指针是指程序员或操作者不能控制的指针。当程序里定义了一个指针而又没有给这个指针一个具体地址指向时，这个指针会随意地指向一个地址，这样的指针就是一个野指针。如果这个地址后面的内存空间没有什么重要的数据则不会造成不严重后果，但是一旦里面存放了有用的数据，那么这些数据随时都有被野指针存取的危险，如果这样，数据就会被破坏，程序也会崩溃。因此在程序里禁止野指针的存在。

面试例题 28：什么是“野指针”？

考点：“野指针”是什么，有什么作用。

出现频率：★★★★

解析

“野指针”不是NULL指针，是指向“垃圾”内存的指针。人们一般不会错用NULL指针，但是“野指针”是很难判断出是很危险的，而且if语句对它不起作用。“野指针”的成因主要有两种。

（1）指针变量没有初始化。任何指针变量刚被创建时不会自动成为NULL指针，它的默认值是随机的，所以，指针变量在创建的同时应当初始化，要么将指针设置为NULL，要么指向合法的内存。

（2）指针p被free或者delete之后，没有置为NULL。

答案

“野指针”不是NULL指针，而是指向“垃圾”内存的指针，其主要成因是：指针变量没有被初始化，或指针p被free或者delete之后，没有置为NULL。

面试例题 29：分析代码查错——“野指针”的危害。

考点：“野指针”的危害。

出现频率：★★★★

下面的程序片段有什么重大的bug?

```
1    short*bufptr;
2    short bufarray[20];
3    short var=0x20;
4    *bufptr = var;
5    bufarry[0] = var;
```

解析

- ❑ 代码第1行正确。声明了一个short *类型的指针，但是没有对它初始化。

- 代码第2行正确。声明了一个含有20个元素的数组，每个元素都是short类型。
- 代码第3行正确。声明了short类型的变量var，并且把它初始化为0x20。
- 代码第4行错误。因为bufptr没有被初始化，是个“野指针”，因此将bufptr指针指向的内容赋为var变量的值是十分危险的，会导致程序崩溃。为了杜绝这种错误，可以对bufptr进行初始化。代码第1行改为：

```
short *bufptr = (short *)malloc(sizeof(short));
```

- 代码第5行正确。把变量var的值赋给bufarray的第一个元素。

答案

代码第4行存在重大bug，bufptr是“野指针”，会导致程序运行崩溃。

面试例题30：有了malloc/free，为什么还要new/delete？

考点：malloc/free和new/delete的区别。

出现频率：★★★★

解析

malloc与free是C++/C语言的标准库函数，new/delete是C++的运算符。它们都可用于申请动态内存和释放内存。

对于非内部数据类型的对象而言，仅用maloc/free无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于malloc/free是库函数而不是运算符，不在编译器控制权限之内，因此不能把执行构造函数和析构函数的任务强加于malloc/free函数。

C++语言需要能完成动态内存分配和初始化工作的运算符，即new运算符，以及能完成清理与释放内存工作的运算符，即delete运算符。注意new/delete运算符不是库函数。请看下面的例子：

```
1    #include <iostream>
2    using namespace std;
3
4    class Obj
5    {
6    public:
7        Obj(void)
8        {
9            cout << "Initialization" << endl;
10       }
11       ~Obj(void)
12       {
13           cout << "Destroy" << endl;
14       }
15   };
16
```

```
17    void UseMallocFree(void)
18    {
19        cout << "in UseMallocFree()..." << endl;
20        Obj*a = (Obj*)malloc(sizeof(Obj));
21        free(a);
22    }
23
24    void UseNewDelete(void)
25    {
26        cout << "in UseNewDelete()..." << endl;
27        Obj*a = new Obj;
28        delete a;
29    }
30
31    int main()
32    {
33        UseMallocFree();
34        UseNewDelete();
35
36        return 0;
37    }
```

类Obj只有构造函数和析构函数，这两个成员函数分别打印一句话。函数UseMallocFree()调用malloc/free函数申请和释放堆内存；函数UseNewDelete ()调用new/delete运算符申请和释放堆内存。可以看出函数UseMallocFree()执行时，类Obj的构造函数和析构函数都不会被调用，而函数UseNewDelete ()执行时，类Obj的构造函数和析构函数会被调用。执行结果如下：

```
in UseMallocFree()...
in UseNewDelete()...
Initialization
Destroy
```

答案

对于非内部数据类型的对象而言，对象在消亡之前要自动执行析构函数。由于malloc/free函数是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于malloc/free函数，因此只能使用new/delete运算符。

面试例题31：程序改错——指针的初始化。

考点："野指针"必须初始化为NULL。

出现频率：★★★

```
1    #include <stdio.h>
2    #include <malloc.h>
3
4    struct Tag_Node
```

```
5    {
6         struct Tag_Node* left;
7         struct Tag_Node* right;
9         int value;
10   };
11
12   typedef struct Tag_Node TNode;
13
14   TNode* root = NULL;
15
16   void append(int N);
17
18   int main()
19   {
20        append(63);
21        append(45);
22        append(32);
23        append(77);
24        append(96);
25        append(21);
26        append(17);
27        print();
28        return 0;
     }
29
30   void append(int N)
31   {
32        TNode* NewNode = (TNode *)malloc(sizeof(TNode));
33        NewNode->value = N;
34
35        if(root == NULL)
36        {
37             root = NewNode;
38             return;
39        }
40        else
41        {
42             TNode* temp;
43             temp=root;
44
45             while((N >= temp->value && temp->left != NULL) ||
46                  (N < temp->value && temp->right != NULL))
47             {
48                  while(N >= temp->value && temp->left != NULL)
49                       temp = temp->left;
50                  while(N < temp->value && temp->right != NULL)
```

```
51                      temp = temp->right;
52              }
53              if(N >= temp->value)
54                  temp->left = NewNode;
55              else
56                  temp->right = NewNode;
57              return;
58          }
59      }
```

解析

TNode是结构体类型，它有left和right两个成员指针，分别代表链接左右两个元素，value成员表示元素节点的数据。在append函数中，数据从左到右按降序排列。因此在代码第45、46、48和50行中使用while循环来查找合适的位置。在这4行都采用temp的指针left或指针right与NULL进行判断，然而对堆中分配的内存只做成员value的初始化（代码第33行），没有把left和right初始化为NULL，因此指针left和指针right与NULL进行判断不起作用。从而程序会对野指针指向的地址进行赋值，从而导致程序崩溃。

改正后的代码如下：

```
1   #include <stdio.h>
2   #include <malloc.h>
3   
4   struct Tag_Node
5   {
6       struct Tag_Node* left;
7       struct Tag_Node* right;
8       int value;
9   };
10  typedef struct Tag_Node TNode;
11  
12  TNode* root = NULL;
13  
14  void append(int N);
15  void print();
16  
17  int main()
18  {
19      append(63);
20      append(45);
21      append(32);
22      append(77);
23      append(96);
24      append(21);
25      append(17);
26      printf("head: %d\n", root->value);
```

```
27      print();                                  //打印链表所有元素
28  }
29
30  void append(int N)
31  {
32      TNode* NewNode = (TNode*)malloc(sizeof(TNode));
33      NewNode->value = N;
34      NewNode->left = NULL;                     //初始化 left
35      NewNode->right = NULL;                    //初始化 right
36
37      if(root == NULL)
38      {
39          root = NewNode;
40          return;
41      }
42      else
43      {
44          TNode* temp;
45          temp=root;
46
47          while((N >= temp->value && temp->left != NULL) ||
48              (N < temp->value && temp->right != NULL))
49          {
50              while(N >= temp->value && temp->left != NULL)
51                  temp = temp->left;
52              while(N < temp->value && temp->right != NULL)
53                  temp = temp->right;
54          }
55          if(N >= temp->value)
56          {
57              temp->left = NewNode;
58              NewNode->right = temp;            //形成双向链表
59          }
60          else
61          {
62              temp->right = NewNode;
63              NewNode->left = temp;             //形成双向链表
64          }
65          return;
66      }
67  }
68
69  void print()
70  {
71      TNode* leftside = NULL;
```

```
72
73          if (root == NULL)
74          {
75              printf("There is not any element1");
76              return;
77          }
78
79          leftside = root->left;
80
81          while(1)
82          {
83              if (leftside->left == NULL)
84              {
85                  break;
86              }
87              leftside = leftside->left;
88          }
89
90          while(leftside != NULL)
91          {
92              printf("%d ", leftside->value);
93              leftside = leftside->right;
94          }
95      }
```

在代码第 34 行和第 35 行中添加了成员指针 left 和 right 的初始化，这样就杜绝了野指针的产生。代码第 58 行、第 63 行的目的是使链表为双向链表。这样在遍历链表时就会比较方便。print 函数从左到右打印链表中所有元素的 value 成员。执行结果如下：

```
head: 63
96 77 63 45 32 21 17
```

可以看出，root 节点是第 1 个插入到链表的节点，其数据值为 63。链表按照从左到右降序排列。

答案

没有对新增加的节点成员指针 left 和 right 作初始化，它们都是野指针，在随后与 NULL 比较时不起判断的作用。最终由于对野指针指向的内存块赋值导致程序崩溃。

6.6 动态内存的传递

C/C++编程时，会多次与内存管理打交道，因此掌握内存分配、传递、赋值与释放等知识十分必要。

面试例题 32：各种内存分配和释放的函数的联系和区别。

考点：各种标准内存分配函数的使用。

出现频率：★★★

解析

C 语言的标准内存分配函数有 malloc、calloc、realloc、free 等。

malloc 与 calloc 的区别如下。

malloc 调用形式为(类型*)malloc(size)。在内存的动态存储区中分配长度为“size”字节的连续区域，返回该区域的首地址，此时内存中的值没有初始化，是一个随机数。

calloc 调用形式为(类型*)calloc(n，size)。在内存的动态存储区中分配 n 块长度为“size”字节的连续区域，返回首地址，此时内存中的值都被初始化为 0。

realloc 调用形式为；(类型*)realloc(*ptr，size)。将 ptr 内存大小增大到“size”，新增加的内存块没有初始化。

free 的调用形式为 free(void*ptr)：释放 ptr 所指向的内存空间。

C++中用 new 和 delete 函数，调用类的构造函数和析构函数。

面试例题 33：程序找错——动态内存的传递。

考点：动态内存的传递。

出现频率：★★★★

```
1   #include <iostream>
2   using namespace std;
3
4   class Base
5   {
6   private:
7       char*name;
8   public:
9       Base(char * className)
10      {
11          name = new char[strlen(className)];
12          strcpy(name, className);
13      }
14      ~Base()
15      {
16          delete name;
17      }
18      char *copyName()
19      {
20          char newname[10] = "";
```

```
21            strcpy(newname, name);
22
23            return newname;
24        }
25        char *getName()
26        {
27            return name;
28        }
29    };
30    class Subclass : public Base
31    {
32    public:
33        Subclass(char * className) : Base(className)
34        {
35        }
36    };
37
38    int main()
39    {
40        Base *pBase = new Subclass("test");
41        printf("name: %s\n", pBase->getName());
42        printf("new name: %s\n", pBase->copyName());
43
44        return 0;
45    }
```

解析

程序有 Base 和 Subclass 两个类，其中 Subclass 是 Base 的子类。Subclass 被用来向类 Base 的构造函数传递字符串参数，从而为 Base 的私有成员赋值。

代码第 11 行，在类 Base 的构造函数中用 new 运算符分配堆内存。其内存是传入的字符串长度大小。这里有个错误，由于字符串是是以"\0"作为结束符的，应该多分配一字节存放"\0"。

类 Base 的成员函数 copyName()，返回其数组的地址。由于数组处于栈中，当 copyName 调用结束后，栈就会被销毁，这里返回堆内存地址。

答案

代码第 11 行改为：

```
name = new char[strlen(className) + 1];
```

代码第 20 行改为：

```
char *newname = new char[strlen(name) + 1];
```

修改后正确的输出为：

```
name: test
new name: test
```

面试例题 34：程序分析——动态内存的传递。

考点：动态内存的传递。

出现频率：★★★★

```
1   #include <iostream>
2   using namespace std;
3
4   void GetMemory(char*p, int num)
5   {
6       p = (char*)malloc(sizeof(char) *num);
7   };
8
9   int main(void)
10  {
11      char*str = NULL;
12
13      GetMemory(str, 10);
14      strcpy(str, "hello");
15
16      return 0;
17  }
```

解析

GetMemory()函数存有错误。因为编译器总是为函数的每个参数制作临时的变量。所以GetMemory()函数体内的p是main函数中str变量在GetMemory()函数中的一个备份。虽然在代码第6行，p申请了堆内存，但是返回到main函数时，str还是NULL，而不指向堆内存，因此在代码第14行调用strcpy时会导致程序崩溃。

GetMemory()函数并不能做任何有用的事情。还要注意，由于GetMemory()函数返回时不能获得堆内存的地址，堆内存就不能被继续引用，也就得不到释放，因此调用一次GetMemory()函数会产生num字节的内存泄漏。

采用以下3种方法来解决动态内存不能传递的问题。

- 在C中，采用指向指针的指针解决这个问题，可以把str的地址传给GetMemory()函数。
- 在C++中，采用传递str指针的引用。
- 使用函数返回值来传递动态内存。

看下面的示例代码：

```
1   #include <iostream>
2   using namespace std;
3
4   void GetMemory(char* p, int num)
5   {
6       p = (char*)malloc(sizeof(char)* num);
7   };
```

```
8
9    void GetMemory2(char**p, int num)
10   {
11       *p = (char*)malloc(sizeof(char) *num);
12   };
13
14   void GetMemory3(char* &p, int num)
15   {
16       p = (char*)malloc(sizeof(char)* num);
17   };
18
19   char *GetMemory4(int num)
20   {
21       char* p = (char*)malloc(sizeof(char)*num);
22
23       return p;
24   }
25
26   int main(void)
27   {
28       char* str1 = NULL;
29       char* str2 = NULL;
30       char* str3 = NULL;
31       char* str4 = NULL;
32
33
34       GetMemory2(&str2, 20);
35       GetMemory3(str3, 20);
36       str4 = GetMemory4(20);
37
38       strcpy(str2, "GetMemory 2");
39       strcpy(str3, "GetMemory 3");
40       strcpy(str4, "GetMemory 4");
41
42       cout << "str1 == NULL? " << (str1 == NULL? "yes":"no") << endl;
43       cout << "str2:" << str2 << endl;
44       cout << "str3:" << str3 << endl;
45       cout << "str4:" << str4 << endl;
46
47       free(str2);
48       free(str3);
49       free(str4);
50       str2 = NULL;
51       str3 = NULL;
52       str4 = NULL;
```

```
53
54        return 0;
55    }
```

在上面的代码中，GetMemory2()函数采用二维指针作参数传递；GetMemory3()函数采用指针的引用作参数传递；GetMemory4()函数采用返回堆内存指针的方式作参数传递。可以看到这 3 个函数起到相同的作用。

注意代码第 47～52 行，在调用主函数之后把指针 str2、str3 和 str4 指向的堆内存释放并把指针赋为 NULL。不再使用堆内存时，应该把堆内存释放，并把指针赋为 NULL，这样可以避免内存泄漏以及野指针的出现。

程序运行结果如下所示：

```
str1 == NULL? Yes
str2:GetMemory 2
str3:GetMemory 3
str3:GetMemory 4
```

答案

调用“strcpy(str, "hello")”时程序崩溃。因为 GetMemory()函数不能传递动态内存，并且 str 始终都是 NULL。

面试例题 35：比较分析两个代码段的输出——动态内存的传递。

考点：动态内存的传递。

出现频率：★★★★★

程序 1

```
1    char*GetMemory()
2    {
3        char p[] = "hello world";
4        return p;
5    }
6
7    void Test(void)
8    {
9        char*str = NULL;
10       str = GetMemory();
11       printf(str);
12   }
```

程序 2

```
1    void GetMemory(char* p)
2    {
3        p=(char*)malloc(100);
4    }
5
6    void Test(void)
```

```
7    {
8         char* str=NULL;
9         GetMemory(str);
10        strcpy(str,"hello world");
11        printf(str);
12   }
```

解析

程序 1 的 GetMemory()返回的是指向栈内存的指针，该指针的地址不是 NULL，但是当栈退出后，内容不定，有可能会输出乱码。

程序 2 的 GetMemory()没有返回值，这个函数不能传递动态内存。在 Test 函数中 str 变量的值通过参数传值的方式赋值给 GetMemory()的局部变量 p。但是 Test()中的 str 一直为 NULL，所以在代码第 10 行的调用会使程序崩溃。此外，由于在 GetMemory()执行之后没有指针引用堆内存，因此会产生内存泄漏。

答案

程序 1 输出结果可能是乱码。

程序 2 有内存泄漏，在 Test 函数调用 strcpy 时程序崩溃。

面试例题 36：程序查错——"野指针"用于变量值的互换。

考点："野指针"不能用于变量值的互换。

出现频率：★★★

```
1    swap(int* p1, int* p2)
2    {
3         int *p;
4         *p = *p1;
5         *p1 = *p2;
6         *p2 = *p;
7    }
```

解析

在代码第 3 行，声明了一个指针 p，由于没有对 p 初始化，因此 p 是一个野指针，它可能指向系统区。因此在代码第 4 行，对 p 指向的内存区赋值非常危险，会导致程序运行时崩溃。程序应改为：

```
1    swap(int* p1, int* p2)
2    {
3         int p;
4         p = *p1;
5         *p1 = *p2;
6         *p2 = p;
7    }
```

答案

swap 函数内的指针变量 p 没有初始化，是一个野指针，运行时导致程序崩溃。

面试例题37：内存的分配方式有几种？

考点：静态存储区、栈、堆的内存分配。

出现频率：★★★★★

解析

- 从静态存储区域分配内存。程序编译的时候内存已经分配好了，并且在程序的整个运行期间都存在，例如全局变量。
- 在栈上创建。在执行函数时，函数内局部变量的存储单元可以在栈上创建，函数结束时这些存储单元自动被释放。处理器的指定集中有关于栈内存的分配运算，因此效率很高，但是分配的内存容量有限。
- 从堆上分配内存，亦称动态内存分配。程序在运行的时候用 malloc 函数或 new 运算符申请任意大小的内存，程序员要用 free 函数或 delete 运算符释放内存。动态内存使用非常灵活，但问题也很多。

6.7 指针与句柄的区别

很多人以为句柄就是指针，实际上句柄并不是那么简单的概念。应聘时会遇到有关句柄的知识，以考查应聘者是否对 Windows 编程有一定的了解。

面试例题38：什么是句柄？

考点：对于 Windows 句柄的理解。

出现频率：★★★

解析

句柄在 Windows 编程中是一个很重要的概念，在很多程序中都扮演着重要的角色。在 Windows 环境中，句柄是用来标识项目的，这些项目包括：

- 模块（module）。
- 任务（task）。
- 实例（instance）。
- 文件（file）。
- 内存块（block of memory）。
- 菜单（menu）。
- 控制（control）。
- 字体（font）。
- 资源（resource）（包括图标（icon），光标（cursor），字符串（string）等。

- GDI 对象（GDI object）（包括位图（bitmap），画刷（brush）、元文件（metafile））。调色板（palette）、画笔（pen）、区域（region）以及设备描述表（device context）。

Windows 是以虚拟内存为基础的操作系统。在这种操作系统下，Windows 内存管理器在内存中来回移动对象，从而满足各种应用程序的内存需要。对象被移动意味着它的地址变化了。由于地址总是如此变化，所以 Windows 操作系统为各应用程序腾出一些内存储地址，专门用来登记各应用对象在内存中的地址变化，而这些地址（存储单元的位置）本身是不变的。Windows 内存管理器在内存中移动对象位置后，把对象新的地址告知句柄地址来保存。这样只需记住这个句柄地址就可以间接地知道对象在内存中具体的位置。这个地址在对象装载（Load）时由系统分配，当系统卸载时（Unload）又释放给系统。

因此 Windows 程序中并不是用物理地址来标识内存块、文件、任务或动态装入模块的，相反，Windows API 给这些项目分配确定的句柄，并将句柄返回给应用程序，然后通过句柄来进行操作。

在 Windows 编程中会用到大量的句柄，比如：HINSTANCE（实例句柄），HBITMAP（位图句柄），HDC（设备描述表句柄），HICON（图标句柄）等，其中还有一个通用的句柄，就是 HANDLE，如下面的语句所示：

```
HINSTANCE hInstance;
HANDLE hInstance;
```

程序执行顺序是：句柄地址（稳定）→记载对象在内存中的地址→对象在内存中的地址（不稳定）→实际对象。但是程序每次重新启动，系统分配给程序的句柄并不一定还是原来的句柄，而且绝大多数情况下是不一样的。

面试例题 39：指针与句柄有什么区别？

考点：对于 Windows 句柄的理解以及其与一般指针的区别。

出现频率：★★★★

解析

指针对应着数据在内存中的地址，利用指针可以自由地修改数据。Windows 并不希望一般程序修改其内部数据结构，因为这样太不安全。所以 Windows 给每个使用 GlobalAlloc 等函数声明的内存区域指定一个句柄，即指向指针的指针。

句柄和指针都是地址，不同之处如下所示。

- 句柄所指可以是一个复杂的结构，并且可以与系统有关，例如线程的句柄，它可以指向一个类或者结构，而且和系统有很密切的关系，当一个线程由于不可预料的原因而终止时，系统就可以返回它所占用的资料，如 CPU、内存等，反之、句柄中的某一些项是与系统进行交互的。
- 指针也可以指向一个复杂的结构，但是通常是由用户定义的，所以必要的工作要由用户完成，特别删除部分的工作。

第7章 字符串

在C/C++语言中没有专门的字符串变量，通常用字符数组来存放字符串。字符串是以“\0”作为结束符。C/C++提供了丰富的字符串处理函数，下面列出了几个最常用的函数。

- 字符串输出函数puts。
- 字符串输出函数gets。
- 字符串连接函数strcat。
- 字符串复制函数strcpy。
- 测字符串长度函数strlen。

字符串是面试的重点考查部分的相关知识，通过考查字符串的相关知识可以考察程序员的编程规范以及编程习惯。并且其中包括了许多知识点，例如内存越界、指针与数组操作等。许多公司在面试时会要求应聘者写一段复制字符串或字符串子串操作的程序。本章列举了一些与字符串相关的面试题，有些题目要求较高的编程技巧。

7.1 数字与字符串的转化

应聘时经常出现数字与字符串之间转化的问题，面试官通过这类题目来考察应聘者能力，例如是否熟悉常用的库函数，是否了解ASCII码以及是否了解字符串的存储格式等。

7.1.1 数字转化为字符串

面试例题1：使用库函数将数字转换为字符串。

考点：C库函数中数字转换为字符串的使用。

出现频率：★★★

解析

C语言提供了几个标准库函数，可以将任意类型（整型、长整型、浮点型等）的数字转换为字符串，下面列举了各函数的方法及其说明。

- itoa()：将整型值转换为字符串。

- ltoa()：将长整型值转换为字符串。
- ultoa()：将无符号长整型值转换为字符串。
- gcvt()：将浮点型数转换为字符串，取四舍五入。
- ecvt()：将双精度浮点型值转换为字符串，转换结果中不包含十进制小数点。
- fcvt()：指定位数为转换精度，其余同 ecvt()。

还可以使用 sprintf 系列函数把数字转换成字符串，其比 itoa()系列函数运行速度慢。下列程序演示了如何使用 itoa()函数和 gcvt()函数：

```
1   # include <stdio.h>
2   # include <stdlib.h>
3
4   int main ()
5   {
6       int num_int = 435;
7       double num_double = 435.10f;
8       char str_int[30];
9       char str_double[30];
10
11      itoa(num_int, str_int, 10);              //把整数 num_int 转成字符串 str_int
12      gcvt(num_double, 8, str_double);         //把浮点数 num_double 转成字符串 str_double
13
14      printf("str_int: %s\n", str_int);
15      printf("str_double: %s\n", str_double);
16
17      return 0;
18  }
```

程序输出结果：

```
1   str_int: 435
2   str_double: 435.10001
```

- 代码第 11 行中的参数 10 表示按十进制类型进行转换，转换后的结果是“435”，如果按二进制类型进行转换，则结果为“1101110011”。
- 代码第 12 行中的参数 8 表示精确位数，这里得到的结果是“435.10001”。

答案

可以使用 atoi 系列函数把数字转换成字符串。

面试例题 2：不使用库函数将整数转换为字符串。

考点：数字转换为字符串，理解相关 ASCII 码。

出现频率：★★★★

解析

如果不使用 atoi 或 sprintf 等库函数，可以通过把整数的各位上的数字加“0”转换成 char 类型并存到字符数组中。但是要注意，需要采用字符串逆序的方法。如以下程序所示：

```
1   #include <iostream>
2   using namespace std;
3
4   void int2str(int n, char *str)
5   {
6       char buf[10] = "";
7       int i = 0;
8       int len = 0;
9       int temp = n < 0 ? -n: n;                  //temp 为 n 的绝对值
10
11      if (str == NULL)
12      {
13          return;
14      }
15      while(temp)
16      {
17          buf[i++] = (temp % 10) + '0';          //把 temp 的每一位上的数存入 buf
18          temp = temp / 10;
19      }
20
21      len = n < 0 ? ++i: i;                      //如果 n 是负数，则多需要一位来存储负号
22      str[i] = 0;                                //末尾是结束符 0
23      while(1)
24      {
25          i--;
26          if (buf[len-i-1] ==0)
27          {
28              break;
29          }
30          str[i] = buf[len-i-1];                 //把 buf 数组里的字符拷到字符串
31      }
32      if (i == 0 )
33      {
34          str[i] = '-';                          //如果是负数，添加一个负号
35      }
36  }
37
38  int main()
39  {
40      int nNum;
41      char p[10];
42
43      cout << "Please input an integer:";
44      cin >> nNum;
```

```
45          cout << "output: " ;
46          int2str(nNum, p);                          //整型转换成字符串
47          cout<< p << endl;
48
49          return 0;
50      }
```

程序中的 int2str 函数完成了 int 类型到字符串类型的转换。在代码第 46 行对 int2str 函数做了测试。程序的执行结果如下所示：

```
Please input an integer: 1234
Output: 1234
```

如果输入的是个负数，程序执行结果如下所示：

```
Please input an integer: -1234
Output: -1234
```

接下来对 int2str 函数的实现进行分析。

- 代码第 9 行，把参数 n 的绝对值赋给 temp，以后在计算各个位的整数时用 temp，这样保证在负数情况下取余不会出现问题。
- 代码第 11～第 14 行判断 str 的有效性，str 不为 NULL。
- 代码第 15～第 19 行的 while 循环中，将 n 的各个位存放到局部数组 buf 中，存放的顺序与整数顺序相反。例如 n 为整数 123 456，while 循环结束后 buf 应为“654 321”。
- 代码第 21 行计算转换后字符串的长度 len，如果是负数，长度应该再加 1。
- 代码第 22～第 31 行把数组 buf 中的非 0 元素逆向复制到参数 str 指向的内存中，如果 n 是负数，则 str 指向的第一个内存存放负号。

7.1.2 字符串转化为数字

面试例题 3：使用库函数将字符串转换为数字。

考点：C 库函数中字符串转换为数字的使用。

出现频率：★★★★

解析

与上节数字转换为字符串类似，C/C++语言提供了几个标准库函数，可以将字符串转换为任意类型（整型、长整型、浮点型等）。以下列举了各函数的方法及其说明。

- atof()：将字符串转换为双精度浮点型值。
- atoi()：将字符串转换为整型值。
- atol()：将字符串转换为长整型值。
- strtod()：将字符串转换为双精度浮点型值，并报告不能被转换的所有剩余数字。
- strtol()：将字符串转换为长整值，并报告不能被转换的所有剩余数字。
- strtoul()：将字符串转换为无符号长整型值，并报告不能被转换的所有剩余数字。

以下程序演示如何使用 atoi ()函数和 atof ()函数。

```
1   # include <stdio.h>
2   # include <stdlib.h>
3
4   int main ()
5   {
6       int num_int;
7       double num_double;
8       char str_int[30] = "435";             //将要被转换为整型的字符串
9       char str_double[30] = "436.55";       //将要被转换为浮点型的字符串
10
11      num_int = atoi(str_int);              //转换为整型值
12      num_double = atof(str_double);        //转换为浮点型值
13
14      printf("num_int: %d\n", num_int);
15      printf("num_double: %lf\n", num_double);
16
17      return 0;
18  }
```

输出结果：

```
num_int: 435
num_double: 436.550000
```

面试例题 4：不使用库函数将字符串转换为数字。

考点：字符串转换为数字时，对相关 ASCII 码的理解。

出现频率：★★★★

解析

程序代码如下：

```
1   #include <iostream>
2   using namespace std;
3
4   int str2int(const char *str)
5   {
6       int temp = 0;
7       const char *ptr = str;                //ptr 保存 str 字符串开头
8
9       if (*str == '-' || *str == '+')       //如果第一个字符是正负号
10      {                                     //则移到下一个字符
11          str++;
12      }
13      while(*str != 0)
14      {
15          if ((*str < '0') || (*str > '9')) //如果当前字符不是数字
16          {                                 //则退出循环
17              break;
```

```
18              }
19              temp = temp * 10 + (*str - '0');        //如果当前字符是数字则计算数值
20              str++;                                  //移到下一个字符
21          }
22          if (*ptr == '-')                            //如果字符串是以"-"开头，则转换成其相反数
23          {
24              temp = -temp;
25          }
26
27          return temp;
28      }
29
30      int main()
31      {
32          int n = 0;
33          char p[10] = "";
34
35          cin.getline(p, 20);                         //从终端获取一个字符串
36          n = str2int(p);                             //把字符串转换成整型数
37
38          cout << n << endl;
39
40          return 0;
41      }
```

程序执行结果：

```
输入：1234
输出：1234
输入：-1234
输出：-1234
输入：+1234
输出：1234
```

程序中的str2int函数作用是将字符串转换成整数。这个函数的转换过程与例题2中的int2str函数相比更加简单，它只需要做一次while循环（代码第13行）就能把数值大小计算出来，如果结果是负数，就加一个负号。

7.2 字符串与数组

字符串一般是用字符数组的方式存储，例如下面的str定义：

```
char str[] = "123456";
```

这里str是一个字符数组，它存放了一个字符串"123456"，由于字符串还有一个结束符"\0"，所以此数组的长度为7而不是6。

7.2.1 strcpy 函数与 memcpy 函数

strcpy 和 memcpy 都是标准 C 库函数，它们有下面的特点。

- strcpy 提供了字符串的复制。即 strcpy 只用于字符串复制，并且它不仅复制字符串内容之外，还会复制字符串的结束符。
- memcpy 提供了一般内存的复制。即 memcpy 对于需要复制的内容没有限制，因此用途更广。

面试例题 5：编程实现 strcpy 函数。

考点：字符串复制的实现。

出现频率：★★★★

已知 strcpy 函数的原型是：

```
char * strcpy(char * strDest,const char * strSrc);
```

要求如下。

（1）不调用库函数，实现 strcpy 函数；

（2）解释为什么要返回 char *。

解析

程序代码如下：

```
1   #include <stdio.h>
2
3   char * strcpy(char * strDest, const char * strSrc)        //实现 strSrc 到 strDest 的复制
4   {
5       if ((strDest == NULL) || (strSrc == NULL))             //判断参数 strDest 和 strSrc 的有效性
6       {
7           return NULL;
8       }
9       char *strDestCopy = strDest;                           //保存目标字符串的首地址
10      while ((*strDest++ = *strSrc++)!='\0');                //把 strSrc 字符串的内容复制到 strDest 下
11
12      return strDestCopy;
13  }
14
15  int getStrLen(const char *strSrc)                          //实现获取 strSrc 字符串的长度
16  {
17      int len = 0; //保存长度
18      while(*strSrc++ != '\0')                               //循环直到遇见结束符'\0'为止
19      {
20          len++;
21      }
22
23      return len;
```

```
24  };
25
26  int main()
27  {
28      char strSrc[] = "Hello World!";                  //要被复制的源字符串
29      char strDest[20];                                //要复制到的目的字符数组
30      int len = 0;                                     //保存目的字符数组中字符串的长度
31
32      len = getStrLen(strcpy(strDest, strSrc));        //链式表达式，先复制后计算长度
33      printf("strDest: %s\n", strDest);
34      printf("Length of strDest: %d\n", len);
35
36      return 0;
37  }
```

（1）strcpy 函数的实现说明。

- 代码第 5～第 7 行判断传入的参数 strDest 和 strSrc 是否为 NULL，如果是则返回 NULL。
- 代码第 9 行把 strDest 的值保存到 strDestCopy 指针中。
- 代码第 10 行对 strSrc 和 strDest 两个指针进行循环移动，并不断复制 strSrc 内存的值到 strDest 内存中。
- 由于已经保存了 strDest 指针的值，因此这里只需返回 strDestCopy 的值，而函数调用完后返回的就是 strDest 的值。

（2）strcpy 函数返回 char *类型的原因是为了能使用链式表达式。首先调用 strcpy 使得 strDest 指针复制 strSrc 的内存数据，然后调用 getStrLen 函数获取 strDest 字符串的长度。这样不仅调用方便，而且程序结构简洁明了。程序的输出结果如下：

```
strDest: Hello World!
Length of strDest: 12
```

面试例题 6：编程实现 memcpy 函数。

考点：内存复制的实现。

出现频率：★★★★

答案

程序代码如下所示：

```
1   #include <stdio.h>
2   #include <assert.h>
3
4   void *memcpy2(void *memTo, const void *memFrom, size_t size)
5   {
6       assert((memTo != NULL) && (memFrom != NULL));    //memTo 和 memFrom 必须有效
7       char *tempFrom = (char *)memFrom;                //保存 memFrom 首地址
```

```
8       char *tempTo = (char *)memTo;                      //保存 memTo 首地址
9
10      while(size -- > 0)                          //循环 size 次，复制 memFrom 的值到 memTo 中
11          *tempTo++ = *tempFrom++ ;
12
13      return memTo;
14  }
15
16  int main()
17  {
18      char strSrc[] = "Hello World!";             //将被复制的字符数组
19      char strDest[20];                           //目的字符数组
20
21      memcpy2(strDest, strSrc, 4);                //复制 strSrc 的前 4 个字符到 strDest 中
22      strDest[4] = '\0';                          //把 strDest 的第 5 个元素赋为结束符'\0'
23      printf("strDest: %s\n", strDest);
24
25      return 0;
26  }
```

memcpy 的实现如下。

与 strcpy 不同，memcpy 用参数 size 决定复制多少个字符（strcpy 遇到结束符“\0”结束）。由于在主程序中只复制了 strSrc 的前 4 个字符（代码第 22 行），程序输出如下：

```
strDest: Hell
```

面试例题 7：strcpy 与 memcpy 的区别。

考点：字符串复制与内存复制之间的区别。

出现频率：★★★★

解析

strcpy 和 memcpy 主要有以下 3 方面的区别。

- 复制的内容不同。strcpy 只能复制字符串，而 memcpy 可以复制任意内容，例如字符数组、整型、结构体、类等。
- 复制的方法不同。strcpy 不需要指定长度，它遇到字符串结束符“\0”便结束。memcpy 则是根据其第 3 个参数决定复制的长度。
- 用途不同。通常在复制字符串时用 strcpy，而需要复制其他类型数据时则一般用 memcpy。

7.2.2 数组越界

C++不会自动检查数组越界，也就是说如果数组越界，程序编译时不会报错，从而在执行时产生非法操作或者得不到正确的结果。因此在使用数组时，一定要在编程中判断是否越界以保证程序的正确性。

面试例题 8：改错——数组越界。

考点：数组越界出现的问题。

出现频率：★★★★

题（1）

```
1   void test1()
2   {
3       char string[10];
4       char* str1 = "0123456789";
5       strcpy(string, str1);
6   }
```

题（2）

```
1   void test2()
2   {
3       char string[10], str1[10];
4       int i;
5       for(i=0; i<10; i++)
6       {
7           str1[i] = 'a';
8       }
9       strcpy(string, str1);
10  }
```

题（3）

```
1   void test3(char* str1)
2   {
3       char string[10];
4       if(strlen(str1) <= 10)
5       {
6           strcpy(string, str1);
7       }
8   }
```

解析

这 3 道题都有数组越界的问题。

- 题（1）中，string 是一个含有 10 个元素的字符数组，str1 指向的字符串长度为 10，在进行 strcpy 调用时，会将 str1 的结束符也复制到 string 数组里，即复制的字符个数为 11，这样会导致 string 出现数组越界。程序不一定会因此而崩溃，但这是一个潜在的危险。解决办法是将 string 的元素个数定义为 11。
- 题（2）中，str1 和 string 都是含有 10 个元素的字符数组，并且 str1 的元素全部被赋为字符“a”，然后再调用 strcpy。这里会出现以下两个问题：一个问题是 str1 表示的字符数组没有以'\0'结束，在随后调用 strcpy 时无法判断什么时候复制结束；另一个问题是 string 的数组长度不够，出现数组越界的现象。解决办法是将 string 和 str1 的元素个数

都定义为11个，并在调用strcpy之前加入一条语句把str1[10]赋为'\0'。

- 题（3）中，if语句使用小于等于“<=”进行比较，如果str1的长度等于10，也会出现数组越界的情况。解决办法是把“<=”换成“<”。

答案

3道题都有数组越界的问题。改正后的程序如下所示。

题（1）

```
1  void test1()
2  {
3      char string[11];                //字符数组长度为11，多分配一个
4      char* str1 = "0123456789";
5      strcpy(string, str1);
6  }
```

题（2）

```
1  void test2()
2  {
3      char string[11], str1[11];      //字符数组长度都为11，均多分配一个
4      int i;
5      for(i=0; i<10; i++)
6      {
7          str1[i] = 'a';
8      }
9      str1[10] = '\0';                //初始化str1为空字符串
10     strcpy(string, str1);
11 }
```

题（3）

```
1  void test3(char*str1)
2  {
3      char string[10];
4      if(strlen(str1) < 10)           //不能用<=
5      {
6          strcpy(string, str1);
7      }
8  }
```

面试例题9：分析程序——数组越界。

考点：不当的循环操作导致数组越界。

出现频率：★★★

下面这个程序执行后会出现什么错误？

```
1  #define MAX 255
2  int main()
3  {
4      unsigned char A[MAX], i;
5
```

```
6         for (i = 0; i <= MAX; i++)
7             A[i] = i;
8     }
```

解析

代码第 6 行的 for 循环中用的是“<=”，当 i=MAX 时数组越界。值得注意：这个程序很容易使人误认为只有数组越界的问题，但只要细心些就能发现，i 是无符号的 char 类型，它的范围是 0～255，所以“i<=MAX”一直都是真，这样会导致无限循环。可以把“i<=MAX”改为“i<MAX”，这样既避免了无限循环又避免了数组越界。

答案

i<=MAX 导致数组越界以及无限循环，应改为 i<MAX。

面试例题 10：分析程序——打印操作可能产生数组越界。

考点：打印操作时可能产生的数组越界问题。

出现频率：★★★

下面这个程序的打印结果是什么？

```
1    #include <stdio.h>
2
3    int main()
4    {
5        int a[5]={0, 1, 2, 3, 4}, *p;
6        p = a;
7        printf("%d\n",*(p + 4*sizeof(int)));
8
9        return 0;
10   }
```

答案

这个程序存在着越界的问题。

代码第 6 行，p 指向 a 的第 1 个元素，所以 p+4 指向 a 的最后一个元素即 4，p + 4 * sizeof(int) 即 p+16，此时指向的是数组 a 的第 17 个元素，显然已经越界了，因此打印结果是个随机数。

7.2.3 其他编程问题

由于字符串非常灵活，因此面试中非常注重字符串操作的考察。本节列出了一些经常出现的字符串考题，例如有关字符串长度检测，查找子串，单向翻转，判断是否回文等的题目。

面试例题 11：编程实现字符串的长度检测。

考点：strcpy 库函数的实现细节。

出现频率：★★★★★

解析

这个题目非常简单，字符串是以'\0'作为结束符，所以只需要做一次遍历就可以了。但是需

要注意的是要尽量把程序写得简单高效。看下面的示例代码：

```
1   #include <stdio.h>
2   #include <assert.h>
3
4   int strlen1(const char* src)
5   {
6       assert( NULL != src);                   //src 必须有效
7       int len = 0;                            //保存 src 的长度
8       while(*src++ != '\0')                   //遇到结束符'\0'退出循环
9           len++;                              //每循环一次 len 加 1
10      return len;
11  }
12
13  int strlen2(const char* src)
14  {
15      assert( NULL != src);                   //src 必须有效
16      const char *temp = src;                 //保存 src 首地址
17      while(*src++ != '\0');                  //遇到结束符'\0'退出循环
18      return (src-temp-1);                    //返回尾部指针与头部指针之差即长度
19  }
20
21  int main()
22  {
23      char p[] = "Hello World!";
24      printf("strlen1 len: %d\n", strlen1(p));   //打印方法 1 得到的结果
25      printf("strlen2 len: %d\n", strlen2(p));   //打印方法 2 得到的结果
26
27      return 0;
28  }
```

strlen1 和 strlen2 这两个函数用来计算字符串长度。它们的区别如下所示。

- strlen1 用局部变量 len 在遍历的时候做自增，然后返回 len。因此每当 while 循环一次，就需要执行两次自增操作。
- strlen2 用局部变量 temp 记录 src 遍历前的位置。while 循环一次只需要一次自增操作，最后返回指针之间的位置差。

当字符串较长的时候 strlen2 比 strlen1 的效率更高。下面是程序的输出结果：

```
strlen1 len: 12
strlen2 len: 12
```

面试例题 12：编程实现字符串中子串的查找。

考点：strstr 库函数的实现细节。

出现频率：★★★★★

编写实现 strstr 即一个函数，即从一个字符串中，查找另一个字符串的位置，如 strstr("12345",

"34")返回值为 2，即在 2 号位置找到字符串 34。

解析

程序如下所示：

```
1   #include <stdio.h>
2   #include <assert.h>
3
4   const char *strstr(const char* src, const char* sub)
5   {
6       const char *bp;
7       const char *sp;
8
9       if (src == NULL || NULL == sub)          //判断 src 与 sub 的有效性
10      {
11          return src;
12      }
13      while (*src) //遍历 src 字符串
14      {
15          bp = src;                            //用于 src 的遍历
16          sp = sub;                            //用于 sub 的遍历
17          do
18          {                                    //遍历 sub 字符串
19              if (!*sp)                        //如果到了 sub 字符串结束符位置
20                  return src;                  //表示找到了 sub 字符串，退出
21          } while (*bp++ == *sp++);
22          src += 1;
23      }
24
25      return NULL;
26  }
27  int main()
28  {
29      char p[] = "12345";
30      char q[] = "34";
31
32      char *r = strstr(p, q);
33      printf("r: %s\n", r);
34
35      return 0;
36  }
```

main 函数中的测试结果为：

```
r: 345
```

代码第 32 行调用 strstr 结束之后，r 指向了数组 p 的第 3 个元素。这里 strstr 函数的方法是循环取 src 的子串与 sub 比较。以本题中的“12345”和“34”为例，比较步骤如下所示。

- "12345"和"34"比较，不满足匹配。
- "2345"和"34"比较，不满足匹配。
- "345"和"34"比较，满足匹配。

面试例题 13：编程实现字符串中各单词的翻转。

考点：字符串相关的综合编程能力。

出现频率：★★★

编写函数，将"I am from Shanghai"倒置为"Shanghai from am I"，即将句子中的单词位置倒置，而不改变单词内部的结构。

解析

第 1 种方法代码如下：

```
1   #include <iostream>
2   using namespace std;
3
4   void RevStr(char *src)
5   {
6       char *start = src, *end = src, *ptr = src;
7
8       while(*ptr++ != '\0')                //遍历字符串
9       {
10          if(*ptr == ' ' || *ptr == '\0')  //找到一个单词
11          {
12              end = ptr - 1;               //end 指向单词末尾
13              while(start < end)
14              swap(*start++, *end--);      //把单词的字母逆置
15
16              start = end = ptr+1;         //指向下一个单词开头
17          }
18      }
19      start = src, end = ptr-2;            //start 指向字符串开头，end 指向字符串末尾
20      while(start < end)
21      {
22          swap(*start++, *end--);          //把整个字符串逆置
23      }
24  }
25
26  int main()
27  {
28      char src[] = "I am from Shanghai";
29      cout << src << "\n";
30      RevStr(src);
31      cout << src << "\n";
```

```
32
33        return 0;
34    }
```

程序输出结果：

```
1    I am from Shanghai
2    Shanghai From am I
```

RevStr 函数有两个转换步骤：代码第 8～第 18 行将 src 中所有的单词进行翻转，其结果是 src 的内容变为“I ma morf iahgnahS”，然后代码第 20～第 23 行再进行全局翻转。

第 2 种方法代码如下：

```
1    #include <iostream>
2    using namespace std;
3
4    void RevStr(char *src)
5    {
6        char *start = src, *end = src, *ptr = src;
7
8        while(*ptr++ != '\0');
9        end = ptr-2;                          //找到字符串末尾
10       while(start < end)
11       {
12           swap(*start++, *end--);           //逆置整个字符串
13       }
14
15       start = src, end = ptr-2;
16       ptr = start;
17       while(*ptr++ != '\0')
18       {
19           if(*ptr == ' ' || *ptr == '\0')   //找到单词
20           {
21               end = ptr - 1;                //end 指向单词末尾
22               while(start < end)
23               swap(*start++, *end--);       //逆置单词
24
25               start = end = ptr+1;          //指向下一个单词开头
26           }
27       }
28   }
29
30   int main()
31   {
32       char src[] = "I am from Shanghai";
33       cout << src << "\n";
34       RevStr(src);
35       cout << src << "\n";
```

```
36
37        return 0;
38    }
```

程序输出结果：

```
I am from Shan ghai
Shanghai from am I
```

RevStr 函数转换步骤与第 1 种方法相反，即代码第 10～第 11 行将 src 进行全局翻转，src 的内容变为“iahgnahS morf ma I”，然后代码第 17～第 27 行再把所有的单词进行翻转。

从上面的代码分析可以看出，两种方法都是采用了两个步骤，即字符串全局翻转和各个单词局部翻转，只是步骤的顺序不同，但是得到的结果都是一致的。

面试例题 14：编程判断字符串是否为回文。

考点：字符串相关的综合编程能力。

出现频率：★★★★

判断一个字符串是不是回文。

解析

根据题目要求，我们可以从一个字符串的两端进行遍历比较。例如对于“level”字符串，我们可以进行如下操作。

- 计算需要比较的次数。由于“level”字符串长度为 5，是奇数，因此比较 2 次。
- 第 1 次比较：看“level”的第 1 个字符与最后 1 个字符是否相等，如果相等进行第 2 次比较。
- 第 2 次比较：看“level”的第 2 个字符与倒数第 2 个字符是否相等，如果相等则是回文。

只要在上面的比较过程中有一个不相等，则字符串不是回文。根据以上思路，程序代码如下所示：

```
1    #include <iostream>
2    using namespace std;
3
4    int IsRevStr(char *str)
5    {
6        int i, len;
7        int found = 1;                  //1 表示是回文字符串，0 表示不是
8
9        if(str == NULL)                 //判断 str 的有效性
10       {
11           return -1;
12       }
13       len = strlen(str);              //获得字符串长度
14       for(i=0; i<len/2; i++)
15       {
```

```
16              if(*(str+i) != *(str+len-i-1))          //遍历中如果发现相应头尾字符不等
17              {                                       //则字符串不是回文
18                  found=0;
19                  break;
20              }
21          }
22          return found;
23      }
24
25      int main()
26      {
27          char str1[10] = "1234321";                  //回文字符串
28          char str2[10] = "1234221";                  //非回文字符串
29
30          int test1 = IsRevStr(str1);                 //测试 str1 是不是回文
31          int test2 = IsRevStr(str2);                 //测试 str2 是不是回文
32
33          cout << "str1 is " << (test1 == 1 ? "": "not ")
34                          << "reverse string." << endl;
35          cout << "str2 is " << (test2 == 1 ? "": "not ")
36                          << "reverse string." << endl;
37
38          return 0;
39      }
```

这个程序中的 IsRevStr()函数用于判断字符串是否是回文字符串，如果是则返回 1，否则返回 0。输出结果：

```
str1 is reverse string.
str2 is not reverse string.
```

面试例题 15：编程实现 stcmp 库函数。

考点：库函数 strcmp 的实现细节。

出现频率：★★★★★

解析

此题实际上就是实现 C/C++库函数中的 strcmp 函数。对于两个字符串 str1 和 str2，若相等则返回 0，若 str1 大于 str2 则返回 1，若 str1 小于 str2 返回–1。

程序代码如下所示：

```
1   #include <iostream>
2   using namespace std;
3
4   int mystrcmp(const char *src, const char *dst)
5   {
6       int ret = 0 ;
7       while( !(ret = *(unsigned char *)src - *(unsigned char *)dst) && *dst)
```

```
8        {                                   //循环比较两个字符是否相等
9            ++src;                          //如果不等或者到了 dst 字符串末尾
10           ++dst;                          //退出循环
11       }
12       if ( ret < 0 )                      //ret 保存着字符比较的结果
13           ret = -1 ;
14       else if ( ret > 0 )
15           ret = 1 ;
16       return( ret );
17   }
18
19   int main()
20   {
21       char str[10] = "1234567";
22       char str1[10] = "1234567";          //str1 == str
23       char str2[10] = "12345678";         //str2 > str
24       char str3[10] = "1234566";          //str3 < str
25
26       int test1 = mystrcmp(str, str1);    //测试 str 与 str1 比较
27       int test2 = mystrcmp(str, str2);    //测试 str 与 str2 比较
28       int test3 = mystrcmp(str, str3);    //测试 str 与 str3 比较
29
30       cout << "test1 = " << test1 << endl;
31       cout << "test2 = " << test2 << endl;
32       cout << "test3 = " << test3 << endl;
33
34       return 0;
35   }
```

mystrcmp()函数对 src 和 dst 两个字符串同时进行遍历，当发现它们存在不同值时停止循环，并根据它们的最后一个字符的大小，返回相应的结果。程序输出结果：

```
test1 = 0
test2 = -1
test3 = 1
```

面试例题 16：编程查找两个字符串的最大公共子串。

考点：字符串相关的综合编程能力。

出现频率：★★★

解析

对于两个字符串 A 和 B，如果 A=“aocdfe”,B=“pmcdfa”,则输出“cdf”。

```
1    #include<stdio.h>
2    #include<stdlib.h>
3    #include<string.h>
4
5    char *commonstring(char *str1, char *str2)
```

```
6      {
7          int i, j;
8          char *shortstr, *longstr;
9          char *substr;
10
11         if (NULL == str1 || NULL == str2)        //判断 str1 与 str2 的有效性
12         {
13             return NULL;
14         }
15
16         if (strlen(str1) <= strlen(str2))         //shortstr 和 longstr 分别指向较短和较长的字符串
17         {
18             shortstr = str1;
19             longstr = str2;
20         } else
21         {
22             shortstr = str2;
23             longstr = str1;
24         }
25
26         if(strstr(longstr, shortstr) != NULL)     //如果在长的字符串中能寻找短的字符串
27         {                                         //返回短字符串
28             return shortstr;
29         }
30
31         substr = (char *)malloc(sizeof(char) * (strlen(shortstr) + 1));  //申请堆内存存放返回结果
32
33         for(i=strlen(shortstr)-1; i>0; i--)
34         {
35             for(j=0; j<=strlen(shortstr)-i; j++)
36             {
37                 memcpy(substr, &shortstr[j], i);         //将短字符串的一部分复制到 substr
38                 substr[i] = '\0';                        //其长度逐渐减小
39                 if(strstr(longstr, substr) != NULL)      //如果在 longstr 中能找到 substr 则返回 substr
40                     return substr;
41             }
42         }
43
44         return NULL;
45     }
46
47     int main()
48     {
49         char *str1 = (char *)malloc(256);              //分配堆内存存放字符串 str1 和 str2
50         char *str2 = (char *)malloc(256);
51         char *common=NULL;
```

```
52
53          gets(str1);                          //从终端输入 str1 和 str2
54          gets(str2);
55
56          common = commonstring(str2, str1);   //取最大的相同子串
57
58          printf("the longest common string is: %s\n", common);
59
60          return 0;
61      }
```

为了方便，可以利用库函数 strstr 寻找一个字符串中的子串。这个程序的步骤如下。

- 代码第 11～第 14 行，检查参数 str1 和 str2 的有效性。
- 计算两个字符串的长短。
- 调用 strstr 直接进行两个字符串的整串比较，如果不为 NULL，说明短串被长串所包含，直接返回短串即可，否则进行下一步。
- 申请一块大小为短串长度加 1 的堆内存，这块内存用于保存最大公共子串。
- 循环取短串的子串放入堆内存，调用 strstr 函数检查长串中是否包含这个子串，如果包含则返回堆内存。值得注意的是，短串的长度是不断减小的。

下面是程序执行结果：

```
aocdfe（输入）
pmcdfa（输入）
the longest common string is: cdf
```

面试例题 17：不能使用 printf，将十进制数以二进制和十六进制的形式输出。

考点：用字符串表示十进制数。

出现频率：★★★

解析

不能使用 printf 系列库函数，可以通过位运算得到十进制数的二进制形式和十六进制形式的字符串，再将字符串打印。程序代码如下所示：

```
1   #include<stdio.h>
2   #include<stdlib.h>
3   #include<string.h>
4
5   char *get2String(long num)              //得到二进制形式的字符串
6   {
7       int i=0;
8       char* buffer;
9       char* temp;
10
11      buffer = (char*)malloc(33);
12      temp = buffer; //temp
```

```
13        for (i=0; i < 32; i++)
14        {                                          //给数组的 32 个元素赋 0 或 1
15            temp[i] = num & (1 << (31 - i));
16            temp[i] = temp[i] >> (31 - i);
17            temp[i] = (temp[i] == 0) ? '0': '1';
18        }
19        buffer[32] = '\0';                         //字符串结束符
20
21        return buffer;
22    }
23
24    char *get16String(long num)                    //得到十六进制形式的字符串
25    {
26        int i=0;
27        char* buffer = (char*)malloc(11);
28        char* temp;
29
30        buffer[0] = '0';                           // "0x" 开头
31        buffer[1] = 'x';
32        buffer[10] = '\0';                         //字符串结束符
33        temp = buffer + 2;
34
35        for (i=0; i < 8; i++)
36        {                                          //给数组的 8 个元素赋值
37            temp[i] = (char)(num<<4 * i>>28);
38            temp[i] = temp[i] >= 0 ? temp[i] : temp[i] + 16;
39            temp[i] = temp[i] < 10 ? temp[i] + 48 : temp[i] + 55;
40        }
41        return buffer;
42    }
43
44    int main()
45    {
46        char *p = NULL;
47        char *q = NULL;
48        int num = 0;
49
50        printf("input num: ");
51        scanf("%d", &num);                         //输入整数
52
53        p = get16String(num);                      //得到 16 进制形式的字符串
54        q = get2String(num);                       //得到 2 进制形式的字符串
55
56        printf("%s\n", p);
57        printf("%s\n", q);
58
```

```
59          return 0;
60
}
```

这个程序中，get2String 和 get16String 分别用于得到二进制字符串和十六进制字符串。

long 型的整数是 4 个字节，即 32 位，每一位用 0 或 1 表示，get2String 函数申请了 33 个字节（包括‘\0’）的堆内存存放结果，get16String 函数申请 11 个字节（包括‘0’、‘x’和‘\0’）。然后把每位的值赋给堆内存的对应位置。程序执行结果：

```
input num: 456789（输入）
0x0006F855
00000000000000000000000001010101
```

面试例题 18：在字符串中，插入字符统计的个数。

考点：字符串综合编程能力。

出现频率：★★★★

解析

根据题意，需要在字符串中插入字符统计的个数。例如字符串 aaab，插入字符个数后变成 aaa3b1。

程序代码如下所示。

```
1    #include <stdlib.h>
2    #include <stdio.h>
3    #include <string.h>
4
5    #define MAXCOUNT 2*100
6
7    char *transformation(char *str)
8    {
9        int len=strlen(str);
10       char *buf=new char[len+1];
11
12       char *p=str;
13       char *q=p+1;
14       int count=1;
15       while(*q)
16       {
17           if(*p==*q)
18           {
19               count++;
20               p++;
21               q++;
22           }
23           else
24           {
25               itoa(count,buf,10);
```

```
26                  int nbits=strlen(buf);
27                  strcat(buf,q);
28                  *q=0;
29                  strcat(str,buf);
30                  q+=nbits;
31                  p=q;
32                  q=p+1;
33                  count=1;
34              }
35          }
36
37          itoa(count,buf,10);
38          strcat(str,buf);
39
40          delete []buf;
41          buf=NULL;
42          return str;
43      }
44
45      int main()
46      {
47          char str[MAXCOUNT];
48
49          printf("please input a string:");
50          scanf("%s",&str);
51          printf("before transformation: %s\n",str);
52          char *pstr=transformation(str);
53          printf("after    transformation: %s\n",pstr);
54
55          return 0;
56      }
```

程序中的 transformation()函数用来转换字符串。我们以字符串 aaab 为例来说明其执行过程。为了计算方便，首先申请了 5 字节的堆内存 buf（aaab 为 4 字节，加一个结束符为 5 字节）来存放字符串数字相关的信息。初始计数为 1，然后进行下面的步骤。

遍历 aaab 直到找到不同的字符，然后在 buf 中保存 3，把 str 变为 aaa（即字符“b”位置内存设为 0）。然后执行 strcat(str, buf)，此时 str 变为 aaa3b，计数为 1。如果到字符串末尾（碰到结束符“\0”），则退出循环，否则继续进行以上的步骤。

如果退出循环，最后一个字符个数存入 buf（这里 b 的个数 1），此时 str 为 aaa3b，经过调用 strcat(str, buf)，str 变为 aaa3b1。

释放 buf 堆内存并返回 str。程序执行结果为：

```
please input a string:aaab
before transformation: aaab
after transformation: aaa3b1
```

面试例题 19：字符串编码例题。

考点：字符串相关的综合编程能力。

出现频率：★★

对一个长度小于 20 的字符串进行编码，遵循 3 个规则。

- 字母用后面的第 4 个字母替换。例如：a->e，A -> E，X -> b，y ->c,，z ->d。
- 如果字符不是字母，字符保持不变。
- 翻转整个字符串。

解析

整个过程可以分为两部分：替换字符和翻转字符串，其中替换字符还包括检查字符是否是在字母表中的。程序代码如下：

```
1    #include <iostream>
2    using namespace std;
3
4    char LowerCaseAlphabets[] =
5                    {'a','b','c','d','e','f','g','h',
6                    'i','j','k','l','m','n','o','p',
7                    'q','r','s','t','u','v','w','x','y','z'};
8    char UpperCaseAlphabets[]=
9                    {'A','B','C','D','E','F','G','H',
10                   'I','J','K','L','M','N','O','P',
11                   'Q','R','S','T','U','V','W','X','Y','Z'};
12
13   char GetFourthChar(char chrSource,char alphabets[])
14   {
15       for(int i=0;i<26;i++)
16       {
17           if(alphabets[i]==chrSource)
18           {
19               int index = (i+4) % 26;
20               return alphabets[index];
21           }
22       }
23       return '\0';
24   };
25
26   void ReplaceChars(char chars[], int len)
27   {
28       for(int i=0; i<len; i++)
29       {
30           if('a' <= chars[i] && chars[i] <= 'z')
31           {
32               chars[i]=GetFourthChar(chars[i], LowerCaseAlphabets);
```

```
33              }
34              else if('A' <= chars[i] && chars[i] <= 'Z')
35              {
36                  chars[i]=GetFourthChar(chars[i], UpperCaseAlphabets);
37              }
38          }
39      };
40
41      void ReverseString(char str[],int len)
42      {
43          int begin=0, end=len-1;
44          if(str[end] == '\0')
45              end--;
46
47          char hold;
48          while(begin < end)
49          {
50              hold = str[begin];
51              str[begin] = str[end];
52              str[end] = hold;
53
54              begin++;
55              end--;
56          }
57      };
58
59      void EncodeString(char str[],int len)
60      {
61          ReplaceChars(str, len);
62          ReverseString(str, len);
63      };
64
65      int main()
66      {
67          char hello[] = "hasd11H";
68
69          EncodeString(hello, strlen(hello));
70          cout << hello << endl;
71
72          return 0;
73      }
```

程序说明如下：

- 代码第 61～第 62 行，EncodeString()函数调用了 ReplaceChars ()函数和 ReverseString()函数，前者替代字符，后者翻转整个字符串。
- ReplaceChars()函数调用 GetFourthChar()函数来查找并替换后面第 4 个字符。

GetFourthChar()函数查找两个全局数组，这两个数组分别包含了所有的字母。

- ReverseString()函数使用了 begin 和 end 两个分别指向字符串头尾的指针，然后头尾的内容不断交换，begin 指针往尾移动，end 指针往头移动，如此循环直到两个指针碰头。

程序执行结果如下：

```
L11hwel
```

面试例题 20：反转字符串，但其指定的子串不反转。

考点：字符串相关的综合编程能力。

出现频率：★★★

给定一个字符串以及该字符串的子串，将字符串反转，但子串部分不反转，示例如下。

输入字符串为"Welcome you, my friend"

子串为"you"

输出为"dneirf ym ,you emocleW"

解析

对于本类型题目，采取的步骤如下所示。

- 扫描一遍字符串，然后用 stack 把它反转，同时记录下子串出现的位置。
- 扫描一遍记录下来的子串，再用 stack 反转。
- 将堆栈里的字符弹出，这样子串又恢复了原来的顺序。

这里使用扫描数组的方法。如果扫描中发现子串，就将子串倒过来压入堆栈。最后再将堆栈里的字符弹出，这样子串又恢复了原来的顺序。C++标准库中的 stack 是后进先出的栈结构，使用 stack 可以轻松地完成这个转换。程序代码如下：

```
1    #include <iostream>
2    #include <cassert>
3    #include <stack>
4    using namespace std;
5
6    const char* reverse(const char* s1, const char* token)
7    {
8         stack<char> stack1;
9         const char* ptoken = token, *head = s1, *rear = s1;
10        assert(s1 && token);
11        while (*head != '\0')
12        {
13            while(*head != '\0' && *ptoken == *head)
14            {
15                 ptoken++;
16                 head++;
17            }
18            if(*ptoken == '\0')
19            {
```

```
20                const char* p;
21                for(p=head-1; p>=rear; p--)
22                {
23                    stack1.push(*p);
24                }
25                ptoken = token;
26                rear = head;
27            }
28            else
29            {
30                stack1.push(*rear++);
31                head = rear;
32                ptoken = token;
33            }
34        }
35        char *pReturn = new char[strlen(s1)+1];
36        int i=0;
37        while(!stack1.empty())
38        {
39            pReturn[i++] = stack1.top();
40            stack1.pop();
41        }
42        pReturn[i]='\0';
43
44        return pReturn;
45    }
46
47    int main(int argc, char* argv[])
48    {
49        char welcome[] = "Welcome you, my friend";
50        char token[] = "you";
51        const char *pRev = reverse(welcome, token);
52
53        cout << "before reverse:" << endl;
54        cout << welcome << endl;
55        cout << "after reverse:" << endl;
56        cout << pRev << endl;
57
58        return 0;
59    }
```

代码第 13～第 17 行，reverse()函数搜索字符串的字串位置。其中指针 ptoken 记录当前扫描的子串位置，如果其内容为结束符'\0'（代码第 18～第 27 行），如果搜索到了这个子串，将它倒过来压入堆栈，否则直接压入堆栈（代码第 28～第 33 行）。最后申请一块堆内存，使用退栈把栈中内容存入堆内存中并返回堆内存。程序执行结果如下：

```
before reverse:
welcome you, my friend
after reverse:
dneirf ym ,you emocleW
```

面试例题 21：编写字符串反转函数 strrev。

考点：字符串相关的综合编程能力。

出现频率：★★★★

编写字符串反转函数：strrev，要求时间和空间效率都尽量高。

测试用例:输入“abcd”，输出应为“dcba”。

解析

这种类型的题目最简单的解法是：遍历字符串，将第 1 个字符和最后 1 个字符交换，第 2 个和倒数第 2 个字符交换，依次循环。解法 1 代码如下：

```
1     char* strrev1(const char* str)
2     {
3         int len = strlen(str);
4         char* tmp = new char[len + 1];
5
6         strcpy(tmp,str);
7         for (int i = 0; i < len/2; ++i)
8         {
9             char c = tmp[i];
10            tmp[i] = tmp[len - i - 1];
11            tmp[len - i - 1] = c;
12        }
13
14        return tmp;
15    }
```

解法 1 是通过数组下标的方式访问字符串，也可以用指针直接操作。解法 2 代码如下：

```
1     char* strrev2(const char* str)
2     {
3         char* tmp = new char[strlen(str) + 1];
4         strcpy(tmp,str);
5         char* ret = tmp;
6         char* p = tmp + strlen(str) - 1;
7
8         while (p > tmp)
9         {
10            char t = *tmp;
11            *tmp = *p;
12            *p = t;
13
```

```
14            --p;
15            ++tmp;
16        }
17
18        return ret;
19    }
```

解法 1 和解法 2 都没有考虑时间和空间的优化，典型的优化策略就是两个字符交换的算法优化，我们可以借助异或运算符（^）完成两个字符的交换，对应的是解法 3 和解法 4。

解法 3：

```
1     char* strrev3(const char* str)
2     {
3         char* tmp = new char[strlen(str) + 1];
4         strcpy(tmp,str);
5         char* ret = tmp;
6         char* p = tmp + strlen(str) - 1;
7
8         while (p > tmp)
9         {
10            *p ^= *tmp;
11            *tmp ^= *p;
12            *p ^= *tmp;
13
14            --p;
15            ++tmp;
16        }
17
18        return ret;
19    }
```

解法 4：

```
1     char* strrev4(const char* str)
2     {
3         char* tmp = new char[strlen(str) + 1];
4         strcpy(tmp,str);
5         char* ret = tmp;
6
7         char* p = tmp + strlen(str) - 1;
8
9         while (p > tmp)
10        {
11            *p = *p + *tmp;
12            *tmp = *p - *tmp;
13            *p = *p - *tmp;
14
15            --p;
```

```
16            ++tmp;
17        }
18
19        return ret;
20    }
```

也可以使用递归算法来解决这个问题。每次交换首尾两个字符，中间部分又变成和原来字符串相同的问题。

解法5：

```
1     char* reverse5(char* str,int len)
2     {
3         if (len <= 1)
4             return str;
5
6         char t = *str;
7         *str = *(str + len -1);
8         *(str + len -1) = t;
9
10        return (reverse5(str + 1,len - 2) - 1);
11    }
```

注意：这5个解法中只有解法5修改了输入字符串，其他4种都是在函数体内申请堆内存。

下面给出测试用的main()函数：

```
1     int main(int argc,char* argv[])
2     {
3         char* str = "123456";
4         cout << str << endl;
5
6         char* str2 = reverse1(str);
7         cout << str2 << endl;
8
9         char* str3 = reverse2(str2);
10        cout << str3 << endl;
11
12        char* str4 = reverse3(str3);
13        cout << str4 << endl;
14
15        char* str5 = reverse4(str4);
16        cout << str5 << endl;
17
18        char* str6 = reverse5(str5,strlen(str5));
19        cout << str6 << endl;
20
21        return 0;
22    }
```

程序输出结果如下：

```
123456
654321
123456
654321
123456
654321
```

面试例题 22：编程实现任意长度的两个正整数相加。

考点：字符串相关的综合编程能力。

出现频率：★★★★

解析

在 C/C++中可以用 int、float、double 等类型表示数字，但是它们的长度都是有限的，而本题要求数字是任意长度，因而需要用字符串表示数字，同样用字符串表示结果。因此所要做的就是做一个类似整数加法的字符串转换，程序代码如下所示：

```
1   #include <stdio.h>
2   #include <string.h>
3   #include <stdlib.h>
4   #include <math.h>
5
6   char* addBigInt(char* num1, char* num2)
7   {
8       int c = 0;                                          //进位，开始最低进位为 0
9       int i = strlen(num1)-1;                             //指向第一个加数的最低位
10      int j = strlen(num2)-1;                             //指向第二个加数的最低位
11      int maxLength = strlen(num1) >= strlen(num2) ?
12          ( strlen(num1)+1 ) : ( strlen(num2)+1 ) ;       //得到 2 个数中较大数的位数
13      char* rst = (char*)malloc(maxLength+1);             //保存结果
14      int k;
15      if (rst == NULL)
16      {
17          printf("malloc error!\n");
18          exit(1);
19      }
20
21      rst[maxLength] = '\0';     //字符串最后一位为'\0'
22      k = strlen(rst) - 1 ;      //指向结果数组的最低位
23      while ( (i >= 0) && (j >= 0) )
24      {
25          rst[k] = ( (num1[i] - '0') + (num2[j] - '0') + c )%10 +'0';   //计算本位的值
26          c = ( (num1[i] - '0') + (num2[j] - '0') + c )/10;             //向高位进位值
27          --i;
28          --j;
29          --k;
30      }
```

```
31          while ( i >= 0 )
32          {
33              rst[k] = ( (num1[i] - '0') + c )%10 + '0';
34              c = ( (num1[i] - '0') + c)/10;
35              --i;
36              --k;
37          }
38          while ( j >= 0 )
39          {
40              rst[k] = ( (num2[j] - '0') + c )%10 + '0';
41              c = ( (num2[j] - '0') + c )/10;
42              --j;
43              --k;
44          }
45          rst[0] = c + '0';
46
47          if ( rst[0] != '0' )                    //如果结果最高位不等于0，则输出结果
48          {
49              return rst;
50          }
51          else
52          {
53              return rst+1;
54          }
55      }
56
57      int main()
58      {
59          char num1[] = "123456789323";
60          char num2[] = "45671254563123";
61          char *result = NULL;
62
63          result = addBigInt(num1, num2);
64          printf("%s + %s = %s\n", num1, num2, result);
65
66          return 0;
67      }
```

程序执行结果如下所示：

```
123 456 789 323 + 45 671 254 563 123 = 45 794 711 352 446
```

面试例题 23：编程实现字符串循环右移。

考点：字符串综合编程能力。

出现频率：★★★★

解析

根据题意，编写的函数能把字符串循环右移 n 位。例如字符串“abcdefghi”，如果 n=2，移

位后是“hiabcdefgh”。

程序代码如下所示：

```
1    #include <stdio.h>
2    #include <stdlib.h>
3
4    void loopMove(char *str, int n)
5    {
6        int i = 0;
7        char *temp = NULL;
8        int strLen = 0;
9        char *head = str;                          //指向字符串头
10
11       while(*str++);
12       strLen = str - head - 1;                   //计算字符串长度
13       n = n % strLen;                            //计算字符串尾部移到头部的字符个数
14       temp = (char *)malloc(n);                  //分配内存
15       for (i = 0; i < n; i++)
16       {
17           temp[i] = head[strLen - n + i];        //临时存放从尾部移到头部的字符
18       }
19       for (i = strLen - 1; i >= n; i--)
20       {
21           head[i] = head[i - n];                 //从头部字符移到尾部
22       }
23       for (i = 0; i < n; i++)
24       {
25           head[i] = temp[i];                     //从临时内存区复制尾部字符
26       }
27
28       free(temp);
29   }
30
31   int main(void)
32   {
33       char string[] = "123456789";
34       int steps = 0;
35
36       printf("string: %s\n", string);
37       printf("input step: ");
38       scanf("%d", &steps);
39       loopMove(string, steps);   //向右循环移位
40       printf("after loopMove %d: %s\n", steps, string);
41
42       return 0;
43   }
```

程序执行结果如下所示：

```
string: 123456789
input step; 6（输入）
after loopMove 6: 456789123
```

程序中首先计算字符串尾部移到头部的字符个数，然后分配一个相同大小的堆内存来临时保存这些字符，最后做两次循环分别把头部字符移位到尾部以及把堆内存中的内容复制到字符串头部。

面试例题 24：从字符串的指定位置开始，删除其指定长度字符。

考点：字符串综合编程能力。

出现频率：★★★★

解析

根据题意，假设一个字符串“abcdefg”，从第 2 个字符开始（索引为 1），删除两个字符。删除后的字符串是“adefg”。

程序代码如下：

```
1   #include <stdio.h>
2   #include <string.h>
3
4   char *deleteChars(char *str,int pos,int len)
5   {
6       char *p = str + pos - 1;                    //指向 pos 位置字符
7       int tt = strlen(str);                       //计算字符长度
8
9       if( (pos < 1) || (p-str) > tt)              //检查 pos 是否不大于 1
10      {                                           //或者 pos 超出字符串长度
11          return str;
12      }
13
14      if( (p+len-str) > tt)                       //len 大于 pos 后剩余的字符个数
15      {                                           //只需对 pos 位置赋'\0'
16          *p = '\0';
17          return str;
18      }
19
20      //删除 len 个字符
21      while(*p && *(p+len))                       //len 小于或等于 pos 后剩余的字符个数
22      {                                           //删除中间 len 个字符
23          *p = *(p+len);
24          p++;
25      }
26      *p = '\0';
27
28      return str;
29  }
```

```
30
31  int main()
32  {
33      char string[] = "abcdefg";
34      int pos = 0;
35      int len = 0;
36      int steps = 0;
37      printf("string: %s\n", string);
38      printf("input pos: ");
39      scanf("%d", &pos);
40      printf("input len: ");
41      scanf("%d", &len);
42      deleteChars(string, pos, len);                //删除 string 的 pos 位置后 len 个字符
43      printf("after delete %d chars behind pos %d: %s\n", len, pos, string);
44
45      return 0;
46  }
```

程序执行结果如下所示：

```
string: abcdefg
input pos: 2（输入）
input len: 2（输入）
after delete 2 chars behind pos 2: adefg
```

deleteChars 函数首先检查 pos 以及 len 的合法性，然后找到字符串的 pos 位置，最后删除 len 个字符。

面试例题 25：字符串的排序及交换。

考点：字符串综合编程能力。

出现频率：★★★

编写一个函数，首先将一条字符串分成两部分，前半部分按 ASCII 码升序排序，后半部分不变，（如果字符串是奇数则中间的字符不变）其次将前后两部分交换，最后输出该字符串。测试字符串“ADZDDJKJFIEJHGI”。

解析

程序代码如下：

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   /* 冒泡排序算法 */
5   void mysort(char *str, int num)
6   {
7       int i, j;
8       int temp = 0;
9
```

```
10        for (i = 0; i < num; i++)
11        {
12            for (j = i+1; j < num; j++)
13            {
14                if (str[i] < str[j])                //如果下一个值比当前值大
15                {                                   //则交换两个元素值
16                    temp = str[i];
17                    str[i] = str[j];
18                    str[j] = temp;
19                }
20            }
21        }
22    }
23
24    char *foo(char *str)
25    {
26        int len = 0;
27        char *start = NULL;
28
29        if (str == NULL)                            //检查参数 str 的有效性
30        {
31            return NULL;
32        }
33
34        start = str                                 //保存头部位置
35        while(*str++);
36        len = str - start - 1;                      //计算字符串长度
37        len = len / 2;                              //计算需要排序的字符个数
38        str = start;
39
40        mysort(str, len);                           //从大到小排序
41
42        return str;
43    }
44
45    int main()
46    {
47        char string[] = "ADZDDJKJFIEJHGI";
48
49        printf("before transformation: %s\n", string);
50        foo(string);
51        printf("after transformation: %s\n", string);
52
53        return 0;
54    }
```

程序执行结果：

```
before transformation: ADZDDJKJFIEJHGI
after transformation: ZKJDDDAJFIEJHGI
```

foo()函数首先获得字符串的长度，然后计算需要排序的字符个数，最后调用 mysort 函数（使用冒泡排序方法）对字符进行排序。

面试例题 26：编程实现删除字符串中所有指定的字符。

考点：字符串综合编程能力。

出现频率：★★★★

解析

根据题意，假设字符串为“cabcdefcgchci”，把该字符串中所有的字符“c”删除掉后，那么结果应该是“abdefghi”。

程序代码如下：

```
1    #include <stdio.h>
2
3    char *deleteChar(char *str,char c)
4    {
5        char *head = NULL;
6        char *p = NULL;
7
8        if (str == NULL)                    //检查 str 的有效性
9        {
10           return NULL;
11       }
12
13       head = p = str;                     //指向字符串头,准备遍历
14
15       while(*p++)
16       {
17           if(*p != c)                     //如果不等于 c 的值则在 str 中记录
18           {
19               *str++ = *p;
20           }
21       }
22       *str = '\0';
23
24       return head;
25   }
26
27   int main(void)
28   {
29       char string[] = "cabcdefcgchci";
```

```
30        char c = 0;
31
32        printf("input char: ");
33        scanf("%c", &c);
34        printf("before delete: %s\n", string);
35        deleteChar(string, c);        //删除所有的 c
36        printf("after delete: %s\n", string);
37
38        return 0;
39    }
```

程序执行结果：

```
input char: c
before delete: cabcdefcgchci
after delete: abdefghi
```

deleteChar()函数首先判断传入字符指针的有效性，然后使用两个指针进行操作。其中一个指针用来做记录，另一个指针进行字符串遍历。

面试例题 27：分析代码——使用 strcat 连接字符串。

考点：字符串综合编程能力。

出现频率：★★★

下面的程序代码有什么问题吗？输出是什么？

```
1     #include<iostream>
2     using namespace std;
3
4     int main()
5     {
6         char *str1 = "hello";
7         char *str2 = " china";
8         char *str3 = NULL;
9
10        str3 = new char[strlen(str1)+strlen(str2)+1];
11        str3[0] = '\n';
12        strcat(str3,str1);
13        strcat(str3,str2);
14        cout << str3 << endl;
15
16        return 0;
17    }
```

解析

代码第 12 行和第 13 行调用 strcat 函数。strcat 函数是库函数，其原型如下：

```
extern char *strcat(char *dest, const char *src);
```

strcat 函数把 src 字符串加到 dest 字符串之后。代码第 10 行中用 new 运算符申请的堆内存是没有被初始化的，内存中的值是随机数。代码第 12 行调用 strcat 不能把 str1 的内容复制到堆

内存中，并且会导致数组越界，同样的问题发生在代码第 13 行的调用中。应在代码第 11 行把 str3[0]赋为结束符'\0'。正确的代码应为：

```
1    #include<iostream>
2    using namespace std;
3
4    int main()
5    {
6        char *str1 = "hello";
7        char *str2 = " china";
8        char *str3 = NULL;
9
10       str3 = new char[strlen(str1)+strlen(str2)+1];      //分配堆内存
11       str3[0] = '\0';                                     //str3[0]赋为结束符'\0'，以便 strcat 能正常调用
12       strcat(str3,str1);                                  //str3 变为"hello"
13       strcat(str3,str2);                                  //str3 变为"hello china"
14       cout << str3 << endl;
15
16       return 0;
17   }
```

程序执行结果：

```
hello china
```

答案

程序 str3 指向的堆内存没有初始化，不含有字符串结束符。其输出结果是随机值。

面试例题 28：编程实现库函数 strcat。

考点：库函数 strcat 的实现细节。

出现频率：★★★★★

解析

库函数 strcat 把字符串内容连接到目标字符串的后面，所以应该从目标字符串的末尾，也就是结束符“\0”的位置开始插入另一个字符串的内容。

```
1    #include <stdio.h>
2    #include <stdlib.h>
3
4    char *mystrcat(char *dest, const char *src)
5    {
6        char *ret;
7
8        ret = dest;                          //保存目的字符串首地址以便返回
9        while (*dest++);
10       dest--;                              //此时 dest 指向字符串结束符
11       while (*dest++ = *src++);            //循环复制
12
```

```
13        return ret;
14    }
15
16    int main(void)
17    {
18        char *dest = NULL;
19        char *str1 = "Hello ";
20        char *str2 = "World!";
21
22        dest = (char *)malloc(256);
23        *dest = '\0';                          //为把目标字符串置为空，将结束符放在其开头
24        mystrcat(mystrcat(dest, str1), str2);  //链式表达式连接 str1 和 str2
25        printf("dest: %s\n", dest);
26        free(dest);
27        dest = NULL;
28
29        return 0;
30    }
```

程序执行结果：

```
Hello World!
```

面试例题 29：计算含有汉字的字符串长度。

考点：字符串综合编程能力。

出现频率：★★

编写 gbk_strlen 函数，计算含有汉字的字符串的长度，汉字作为一个字符处理。已知汉字编码为双字节，其中首字节<0，尾字节在 0～63 以外（如果一个字节范围为-128～127）。

解析

程序代码如下：

```
1     #include <iostream>
2     using namespace std;
3
4     int gbk_strlen(const char* str)
5     {
6         const char* p = str;                          //p 用于后面遍历
7
8         while(*p)                                     //若是结束符 0 则结束循环
9         {
10            if (*p < 0 && (*(p+1)<0 || *(p+1)>63))    //中文汉字情况
11            {
12                str++;                                //str 移动一位，p 移动两位，因此长度加 1
13                p += 2;
14            }
15            else
```

```
16              {
17                  p++;                                //str 不动，p 移动一位，长度加 1
18              }
19          }
20
21          return  （p-str）;//返回地址之差
22      }
23
24      int main()
25      {
26          char str[] = "abc 你好 123 中国 456";           //含有中文汉字的字符串
27
28          int len = gbk_strlen(str);                  //获得字符串长度
29          cout << str << endl;
30          cout << "len = " << len << endl;
31
32          return 0;
33      }
```

gbk_strlen()函数中使用两个指针指向的地址之差来获得字符串长度。当遇到中文汉字时，p 移动两次指向中文汉字的下一个字符，同时为了使汉字的长度计算 1 次，则需要将 src 移动一位。程序输出结果如下：

```
abc 你好 123 中国 456
len = 13
```

面试例题 30：找出 01 字符串中 0 和 1 连续出现的最大次数。

考点：字符串综合编程能力。

出现频率：★★

解析

程序代码如下：

```
1       #include <iostream>
2       using namespace std;
3
4       void Calculate(const char *str, int *max0, int *max1)
5       {
6           int temp0 = 0;                              //保存连续是'0'的最大长度
7           int temp1 = 0;                              //保存连续是'1'的最大长度
8
9           while(*str)                                 //遍历 01 字符串
10          {
11              if (*str == '0')                        //当前字符是'0'
12              {
13                  (*max0)++;                          //'0'的计算长度加 1
14                  if (*(++str) == '1')                //如果下一个字符是'1'
15                  {
```

```
16                  if (temp0 < *max0)              //判断当前最大长度是否需要保存
17                  {
18                      temp0 = *max0;
19                  }
20                  *max0 = 0;
21              }
22          }
23          else if (*str == '1')                   //当前字符是'1'
24          {
25              (*max1)++;                          //'1'的计算长度加 1
26              if (*(++str) == '0')                //如果下一个字符是'0'
27              {
28                  if (temp1 < *max1)              //判断当前最大长度是否需要保存
29                  {
30                      temp1 = *max1;
31                  }
32                  *max1 = 0;
33              }
34          }
35      }
36
37      *max0 = temp0;                              //'0'的最大长度返回 max0
38      *max1 = temp1;                              //'1'的最大长度返回 max1
39  }
40
41  int main(int argc, char *argv[])
42  {
43      char string[] = "00001110110000001100110101101001010101011111010";
44
45      int max0 = 0;
46      int max1 = 0;
47
48      Calculate(string, &max0, &max1);            //计算 max0 和 max1
49      cout << "max0 = " << max0 << endl;
50      cout << "max1 = " << max1 << endl;
51
52      return 0;
53  }
```

程序输出结果：

```
max0 = 6
max1 = 5
```

面试例题 31：实现字符串的替换。

考点：字符串综合编程能力

出现频率：★★★

用 C++编写一个小程序，先请用户输入 3 个字符串，然后在第 1 个字符串中找出所有的第

2 个字符串，第 3 个字符串替换掉第 2 个字符串，最后输出新的字符串。

解析

程序代码如下：

```
1    #include <iostream>
2    using namespace std;
3
4    char* replace(const char* str, const char* sub1, const char* sub2, char* output)
5    {
6        char* pOutput = NULL;
7        const char* pStr = NULL;
8        int lenSub1 = strlen(sub1);                   //子串 sub1 的长度
9        int lenSub2 = strlen(sub2);                   //子串 sub2 的长度
10
11       pOutput = output;
12       pStr = str;                                   //用于寻找子串
13       while(*pStr != 0)
14       {
15           pStr = strstr(pStr, sub1);                //在 str 中寻找 sub1 子串
16           if (NULL != pStr)                         //找到 sub1 子串
17           {
18               memcpy(pOutput, str, pStr-str);       //复制 str 的前一部分 output
19               pOutput += pStr-str;
20               memcpy(pOutput, sub2, lenSub2);       //复制 sub2 子串到 output
21               pOutput += lenSub2;
22               pStr += lenSub1;                      //为了下一次复制作准备
23               str = pStr;
24           }
25           else                                      //找不到 sub1 子串
26           {
27               break;
28           }
29       }
30       *pOutput = '\0';
31       if (*str != '\0')
32       {
33           strcpy(pOutput, str);                     //复制 str 剩余部分到 ouput
34       }
35
36       return output;
37   }
38
39   int main()
40   {
41       char str[50] = "";                            //源字符串 str
```

```
42        char sub1[10]   = "";                         //被替换的字符串 sub1
43        char sub2[10]   = "";                   //用来替换 sub2
44        char output[100] = "";                  //输出字符串
45
46        cout << "str: ";
47        cin >> str;
48        cout << "sub1: ";
49        cin >> sub1;
50        cout << "sub2: ";
51        cin >> sub2;
52
53        cout << replace(str, sub1, sub2, output) << endl;
54
55        return 0;
56    }
```

程序执行结果：

```
str: abcdefcdg
sub1: cd
sub2: 123
ab123ef123g
```

replace()函数把 str 中所有 sub1 子串替换为 sub2 子串，结果存入 output 指向的内存块中。它循环使用库函数 strstr 寻找字符串中的子串，如果找到 sub1 子串，则 output 复制 str 中不属于 sub1 子串的部分以及 sub2 字符串。如果找不到 sub1 子串，则退出循环，并且把 str 剩余的部分复制到 output 中。

第8章 位运算与嵌入式编程

C 语言是嵌入式开发所必需的编程技术，因此在招聘嵌入式系统程序员时它是必备的知识。

对于应聘者来说，可以通过测试题了解出题者的一些情况。例如出题者是擅长于开发微机还是嵌入式系统。

对于招聘方来说，一个测试题能从多个方面反映应聘者的素质，应聘者采取的回答方式可以反应出他的基本素质。本章列出了一些针对嵌入式系统的面试题，希望能给应聘者一些帮助。

8.1 位制转换与位运算

位制转换与位运算是程序员使用 C 语言进行嵌入式编程时必须注意的方面。这是因为嵌入式程序员需要很清楚各种类型变量的内存结构，以及内存大小和不同类型之间的转换。

8.1.1 位制转换

C 语言中的不同类型变量需要不同的字节长度进行存储(例如 char 和 int 的长度就不相同)，如果存在不同类型变量进行相互赋值时，就会发生位制转换，例如当 char 和 int 之间的相互转换时，由于 char 类型变量所占的内存小于 int 类型，因此不会发生数据丢失；而当 int 类型转换为 char 类型时，会发生数据字节丢失。

面试例题 1：分析代码写出结果——位制转换。

考点：使用 printf 输出不同类型变量。

出现频率：★★★★

```
1    #include <stdio.h>
2
3    int main(int argc, char* argv[])
4    {
```

```
5        int i = 5.0l;
6        float f = 5;
7
8        printf("%f\n", 5);
9        printf("%lf\n", 5.0l);
10       printf("%f\n", f);
11
12       printf("%d\n", 5.0l);
13       printf("%d\n", i);
14
15       return 0;
16   }
```

解析

32位平台中int类型和float类型都占4字节，double占8字节。以下的讨论都是基于32位平台。

printf根据说明符“%f”，认为参数是double类型的参数(在printf函数中，float会自动转换成double)，因此从栈中读8字节。类似的当printf后的说明符为“%d”，会认为参数是int类型的参数，因此从栈中读4字节。

代码第8行，参数5为int型，所以在栈中分配了4字节的内存存放参数5。然后printf从栈中读8字节。很显然，内存访问越界，会有不可预料的情况发生。

代码第9行和第10行，参数5.0l和f分别是double类型和float类型 (注意这里f在赋值时已经作了int到float的转化)，而float和double都是浮点类型，其相互转化是相对安全的，因此这两段代码输出都是5.000 000。

代码第12行参数5.0l占8字节，而printf读4字节，同样会出现不可预料的情况。

代码第13行参数i是由5.0l转换过来的，这里的转换不同于printf中的读取，转换是在数据大小允许的范围之内（没有溢出）进行的。转换的结果是i等于5，占4字节，因此这段代码输出为5。

从上述分析中可以看出，如果在printf或者scanf中指定为“%f”，那么后面对应的参数列表也应该是浮点数，或者是指向浮点变量的指针，否则就不应该加载支持浮点数的函数。

答案

```
0.000000
5.000000
5.000000
0
5
```

8.1.2 位运算

程序设计中的许多运算都是以字节作为最基本位进行的，但在很多系统程序中常常要进行位(bit)一级运算或处理。C语言提供了位运算的功能，这使得C语言可以像汇编语言一样用来编写系统程序。C语言的位运算功能区别于其他大多数高级程序设计语言，是C语言编写系统程序的基础。

面试例题 2：分析代码写出结果——位运算。

考点：使用位操作符>>和<<。

出现频率：★★★★

```
1   #include <stdio.h>
2
3   int main(int argc, char* argv[])
4   {
5       unsigned short int i = 0;
6       unsigned char ii = 255;
7       int j = 8, p, q;
8
9       p = j << 1;
10      q = j >> 1;
11      i = i - 1;
12      ii = ii + 1;
13
14      printf("i = %d\n", i);
15      printf("ii = %d\n", ii);
16      printf("p = %d\n", p);
17      printf("q = %d\n", q);
18
19      return 0;
20  }
```

解析

本题有两个考点：一是无符号类型的计算；二是用移位的方式代替乘除法。

变量 i 是一个 unsigned short int 类型，在 32 位平台下是 2 字节，因此其范围为 0～65535。i 赋值后为 0，内存中的数据为 0x0000。当执行了"i = i – 1";之后，内存中的数据变为 0Xffff，结果为 65535。

变量 ii 是一个 unsigned short char 类型，占 1 字节，因此其范围为 0～255。ii 赋值之后为 255，内存中的数据为 0xff。当执行了代码 12 行之后，内存中的数据变为 0X00，即结果是 0。

左移操作<<相当于乘法操作，<< n 相当于乘以 2n。右移操作>>相当于除法操作，>> n 相当于除以 2n。因此变量 p 和 q 的输出分别是 16 和 4。

通过以上的分析可以看出，对于移位操作需要清楚地了解数据在内存中的大小以及其表达形式。另外移位操作是种高效率的（可以代替乘除法），计算方式因此在嵌入式系统或者其他需要高效率计算的地方得到广泛运用。

答案

```
i = 65535
ii = 0
p = 16
q = 4
```

面试例题 3：设置或清除特定的位。

考点：使用&和|位操作符。

出现频率：★★★★

嵌入式系统要求用户对变量或寄存器进行位操作。给定一个整型变量 a，写两段代码，第 1 个设置 a 的 BIT 3，第 2 个清除 a 的 BIT 3。在以上两个操作中，要保持其他位不变。

解析

通常情况下应聘者对这个问题有 3 种反应。

- 不知道如何下手。应聘者从没做过任何嵌入式系统的工作。
- 用 bit fields。bit fields 是 C 语言的死角，它使代码在不同编译器之间是不可移植的，同时也代码是不可重用的。
- 用#define 和 bit masks 操作。这是一种有极高可移植性的方法，应该被广泛应用。

最佳的解决方案为：

```
1    #define BIT3 (0x1 << 3)
2    static int a;
3
4    void set_bit3(void)
5    {
6        a |= BIT3;
7    }
8
9    void clear_bit3(void)
10   {
11       a &=  ~BIT3;
12   }
```

BIT3 用来计算需要操作的位，|=和&=分别用来将指定位置 1 和位置 0。

面试例题 4：计算一字节里有多少位被置 1。

考点：各种位操作符的使用。

出现频率：★★★

解析

```
1    #include <stdio.h>
2
3    #define BIT7 (0x1 << 7)
4
5    int calculate(unsigned char c)
6    {
7        int count = 0;
8        int i = 0;
9        unsigned char comp = BIT7;
```

```
10
11          for (i = 0; i < sizeof(c) * 8; i++)
12          {
13               if ((c & comp) != 0)
14               {
15                    count++;
16               }
17               comp = comp >> 1;
18          }
19
20          return count;
21     }

22     int main(int argc, char* argv[])
23     {
24          unsigned char c = 0;
25          int count = 0;
26
27          printf("c = ");
28          scanf("%d", &c);
29
30          count = calculate(c);
31          printf("count = %d\n", count);
32          return 0;
33     }
```

程序说明如下：

1 个字节（byte）有 8 位，因此首先在宏定义 BIT7 中将最高位置成 1；然后在 calculate 函数中比较每一位是否被置 1，如果是则 count++，循环结束并返回 count 的值；最后在 main 函数中进行测试，输入 97（二进制为 1100001），最后打印出来的是 count= 3。

面试例题 5：位运算改错。

考点：正确使用位运算符和逻辑运算符。

出现频率：★★★

```
1     // 函数将相应的位置 0
2     #define BIT_MASK(bit_pos) (0x01<<(bit_pos))
3
4     int Bit_Reset(unsigned int* val, unsigned char pos)
5     {
6          if (pos >= sizeof(unsigned int) * 8)
7          {
8               return 0;
9          }
10         val = (val &&  ~BIT_MASK(pos));
```

```
11        return 1;
12    }
```

解析

Bit_Reset 函数的作用是把相应的位置 0。首先是定义位掩码 BIT_MASK(bit_p os)，它需要置 0 的位是 1，其他位都是 0。然后在 Bit_Reset 函数中判断 pos 是否超出整型的字节范围，如果超出则返回 0 表示失败。最后代码第 10 行将 val 的 pos 位置 0。

值得注意的是，这里存在位运算的问题。在代码第 10 行置 0 操作中用的是“&&”操作符，它表示 “逻辑与”，只要 val 不为 0，调用完代码第 10 行之后 val 变成 1，否则为 0。这与设计函数的初衷不符。应该用“&”替换“&&”。

面试例题 6：运用位运算交换 a、b 两个数。

考点：位运算的灵活使用。

出现频率：★★★

解析

```
1     #include <stdio.h>
2
3     int main()
4     {
5         int a = 3;
6         int b = 5;
7         //进行三次异或操作
8         a ^= b;
9         b ^= a;
10        a ^= b;
11
12        printf("a = %d, b = %d\n", a, b); //打印 a、b 值
13        return 0;
14    }
```

^是位异或的运算符，用来比较两位的异同，如果相同则赋值为 0，否则为 1。

此例中 a、b 两数的初始值分别为 3 和 5，对应的二进制分别为 00000011 和 00000101。经过以下的 3 个步骤交换 2 个变量的值：

代码第 8 行，a 的二进制变为 00000110，b 仍为 00000101。即 b 不变，取出不相等的位存入 a。

代码第 9 行，a 的二进制为 00000110，b 变为 00000011。即 a 不变，取出所有不相等的位存入 b。此时 b 的值是 a 的初始值。

代码第 10 行，a 的二进制变为 00000101，b 为 00000011。即 b 不变，取出所有不相等的位存入 a。此时 a 的值为 b 的初始值。到此完成 a、b 两变量值的变换。

算法最大的优点是省略了中间变量，但只能用于相同类型数的交换。程序执行结果：

```
a = 5, b = 3
```

面试例题 7：**列举并解释 C++中的四种运算符转化，说明它们的不同点。**

考点：位运算的灵活使用。

出现频率：★★★★

解析

4 种运算符如下。

（1）const_cast 操作符：即供程序设计师在特殊情况下将限制为 const 成员函数的 const 定义解除，使其可以更改特定属性。

（2）dynamic_cast 操作符：如果启动了支持运行时间类型信息(RTTI)，dynamic_cast 有助于判断在运行时所指向对象的确定类型。它与 typeid 运算符有关，可以将一个基类的指针指向不同的子类型（派生类），然后将被转型作为基础类的对象还原成原来的类。不过，仅限于对象指针的类型转换。

（3）reinterpret_cast 操作符：将一个指针转换成其他类型的指针，新类型的指针与旧指针可以毫不相干。通常用于非标准的指针数据类型转换，例如将 void *转换为 char *。它也可以用于指针和整形数之间的类型转换。注意：它存在潜在的危险，除非有使用它的充分理由，否则就不要使用它。例如，将一个 int *类型的指针转换为 float *类型的指针，会很容易造成整数数据不能被正确地读取。

（4）static_cast 操作符：在相关的对象和指针类型之间进行类型转换。有关的类之间必须通过继承构造函数或者转换函数进行联系。static_cast 操作符还能在数字（原始的）类型之间进行类型转换。通常情况下，static_cast 操作符用在将数域较大的类型转换为较小的类型。当转换的类型是原始数据类型时，这种操作可以有效地禁止编译器发出警告。

8.2 嵌入式编程

嵌入式系统编程建立在特定的硬件平台上，要求编程语言具备较强的直接操作硬件能力。无疑，汇编语言具备这样的特质，但是由于汇编语言开发的复杂性，它并不是嵌入式系统开发的最佳选择。与之相比，C 语言作为一种“高级的低级”语言，成为嵌入式系统开发的最佳选择。因此在应聘中会着重考查有关 C 语言底层操作的知识。

面试例题 8：**用#define 声明一个常数，用以表明一年中有多少秒。**

考点：#define 的使用规范。

出现频率：★★★★

解析

```
#define SECONDS_PER_YEAR (60 * 60 * 24 * 365)UL
```

这里想考查以下几个方面。

- #define 语法的基本知识（例如不能以分号结束，括号的使用规则等）。
- 会用预处理器计算常数表达式的值，因此，直接写出计算一年中有多少秒而不是计算出实际的值，是清晰而没有代价的。
- 理解这个表达式会使 16 位机的整型数溢出，因此需要用长整型符号 L 告诉编译器这个常数是长整型数。
- 如果在表达式中用到 UL（表示无符号长整型），那么可以有一个好的起点。

面试例题 9：如何用 C 编写死循环？

考点：*死循环的编写方式。*

出现频率：★★★★

解析

这个问题有几个解决方案。第 1 个方案如下所示：

```
while(1) { }
```

第 2 个方案如下所示：

```
for(;;) { }
```

这个实现方式使语法没有确切地表达出要做的事情。如果应聘者采用这个方案，面试官可能会进一步探究应聘者这样做原因。

第 3 个方案是用 goto：

```
Loop:
...
goto Loop;
```

应聘者如果给出上面的方案，这将反映出他是一个汇编语言程序员（这也许是好事）或者他是一个想进入新领域的 BASIC/FORTRAN 程序员。

面试例题 10：如何访问特定位置的内存？

考点：*合理编写代码访问特定内存。*

出现频率：★★★★

嵌入式系统经常要求程序员去访问特定的内存位置。在某工程中，一个整形变量的绝对地址 0x67a9，请将其值设为 0xaa55，并且已知编译器是一个纯粹的 ANSI 编译器，请编写程序代码。

解析

```
int* ptr;
ptr = (int*)0x67a9;
*ptr = 0xaa55
```

还可以用下面方法，但是这种方法比较晦涩难懂。

```
*(int * const)(0x67a9) = 0xaa55;
```

建议在应聘时使用第 1 种方案。

面试例题 11：对中断服务代码的评论。

考点：对嵌入式系统中断服务的理解。

出现频率：★★★

中断是嵌入式系统中重要的组成部分，因此很多编译开发商提供使标准 C 支持中断的扩展。从而产生了一个新的关键字-interrupt。下面的代码就使用-interrupt 关键字定义中断服务子程序(ISR)，请评论一下这段代码优劣。

```
1   __interrupt double compute_area (double radius)
2   {
3       double area = PI * radius * radius;
4       printf(" Area = %f", area);
5       return area;
6   }
```

解析

这个函数有以下几方面的错误。

- ISR 不能返回一个值。
- ISR 不能传递参数。
- 在许多的处理器或编译器中，浮点数一般都是不可重入的。有些处理器或编译器需要使用额外的寄存器入栈，有些处理器或编译器是不允许在 ISR 中做浮点运算。此外，ISR 应该是短且有效率的，在 ISR 中做浮点运算是不明智的。
- printf()经常出现重入和性能上的问题。

面试例题 12：分析代码写结果——整数的自动转换。

考点：对 C 语言中整数自动转换原则的理解。

出现频率：★★★

```
1   Void foo(void)
2   {
3       unsigned int a = 6;
4       int b = -20;
5       if (a+b > 6)
6           puts("> 6");
7       else
8           puts("<= 6");
9   }
```

解析

这个问题重点测试应聘者是否懂得 C 语言中的整数自动转换原则。这无符号整型问题的答案输出是“>6”。原因是当表达式中存在有符号类型和无符号类型时所有的操作数都自动转换为无符号类型。因此“-20”变成了一个非常大的正整数，所以该表达式计算出的结果是“>6”。这一点对于频繁用到无符号数据类型的嵌入式系统来说是非常重要的。

面试例题 13：关键字 static 的作用是什么？

考点：对 C 语言中 static 关键字的理解。

出现频率：★★★★★

解析

在 C 语言中，关键字 static 有 3 个明显的作用。

- 在函数体，被声明为静态的变量在这一函数被调用的过程中维持其值不变。
- 在模块内（但在函数体外），被声明为静态的变量可以被模块内所有函数访问，但不能被模块外其他函数访问。它是一个本地的全局变量。
- 在模块内，一个被声明为静态的函数只可被这一模块内的其他函数调用。即这个函数被限制在声明它的本地范围内。

大多数应聘者能正确回答第 1 部分，另一部分应聘者能正确回答第 2 部分，同时很少的人能懂得第 3 部分。这是应聘者的一个严重的知识缺陷，因为显然应聘者不懂得本地化数据和代码范围的好处及重要性。

面试例题 14：关键字 volatile 有什么含义？

考点：对 C 语言中 volatile 关键字作用的理解。

出现频率：★★★

解析

定义为 volatile 的变量可能会被意想不到地改变，这样，编译器就不会去假设这个变量的值。准确地说，优化器在用到 volatile 变量时必须小心地重新读取该变量的值，而不是使用保存在寄存器里的备份。下面是使用 volatile 变量的 3 个例子。

- 并行设备的硬件寄存器（如：状态寄存器）。
- 一个中断服务子程序中会访问到的非自动变量(Non-automatic variables)。
- 多线程应用中被几个任务共享的变量。

面试例题 15：判断处理器使用 Big_endian 还是 Little_endian 模式存储数据。

考点：处理器的字节序以及 union 的作用。

出现频率：★★★

编写一个函数，若处理器使用 Big_endian 模式存储数据，则返回 0；若是用 Little_endian 模式存储数据，则返回 1。

解析

```
1    int checkCPU()
2    {
3        union w
4        {
```

```
5            int a;
6            char b;
7        } c;
8        c.a = 1;
9        return (c.b == 1);
10   }
```

嵌入式系统开发者应该对 Little-endian 和 Big-endian 模式非常了解。采用 Little-endian 模式的 CPU 对操作数的存储方式是从低字节到高字节，而 Big-endian 模式对操作数的存储方式是从高字节到低字节。例如，16 位的数 0x1234 在 Little-endian 模式 CPU 内存中的存放方式（假设从地址 0x4000 开始存放）为：

```
0x4000:        0x34
0x4001:        0x12
```

而在 Big-endian 模式 CPU 内存中的存放方式则为：

```
0x4000:        0x12
0x4001:        0x34
```

32 位宽的数 0x12345678 在 Little-endian 模式 CPU 内存中的存放方式（假设从地址 0x4000 开始存放）为：

```
0x4000:        0x78
0x4001:        0x56
0x4000:        0x34
0x4001:        0x12
```

而在 Big-endian 模式 CPU 内存中的存放方式则为：

```
0x4000:        0x12
0x4001:        0x34
0x4000:        0x56
0x4001:        0x78
```

联合体 union 的存放顺序是所有成员都从低地址开始存放，利用该特性，轻松地了解 CPU 对内存采用的读写模式。

面试例题 16：评价代码片断——处理器字长。

考点：认识处理器字长。

出现频率：★★★

```
unsigned int zero = 0;
unsigned int compzero = 0xFFFF;
```

解析

对于 int 型不是 16 位的处理器来说，上面的代码是不正确的。应编写如下：

```
unsigned int compzero = ~0;
```

对于这一问题的回答能真正反映出应聘者是否懂得处理器字长的重要性。优秀的嵌入式程序员能够非常准确地明白硬件的细节和它的局限性，然而 Windows 程序员往往把硬件作为一个无法避免的烦恼。

第9章 C++面向对象

C 语言是面向过程的，而 C++作为 C 语言的超集支持，它是面向对象的编程。面向对象（Object Oriented）是当前计算机界关心的重点，它是当今软件开发方法的主流。因此也是笔试时的重要考点。在面试过程中，应聘者应该对面向对象的基本概念、类、对象构造函数以及多态性等有清晰的认识，具备面向对象编程的能力。

9.1 面向对象的基本概念

面向对象其实是现实世界模型的自然延伸。现实世界中任何实体都可以看做是对象。对象之间通过消息相互作用。另外，现实世界中任何实体都归属于某类事物，任何对象都是某一类事物的实例。传统的过程式编程语言是以过程为中心、以算法为驱动，而面向对象的编程语言则是以对象为中心，以消息为驱动。用公式表示，过程式编程语言为：程序=算法+数据。面向对象编程语言为：程序=对象+消息。

面试例题 1：面向对象技术的基本概念是什么？

考点：面向对象的基本概念。

出现频率：★★★★

解析

面向对象的基本概念：按照人们认识客观世界的思维方式，采用基于对象（实体）的概念建立模型，通过面向对象的理念使计算机软件系统与现实世界中的系统一一对应。它包括下面几方面的内容。

- 类（class）：具有相似的内部状态和运动规律的实体集合。人们认识事物时主要使用两种方法：由特殊到一般的归纳法和由一般到特殊的演绎法。在归纳的过程中，从具体的事物中抽取共同的特征，形成一般的概念。在演绎的过程中，把同类的事物根据不同的特征分成不同的小类。
- 对象（object）：指现实世界中各种各样的实体，也就是类（class）的实例。它既可以

指具体的事物，也可以指抽象的事物。每个对象都有自己的内部状态和运动规律。在面向对象概念中把对象的内部状态称为属性，运动规律称为方法或事件。

- 消息（Message）：指对象间相互联系和相互作用的方式。一个消息主要由 5 部分组成，发送消息的对象、接收消息的对象、消息传递办法、消息内容（参数）、反馈。
- 类的特性：抽象、继承、封装、重载、多态。

答案

面向对象是指按人们认识客观世界的思维方式，采用基于对象（实体）的概念建立模型。

面试例题 2：判断题——类的基本概念。

考点：面向对象的基本概念。

出现频率：★★★★

对于类，下面哪一个是不正确的？

A．类是对象的设计蓝图。

B．使用关键字 Class 创建类结构。

C．类被声明后，类名成为类型名并且可以用来声明变量。

D．类与结构体相同，它们之间没有区别。

解析

A 正确。因为对象是类的实例，只有类设计好了，对象才可以被创建。

B 正确。Class 是“类型”的意思。不仅在 C++中使用 class 创建类，而且在 Java 和 C#里都使用 class 关键字创建类。

C 正确。一个类一旦被声明了，这个类名就成为一个类型名并可以使用它来声明变量。

D 错误。因为类的访问权限是 private，而结构体的访问权限是 public。

答案

D

面试例题 3：选择题——与 C 相比，C++的改进。

考点：C++的改进点。

出现频率：★★★★

C++是从早期的 C 语言逐渐发展演变来的，与 C 语言相比，它在求解问题方法上最大的改进是什么？

A．面向过程　　　B．面向对象

C．安全性　　　　D．复用性

解析

C++是从 C 语言发展演变来的。C 语言是过程式编程语言，它是以过程为中心、以算法为驱动。而 C++能够使用面向对象的编程方式，即以对象为中心，以消息为驱动。这是 C++在 C

语言基础上的最大改进。

答案

B

9.2 class和struct的区别

struct是C语言的关键字，C++中引用了class，并且为了与C语言兼容，保留了struct关键字。但是C语言的struct与C++的class和struct有着很大的区别。

面试例题4：class和struct有什么区别？

考点：class和struct的区别。

出现频率：★★★★

解析

这里有两种情况下的区别。

- C的struct与C++的class的区别。
- C++中的struct和class的区别。

在第1种情况下，struct与class有着非常明显的区别。C是一种过程化的语言，struct作为一种复杂数据类型定义只能定义成员变量，不能定义成员函数。例如下面的C程序片断：

```
1   struct Point
2   {
3       int x; // 合法
4       int y; // 合法
5       void print()
6       {
7           printf("Point print\n"); //编译错误
8       };
9   };
```

这里代码第7行会出现编译错误，提示错误消息如下："函数不能作为Point结构体的成员"。因此在第1种情况下struct只是一种数据类型，不能面向对象编程。现在来看第2种情况。程序代码如下：

```
1   #include <iostream>
2   using namespace std;
3
4   class CPoint
5   {
6       int x;              //默认为private
7       int y;              //默认为private
8       void print()        //默认为private
```

```
9          {
10             cout << "CPoint: (" << x << ", " << y << ")" << endl;
11         }
12     public:
13         CPoint(int x, int y)        //构造函数，指定为 public
14         {
15             this->x = x;
16             this->y = y;
17         }
18         void print1()
19         {
20             cout << "CPoint: (" << x << ", " << y << ")" << endl;
21         }
22     };
23
24     struct SPoint
25     {
26         int x;                  //默认为 public
27         int y;                  //默认为 public
28         void print()            //默认为 public
29         {
30             cout << "SPoint: (" << x << ", " << y << ")" << endl;
31         }
32         SPoint(int x, int y)   //构造函数,默认为 public
33         {
34             this->x = x;
35             this->y = y;
36         }
37     private:
38         void print1()          //private 类型的成员函数
39         {
40             cout << "SPoint: (" << x << ", " << y << ")" << endl;
41         }
42     };
43
44     int main(void)
45     {
46         CPoint cpt(1, 2);   //调用 CPoint 带参数的构造函数
47         SPoint spt(3, 4);   //调用 SPoint 带参数的构造函数
48
49         cout << cpt.x << " " << cpt.y << endl;   //编译错误
50         cpt.print();          //编译错误
51         cpt.print1();         //合法
52
53         spt.print();          //合法
```

```
54        spt.print1();        //编译错误
55        cout << spt.x << " " << spt.y << endl;   //合法
56
57        return 0;
58    }
```

在上面的程序里，struct 包含构造函数和成员函数，其实它还拥有 class 的其他特性，例如继承、虚函数等。因此 C++中的 struct 扩充了 C 的 struct 功能。那么它们有什么不同呢？

Main()函数的编译错误全部是因为访问 private 成员而产生的。因此可以看到 class 中默认的成员访问权限是 private 的，而 struct 中则是 public 的。在类的继承方式上，struct 和 class 又有什么区别？请看下面的程序：

```
1     #include <iostream>
2     using namespace std;
3
4     class CBase
5     {
6     public:
7         void print()                    //public 成员函数
8         {
9             cout << "CBase: print()..." << endl;
10        }
11    };
12
13    class CDerived1 : CBase           //默认 private 继承
14    {
15    };
16
17    class CDerived2 : public Cbase    //指定 public 继承
18    {
19    };
20
21    struct SDerived1 : Cbase          //默认 public 继承
22    {
23    };
24
25    struct SDerived2 : private Cbase  //指定 public 继承
26    {
27    };
28
29    int main()
30    {
31        CDerived1 cd1;
32        CDerived2 cd2;
33        SDerived1 sd1;
34        SDerived2 sd2;
35
```

```
36        cd1.print();    //编译错误
37        cd2.print();
38        sd1.print();
39        sd2.print();    //编译错误
40
41        return 0;
42    }
```

可以看出，以 private 方式继承父类的子类对象不能访问父类的 public 成员。class 继承默认是 private 继承，而 struct 继承默认是 public 继承。

另外，在 C++模板中，类型参数前面可以使用 class 或 typename，如果使用 struct，则含义不同，struct 后面跟的是“non-type template parameter”，而 class 或 typename 后面跟的是类型参数。

事实上，C++中保留 struct 的关键字是为了使 C++编译器能够兼容 C 开发的程序。

答案

分两种情况进行分析。

- C 的 struct 与 C++的 class 的区别：struct 只是作为一种复杂数据类型定义，不能用于面向对象编程。
- C++中的 struct 和 class 的区别：对于成员访问权限以及继承方式，class 中默认的是 private，而 struct 中则是 public。class 还可以用于表示模板类型，struct 则不行。

面试例题 5：改错——C++类对象的声明。

考点：C++类对象的声明方法。

出现频率：★★★

```
1    struct Test
2    {
3        Test( int ) {}
4        Test() {}
5        void fun() {}
6    };
7
8    void main( void )
9    {
10       Test a(1);
11       a.fun();
12       Test b();
13       b.fun();
14   }
```

答案

题中的 Test 有两个构造函数，一个是带参数的，另一个是不带参数的。在调用不带参数的构造函数时不需要加小括号，因此代码第 12 行是错误的。应该改为：

```
Test b;//去掉小括号
```

9.3 成员变量

C++中类的成员有着不同的类型，例如普通类型、static类型、const类型。了解这些类型成员的作用、区别以及对它们初始化的方法是十分必要的。

面试例题6：分析代码写结果——C++类成员的访问。

考点：类对象的私有成员函数不能用对象访问。

出现频率：★★★

请指出下面程序中标记为（1）（2）（3）的代码哪一个是错误的。

```
1   #define public private          // (1)
2
3   class Animal
4   {
5   public:                         //(2)
6       void MakeNoise();
7   };
8
9   int main(void)
10  {
11      Animal animal;
12      animal.MakeNoise();         // (3)
13      return 0;
14  }
```

A.（1） B.（2） C.（3） D.（1）（2）（3）

解析

（1）正确。把public宏定义为private。

（2）正确。注意由于public已经被定义为private，因此这里的MakeNoise()成员函数实际上是private的。

（3）错误。不能调用Animal对象的私有成员函数。

答案

C

面试例题7：找错——类成员的初始化。

考点：初始化列表的构造顺序。

出现频率：★★★

```
1   #include <iostream>
2   using namespace std;
```

```
3
4	class Obj
5	{ public:
6		Obj(int k) : j(k), i(j)
7		{
8		}
9		void print(void)
10		{
11			cout << i << endl << j << endl;
12		}
13	private:
14		int i;
15		int j;
16	};
17
18	int main(int argc, char *argv[])
19	{
20		Obj obj(2);
21		obj.print();
22
23		return 0;
24	}
```

解析

本题考查的是初始化列表方面的知识。这里很容易让人误认为先用 2 对 j 进行初始化，然后再用 j 对 i 进行初始化，那么 i 和 j 都是 2。

实际上，初始化的顺序正好与想象中的相反。

初始化列表的初始化顺序与变量声明的顺序一致，而不是按照出现在初始化列表中的顺序。这里成员 i 比成员 j 先声明，因此正确的顺序是先用 j 对 i 进行初始化，然后再用 2 对 j 进行初始化。由于在对 i 进行初始化时 j 尚未被初始化，j 的值为随机值，故 i 的值也为随机值；然后用 2 对 j 进行初始化，j 的值为 2。

答案

i 为随机值，j 为 2。程序输出结果：

```
-858993460
2
```

面试例题 8：分析代码写结果——静态成员变量的使用。

考点：静态成员变量的理解和使用。

出现频率：★★★★

```
1	#include <iostream>
2	using namespace std;
3
4	class Myclass
```

```
5   {
6   public:
7       Myclass(int a, int b, int c);
8       void GetNumber();
9       void GetSum();
10  private:
11      int A;
12      int B;
13      int C;
14      int Num;
15      static int Sum;
16  };
17
18  int Myclass::Sum = 0;
19
20  Myclass::Myclass(int a, int b, int c)
21  {
22      A = a;
23      B = b;
24      C = c;
25      Num = A+B+C;
26      Sum = A+B+C;
27  }
28
29  void Myclass::GetNumber()
30  {
31      cout << "Number = " << Num << endl;
32  }
33
34  void Myclass::GetSum()
35  {
36      cout << "Sum = " << Sum << endl;
37  }
38
39  void main()
40  {
41      Myclass M(3, 7, 10),N(14, 9, 11);
42
43      M.GetNumber();
44      N.GetNumber();
45      M.GetSum();
46      N.GetSum();
47  }
```

解析

本题考查的是静态成员与非静态成员的区别。静态成员是该类类型的全局变量。对于非静态成员，每个类对象都有自己的复制品，而静态成员对每个类的类型只有一个复制品。静态成员只有一个，由该类类型的所有对象共享访问。

Myclass 类有 GetNumber()和 GetSum()两种方法，它们分别输出成员变量 Num 和 Sum 的值。main()函数中定义了两个 Myclass 的对象，并分别调用它们的 GetNumber()和 GetSum()。

Num 成员为普通类型，它被 Myclass 类的对象所有。因此两次打印出来的值不一样。

Sum 成员为静态类型，它被 Myclass 类所有，也被 Myclass 类的所有对象所共享。因此两次打印出来的值是相同的。

答案

程序输出结果为：

```
Number = 20
Number = 34
Sum = 34
Sum = 34
```

面试例题 9：与全局对象相比，使用静态数据成员有什么优势？

考点：对静态成员变量的理解。

出现频率：★★★★

答案

主要有以下所述两种优势。

- 静态数据成员没有进入程序的全局名字空间，因此不存在程序中其他全局名字冲突的问题。
- 使用静态数据成员可以隐藏信息。因为静态成员可以是 private 成员，而全局对象不能。

面试例题 10：有哪几种情况只能用初始化列表（intialization list），而不能用赋值（assignment）？

考点：初始化列表和赋值的区别。

出现频率：★★★

解析

无论是在构造函数初始化列表中初始化成员，还是在构造函数体中对它们赋值，最终结果都是相同的。不同之处在于，构造函数在初始化列表初始数据成员，没有定义初始化列表的构造函数在构造函数体中对数据成员赋值。

一种情况是，const 和 reference 类型成员变量只能被初始化而不能做赋值操作。

另一种情况是，类的构造函数需要调用其基类的构造函数。请看下面的程序：

```
1    #include <iostream>
2    using namespace std;
3
4    class A                          //A 是父类
5    {
6    private:
7        int a;                       //private 成员
8    public:
```

```
9          A() {}
10         A(int x):a(x) {}              //带参数的构造函数对 a 初始化
11         void printA()                 //打印 a 的值
12         {
13             cout << "a = " << a << endl;
14         }
15     };
16
17     class B : public A //B 是子类
18     {
19     private:
20         int b;
21     public:
22         B(int x, int y) : A(x)        //需要初始化 b 以及父类的 a
23         {
24             a = x;                //a 为 private，无法在子类被访问，编译错误
25             A(x);                 //调用方式错误，编译错误
26             b = y;
27         }
28         void printB() //打印 b 的值
29         {
30             cout << "b = " << b << endl;
31         }
32     };
33
34     int main()
35     {
36         B b(2,3);
37
38         b.printA();                   //调用子类的 printA()
39         b.printB();                   //调用自己的 printB()
40
41         return 0;
42     }
```

从上面的程序可以看出，如果在子类的构造函数中需要初始化父类的 private 成员，直接对其赋值是不行的（代码第 24 行），只有调用父类的构造函数才能完成对它的初始化。但在函数体内调用父类的构造函数也是不合法的（代码第 25 行），只能采取代码第 22 行中的初始化列表调用子类构造函数的方式。程序的执行结果如下：

```
a = 2
b = 3
```

答案

当类中含有 const、reference 成员变量以及基类的构造函数都需要初始化列表。

面试例题 11：代码改错——静态成员的使用。

考点：静态成员与非静态成员的理解。

出现频率：★★★★

```
1   #include <iostream>
2   using namespace std;
3
4   class test
5   {
6   public:
7       static int i;
8       int j;
9       test(int a) : i(1), j(a) {}
10      void func1();
11      static void func2();
12  };
13
14  void test::func1() { cout << i << ", " << j << endl; }
15
16  void test::func2() { cout << i << ", " << j << endl; }
17
18  int main()
19  {
20      test t(2);
21      t.func1();
22      t.func2();
23      return 0;
24  }
```

解析

这个程序会出现如下所示的错误。

- 代码第 9 行，不能初始化 i。这是关于静态成员变量 i 的初始化问题。为了与非静态成员变量相区别，i 不能在类内部被初始化。可以把 i 放在类定义外面初始化（例如在代码 13 行初始化）。
- 代码第 16 行，在静态成员函数中非法引用了数据成员 test::j。这是关于静态成员函数访问非静态成员的错误。要知道，静态成员函数和静态成员变量一样，不属于类的对象，因此不含 this 指针，也就无法调用类的非静态成员。

正确的程序代码如下所示。

```
1   #include <iostream>
2   using namespace std;
3
4   class test
5   {
6   public:
7       static int i;
8       int j;
9       test(int a) : j(a) {}
10      void func1();
```

```
11        static void func2();
12    };
13    int test::i = 1;
14    void test::func1() { cout << i << ", " << j << endl; }
15
16    void test::func2() { cout << i << /*", " << j <<*/ endl; }    //注释对 j 的调用
17
18    int main()
19    {
20        test t(2);
21        t.func1();
22        t.func2();
23        return 0;
24    }
```

程序执行结果为：

```
1    1, 2
2    1
```

面试例题 12：选择题——对静态数据成员的正确描述。

考点：静态数据成员的理解和使用。

出现频率：★★★

下面对静态数据成员的描述中，正确的选项是哪一个？

A．静态数据成员可以在类体内进行初始化；

B．静态数据成员不可以被类的对象调用；

C．静态数据成员不受 private 控制符作用；

D．静态数据成员可以直接用类名调用。

答案

A 错误。静态数据成员必须在类外面初始化，以示与普通数据成员的区别。

B 错误。静态数据成员是类的成员，它为类的所有的对象所共享。

C 正确。

D 正确。

9.4 构造函数和析构函数

9.4.1 构造函数

构造函数是一种特殊的类成员函数，它的作用是对类的数据成员进行初始化和分配内存。构造函数要和所属类具有相同的名称，当创建一个类的对象时，类的构造函数将自动被调用。

面试例题 13：main 函数执行以前，会执行什么代码？

考点：对构造函数调用期的理解。

出现频率：★★★

解析

请看下面的程序代码：

```
1   #include <iostream>
2   using namespace std;
3
4   class Test
5   {
6   public:
7       Test()          //构造函数
8       {
9           cout << "constructor of Test" << endl;
10      }
11  };
12
13  Test a;             //全局变量
14
15  int main()
16  {
17      cout << "main() start" << endl;
18      Test b;         //局部变量
19      return 0;
20  }
```

程序输出结果：

```
constructor of Test
main() start
constructor of Test
```

显然，程序的执行顺序是：首先构造全局对象 a，其次进入 main 函数，最后构造局部对象 b。

答案

全局对象的构造函数在 main 函数之前执行。

面试例题 14：C++中的空类，默认情况下会产生哪些类成员函数？

考点：编译器对 C++类的默认处理。

出现频率：★★★★★

解析

对于一个 C++的空类，例如 Empty：

```
class Empty
{
};
```

虽然 Empty 类定义中没有任何成员，但为了进行一些默认的操作，编译器会加入以下成员函数，这些成员函数使得类的对象拥有一些通用的功能。

- 默认构造函数和复制构造函数，它们被用于类的对象的构造过程。
- 析构函数，它被用于类的对象的析构过程。
- 赋值函数，它被用于同类的对象间的赋值过程。
- 取值运算，当对类的对象进行取地址（&）时，此函数被调用。

由上可知即使程序没有定义类的任何成员，编译器也会插入一些函数，完整的 Empty 类定义如下所示：

```
1   class Empty
2   {
3   public:
4   Empty();  // 缺省构造函数
5   Empty( const Empty& );                   //复制构造函数
6   ~Empty();                                //析构函数
7   Empty& operator=( const Empty& );    //赋值运算符
8   Empty* operator&();  // 取址运算符
9   const Empty* operator&() const;       //取址运算符 const
10  };
```

答案

C++的空类中，默认情况下会产生默认构造函数、复制构造函数、析构函数、赋值函数以及取值运算。

面试例题 15：构造函数和析构函数是否可以被重载？

考点：构造函数和析构函数的理解。

出现频率：★★★★

答案

构造函数可以被重载，因为构造函数可以有多个，而且可以带参数。

析构函数不可以被重载。因为析构函数只能有一个，并且不能带参数。

面试例题 16：分析代码——重载构造函数的调用。

考点：重载构造函数的调用。

出现频率：★★★★

```
1   class Test
2   {
3   public:
4       Test() {}
5       Test(char *Name, int len = 0) { }
6       Test(char *Name) { }
7   };
8   int main()
```

```
9    {
10       Test obj("Hello");
11       return 0;
12   }
```

A．将会产生运行时错误。

B．将会产生编译错误。

C．将会执行成功。

D．以上说法都不正确。

解析

Test 类定义了两个构造函数。当编译到代码第 10 行时，由于构造函数的模糊语义，编译器无法决定调用哪一个构造函数，因此会产生编译错误。

如果把代码第 10 行注释掉，编译器将不会产生错误。因为 C++编译器认为潜在的二义性不是一种错误。

答案

B

面试例题 17：分析代码——构造函数的使用。

考点：构造函数的使用。

出现频率：★★★★

以下代码中的输出语句输出 0 吗，为什么？

```
1    #include <iostream>
2    using namespace std;
3
4    struct CLS
5    {
6        int m_i;
7         CLS( int i ) : m_i(i) {}
8         CLS()
9         {
10            CLS(0);
11        }
12   };
13   int main()
14   {
15       CLS obj;
16       cout << obj.m_i << endl;
17       return 0;
18   }
```

解析

在代码第 10 行，不带参数的构造函数直接调用了带参数的构造函数。这种调用，被很多人

误以为可以通过构造函数的重载和相互调用实现一些类似默认参数的功能，其实是不行的，而且还会有副作用。下面加几条打印对象地址的语句到原来的程序中：

```
1   #include <iostream>
2   using namespace std;
3
4   struct CLS
5   {
6       int m_i;
7       CLS( int i ) : m_i(i)
8       {
9           cout << "CLS(): this = " << this << endl;
10  }
11      CLS()
12      {
13          CLS(0);
14          cout << "CLS(int): this = " << this << endl;
15      }
16
17  };
18
19  int main()
20  {
21      CLS obj;
22      cout << "&obj = " << &obj << endl;
23      cout << obj.m_i << endl;
24      return 0;
25  }
```

程序执行结果如下：

```
CLS(): this = 0012FF20
CLS(int): this = 0012FF7C
&obj = 0012FF7C
-858993460
```

可以看到，在带参数的构造函数里打印出来的对象地址和对象 obj 的地址不一致。实际上代码第 13 行的调用只是在栈上生成一个临时对象，对于自己毫无影响。还可以发现，构造函数的互相调用引起的后果不是死循环，而是栈溢出。

答案

输出不为 0，而是个随机数。

原因是构造函数内调用构造函数只是在栈上生成了一个临时对象，对于其毫无影响。

面试例题 18：构造函数 explicit 与普通构造函数的区别。

考点：explicit 构造函数的作用

出现频率：★★★

解析

explicit 构造函数是用来防止隐式转换的。请看下面的代码：

```
1	class Test1
2	{
3	public:
4		Test1(int n) { num = n; }		//普通构造函数
5	private:
6		int num;
7	};
8	
9	class Test2
10	{
11	public:
12		explicit Test2(int n) { num = n; }	//explicit(显式)构造函数
13	private:
14		int num;
15	};
16	
17	int main()
18	{
19		Test1 t1 = 12;			//隐式调用其构造函数，成功
20		Test2 t2 = 12;			//编译错误,不能隐式调用其构造函数
21		Test2 t3(12);			//显示调用成功
22		return 0;
23	}
```

Test1 的构造函数带一个 int 型的参数，代码第 19 行会隐式转换成调用 Test1 的这个构造函数 Test2 的构造函数被声明为 explicit，这表示不能通过隐式转换来调用这个构造函数，因此代码第 20 行会出现编译错误。

答案

普通构造函数能够被隐式调用，而 explicit 构造函数只能被显示调用。

面试例题 19：分析代码——explicit 构造函数的作用。

考点：explicit 构造函数的作用。

出现频率：★★★

下面的程序函数 show()被调用时，输出结果是什么？

```
1	#include <iostream>
2	#include <string>
3	using namespace std;
4	
5	class Number
6	{
7	public:
```

```
8        string type;
9        Number(): type("void") { }
10       explicit Number(short) : type("short") { }
11       Number(int) : type("int") { }
12   };
13
14   void Show(const Number& n) { cout << n.type; }
15
16   void main()
17   {
18       short s = 42;
19       Show(s);
20   }
```

解析

Show()函数的参数类型是 Number 类对象的引用，代码第 19 行调用 Show(s)时采取了以下所示步骤。

- Show(s)中的 s 为 short 类型，其值为 42，因此首先检查参数为 short 的构造函数能否被隐式转换，由于代码 10 行的构造函数被声明为显式调用，因此不能隐式转换。于是进入下一步操作。
- 42 自动转换为 int 类型。
- 检查参数为 int 的构造函数能否被隐式转换，由于代码第 11 行参数为 int 的构造函数没有被声明为显式调用，因此调用此构造函数构造出一个临时对象。
- 打印上一步临时对象的 type 成员。

答案

C

9.4.2　析构函数

析构函数是与构造函数互补的用户自定义成员函数，在对象最后一次使用之后它会被自动应用在每个类对象上。析构函数主要用于释放类的构造函数在整个生命期中获得的资源。

面试例题 20：C++中虚析构函数的作用是什么？

考点：虚析构函数的理解。

出现频率：★★★

解析

析构函数是为了在对象不被使用之后释放它的资源，虚函数是为了实现多态。那么把析构函数声明为 vitual 有什么作用呢？请看下面的程序代码：

```
1    #include <iostream>
2    using namespace std;
3
```

```
4   class Base
5   {
6   public:
7       Base() {};          //Base 的构造函数
8       ~Base()             //Base 的析构函数
9       {
10          cout << "Output from the destructor of class Base!" << endl;
11      };
12      virtual void DoSomething()
13      {
14          cout << "Do something in class Base!" << endl;
15      };
16  };
17
18  class Derived : public Base
19  {
20  public:
21      Derived() {};       //Derived 的构造函数
22      ~Derived()          //Derived 的析构函数
23      {
24          cout << "Output from the destructor of class Derived!" << endl;
25      };
26      void DoSomething()
27      {
28          cout << "Do something in class Derived!" << endl;
29      };
30  };
31
32  int main()
33  {
34      Derived *pTest1 = new Derived();    //Derived 类的指针
35      pTest1->DoSomething();
36      delete pTest1;
37
38      cout << endl;
39
40      Base *pTest2 = new Derived();       //Base 类的指针
41      pTest2->DoSomething();
42      delete pTest2;
43
44      return 0;
45  }
```

程序输出结果：

```
Do something in class Derived!
Output from the destructor of class Derived!
Output from the destructor of class Base!
```

```
Do something in class Derived!
Output from the destructor of class Base!
```

代码第36行释放pTest1的资源，而代码第42行不能释放pTest2的资源，因为从结果可以看Derived类的析构函数并没有被调用。通常情况下类的析构函数里面都是释放内存资源，而析构函数不被调用就会造成内存泄漏。原因是指针pTest2是Base类型的指针，释放pTest2时只进行Base类的析构函数。在代码第8行前面加上virtual关键字后的运行结果如下：

```
Do something in class Derived!
Output from the destructor of class Derived!
Output from the destructor of class Base!

Do something in class Derived!
Output from the destructor of class Derived!
Output from the destructor of class Base!
```

此时释放指针pTest2，由于Base的析构函数是virtual，因此先执行Derived类的析构函数，然后再执行Base类的析构函数，这样资源正常释放，避免了内存泄漏。

由此可知，只有当一个类被用来作为基类的时候，才会把析构函数写成虚函数。

面试例题21：分析代码写结果——析构函数的执行顺序。

考点：构造函数的执行顺序与构造函数相反。

出现频率：★★★★

```
1   #include<iostream.h>
2   class A
3   {
4   private:
5       int a;
6
7   public:
8       A(int aa) { a = aa; };
9       ~A() { cout<<"Destructor A!"<<a<<endl; };
10  };
11
12  class B:public A
13  {
14  private:
15      int b;
16
17  public:
18      B( int aa = 0, int bb = 0 ):A(aa) { b = bb; };
19      ~B(){ cout<<"Destructor B!"<<b<<endl; };
20  };
21
22  void main()
```

```
23    {
24          B obj1(5), obj2(6, 7);
25          return;
26    };
```

解析

本题考查的是析构函数的执行顺序。析构函数的执行顺序与构造函数的执行顺序相反。

main()函数中定义了两个类 B 的对象，它们的基类是 A。由于这两个对象都是栈中分配的，当 main()函数退出时会发生析构，又因为 obj1 比 obj2 先声明，所以 obj2 先析构。它们析构的顺序是首先执行 B 的析构函数，然后再执行 A 的析构函数。

答案

程序输出如下：

```
Destructor B!7
Destructor A!6
Destructor B!0
Destructor A!5
```

9.5 复制构造函数和赋值函数

9.5.1 复制构造函数

可以用类对象初始化另一个对象，默认是通过依次复制每个非静态数据成员来实现的，也可以通过提供特殊的复制构造函数（copy constructor）来改变默认行为。

面试例题 22：复制构造函数是什么？什么情况下会用到它？什么是深复制和浅复制？

考点：复制构造函数的理解。

出现频率：★★★

解析

首先来解释什么是复制构造函数，以及使用它的场合。

复制构造函数是一种特殊的构造函数，它由编译器调用来完成一些基于同一类的其他对象的构造及初始化。

如果在类中没有显式声明一个复制构造函数，那么，编译器会私下里制定一个函数来进行对象之间的位复制（bitwise copy）。这个隐含的复制构造函数简单地关联了所有的类成员。

在 C++中，3 种对象需要复制，此时复制构造函数将会被调用。

- 一个对象以值传递的方式传入函数体。
- 一个对象以值传递的方式从函数返回。
- 一个对象需要通过另外一个对象进行初始化。

下面的程序代码说明了上述 3 种情况。

```
1   #include <iostream>
2   using namespace std;
3
4   class Test
5   {
6   public:
7       int a;
8       Test(int x)
9       {
10          a = x;
11      }
12      Test(Test &test)        //复制构造函数
13      {
14          cout << "copy constructor" << endl;
15          a = test.a;
16      }
17  };
18
19  void fun1(Test test)        //(1)值传递传入函数体
20  {
21      cout << "fun1()..." << endl;
22  }
23
24  Test fun2()                 //(2)值传递从函数体返回
25  {
26      Test t(2);
27      cout << "fun2()..." << endl;
28      return t;
29  }
30
31  int main()
32  {
33      Test t1(1);
34      Test t2 = t1;           //(3)用 t1 对 t2 做初始化
35      cout << "before fun1()..." << endl;
36      fun1(t1);
37
38      Test t3 = fun2();
39      cout << "after fun2()..." << endl;
40      return 0;
41  }
```

程序执行结果如下：

```
copy constructor
before fun1()...
```

```
copy constructor
fun1()...
fun2()...
copy constructor
copy constructor
after fun2()...
```

fun1()、fun2()以及代码第 34 行分别对应了上面 3 种调用复制构造函数的情况。

接下来说明深复制与浅复制。

既然系统会自动提供一个默认的复制构造函数来处理复制，那为什么要去自定义复制构造函数呢？下面的程序代码说明了这个问题。

```
1   #include <iostream>
2   using namespace std;
3
4   class Test
5   {
6   public:
7       char *buf;
8       Test(void)              //不带参数的构造函数
9       {
10          buf = NULL;
11      }
12      Test(const char* str)   //带参数的构造函数
13      {
14          buf = new char[strlen(str) + 1];     //分配堆内存
15          strcpy(buf, str);    //复制字符串
16      }
17      ~Test()
18      {
19          if (buf != NULL)
20          {
21              delete buf;   //释放 buf 指向的堆内存
22              buf = NULL;
23          }
24      }
25  };
26
27  int main()
28  {
29      Test t1("hello");
30      Test t2 = t1;           //调用默认的复制构造函数
31
32      cout << "(t1.buf == t2.buf) ? " <<
33          (t1.buf == t2.buf ? "yes": "no") << endl;
34
35      return 0;
36  }
```

这里 Test 类的 buf 成员是一个字符指针，在带参数的构造函数中为它分配了一块堆内存来存放字符串，然后在析构函数中又将堆内存释放。在 main()函数（代码第 30 行）使用了对象复制，因此会调用默认的复制构造函数。程序的执行结果如下：

```
(t1.buf == t2.buf) ? yes
程序崩溃
```

这里程序崩溃发生在 main()函数退出对象析构的时候。由上面的打印结果可以看出，默认复制构造函数只是简单地把两个对象的指针作赋值运算，它们指向的是同一个地址，产生两次析构，释放同一块堆内存时发生崩溃。可以在 Test 类里通过添加一个自定义的复制构造函数解决两次析构的问题：

```
1    Test(Test &test)
2    {
3        buf = new char[strlen(test.buf) + 1];
4        strcpy(buf, test.buf);
5    }
```

程序执行结果如下：

```
(t1.buf == t2.buf) ? no
```

由于此时 buf 又分配了一块堆内存来保存字符串，t1 的 buf 和 t2 的 buf 分别指向不同的堆内存，析构时就不会发生程序崩溃。

总结：如果复制的对象中引用了某个外部的内容（例如分配在堆上的数据），那么在复制这个对象的时候，让新旧两个对象指向同一个外部的内容，就是浅复制；如果在复制这个对象的时候为新对象制作了外部对象的独立复制，就是深复制。

答案

复制构造函数是一种特殊的构造函数，它由编译器调用来完成一些基于同一类的其他对象的构造及初始化。

浅复制是让新旧两个对象指向同一个外部的内容，而深复制是指为新对象制作了外部对象的独立复制。

面试例题 23：什么时候编译器会生成默认的 copy constructor 呢？如果已经写了一个构造函数，编译器还会生成 copy constructor 吗？

考点：复制构造函数的理解。

出现频率：★★★

答案

如果用户没有自定义复制构造函数，并且在代码中使用到了复制构造函数，编译器就会生成默认的复制构造函数；但是如果用户定义了复制构造函数，编译器就不会再生成复制构造函数。

如果用户定义了一个构造函数，但不是复制构造函数，而此时在代码中又使用到了复制构造函数，编译器也还会生成默认的复制构造函数。

面试例题 24：写一个继承类的复制函数。

考点：继承类的复制构造函数的理解。

出现频率：★★★

解析

如果基类没有私有成员，即所有成员都能被派生类访问，则派生类的复制构造函数很容易编写。但如果基类有私有成员，并且这些私有成员必须在调用派生类的复制构造函数时初始化，在这种情况下又该怎么做呢？

编写继承类的复制函数有一个原则：使用基类的复制构造函数。这个原则其实就是解决上述问题的方案。请看下面的程序：

```
1   #include <iostream>
2   using namespace std;
3
4   class Base
5   {
6   public:
7       Base():i(0) {cout << "Base()" << endl;}                 //默认普通构造函数
8       Base(int n):i(n) {cout << "Base(int)" << endl;}         //普通构造函数
9       Base(const Base &b) :i(b.i)   //复制构造函数
10      {
11          cout << "Base(Base&)" << endl;
12      }
13  private:
14      int i;   //私有成员
15  };
16
17  class Derived : public Base
18  {
19  public:
20      Derived():Base(0), j(0) {cout << "Derived()" << endl;}   //默认普通构造函数
21      Derived(int m, int n):Base(m), j(n) {cout << "Derived(int)" << endl;}   //普通构造函数
22      Derived(Derived &obj) : Base(obj), j(obj.j)             //Derived 类的复制构造函数
23      {                                                         //调用了 Base 的复制构造函数
24          cout << "Derived(Derived&)" << endl;
25      }
26  private:
27      int j;
28  };
29
30  int main()
31  {
32      Base b(1);
33      Derived obj(2, 3);
```

```
34        cout << "---------------" << endl;
35        Derived d(obj);                                     //调用 Derived 的复制构造函数
36        cout << "---------------" << endl;
37        return 0;
38    }
```

Derived 类继承自 Base 类，因此在 Derived 类内不能使用 obj.i 或 Base::i 的方式访问 Base 的私有成员 i。很明显，其复制构造函数只能使用 Base(obj)（代码第 22 行）的方式调用其基类的复制构造函数来给基类的私有成员 i 初始化。

9.5.2 赋值函数

当一个类对象向该类的另一个对象的赋值操作时，需要用到该类的赋值函数。

面试例题 25：复制构造函数与赋值函数有什么区别？

考点：构造函数与赋值函数的区别。

出现频率：★★★

解析

构造函数与赋值函数有以下 3 个区别。

- 复制构造是一个对象初始化一块内存区域，这块内存就是新对象的内存区；例如：

```
1    class A;
2    A a;
3    A b=a;  //复制构造函数调用
4
5    A b(a);  //复制构造函数调用
```

而赋值函数是对于一个已经被初始化的对象来进行赋值操作。例如：

```
1    class A;
2    A a;
3    A b;
4    b = a;    //赋值函数调用
```

- 一般来说在数据成员包含指针对象的时候，应考虑两种不同的处理需求：一种是复制指针对象，另一种是引用指针对象。复制构造函数大多数情况下是复制，赋值函数则是引用对象。
- 实现不一样。复制构造函数首先是一个构造函数，它调用的时候是通过参数的对象初始化产生一个对象。赋值函数则是把一个新的对象赋值给一个原有的对象，所以如果原来的对象中有内存分配要先把内存释放掉，而且还要检查一下两个对象是不是同一个对象，如果是，则不做任何操作。

面试例题 26：编写类 String 的构造函数、析构函数和赋值函数。

考点：构造函数、析构函数和赋值函数的编写方法。

出现频率：★★★★★

已知类 String 的原型如下所示：

```
1   class String
2   {
3   public:
4       String(const char *str = NULL);         //普通构造函数
5       String(const String &other);            //复制构造函数
6       ~ String(void);                          //析构函数
7       String & operator =(const String &other);   //赋值函数
8   private:
9       char *m_String;                          //私有成员，保存字符串
10  };
```

解析

程序代码如下：

```
1   #include <iostream>
2   using namespace std;
3   
4   class String
5   {
6   public:
7       String(const char *str = NULL);               //普通构造函数
8       String(const String &other);                  //复制构造函数
9       ~ String(void);                                //析构函数
10      String & operator =(const String &other);   //赋值函数
11  private:
12      char *m_String;      //私有成员，保存字符串
13  };
14  
15  String::~String(void)
16  {
17      cout << "Destructing"<< endl;
18      if (m_String != NULL)                        //如果 m_String 不为 NULL，释放堆内存
19      {
20          delete [] m_String;
21          m_String = NULL;                         //释放后置为 NULL
22      }
23  }
24  
25  String::String(const char *str)
26  {
27      cout << "Construcing" << endl;
28      if(str == NULL)                              //如果 str 为 NULL，存空字符串""
29      {
30          m_String = new char[1];                  //分配一个字节
31          *m_String = '\0';                        //将之赋值为字符串结束符
```

```
32          }
33          else
34          {
35              m_String = new char[strlen(str) + 1];   //分配空间容纳 str 内容
36              strcpy(m_String, str);                  //复制 str 到私有成员
37          }
38      }
39
40      String::String(const String &other)
41      {
42          cout << "Constructing Copy" << endl;
43          m_String = new char[strlen(other.m_String) + 1];   //分配空间容纳 str 内容
44          strcpy(m_String, other.m_String);                  //复制 str 到私有成员
45      }
46
47      String & String::operator = (const String &other)
48      {
49          cout << "Operate = Function" << endl;
50          if(this == &other)              //如果对象与 other 是同一个对象
51          {                               //直接返回本身
52              return *this;
53          }
54          delete [] m_String;             //释放堆内存
55          m_String = new char[strlen(other.m_String)+1];
56          strcpy(m_String, other.m_String);
57
58          return *this;
59      }
60
61      int main()
62      {
63          String a("hello");              //调用普通构造函数
64          String b("world");              //调用普通构造函数
65          String c(a);                    //调用复制构造函数
66          c = b;                          //调用赋值函数
67
68          return 0;
69      }
```

函数分析如下。

- 普通构造函数：这里判断了传入的参数是否为NULL。如果是NULL，初始化一个字节的空字符串（包括结束符“\0”）；如果不是，分配足够长度的堆内存保存字符串。
- 复制构造函数：只分配足够长度的堆内存保存字符串。
- 析构函数：如果类私有成员 m_String 不为 NULL，释放 m_String 指向的堆内存，为了避免产生野指针，将 m_String 赋为 NULL。
- 赋值函数：首先判断当前对象与引用传递对象是否是同一个对象，如果是，不做操作

直接返回；否则先释放当前对象的堆内存，然后分配足够长度的堆内存复制字符串。

程序的执行结果如下：

```
Construcing
Construcing
Construcing Copy
Operate = Function
Destructing
Destructing
Destructing
```

代码第 63～66 行会发生构造函数以及赋值函数的调用，而析构函数的调用发生在 main() 函数退出时。

面试例题 27：分析代码写结果——C++类各成员函数关系。

考点：构造函数、析构函数和赋值函数的关系。

出现频率：★★★★

```
1   #include <iostream.h>
2
3   class A
4   {
5   private:
6       int num;
7   public:
8       A()
9       {
10          cout<<"Default constructor"<< endl;
11      }
12      ~A()
13      {
14          cout << "Desconstructor" << endl;
15          cout << num << endl;
16      }
17      A(const A &a)
18      {
19          cout << "Copy constructor" << endl;
20      }
21      void operator = (const A &a)
22      {   cout << "Overload operator" << endl;
23      }
24      void SetNum(int n)
25      {   num = n;
26      }
27  };
28
```

```
29  void main()
30  {
31      A a1;
32      A a2(a1);
33      A a3=a1;
34      A &a4=a1;
35      a1.SetNum(1);
36      a2.SetNum(2);
37      a3.SetNum(3);
38      a4.SetNum(4);
39  }
```

解析

代码第31行，定义了对象a1，调用的是默认的构造函数。

代码第32行，用a1初始化对象a2，调用的是复制构造函数。

代码第33行，同上。注意这里不是调用赋值函数，这里属于对象a3的初始化，而不是赋值。若要调用赋值必须是如下形式：

```
A a3;
a3 = a1;
```

代码第34行，定义a4为a1的一个引用，不调用构造函数或赋值函数。

代码第35～38行，调用各个对象的SetNum()成员函数为私有成员num赋值。注意由于a4为a1的引用，因此a4.SetNum()实际上和a1.SetNum()是相同的。

当main()函数退出时，对象析构顺序与调用构造函数顺序相反，依次为a3、a2和a1。

答案

程序执行结果如下所示：

```
Default constructor
Copy constructor
Copy constructor
Desconstructor
3
Desconstructor
2
Desconstructor
4
```

面试例题28：分析代码写输出——C++类的临时对象。

考点：构造函数、析构函数和赋值函数的编写方法。

出现频率：★★★★

已知class B以及Play()函数定义如下：

```
1   #include <iostream.h>
2
3   class B
```

```
4   {
5   public:
6       B()
7       {
8           cout<<"default constructor"<<endl;
9       }
10
11      ~B()
12      {
13          cout<<"destructed"<<endl;
14      }
15
16      B(int i) : data(i)              //初始化私有成员 data
17      {
18          cout<<"constructed by parameter " << data <<endl;
19      }
20
21  private:
22      int data;
23  };

    B Play( B b)
    {
        return b ;
    }
```

分析下面两个 main()函数的输出。

第 1 个 main()函数

```
1   int main(int argc, char* argv[])
2   {
3       B t1 = Play(5);
4       B t2 = Play(t1);
5
6       return 0;
7   }
```

第 2 个 main()函数

```
1   int main(int argc, char* argv[])
2   {
3       B t1 = Play(5);
4       B t2 = Play(10);
5
6       return 0;
7   }
```

解析

这里调用 Play()函数时，有两种参数类型的传递方式。

- 如果传递的参数是整型数，那么在其函数栈中首先会调用带参数的构造函数产生一个临时对象，然后返回前（在 return 代码执行时）调用类的复制构造函数生成临时对象（这样函数返回后主函数的对象就被初始化了），最后这个临时对象会在函数返回时（在 return 代码执行后）析构。
- 如果传递的参数是 B 类的对象，只有第 1 步与上面的不同，即其函数栈中会首先调用复制构造函数产生一个临时对象，其余步骤完全相同。

可以看出，两种情况的区别是采用了不同的方法生成临时对象（一种是调用带参数的构造函数，另一种是调用复制构造函数）。

在第 1 个 main()函数中，对象 t1 使用了传入整型数的方式调用 Play()函数，而对象 t2 使用了传入 B 的对象的方式调用 Play()函数。在第 2 个 main()函数中，对象 t1 和 t2 都使用了传入整型数的方式调用 Play()函数。第 1 个 main()函数的执行结果为：

```
constructed by parameter 5 （调用带参数的构造函数在 fun 内生成临时对象）
destructed （5 传入 fun 时生成的临时对象析构）
destructed （t1 传入 fun 时产生的返回的临时对象析构）
destructed （t2 析构）
destructed （t1 析构）
```

第 2 个 main()函数的执行结果为：

```
constructed by parameter 5 （调用带参数的构造函数在 fun 内产生临时对象）
destructed （5 传入 fun 时生成的临时对象析构）
constructed by parameter 10 （调用带参数的构造函数在 fun 内产生临时对象）
destructed （10 传入 fun 时生成的临时对象析构）
destructed （t2 析构）
destructed （t1 析构）
```

为了更加详细说明结果，在 B 类中加入一个自定义的复制构造函数：

```
B(B &b)
{
    cout << "copy constructor" << endl;
    data = b.data;
}
```

第 1 个 main()函数下的执行结果为：

```
constructed by parameter 5 （调用带参数的构造函数在 fun 内产生临时对象）
copy constructor （调用复制构造函数把临时对象复制到 t1）
destructed （fun 内的临时对象析构）
copy constructor （调用复制构造函数在 fun 内产生临时对象）
copy constructor （调用复制构造函数把临时对象复制到 t2）
destructed （fun 内的临时对象析构）
destructed （t2 析构）
destructed （t1 析构）
```

第 2 个 main()函数下的执行结果为：

```
constructed by parameter 5 （调用带参数的构造函数 fun 内产生临时对象）
copy constructor （调用复制构造函数把临时对象复制到 t1）
```

```
destructed （fun 内的临时对象析构）
constructed by parameter 10 （调用带参数的构造函数 fun 内产生临时对象）
copy constructor （调用复制构造函数把临时对象复制到 t2）
destructed （fun 内的临时对象析构）
destructed （t2 析构）
destructed （t1 析构）
```

此时，两个 main()函数的输出结果只有第 4 行不一样，这是因为传入不同类型参数（整型与对象类型），从而生成不同的临时对象。

面试例题 29：分析代码写输出——复制构造函数和析构函数。

考点：复制构造函数和析构函数的理解。

出现频率：★★★★

```
1   #include <iostream.h>
2   classA
3   {
4   public:
5       A()
6       {
7           cout << "This is A Construction" << endl;
8       }
9       virtual ~A()
10      {
11          cout << "This is A destruction" << endl;
12      }
13  };
14
15  A fun()
16  {
17      A a;
18      return a;
19  }
20
21  void main()
22  {
23      {
24          A a;
25          a = fun();
26      }
27  }
```

已知程序输出为：

```
This is A Construction
This is A Construction
This is A destruction
This is A destruction
```

```
This is A destruction
```

构造函数和析构函数不是成对出现的吗？为什么少了一个构造函数呢？

解析

构造函数和析构函数确实是成对的。构造函数除了普通构造函数之外，还包括复制构造函数。

上面的程序中一共构造了3个对象，分别是main()函数中的对象a（代码第24行）、fun()函数中的对象a（代码第17行）以及fun返回时生成的临时对象（代码第18行）。前两个对象都是用普通构造函数构造的，而fun返回时生成的临时对象是由复制构造函数生成的。由于上面程序中只是在普通构造函数中打印信息。加入自定义复制构造函数和赋值函数后，程序代码如下所示：

```
1    A(A &a)
2    {
3        cout << "This is A Copy Construction" << endl;
4    }
5    A& operator =(const A &a)
6    {
7        cout << "This is an assignment function" << endl;
8        return *this;
9    }
```

程序执行结果：

```
This is A Construction
This is A Construction
This is A Copy Construction
This is A destruction
This is an assignment function
This is A destruction
This is A destruction
```

可以看出，此时的构造函数和析构函数都被执行了3次，另外在main()函数中把fun()返回的临时对象赋给了对象a，此时会调用赋值函数。

答案

构造函数和析构函数确实是成对的，原程序中的fun返回时生成的临时对象是由复制构造函数生成的。这里没有在复制构造函数中输出信息（编译器生成默认的复制构造函数），所以编写程序时构造函数比析构函数少了一个。

面试例题30：分析代码写结果——C++静态成员和临时对象。

考点：C++静态成员和临时对象的理解。

出现频率：★★★★

```
1    #include <iostream>
2    using namespace std ;
3
4    class human
```

```
5    {
6    public:
7        human()
8        {
9            human_num++;
10       }
11       static int human_num;
12       ~human()
13       {
14           human_num--;
15           print();
16       }
17       void print()
18       {
19           cout<<"human nun is: "<<human_num<<endl;
20       }
21   };
22
23   int human::human_num = 0;
24
25   human f1(human x)
26   {
27       x.print();
28       return x;
29   }
30
31   int main(int argc, char* argv[])
32   {
33       human h1;
34       h1.print();
35       human h2  = f1(h1);
36       h2.print();
37
38       return 0;
39   }
```

解析

程序的 human 类有一个静态成员 human_num，每执行一次普通构造函数 human_num 加 1，每执行一次析构函数 human_num 减 1。注意 f1()函数中使用默认的复制构造函数，然而默认的复制构造函数没有对 human_num 处理。

代码第 34 行，构造了对象 h1（调用普通构造函数），因此打印 1。

代码第 35 行，使用值传递参数的方式调用 f1()函数，这里分为 3 步。

- f1()函数首先调用复制构造函数生成一个临时对象，因此代码第 27 行打印 1。
- f1()函数调用复制构造函数给 main 的对象 h2 初始化（复制临时对象）。
- f1()函数返回后，临时对象发生析构，此时 human 的静态成员 human_num 为 0，打印出 0。

代码第 36 行打印 0。

main()函数结束时有 h1 和 h2 两个对象要发生析构，所以分别打印出−1 和−2。

程序的意图其实很明显，即静态成员用 human_num 记录类 human 的实例数，然而由于默认的复制构造没有对静态成员操作，导致了执行结果的不正确。这里可以通过添加一个自定义的复制构造函数解决。

```
human(human &h)
{
    human_num++;
}
```

此时 human_num 能起到应有的作用了。

答案

程序执行结果：

```
1
1
0
0
-1
-2
```

面试例题 31：什么是临时对象？临时对象在什么情况下产生？

考点：C++临时对象的理解。

出现频率：★★★★

解析

有 C++程序中经常把仅使用一小段时间的变量称为临时变量。例如下面的 swap()函数里：

```
1    void swap(int &a, int &b)
2    {
3        int temp = a;
4        a = b;
5        b = temp;
6    }
```

称 temp 为临时变量。但是在 C++里，temp 根本不是临时变量，实际上它只是函数的局部变量。

真正的临时变量是看不见的，它不会出现在程序代码中。但大多数情况下，它会影响程序执行的效率，所以尽量避免临时对象的产生。它通常在以下所示的两种情况下产生。

- 参数按值传递。
- 返回值按值传递。

参考下面的程序代码：

```
1    #include <iostream>
2    using namespace std ;
```

```
3
4   class Test
5   {
6   public:
7       Test() : num(0) { }                      //默认构造函数
8       Test(int number) : num(number) { }       //带参数的构造函数
9       void print()                             //打印私有成员 num
10      {
11          cout << "num = " << num << endl;
12      }
13      ~Test()                                  //析构函数，打印 this 指针和私有成员 num
14      {
15          cout << "destructor: this = " << this << ", num = " << num << endl;
16      }
17  private:
18      int num;
19  };
20
21  void fun1(Test test)                         //参数按值传递
22  {
23      test.print();
24  }
25
26  Test fun2()
27  {
28      Test t(3);
29      return t;                                //返回值按值传递
30  }
31
32  int main(int argc, char* argv[])
33  {
34      Test t1(1);
35
36      fun1(t1);                                //对象传入
37      fun1(2);                                 //整型数 2 传入
38      t1 = fun2();
39
40      return 0;
41  }
```

程序中的 fun1()函数的参数是按值传递的，fun2()函数的返回值是按值传递的。在 Test 类的析构函数中打印出 this 指针的值，下面是程序执行结果：

```
num = 1
destructor: this = 0012FF0C, num = 1  （fun1 内的临时对象析构）
num = 2
destructor: this = 0012FF0C, num = 2  （fun1 内的临时对象析构）
destructor: this = 0012FEF4, num = 3  （fun2 内的局部变量析构）
destructor: this = 0012FF68, num = 3  （fun2 返回临时对象析构）
destructor: this = 0012FF70, num = 3  （main 内的 t1 析构）
```

这里代码第 36 和代码第 37 行使用了两种类型的参数传入 fun1()函数，它们都会生成临时变量，但是代码第 36 行的调用使用了复制构造函数创建临时变量，而代码第 37 行调用使用的则是带参数的构造函数创建临时变量。

如何避免产生临时变量呢？可以利用引用传递代替值传递。例如把上面的 fun1()函数改成如下形式：

```
void fun1(Test &test)
{
    test.print();
}
```

这样 fun1()函数的参数就是一个已经存在的对象引用，此时整型值是不能传进来的。执行下面的主程序：

```
1   int main(int argc, char* argv[])
2   {
3       Test t1(1);
4       fun1(t1);   //对象引用传入
5       return 0;
6   }
```

程序的执行结果如下：

```
num = 1
destructor: this = 0012FF70, num = 1  （main 内的 t1 析构）
```

可以看到，此时就不会产生临时对象了。

注意：必须引用实在的、可引用的对象，否则引用是错误的。因此在没有实在的、可引用的对象时，只能依赖临时对象。

9.6 函数重载和运算符重载

9.6.1 函数重载

C++引入了函数重载的功能。函数重载是指一系列具有相同或者相似功能，但数据类型或者参数不同的同名函数。应聘时不仅会考查函数重载的操作，还会考查函数重载与覆写（override）之间的区别。

面试例题 32：什么是函数重载？为什么 C 不支持函数重载，而 C++能支持函数重载？

考点：函数重载以及 C++和 C 之间的差异性的理解。

出现频率：★★★★

解析

要进行两种不同数据类型的和操作，在 C 语言里需要写两个不同名称的函数来进行区分：

```
1    int add1(int a, int b)
2    {
3        return (a +b);
4    }
5
6    float add2(float a, float b)
7    {
8        return (a +b);
9    }
```

上面的代码编写的不太完善，这两个具备相似操作的函数，却给它们取了两个不同的名字，这样不便于管理。因此C++为了编写方便，引入了函数重载的概念。例如下面的代码：

```
1    #include <iostream>
2    using namespace std;
3
4    class Test
5    {
6    public:
7        int add(int x, int y)          //相加,传入参数以及返回值都是 int
8        {
9            return (x+y);
10       }
11       float add(float x, float y)    //相加,传入参数以及返回值都是 float
12       {
13           return (x+y);
14       }
15   };
16
17   int add(int x, int y)              //相加,传入参数以及返回值都是 int
18   {
19       return (x+y);
20   }
21
22   float add(float x, float y)        //相加,传入参数以及返回值都是 float
23   {
24       return (x+y);
25   }
26
27   int main(int argc, char* argv[])
28   {
29       int i = add(1, 2);
30       float f = add(1.1f, 2.2f);
31       Test test;
32       int i1 = test.add(3, 4);
33       float f1 = test.add(3.3f, 4.4f);
34
```

```
35        cout << "i = " << i << endl;
36        cout << "f = " << f << endl;
37        cout << "i1 = " << i1 << endl;
38        cout << "f1 = " << f1 << endl;
39
40        return 0;
41    }
```

上面的程序中使用了全局函数和类成员函数的重载，代码第29～第38行是对它们的调用与测试。可以看到，在C++中可以根据传入参数类型和返回类型来区分不同的重载函数。

C不支持函数重载，而C++却支持，这是因为C++的重载函数经过编译器处理之后，两个函数的符号是不相同的。例如代码第17行的add函数，经过处理后变成_int_add_int_int，而代码第22行经过处理后变成了_float_add_float_float。这样的名字包含了函数名、函数参数数量及返回类型信息，C++就是靠这种机制来实现函数重载的。

答案

函数重载是用来描述同名函数具有相同或者相似功能，但数据类型或者是参数不同的函数管理操作。

函数名经过C++编译器处理后包含了原函数名、函数参数数量及返回类型信息，而C不会对函数名进行处理。

面试例题33：判断题——函数重载的正确声明。

考点：函数重载的正确声明。

出现频率：★★★★

```
1     int calc(int,int);                          //A
2     int calc(const int,const int);
3
4     int get();                                  //B
5     double get();
6
7     int *reset(int *);                          //C
8     double *reset(double *);
9
10    extern "C" int compute(int *,int);          //D
11    extern "C" double compute(double *,double);
```

解析

A错误。第2个函数被视为重复声明，第2个声明中的const修饰词会被忽略。

B错误。第2个声明是错误的，因为单就函数的返回值而言，不足以区分两个函数的重载。

C正确。这是合法的声明，reset()函数被重载。

D错误。第2个函数声明是错误的，因为在一组重载函数中，只能有一个函数被指定为extern "C"。

答案

C

面试例题 34：重载和覆写有什么区别？

考点：重载和覆写之间区别的理解。

出现频率：★★★★

解析

重载（overriding）是指子类改写父类的方法；覆写（overloading）是指同一个函数的不同版本之间参数不同。

重载是编写一个与已有函数同名但是参数表不同（参数数量或参数类型不同）的方法，它具有如下所示的特征。

- 方法名必须相同。
- 参数列表必须不相同，与参数列表的顺序无关。
- 返回值类型可以不相同。

覆写是派生类重写基类的虚函数，它具有如下所示的特征。

- 只有虚方法和抽象方法才能够被覆写。
- 具有相同的函数名。
- 具有相同的参数列表。
- 具有相同的返回值类型。

重载是一种语法规则，由编译器在编译阶段完成，不属于面向对象的编程；而覆写是由运行阶段决定的，是面向对象编程的重要特征。

9.6.2 运算符重载

C++允许类的对象构造运算符实现单目或者双目运算，这个特性就叫运算符重载。对于每一个运算符@，在 C++中都对应一个运算符函数 operator@（@为 C++各种运算符）。运算符函数的一般原型为：

```
type operator@ (arglist)
```

其中 type 为运算结果的类型，arglist 为操作数列表。

对于不同的情况需要进行不同运算符的重载：

- 在定义的对象间相互赋值时，重载赋值运算符。
- 在数字类型增加算术属性时，重载算术运算符。
- 为定义的对象进行逻辑比较时，重载关系运算符。
- 对于容器（container），重载下标运算符[]。
- 从 I/O 流中读写对象时，重载“<<”和“>>”运算符。
- 实现 smart 指针时，重载成员指针运算符“->”。

- 在少数情况下重载 new、delete 运算符。

运算符重载需要遵循以下所示的规则：

- 重载的运算符不能违反语言的语法规则。
- 如果一个运算符可以放在两个操作数之间，就可以重载它来满足类操作的需要，哪怕这种用法原本为编译器所不能接受。
- 不能创造 C++语言中没有的运算符。
- 重载时不能改变运算符的优先级。

面试例题 35：编程题——MyString 类的编写。

考点：重载=和+运算符。

出现频率：★★★

对于下面的类 MyString，要求重载运算符后可以计算表达式 a=b+c（a、b、c 都是类 MyString 的对象）。请重载相应的运算符并编写程序测试。

```
1    class MyString
2    {
3    public:
4        MyString(char *s)
5        {
6            str = new char[strlen(s)+1];
7            strcpy(str,s);
8        }
9        ~MyString()
10       {
11           delete []str;
12       }
13   private:
14       char *str;
15   };
```

解析

为了实现 a=b+c 这个表达式，需要重载两个运算符，一个是“+”运算符，用于 b+c，另一个是“=”运算符，用于对象 a 的赋值。程序代码如下：

```
1    #include <iostream>
2    using namespace std;
3
4    class MyString
5    {
6    public:
7        MyString(char *s)                    //参数为字符指针的构造函数
8        {
9            str = new char[strlen(s)+1];
10           strcpy(str,s);
```

```
11          }
12          ~MyString()                                    //析构函数释放 str 堆内存
13          {
14              delete []str;
15          }
16          MyString & operator = (MyString &string)   //赋值函数，重载 =
17          {
18              if (this == &string)
19              {
20                  return *this;
21              }
22              if (str != NULL)                           //释放内存
23              {
24                  delete []str;
25              }
26              str = new char[strlen(string.str) + 1];    //申请内存
27              strcpy(str, string.str);                   //复制字符串内容
28              return *this;
29          }
30
31          MyString & operator + (MyString &string)    //重载 +(改变被加对象)
32          {
33              char *temp = str;
34              str = new char[strlen(temp) + strlen(string.str) + 1];
35              strcpy(str, temp);                         //复制第 1 个字符串
36              delete []temp;
37              strcat(str, string.str);                   //连接第 2 个字符串
38              return *this;
39          }
40
41
42          /*MyString & operator + (MyString &string)  //重载 +(不改变被加对象)
43          {
44              MyString *pString = new MyString("");  //堆内存中构造对象
45              pString->str = new char[strlen(str) + strlen(string.str) + 1];
46              strcpy(pString->str, str);                 //复制第 1 个字符串
47              strcat(pString->str, string.str);          //连接第 2 个字符串
48              return *pString;                           //返回堆中的对象
49          }*/
50
51          void print()                                   //测试打印 str 成员
52          {
53              cout << str << endl;
54          }
55      private:
56          char *str;
```

```
57  };
58
59  /* //MyString 类的友员，要求 str 成员是 public 访问权限
60  MyString & operator +(MyString &left, MyString &right)   //重载 +(不改变被加对象)
61  {
62      MyString *pString = new MyString("");
63      pString->str = new char[strlen(left.str) + strlen(right.str) + 1];
64      strcpy(pString->str, left.str);
65      strcat(pString->str, right.str);
66
67      return *pString;
68  } */
69
70  int main(int argc, char* argv[])
71  {
72      MyString a("hello ");
73      MyString b("world");
74
75      MyString c("");
76      c = c + a;              //先做加法，再赋值
77      c.print();
78      c = c + b;              //先做加法，再赋值
79      c.print();
80
81      c = a + b;
82      a.print();
83      c.print();
84
85      return 0;
86  }
```

这里有 3 个版本的“+”操作符重载函数，它们都是调用 strcpy 复制第 1 个字符串，然后调用 strcat 连接第 2 个字符串。

第 1 个版本返回*this 对象，它改变了被加对象的内容。使用第 1 个“+”操作符重载函数版本的执行结果如下：

```
hello
hello world
hello world（对象 a 的 str 成员被改变了）
hello world
```

第 2 个版本和第 3 个版本都是返回堆中构造的对象，它们没有改变被加对象内容。它们的区别如下。

- 第 2 个版本属于类的成员函数，而第 3 个版本是类的友员函数。
- 第 2 个版本的参数为 1 个，而第 3 个版本的参数为 2 个，因为友员函数不含有 this 指针。
- 由于类的友员函数不能使用私有成员，因此在这里使用第 3 个版本时需要把 str 成员的

访问权限改为 public。

使用这两个“+”操作符重载函数版本的执行结果如下：

```
hello
hello world
hello（对象 a 的 str 成员没有被改变）
hello world
```

选择何种版本的“+”操作符重载函数要取决于实际情况。

面试例题 36：编程题——各类运算符重载函数的编写。

考点：编写各类运算符重载函数。

出现频率：★★★★

用 C++实现一个 String 类，它具有比较、连接、输入、输出功能，并且提供一些测试用例说明如何使用这个类。注意不要用 MFC、STL 以及其他库。

解析

要实现本题要求功能，需要重载下面的运算符。

- ❑ <、>、==和!=比较运算符。
- ❑ += 连接运算符以及赋值运算符
- ❑ 输出运算符和输入运算符。

根据分析，可得到如下 String 类（String.h 文件）的定义。

```
1   #ifndef STRING_H
2   #define STRING_H
3
4   #include <iostream>
5   using namespace std;
6
7   class String
8   {
9   public:
10      String();                                  //默认构造函数
11      String(int n,char c);                      //普通构造函数
12      String(const char* source);                //普通构造函数
13      String(const String& s);                   //复制构造函数
14      String& operator = (char* s);              //重载=,实现字符串赋值
15      String& operator = (const String& s);      //重载=,实现对象赋值
16      ~String();                                 //析构函数
17      char& operator[](int i);                   //重载[],实现数组运算
18      const char& operator[](int i) const;       //重载[],实现数组运算(对象为常量)
19      String& operator += (const String& s);     //重载+=,实现与字符串相加
20      String& operator += (const char* s);       //重载+=,实现与对象相加
21      friend ostream& operator << (ostream&out, String& s);        //重载 <<,实现输出流
22      friend istream& operator >> (istream& in, String& s);        //重载 >>,实现输入流
```

```
23        friend bool operator < (const String& left,const String& right);    //重载 <
24        friend bool operator > (const String& left, const String& right);   //重载 >
25        friend bool operator == (const String& left, const String& right) ; //重载 ==
26        friend bool operator != (const String& left, const String& right);  //重载 !=
27        char* getData();                                  //获取 data 指针
28    private:
29        int size;                                         //data 表示的字符串长度
30        char* data;                                       //指向字符串数据
31    };
32    #endif
```

为了实现与对象操作和与字符串操作，=和+=运算符的重载函数都有两个，它们的参数分别为 String 对象引用和字符指针。String.h 文件的代码第 21～第 26 行声明运算符重载函数都是友员，并且这些函数所重载的运算符都是双目运算符，因此参数是两个。也可以把这些友员函数改为成员函数，此时参数是一个。需要注意：输入输出流操作符的重载最好是声明为友员函数。

String 类声明的所有函数实现在 String.cpp 文件中，下面是 String.cpp 的清单：

```
1    #include "String.h"
2
3    String::String()                        //默认构造函数，构造空字符串
4    {
5        data = new char[1];             //空字符串只含有'\0'一个元素
6        *data = '\0';
7        size = 0;
8    }
9    String::String(int n,char c)            //普通构造函数
10   {                                        //含有 n 个相同字符的字符串
11       data = new char[n + 1];
12       size = n;
13       char *temp = data;              //保存 data
14       while(n--) //做 n 次赋值
15       {
16           *temp++ = c;
17       }
18       *temp = '\0';
19   }
20   String::String(const char *source)      //普通构造函数
21   {                                        //字符串内容与 source 相同
22       if (source == NULL)             //source 为 NULL
23       {                                //将 data 赋为空字符串
24           data = new char[1];
25           *data = '\0';
26           size = 0;
27       }
28       else
29       {                                //source 不为 NULL
```

```
30              size = strlen(source);      //复制 source 字符串
31              data = new char[size + 1];
32              strcpy(data, source);
33          }
34  }
35  String::String(const String &s)        //复制构造函数
36  {                                       //字符串内容与对象 s 的相同
37          data = new char[s.size + 1];
38          strcpy(data, s.data);
39          size = s.size;
40  }
41  String& String::operator = (char *s)    //= 重载
42  {                                       //目标为字符串
43          if (data != NULL)
44          {
45              delete []data;
46          }
47          size = strlen(s);
48          data = new char[size + 1];
49          strcpy(data, s); //复制目标字符串
50          return *this;
51  }
52  String& String::operator = (const String& s)    //= 重载
53  {                                               //目标为 String 对象
54          if (this == &s)                 //如果对象 s 就是自己，直接返回*this
55          {
56              return *this;
57          }
58          if (data != NULL)               //释放 data 堆内存
59          {
60              delete []data;
61          }
62          size = strlen(s.data);
63          data = new char[size + 1];      //分配堆内存
64          strcpy(data, s.data);           //复制对象 s 的字符串成员
65          return *this;
66  }
67  String::~String()
68  {
69          if (data != NULL)               // data 不为 NULL，释放堆内存
70          {
71              delete []data;
72              data = NULL;
73              size = 0;
74          }
```

```
75  }
76  char& String::operator [](int i)          //[]重载
77  {                                          //取数组下标为 i 的字符元素
78      return data[i];
79  }
80  const char& String::operator[] (int i) const
81  {
82      return data[i];
83  }
84  String& String::operator +=(const String& s)    // += 重载
85  {                                          // 连接对象 s 的字符串成员
86      int len = size + s.size + 1;
87      char *temp = data;
88      data = new char[len];       //申请足够的堆内存来存放连接后的字符串
89      size = len - 1;
90      strcpy(data, temp);         //复制原来的字符串
91      strcat(data, s.data);       //连接目标对象内的字符串成员
92      delete []temp;
93      return *this;
94  }
95  String& String::operator +=(const char *s)    //+= 重载
96  {                                              //连接 s 字符串
97      if (s == NULL)
98      {
99          return *this;
100     }
101     int len = size + strlen(s) + 1;
102     char *temp = data;
103     data = new char[len];             //申请足够的堆内存来存放连接后的字符串
104     size = len - 1;
105     strcpy(data, temp);              //复制原来的字符串
106     strcat(data, s);                 //连接目标字符串
107     delete []temp;
108     return *this;
109 }
110 String::length()                     //获取字符串长度
111 {
112     return size;
113 }
114 ostream& operator << (ostream &out, String &s)    //重载<<
115 {                                     //打印对象 s 内字符串成员的所有字符元素
116     for(int i = 0; i < s.length(); i++)
117     {
118         out << s[i] << " ";          //输出字符串中每一个字符元素
119     }
```

```
120         return out;
121     }
122     istream& operator >> (istream& in, String& s)
123     {
124         char p[50];
125         in.getline(p, 50);                  //从输入流接收最多 50 个字符
126       s = p;                              //调用赋值函数
127         return in;
128     }
129     bool operator < (const String& left, const String& right)   //重载 <
130     {
131         int i = 0;
132         while(left[i] == right[i] && left[i] != 0 && right[i] != 0)
133         {
134             i++;
135         }
136         return left[i]-right[i] < 0 ? true : false;
137     }
```

重载=、>、!=和重载<非常相似，这里就不列举了。

由于以上的代码清单中有详细的注释，因此就不再赘述了。另外还有两点需要说明。

- 友员函数不能访问 String 类的私有成员，但由于重载了[]操作符，所以可以使用对象索引（left[i]和 right[i]的方式）进行访问。
- 不能使用字符串复制（私有成员 data 不能访问），而是调用赋值函数给对象 s 赋字符串的内容（代码 126 行）。

程序代码如下：

```
1     #include <iostream>
2     #include "String.h"
3     using namespace std;
4
5     int main(void)
6     {
7         String str(3, 'a');                         //普通构造函数测试
8         String str1(str);                           //复制构造函数测试
9         String str2("asdf");                        //普通构造函数
10        String str3;                                //默认构造函数测试
11
12        cout << "str: " << str << endl;
13        cout << "str1: " << str1 << endl;
14        cout << "str2: " << str2 << endl;
15        cout << "str3: " << str3 << endl;
16
17        str3 = str2;                                //赋值函数测试
18        cout << "str3: " << str3 << endl;
```

```
19        str3 = "12ab";                              //赋值函数测试
20        cout << "str3: " << str3 << endl;
21
22        cout << "str3[2] = " << str3[2] << endl;    //[] 重载函数测试
23
24        str3 += "111";                              //+= 重载函数测试
25        cout << "str3: " << str3 << endl;
26        str3 += str1;                               //+= 重载函数测试
27        cout << "str3: " << str3 << endl;
28
29        cin >> str1;                                // >>重载函数测试
30        cout << "str1: " << str1 << endl;
31
32        String t1 = "1234";
33        String t2 = "1234";
34        String t3 = "12345";
35        String t4 = "12335";
36
37        cout << "t1 == t2 ? " << (t1 == t2) << endl;     // == 重载函数测试
38        cout << "t1 < t3 ? " << (t1 < t3) << endl;       // < 重载函数测试
39        cout << "t1 > t4 ? " << (t1 > t4) << endl;       // > 重载函数测试
40        cout << "t1 != t4 ? " << (t1 != t4) << endl;     // != 重载函数测试
41
42        return 0;
43    }
```

程序执行结果：

```
str: a a a
str1: a a a
str2: a s d f
str3:
str3: a s d f
str3: 1 2 a b
str3[2] = a
str3: 1 2 a b 1 1 1
str3: 1 2 a b 1 1 1 a a a
123 456 abc def（终端输入）
str1: 1 2 3    4 5 6    a b c    d e f
t1 == t2 ? 1
t1 < t3 ? 1
t1 > t4 ? 1
t1 != t4 ? 1
```

面试例题 37：分析代码写输出——new 操作符重载的使用。

考点： new 操作符重载的使用。

出现频率：★★

下面程序中主函数的 new 操作符是类中 new 操作符的重载，但是 new 后面只有一个参数 0xa5，而类中函数的声明有两个参数，那么调用这个类具体过程该如何理解？

```
1   #include <malloc.h>
2   #include <memory.h>
3
4   class Blanks
5   {
6   public:
7       void *operator new( size_t stAllocateBlock, char chInit );
8   };
9
10  void *Blanks::operator new( size_t stAllocateBlock, char chInit )
11  {
12      void *pvTemp = malloc( stAllocateBlock );
13      if( pvTemp != 0 )
14          memset( pvTemp, chInit, stAllocateBlock );
15      return pvTemp;
16  }
17
18  int main()
19  {
20      Blanks *a5 = new( 0xa5 ) Blanks;
21
22      return a5 != 0;
23  }
```

解析

这里有以下几点需要说明。

- 重载 new 操作符第 1 个参数必须是 size_t 类型的，并且传入的值就是类的大小。本例题中类的大小为 1。如果类中含有一个 int 类型成员（int 占 4 个字节），那么参数 stAllocateBlock 的值为 4。
- 代码第 20 行，用 chInit 初始化分配的内存。
- 代码第 14 行中的 0xa5 表示第 2 个参数的大小，也就是 chInit 为 0xa5。
- 当执行代码第 20 行时，首先调用 Blanks 重载的 new 操作符函数，然后使用默认的构造函数初始化对象，最后用这个 Blanks 对象地址初始化 a5。

第10章 C++继承和多态

继承和多态是C++面向对象程序设计的关键。继承机制使得派生类能够获得基类的成员数据和方法。多态是建立在继承基础上的，它使用了C++编译器最核心的一个技术，即动态绑定技术。

10.1 继承的概念

继承是面向对象程序设计中最重要的机制，它支持层次分类的观点。继承使得程序员可以在一般类的基础上很快地建立一个新类，而不必从零开始设计每个类。

下面举例简单地说明继承的概念，如图10.1所示。

上图是一个抽象描述的特性继承表。

生物是所有类的基类，所有生物都有寿命，所以可以把年龄作为生物类的属性。继续给生物分类，分为动物类和植物类。当建立动物类和植物类的时候无需再定义基类已经有的数据成员，而只需要描述动物类和植物类所特有的特性即可。比如动物类有奔跑、睡觉等行为。

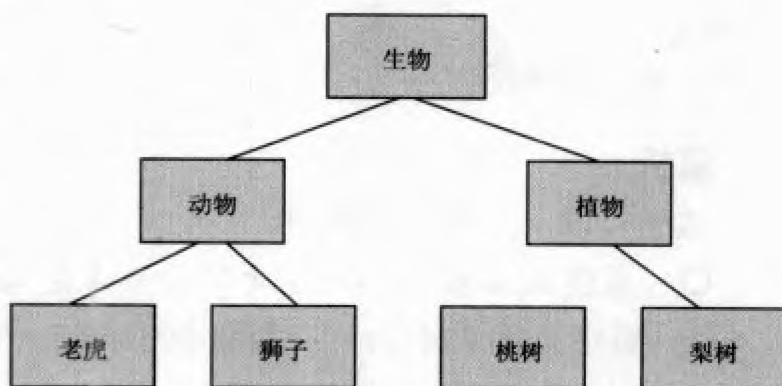


图10.1　继承的概念

动物类和植物类的特性是在生物类原有特性基础上增加而来的，那么动物类和植物类就是生物类的派生类（也称做子类）。同样老虎类和狮子类也是动物类的派生类，它们拥有动物类的一切特性。这种子类获得父类特性的概念就是继承。

面试例题1：C++类继承的3种关系。

考点：对C++类继承的3种关系的理解。

出现频率：★★★★★

解析

C++中继承主要有3种关系public、protected和private。

（1）public 继承

public 继承是一种接口继承，子类可以代替父类完成父类接口所声明的行为。此时子类可以自动转换成为父类的接口，完成接口转换。从语法角度上来说，public 继承会保留父类中成员（包括函数和变量等）的可见性，也就是说，如果父类中的某个函数是 public，那么被子类继承后仍然是 public。

（2）protected 继承

protected 继承是一种实现继承，子类不能代替父类完成父类接口所声明的行为，此时子类不能自动转换成父类的接口。从语法角度上来说，protected 继承会将父类中的 public 可见性的成员修改成为 protected 可见性，这样在子类中同样可以调用父类的 protected 和 public 成员，子类的子类也可以调用被 protected 继承的父类的 protected 和 public 成员。

（3）private 继承

private 继承是一种实现继承，子类不能代替父类完成父类接口所声明的行为，此时子类不能自动转换成为父类的接口。从语法角度上来说，private 继承会将父类中的 public 和 protected 可见性的成员修改成为 private 可见性，这样虽然子类中同样还是可以调用父类的 protected 和 public 成员，但是在子类的子类就不可以再调用被 private 继承的父类的成员了。

下面的程序代码说明了 protected 继承和 private 继承的区别。

```
1   #include <iostream>
2   using namespace std;
3
4   class Base
5   {
6   protected:
7           void printProtected() {cout << "print Protected" << endl;}
8   public:
9   void printPublic() {cout << "print Public" << endl;}
10  };
11
12  class Derived1 : protected Base     //protected 继承
13  {
14  };
15
16  class Derived2 : private Base       //private 继承
17  {
18  };
19
20  class A : public Derived1
21  {
22  public:
23          void print()
24          {
25                  printProtected();
```

```
26                  printPublic();
27            }
28      };
29
30      class B : public Derived2
31      {
32      public:
33            void print()
34            {
35                  printProtected();                 //编译错误，不能访问
36                  printPublic();                    //编译错误，不能访问
37            }
38      };
39
40      int main()
41      {
42            class A a;
43            class B b;
44            a.print();
45            b.print();
46            return 0;
47      }
```

Derived1 类通过 protected 继承 Base 类，因此它的派生类 A 可以访问 Base 基类的 protected 和 public 成员函数。

Derived2 类是通过 private 继承 Base 类，因此它的派生类 B 不可以访问 Base 基类的任何成员函数。

面试例题 2：分析代码——C++继承关系。

考点：对 C++类继承的 3 种关系的理解。

出现频率：★★★★

请考虑下面的标记为 A～J 的语句在编译时可能出现的情况。如果能够成功编译，请记为 “RIGHT”，否则记为 “ERROR”。

```
1     #include <iostream>
2     using namespace std;
3
4     class Parent
5     {
6     public:
7     Parent(int var = -1)
8     {
9           m_nPub = var;
10                m_nPtd = var;
11                m_nPrt = var;
```

```
12          }
13      public:
14          int m_nPub;
15      protected:
16          int m_nPtd;
17      private:
18          int m_nPrt;
19      };
20
21      class Child1 : public Parent
22      {
23      public:
24          int getPub() {return m_nPub;}
25          int getPtd() {return m_nPtd;}
26          int getPrt() {return m_nPrt;}       //A
27      };
28
29      class Child2 : protected Parent
30      {
31      public:
32          int getPub() {return m_nPub;}
33          int getPtd() {return m_nPtd;}
34          int getPrt() {return m_nPrt;}      //B
35      };
36
37      class Child3 : private Parent
38      {
39      public:
40          int getPub() {return m_nPub;}
41          int getPtd() {return m_nPtd;}
42          int getPrt() {return m_nPrt;}      //C
43      };
44
45
46      int main()
47      {
48          Child1 cd1;
49          Child2 cd2;
50          Child3 cd3;
51
52          int nVar = 0;
53
54          //public inherited
55          cd1.m_nPub = nVar;              //D
56          cd1.m_nPtd = nVar;              //E
57          nVar = cd1.getPtd();            //F
58          //protected inherited
```

```
59          cd2.m_nPub = nVar;              //G
60          nVar = cd2.getPtd();            //H
61          //private inherited
62          cd3.m_nPub = nVar;              //I
63          nVar = cd3.getPtd();            //J
64
65          return 0;
66      }
```

解析

A、B、C 错误。m_nPrt 是基类 Parent 的私有变量，不能被派生类访问。

D 正确。Child1 是 public 继承，可以访问并修改基类 Parent 的 public 成员变量。

E 错误。m_nPtd 是基类 Parent 的 protected 成员变量，通过公有继承变成了派生类 Child1 的 protected 成员，因此只能在 Child1 类内部访问，不能使用 Child1 对象访问。

F 正确。可以通过 Child1 类的成员函数访问其 protected 变量。

G 错误。Child2 是 protected 继承，其基类 Parent 的 public 和 protected 成员变成了它的 protected 成员，因此 m_nPub 只能在 Child2 类内部访问，不能使用 Child2 对象访问。

H 正确。可以通过 Child2 类的成员函数访问其 protected 变量。

I 错误。Child3 是 private 继承，其基类 Parent 的 public 和 protected 成员变成了它的 private 成员，因此 m_nPub 只能在 Child2 类内部访问，不能使用 Child2 对象访问。

J 正确。可以通过 Child3 类的成员函数访问其 private 变量。

答案

A、B、C、E、G、I 为“ERROR”。

F、H、J 为“RIGHT”。

面试例题 3：分析代码找错——C++继承。

考点：对 C++类继承的 3 种关系的理解。

出现频率：★★★★

```
1       #include <iostream>
2       using namespace std;
3
4       class base
5       {
6       private:
7           int i;
8       public:
9       base(int x) { i=x; }
10      };
11
12      class derived: public base
13      {
```

```
14      private:
15              int i;
16      public:
17              derived(int x, int y) { i=x;}
18              void printTotal()
19              {
20                      int total = i + base::i;
21                      cout << "total = " << total << endl;
22              }
23      };
24
25      int main()
26      {
27              derived d(1, 2);
28              d.printTotal();
29              return 0;
30      }
```

答案

这个程序有如下所示两个错误。

（1）在 derived 类进行构造时，它首先要调用其基类（base 类）的构造方法，由于没有指明构造方法，默认调用 base 类不带参数的构造方法，然而基类 base 中已经定义了带一个参数的构造函数，所以编译器就不会给它定义默认的构造函数。因此代码第 17 行会出现“找不到构造方法”的编译错误。解决办法是在 derived 的构造函数中显示调用 base 的构造函数，如下所示：

```
d erived(int x, int y) : base(y) { i=x;}    //代码第 17 行
```

（2）derived 类的 printTotal()中，使用 base::i 的方式调用 base 类的私有成员 i，这样会得到“不能访问私有成员”的编译错误。解决办法是把成员 i 的访问权限设为 public。

10.2 私有继承

私有继承之后，许多在父类外界（或派生类）可以访问的特性，到了派生类全部变成不能访问的了。

面试例题 4：私有继承有什么作用？

考点：对 C++类私有继承的理解。

出现频率：★★★★★

解析

分析下面的代码：

```
1       #include <iostream>
2       using namespace std;
```

```
3
4     class Person
5     {
6     public:
7         void eat() { cout << "Person eat" << endl;}
8     };
9
10    class Student : private Person       //私有继承
11    {
12    public:
13        void study() {cout << "Student Study" << endl;}
14    };
15
16    int main()
17    {
18        Person p;
19        Student s;
20
21        p.eat();
22        s.study();
23        s.eat();                    //编译错误
24        p = s;                      //编译错误
25
26        return 0;
27    }
```

此程序的两个编译错误分别说明了私有继承的两个规则。

第 1 个规则：和公有继承相反，如果两个类之间的继承关系为私有，编译器一般不会将派生类对象（如 Student）转换成基类对象（如 Person），这就是代码 24 行失败的原因。

第 2 个规则：从私有基类继承而来的成员成为派生类的私有成员，即使它们在基类中是保护或公有成员，这就是代码第 23 行失败的原因。

可以看出私有继承时派生类与基类不是“is a”的关系，而是“IsImplementInTermsOf”（即以...实现）的关系。使类 D 私有继承于类 B，这样做是因为你想利用类 B 中已经存在的某些代码，而不是因为类型 B 的对象和类型 D 的对象之间有什么概念上的关系。

面试例题 5：私有继承和组合的相同点和不同点是什么？如何在两者之间作出选择？

考点：私有继承和组合的理解。

出现频率：★★★

解析

使用组合表示“（hasa）”（即有一个）的关系。如果在组合中需要使用对象的某些方法，则完全可以利用私有继承代替。

私有继承使派生类获得基类的一份备份，同时可以访问基类的公共以及保护接口以及重写

基类虚拟函数。它意味着“IsImplementInTermsOf”（即以…实现），它是组合语法上的一种变形（即聚合或者“有一个”）。

分析下面的代码：

```
1   #include <iostream>
2   using namespace std;
3
4   class Engine
5   {
6   public:
7   Engine(int num) : numCylinders(num) {}     //Engine 构造函数
8   void start()
9   {
10          cout << "Engine start, " << numCylinders << " Cylinders" << endl;
11      }
12  private:
13      int numCylinders;
14  };
15
16  class Car_pri : private Engine                  //私有继承
17  {
18  public:
19      Car_pri() : Engine(8) {}                    //调用基类的构造函数
20      void start()
21      {
22          Engine::start();                        //调用基类的 start()
23      }
24  };
25
26  class Car_comp
27  {
28  private:
29      Engine engine;                              //组合 Engine 类对象
30  public:
31      Car_comp() : engine(8) {}                   //给 engine 成员初始化
32      void start()
33      {
34          engine.start();                         //调用 engine 的 start()
35      }
36  };
37
38  int main()
39  {
40      Car_pri car_pri;
41      Car_comp car_comp;
42
43      car_pri.start();
```

```
44          car_comp.start();
45
46          return 0;
47      }
```

由此看出，“has a”关系既可以用私有继承表示，也可以用单一组合表示。

类 Car_pri 和类 Car_comp 有很多相似点，如下所示。

- 它们都只有一个 Engine 被确切地包含在 Car 中。
- 它们在外部都不能进行指针转换，如不可以将 Car_pri *转换为 Engine * 。
- 它们都有一个 start()方法，并且都在包含的 Engine 对象中调用 start()方法。

  也有以下的区别。
- 如果想让每个 Car 都包含若干 Engine，那么只能用单一组合的形式。
- 私有继承形式可能引入不必要的多重继承。
- 私有继承形式允许 Car 的成员将 Car *转换成 Engine *。
- 私有继承形式允许访问基类的保护（protected）成员。
- 私有继承形式允许 Car 重写 Engine 的虚函数。

在组合和私有继承之间选择的原则是尽可能使用组合，万不得已时才用私有继承。请看下面的程序代码：

```
1       #include <iostream>
2       using namespace std;
3
4       struct Base                                             //抽象
5       {
6       public:
7       virtual void Func1() = 0;                               //纯虚函数
8       virtual void Func2() = 0;                               //纯虚函数
9       void print()
10          {
11              Func1(); //调用派生类的 Fun1()
12              Func2(); //调用派生类的 Fun1()
13          }
14      };
15
16      struct T : private Base
17      {
18      public:
19          virtual void Func1() {cout << "Func1" << endl;}     //覆盖基类的 Fun1
20          virtual void Func2() {cout << "Func2" << endl;}     //覆盖基类的 Fun2
21          void UseFunc()
22          {
23              Base::print();                                  //调用基类的 print()
24          }
25      };
```

```
26
27  int main()
28  {
29          T t;
30          t.UseFunc();
31          return 0;
32  }
```

程序输出如下：

```
Func1
Func2
```

上面的代码中 Base 类含有纯虚函数 Fun1()和 Fun2()，它通过虚拟函数调用 T 中的重写版本。这种情况不能使用组合，这是因为组合的对象关系中不能使用一个抽象类，即抽象类不能实例化。

答案

相同点是都可以表示“has a”关系。

不同点是私有继承中派生类能访问基类的 protected 成员，并且可以重写基类的虚拟函数(甚至当基类是抽象类时)。组合不具有这些功能。

选择它们的原则为尽可能使用组合，万不得已时使用私有继承。

10.3 多态的概念

当不同的对象调用相同的名称的成员函数时，可能引起不同的行为（即执行不同的代码）。这种现象称为多态性。将函数调用链接相应函数体的代码的过程称为函数联编（简称联编）。在 C++中，根据联编时刻不同，分为静态联编和动态联编。

静态联编，不同的类可以有相同名称的成员函数(甚至还可以有相同的参数，编译器在编译时就对它们进行函数联编，这种在编译时刻进行的联编称为静态联编。静态联编所支持的多态性称为编译时多态性。函数重载就属于编译时多态性。

动态联编，在动态联编中，在程序运行时才能确定调用哪个函数。这种在运行时的函数联编称为动态联编。动态联编所支持的多态性称为运行时多态性。在 C++中，只有虚函数才可能是动态联编的。可以通过定义类的虚函数和创建派生类，然后在派生类中重新实现虚函数，实现具有运行时的多态性。

面试例题 6：什么是多态？

考点：对 C++多态的理解。

出现频率：★★★★★

解析

多态（Polymorphism）、封装（Encapsulation）和继承（Inheritance）是面向对象思想的“三

大特征”。而多态性的定义是：同一操作作用于不同的对象，产生不同的执行结果。有两种类型的多态性：

（1）编译时的多态性

编译时的多态性是通过重载来实现的。对于非虚成员来说，系统在编译时，根据传递的参数、返回的类型等信息决定实现何种操作。

（2）运行时的多态性

运行时的多态性是指直到系统运行时，才根据实际情况决定实现何种操作。C++中，运行时的多态性通过虚成员实现。程序代码如下所示。

```
1    #include <iostream>
2    using namespace std;
3
4    class Person
5    {
6    public:
7    virtual void print() {cout << "I'm a Person" << endl;}
8    };
9    class Chinese : public Person
10   {
11   public:
12           virtual void print() {cout << "I'm from China" << endl;}
13   };
14
15   class American : public Person
16   {
17   public:
18           virtual void print() {cout << "I'm from USA" << endl;}
19   };
20
21   void printPerson(Person &person)
22   {
23           person.print();                    //运行时决定调用哪个类中的 print()函数
24   }
25
26   int main()
27   {
28           Person p;
29           Chinese c;
30           American a;
31           printPerson(p);
32           printPerson(c);
33           printPerson(a);
34           return 0;
35   }
```

执行结果如下：

```
I'm a Person
I'm from China
I'm from USA
```

可以看到，在运行时通过基类 Person 的对象调用派生类 Chinese 和 American 的实现方法。Chinese 和 American 的方法都是通过覆盖基类中的虚函数来实现。

面试例题 7：虚函数是怎么实现的？

考点：C++虚函数实现细节。

出现频率：★★★★

解析

简单地说，虚函数是通过虚函数表实现的。事实上，如果一个类中含有虚函数，则系统会为这个类分配一个指针成员指向一张虚函数表(vtbl)，表中每一项指向一个虚函数的地址，实现上就是一个函数指针的数组。为了说明虚函数表，请看下面的程序代码：

```
1    class Parent
2    {
3    public:
4     virtual void foo1() {}
5     virtual void foo2() {}
6     void foo3();
7    };
8
9     class Child1
10    {
11    public:
12          void foo1() {}
13          void foo3();
14    };
15
16    class Child2
17    {
18    public:
19          void foo1() {}
20          void foo2() {}
21          void foo3();
22    };
```

下面列出了各个类的虚函数表（vtbl）的内容。

Parent 类的 vtbl：Parent::foo1()的地址、Parent::foo1()。

Child1 类的 vtbl：Child1::foo1()的地址、Parent::foo1()。

Child2 类的 vtbl：Child1::foo1()的地址、Child2::foo1()。

可以看出，虚函数表既有继承性又有多态性。每个派生类的虚函数表继承了它各个基类的虚函数表，如果基类虚函数表中包含某一项，则派生类的虚函数表中也将包含同样的一项，但

是两项的值可能不同。如果派生类覆盖(override)了该项对应的虚函数，则派生类虚函数表的该项指向重载后的虚函数，在没有重载的情况下，则沿用基类的值。

在类对象的内存布局中，首先是该类的虚函数表指针，然后才是对象数据。在通过对象指针调用虚函数时，编译器生成的代码将先获取对象类的虚函数表指针，然后调用虚函数表中对应的项。对于通过对象指针调用的情况，在编译期间无法确定指针指向，但是在运行期间执行到调用语句时，这一点已经确定，编译后的调用代码能够根据具体对象获取正确的虚函数表，调用正确的虚函数，从而实现多态性。

分析一下，问题的实质是对于发出虚函数调用的对象指针，在编译期间缺乏更多的信息，而在运行期间却具备足够的信息，但此时已不再进行绑定了，在二者之间的一个过渡是把绑定的信息用通用的数据结构记录下来，该数据结构同对象指针相联系，在编译时只需要使用这个数据结构进行抽象的绑定，而在运行期间将会得到真正的绑定。这个数据结构就是虚函数表。可以看到，实现用户所需的抽象和多态需要进行后绑定，而编译器又是通过抽象和多态实现后绑定的。

面试例题 8：分析代码输出——构造函数调用虚函数。

考点：C++虚拟机制的理解。

出现频率：★★★★★

下面的代码执行的结果是什么？为什么会出现这样的结果？

```
1    #include <stdio.h>
2
3    class A
4    {
5    public:
6    A() { doSth(); }                        //构造函数调用虚函数
7    virtual void doSth() { printf("I am A"); }
8    };
9
10   class B : public A
11   {
12   public:
13          virtual void doSth() { printf("I am B"); }
14   };
15
16   int main()
17   {
18          B b;
19          return 0;
20   }
```

解析

在构造函数中，虚拟机制不会发生作用，因为基类的构造函数在派生类构造函数之前执行，

当基类构造函数运行时，派生类数据成员还没有被初始化。如果基类构造期间调用的虚函数向下匹配到派生类，派生类的函数理所当然会涉及本地数据成员，但是数据成员还没有被初始化，而调用涉及对象还没有被初始化的部分自然是危险的，所以C++会提示编译错误。因此虚函数不会向下匹配到派生类，而是直接执行基类的函数。

答案

在构造函数中，虚拟机制不会发生作用，执行结果如下：

```
I am A
```

面试例题 9：分析代码写输出——虚函数的作用。

考点：C++类虚拟机制的理解。

出现频率：★★★★★

下面的代码执行的结果是什么？

```
1	#include <iostream>
2	using namespace std;
3	class A
4	{
5	public:
6	virtual void print(void)
7	{
8		cout << "A::print()" << endl;
9	}
10	};
11	class B:public A
12	{
13	public:
14		virtual void print(void)
15		{
16			cout << "B::print()" << endl;
17		}
18	};
19	class C:public A
20	{
21	public:
22		void print(void)
23		{
24			cout << "C::print()" << endl;
25		}
26	};
27	void print(A a)
28	{
29		a.print();
30	}
```

```
31    void main(void)
32    {
33        A a, *pa, *pb, *pc;
34        B b;
35        C c;
36
37        pa = &a;
38        pb = &b;
39        pc = &c;
40
41        a.print();
42        b.print();
43        c.print();
44
45        pa->print();
46        pb->print();
47        pc->print();
48
49        print(a);
50        print(b);
51        print(c);
52    }
```

解析

代码第41～第43行，分别使用类A、类B和类C的各个对象来调用其print()成员函数，因此执行的是各个类的print()成员函数。

代码第45～第47行，使用3个类A的指针来调用print()成员函数，而这3个指针分别指向类A、类B和类C的3个对象，由于print()函数是虚函数，因此这里有多态，执行的是各个类的print()成员函数。

代码第49～第51行，全局的print()函数的参数使用传值的方式（注意与传引用的区别，如果是引用，则又是多态），在对象a、b、c分别传入时，在函数栈中会分别生成类A的临时对象，因此执行的都是类A的print()成员函数。

答案

执行结果如下：

```
A::print()
B::print()
C::print()
A::print()
B::print()
C::print()
A::print()
A::print()
A::print()
```

面试例题 10：分析代码写结果——虚函数。

考点：C++类虚拟机制的理解。

出现频率：★★★★★

下面的代码执行的结果是什么？

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   void println(const std::string& msg)
6   {
7       cout << msg << "\n";
8   }
9
10  class Base
11  {
12  public:
13      Base()
14      {
15          println("Base::Base()");
16          virt();
17      }
18      void f()
19      {
20          println("Base::f()");
21          virt();
22      }
23      virtual void virt()
24      {
25          println("Base::virt()");
26      }
27  };
28
29  class Derived : public Base
30  {
31  public:
32      Derived()
33      {
34          println("Derived::Derived()");
35          virt();
36      }
37      virtual void virt()
38      {
39          println("Derived::virt()");
```

```
40          }
41      };
42
43      int main(int argc,char* argv[])
44      {
45          Derived d;
46          Base *pB=&d;
47          pB->f();
48          return 0;
49      }
```

解析

程序分析如下。

代码第45行，构造Derived对象d。首先调用Base的构造函数，然后再调用Derived的构造函数。在Base类的构造函数中，调用了虚函数virt()，此时虚拟机制还没有开始作用（因为是在构造函数中），所以执行的是Base类的virt()函数。同样在Derived类的构造函数中，执行的是Derived类的virt()函数。

代码第47行，通过Base类的指针pB访问Base类的公有成员函数f()。f()函数又调用了虚函数virt()，这里出现多态。由于指针pB是指向Derived类对象的，因此实际执行的是Derived类中的virt()成员。

答案

```
Base::Base()
Base::virt()
Derived::Derived()
Derived::virt()
Base::f()
Derived::virt()
```

面试例题11：选择题——虚函数相关。

考点：C++类虚拟机制的理解。

出现频率：★★★★

分析下面类和变量的定义，并从下面的选择题中选出正确的选项。

```
1       #include <iostream>
2       #include <complex>
3       using namespace std;
4       class Base
5       {
6       public:
7       Base() { cout<<"Base-ctor"<<endl; }
8        ~ Base() { cout<<"Base-dtor"<<endl; }
9       virtual void f(int) { cout<<"Base::f(int)"<<endl; }
10      virtual void f(double) {cout<<"Base::f(double)"<<endl; }
```

```
11      virtual void g(int i = 10) {cout<<"Base::g()"<<i<<endl; }
12      };
13
14      class Derived: public Base
15      {
16      public:
17          Derived() { cout<<"Derived-ctor"<<endl; }
18          ~ Derived() { cout<<"Derived-dtor"<<endl; }
19          void f(complex<double> c){ cout<<"Derived::f(complex)"<<endl;}
20          virtual void g(int i = 20) {cout<<"Derived::g()"<<i<<endl; }
21      };
22      Base b;
23      Derived d;
24      Base* pb = new Derived;
```

（1）cout << sizeof(Base) << endl;

A. 4　　B. 32　　C. 20　　D.与平台相关

（2）cout << sizeof(Derived) << endl;

A. 4　　B. 8　　C. 36　　D.与平台相关

（3）pb->f(1.0);

A. Derived::f(complex)　　B. Base::f(double)　　C. Base::f(int)　　D.Derived::f(double)

（4）pb->g();

A. Base::g()10　　B. Base:g()20　　C. Derived::g()10　　D.Derived::g()20

解析

（1）Base 类没有任何数据成员，并且含有虚函数，所以系统会为它分配一个指针指向虚函数表。指针的大小是 4 字节。

（2）Derived 类没有任何数据成员，它是 Base 的派生类，因此它继承了 Base 的虚函数表。系统会为它分配一个指针指向这张虚函数表。

（3）Base 类中定义了两个 f()的重载函数，Derived 只有一个 f()，其参数类型为 complex，因此 Derived 并没有 Base 的 f()进行覆盖，参数由于参数 1.0 默认是 double 类型的，因此调用的是 Base:: f(double)。

（4）Base 和 Derived 都定义了含有相同参数列表的 g()，因此这里发生多态。Pb 指针指向的是 Derived 类的对象，因此调用的是 Derived 类的 g()。这里要注意，由于参数值是在编译期就已经决定的，因此参数 i 应该取 Base 类的默认值，即 10。

答案

（1）A

（2）A

（3）B

（4）C

10.4 多重继承和虚拟继承

如果派生类只有一个基类，称为单基派生或单一继承。在实际运用中，经常需要派生类同时具有多个基类，这种方法称为多重继承。多重继承中容易产生向上继承的二义性，这个问题由虚拟继承来解决。

面试例题 12：为什么需要多重继承？它的优缺点是什么？

考点：C++多重继承的理解。

出现频率：★★★★

解析

实际生活中，一些事物往往会拥有两个或两个以上事物的属性，为了与实际情况相符，C++引入了多重继承的概念，C++允许为一个派生类指定多个基类，这样的继承结构被称做多重继承。例如，人（Person）派生出作者（Author）和程序员（Programmer），而程序员作者同时拥有作家和程序员的这两个属性，既能编程又能写作，如图 10.2 所示。

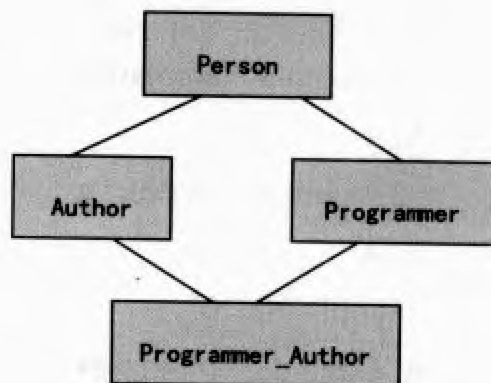


图 10.2 各个类继承关系

使用多重继承的程序代码如下：

```
1	#include <iostream>
2	using namespace std;
3
4	class Person
5	{
6	public:
7	void sleep() {cout << "sleep" << endl;}
8	void eat() {cout << "eat" << endl;}
9	};
10
11	class Author : public Person			//Author 继承自 Person
12	{
13	public:
14		void writeBook() {cout << "write Book" << endl;}
15	};
16
17	class Programmer : public Person		//Programmer 继承自 Person
18	{
19	public:
20		void writeCode() {cout << "write Code" << endl;}
21	};
22
```

```
23    class Programmer_Author : public Programmer, public Author      //多重继承
24    {
25    };
26
27    int main()
28    {
29        Programmer_Author pa;
30
31        pa.writeBook();    //调用基类 Author 的方法
32        pa.writeCode();    //调用基类 Programmer 的方法
33        pa.eat();          //编译错误，eat()定义不明确
34        pa.sleep();        //编译错误，sleep()定义不明确
35
36        return 0;
37    }
```

多重继承的优点很明显，即对象可以调用多个基类中的接口，如代码第 31 行和代码第 32 行对象 pa 分别调用 Author 类的 writeBook()函数和 Programmer 类的 writeCode()函数。

多重继承的缺点是如果派生类所继承的多个基类有相同的基类，而派生类对象需要调用这个祖先类的接口方法，容易产生二义性。代码第 33 行和代码第 34 行就是因为这个原因而出现编译错误的。因为通过多重继承的 Programmer_Author 类复制了 Author 类和 Programmer 类，而 Author 类和 Programmer 类都分别复制了 Person 类，所以 Programmer_Author 类对 Person 类进行了两次复制，在调用 Person 类的接口时，编译器会不清楚需要调用哪一次的复制内容，从而产生错误。对于这个问题通常有两个解决方案：

（1）加上全局符确定调用复制。比如 pa.Author::eat()调用属于 Author 的复制。

（2）使用虚拟继承，使得多重继承类 Programmer_Author 只拥有 Person 类的一份复制。在代码第 11 行和代码第 17 行的继承语句中加入 virtual 就可以了，如下所示：

```
class Author : virtual public Person          //Author 虚拟继承自 Person
class Programmer : virtual public Person      //Programmer 虚拟继承自 Person
```

答案

实际生活中，一些事物往往会拥有两个或两个以上事物的属性，针对这个问题，C++引入了多重继承的概念。多重继承的优点是对象可以调用多个基类的接口。

多重继承的缺点是容易出现继承向上的二义性。

面试例题 13：分析代码找错——多重继承中的二义性。

考点：C++多重继承的理解。

出现频率：★★★★★

下面程序代码中的多重继承有什么错误?

```
1    #include <iostream.h>
2    class cat
3    {
```

```
4	public:
5	void show()
6	{
7		cout<<"cat"<<endl;
8	}
9	};
10
11	class fish
12	{
13	public:
14		void show()
15		{
16			cout<<"fish"<<endl;
17		}
18	};
19
20	class catfish:public cat, public fish
21	{
22	};
23
24	int main()
25	{
26		catfish obj;
27		obj.show();
28
29		return 0;
30	}
```

解析

程序中 catfish 类多重继承 cat 类和 fish 类，因此继承了 cat 的 show()方法和 fish 的 show()方法。由于这两个方法同名，代码第 27 行直接用 obj.show()时，无法区别执行哪个基类的 show()方法，因此会出现编译错误。

答案

代码第 27 行出现编译错误是因为 obj.show()无法区分执行哪个基类的 show()方法。可以改成 obj.car::show()访问 car 的 show()成员。

面试例题 14：多重继承二义性的消除。

考点：多重继承中二义性的消除。

出现频率：★★★★

类 A 派生 B 和 C，类 D 从 B 和 C 派生，如何将类 A 的指针指向类 D 的一个实例？

解析

这道题实际上考查的是如何消除多重继承引起的向上继承二义性问题。程序代码如下所示：

```
1   class A {};
2   class B : public A {};
3   class C : public A {};
4   class D : public B, public C {};
5
6   int main()
7   {
8   D d;
9   A *pd = &d; //编译错误
10      return 0;
11  }
```

由于 B 和 C 继承自 A，B 和 C 都拥有 A 的一份复制，类 D 多重继承自 B 和 C，因此拥有 A 的两份复制，如果此时类 A 的指针指向类 D 的一个实例，会出现“模糊转换”的编译错误。解决办法如以下代码所示：

```
1   class A {};
2   class B : virtual public A {};          //B 虚拟继承自 A
3   class C : virtual public A {};          //C 虚拟继承自 A
4   class D : public B, public C {};
5
6   int main()
7   {
8   D d;
9   A *pd = &d; //成功转换
10      return 0;
11  }
```

将 B、C 都改为虚拟继承自 A，则类 D 多重继承自 B、C 时，这样就不会重复拥有 A 的复制了，因此也就不会出现转换错误了。

答案

把 B、C 都改为虚拟继承自 A，从而消除继承的二义性。

面试例题 15：分析代码写结果——多重继承和虚拟继承。

考点：多重继承和虚拟继承的理解。

出现频率：★★★★

下面的代码输出结果是什么？如果类 Child1 和 Child2 都改为虚拟继承 Parent，输出结果又是什么？

```
1   #include <iostream>
2   using namespace std;
3
4   class Parent
5   {
6   public:
7   Parent() : num(0) { cout << "Parent" << endl;}
```

```
8       Parent(int n) :num(n) {cout << "Parent(int)" << endl;}
9       private:
10      int num;
11      };
12      class Child1 : public Parent
13      {
14      public:
15              Child1() {cout << "Child1()" << endl;}
16              Child1(int num) : Parent(num) {cout << "Child1(int)" << endl;}
17      };
18      class Child2 : public Parent
19      {
20      public:
21              Child2() {cout << "Child2()" << endl;}
22              Child2(int num) : Parent(num) {cout << "Child2(int)" << endl;}
23      };
24      class Derived : public Child1, public Child2
25      {
26      public:
27              Derived() : Child1(0), Child2(1) {}
28              Derived(int num) : Child2(num), Child1(num+1) {}
29      };
30
31      int main()
32      {
33              Derived d(4);
34              return 0;
35      }
```

解析

首先讨论不存在虚拟继承的情况。

多重继承类对象的构造顺序与其继承列表中基类的排列顺序一致，而不是与构造函数的初始化列表顺序一致。在这里 Derived 继承的顺序是 Child1、Child2（代码第 24 行），因此按照下面步骤构造。

（1）构造 Child1。由于 Child1 继承自 Parent，因此先调用 Parent 的构造函数，然后再调用 Child1 的构造函数。

（2）调用 Child2。先调用 Parent 的构造函数，然后再调用 Child2 的构造函数。

（3）调用 Derived 类的构造函数。

因此输出结果为：

```
Parent(int)
Child1(int)
Parent(int)
Child2(int)
```

其次说明 Child1 和 Child2 都改为虚拟继承 Parent 的情况。

当系统碰到多重继承的时候就会自动先加入一个虚拟基类（Parent）的复制，即首先调用虚拟基类（Parent）默认的构造函数，然后再调用派生类（Child1 和 Child2）的构造函数和自己（Derived）的构造函数。由于只生成一份复制，因此以后再也不会调用虚拟基类（Parent）的构造函数了，从而 Child1 和 Child2 指定调用 Parent 的构造函数就无效了。

输出结果为：

```
Parent
Child1(int)
Child2(int)
```

总结一下，多继承中的构造函数顺序如下。

（1）任何虚拟基类的构造函数按照它们被继承的顺序构造。

（2）任何非虚拟基类的构造函数按照它们被构造的顺序构造。

（3）任何成员对象的构造函数按照它们声明的顺序调用。

（4）类自身的构造函数。

答案

不存在虚拟继承时的输出结果为：

```
Parent(int)
Child1(int)
Parent(int)
Child2(int)
```

存在虚拟继承时的输出结果为：

```
Parent
Child1(int)
Child2(int)
```

10.5 纯虚函数和抽象基类

纯虚函数就是基类只定义了函数体，没有实现过程。如果基类含有一个或多个纯虚函数，那么它就属于抽象基类，不能被实例化。

面试例题 16：为什么要引入抽象基类和纯虚函数？

考点：抽象基类和纯虚函数的理解。

出现频率：★★★★★

解析

纯虚函数在基类中没有定义，必须在子类中加以实现。如果基类含有一个或多个纯虚函数，那么它就属于抽象基类，不能被实例化。

引入抽象基类和纯虚函数的原因有以下两点。

（1）为了方便使用多态特性。

（2）在很多情况下，基类本身生成对象是不合情理的。例如，动物作为一个基类可以派生出老虎、狮子等子类，但动物本身生成对象明显不合常理。抽象基类不能够被实例化，它定义的纯虚函数相当于接口，能把派生类的共同行为提取出来。

以上面的动物、老虎、狮子类为例，程序代码如下所示：

```
1    #include<iostream>
2    #include<memory.h>
3    #include<assert.h>
4    using namespace std;
5
6    class Animal
7    {
8    public:
9    virtual void sleep() = 0;          //纯虚函数，必须在派生类被定义
10   virtual void eat() = 0;            //纯虚函数，必须在派生类被定义
11   };
12
13   class Tiger : public Animal
14   {
15   public:
16        void sleep() {cout << "Tiger sleep" << endl;}
17        void eat() {cout << "Tiger eat" << endl;}
18   };
19
20   class Lion : public Animal
21   {
22   public:
23        void sleep() {cout << "Lion sleep" << endl;}
24        void eat() {cout << "Lion eat" << endl;}
25   };
26
27   void main()
28   {
29        Animal *p;      //Animal 指针,不能使用 Animal animal 定义对象
30        Tiger tiger;
31        Lion lion;
32
33        p = &tiger;          //指向 Tiger 对象
34        p->sleep();          //调用 Tiger::sleep()
35        p->eat();            //调用 Tiger::eat()
36        p = &lion;           //指向 Lion 对象
37        p->sleep();          //调用 Lion::sleep()
38        p->eat();            //调用 Lion::eat()
39   }
```

执行结果：

```
Tiger sleep
Tiger eat
Lion sleep
Lion eat
```

程序利用抽象类 Animal 把动物的共同行为抽出来了，即不管是什么动物，都需要睡觉和吃食物。在上面的代码中，Animal 有两个纯虚函数分别对应这两个行为，因此 Animal 为抽象基类，不能被实例化。Animal 的两个纯虚函数 sleep()和 eat()在它的子类 Tiger 和 Lion 中都被定义了（如果子类中有一个基类的纯虚函数没有定义，那么子类也是抽象类）。虽然不能使用 Animal animal 的方式生成 Animal 对象，但可以使用 Animal 的指针指向 Animal 的派生类 Tiger 和 Lion，使用指针调用 Animal 类中的接口（纯虚函数）完成多态。

面试例题 17：虚函数与纯虚函数有什么区别？

考点：虚函数和纯虚函数的理解。

出现频率：★★★★

解析

虚函数和纯虚函数有以下区别。

（1）类里声明虚函数的作用是为了能让这个函数在它的子类里面被覆盖，这样编译器就可以使用后期绑定来达到多态了。纯虚函数只是一个接口，是个函数的声明而已，它要留到子类里去实现。

（2）虚函数在子类里面也可以不重载，但是纯虚函数必须在子类去实现。通常，很多函数加上 virtual 修辞，虽然牺牲掉一些性能，但是增加了面向对象的多态性，可以阻止父类里的这个函数在子类里被修改实现。

（3）虚函数的类用于“实作继承”，也就是说继承接口的同时也继承了父类的实现。当然大家也可以完成自己的实现。纯虚函数的类用于“介面继承”，即纯虚函数关注的是接口的统一性，实现由子类完成。

（4）带纯虚函数的类叫虚基类。这种基类不能直接生成对象，只能被继承，并重写其虚函数后才能使用，这样的类也叫抽象类。

面试例题 18：程序找错——抽象类不能实例化。

考点：抽象类不能实例化的理解。

出现频率：★★★★★

```
1    #include<iostream>
2    using namespace std;
3
4    class Shape
5    {
```

```
6    public:
7        Shape() {}
8        ~ Shape() {}
9        virtual void Draw() = 0;
10   }
11
12   void main()
13   {
14       Shape s1;
15   }
```

答案

Shape 类的 Draw()函数是一个纯虚函数，因此 Shape 类就是一个抽象类，它是不能实例化对象的。因此代码第 14 行出现编译错误。解决办法是把 Draw 函数修改成一般的虚函数或者把 s1 定义成 Shape 的指针。

面试例题 19：应用题——用面向对象的方法进行设计。

考点：面向对象编程的理解。

出现频率：★★★

编写一个图形相关的应用程序，它要具有处理大量图形（Shape）信息的功能，图形有矩形（Rectangle）、正方形（Square）、圆形 （Circle）等种类，应用程序需要计算这些图形的面积，并且需要在某个设备上进行显示（使用在标准输出上打印信息的方式），要求如下。

（1）请用面向对象的方法对以上应用进行设计，编写可能需要的类。

（2）请给出实现以上应用功能的示例性代码，从某处获取图形信息，并且进行计算和绘制。

（3）Square 是否继承自 Rectangle？为什么？并且请比较两种方式的优劣。

解析

显然形状无法具备对象，但是长方形或圆形等具体的图形类有对象。因此 Shape 为抽象类，其派生类有 Rectangle 和 Circle 等具体图形类。

定义形状（Shape）类的用处是因为任何图形都有面积（Area），并且都能被显示（Draw），因此把这些共同的行为抽象出来作为 Shape 类的方法。由于 Shape 类为抽象类，因此这些方法在 Shape 类中就是纯虚函数。代码如下：

```
1    #include<iostream>
2    using namespace std;
3    #define PI 3.14159     //圆周率
4
5    //形状类
6
7    class Shape
8    {
9    public:
10       Shape() {}
```

```
11            ~ Shape() {}
12            virtual void Draw() = 0;          //纯虚函数
13            virtual double Area() = 0;        //纯虚函数
14    };
15
16
17    //长方形类
18
19    class Rectangle : public Shape
20    {
21    public:
22            Rectangle() : a(0), b(0) {}
23            Rectangle(int x, int y) : a(x), b(y) {}
24            virtual void Draw()
25            {
26                    cout << "Rectangle, area: " << Area() << endl;
27            }
28            virtual double Area() {return a * b; }
29    private:
30            int a;
31            int b;
32    };
33
34
35    //圆形类
36
37    class Circle : public Shape
38    {
39    public:
40            Circle(double x) : r(x) {}
41            virtual void Draw()
42            {
43                    cout << "Circle, area: " << Area() << endl;
44            }
45            virtual double Area() { return PI * r * r; }
46    private:
47            double r;
48    };
49
50
51    //正方形类
52
53    class Square : public Rectangle
54    {
55    public:
56            Square(int length) : a(length) {}
```

```
57          virtual void Draw()
58          {
59                  cout << "Square, area: " << Area() << endl;
60          }
61          virtual double Area()
62          {
63                  return a * a;
64          }
65  private:
66          int a;
67  };
68
69
70  int main()
71  {
72          Rectangle rect(10, 20);
73          Square square(10);
74          Circle circle(8);
75
76          Shape *p;                          //抽象类指针
77          p = &rect;
78          cout << p->Area() << endl;         //调用 Rectangle::Area()
79          p->Draw();                         //调用 Rectangle::Draw()
80
81          p = &square;
82          cout << p->Area() << endl;         //调用 Square::Area()
83          p->Draw();                         //调用 Square::Draw()
84
85          p = &circle;
86          cout << p->Area() << endl;         //调用 Circle::Area()
87          p->Draw();                         //调用 Circle::Draw()
88
89          return 0;
90  }
```

在主函数 main()中，使用 Shape 类的指针去访问不同图形类的 Draw()和 Area()方法。这样，Shape 类中的 Draw()和 Area()纯虚函数被认为是接口，只要使用 Shape 类指针操作这些接口即可，而不用关心子类中的具体实现。

实际上正方形也可以直接继承于 Shape 类，但由于正方形可以被看成是长和宽相等的长方形，可以被认为是一种特殊的长方形，所以它继承自 Rectangle 类。

这样做的好处是操作方便，例如 Rectangle 类中可以存在一个如下的虚函数：

```
virtual void foo() {cout << "Rectangle" << endl;}
```

注意这个虚函数表示的是长方形的属性，而不是形状（Shape）的属性，并且如果 foo()同时属于正方形，那可以在 Square 类中对其进行覆盖：

```
virtual void foo() {cout << "Square" << endl;}
```

这样就可以用 Rectangle 类指针操作 Square 类对象以达到多态。

当然也会带来一些性能上的问题。Square 类继承自 Rectangle 类，于是 Square 继承了 Rectangle 的虚表，如果 Rectangle 存在不同于 Shape 类的虚函数，这张虚表所包括的项目就会增加。因此 Square 会有更多的虚表使用开销，导致程序执行效率下降。

10.6 COM（组件对象模型）

面试例题 20：什么是 COM（组件对象模型）？

考点：理解 COM（组件对象模型）。

出现频率：★★★

解析

Windows 使用 DLLs（动态链接库）在二进制级共享代码，这也是 Windows 程序运行的关键——重用 kernel32.dll, user32.dll 等。但 DLLs 是针对 C 接口写的，它们只能被 C 或理解 C 调用规范的语言使用，由编程语言实现共享代码，而不是由动态链接库本身实现。这样动态链接库的使用受到了限制。

COM（Component Object Model，组件对象模型）通过定义二进制标准解决这些问题。这是因为 COM 明确指出二进制模块（动态链接库和可执行文件）必须被编译成与指定的结构匹配。这个标准也确切规定了在内存中如何组织 COM 对象。COM 定义的二进制标准还必须独立于任何编程语言（如 C++中的命名修饰）。一旦满足了这些条件，就可以轻松地从任何编程语言中存取这些模块。由编译器所产生的二进制代码可以与标准兼容。

在内存中，COM 对象的这种标准形式在 C++虚函数中可用，这是许多 COM 代码使用 C++的原因。但是注意这与编写模块所用的语言无关，因为二进制代码是所有语言都可用的。

答案

COM 即组件对象模型，定义了一种二进制标准，使得任何编程语言都能存取它所编写的模块。

面试例题 21：COM 组件有什么特点？

考点：对 COM（组件对象模型）特点的理解。

出现频率：★★★

答案

COM 组件是遵循 COM 规范编写，以 Win32 动态链接库（DLL） 或可执行文件（EXE）形式发布的可执行二进制代码，能够满足组件架构的所有需求。遵循 COM 的规范标准，组件与应用、组件与组件之间可以相互操作，极其方便地建立可伸缩的应用系统。COM 是一种技术

标准，其商业品牌被称为 ActiveX。

组件在应用开发方面具有以下特点。

（1）组件与开发工具语言无关。开发者可以根据特定情况选择特定语言工具实现组件的开发。编译之后的组件以二进制的形式发布，可以跨 Windows 平台使用，而且源程序代码不会外泄，有效地保证了组件开发者的版权。

（2）通过接口有效保证了组件的复用性。一个组件具有若干个接口，每个接口代表组件的某个属性或方法。其他组件或应用程序可以设置或调用这些属性和方法来进行特定的逻辑处理，组件和应用程序的连接是通过其接口实现的。负责集成的开发者无需了解组件功能是如何实现的，只需要简单地创建组件对象并与其接口建立连接。在保证接口一致性的前提之下，可以调换组件，更新版本，也可以把组件安插在不同的应用系统中。

（3）组件运行效率高，便于使用和管理。因为组件是二进制代码，运行效率比 ASP 脚本高很多。核心的商务逻辑计算任务必须由组件来担当，ASP 脚本只起组装的角色。而且组件在网络上的位置可以被透明分配，组件和使用它的程序能在同一进程中、不同进程或不同机器上运行，并且组件之间是相互独立的。组件对象通过一个内部引用计数器来管理它的生存周期，这个计数器存放任何时候连接到该对象的客户数。当引用计数变为 0 时，对象可以把自己从内存中释放掉，这使程序员不必考虑与提供可共享资源有关的问题。

面试例题 22：如何理解 COM 对象和接口？

考点：COM 对象和接口的理解。

出现频率：★★★

答案

一个对象实现一个接口，即对象使用代码实现接口的每个方法并且为这些函数通向 COM 库提供了 COM 的二进制指针，然后 COM 使这些函数运行在请求了一个指向该接口的任何客户端。

COM 在接口的定义和实现上有根本的差别。接口实际上是由一组定义了用法的相互联系的函数原型组成，只是它不能够被实现。这些函数原型就相当于 C++中含有纯虚拟函数的基类。

一个接口定义了接口的成员函数、调用方法、返回类型、它们的参数类型和参数数量，以及这些函数要干什么。但是这里并没有与接口实现相关的东西。

接口的实现就是程序员在一个接口定义上提供的执行相关动作的代码。客户调用完全取决于接口的定义。接口实现的一个实例就是一个指向一组方法的指针，即指向一个接口的函数表，该函数表引用了该接口所有方法的实现。在 COM 模型中，对象本身对于客户来说是不可见的，客户请求服务时，只能通过接口实现，每个接口都由一个 128 位的全局标识符（GUID，Globally unique Identifier）来标识。客户通过 GUID 获得接口的指针，并通过接口指针调用其成员函数。

一个 COM 接口与 C++类是不一样的。一个 COM 接口本身对客户来说是不可见的，它不是一个对象，只是简单地关联一组函数，是客户和程序之间通信的二进制标准。只要它提供了指向接口方法的指针，对象就可以用任何语言来实现。与接口类似，每个对象也用一个 GUID 来

标识，称为 CLSID（class ID）。

继承在 COM 里并不意味着代码的重用。因为接口没有实现关联，一个接口同一个合同关联，就像 C++的纯虚拟基类的创建和修改一样，可以添加方法或者加强方法的使用。在 COM 里没有选择性继承。如果一个接口由另一个接口继承的话，它就包含了另一个接口定义的所有的方法。

管理实现 COM 对象的 IUnknown::QueryInterface 方法有 3 个主要规则：

（1）对象必须要有一个标识符。

（2）对象实例的接口集合必须是静态的（static）。

（3）在对象中从任何其他的接口查询此接口都应该成功。

通过引用计数来管理对象的生命周期。

```
AddRef ( )      //增加引用
Realase ( )     //减少引用
```

面试例题 23：什么是 ActiveX，简述 DCOM。

考点：ActiveX 以及 DCOM 的理解。

出现频率：★★★

答案

ActiveX 是 Microsoft 提出的一套基于 COM 的构件技术标准，实际上是对象嵌入与链接（OLE）的新版本。

基于分布式环境下的 COM 被称作 DCOM（Distribute COM，分布式组件对象模型），它实现了 COM 对象与远程计算机上的另一个对象之间相互交互。DCOM 定义了分散对象创建和对象间通信的机制，DCOM 是 ActiveX 的基础，因为 ActiveX 主要是针对 Internet 应用开发（与 OLE 相比）的技术，当然也可以用于普通的桌面应用程序。

面试例题 24：什么是“DLL HELL”？

考点：对“DLL HELL”的理解。

出现频率：★★

答案

“DLL HELL”是指 DLL（动态链接库）版本冲突的问题。一般情况下 DLL 新版本会覆盖旧版本，那么使用旧版本的应用程序会不能继续正常工作。

扩展知识：虚函数表。

虚函数（Virtual Function）是通过一张虚函数表（Virtual Table）来实现的。在这个表中，主要是一个类的虚函数的地址，这张表解决了继承、覆盖的问题，其内容真实反映了实际函数。这样，在有虚函数的类的实例中虚函数表被分配在实例的内存中，所以，当用父类的指针操作一个子类的时候，这张虚函数表就显得尤为重要了，它就像一张地图，指明了实际所应该调用

的函数。

在 C++的标准规格说明书中写道，编译器必须保证虚函数表的指针存在于对象实例中最前面的位置（这是为了保证正确取到虚函数的偏移量）。这意味着通过对象实例的地址可以得到虚函数表，然后就可以遍历其中的函数指针，并调用相应的函数了。请看下面代码：

```
1   #include <iostream>
2   using namespace std;
3
4   class Base
5   {
6   public:
7   virtual void fun1() {cout << "Base::fun1" << endl;}
8   virtual void fun2() {cout << "Base::fun2" << endl;}
9   virtual void fun3() {cout << "Base::fun3" << endl;}
10  private:
11      int num1;
12      int num2;
13  };
14
15  typedef void (*Fun)(void);
16
17  int main()
18  {
19      Base b;
20      Fun pFun;
21
22      pFun = (Fun)*((int*)*(int*)(&b)+0);      //取得 Base::fun1()地址
23      pFun();                                   //执行 Base::fun1()
24      pFun = (Fun)*((int*)*(int*)(&b)+1);      //取得 Base::fun1()地址
25      pFun();                                   //执行 Base::fun2()
26      pFun = (Fun)*((int*)*(int*)(&b)+2);      //取得 Base::fun1()地址
27      pFun();                                   //执行 Base::fun3()
28
29      return 0;
30  }
```

上面程序的执行结果如下：

```
Base::fun1
Base::fun2
Base::fun3
```

可以看到通过函数指针 pFun 的调用，分别执行了对象 b 的 3 个虚函数。通过这个示例发现，可强行把&b 转成 int *，取得虚函数表的地址，然后再次取址就可以得到第 1 个虚函数的地址了，也就是 Base::fun1()。如果调用 Base::fun2()和 Base::fun3()，只需要把&b 先加上数组元素的偏移，后面的步骤类似。

程序中的 Base 对象 b 内存结构图，如图 10.3 所示。

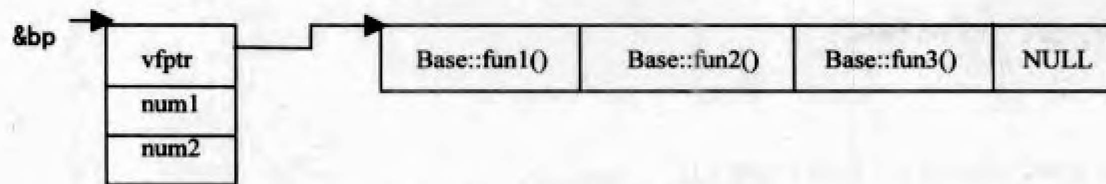


图 10.3 Base 虚函数表图

对于一个单继承的类，如果它有虚拟函数，则只有一张虚函数表。对于多重继承的类，它可能有多张虚函数表。

考虑下面代码中的各个类的定义：

```
1    #include <iostream>
2    using namespace std;
3
4    class Base1
5    {
6    public:
7    Base1(int num) : num_1(num) {}
8    virtual void foo1() {cout << "Base1::foo1 " << num_1 << endl;}
9    virtual void foo2() {cout << "Base1::foo2 " << num_1 << endl;}
10       virtual void foo3() {cout << "Base1::foo3 " << num_1 << endl;}
11   private:
12       int num_1;
13   };
14
15   class Base2
16   {
17   public:
18       Base2(int num) : num_2(num) {}
19       virtual void foo1() {cout << "Base2::foo1 " <<num_2 << endl;}
20       virtual void foo2() {cout << "Base2::foo2 " <<num_2 << endl;}
21       virtual void foo3() {cout << "Base2::foo3 " <<num_2 << endl;}
22   private:
23       int num_2;
24   };
25
26   class Base3
27   {
28   public:
29       Base3(int num) : num_3(num) {}
30       virtual void foo1() {cout << "Base3::foo1 " << num_3 << endl;}
31       virtual void foo2() {cout << "Base3::foo2 " << num_3 << endl;}
32       virtual void foo3() {cout << "Base3::foo3 " << num_3 << endl;}
33   private:
34       int num_3;
```

```
35  };
36
37  class Derived1 : public Base1
38  {
39  public:
40          Derived1(int num) : Base1(num) {}
41          virtual void faa1() {cout << "Derived1::faa1" << endl;}        //无覆盖
42          virtual void faa2() {cout << "Derived1::faa2" << endl;}
43  };
44
45  class Derived2 : public Base1
46  {
47  public:
48          Derived2(int num) : Base1(num) {}
49          virtual void foo2() {cout << "Derived2::foo2" << endl;}        //只覆盖了 Base1::foo2
50          virtual void fbb2() {cout << "Derived2::fbb2" << endl;}
51          virtual void fbb3() {cout << "Derived2::fbb3" << endl;}
52  };
53
54  class Derived3 : public Base1, public Base2, public Base3          //多重继承，无覆盖
55  {
56  public:
57          Derived3(int num_1, int num_2, int num_3) :
58                  Base1(num_1), Base2(num_2), Base3(num_3) {}
59          virtual void fcc1() {cout << "Derived3::fcc1" << endl;}
60          virtual void fcc2() {cout << "Derived3::fcc2" << endl;}
61  };
62
63  class Derived4 : public Base1, public Base2, public Base3          //多重继承，有覆盖
64  {
65  public:
66          Derived4(int num_1, int num_2, int num_3) :
67                  Base1(num_1), Base2(num_2), Base3(num_3) {}
68          virtual void foo1() {cout << "Derived4::foo1" << endl;}        //覆盖了 Base1::foo1,
69                                                                         //Base2::foo1, Base3::foo1
70          virtual void fdd() {cout << "Derived4::frr" << endl;}
71  };
```

这个例子说明了 4 种继承情况下的虚函数表。

（1）一般继承（无虚函数覆盖）：Derived1 类继承自 Base1 类，Derived1 的虚函数表如图 10.4 所示。

Derived1 类内没有任何覆盖基类 Base1 的函数，因此两个虚拟函数 foo1()和 foo2()被依次添加到了虚函数表的末尾。

（2）一般继承（有虚函数覆盖）：Derived2 类继承自 Base1 类，并对 Base1 类中的虚函数 foo2()进行了覆盖。Derived2 的虚函数表如图 10.5 所示。

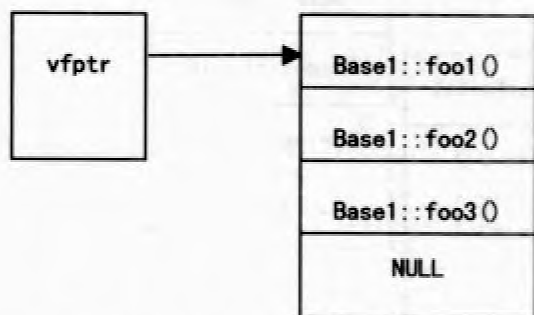


图 10.4 Derived1 虚函数表图

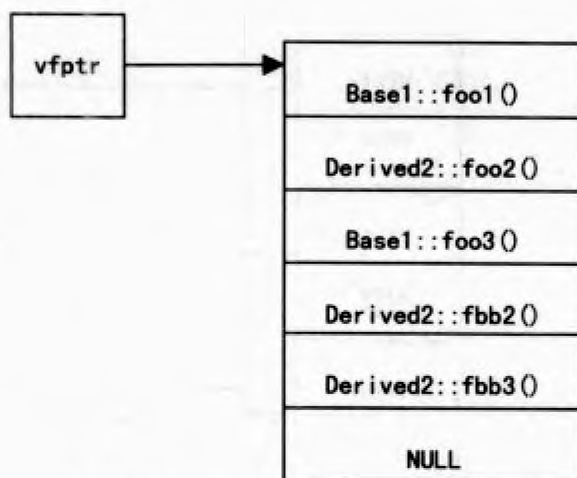


图 10.5 Derived2 虚函数表图

Derived2 覆盖了基类 Base1 的 foo1()，因此其虚函数表中 Derived2::foo2()替换了 Base1::foo2()，foo2()和 foo3()被依次添加到虚函数表的末尾。

（3）多重继承（无虚函数覆盖）：Derived3 类继承自 Base1 类、Base2 类、Base3 类，Derived3 的虚函数表如图 10.6 所示。

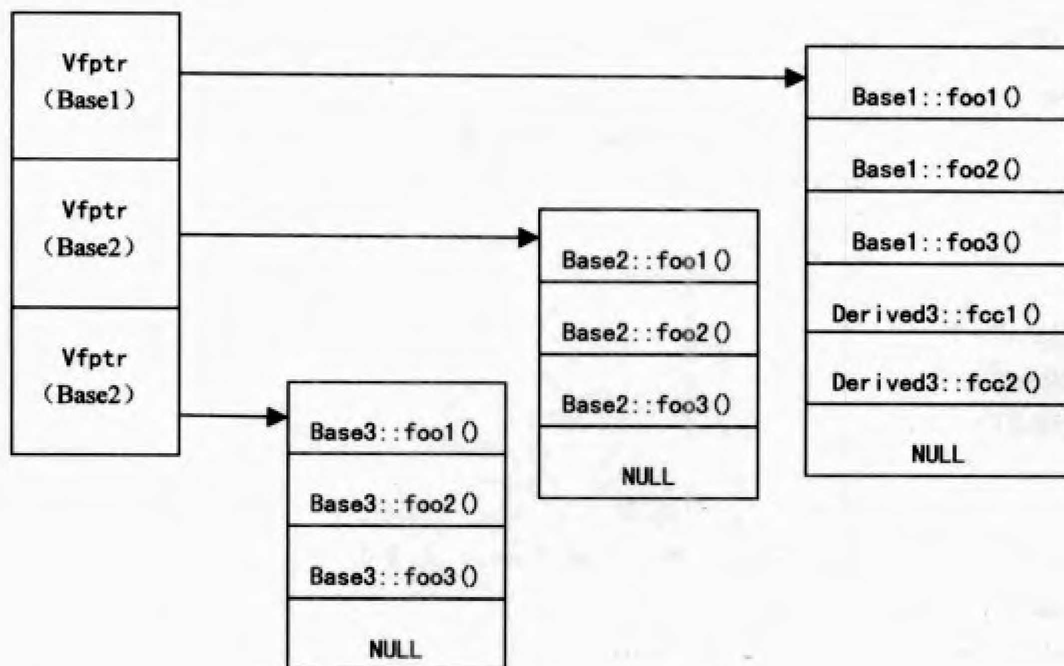


图 10.6 Derived3 虚函数表图

每个父类都有自己的虚表，Derived3 也有了 3 个虚表，这里父类虚表的顺序与声明继承父类的顺序一致。这样做是为了解决不同的父类类型的指针指向同一个子类实例，而能够调用到实际的函数。例如：

```
Base2 *pBase2 = new Derived3();
pBase2->foo2();    //调用 Base2::foo2()
```

把 Base2 类型的指针指向 Derive3 实例，那么调用是对应 Base2 虚表里的函数。

（4）多重继承（有虚函数覆盖）：Derived4 类继承 Base1 类、Base2 类、Base3 类，并对 Base1 类的 foo1()、Base2 类的 foo1()、Base3 类的 foo1()都进行覆盖。Derived4 的虚函数表如图 10.7 所示。

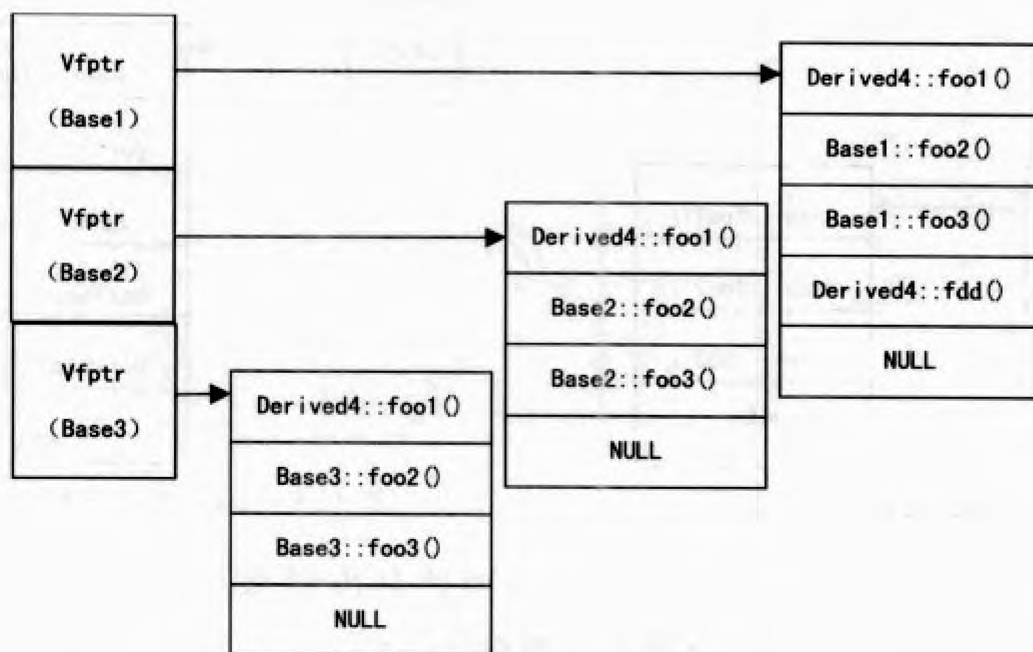


图 10.7 Derived4 虚函数表图

可以看出，Base1::foo1()、Base2::foo1()和 Base3::foo1()都被替换成了 Derived::foo1()。这样，我们就可以把任一个静态类型的父类指向子类，并调用子类的 f()函数了。如下程序所示：

```
1    Base1*pBase1 = new Derived4();
2    pBase1->foo1();                //调用从 Base1 继承的虚表中的 Derived4::foo1()
```

下面是我们所讨论的 4 种继承情况下的测试代码：

```
1    int main()
2    {
3    Base1 *pBase1 = NULL;
4    Base2 *pBase2 = NULL;
5    Base3 *pBase3 = NULL;
6
7    cout << "-----   一般继承自 Base1，无覆盖   ----------" << endl;
8    Derived1 d1(1);          //Derived1 一般继承自 Base1，无覆盖
9    pBase1 = &d1;
10       pBase1->foo1();   //执行 Base1::foo1();
11
12       cout << "-----   一般继承自 Base1，覆盖 foo2()   ---------" << endl;
13       Derived2 d2(2);          //Derived2 一般继承自 Base1，覆盖了 Base1::foo2()
14       pBase1 = &d2;
15       pBase1->foo2(); //执行 Derived2::foo2();
16
17       cout << "-----   多重继承，无覆盖   ----------" << endl;
18       Derived3 d3(1, 2, 3);    //Derived3 多重继承自 Base1,Base2,Base3,没有覆盖
19       pBase1 = &d3;
20       pBase2 = &d3;
21       pBase3 = &d3;
22       pBase1->foo1();          //执行 Base1::foo1();
23       pBase2->foo1();          //执行 Base2::foo1();
```

```
24          pBase3->foo1();           //执行 Base3::foo1();
25
26          cout << "------   多重继承，覆盖 foo1()   ---------" << endl;
27          Derived4 d4(1, 2, 3);     //Derived4 多重继承自 Base1,Base2,Base3,覆盖 foo1()
28          pBase1 = &d4;
29          pBase2 = &d4;
30          pBase3 = &d4;
31          pBase1->foo1();           //执行 Derived4::foo1();
32          pBase2->foo1();           //执行 Derived4::foo1();
33          pBase3->foo1();           //执行 Derived4::foo1();
34          return 0;
35      }
```

测试结果如下：

```
一般继承自 Base1，无覆盖  ----------
Base1::foo1 1
一般继承自 Base1，覆盖 foo2()
Derived2::foo2
-----  多重继承，无覆盖  ----------
Base1::foo1 1
Base2::foo1 2
Base3::foo3 3
------  多重继承，覆盖 foo1()  ---------
Derived4::foo1
Derived4::foo1
Derived4::foo1
```

第11章 数据结构

数据结构主要研究数据的组织方式以及相应的操作方法。它除了描述数据本身之外，还描述数据之间的相互关系。它不仅是一般程序设计的基础，而且是设计编译程序、操作系统、数据库、人工智能及其他大型应用程序的基础。如今数据结构在计算机科学中占有重要的地位，对于相当多的程序设计来说，认清数据的内在关系，可获得对问题的正确认识，看清问题的结构甚至解法。在一定意义上，程序所描述的就是在数据结构上实现的算法。算法的设计依赖于数据的逻辑结构，算法的实现依赖于数据的存储结构，所以数据结构选择的好坏，对程序的质量影响甚大。掌握基本的数据结构知识，是程序设计水平提高的必要条件。

因此无论在笔试还是面试中，数据结构都是重要的考察内容。

11.1 单 链 表

单链表的结构是数据结构中最简单的，它的每一个节点只有一个指向后一个节点的指针，其模型如图 11.1 所示。

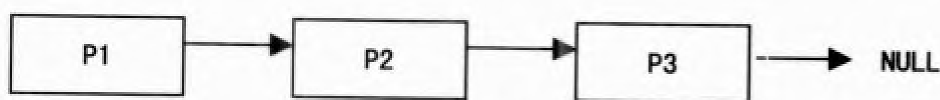


图 11.1 单链表模型

面试例题 1：**编程实现单链表的建立。**

考点：单链表的操作。

出现频率：★★★★

解析

链表节点的定义：

```
typedef struct node
{
    int data;        //节点内容
```

```
    node *next;    //下一个节点
}node;
```

单链表的创建：

```
1    //创建单链表
2    node *create()
3    {
4        int i = 0;                          //链表中数据的个数
5        node *head, *p, *q;
6        int x = 0;
7        head = (node *)malloc(sizeof(node)); //创建头节点
8
9        while(1)
10        {
11            printf("Please input the data: ");
12            scanf("%d", &x);
13            if (x == 0)                 //data 为 0 时创建结束
14                break;
15            p = (node *)malloc(sizeof(node));
16            p->data = x;
17            if (++i == 1)
18            {                           //链表只有一个元素
19                head->next = p;         //连接到 head 的后面
20            }
21            else
22            {
23                q->next = p;            //连接到链表尾端
24            }
25            q = p;                      //q 指向末节点
26        }
27        q->next = NULL;                 //链表的最后一个指针为 NULL
28        return head;
29    }
```

上面的代码中，while 循环每次从终端读入一个整型数据，并调用 malloc 动态分配链表节点内存存储这个整型数据，然后再插入到单链表的末尾。最后当数据为 0 时表示插入数据结束，此时把末尾节点的 next 指针置为 NULL。

面试例题 2：编程实现单链表的测长。

考点：单链表的操作。

出现频率：★★★★

解析

单链表的测长：

```
//返回单链表长度
int length(node *head)
```

```
{
    int len = 0;
    node *p;
    p = head->next;
    while(p != NULL)                    //遍历链表
    {
        len++;
        p = p->next;
    }
    return len;
}
```

由于链表末尾节点的 next 指针被置为 NULL，因此可以使用 while 循环进行遍历链表所有节点，当遇到 NULL 时结束循环。

面试例题 3：编程实现单链表的打印。

考点：单链表的操作。

出现频率：★★★★

解析

```
1   //打印单链表
2   void print(node *head)
3   {
4       node *p;
5       int index = 0;
6       if (head->next == NULL)        //链表为空
7       {
8           printf("Link is empty!\n");
9           return;
10      }
11      p = head->next;
12      while(p != NULL)               //遍历链表
13      {
14          printf("The %dth node is: %d\n", ++index, p->data);      //打印元素
15          p = p->next;
16      }
17  }
```

单链表的打印与单链表的测长方法类似，使用 while 循环进行遍历链表所有节点并打印各个节点内容，当遇到 NULL 时结束循环。

面试例题 4：编程实现单链表的节点查找。

考点：单链表的操作。

出现频率：★★★★

解析

单链表的查找节点，请看下面的代码：

```
1   //查找单链表pos位置的节点,返回节点指针
2   //pos从0开始,0返回head节点
3   node *search_node(node *head, int pos)
4   {
5       node *p = head->next;
6       if (pos < 0)                    //pos位置不正确
7       {
8           printf("incorrect position to search node!\n");
9           return NULL;
10      }
11      if (pos == 0)                   //在head位置，返回head
12      {
13          return head;
14      }
15      if(p == NULL)
16      {
17          printf("Link is empty!\n");  //链表为空
18          return NULL;
19      }
20
21      while(--pos)
22      {
23          if ((p = p->next) == NULL)
24          {
25              printf("incorrect position to search node!\n");
26              break;              //超出链表返回
27          }
28      }
29      return p;
30  }
```

面试例题 5：编程实现单链表的节点插入。

考点：单链表的操作。

出现频率：★★★★

解析

在单链表中某个位置（第pos个节点）后面插入节点，这里分为插入到链表首部、插入到链表中间，以及链表尾端3种情况，请看下面的代码：

```
1   //在单链表pos位置处插入节点，返回链表头指针
2   //pos从0开始计算,0表示插入到head节点后面
3   node *insert_node(node *head, int pos, int data)
4   {
5       node *item = NULL;
6       node *p;
7
```

```
8          item = (node *)malloc(sizeof(node));
9          item->data = data;
10         if (pos == 0)                     //插入链表头后面
11         {
12                head->next = item;         //head 后面是 item
13                return head;
14         }
15         p = search_node(head, pos);       //获得位置 pos 的节点指针
16         if (p != NULL)
17
18                item->next = p->next;      //item 指向原 pos 节点的后一个节点
19                p->next = item;            //把 item 插入到 pos 的后面
20         }
21         return head;
22     }
```

面试例题 6：编程实现单链表的节点删除。

考点：单链表的操作。

出现频率：★★★★

解析

```
1     //删除单链表的 pos 位置的节点，返回链表头指针
2     //pos 从 1 开始计算，1 表示删除 head 后的第一个节点
3     node *delete_node(node *head, int pos)
4     {
5          node *item = NULL;
6          node *p = head->next;
7          if (p == NULL)                          //链表为空
8          {
9                printf("link is empty!\n");
10               return NULL;
11         }
12         p = search_node(head, pos-1);           //获得位置 pos 的节点指针
13         if (p != NULL && p->next != NULL)
14         {
15               item = p->next;
16               p->next = item->next;
17               delete item;
18         }
19         return head;
20    }
```

下面代码是上面各个函数的测试程序：

```
1     int main()
2     {
3          node *head = create();                  //创建单链表
```

```
4        printf("Length: %d\n", length(head));    //测量单链表长度
5        head = insert_node(head, 2, 5);          //在第 2 个节点之后插入 5
6        printf("insert integer 5 after 2th node: \n");
7    print(head);                                 //打印单链表
8        head = delete_node(head, 2);             //删除第 2 个节点
9        printf("delete the 3th node: \n");
10       print(head);
11       return 0;
12   }
```

程序执行结果：

```
Please input the data: 1
Please input the data: 2
Please input the data: 3
Please input the data: 4
Please input the data: 0
Length: 4
insert integer 5 after 2th node:
The 1th node is: 1
The 1th node is: 2
The 1th node is: 5
The 1th node is: 3
The 1th node is: 4
delete the 3th node:
The 1th node is: 1
The 2th node is: 5
The 3th node is: 3
The 4th node is: 4
```

面试例题 7：编程实现单链表的逆置。

考点：单链表的操作。

出现频率：★★★★

解析

关于这个问题，最容易想到的方法是遍历一遍链表，利用一个辅助指针，存储遍历过程中当前指针指向下一个元素，然后将当前节点元素的指针反转，利用已经存储的指针往后继续遍历。

```
1    node *reverse(node *head)
2    {
3        node *p, *q, *r;
4
5        if (head->next == NULL)         //链表为空
6        {
7            return head;
8        }
9
10       p = head->next;
```

```
11          q = p->next;                    //保存原第二个节点
12          p->next = NULL;                 //原第一个节点为末节点
13
14          while(q != NULL)                //遍历，各个节点的 next 指针反转
15          {
16              r = q->next;
17              q->next = p;
18              p = q;
19              q = r;
20          }
21          head->next = p;                 //新的第一个节点为原末节点
22          return head;
23      }
```

面试例题 8：寻找单链表的中间元素。

考点：单链表的操作。

出现频率：★★★★

解析

这里采用一遍扫描的方法，描述如下。

假设 mid 指向当前已经扫描的子链表的中间元素，cur 指向当前已扫描链表的尾节点，继续扫描即移动 cur 到 cur->next，这时只需判断一下是否移动 mid 到 mid->next 就可以了。所以一遍扫描就能找到中间位置。代码如下：

```
1   node *search(node *head)
2   {
3       int i = 0;
4       int j = 0;
5       node *current = NULL;
6       node *middle = NULL;
7
8       current = middle = head->next;
9       while(current != NULL)
10      {
11          if( i / 2 > j)
12          {
13              j++;
14              middle = middle->next;
15          }
16          i++;
17          current = current->next;
18      }
19
20      return middle;
21  }
```

面试例题 9：单链表的正向排序。

考点：单链表的操作。

出现频率：★★★★

解析

下面是结构体定义的代码：

```
1   typedef struct node
2   {
3       int data;
4       node *next;
5   }node;
6
7   node* InsertSort(void)
8   {
9       int data = 0;
10      struct node *head=NULL,*New,*Cur,*Pre;
11      while(1)
12      {
13          printf("please input the data\n");
14          scanf("%d", &data);
15          if (data == 0)                    //输入 0 结束
16          {
17              break;
18          }
19          New=(struct node*)malloc(sizeof(struct node));
20          New->data = data;                 //新分配一个 node 节点
21          New->next = NULL;
22          if(head == NULL)
23          {                                 //第一次循环时对头节点赋值
24              head=New;
25              continue;
26          }
27          if(New->data <= head->data)
28          { //head 之前插入节点
29             New->next = head;
30             head = New;
31             continue;
32          }
33          Cur = head;
34          while(New->data > Cur->data &&   //找到需要插入的位置
35                Cur->next!=NULL)
36          {
37             Pre = Cur;
38             Cur = Cur->next;
39          }
```

```
40              if(Cur->data >= New->data)       //位置在中间
41              {                                //把 new 节点插入到 Pre 和 cur 之间
42                  Pre->next = New;
43                  New->next = Cur;
44              }
45              else                             //位置在末尾
46                  Cur->next = New;             //把 new 节点插入到 cur 之后
47          }
48          return head;
49      }
```

面试例题 10：判断单链表是否存在环型链表问题。

考点：单链表的操作。

出现频率：★★★★

解析

这里有一种比较简单的解法。假设两个指针分别为 p1 和 p2。每循环一次 p1 向前走一步，p2 向前走两步，直到 p2 碰到 NULL 指针或者两个指针相等时循环结束。如果两个指针相等则说明存在环。

程序代码如下：

```
1   //判断是否存在回环
2   //如果存在，start 存放回环开始的节点
3   bool IsLoop(node* head, node **start)
4   {
5       node* p1=head, *p2 = head;
6
7       if (head == NULL || head->next ==NULL)
8       {                                       //head 为 NULL 或
9           return false;                       //链表为空时返回 false
10      }
11      do
12      {
13          p1 = p1->next;                  //p1 走一步
14          p2 = p2->next->next;            //p2 走两步
15      } while(p2 && p2->next && p1 != p2);
16
17      if(p1 == p2)
18      {
19          *start = p1;                    //p1 为回环开始节点
20          return true;
21      }
22      else
23          return false;
24  }
25
26
```

```
27    int main()
28    {
29        bool bLoop = false;
30        node *head = create();                    //创建单链表
31        node *start = head->next->next->next;  //使第 4 个节点为回环开始位置
32        start->next = head->next;                //回环连接到第 2 个节点
33
34            node *loopStart = NULL;
35        bLoop = IsLoop(head, &loopStart);
36        printf("bLoop = %d\n", bLoop);
37        printf("bLoop == loopStart ? %d\n", (loopStart == start));
38        return 0;
39    }
```

main()函数中对 IsLoop()函数做了测试，这里代码第 31 行和代码第 32 行手动把第 2 个节点接到了原来的第 4 个节点之后，于是节点 4 就成了回环开始的节点。因此代码第 36 行和代码第 37 行的打印语句输出都是 1。

面试例题 11：有序单链表的合并。

考点：单链表的操作。

出现频率：★★★★

已知两个链表 head1 和 head2 各自有序，请把它们合并成一个链表并且其依然有序，分别使用非递归方法以及递归方法。

解析

首先介绍非递归方法。因为两个链表 head1 和 head2 都是有序的，所以只需要把较短链表的各个元素有序地插入到较长的链表之中就可以了。

程序代码如下：

```
1     node* insert_node(node *head, node *item) //head != NULL
2     {
3         node *p = head;
4         node *q = NULL;  //始终指向 p 之前的节点
5
6         while(p->data < item->data && p != NULL)
7         {
8             q = p;
9             p = p->next;
10        }
11        if (p == head)     //插入到原头节点之前
12        {
13            item->next = p;
14            return item;
15        }
16        //插入到 q 与 p 之间之间
```

```
17          q->next = item;
18          item->next = p;
19          return head;
20     }
21
22  /* 两个有序链表进行合并 */
23  node* merge(node* head1, node* head2)
24  {
25          node* head;              //合并后的头指针
26          node *p;
27          node *nextP;             //指向 p 之后
28
29          if ( head1 == NULL )  //有一个链表为空的情况，直接返回另一个链表
30          {
31               return head2;
32          }
33          else if ( head2 == NULL )
34          {
35               return head1;
36          }
37
38          // 两个链表都不为空
39          if(length(head1) >= length(head2))      //选取较短的链表
40          {                                       //这样进行的插入次数要少些
41                 head = head1;
42                 p = head2;
43           }
44           else
45           {
46                 head = head2;
47                 p = head1;
48           }
49
50           while(p != NULL)
51           {
52                 nextP = p->next;                //保存 p 的下一个节点
53                 head = insert_node(head, p);    //把 p 插入到目标链表中
54                 p = nextP;                      //指向将要插入的下一个节点
55          }
56
57          return head;
58     }
```

这里 insert_node()函数是有序节点的插入，注意与前面例题中的函数有区别，这里它传入的参数是 node*类型。然后在 merge()函数中（代码第 52～第 55 行）循环把短链表中的所有节点插入到长链表中。

接下来介绍递归方法。假设有下面两个链表。

链表 1：1->3->5。

链表 2：2->4->6。

递归方法的步骤如下。

（1）比较链表 1 和链表 2 的第 1 个节点数据，由于 1<2，因此把结果链表头节点指向链表 1 中的第 1 个节点，即数据 1 所在的节点。

（2）对剩余的链表 1（3->5）和链表 2 再调用本过程，比较得到结果链表的第 2 个节点，即 2 与 3 比较得到 2。此时合并后的链表节点为 1->2。如此递归直到两个链表的节点都被加到结果链表中，程序代码如下所示：

```
1    node * MergeRecursive(node *head1, node *head2)
2    {
3        node *head = NULL;
4
5        if (head1 == NULL)
6        {
7            return head2;
8        }
9        if (head2 == NUL)
10       {
11           return head1;
12       }
13
14       if ( head1->data < head2->data )
15       {
16           head = head1 ;
17           head->next = MergeRecursive(head1->next,head2);
18       }
19       else
20       {
21           head = head2 ;
22           head->next = MergeRecursive(head1,head2->next);
23       }
24
25       return head ;
26   }
```

下面是测试程序：

```
1    int main()
2    {
3        node *head1 = create();          //创建单链表 1
4        node *head2 = create();          //创建单链表 2
5        //node *head = merge(head1, head2);
6        node *head = MergeRecursive(head1, head2);
7        print(head);
```

```
8
9       return 0;
10  }
```

通过使用merge()函数和MergeRecursive()函数测试，其结果一致。

11.2 循环链表

循环链表与单链表一样，是一种链式的存储结构。所不同的是，循环链表的最后一个节点的指针指向该循环链表的第1个节点或者表头节点，从而构成一个环形的链。

循环链表的运算与单链表的运算基本一致。所不同的有以下几点。

- 在建立一个循环链表时，必须使其最后一个节点的指针指向表头节点，而不是像单链表那样置为NULL。此种情况适用于在最后一个节点后插入一个新的节点。
- 判断是否到表尾采用判断该节点链域的值是否是表头节点的方法，当链域值等于表头指针时，说明已到表尾，而不是像单链表那样判断链域值是否为NULL。

循环链表的结构模型如图11.2所示。

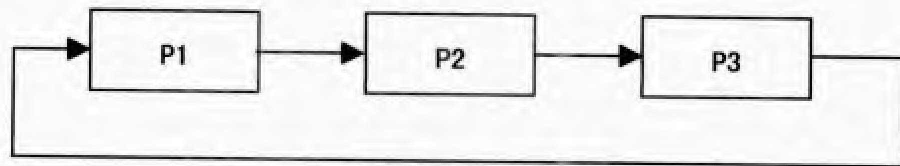


图11.2 循环链表模型

面试例题12：约瑟夫问题的解答。

考点：循环链表的操作。

出现频率：★★★★

编号为1～N的N个人按顺时针方向围坐一圈，每人持有一个密码(正整数)，开始任选一个正整数作为报数上限值M，从第1个人按顺时针方向自1开始顺序报数，报到M时停止报数。报M的人出列，将他的密码作为新的M值，从他顺时针方向上的下一个人开始从1报数，如此下去，直至所有人全部出列为止。试设计一个程序求出出列顺序。

解析

显然当有人退出圆圈后，报数的工作要从下一个人开始继续，而其剩的人仍然围成一个圆圈，因此可以使用循环单链表。退出圆圈的工作对应着表中节点的删除操作，对于这种删除操作频繁的情况，选用效率较高的链表结构，为了程序指针每一次都指向一个具体的代表一个人的节点而不需要判断，链表不带头节点。因此，对于所有人围成的圆圈所对应的数据结构采用一个不带头节点的循环链表来描述。设头指针为p，并根据具体情况移动。

为了记录退出的人的先后顺序，采用一个顺序表进行存储。程序结束后再输出依次退出人

的编号顺序。由于只记录各个节点的数据值就可以，所以定义一个整型一维数组。如：在“int quit[n]”中n为根据实际问题定义的足够大的整数。

程序代码如下：

```
1   #include <iostream>
2   using namespace std;
3
4   /* 结构体和函数声明 */
5   typedef struct node
6   {
7       int data;
8       node *next;
9   } node;
10
11  node *node_create(int n);
12
13  //构造节点数量为 n 的单向循环链表
14  node * node_create(int n)
15  {
16      node *pRet = NULL;
17
18      if (0 != n)
19      {
20          int n_idx     = 1;
21          node *p_node   = NULL;
22
23          /* 构造 n 个 node */
24          p_node = new node[n];
25          if (NULL == p_node) //申请内存失败，返回 NULL
26          {
27              return NULL;
28          }
29          else
30          {
31              memset(p_node, 0, n * sizeof(node)); //初始化内存
32          }
33          pRet = p_node;
34          while (n_idx < n) //构造循环链表
35          {                       //初始化链表的每个节点,从 1 到 n
36              p_node->data = n_idx;
37              p_node->next = p_node + 1;
38              p_node = p_node->next;
39              n_idx++;
40          }
41          p_node->data = n;
42          p_node->next = pRet;
43      }
```

```
44
45        return pRet;
46    }
47
48    int main()
49    {
50        node *pList = NULL;
51        node *pIter = NULL;;
52        int n = 20;
53        int m = 6;
54
55        /* 构造单向循环链表 */
56        pList = node_create(n);
57
58        /* 约瑟夫 循环取数 */
59        pIter = pList;
60        m %= n;
61        while (pIter != pIter->next)
62        {
63            int i = 1;
64
65            /* 取到第 m-1 个节点 */
66                for (; i < m - 1; i++)
67            {
68                pIter = pIter->next;
69            }
70
71            /* 输出第 m 个节点的值 */
72            printf("%d ", pIter->next->data);
73
74            /* 从链表中删除第 m 个节点 */
75            pIter->next = pIter->next->next;
76            pIter = pIter->next;
77        }
78        printf("%d\n", pIter->data);
79
80        /* 释放申请的空间 */
81        delete []pList;
82        return 0;
83    }
```

11.3 双向链表

双向链表其实是单链表的改进。对单链表进行操作时，有时要对某个节点的直接前驱进行

操作，又必须从表头开始查找。由于单链表每个节点只有一个直接后继节点存储地址的链域，因此运用单链表是无法办到的。这样就引出了一个既有存储直接后继节点地址的链域，又有存储直接前驱节点地址的链域的双链域节点结构，即双向链表。

在双向链表中，节点除含有数据域外，还有两个指针：一个存储直接后继节点地址，另一个存储直接前驱节点地址。

双向链表的模型图如图 11.3 所示。

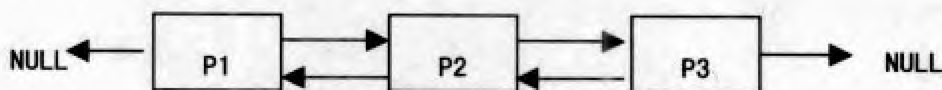


图 11.3 双向链表模型

面试例题 13：建立一个双向链表。

考点：双向链表的操作。

出现频率：★★★★

解析

双向链表的定义如下：

```
1    typedef struct DbNode
2    {
3        int data;              //节点数据
4        DbNode *left;          //前驱节点指针
5        DbNode *right;         //后继节点指针
6    } DbNode;
```

建立双向链表（为方便，这里定义了 3 个函数）如下所示。

- CreateNode()根据数据创建一个节点，返回新创建的节点。
- CreateList()函数根据一个节点数据创建链表的表头，返回表头节点。
- AppendNode()函数总在表尾插入新节点（其内部调用 CreateNode()生成节点），返回表头节点。

```
1    //根据数据创建创建节点
2    DbNode *CreateNode(int data)
3    {
4        DbNode *pnode = (DbNode *)malloc(sizeof(DbNode));
5        pnode->data = data;
6        pnode->left = pnode->right = pnode;   //创建新节点时
7                                              //让其前驱和后继指针都指向自身
8    return pnode;
9    }
10
11   //创建链表
12   DbNode *CreateList(int head)              //参数给出表头节点数据
13   {                                         //表头节点不作为存放有意义数据的节点
```

```
14          DbNode *pnode = (DbNode *)malloc(sizeof(DbNode));
15          pnode->data = head;
16          pnode->left = pnode->right = pnode;
17
18          return pnode;
19      }
20
21      //总是在表尾插入新节点，返回表头节点
22      DbNode *AppendNode(DbNode *head, int data)   //参数 1 是链表的表头节点
23      {                                             //参数 2 是要插入的节点,其数据为 data
24          DbNode *node = CreateNode(data);          //创建数据为 data 的新节点
25          DbNode *p = head, *q;
26
27          while(p != NULL)
28          {
29              q = p;
30              p = p->right;
31          }
32          q->right = node;
33          node->left = q;
34
35          return head;
36      }
```

可以使用其中的 CreateList()和 AppendNode()来生成一个链表，下面是一个生成从节点 0～9 的循环链表。

```
1   DbNode *head = CreateList(0);             //生成表头，表头数据为 0
2
3   for (int i = 1; i < 10; i++)
4   {
5       head = AppendNode(head, i);           //添加 9 个节点，数据为从 1 到 9
6   }
```

面试例题 14：双向链表的测长。

考点：双向链表的操作。

出现频率：★★★★

解析

为了得到双向链表的长度，需要使用 right 指针进行遍历，直到得到 NULL 为止。

```
1   //获取链表的长度
2   int GetLength(DbNode *head)               //参数为链表的表头节点
3   {
4       int count = 1;
5       DbNode *pnode = NULL;
6
7       if (head == NULL)                     //head 为 NULL 表示链表空
```

```
8         {
9             return 0;
10        }
11        pnode = head->right;
12        while (pnode != NULL)
13        {
14            pnode = pnode->right;              //使用 right 指针遍历
15            count++;
16        }
17
18        return count;
19    }
```

面试例题 15：双向链表的打印。

考点：双向链表的操作。

出现频率：★★★★

解析

与测量长度的方法一样，使用 right 指针进行遍历。

```
1     //打印整个链表
2     void PrintList(DbNode *head)              //参数为链表的表头节点
3     {
4         DbNode *pnode = NULL;
5
6         if (head == NULL)                     //head 为 NULL 表示链表空
7         {
8             return;
9         }
10        pnode= head;
11        while (pnode != NULL)
12        {
13            printf("%d ", pnode->data);
14            pnode = pnode->right;          //使用 right 指针遍历
15        }
16        printf("\n");
17    }
```

面试例题 16：双向链表的节点查找。

考点：双向链表的操作。

出现频率：★★★★

解析

使用 right 指针遍历，直至找到数据为 data 的节点，如果找到节点，则返回节点，否则返回 NULL，如下所示：

```
1   //查找节点，成功则返回满足条件的节点指针，否则返回 NULL
2   DbNode *FindNode(DbNode *head, int data)    //参数 1 是链表的表头节点
3   {                                            //参数 2 是要查找的节点,其数据为 data
4       DbNode *pnode = head;
5   
6       if (head == NULL)                        //链表为空时返回 NULL
7       {
8           return NULL;
9       }
10  
11      /*找到数据或者到达链表末尾退出 while 循环*/
12      while (pnode->right != NULL && pnode->data != data)
13      {
14          pnode = pnode->right;                //使用 right 指针遍历
15      }
16  
17      //没有找到数据为 data 的节点，返回 NULL
18      if (pnode->right == NULL)
19      {
20          return NULL;
21      }
22  
23      return pnode;
24  }
```

面试例题 17：双向链表的节点插入。

考点：双向链表的操作。

出现频率：★★★★

解析

节点 P 后插入一个节点，这里分为两种情况：一种是插入位置在中间，另一种是插入位置在末尾。两种情况有一点不同：插入位置在中间时，需要把 P 的原后继节点的前驱指针指向新插入的节点。

```
1   //在 node 节点之后插入新节点
2   void InsertNode(DbNode *node, int data)
3   {
4       DbNode *newnode = CreateNode(data);
5       DbNode *p = NULL;
6   
7       if (node == NULL)                //node 为 NULL 时返回 NULL
8       {
9           return NULL;
10      }
11      if (node->right == NULL)         //node 为最后一个节点
12      {
```

```
13              node->right = newnode;
14              newnode->left = node;
15          }
16          else
17          {                                   //node 为中间节点
18              newnode->right = node->right;   //newnode 向右连接
19              node->right->left = newnode;
20              node->right = newnode;          //newnode 向左连接
21              newnode->left = node;
22          }
23      }
```

面试例题 18：双向链表的节点删除。

考点：双向链表的操作。

出现频率：★★★★

解析

这里有 3 种情况：删除头节点、删除中间节点以及删除末节点。

下面的 DeleteNode()删除数据为 data 的节点，并返回表头节点。如果不存在节点，则删除失败返回 NULL，如果删除后的链表为空，也返回 NULL。

```
1   //删除满足指定条件的节点，返回表头节点，删除失败返回 NULL（失败的原因是该节点不存在）
2   DbNode *DeleteNode(DbNode *head, int data)          //参数 1 是链表的表头节点
3   {                                                   //参数 2 是要插入的节点,其数据为 data
4       DbNode *ptmp = NULL;
5
6       DbNode *pnode = FindNode(head, data);           //查找节点
7       if (NULL == pnode)                              //节点不存在，返回 NULL
8       {
9           return NULL;
10      }
11      else if (pnode->left == NULL)                   //node 为第一个节点
12      {
13          head = pnode->right;
14          if (head != NULL)                           //链表不为空
15          {
16              head->left = NULL;
17          }
18      }
19      else if (pnode->right == NULL)                  //node 为最后一个节点
20      {
21          pnode->left->right = NULL;
22      }
23      else                                            //node 为中间的节点
24      {
25          pnode->left->right = pnode->right;
```

```
26                pnode->right->left = pnode->left;
27        }
28
29        free(pnode);                                    //释放已被删除的节点空间
30        return head;
31    }
```

11.4 双向循环链表

双向循环链表其实就是把双向链表的首尾相连。双向循环链表的模型图如图 11.4 所示。

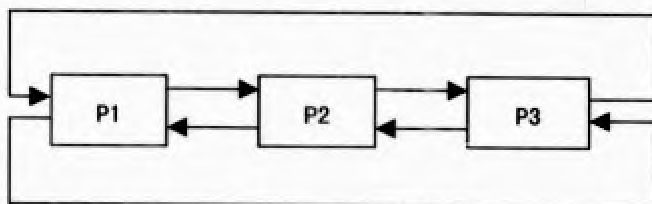


图 11.4 双向循环链表模型

面试例题 19：实现有序双向循环链表的插入操作。

考点：双向循环循环链表的操作。

出现频率：★★★

解析

程序代码如下：

```
1    #include <stdio.h>
2    #include <stdlib.h>
3
4    typedef struct DbNode
5    {
6        int data;           //节点数据
7        DbNode *left;       //前驱节点指针
8        DbNode *right;      //后继节点指针
9    } DbNode;
10
11   //根据数据创建创建节点
12   DbNode *CreateNode(int data)
13   {
14       DbNode *pnode = (DbNode *)malloc(sizeof(DbNode));
15       pnode->data = data;
16       pnode->left = pnode;
17       pnode->right = pnode;        //创建新节点时
18                                    //让其前驱和后继指针都指向本身
19       return pnode;
```

```
20  }
21
22  //在表尾插入新节点，返回表头节点
23  DbNode *AppendNode(DbNode *head, int data) //参数 1 是链表的表头节点
24  {                                          //参数 2 是要插入的节点,其数据为 data
25      DbNode *node = CreateNode(data);       //创建数据为 data 的新节点
26      DbNode *p = NULL;
27      DbNode *q = NULL;
28
29      if (head == NULL)
30      {
31          return NULL;
32      }
33
34      q = p = head->right;
35      while(p != head)
36      {
37          q = p;
38          p = p->right;
39      }
40      q->right = node;                       //与原最后一个节点互连
41      node->left = q;
42      node->right = head;                    //与头节点互连，形成环状
43      head->left = node;
44
45      return head;
46  }
47
48  //打印整个链表
49  void PrintList(DbNode *head)               //参数为链表的表头节点
50  {
51      DbNode *pnode = NULL;
52
53      if (head == NULL)                      //head 为 NULL 表示链表空
54      {
55          return;
56      }
57      printf("%d ", head->data);
58      pnode= head->right;
59      while (pnode != head)
60      {
61          printf("%d ", pnode->data);
62          pnode = pnode->right;              //使用 right 指针遍历
63      }
64      printf("\n");
65  }
```

```
66
67  //插入一个有序链表（从小到大排列），返回表头
68  DbNode *InsertNode(DbNode *head, int data)
69  {
70      DbNode *p = NULL, *q = NULL;
71      DbNode *node = NULL;
72
73      node = CreateNode(data);                //新建数据节点
74      if (head == NULL)                       //空链表,返回新建节点
75      {
76          head =   node;
77          return node;
78      }
79
80      if (head->data > data)                  //data 小于表头数据，插入到表头之前
81      {                                       //把新建节点作为表头
82          head->left->right = node;           //末节点后继指针指向 node
83          node->left = head->left;            //node 的前驱指针指向末节点
84          node->right = head;                 //node 的后继指针指向 head
85          head->left = node;                  //head 的前驱指针指向 node
86          return node;
87      }
88
89      p = head->right;
90      while(p->data <= data && p != head)
91      {
92          p = p->right;
93      }
94
95      p = p->left;
96      q = p->right;
97
98      // 把 node 插入 p 和 q 之间
99      p->right = node;
100     node->left = p;
101     node->right = q;
102     q->left = node;
103
104     return head;
105 }
106
107 int main()
108 {
109     DbNode *head = CreateNode(1);       //创建表头节点，其数据为 1
110     AppendNode(head, 3);                //添加节点 3 6 8
111     AppendNode(head, 6);
```

```
112         AppendNode(head, 8);
113         PrintList(head);
114         head = InsertNode(head, 0);         //插入 0，位置在开头
115         PrintList(head);
116         head = InsertNode(head, 4);         //插入 4，位置在中间
117         PrintList(head);
118         head = InsertNode(head, 10);        //插入 10，位置在末尾
119         PrintList(head);
120
121         return 0;
122  }
```

CreateNode()根据数据创建节点，包括头节点和一般节点。创建头节点时，left 和 right 指针都指向本身从而形成环状。

AppendNode()插入新节点，总是在末尾插入，返回表头节点。

PrintList()打印整个链表，由于属于循环链表，因此遍历回到 head 节点时结束打印。

InsertNode()数据插入一个升序的链表，返回表头节点。这里插入位置分为表头和非表头两种情况。如果是表头，返回的指针是新的节点，否则返回的是原来的表头节点。

main()函数建立了一个升序链表，并对它的不同位置进行了插入操作。

执行结果如下：

```
 1 3 6 8
0 1 3 6 8
0 1 3 4 6 8
0 1 3 4 6 8 10
```

面试例题 20：删除两个双向循环链表的相同节点。

考点：双向链表的操作。

出现频率：★★★

有两个双向循环链表 A、B，知道其头指针为：pHeadA，pHeadB，请写一函数将两链表中数据值相同的节点删除。

解析

本题可以这样来处理。

（1）首先把 A 中含有与 B 中相同的数据节点抽出来组成一个新的链表，例如：

链表 A：1 2 3 4 2 6 4

链表 B：10 20 3 4 2 10

新建链表 C: 2 3 4

（2）然后遍历链表 C，删除 A 和 B 的所有节点。如果 A 含有数据相等节点（A 有两个 4），且 B 也含有此数据节点（B 也有 4），则 A 中数据相等的节点全部被删除。

根据以上分析，实现了以下函数：

GetLink()函数，其程序代码如下：

```
1   //GetLink()通过 headA 链表和 headB 链表中相同数据节点，得到一个新的链表
2   DbNode *GetLink(DbNode *headA, DbNode *headB)
3   {
4       int i = 0;
5       DbNode *newHead = NULL;     //返回新的链表
6       DbNode *pnodeA = headA;     //遍历 headA
7       DbNode *pnodeB = headB;     //遍历 headB
8       DbNode *pnode = NULL;       //遍历 newHead
9
10      do
11      {
12          DbNode *node = NULL;    //用于查看新的链表中是否已经存在节点数据
13          pnodeB = headB;
14
15          if ((node = FindNode(newHead, pnodeA->data)) != NULL)
16          {
17              //若 newHead 中已含有此节点数据，则不考虑此节点的比较，进行下一次遍历
18              pnodeA = pnodeA->right;
19              continue;
20          }
21          do
22          {
23              if (pnodeA->data == pnodeB->data)    //找到 A 与 B 中相同的数据节点
24              {
25                  i++;
26                  if (i == 1)         //第一次生成头节点
27                  {
28                      newHead = CreateNode(pnodeA->data);
29                  }
30                  else                //不是第一次，添加节点
31                  {
32                      AppendNode(newHead, pnodeA->data);
33                  }
34                  break;
35              }
36              pnodeB = pnodeB->right;     //对链表 B 进行遍历
37          } while(pnodeB != headB);
38          pnodeA = pnodeA->right;         //对链表 A 进行遍历
39      } while(pnodeA != headA);
40
41      return newHead;
42  }
```

GetLink()把 A 和 B 的两个链表头节点指针传入，返回一个链表头节点指针。这里有下面几点需要说明。

- 由于要遍历 A 和 B，因此使用了嵌套的循环。外层是对链表 A 进行遍历，内层是对链表 B 进行遍历。

- 当找到 A、B 相同数据后，由于新建表头节点（使用 CreateNode 函数）与插入节点（AppendNode 函数）不同，因此区别对待代码第 25～第 33 行。
- 出于效率考虑，代码第 15～第 20 行首先判断 A 当前节点的数据是否已经在新建的链表中，如果已经存在，则不进行 B 的遍历，继续进行 A 的下一个节点的判断。

DeleteNode 函数，其程序代码如下：

```
1   //删除链表中所有数据等于 Value 的节点
2   DbNode *DeleteNode(DbNode *pHeader, int Value)
3   {
4       DbNode *pNode = NULL;
5       DbNode *pNodeRight = NULL;
6       int bRet = 0;
7
8       if (pHeader == NULL)
9           return NULL;
10      while(pHeader->data == Value)          //头节点的数据为 Value,删除
11      {
12          pNode = pHeader->right;
13          if (pHeader == pHeader->right)  //链表只剩下一个元素
14          {
15              free(pHeader);              //删除节点，此时链表为空，返回 NULL
16              return NULL;
17          }
18          //链表还有其他节点，把原来头节点的左右两个节点相连
19          pHeader->left->right = pHeader->right;
20          pHeader->right->left = pHeader->left;
21          free(pHeader);                  //释放头节点内存
22          pHeader = pNode;                //把原来头节点的下一个节点作为新的头节点
23      }
24
25      pNode = pHeader->right;            //要删除的不是头节点
26      while(pNode != pHeader)            //遍历链表直到回到头节点
27      {
28          pNodeRight = pNode->right;  //保存下一个节点
29          if (pNode->data == Value)   //如果搜索到 Value,删除此节点
30          {
31              //把此节点的原来的左右两个节点相连
32              pNode->left->right = pNodeRight;
33              pNodeRight->left = pNode->left;
34              free(pNode);
35          }
36          pNode = pNodeRight;         //指向下一个节点
37      }
38
39      return pHeader;
40  }
```

DeleteNode()传入的参数为链表头节点的指针以及需要删除的数据值，返回删除后的头节点指针，有下面几点需要说明。

- 当删除的节点为头节点时，此时需要考虑两个情况，一种是链表中只有一个头节点，另一种是删除后的头节点数据也等于 Value 值。代码第 13～17 行对第 1 种情况进行了判断，代码第 10 行的 while 循环是出于第 2 种情况的考虑。并且头节点为下一个节点，即 pHeader 发生了变化。
- 当删除的节点不是头节点时，简单地循环搜索数据为 Value 的节点并删除，直到当前节点回到头节点后结束。

DeleteEqual ()函数，其程序代码如下：

```
1   //DeleteEqual 删除 ppHeadA 与 ppHeadB 两个链表所有含相同数据的节点
2   //参数 ppHeadA 和 ppHeadB 分别为链表 A 和链表 B 头节点指针的地址
3   void DeleteEqual(DbNode **ppHeadA, DbNode **ppHeadB)
4   {
5       DbNode *pHeadA = *ppHeadA, *pHeadB = *ppHeadB;
6       DbNode *head = NULL;
7
8       if (ppHeadA == NULL || ppHeadB == NULL)      //ppHeadA 或 ppHeadB 不合法
9       {
10          return;
11      }
12      if (pHeadA == NULL || pHeadB == NULL)        //链表 A 或 B 有一个为空
13      {
14          return;
15      }
16
17      head = GetLink(pHeadA, pHeadB);      //获得 A 和 B 的相同数据节点所构成的链表
18      while(head != NULL)
19      {
20          pHeadA = DeleteNode(pHeadA, head->data);   //删除 A 中所有 head 的数据节点
21          pHeadB = DeleteNode(pHeadB, head->data);   //删除 A 中所有 head 的数据节点
22          head = DeleteNode(head, head->data);        //删除 head 当前节点
23      }
24
25      *ppHeadA = pHeadA; //返回 ppHeadA
26      *ppHeadB = pHeadB; //返回 ppHeadB
27  }
```

DeleteEqual()是最终的调用函数，它首先判断传入参数的有效性，然后调用前面的 GetLink()得到链表 A 和链表 B 的相同数据所构成的新链表（代码第 17 行），最后调用前面的 DeleteNode()循环删除各个链表中的节点（代码第 18～第 23 行）。

另外注意一点，由于 DeleteEqual()函数有可能删除链表 A 和 B 的头节点，为了保存函数返回后 A 和 B 的头节点，参数必须使用指针的地址或者指针的引用。这里使用的是指针的地址，所以在最后（代码第 25 行和代码第 26 行）保存链表头节点指针。

11.5 队 列

队列（Queue）是一种数据结构，可以在队列的一端插入元素而在队列的另一端删除元素，它有以下特点。

- 允许删除的一端称为队头（front）。
- 允许插入的一端称为队尾（rear）。
- 当队列中没有元素时称为空队列。
- 队列亦称作先进先出（First In First Out）的线性表，简称为 FIFO 表。队列的修改依据先进先出的原则，新来的成员总是加入队尾（即不能中间插入），每次离开的成员总是队头的成员（不允许中途离队）。

多任务系统是一个典型的队列示例，假设有 5 个任务等待执行，它们将被放入一个队列，如果有第 6 个任务要执行，它将被放在队列的末尾。队列中首位任务首先执行。

面试例题 21：编程实现队列的入队、出队、测长、打印。

考点：队列的各项基本操作。

出现频率：★★★★

解析

队列的实现可以使用链表和数组，本题中采用单链表来实现队列。如图 11.5 所示，构造一个如下结构的队列，使用下面结构体定义队列和队列的节点。

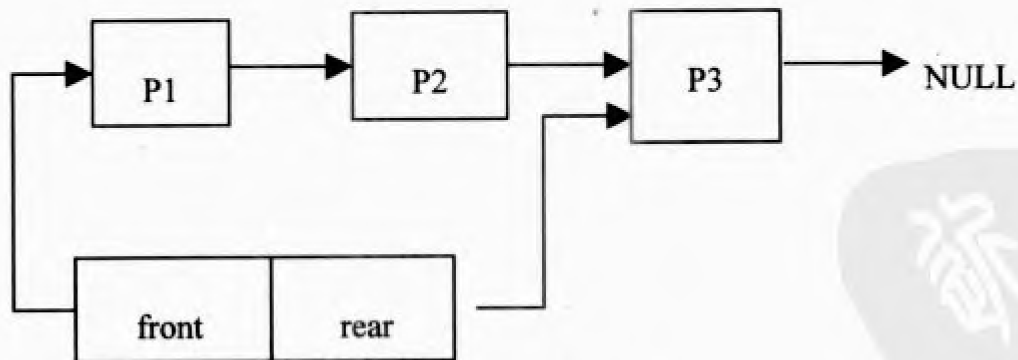


图 11.5 链表实现队列

```
1    typedef struct _Node
2    {
3         int data;
4         struct _Node *next; //指向链表下一个指针
5    } node;
6
7    typedef struct _Queue
```

```
8    {
9        //node 表示队列中的每个节点元素，My Queue 表示队列
10       node *front;          //队头
11       node *rear;           //队尾
12   } MyQueue;
```

在进行编程之前，首先要构造一个空的队列，代码如下：

```
1    //构造空的队列
2    MyQueue *CreateMyQueue()
3    {
4        MyQueue *q = (MyQueue *)malloc(sizeof(MyQueue));
5        q->front = NULL;      //把队首指针置空
6        q->rear = NULL;       //把队尾指针置空
7        return q;
8    }
```

下面进行各个函数的编程。

（1）入队。

```
1    //入队，从队尾一端插入节点
2    MyQueue *enqueue(MyQueue *q, int data)
3    {
4        node *newP = NULL;
5        newP = (node *)malloc(sizeof(node));  //新建节点
6        newP->data = data;                     //复制节点数据
7        newP->next = NULL;
8        if (q->rear == NULL)
9        {
10           //若队列为空，新节点既是队首又是队尾
11           q->front = q->rear = newP;
12       }
13       else
14       {
15           //若队列不为空，新节点放到队尾，队尾指针指向新节点
16           q->rear->next = newP;
17           q->rear = newP;
18       }
19       return q;
20   }
```

由队列的特点可知，入队操作是在队尾一端进行插入的，enqueue 函数的说明如下。

- 代码第 5～第 7 行，根据数据 data 新建一个数据节点。
- 代码第 8～第 12 行，如果队列为空，把新建的节点作为队首，同时也作为队尾。

注意：此时不仅要操作 rear 指针，同时也要操作队列的 front 指针。

- 代码第 13～18 行，如果队列不为空，把新建的节点连接到队尾，并将它作为新的队尾，此时只需要对 front 指针进行操作。

（2）出队。

```
1   // 出队，从队头一端删除节点
2   MyQueue *dequeue(MyQueue *q)
3   {
4       node *pnode = NULL;
5       pnode = q->front;                    //指向队头
6       if (pnode == NULL)                   //队列为空
7       {
8           printf("Empty queue!\n");
9       }
10      else
11      {
12          q->front = q->front->next;       //新队头
13          if (q->front == NULL)            //若删除后队列为空时，对 rear 置空
14          {
15              q->rear = NULL;
16          }
17          free(pnode);                     //删除原队头节点
18      }
19      return q;
20  }
```

出队与入队的操作不同，它是在队首一端进行删除的，dequeue 函数的说明如下。

- 代码第 5 行，指针 pnode 指向队头节点，也就是即将要删除的指针节点。
- 代码第 6～第 9 行，如果队列为空，打印“Empty queue”消息并返回。
- 代码第 12～第 18 行，把队头节点往后移，然后删除原来的对头节点，如果删除后的队列为空，则把 rear 指针置空。

为了简单起见，dequeue 函数删除节点的同时释放了节点内存，如果要得到被删除的节点，可以不释放内存并且返回此节点。

（3）测长。

```
1   //队列的测长
2   int GetLength(MyQueue *q)
3   {
4       int nlen = 0;
5       node *pnode = q->front;              //指向队头
6       if (pnode != NULL)                   //队列不为空
7       {
8           nlen = 1;
9       }
10      while(pnode != q->rear)              //遍历队列
11      {
12          pnode = pnode->next;
13          nlen++;                          //循环一次 nlen 递增 1
14      }
```

```
15        return nlen;
16    }
```

GetLength 函数的实现很简单，只需要遍历一次队列中的节点就可以获得，只有一点需要注意：在代码 10 行中，循环结束的条件是"pnode != q->rear"，而不应该与 NULL 做比较，即"pnode != NULL"，这是因为队尾有可能指向的不是一个链表的末节点。

（4）打印。

```
1    //队列的打印
2    void PrintMyQueue(MyQueue *q)
3    {
4        node *pnode = q->front;
5        if (pnode == NULL)                  //队列为空
6        {
7            printf("Empty Queue!\n");
8            return;
9        }
10       printf("data: ");
11       while(pnode != q->rear)             //遍历队列
12       {
13           printf("%d ", pnode->data);     //打印节点数据
14           pnode = pnode->next;
15       }
16       printf("%d ", pnode->data);         //打印队尾节点数据
17   }
```

PrintMyQueue 函数的实现也很简单。与 GetLength 函数一样，都需要进行一次遍历队列中的节点，而且循环结束的条件也是"pnode != q->rear"，最后打印队尾节点数据（代码第 16 行）。

下面是对上面各个函数的简单测试：

```
1    int main()
2    {
3        int nlen = 0;
4        MyQueue *hp = CreateMyQueue();      //建立队列
5        enqueue(hp, 1); //入队 1 2 3 4
6        enqueue(hp, 2);
7        enqueue(hp, 3);
8        enqueue(hp, 4);
9        nlen = GetLength(hp);               //获得队列长度
10       printf("nlen = %d\n", nlen);
11       PrintMyQueue(hp);                   //打印队列数据
12       dequeue(hp);                        //出队两次
13       dequeue(hp);
14       nlen = GetLength(hp);               //再次获得队列长度
15       printf("\nnlen = %d\n", nlen);
16       PrintMyQueue(hp);                   //再次打印队列数据
17       return 0;
18   }
```

执行结果如下：

```
nlen = 4
data: 1 2 3 4
nlen = 2
data: 3 4
```

11.6 栈

栈（Stack）是一种数据结构，只能从表的一端进行插入和删除。它有以下特点。

- 插入、删除的一端为栈顶（Top），另一端为栈底（Bottom）。
- 当表中没有元素时称为空栈。
- 当栈中没有元素时称为空队列。
- 栈为后进先出（Last In First Out）的线性表，简称为 LIFO 表，栈的修改是依后进先出的原则进行的。每次删除（退栈）的总是当前栈中“最新的”元素，即最后插入（进栈）的元素，而最先插入的是被放在栈的底部，要到最后才能删除。

面试例题 22：队列和栈有什么区别？

考点：队列和栈的区别。

出现频率：★★★★★

解析

队列与栈是两种不同的数据结构，它们有以下的区别。

- 操作的名称不同。队列的插入称为入队，队列的删除称为出队。栈的插入称为进栈，栈的删除称为出栈。
- 可操作的方向不同。队列是在队尾入队，队头出队，即两边都可操作。而栈的进栈和出栈都是在栈顶进行的，无法对栈底直接进行操作。
- 操作的方法不同。队列是先进先出（FIFO），即队列的修改是依据先进先出的原则进行的。新来的成员总是加入队尾（即不能中间插入），每次离开的成员总是队头上的成员（不允许中途离队）。而栈为后进先出（LIFO），即每次删除（退栈）的总是当前栈中“最新的”元素，即最后插入（进栈）的元素，而最先插入的元素是被放在栈的底部，要到最后才被删除。

面试例题 23：简答题——队列和栈的使用。

考点：队列和栈的使用。

出现频率：★★★★★

顺序为 1，2，3，4，5，6 的栈，依次进入一个栈，然后再进入队列，执行顺序是什么？

解析

首先顺序为 1，2，3，4，5，6 的栈，即其进栈的顺序是 1，2，3，4，5，6，按照栈的结构，1 最先进栈，被放入栈底，6 最后进栈位于栈顶。

然后进入一个栈，因为只能在栈顶进行退栈操作，也就是说 6 最先退栈，1 最后退栈。因此栈的入队顺序（也就是栈的退栈顺序）为 6，5，4，3，2，1。

最后再进队列，队列是个 FIFO（先进先出）的结构，因此出队顺序与入队的顺序相同，即 6，5，4，3，2，1。也就是 6 最先队列，1 最后进队列。因此此时 6 位于队首，1 位于队尾。

答案

最后的进入队列顺序为：6，5，4，3，2，1。

面试例题 24：选择题——队列和栈的区别。

考点：队列的和栈的区别。

出现频率：★★★★

下列关于栈和队列描述中哪一个是正确的？

A．栈是先进后出结构，队列是先进先出结构

B．栈是先进先出结构，队列是后进先出结构

C．栈和队列都是先进后出结构

D．栈和队列都是先进先出结构

解析

本题考查的是栈与队列的区别，栈是先进后出（FILO），而队列是先进先出（FIFO）。

答案

A

面试例题 25：使用队列实现栈。

考点：队列的使用以及栈的实现。

出现频率：★★★★

编程实现下面的栈的操作，并根据这个栈完成队列的操作。

```
1    class MyStack
2    {
3        void push(data);
4        void pop(&data);
5        bool isEmpty();
6    };
```

解析

显然这里需要实现栈的 3 种基本操作，即进栈、退栈以及判空。为方便起见，使用单链表结构实现栈并且使用类的形式来定义栈及其节点。首先是节点类和栈类的具体定义，程序代码

如下：

```
1   class MyData
2   {
3   public:
4       MyData() : data(0), next(NULL) {}                //默认构造函数
5       MyData(int value) : data(value), next(NULL) {}   //带参数的构造函数
6       int data;                    //数据域
7       MyData *next;                //下一个节点
8   };
9
10   class MyStack
11  {
12  public:
13      MyStack() : top(NULL) {}   //默认构造函数
14      void push(MyData data);    //进栈
15      void pop(MyData *pData);   //退栈
16      bool IsEmpty();            //是否为空栈
17      MyData *top;               //栈顶
18  };
```

这里 MyData 定义了单链表的节点，其中 data 表示节点的数据域，next 表示指向下一个节点的指针。MyStack 表示栈的定义，其中 private 成员 top 表示栈顶，由于不能直接操作栈底，因此这里没有定义栈底的指针。在 MyStack 的默认构造函数中，把栈顶指针 top 置空，表示此时栈为空栈。

接下来是进栈、退栈以及判空的代码实现，程序代码如下所示：

```
1   //进栈
2   void MyStack::push(MyData data)
3   {
4       MyData *pData = NULL;
5       pData = new MyData(data.data);     //生成新节点
6       pData->next = top;                  //与原来的栈顶节点相连
7           top = pData;                        //栈顶节点为新加入的节点
8   }
9
10  //出栈，返回栈顶节点内容
11  void MyStack::pop(MyData *data)
12  {
13      if (IsEmpty())                      //如果栈为空，则直接返回
14      {
15          return;
16      }
17      data->data = top->data;        //给传出的参数赋值
18      MyData *p = top;                //临时保存原栈顶节点
19      top = top->next;               //移动栈顶，指向下一个节点
20      delete p;                      //释放原栈顶节点内存
```

```
21  }
22
23  //判断栈是否为空栈
24  bool MyStack::IsEmpty()
25  {
26      return (top == NULL);      //如果 top 为空返回 1，否则返回 0
27  }
```

MyStack::push 函数表示进栈操作，它实际上就是在单链表的首部进行插入操作，并且 top 一直指向这个单链表的首部。

MyStack::pop 函数表示退栈操作，它实际上就是在单链表的首部进行删除操作，并且 top 一直指向这个单链表的首部。另外它还把删除节点的数据保存到 data 参数中（代码第 17 行）。

MyStack::IsEmpty 函数非常简单，当空栈时，top 指针为 NULL，而当栈中有节点时，top 指向链表头，因此只需要用 top 与 NULL 进行比较即可。

下面是栈操作的测试代码：

```
1   int main()
{
    MyData data(0);           //定义一个节点
    MyStack s;                //定义一个栈结构
    s.push(MyData(1));        //进栈三次: 1, 2, 3
    s.push(MyData(2));
    s.push(MyData(3));

    s.pop(&data);             //第 1 次退栈
    cout << "pop " << data.data << endl;
    s.pop(&data);             //第 2 次退栈
    cout <<  "pop " << data.data << endl;
    s.pop(&data);             //第 3 次退栈
    cout <<  "pop " << data.data << endl;
    cout << "Empty = " << s.IsEmpty() << endl;      //打印判空

    return 0;
}
```

执行结果为：

```
1   pop 3
2   pop 2
3   pop 1
4   IsEmpty = 1
```

可以看出，进栈的顺序和退栈的顺序相反。

在前面的小节里已经实现了 queue，当时所采用的是 front 和 rear 两个指针分别指向队头和队尾。由于题目限制，本题不能使用这些指针。

如何只使用 stack 实现 queue 呢？由于 stack 是先进后出的（FILO），而 queue 是先进先出的（FIFO）。也就是说 stack 进行了一次反向，进行两次反向就能实现 queue 的功能，所以可以

用两个 stack 实现 queue。

假设两个栈 A 和 B，且都为空。可以认为栈 A 提供入队列的功能，栈 B 提供出队列的功能。下面是入队和出队的具体算法。

```
入队:
入栈 A
出队:
```

- 如果栈 B 不为空，直接弹出栈 B 的数据。
- 如果栈 B 为空，则依次弹出栈 A 的数据，放入栈 B 中，再弹出栈 B 的数据。

于是可以得到下面的 MyQueue 定义及实现：

```
1   class MyQueue
2   {
3   public:
4       void enqueue(MyData data);     //入队
5       void dequeue(MyData &data);    //出队
6       bool IsEmpty();                //是否为空队
7   private:
8       MyStack s1;                    //用于入队
9       MyStack s2;                    //用于出队
10  };
11
12  //入队
13  void MyQueue::enqueue(MyData data)
14  {
15      s1.push(data);                 //只对 s1 进行进栈操作
16  }
17
18  //出队
19  void MyQueue::dequeue(MyData &data)
20  {
21       MyData temp(0);               //局部变量，用于临时存储
22
23       if (s2.IsEmpty())
24       {
25           // 如果 s2 为空，把 s1 的所有元素 push 到 s2 中
26           while(!s1.IsEmpty())
27           {
28               s1.pop(temp);    //弹出 s1 的元素
29               s2.push(temp);   //压入 s2 中
30           }
31       }
32      if (!s2.IsEmpty())
33      {
34           //此时如果 s2 不为空，弹出 s2 的栈顶元素
35           s2.pop(data);
```

```
36         }
37     }
38
39     //队列判空
40     bool MyQueue::IsEmpty()
41     {
42         //如果两个栈都为空，则返回 1，否则返回 0
43         return (s1.IsEmpty() && s2.IsEmpty());
44     }
```

测试 MyQueue 的程序如下：

```
1   int main()
2   {
3       /*测试队列*/
4       MyData data(0);         //定义一个节点
5       MyQueue q;
6
7       q.enqueue(MyData(1));
8       q.enqueue(MyData(2));
9       q.enqueue(MyData(3));
10
11      q.dequeue(data);
12      cout << "dequeue " << data.data << endl;
13      q.dequeue(data);
14      cout << "dequeue " << data.data << endl;
15      q.dequeue(data);
16      cout << "dequeue " << data.data << endl;
17      cout << "IsEmpty: " << q.IsEmpty() << endl;
18
19      return 0;
20  }
```

执行结果：

```
dequeue 1
dequeue 2
dequeue 3
IsEmpty: 1
```

通过结果说明：入队顺序与出队顺序相同。

面试例题 26：选择题——栈的使用。

考点：队列和栈的区别。

出现频率：★★★★

设栈最大长度为 3，入栈序列为 1，2，3，4，5，6，则不可能得出栈序列是：

A. 1，2，3，4，5，6

B. 2，1，3，4，5，6

C. 3，4，2，1，5，6

D. 4，3，2，1，5，6

解析

此题包含一个前提，就是任何时候都能进栈和退栈。

下面具体分析每一个选项。

- 选项 A，顺序为 1，2，3，4，5，6。很明显，每次有一个元素进栈，然后马上退栈，即：1 进栈，1 退栈，2 进栈，2 退栈，如此进行就可以遵循这个顺序。
- 选项 B，顺序为 2，1，3，4，5，6。先入栈 1，2，然后全部退栈，即以 2，1 顺序退栈。接着后面的元素和选项 A 的方式一样有一个元素进栈，然后马上退栈，这样也可以遵循这个顺序。
- 选项 C，顺序为 3，4，2，1，5，6。先入栈 1，2，3，然后做一次退栈，即 3 退栈。接着 4 进栈，并且马上全部退栈，即 4，2，1 退栈。后面的 5，6 的入栈退栈方式和选项 A 的一样，这样也可以遵循这个顺序。
- 选项 D，顺序为 4，3，2，1，5，6。注意这里前面 4 个元素是 4，3，2，1，如果栈的最大长度为 4，是可以实现按这个顺序退栈的，然而栈的最大长度为 3，即元素 4 无法和 1，2，3 同时存在栈内。所以无法按 4，3，2，1 顺序退栈。

答案

D

11.7 二 叉 树

二叉树是树形结构的一个重要类型。许多实际问题抽象出来的数据结构都是二叉树的形式，而且二叉树的存储结构及其算法都较为简单，因此二叉树使用极为广泛。

二叉树（BinaryTree）是 n（n≥0）个节点的有限集合，它可以是空集（n=0），也可以是由一个根节点及两棵互不相交、分别称作这个根的左子树和右子树的二叉树组成。

面试例题 27：用 C++实现一个二叉排序树，完成创建节点、插入节点、删除节点、查找节点等功能。

考点：二叉排序树的各项基本操作。

出现频率：★★★★

解析

二叉排序树（Binary Sort Tree）又称二叉查找树（Binary Search Tree）。其定义为：二叉排序树是空树，或者是满足如下性质的二叉树。

- 若它的左子树非空，则左子树上所有节点的值均小于根节点的值。
- 若它的右子树非空，则右子树上所有节点的值均大于根节点的值。
- 左、右子树本身又各是一棵二叉排序树。

一个简单的二叉排序树如图 11.6 所示。

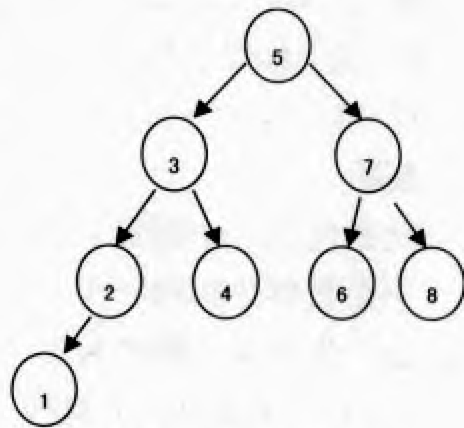


图 11.6 二叉排序树模型

对于二叉排序树模型首先需要定义节点类以及二叉排序树类。对于节点类来说，每个节点有向下的两个指针，而且每个节点都只有一个父节点；对于二叉排序树类来说，只需要有数的根节点，就可以进行操作了。因此它们的数据结构如下：

```
1   //节点类定义
2   class Node
3   {
4   public:
5       int  data;          //数据
6       Node *parent;       //父节点
7       Node *left;         //左子节点
8       Node *right;        //右子节点
9   public:
10      Node() : data(-1), parent(NULL), left(NULL), right(NULL) { };
11      Node(int num) : data(num), parent(NULL), left(NULL), right(NULL) { };
12  };
13
14  //二叉排序树类定义
15  class Tree
16  {
17  public:
18      Tree(int num[], int len);           //插入 num 数组的前 len 个数据
19      void insertNode1(int data);         //插入节点，递归方法
20      void insertNode(int data);          //插入节点，非递归方法
21      Node *searchNode(int data);         //查找节点
22      void deleteNode(int data);          //删除节点及其子树
23  private:
24      void insertNode(Node* current,int data);        //递归插入方法
25      Node *searchNode(Node* current,int data);       //递归查找方法
26      void deleteNode(Node* current);                 //递归删除方法
27  private:
28      Node* root;                                     //二叉排序树的根节点
29  };
```

代码第 18 行的构造函数定义了根据 int 数组创建二叉排序树的方法。

代码第 19～10 行分别定义两种插入节点的方法，分别为递归和非递归方法。

代码第 21～22 行分别定义了查找和删除节点的方法，这两个方法均为递归方法。

另外注意，代码第 24～26 行中有 3 个定义为 private 的方法。这是出于封装性的考虑，指针 root 定义为私有成员变量，于是 3 个使用递归方法的函数（代码第 20～第 22 行）都只能有一个参数 data，然而只含有一个参数 data 的函数是无法进行递归运算的，所以它们内部直接调用了这 3 个 private 的对应方法，而这 3 个 private 方法直接使用递归。

接下来具体说明各个方法的实现。

构造函数中创建二叉排序树的方法。这里首先生成根节点，然后循环调用插入节点对二叉树进行插入操作。

```
1   //插入 num 数组的前 len 个数据
2   Tree::Tree(int num[], int len)
3   {
4       root = new Node(num[0]); //建立 root 节点
5       //把数组中的其他数组插入到二叉排序树中
6       for(int i=1; i < len; i++)
7       {
8           //insertNode(num[i]);
9           insertNode1(num[i]);
10      }
11  };
```

插入节点操作。通常在二叉树的遍历中可以选择递归与非递归的方法。由于篇幅有限，本题仅给出插入时的这两种方法。

insertNode1()使用非递归方法插入节点，其代码如下：

```
1   //插入数据为参数 data 的节点,非递归方法
2   void Tree::insertNode1(int data) //插入节点
3   {
4       Node *p, *par;
5       Node *newNode = new Node(data);     //创建节点
6
7       p = par = root;
8       while(p != NULL)                    //查找插入在哪个节点下面
9       {
10          par = p;  //保存节点
11          if (data > p->data)             //如果 data 大于当前节点的 data
12              p = p->right;               //下一步到左子节点,否则进行到右子节点
13          else if(data < p->data)
14              p = p->left;
15          else if (data == p->data)       //不能插入重复数据
16          {
17              delete newNode;
18              return;
19          }
20      }
21      newNode->parent = par;
22      if (par->data > newNode->data)      //把新节点插入在目标节点的正确位置
```

```
23              par->left = newNode;
24              else
25              par->right = newNode;
26  }
```

其操作下面的步骤如下所示。

（1）创建节点（代码第 5 行）。

（2）查找新建节点的插入位置。

根据前面介绍过的二叉排序树的性质，可以用节点的比较（代码第 11～第 14 行）进行节点的遍历，如果二叉树中已经含有相同数据的节点，则不予插入（代码第 15～第 19 行）且直接返回。循环完毕后，p 为 NULL，par 指向目标节点。

（3）把新节点插入在目标节点的正确位置。同样需要考虑到二叉树排序的性质。

insertNode()使用非递归的方法插入节点，它的内部调用了 private 成员函数 insertNode()，代码如下：

```
1   //插入数据为参数 data 的节点,调用递归插入方法
2   void Tree::insertNode(int data)
3   {
4       if(root != NULL)
5       {
6           insertNode(root, data);          //调用递归插入方法
7       }
8   }
9
10  //递归插入方法
11  void Tree::insertNode(Node* current,int data)
12  {
13      //如果 data 小于当前节点数据，在当前节点的左子树插入
14      if(data < current->data)
15      {
16          if(current->left == NULL)        //如果左节点不存在，则插入到左节点
17          {
18              current->left = new Node(data);
19              current->left->parent = current;
20          }
21          else
22              insertNode(current->left,data);    //对左节点进行递归调用
23      }
24
25      //如果 data 大于当前节点数据，在当前节点的右子树插入
26      else if(data > current->data)
27      {
28          if(current->right == NULL)            //如果右节点不存在，则插入到右节点
29          {
30              current->right = new Node(data);
```

```
31                 current->right->parent = current;
32             }
33             else
34                 insertNode(current->right,data);  //对右节点进行递归调用
35         }
36 
37         return;                              //data 等于当前节点数据时，不插入
38     };
```

现在说明 private 成员函数 insertNode 的步骤。

（1）如果当前节点的数据值小于 data，则应该在它的左子树插入。如果当前节点没有左子节点，则直接插入新节点作为它的左子节点。否则把它的左子节点作为参数，再进行整个过程。

（2）如果当前节点的数据值大于 data，则应该在它的右子树插入。此时如果当前节点没有右子节点，则直接插入新节点作为它的右子节点。否则把它的右子节点作为参数，再进行整个过程。

比较这两个方法的插入函数可以看出，使用递归方法写出的程序简单容易理解，而非递归方法，与递归方法的程序结构相比显得臃肿。递归时需要不断的进行函数入栈、退栈操作，因此效率比较低，并且如果递归的深度较大，可能会引发栈溢出。

查找节点也使用了递归，由于与插入节点类似，这里只列出 private 方法：

```
1   //递归查找方法
2   Node* Tree::searchNode(Node* current,int data)
3   {
4       //如果 data 小于当前节点数据,递归搜索其左子树
5       if(data < current->data)
6       {
7           if (current->left == NULL)       //如果不存在左子树，返回 NULL
8               return NULL;
9           return searchNode(current->left, data);
10      }
11 
12      //如果 data 大于当前节点数据,递归搜索其左子树
13      else if(data > current->data)
14      {
15          if (current->right == NULL)      //如果不存在右子树，返回 NULL
16              return NULL;
17          return searchNode(current->right, data);
18      }
19 
20 
21      return current; //如果相等返回 current
22  }
```

searchNode 的步骤如下。

（1）如果 data 小于当前节点（current）的值，且 current 的左子树存在，则继续搜索 current 的左子树，否则返回 NULL。

（2）如果 data 大于当前节点（current）的值，且 current 的右子树存在，则继续搜索 current 的左子树，否则返回 NULL。

（3）如果 data 等于当前节点（current）的值，返回 current。

最后是删除节点的操作，代码如下：

```
1    //删除数据为 data 的节点及其子树
2    void Tree::deleteNode(int data)
3    {
4        Node *current = NULL;
5
6        current = searchNode(data); //查找节点
7        if (current != NULL)
8        {
9            deleteNode(current); //删除节点及其子树
10       }
11   }
12
13   //删除 current 节点及其子树的所有节点
14   void Tree::deleteNode(Node *current)
15   {
16       if (current->left != NULL)   //删左子树
17           deleteNode(current->left);
18       if (current->right != NULL)   //删右子树
19           deleteNode(current->right);
20
21       if (current->parent == NULL)
22       {
23           //如果 current 是根节点，把 root 置空
24           delete current;
25           root = NULL;
26           return;
27       }
28
29       //将 current 父亲节点的相应指针置空
30       if (current->parent->data > current->data) //current 为其父节点的左子节点
31           current->parent->left = NULL;
32       else  //current 为 parNode 的右子节点
33           current->parent->right = NULL;
34
35       //最后删此节点
36       delete current;
37   }
```

public 成员函数 deleteNode()删除数据为 data 的节点及其子树，它首先调用了 searchNode()查找数据等于 data 的节点（代码第 6 行），如果找到节点，则调用 private 成员函数 deleteNode 函数节点及其子树。

private 成员函数也是使用递归方法进行删除操作的。它的步骤如下。

（1）如果 current 左子树存在，则递归删除 current 左子树。

（2）如果 current 右子树存在，则递归删除 current 右子树。

（3）最后删除 current 节点。此时如果 current 是根节点，需要把 root 置空，否则把其父节点相应的指针置空。

面试例题 28：使用递归与非递归方法实现中序遍历。

考点：中序遍历算法的实现。

出现频率：★★★

解析

中序遍历的递归算法定义为，若二叉树非空，则依次执行如下操作。

- 遍历左子树。
- 访问根节点。
- 遍历右子树。

中序遍历的递归算法程序代码如下：

```
1   //中序遍历
2   void Tree::InOrderTree()
3   {
4       if(root == NULL)
5           return;
6       InOrderTree(root);
7   }
8
9   void Tree::InOrderTree(Node* current)
10  {
11      if(current!= NULL)
12      {
13          InOrderTree(current->left);      //遍历左子树
14          cout << current->data << " ";    //打印节点数据
15          InOrderTree(current->right);     //遍历右子树
16      }
17  }
```

对于中序遍历非递归算法，可以采用栈（stack）来临时存储节点。方法如下。

- 先将根节点入栈，遍历左子树。
- 遍历完左子树返回时，栈顶元素应为根节点，此时出栈，并打印节点数据。
- 再中序遍历右子树。

代码如下：

```
1   void Tree::InOrderTreeUnRec()
2   {
3       stack<Node *> s;
```

```
4          Node *p = root;
5          while(p != NULL || !s.empty())
6          {
7              while(p != NULL)            //遍历左子树
8              {
9                  s.push(p);             //把遍历的节点全部压栈
10                 p = p->left;
11             }
12
13             if (!s.empty())
14             {
15                 p = s.top();           //得到栈顶内容
16                 s.pop();               //出栈
17                 cout << p->data << " "; //打印
18                 p = p->right;          //指向右子节点，下一次循环时就会中序遍历右子树
19             }
20         }
21     }
```

main 函数测试如下：

```
1     int main(void)
2     {
3         int num[] = {5, 3 ,7, 2, 4, 6, 8, 1};
4         Tree tree(num, 8);
5         cout << "InOrder: ";
6         tree.InOrderTree();              //中序遍历，递归方法
7         cout << "\nInOrder: ";
8         tree.InOrderTreeUnRec();         //中序遍历，非递归方法
9         return 0;
10    }
```

测试结果为：

```
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
```

另外，从这个结果可以看出，中序遍历的结果就是二叉排序树的排序结果。

面试例题 29：使用递归与非递归方法实现先序遍历。

考点：先序遍历算法的实现。

出现频率：★★★

解析

若二叉树非空，则先序遍历的递归算法，依次执行如下操作。

- 访问根节点。
- 遍历左子树。
- 遍历右子树。

先序遍历的程序代码如下：

```
1   //先序遍历
2   void Tree::PreOrderTree()
3   {
4       if(root == NULL)
5           return;
6       PreOrderTree(root);
7   };
8
9   void Tree::PreOrderTree(Node* current)
10  {
11      if(current!= NULL)
12      {
13          cout << current->data << " ";     //打印节点数据
14          PreOrderTree(current->left);     //遍历左子树
15          PreOrderTree(current->right);    //遍历右子树
16      }
}
```

对于先序遍历的非递归算法，使用栈（stack）来临时存储节点，方法如下。

- 打印根节点数据。
- 把根节点的 right 入栈，遍历左子树。
- 遍历完左子树返回时，栈顶元素应为 right，出栈，遍历以该指针为根的子树。

```
1   void Tree::PreOrderTreeUnRec()
2   {
3       stack<Node *> s;
4       Node *p = root;
5       while(p != NULL || !s.empty())
6       {
7           while(p != NULL)                  //遍历左子树
8           {
9               cout << p->data << " ";       //打印
10              s.push(p);                    //把遍历的节点全部压栈
11              p = p->left;
12          }
13
14          if (!s.empty())
15          {
16              p = s.top();                  //得到栈顶内容
17              s.pop();                      //出栈
18              p = p->right;                 //指向右子节点，下一次循环时就会先序遍历左子树
19          }
20      }
21  }
```

main 函数测试如下：

```
1     int main(void)
2     {
3         int num[] = {5, 3 ,7, 2, 4, 6, 8, 1};
4         Tree tree(num, 8);
5         cout << "PreOrder: ";
6         tree.PreOrderTree();          //先序遍历，递归方法
7         cout << "\nPreOrder: ";
8         tree.PreOrderTreeUnRec();  //先序遍历，非递归方法
9         return 0;
10    }
```

测试结果为：

```
5 3 2 1 4 7 6 8
5 3 2 1 4 7 6 8
```

面试例题 30：使用递归与非递归方法实现后序遍历。

考点：后序遍历算法的实现。

出现频率：★★★

解析

若二叉树非空，则后序遍历得递归算法依次执行如下操作。

- ❑ 遍历左子树。
- ❑ 遍历右子树。
- ❑ 访问根节点。

后序遍历的程序代码如下：

```
1     //后序遍历
2     void Tree::PostOrderTree()
3     {
4         if(root == NULL)
5             return;
6         PostOrderTree(root);
7     }
8
9     void Tree::PostOrderTree(Node* current)
10    {
11        if(current!= NULL)
12        {
13            PostOrderTree(current->left);      //遍历左子树
14            PostOrderTree(current->right);     //遍历右子树
15            cout << current->data << " ";      //打印节点数据
16        }
17    }
```

现在说明后序遍历的非递归方法。假设 T 是要遍历树的根指针，后序遍历要求在遍历完左右子树后，再访问根节点。需要判断根节点的左右子树是否均遍历过。

可采用标记法，节点入栈时，配一个标志 tag 一同入栈（tag 为 0 表示遍历左子树前的现场保护，tag 为 1 表示遍历右子树前的现场保护）。

首先将 T 和 tag（为 0）入栈，遍历左子树；返回后，修改栈顶 tag 为 1，遍历右子树；最后访问根节点程序。代码如下：

```
1   void Tree::PostOrderTreeUnRec()
2   {
3       stack<Node *> s;
4       Node *p = root;
5
6       while(p != NULL || !s.empty())
7       {
8           while(p != NULL)
9           {
10              s.push(p);                  //压栈
11              p = p->left;                //遍历左子树
12          }
13
14          if (!s.empty())
15          {
16              p = s.top();                //得到栈顶元素
17              if (p->tag)                 //tag 为 1 时
18              {
19                  cout << p->data << " ";  //打印节点数据
20                  s.pop();            //退栈
21                  p = NULL;           //第二次访问标志其右子树已经遍历
22              }
23              else
24              {
25                  p->tag = 1;         //修改 tag 为 1
26                  p = p->right;       //指向右节点，下次遍历其左子树
27              }
28          }
29      }
30  }
```

根据上面的分析，代码中 Node 类添加了 int 型的成员变量 tag。main 函数测试如下：

```
1   int main(void)
2   {
3       int num[] = {5, 3 ,7, 2, 4, 6, 8, 1};
4       Tree tree(num, 8);
5       cout << "PostOrder: ";
6       tree.PostOrderTree();            //后序遍历，递归方法
7       cout << "\nPostOrder: ";
8       tree.PostOrderTreeUnRec();    //后序遍历，非递归方法
9       return 0;
10  }
```

测试结果为：

```
12436875
12436875
```

面试例题 31：编写层次遍历二叉树的算法。

考点：层次遍历算法的实现。

出现频率：★★

解析

层次遍历就是一层一层地进行遍历。例如图 11.6 共有 4 层，各层分别如下所示。

第 1 层：5。

第 2 层：3、7。

第 3 层：2、4、6、8。

第 4 层：1。

本题很难直接用节点的指针（left、right、parent）来实现，但是借助于队列就可以轻松地实现。例如图 11.7 所示的二叉树结构，可以按照下面的方式执行。

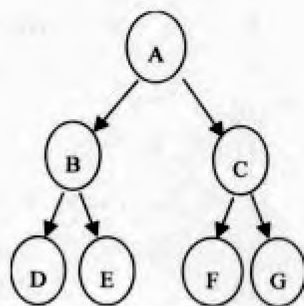


图 11.7 二叉树图

- A 入队。
- A 出队，同时 A 的子节点 B、C 入队（此时队列有：B、C)。
- B 出队，同时 B 的子节点 D、E 入队（此时队列有：C、D、E)。
- C 出队，同时 C 的子节点 F、G 入队（此时队列有：D、E、F、G)。

显然这样就能使出队的顺序满足层序遍历。

为了方便，可以使用 C++标准库中的 queue 模板类。代码如下：

```
1    //层次遍历
2    void Tree::LevelOrderTree()
3    {
4        queue<Node *> q; //定义队列 q，它的元素为 Node *类型指针
5        Node *ptr = NULL;
6
7        q.push(root);    //根节点入队
8        while(!q.empty())
9        {
10           ptr = q.front();              //得到队头节点
11           q.pop();                      //出队
12           cout << ptr->data << " ";     //打印当前节点数据
13           if (ptr->left != NULL)        //当前节点存在左节点，则左节点入队
14           {
15               q.push(ptr->left);
16           }
17           if (ptr->right != NULL)       //当前节点存在右节点，则右节点入队
18           {
```

```
19                  q.push(ptr->right);
20              }
21          }
22      }
```

程序中对队列 q 进行入队和出队的操作。每次出队时，就打印此出队的元素，并且对这个元素节点的左子节点和右子节点进行入队操作。

测试代码如下：

```
1   int main(void)
2   {
3       int num[] = {5, 3 ,7, 2, 4, 6, 8, 1};
4       Tree tree(num, 8);
5       cout << "InOrder: ";
6       tree.LevelOrderTree();          //层序遍历
7       return 0;
8   }
```

执行结果如下：

```
5 3 7 2 4 6 8 1
```

面试例题 32：编写判断给定二叉树是否为二叉排序树的程序。

考点：二叉排序树判定算法的实现。

出现频率：★★★

解析

前面例题中已经编写过中序遍历的算法，使用中序遍历的结果就是二叉排序树的排序输出，因此我们可以使用中序遍历来实现判定二叉树是否为二叉排序树。程序代码如下：

```
1   //使用中序遍历判断二叉树是否为二叉排序树
2   bool IsSortedTree(Tree tree)
3   {
4       int lastvalue = 0;
5       stack<Node *> s;
6       Node *p = tree.root;
7       while(p != NULL || !s.empty())
8       {
9           while(p != NULL)        //遍历左子树
10          {
11              s.push(p);      //把遍历的节点全部压栈
12              p = p->left;
13          }
14
15          if (!s.empty())
16          {
17              p = s.top();    //得到栈顶内容
18              s.pop();        //出栈
19              if (lastvalue == 0 || lastvalue < p->data)
```

```
20                {
21                    //如果是第一次弹出或者 lastvalue 小于当前节点值，给 lastvalue 赋值
22                    lastvalue = p->data;
23                }
24                else if (lastvalue >= p->data)
25                {
26                    //如果 lastvalue 大于当前节点值,返回 false
27                    return false;
28                }
29
30                p = p->right;  //指向右子节点，下一次循环时就会中序遍历右子树
31            }
32        }
33
34        return true; //到这里说明节点数据是升序排列，返回 true
35    }
```

这里使用了非递归的二叉树遍历方式，其中使用了一个 lastvalue 来记录上一次出队的节点数据，如果出现 lastvalue 大于或等于当前出队的节点值，则返回 false，否则一直入队和出队。如果 lastvalue 始终小于当前出队的节点值，则是升序排列，返回 true。

下面是主函数的测试：

```
1   int main(void)
2   {
3       int num[] = {5, 3 ,7, 2, 4, 6, 8, 1};
4       Tree tree(num, 8);
5       cout << "InOrder: ";
6       tree.InOrderTreeUnRec();          //中序遍历，非递归方法
7       cout << "\nIsSortedTree: " << IsSortedTree(tree) << endl;
8       Node *node = tree.searchNode(4);
9       node->data = 1;                    //手动把节点的数据改成 1
10      cout << "InOrder: ";
11      tree.InOrderTreeUnRec();          //中序遍历，非递归方法
12      cout << "\nIsSortedTree: " << IsSortedTree(tree) << endl;
13      return 0;
14  }
```

测试结果：

```
1 2 3 4 5 6 7 8
IsSortedTree: 1
1 2 3 1 5 6 7 8
IsSortedTree: 0
```

第12章 排序

所谓排序，就是要整理数组中的记录，使之按关键字递增(或递减)次序排列起来。其准确定义如下所示。

输入：n 个记录 R_1，R_2，…，R_n，其相应的关键字分别为 K_1，K_2，…，K_n。

输出：R_{i1}，R_{i2}，…，R_{in}，使得 $K_{i1} \leqslant K_{i2} \leqslant \ldots \leqslant K_{in}$（或 $K_{i1} \geqslant K_{i2} \geqslant \ldots \geqslant K_{in}$）。

排序可以分为下面 5 类。

- 插入排序。
- 选择排序。
- 交换排序。
- 归并排序。
- 分配排序。

12.1 插入排序

插入排序（Insertion Sort）的基本思想是：每次将一个待排序的记录，按其关键字大小插入到前面已经排好序的子数组中的适当位置，直到全部记录插入完成为止。

直接插入排序和希尔（Shell）排序都属于插入排序方法。

12.1.1 直接插入排序

直接插入排序是稳定的排序方法。直接插入排序的基本思想是假设待排序的记录存放在数组 R[1…*n*]中。初始时，R[1]自成 1 个有序区，无序区为 R[2…*n*]。从 $i=2$ 开始到 $i=n$ 结束，依次将 R[*i*]插入当前的有序区 R[1...*i*-1]中，生成含 n 个记录的有序区。

第（*i*-1）次直接插入排序：

通常将记录 R[*i*]（$i=2$，…，n-1）插入到当前的有序区，使得插入后仍保证该区间里的记录是按关键字有序地操作，称第 *i*-1 次直接插入排序。

排序过程的某一中间时刻，R 被划分成两个子区间 R[1…*i*-1]（已排好序的有序区）和 R[*i*…*n*]

（当前未排序的部分，可称无序区）。

直接插入排序的基本操作是将当前无序区的第1个记录R[*i*]插入到有序区R[1…*i*-1]中适当的位置上，使R[1…*i*]变为新的有序区。这种方法每次使有序区增加1个记录，通常称增量法。

插入排序与打扑克时整理手上的牌非常类似。摸来的第1张牌无须整理，此后每次摸1张插入左手的牌（有序区）中正确的位置上。为了找到这个正确的位置，需自左向右（或自右向左）将摸来的牌与左手中的牌逐一比较。

由直接插入排序的基本思想很容易得到下面的方法。

- 在当前有序区R[1…*i*-1]中查找R[*i*]的正确插入位置k（$1 \leqslant k \leqslant i-1$）。
- 将R[*k*…*i*-1]中的记录均后移一位，腾出k位置上的空间插入R[*i*]。

这里采用升序排序，也就是说如果R[*i*]的关键字大于等于R[1…*i*-1]中所有记录的关键字，则R[*i*]就是插入的位置。

还有一种改进的方法，即查找比较操作和记录移动操作交替地进行。将待插入记录R[*i*]的关键字从右向左依次与有序区中记录R[*j*]（$j = i-1, i-2, \dots, 1$）的关键字进行比较。

- 如果R[*j*]的关键字大于R[*i*]的关键字，则将R[*j*]后移一位；
- 如果R[*j*]的关键字小于或等于R[*i*]的关键字，则查找过程结束，（$j+1$）即为R[*i*]的插入位置。

关键字比R[*i*]的关键字大的记录均已后移，所以（$j+1$）的位置腾空，只要将R[*i*]直接插入此位置即可完成一次直接插入排序。

面试例题1：编程实现直接插入排序。

考点：直接插入排序算法的实现。

出现频率：★★★★

解析

使用上面介绍的改进方法，即查找比较操作和记录移动操作交替地进行。代码如下所示：

```
1   #include <iostream>
2   using namespace std;
3
4   //直接插入排序
5   void insert_sort(int a[], int n)
6   {
7       int i, j, temp;
8
9       for (i=1; i<n; i++) //需要选择 n-1 次
10      {
11          //暂存下标为 i 的数。下标从 1 开始，因为开始时
12          //下标为 0 的数，前面没有任何数，此时认为它是排好顺序的
13          temp = a[i];
14          for (j=i-1; j>=0 && temp<a[j]; j--)
```

```
15              {
16                  //如果满足条件就往后挪。最坏的情况就是 temp 比 a[0]小，它要放在最前面
17                  a[j+1] = a[j];
18              }
19
20              a[j+1] = temp;            //找到下标为 i 的数的放置位置
21          }
22  }
23
24  void print_array(int a[], int len)
25  {
26      for(int i = 0; i < len; i++)    //循环打印数组的每个元素
27          {
28              cout << a[i] << " ";
29      }
30          cout << endl;
31  }
32
33  int main()
34  {
35      int a[] = {7, 3, 5, 8, 9, 1, 2, 4, 6};
36          cout << "before insert sort: ";
37          print_array(a, 9);
38          insert_sort(a, 9);          //进行直接插入排序
39      cout << "after insert sort: "
40        print_array(a, 9);
41          return 0;
42  }
```

insert_sort 函数的插入次数是（len-1），因为当数组只有 a[0]时，认为 a[0]就是已经排好序了。局部变量 *i* 用于表示对哪一个元素进行插入操作，*j* 表示插入到哪个目标元素的后面，temp 保存需要插入的元素。这里最坏的情况就是 temp 比 a[0]都小，此时 *j* 为－1，需要把 temp 作为新的 a[0]。

测试结果如下：

```
before insert sort: 7 3 5 8 9 1 2 4 6
before insert sort: 1 2 3 4 5 6 7 8 9
```

12.1.2 希尔（Shell）排序

希尔（Shell）排序是 D.L.shell 于 1959 年提出的，它属于插入排序方法，是不稳定的排序方法。

在直接插入排序算法中，每次插入一个数，使有序序列只增加 1 个节点，并且对插入下一个数没有提供任何帮助。如果比较相隔较远距离（称为增量）的数，使得数移动时能跨过多个元素，则进行一次比较就可能消除多个元素交换。

希尔（Shell）排序算法先将要排序的一组数按某个增量 d 分成若干组，对每组中全部元素

进行排序，然后用一个较小的增量对它进行再次分组，并对每个新组进行排序。当增量减到1时，整个要排序的数被分成一组，排序完成。因此希尔排序实质上是一种分组插入方法。

希尔排序的时间性能优于直接插入排序，其原因如下。

- 当数组初始状态基本有序时，直接插入排序所需的比较和移动次数均较少。
- 当 *n* 值较小时，*n* 和 *n*2 的差别也较小，即直接插入排序的最好时间复杂度 O（*n*）和最坏时间复杂度 O（*n*2）差别不大。
- 在希尔排序开始时增量较大，分组较多，每组的记录数目少，故各组内直接插入较快，后来增量 d 逐渐缩小，分组数逐渐减少，而各组的记录数目逐渐增多，但由于已经按 d-1 作为距离完成排序，使数组较接近于有序状态，所以新的一次排序过程也较快。

因此，希尔排序在效率上比直接插入排序有较大的改进。

注意：由于分组的存在，相等的元素可能会分在不同组，导致它们的次序可能发生变化，因此希尔排序是不稳定的。

面试例题 2：编程实现直接 Shell 排序。

考点：Shell 排序算法的实现。

出现频率：★★★★

解析

可以这样来设置增量。初始时取序列的一半为增量，以后每次减半，直到增量为1。

代码如下所示：

```
1    #include <iostream>
2    using namespace std;
3
4    void shell_sort(int a[], int len)
5    {
6        int h, i, j, temp;
7
8        for (h=len/2; h>0; h=h/2)      //控制增量
9        {
10           for (i=h; i<len; i++)       //这个 for 循环就是前面的直接插入排序
11           {
12               temp = a[i];
13               for (j=i-h; (j>=0 && temp<a[j]); j-=h)
14               {
15                   a[j+h] = a[j];
16               }
17               a[j+h] = temp;
18           }
19       }
20   }
21
```

```
22  void print_array(int a[], int len)
23  {
24      for(int i = 0; i < len; i++)     //循环打印数组的每个元素
25      {
26          cout << a[i] << " ";
27      }
28      cout << endl;
29  }
30
31  int main()
32  {
33      int a[] = {7, 3, 5, 8, 9, 1, 2, 4, 6};
34      cout << "before shell sort: ";
35      print_array(a, 9);
36      shell_sort(a, 9);            //进行 Shell 排序
37      cout << "after shell sort: ";
38      print_array(a, 9);
39      return 0;
40  }
```

shell_sort 函数使用了循环设置增量（代码第 8 行），里面又嵌套了一个直接插入排序的算法。注意这个嵌套的算法代码实现，与上个例题中的代码相比只有一点不同，就是现在的增量是 h，而原来的增量是 1。

测试结果如下：

```
before shell sort: 7 3 5 8 9 1 2 4 6
before shell sort: 1 2 3 4 5 6 7 8 9
```

12.2 交换排序

交换排序的基本思想是：两两比较待排序记录的关键字，发现两个记录的次序相反时即进行交换，直到没有反序的记录为止。

应用交换排序基本思想的主要排序方法有冒泡排序和快速排序。

12.2.1 冒泡排序

冒泡排序的方法是：将被排序的记录数组 A[1…*n*]垂直排列，每个记录 A[*i*]看做是重量为 A[*i*]的气泡。根据轻气泡不能在重气泡之下的原则，从下往上扫描数组 A，凡扫描到违反本原则的轻气泡，就使其向上“飘浮”。如此反复进行，直到最后任意两个气泡都是轻者在上，重者在下为止。

冒泡排序是稳定的排序。下面是具体的算法。

（1）初始状态下，A[1…*n*]为无序区。

（2）第1次扫描。

从无序区底部向上依次比较相邻的两个气泡的重量，若发现轻者在下、重者在上，则交换二者的位置。即依次比较（A[*n*]，A[*n*-1]），（A[*n*-1]，A[*n*-2]），…，（A[2]，A[1]）。对于每对气泡（A[*j*+1]，A[*j*]），若A[*j*+1]<A[*j*]，则交换A[*j*+1]和A[*j*]的内容。

第1次扫描完毕时，“最轻”的气泡就飘浮到该区间的顶部，即关键字最小的记录被放在最高位置A[1]上。

（3）第2次扫描。

扫描A[2…*n*]。扫描完毕时，“次轻”的气泡飘浮到A[2]的位置上。

（4）第*i*次扫描。

A[1…*i*-1]和A[*i*…*n*]分别为当前的有序区和无序区。扫描仍是从无序区底部向上直至该区顶部。扫描完毕时，该区中最轻气泡飘浮到顶部位置A[*i*]上，结果是A[1…*i*]变为新的有序区。

最后，经过（*n*-1）次扫描可得到有序区A[1…*n*]。

面试例题3：编程实现直接冒泡排序。

考点：冒泡排序算法的实现。

出现频率：★★★★

解析

根据前面冒泡扫描的方法，可以写出下面的排序代码。

```
1   void bubble_sort_1(int a[], int len)
2   {
3       int i = 0;
4       int j = 0;
5       int temp = 0; //用于交换
6
7       for(i=0; i<len-1; i++)          //进行（n-1）次扫描
8       {
9           for(j=len-1; j>=i; j--)   //从后往前交换，这样最小值冒泡到开头部分
10          {
11              if(a[j+1] < a[j])   //如果 a[j]小于 a[j-1],交换两元素值
12              {
13                  temp = a[j];
14                  a[j] = a[j+1];
15                  a[j+1] = temp;
16              }
17          }
18      }
19  }
```

这个代码存在一个问题，就是假如进行第 i 次扫描前，数组已经是排好序了，它还会进行下一次的扫描，显然以后的扫描都是没有必要的。

可以对上面的这个代码进行一点改进，代码如下所示：

```
1    void bubble_sort_2(int a[], int len)
2    {
3        int i = 0;
4        int j = 0;
5        int temp = 0;              //用于交换
6        int exchange = 0;          //用于记录每次扫描时是否发生交换
7
8        for(i=0; i<len-1; i++)     //进行（n-1）次扫描
9        {
10           exchange = 0;          //每次扫描之前对 exchange 置 0
11           for(j=len-1; j>=i; j--)        //从后往前交换，这样最小值冒泡到开头部分
12           {
13               if(a[j+1] < a[j])          //如果 a[j]小于 a[j-1],交换两元素值
14               {
15                   temp = a[j];
16                   a[j] = a[j+1];
17                   a[j+1] = temp;
18                   exchange = 1;  //发生交换，exchange 置 1
19               }
20           }
21           if (exchange != 1)             //此趟扫描没有发生过交换，说明已经是排序的
22               return;                    //不需要进行下次扫描
23       }
24   }
```

这里使用一个局部变量exchange来记录在本次扫描时有没有进行数据交换。每次扫描之前，把 exchange 置 0（代码第 10 行），如果扫描时发生数据交换，则把 exchange 置 1（代码第 18 行），如果没有则说明数组已经是排序了，不需要进行下一次扫描（代码第 22 行）。

对两种冒泡排序的测试 main 函数如下所示：

```
1    int main()
2    {
3        int a[] = {7, 3, 5, 8, 9, 1, 2, 4, 6};
4        cout << "before bubble sort: ";
5        print_array(a, 9);
6        //bubble_sort_1(a, 9);       //冒泡排序
7        bubble_sort_2(a, 9);         //改进的冒泡排序
8        cout << "after bubble sort: ";
9        print_array(a, 9);
10        return 0;
11   }
```

测试结果如下：

```
before bubble sort: 7 3 5 8 9 1 2 4 6
before bubble sort: 1 2 3 4 5 6 7 8 9
```

12.2.2 快速排序

快速排序是C.R.A.Hoare于1962年提出的一种划分交换排序，它采用了分治的策略，通常称其为分治法（Divide-and-Conquer Method）。分治法的基本思想是：将原问题分解为若干个规模更小但结构与原问题相似的子问题，递归地解答这些子问题，然后将这些子问题的解组合为原问题的解。

快速排序的基本思想是：假设当前待排序的无序区为A[low…high]，利用分治法描述如下所示。

- 分解：在A[low…high]中任选一个记录作为基准（Pivot），以此为基准将当前无序区划分为左、右两个较小的子区间A[low…（pivotpos-1）]和A[（pivotpos+1）…high]，并使左边子区间中所有记录的关键字均小于等于基准记录（pivot），右边的子区间中所有记录的关键字均大于等于pivot，而基准记录pivot则位于正确的位置上，无须参加后续的排序。

注意：划分的关键是求出基准记录所在位置pivotpos。划分的结果可以简单地表示为：A[low…(pivotpos-1)]≤A[pivotpos]≤A[(pivotpos+1)..high]，其中pivot=A[pivotpos]，low≤pivotpos≤high。

- 求解：通过递归调用快速排序对左、右子区间A[low…（pivotpos-1）]和A[（pivotpos+1）…high]快速排序。
- 组合：当“求解”步骤中的两个递归调用结束时，其左、右两个子区间已有序，因而对快速排序而言，“组合”步骤无须做什么，可看做是空操作。

面试例题4：编程实现快速排序。

考点：快速排序算法的实现。

出现频率：★★★★

解析

程序代码如下所示：

```
1    void quick_sort(int a[], int low, int high)
2    {
3        int i, j, pivot;
4        if (low < high)
5        {
6            pivot = a[low];
7            i = low;
8            j = high;
9            while(i<j)
10           {
11               while (i<j && a[j]>=pivot)
```

```
12                          j--;
13                  if(i<j)
14                          a[i++]=a[j];        //将比 pivot 小的元素移到低端
15
16                  while (i<j && a[i]<=pivot)
17                          i++;
18                  if(i<j)
19                          a[j--] = a[i];      //将比 pivot 大的元素移到高端
20          }
21          a[i] = pivot;                       //pivot 移到最终位置
22          quick_sort(a, low, i-1);    //对左区间递归排序
23          quick_sort(a, i+1, high);   //对右区间递归排序
24      }
25  }
```

pivot 的初始值为 a[low]。局部变量 *i* 和 *j* 分别代表 low 和 high 的位置。按照下面的步骤进行一次交换。

- 把比 pivot 小的元素移到低端（low）。
- 把比 pivot 大的元素移到高端（high）。
- 将 pivot 移到最终位置，此时这个位置的左边元素的值都比 pivot 小，而其右边元素的值都比 pivot 大。
- 对左、右区间分别进行递归排序，从而把前面所进行的粗排序逐渐细化，直至最终 low 和 high 交汇。

测试程序如下所示：

```
1   void main()
2   {
3       int data[9] = {54,38,96,23,15,72,60,45,83};
4       quick_sort(data, 0, 8); //快速排序
5       for(int i = 0; i<9; i++)
6           cout << data[i] << " ";  //打印排序后的数组
7   }
```

执行结果如下：

```
15 23 38 45 54 60 72 83 96
```

12.3 选择排序

选择排序（Selection Sort）的基本思想是：每一次从待排序的记录中选出关键字最小的记录，顺序放在已排好序的子文件的最后，直到全部记录排序完毕。

常用的选择排序方法有直接选择排序和堆排序。

12.3.1 直接选择排序

直接选择排序的基本思想：*n* 个记录的直接选择排序可经过（*n*-1）次直接选择排序得到有序结果。

- 初始状态：无序区为 A[1…*n*]，有序区为空。
- 第 1 次排序：在无序区 A[1…*n*]中选出最小的记录 A[*k*]，将它与无序区的第 1 个记录 A[1]交换，使 A[1…1]和 A[2…*n*]分别为记录个数增加 1 个的新有序区和记录个数减少 1 个的新无序区。
- 第 *i* 次排序：第 *i* 次排序开始时，当前有序区和无序区分别为 A[1…i-1]和 A[*i*…*n*]（1≤*i*≤*n*-1）。该次排序从当前无序区中选出关键字最小的记录 A[*k*]，将它与无序区的第 1 个记录 A[*i*]交换，使 A[1…*i*]和 A[（*i*+1）…*n*]分别为记录个数增加 1 个的新有序区和记录个数减少 1 个的新无序区。

这样，*n* 个记录的文件的直接选择排序经过（*n*-1）次直接选择排序得到有序结果。

注意：直接选择排序是不稳定的。

面试例题 5：编程实现直接选择排序。

考点：直接选择排序算法的实现。

出现频率：★★★

解析

程序代码如下所示：

```
1   #include <iostream>
2   using namespace std;
3
4   void select_sort(int a[], int len)
5   {
6       int i,j,x,l;
7
8       for(i=0; i<len; i++)          //进行 n-1 次遍历
9       {
10          x = a[i];                 //每次遍历前 x 和 l 的初值设置
11          l = i;
12          for(j=i; j<len; j++)      //遍历从 i 位置向数组尾部进行
13          {
14              if(a[j] < x)
15              {
16                  x = a[j];         //x 保存每次遍历搜索到的最小数
17                  l = j;            //l 记录最小数的位置
18              }
19          }
```

```
20                  a[l] = a[i];              //把最小元素与 a[i]进行交换
21                  a[i] = x;
22          }
23}
24
25      void main()
26      {
27              int data[9] = {54,38,96,23,15,72,60,45,83};
28              select_sort(data, 9);       //快速排序
29              for(int i = 0; i<9; i++)
30                      cout << data[i] << " "; //打印排序后的数组
31      }
```

select_sort 函数进行（*n*-1）次排序。局部变量 *x* 和 l 分别记录每次遍历时所得的最小元素值及所在位置，代码 20～21 行利用它们与 a[*i*]的交换。以 main 函数中的 data 数组为例，说明其具体步骤。

- 第 1 次排序：数组各元素为 54,38,96,23,15,72,60,45,83，此时 *i* 为 0，遍历整个数组得到最小元素 15，然后与 a[0]交换，结果为：15,38,96,23,54,72,60,45,83。
- 第 2 次排序：此时 *i* 为 1，遍历从 a[1]开始到数组末尾得到最小元素 23，然后与 a[1]进行交换，结果为：15,23,96,38,54,72,60,45,83。
- 第 3 次排序：此时 *i* 为 2，遍历从 a[2]开始到数组末尾得到最小元素 38，然后与 a[2]进行交换，结果为：15,23,38,96,54,72,60,45,83。

显然，每次排序都选出了一个最小的元素与遍历起始位置的元素进行交换。通过（*n*-1）次这样的排序，最终把整个数组进行了排序。

执行结果为：

```
15 23 38 45 54 60 72 83 96
```

12.3.2 堆排序

堆排序定义：*n* 个序列 A_1，A_2，…，A_n 称为堆，有下面两种不同类型的堆。

- 小根堆：所有子节点都大于其父节点，即 $A_i \leqslant A_{2i}$ 且 $A_i \leqslant A_{2i+1}$。
- 大根堆：所有子节点都小于其父节点，即 $A_i \geqslant A_{2i}$ 且 $A_i \geqslant A_{2i+1}$。

若将此序列所存储的向量 A[1...*n*]看为一棵完全二叉树的存储结构，则堆实质上是满足如下性质的完全二叉树：树中任意非叶节点的关键字均不大于（或不小于）其左右子节点（若存在）的关键字。

因此堆排序（HeapSort）是一树形选择排序。在排序过程中，将 R[l...*n*]看成是一棵完全二叉树的顺序存储结构，利用完全二叉树中双亲节点和孩子节点之间的内在关系，在当前无序区中选择关键字最大（或最小）的记录。

使用大根堆排序的基本思想方法如下。

（1）先初始化 A[1…*n*]建成一个大根堆，此堆为初始的无序区。

（2）再将关键字最大的记录 A[1]（即堆顶）和无序区的最后一个记录 A[*n*]交换，由此得到新的无序区 A[1...*n*-1]和有序区 A[*n*]，且满足 A[1...（*n*-1）]≤A[*n*]。

（3）由于交换后新的根 A[1]可能违反堆性质，故应将当前无序区 A[1...（*n*-1）]调整为堆。然后再次将 A[1...（*n*-1）]中关键字最大的记录 A[1]和该区间的最后一个记录 A[*n*-1]交换，由此得到新的无序区 A[1...（*n*-2）]和有序区 A[（*n*-1）...*n*]，且仍满足关系 A[1...（*n*-2）]≤A[（*n*-1）...*n*]，同样要将 A[1...（*n*-2）]调整为堆。

（4）对调整的堆重复进行上面的交换，直到无序区只有一个元素为止。

构造初始堆必须使用到调整堆的操作， Heapify 函数思想方法如下。

每次排序开始前 A[l...*i*]是以 A[1]为根的堆，在 A[1]与 A[*i*]交换后，新的无序区 A[1...(*i*-1)]中只有 A[1]的值发生了变化，故除 A[1]可能违反堆性质外，其余任何节点为根的子树均是堆。因此，当被调整区间是 A[low...high]时，只须调整以 A[low]为根的树即可。

可以使用“筛选法”进行堆的调整。A[low]的左、右子树（若存在）均已是堆，这两棵子树的根 A[2low]和 A[2low+1]分别是各自子树中关键字最大的节点。若 A[low]不小于这两个子节点的关键字，则 A[low]未违反堆性质，以 A[low]为根的树已是堆，无须调整；否则必须将 A[low]和它的两个子节点中关键字较大者进行交换，即 A[low]与 A[large]（A[large]=max（A[2low]，A[2low+1]））交换。交换后又可能使节点 A[large]违反堆性质，同样由于该节点的两棵子树（若存在）仍然是堆，故可重复上述的调整过程，对以 A[large]为根的树进行调整。此过程直至当前被调整的节点已满足堆性质，或者该节点已是叶子为止。上述过程就像过筛子一样，把较小的关键字逐层筛下去，而将较大的关键字逐层选上来。

面试例题 6：编程实现堆排序。

考点：堆排序算法的实现。

出现频率：★★★

解析

程序代码如下所示：

```
1   #include <iostream>
2   using namespace std;
3
4   int heapSize = 0;
5
6   //返回左子节点索引
7   int Left(int index) { return ((index << 1) + 1);}
8
9   //返回右子节点索引
10   int Right(int index) {return ((index << 1) + 2);}
11
12   //交换 a、b 的值
13   void swap(int *a, int *b) {int temp = *a;*a = *b;*b = temp;}
```

```
14
15      //array[index]与其左右子树进行递归对比
16      //用最大值替换 array[index],index 表示堆顶索引
17      void maxHeapify(int array[], int index)
18      {
19              int largest = 0;                //最大数
20              int left = Left(index);         //左子节点索引
21              int right = Right(index);       //右子节点索引
22
23              //把 largest 赋为堆顶与其左子节点的较大者
24              if ((left <= heapSize) && (array[left] > array[index]))
25                      largest = left;
26              else
27                      largest = index;
28
29              //把 largest 与堆顶的右子节点比较，取较大者
30              if ((right <= heapSize) && (array[right] > array[largest]))
31                      largest = right;
32
33              //此时 largest 为堆顶,左子节点,右子节点最大者
34              if (largest != index)
35              {
36                      //如果堆顶不是最大者,则交换,并递归调整堆
37                      swap(&array[index], &array[largest]);
38                      maxHeapify(array, largest);
39              }
40      }
41
42      //初始化堆，将数组中的每一个元素置放到适当的位置
43      //完成之后，堆顶的元素为数组的最大值
44      void buildMaxHeap(int array[], int length)
45      {
46              int i;
47              heapSize = length;              //堆大小赋为数组长度
48              for (i = (length >> 1); i >= 0; i--)
49                      maxHeapify(array, i);
50      }
51
52      void heap_sort (int array[], int length)
53      {
54              int i;
55
56              //初始化堆
57              buildMaxHeap(array, (length - 1));
58
```

```
59              for (i = (length - 1); i >= 1; i--)
60              {
61
62                      swap(&array[0], &array[i]);
            //堆顶元素 array[0]（即数组的最大值）被置换到数组的尾部 array[i]
63                      heapSize--;                     //从堆中移除该元素
64                      maxHeapify(array, 0);           //重建堆
65              }
66      }
67
68      int main(void)
69      {
70              int a[8] = {45, 68, 20, 39, 88, 97, 46, 59};
71              heap_sort (a, 8);
72              for(int i=0; i<8; i++)
73              {
74                      cout << a[i] << " ";
75              }
76              cout << endl;
77              return 0;
78      }
```

heap_sort 函数按照下面步骤进行排序。

（1）调用 buildMaxHeap 对数组进行堆的初始化（代码第 57 行）。

（2）由于堆顶元素（即 array[0]）的值是最大的（大根堆），因此把它与数组尾部进行交换，并把 heapSize 递减 1，即从堆中移除数组尾部元素。

（3）由于只有剩下的堆顶元素（array[0]）不满足堆，因此调用 maxHeapify 重建堆。

（4）对前面两步进行循环调用，直到堆中只含有堆顶，此时 heapSize 变为 1（即 *i* 为 0）。

其中 buildMaxHeap 函数初始化堆时也调用了 maxHeapify 函数，而 maxHeapify 函数使用递归的方法把堆调整为大根堆。

测试结果如下：

```
20 39 45 46 59 68 88 97
```

12.4 归并排序

归并排序（Merge Sort）是利用“归并”技术来进行排序。归并是指将若干个已排序的子文件合并成一个有序的文件。

两路归并算法的基本思路是：设两个有序的子文件（相当于输入堆）放在同一向量中相邻的位置上，即 A[low...*m*]与 a[（*m*+1）...high]，先将它们合并到一个局部的暂存向量 Temp（相当于输出堆）中，待合并完成后将 Temp 复制回 A[low...high]中。

归并排序有两种实现方法：自底向上和自顶向下。

自底向上的方法的基本思想如下所示。

- 第1次归并排序时，将待排序的文件A[1...n]看做是n个长度为1的有序子文件，将这些子文件两两归并，若n为偶数，则得到n/2个长度为2的有序子文件；若n为奇数，则最后一个子文件不参与归并，故本次归并完成后，前（n−1）/2个有序子文件长度为2，但最后一个子文件长度为1。
- 第2次归并则是将第1次归并所得到的n/2个有序的子文件两两归并，如此反复，直到最后得到一个长度为n的有序文件为止。
- 上述的每次归并操作均是将两个有序的子文件合并成一个有序的子文件，故称其为“二路归并排序”。类似地有k（k>2）路归并排序。

自顶向下算法的设计形式更为简洁。设归并排序的当前区间是A[low...high]，步骤如下。

- 分解：将当前区间一分为二，即求分裂点。
- 求解：递归地对两个子区间A[low...mid]和A[（mid+1）...high]进行归并排序。
- 组合：将已排序的两个子区间A[low...mid]和A[（mid+1）...high]归并为一个有序的区间R[low...high]。
- 递归的终结条件：子区间长度为1（一个记录自然有序）。

面试例题7：归并排序算法的实现（使用自顶向下的方法）。

考点：归并排序算法的实现。

出现频率：★★★

解析

归并排序算法采用顺序存储结构，这种结构易于在链表上实现。本题使用数组结构。根据前面介绍过的自顶向下算法步骤，可以实现如下的程序。

```
1    #include <iostream>
2    using namespace std;
3
4    //将分治的两端按大小次序填入临时数组最后把临时数组复制到原始数组中
5    //lPos 到 rPos-1 为一端，rPos 到 rEnd 为另外一端
6    void Merge(int a[], int tmp[], int lPos,int rPos, int rEnd )
7    {
8            int i, lEnd, NumElements, tmpPos;
9            lEnd = rPos - 1;
10           tmpPos = lPos;                      //从左端开始
11           NumElements = rEnd - lPos + 1;      //数组长度
12
13           while( lPos <= lEnd && rPos <= rEnd )
14           {
15                   if( a[lPos] <= a[rPos] )              //比较两端的元素值
```

```
16                              tmp[tmpPos++] = a[lPos++];  //把较小的值先放入 tmp 临时数组
17                      else
18                              tmp[tmpPos++] = a[rPos++];
19              }
20
21              //到这里，左端或右端只能有一端还可能含有剩余元素
22              while( lPos <= lEnd )                   //把左端剩余的元素放入 tmp
23                      tmp[tmpPos++] = a[lPos++];
24
25              while( rPos <= rEnd )                   //把右端剩余的元素放入 tmp
26                      tmp[tmpPos++] = a[rPos++];
27
28              for( i = 0; i < NumElements; i++, rEnd--)
29                      a[rEnd] = tmp[rEnd];                    //把临时数组复制到原始数组
30      }
31
32      void msort(int a[], int tmp[], int low, int high )
33      {
34              if(low >= high)  //结束条件，原子节点 return
35                      return ;
36
37              int middle = (low + high) / 2;                  //计算分裂点
38              msort(a, tmp, low, middle);                     //对子区间[low,middle]递归做归并排序
39              msort(a, tmp, （middle+1）, high);  //对子区间[（middle+1）,high]递归做归并排序
40              Merge(a, tmp, low, （middle+1）, high);  //组合,把两个有序区合并为一个有序区
41      }
42
43      void merge_sort( int a[], int len )
44      {
45              int *tmp = NULL;
46              tmp = new int[len];                             //分配临时数组空间
47              if(tmp != NULL)
48              {
49                      msort(a, tmp, 0, len-1 );               //调用 msort 归并排序
50                      delete []tmp;                           //释放临时数组内存
51              }
52      }
53
54      int main()
55      {
56              int a[8] = {8, 6, 1, 3, 5, 2, 7, 4};
57              merge_sort(a, 8);
58              return 0;
59      }
```

merge_sort 函数是归并的最外层调用，它调用了 msort 函数，msort 是归并算法的递归实现。

它的步骤与前面介绍的相同，分为 3 个步骤。

- 代码第 37 行，计算分裂点，把区间一分为二。
- 代码第 38～第 39 行，递归地对两个子区间 A[low...middle]和 A[（middle+1）...high]进行归并排序。
- 代码第 40 行，调用 Merge 函数合并两个排序后的区间。

Merge 函数将分治的两端（这两端是已经排好序的）按大小次序填入临时数组，最后把临时数组复制到原始数组中。

12.5 基数排序

基数排序是箱排序的改进和推广。

箱排序也称桶排序（Bucket Sort），其基本思想是：设置若干个箱子，依次扫描待排序的记录 R[0]，R[1]，…，R[*n*-1]，把关键字等于 *k* 的记录全都装入到第 *k* 个箱子里（分配），然后按序号依次将各非空的箱子首尾连接起来（收集）。

例如，要将一副混洗的 52 张扑克牌按点数 A<2<…<J<Q<K 排序，需设置 13 个“箱子”，排序时依次将每张牌按点数放入相应的箱子里，然后依次将这些箱子首尾相接，就得到了按点数递增排列的一副牌。

基数排序基于多关键字，如果文件中任何一个记录 R[*i*]的关键字都是由 d 个分量构成，而且这 d 个分量中每个分量都是一个独立的关键字，则文件是多关键字的（比如扑克牌有两个关键字：点数和花色）。

通常实现多关键字排序有两种方法，如下所示。

- 最高位优先 MSD（Most Significant Digit first）。
- 最低位优先 LSD（Least Significant Digit first）。

基数排序是典型的 LSD 排序方法，其基本思想是：从低位到高位依次对数据进行箱排序。在 d 次箱排序中，所需的箱子数是基数 rd（可能的取值个数）。

比如对于值为 10～99 的整数序列：45，13，58，64，29，74，39，18，使用基数排序需要 10 个箱子（0～9 标号）进行分配和收集。如果把每一个数看成是由两个关键字构成（个位数和十位数），那么可以对它们进行两次分配和收集（分别对于个位和十位），具体步骤如下。

- 对序列的各个元素按个位进行顺序装箱，即 45 装入 5 号箱，13 装入 3 号箱，58 和 18 装入 8 号箱，64 和 74 装入 4 号箱，29 和 39 装入 9 号箱。
- 从 0 到 9 号箱顺序依次收集到原序列。即 3 号箱的 13，4 号箱的 64 和 74，5 号箱的 45，8 号箱的 58 和 18，9 号箱的 29 和 39 被依次收集。序列变为：13，64，74，45，58，18，29，39。
- 对序列的各个元素按十位进行顺序装箱，即 13 和 18 装入 1 号箱，29 装入 2 号箱，39

装入3号箱，45装入4号箱，58装入5号箱，64装入6号箱，74装入7号箱。

- 再次从0到9号箱顺序收集到原序列，序列变为：13，18，29，39，45，58，64，74。此时完成基数排序。

对于一个两位数来说，其十位数当然比个位数关键，因此使用LSD方法时，先对个位数开始分配和收集。

面试例题8：使用基数排序对整数进行排序。

考点：基数排序算法的实现。

出现频率：★★

解析

前面已经分析了使用基数排序对整数序列进行排序的具体步骤。如果整数的范围没有指明时，需要查找数组最大的元素有多少位数以便确定需要进行几次分配和收集，还需要知道每一位是什么。例如数据167，不仅需要知道167是一个3位数，而且还需要知道它的个位是7，十位是6，百位是1。

程序代码如下所示：

```
1    #include <iostream>
2    #include <math.h>
3    using namespace std;
4
5    int find_max(int a[], int len)         //查找长度为 len 数组的最大元素
6    {
7        int max = a[0];                    //max 从 a[0]开始
8        for(int i=1; i<len; i++)
9        {
10           if( max < a[i] )               //如果发现元素比 max 大
11               max = a[i];                //就重新给 max 赋值
12       }
13       return max;
14   }
15
16   //计算 number 有多少位
17   int digit_number(int number)
18   {
19           int digit = 0;
20           do
21           {
22                   number /= 10;
23                   digit++;
24           } while(number != 0);
25           return digit;
26   }
```

```
27
28      //返回 number 上第 Kth 位的数字
29      int kth_digit(int number, int Kth)
30      {
31              number /= pow(10, Kth);
32              return number % 10;
33      }
34
35      //对长度为 len 数组进行基数排序
36      void radix_sort(int a[], int len)
37      {
38              int *temp[10];                          //指针数组,每一个指针表示一个箱子
39              int count[10] = {0,0,0,0,0,0,0,0,0,0};  //用于存储每个箱子装有多少元素
40              int max = find_max(a, len);             //取得序列中的最大整数
41              int maxDigit = digit_number(max);       //得到最大整数的位数
42              int i, j, k;
43              for(i=0; i<10; i++)
44              {
45                      temp[i] = new int[len];             //使每一个箱子能装下 len 个 int 元素
46                      memset(temp[i], 0, sizeof(int) * len);  //初始化为 0
47              }
48              for(i=0; i<maxDigit; i++)
49              {
50                      memset(count, 0, sizeof(int) * 10);  //每次装箱前把 count 清空
51                      for(j=0; j<len; j++)
52                      {
53                              int xx = kth_digit(a[j], i);      //将数据安装位数放入到暂存数组中
54                              temp[xx][count[xx]] = a[j];
55                              count[xx]++;                      //此箱子的计数递增
56                      }
57                      int index = 0;
58                      for(j=0; j<10; j++)                   //将数据从暂存数组中取回，放入原始数组中
59                      {
60                              for(k=0; k<count[j]; k++)     //把箱子里所有的元素都取回到原始数组
61                              {
62                                      a[index++] = temp[j][k];
63                              }
64                      }
65              }
66      }
67
68      int main(void)
69      {
70              int a[] = {22, 32, 19, 53, 47, 29};
71              radix_sort(a, 6);
```

```
72
73          return 0;
74      }
```

下面简单说明一下 radix_sort 函数的执行步骤。

- 代码第 40～41 行，调用 find_max 取得序列中的最大整数，并调用 digit_number 得到其最大位数 maxDigit。
- 代码第 43～47 行，分配 10 个足够大的箱子来存放序列中的整数。
- 代码第 51～56 行，针对序列中整数的个位数，进行第 1 次分配箱子。
- 代码第 57～64 行，依次收集每个箱子的元素，放回到原始数组中。

一共需要进行 maxDigit 次，每一次针对序列中整数的不同位数分配箱子，代码第 53 行调用了 kth_digit 计算元素各个位数的数字，以确定放入哪一个箱子。另外在 radix_sort 函数里还有一个局部数组 count，它被用来在每次分配箱子后，保存各箱子里所含的整数元素。

12.6 各种排序方法比较

已经介绍的各种排序算法，按平均时间将排序分为 4 类，如下所示。

（1）平方阶（O（*n*2））排序。

一般称为简单排序，例如直接插入、直接选择和冒泡排序。

（2）线性对数阶（O（nlgn））排序。

如快速、堆和归并排序。

（3）O（*n*1+￡）阶排序。

￡是介于 0 和 1 之间的常数，即 0<￡<1，例如希尔排序。

（4）线性阶（O（*n*））排序。

如桶、箱和基数排序。

各种排序方法比较结果是：简单排序中直接插入最佳，快速排序最快，当文件为正序时，直接插入和冒泡最好。

影响排序效果的因素是：因为不同的排序方法适应于不同的应用环境和要求，所以选择合适的排序方法应综合考虑下列因素。

- 待排序的记录数目 *n*。
- 记录的大小（规模）。
- 关键字的结构及其初始状态。
- 对稳定性的要求。
- 语言工具的条件。
- 存储结构。
- 时间和空间复杂度。

不同条件下，排序方法的选择如下。

- 若 n 较小（如 $n \leq 50$），可采用直接插入或直接选择排序。

当记录规模较小时，直接插入排序较好；否则因为直接选择移动的记录数少于直接插入，应选直接选择排序为宜。

- 若文件初始状态基本有序，则应选用直接插入、冒泡或随机的快速排序为宜。
- 若 n 较大，则应采用时间复杂度为 O（$n\lg n$）的排序方法：快速排序、堆排序或归并排序。快速排序是目前基于比较的内部排序中被认为是最好的方法，待排序的关键字是随机分布时，快速排序的平均时间最短；堆排序所需的辅助空间少于快速排序，并且不会出现快速排序可能出现的最坏情况，这两种排序都是不稳定的。若要求排序稳定，则可选用归并排序。通常将归并排序和直接插入排序结合在一起使用，先利用直接插入排序求得较长的有序子文件，然后再两两归并之。因为直接插入排序是稳定的，所以改进后的归并排序仍是稳定的。
- 在基于比较的排序方法中，每次比较两个关键字的大小之后，仅仅出现两种可能的转移，因此可以用一棵二叉树来描述比较判定过程。

当文件的 n 个关键字随机分布时，任何借助于“比较”的排序算法，至少需要 O（$n\lg n$）的时间。

箱排序和基数排序只需一步就会引起 m 种可能的转移，即把一个记录装入 m 个箱子之一，在一般情况下，箱排序和基数排序可能在 O（n）时间内完成对 n 个记录的排序。但是，箱排序和基数排序只适用于像字符串和整数这类有明显结构特征的关键字，而当关键字的取值范围属于某个无穷集合（例如实数型关键字）时，无法使用箱排序和基数排序，这时只有借助于“比较”的方法来排序。

若 n 很大，记录的关键字位数较少且可以分解时，采用基数排序较好。虽然桶排序对关键字的结构无要求，但它只有在关键字是随机分布时才能使平均时间达到线性阶，否则为平方阶。同时要注意，箱、桶、基数这 3 种分配排序均假定了若关键字为数字，则其值是非负的，否则将其映射到箱号时，又要增加相应的时间。

面试例题 9：选择题——各种排序算法速度的性能比较。

考点：各排序算法速度的性能比较。

出现频率：★★★★★

下面哪种排序法对 12354 排序最快？

A．快速排序法。

B．冒泡排序法。

C．归并排序法。

解析

选择排序算法的时候，需要考虑以下几点。

- 数据的规模。
- 数据的类型。
- 数据已有的顺序。

一般来说，当数据规模较小时，应选择直接插入排序法或冒泡排序法。任何排序算法在数据量小时基本体现不出来差距。考虑数据的类型，比如全部是正整数时，应该考虑使用桶排序法。

考虑数据已有顺序，快速排序是一种不稳定的排序（当然可以改进），对于大部分排好的数据，快速排序会浪费大量不必要的步骤。认为快速排序好，是指大量随机数据下，使用快速排序效果最理想。而不是指所有情况。

所以，根据题目分析，12 354 数据量极小，已经基本排好序。所以此时冒泡排序法是最佳选择。

答案

B

面试例题 10：各排序算法的时间复杂度比较。

考点：各排序算法时间复杂度比较。

出现频率：★★★

写出下列算法的时间复杂度。

- 冒泡排序法。
- 选择排序法。
- 插入排序法。
- 快速排序法。
- 堆排序法。
- 归并排序法。

答案

冒泡排序算法时间复杂度是 O（n^2）。

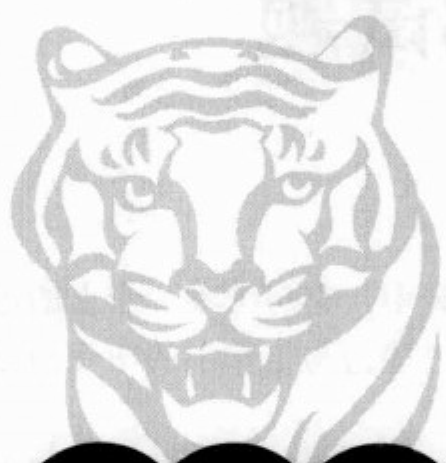
选择排序算法复杂度是 O（n^2）。

插入排序算法时间复杂度是 O（n^2）

快速排序法是不稳定的。最理想情况算法时间复杂度为 O（$n\log2n$），最坏为 O（n^2）。

堆排序算法时间复杂度为 O（$n\log n$）。

归并排序的时间复杂度是 O（$n\log2n$）。



第3篇

智力测试

第13章 智力测试题

有很多有趣的逻辑思考题目出现在跨国企业的面试招聘中，它对考察一个人的思维方式及思维方式转变能力有极其明显的作用。据研究显示，这样的能力与工作中的应变与创新思维息息相关。

这类智力型题目看似稀奇古怪，其实一般来说，它们并不是真要考应聘者的智力，而是考察以下几点。

- 思维的方式。这类问题的典型就是微软公司的“美国有多少个加油站”。此类问题，其实并没有什么惟一正确的答案，招聘方希望看到的是应聘者在分析和解决此问题中展示出来的思维过程。应对此类问题，注意力应放在合理假设、正确推理上，尽量把自己清晰的思维过程让面试者完全明白，而非什么数字是正确。
- 逻辑推理能力。这类题目一般都会给出一系列条件，然后让求职者从相互关系中推理出答案。题目的逻辑关系可能较复杂，建议拿纸写下来推理过程，理顺自己的思路，即使最终没能得出正确答案，也能让招聘方知道求职者是怎么推理的。
- 处世应变能力。此类问题中有很多大家常见的脑筋急转弯题。有些超常问题要靠平时多积累，有些问题无标准答案，冷静处理即可。

所以，回答这类题目时，必须冲破思维定式，试着从不同的角度考虑问题，不断进行逆向思维、换位思考，并且把题目与自己熟悉的场景联系起来，切忌思路混乱。最重要的是要尽量让面试官知道是怎么得出结论的，至于具体答案是否靠谱，反而是次要的。

13.1 数学能力

这种类型的智力测试题重点考察应聘者应用数学解决实际问题的基本能力。通常这类题并不需要很高的数学技巧。应聘者只要保持清醒的头脑，并且注意一些细节就能够顺利完成。

面试例题 1：元帅统领 8 员将军，每位将军各分 8 个营，每个营里面摆 8 个阵，每个阵配置 8 位先锋，每位先锋 8 面旗头，每面旗头有 8 个队，每个队分设 8 个

组，每个组带领 8 个兵。请问，元帅共有多少兵？

解析

首先，元帅统领 8 员将军，每位将军各分 8 个营，即元帅统领 8×8 个营；

其次，每营里面摆 8 个阵，即元帅统领 $8 \times 8 \times 8$ 个阵。

以下的计算和上面类似，即不断地乘以一个整数。

由于 8 个条件都是数字 8，所以元帅总共有 8^8 个兵士。

答案

元帅总共有 8^8 个兵士。

面试例题 2：有两只乌龟一起赛跑。甲龟到达 10m 终点线时，乙龟才跑了 9m。现在如果让甲龟的起跑线退后 1m，这时两龟再同时起跑比赛，问甲、乙两龟是否同时到达终点？

解析

这道题有一个陷阱，如果不仔细思考，仅从距离方面来判断，很容易得出甲、乙两龟同时到达终点的错误判断。

甲、乙两龟各自的用时等于各自的距离除以各自的速度。由前面的条件，也就是甲龟到达 10m 终点线时，乙龟才跑到 9m 可知，甲龟的速度是乙龟的 10/9 倍。设甲龟的速度为每小时 vm，则乙龟的速度为每小时 $0.9v$m。

如果让甲龟的起跑线退后 1m，则甲龟一共跑 11m，而乙龟仍然是 10m，则甲龟跑完 11m 需时 $11/v$，而乙龟跑完 10m 需时 $10v/9$。显然 $10v/9$ 大于 $11/v$，所以乙龟跑完 10m 比甲龟跑完 11m 用时要长，即甲龟先到。

答案

甲龟先到终点。

面试例题 3：麦克因工作繁忙，决定临时请尼克来协助他工作。规定以一年为期限，一年的报酬为 600 美元与一台电视机。

可是尼克工作了 7 个月后，因急事必须离开麦克，并要求麦克付给他应得的钱和电视机。由于电视机不能拆散付给他，结果尼克得到了 150 美元和一台电视机。

现在请你想一想：这台电视机值多少钱？

解析

根据题意，可以得到下面两个条件。

- 尼克一年的报酬为 600 美元与一台电视机。
- 尼克工作 7 个月后得到了 150 美元和一台电视机。

假设尼克一个月的报酬是 x 美元，并且这台电视机的价值是 y 美元，则我们可以把上面两个条件转换成一个二元一次方程组，通过解方程组得到电视机的价值。

方程组如下。

```
12x = 600 + y;
7x = 150 + y;
```

方程组的解为：

```
x = 90
y = 480
```

因此电视机值480美元。

答案

电视机值480美元。

面试例题4：有4个小孩看见一块石头正沿着山坡滚下来，便议论开了。

"我看这块石头有17kg重，"第1个孩子说。

"我说它有26kg，"第2个孩子不同意地说。

"我看它重21kg"，第3个孩子说。

"你们都说得不对，我看它的正确重量是20kg，"第4个孩子争着说。

他们4人争得面红耳赤，谁也不服谁。最后他们把石头拿去称了一下，结果谁也没猜准。其中一个人所猜的重量与石头的正确重量相差2kg，另外两个人所猜的重量与石头的正确重量之差相同。当然，这里所指的差，不考虑正负号，取绝对值。请问这块石头究竟有多重？

解析

根据题意，可以整理为下面几个条件。

（1）石头的重量不为17kg、20kg、21kg、26kg中的一个。

（2）一人所猜的重量与石头的正确重量相差2kg。

（3）另外两个人所猜的重量与石头的正确重量之差相同（取绝对值）。

由条件可知，石头的重量只有下面几种可能。

15kg、19kg、18kg、22kg、23kg、24kg、28kg。

为了方便叙述，我们称4个孩子分别为甲、乙、丙、丁。接下来对每一种可能情况进行分析。

- 石头若为15kg，此时只有甲满足条件（2），而与条件（3）相符的可能是乙、丙、丁，由于乙、丙、丁猜的重量与石头的重量差分别为5、6、11，3个人猜的重量与石头的正确重量之差都不相同，所以与条件（3）不可能相符。
- 石头若为19kg，此时甲或乙都满足条件（2），如果甲满足条件（2），则条件（3）中的两人可能是乙、丙、丁，由于乙、丙、丁猜的重量与石头的重量差分别为1、2、7，都不可能相同，所以与条件（3）不可能相符；如果乙满足条件（2），则条件（3）中的两人可能是甲、丙、丁，由于甲、丙、丁猜的重量与石头的重量差分别为1kg、2kg、7kg，都不可能相同，所以与条件（3）不可能相符。
- 由于判断原则简单，在这里对剩余情况的分析不再赘述，由此可以推断出石头为23kg。

此时丙满足条件（2），乙和丁满足条件（3）。

答案

石头为23kg。

面试例题5：一家有4个兄弟，他们4人年龄的乘积为14。那么，他们各自的年龄是多大?当然年龄应该是整数。

解析

本题只有一个条件，就是4人年龄乘起来的积为14。由于数字14只能被分解为下面两种乘积：一种是1×1×2×7，另一种是1×1×1×14。

因此4兄弟只能是以上两种组合。

答案

4个兄弟的年龄分别是：1、1、2、7或者1、1、1、14。

面试例题6：一位先生要到10层楼的第8层去办事，不巧正赶上停电，电梯无法使用，他只能够步行上楼。如果他从第1层爬到第4层需要用48s，那么请问，以同样的速度走到第8层需要多少秒？

解析

因为第1层是起点，所以从第1层到第4层的距离是3个楼层，而从第4层爬到第8层的距离是4个楼层。根据条件可知，这位先生步行3个楼层需要48s，则每个楼层需要16s，因此他步行4个楼层需要64s。

答案

以同样的速从第4层走到第8层需要64s。

面试例题7：现在有3种不同重量的标准砝码1克、3克、9克。请问可以称出多少不同重量的物品？在进行称量时，要称的东西与已知的标准砝码可以任意地放在天平的两边的称盘上，每种砝码只有一只，并且不准复制。

解析

现在有3种不同重量的标准砝码1克、3克、9克，显然可能称量的范围必定在1～13克。现在对于1克到13克之间的各个重量进行讨论。

- 称量1克的物品时，天平左边放1克砝码，右边放物品。
- 称量2克的物品时，天平左边放3克砝码，右边放物品+1克砝码。
- 称量3克的物品时，天平左边放3克砝码，右边放物品。
- 称量4克的物品时，天平左边放1克砝码和3克砝码，右边放物品。
- 称量5克的物品时，天平左边放9克砝码，右边放1克砝码和3克砝码+物品。
- 称量6克的物品时，天平左边放9克砝码，右边放物品+3克砝码。
- 称量7克的物品时，天平左边放9克砝码+1克砝码，右边放物品+3克砝码。

- 称量 8 克的物品时，天平左边 9 克砝码，右边放物品+1 克砝码。
- 称量 9 克的物品时，天平左边放 9 克砝码，右边放物品。
- 称量 10 克的物品时，天平左边放 1 克砝码+9 克砝码，右边放物品。
- 称量 11 克的物品时，天平左边放 9 克砝码+3 克砝码，右边放物品+1 克砝码。
- 称量 12 克的物品时，天平左边放 9 克砝码+3 克砝码，右边放物品。
- 称量 13 克的物品时，天平左边放 1 克砝码+9 克砝码+3 克砝码，右边放物品。

所以 1～13 克中任何重量的物品都能称量。

答案

1～13 克中任何重量的物品都能称量。

面试例题 8：现有 9 千克米以及 50 克和 200 克的砝码各一个。问怎样在天平上只称量 3 次而称出 2 千克米？

解析

根据题意，有 50 克和 200 克的砝码各一个，即总共是 250 克（0.25 千克）的砝码。如果用砝码直接称量，例如：

第 1 次使用两个砝码称出 250 克米；

第 2 次砝码和米称出 500 克米；

第 3 次所有的砝码和米称出 1 千克米。

按照上面这种方式称量不可能得到 2 千克米，难道有什么条件没有注意吗？需要注意的是已知米的初始重量 9 千克，需要称量出来的 2 千克米与砝码重量之和为 2.25 千克，而 2.25 千克乘以 4 正好是 9 千克，于是可以采取如下的步骤进行称量。

- 第 1 次称量，不用砝码，直接把 9 千克米平均放到天平两端，平衡后，天平两端都称出 4.5 千克米；
- 第 2 次称量，同第 1 次一样，也不用砝码，直接把 4.5 千克米平均放到天平两端，平衡后，天平两端都称出 2.25 千克米；
- 第 3 次称量，把两个砝码都放在天平的一端，然后把第 2 次称出的 2.25 千克米放在天平的另一端，此时肯定是米那一端重，最后把全下的米逐渐放入砝码端直至两端平衡。此时，砝码端含有 2 千克的米。

答案

第 1 次称量使用 9 千克米称出 4.5 千克米；第 2 次与第 1 次相同，称出 2.25 千克米；第 3 次使用 2.25 千克米和 0.25 千克砝码称出 2 千克米。

面试例题 9：在我最喜欢的比萨饼店中，10 寸的比萨卖 4.99 美元。店主说，他们有 12 寸比萨，定价为每份 5.39 美元。请问：对于 12 寸的比萨饼该店给予买方多少折扣？

解析

根据条件 10 寸的比萨卖 4.99 美元，即每寸比萨卖 0.499 美元。那么 12 寸的比萨饼按规定要卖 $0.499 \times 12 = 5.988$ 美元。然而这一笔 12 寸比萨饼定价为每份 5.39 美元，也就是便宜了 $5.988 - 5.39 = 0.598$ 美元，折扣为 0.598/5.988，约为 0.1。

答案

折扣约为 0.1。

面试例题 10：纽约伊沙贝拉时装精品屋，从意大利购进了一件女式冬装。这衣服的购入价格再加二成是该店标出的销售价。由于半个月内未卖出去，女老板又将这个定价减去了一成，很快被一位漂亮小姐买走了，而女老板获利 400 元。

请问，这件高档女式冬装购入价是多少？

解析

本题有下面两个条件：

- 衣服的购入价格再加二成，是该店标出的销售价；
- 定价减去了一成，很快被一位漂亮小姐买走了。女老板获利 400 元。

假设衣服购入价为 x 元，则由条件可知，原来的销售价为 $1.2x$ 元，可得到下面的方程：

$1.2x \times 0.9 = x + 400$

解方程得到 $x = 5000$，即高档女式冬装购入价为 5000 元。

答案

高档女式冬装购入价为 5000 元。

面试例题 11：烧一根不均匀的绳，从头烧到尾总共需要 1 小时。现在有若干条材质相同的绳子，问如何用烧绳的方法来计时 1 小时 15 分钟呢？

解析

由于绳子是不均匀的，所以燃烧的速度不一样，从一头燃烧用 1 小时，那么从两头燃烧就会用 30 分钟，但要注意，若燃烧 1/4 绳子就不一定用 15 分钟了。因此可以采用以下步骤计算。

- 取 A、B、C3 条同长的绳子，点燃 A 绳的一端，同时点燃 B 绳的两端。
- 两端同时点燃的 B 绳燃尽用时 30 分钟，此时一端点燃的 A 绳也燃烧了 30 分钟，剩下的燃烧还要用 30 分钟。
- 把 A 绳另一端点燃，这样当 A 绳燃尽时时间过了 45 分钟。
- 最后把 C 绳两端同时点燃，30 分钟燃尽。

这样就一共用了 1 小时 15 分钟。

答案

A 绳从一头烧，同时 B 绳从两头烧。当 B 绳烧尽时，点燃 A 绳的另一头。A 绳燃尽后 C 绳从两头烧。结束时即为 1 小时 15 分钟。

面试例题 12：工人每周工作 7 天，报酬是 1 根金条，这根金条平分成相连的 7 段，必须在每天结束的时候给他们 1 段金条。如果只允许你两次把金条弄断，该如何给工人付酬？

解析

由于只允许两次把金条弄断，因此金条最多能被分成 3 份，由于每天都必须付给工人金条，因此必然有 1/7 那一份。剩下 6/7 金条的分割方式，有下面几种组合。

- 3/7 和 3/7
- 1/7 和 5/7
- 2/7 和 4/7

对于第 1 种组合，在第 2 天就不能够付给工人金条了，而对于第 2 种组合，在第 3 天就不能够付给工人金条了。

因此只有第 3 种组合，即把金条分成 1/7、2/7 和 4/7 3 份。付酬过程如下。

- 第 1 天必然给他 1/7；
- 第 2 天给他 2/7，让他找回 1/7；
- 第 3 天再给他 1/7，加上原先的 2/7 就是 3/7；
- 第 4 天给他那块 4/7，让他找回那两块 1/7 和 2/7 的金条；
- 第 5 天，再给他 1/7；
- 第 6 天和第 2 天一样；
- 第 7 天给他找回的 1/7。

面试例题 13：有 4 个装药丸的罐子，每个药丸都有一定的重量，被污染的药丸是没被污染的药丸的重量+1。只称量一次，如何判断哪个罐子的药被污染了？

解析

本题的限制是只称量一次，也就是需要一次确定是哪一个罐子中的药丸被污染，只有根据称出的总重量大小来判断。并且根据题意，每个药丸都有一定重量，所以一定重量是已知的。为了对 4 个药罐进行区分，必须拿出不同数量的药丸。这是因为，假如从 A、B 两个药罐中拿出了相同的药丸，如果被污染的药罐是 A、B 中的一个，则此时不能区分。

采取的称量策略如下。

- 从第 1 个药罐中取出 1 个药丸，从第 2 个药罐里取出 2 个药丸，第 3 个取出 3 个，第 4 个取出 4 个，一起放在电子秤上称；
- 如果与标准重量比重 1，就是 1 号罐被污染；
- 如果与标准重量比重 2，就是 2 号罐被污染；
- 如果与标准重量比重 3，就是 3 号罐被污染；
- 如果与标准重量比重 4，就是 4 号罐被污染。

答案

4 个罐子中分别取 1、2、3、4 个药丸，称出比正常重多少，即可判断出哪个罐子的药被污染。

面试例题 14：在罐头工厂工作的送货员 A，给一家食品公司送了 10 箱菠萝罐头。每个罐头重量是 800g，每箱装 20 个。当他送完货要回工厂的时候，接到了从工厂打来的电话，说这 10 箱中有一箱由于机器出了问题而混进了次品，每个罐头缺 50g 的分量，要送货员把这箱罐头送回工厂以便更换。但是，怎样从中找出这箱是次品罐头呢？

离他不远的路旁有一台自动称量体重的机器，也就是投进去 1 元硬币就可以称量一次重量。他的口袋里刚好就有一个 1 元硬币，那么他应该怎么充分利用这一次的机会，来找到那一箱不符合规格的产品呢？

解析

本题与前面称药罐的面试例题是同一类题，因此可以采取相同的方法。在前面的例题中，为了对 4 个药罐进行区分，分别拿出了 1、2、3、4 个药丸。这里为了对 10 箱菠萝罐头进行区分，可以分别拿出 1～10 个罐头，具体步骤如下。

将罐头排成一排，从左向右（反之亦然）取罐头，第 1 箱取 1 个，第 2 箱取 2 个，以次类推，第 9 箱取 9 个，第 10 箱取 10 个。全部一起过秤，若少 50 克，则第 1 箱为不合格，若少 100g，则第 2 箱为不合格，以此类推，少几个 50g，即为第几箱不合格。

答案

第 1 箱取 1 个，第 2 箱取 2 个，以此类推，第 9 箱取 9 个，第 10 箱取 10 个。全部一起过秤，若少 50g，则第 1 箱为不合格，若少 100g，则第 2 箱为不合格，由此得出，少几个 50g，即为第几箱不合格。

面试例题 15：1 元钱一瓶汽水，喝完后两个空瓶换一瓶汽水，问：你有 20 元钱，最多可以喝到几瓶汽水？

解析

根据题意，两个空瓶可以换一瓶汽水，有些人会不假思索地得出 30 瓶，即 20 元钱买 20 瓶，喝完后这 20 个空瓶换 10 瓶汽水。但是要注意，使用两个空瓶得到的一瓶汽水喝完后，这个空瓶也能继续进行交换汽水瓶。解题步骤如下。

- 20 元钱买了 20 瓶汽水，喝完后有 20 个空瓶。
- 20 个空瓶换了 10 瓶汽水，喝完后有 10 个空瓶。
- 10 个空瓶换了 5 瓶汽水，喝完后有 5 个空瓶。
- 5 个空瓶换了 2 瓶汽水，喝完后有 3 个空瓶。
- 3 个空瓶换了 1 瓶汽水，喝完后有 2 个空瓶。

- 2 个空瓶换了 1 瓶汽水，喝完后有 1 个空瓶。
- 最后一个空瓶换 1 瓶汽水，喝完换来的那瓶再把瓶子还给人家即可。

这样总共有 20+10+5+2+1+1+1，即 40 瓶汽水。

本题还有另一种解法，即由于两个空瓶可以换一瓶汽水，也就是说一个空瓶和瓶里的汽水价值相等，即都是 0.5 元，那么 20 元钱就恰恰能喝 40 瓶汽水。

答案

最多可以喝 40 瓶汽水。

面试例题 16：有一辆火车以 15km/h 的速度从北京开往广州，同时另一辆火车以 20km/h 的速度从广州开往北京。如果有一只鸟，以 30km/h 的速度和两辆火车同时启动，从北京出发，碰到另一辆车后就向相反的方向返回去飞，就这样依次在两辆火车之间来回地飞，直到两辆火车相遇。请问，这只鸟共飞行了多长的距离？

解析

由于鸟的飞行速度比两列火车的速度都要快，因此从两列火车同时出发到它们相遇前，鸟会进行许多次往返飞行。如果直接计算，需要计算多次往返距离，这是一个求极限的过程。但实际上，由于鸟是以 30km/h 的速度飞行的，只要计算它飞行了多长的时间，然后再乘以速度，就能轻易获得它飞行的距离。

假设北京和广州之间的距离是 Lkm，由于两辆火车都是匀速行驶，则它们相遇的时间为：Lkm/（15km/h+20km/h）=（L/35）h。注意两辆火车从出发到相遇的时间也是鸟飞行的时间。

鸟的速度乘以飞行的时间，即（30km/h）*（L / 35）h 等于（6L/7）h，也就是说鸟总共飞行了 6/7 倍的北京到广州的距离。

答案

鸟总共飞行的距离是北京到广州的距离的 6/7 倍。

面试例题 17：有一个农场主，雇用了 2 个临时工帮忙种小麦。其中一个叫汤姆，是一个耕地能手，但是他不会播种；而另一个叫尼克，他是播种的好手但不擅长耕地。这个农场主决定要种 10 公顷小麦，让他们各自包一半，于是，汤姆从东头开始耕地，而尼克从西头开始耕地。耕 1 亩地汤姆只要用 20 分钟，而尼克却需要 40 分钟，但是尼克播种的速度比汤姆要快 3 倍。

他们播种完工后，农场主按照他们的工作量给予他俩一共 100 元的工钱。请问：他们应该怎样分这份工钱才最合理？

解析

要计算如何分配最后的 100 元，首先需要知道他们各自的总工作量。

对于两个人来说，虽然耕地速度和播种的速度各不相同，导致他们最后完成的时间不相同，但是他们各自的总工作量是一样的，即都是耕种了 5 公顷的小麦。因此按劳取酬应该每人对半

分，即两人都是 50 元。

答案

两人都是 50 元。

面试例题 18：在一架飞机上，中间有一条过道，过道两边是座位，每一排有 3 个座位。两位空姐 A 和 B 分别给两边旅客分发旅行物品。刚开始，A 给右边的旅客发放了 6 份物品，空姐 B 过来告诉 A 左边的旅客才由 A 负责。于是 A 重新给左边旅客分发物品，B 给右边剩下的旅客发完物品，又帮 A 给左边旅客发了 15 份物品，最后两人同时结束工作。

请问：A 和 B 谁发放的物品多？多发了多少份？

解析

本题中虽然提到了每一排为 3 人，但是没有说总共有多少排座位，因此无法计算两位空姐 A 和 B 各自都发了多少份物品。在这里实际上没有必要计算发放的总数，由于飞机上过道的两边座位是一样的，因此 A 和 B 各自应发的物品总数是相同的，都假设为 N 个。

开始的时候，A 给右边的旅客发放了 6 份，导致 B 给右边剩下的旅客发放物品，如果 B 发完右边的物品后没有帮 A 发物品，则 A 发了 N+6 个，而 B 发了 N-6 个。

但是 B 又帮 A 发了 15 份，导致 A 发给左边旅客的物品少了 15 个，也就是说 A 发了 N+6-15，即 N-9 个，而 B 发了 N-6+15，即 N+9 个。

显然，B 发的多，B 比 A 多发了（N+9）-（N-9）=18 个物品。

答案

B 发的多，B 比 A 多发了 18 个物品。

面试例题 19：有 3 个人去住旅馆，订了 3 间一样的客房，一共付给老板 30 元。第 2 天，老板认为 3 间房 25 元就够了，于是让伙计退 5 元给 3 位客人，谁知伙计贪心，只每人退回 1 元，自己拿走 2 元，这样一来便等于那 3 位客人每人各花了 9 元。3 个人一共花了 27 元，再加上伙计独吞了 2 元，总共是 29 元，可是当初他们 3 个人一共付出 30 元，那么还有 1 元在哪里呢？

解析

这是一道著名的偷换概念的数学题！这里伙计独吞的 2 元被出题者偷换了概念。

这 3 个人每人最后花了 10-1=9 元，也就是一共花了 9×3=27 元。

这 27 元包括老板得到的 25 元+伙计拿走的 2 元。如果加上他们 3 人每人拿回的 1 元，正好是最初所付的 30 元。

伙计拿走的 2 元是包含在 27 元里的，是 3 位房客付出的钱，而不是他们拿回去的钱。本题中拿 27 元与伙计拿走的 2 元相加纯属偷换概念。

答案

伙计拿走的 2 元是包含在 27 元里的，用这 2 元和 27 元相加属于混淆概念。

面试例题 20：有 7g、2g 砝码各一个，天平一个，如何只用这些物品分 3 次将 140g 的盐分成 50g、90g 各一份？

解析

这道题的意图很明显，如果只拿一个 7g 砝码和一个 2g 砝码放在天平上，称 3 次最多只能得到（7+2）×3＝27g 的盐，这与 50g 盐的差距很大。所以必须利用已经称出的盐去称盐。

这里给出两种称量方法。

第 1 种方法，3 次称量步骤如下。

- 第一次称量时，只有一个 7g 砝码和一个 2g 砝码，此时把它们放在天平的左边，右边放上盐，平衡时天平右边即可得到 9g 盐。
- 第 2 次称量时，除了一个 7g 砝码和一个 2g 砝码之外，还有第 1 次称出的 9g 盐。把 7g 砝码和 9g 盐放在天平的左边，右边放上盐，平衡时天平右边即可得到 16g 盐。
- 第 3 次称量时，除了一个 7g 砝码和一个 2g 砝码之外，还有第 1 次称出的 9g 盐以及第 2 次称出的 16g 盐。把前两次称出的 25g 盐放在天平的左边，右边放上盐，平衡时即可得到 25g 盐。此时天平左边和右边都是 25g 盐，总共是 50g 盐。

第 2 种方法，3 次称量步骤如下。

- 第 1 次称量时，只有一个 7g 砝码和一个 2g 砝码，此时把它们放在天平的左边，右边放上盐，平衡时天平右边即可得到 9g 盐。
- 第 2 次称量时，除了一个 7g 砝码和一个 2g 砝码之外，还有第 1 次称出的 9g 盐。把所有砝码和 9g 盐放在天平的左边，右边放上盐，平衡时天平右边即可得到 18g 盐。
- 第 3 次称量时，除了一个 7g 砝码和一个 2g 砝码之外，还有第 1 次称出的 9g 盐以及第 2 次称出的 18g 盐。把 7g 砝码和第 2 次称出的 18g 盐放在天平的左边，右边放上 2g 砝码和盐，平衡时天平右边即可得到 23g 盐。总共得到：9+18+23=50g 盐。

上面两种方法相比，第 1 种方法优于第 2 种方法。这是因为在第 2 种方法的第 3 次称量时，其第 1 次称出的盐并没有放在天平的左边，所以必须找一个地方存放第 1 次称出的盐。

面试例题 21：如果你有无穷多的水，一个 3L 和一个 5L 的提桶，你如何准确称出 4L 的水？

解析

这道题可以使用倒推法解答。

一个 3L 和一个 5L 的提桶，而只有 5L 的提桶才能装下 4L 的水，所以最后称出的 4L 的水必定在 5L 的桶内。

于是最后的操作只有下面两种。

- 5L 的桶内只有 1L 的水，把满的 3L 提桶的水全部倒入 5L 提桶内。
- 3L 的桶内只有 2L 的水，把满的 5L 提桶的水往 3L 提桶内倒入 1L 的水，则 5L 提桶内剩余 4L 的水。

由于 5L 的提桶和 3L 的提桶相差 2L 的含水量，因此上面的第 2 种操作很容易实现。即把 5L 的提桶注满水，然后倒入到 3L 的提桶中，这样，5L 的提桶内只剩下 2L 的水。

整理完思路，按下面的步骤操作了。

（1）5L 的桶注满水，倒入 3L 的桶中，此时 5L 桶内剩余 2L 水。

（2）将 3L 桶中的水倒掉，将 5L 桶中的 2L 水倒入 3L 桶中，此时 3L 桶中有 2L 水，并且 3L 桶只剩余 1L 水的空间。

（3）将 5L 的桶装满水，然后向刚才的 3L 的桶中倒，使其注满水，此时 5L 桶中剩余的水就是 4L。

13.2 推理能力

逻辑推理题是大公司招聘中最为常见的一类试题，招聘方非常看重求职者的逻辑思维能力，并相信这种能力是漂亮地完成工作的基础。所以，求职者在回答这类问题时，重要的在于思路，往往思路正确，答案就近在咫尺；思路错误，往往就越走越远。许多难度较高的智力题都是这种类型的。

面试例题 22：一个岔路口分别通向诚实国和说谎国。有两个人，已知一个是诚实国的，另一个是说谎国的。诚实国的人永远说实话，说谎国的人永远说谎话。现在你要去说谎国，但不知道应该走哪条路，需要问这两个人。请问应该怎么问？

解析

此题的重点是需要分辨出这两个人中哪个人是说谎国的，哪个人是诚实国的。一旦判断出之后，就能顺利问路了。

根据已知条件，说谎国的人永远说假话，而诚实国的人永远说实话，于是可以设计一个答案是众所周知的问题问他们，例如以下问题。

（1）这是一个岔路口吗？

（2）你们一共两个人吗？

（3）你们俩都是说谎国的吗？

（4）你们俩都是诚实国的吗？

对于此类问题，说谎者必然这样回答。

（1）这不是一个岔路口。

（2）我们一共不是两个人。

（3）我们俩都是说谎国的。

（4）我们俩都是诚实国的。

而诚实者的回答如下。

（1）这是一个岔路口。

（2）我们一共是两个人。

（3）我们俩不都是说谎国的。

（4）我们俩不都是诚实国的。

于是说谎者和诚实者就能被立刻分辨，接下来就可以顺利地问路了。

答案

首先设计一个答案是众所周知的问题问他们，以此分辨说谎者和诚实者，例如“你们俩都是说谎国的吗？”等问题，然后再问路。

面试例题 23：有 4 个大小相同的球，分别为甲、乙、丙、丁。

甲和乙放在天平的一边，丙和丁在另一边，天平基本保持平衡。

乙和丙调换，乙和丁那一边较重。

如果天平一边是甲和丁，另一边是乙，则乙重。

请按重量由大到小排序。

解析

推理步骤如下。

根据 3 个条件，可以分别列出如下的关系式。

a．甲 +乙= 丙+丁

b．甲+丙<乙+丁

c．甲+丁<乙

（1）把 a 式和 b 式左右两边分别相加后可以得出以下结论。

（甲+乙）+（甲+丙）≤（丙+丁）+（乙+丁），即甲≤丁。

（2）b 式左右两边分别减去 a 式左右两边，可以得出以下结论。

（甲+丙）－（甲+乙）≤（乙+丁）－（丙+丁），即丙≤乙。

（3）把 a 式和 c 式左右两边分别相加后可以得出以下结论。

（甲+乙）+（甲+丁）<（丙+丁）+乙，即 2×甲<丙

（4）根据上面 3 步以及关系式 c 可得，按重量由大到小的排序方式可能有以下两种情况。

乙、丙、丁、甲和乙、丙、甲、丁。

如果为乙、丙、甲、丁，则不满足关系式 a，所以排序为：乙、丙、丁、甲。

答案

重量从大到小排序为：乙、丙、丁、甲。

面试例题 24：有一桶果冻，果冻分为黄色、绿色、红色 3 种不同颜色。闭上眼睛抓取果冻，一次抓取多少个就可以确定手中肯定有两个同一颜色的果冻？

解析

这道题很简单，分析步骤如下。

(1) 如果只抓取了 1 个果冻，则其颜色是黄色、绿色、红色 3 种之中的 1 种，不可能存在两个同一颜色的果冻。

(2) 如果抓取了 2 个果冻，则其颜色可能是黄色、绿色、红色 3 种之中的 1 种或 2 种，此时不能肯定有 2 个同一颜色的果冻。

(3) 如果抓取了 3 个果冻，则其颜色可能是黄色、绿色、红色 3 种之中的 1 种或 2 种或 3 种，此时也不能肯定有 2 个同一颜色的果冻。

(4) 如果抓取了 4 个果冻，则其颜色必然有重复，可以肯定至少有 2 个同一颜色的果冻。

答案

抓取 4 个就可以确定肯定有 2 个同一颜色的果冻。

面试例题 25：张老师的生日是 m 月 n 日，小明和小强都是张老师的学生，2 人都知道张老师的生日是下列 10 组日期中的一天。张老师把 m 值告诉了小明，把 n 值告诉了小强。张老师问小明和小强是否知道他的生日是哪一天？

```
3月4日  3月5日  3月8日
6月4日  6月7日
9月1日  9月5日
12月1日  12月2日  12月8日
```

小明说：如果我不知道的话，小强肯定也不知道。

小强说：本来我也不知道，但是现在我知道了。

小明说：哦，那我也知道了。

请根据以上对话推断出张老师的生日是哪一天？并说明原因。

解析

分析步骤如下。

(1) 小明说：如果我不知道的话，小强肯定也不知道。

小明能肯定小强不知道，这说明小强知道的 m 值肯定不是 7 和 2（因为 7 和 2 直接可以确定是 6 月 7 日和 12 月 2 日）。小明能肯定小强知道的 m 值不是 7 和 2，那么他自己知道的 m 值肯定不是 6 和 12。

于是范围变为：

```
3月4日  3月5日  3月8日
9月1日  9月5日
```

(2) 小强说：本来我也不知道，但是现在我知道了。

当小强知道了 m 值是 3 或者 9，他马上就知道了准确日期，所以小强知道不可能是 5，只能是 1、4、8 中的一个。于是范围又变为：

```
3月4日  3月8日
9月1日
```

（3）小明说：哦，那我也知道了。

小明知道了，这是因为月份是惟一的，即月份为9。

最后得出老师的生日为9月1日。

答案

老师的生日为9月1日。

面试例题26：一个经理有3个女儿，3个女儿的年龄加起来等于13，3个女儿的年龄乘起来等于经理自己的年龄。有一个下属知道经理的年龄，但不能确定经理3个女儿的年龄，这时经理说只有一个女儿的头发是黑的，然后这个下属就知道了经理3个女儿的年龄。请问经理的3个女儿的年龄分别是多少？为什么？

解析

分析过程如下。

（1）显然3个女儿的年龄都不为0，不然经理的年龄就为0了。

（2）因为3个女儿的年龄加起来等于13，因此可以得到下面几种组合：

```
1*1*11 = 11        1*6*6 = 36        3*3*7 = 63
1*2*10 = 20        2*2*9 = 36        3*4*6 = 72
1*3*9 = 27         2*3*8 = 48        3*5*5 = 75
1*4*8 = 32         2*4*7 = 42        4*4*5 = 80
1*5*7 = 35         2*5*6 = 60
```

（3）根据题意，有一个下属已知道经理的年龄，但仍不能确定经理3个女儿的年龄，这说明经理的年龄为36，因为以上的组合中，只有36有两项组合，即1、6、6或者2、2、9。

（4）最后一个条件，经理说只有一个女儿的头发是黑的，即只有2、2、9能够满足。所以3个女儿的年龄分别为2、2、9。

答案

经理的3个女儿的年龄分别为2、2、9。

面试例题27：有2位盲人，他们都各自买了2双黑袜和2双白袜，8双袜子的质地、大小完全相同，并且每双袜子都由一张商标纸连着。两位盲人不小心将8双袜子混在一起。他们每人怎样才能取回各自的黑袜和白袜呢？

解析

由于黑袜和白袜的质地、大小完全相同，所以盲人不能通过用手去摸来区别黑袜和白袜，当然也不能通过看来区别。由于既不能通过看，也不能通过摸，只有寻找别的途径了。本题中，还有一个重要的条件，就是每双袜子都有一张商标纸连着。是不是可以通过这个条件来解决问题呢？

由于袜子是2只成一双的，并且8双袜子有4双黑袜和4双白袜。由于是两个盲人，恰好能平均分配，因此把每双袜子的商标撕开，然后每人拿每双的一只，直到把8双袜子全部分完，结果每

人都拿到了4只黑袜和4只白袜，即2双黑袜和2双白袜。

答案

把每双袜子的商标撕开，然后每人拿每双中的1只。

面试例题28：在击鼠标比赛中，参赛者拉尔夫10秒钟能击10次鼠标，威利20秒钟能击20次鼠标，保罗5秒钟能击5次鼠标。以上各人所用的时间是这样计算的：从第1击开始，到最后1击结束。

他们是否打平手？如果不是，谁最先击完40次鼠标？

解析

根据第一个条件可知，拉尔夫10秒钟能击10次鼠标，威利20秒钟能击20次鼠标，保罗5秒钟能击5次鼠标。容易让人不假思索地认为他们3个人单击鼠标的速度都是每1秒钟单击1次，因此打成平手。

这样的答案是错误的，对于智力类型的题目不可能有这么简单的推理。如果仔细阅读题目，不难发现还有第2个条件——各人所用的时间的计算方式，即从第1击开始，到最后1击结束。

对于拉尔夫来说，从第1击开始，到最后1击结束一共花了10秒钟，也就是说他10秒钟能单击9次。同样，威利20秒钟能单击19次，保罗5秒钟能击4次。他们点击鼠标的速度分别如下所示。

拉尔夫每秒击0.9次。

威利每秒击0.95次。

保罗每秒击0.8次。

显然，威利的单击鼠标的速度最快，拉尔夫次之，保罗速度最慢。因此，他们不会打成平手，如果比赛单击40次鼠标，当然是威利获胜。

答案

他们不会打成平手，威利最先击完40次鼠标。

面试例题29：从前有A、B两个相邻的国家，它们的关系很好，不但互相之间贸易交往频繁，而且货币可以通用，汇率也相同，也就是说A国的100元等于B国的100元。可是两国因为一次事件而关系破裂了，虽然贸易往来仍然继续，但两国国王却宣布对方货币的100元只能兑换本国货币的90元。有一个聪明人，他手里只有A国的100元钞票，却借机捞了一大把，发了一笔横财。请你想一想，这个聪明人是怎样从中发财的？

解析

这道题考察的是应聘者能否运用逆向思维来解决问题。

根据题目条件，A、B两个国家由于关系破裂，虽然贸易往来仍然继续，但两国国王却宣布对方货币的100元只能兑换本国货币的90元。

用定向思维来思考，假如A国人要去B国做旅游或者生意，那么拿着A国的货币去B国，

并在B国换成B国的货币，则带去的钱会缩水至9/10。而同样，B国人带着B国的货币去A国交换A国的货币，也会缩水至9/10。可见，按照这样方式来交换，口袋里的钱会越来越少。

现在换一个方式来思考，如果使用A国的货币在A国内交换B国的货币呢？当然是能换更多的B国的钱。假如此时有100元A国的货币，由于在A国内90元A国货币能换100元B国货币，所以在A国内100元A国的货币能换111元B国的货币。同样，在B国内100元B国的货币能换111元A国的货币。按照这种方式不断在这两个国家进行钱币兑换，这个聪明人就发了一笔横财。

答案

在A国内用A国的货币换取B国的货币，然后在B国内把B国的货币全部换成A国的货币。如此进行不断交换，钱会越换越多。

面试例题30：如果下列每个人说的话都是假话，那么是谁打碎了花瓶？

夏克：吉姆打碎了花瓶。

汤姆：夏克会告诉你谁打碎了花瓶。

埃普尔：汤姆，夏克和我不太可能打碎花瓶。

克力斯：我没打碎花瓶。

艾力克：夏克打碎了花瓶，所以汤姆和埃普尔不太可能打碎花瓶。

吉姆：我打碎了花瓶，汤姆是无辜的。

解析

由于本题中的6个人说的话都是假话，所以首先可以把他们的话转化为下面的真话。

（1）夏克：吉姆没有打碎花瓶。

（2）汤姆：夏克不会告诉你谁打碎了花瓶。

（3）埃普尔：汤姆，夏克和我都可能打碎花瓶。

（4）克力斯：我打碎了花瓶。

（5）艾力克：夏克没有打碎花瓶，所以汤姆和埃普尔有可能打碎花瓶。

（6）吉姆：我没有打碎花瓶，汤姆打碎了花瓶。

接下来对每个人进行检查。

- 对夏克的检查，在（5）中明确说明了夏克没有打碎花瓶。
- 对汤姆的检查，在（6）中明确说明了汤姆打碎花瓶。
- 对埃普尔的检查，上面6条中都没有明确说明埃普尔是否打碎花瓶，与埃普尔有关的叙述只有（3）和（5），他们都是说埃普尔有打碎花瓶的可能。但是注意（3）是埃普尔自己说的话，因此埃普尔打碎了花瓶。
- 对克力斯的检查，在（4）中明确说明了克力斯打碎花瓶。
- 对艾力克的检查，上面6条中都没有出现艾力克是否打碎花瓶的记录，因此不能确定艾力克有没有打碎花瓶。

❑ 对吉姆的检查，在（6）中明确说明了吉姆没有打碎花瓶。

因此得出结论，夏克、吉姆没有打碎花瓶；汤姆、埃普尔、克力斯打碎了花瓶；不能确定艾力克有没有打碎花瓶。

答案

夏克、吉姆没有打碎花瓶；汤姆、埃普尔、克力斯打碎了花瓶；不能确定艾力克有没有打碎花瓶。

面试例题 31：鲁道夫、菲利普、罗伯特 3 位青年，一个当了歌手，一个考上大学，一个加入美军陆战队，个个都大有作为。现已知如下信息。

A．罗伯特的年龄比战士的大。

B．大学生的年龄比菲利普小。

C．鲁道夫的年龄和大学生的年龄不一样。

请问：3 个人中谁是歌手？谁是大学生？谁是士兵？

解析

推理过程如下。

首先可以确定罗伯特是大学生。这是因为由 B 和 C 可知，菲利普和鲁道夫的年龄与大学生的年龄都不一样，即他们都不是大学生，所以只有罗伯特是大学生。

当确定了罗伯特是大学生后，就出现了下面两种组合。

❑ 鲁道夫是歌手，菲利普是战士。

❑ 鲁道夫是战士，菲利普是歌手。

如果第 1 种组合成立，则原题的 3 个条件变为：

A．罗伯特的年龄比菲利普大。

B．罗伯特的年龄比菲利普小。

C．鲁道夫的年龄和罗伯特的年龄不一样。

很明显，上面的 A 和 B 相矛盾。

如果考虑第 2 种组合成立，此时原题的 3 个条件变为。

A．罗伯特的年龄比鲁道夫大。

B．罗伯特的年龄比菲利普小。

C．鲁道夫的年龄和罗伯特的年龄不一样。

此时他们之中，鲁道夫最小，然后是罗伯特，菲利普最大。条件不存在矛盾。因此结论是：罗伯特是大学生，鲁道夫是战士，菲利普是歌手。

答案

罗伯特是大学生，鲁道夫是战士，菲利普是歌手。

面试例题 32：在某宾馆的宴会厅里，有甲、乙、丙、丁 4 位朋友围桌而坐，侃侃而谈。

他们用中、英、法、日4种语言。现已知如下条件。

A．甲、乙、丙各会2种语言，丁只会1种语言。

B．有1种语言，4人中有3人都会。

C．甲会日语，丁不会日语，乙不会英语。

D．甲与丙、丙与丁不能直接交谈，乙与丙可以直接交谈。

E．没有人既会日语，又会法语。

请问：甲、乙、丙、丁各会什么语言？

解析

推理过程如下。

首先看条件B，有1种语言4人中有3人都会。假设这种语言不是丁会的那种，则甲、乙、丙都会这种语言，可以推出甲、乙、丙3人能够直接交谈，这与条件D矛盾（甲与丙不能直接交谈）。因此这种语言一定是丁会的那种语言。再根据条件D，丙与丁不能直接交谈，这说明甲、乙会丁说的语言，而丙不会。

现在根据以上推论的初步结果，可以把上面的A～E条件整理为：

a．甲、乙、丙各会2种语言，丁只会1种语言。

b．甲、乙会丁说的语言，而丙不会丁说的语言。

c．甲会日语，丁不会日语，乙不会英语。

d．甲与丙不能直接交谈，乙与丙可以直接交谈。

e．没有人既会日语，又会法语。

现在推理丁说的是何种语言，步骤如下。

- 由c可知，丁不会日语，乙不会英语。这说明丁说的语言是中文和法语之中的1种。
- 如果丁说的是法语，则由b和c可知，甲会日语和法语，这与e矛盾。
- 因此丁说的只能是中文。

甲、乙、丙的推断步骤如下。

- 由于丁说的是中文，由a、b、c可以推出甲会中文和日语。
- 因为甲会中文和日语，又从d可知，丙不会中文和日语，所以丙只能会英语和法语（丙会两种语言）。
- 乙会丁说的语言，即中文，并且他与丙可以直接交谈，这说明他会的另一种是丙会的英语和法语之中的一种。由条件c可知，乙不会英语，因此乙会的另一种语言只能是法语。所以乙会中文和法语两种语言。

通过上面的推理，可以得出下面的结论。

甲会中文和日语两种语言；乙会中文和法语两种语言；丙会英语和法语两种语言；丁会中文。

答案

甲会中文和日语两种语言；

乙会中文和法语两种语言；
丙会英语和法语两种语言；
丁会中文。

面试例题 33：有 4 个女人要过一座桥。她们都站在桥的一边，她们要在 17 分钟内全部通过这座桥。这时已是晚上，她们只有 1 个手电筒，并且最多只能 2 个人同时过桥。不管是谁过桥，不管是 1 个人还是 2 个人，必须要带着手电筒。每个女人过桥的速度不同，2 个人同时过桥速度以较慢的那个人为准。4 个女人过桥的时间如下。

第 1 个女人：过桥需要 1 分钟。
第 2 个女人：过桥需要 2 分钟。
第 3 个女人：过桥需要 5 分钟。
第 4 个女人：过桥需要 10 分钟。

如果第 1 个女人与第 4 个女人首先过桥，耗费 10 分钟。如果让第 4 个女人将手电筒送回去，那么等她回到桥头时，总共用去了 20 分钟，行动也就失败了。怎样可以让这 4 个女人在 17 分钟内顺利过桥呢？

解析

如果始终让第 1 个女人来回传递手电筒（因为第一个女人回来的时间最短），会得到下面的方案。

- 第 1 个女人和第 2 个女人过桥，需要 2 分钟。
- 第 1 个女人回来，需要 1 分钟。
- 第 1 个女人和第 3 个女人过桥，需要 5 分钟。
- 第 1 个女人回来，需要 1 分钟。
- 第 1 个女人和第 4 个女人过桥，需要 10 分钟。

一共用时 2+1+5+1+10=19 分钟。

显然上面的方案不成功，而让两个最慢的女人（第 3 个女人和第 4 个女人）同时过桥更能节省时间。下面为正确的方案。

- 第 1 个女人和第 2 个女人过桥，需要 2 分钟。
- 第 1 个女人回来，需要 1 分钟。
- 第 3 个女人和第 4 个女人过桥，需要 10 分钟。
- 第 2 个女人回来，需要 2 分钟。
- 第 1 个女人和第 2 个女人过桥，需要 2 分钟。

一共用时 2+1+10+2+2=17 分钟。

面试例题 34：S 先生在家里休息时，接到了一个陌生人打来的预约电话。对方很想在下

下个星期五去拜访 S 先生。但是 S 先生并不想见这个陌生人，于是他说："下下个星期五我非常忙。上午要开会，下午 1 点钟要去参加一个学生的婚礼，接着 4 点钟要去参加一个朋友孩子的葬礼，随后是我的叔叔的 70 寿辰宴会。所以那天我实在是没有时间来接待您的来访了。"

请仔细看题，S 先生的话里有一处是不可信的，是哪个地方？

解析

由于 S 先生不想见这个陌生人，因此他说自己下下个星期五很忙，并找了以下的借口。

- 上午要开会。
- 下午 1 点钟要去参加一个学生的婚礼。
- 4 点钟要去参加一个朋友的孩子的葬礼。
- 随后是叔叔的 70 寿辰宴会。

乍看之下，这 4 个理由都没有漏洞，那为什么说 S 先生的话里有一处是不可信的？

再仔细些就可以发现本题中还有一个时间限制，也就是陌生人要约定的时间是下下个星期五。

如果现在是星期五，下下个星期五距离现在有 12 天，所以陌生人预约的时间距离此刻至少有 12 天的时间。对上面 4 个借口分析可知参加葬礼是不可信的。因为 1 个人死后，他的葬礼一般在一个星期内举行，如果葬礼定于 12 天之后举行，那么除非知道这个人准确的死亡时间。在本题中，S 先生借口说 12 天后要去参加一个朋友的孩子的葬礼，显然是不可信的。

答案

S 先生借口说下下个星期五要去参加一个朋友的孩子的葬礼，这句话不可信。因为没有人会知道发生在两周后的死亡时间。

面试例题 35：5 个海盗抢到了 100 颗宝石，每一颗都有一样的大小和价值。他们决定这么分：

第 1 步，抽签决定自己的号码（1、2、3、4、5）。

第 2 步，首先，由 1 号提出分配方案，然后 5 个人进行表决，当且仅当超过半数的人同意时，按照 1 号的提案进行分配，否则 1 号将被扔入大海喂鲨鱼。

第 3 步，1 号死后，再由 2 号提出分配方案，然后 4 人进行表决，当且仅当超过半数的人同意时，按照 2 号的提案进行分配，否则 2 号将被扔入大海喂鲨鱼。

以此类推。

前提：每个海盗都是很聪明的人，都能很理智的判断得失，从而做出选择。

问题：最后的分配结果如何？

提示：海盗的判断原则：A．保命；B．尽量多得宝石；C．尽量多杀人。

解析

逻辑推理最重要的是要找对思路。

如果从 1 号开始往后（到 5 号结束）推理，则中间的假设非常多，因此很难完成。正确的推理过程是从后向前推理。

- 如果 1、2、3 号强盗都喂了鲨鱼，只剩 4 号和 5 号。此时 5 号一定投反对票使 4 号喂鲨鱼，这样 5 号就能独吞所有宝石。
- 当然 4 号也知道如果只剩两个人时 5 号肯定不会支持他，因此为了保命，无论 3 号给自己多少宝石，他也必须支持 3 号。
- 3 号也知道 4 号肯定会支持他，因此 3 号就会提（100，0，0）的分配方案，即对 4 号、5 号一毛不拔而将全部金币据为己有，因为他知道 4 号即使一无所获但还是会投赞成票，再加上自己一票，他的方案即可通过。
- 2 号推知到 3 号的方案，就会提出（98，0，1，1）的方案，即放弃 3 号，而给 4 号和 5 号各一枚金币。由于该方案对 4 号和 5 号来说比 3 号将会提出的分配方案更为有利，因此 4 号和 5 号得到好处后将支持 2 号，而不希望由 3 号来分配。这样，2 号将拿走 98 枚金币。
- 不过，1 号也能推知到 2 号的方案，1 号并将提出（97，0，1，2，0）或（97，0，1，0，2）的方案，即放弃 2 号，而给 3 号一枚金币，同时给 4 号（或 5 号）2 枚金币。由于 1 号的这一方案对于 3 号和 4 号（或 5 号）来说，相比 2 号分配时的方案更优，他们将投 1 号的赞成票，再加上 1 号自己的票，1 号的方案可获通过，97 枚金币可轻松落入囊中。这无疑使 1 号能够获取最大收益！

可以看出，本题的推理过程就优先考虑简化的极端情况，从而顺藤摸瓜，得出最后的结果。

面试例题 36：村子里有 50 个人，每人养 1 条狗。在这 50 条狗中有病狗（这种病不会传染）。于是人们就要找出病狗。每个人可以观察其他的 49 条狗，以判断它们是否生病，只是自己的狗不能观察。观察后得到的结果不得交流，也不能通知病狗的主人。主人一旦推算出自己家的是病狗就要枪毙自己的狗，而且每个人只有权利枪毙自己的狗，没有权利打死其他人的狗。第 1 天，第 2 天都没有枪响。到了第 3 天传来一阵枪声，问有几条病狗，如何推算得出？

解析

本题一开始就说 50 条狗中有病狗，说明了病狗至少 1 条。

因为可能有多条病狗，为了叙述方便，病狗的主人就用病狗主人 1、病狗主人 2、病狗主人 3 等来区分。

- 第 1 天，病狗的主人 1 如果看到其他的 49 个人的家里的狗都没病，而 50 条狗中至少有 1 条病狗，那么就会立即枪毙自己家里的狗。但是第 1 天没有枪声，说明了病狗的主人 1 在其他的 49 人家里看到至少 1 条病狗。其他病狗的主人同样观察到这个现象，会认为自己家的狗没有病，所以没开枪。由此确定病狗数大于 1。

- 第2天，病狗主人1和病狗主人2发现昨天没人开枪。如果他们第1天只看到另外49个人家中只有1条病狗，同时第1天已确定病狗数大于1，那么病狗主人1就会在第2天枪毙自己家里的狗。但是第2天也没有枪声，所以证明病狗主人1第1天看到了另外49个人家里还至少有2只狗，确定病狗数大于2。
- 第3天，病狗主人1和病狗主人2和病狗主人3看到昨天没人开枪，而第3天传来一阵枪声，说明3个病狗主人已经确定自己家里有病狗。所以3个病狗主人在其他人家里各看到2只病狗，所以确定病狗数为3。

答案

总共有3条病狗。

面试例题37：法院开庭审理一起盗窃案件，A、B、C三人被押上法庭。负责审理案件的法官这样想：肯提供真实情况的不可能是盗窃犯；与此相反，真正的盗窃犯为了掩盖罪行，一定会编造口供。因此，法官得出结论：说真话的肯定不是盗窃犯，说假话的肯定就是盗窃犯。审判的结果也证明了法官的想法是正确的。

法官先问A：“你是怎样进行盗窃的？”

A回答法官的问题：“叽哩咕噜，叽哩咕噜……”

A讲的是某地的方言，法官根本听不懂他讲的是什么意思。法官又问B和C：“刚才A是怎样回答我的提问的？叽哩咕噜，叽哩咕噜，是什么意思？”

B说：“禀告法官，A的意思是说，他不是盗窃犯。”

C说：“禀告法官，A刚才已经招供了，他承认自己就是盗窃犯。”B和C说的话法官是能听懂的。听了B和C的话之后，这位法官马上断定：B无罪，C是盗窃犯。

请问：聪明的法官为什么能根据B和C的回答，做出这样的判断？A是不是盗窃犯？

解析

根据本题的条件，说真话的肯定不是盗窃犯，说假话的肯定就是盗窃犯，A、B、C3人有一个人是盗窃犯，因此只有他一个人会说假话，其他两个人都会说真话。推理过程如下。

- 首先是对A的判断，如果A不是盗窃犯，那么A说的是真话，他会说自己“不是盗窃犯”；如果A是盗窃犯，那么A说的是假话，他也必然会说自己“不是盗窃犯”。因此A对于法官的提问，回答肯定都是说自己不是盗窃犯。因此不能判断A是否为盗窃犯。
- 然后是对B的判断，B禀告法官说A不是盗窃犯，由上可知不管A是不是盗窃犯，他的回答都是否定的，由此可知B如实地禀告了法官，所以B说的是真话，因此B不是盗窃犯。
- 最后是对C的判断，C说A承认自己是盗窃犯，由于B说的是真话，因此C说的必然是假话，所以C肯定是盗窃犯。

答案

法官问 A 的问题，不管 A 是否为盗窃犯，A 的回答都一定是否定的。在这种情况下，B 如实地转述了 A 的话，而 C 没有。因此 B 说了真话，C 说了假话。不能判断 A 是否为盗窃犯。

面试例题 38：在大西洋的"说谎岛"上，住着 X 和 Y 两个部落。X 部落总是说真话，Y 部落总是说假话。

有一天，一个旅游者迷路了，恰巧遇见一个土著人 A。

旅游者问："你是哪个部落的人？"

A 回答说："我是 X 部落的人。"

旅游者相信了 A 的回答，请他做向导。

他们在路途中，看到另一位土著人 B，旅游者请 A 去问 B 是属于哪一个部落。A 回来说："B 说他是 X 部落的人。"旅游者糊涂了，他问同行的逻辑博士：A 是 X 部落的人，还是 Y 部落的人呢？逻辑博士说：A 是 X 部落的人。为什么？

解析

1. 假设 A 是 X 部落的人

- 如果 A 遇见的 B 是 X 部落的人，由于 X 族人说真话，那么 A 向旅游者如实地传达了 B 的回答。
- 如果 A 遇见的 B 是 Y 部落的人，由于 Y 族人说假话，那么 A 也向旅游者如实地传达了 B 的回答。

2. 假设 A 是 Y 部落的人

- 如果 A 遇见的 B 是 X 部落的人，由于 X 族人说真话，那么 B 说自己是 X 部落的人，由于 A 是 Y 部落的人，他是说假话的，所以，他会把 B 的回答向旅游者传达为"B 说他是 Y 部落的人"。
- 如果 A 遇见的 B 是 Y 部落的人，由于 Y 族人是说假话的，那么 B 会说自己是 X 部落的人，而 A 也是 Y 部落，也会把 B 的回答传达为"B 说他是 Y 部落的人"。

从题目的给定条件可知，A 对旅游者传达的话是："B 说他是 X 部落的人。"可见，假定 A 是 Y 部落的人时得出的两个结论，都与题目给定条件矛盾，只有前一种假设（A 是 X 部落的人）符合题目给定条件。所以，做向导的 A 是 X 部落的人。

答案

不管 B 是 X 部落的人还是 Y 部落的人，B 都会说自己是 X 部落的人，A 如实地转述了 B 的话，因此 A 是 X 部落的人。

面试例题 39：S、P、Q3 位先生都具有足够的推理能力。这天，他们正在接受推理面试。

桌子的抽屉里有如下 16 张扑克牌。

```
红桃A、Q、4
黑桃J、2、3、4、7、8
草花K、Q、4、5、6
方块A、5
```

约翰教授从16张牌中挑出一张牌，并把这张牌的点数告诉P先生，把这张牌的花色告诉Q先生。然后，约翰教授问P先生和Q先生：你们能从已知的点数或花色中推知这张牌是什么牌吗？

于是，S先生听到如下的对话。

P先生："我不知道这张牌。"

Q先生："我知道你不知道这张牌。"

P先生："现在我知道这张牌了。"

Q先生："我也知道了。"

听完以上的对话，S先生思考了一会儿，正确推理出这张牌。请问：这张牌是什么牌？

解析

推理过程如下。

P先生知道这张牌的点数，Q先生知道这张牌的花色。

（1）首先由P先生说"我不知道这张牌。"以及随后Q先生对P先生说"我知道你不知道这张牌。"可知，这张牌的点数与其他牌的点数有重复。由于点数J、8、2、3、7、K、6都是惟一的，因此可以过滤掉筛选出点数重复的牌，如下所示。

```
红桃A、Q、4
黑桃4
草花Q、5、4
方块A、5
```

（2）然后P先生和Q先生都说知道什么牌了。A、Q、5在红桃、草花、方块中各有一张，并且这3种花色至少有2种点数，因此点数如果是A、Q、5中的一个，P先生和Q先生无论如何都是猜不出来的。只有花色为黑桃的点数只有一种，因此这张牌是黑桃4。

答案

这张牌是黑桃4。

面试例题40：一列火车上有3个工人，史密斯、琼斯、罗伯特，3人分别为消防员、司闸员、机械师。火车上同时有3个乘客与这3个工人名字相同。已知的条件如下。

A．罗伯特住在底特律。

B．司闸员住在芝加哥和底特律中间的地方。

C．琼斯一年赚2万美金。

D．有一个乘客和司闸员住在一个地方，每年的薪水是司闸员的3倍整。

E．史密斯台球打得比消防员好。

F. 和司闸员同名的乘客住在芝加哥。

根据上面条件，请问谁是机械师？

解析

实际上一共有 6 个人，分为两组，每组 3 个人的名字分别为史密斯、琼斯、罗伯特。下面是推理过程。

首先要搞清楚和司闸员住同一地方的乘客的名字。

- 条件 C 中，因为 2 万美金不能被 3 整除。这说明该乘客名字不是琼斯（当然还有第 2 个琼斯），于是他的名字可能为史密斯或者罗伯特。
- 条件 A 中，罗伯特住在底特律。这里注意题目中并没有指明哪个是罗伯特（是工人还是乘客），因此 2 个罗伯特都住在底特律。所以和司闸员同住一个地方的乘客名字必定是史密斯。

接下来推理出司闸员的名字。

- 条件 F 中，和司闸员同名的住芝加哥，罗伯特住底特律，所以司闸员既不叫史密斯又不叫罗伯特，司闸员名字是琼斯。

最后推出机械师的名字。

- 条件 E 中，史密斯不是消防员，并且也不是司闸员，因此史密斯就是机械师。

答案

史密斯是机械师。

面试例题 41：10 个人站成一列纵队，从 10 顶黄帽子和 9 顶蓝帽子中，取出 10 顶分别给每个人戴上。站在最后的第 10 个人说：“我虽然看见了你们每个人头上的帽子，但仍然不知道自己头上帽子的颜色。你们呢？”
第 9 个人说：“我也不知道。”
第 8 个人说：“我也不知道。”
第 7 个、第 6 个……
直到第 2 个人，依次都说不知道自己头上帽子的颜色。

出乎意料的是，第 1 个人却说：“我知道自己头上帽子的颜色了。”聪明的你知道第 1 个人戴的是什么颜色的帽子吗？

解析

对于第 10 个人来说，他能看到 9 顶帽子，如果 9 顶帽子都是蓝帽子，他肯定知道自己戴的是黄帽子，而他不知道，说明前面 9 顶帽子至少有 1 顶帽子是黄帽子，即他至少看到 1 顶黄帽子。

第 9 个人也知道第 10 个人的想法，如果他没看到黄帽子，肯定知道自己戴的是黄帽子，而他也不知道，说明前面 8 顶帽子至少有 1 顶帽子是黄帽子，即他也至少看到 1 顶黄帽子。

同理可知，第 8 个、第 7 个……直到第 2 个人，都至少看到 1 顶黄帽子。因此第 1 个人头上戴的是黄帽子。

第 1 个人通过以上推理，可知自己戴的是黄帽子。

答案

在 10 个人的情况下，只有第 1 个人是黄帽子，其他人才不能确定。如果第 1 个人是蓝帽子，则剩下的 9 个人中，必定有 1 个人能确定。

面试例题 42：有两个大于 1 且小于 10 的整数，把两数之和告诉甲，两数之积告诉乙。一开始两人都说不知道两数分别是什么。沉思一会儿后，乙说知道这两个数了，接着甲也说知道了。请问这两个数各是多少？

解析

由题意可知，这两个整数在 2～9。

把 2～9 的数分别与 2～9 的数相加，结果如下。

加 2：4、5、6、7、8、9、10、11

加 3：5、6、7、8、9、10、11、12

加 4：6、7、8、9、10、11、12、13

加 5：7、8、9、10、11、12、13、14

加 6：8、9、10、11、12、13、14、15

加 7：9、10、11、12、13、14、15、16

加 8：10、11、12、13、14、15、16、17

加 9：11、12、13、14、15、16、17、18

把 2～9 的数分别与 2～9 的数相乘，结果如下。

乘 2：4、6、8、10、12、14、16、18

乘 3：6、9、12、15、18、21、24、27

乘 4：8、12、16、20、24、28、32、36

乘 5：10、15、20、25、30、35、40、45

乘 6：12、18、24、30、36、42、48、54

乘 7：14、21、28、35、42、49、56、63

乘 8：16、24、32、40、48、56、64、72

乘 9：18、27、36、45、54、63、72、81

甲知道两数之和，乙知道两数之积。

很明显，甲在乙之后确定这两个数，也就是说甲刚开始并不能确定。这说明两数之和在上面的相加列表中不惟一，可以推出两数之和的范围应该在 6～16。因为 4 只能是 2 + 2，5 只能是 2 + 3，17 只能是 8 + 9，18 只能是 9 + 9，而其他的和数有至少两种相加组合，比如 7 = 3 + 4 或 7 = 2 + 5，16 = 8 + 8 或 16 = 7 + 9。

乙首先说自己知道这两个数了，这说明两数之积在上面相乘的列表中是惟一的（除去对称性相同，例如 3*5 = 5*3）。因此可以得到以下积数：

4、6、8、10、14、15、21、25、27、30、35、

40、42、45、48、54、56、63、64、72、81。

而甲随后也跟着说他也知道这两个数了，也就是说当他知道两数之积在上面相乘的列表中是惟一的时候，他就能确定两个数。由于两数之和的范围应该在 6～16，因此上面的积数可以首先除去 4、6、72、81。剩下的乘积为：

8、10、14、15、21、25、27、30、35、

40、42、45、48、54、56、63、64。

现在对每一个积数进行分解。

A—8：8 = 2*4，2 + 4 = 6

B—10：8 = 2*5，2 + 5 = 7

C—14：14 = 2*7，2 + 7 = 9

D—15：15 = 3*5，3 + 5 = 8

E—21：21 = 3*7，3 + 7 = 10

F—25：25 = 5*5，5 + 5 = 10

G—27：27 = 3*9，3 + 9 = 12

H—30：30 = 5*6，5 + 6 = 11

I—35：35 = 5*7，5 + 7 = 12

J—40：40 = 5*8，5 + 8 = 13

K—42：42 = 6*7，6 + 7 = 13

L—45：45 = 5*9，5 + 9 = 14

M—48：48 = 6*8，6 + 8 = 14

N—54：54 = 6*9，6 + 9 = 15

O—56：56 = 7*8，7 + 8 = 15

P—63：63 = 7*9，7 + 9 = 16

Q—64：64 = 8*8，8 + 8 = 16

显然，只有上面 A～D 中的加数没有重复，即这两个数有以下 4 种组合：2 和 4、2 和 5、3 和 5、3 和 7。

答案

这两个数可能有 4 种组合：2 和 4、2 和 5、3 和 5、3 和 7。

面试例题 43：A、B、C、D、E、F6 个人被邀请参加一项会议，到会情况满足下面的条件。

（1）A、B 至少有 1 个人参加会议。

（2）A、E、F3 个人中有 2 个参加会议。

（3）B 和 C2 个人一致决定，要么 2 个人都参加，要么 2 个人都不参加。

（4）A、D2个人中只要1个人参加。

（5）C、D2个人中也只有1个人参加。

（6）如果D不去，那么E也决定不去。

那么最后究竟有哪些人参加了会议呢？为什么？

解析

推理步骤如下。

- 首先判断D会不会去参加会议。

假如D参加会议，则可进行下面推断

根据条件（4）和（5），可知A和C都不去参加会议，再由条件（3），可知由于C不去导致B也不去，因此A、C、B都不会去参加会议。这与条件1矛盾（A、B至少有一个人参加会议）。

至此推断D一定没有参加会议。

- D没有参加，根据条件（4）推出A必然参加。
- D没有参加，根据条件（5）推出C也必然参加。
- D没有参加，根据条件（6）推出E必然没有参加。
- 由于C参加，根据条件（3）推出B必然参加。因此推出A、B、C参加会议，而D和E没有参加会议。
- 由于A参加，而E没有参加，所以根据条件（2）可知，F必然参加会议。

答案

参加会议的有A、B、C、F，D和E没有参加会议。

13.3 反应能力

有关反应能力的题目通常不会很难，它主要考察应聘者是否能对外来信号迅速做出相应的反应，因此，应聘者需要尽可能缩短答题时间。

面试例题44：有1种小虫，每隔2秒钟分裂一次。分裂后的2只新的小虫经过2s后又会分裂。如果最初某瓶中只有1只小虫，那么2s后变2只，再过2s后就变4只……2min后，正好满满一瓶小虫。

现在往空瓶内放入2只这样的小虫。问：经过多少时间后，正巧也是满满的一瓶？

解析

为了进行快速答题，不需要考虑总共需要小虫在规定时间内裂变多少次，或者这个瓶子总共能装多少只小虫。

实际上，只需要考虑2个条件。

- 裂变的规律是 2s 1 只小虫变成 2 只小虫。
- 原来是只一小虫需要 2 min 裂变装满小瓶，现在是 2 只小虫裂变。

因此，小虫是以 2 的乘方的速度增长的，而现在与原来的区别是初始为 2，也就是总裂变比原来少了一次，也就是比原来缩短了 2s，即 1min 58s。

答案

1 分钟 58 秒。

面试例题 45：某小镇车队有 17 辆公共汽车，每天在相距 197km 的青山镇与绿水镇之间往返运客。每辆车到达小镇后司机都要休息 8min。司机杰克上午 10 点 20 分开车从青山镇出发，在途中不时地遇到（有时是迎面驶来，有时是互相超越）本车队的车。下午 1 点 55 分他到达绿水镇，休息时，发现本队的其他司机一个都不在。请问：杰克一共遇到了本车队的几辆车？

解析

本题中有很多条件，但绝大多数都与解题无关，例如下面的条件。

- 青山与绿水两个小镇之间相距 197 千米。
- 每辆车到达小镇后司机都要休息 8 分钟。
- 司机杰克上午 10 点 20 分开车从青山镇出发。
- 下午 1 点 55 分他到达绿水镇。

看似有用的数据，但实际上仅有下面 2 条有用：

- 小镇车队共有 17 辆小公共汽车。
- 休息时发现本队的其他司机一个都不在。

由于司机杰克的车也属于这种小公共汽车，因此他遇到的是其他司机开的车，即总共为 17-1=16 辆车。

答案

杰克在路上遇到了 16 辆公共汽车。

面试例题 46：请在一分钟内回答下面的 10 道问题。

（1）您从西向东行走，不久向左转 270° 角行走，再向后转走，接着又向左转 90° 角走，最后又向后转走。请问，最终您是朝哪一个方向行走的？

（2）在 21 世纪前有这样一个年份，把它写成阿拉伯数字时，正看是这一年，倒过来看还是这一年。请问是哪一年？

（3）用 3 根火柴要摆成一个最小的数（不许把火柴折断或弯曲），这个数是多少？

（4）有一个高且狭窄的玻璃筒，筒里放着一只鲜鸡蛋。如果不许把玻璃筒倾斜，也不许用任何夹具把鲜鸡蛋夹起，那么，您有什么办法取出鲜鸡蛋？

（5）英国伦敦某公司采购员杰夫经常出差去法国巴黎，而且每次都是乘坐火车去的。有一

次，他又要出差去法国巴黎，但前一半路程是坐飞机去的，这比他平常坐火车去的速度要快 8 倍；后一半路程他是坐火车和汽车到达法国巴黎的，速度比他平常坐火车要慢一半。请问他这一次出差去法国巴黎，是否比他平常坐火车去节省时间？为什么？

（6）一只挂钟在 24 小时里分针和时针重合多少次？

（7）给您一根较长的粗铜线，用这根铜线将点燃着的蜡烛火焰熄灭，但又不许用铜线碰到蜡烛，请问有什么办法？

（8）有一根铁丝，如果用钳子把它剪断后，它仍然是一根与原来长度相等的铁丝。请问这是一根什么形状的铁丝？

（9）宇航员卡特在乘宇宙飞船进入太空前，用他所带的自来水笔为来访者签名留念。当他进入太空以后，他正忙着用这支笔写日记。您相信吗？

（10）有 12 个人要过河去，河边只有一条能够载 3 个人的小船。请问这 12 个人都过河，需要往返几次？

解析

- 题（1）：向左转 270° 角后，方向是南；再向后转走，方向是北；又向左转 90° 角走，方向是西，最后又向后转走，方向东，因此最后朝东走。
- 题（2）：数字 0～9 中，数字 0、1 和 8 是上下对称的，因此年份为 1881 和 1001。
- 题（3）：负数小于 0，因此 3 根火柴有一根用作负号，显然剩余两根摆成 11，即−11。
- 题（4）：因为不可以直接取出鲜鸡蛋，所以只有让鸡蛋自己出来，可以增加水的浮力（例如加醋）。
- 题（5）：本题中不需要计算坐飞机比坐火车节省了多少时间，只需要注意后一半路程他是坐火车和汽车到达法国巴黎的，速度比他平常坐火车要慢一半，也就是说后一段路程花的时间等于他平常坐火车去的总时间，再加上飞机的时间，显然比平常坐火车多用了时间。
- 题（6）：0 点开始，分针和时针碰到的时间大致为：0:00、1:05、2:11、3:16、4:22、5:27、6:33、7:38、8:43、9:49、10:54，一共有 11 次。

然后又从头再来，一共 22 次。

如果最后到达 00：00，计算在内一共是 23 次。

- 题（7）：可以把铜线绕成一个扇子形状，然后扇风将蜡烛吹灭。
- 题（8）：这个铁线是环状的，剪断后铁线没有了环，但长度和原来相同。
- 题（9）：进入太空之后，由于没有了地球的引力，所有东西都失去了重量，自来水笔中的墨水也是如此，因此不可能用自来水笔写日记。
- 题（10）：这里相当于一个船夫运 11 个人，去河对岸可坐 3 个人，从河对岸回来至少需要一个人（作为船夫）。所以每次送 2 个人去河对岸，总共需要渡 6 次。

面试例题 47：有 100 盏灯，并用 1～100 编上号，开始时所有的灯都是关着的。第 1 次，把所有编号是 1 的倍数的灯的开关状态改变一次；第 2 次，把所有编号是

2 的倍数的灯的开关状态改变一次；第 3 次，把所有编号是 3 的倍数的灯的开关状态改变一次；以此类推，直到把所有编号是 100 的倍数的灯的开关状态改变一次。请问此时所有开着的灯的编号。

解析

本题中灯是 100 个，因此不可能进行 100 次试验后得到结果。

显然 100 盏小灯是否开着与它们状态改变次数的奇偶性有关。由于初始时所有的灯都是关着的，因此如果改变次数为奇数，则灯是开着的；如果改变次数为偶数，则灯是关着的。

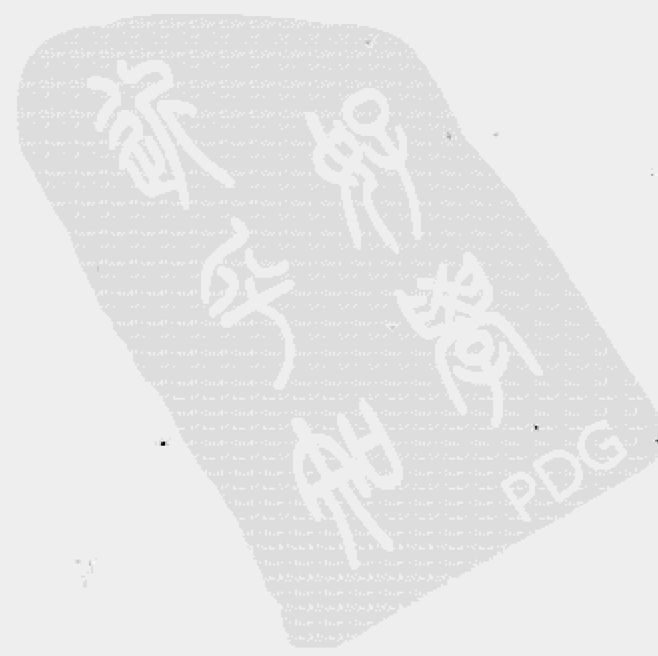
如何判断各个小灯的状态改变次数是奇数还是偶数呢？任何一个数可以分解成 1 和它自身，也有可能分解成另外 2 个较小的数的乘积，由于是两两相乘，因此可知一般都是偶数次数。只有一种情况例外，就是这个数能表示为某个数的二次方。比如：9 可以分解成 3 的二次方，即 3×3。

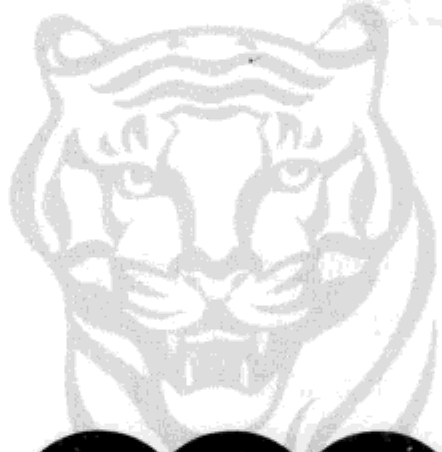
显然，1～100 的数只可能出现 1～10 的乘方，因此所有开着的灯的编号为：

1、4、9、16、25、36、49、64、81、100。

答案

所有开着的灯的编号为：1、4、9、16、25、36、49、64、81、100。





第4篇

职场生涯

第14章 踏上征途

经过了漫长的等待，一轮轮的面试，笔试，然后某天早上你接到人事部的电话，告诉你你已经被录用了，这难道不是人生中最美好的时刻之一吗？在签订了劳动合同，约定了上班日期之后，你终于要成为这个企业的一员了。让我们来看看还有什么需要注意的，让你能有一个出色的开始。

14.1 第一天上班

到目前为止，你应该已经有过很多次第一天加入某个集体的经历了，第一天上小学，第一天上中学，第一天上大学，等等。第一天上班会有什么不同呢？最大的不同就是你是这里唯一的新人。不会有老师带着你参观校园，告诉你洗手间在什么地方，不会有老师教你如何使用打印机，到什么地方喝水，甚至不会有人告诉你如何打电话，去哪儿领电脑，更不用说老板是什么脾气的，说什么样的话会让老板喜欢等，所有这些都等待你去摸索、发掘。人与人之间，员工与员工之间的差距从第一天就开始显现出来了。

如何顺利度过入职初期这段时间呢？以下几个忠告供你借鉴：

1. 签合同的入职时间时，稍微留出一点空隙时间来

给自己一周到两周的准备时间，尽量不要刚刚从上一个工作岗位离开，立刻进入新环境中去，除非公司有确实迫切的要求。

有这段准备时间，你可以请以前的同事们一起吃个饭，联络联络感情。虽然有些人你也许根本就不喜欢，但毕竟这么多年过来了，他们的风格你也都适应了，而且山不转水转，也许哪天大家还有可以互相帮忙的时候，所以彼此留一下联系方式，以供以后联系。

更重要的是，利用这段时间，尽一切可能了解公司的文化、理念、产品线、管理层，以及你所要加入的项目组的研究方向，所用的语言、工具等。最理想的情况是你要是有一个熟悉的朋友在公司里，通过他了解一下公司的情况以及你所在的部门和项目组的情况，所得到的信息就会更加准确可靠。

你还可以利用这段时间准备一下第一周要穿的衣服，理个合适的发型。第一周你最好就穿

古板一些保守一些的服装，不要穿奇装异服，也不要衣冠不整。你总不希望给老板留下的第一印象是你比他穿的还高档吧？然后注意观察你周围的人穿什么样的服装，以后你就可以做相应的调整了。

研究一下从家到公司该怎么走，开车的话会不会堵车，特别是上下班高峰期大概要用多少时间，如果是乘公共交通，看一看有几种可能的路线，需要倒几趟车，车厢里人群的拥挤程度等，都是你要考虑的要素。最好实地考察一下，地图上的数据不一定完全符合实际情况，别到时候发现该坐的公交车取消了就麻烦了。

2．保持好心情

时光转眼飞逝，第一天很快就来到了。跟你去面试一样，上班第一天，穿戴整齐，刷牙洗脸，保持一份自信友好的好心情去上班。为什么第一天上班很重要呢？尽管你在面试的时候可能已经见过一些上级或者同事，但今天你会见到更多的人，他们以后都将和你有工作上的联系，所以给每个人都留下一个好印象对你以后的职业生涯有益无害。

出门尽量早一点，别第一天上班就迟到。到了公司门口，深吸一口气，调整你的心情。美国作家曼狄诺说过："微笑可以换取黄金。"所以能微笑就尽量微笑，不习惯微笑也要保持一个友好的态度，不管你碰到的是谁，不管是前台接待还是扫地的阿姨，都要态度友善。说话要有礼貌，多用请问和谢谢。新人来到，多和人接触，多问问题是应该而且必要的。不要因为要表现你的自信和解决问题的能力，就试图全靠自己解决一切，那样只会使你显得孤傲。

3．别总说你以前的公司怎么怎么样

前面讲面试的章节里我们提到过，不要说你以前的公司和你前任老板的坏话。这一条在这里依然适用，但还要加上一条，也别说他们的好话。有时候你确实会遇上那种情况，就是你原来公司的工作方法比新公司的好，你可能会想使用原来的好的方法，但由于你是初来乍到，不了解情况，每一种方式方法的形成都有它的原因，所以在最初的几周甚至几个月里，你最好按照新公司的要求做事，而不要去试图搞什么改革。最忌讳的话是："我们原来公司不是这么做事的。"你的新同事会对此极为反感，他们肯定会这样看你："这是我们公司，又不是你原来的公司，再说你原来公司要那么好的话，你为什么要离开？"所以闭好你的嘴巴，现在还不是你发表评论的时候。

14.2 入职事项

一般来讲，你入职的时候，人事部门或者你所在部门的秘书会给你一份资料，告诉你有哪些事情要办，在哪里办，找谁办，等等，你只需要按资料上要求的在规定时间内办好就可以了。

接下来就是培训。几乎所有大公司都会安排专门的入职培训，时间长短不等，短则几天，长的会有好几周。公司的培训主要是介绍公司的历史、文化，以及各项规章制度包括保密制度、差旅制度和报销制度等。小公司可能不会安排这样正规的培训，但也会有相关资料，如果没有

的话，就多向老员工请教，主要还是通过自身的自觉来多了解这方面情况，以便尽快融入公司的氛围当中，不必事事求人。

除了公司培训之外，招聘你的部门还会安排部门培训，以及岗位培训，这些培训主要是技术培训和专用流程的培训，是与你每日工作最密切相关的部分。

同时，这里也有一份备忘录，你可以按照这些事项逐一检查，确保没有遗漏的。

（1）签劳动合同。

（2）体检。

（3）转档案。

（4）转公积金。

（5）领门卡及办公室钥匙，一般是在保安处领取。

（6）领工位及工位文件箱钥匙，一些基本的办公用品，比如签字笔、曲别针、订书器和胶棒等。

（7）领电话，向同事问清楚外线怎么拨，国内长途怎么打，国际长途怎么打，三方通话及电话会议怎么开，费用有否限制。

（8）咨询清楚基本办公电器的用法。离你最近的打印机在哪里，离你最近的复印机在哪里，有没有密码，传真机在哪里，传真号码是多少。现在就应该了解清楚这些，否则也许未来几周后某个深夜你在加班时需要复印却复印不了，会让你很尴尬。

（9）领电脑，了解清楚电脑的型号和CPU、内存、硬盘的状况，以及是否已经安装了工作必需的软件等。同时和IT部门搞好关系，以后你还会时不时地找他们的。

（10）领上网账号及密码，以及其他工作中要用到的服务器的账号及密码。

（11）领E-mail地址。

（12）向行政部门了解清楚怎么印名片。

（13）工资如何发放，是否有对应银行的银行卡。

（14）搞清楚上下班时间。

（15）中午在哪儿吃饭。

等以上这些事情都办完，你差不多才算是在公司里安顿下来了。

14.3 最初几周

适应新工作并进入角色的时间，每人和每人不同，工种和工种也不同。你可能能很快适应某种工作，但对另外一种工作却要花很久才能上路，而有的人则似乎不论干什么都能很快上手。不管别人怎样，你所要做的就是尽你所能地用一切你知道的最好的办法完成你手头的工作。这里的一些建议或许会对你有所帮助。

（1）多问问题。你是新来的人，所以在做你不熟悉的事情之前一定要多问问周围的同事，

省得到时候做的不对还得从头再来。

（2）保持微笑，多交朋友。尽快熟悉并适应你周围的环境，记住每个人的名字，多和人打招呼。

（3）午饭是非常重要的社交时间，选择和谁以及哪个团体经常出去吃午饭，可能会影响到你日后的工作成绩乃至升职加薪等，不可不慎重。

（4）搞清楚哪些事情是你必须要做的，哪些事则不在你的业务范围之内。你有时候会遇到某个不是你老板的人过来对你说，把这个事情做一下。这时候不要急着动手做，而要分辨清楚这是不是该你做的。

（5）遇到流言蜚语，听着就行了，不要插话，更不要四处传播，因为你初来乍到，还不清楚人与人之间都是什么关系，通过倾听可以让你熟悉环境，但如果四处散播流言，弄不好会给自己惹来麻烦。

（6）切记对任何人都不可抱怨你的现任老板，不论他对你表现得有多友好，甚至他会主动说几句老板的坏话，引发你的赞同等。如果你躲不开，则以沉默应对。

（7）也不要说其他同事或者以前的老板及同事的坏话。

（8）继续保持早到公司的习惯，至少坚持 3 个月。

（9）不要一到下班点就立刻收拾东西准备走人，观察观察你周围的人，看看别人怎么做，你不要当那第一个冲出大门的人。

（10）保持积极乐观的态度。一份新的工作是一个新开始，肯定需要时间来适应。

14.4 蘑菇管理定律

最后，作为新人，有必要了解一下什么是蘑菇管理定律。

“蘑菇管理”指的是组织或个人对待新进者的一种管理心态。因为初学者常常被置于阴暗的角落，不受重视的部门，只是做一些打杂跑腿的工作，有时还会受到无端的批评、指责、代人受过，组织或个人任其自生自灭，初学者得不到必要的指导和提携，这种情况与蘑菇的生长情景极为相似。

1. 蘑菇管理定律的由来

据称，蘑菇管理定律一词来源于 20 世纪 70 年代一批年轻的电脑程序员的创意。由于当时许多人不理解他们的工作，持怀疑和轻视的态度，所以年轻的电脑程序员就经常自嘲“像蘑菇一样的生活”。电脑程序员之所以如此自嘲，这与蘑菇的生存空间有一定的关系。

蘑菇的生长特性是需要养料和水分的，但同时也要注意避免阳光的直接照射，一般必须在阴暗的角落里培育，过分的曝光会导致过早夭折。而且古代的时候，蘑菇的养料一般为人、兽的排泄物，虽然不洁净但对蘑菇来说是必需品。从两者的关系来看，地点、养料两方面的条件给予了蘑菇的生存空间，但如果要存活蘑菇必须自生自灭，新进学者也是如此。

2．管理中的蘑菇定律

一个组织，一般对新进的人员都是一视同仁，从起薪到工作都不会有大的差别。无论你是多么优秀的人才，在刚开始的时候，都只能从最简单的事情做起。“蘑菇”的经历，对于成长中的年轻人来说，就像蚕茧，是羽化前必须经历的一步。所以，如何高效率地走过生命的这一段，从中尽可能汲取经验，成熟起来，并树立良好的值得信赖的个人形象，是每个刚入社会的年轻人必须面对的课题。

古人云“吃得苦中苦，方为人上人”、“天降大任于斯人，必先苦其心志，劳其筋骨、饿其体肤”，吃苦受难并非是坏事，特别是刚走向社会步入工作岗位，当上几天 “蘑菇”，能够消除很多不切实际的幻想，也能够对形形色色的人与事物有更深的了解，为今后的发展打下坚实的基础。当“蘑菇”的经历对于成长中的年轻人来说犹如破茧成蝶，如果承受不起这些磨难就永远不会成为展翅的蝴蝶。如果能够平和地走过生命的这一“蘑菇”阶段，就能够汲取经验，尽快成熟起来。当然，如果当“蘑菇”时间过长，有可能成为众人眼中的无能者，自己也会渐渐认同这个角色。

3．从打杂到惠普 CEO

卡莉•费奥丽娜从斯坦福大学法学院毕业后，第一份工作是在一家地产经纪公司做接线员，她每天的工作就是接电话、打字、复印、整理文件。尽管父母和朋友都表示支持她的选择，但很明显这并不是一个斯坦福毕业生应有的本分。她毫无怨言，在简单的工作中积极学习。一次偶然的机会，几个经纪人问她是否还愿意干点别的什么，于是她得到了一次撰写文稿的机会，就是这一次，她的人生从此改变，这位卡莉•费奥丽娜成为了惠普公司前 CEO，被尊称为世界第一女 CEO。

一个组织，一般对新进人员都一视同仁，无论你是多么优秀的人才，都只能从最简单的事情做起。“蘑菇”的经历，对于成长中的年轻人来说，就像蚕茧，是羽化前必须经历的一步。

如今就业形势非常紧迫，刚出校门的毕业生由于没有从业经历，很难找到满意的工作，于是有些人选择了先就业后择业的道路。在社会上工作和在学校里生活有天壤之别，首先需要的就是磨去棱角适应社会，把年轻人的傲气和知识分子的清高去掉，摆正心态，放低姿态。这些社会新人如果明白蘑菇管理的道理，就能从最简单最单调的事情中学习，努力做好每一件小事，多干活少抱怨，更快进入社会角色，赢得前辈们的认同和信任，从而较早地结束蘑菇时期，进入真正能发挥才干的领域。

以下是几条对处于蘑菇期的年轻人的忠告。

- 初出茅庐不要抱太大希望，当上几天“蘑菇”，能够消除我们很多不切实际的幻想，让我们更加接近现实，看问题也更加实际。
- 耐心等待出头机会，千万别期望环境来适应你，做好单调的工作，才有机会干一番真正的事业。
- 争取养分，茁壮成长，要有效的从做蘑菇的日子中汲取经验，令心智成熟。

总之，蘑菇管理是一种特殊状态下的临时管理方式，被管理者一定要诚心领会，早经历早受益。

第15章 渐入佳境

经过了入职培训，度过试用期，你已经对公司的事务逐渐了解，甚至已经开始在项目中承担一部分工作。除了具体的项目任务之外，每天的日常工作会占到 20%～50%左右的时间，这部分日常事务性工作的重要不但毫不亚于项目工作，在某种程度上来说甚至比项目工作还重要。如何把日常工作做好，会直接影响到你在公司里的地位。

15.1 从依赖走向独立

新员工从入职开始到真正成长为一个能够独当一面，能够被老板信任的人的过程，就是一个从依赖走向独立的过程。

当我回想以往职业生涯中对我最重要、帮助最大的一本书时，首先想到的就是史蒂芬·柯维博士的《高效能人士的七个习惯》。这本书的英文版首次发行是 1989 年，随后被翻译为 38 种语言，印刷了超过 1 500 万册，被列为最具影响力的 10 大管理类书籍之一。实际上，柯维在这本书里所谈的 7 个习惯绝不仅仅关乎管理，而是可以指导你一生中的方方面面，包括事业、家庭、朋友等人生态度。

柯维博士把这 7 个习惯分为 3 组，第 1 组的 3 个习惯主要是个人自我管理的习惯，按照顺序分别是，积极主动、以终为始和要事第一，如果把这 3 个习惯掌握好并能切实应用，你就可以从工作或生活当中的依赖别人变为独立自主；第 2 组的 3 个习惯主要是讲团队合作和人际关系，按照顺序分别是，双赢思维、知彼解己和统合综效，掌握好这 3 个习惯，你在工作中就不仅是独立自主，并且可以和团队成员一起走向共同胜利；第 3 组只有一个习惯，就是不断更新，它指导你如何巩固前 6 个习惯，从一个成功走向下一个成功。

本章的后面几个小章节都是谈一些办公室事务的具体操作细节，但万变不离其宗，所有这些细节归根结底还是来源于你的处理事务的态度，所以在这里，我们先了解和学习一下高效能人士的 7 个习惯中的前 3 个。

15.1.1 习惯一：积极主动

7 个习惯里的第一个就是积极主动，积极主动是一切其他习惯的基础，也往往是最难做到的一个习惯。

1．负责任的态度是积极主动的第一步

2001 年南斯拉夫足球教练米卢蒂诺维奇来到中国执教中国足球队并成功地把中国队带入世界杯，他提出的一句名言是“Attibude is everything（态度决定一切）”。积极主动其实就是一种态度，如果没有积极主动的态度，则不论其他习惯再好也不可能达到目标。你想不想做这件事情？想不想把这件事情做好？只有具有了强烈的想做事，想把事情做好的愿望，才会开始认真着手准备，并想方设法把它完成。

在威斯敏斯特教堂地下室里，一位英国圣公会主教的墓碑上写着这样的一段话：

“当我年轻自由的时候，我的想象力没有任何局限，我梦想改变这个世界。当我渐渐成熟明智的时候，我发现这个世界是不可能改变的，于是我将眼光放得短浅了一些，那就只改变我的国家吧！但我的国家似乎也是我无法改变的。当我到了迟暮之年，抱着最后一丝努力的希望，我决定只改变我的家庭、我亲近的人。但是，唉！他们根本不接受改变。现在在我临终之际，我才突然意识到，如果起初我只改变自己，接着我就可以依次改变我的家人；然后，在他们的激发和鼓励下，我也许就能改变我的国家。再接下来，谁又知道呢，也许我连整个世界都可以改变。”

每个人都会有很多梦想，但如何实现这些梦想，则必须从自身做起。一个人首先要对自己负责任，然后才有可能对家庭负责，对社会负责。很难想象一个连对自己都不负责任的人会对家庭负责，对社会负责。中国古代典籍《礼记·大学》中所提到的“正心、修身、齐家、治国、平天下”也无不是在强调一切先要从自身做起。

所以，积极主动这个习惯还包含着强烈的责任感和使命感。一个积极主动的人是能够为自己的过去、现在及未来的行为负责任的人，他的一切行为所依据的是客观原则及价值观，而不是他当时的情绪或外在环境的压力。正如范仲淹在《岳阳楼记》中所说“不以物喜，不以己悲”，这才是一个积极主动的人所应该具备的态度。

2．热情是积极主动的催化剂

美国自然科学家、作家杜利奥曾经说过：“没有什么比失去热忱更使人觉得垂垂老矣。”一个积极主动的人一定是一个充满热情的人。人与人之间只有很小的差异，但这种很小的差异结果却往往造成了巨大的差异。很小的差异就是所具备的心态是积极的还是消极的，巨大的差异就是成功与失败。成功人士的首要标志，就在于他们有热情积极的心态。一个人如果心态积极，乐观地面对人生，乐观地接受挑战和应付麻烦事，那他就成功了一半。

积极主动的人是改变的催生者，他们扬弃被动的受害者角色，从不怨天尤人，而是通过发挥人类 4 项独特的禀赋——自觉、良知、想象力和自主意志来改变自身处境及周边环境，以由内而外的方式来创造改变，积极面对一切。积极主动者的生命和生活是由自己选择和创造的，

同样，不积极主动者的生命和生活从表面上来看是被他人所主宰，但归根结底也是他们自己的选择。所以，与其选择被动接受，不如自己积极开始改变自己的人生。

成功学大师拿破仑·希尔曾讲过这样一个故事。

塞尔玛陪伴丈夫驻扎在一个沙漠的陆军基地里。丈夫奉命到沙漠里去演习，她一个人留在陆军的小铁皮房子里，天气热得受不了，在仙人掌的阴影下也有 51.7℃。她没有人可谈天，身边只有墨西哥人和印第安人，而他们不会说英语。她非常难过，于是就写信给父母，说要丢开一切回家去。她父亲的回信只有两行，这两行信却永远留在她心中，完全改变了她的生活：

“两个人从牢中的铁窗望出去，一个看到泥土，一个却看到了星星。”

塞尔玛一再读这封信，觉得非常惭愧。她决定要在沙漠中找到星星。

塞尔玛开始和当地人交朋友，他们的反应使她非常惊奇，她对他们的纺织、陶器表示兴趣，他们就把最喜欢但舍不得卖给观光客人的纺织品和陶器送给了她。塞尔玛研究那些引人入迷的仙人掌和各种沙漠植物、物态，又学习了有关土拨鼠的知识。她观看沙漠日落，还寻找海螺壳，这些海螺壳是几百万年前这沙漠还是海洋时留下来的……原来难以忍受的环境变成了令人兴奋、流连忘返的奇景。

沙漠没有改变，印第安人也没有改变。是什么使塞尔玛发生了这么大的转变呢？是她的心态，是她对生活的一种热情。重燃的生活热情使她把原先认为恶劣的情况变为一生中最有意义的冒险。她为发现新世界而兴奋不已，并为此写了一本书，以《快乐的城堡》为书名出版了。她从自己造的牢房里看出去，终于看到了星星。

“一个人如果缺乏热情，那是不可能有所建树的。”作家拉尔夫·爱默生说，“热情像浆糊一样，可让你在艰难困苦的场合里紧紧地粘在这里，坚持到底。它是在别人说你‘不行’时，发自内心的有力声音——‘我行’。”麦当劳的老板克罗克的故事很好地说明了这一点。

克罗克一出生，就与一个本来可以发大财的时代擦肩而过，向西部淘金的运动结束了。而正当他准备上大学时，又迎来了 1931 年的美国经济大萧条。他不得不顺从囊中羞涩的现实，辍学去搞房地产。可房地产生意刚有起色，第二次世界大战又打起来了。人们都只顾逃命，哪有心思买房？于是房价急转直下，克罗克又是竹篮打水一场空。这以后，他到处求职，曾做过急救车司机、钢琴演奏员和搅拌器推销员。但似乎一切都不顺，不幸几乎就没离开过克罗克。

尽管如此，克罗克仍是热情不减，执著追求，毫不气馁。1955 年，在外面闯荡了半辈子的他空手回到了老家。在卖掉了家里的一份小产业后，克罗克开始做生意。这时，他发现迪克·麦当劳和迈克·麦当劳开办的汽车餐厅生意十分红火。经过一段时间的观察，他确认这种行业很有发展前途。当时克罗克已经 52 岁了，对于多数人来说这正是准备退休的年龄，可这位门外汉却决心从头做起，到这家餐厅打工，学做汉堡包。后来，他毫不犹豫地借债 270 万美元买下了麦氏兄弟的餐厅。经过几十年的苦心经营，麦当劳现在已经成为全球最大的以汉堡包为主食的快餐公司，在国内外拥有 7 万多家连锁分店，年销售额高达近 200 亿美元。克罗克也被誉为“汉堡包之王”。

生活处处有磨难，关键在于你用怎样的心态去面对。拿破仑·希尔说：“一个人能否成功，

关键在于他的心态。”成功人士与失败人士的差别在于成功人士有积极的心态和高昂的热情。正因为克罗克拥有热情的心态，才使得命运瑰丽多彩。的确，心态是真正的主人，你的心态决定了谁是坐骑，谁是骑师。积极的心态使你充满力量，去获得财富、成功、幸福和健康，攀登到人生的顶峰。而消极的心态却把一切让你的生活里有意义的东西剥夺得一干二净，在人生的整个航程中处于一种长期的晕船状态，对将来总感到失望。要记住，宽广成功之路总是为那些自强不息、审时度势的人准备的。

3．信心是积极主动的有力支撑

在很多时候，强者不一定是胜利者，但胜利迟早都属于有信心的人。可以说，信心决定成败。如果你只接受最好的，你最后得到的往往也是最好的，只要你有信心。

有一个人经常出差，经常买不到对号入座的车票。可是无论长途短途，无论车上多挤，他总能找到座位。

他的办法其实很简单，就是耐心地一节车厢一节车厢地找过去。这个办法听上去似乎并不高明，但却很管用。每次，他都做好了从第一节车厢走到最后一节车厢的准备，可是每次他都用不着走到最后就会发现空位。他说，这是因为像他这样锲而不舍找座位的乘客实在不多。经常是在他落座的车厢里尚余若干座位，而在其他车厢的过道和车厢接头处，居然人满为患。

他说，大多数乘客轻易就被一两节车厢拥挤的表面现象迷惑了，不大细想在数十次停靠之中，从火车十几个车门上上下下的流动中蕴藏着不少提供座位的机遇，即使想到了，他们也没有那一份寻找的耐心。眼前一方小小立足之地很容易让大多数人满足，为了一两个座位背负着行囊挤来挤去，有些人也许会觉得不值。他们还担心万一找不到座位，回头连个好好站着的地方也没有了。与生活中一些安于现状不思进取害怕失败的人，永远只能滞留在没有成功的起点上一样，这些不愿主动找座位的乘客大多只能在上车时的落脚之处一直站到下车。

自信、执着、富有远见、勤于实践，会让你握有一张人生之旅的永远坐票。

有世界第一CEO之称的前通用电气公司董事长杰克•韦尔奇出生在一个典型的美国中产阶级家庭。父母结婚16年后才有了这个独生子，父亲为波士顿与缅因铁路公司工作，早出晚归，所以培养孩子的任务就落在了母亲的肩上。

与其他独生子女母亲不太一样的是，她对儿子的关心更主要体现在提升他的能力和意志上。杰克非常尊敬乃至崇拜母亲：“她是一位非常有权威性的母亲，总是让我觉得自己什么都能干，是我母亲训练了我，要我学习独立。每次当我的行为稍有越轨，她就一鞭子把我抽回来，但通常都是正面而且有建设性的，还能促使我振作起来。她向来不说什么多余的话，总是那么坚决，那么积极，那么豪迈。我总是对她心服口服。”

母亲教给杰克3门非常重要的功课：坦率的沟通，面对现实，并且主宰自己的命运，这是母亲始终抱持的理念。日后证明在杰克的管理生涯中，这种禀赋被发挥得淋漓尽致。

要掌握自己的命运就必须树立自信。尽管杰克到了成年还略带口吃，可母亲说这算不了什么缺陷，只不过是想的比说的快些罢了。结果，略带口吃的毛病并没有阻碍杰克的发展，而实际上注意到这个弱点的人大都对他产生了某种敬意。美国全国广播公司新闻部总裁迈克尔对他

十分敬佩，甚至开玩笑地说："他真有力量，真有效率，我恨不得自己也口吃。"

在韦尔奇看来，我们所经历的一切都会成为我们信心建立的基石。当你被选为一支球队的队长时，当你在球场中选队员时，你就掌握了这支队伍。然后事情就这么发生了，渐渐地，你会习惯这些经验，而且人们也会信任你，给予你善意的回应。

韦尔奇的中学成绩应该是可以保证他进入美国最好的大学，但因种种原因而事与愿违，只进了麻州大学。开始他感到非常沮丧，但进入大学之后，沮丧就变成了庆幸。

"如果当时我选择了麻省理工学院，那我就会被昔日的伙伴们打压，永远没有出头的一天，然而这所较小的州立大学，让我获得了许多自信。我非常相信一个人所经历的一切，都会成为建立自信的基石，包括母亲的支持、运动、上学、取得学位。"事实证明杰克是麻州大学最顶尖的学生，看来没有到麻省理工学院是对的。担任杰克大学班主任的威廉当时也看出了杰克成功的初期征兆"是他的双眼，他总是很自信，他痛恨失败，即使在足球比赛中也一样"。

"自信"在日后成为了通用电气的核心价值观之一。杰克说："所有的管理都是围绕'自信'展开的。"韦尔奇 1981 年成为 GE 历史上最年轻的 CEO。17 年来，公司的市场价值从原来的 120 亿美元，升到了如今的超过 4 000 亿美元，而且一直被公认为是管理最优秀和最受推崇的公司之一。

对事业怀有信心，相信自己，乃是获得成功不可或缺的前提。当然其他因素也非常重要，但最基本的条件，是激励自己达到所希望的目标的积极态度。怀有信念的人是了不起的。他们遇事不畏缩，也不恐惧，就是稍感不安，最后也都能自我超越。他们健壮而充满活力，能解决任何问题，凡事全力以赴，最终成为伟大的胜利者。他们都有一个神奇的座右铭，那就是"信念"。

15.1.2 习惯二：以终为始

有了习惯一积极主动的态度之后，习惯二的以终为始和习惯三的要事第一都是在讲方法论，以终为始强调目标的重要性，而要事第一则更多地关注做事的顺序。

我们人类所创造的所有事物都经过两次的创造，先是在脑海里酝酿，其次才是实质的创造。个人、家庭、团队和组织在做任何计划时，都应当先拟定出前景和目标，并据此塑造未来，全心投注于自己最重视的原则、价值观、关系及目标之上。

1. 要有大局观

公元前 204 年，汉王刘邦手下大将韩信带兵攻打赵王歇，出井陉口后派一万军队故意背靠河水排列阵势来引诱赵军。到了天明，韩信率军发动进攻，双方展开激战。不一会儿，汉军假意败回水边阵地，赵军全部离开营地，前来追击。这时，韩信命令主力部队出击，背水结阵的士兵因为没有退路，也回身猛扑敌军。赵军无法取胜，正要回营，忽然营中已插遍了汉军旗帜，于是四散奔逃。汉军乘胜追击，打了一个大胜仗。在庆祝胜利的时候，将领们问韩信："兵法上说，列阵可以背靠山，前面可以临水泽，现在您让我们背靠水排阵，还说打败赵军再饱饱地吃一顿，我们当时不相信，然而竟然取胜了，这是一种什么策略呢？"韩信笑着说："这也是兵法上有的，只是你们没有注意到罢了。兵法上不是说'陷之死地而后生，置之亡地而后存'吗？

如果是有退路的地方，士兵都逃散了，怎么能让他们拼命呢！”

400多年后的公元228年，诸葛亮一出祁山时任命马谡防守街亭。临行前，诸葛亮再三嘱咐马谡：“街亭虽小，关系重大。它是通往汉中的咽喉。如果失掉街亭，我军必败。”并具体指示让他“靠山近水安营扎寨，谨慎小心，不得有误”。马谡到达街亭后，不按诸葛亮的指令依山傍水部署兵力，却骄傲轻敌，自作主张地想将大军部署在远离水源的街亭山上，当王平反对时，他辩解说：“居高临下，势如破竹，置之死地而后生，这是兵家常识，我将大军布于山上，使之绝无反顾，这正是致胜之秘诀。”魏明帝曹睿得知蜀将马谡占领街亭，立即派骁勇善战，曾多次与蜀军交锋蜀大军张部领兵抗击，张部进军街亭，侦察到马谡舍水上山，心中大喜，立即挥兵切断水源，掐断粮道，将马谡部围困于山上，然后纵火烧山。蜀军饥渴难忍，军心涣散，不战自乱。张命令乘势进攻，蜀军大败。马谡失守街亭，战局骤变，迫使诸葛亮退回汉中。

同样是“置之死地”，为什么韩信能获胜，而马谡却落了个全军覆没呢？因为马谡的思维方式就明显不是以终为始式的。守街亭的最主要目的是什么？诸葛亮并没有命令他在街亭全歼或者重创魏军，只是要他守住街亭这个隘口，挡住敌人即可。可是马谡却狂妄自大，妄图用自己薄弱的兵力战胜强大的敌人，他以一己之浅见破坏了全军部署，导致自己全军覆没，并且彻底破坏了诸葛亮的全盘计划。

以终为始首先是要有大局观，始终知道自己的终极目标是什么，在行动过程中不断纠正自己的方向，在大方向上不偏离目标太远。这里要搞清楚什么是终极目标，终极目标是否正确？如果终极目标正确，再把终极目标分解为小目标，并仔细检讨这些小目标是否确实必要，是否确实会导向终极目标，找出关键路径。这样做事情的时候不会拘泥于非关键路径上的个别小目标是否实现，关心点只在于距离终极目标还有多远，这样行动起来才不会迷失方向。

2．按照目标制订计划

美国行为科学家艾得•布利斯提出：“用较多的时间为一次工作事前计划，做这项工作所用的总时间就会减少。”

美国的几个心理学家曾做过这样一个实验，把学生分成3组进行不同方式的投篮技巧训练。第1组学生在20天内每天练习实际投篮，把第一天和最后一天的成绩记录下来。第2组学生也记录下第一天和最后一天的成绩，但在此期间不做任何练习。第3组学生记录下第一天的成绩，然后每天花20分钟做想象中的投篮，如果投篮不中时，他们便在想象中做出相应的纠正。实验结果表明，第2组没有丝毫长进；第1组进球增加了24%；第3组进球增加了26%。由此，他们得出结论，行动前进行头脑热身，构想要做之事的每个细节，梳理心路，然后把它深深铭刻在脑海中，当你行动的时候，你就会得心应手。

这个实验告诉我们的，就是计划的重要性。做事没有计划，行动起来就必然会是一盘散沙。只有事前拟订好了行动的计划，梳理通畅了做事的步骤，做起事来才会应付自如。好的规划是成功的开始。

3．避免目标置换效应

强调以终为始的重要性就是为了要避免出现“目标置换效应”。所谓“目标置换效应”就是

说对于工作如何完成的关切，致使渐渐地让方法、技巧、程序的问题占据了一个人的心思，反而忘了整个目标的追求。换言之，“工作如何完成”逐渐代替了“工作完成了没有”。据美国管理学家约翰•卡那做过的一项调查显示，在所有影响目标达成的因素中，“目标置换”因素占了67%以上。

应该说，在实施目标的过程中，总是有或多或少、或直接或间接、或潜在或明显的因素干扰和障碍着目标的达成。若从“目标置换”的角度上分析大致有两类。

一类是客观上的，具体表现：1 是目标不明确，对目标的完成在数量、质量、时限和标准等方面规定得比较笼统，使目标缺乏方向感；2 是目标或过高，超出了人们的实现能力，或过低，激不起人们的兴趣，难以起到真正的激励作用；3 是目标实现周期长，随着时间的推移和环境的改变，达成目标的现实条件逐渐丧失；4 是出现了不可预料的事件，分散了目标实施者的精力和注意力。

另一类是主观上的，具体表现：1 是目标实施者对目标的理解出现偏差，无意中使自己的行为偏离了既定目标；2 是因循守旧，思维僵化，不敢变通和创新，生怕“越雷池一步”；3 是缺乏团队精神，难以得到上级或同事的有力配合与支持；4 是实际操作能力低，缺乏达成目标的相关方法与手段；5 是缺乏信息意识，不能积极了解目标的实施进展情况并通过负反馈来及时调整和纠偏。

“目标置换”是实施目标过程中一种“偏差”行为和“错位”现象，若不及时发现和矫正，必然影响目标的达成。那么，如何避免“目标置换”现象的发生呢？

（1）要建立动态的目标体系。

一方面，在目标的设立、分解、定责等过程中要使诸目标间形成一个相互支持、关联、照应的有机整体，总目标要成为分目标的“标杆”，各分目标要自觉地、主动地、经常地向总目标“看齐”；另一方面，要使一些目标具有相应的弹性，以便在出现新情况和新问题时能根据具体情况进行调整与完善。

（2）要实施全方位的目标管理。

主要应抓好以下各环节，其 1，目标应建立在上下达成共识的基础之上，不能人为地“压任务”、“下指标”，最好由上下级协商确定，否则上下级往往会在目标问题上形成“博弈”关系。管理实践证明，上级亲自参与下级目标的制定过程，生产率的平均改进幅度可达 56%，反之则仅达 6%。其 2，目标要适度，过低则人忽其易，太高则人畏其难，应以“跳一跳够得着”为最好。其 3，目标间要建立其支持关系，以便于目标承担者之间的积极互动。其 4，为目标实施者创造必要的实施条件（设备、技术、培训、资金等）。其 5，赋予目标实施者充足的权力，并使目标与权力、责任和利益挂钩，以更好体现“目标激励”。其 6，调整式改革一些有碍于达成目标的规章、制度。其 7，鼓励目标承担者在权限以内大胆创新、独立自主。

（3）要解决“信息不对称”问题。

从某种意义上讲，达成目标的过程，也是处理信息的过程，能否拥有充分、及时、准确、优质的信息对达成目标起着至关重要的作用，否则就会因信息不对称而导致目标实施者“逆向

选择”行为和“道德风险”现象的发生。因此，一方面，管理者要为目标责任人提供必要的信息支持，并与其经常进行信息交流与沟通，帮助其正确分析形势、研究问题和解决问题；同时对目标责任者所采取的一些行之有效的新方法和取得的新进展、新成果要及时给予肯定和鼓励。另一方面，要定期对目标的进展情况进行检查和考评，并及时将检查和考评的结果反馈给实施者，因为知道干得怎样的人，往往也最易知道怎样干得更好。

15.1.3 习惯三：要事第一

随着你对公司里的业务越来越熟悉，你会发现你手头的事情越来越多，有些是老板交代下来必须今天完成的，有些是项目经理给安排的任务，一大堆邮件还等待回复，同时还有客户要你去参加会议，面对这么多同时进行的任务，你是否感觉疲于奔命，力不从心？如果是这样，那么这第 3 个习惯会对你有很大帮助。

1．事务的 4 种类型

要事第一的原则就是永远把最重要的事情放在第一要完成的列表中，而不是把最紧急的事情放在前面。事务可以分为 4 大类，如图 15.1 所示。

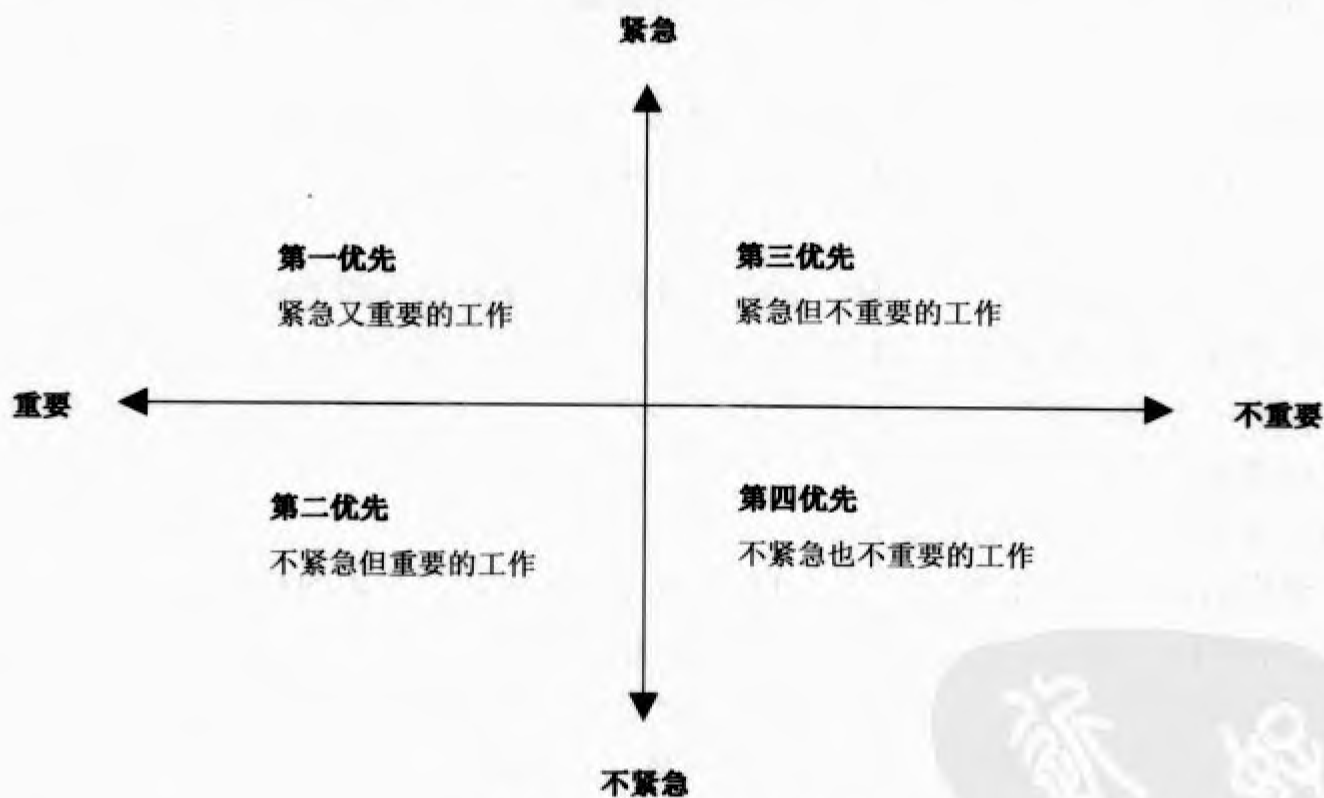


图 15.1 事务的 4 种类型

第 1 类，紧急又重要的工作。毫无疑问，这一类工作必须先做。比如，仓库失火了，如果不及时救火，火势会蔓延，因此非常紧急，同时仓库里有重要物资，不救不行，因此非常重要，类似这样的工作就属于第 1 类。

第 2 类，不紧急但重要的工作。这一类问题是整个时间管理的精髓所在，如果把这一类工作做好了，你就彻底掌握了时间管理。为什么要先做这一类工作？因为如果你把这些不紧急但

重要的工作做好了，会极大地提高你的工作效率。我们想一下，为什么会出现上面第 1 类里那些紧急又重要的工作？很大程度上是因为我们没有把这第 2 类工作做好，才导致了那些本来应该是第 2 类的工作结果变成了第 1 类的工作，结果把我们每天的生活弄得非常紧张。还以上例来说，既然仓库如此重要，那我们是不是应该及早做好消防准备，而不是等到火烧眉毛了才去挽救？消防工作就是这种不紧急但很重要的工作，如果你把消防工作做好了，火烧不起来，那你就不必去为救火这种第 1 类工作操心，工作生活自然就会轻松许多。

第 3 类，紧急但不重要的工作。这是一类什么样的工作呢？这类工作有时候往往会占据我们大量时间，因为它紧急啊，但我们往往在做这类事情之前忘了问个为什么，忘了问它是不是真的很重要？它对我来说是真的重要吗，还是只是对别人重要，对我其实并不那么重要？如果你能够分辨清楚，那你也就找到了答案。我们当然鼓励乐于助人，但当我们手头既有第 2 类工作也有第 3 类工作时，面临这样的情况，一个高效能的人会理所当然地选择先做第 2 类工作，而把第 3 类工作交给别人去完成。

第 4 类，既不紧急也不重要的工作。这类其实都不算是工作了，基本上属于娱乐的范畴。我们每个人的工作生活压力很重，娱乐是必不可少的，所以第 4 类工作有其存在的合理性。但什么时候做，排在哪个顺序，显然我们不能够把它排在其他类工作的前面。当然，如果你处于长期压力之下，娱乐活动对你来说已经变成了既紧急又重要或者不紧急但重要的工作，那它也就不再属于这一范畴，你应该认真地把它当做第 1 类或者第 2 类工作来看待，给自己好好安排一个假期吧。

要事第一即实质的创造，是梦想（你的目标、愿景、价值观及要事处理顺序）的组织与实践。次要的事不必摆在第一，要事也不能放在第二。无论迫切性如何，个人与组织均针对要事而来，重点是，把要事放在第一顺位。

2．避免蔡戈尼效应

蔡戈尼效应是指，人们天生有一种办事有始有终的驱动力，人们之所以会忘记已完成的工作，是因为欲完成的动机已经得到满足，如果工作尚未完成，这同一动机便使他对此留下深刻印象。

1927 年，心理学家蔡戈尼做了一个实验，将受试者分为甲乙两组，同时演算相同的数学题。其间让甲组顺利演算完毕，而乙组演算中途，突然下令停止。然后让两组分别回忆演算的题目，乙组明显优于甲组。这种未完成的不爽深刻地留存于乙组人的记忆中，久搁不下。而那些已完成的人，“完成欲”得到了满足，便轻松地忘记了任务。

这种解答未遂的问题，深刻地留存记忆中的心态就叫蔡戈尼效应。

关于这种心理，曾有过这样一段佳话，一位爱睡懒觉的大作曲家的妻子为使丈夫起床，便在钢琴上弹出一组乐句的头 3 个和弦。作曲家听了之后，辗转反侧，终于不得不爬起来，弹完最后一个和弦。趋合心理逼使他在钢琴上完成他地脑中早已完成的乐句。

很多人有与生俱来的完成欲。要做的事一日不完结，一日不得解脱。蔡戈尼效应使人走入两个极端，一个是过分强迫，面对任务非得一气呵成，不完成便死抓着不放手，甚至偏执

地将其他任何人或事物置身事外；另一端是驱动力过弱，做任何事都拖沓啰嗦，时常半途而废，总是不把一件事情完全完成后再转移目标，永远无法彻底地完成一件事情。

举例来说，倘若信才写了一半，圆珠笔突然笔用光了，你是随手拿起另一支笔继续写下去，还是四处找一支颜色相同的笔，在寻找时思路又转到别的方面去了，而丢下没写完的信不理？或者，你是否被一本间谍小说迷住了，哪怕明天早上有一个重要会议，也要读到凌晨4点仍不释卷？之所以出现这种现象，是因为人们天生有一种办事有始有终的驱动力。请试画一个圆圈，在最后留下一个小缺口，现在请你再看它一眼，你的心思会倾向于要把这个圆完成。

对大多数人来说，蔡戈尼效应是推动我们完成工作的重要驱动力。但是有些人会走向极端，要么因为拖拉永远也完不成一件事，要么非得一口气把事做完不可。这两种人都需要调整他们的完成驱动力。

对于驱动力过弱的人来说，做事半途而废也许只是因为害怕失败。他永远不去把一件作品完成，以避免受到批评，同样，只愿永远当学生而不想毕业的人，也许是因为这样就可不必到社会上去工作，也可能由于他在潜意识中就不相信自己会成功，于是害怕成功，因此也就下意识地逃避成功。泰克医生为有这样心理的人提出一个解决的方法，他说："如果你精力集中的时间限度是10分钟，而工作要一小时才能做完，那么，你的脑筋一开始散漫你就要停止工作，然后用3分钟的时间活动筋骨，例如跳几下，去倒一杯水，或是做些静力锻炼的肌肉运动，活动过后，再把另一个10分钟花在工作上。"

一个从不把工作做完的人，至少能够扩展自己的生活，而且可能生活得丰富多彩，但是一个非把每件事都做完不可的人，驱动力过强，可能导致生活没有规律、太紧张、太狭窄。只有减弱过强的驱动力，才可以使人一面做事一面享受人生乐趣。在工作方面，不做完不罢休的人可能是个工作狂。如果把这种态度缓和一下，不仅使你能在周末离开办公室，还有时间去应付因工作狂带来的问题，自我怀疑，感觉自己能力不够或不能应付紧张等。非做完不可的人为了避免半途而废，很可能冒把自己封死在一份没有前途的工作上的危险。兴趣一旦变成狂热，就可能是一个警告信号，表示过分强烈的完成驱动力正在渐渐主宰你的消遣活动。有人会强迫自己织完一件毛衣，结果虽然不喜欢那件毛衣，但却觉得非穿它不可。对于某些事，不应该害怕半途而废。

怎样才能把脱缰之马一般的完成驱动力抑制住呢？

第一，在看事物的时候运用自己的价值观标准，如果我们发现一个工作计划不值得做，那么我们就勇敢地放弃。

第二，编制一个时间表，把必须做的事以及要在的时间都写下来。努力培养出一种较符合实际的意识，把期限定在要求办妥的时间以前。如果有笔账必须在12月1日缴付，那就预订在11月25日付出。

第三，一点一滴地强化意志力，我们可以先从一件小事来训练自己，比如强迫自己在洗碗槽里留下几只碟子不洗，看一本书的时候，尝试停一下，想想自己是否在浪费时间和精力，如果是的，要不要继续看下去？

3．学会管理你的时间

在理解了事务的 4 种类型之后，时间管理就不再是一件难事。最简单的时间管理方法是，每天上班后的第一件事情，先不急着马上开始工作，而是把这一天所需要的工作先列出来，写在纸上，按重要度进行排序，重要的先做，不重要的后做。依此类推，每周开始之前，也可以列这样的一张周计划表，每个月开始之前列月计划表。

另外，如果觉得纸和笔不方便的话，还有很多时间管理的软件也可以帮助你做这些事情，原理都是一样的。

不管用什么方法进行时间管理，最重要的是能够坚持下去，持之以恒，养成一种习惯。如果做了两天，出于惰性，坚持不下去，或者给自己找借口说每天工作太忙了，没有时间做时间管理，则恰好落入了时间管理失败者的轮回，永远无法自拔。所以，最好的方法是，从现在开始，坚持到底，你一定能够从中受益无穷。

15.2 建立目标

在第 1 章里我们就谈到要建立长远的职业规划，现在已经入职的你，依然不应该忘记当初自己的计划，现在要做的只是把这个长远规划和你目前的职位结合起来，进行具体的计划。

每年年末年初的时候，总结一下自己在上一年里做了哪些工作，学到了哪些知识技能，明年准备如何做，自己还有哪些不足，需要学习什么新的知识，这些知识以及经验如何与你的长远职业规划相匹配。

有人说，制订计划是老板的事情，员工只要负责执行就行了。实际情况不是这样的，在很多大企业里，员工的工作计划甚至部门计划都是需要员工参与设计的，如果没有员工的参与而由老板擅自指定的计划往往最后变得无法实现。所以作为员工的你也有必要了解如何制订一个良好的切实可行的计划。

在做年度计划的时候，最常听到一个词就是 SMART，比如我们要制订一个 SMART 计划。SMART 这个在英文里有“聪明”的意思，但把它分解成 5 个独立的字母，每个字母代表了制订一个“聪明”计划里所需要考虑的 5 个方面：

1．S（Specific）：明确性

是指计划要有明确的目标，不是泛泛空谈。什么样的计划是一个明确的计划，可以通过回答这个 6W 的问题来衡量。6W 是指 6 个以 W 开头的单词。

- Who（谁），由谁来完成这个计划？是你自己单独一个，还是需要一个项目组配合？
- What（什么），完成什么事情？
- Where（哪里），在哪儿完成，需不需要出差？
- When（何时），从什么时候开始，到什么时候结束？
- Which（哪个），进一步明确 What，这个计划的具体需求是什么？

- Why（为什么），为什么要做这件事情？这件事情能够带来哪些价值？

比如说，我准备开发软件，或者我准备开发手机上的软件，这些都不是明确的目标。一个明确的目标必须能够表述清楚你所要做的事情的具体方向甚至详细功能。作为一个具备了S特性的目标，至少要描述到这样才够清楚，我准备开发手机上的三维实时对战游戏。

2．M（Mesurable）：可度量性

一个S目标并不天然具备M的特性，比如上文说到的开发手机游戏的目标并没有指明开发多少个这样的游戏，所以还需要用M特性来约束它，加上了M特性的目标就变成了，我准备开发3款手机上的三维实时对战游戏。

3．A（Attainable）：可实现性

过高或过低的计划都不符合A特性。A特性是和S特性和M特性都有相关性的。如果S特性说我准备登上月球，显然是不切实际的，不符合A特性；或者M特性说我准备开发1 000款手机游戏，这也不现实，所以在这一步你需要用A特性去检验你前面根据S和M特性制订的计划是否具有可实现性。同时，也要考虑所制订的计划是否要求太低了，比如去年每年都能开发5款游戏，那么今年只制订3款游戏的计划则属于要求过低的情况。一般原则是宁高勿低，只要不是高到不切实际就可以。如果目标比能力还略高的话，会激发人努力去完成它的愿望，这样即使完不成，也会得到一个较好的结果；而如果目标过低，很容易实现的话，会使人产生惰性。

美国行为学家吉格勒认为：“设定一个高目标就等于已经达到了目标的一部分。”不少人认为天才或成功是先天注定的。但是，世上被称为天才的人，肯定比实际上成就天才事业的人要多得多。为什么？许多人一事无成，就是因为他们缺少雄心勃勃、排除万难、迈向成功的动力，不敢为自己制定一个高远的奋斗目标。不管一个人有多么超群的能力，如果缺少一个认定的高远目标，他将一事无成。

4．R（Relevant）：相关性

也就是说你制订的计划要与你的工作相关，如果你们公司不是搞游戏开发的，那么你前面制订的关于游戏开发的计划就是与R特性违背的，也不可能得到公司的批准。

5．T（Time-based）：时间限制

时间是压力的来源，没有时间压力，很多事情将一拖再拖，最后变得无法实现，所以一个好的计划必须要有时间限制。按上例来说，在10月31日之前完成3款手机上的三维实时对战游戏就是一个不错的目标。

15.3 评价工作表现

在很多大企业里，每到年底的时候，各个单位，各个部门，以至每个人都要写年终总结和次年计划。总结写得好不好，会对年终考评有一定的影响。

15.3.1 年终总结

我们先来看一下年终总结要怎么写。写年终总结和下年计划要注意以下几点。

（1）年终总结不是表彰大会，也不是批判大会，而是一年得失的检讨和经验的总结。

（2）年终总结是个人、部门、公司的指南针。

（3）年度计划不是空想，也不是喊口号，而是建立在公司的远景、目标和发展方向的基础上，结合市场态势、资源整合等而展开的有目标、有方法、有期限、有责任的工作分解与细化。

写总结是必须的也是对自己的一个思考，有几项是必须写的，如已完成的工作，未完成的工作，项目完成情况，项目拖延原因，改进项目方法，去年工作预测对比以及来年工作预测和安排。计划和实际是有差距的，做好计划就可以实现，遇到实际问题也可以按照计划做一些调整。

实际上，年终总结不是到了年终才开始写的，是用一整年的时间在写，这是一个很重要的概念，如果不理解这个概念，那你就不是在写年终总结，而是在编一些东西糊弄你的领导。

这是一个什么概念呢？也就是在一年的工作计划中，需要对下一年的年终总结的形成做一个计划。在一年的计划中，需要定义一个日常性小结或记录，阶段性总结或整理的要求和格式，坚持做下去，到了年底进行一下汇总和综合分析，就可以形成客观、真实的年终总结报告。

因此反过来说，年初计划一定要考虑到阶段小结和记录的要求，将下一年的年终总结的形成方式做到计划之中去。

15.3.2 绩效考评

还记得以前期中考试和期末考试后老师公布成绩时你的心情吗？不管你是否有自信取得好成绩，你都会感到担心和紧张。学校的岁月已经离我们远去，你是否觉得成绩单也已经不复存在？不幸的是，事实并非如此。作为一个有工作的成年人，我们依然必须面对员工绩效评估或者员工考评等这样的评价。名称千变万化，但宗旨无非一个，考核我们一年或者一段时期来的工作成绩。然后，考核成绩会直接影响到我们的升职、提薪，甚至影响到我们是不是还能保住饭碗。所以，即使是成年人，这些“成绩单”也一样会让我们感到不安。

1．搞清楚评估方式

就像小时候参加考试一样，你不一定知道考题，但至少得清楚这次考试考的是什么内容。恐惧来自于无知，如果你知道考的是什么，自然有信心应对。关于绩效考评，你最需要了解的是考评的目的，也就是说企业为什么要采用这种方式来评价员工。一般来讲，考评的目的有以下几个，增强沟通，建立清晰的工作衡量指标，鼓励员工的优良表现，改善不良表现，以及培养企业文化和团队精神等。每个企业的考评方式不同，有些企业采用直接主管考评，有些企业采用 360° 考评。事实上，每种考评方式都各有其优缺点，不存在哪种最好哪种不好之分。对于企业员工来说，不论是哪种方式，你只要事先有所了解，并且在日常工作中及时注意这些方

面，就不会在考评时感到手足无措。

2．准备考评

把今年以来的工作成绩写成文档，列出所有你想在考评时和老板讨论的问题。如果你的工作成绩没有文字记录可供参考的话，需要花点时间仔细想想从上次考评到现在你做了哪些工作，最重要的是，你的这些工作给企业或部门带来了什么样的好处，比如，增加了效益，改善了客户关系或减少了开销等。

3．得了差评怎么办

这是每个人都不想看到的，包括你老板。但是很多公司有规定，一个部门里不可能人人都好，总要有优良中差，而你不幸地就是那个人。首先，要弄清楚，客观公正地说，你是不是表现的不太好？如果确实如此，那就诚恳地向老板请教，如何能让你明年的工作有所改善？你的不足之处通过什么样的方式可以弥补？也许你会因此而争取到难得的培训机会，这反倒成了一件好事。其次，如果你觉得你表现得非常棒，但就是因为老板不喜欢你，或者你以前得罪过他，所以他给你穿小鞋，打击报复，这时候事态就比较复杂了。也许你可以约老板再来一次长谈，把大家心中的过节说清楚，希望以后不再带着包袱上路，或者干脆就另谋高就算了，此处不留人，自有留人处。当然也有人选择去老板的老板或者人事部门那里告状，但告状的结果即使你胜诉了，再留在这里以后还不知道会有什么，所以一般不推荐这种选择。

4．你应该从考评中学到什么

客观来说，应该把每一次考评看做是一次良好的学习机会。因为一年到头来，大家都是忙于工作，难得有这样的机会和老板一起坐下来总结分析一下这一年来工作当中的成绩和不足。你应该抓住这个机会获取一些宝贵的信息，这些信息包括，你的优点和缺点是什么？你老板希望你成长为什么样的人？你可以通过什么样的途径获得提高？等等。

15.4 开会及发言

开会是办公室里进行沟通的必要手段。毫无疑问，开会会占用一部分的工作时间，如果会议开得太多，拖得太长，而又达不到应有的效果，所以有相当多的人不喜欢开会，但开会又是必不可少的，如何把会开好，开得有效率，这需要会议的组织者和参与者共同努力。

15.4.1 会议主持人

会议的重要性在于其历史性。会议的形式往往重要于内容，会议的准备、会议召开和结束事项，构成了会议的全部，因此作为会议的主持人，必须从会议的准备开始一直到会议的结束始终对会议保持高度的关注。

“机遇只偏爱那些有准备的头脑”对一个人而言，无论做任何事，如果事前没有做好充分的准备，那么等待他的必然是一个注定的失败。对一次会议而言，高效的会议其实在正式宣布开

始之前就开始准备了。作为会议主持人必须要付出比别人多的多的努力。

1. 要不要开会

一次高效会议的第一步准备工作是确定是否真的有开会的必要性。这个确定工作看似多余，实际上必不可少。这一步确定工作的作用相当于一个“筛选器”，它使那些根本不符合发生条件的“会议”提前落选，从而为高效会议的产生把好第一关。

这些情况下不需要开会。

（1）如果还有比开会更好的方法，即如果能够通过一些看似原始的方法达到与开会同样的目的，就无须举行会议，例如写纸条、打电话或发邮件等。这些方式可能在感觉上不如开会那么痛快淋漓，然而却简洁有效，对于一些问题的解决具有经济实用的优势。

（2）如果你是唯一可以做决策的人，那么一定不要浪费自己和他人的时间，召开耗时长久的所谓会议。首先，因为他人的参与可能只能把问题变得更糟；其次，决策既然只能由你做出，就无须聚集他人集体讨论，结果又全盘推翻他人意见，这种做法只会既不利于解决问题，又容易令自己陷入尴尬境地。

（3）会议成本是一个绝对不容忽视的问题，一定要做好会议成本的预算，不能用高昂的会议成本来换取某些会议的发生，因为长此以往公司必然会不堪重负、得不偿失。

这些情况下必须开会。

（1）听取部门汇报工作最好采用会议的形式，不能用每周或每月交一次书面报告的方式取代。定时交书面报告往往会流于形式，往往会使公司上下形成为了交报告而交报告的不良风气，使公司内部缺乏沟通，使问题和建议无法顺畅流通，长此以往，后果不堪设想。因此，每月一定要召开正式的会议，听取部门汇报工作，使各种信息在公司内部高效流通。

（2）表扬和批评是 2 种很好的激励方式，而当众的表扬和批评具有更强的影响力。采用会议的形式表扬或批评，有利于充分发挥表扬或批评的作用。在大会上当众表扬“明星员工”可以树立标杆，充分发挥榜样的好作用，有效推动竞争；当众批评某些情节严重的错误行为，有利于警醒员工，避免同类事情再度发生。

（3）当你给某一个项目小组，或者给你的部门分配任务时，一定要举行会议，不能采用邮件或者其他书面形式。因为分派任务意味着要达到团结上下、协作完成任务的目的，只有通过会议把任务公共化、明确化，才能促进协作，从而保证任务的及时完成。

2. 准备工作的重要性

当你确定有必要召开会议时，首先要了解准备工作究竟有什么作用。只有认识到准备工作的作用，才能有更大的积极性，更有信心地做好准备工作。一般来说，会议的准备工作具有以下几种作用。

（1）它可以帮助你确定在会议中应该采取哪种讨论方式达到最好的会议效果。

（2）通过充分的会议准备，以书面通知的形式告知参会者会议的目的、意图以及会议对与会者发言的要求等情况。

（3）良好的会议准备工作能够帮助节省会议时间，同时减少会议中可能出现的各种冲突，

使会议井然有序、卓有成效。

一次成功的会议背后一定是非常充分的准备工作。高效会议要求我们循序渐进、按部就班的做好会前的各项准备工作。好的准备是成功的一半，一定要做好会议的准备工作。

3．参加会议的人数

根据统计调查显示，会议人数最多不应超过9个，最为合适的人数是5至7人。这个人数既能够防止人数太少，不能集中反映群体思想，又能避免因为人数过多，从而造成一部分人的消极参与。当然，会议的合适人数应该根据会议的议题以及重要性具体决定。

4．会议准备工作核对单

会议的具体准备工作相当于会议的硬件，因此必须要面面俱到。将这些具体的工作一一列出就是一张会议准备工作核对单。这张核对单虽然数目繁多，然而却是会议成功的有效保证。

5．会场布置

如果与会人员超过10个，会场最好布置成“U”字形，主持人可以站在会场中央，这样有利于协调讨论。如果是全体会议，可以布置成阶梯教室的形式。如果是5到7人的会议，最好安排成圆桌形或者半岛形。

同时要注意，座位的位置非常重要。主席要考虑到谁坐在自己的对面，如果并列坐在一起，易形成不一致，如果开会只是为了向世界显示大家是团结的，就故意让反对者坐在自己的身边。通常，主席是一种荣誉、地位的象征。尤其在一个狭长的桌子两旁开会时距离主席越远者，是地位越低的人。所以，要根据自己的地位为自己选择适当的位置。

6．开会要准时

即使有人缺席也要按时开会。一旦人们看到你在会议开始前等待迟到者，将来人人都会迟到。

同时在会议开始前委任一名“计时员”，当超过或接近预定的会议结束时间时，此人会给你提示，以便能准时结束会议。要记住，准时结束会议和准时开始会议一样重要，大家的时间都很宝贵，谁也不愿意把时间浪费在冗长的会议中。

7．会议节奏的掌控

对于爱多说话的人，可以让他写详细报告，或者在发言时做出某些暗示，以及故意与别人交谈、打岔等。同时，要鼓励沉默的与会者多参与，这样既可以保护弱者同时又鼓励了不同意见。不要企图一切问题都有答案。安排最重要的人物最后发言。

8．会议总结

会后必须要把会议的内容以及形成的决议总结成文字发给所有与会者。这个工作不一定必须要由主席来做，但如果主席自己不准备做这件事情的话，必须在会议开始前指定一名会议秘书在会议过程中做文字记录工作并负责在会后发给大家。

会议总结应该包括以下内容：会议的时间、地点、参加者姓名，被讨论的议题和达成的决策（必要时请有关人员签字），以及下次会议的日期、时间和地点等。

15.4.2 会议参与者

（1）按时参加会议。无论是开会、赴约，有教养的人从不迟到。他们懂得，即使是无意迟到，对其他准时到场的人来说，也是不尊重的表现。

（2）如果是大型会议，进入或离开会场时要服从指挥，注意秩序，不一哄而上，不争先恐后，以免造成拥挤堵塞和防止事故。集会中应注意遵守会场纪律，不随便走动或发出声响，不破坏会场气氛，恭候报告人到来。当报告人到来时，会场应立即安静下来，并报以热烈的掌声。在报告过程中，应端坐静听，不交头接耳窃窃私语，不打瞌睡，不无故中途离席。报告人说到精彩处应鼓掌表示赞同，报告结束，也应以长时间的热烈掌声，以表示感谢。集会时如有上级领导或客人参加，应在他们到达时以热烈的掌声表示欢迎，离场时应让领导或客人先走，并以热烈的掌声欢送。

（3）有人发言时，不要在台下窃窃私语。只要有人发言，台下窃窃私语是相当不礼貌的行为。最重要的是把握尊重说话者的原则，切莫干扰他人谈话。

（4）如果有不同意见，注意不要随便打断别人的谈话，先听完对方的发言，然后再去反驳或者补充对方的看法和意见。

（5）轮到自己发言时，要注意，讲话要有条理，根据要介绍给大家的文档进行介绍，但在说的时候，也需要把各个点串起来。文档只是以点代面的东西，讲的时候要把点串起来，也就是说不能照着自己写的东西念，大家都在看这个文档，念是没有意义的。这是开会中的一大忌讳。要在骨架的基础上进一步丰富要讲述的东西。

（6）讲一个问题时要能想到很多的点，从不同的角度，不同的层次来阐述议题。比如项目总结会议，说这个项目成功了，为什么说它成功了，在哪些方面体现了它是成功的，是资源的分配与利用、技术的运用、完成的东西、质量、沟通，还是从中得到了什么等。

（7）不要大喊大叫，但也要声音清晰明亮，保证在场的所有与会人员都可以听到你说的话。

（8）会议中如果内急时，没有必要告诉他人所去之处，只需要道个歉说："对不起，我需要离开一下。"就可以了。

15.4.3 PowerPoint 技巧

现在很多公司在开会时会用到 PowerPoint，在这里我们简单介绍一下使用 PowerPoint 做开会辅助材料时的一些注意事项。

1．采用强有力的材料支持你的演示

在某种程度上，PowerPoint 的易用性可以成为它最致命的弱点。虽然利用它可以轻松创建引人注目的幻灯片和图形，但你需要知道 PowerPoint 并非独立存在。观众前来并不只是为了观看屏幕上显示的影像，更多的是为了聆听你的演说。你需要构建一组强有力的 PowerPoint 程序，但也应确保你的口头评论同样引人入胜。

2．简单化

我们可能都曾见过演讲者在PowerPoint和其他演示中似乎准备充分胸有成竹的情形。毕竟，他引以为傲的明显只是各个功能、特殊效果以及其他可用的小技巧。其实最有效的PowerPoint很简单，只需要易于理解的图表和反映演讲内容的图形。权威人士建议每行不超过5个字，每张幻灯片不超过5行，不要用太多的文字和图形破坏演示。

3．最小化幻灯片数量

PowerPoint的魅力在于能够以简明的方式传达观点和支持演讲者的评论。通过大量的数字和统计信息很难做到这点。对于这方面，最有效的PowerPoint展示绝不会用太多的图形和数字把观众淹没。取而代之的是，将这些图形和数字留在演示结尾分发的讲义中，以便观众彻底消化演示内容。如果你要强调PowerPoint中的某处统计信息，可以考虑采用图表或影像来传达。

4．不要照念PowerPoint

仅仅为观众照念可见的演示是PowerPoint用户最常见也最不好的习惯。这样说来，演讲者简直就是多余的，毫无存在的必要。不仅如此，即使是最具视觉吸引力的演示也会因此变得索然无味。PowerPoint与扩充性和讨论性的口头评论搭配才能达到最佳效果，而不是照念屏幕上的内容。

5．安排评论时间

另一个潜在的不良习惯是演讲者在一张新PowerPoint幻灯片出现时就立即开始评论。那只会分散观众的注意力。恰当编排的PowerPoint程序在展示新幻灯片时，先会给观众阅读和理解的机会，然后才会加以评论，拓展并增补屏幕内容。

6．要有一定的间歇

再次说明，PowerPoint是口头评语最有效的视觉搭配。经验丰富的PowerPoint用户会不失时机地将屏幕转为空白。那不仅能带给观众视觉上的休息，还能有效地将注意力集中到更需要口头强调的内容中，例如分组讨论或问答环节。

7．使用鲜明的颜色

文字、图表和背景颜色的强烈反差在传达信息和情感方面均非常有效。

8．导入其他影像和图表

不要把演示限制在PowerPoint提供的范围内。使用外部影像和图表（包括视频）能增强多样性和视觉吸引力。

9．在演示结尾分发讲义，而不是在演示过程中

这里可能有人会向我表示异议。可是没人希望对着一群忙于阅读评论总结的观众演讲。除非规定观众在你演示时要对照讲义，否则请你演示完再分发讲义。

10．演示前要严格编辑

永远不要忽视观众的看法。一旦你完成PowerPoint幻灯片草稿，在复查时你可以假设自己只是一名观众。如果发现不吸引人、跑题或令人疑惑的内容，要严格进行编辑。这是完善你的总体演示的好机会。

15.5 电 子 邮 件

电子邮件正日益成为人们日常生活和工作中必不可少的工具。作为一个工具来说，一定有它好的一面也有它不好的一面，只有充分了解这种工具的特点，才能真正掌握它，使它为己所用，而又不会被它伤害。

微软创始人比尔•盖茨在回答别人有关电子邮件作为沟通工具的特点时说："电子邮件在很多方面是一种独特的沟通工具，但它绝对不是直接交流的替代品。有许多人是和我在见面之前曾经用电子邮件联系过好几个月的。如果你对某人的言论不感兴趣，不回复他们的邮件比不接他们的电话要容易得多。事实上我几乎从不给别人留我的电话号码，但我的电子邮件地址知道的人却很多。我是唯一能读到发给我的邮件的人，所以没有人会担心他们发给我的邮件被公开。我们的邮件系统非常安全。同时电子邮件也为其他沟通方式提供了便利。比如它使你能够在开会前交流很多信息，从而使会议开得更有效。另外，电子邮件绝不是一个发脾气的好工具，因为你一旦发出去之后就无法收回它了。同时你也可以非常容易地发送表示友好的信息，这些信息不会被别人误解。"

15.5.1 基本注意事项

在日常工作中写电子邮件时要注意以下细节。

1．写 E-mail 要慎重，E-mail 是会留下证据的

一般来讲，如果是不确定的事情，比如提议等，最好不要用 E-mail 的方式，而应该面谈或电话沟通，等双方都认可之后再以 E-mail 的形式加以确认。E-mail 一定要及时备份，以备日后查阅时用。我曾经经历过的一件事情是，部门交接的时候，对方部门的一个同事说我没有把一个软件的 License 给她，幸亏我留了那封 E-mail，及时找了出来，我告诉她在某年某月某日下午几点几分，我曾经给她发过 E-mail，E-mail 里包含了所有信息，证据俱在，是保护自己的最佳手段，否则你也许要花很多时间去解释，有时候也未必能解释清楚。

同时，鉴于 E-mail 的这种证据性，在写 E-mail 的时候一定要慎重，否则你自己写的每一行字会反回来变成打击你的最佳武器。我们在工作中经常遇到的事情是，A 部门由于种种原因没有按时完成工作，老板问责的时候，他们抱怨说 B 部门没有及时配合他们，如果这样的抱怨是有证据的，就非常有力，但如果 B 部门只是在电话里没有配合，而在 E-mail 上给了及时回复的话，这种情况就对 A 部门非常不利。我们有时候会看到，B 部门马上翻出以前的邮件，充分证明自己对每一封邮件都做了及时的解答，这时候老板会怪谁呢？

2．注意 To 和 Cc 以及 Bcc 的不同

特别是在回复邮件的时候，很多人经常很随意地使用"回复所有人"，这是不对的。特别是当信件是发给老板的时候，你必须反复检查，这些人里有没有你的老板？有没有对方的老板？有

没有其他老板？有没有更大的老板？这些老板放在这里是不是合适？原信是发给所有老板的，那我的回信是否也有必要发给所有老板？这封信是不是有很细节的东西，不需要老板过问，发给老板除了给老板增加麻烦以外不会有任何好处？这封信里是不是有很重要的内容，自己无法承担这个责任，如果不让老板知道的话后果会很严重？所有这些问题都想清楚了，再来决定把信发给谁。

在撰写邮件时，要分清楚谁应该放在 To 里，谁应该放在 Cc 里。放在 To 里的人都是和这封邮件密切相关的人或群组，放在 Cc 里的人虽然关注这件事情的进展，但并不直接参与，一般是相关的领导。Bcc 则是你希望他也能了解这件事情，但不希望在 To 里和在 Cc 里的人知道他的存在。

在使用 Bcc 时要特别注意，如果你把某人放在 Bcc 里的话，则当别人“回复所有人”的时候，他是收不到回复的邮件的。反之，如果你是被放在 Bcc 列表当中的话，要注意不要轻易回复所有人，因为你的回复能被所有人看到，这样就暴露了当时把你加到 Bcc 列表中的那个人，很显然他的本意是不希望你的存在被别人所知觉的。

3．邮件标题要和邮件的内容相关

邮件标题要说清楚这封邮件的主题。如果是关于某件事情的报告，可以写“[报告]某月某日会议纪要”等。如果是要求，可以写“[申请]关于某事的申请”等，不要仅写一个“你好”或者什么也不写空在那里。

4．发送邮件之前要再次检查

我们经常会遇到某人发来的邮件里说，请看附件，但结果是邮件里根本没有附件，犯这样的低级错误很容易让人感觉你做事不认真。

特别是发给老板的邮件，以及 Cc 里包含了老板的邮件，更要仔细检查，看有没有拼写错误，有没有语法错误，有没有不该说的话，每句话是不是得体以及会不会让别人产生误解等。部门之间的邮件也要注意，每句话说出去是不是会成为别人手中的把柄。举例来说，某部门要你合作做个什么事情，但你暂时做不了或者不愿做，回信的时候千万不能直接说“这事情我不想做”而必须以更正式的合乎组织原则的方式加以拒绝，比如说，“我需要和我的老板商量一下”，或者说“按照公司的规定我目前不能做这件事情，如果确有需要，请按规定向我的老板反映”等。

总之，邮件一旦发出就无法收回，虽然从技术上说，有收回邮件的可能，但实际上这种措施有时候往往并不能奏效，所以，未发邮件之前，先想好退路，想好如何保护自己，这样的邮件才会是一件工作的利器而又不会伤到自己。

15.5.2 电子邮件类型

从用途来分，电子邮件有很多种类型，一般包括请求信、感谢信、表扬信、慰问信、推荐信、介绍信、证明信、咨询信、申请书、聘书、请柬、决心书、倡议书、建议书和挑战书等。这里我们列出最常用的几种，供你参考。

1．请求信

工作中常用到的一类邮件是请求信，如请求对方部门配合做某件事情，或者请求对方提供

某项资料等。在写请求信之前，先分清楚是哪种类型的请求，如果请求的事直接或间接地对对方有利，对方也可以从中得到好处，那就不必特别谦虚，过度客气。如果所请求的事对对方没有任何利处，则要有礼有节，按照礼仪信函的格式郑重书写，同时坦率地写明原因和情况，力求得到对方的理解。如果是实际上等于通知的请求信，由于是己方单方面的请求（或通知），所以信件的态度一定要谦和、恭敬，最重要的是写明理由和状况。不必要让对方做什么。

下面给出一封请求信的样例供参考。

老李，你好！

按照我们昨天开会讨论的结果，请协助尽快提供有关某项目的需求文档。文档的格式请参考上一项目的档案。由于此文档的按时提交与否会直接决定此项目能否按时完成，在此还需要得到您的大力配合。谢谢！

此致

署名

2．感谢信

太笼统的感谢显得感谢不够真诚，往往失去感谢的力度。感谢信的 3 段论法，首先笼统感谢对方，然后说明对方做的什么事情使你想要感谢他，再次说明他做的事情起到了什么样的效果，最后再次感谢并提出希望。举例如下。

老李，你好！

首先，感谢您的积极配合！由于您及时提供了关于某技术的指导，我们的某个技术难题得到了顺利解决，使整个项目的进度提前了三天。如果没有您的努力，这些成绩的产生是不可能的。

在此再次感谢您对项目的无私奉献！希望您能够继续关心我们的项目，并提出宝贵建议。

此致

署名

3．抱怨信

正如我们上面谈到的，抱怨信一般是让老板知晓的。当你开始发出一封抱怨信的时候，你也许挑起了一场战争。除非万不得已，否则应当尽量利用面谈或者电话沟通解决双方的分歧。只有当对方明显不配合，而没有对方的配合你无法工作的时候，你需要运用这一武器。写抱怨信时，一定要注意收集证据。举例如下。

老李，你好！

按照上次会议的要求，您应该在某月某日之前给我们提供与某项目有关的技术资料，但时至今日，我们一直没有收到。在本周二的时候，我们向您再次发信，但是却没有回音。由于缺少这些关键性的资料，我们的项目进度已经落后于计划三天了。如果在本周四之前我们还是无法得到相关资料的话，恐怕项目将无法正常完成。

希望您尽快提供相关资料，谢谢！

此致

署名

15.6 电话沟通

电话和面谈的相似之处是它是一种非正式会谈，这种非正式会谈不同于 E-mail，它不会留

下证据，当然除非是电话录音，不过一般公司内部很少这样做。但电话和面谈又有明显的不同，在打电话时你看不到对方的肢体语言和面部表情，所以它并不是最理想的沟通方式。我的一个管理课老师把电话会谈说成是吵架的最理想方式。如果你想吵架，那你就选择电话会谈而不是面谈。当然由于电话联系的高效率，我们工作生活当中又都离不开电话，所以注意一些电话时的礼节和技巧会有助于保持一个良好的人际关系。我们应该像塑造我们的视觉形象一样，来塑造我们打电话时的形象。说话使用的词汇、语音和语调都能帮助我们传递信息，并有助于我们抓住语言背后所蕴涵着的说话人的状态和情绪。

15.6.1 打电话的技巧

（1）打电话前先“清场”，环顾四周情况，是否嘈杂不安，以免对方深受干扰。

（2）注意挑选打电话的时机，避免在吃饭的时间里或者快下班的时候与对方联系。除非确有急事，否则凌晨或半夜打电话给对方，是极不受欢迎的事情。

（3）在打电话前先罗列一下要点，然后看着电脑里或手上的要点清单打电话。

（4）准备好纸笔在旁边，同时养成左手持话筒，右手拨电话号码的习惯。这样如果需要详细记载电话内容时，就可以用右手，逐一详细记载了。

（5）电话接通后，将你的名字告诉接电话者。让接电话的人知道你是谁，就好像你在和他面对面交谈一样。

（6）对着话筒微笑。在某些情况下，人们能够通过话筒感觉到对方的微笑。有些习惯于给别人打电话的人在打电话时往往在面前放一面镜子，可以在打电话时看着在镜子里的自己，注意不要阴沉着脸，或缺乏生机。

（7）如果想宣传某个主张，可以站起来说，这样语气更有力而热情，如果不方便的话，也必须挺直背脊来说话。

（8）让你的声音在语调、语速和音高方面显得富于变化。你应该往自己声音中注入一点儿活力，从而使对方对你保持注意，如果你想作出诚恳的反应，一定要富有表情，使你的语调显得自然而又亲切。

（9）讲话要简洁明了。因为电话同时只能容 2 个人谈话，在你跟这个人谈得太久的时候，可能另一个人打电话找你或者找他老打不通，甚至误了一件重要的事情。因此，交谈要长话短说，简而言之，除了必要的寒暄与客套之外，一定要少说与业务无关的话题，杜绝电话长时间占线的现象存在。每一句话都要有适当的间隔，并且主旨明确，不要拖泥带水，说了半天也没有直入核心。

（10）言语要富有条理性，不可语无伦次前后反复，让对方产生反感或啰唆。

（11）打电话要专心，不要一边说着话，一边干别的或者同时和座位旁边的人说话。如果你需要查阅信息，你必须把自己的做法告诉对方。切记，不要让对方手握着话筒时鸦雀无声，不知道你究竟是否还要与他通话。

（12）谨慎使用“稍微等一下”。接电话的人一般都讨厌对方让自己等一下，假如真的有必

要使用“稍等”这个字眼，那么尽量向对方讲明原因，并不时的插入一两句话，让接听电话的人知道他并没有被你所遗忘，假如你手头的事情所需要的时间并不止几分钟，那么建议你征求一下对方的意见，看对方是否同意你过一会儿再给他打过去，将给对方打电话的承诺写下来并且千万不可失约。

（13）当你准备结束电话时，向对方表示感谢。这样做会让对方意识到你们之间的谈话马上就要结束了。然后彼此客气地道别，说一声“再见”，再挂电话，不要只管自己讲完就挂断电话。

15.6.2 接电话的技巧

（1）把电话放在安全的地方。平时放置桌面的物品时，就应极力避免将可能发出声响的物品，比如茶杯、胶水与花瓶等物品放于电话旁边，否则，匆促举起话筒之时，极有可能使电话线碰到上述物品，导致翻覆而弄得满桌狼藉。对于卷起的电话线，也务必于拨电话前将其拉直，以免有碍通话。

（2）在电话机旁最好摆放一些纸和笔这样可以一边听电话一边随手将重点记录下来，电话结束后，接听电话应该对记录下来的重点妥善处理或上报认真对待。

（3）当你正在和某个人谈话时，尽量不要接电话。因为这时接电话是一种非常不礼貌的行为，这样做其实是在告诉你对面的人，电话那端的人比他更重要。除非你是在等一个非常重要的电话，否则就让语音信箱留信息。如果你必须要接电话，也应该让对方了解你为什么接这个电话，比如说“我正在等老板的电话”。

（4）一定要等电话响两声之后再接。你要利用这段时间平静自己的情绪，在你不知道来电人是谁、来电内容是什么之前，不要将你的情绪带给将要和你对话的那个人。同时，电话响 2 声，对打电话的人来说，也是他期待你拿起电话的最佳状态。

（5）电话接通后，国际通行惯例是接听者要先自报家门。但这样的做法和中国的传统习惯不符，因为是对方主动打过来的，他是有准备方，我方是无准备方，所以不必拘泥于必须自报家门。你可以采用一种变通的做法。先说：“你好！”这时对方一般会说：“你好，我是谁谁，我想找谁。”如果对方不说想找谁，你再自报家门也不迟。

（6）寒暄之后，诱导打电话者尽快切入正题。你可以向对方提出一些问题诱导对方，比如，“我今天应该如何帮助您？”或者“您需要我为您做些什么吗？”等。

（7）对于时间、地点、数字等信息，不仅要记录下来，还应该向对方复述一遍，以确定无误。

（8）用纸和笔记录接电话者的需求，并详细告诉接电话的人你要做些什么，以及你什么时候再和他进行联系。

（9）如果通话过程中，需要对方等待，接听者必须说：“对不起，请您稍等一下。”之后还要说出让他等候的理由，以免因等候而焦急。再次接听电话时必须向对方道歉：“对不起，让您久等了。”如果让对方等待时间较长，接听者应告知理由，并请他先挂掉电话待处理完后再拨电

话过去。

（10）如果对方语音太小，接听者可直接说："对不起，请您声音大一点好吗？我听不太清楚您讲话。"不要大声喊，因为需要大声的是对方，不是你。

（11）要养成在24小时之内一定回电话的习惯。

（12）如果遇到找人的电话，应迅速把电话转给被找者，如果被找者不在应对对方说："对不起，他现在不在座位上，我是某某，如果方便的话，可不可以让我帮你转达呢？"也可以请对方留下电话号码，等被找人回来，立即通知他给对方回电话。

15.7 面　谈

在日常工作中，如果想使某项工作尽快有进展，尽快得到解决的话，往往面谈会比电子邮件和电话有更好的效果。我的一个朋友讲过这样一个故事，他在某部门工作的时候，经常要和一个香港人打交道，E-mail经常发，电话也经常谈，总觉得这个人很难对付，甚至有点抵触情绪。但有一次这个香港同事到北京来出差，大家见了一面之后，发现这个人很好相处，从那以后就成了朋友，在以后的工作中也不觉得困难了。

面谈时候的一些基本礼仪必须注意。

1. 准备好谈话时所要用到的资料

很多面谈不是空手而去的，往往需要事先准备资料，即使没有资料，也应该准备好一个笔记本，以便在谈话过程中随时记录谈话要点。在面谈之前先将资料准备好，一方面可以使谈话有的放矢，另一方面可由此显示个人做事的计划性。

2. 约定好面谈的时间

面谈前充分估计这次会谈要用多长时间，按照时间长短和对方约定好合适的开始时间。如果是正式会谈，则必须选择正常上班时间谈，不要安排在快下班时或者中午休息时间。

3. 注意自我形象

千万不要小看形象的作用。虽然不是求职面试，但工作中的衣着与形象也会影响面谈的成效。好的形象总会使人产生愉悦的感觉，所提出的要求也往往容易被对方接受，如果衣冠不整，给人的第一感觉很糟糕，那后面会谈时会带来额外的困难。根据谈话场合谈话对象不同，衣着不一定非要局限于西装，有时候稍正规些的休闲装也可以是一种选择，会使人产生活力感，但不建议过于随便，比如穿背心短裤去面谈绝对是大忌。另外，不论什么服装，必须整洁干净，注意个人卫生，没有人喜欢和你谈话时闻你的汗臭味和口臭。

4. 注意基本礼节

面谈之前不要喝太多水，以免老想上洗手间。面谈前约好的时间不要迟到。见面时应面带微笑。谈话过程中多听少说，理解对方的意图，自己说话时用词要谨慎。谈话时注意个人仪态，特别是坐姿，不要盘腿或翘脚，即使对方资格比你低也不应该如此。

15.8 培　训

除了入职培训以外，很多公司还提供各种各样的培训机会，有技术方面的，也有管理方面的。作为企业一员，适度地参加各方面培训，对于自己全面发展有很大好处。

注意选择培训的时机，如果项目组工作很紧张，则不适合参加与本职工作无关的培训，可以选择项目工作不太紧张的时候参加。

要积极向老板争取培训机会。很多企业内部培训机会并不均等，如果你不去争取，也许好几年也轮不到你。所以如果看到有好的培训机会，不管是不是会遭到老板拒绝，都要积极要求。要求时必须摆出充分的理由来说明这次培训为什么有用，对部门工作会有什么帮助。遭到拒绝也不要怕，拒绝了一次两次之后，下次再有机会老板就会第一个想到你。

报名培训之后，一定要准时参加。因为不管是内部培训还是外部培训，都有人力财力的成本在内，如果报了名不去，或者去了也随随便便，达不到培训的效果，既浪费了企业的资源，又浪费了自己的时间，还不如一开始就不要报名。

培训过程中要带着笔和脑子去，把重要的内容用笔记录下来或者在讲义上做记号，以便回来复习。就像上学时一样，听课的时候要注意理解老师讲的理论性的东西，不要听理论时昏头大睡，讲笑话时来精神了，笑过之后什么也没学到，这培训就白上了。

和上学时照本宣科不同，很多培训讲师很能活跃课堂气氛，鼓励学员积极发言，遇到这样的机会，不要胆怯，畏葸不前，培训是最好的纠正你行为当中偏差的机会，积极配合老师就可以了。

同时，培训还是一个很好的认识本部门之外其他部门同事的机会，多交些朋友，也许会获得一些意想不到的收获。

15.9 加　班

对于软件工程师这个行业来说，不管在哪个公司，不管是中资还是外企，由于项目的压力，加班几乎成了家常便饭，加的时间长不说，而且不分周末、节假日，也没有什么加班费一说。虽然国家规定要有加班费，但很多企业以弹性工作制为由，很少认真执行国家标准。身处这种情况之下，作为员工一定要善于保护自己。

如果别人都在加班，你从不加班或者很少加班，肯定说不过去。特别是连老板都在加班的情况下，更别指望每天早早离开办公室了。但是老加班，不但失去了和家人团聚的时间，而且身体上也吃不消。所以如何平衡加班就需要一点技巧了。

（1）准确估计手头的工作量，如果加一会班能完得成，就加班。如果加班也完不成，一定

要提前跟老板说明情况，留待明天再解决。

（2）加班以后要注意补偿自己，明天可以晚来一会儿。或者一段时间老加班，就要给自己留出一段不忙的时间来充分放松一下。

（3）如果工作没完没了，永远是加班，永远没有休息的时候，那就看你自己选择吧，也许你觉得这样下去前途很远大，那就留下来继续这样忍受，也不要抱怨了，抱怨也没用，这是你自己的选择。

（4）如果觉得受不了，还是家庭和身体重要，那就早做打算，另谋高就吧。

下面是一些对于加班的战术上的提示，也许对你有一点帮助。

（1）熬夜工作者要供给充足的维生素 A，因为维生素 A 可调节视网膜感光物质——视紫的合成，能提高熬夜工作者对昏暗光线的适应力，而防止视觉疲劳。

（2）熬夜工作者劳动强度大，耗能多，应注意优质蛋白质的补充。动物蛋白质最好能达到蛋白质供应总量的一半。因为动物蛋白质含人体必需氨基酸，这对于保证熬夜工作者提高工作效率和身体健康是有好处的。

（3）应适当补充热量，吃一些水果、蔬菜及蛋白质食品，如肉、蛋等来补充体力消耗，但千万不要大鱼大肉地猛吃。吃一些花生米、杏仁、腰果和胡桃等干果类食品，它们含有丰富的蛋白质、维生素 B、维生素 E、钙和铁等矿物质以及植物油，而胆固醇的含量很低，对恢复体能有特殊的功效。

（4）熬夜中如感到精力不足或者欲睡，就应做一会儿体操或到户外活动一下。

（5）不要吃方便面来填饱肚子，以免火气太大。最好尽量以水果、面包和清粥小菜来充饥。

（6）开始熬夜前，来一颗维他命 B 群营养丸。维他命 B 能够解除疲劳，增强人体抗压力。

（7）提神饮料，最好以绿茶为主，可以提神，又可以消除体内多余的自由基，让你神清气爽。但是胃肠不好的人，最好改喝枸杞子泡水的茶，不仅可以解压，还可以明目。

15.10 请　　假

生活中难免会出现各种意外事件。比如，你突然生病了，或者是发生重要事件，这些都可能让你没法工作，那就只好请假。但在请假前你需要先了解清楚公司的请假政策。比如有几种假期，分别多少天等。

在这个基础之上，因为每人情况不同，你还需要再具体了解清楚你自己还剩多少天假期可以休。

如果同时有年假和事假，很多公司会要求先休年假，等年假休完后才可以请事假。

根据公司业务安排，尽可能在每年年初就提前安排好全年的年假，不要今天请假明天就休。

如果是请病假，则必须给老板打电话请假，不能只发个 E-mail 或者短信。

请假回来之后，要到公司系统里销假。

下面给出一封请假信范例。

首先，在邮件标题里写明是请假，然后，在假条的第一段，要开门见山但是有礼貌地提出请假，并写清你要请假的日期。例如，

Peter, I would like to know if I could ask for a casual leave of absence for one day on May 24th, this Wednesday.（王经理，我想在本周三5月24日请一天年假。）

接着，在假条的第二段，你要简单明了地陈述请假的原因事由，并且要表达对此给工作带来不便的歉意。例如：

This morning I received a telephone call from my dentist, urging me to come to his practice for immediate treatment of my teeth. I have been experiencing a stinging pain, depriving me of my sleep during the past fortnight. The situation could be worse, should infection occur.（今天早上我接到牙医的电话，催我尽快去诊看牙。我的牙已经痛了两周，晚上经常休息不好，如果再不看的话，可能会造成感染。）

Concerning my workload: As Wednesday is not as busy as the other weekdays, I think a one-day leave this Wednesday may be the best solution. I apologize for the inconvenience my absence from work may cause.（考虑到我的工作安排，由于星期三不像其他日子那么忙，我觉得在周三请一天假应该是最好的安排。很抱歉由于我的缺席而对工作造成的不便。）

最后，在假条的最后一段，要写上你希望获得准假的请求，或者具体等候答复的时间。例如，

Thanks. I will call you at 1:30p.m. Or you can call me at any time.（我会在今天下午1：30给您打电话，或者您也可以随时打我电话。谢谢！）

15.11 出　差

（1）出差前制订出差计划，根据出差业务需要计算所需天数，以及每天工作安排。现在各公司都开始节约经费，能不出差尽量不出差，如果确实需要出差，也要本着早去早回的原则。

（2）制订了工作安排之后，根据工作需要提前预定机票和酒店。可以委托部门秘书代理。预定房间时要注明尽量不要靠近马路的一侧，否则马路上的噪声可能会吵得你无法入睡。要含早餐，能上网。

（3）根据出差计划收拾行囊，包括办公用品、所需文件和换洗衣物等。

（4）身份证和信用卡必须随身携带，所有的消费尽可能用信用卡。

（5）用卡之后记得索要发票，发票名称注意让开票方写公司全称，不要写简称。

（6）现金保持在2 000元以上，以备不时之需。钱包里保持1 000元，电脑包里要有1 000元。如果丢了钱包，还有钱回家。

（7）如果经常坐飞机，办一张国航知音卡或者别的航空公司的优惠卡，可以积里程换取免

费机票。去机场时随身携带上此卡，办登机牌时可以出示。

（8）提前到达机场。国内航班要求提前1小时到，国际航班要求提前2小时，同时还要考虑到路上堵车的时间，特别是上下班高峰期的话，更要多提前。

（9）坐飞机和火车尽量要靠窗一侧的位置，便于用电脑，看风景。

（10）穷家富路，尽可能找大一点的饭店吃好的。同时记得天天吃水果，避免上火。

（11）去客户处要提前走，因为你不熟悉路，一旦堵车或者走了弯路还能补救。

（12）为了安全，不要一个人拎着电脑在马路上走很远的路，尽可能打车。

（13）出差前问问周围的同事有没有想要你从目的地带回来的东西，出差回来时可以一并捎回。即使没有这样的需求，出差时也可以带一些当地的小礼品送给亲朋好友、领导同事，可以大大地增加感情，以及加强以后在工作的默契。

15.12 报　销

首先，要了解清楚公司的报销制度。知道哪些票能报，哪些票不能报，票上的单位名称如何写，报销的有效时间有多长，报销系统如何使用等。可以去向公司财务部门咨询，也可以向有经验的员工了解情况。

然后，出差回来之后要尽早填写报销单，以免票据过期。

报销前，把所有发票按照财务要求整理好，按日期排序，并填写报销单。第一次报销的时候，如果不放心的话，可以请公司里的老同事帮自己检查一遍。

如果公司有规定，还需要在出差报告单上，按日期写清楚每张票据的用处，包括出租车始发地、目的地，餐费是午餐还是晚餐等，甚至写明每餐是和多少人吃的，分别是谁等。

如果公司是按天发放出差补贴，则需要按照公司规定，计算实际出差天数，申请出差补贴。

第16章 风雨江湖

俗话说："有人的地方就有政治。"也有人说："有人的地方就有江湖。"政治是什么？政治其实就是江湖。虽然我们是做技术的，做工程师的，但一样离不开政治。关于政治的确切定义，辞海里的解释很简单："经济的集中表现。"维基百科里的解释则是："政治是各种集团进行集体决策的一个过程，也是各种集团或个人为了各自的利益所结成的特定关系。"我们可以看到，这里强调了政治的两个特性：其一是它和利益相关，没有利益就没有政治；其二是它是一种关系，所谓关系，必须是两个人以上才有人与人之间的关系，一个人独自去打猎构不成政治，而两个人结伙去就会有政治发生。

很多情况下，你不想去找政治，但政治会主动来找你。举个例子来说，你是一名软件工程师，老板让你去向另外一个部门的一个工程师了解一项新技术或者合作开发一个模块，老板说已经和对方领导协商好了，你去工作就可以了，但实际工作的时候你会发现对方不愿意配合，因为什么？因为如果他教会了你，那就意味着他的饭碗可能要受影响，事关利益，这就是政治。如何解决这样的矛盾，也是对身为技术工程师的一个考验。再比如，很多工程师经常遇到这样的情况，他们抱怨说："我做了大部分的工作，但到年终评奖的时候或者升迁的时候，总是没有我的份，反倒是那个看起来平常工作不怎么积极的人得到升迁，为什么？"这样的事情如果发生的只是一回两回，那也许是运气问题，有偶然因素存在，但事实是我们经常在我们周边的环境中看到这样的事情发生，有时甚至直接发生在自己身上。所以谁敢说工程师的环境里就没有政治呢？

正是因为政治是各种关系的总和，这一章我们通过说明如何处理好各种关系来解读办公室政治。鉴于我们工程师的身份，我们处理政治问题的原则是，不主动营造敌人，但要学会保护自己。能做好这两条，基本上我们的技术生涯地就不会有什么太大的障碍。

16.1 从独立走向互赖

在上一章里，我们谈了高效能人士的 7 个习惯中的前 3 个习惯，这 3 个习惯可以帮助你从依赖走向独立。这里我们主要介绍一下后 4 个习惯。

如果说前 3 个习惯主要是与个人修养有关，那么这后 4 个习惯中的前 3 个主要讲的是如何处理人与人之间的关系，也就是帮你从独立走向互赖。互赖与依赖不同，依赖是单向的，你就像是

一棵没有成熟的小树苗，必须得到别人的帮助与督促才能完成事务，互赖则建立在独立的基础上，你自己已是一棵参天的大树，但你需要和其他大树一起互为依靠，才能成长出一片森林。

16.1.1 习惯四：双赢思维

第4个习惯是双赢思维。双赢思维是一种基于互敬，寻求互惠的思考框架与心意，目的是更丰盛的机会、财富及资源，而非患不足的敌对式竞争。双赢既非损人利己（赢输），亦非损己利人（输赢）。我们的工作伙伴及家庭成员要从互赖式的角度来思考（“我们”，而非“我”）。双赢思维鼓励我们解决问题，并协助个人找到互惠的解决办法，是一种资讯、力量、认可及报酬的分享。

最常用到双赢思维的地方是谈判过程中，这里的谈判不只是指商业市场中的谈判，实际上，谈判活动发生在我们日常生活的很多情景中。比如你想让孩子吃饭，他不肯吃，这里就有谈判的影子存在。

通常，我们在一般谈判时都会想到要讲究一些原则技巧。在买卖双方达成一笔买卖交易时，通常我们会看到，双方都会竭尽全力维护自己的报价。通常的谈判也最容易将谈判的焦点集中在价格上。例如，一位精明的卖主会把自己的产品讲的天花乱坠，尽量抬高自己产品的身价，报价要尽量高；而另一位出手不凡的买主也会在鸡蛋里挑骨头，从不同的角度指出产品的不足之出，从而将还价至少压低到对方出价的一半。最后双方都会讲出无数条理由来支持自己的报价，最后谈判在无奈情况下成为僵局。如果不是僵局，那么通常是一方作出了一定的让步，或双方经过漫长的多个回合，各自都进行了让步，从而达成的是一个中间价。这样的谈判方式，我们在商务活动中是非常常见的。

上述谈判方式，我们在谈判学上称之为“立场争辩式谈判”。立场争辩式谈判的特点是，谈判每一方都在为自己的既定立场争辩，欲通过一系列的让步而达成协议。立场争辩式的谈判属于最普通的传统谈判方式。许多介绍的谈判技巧也都是从这个出发点来谈的。然而，我们认为，如果在商业活动中，大家都遵循这样的谈判原则与技巧，往往会使谈判陷入一种误区。我们从实践中得到的教训却是，这种谈判方式有时最后谈判各方会不欢而散，甚至会破坏了双方今后的进一步合作机会。

因此，我们在这里就提出一个谈判要达到什么目的和遵循什么标准的问题。从商务角度来看，谈判应使得双方都得到商务发展的机会。为此，我们遵循的谈判原则与技巧至少应满足以下3个标准。

（1）谈判要达成一个明智的协议。

（2）谈判的方式必须有效率。

（3）谈判应该可以改进或至少不会伤害谈判各方的关系。

双赢谈判的4大原则。

（1）要知道人们在谈判中，不是为同样的东西而来的。不要认为对方想得到的东西，你就一定有所损失。你们要的不一定是同样的东西。糟糕的谈判对手试图强迫对方改变立场，而高明的谈判对手知道即使立场差别很大，双方的利益也可以是共同的，所以他们通过行动让对方改变立场，关注双方共同的利益。

（2）不要把谈判局限在一个问题上。

（3）不要太贪心。不要企图拿走谈判桌上的最后一分钱。

（4）把一些东西放回到谈判桌上，给一些额外的优惠。

16.1.2 习惯五：知彼解己

这个习惯的原文是 Synergize，有时候也翻译成“同理心倾听”。它主要强调了与人沟通时倾听和理解的重要性。

美国心理学家斯坦纳说过：“在哪里说得愈少，在哪里听到的就愈多。”只有很好听取别人的，才能更好说出自己的，虚心听取别人的意见是一个人进步必要条件。英国联合航空公司总裁兼总经理费斯诺也说过：“人有两只耳朵却只有一张嘴巴，这意味着人应该多听少讲。”所以自己意见不成熟时不能发表，说得过多了，说的就会成为做的障碍。另外，多听、多做、少说是一个人成熟的表现。

知彼解己这个习惯的第一要诀是你要用心去倾听，不但要用心听，而且要真正站在对方的角度，用你自己生活中的切身感受来做同样道理的倾听，所以叫“同理心倾听”。

第二要诀是在回答的时候，这里不是知彼知己，而是要能“解己”。知己容易做到，但要解己则并不容易。

当我们舍弃回答心，改以了解心去聆听别人，便能开启真正的沟通，增进彼此的关系。双方获得了解后，会觉得受到尊重与认可，进而卸下心防，坦然而谈，双方对彼此的了解也就更流畅自然。知彼需要仁慈心，解己需要勇气，能平衡两者，则可大幅提升沟通的效率。

16.1.3 习惯六：统合综效

统合综效谈的是创造第 3 种选择，既不是完全按照我的方式，也不是完全遵循你的方式，而是采取第 3 种远胜过个人之见的办法。它是互相尊重的成果，不但是了解彼此，甚至是称许彼此的差异，欣赏对方解决及掌握机会的手法。个人的力量是团队和家庭统合综效的利基，能使整体获得一加一大于二的成效。实践统合综效的人际关系和团队会扬弃敌对的态度（1+1=1/2），不以妥协为目标（1+1=1 1/2），也不仅止于合作（1+1=2），他们要的是创造式的合作（1+1=3 或更多）。

英国管理学家特雷默提出：“每个人的才华虽然高低不同，但一定是各有长短，因此在选拔人才时要看重的是他的优点而不是缺点，利用个人特有的才能再委以相应责任，使各安其职，这样才会使诸方矛盾趋于平衡。否则，职位与才华不能适合，使应有的能力发挥不出，彼此之间互不信服，势必造成冲突的加剧。”在一个团队中，每个人各有所长，但更重要的是领导者能将这些人依其专长运用到最适当的职位，使其能够发挥自己所长，进而让整个企业繁荣强盛。没有无用的人，只有不会用人的人。

在一次宴会上，唐太宗对王珐说：“你善于鉴别人才，尤其善于评论，你不妨从房玄龄等人开始，都一一做些评论，评一下他们的优缺点，同时和他们互相比较一下，你在哪些方面比他们优秀？”

王珐回答说：“孜孜不倦地办公，一心为国操劳，凡所知道的事没有不尽心尽力去做，在这方面我比不上房玄龄。常常留心于向皇上直言建议，认为皇上能力德行比不上尧舜很丢面子，这方面我比不上魏征。文武全才，既可以在外带兵打仗做将军，又可以进入朝廷搞管理担任宰相，在这方面，我比不上李靖。向皇上报告国家公务，详细明了，宣布皇上的命令或者转达下

属官员的汇报，能坚持做到公平公正，在这方面我不如温彦博。处理繁重的事务，解决难题，办事井井有条，这方面我也比不上戴胄。至于批评贪官污吏，表扬清正廉署，疾恶如仇，好善喜乐，这方面比起其他几位能人来说，我也有一日之长。”唐太宗非常赞同他的话，而大臣们也认为王珐完全道出了他们的心声，都说这些评论是正确的。从王珐的评论可以看出唐太宗的团队中，每个人各有所长，但更重要的是唐太宗能将这些人依其专长运用到最适当的职位，使其能够发挥自己所长，进而让整个国家繁荣强盛。

每个人的才华虽然高低不同，但一定是各有长短，因此在选拔人才时要看重的是他的优点而不是缺点，利用个人特有的才能再委以相应责任，使各安其职，这样才会使诸方矛盾趋于平衡。否则，职位与才华不能适合，使应有的能力发挥不出，彼此之间互不信服，势必造成冲突的加剧，这是我们选择人才时要考虑的一个重要问题。

16.1.4 习惯七：不断更新

最后一个习惯是不断更新，它指的是，如何在 4 个基本生活面向（生理、社会/情感、心智及心灵）中，不断更新自己。这个习惯提升了其他 6 个习惯的实施效率。对组织而言，7 个习惯提供了愿景、更新及不断的改善，使组织不至呈现老化及疲态，并迈向新的成长之径。对家庭而言，7 个习惯透过固定的个人及家庭活动，使家庭效能升级，就像建立传统，使家庭日新月异，即是一例。

16.1.5 总图

为了解释这 7 个习惯互相之间的关系，柯维博士还画了一张图来描述它们（如图 16.1），记住了这张图就很容易记住这 7 个习惯。

图 16.1 高效能人士的 7 个习惯

在这张图里，我们可以看到，习惯 1、2、3 属于个人成功范畴，而习惯 4、5、6 属于公众成功范畴。如果你不具备习惯 1、2、3，那么你还处于依赖阶段，需要别人的帮助才能生存，如果你具备了习惯 1、2、3，那么你就上升到独立阶段，可以不依赖别人而活着，但如果想真正成功，你还需要习惯 4、5、6 的配合，有了习惯 4、5、6，你和你的朋友，你的团队变成一种互相依赖，共同提高的关系，从而达到全面的成功。

16.2 和老板的关系

自古至今，管理者和被管理者之间始终存在矛盾，如何正视这种矛盾是破解和老板关系谜局的窍门。从本质上来说，管理者和被管理者之间的矛盾实质上来自于我们自己本性的矛盾，也就是社会性与自由性之间的矛盾。日本管理学家矢泽清弘提出："谁都想做自己的主宰，而不愿受别人驱使。"愈是有主见的人，愈想成为自己的主人。人是社会动物，每个人都有其社会性的一面，必须依赖社会才能生存，我们为老板打工就是这种社会性的体现，如果一个人可以完全不依赖这种社会性就能生存，自然也就不会有这种矛盾。同时，作为一个独立的个体，我们每个人又都向往自由，希望自己能够不受拘束地活着，如果这种自由性与社会性发生激烈冲突，这时候就体现了我们与老板之间的矛盾与冲突。

和老板的关系可以说是所有关系里最重要的一种关系，说它是最重要的关系一点不为过。因为你的老板是直接发钱给你的人，你的升与降，得与失完全控制在老板手里，同时，你的工作业绩也会直接或间接影响到他的业绩。

实际上，员工和老板之间存在着一种相互依存的关系。老板是组织者，员工是执行者。组织者需要执行者来实现指令，如果员工经常无法按时或按要求完成任务，老板的组织结果会是失败的。同时，执行者需要组织者提供正确的信息和指导，如果在一些关键点上，老板没有能及时给予帮助，员工也无法把事情办好。

在合作基础之上，由于每个老板的性格不同，阅历不同，会有各种各样不同风格的老板。可以说，没有两个老板是完全一样的，有些老板可能比较容易相处，有些老板可能会非常严厉，有些老板思路清晰，讲话头头是道，但也有些老板并不爱把一切事情都讲得那么清楚明白，所以对付不同的老板有不同的方法。英文里对此有一个专门的短语称之为 Manage up，意思是向上管理，很形象。Manage up 对每个员工来讲都不是一件简单的事情，需要花时间和精力去摸索，去适应，但这些花费绝对是值得的。

16.2.1 给你的老板分个类

不同的老板有不同的风格，如果我们了解了他们的风格，和老板处理起关系来就会容易很多。有的老板很关心员工，有的老板只关心业绩，但不管什么样的老板，最后都能在下面的管理方格图里找到他们的坐标。同时，了解了管理方格理论，在你自己日后当了老板也可以有意

识地往良好的管理方式努力，而避免犯一些常规的错误。

管理方格图（Management Grid Theory），如图 16.2，是由美国德克萨斯大学的行为科学家罗伯特·布莱克（Robert R. Blake）和简·莫顿（Jane S. Mouton）在 1964 年出版的《管理方格》（1978 年修订再版，改名为《新管理方格》）一书中提出的。它是一张纵轴和横轴各 9 等分的方格图，纵轴表示企业领导者对人的关心程度（包含了员工对自尊的维护、基于信任而非基于服从来授予职责、提供良好的工作条件和保持良好的人际关系等），横轴表示企业领导者对业绩的关心程度（包括政策决议的质量、程序与过程、研究工作的创造性、职能人员的服务质量、工作效率和产量），其中，第 1 格表示关心程度最小，第 9 格表示关心程度最大。

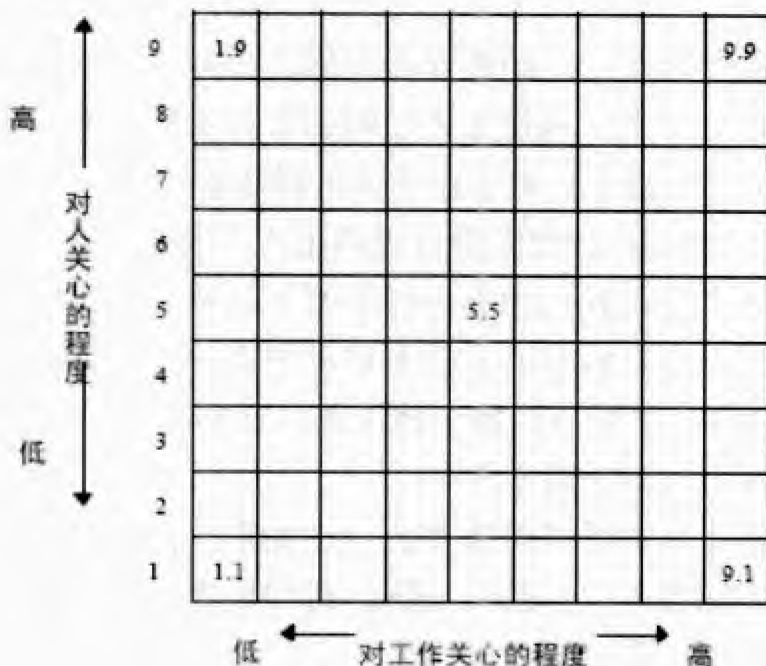


图 16.2 管理方格图

粗略地来分，老板一般可以分为 5 个大类。在管理方格图中，1.1 方格表示对人和工作都很少关心，这种领导必然失败。9.1 方格表示重点放在工作上，而对人很少关心。领导人员的权力很大，指挥和控制下属的活动，而下属只能奉命行事，不能发挥积极性和创造性。1.9 方格表示重点放在满足职工的需要上，而对指挥监督、规章制度却重视不够。5.5 方格表示领导者对人的关心和对工作的关心保持中间状态，只求维持一般的工作效率与士气，不积极促使下属发扬创造革新的精神。只有 9.9 方格表示对人和工作都很关心，能使员工和生产两个方面最理想、最有效地结合起来。这种领导方式要求创造出这样一种管理状况：职工能了解组织的目标并关心其结果，从而自我控制，自我指挥，充分发挥生产积极性，为实现组织的目标而努力工作。

除了那些基本的定向外，还可以找出其他一些组合。比如，5.1 方格表示准生产中心型管理，比较关心生产，不大关心人；1.5 方格表示准人中心型管理，比较关心人，不大关心生产；9.5 方格表示以生产为中心的准理想型管理，重点抓生产，也比较关心人；5.9 方格表示以人为中心的准理想型管理，重点在于关心人，也比较关心生产。还有，如果一个管理人员与其部属关系会有 9.1 定向和 1.9 体谅，就是家长作风；当一个管理人员以 9.1 定向方式追赶生产，而在这样做的时候激起了怨恨和反抗时，又到了 1.9 定向，这就是大弧度钟摆。还有平衡方法、双帽方

法、统计的5.5方法等。

下面我们对这5种基本的管理类型做一下解析。

1. 贫乏型管理（1，1）

这是一种既不关心生产，也不关心人员的管理方式。这种方式的领导者并不是组织的叛逆，恰恰相反，他们对组织有高度的依恋，而仅仅是缺乏热情和上进心而已。他们是理性的而不是糊涂的，其行为总是试图以最小的付出来保住自己的职位。他们往往具有"熬"出来的资历优势。国内的书籍在介绍这一类型时，经常出现一些误导性的解释，如把贫乏型管理者理解为能力低下者，这是有偏差的。管理水平低下不等于领导人的能力低下，不努力工作不等于不依赖组织。有时情况恰恰相反，领导人能力很高却管理效果不好，越是混日子的职员对组织的依附性越强。在中国，判断是否为贫乏型管理，有一个简易标准，凡是那种把"没有功劳也有苦劳，没有苦劳也有疲劳"挂在嘴边的领导者，恰恰是布莱克和莫顿强调的贫乏型管理者的写照。

贫乏型管理者会用漠不关心的态度，最小的努力去完成必要的工作并维持人际关系。他们所求不多，但付出更少。他们只是按符合规定的标准去做事，且认为多一事不如少一事。用8个字形容最合适——无精打采，放任自流。他们的心理是要抓住现状，而不是抓住未来。他们为自己辩解的理由往往是，"现代化的激烈竞争会导致人性丧失"、"公司追逐利润是钱迷心窍"、"不遗余力向上爬肯定会不择手段不讲道德"等，其实不过是给自己无所作为寻找一个借口，实现心理平衡。

这种领导人，往往会以授权的名义把工作交给部属。与其说是授权，不如说是弃权。他们的理由是，工作要靠部下，一线工作者比领导者更清楚问题所在和关键环节，上级不宜对工作目标和进度加以硬性规定，传递信息要尽可能准确（实际是上传下达的传声筒而已），人员安排上给谁都行，上面说什么就是什么，下面做什么就是什么。决策靠上级，干活靠下级，舒适靠自己。凡是那种遇事就声明"我不对这事负责，我只是在这儿工作"的人，就是这种类型领导者的口头禅。他们多数希望避免使自己处于被关注、被议论的地位。当组织内爆发冲突时往往会保持中立，摆出与世无争的姿态，不会激起别人对他的过分不满。只有在涉及自身利益时，他们才会紧紧盯着。可以说，是对自身利益"与世无争"，还是对事业发展"与世无争"，是对领导责任"难得糊涂"，还是对组织使命"难得糊涂"，是判断贫乏型管理与其他类型管理的分水岭。这种领导者，在组织发展史上不会留下自己的业绩，组织的记忆中也没有他们的空间。

2. 权威型管理（9，1）

这是一种"一心扑在工作上"的管理方式。这种领导人可能表现出对工作非常关心，但忽略对人的关心。他们往往强调工作环境对工作效率的影响，而不大重视人的因素对工作效率的影响。这种方式的领导者往往有极强的控制欲，希望可以有效地控制部属。从个性上看，这种领导者通常具有坚强的信念，不易屈服，并且有信心做好管理工作。所以他会主动选定工作方向，命令下属去服从。若有人不服从便会被他认为是工作中的障碍。当遭遇失败时，他会以暴怒来发泄。当组织中发生冲突时，他会以压制的态度来对待。

权威型管理者在潜意识里惧怕失败，他们总要显示出自己的强大和优势，所以，一旦遇到

挫折，他们往往会把失败的原因归于别人，不能反求诸己。因为承认自己失误，就等于否定自己的能力。同样，他们常常不太听得进去别人的忠告，尽管在内心里也有可能觉得别人的意见有道理，但接受别人的意见似乎就会显得自己没有能力或者失去独立性。这种人遭到失败的典型特征是发怒。如果有人说“他是一个容易被激怒的人”，那么，八九不离十他就属于权威型管理者。他们往往在外表上喜欢充当硬汉，缺乏实力时则采取虚张声势的策略。

权威型管理者往往对等级制度极为重视，他们习以为常的是“命令、服从”关系。在对下级的指示中，他们会尽可能把工作内容细节、时间地点和方法技术都交代清楚。而且在部属工作时总是频繁地检查，流露出几分不放心。当然，这中间会夹杂着一点父辈对子女的真诚。他们的信条是“家有千口，主事一人”。在他们眼里，为了搞好工作肯定要得罪人，领导者不可能讨好每一个下级，优胜劣汰是十分合理的。他们喜欢出成绩的部下。如果要推行目标管理，他们会抛弃员工自主性的内核，而把目标管理变成限额管理。在听取部下汇报时，他们往往以怀疑的甚至是挑剔的眼光打量部下，毫不客气地打断他们的发言，要求用翔实具体的数据说话。他们的提问往往是进攻性的，而倾听往往是防御性的。在工作中，他们不浪费自己的时间也不浪费部下的时间。这种人容易固执己见。他们的口头禅是“绝不”、“必须”、“应该”。有时候，即使在开玩笑时也能显示出争强好胜。比如说，“打个赌，如果我错了赔你多少?你先把你的赌本放下!”对待出错的下级，他们就像猫扑向耗子。有意思的是，这种人一般语速较快，声音较大，绝少慢条斯理。运用位置和权力上的强势压服下属是他们的拿手好戏。

在权威型管理的支配下，部下要么形成服从的习惯，要么采取对抗的措施。常见的对抗反应，就是部下的积极性受到打击，撤退到贫乏型管理的领域，表现出中立和冷淡，甚至麻木不仁。而这时的权威型管理者，看到下属果如自己所料，还会对自己预料准确而沾沾自喜。他真诚地认为，对懒惰和麻木的部下就应该来点强硬手段，却不知道很可能是他的强硬手段引发了部下的懒惰和麻木。有人认为，正是权威型管理，制造了依赖、归顺和服从，压抑甚至扼杀了部下的自主性和创造性，而反过来又促使领导人进一步采取权威型管理方式。在工业化社会里，权威型管理是十分常见的，组织中的厌倦、疏远、冷漠以及效率下降，往往是这种管理类型的长期反应。

3. 乡村俱乐部型管理（1，9）

这是一种追求下级拥戴和同情的管理方式。这种领导人对人非常关心，十分重视自己与下属、上司和同僚的关系，但忽视工作的状况。这类领导者往往认为下属的态度和情感是自己进行管理工作的支撑，如果得到下属发自内心的支持，他就是安全的。因此，他会主动去关心下级的需求是否满足，避免将自己的意志强加于人，致力于搞好上下级关系，创造一个友好的温暖的气氛。

乡村俱乐部型管理的本质，是领导人担心遭到抵制。恐惧使他们尽可能对别人表现出默许和顺从。他期望得到他人的感情和认可，却又害怕失去拥戴。直观地看，这种领导人是用“讨好”的方式来增强他人的认可。他们的信念是只要我对别人好，别人就不会伤害我。用中国话来说，就是以“多栽花，少栽刺” 为宗旨。冠冕堂皇的理由，是领导者应当维持组织中的融洽

气氛并保持高昂的士气，做到人人心情舒畅。他们外表上是笑面弥勒，而内心却充满焦虑。在表象上，他们能够给部属尽可能的支持和帮助，而在实质上，他们把部属看做自己的最重要财富，其目的是要换取部下投桃报李式的追随。所以，这种组织会形成懒散、自由，类似于中国乡村“老碗会”式的氛围。“乡村俱乐部”的名称即由此而来。

这种领导者在下达指令时，往往会采取委婉并鼓励部下的方式，笑容可掬。“我相信你能把这件事干好，我只是给大家跑腿服务。”他们常常对部下标榜，“有事没事都欢迎大家来我这里坐坐，我的门永远是敞开的。”在他们眼里，控制就意味着不信任，心里总是盘算着如何对“众口难调”的部下熬出一锅人人都感到美味的好汤。他们会有意无意地强调“我们是一个融洽的大家庭”，而且还会对员工的家人、亲属表达出足够的关怀。如果这种领导人主持会议，那么，他会等人到齐了再宣布会议开始，哪怕有人迟迟不来，他也有耐心等下去。这意味着会议的主题已经由工作变成了社交。正因为如此，他们更重视会前的私下沟通和协商。有意思的是，权威型管理者开会总是最先讲话，而乡村俱乐部型管理者开会总是最后发言。这种领导人最担心组织内部发生冲突。所以，他们常常力图制造出组织内的愉快气氛来，不断的闲谈就是增进友谊的有效方式之一。他们特别注意“礼让三分”，总是让别人走在前面。在他们口里，很难听到“你错了”、“我不同意”等否定性词汇。一旦有不同意见，他们一般会保持沉默，必须表态的情况下，他们会模棱两可，实在不能中立时，则会把要否定的意见用“但是”予以适度肯定。当工作业绩下降或出现明显问题时，他们会向大伙儿道歉并表示承担责任，但同时又会强调“下不为例”。即使是真心想解决问题，也会设法掩饰产生问题的原因，大事化小，追求皆大欢喜的效果。如果说，有些权威型领导人靠“骂”，那么，有些乡村俱乐部领导人则靠“骗”。他们对面带难色的部属，往往不是用纪律威胁，而是许诺给他一个自己也把握不大的未来好处。所以，他们不轻易说“不”，但却常常“遗忘”。对于这种领导人，部下可能先是惬意，继而失望，最后则发展为蔑视，有雄心的人会选择离开。在无须努力竞争就能获得高额利润的企业，或者是缺乏竞争压力的垄断性组织，都容易滋生出这种管理方式。

4．中庸型管理（5，5）

这是一种介于“铁面包公”（9，1）式和“笑面弥勒”（1，9）式之间的管理方式，不走极端。这种领导人对工作的关心和对人的关心兼顾，尤其重视群体归属和组织人格。他们很重视同僚和员工对自己的评价，力求在群体中稍有优势。所以，他们往往健谈，喜欢交友，善于应酬，不失风趣，仪态得体。他们的适应性很强，流行的意见就是他的意见，别人抵制东西他也抵制，大家的看法等于他的看法，始终能够与多数人保持一致。但他们还是努力的，如果得到别人好的评价，会使他们感到发自内心的喜悦。他们的信念是，只要能站在大众一边而且比较出色，就是一个地位牢靠的经理。所以，他们的管理风格是组织化的，很少形成鲜明的个人特色。

中庸型管理者往往采取“应答式”策略，力图与大家保持一致，办事有度，处理适中，不标新立异。所以，他的工作往往不是开拓式的，而是修修补补式的。这种领导的实质，是突出“权宜”二字。他们在领导方法上，不太赞成命令式，也不大喜欢放任自流式，而是以激励和沟

通为主。在制订计划时，他们重视部下的想法和意见，尽可能做到实施阻力最小。在布置工作时，他们常用说服甚至恳求的方法，对部下以鼓励为主，也乐意给部下以力所能及的支持和帮助。人事安排上，注重能够配合的程度，技术能力反倒在其次。在他们眼里，多数人都是通情达理的，偶尔有怪话和牢骚也很正常。这种领导一般不会着眼于最大产量，而是把定额控制在人们乐于接受同时又要适度努力的界限内。“讲求实际”、“权衡”、“不仅……而且……”是他们的口头禅。他们不是寻求最佳，而是寻求妥协。在沟通中，他们更乐意采用非正式的和轻松的方式。在他们看来，领导活动不过是催化剂和促进剂。所以，他们在表扬部下时不忘记还要附带提出更好的希望，而在批评部下时则特别要强调还有哪些是可取之处。

一旦有什么疑难或冲突，这种领导人往往求助于传统和惯例，所以他们非常重视组织中长期形成的不成文规则，要尽可能回避和减少管理中的不确定性。如果拿不准，往往采用民意测验或者市场调查作为依据，看重多数人的反应，信奉“法不治众”。为了减少不确定性，他们有广泛采用非正式系统的偏好。他们与部下的关系通常比较融洽，而部下也不知不觉中会受到感染。部下在工作中的第一反应，往往不是问“怎样做最好”，而是考虑“我的头儿希望我怎样做”。创新力会大大削弱，谁要打破常规，则会承受被视为异端的风险。如果在工作中受挫，有可能滑向贫乏型管理。

5．团队型管理（9，9）

这是一种个人与组织、工作与情感达到高度和谐的管理方式。领导人不但对工作和人员都予以高度关心，而且还会把二者融为一体，他们的才智和热心兼备，能够推动自愿合作、自主创新、组织开放和责任分担。他们寻求组织发展与个人成长的吻合，追求做出重大贡献的喜悦和兴奋。他们也有可能在复杂问题面前受挫，从而出现短期的心神不宁甚至沮丧，但他们不会气馁，而是相信采取慎重态度、通过献身精神和进行多种探索，能够解决真正的棘手问题，走向成功。

团队型管理的本质是建立个人发展与组织成长之间的内在联系，个人通过组织目标凝聚为团队，组织在个人自我实现中获得成就。这种类型的领导人，会把自己的精力集中于决策上，计划的制订要同利害相关者一起完成，组织构架能做到责任明确、程序清晰、规则完善。对部下，富有前瞻性的指导和真诚的帮助同步进行，组织的目标控制和员工的自我控制互为补充。在人事配置上，把工作要求和员工能力开发结合起来。总之，这种管理，是要在员工参与中实现对组织目标的理解、赞同和支持，从而实现真正的协作（这需要把协作与妥协折中加以明确区别）。按照布莱克和莫顿的观点，只有这种团队型管理，才能真正推行目标管理。

即使在团队型管理中，也不可能没有冲突。但在这种管理方式下，冲突是可以预防和解决的，而且有可能成为创新的契机。预防冲突的手段，首先是开诚布公、以信任为基础的自由沟通。领导人善于采用双向沟通方式，比如说，对有不同看法的员工提出：“我们能不能协商一下，我不能保证会赞同你的意见，但却对你的意见很感兴趣。”这种双向沟通，可以大大降低信息误解的几率。团队的管理，能够通过讲清道理，寻求事实，推演逻辑，鼓励人们思考、分析和评价相关问题。在发生对立意见甚至发生对抗时，领导者可以通过这种方式来探究分歧产生的动

机、理由和原因。所以，这种方式很少发生愤怒、恐惧和敌意。部属能够在这样的管理下产生骨干精神，勇于承担责任。当然，有时也会出现对领导者要求过高的不理解，对过于激烈的创新认为不切实际，但这都属于可解决的问题。

16.2.2 老板的话必须要听

了解了老板的分类后，我们来看一下几个和老板相处的基本原则。这些原则对于不同类型的老板都适用，唯一的区别只在于程度上的不同。

第一个基本原则是，老板的话必须要听。基本上来讲，这个原则谁都明白，但真正做的时候，却不一定每时每刻都能做到，也不一定每个人都能做好。

先讲一个真实的故事。小李 24 岁那一年在一家私企里工作，他的主要职责是技术工程师。有一天，小李正在钻研技术的时候，一个销售员小彭来找他，让小李帮她改一份标书里的技术内容。因为每个人手头都有自己的任务，不可能随时随地帮任何人的忙，于是小李跟小彭说："我现在正忙，你去找别人吧。"但是销售小彭说这份标书很重要，必须要小李马上帮她。小李还是不同意帮，并且当时小李的部门经理赵亮就坐在他旁边，什么话也没说，那意思肯定是说你自己处理好了，你愿意帮就帮，不愿意帮就不帮。

销售小彭就去找来了总经理，总经理认为这件事情很重要，于是直接走到小李的位子跟前，和颜悦色地对小李说："你先帮她改一下这份标书。"恰巧小李这几天正好在读史记，讲到"周亚夫军细柳"的故事，给他留下了很深刻的印象。故事是这样的。

说汉文帝后元六年的时候，匈奴大规模侵入汉朝边境。于是，朝廷委派宗正官刘礼为将军，驻军在霸上；祝兹侯徐厉为将军，驻军在棘门；又派河内郡太守周亚夫为将军，驻军细柳，以防备胡人侵扰。有一天，皇上亲自去这几个地方慰劳军队。到了霸上和棘门的军营，长驱直入，将军及其属下都骑着马迎送。随后来到了细柳军营，只见官兵都披戴盔甲，兵器锐利，开弓搭箭，弓拉满月。皇上的先行卫队到了营前，不准进入。先行的卫队说："皇上即将驾到。"镇守军营的将官回答："将军有令'军中只听从将军的命令，不听从天子的诏令。'"过不多久，皇上驾到，也不让入军营。于是皇上就派使者拿了天子的凭证去告诉将军："我要进营慰劳军队。"周亚夫这才传令打开军营大门。事后汉文帝不但没有处罚周亚夫，并且长时间对周亚夫赞叹不已还给他升了官。

因为小李脑子里正想到这个故事，于是他眼睛盯着电脑屏幕，头也没抬地指了指坐在旁边的部门经理赵亮对总经理说："你去找赵亮说吧。"没想到，总经理勃然大怒，大吼了一声："谁是你老板？"小李当即愣怔在那里，好半天才回过神来。总经理说完这句话立刻怒气冲冲地回自己办公室了。接下来的事情可想知，小李不但必须要把这个事情做好，并且以后也不可能再在这个公司发展了。

实际上，在这个故事里小李所犯的错误不只一处，比如同事关系的处理。如果和销售的关系处理得好的话，不至于发展到找总经理的地步，关于如何处理同其他部门同事的关系，我们后面还会讲到，这是其一。而最关键的是和老板的关系。在这个故事里，如果事情的经过是作

为部门经理的赵亮主动不让销售小彭来找小李，或者拒绝总经理指派的任务是赵亮干的，那么这个矛盾是赵亮和总经理之间的矛盾，和小李没什么关系。但现在的情况是人家部门经理赵亮什么话都没说，结果变成了小李和总经理之间的矛盾，是小李不听总经理的话，其后果当然很糟糕。

所以，这个故事告诉我们，老板的话一定要听。除非你有明确的指令，比如周亚夫的将军说的："将军有令'军中只听从将军的命令，不听从天子的诏令。'"否则越大的老板的话就越得要听。

16.2.3 老板的话要听懂

俗话说：唱戏听声，锣鼓听音。听了话不一定表示听懂了话，和老板处理关系的第二个秘诀就是一定要听懂老板的话。

小王是某公司一个内部项目的经理，有一天老板和他谈话，她先夸了小王项目管理的不错，开发的软件产品很适合部门使用。然后又说听说最近另外一个分公司也开发了一款类似产品，功能更为强大，最后问小王如果他做部门主管的话，会不会考虑引进另外那个分公司的产品？

小王其实对那款产品也有所耳闻，并且从纯粹技术的角度来说，如果全公司都使用同一款产品的话，从开发和维护角度来说更轻松，并且可以减少很多重复开发的人力物力，为总公司节省一笔相当不小的费用。于是小王天真地说："会啊，这样做对公司有很多好处，比如……"然后滔滔不绝地列举了很多好处。就看老板的脸色越来越不好看，随后过不了多久，另外一个同事升了部门主管，并且部门很快扩大，产品也越来越丰富。

事后，小王才知道，老板正是由于对别的分公司的产品功能有所了解，所以才希望自己的产品必须有竞争力，否则真如小王所说，彻底引进别的分公司的产品的话，那自己这个部门就没有存在的必要了。对总公司有利的事情未必对分公司有利，怎么样给总公司省钱那是总公司的人考虑的事情，作为分公司一员首先要考虑的是分公司的利益。小王正是由于不明此理，所以才丢掉了升职的机会。

听老板说话一定要善于揣摩老板话里的意思。相当多的老板说话的风格是话里藏话的，因为有些话不方便直接说出来，而必须靠你去理解体会。和老板说话一要小心，二要过脑子，不是随声附和当应声虫就能简单应付的。比如在这个故事里，老板说某分公司产品好，小王随声附和，恰恰犯了大忌讳。

听懂老板话的几个方法。

1. 搞清楚背景

背景信息包括很多很重要的信息，包括立场问题、目前形势等。尽可能搞清楚情况再发言，如果实在不清楚的话，就宁可装糊涂，承认自己无知，也不可贸然发表意见。

2. 用心听讲

老板前面说的铺垫性的话语里往往会隐含着微妙的倾向性。比如上文里的老板首先夸了小王的产品适合本部门需要，像这样重要的信息就是在说明本部门产品的不可替代性，像这样重

要的信息却被小王轻易忽略，殊为可惜。

3．要善于察颜观色

俗话说："眼睛是心灵的窗口。"听讲和说话的时候都要注意看着对方的眼睛，不要只顾埋头整理自己的思路。看着眼睛说话，如果哪里有问题就要尽快停下来，不要在错误的路上越走越远。

在听懂老板话的基础上，如何应答也是一门学问。美国心理学家古德指出："成功的沟通，靠的是准确地把握别人的观点。"如果能准确地把握了老板内心深处的真实观点，自然可以做出巧妙的回答。

三国时期，曹操很喜爱曹植的才华，因此想废了曹丕转立曹植为太子。当曹操将这件事征求贾诩的意见时，贾诩却一声不吭。曹操就很奇怪地问："你为什么不说话？"

贾诩说："我正在想一件事呢！"

曹操问："你在想什么事呢？"

贾诩答："我正在想袁绍、刘表废长立幼招致灾祸的事。"

曹操听后哈哈大笑，立刻明白了贾诩的言外之意，于是不再提废曹丕的事了。

若干年以后的南朝，也发生了一件类似的故事。

在南朝时，齐高帝曾与当时的书法家王僧虔一起研习书法。有一次，高帝突然问王僧虔说："你和我谁的字更好？"

这问题比较难回答，说高帝的字比自己的好，是违心之言；说高帝的字不如自己，又会使高帝的面子搁不住，弄不好还会将君臣之间的关系弄得很糟糕。

王僧虔的回答很巧妙："我的字臣中最好，您的字君中最好。"

皇帝就那么几个，而臣子却不计其数，王僧虔的言外之意是很清楚的。

高帝领悟了其中的言外之意，哈哈一笑，也就作罢，不再提这事了。

曹操和齐高帝所提的问题对于下属来说可谓是非常棘手，稍有不慎就会引起龙颜大怒。而贾诩和王僧虔没有正面地回答问题，这一点相当聪明，既避免了冒犯领导权威，也没有给人阿谀奉承的感觉。这正是建立在属下准确理解领导背后意图的基础之上的。不知道别人想什么，你说什么也会不着边际。

16.2.4 老板说的总是对的

有一个笑话说，员工做事应有两个原则：第一，老板的话总是对的；第二，如果不对，请参考第一条。虽然是笑话，其实是有道理的。

有一天，儿子拿回来一张语文试卷让我签字，我看到他标的拼音有一个地方不太对，画画的第二个画字标成了轻声，于是我对他说："这个字应该是四声。"但是儿子坚持认为应该是轻声，他的原则就是，这是老师说的，老师说的就是对的。不论我如何解释，并且从网上找到正确答案，证明我说的是对的，但他仍然坚持是轻声。最后我放弃了，因为他说的对，现在给他们判卷子的是他们的班主任，如果班主任说这是轻声，尽管是错的，那也必须是轻声，否则就

得不了100了。我想唯一能解决这个问题的是将来等他长大了以后几个老师交叉判卷，到那个时候他才不得不遵守客观真理，而不能以老师的话为金科玉律。

后来我就想，我们是从什么时候开始不再相信老师说的都是对的，而转而相信客观真理呢？应该就是从交叉判卷开始，因为有了交叉判卷，我们不必再唯班主任马首是瞻，而转而相信客观真理。但多年培养下来的这个习惯，到了职场里又行不通了。因为职场里没有交叉判卷，只有一个老板，所以你需要重新找回你小学一年级时候的感觉，老师说的永远是对的，就这么简单。

后来在网上看到另一篇文章详细阐述了为什么老板说的话是对的，他讲了主要是3个不对称。

1．信息不对称

有人认为老板高高在上，不懂技术，我在底层，技术的情况我最了解，所以我不执行，这是很多不执行员工的借口，这是信息不对称。你站在这么小的一个地方，你在井底，你把井里的东西都看得明明白白的，与公司大局相比也是井底之蛙，你就看那么大的一个天，老板要在上面看井，想要看得明白，放个探测器下去，甚至放个人下去，什么都解决了。但是你看上面看不见，老板想知道就能知道，不想知道是不让这事情阻碍整体的行动，而且在执行细节上，老板会给你空间，比如说走的这个大方向必须是正确的，至于怎么走，走哪条路老板是不会干扰的，这与个人能力展示不矛盾。

2．资源不对称

下级掌握的资源，人力、物力相比公司而言只是一小部分，而老板既然是你的上司，他掌握的人、财、物肯定要比你多，而你如果和公司的执行方向相反，这是分散人财物，是破坏性的，反方向作为的时候会造成一些干扰，甚至是负作用，把原来剩下的那部分资源也给侵占了，失去了效用。

3．目标、任务的不对称

当兵的不能站在高处理解公司的使命，比如说诺曼底登陆，大家都知道历史上诺曼底战役是因为佯攻才成功的，假设有一个军队去佯攻，这支军队担负任务的危险性很大，要么死半个军，要么全被歼灭，结果这个军长想，我要死半个军，甚至是一个军的，我坚决不做无谓的牺牲，我坚决不执行，这个佯攻就没办法打。而“二战”中正是盟军坚决执行了佯攻计划，希特勒虽然也做了大量的间谍工作，但是因为不知道盟军从哪攻，正是这种战略上的“耍花枪”，才为诺曼底战役赢得了最后胜利。

屁股决定脑袋，作为一军之长，他是不会告诉士兵战略是佯攻的还是正面进攻的，就让你打，就让你死半个军，你不站在那个位置不做这样的决定你就不是个军长，战士的任务就是执行，这么大的目标你没必要全明白，就让你干，这种情况在很多公司是很常见的。以我们员工自己的想法，这样损失太大了，这样不赚钱啊，这样不合算啊！是，今年就是不赚钱，就是投几千万，就是建设终端，就是战略性的亏损。假设领导是这么想的，你那种相反的声音，那种不执行的声音恰恰导致整个战略彻底的失败。

所以在执行这件事情上不要怀疑你的上级，即使他有错，但是有概率，我们在部队执行命令的时候，从来没有说我们的命令是100%的正确，但是他如果90%是对的，10%是错的，下面的人乱议论，如果让下面的人来指挥，下个命令，因为他在底层，他30%都不见的是对的，我们只能这么看问题，不能说你的上司有一部分错误，有一些风险你就不执行，你不执行的风险是100%，你执行的风险只有10%，两害取其轻，两益取其重，没有绝对的东西，不要用这些没有绝对的东西来给自己不执行、不作为找理由。有好多人说我工作干不好，我没有努力工作是因为我的领导不好，环境不好，这是典型的找借口。你是你，领导是领导，你执行了命令然后有反对意见都没问题，人是活的，这次打败了，我们总结的时候讲这个都来得及，但绝对不能在打仗的过程当中不执行。

当然在决策前征求意见时，可以畅所欲言，鼓励人尽其言，言尽其意，但如果搞马后炮，会前不说会后乱说就不对。真正优秀的职员，是那种真让你发言的时候敢于提出自己的不同意见，但是如果上级否定了你的建议再让你执行的时候，你必须执行，这才是真正的敬业精神。

众所周知，诺曼底登陆不是盟军一方在打仗，别忘了对面还有几乎相同数量的德军，假如你运气不好，恰好是在德军中间呢，那老板说的还是对的吗？还是90%正确吗？我觉得这个问题要分两个层面看：第一，老板说的至少在战术层面上是对的，如果不对的话，那可能失败得会更早更快，所以在这一点上，老板要你去冲锋陷阵，有他的道理；第二，如果你认为老板是战略上的错误，或者原则错误，这仗根本就不该打，我觉得唯一的选择就是你赶紧脱离这支部队，否则留下来只能是白白送死。

16.2.5 不要给老板惊奇

英语有个谚语：Nobody like surprise（没有人喜欢惊奇）。真的，千万不要给你老板惊奇，哪怕是惊喜也不行。

小王是项目组的工程师，做事情喜欢做到尽善尽美，生怕一个不成熟的作品拿出去给人笑话或者被老板认为自己能力低下。一次老板要他完成一个技术方案，他接受任务之后，冥思苦想，熬了好几个通宵，但总觉得方案写得不完美，还有很多可以改进的地方，眼看就到了必须交稿的日子，而方案还有很多地方没有完成，老板催了好几次，他总说还差一点儿就完工了。到最后终于写完了，交上去之后却又被老板训斥了一顿，发回去重写。为什么？因为有好几个地方，在写的过程中情况已经发生了变化，而且老板也并不喜欢这个方案的整体风格，结果小王不得不从头开始，经过一次次修改，最后才定了稿，时间已经过了最后期限好几天。本来想追求完美的小王为什么最后落得个既不完美又错过了时间呢？

1. 没有人是完美的

工作不是艺术作品，而是多人协作的结果，在工作中比独立完成更重要的是合作与沟通，衡量一个人在工作岗位上是不是成功，是不是能很好地完成工作，不看你是否能“独自”完成任务，而是看你是否“能”完成任务，只要能完成任务，调动多少资源，使多少人心甘情愿地帮助你完成任务，这才是一个人的能力。如果一个人能够画出蒙娜丽莎来，但却无法与别人合

作的话，像这样的人是不适合在职场中生存的。所以做事情千万不要想独自一个人把事情做完美了，有功独享，得到老板赞扬，这样的想法本身是错误的，也是很危险的。

2. 给老板提点你的余地

曾经有一个很有名的漫画师，他不但追求完美，而且认为自己的作品就是最完美的，不喜欢别人修改他的作品。但是他的作品要出版，必须要经过编辑部，而编辑部的人不提意见就原封不动地照搬也是不可能的，这样会被人认为太不负责任。为了解决这个矛盾，这个漫画师每次都在画好了作品之后，在某个角落里画几只很醒目的丑陋的小狗，这样每次编辑部提的意见都是要他把那几只小狗去掉，于是他便欣然从命，把小狗删掉，付诸出版。这样既保住了他的作品不必随意乱改，又满足了编辑部的人"挑刺"的愿望。你的老板既然身为你的老板，就理所当然地有提意见的权力，所以永远别指望你的东西能让老板一次通过，聪明人不但不怕老板提意见，有时候反而故意卖个破绽给老板，满足老板挑剔下属的愿望。当然，凡事有个度，破绽可以留，但不能在关键地方露出破绽，否则那就不是破绽而是能力低下了。

3. 及时沟通避免错误

一件作品从有创意到最终完成，不是一朝一夕之功，在整个过程中，随着时间的推移，形势在逐渐发生变化，有时候即使形势不变化，人的认识和心情的变化也会使人对同一件作品评价不同。如何避免使"惊喜"变为"惊悲"，最简单的一条就是，什么惊也不要有。工作不是玩儿童游戏，工作是很严肃的事情，一就是一，二就是二。老板让你写一个论文，假定需要 30 天的时间，那你应该做的是，只要老板不嫌烦，能沟通的就天天沟通，写完提纲让他看一看，写完第一章再让他看看，哪怕他不看，也要及时汇报工作进度，让他心中有数。软件工程师应该知道，错误有个放大效应，如果一个错误在需求阶段被及时发现，可能只需要几个小时的修改时间，反之，如果一个错误在需求阶段被漏过了，到了最后系统测试阶段才发现，则所有的设计档案要重新修改，所编的代码要重新改写，花费的修改时间会成倍甚至数十倍地增加。及时沟通就是力争能在早期发现错误，花费一些沟通的时间以避免后期推倒重来的损失。

4. 让老板知道你的进度

中国有句俗话：计划赶不上变化，计划在执行过程中可能会出现各种各样意料之外的情况，它们都有可能会在某种程度上改变计划的执行方式以及最终时间，如果老板能够及时知道你这边发生的状况，一方面他会指导你如何应对；另一方面假使真的发生了无法避免的拖延，而他已经事先了解了整个状况，你就不必独自承担那么大的责任。但如果有任何情况你隐瞒不报，一旦出了问题，你老板完全可以说自己一点责任也没有，你没有告诉我啊。最多老板算是个用人不当，把你牺牲当替罪羊了事。所以为了自己的职场安全起见，有问题必须及早通知老板。

16.2.6 和老板建立良好关系

除以上几条基本原则之外，以下这些基本原则可以帮助你和你的老板建立一种积极有效的、可持续的、双赢互利的工作关系，使你从中受益，使你的老板从中受益，并最终使整个部门整个公司受益。

1. 建立积极的工作关系

所谓积极的工作关系，其建立的前提是互相信任。这种信任来自于日常工作中的每一件小事，包括但不限于，说到的事情一定做到，遵守时间，让老板及时知道项目进行中遇到的问题和困难，随时报告项目进展，以及和其他部门协作的情况等。

特别要强调的是，如果你或者你的下属犯了错误，必须及时让老板知道。掩盖事实不但无助于建立良好关系，而且为了掩盖事实而撒的谎最后会给你自己带来很大压力。谎言如果被揭穿，后果非常严重就不用说了，就算谎言暂时没有揭穿，你也总得担心自己说的每一句话会不会自相矛盾。

通常情况下，每天或者每周给老板做一次工作汇报，是建立积极工作关系的第一步。同时，也可以借此机会熟悉老板的性格。老板也是人，也有喜怒哀乐，也有通常人有的烦心事，接触多了就可以了解老板更为人性的一面。

2. 把老板的利益放在首位

时刻记住，你的工作的成败不是你个人的成败，而必须把老板的需求放在首位。要搞清楚，老板目前面临的最大的问题和挑战是什么，你如何能帮得上忙？老板目前最大的担忧是什么，你做什么可以减轻他的担忧？老板的目标是什么，什么优先级最高？然后你也需要按照他的优先级来调整你工作的优先级。做任何事情之前，看问题的角度要从部门和公司的角度出发，而不要只盯着你手头的那点职责。

3. 寻找并专注于老板的优点

人无完人，每个人都有自己的优缺点，不要只盯着老板的缺点，而应多关注他的优点。从心理学上来讲，当我们不喜欢某人的时候，就倾向于盯着他的缺点而对他的优点视而不见，然后更加不喜欢他，结果就会造成一种恶性循环。这种情绪不但会使你工作时心情不好，并且会直接影响你在公司的成功。相反，你应该主动寻找你老板身上的闪光点，并且在适当的场合大胆而真诚地赞美他，比如当老板帮你解决了一个问题的时候，你除了应该表示感谢之外，还应该表扬他判断事物准确等，这样他会觉得自己做的事情有价值，以后会更愿意帮助你。其实，你不是也希望从他那里听到对你的表扬吗？

4. 别指望改变你老板

回想一下，你改变你自己容易吗？除非你自己想改变，否则谁也不可能改变你。老板也是一样，他的管理风格是在和无数人无数事打交道过程中日积月累形成的，决不可能轻易改变。所以，别指望你能改变你老板或奢望他自己会改变，而应当专注于了解他的做事方式。

了解他喜欢什么样的员工，他喜欢经常沟通，还是更希望员工能自我管理，他喜欢开会时有正式的记录，还是更注重在走廊上相遇时的礼节。这些细节都是非常重要的，你了解的越透彻，就越容易和你的老板相处。

5. 会察颜观色

会察颜观色是有效沟通的必备技能。谁都有不想被打扰的时候，比如说你老板正忙着准备项目进展报告，你却拿着一份新技术提议想找他讨论，这显然不合时宜。而且工作中他是

老板，日常生活中他也是凡人，都会有家长里短，家人身体健康问题或者甚至出门汽车被人划了等心情不好的时候，像这种时候，你就不要老拿什么技术问题去烦他。另外，如果老板对某一类似的提议反复做了拒绝或者相同指示的话，就不要再重复提议，而应当了解一下深层次的原因。

6．多向老板学习

尽管很多工程师认为自己的技术能力比老板还要强，但实际上老板有很多东西是可以教你的。简单来说，他能得到晋升就是公司对他的工作成绩和管理风格的认可。如果你想得到晋升，显然可以从他那里学到很多关于如何有效率地工作以及如何做出最有效成绩的方法。所以，多向老板问问题而且多听少说可以有效地建立和老板的良好关系。

7．寻求老板认可

事情做完之后或者进行过程当中，都应该多向老板征求意见，让老板来充当裁判和导师的角色。如果你只做不说，老板不可能猜到你在想什么，所以如果你做了某件事，觉得做的还不错，想要得到老板的认可，就一定要主动把你的工作成绩向老板汇报，并且说清楚这件事情的意义，这样他才能表彰你。说话的时候不要滔滔不绝，要给老板留出充分的时间来表示对你工作的认可和感谢等。

8．珍惜老板时间

一般来说，老板会约定周会讨论业务进展和技术问题。如果时间不够的话，你也可以主动和老板约定一周一次或者一周几次的会议来讨论一些技术细节，这样在会前你可以把遇到的问题总结清楚，在会上老板可以集中解决，这样就省去了你一次次去打断他工作的不便。

9．使你工作和需求与老板的目标保持一致

在向老板提建议之前，要先从全局着眼。你的建议不被采纳可能有很多原因：人力因素、时间因素、目标因素和方向因素等，不要对此心怀不满。同时，在你提建议之前和之后都要保持严格的保密性，这是保护你自己和老板的窍门。

10．不要抱怨

再好的关系也会有不同意见的时候，甚至会有情绪上的抵触。但你要注意，不要让这种不满情绪一直延烧，更不要威胁要辞职之类。你必须明白，老板（至少在这个位子上）是比你有权力的，当矛盾发生的时候，不可能什么事情都得按照你的想法来。

16.2.7 被老板表扬

经过了一段时间的努力工作，你做出了很大的成效，老板把你叫到他的办公室里去，对你说："小王啊，这段时间你的表现不错，希望你再接再厉！"那么你该如何回应呢？光嘿嘿傻笑不是个办法，你说什么话做什么事才会比较得体呢？

这种情况下一般有以下几种处理方法：

首先是要端正心态。领导表扬你，是对你前期工作的肯定，同时也包含着希望你未来能做出更好成绩的期望，所以从心态上来讲，应该戒骄戒躁，保持一种积极稳重的态度。

其次，注意自己的态度，处理好自己的脸部表情。保持谦虚，不要老板一夸就面露得意之色，这是不成熟不稳重的表现。

再次，表示这是自己分内工作，应该做好。

最后，让利于人，说某某也在工作中帮了你很大的忙，整个团队的贡献等。

有时候，老板是在某个比较正式的场合公开提出表扬，并伴随着物质奖励，在这种情况下，你可能需要考虑请同事们一起吃顿饭之类的，一方面表示一下自己对他们日常工作当中对你的帮助与支持的感谢；另一方面也是希望借此机会向他们表达自己以后还希望大家能够继续友好合作的心态。

另外，受到表扬之后，也要注意阿伦森效应。

什么是阿伦森效应呢？阿伦森效应是指人们最喜欢那些对自己的喜欢、奖励、赞扬不断增加的人或物，最不喜欢那些显得不断减少的人或物。阿伦森是一位著名的心理学家，他做了一个实验，将实验人分 4 组对某一人给予不同的评价，借以观察某人对哪一组最具好感。第 1 组始终对之褒扬有加，第 2 组始终对之贬损否定，第 3 组先褒后贬，第 4 组先贬后褒。此实验对数十人进行过后，发现绝大部分人对第 4 组最具好感，而对第 3 组最为反感。

阿伦森效应提醒人们，在日常工作与生活中，应该尽力避免由于自己的表现不当所造成的他人对自己印象不良方向的逆转。同样，它也提醒我们在形成对别人的印象过程中，要避免受它的影响而形成错误的态度。

小刚大学毕业后分到一个单位工作，刚一进单位，他决心好好地积极表现一番，以给领导和同事们留下非常好的第一印象。于是，他每天提前到单位打水扫地，节假日主动要求加班，领导布置的任务有些他明明有很大的困难，也硬着头皮一概承揽下来。

本来，刚刚走上工作岗位的青年人积极表现一下自我是无可厚议的。但问题是小刚的此时表现与其真正的思想觉悟、为人处世的一贯态度和行为模式相差甚远，夹杂着 “过分表演” 的成分。因而就难以有长久的坚持性。没过多久，小刚水也不打了，地也不拖了，还经常迟到，对领导布置的任务更是挑肥拣瘦。结果，领导和同事们对他的印象由好转坏，甚至比那些刚开始来的时候表现不佳的青年所持的印象还不好。因为大家对他已有了一个“高期待、高标准”，另外，大家认为他刚开始的积极表现是“装假”，而“诚实”是我们社会评定一个人所运用的“核心品质”。

实际上，“阿伦森效应”在组织生活中也是常见的。比如以为刚刚毕业的大学生来到政府部门工作，从被保护的环境一下子跳入了一个竞争性的环境，很容易发生“适应不良症”。作为新人，开始时的勤奋工作可能被领导和同事重视并得到赞扬，但日子一长，从局外人逐渐成为局内人，领导的表扬没了，同事的赞赏少了，他会感到不自在，感到自己可有可无，无足轻重，产生挫折心理，所以工作积极性大受影响，没有初来时的那股干劲了，孰不知这种由勤到不勤的转变，对领导和同事而言，同样会产生“褒奖递减”作用，形成“阿伦森效应”，对其表露出不满。这会进一步加剧该学生的挫折感，使其更加懒散，进而大家更没有好印象。这种恶性循环会使这位大学生越来越陷入一种非常失败的关系之中。

16.2.8 被老板批评

1．批评其实是反馈的一种

批评和表扬一样，都是反馈的一种，是同一事物的一体两面。

作为政治上成熟的工程师，应该理性地看待老板的批评，要有闻过则喜的雅量。老板肯批评你，说明他认为你还有改进的余地，如果你是属于那种彻底不可救药的，早就考虑把你开掉，而不是批评这么简单了，批评你还浪费口舌。但也切不可不重视批评，因为每一次批评都是一次督促，如果批评了你不改，日后这会成为你被免职或开除的证据，所以面对老板的批评，一定要确实弄清楚老板批评的是什么，以便于日后改进。

身为人类，每个人都有自尊心，愿意听表扬，不愿意听批评是人之常情。但想一辈子不受批评是不可能的，即使贵为天子，也免不了要受大臣们的批评，更何况是凡夫俗子的我们？而且有时候，被批评反倒是一个提高与改进自己的好机会。我认识的一个同事很会借此机会为自己争取好处，比如说一年一度的年终总结，老板说你今年一年来工作效率低下，影响了项目进度，他不但很虚心地听取了老板的批评，并且征求老板意见如何能够迅速改正这一缺点，是否有什么样的培训或者书籍可以帮助自己改善，结果老板很快给他安排了一次关于时间管理的培训。很多同事抱怨说为什么别人经常有各种各样的培训机会，而自己一次也没有，如果你是这种情况的话，那你真应该注意利用每一次被批评的机会。

心理学家赫洛克曾经做过一个心理实验，他把被试者分成4个组，在4个不同诱因的情况下完成任务。第1组为激励组，每次工作后预以鼓励和表扬；第2组为受训组，每次工作后对存在的第一点问题都要严加批语和训斥；第3组为被忽视组，每次工作后不给予任何评价，只让其静静地听其他2组受表扬和挨批评；第4组为控制组，让他们与前3组隔离，且每次工作后也不给予任何评价。

实验结果表明，成绩最差者为第4组（控制组），激励组和受训组的成绩则明显优于被忽视组，而激励组的成绩不断上升，学习积极性高于受训组，受训组的成绩有一定波动。这个实验证明，及时对学习和活动结果进行评价，能强化学习和活动动机，对工作起促进作用。适当激励的效果明显优于批评，而批评的效果比不闻不问的效果好。

在生活中，有反馈（知道学习后的测验成绩）比没有反馈（不知道测验成绩），学习效果要好得多。而且，即时反馈（每天知道测验成绩）比远时反馈（测验成绩要一周后才知道）所产生的效应（激励作用）更大。

这个效应提醒我们，有效的反馈机制是活动目标达成的必要条件，对于别人的活动必须及时地反馈调节。无论是在管理还是在指导活动中，要多种多样的手段即时地搜集和评定活动效果，如观察交谈、现场提问和效果评价等，然后及时反馈信息，随时调节活动过程，对存在的问题，也不必马上实施惩罚性的方式，而要有针对性地讲解疑难，不使问题累积。

在反馈时，要正确运用鼓励和批评。鼓励和批评都是把握的基本方式，不能偏废。鼓励很重要，但不能夸大其词，对错误和问题的批评要及时、慎重，不能讥笑和嘲讽。要使鼓励和批

评收到实效，关键是理解和尊重，凭敏锐的感觉和沟通的智慧对症下药。

2．承认错误与自我批评

人吃五谷杂粮，也会生百病。人不是神，总有自己的缺点，谁都难免会犯一些错误。当我们犯错误的时候，脑子里往往会出现想隐瞒自己错误的想法，害怕承认之后会很没面子。其实，承认错误并不是什么丢脸的事。反之，在某种意义上，它还是一种具有“英雄色彩”的行为。因为错误承认得越及时，就越容易得到改正和补救。而且，由自己主动认错也比别人提出批评后再认错更能得到别人的谅解。更何况一次错误并不会毁掉你今后的道路，真正会阻碍的，是那不愿承担责任，不愿改正错误的态度。

新墨西哥州阿布库克市的布鲁士•哈维，错误地核准付给一位请病假的员工全薪。在他发现这项错误之后，就告诉这位员工并且解释说必须纠正这项错误，他要在下次薪水支票中减去多付的薪水金额。这位员工说这样做会给他带来严重的财务问题，因此请求分期扣回多领的薪水。但这样哈维必须先获得他上级的核准。哈维说：“我知道这样做一定会使老板大为不满。在我考虑如何以更好的方式来处理这种状况的时候，我了解到这一切的混乱都是我的错误，我必须在老板面前承认。”

于是，哈维找到老板，说了详情并承认了错误。老板听后大发脾气，先是指责人事部门和会计部门的疏忽，后又责怪办公室的另外两个同事，这期间，哈维则反复解释说这是他的错误，不干别人的事。最后老板看着他说：“好吧，这是你的错误。现在把这个问题解决吧。”这项错误改正过来，没有给任何人带来麻烦。自那以后，老板就更加看重哈维了。

勇于承认错误，为哈维带来了老板的信任。其实，一个人有勇气承认自己的错误，也可以获得某种程度的满足感。这不只可以清除罪恶感和自我卫护的气氛，而且有助于解决这项错误所制造的问题。

后来，心理学家布朗通过实验对反馈效应进行了进一步的研究，结果表明，反馈主体和方式的不同对学习和工作的促进作用也不相同。一般来说，自己进行的主动反馈要优于别人的反馈。这给我们的启示在于：

（1）在学习过程中，我们一定要及时地进行自我反馈，避免毫无目的的学习和不知道自己的学习结果的学习方式。

（2）重视别人所作的评价，认真总结自己的优缺点，从而明确自己的努力方向。

（3）正确对待自己的进步，成功时不骄傲，仍坚持继续努力，进展不理想时不要丧失信心，决心迎头赶上。

16.2.9 意见和建议

我们大家肯定听说过唐太宗和魏征的故事，魏征直言敢谏，而唐太宗则能够积极听取，从而使贞观之治传为美谈。如果你能遇上像唐太宗这样的老板，当然是好事，但现实生活中更常发生的情况是老板不是唐太宗，你也不是魏征，或者换位思考一下，把你自己放在你老板那个位置，你敢保证你就能做到像唐太宗那样虚怀若谷吗？人非圣贤，孰能无过，所以在有任何意

见或者建议的时候，首先要考虑的是提意见的技巧和礼节。这样既可以使你的意见得到足够的重视，使问题得到解决，同时又保全了人际关系。

提意见的一个最基本原则是，对事不对人。其实这一点等你自己有一天当了老板，对待下属的时候也是一样。你需要说清楚的是某一天发生的某件具体的事情造成了什么样的不良后果，然后跟老板以商量的口气说看是否可以以某种方式加以改进。比如说老板给你们项目组的工作压力非常重，经常加班加点也干不完，这时候你试图给老板提建议，应该如何提？给老板提建议时候一般要多准备几个选择项，由他来选择一个，这样既保存了他的权威，同时你又得到了实惠。所以，对于上面的情况，你可以说，或者砍掉项目中的某几项不太重要的功能，确保重要功能首先得到实现，或者增加人手等。

意见遭到拒绝时有发生，也是一件很正常的事情，因为有些时候，某个意见从你这个角度看上去似乎很正确，但从老板统观全局的角度来看并不尽然。遇到这种情况时千万不可固执己见，而应学会妥协。妥协不意味着全面退让，而是从积极思考的角度看是否能做出另外一个或者几个可以让大家双赢的选择。如果老板说项目虽然紧，但功能是合同里签订死的肯定不能减，而项目预算有限，人员也不可能增加，在这种情况下，也许你可以考虑第 3 种选择，让老板给大家发奖金等激励士气的行为。

16.2.10 尽快投靠新老板

小赵在一家外企做技术工作，原来的老板对他不错，技术又强，又很能体谅下属。小赵遇上了这样的老板，真有一种士为知己者死的冲动，干工作也非常卖力，加班加点，任劳任怨，本来很有希望在短期内获得晋升。

恰巧在这时候，某一天公司突然宣布要改组，原来的老板调到别的部门去了，小赵的部门换上了一位新老板。新来的老板风格和原来的完全不同，小赵觉得他技术不怎么样不说，还老瞎指挥，明明以前这样做就挺好，现在却要全部推倒重来，浪费时间精力不说，效果还很差。当然小赵明白老板的话是必须要听的，但做起事情来总是阳奉阴违，新来的老板很快提拔了另外一个同事做自己的副手，从此小赵就失去了工作的热情，做事拖拖拉拉，绩效考评也差，他总觉得这个新老板在给自己穿小鞋，最后不得不辞职离开了这家公司。

从一个以前的优秀员工怎么会变成极差员工呢？仅仅换了一个老板就影响了自己珍爱已久的事业，这样的事情可以说在职场司空见惯。我们常说，一朝天子一朝臣，一换老板，风格变了，以前得宠的红人现在失宠了，但不管怎么说，大浪淘沙，总会有一批人在这样的变换中幸存下来，比如五代时期的冯道先后辅佐了 5 个朝代（后唐、后晋、后汉、后周、契丹）的 11 位君主，而且朝朝公卿，担任过 6 个皇帝的宰相，最后全身而退，被称为中国政治史上的一个奇迹。每当政权鼎革之际，冯道都躲到了幕后。而当新政权全面控制局面时，冯道往往会跳到台前来帮助新主子“稳定”局势，理顺方方面面的头绪，使新政权尽快进入角色。身为公司里的一名员工，特别是搞技术的人员，我们根本无法阻挡公司的改组，有时甚至无法预知，那么在这样的动荡之中，我们有必要学习一些冯道的为官之道，尽可能使我们的人生之路不必太过大

起大落。

冯道为官的秘诀之一是及时辨明政治风向，政治立场和效忠对象可以随时改变。后唐明宗死后，愍帝即位，冯道仍为宰相。这时潞王李从珂反于凤翔，愍帝遂出奔卫州。一看愍帝大势已去，冯道“视其君如路人”，亲率百官迎新主子潞王李从珂入朝，接着拥立李从珂为后唐末帝，冯道则继续担任宰相一职，百官在他的带领下迅速各归其位。末帝即位时，愍帝还在卫州，3日后，愍帝被杀。后来每次的政治风云中，冯道都能及时调转船头，短时间内把在前一朝累积的官声名望转换为对后一朝的“赤胆忠心”，尽管这种“赤胆忠心”也许在下一次江山易代时烟消云散。

任何新老板上台，肯定会有和以前不同的政策，所谓“新官上任三把火”。作为员工的小李如果想阻挠这种改变只能是螳臂当车。所以作为小李来说，最佳的策略是不但不阻挠这种改变，并且积极顺应趋势。试想，如果小李能在部门改组后及时和新老板进行沟通，并以身作则地带领全组同事推进新制度，比以前更积极的热情把各项工作做好，他还会成为新任老板的眼中钉，肉中刺吗？

16.2.11 外国老板

2006年，网络上风传的“史上最牛女秘书”信件轰动外企圈，成为了职场里不同文化冲击的典型代表。事情的缘由是这样的。

一天晚上下班后，某知名网络存储设备公司大中华区总裁L回办公室取东西，走到门口时才发现自己没带钥匙，而他的私人秘书瑞贝卡（中国员工）已经下班回家了。L几次联系未果后大发雷霆，于次日凌晨通过内部电子邮件系统发给瑞贝卡一封措辞严厉的英文信并将信件抄送给公司其他几位高管。

> 瑞贝卡，这个礼拜二我刚告诉你，想东西、做事情不要想当然，今天晚上你就把我锁在门外，我要的东西都还在办公室里。问题就在于你以为我随身带了钥匙。从现在起，无论是午餐时段还是晚上下班后，你要跟你服务的每一名经理都确认无事后才能离开办公室，明白了吗？（注：原文为英文）

瑞贝卡收到信后，不但不承认错误，并且回了封咄咄逼人的中文邮件。

> 第1，我做这件事是完全正确的，我锁门是从安全角度考虑的，北京这里不是没有丢过东西，如果一旦丢了东西，我无法承担这个责任。
>
> 第2，你有钥匙，你自己忘了带，还要说别人不对。造成这件事的主要原因都是你自己，不要把自己的错误转移到别人的身上。
>
> 第3，你无权干涉和控制我的私人时间，我一天就8小时工作时间，请你记住中午和晚上下班的时间都是我的私人时间。
>
> 第4，从到公司的第一天到现在为止，我工作尽职尽责，也加过很多次的班，我也没有任何怨言，但是如果你们要求我加班是为了工作以外的事情，我无法做到。
>
> 第5，虽然咱们是上下级的关系，也请你注重一下你说话的语气，这是做人最基本的礼貌问题。
>
> 第6，我要在这强调一下，我并没有猜想或者假定什么，因为我没有这个时间也没有这个必要。

而最令人吃惊的是，瑞贝卡的回信对象不止总裁一个，而是包括了该公司在中国所有分公司的全体员工。随后，这封“女秘书PK老板”的火爆邮件被转发至全国数千外企，在网上引起一阵热议，有的版本后跟帖多达1 000页。事后，瑞贝卡和L先后辞职离开了该公司，造成

了一个双输的结局。

现在我们来分析一下这里面的一些问题。公正地来说，一个巴掌拍不响，在这个事件里身为老板的L在做事方式上肯定有他自己的问题，否则不会让一件小事发展到让自己如此被动的局面。但身为员工的我们更应该多从此事件中看到员工方面的不足，而避免此类事件的发生。

（1）按照外企的通行做法，回复上司的英文邮件时，中国职员一般也应用英文。而瑞贝卡之所以用中文回信，很显然是在故意对抗。外企在中国开公司，为什么不用中文，而必须用英文，这里是否有文化歧视的嫌疑？我觉得一个公司有一个公司的文化，除非法律规定外企在中国开公司工作语言必须用中文，否则任何公司是有权力决定自己公司内部的工作语言的，如果你对此感觉到不高兴的话，那就不要加入该公司好了。

（2）批评性的邮件只应涉及当事人。有人说，总裁L把邮件抄送给其他高管本身已经违反了这一原则，但别忘了，他的邮件里特意提到了“你服务的每一名经理”，从规则上来说这样的邮件显然是有必要抄送给所有高管的。而瑞贝卡的信件并无必要抄送给全中国所有员工，显然是在故意扩大事端。

（3）理解有误。L 的信件里只要求瑞贝卡“跟你服务的每一名经理都确认无事后才能离开办公室”，并没有命令她必须留在办公室等所有经理，实际上这里的“确认”很容易操作，打个电话给老板，问问清楚还有没有什么事情需要留在办公室的就好了。而瑞贝卡把它理解成老板是在“干涉和控制我的私人时间”，以及“要求我加班是为了工作以外的事情”。

（4）语气问题。我想这可能是整个事件中最让瑞贝卡感到不满的地方。凭什么老板就可以以这样的语气对我说话？很简单，凭的就是他是老板而你不是。因为不同的位置承担的后果不同，比如瑞贝卡这样说话造成的后果就是她自己的离职，而老板说话不客气并不会造成他被开除，所以只要不是侮辱谩骂的话，老板有资格训斥员工，而员工则没有这一权力。

除此之外，和外国老板相处的基本原则和中国老板其实差不多。主要差别在于语言和一些文化上的细节。

1．注意沟通方式

一般从欧美来的老板做事比较喜欢直来直去。在汇报工作时，他们喜欢的方式是先讲结果再说原因，这一点和东方人不同，东方人一般喜欢先寒暄几句，把周边情况先介绍清楚了，然后再引出正题。西方老板重视做事和说话时的条理性和逻辑性，汇报工作时先把工作的状况说出来，然后再分析原因。

同时，如果你有任何的要求或者建议，最好不要用暗示或者等待的办法，而必须主动提出，否则如果事情做得不好，他反过来会责怪你为什么没有及时提出。

2．要诚实守信

说到就要做到，做不到就不要说。和中国老板不同，外国老板注重事情的成败更甚于面子，所以如果有些事情是无法做到的，必须当面直说，而不要想着先答应下来，事后再找借口，这样他们会认为你不诚实。如果确有实际情况导致当初的承诺无法兑现，则必须清楚明白地说明是什么具体的事件导致计划无法实现，并采取了什么样的措施避免以后再发生类似的情况。

3．守时

不论是开会还是打电话，必须严格按照约定好的时间准时到达会场或接听电话。和外国老板共事，应该通过日常工作生活把守时培养成一种习惯，习惯成自然，就不会觉得守时有多么困难了。

4．随机应变

文化上互相适应不代表业务上就能完全无缝地结合，有时候由于双方利益不同，冲突与矛盾在所难免。面对这种情况，首先应当阐明自己的立场，让对方了解你的利益所在，了解你坚持立场的原因，如果这样依然不能解决问题，则要考虑第3方案，看是否可以有让双方皆大欢喜的选择。如果连这样选择的余地也没有，那你就参考上文对付中国老板的办法，在人屋檐下，不得不低头，退一步海阔天空，毕竟对方是老板，外国老板也是老板，就按老板的要求做，再不然就只剩一条路可走，换个部门或者离开公司。

5．面子协商

此外，了解一下汀·图梅的“面子·协商”理论会有助于你理解东西方文化的差异，这样在遇到类似文化冲突的时候不会感到太惊讶。

汀·图梅（Stella Tin-Toomey）是华盛顿大学博士，加利福尼亚州州立大学富尔顿分校人类交际研究学教授。她对于东西方文化造成的传播差异作出了有趣的解释。她指出，在每一种文化里都有某种用于协商的“面子”。面子是一个关于在公众中建立的自我形象的隐喻。营造面子是一套操作，包括面子策略的扮演、语言和非语言的动作、自我表现行为、印象管理互动等。霍尔曾经把文化分为“高度语境文化”（high context culture）和“低度语境文化”（low context culture）。汀·图梅则作了更具体的解释。“高度语境文化”如中国、日本、韩国等历史悠久的东方文化，在既定的文化系统中解释信息时，强调意义对语境（context）的关联的重要性，也就是说，任何解释都是联系到语境的解释，从而没有绝对固定的解释，意义依赖于语境而不是被固定于语词。“低度语境文化”如美国、欧洲等西方文化，则是更加重视语言符号本身既定的意义和意思。语言和符号的既定意义在“高度语境文化”中，不是意义的最重要的来源，意义只是隐含在语境和关系当中的。隶属于“高度语境文化”的成员，崇尚集体需求和目标，将它置于个体需求和目标之上。假定说，在一个漫长的过程中，个体的决定都会影响到群体中的每个人，那么，个体的行为就理所应当受到群体规范的控制。是“我们”而不是“我”才代表最高的认同。相反，在“低度语境文化”中，个体的价值、需求、目标均高于群体。个人权利比群体责任更值得重视。“我”自身的认同才是最高的认同。霍尔以日本人和美国人的差异为例，指出“高度语境文化”更多的依靠非语言传达，更习惯于将人群区分为“我们”或“他们”，更关心外来者进入“我们”的圈子时，是否能举止恰当，并不关心外来者究竟如何想、其真实的态度或感情如何。“低度语境文化”则认为，人们所用语言表达的就应是他真实的思想感情，沟通成败全系于能否恰当和准确的表达。因此，在后者看来，“高度语境文化”是含义暧昧的文化。在既定的语词辞典中，很难掌握到确切的解答。这样，分属于两种文化的人之间，存在着大量的误解。问题是，这两种文化如何达到沟通？汀·图梅提出的解决方案是通过对“自我面子关

切”和“他者面子关切”的协商式行为，达到沟通目的。

汀·图梅认为“面子”有消极的和积极的两类。“低度语境文化”追求的是消极的面子，“高度语境文化”追求的是积极的面子。所谓消极的面子包括“挽回面子”（Face-restoration）即要求自我的自由、空间，避免他人侵害个人的独立自治，“留面子”（Face-saving）即表现出对他人自由、空间和某种孤僻的尊重。挽回面子和留面子被定义为“消极”或“被动”性的，是因为这类面子，主要作用是维护自我的最起码的尊严，不具有对他人的控制和支配作用。所谓积极的面子包括“要面子”（Face-assertion）和“给面子”（Face- giving）。“要面子”，表示面子有极高的价值，人们生活在群体当中，有被接纳被保护被包容的要求。要面子被认为是最合理的。“给面子”是鼓励支持并满足人们对被包容被接纳被承认的需求。在心理动因方面，显然消极的面子谋求“个体自治”，积极的面子谋求“群体包容”。不同的文化类型决定了不同的保全面子的方式，从而决定了不同的处理冲突的方式。群体价值导向的高度语境文化，追求积极的面子，处理冲突的策略一般是亲切随和、协商妥协、退缩、避免冲突，也就是通过不断的“给面子”，来满足人们的“要面子”，从而化解冲突。个体价值导向的低度语境文化，追求消极的面子，处理冲突的策略一般是整合的、解决问题式的，或者通过竞争，谋求独断权威。也就是说，彼此都要保全面子、找回面子，只能订立契约；或者订立基本游戏规则，按照规则竞争。只要是按照规则竞争的，无论输赢，都有面子。汀·图梅对于两类面子的分别，实际上揭示出面子（自我的公众形象）是个体在群体生活中的最基本的符号资源。这种符号资源，深刻地联系着个体的心灵、人格结构、关于安全和恐惧的潜意识（消极面子），还深刻地联系着人与人之间建立的权力支配关系、礼仪交换关系（积极面子）。

了解了东西方文化之间对于“面子”的理解的不同，一些在东方文化中看似不可理解的举动就得到了很好的解释，从而可以有效地减少外国老板和本地员工之间的误解。

16.2.12 异地老板

随着经济全球化的发展以及现代通信手段的先进性，异地管理的情况正逐渐增多。如果你正好有一个位于异地的老板，则除了上文所描述的一般性原则之外，还应该注意以下几点。

1. 多沟通多交流

这里说的“多”是指相对于本地管理来说。越是异地老板，越要多主动沟通。因为他本来就在异地，和你的工作环境接触少，不了解情况，如果沟通再少的话，就更无法做出合理决策。一个错误决策伤害的不只是部门或者公司，并且会影响到你个人。沟通方式主要是电话和E-mail，如果条件许可的话，多出差见面沟通更理想。

2. 分清职责

由于地理条件上并不方便大家实时沟通，所以需要在做事情之前，通过几次集中的会议把一段时间内的工作内容确定下来，做到每个人心中有数，以提高工作效率。每隔一段时间，还应该及时碰头，把前一段时间工作中的问题和经验总结一下，该调整的调整，该推广的推广。

Offer

3. 了解清楚考核标准

异地管理最注重就是实际成绩的考核，所以在做事情之前先要和老板沟通清楚衡量自己业绩的指标是什么，然后努力完成要求。

16.3 和同事的关系

办公室政治中重要程序仅次于老板关系的就是同事关系。与老板不同，这些同事是你每日工作中都要接触的，他们可能和你分工不同，但地位相当，有合作但也有竞争。并不是说只要和老板把关系搞好了就可以，有些时候如果和同事关系处理不好也会影响你在公司的命运，最低限度来说，可以让你工作的时候没有好心情。所以，保持一个好的同事关系，不但对你把工作顺利完成有帮助，而且可以使你保持一份愉快的好心情。

16.3.1 合作

假设一个人一分钟可以挖一个洞，那么找 60 个人来用一秒种的时间能挖出一个洞来吗？答案是不可能的。合作是一个问题，如何合作也是一个问题。你需要有计划才能和同事们合作愉快。

有一个寓言故事是这样说的，在远古的时候，上帝创造了人类。随着人的增多，上帝开始担忧，他怕人类的不团结，会造成世界大乱，从而影响他们稳定的生活。为了检验人类之间是否具备团结协作、互助互帮的意识，上帝做了一个试验，他把人类分为两批，在每批人的面前都放了一大堆可口美味的食物，但是，却给每个人发了一双很细很长的筷子，要求他们在规定的时间内，把桌上的食物全部吃完，并且不许有任何的浪费。

比赛开始了，第一批人各自为政，只顾拼命地用筷子夹取食物往自己的嘴里送，但因筷子太长，总是无法够到自己的嘴，而且因为你争我抢，造成了食物极大的浪费。上帝摇了摇头，为此感到失望。

轮到第二批人了，他们一上来并没有急着要用筷子往自己的嘴里送食物，而是大家一起围坐成了一个圆圈，一个人先用自己的筷子夹取食物送到坐在自己对面人的嘴里，然后，由坐在自己对面的人用筷子夹取食物送到他的嘴里。就这样，每个人都在规定时间内吃到了整桌的食物，并丝毫没有造成浪费。第二批人不仅仅享受了美味，还获得了更多彼此的信任和好感。

上帝看了，点了点头，为此感到欣慰。

于是，上帝为第一批人的背后贴上 5 个字，叫“利己不利人”；而在第二批人的背后也贴上 5 个字，叫“利人又利己”。

这个故事告诉我们，合作才是制胜的法宝。

那么，到底如何与同事们合作才能取得成功呢？

掌握 5P 和 3C 法则，可以让你在办公室环境中一展优雅手段，游刃有余地工作。

什么是5P法则呢？5P实际上是一句包含了5个英语单词的句子，这5个单词都是以P开头的，Prior、Preparation、Prevents、Poor、Performance，意思是说，事先的准备可以预防糟糕的表现。与客户谈判时是如此，同事之间的相处也是如此。就像教师上课之前，事无巨细，充分备课，临场时自然应付自如。在人际交往上也需要充分“备课”，从企业的文化、工作氛围，到老板的个人脾性和管理风格，再到前任的工作方式等，通过“备课”，把自己“武装”起来，这样你才能从容面对职场的尴尬和敌意，应付自如，无往不利。运用5P，好好备课，是办公室优雅生存的不二法门。

3C则是3个以C开头的英文单词，分别是Courtesy、Caring、Charming，意思是“以礼相待，以诚待人，以德服人”。三者缺一不可，同事之间，如果能够做到礼貌与关爱，真诚地关心以及尊重他人，相应地也会得到他人的尊重与谅解。3C原则做起来，其实很简单。比如说，多用敬语“您”、“谢谢”，用身体语言表达，如微笑等都是很好的方式。融洽的同事关系，默契的配合，往往在工作中会达到事半功倍的效果。

除此之外，经常使用以下技巧也可以帮你和同事建立一种良好的合作关系。

1. 让同事觉得比你高明

在工作中，每个员工都少不了与同事进行合作。如果老是让同事觉得在你面前受压抑，他不但会抵制合作，当彼此分开独立工作的时候，你需要向他求助，即使他能帮助你，也会找理由推辞。所以，在与同事交往时，即使你比对方优秀，你也要压住自己的风头，让同事心情舒畅，才能保持和谐的关系。

有这样一个案例，韩枫因为资历深，上司让他牵头做一个项目策划。另外2个同事的工作时间虽然短，但业务能力也很棒。为了体现自己项目负责人的身份，韩枫跟伙伴约法三章。

- 有好的创意要及时向他汇报，由他来判断是否可行。
- 每天下班前向他汇报工作进度。
- 他提出的意见，第二天必须整改完毕。

对于韩枫的独断专行，另外2个同事颇有微词。虽然上司让韩枫牵头，但是项目需要3个人来做，只有3个人精诚合作，融合大家的智慧，才会把策划做得最好。韩枫的约法三章，明显凌驾于2个同事之上，似乎他是最高明的，别人都不如他。2个人的心里自然都不痛快，并产生了抵触情绪。

随着工作的开展，韩枫与2个同事的矛盾不断升级。有一次，一个同事想出了一个绝妙的创意，向韩枫汇报后，却没有得到一句肯定的话。韩枫板着脸说他再斟酌一下。后来，这个创意被韩枫改动了一个无关紧要的地方，付诸实施了。还有一次，韩枫让另一个同事修改一处文字，可这处文字并无错误，只是2人的表达习惯不同。同事没改，韩枫责骂同事阳奉阴违，结果2个人争吵起来。

2个同事被韩枫一提醒，果真阳奉阴违，开始出工不出力。完工的期限快到了，策划案还没有完成。韩枫很着急，请求2个同事加把劲。他没想到2人几乎异口同声地说：“我们能力有限。你那么高明，还是你自己加油吧。”

韩枫明白这是借口，也明白2个人想看他的笑话。他找到上司，诽谤这2个同事不配合他的工作。上司经过调查知道了事情的真相，就不再让他负责这个项目策划了。

上司这样对他说：“你太优秀了，他们感到压抑，你还是等下次独自负责一个项目吧。”

韩枫听不出上司是表扬他，还是批评他，但他隐约觉得这是上司的一个借口而已，他因此而不被重用了。

从这个案例可以看出，在与同事合作的过程中，如果你能营造让同事觉得自己很高明的氛围，同事就会与你合作得很顺利；如果让同事觉得他屈从于你，他就会感到压抑，进而产生抵触情绪，并找借口抵制合作。这样，你们的合作就形同虚设，发挥不出应有的作用。也容易让上司抓住把柄，影响在职场中的发展。

2．学会利用出丑效应

出丑效应是指才能平庸者固然不会受人倾慕，而全然无缺点的人，也未必讨人喜欢。最讨人喜欢的人物是精明而带有小缺点的人。精明人不经心中犯点小错，不仅是瑕不掩瑜，反而更使人觉得他具有和别人一样会犯错的缺点，而成为其优点，让人更加喜爱他。

一位著名的心理学教授曾做过这样一个试验，他把4段情节类似的访谈录像分别放给他准备要测试的对象。

在第1段录像里接受主持人访谈的是个非常优秀的成功人士，他在自己所从事的领域里面取得了非常辉煌的成就，在接受主持人采访时，他的态度非常自然，谈吐不俗，表现得非常有自信，没有一点羞涩的表情，他的精彩表现，不时地赢得台下观众的阵阵掌声。

第2段录像中接受主持人访谈的也是个非常优秀的成功人士，不过他在台上的表现略有些羞涩，在主持人向观众介绍他所取得的成就时，他表现得非常紧张，竟把桌上的咖啡杯碰倒了，咖啡还将主持人的裤子淋湿了。

第3段录像中接受主持人访谈的是个非常普通的人，他不像上面2位成功人士那样有着不俗的成绩，整个采访过程中，他虽然不太紧张，但也没有什么吸引人的发言，一点也不出彩。

第4段录像中接受主持人访谈的也是个很普通的人，在采访的过程中，他表现得非常紧张，和第2段录像中一样，他也把身边的咖啡杯弄倒了，淋湿了主持人的衣服。

当教授向他的测试对象放完这4段录像，让他们从上面的这4个人中选出一位他们最喜欢的，选出一位他们最不喜欢的。

想知道测试的结果吗？最不受测试者们喜欢的当然是第4段录像中的那位先生了，几乎所有的被测试者都选择了他，可奇怪的是，测试者们最喜欢的不是第1段录像中的那位成功人士，而是第2段录像中打翻了咖啡杯的那位，有95%的测试者选择了他。

从这个实验里我们看到了心理学里著名的“出丑效应”。就是对于那些取得过突出成就的人来说，一些微小的失误比如打翻咖啡杯这样的细节，不仅不会影响人们对他的好感，相反，还会让人们从心理感觉到他很真诚，值得信任。而如果一个人表现得完美无缺，我们从外面看不到他的任何缺点，反而会让人觉得不够真实，恰恰会降低他在别人心目中的信任度，因为一个人不可能是没有任何缺点的，尽管别人不知道，他心里对自己的缺点也可能是心知肚明的。

所以说人们还是会更喜欢优秀且真诚、值得信任的人，如果一位一直令人尊敬的企业领袖人物当众犯了一点小错误，想想如果你是他公司的下属，你感觉会如何，会因为这个小失误而对他的印象大打折扣吗？当然这一切发生的首要条件就是这个人本身非常优秀和值得尊敬，他至少应该留给

别人非常好的第一印象，否则会适得其反。如何让同事更加欣赏你，并非得高高在上，做个完美无缺的人，有时犯点无伤大雅的小错误，反而更可爱，会让同事更加喜欢你，更加信任你。

3．赞美你的同事

当你作出成绩受到同事由衷赞美的时候，你有什么感觉？你先是觉得很自豪，然后就是觉得自己很高明、很优秀了。所以，反过来，你要学会赞美你的同事，这是让他觉得自己很高明的最有效的方法。

赞美的方法有以下几种。

（1）直接赞美。可以开门见山，真切赞美的主题。比如，“你很棒！”或“你是最优秀的！”

（2）间接赞美。用比较含蓄的方式表达你的赞美之意。比如，“我们这个项目，如果没有小张参与，简直不可想象！”或“你的那个创意，让我想一年也想不出来。”

（3）赞美要有真情实感。赞美同事，真诚是必需的。让同事觉得虚假，就是嘲讽了。

4．虚心向同事征询意见

在合作过程中，你遇到难题或者拿不定主意的时候，虚心征询同事的意见，就会让同事觉得他很高明。首先，你向他请教，说明你认为在这个问题上他比你强。其次，即使他解答不了，说明他仅仅与你处于同一水平上，也没什么丢脸的，况且，他会给你提一些建议，一下又觉得比你高明了许多。

向同事征询意见，态度一定要诚恳。切忌用命令的口气，比如，“说说你的看法！”或“解决一下这个问题！”也不要用调侃的语气，比如，“你那两下子能否解决这个问题？”或“见识见识你的高见。”而应该这样说：“打扰了，请你看一下这几条措施可行吗？”或者，“请你谈谈对这个方案的意见”等。

5．让同事说出你的结论

有时候，为了让同事觉得自己很高明，甚至比你还高明，你不妨引导着同事说出你的结论，让他分享成功的喜悦，盖过你的风头。

比如，当你跟同事讨论某项方案的时候，你已经明了怎样做才是正确的，但你并不急着说出来，而是引导着同事说，比如，“这个问题朝着××方向考虑是不是更好？”同事根据你的提示，兴奋地说出正确的做法，他一定会觉得自己很高明。你也不要怕被同事瞧不起，你只要说“我也是这样想的”，就足以证明你的能力了

6．乐于助人

如果同事向你寻求帮助，应该把它看成是建立友好关系的好机会加以珍惜，而不应总是以自己正忙为借口而断然拒绝。如果自己手头恰好正在非常忙，也要客气地说明状况，看是不是可以稍后再帮。如果自己帮不了，可以建议他去别的可能知情的人那里寻求帮助（但你自己要先确信那个知情的人是确实能帮忙的人，否则你是在给自己找麻烦）。如果确实不想帮或者不能帮，最好能说清楚原因，寻求对方的理解。

7．坚持原则

同事之间的友好合作也并不是说无原则地友好，有一些该拒绝的要求必须坚决拒绝。有些时候这

种要求比较容易分辨，比如某同事要求你做一些违反公司管理制度的事情，你肯定知道拒绝。另一些时候这种要求看上去似乎不那么严重，比如说对方提交给你一份完全没有说明也没有文档的代码，要你接着往下做，像这种情况下，你最好不要轻易接受，因为你是这件事情的接手人，如果你接手了这样不完善的代码，则意味着你要对此承担一部分后果，正确的做法是，请示了老板之后再决定。

8. 认识自己，尊重他人

最后，如果你能在和同事的交流与沟通中自觉或不自觉地经常使用以下 8 句话，你就一定能和同事建立起一种合作与信任的关系。

（1）最重要的 8 个字是：我承认我犯过错误

（2）最重要的 7 个字是：你干了一件好事

（3）最重要的 6 个字是：你的看法如何

（4）最重要的 5 个字是：咱们一起干

（5）最重要的 4 个字是：不妨试试

（6）最重要的 3 个字是：谢谢您

（7）最重要的 2 个字是：咱们

（8）最重要的 1 个字是：您

16.3.2 处理争议

有合作就有争议，争议是工作中很常见的事情，争议处理得妥当，甚至能够化敌为友，争议处理得不好，朋友也会变成敌人。英国心理学家欧弗斯托指出：“说服一个人的时候，开头就让他不反对，是实在要紧不过的事。”要使人不反对自己，先要令人不反感自己。

如果有好的意见却不被人接受或采纳，那么就得想法说服对方。而说服力产生的最大要素，就是要因人而异去使用说服方法。简单地说，就是因人而选择适宜的说词。如果不管对方是谁，都用同一种方法去说服，就很难顺利达成目标。因为对某些人只要解说大意即可，而对某些人就要动之以情，晓之以理。要想说服人就必须巧妙妥善地运用各种方法才行。此外，要能适当地因人而选择说服的方法，自己也必须具备相应的知识和体验。

东汉末年，有一场著名的赤壁之战。曹操统率百万大军准备攻打吴国，当时吴国分为主战、主和两派。诸葛亮为了说服孙权和蜀汉联手抗曹，不远千里来到东吴，企图增加主战派的声势。

这时，吴国的主战论者鲁肃对诸葛亮说：“为了促使孙权下决心打仗，希望你能把曹操的实力说得弱一点。”可是，当孙权向诸葛亮询问曹操兵力时，诸葛亮却说：“据说曹操有一百万的精锐兵力，可是实际上并不止这个数字。所以，在这个时候，求和是比较明智的。”孙权很惊讶地问道：“那为什么兵力比吴国还弱的刘备，敢和曹操打仗呢？”诸葛亮说：“我的主公为了要复兴大汉皇室，所以必须和曹操一战。所谓正义之战，兵力乃是次要的问题。为了吴国的安全着想，我劝你还是谋和。”听了孔明这番话，孙权也立志要和曹操决一胜负。于是蜀吴两国合力抗曹，终于打胜了赤壁之战，而在历史上写下辉煌的一页。

诸葛亮知道孙权是一位英雄人物，所以如果把敌方的兵力说弱了，他不会因此而参加战争，

反而因为敌人的强大，更容易激起他的斗志。由孔明游说孙权的例子中可以证明，诸葛亮“说话因人而异”是成功的。

除此之外，处理职场中的争议还有以下一些基本原则。

1．就事论事

不论是多么大的争议，也绝不应挑战对方的人品、智商等因素，而应回到事情本身来。英国伦敦经济政治学院前董事狄伦多说过：“一个团体或机构中所发生的激烈冲突，往往是因为面子问题引起的。解决任何问题的办法在于把握问题未发生前的契机，并将它消解于无形。”善正者正于始，能禁者禁于微，与其争面子，不如挣面子。适当表达对对方的尊重，你就能够说服对方。

同时，正因为“就事论事”这一原则是如此重要，有时候一些很精明的辩论对手会故意抛出一些看似很愚蠢的话，以故意引诱你发动人身攻击，而你一旦落入这个圈套，就全盘皆输了，哪怕你前面坚持的再有道理，你的言论违背了争论规则，就使你的任何理论都站不住脚。所以，做一个好的辩论者必须有好的心态。特别是同事之间的争论更需注意，因为争论结束之后大家还要在一起共事，所以不能因为对事情本身的观点不同而造成大家无法理智共事的局面。

2．心态开放

经验告诉我们，任何争议观点的背后都有它一定的道理，所不同的只是其强弱程度，所以一个有着开放胸怀的人善于吸收各种观点里有道理的部分为己所用，最终形成一个最完善的意见。有的时候争议发生时，不妨先换位思考一下，从对方的角度看问题，然后结合自己的利益点，看有没有可能提出一个让双方都受益或者双方都能接受的方案。在这里我们要强调的是，双赢不是妥协，不是单方面或者双方面的牺牲与退让，而是谋求集中双方的智慧，在现有方案基础上提出一个更高更好意义上的方案，如果没有这样的方案，则必须坚持自己的原则，并寻求上级领导支持。

3．争论的升级

如果争论双方相持不下，何时应当请老板出来调停是一件很艺术的事情。有的人喜欢一争论就把老板牵扯进来，有些时候这会让老板很被动。因为如果事情本身并不大，自己能解决就解决了，凡事必让老板出面，会让老板觉得你这个人一点办事能力都没有。还有些人喜欢捂着，事情都已经很严重了，也不让老板知道，但纸包不住火，一旦后果严重到老板知道的那一天，恐怕自己就得吃不了兜着走了。什么时候该升级，什么时候该自己控制，要根据自己的经验来判断。

4．留存证据

如果是E-mail里的争论，在发E-mail之前就必须要注意自己不能写任何不得体的话，同时要保留一切与争论相关的证据，以备将来查阅。如果是电话里的或者会场上的争论则需要通过撰写会议纪要的方式留下证据。曾经有朋友就因为完整保留了这样的证据，他可以清楚地说明每一次争论的前因后果，以及相关决策是如何做出的和谁做的等，而保住了自己的饭碗。

16.3.3 沟通

虽然技术工作中有比较多的时间是用于处理客观对象，但如果失去了有效的沟通，一样会一败涂地。沟通的重要性并不因为技术人员水平的高低而有不同。

1．有效沟通避免集体性愚蠢

1999 年，美国宇航局进行的火星气象人造卫星任务失败，就是因为一组工程师使用公里和千克的公制单位撰写程序，另一组却使用英里和英镑的英磅单位运算。结果即使两组人马协作无间，最终还是发生搭错线的乌龙。这个例子正是“满坑满谷聪明人的组织，到头来还是做出蠢事”的写照。

国际知名的未来学者、演说家及管理顾问卡尔•阿尔布莱特提出：“把一群聪明人收编进组织后，结果往往变成集体性愚蠢。”

为了防止出现这种集体性愚蠢，必须注意以下几点。

（1）让每个人都了解企业的理念与价值观。

（2）同舟共济、荣辱与共意识。

（3）人人都做好准备，接受挑战与改变。

（4）组织里的全体员工都愿意付出更多，以换取更大成就。

（5）团队合作。

（6）有效运用组织的知识资产。

（7）人人自觉追求卓越，而不是浑水摸鱼。

2．学会倾听，没有沉默就没有沟通

曾经有个小国到中国来，进贡了 3 个一模一样的金人，金碧辉煌，把皇帝高兴坏了。可是这小国不厚道，同时出一道题目：这 3 个金人哪个最有价值？

皇帝想了许多的办法，请来珠宝匠检查，称重量、看做工，都是一模一样的。怎么办？使者还等着回去汇报呢。泱泱大国，不会连这个小事都不懂吧？

最后，有一位退休的老大臣说他有办法。

皇帝将使者请到大殿，老臣胸有成竹地拿着 3 根稻草，插入第 1 个金人的耳朵里，这稻草从另一边耳朵出来了。第 2 个金人的稻草从嘴巴里直接掉出来，而第 3 个金人，稻草进去后掉进了肚子，什么响动也没有。老臣说：“第 3 个金人最有价值！”使者默默无语，答案正确。

最有价值的人，不一定是最能说的人。老天给我们两只耳朵一个嘴巴，本来就是让我们多听少说的。善于倾听，才是成熟的人最基本的素质。当你能够心领神会的时候，沉默便胜过千言万语。

倾听也是有技巧的，让对方知道你在注意听是很重要的。以下有 10 种增进倾听技巧的方法。

（1）消除外在与内在的干扰。

外在和内在的干扰，是妨碍倾听的主要因素。因此要改进聆听技巧的首要方法就是尽可能地消除干扰。必须把注意力完全放在对方的身上，才能掌握对方的肢体语言，明白对方说了什么、没说什么，以及对方的话所代表的感觉与意义。

（2）鼓励对方先开口。

首先，倾听别人说话本来就是一种礼貌，愿意听表示我们愿意客观地考虑别人的看法，这会让说话的人觉得我们很尊重他的意见，有助于我们建立融洽的关系，彼此接纳。其次，鼓励对方先开口可以降低谈话中的竞争意味。我们的倾听可以培养开放的气氛，有助于彼此交换意见。说话的人由于不必担心竞争的压力，也可以专心掌握重点，不必忙着为自己的矛盾之处寻

找遁词。然后，对方先提出他的看法，你就有机会在表达自己的意见之前，掌握双方意见一致之处。倾听可以使对方更加愿意接纳你的意见，让你再说话的时候，更容易说服对方。

（3）使用并观察肢体语言。

当我们在和人谈话的时候，即使我们还没开口，我们内心的感觉，就已经透过肢体语言清清楚楚地表现出来了。听话者如果态度封闭或冷淡，说话者很自然地就会特别在意自己的一举一动，比较不愿意敞开心胸。从另一方面来说。如果听话的人态度开放、很感兴趣，那就表示他愿意接纳对方，很想了解对方的想法，说话的人就会受到鼓舞。而这些肢体语言包括：自然的微笑、不要交叉双臂、手不要放在脸上、身体稍微前倾、常常看对方的眼睛及点头等。

（4）非必要时，避免打断他人的谈话。

善于听别人说话的人不会因为自己想强调一些枝微末节，想修正对方话中一些无关紧要的部分，想突然转变话题，或者想说完一句刚刚没说完的话，就随便打断对方的话。经常打断别人说话就表示我们不善于听人说话，个性激进、礼貌不周，很难和人沟通。

虽然说打断别人的话是一种不礼貌的行为，但是如果是乒乓效应则是例外。所谓的乒乓效应是指听人说话的一方要适时的提出许多切中要点的问题或发表一些意见感想，来响应对方的说法。还有一但听漏了一些地方，或者是不懂的时候，要在对方的话暂时告一段落时，迅速地提出疑问之处。

（5）听取关键词。

所谓的关键词，指的是描绘具体事实的字眼，这些字眼透露出某些讯息，同时也显示出对方的兴趣和情绪。透过关键词，可以看出对方喜欢的话题，以及说话者对人的信任。

另外找出对方话中的关键词，也可以帮助我们决定如何响应对方的说法。我们只要在自己提出来的问题或感想中，加入对方所说过的关键内容，对方就可以感觉到你对他所说的话很感兴趣或者很关心。

（6）反应式倾听。

反应式倾听指的是重述刚刚所听到的话，这是一种很重要的沟通技巧。我们的反应可以让对方知道我们一直在听他说话，而且也听懂了他所说的话。但是反应式倾听不是像鹦鹉一样，对方说什么你就说什么，而是应该用自己的话，简要的述说对方的重点。比如说“你说你住的房子在海边？我想那里的夕阳一定很美”，反应式倾听的好处主要是让对方觉得自己很重要，能够掌握对方的重点，让对话不至于中断。

（7）弄清楚各种暗示。

很多人都不敢直接说出自己真正的想法和感觉，他们往往会运用一些叙述或疑问，百般暗示，来表达自己内心的看法和感受。但是这种暗示性的说法有碍沟通，因为如果遇到不良的听众，他们话中的用意和内容往往被人所误解，最后就可能会导致双方的失言或引发言语上的冲突。所以一但遇到暗示性强烈的话，就应该鼓励说话的人再把话说得清楚一点。

（8）抓住重点。

找出重点，并且把注意力集中在重点上面讨论问题的细节也许很有趣，可是找出对方话中的重点，并且把注意力集中在重点上面，这样我们才比较容易从对方的观点了解整个问题。只

要我们不再注意各种枝微末节，就不会因为没听到对方话中的重点或是错过主要的内容，而浪费了宝贵的时间，或者做出错误的假设。

（9）回顾与总结。

当我们和人谈话的时候，我们通常都会有几秒钟的时间，可以在心里回顾一下对方的话，整理出其中的重点所在。我们必须删去无关紧要的细节，把注意力集中在对方想说的重点和对方主要的想法上，并且在心中熟记这些重点和想法。暗中回顾并整理出重点，也可以帮助我们继续提出问题。如果我们能指出对方有些地方话只说到一半或者语焉不详，说话的人就知道，我们一直都在听他讲话，而且我们也很努力地想完全了解他的话。如果我们不太确定对方比较重视那些重点或想法，就可以利用询问的方式，来让他知道我们对谈话的内容有所注意。

（10）接受说话者的观点。

如果我们无法接受说话者的观点，那我们可能会错过很多机会，而且无法和对方建立融洽的关系。就算是说话的人对事情的看法与感受，甚至所得到的结论都和我们不同，他们还是可以坚持自己的看法、结论和感受。尊重说话者的观点，可以让对方了解，我们一直在听，而且我们也听懂了他所说的话，虽然我们不一定同意他的观点，我们还是很尊重他的想法。若是我们一直无法接受对方的观点，我们就很难和对方彼此接纳，或共同建立融洽的关系。除此之外，也能够帮助说话者建立自信，使他更能够接受别人不同的意见。

16.3.4 竞争

正如商业社会中合作与竞争是并存的一样，在同事关系中也是如此。竞争与合作缺一不可，如果没有合作只有竞争，大家做起事情来各不相让互相掣肘，最后肯定一事无成；同时如果没有竞争只有合作，则部门内部失去了积极向上的动力，最后也会伤害部门利益。所以一个正常的公司内部，这两种关系必然是同时存在的。

一棵树如果孤零零地生长于荒郊，即使成活也多半是枯矮畸形；如果生长于森林丛中，则枝枝争抢水露，棵棵竞取阳光，以致参天耸立郁郁葱葱。个人的成长是在集体中通过与人交往、与人竞争而成长的，集体的要求、活动与论评价和成员素质等都对个人成长具有举足轻重的作用。良好的集体往往造就心智健康的人，不良的集体往往造就心智不健康的人。

1．理解竞争优势效应

在双方有共同的利益的时候，人们也往往会优先选择竞争，而不是选择对双方都有利的“合作”。

一只河蚌正张开壳晒太阳，不料，飞来了一只鹬鸟，伸嘴去啄它的肉，河蚌急忙合起两张壳，紧紧地钳住鹬鸟的嘴巴。

鹬鸟说：“今天不下雨，明天不下雨，就会有死蚌肉。”

河蚌说：“今天不放你，明天不放你，就会有死鹬鸟。”

两个谁也不肯松口。这时，一个渔夫走过来看见了这种情景，便走过来，不费吹灰之力就把它们一起捉走了。

虽然这只是一个寓言，但是因为鹬蚌相争而被别人得利的事情，代不乏人。它形象地说明

了人们的竞争意识有多么强烈，拼着自己与对手同归于尽，也不想给对方让步。这种现象，被心理学家称为“竞争优势效应”。

心理学上有这样一个经典的实验，让参与实验的学生两两结合，但是不能商量，各自在纸上写下来自己想得到的钱数。如果两个人的钱数之和刚好等于 100 或者小于 100，那么，两个人就可以得到自己写在纸上的钱数；如果两个人的钱数之和大于 100，比如说是 120，那么，他们两就要分别付给心理学家 60 元。结果如何呢？几乎没有哪一组的学生写下的钱数之和小于 100，当然他们就都得付钱。

社会心理学家认为，人们与生俱来有一种竞争的天性，每个人都希望自己比别人强，每个人都不能容忍自己的对手比自己强，因此，人们在面对利益冲突的时候，往往会选择竞争，拼个两败俱伤也在所不惜，就是在双方有共同的利益的时候，人们也往往会优先选择竞争，而不是选择对双方都有利的“合作”。

有这样一个笑话，上帝向一个人允诺说：“我可以满足你的 3 个愿望，但有一个条件——你在得到你所想要的东西的时候，你的邻居将得到你所得到的两倍。”于是这人开始提出自己的愿望，第 1 个愿望和第 2 个愿望都是一大笔财产，第 3 个愿望却是“请你把我打个半死吧!”。

这句话背后的心理活动是，如果把我打个半死的话，那么邻居岂不是要被“完全”打死？如果是这样，那么他从我身上赚到的“便宜”，岂不是要付出生命的代价？虽然为此要被打个半死，但为了不让人白得好处，也是值得的!

这样的笑话并不仅仅是笑话，现实生活中也有不少这样的例子。曾经听别人讲起这样一个令人啼笑皆非的故事，一对夫妻离异，根据法官的判决，丈夫应该把自己财产的一半转让给妻子，因此，丈夫开始出售自己的车、房。为了不让妻子平白无故的得到一大笔财产，丈夫将自己价值几百万美元的车子和房子以 10 美元的“天价”贱价出售，妻子固然没有得利，丈夫也损失了一大笔。

利益冲突会导致人们优先选择竞争，这是不言自明的，但是在有共同利益的情况下，人们是否仍然会选择竞争呢？

战国时，秦昭襄王对范雎说：“天下的贤才武士，以合纵为目标，相聚在赵国，而且要攻击秦国，我们该如何对付。”

范雎说：“大王不必忧愁，让我来破解他们的合纵关系。秦国与天下的贤才武士，并没有什么仇恨呀！他们相聚要来攻打秦国，只是为求一己的富贵。一群狗在一处，卧的卧、立的立、走的走、停的停、不会互相争斗，如果投一块骨头过去，每只狗就起来抢夺，并且互相撕咬，这是什么原因呢？因为那块骨头，彼此都有起了争夺之意。”

秦王于是派范雎带了 5 000 金，在武安大摆宴会，散给合纵之士的黄金不到 3 000 金，他们就互相争斗起来，也不再策划攻击秦国了。

由这个故事可见，即使在有共同利益的情况下，因为利益分配的不平均，以及长期利益与眼前利益的矛盾，人们仍然会选择竞争。

除此之外，心理学家还认为，沟通的缺乏也是人们选择竞争的一个重要原因。如果双方曾经就利益分配问题进行商量，达成共识，合作的可能性就会大大增加。如果在上面的实验中允许参

加实验的两个人互相商量，或者两个人对对方的选择有充分的把握，结果必然会是另外一个样子。

2．职场竞争的基本原则

千辛万苦，担任项目组高级工程师的小孙赢得了上司的赏识和同事的拥戴，眼看就要成为部门主管，可以率领一个团队了。可惜，天不随人愿，公司高层领导不知出于何种考虑，高薪从别处挖来一名业内精英小赵进入部门工作。而且，小赵很快就熟悉了公司环境，利用原有的客户资源，很快就有出色表现。每个人都知道，部门主管将由这两人的角逐结果产生。小孙自己更清楚，多年的奋斗和拼搏面临强大的挑战。如何迎战？怎样取胜？败了，如何自处？太多问题困扰。

职场是冷酷无情的。我们常常会遭受到一些突如其来的打击。就像小孙一样，辛苦打拼，正当要有收获时，莫名其妙要经历一场 PK。这多少有些不公平。小孙很可能会想不通。我们在这里不追究公司这样安排的原因，我们要做的是抓住竞争的机会，去战斗。

（1）知难而上。

必须要迎战，这是毋庸置疑的。就算知道这是一场势均力敌的竞争，自己胜算并不大，甚至知道这是一场不那么公正的比赛，也要去争，去斗。战而不胜，虽败犹荣。

（2）要看到自己的优势。

我们知道，小孙和小赵都是业务出色的精英人才，这方面可能不分上下。可是，要成为一个团队的领导者和管理者，是需要与团队有长期的结合、配合、磨合过程的。小孙在同事中有普遍威信，显然具有优势，这是应该充分利用和发挥的。

（3）反省自己的不足。

在看到自己优势的同时，也应该认真思考一下，公司如此安排是出于何种考虑？既然自己已经在业务和领导力方面具备了主管资格，公司为何还要安排这样的 PK？是不是自己哪里做得不好存在问题？可以跟领导和同事多沟通一下，帮助自己找到问题的所在。竞争的过程也是一个难得学习的过程，在高压力高强度下，人是会迅速成长的。

（4）正面竞争，妥善处理与竞争对手的关系。

在竞争上岗的过程中，也要注意方式方法，不要把关系搞僵。毕竟存在竞争失败的可能，如果还想在这家公司继续做下去，就必须给自己留条后路。

（5）考虑到失败的结果。

如果竞争结果是失败了，经过这样一场竞争以后，参与竞争的双方可能会成为很好的合作伙伴，也可能就很难开展工作了。如果实在合不来，就应该考虑申请一下公司内部调整，或者考虑跳槽了。

16.4 和客户的关系

软件工程的项目种类不同，有些项目，如系统集成等会比较频繁地和客户打交道，而另一些制造类型的项目，如为手机开发软件等，工程师可能并不直接接触最终客户，但这并不意味着这个项目就没有客户。广义上来讲，对于任何项目来说，从谁手里接受的需求，谁就是客户。

客户可能是本公司以外的其他公司的，也可能是本公司内部其他部门的，甚至有可能就在同一部门内部。软件工程的成败在很大程度上取决于与客户的关系。

16.4.1 态度决定成败

做软件项目的时候，很多项目经理以及工程师经常会遇到客户修改需求的时候，即使是已经签过字的需求，也有可能客户会提出要改某个部分，甚至修改基本框架。遇到这样的问题怎么办呢？

在某公司的项目管理课上，小李、小王等人正在七嘴八舌地议论纷纷。原来，大家正在讨论小王的公司最近遇到的2个颇为有趣的项目。

据小王介绍，这2个项目分别由2个项目经理来担任。其中，项目经理A属于“谦虚”型，对于客户提出的问题，无论大小都给予解决，客户对此非常满意，然而，项目进度却拖得比较长，而且，客户总想把所有的问题都改完再说，项目已经一再延期。

相比之下，项目经理B显得稍有些“盛气凌人”，对于客户提出的问题，大多都不予理睬，客户对此不是很满意，不过，该项目的进度控制得比较好，基本能够按期完成项目。

话刚一说完，小李就抢着说：“A比较像我，一般在和我的一些战略客户打交道的时候，我基本是有求必应，与客户的关系处得如鱼得水，这样做肯定不会错。就像前天我连合同都写错了，找到客户，人家二话没说就同意改了。你说如果是B的话可能吗？”

小王对此则不以为然，“对项目经理来说，成本、质量和时间是最为重要的3要素。与客户的关系当然很重要，但也要全盘考虑项目的各要素。对于用户的要求，应该在有限的范围内给予解决，但不可以做出太大的牺牲。一味的迁就用户将会使整个项目失败。”

小林接着小王的话说：“当前，国内的项目一般情况下是由销售处面签单，再由项目经理接手后续的工作，因此客户关系多在事前已经搞定。发生新的情况后，可以由公司的公关部出面与客户进行协调，项目经理可以在此过程中坚持一下原则，与公司的公关部一个红脸，一个白脸，唱出一出好戏。”

小赵反驳道：“不管怎样，客户才是第一位的。客户可以给你带来收入，也可以给你带来更多的客户和工作，有什么道理不多配合一下他们呢？说实话我对B的做法蛮欣赏的，可惜行不通。因为客户是上帝，如果照B的做法，后果会造成做一次项目丢掉一个客户，太不划算了。”

对待这样的问题，不同性格的人会有不同的处理方法，不同的处理方法也各有优缺点，但在对待客户的态度上不能傲慢自大，目中无人，而应该谦虚谨慎，不卑不亢。多听少说，既不轻易答应客户的漫天要价，也不断然拒绝，学会在适当的时机说：“我们回去研究讨论后再确定。”

面对客户的需求变更，应该站在更高的层面去看待问题，不能仅仅局限于一个简单的需求变更，更应该多方面的衡量利弊最终决定需求变更是否应该满足，甚至用一个简单的需求变更来回绝一个复杂的需求变更，要懂得利益均衡。即便不满足客户需求，也不能不大搭理客户，这是绝对错误的。

做项目前，首先要给客户定位，客户的定位决定了你后期的做法，这其中有些恶劣客户，那不需要考虑。在职权范围内，如果可以满足客户的，应该尽最大可能去满足。超出职权的，应协调更多资源确定是否满足客户需求的变更，这会与公司的策略、客户的长期发展等多种因素有关，自己不要擅自决定，也许公司为了做个重点项目赔钱也做，为的就是市场效应。这些因素都应该综合考虑。

无论需要变更满足与否，态度是第一位的。生意不成人意在，需求变更不成，但客户关系不能毁在自己手中，这种事情已经屡见不鲜了。

1. 要分清楚客户和用户

客户并不必然等于用户。举例来说，我们为一家大型图书馆设计借书系统，读者是这个借书系统的主要使用者，但他们并不是这个系统的客户，因为他们不直接为这个系统买单，如果我们设计的系统完全从读者角度出发，远远超出了项目预算，真正的客户图书馆的管理层拒绝接受，这就不能称之为一个合格的系统。所以，要设计一个合格系统，首先要分析清楚谁才是这个项目里真正的客户，只有完全满足了客户需求的项目才是成功的项目。

2. 要永远尊重客户

要坚信，客户永远是对的，客户永远比我们聪明。任何初听上去很荒谬的要求，那只是因为你不了解他的业务模式。作为一个已经在该领域工作了若干年的资深人士，客户对他领域的理解肯定要胜过我们。我们的长项只在于软件开发领域，所以我们所要做的不是去主动找出客户现存模式中的问题，而是提供参考信息，引导客户自己来发现他系统当中可以改进的地方。我曾经参加过的一个项目，我们为它设计了一整套的条码打印和识别系统，这样可以很方便地查询和搜索书籍，但客户要求仍旧保留原有的编目系统，最后，事实证明客户提的要求是正确的。

3. 合理安排工作顺序

理解客户的新需求，合理适当地满足，既不完全拒绝，也不完全接受，而是根据项目资金，人员情况进行合理安排。要看一下客户的需求是全部合理，还是全部必需，还是都非常迫切。迫切且合理必需的，就安排人手做；不迫切但却合理必需的，就考虑有时间时再安排人手来做；如果合理，但却不是必需的，一般就不做了；如果是不合理的，自然要说“不”了。

如果某些需求并不是现在必需的，可以对客户讲：“这需求比较好，只是现在我们的重点是……，先保证项目的进度，这样就有更充足的时间来做这个需求了。你所提的需求我们会在下期版本或升级时考虑。”这样既尊重了客户，又肯定了他的提议，只是现在的工作重点不是这个，项目的进度又是工作的重中之重，一般情况下没有人会毫无理由地反对。

对固执的客户有时需要反其道而行之。首先肯定客户的需求，再扩充客户的需求，然后告诉他要先做哪些，再做哪些，需要多少人手、多少时间才能完成，项目进度要推迟到什么时候，要增加多少人员才行，这样的话要双方老总重新签字才可以开始做。这样的客户相当一部分都是底层无关紧要的人员，为了表现自我，而提出不可理喻的需求。比如，他会提你所采用的编程语言为何不是他熟悉的工具，你的界面为何不是他之前用过的软件的界面，为何不采用流行的技术等。这种人在公司内部往往会大声说话压倒人，来展示他有一定的影响力。即使对这种人的意见不理睬，但还是应该客气地告诉他，让他自己觉得这样做的结果非常不可行。这种人一般对其表示一下尊重，请上饭桌一两次，再展示一下团队的真正实力，让其了解自己无畏是因为无知，基本就可以摆平了。

最后，要把所有的商谈结果整理成正式文档，告之客户哪些需求以后再做，并向其主管与直属上司表扬其人其事。这样，一定可以维持一个良好的客户关系。

16.4.2 谈判风格

与客户沟通中另外一个很重要的环节就是谈判。谈判发生的环节不只是项目开始之初，而实际上贯穿整个项目始终。

在项目初起时，要谈判项目需求；在项目进展过程中，客户经常会提各种各样的需求变更要求，这时候也需要谈判；项目收尾阶段，客户验收同样需要谈判。

在开始谈判之前，先了解一下你是什么类型的谈判选手。

有这样一个小游戏：想象一下10个完全陌生的人坐在一张长桌子的两侧，面面相对，你是其中之一。这时候进来一个老师说："我发令之后，前两名能说服坐在你对面的人主动站到你的椅子背后的人每人可以得到1 000元。"

你是10个陌生人之一，你盯着对面的人看，他同时也在看你。你们两个都清楚，前两名能说服对面的人站到自己背后的人可以得到1 000元，而其他人只能得到0元。

你会如何处理？你需要快速行动，因为所有其他人也在思考如何行动。

注意你脑子里产生的第一个念头。如果你有了决定，再往下读。

第1种选择是，什么也不做，你觉得这可能是一场骗局，或者如果谁傻乎乎站起来跑到椅子背后的话看上去很丢人。这种反应是"回避型"的典型表现，回避型的人不喜欢人际冲突，不喜欢有输赢胜负的游戏，他们努力避免可能产生不和的情况，不论是在职业选择上还是个人生活中都喜欢保持一种和平与宁静的气氛。

第2种选择是，努力说服坐在你对面的人站到你的椅子背后来，并答应分给他500元。但是你考虑过没有，他很有可能会要你做同样的事情？你会立刻同意他的要求，站起来跑到他椅子背后去吗？如果你倾向于立刻同意并开始跑动，那么你属于"妥协型"，你更关注于和他人维持一种富有成效的关系，喜欢通过讨论达成一种对双方来讲比较公平的协议，但在紧急情况下可能会屈从于一个只对单方面有益的解决方案。你既不非常贪婪也并不非常害羞。

第3种选择的处理方式比妥协型更简单，几乎不用怎么说服就站起来跑到对方椅子背后。你基本上不谈判。按照规定，你对面的人会得到1 000元。你做了事情，但得好处的是一个陌生人，他拿走了1 000元，而你只得到0。这是"顺应型"的典型表现。顺应型的人喜欢通过帮别人解决问题来处理人际间的冲突。如果你对面的那个人是和你性格相似的人，他会分给你一半钱并感谢你的配合。但如果他们是贪婪而自私的人，那很可能你什么也得不到或者只得到一丁点补偿。

第4种选择是"竞争型"。正如你预料到的，竞争型的人喜欢获胜，并且愿意在游戏中冒险去尝试比别人赢得更多的金钱。一个竞争型的人可能会向对方大吼："快点，站到我背后来，我会和你分钱！"然后竞争型的人就坐在那里等着。他可不像妥协型的人那么好说服，他会和对方反复争论到底应该是谁站起来站到对方背后去。更有趣的是，竞争型的人很有可能会撒谎，说自己的脚扭了，不能动。如果这一招奏效的话，竞争型的人会处于非常有利的位置，他可以来决定到底如何分这个钱，这样他就完全控制了谈判局面。很多成功的律师都属于这种类型，他们喜欢获胜的感觉。

第5种选择是最富有想象力的选择。你立刻起身离开你的座位，并向你对面的人大喊："快

站起来，我们一起站到对方椅子背后去！我们每人都能得到 1 000 元！”如果你们的动作足够快的话，这招是可能成功的。这就是“合作型”的表现方式，他们不去探讨在两个人中间如何平分 1 000 元的问题，而更专注于如何使对方都能得到 1 000 元。这是最难实现的类型，但也是在复杂谈判中最有效的方式。但如果时间非常紧迫或者对方故意要耍强硬，这一招也可能不奏效。

5 种类型没有对错之分，因为不同的场合所需要用到的技巧不同。你了解了你是哪种类型的人以后，在遇到相应情形时可以有意识地纠正自己在某项上的弱势。

这 5 种类型的人都可以在下面这个图形里找到自己的位置，这个图形叫做“TKI 冲突模型”，如图 6.3 所示。横轴代表对手的利益，越靠左侧表示你越不关心对手的利益，相反，越靠右侧表示你越关心对手利益；而纵轴代表你自己的利益，越靠上方表示你越关心自己的利益，相反，越靠下方表示你越不关心自己的利益。举例来说，如果你只关心自己的利益而不在乎对手的利益，那你会在纵轴上处于很高的位置，而在横轴上位于很靠左的位置，所以你会是一个竞争者（Competor）。

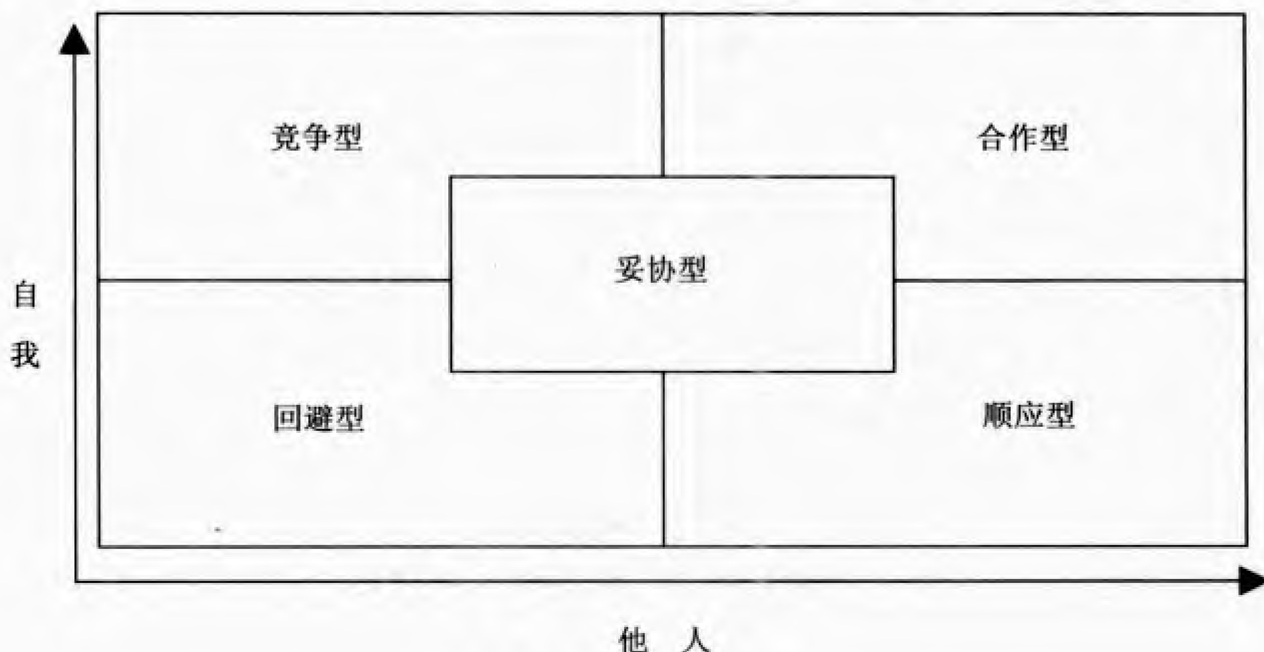


图 16.3 TKI 冲突模型

1．回避型

回避型的目标是拖延或者避免冲突。表现方式是不强硬但也不合作。

作为回避型的人来说，他们不直接追求自己或他人的关注，也不着力解决冲突。回避型可能采取的形式有，很外交地规避问题，推迟事宜直至更合适的时间或者简单地从具有威胁性的情境中撤出。

在以下情形中可以采用回避型的谈判策略。

（1）当事情琐碎，仅仅是一时重要或其他更重要的事情迫在眉睫时。

（2）当你认识到没有机会满足你的关注时，比如说当你的权力较低或对某事灰心丧气，但它又是你难以改变的时候（如国家政策、某人的个性结构等）。

（3）当直面冲突所带来的潜在伤害大于解决问题所带来的好处时。

（4）当想让人们冷静下来时，将紧张降低到能够进行生产的程度，重新获得判断事物的方法并冷静下来。

（5）当搜集更多的信息优于立即决定所带来的好处时。

（6）当他人能更有效地解决冲突时。

（7）当问题看起来有分歧或预示着其他更基本的问题时。

2．妥协型

妥协型的目标是找到中间地带。表现方式是在强硬性和合作性上均居于中间位置。

妥协型的目的是找到能部分满足双方的、有利的、双方均能接受的办法。它居于竞争型和顺应型的中间。妥协型放弃的多于竞争型，但少于顺应型。同样它处理问题比回避型更直接，但不如合作型探索得深入。妥协型可能意味着折衷，相互妥协或者寻找一个快速、中间的立场。

在以下情形中可以采用妥协型的谈判策略。

（1）当目标虽然中等重要，但不值得付出努力或因使用更强硬的方式而引起潜在的不和时。

（2）当两个势均力敌的对手均致力于相互排他的目标时。

（3）当想要积极地暂时解决复杂问题时。

（4）当想要在时间压力下达成有利的解决方案时。

（5）当合作或竞争不成功时，作为备用方法。

3．顺应型

顺应型的目标是屈服。表现方式是不强硬但合作。

顺应型是与竞争型相对而言的。顺应他人时，个体忽视自己的利害关系，以满足他人的关注。此种模式含有自我牺牲的成分。顺应可能采取的形式有，无私慷慨或慈善义举，在不情愿时仍能服从他人的命令或屈从于他人的观点。

在以下情形中可以采用顺应型的谈判策略。

（1）当你意识到做错时，允许他人发表更好的意见，向他人学习，并且表现出你通情达理。

（2）当问题对他人比对你自己重要得多时，满足他人的需要，并且以此作为良好姿态来维系合作关系。

（3）当为了便于日后解决问题而建立良好的社会信誉对你很重要时。

（4）当你在不如别人和失败的情境下，不断的竞争将只会给你的事业带来伤害时。

（5）当保持和谐和避免不和尤为重要时。

（6）当想让下属尝试并从错误中学习来帮助他们在管理上发展时。

4．竞争型

竞争型的目标就是要获胜。表现方式是强硬且不合作。

竞争型的人以他人的代价来追逐自己的关注。这是一种权力定位的模式，在此模式下，人们使用一切看似恰当的手段以处于优势地位，如争辩能力、头衔或经济制裁。竞争型可能意味着维护自己的权力，捍卫他们认为正确的立场，或仅仅是设法取胜。

在以下情形中可以采用竞争型的谈判策略。

（1）当迅速果断的行动至关重要时（如在紧急情况下）。

（2）当在重大问题上需要采取不受欢迎的措施时（如削减成本，实行不得人心的规章制度、

Offer

纪律等）。

（3）在处理对于组织福利至关重要的事情且当你知道你正确时。

（4）当为了防范利用非竞争性行为的人时。

5．合作型

合作型的目标是找出一个双赢的解决方案。表现方式是既强硬又合作。

合作型是与回避型相对的。合作型涉及一种努力，即试图与他人合作以找到能充分满足双方关注的解决办法。这意味着深入研究问题以识别出两人的潜在关注，并且找到能满足双方关注的办法。两人间的合作可能采取的形式有，探索分歧以相互学习对方的见识，一致解决某个情形，否则，它将引起他们争夺资源或发生冲突；形式还包括尽力找到创造性解决人际问题的方法。

在以下情形中可以采用合作型的谈判策略。

（1）当两组关注均重要得难以妥协，想要找到一个协调整合的解决办法时。

（2）当你的目的是学习时（如检查你自己的推断、理解他人的观点）。

（3）当想融合对问题角度不同的人的见识时。

（4）当想通过将他人的关注纳入共识，以获得承诺时。

（5）当为了克服已经干扰人际关系的情绪时。

16.4.3 谈判技巧

1．谈判之前要有准备

准备的内容包括，了解清楚己方的底线、分析谈判对手的特点，等等。了解了你的谈判风格之后，最好也能了解清楚对方的风格，这样在谈判过程中就能有的放矢，最终达成双赢的局面。

2．选择好谈判地点

在谈判地点的选择上，最好选择自己熟悉的环境。心理学家泰勒做过一个试验。他把一群大学生，分成几个小组，让他们分别讨论学校 10 个预算削减计划中哪个最好。每个小组里，既有支配能力（即影响他人能力）高的学生，也有支配能力低的学生。泰勒安排一半小组在支配能力高的学生寝室里，让另一半小组在支配能力低的学生寝室里。泰勒发现，讨论的结果，总是倾向寝室主人的意见，即使主人是低支配力的学生。

试验结果发现，一个人在自己熟悉的环境里比在别人熟悉的环境里更有影响力。在交际活动中，选择熟悉环境是一种蓄势方法。如果你想要提高自身的影响力，就应该把交际场合尽量安排在自己熟悉的环境里，从而能够借助熟悉环境来为自己增添力量。这个熟悉环境，可以是自己家、自己办公室，也可以是自己居住的地区、经常去的公园，等等。

当然，有些时候你的竞争对手也会意识到谈判地点对谈判结果的影响作用，那么对于你来说，谈判地点的第二选择就绝不能是在竞争对手那边。

3．谈判过程中要小心泄密

不光是谈判开始后，在平常和客户聊天时，都必须注意保密。因为我们和客户之间毕竟是

不同的群体，有各自的经济利害关系，不该说的话不能乱说。这些话包括，谈判的底线、技术手段、实施细节和公司里的人际矛盾等。

4．小心谈判收官时的陷阱

谈判有时候会是一个漫长的过程，在任何一个阶段都不应当掉以轻心，特别是在谈判快要结束，主要议题基本都已谈妥的情况下，有时候客户会忽然提出一些额外加码的要求，这时候尤其要格外小心，因为很多人觉得大方向都谈拢了，一点小要求不过分，但有时候谈判的利益差别就在这些小要求里。

有一个关于“诱敌深入法”的有趣实验。人们对报纸售价涨了50元或汽车票由200元涨到250元会十分敏感，但如果房价涨了100万元甚至200万元，人们都不会觉得涨幅很大。人们一开始受到的刺激越强，对以后的刺激也就越迟钝。一个人右手举着300g重的砝码，这时在其左手上放305g的砝码，他并不会觉得有多少差别，直到左手砝码的重量加至306g时才会觉得有些重。如果右手举着600g，这时左手上的重量要达到612g才能感受到差异。即比前一种情况要多给一倍以上的刺激才会有所反应。所以要想辨别出刺激间的差异，刺激总量越大，其差额也必须越大。

客户有时候从一开始提出一个令人难以拒绝的优厚条件，等谈判基本结束后再指出一些不好的细节并希望你接受。如果你被一开始的优厚条件所诱惑，对后来才知道的不好的部分也就会较轻易地接受了。所以在谈判时一定要注意这一点，不要轻易上当。

5．谈判之后要有记录

最后，谈判结束必须要有书面记录，而且最好对双方都有约定性的要求。这样谈判的结果才会使双方都有履行合同的动力。

16.5 和其他部门的关系

在公司里做事，仅处理好和自己老板的关系，和自己部门同事的关系以及和客户的关系还不够，同时还必须考虑和其他部门的关系。因为很多事情是需要其他部门来配合完成的。

16.5.1 和业务部门之间的关系

在数学中1＋1＝2，但把它用在团队协作上，那就不一定是2，有可能是3，或者是10，但也有可能为零或负数。

公司里各部门像一座座孤岛，只顾自家门前雪。一线同事在前线拼得“你死我活”，好不容易搞掂客户，没想到却“后院起火”。在公司里，员工和中级主管花在内部沟通的时间大约占其工作时间的40%～50%，而对于高层主管，这个比率会更高。如何提高公司内部沟通的有效性以改善运营效率呢？怎样才能打破门墙，克服部门间的沟通障碍，让员工发挥1＋1＞2的功效？现在，是直面这些严峻问题的时候了。

1．跨部门合作的特点

（1）合作双方工作在不同地方，对项目沟通造成一定影响。

（2）合作双方隶属于不同的部门，双方的项目开发流程可能完全不同，在项目执行过程中需要考虑到这个因素。

（3）合作项目需要双方共同完成，如果一方的工作进度出现延误，那么整个项目的进度都会受到影响。

2．跨部门合作的障碍

除了上述客观特点之外，跨部门合作有时候还会遇到一些人为的障碍，具体表现为：

（1）表面上积极，实际上得过且过。预约的时候，积极相应，沟通的时候，也承诺积极配合，但是一谈到具体实质工作，特别是需要做出改变或工作时，他们就会找到各种听起来合理的理由。这类部门在不影响自己利益的前提下还算配合，但是要想让他为了公司利益而多承担一点工作或自残一点，是很难办到的。这种情况的主要原因是完全以部门利益为出发点。

（2）能推就推，能拖就拖。很多部门会以各种理由推托沟通，从内心就不愿意了解更不愿意配合部门外的非常规工作。即使勉强接受，但是效果也难以达到预期目的。

（3）在沟通前就已经有了自己的立场，不予配合。很多部门在你还没开口前就把你定位为其布置任务、寻找其工作漏洞的角色，所以一开始就抱着如何逃脱责任的心态。

3．建立跨部门合作的文化

（1）在日常的工作中和各部门管理人员建立良好的人际关系。在双方协调合作中保持积极的态度，经常换位思考，为对方的工作提供很好的配合。中国有句俗话：将心比心。当彼此之间能真诚相待、相互支持，如果你遇到困难或需要对方帮助时，一般情况下不会被拒绝，那么和各部门管理者都能建立相对良好的关系时，其下属人员在参与项目工作时自然也不会太怠慢。

（2）制定项目说明，并准确描述各部门职责及其对整个项目的贡献性。项目说明既起到统筹计划的作用又起到沟通的作用，各部门从中了解自己的工作范畴及需要提供的人员配置，并能根据它预测可能遇到的各种问题。同时，给各部门说明各自的贡献性，可以激发大家的参与热情，防止推脱。

（3）制定项目进度表并按期汇报。向上级及时汇报工作进度，其实质是起到公开监督的作用，通过项目进度表，领导可以一目了然地了解项目进展状况。无论哪个关节出现问题都将导致项目的整体滞后，谁都不会想当出头鸟，所以谁都不会让自己成为拖项目组后退的人。

（4）项目终结后及时提供总结报告。凡事都要有始有终，项目报告既是对整个项目的一个总结，也是对各部门配合工作的评述，更是为相关工作总结经验教训，以便在以后的工作中能更加顺利、有效地进行。

4．建立跨部门合作的制度

对于经常有成立项目组必要的企业，应该制定一套完善的项目管理制度。一套良性循环的项目流程图可以帮忙项目组步步为营、稳步前进，及时发现问题、及时处理。在项目管理中可以分为硬件管理和软件管理两部分。

硬件管理包括项目组成员的组合评估、各成员项目工作评价表。这部分管理实际上是为了“做正确的事”。

软件管理包括项目执行中运用的专业工具、项目笔记、阶段成果确认、工作会议纪要等，这些内容都属于基础的方法论，也就是“正确地做事”。

没有规矩不成方圆。在一个严格的管理制度监控下，可以避免很多不必要的失误，减少很多无效的拉锯战。而一个正效应的亚文化则在完善的管理制度上，对整个项目管理锦上添花。

16.5.2 和服务部门之间的关系

在企业中，除了和业务部门之间的跨部门合作外，日常工作中还经常要和各个服务部门打交道，这些服务部门包括后勤部（IT 部门）、秘书、人事部门等。如果没有他们的积极配合，很多事情做起来也会不顺利。

1．注意按流程办事

每个部门都有自己的服务规范，在要求别人配合的时候，要问清楚对方需要通过什么样的手续。该填写的表格要认真填写，并尽可能备齐一切相关资料，然后再去找对方。

2．报告—联络—商量

报告、联络和商量是日本企业管理中最为普通的法则。无论是生产型的企业还是非生产型的企业，员工从进公司的第一天开始到离开公司，都会听到上司或社长嘴里常说的这句话。而且，在日本企业里，只要哪个下级挨批、挨骂，一定会听到“你为什么不报告？你跟他联络了吗？你找他商量了吗？”可见，报告、联络和商量在日本企业中何等重要。

单从报告联络商量这三个词汇来看，它不仅不是日本企业的专利，就是在我们的日常生活中，也是每天都能接触到，并在进行着的基本内容。而且，这6个字，谁都明白，谁都能做。但是，要持之以恒地做下去，并确实做得合符规范，它就像军人从参军第一天就开始学习“立正、稍息”这两个动作一样，不仅极难，而且还会有意想不到的好处。

3．搞好和具体负责人员的关系

平时见面主动打招呼，逢年过节请他们吃个饭等。每个部门都有一些变通的做法，如果你和他们把关系搞好了，一些可以免除的手续就帮你免了。否则你只好一切都按手续走。举个例子来说，公司的IT部门每年都会采购大量的电脑，你如果和他们搞好关系，如果有好电脑到了，他们就会主动通知你，否则你就只能领别人用过的不太好的电脑。其他还有很多类似的细枝末节的地方。如果你觉得自己受到不公正待遇，尽管你可以去投诉，但投诉的结果即使你这一次获胜了，也只会把双方的关系越搞越僵，在这样的争斗中你不可能获益。与其如此，还不如从一开始就把关系搞好。

4．注意留存证据

部门之间办事不同于私人之间，一旦发生争执，就是部门之间的冲突，留好证据可以有力地保证己方的利益不被别人侵犯。特别是一些财物往来的凭证，更应该放在妥善的地方长期保存。

第17章 更上层楼

上学的时候，老师经常说："学如逆水行舟，不进则退。"事实上，在事业当中也是如此，一方面，在工作中积累了相当时间经验之后，你本身会有希望向上晋升的机会；另一方面，如果你长期从事相同的工作，也会面临被淘汰的局面。所以，寻求晋升以及转换工作不论是从个人角度还是从公司角度都是必要且重要的事情。本章我们来探讨一下如何升迁以及转换职业的问题。

17.1 升职加薪

看到别的同事春风满面的从主管办公室走出来，又是一个被加薪的职员，多少次幻想着自己也能够如此，但是为什么加薪总是轮不到自己呢？为什么别人加薪如同喝水一样容易，但是自己要被加薪却难如登天呢？这样的问号是不是不断的在你的心中浮现？或许正是你该好好正视这问题的时候了，检讨一下问题是不是出在自己身上。

17.1.1 如何谋求升职

胡安工作很出色，具有创造力并善于与人沟通，办公室里的人都经常征求他对一些十分重要的计划的意见。他在职业发展方面，下一步顺理成章的是晋升到一个权力和责任更大的岗位，这也是胡安所希望的。但是，日复一日、年复一年，他仍在原来的岗位上工作。人们不禁要问，胡安工作这么出色，为什么没有晋升到更重要的岗位上去呢？

工作能力强和工作效率高的人没有得到晋升的一个最普遍的原因是，他们在目前的岗位上干得如此出色，以至于上级希望他们留在原岗位上工作。为了避免出现这种情况，你可以采取如下策略。

1. 毛遂自荐

很多人没有要求晋升是因为他们认为，如果工作出色，晋升是自然而然的事情。不幸的是，如果你从来不提出要求，你的上级就可能一直不考虑把你提升到一个新的岗位上去。

中国有句俗话："爱哭的孩子有奶吃。"如果你有上进的要求，但你从来不表现出来，领导

又不是你肚里的蛔虫，怎么会猜到你想什么呢？如果你不说，领导就会认为你对目前的职位和待遇很满意，甚至会觉得多付了你钱，老板吃亏了。你只有表现出你的不满意来，领导才会意识到吃亏的一方是你而不是他，由此才有可能为你升职加薪，当然前提是你的工作做的确实不错，值得他给你升职。

2. 积极表现

如果你在现在的岗位上发挥不出才能，要拿出具体的事例来向上级证明，在一个更重要的岗位上，你可以为公司作出更大的贡献。也可以向上级表明，如果把你放到一个新的岗位上，上级的工作会更加得心应手。

另外，如果你有加薪的内涵，却没有加薪的外表，永远看起来像个没睡醒的人，或衣着看起来很不像可以升职的人，加薪升官就不会想到你了。这个可怕的事实是，“外表给人的印象”的重要性，远远超过你的想象。所以必须重视自己的外表，所谓“佛要金装，人要衣装”，在职场中，若没有一个称头的外表是不行的，你要让人看起来很撑得住场面，所以奉劝你好好整顿一下自己的外表，否则即使有再多的内涵还是无法引起别人的注意。

3. 能者多劳

除了具备基本的能力，如计算机能力、英语会话能力之外，更要充实与自己职务相关的专业能力，当然为了提升自己的价值，若还能拥有其他的技能或第二专长，就更容易受到上级的赏识了。具备相当的工作能力和与人沟通协调的能力，不排斥任何新观念，善于学习新技术。你要具备冷静过人的能力，即使面对巨大的问题或挑战，都要有“泰山崩于前仍面不改色”的能力，在混乱的状况中，能够迅速理清头绪，找出最有效的解决方法，如此必能赢得老板的赏识。

要使自己成为可以被托付责任的人，了解老板的意图，对公司忠诚，能尽心尽力处理公司所交付的任务。要勇于承担责任，坚守职业道德，做到所谓“不达工作目标，决不轻言放弃”的地步，一旦博得老板的信任，加薪就靠近你一步了。

同时，证明你能够胜任新工作的一个可靠方式是亲自去做那种工作。你应在短期内同时干两种工作以向你的上级表明：你已经意识到必须完成现在的工作；你能够履行更重要岗位上的职责；你有能力同时从事两种工作。

小王常加班，可是他也常迟到，加班到半夜3点，没人看得到他在做什么，反而，大家都需要他时，他永远不在，看起来很没责任感，其实小王累得要死，却成为大家最常抱怨的对象。如果你是这样的情况，那你应该把时间管理切实搞好。绝对不迟到，也不轻易的请假，最好连年假或特休都还到公司加班。平常不早退，做事有效率，别人做一件事的时间，你最好可以做上二三件事，而且要做的又快又好，让主管无法挑剔。

4. 培养新人

当你晋升的时候，你原有的岗位应得到填补。因此，经常出现的情况是上级对提拔一名职员感到很为难。应协助领导寻找和培训取代你的人，你应向上级保证，你将尽可能使新人在这一岗位上干得像你一样出色。

Offer

在工作中，有些人总害怕把自己的技能传授给别人，害怕别人掌握了这些技能后顶掉自己。其实越是这样的人越不可能得到升迁，因为你的技能别人没有，这个岗位离了你就会运转不畅，自然老板就会永远让你停留在这个岗位上，反倒会提拔那个什么也不会干的人当你的领导，这样的情况屡见不鲜了吧？很多人抱怨说为什么总是“外行领导内行”呢，想解决这个问题其实很简单，把你手头的事情交一部分给外行，并且指导他如何做好，让他成为内行，成为可以替代你的角色，而你自己则把目光放远，更多地关注整体利益而非局部细节，很快老板就会意识到你的重要性。

5．切忌威胁

不论采用何种策略，除非喜欢置之死地而后生，或是已经找到其他退路，否则千万不要用威胁的方式来要求加薪或升职。得不到晋升就发出最后通牒威胁领导，这不是一个好主意。即使你靠这一策略得到了新职位，也不过是赢了战役而输了战争。你与上级和公司的关系将陷入僵局。

处理事情之前必须多为老板或主管想一想，从老板的角度看问题，以老板的感受指导处理方式。多进行换位思考，如果你是老板，你愿意你的下属威胁你吗？推己及人，己所不欲，勿施于人，连你自己都不希望别人对你做的事，当然不应该对你老板这样做。同时，你要显现绝对的忠诚与可靠，不要处处与老板或主管针锋相对，尽可能的照着他们的指示去做事。

6．提防安抚

在你提出晋升要求之后，你应对任何一种暂时安慰你的做法表示怀疑。你应提防假晋升。假晋升的牺牲品被赋予一个新的岗位，而不负有任何新的责任。你也应提防公司专门为你一个人设立一个新的岗位。尽管这个岗位可能合理，但是当实际上没有晋升你的计划的时候，这也可能是上级把你留在公司的手段。

7．创造机会

当上级不准备晋升你的时候，你应变换岗位。如果你的公司环境、你的同事和你的工作使你感到舒心，你就在这个公司待下去并变换一个岗位，这可能有很多好处。因为其他岗位可能为你提供机会，使你获得新的能力并扩展你的经验和联络网。

8．适时离开

如果你感觉在公司要求晋升的努力没有结果，就可以考虑换换地方了。那么，如何知道什么时候离开这家公司好呢？应注意如下情况：3～5 年你没有得到晋升；你晋升的要求被否决；确实证实了你得不到晋升的事实。

如果你觉得以上的条件你都具备了，而且无怨无悔的为公司做牛做马，但仍然无法加薪，或是你觉得你根本无法达到这些标准，建议你就趁早换份工作吧！

17.1.2 如何要求加薪

通常来讲，加薪是伴随着升职而来的，升了职自然就加了薪。如果不升职，是否有可能加薪呢？答案是肯定的。首先，你需要了解清楚你们公司的加薪规则，一般比较大型的公司都会

有比较固定的加薪规则。如果公司规定每年只有一次加薪，是在年终评审结束之后才加的话，那你恐怕很难拿到除此之外的额外的加薪。另外，如果公司宣布说董事会今年的决定是每人加薪在4%左右，那你基本上也很难再和老板要到更多的加薪。如果是在小公司里，由于公司没有固定的加薪机制，导致多年得不到加薪，或者升职空间已经走到了尽头，甚至升了职却加不了薪，这时候就必须考虑如何与老板谈判，要求加薪。

不必害怕要求老板加薪，这是因为，第 1，虽然没有人是不可取代的，但取代你的成本却可能超过为你加薪。第 2，企业是根据能力来决定加薪与否，因此只要还没有被裁掉，就表示可能仍是公司中的强者。第 3，企业必须维持其竞争力，若所付薪资低于员工的应有价值，就会出现士气及生产力低落的问题。

当然，在目前这段艰难的经济时期里，由于薪水冻结、失业停工，以及公司普遍实行严厉的紧缩政策，请求获得一次加薪或提升并不是一件容易做到的事情。如果你在几年前接受的工作其报酬水平低于你的自身价值，或许你渴望能够提高自己的报酬水平，即便在过去两年中，你对自己的报酬收入感到满意，你或许也希望能够尽量使收入保持在稳定阶段。

无论在任何困境中，人们都希望自己能够得到提升，然而你不能够确定，应该采取何种途径达成自己的愿望。这里提供一些争取获得加薪的策略，以及应该避免的事项，帮助你选择正确的方式，从而把握住下一次的提升机会。

1. 认清你的价值

在你要求上司给你加薪之前，应该就你本人对公司的实际价值有一个清楚的认识。你可能期望工资水平能够达到 8 500 元，然而，如果你是刚刚毕业参加工作，目前正为一家小型软件公司工作，那么这个想法于你而言只是一个不实际的梦想。对于你所期望的生活方式，你不能仅仅是简单或任意的进行想象，而是必须实际的了解，在同一职位上的其他人能够获取的收入水平。你可以核对薪水调查表，将你的全部薪水与别人的进行比较。另外，对你自己进行一次考核前的自我评估的最佳办法就是把你的脚伸到就业市场里去，再没有什么能够比经历几次面谈更能让你现实地认识你的服务价值了。

当然，一旦你开始将自己的薪水与当地的市场水平进行比较时，你一定会感到吃惊。你可能会发现你的收入远远低于同行业的平均水平。在这种情况下，可能该是你在现在的工作上来一次最大、最坏、最大胆的赌博的时候了，你可以说："老板，我在别处得到了一份工资远远高于我现在所得的工作机会，因此，要么把我的工资加得跟他们一样高，要么我只好离开这里。"但不要真的这么做，除非你真的是准备离开，因为如果你错误地估计了自己的价值，你的上司可能就会建议你离开。

假如你的研究结果表明你的薪水高出了同类水平，那又该怎么办？是否该撕掉有关的证明材料并且永远不在你认为无知的上司面前提及此事？绝对不是。应该让这件事对你有利。可以对你的上司说，"我了解到这份工作的年平均工资是 50 000 美元，而你给我的却是 55 000 美元。这表明你很看重我的贡献，我还需要做些什么才能晋级？"

你现在岗位的薪金范围并不能告诉你一切，因为这在很大程度上是取决于你属于哪个专

业级别。要想知道自己有多大的价值，应该诚实地考虑你的业绩如何。如果上一次的考评结果很好；如果你的任务越来越具有挑战性；如果你拥有独特的别人所需要的才能；如果你的上司以及上司的上司都喜欢你，说明你已被看成了一位很有价值的员工。而你的薪水应该反映出这一点。

2．寻求得到提升机会的途径

在职场上，工作的分量增加，不全然就代表被赋予更重大的责任。换言之，只有证明自己以更有效率且更有创造力的方式，承担了分外的工作，也就是做到“程序上的精进”，才能作为要求加薪的筹码。

不要指望你的雇主能够了解你为公司引入了多少新业务，或者你为公司做出了多少贡献。工作不同于你在校学习，并且，唯一了解你的工作业绩的人，还是你自己。

因此，你需要坐下来考虑，应该从哪些方面入手才能切实地有助于平衡公司的资产负债表。如果你想不出任何能够帮助公司赚钱的主意，那么说明你自身存在问题。如果你认为作为一名IT经理，并不意味着需要为公司引入新业务，或者不认为这是自己的职责所在，这种观念是错误的。当你为一家公司效力时，你应该总是设法使公司得到不断发展，一旦你充分发挥自己的能力，将自己为公司创造的价值或节约的成本展现在上司面前，将使你的提升或加薪的希望更容易得到实现。

你可以列两张清单，一张是你在去年一年里或者过去几年里为公司的业务达成了什么样的目标，创造了多少产值，节省了多少成本，完成了多少重要项目等这样的成绩。另一张清单是你在本职工作之外又额外承担了多少其他工作，工作职责的增加是最好的加薪理由。

假如说你希望自己的工资能够提高到 8 500 元。你至少应该在提出加薪要求的一周之前，每天至少 20 次对自己说：“我的工资是 8 500 元，并且我的收获是与付出相等的。”这样做或许有一点矫揉造作，但并不会让你有所损失。

你所付出的努力将逐渐被人们所了解和认可，如果你保持积极、愉快的工作态度，并且坚定必胜的信念，人们将乐于接近你。减少对工作的埋怨，并且树立成功的信念。当然这不是一朝一夕能够做到的事情。

同时，你在一年中应该经常向你的上司提到你的进步和成就。你应该经常拿出定期的工作现状报告，既可以是每周的，也可以是每月的，要把它们直接发给上司的邮箱。这种报告至少应该有 3 栏。第 1 栏是我正在做的事情，第 2 栏是现状，第 3 栏是完成日期。

这种报告要保证短小而有吸引力，只列出那些你正在做的重要的事情。而那些微不足道的项目则不要列上去。

通过提醒你的上司你正在干些什么事情，至少可以达到 3 个目的。第 1，你将会让你的上司知道你是多么努力工作的一个人；第 2，你通过帮助上司注意到所发生的事情可以使他的工作更加轻松；第 3，通过保留你所有报告的副本，你就会在年底有足够的证据帮助你为晋级进行的申辩。

从某种程度上来说，即使你有计划有准备，找老板要求加薪都始终会是一件令人感到害怕

的事情，那么无计划无准备地去找老板要加薪就更加是一种妄想了，而且很可能是一场得不偿失的妄想。一旦失败之后，你老板肯定几乎再也不愿跟你谈什么加薪的事情，除非你在工作中发生了质的变化。反过来说，如果你学会了有计划有准备地去要求加薪，那么这就会容易很多。一次成功的加薪谈判还会带来两点好处：你通过谈判增强了自信心，同时，你在选择的职业方向上为达到最佳收益又迈进了坚实的一步。

3．注重策略，要求应该具体

许多员工在要求加薪时会感到难为情，因此也就结结巴巴地取消了他们的要求。如果你觉得自己在谈判方面不是高手的话，那么学习一些谈判技巧绝对是有必要的。可以通过读一些谈判方面的书和资料来增加自己的理论知识，或者咨询一下身边的朋友有没有类似经验也可以让你获益无穷。另外最好的方法还是反复不断的练习，与每一个家人或朋友一起一次又一次地练习，以便你在走进上司的办公室时就能确切地知道你打算说些什么，而且你能够有力地说出来。

在开始谈判之前，一定要先和老板打好招呼，约好时间、地点和谈话的内容，明确告诉他你想和他谈一下你的薪水问题。如果你事先不打招呼，猛然和老板谈起薪水，肯定什么结果也不会有。因为你老板也需要时间和人事部门了解情况，甚至亲自跑人才市场了解行情，不管是哪种情况，谈判肯定是对等的，不要指望突然袭击能让对方就范。

谈判开始后，作为开场白，你应该说明你为什么觉得自己的薪水应该比现在多些。一次成功的加薪谈判总是基于你的价值和工作成绩，绝不可能基于你为什么需要更多的钱上，比如说你上有老下有小，有房贷有车贷等，这些都不是理由。尽管说你老板有可能很关心员工，但对老板来说提供资金来满足你的个人生活需要，这绝不是他的义务。

假如你有任何可以被证实的比较数字，可以礼貌地提到。你可以说几句类似这样的话："我已注意到像我这个职位的平均工资为 8 000 元。考虑到我在过去的 6 个月里完成了这些工作，我希望你能重新评估我目前 6 000 元的月薪。"在谈到你的工作为上司以及公司带来利益的几个方面时应该具体，然后提出一个具体的数额。如果你希望得到 8 000 元，经过反复思考后，你可以把这个数额定在 8 500 元。所有那些优秀的谈判者是怎么做的你就怎么做，开价应该高于你觉得自己所值的数额。

如果老板表示根据你的表现目前不能加薪，你可以问他你还需要做什么才能在下一次有加薪机会时符合条件。同时要注意"有价值的员工"和"真正创造了价值的员工"之间的区别，加薪只可能发生在后者身上。

4．避免造成以下几方面的错误

千万别拿别的公司给你的开价作为谈判的筹码，否则你就准备好失败吧。如果谈判过程中老板了解到你正在找下家，那基本上你在这家公司的事业也就走到头了，即使你并没有离职，即使你拿到了要求的薪水，但随后的一切职业发展，培训计划，以及其他种种机会可能都不会再降临到你身上。因为没有哪个老板喜欢被人胁迫的感觉，而且他肯定会永远记住你。这种情况一旦发生，就会是恶性循环。

同样，威胁说如果得不到加薪就辞职也是非常糟糕的表现。如果你真想辞职，也应该是安

静地悄悄地进行，没有必要大张旗鼓。同时还要注意你身边的同事，不要流露你想辞职的想法，即使他也流露出对老板的不满，这也不能成为你和他交心的理由。切记同事之间无朋友，特别是在薪水这种敏感的事情上，更不能随便和别人谈。所以加薪完全是私密的行为，不要拉着别人和你一起去求老板加薪。

另外，也不要因为你在公司的任职已经超过了一年，就认定自己应该获得提升或加薪。获得提升或加薪取决于你的个人能力和工作表现，而不是取决于工作时间的长短。在公司任职的时间仅仅只有一年，即便在此期间你废寝忘食地多次加班，也不能因此作为提升加薪的要求。

因此，在与上司进行交谈时，你不应该表现出情绪化或小气量。没错，你工作得很卖力，但是所获得的报酬过低。如果你的上司并非这家公司的所有者，当他支付你过低的薪水时，并不意味着他本身存在过错，或许他的处境也和你一样，也被支付过低的薪水。另一种情形下，你的上司拥有这家公司，那么你必须控制自己的情绪更努力的工作，因为你所挣得的每一块钱都是由你的老板直接支付，因此你需要认真工作，使他乐意为你支付令人满意的薪水。

需要注意的是，薪水下降是个普遍存在的现象，抱怨它并不能起到任何帮助作用。不要使你的雇主猜测你有什么想法。当上司询问“你现在期望得到些什么”时，表明这个询问非同寻常，此时你不能因为腼腆而不予回答，也不能故作聪明的回答：“嗯，我希望得到一次提升和加薪，才能反映出我给公司带来的效益和价值。”当你明确的提出：“基于我现在从事的工作，以及在过去 18 个月中取得的附加收入，我相信自己能够挣到 7 500～8 000 元。”事实上，你已经对雇主明确地提出了你的希望。在你明确地提出要求时，需要确定的是，你的请求能够得到可靠的数据支持。

5．结论性意见

如果你别无选择地接受了一份工作，然而薪水与自己所期望的报酬标准相差甚远。此时，你需要使自己的上司了解，你乐意从事这份薪水过低的工作。这是一个重要的心态转变过程。

当你的上司不认为应该支付你更高的薪水时，你不应该使自己陷入不利的境况。因此，如果你的加薪请求被拒绝，这可能只是时机不恰当的问题。

当公司不愿加薪时，你最好的做法是继续留在公司。如果你对公司的发展前景充满希望，并且能够得到老板的器重，那么你应该下定决心经守住这段困难时期。然而，当你们在讨论加薪问题时，你需要得到一些明确的书面形式的承诺。

对高科技领域来说，收入水平正在逐步得到回升。因此，如果你清楚自己的价值，你就能够获得自己应该享有的权益。这仅仅是一个如何采取正确方法和途径的问题。

17.2 换 部 门

从某种意义上来说，在公司内部换部门比辞职还要复杂。如果你想辞职离开公司的话，老板基本上是拦不住你的。但如果你只是想换个部门，那么你不但需要新老板批准，并且还必须

要现任老板批准，如果他不同意，你永远不可能换部门成功。

在某些大公司里，公司文化就鼓励你换工作，那当然是一件好事。但是如果公司文化里没有这一条，你能不能换部门成功，在很大程度上取决于你和老板的关系。如果你和老板之间的关系非常糟糕，基本上你也就不用考虑换部门这件事了，直接看下一章准备辞职吧，因为可以预料的是，即使你在公司内部找到了想要接收你的下家，你的现任老板也会百般阻挠的。所以如果真想换部门，就一定不能得罪现任领导。

17.2.1 在现岗上做出成绩

一种状况是你对现在的工作产生倦怠或低潮。首先，你应该在自己的现有岗位上有所表现，再去争取换部门。毕竟，如果你在现有的岗位上已经表现不好，公司哪敢把你换到新部门去？职场很现实，你得在现在负责的工作上做出成绩，才能跟公司谈。

如果你的工作表现还不够好，而你对现职真的没兴趣或是觉得无法发挥自己的特长，那你起码也得先把分内的工作完成，然后再向老板表示想尝试一下不同的工种或是比较有挑战性的工作。你可以先请老板增加一些新的尝试机会给你，而不是直接要求老板让你换工作。接着，你应该好好把握这些新增加的机会，做出一点成绩，然后再积累调换部门的信用。

17.2.2 学会谦虚与等待

另一种情况是你对现在负责的工作已经轻车熟路，或是对目前负责的业务范围已经非常熟悉，于是想接受新挑战。一般公司都会有良性的工作轮调。轮调一方面是为了给员工新的成长机会；另一方面，也避免员工负责某个工作太久，容易产生盲点。

因此，如果你在现在的岗位上表现得不错，你可以主动向主管提出想要有新的学习机会，在不影响业务的前提下，公司通常都会协助轮调。但是这时候，你的态度应该要更谦虚，不要一副好像是拿现在的成绩与公司谈条件。而且，应该要懂得等待，不能一提出换工作的要求，就希望公司马上帮你安排，公司通常至少要有2～3个月的时间处理业务交接和人事的问题。

17.2.3 如何提？把戏演好

如果你必须调到另一个部门或是另一个团队才能满足你职业发展的需求，此时，你要更小心翼翼地处理这种牵涉到调换团队层次的问题。

1．了解新岗位

决定调换团队的行动后，你可以先通过公司内部征才的公告或网页，了解是否有满意的工作空缺，然后去打听该部门的工作内容和运作情形。如果你对新职位很了解，会让人力资源部门和该部门主管知道你是经过深思熟虑，而更愿意帮你调换部门，千万不要不清不楚就提出调换部门的要求，这会让人质疑你的判断力与做事态度。

2．和人事部门沟通

接着，你得向人力资源部门表达调换工作的意愿，人力资源部门通常会协助你询问想新调

往部门的老板，是否有职位空缺、是否愿意让你加入。当然，人力资源部门也会基于保护你的立场，不会张扬你调换部门的动作。

3．向老板提申请

如果你希望调入的新部门或新团队有意愿让你加入，你就必须正式地向你的现任老板提出申请。当然，你要摸清楚你现在部门主管的个性，如果他够开明，你可以主动跟他提，如果不是，就由人力资源部门帮你协调，这是人力资源部门的工作。

有一种情形常发生，你可能已经私底下跟你想新调往的部门或团队主管谈好，但是无论如何，最后你都必须遵守公司的人事管理制度，你得先向原来的老板提轮调的申请。虽然你已经确定可以调往其他部门，基于尊重老板，即使是演戏，也千万不要让你原来的老板认为他是最后一个知道的人。

17.3 辞　职

好吧，最后这一节里我们来谈一下辞职。辞职是一段职业生涯的结束，也是另一段新生涯的开始，有很多种理由使你觉得必须要辞职，比如你遇到了一个糟糕的老板，工作环境很差，薪水很低等，但同时你也要充分意识到辞职之后会面临的一些新挑战。正确采取以下步骤有助于你顺利解决辞职之前及之后遇到的问题。

17.3.1 决定是否要辞职

正如你在做所有其他事情之前一样，首先要想清楚，我为什么要辞职？是不是到了非辞不可的程度？任何事情都有其两面性，如果你辞了职，你的生活肯定会发生改变，在某些方面会变好，在某些方面会变坏，这些你都要事先考虑清楚。如果是由于老板糟糕，同事恶劣而辞职，也许你会摆脱他们，但在此同时你会面临一些其他新的挑战，所以在提出辞职之前，先要确保这是一个正确的选择。

有很多足够你提出辞职的好理由，看一下你是否属于其中之一。

1．工作压力太大

有些人因为性格内向，特别不愿意在众人面前讲话，每当遇到那样的场合，都觉得是在受刑；还有的人对所从事的工作感到力不从心，为无形的压力所苦。俗话说："身体是革命的本钱。"工作压力太大会使人感到头痛、背痛甚至失眠。如果不论你自己如何调整，都无法解决工作压力的问题，你必须把自己的身体健康放在第一位。

2．老板给你穿小鞋

如果不知是出于什么原因，老板总是拿走你的工作职责，不让你参与部门事务，有重要的会议也不让你参加，你感觉自己被排斥，那么在你作出辞职决定之前，不妨约你的老板谈谈，向他解释解释你目前的感受，问问他，你如何做才能更好，弄明白你还能在这个行当中走多远。

但也有可能老板就是想借这样的方式来暗示你应该辞职了，如果事情没有任何改善余地的话，也许你应该考虑接受他的暗示。

3．你对工作已经失去兴趣

刚接手这个工作的时候，你还是一个新人，充满了工作激情。但随着时间的推移，日复一日重复同样的工作，虽然你已经积累了一大堆工作经验，但你感觉这些经验对你未来的职业发展其实毫无用处，并且你以往的经验也用不到现在的职位上，也许是时候开始寻找一份能充分发挥你的特长的工作了。面对一个无望的职业，你可能不再关心能从工作中学到什么。而在一个令人倾心的工作中，人们寻找不断的发展。你的职业有时候如同你结交的异性朋友一样，你总想知道，有一天，你能否得到一声意味深长的承诺，否则，你就该尽早抽身退出。

在这里，我们有必要介绍一下不值得定律“不值得做的事情，就不值得做好”，这个定律似乎再简单不过了，但它的重要性却时时被人们遗忘。

哪些事值得做呢？一般而言，这取决于3个因素。

（1）价值观。关于价值观我们已经谈了很多，只有符合我们价值观的事，我们才会满怀热情去做。

（2）个性和气质。一个人如果做一份与他的个性气质完全背离的工作，他是很难做好的，如一个好交往的人成了档案员，或一个害羞者不得不每天和不同的人打交道。

（3）现实的处境。同样一份工作，在不同的处境下去做，给我们的感受也是不同的。例如，在一家大公司，如果你最初做的是打杂跑腿的工作，你很可能认为是不值得的，可是，一旦你被提升为领班或部门经理，你就不会这样认为了。

总的来说，值得做的工作是，符合我们的价值观，适合我们的个性与气质，并能让我们看到期望。如果你的工作不具备这3个因素，你就要考虑换一个更合适的工作，并努力做好它。

4．你接到了别的公司的更优厚的待遇

你已经在目前这个工资水平上徘徊很久了，而且未来也看不到有提升的希望。正在这个时候，你忽然发现有一个非常合适你的机会，对方的开价使你无法拒绝，并且你的特长可以在新岗位上得到充分的发挥，这是你最需要考虑辞职的时候。

17.3.2 如果不能辞职

也许你觉得你有很充分的理由想辞职，但出于现实考虑你目前暂时还不能辞职，比如你还没有找好下家，而一旦失去了收入你的生活会陷入窘境。如果是这种情况，那现在就不是辞职的好时机，直到你找好了下家为止。如果你暂时还不能辞职，你必须想办法改善你目前的处境，直到你能辞职为止。

而且有些问题其实是可以不必通过辞职这种激烈的手段也能解决的。如果是下面这几种情况，也许你可以考虑通过辞职以外的其他方法来解决。

1．你的工作和家庭生活有冲突，或者上班距离太远

也许你可以和老板协商，通过一些调配手段来解决这个问题，比如活动工作制，工作共享，

远程办公等。

2. 你和某个同事相处不愉快，或者你和所有同事相处不愉快，或者你和老板处不来

如果这种情况发生，就像婚姻关系糟糕一样让人难受。毕竟每周 5 天你要在工作单位花至少 8～9 个小时的时间工作。在你最终决定辞职之前，最好还是先想想办法看有没有可能和老板及同事们处好关系，如果能处好，那样就不必辞职了。

3. 你在年终评审时得到了差评

如果你得了差评，是不是意味着你必须辞职？也不必然。如果评审是公平公正的，你应该想办法改善你的表现。如果你觉得评审不公平，你可以先和评审你的人心平气和地讨论一下。然后再做决定。

4. 老板实行的一些新政策让你很不爽

对任何人来讲，改变都是困难的。你需要分析清楚，你对老板的新政策不爽是因为你本性抗拒改变，还是你真心相信新政策会对公司造成损害。如果是后者的话，你应该和老板讨论一下你的顾虑，并清晰准确地提出你的改进建议。

17.3.3 开始找新工作

美国管理学家费斯指出：“在拿到第二个以前，千万别扔掉第一个。”尤其是手中的东西对你来说很重要时更应该如此。辞职也是一样，你必须先确保你已经能拿到新的职位了才可以提，否则最好沉默。

在开始寻找新工作之前，你需要做很多工作，包括重新修改你的简历、联系猎头、寻找合适的职位及准备面试等。你也可以让你以前的同事，朋友知道你想要找工作的意愿，但如果你准备找好工作再辞职的话，那现在还不是广播你的求职计划的时候。

关于如何找新工作，可以翻回头看本书前 3 章。

17.3.4 提出辞职

如果你要辞职，很显然你必须告诉你的老板。如果有什么特殊情况你还必须在现在的岗位上多待一段时间的话，那你现在就不能提辞职的事情，直到你明确地知道哪天是你在这里的最后一天。而且在告诉你老板之前，不要对任何其他员工提你要辞职的事情。你需要写一封正式的辞职信，并且和老板约时间单独面谈。如果老板在国外的话，也要约好在电话里交谈。在正式离职之前要留出充足的交接时间，通常两周的时间是合适的，但如果你从事的工作专业化程度比较高，比较难以交接的话，也许需要更多时间，最长可能需要 1 个月。如果你目前正参与一个大型项目，老板也许会希望你对新接手的人进行培训。

另外，要注意老板可能会对你进行挽留，甚至提出一些更为优厚的条件。在这种情况下，要毫不犹豫，当机立断，不要像布利丹之驴一样。14 世纪时法国经院哲学家布利丹，在一次议论自由问题时讲了这样一个寓言故事：“一头饥饿至极的毛驴站在两捆完全相同的草料中间，可是它却始终犹豫不决，不知道应该先吃哪一捆才好，结果活活被饿死了。”凡事都有好有坏，该

采取措施的时候就必须积极行动，按照自己内心的想法行动，而不是轻易被别人所左右。

17.3.5 离开工作岗位

不管你是主动辞职还是被公司裁员或者开除，离职总是一件令人难过的事情。

1．不要恶语伤人

终于到了离职的时候，或许你心情会很激动，特别是如果你是在一种有怨言的情况下离职的时候更是如此。也许你会想告诉你老板或者某个同事其实你内心深处是如何看待他们的。千万别这样做，就算他们完全配得上你对他们这样的评价也不要这样做。山不转水转，你永远不知道也许什么时候你又会遇上他们或者有需要他们帮助的时候，做人肚量要大一些，过去的事情就过去算了，人应该更多的是着眼未来。

2．不要破坏公司财物或者盗窃资料

你也许觉得你老板对你很糟糕，你也许真的非常生气。但是，破坏公物和盗窃不光是道德问题，更是违法行为。一旦被人发现，不只是你的职业生涯受影响，你还有可能进监狱，这一点都不夸张。

3．别忘了要一份推荐信

如果你是在一种并不友好的气氛下离职的话，这也许听上去有点奇怪。但是，你未来的简历里肯定会包含这一段经历，所以你应该努力争取一个良好的或者哪怕是中立的评价。如果你是由于犯了重大过失被开除，这基本上是没希望的事情。但如果你的离职是由于不那么严重的原因的话，你还是可以找你的老板争取要一个评价，别去考虑什么“也许他根本不会给我评价”这样的事情，反正你也要离开了，你怎么知道他一定不会给你呢？就算他不给的话，那就不给好了，对你也没什么损失。

4．别对接手你的人说你老板或者任何同事的坏话

首先，这种行为使你自己看上去像那只吃不着葡萄说葡萄酸的狐狸；其次，接手你的人，人家自己会分辨事情的好坏，不需要你来指手画脚；最后，对你来讲很糟糕的经历，也许在他处理起来感觉非常好呢？所以交接的时候只把与业务相关的部分交接了就行了，对任何人不作评价。