

- 在这里，众多知名企业面试官将为你撕开神秘的求职面纱
- 在这里，多位求职达人将现身说法为你揭开求职谜团
- 在这里，各种类型的企业招聘细节都会被展露无疑
- 在这里，我们将为你抽丝剥茧，还原IT行业的真相
- 在这里，我们将为你指点迷津，告诉你职场上的风风雨雨
- 在这里，我们将为你点石成金，成为那盏指引你前进的灯塔

Java 程序员 面试笔试

何昊 薛鹏 叶向阳 编著

宝典



本书覆盖了历年来各大IT名企95%以上的面试笔试题，当你细细品读完本书的知识后，各类企业的offer将任由你挑选。本书将带你走进神奇的求职之旅。



机械工业出版社
CHINA MACHINE PRESS

官方微博: <http://weibo.com/cmpjsj>
官方博客: <http://blog.sina.com.cn/cmpbookjsj>
读者信箱: cmp_itbook@163.com

作者简介



何 昊

何昊, 湖北仙桃人, 软件系统分析师, 西安电子科技大学计算机软件与理论专业硕士。擅长C/C++/C#/Java等主流编程语言, 具有丰富的项目经验。酷爱体育锻炼与历史, 并喜欢尝试各种新鲜事物。现就职于某知名科研设计单位, 从事软件研发工作, 成功指导众多求职者进入各大IT名企工作。编著有《程序员面试笔试宝典》、《程序员求职之道》等书。



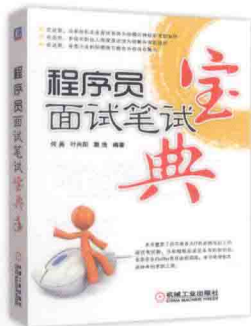
薛 鹏

薛鹏, 陕西榆林人, 软件系统分析师, 西安电子科技大学计算机软件与理论专业硕士。现就职于某知名外企, 从事软件研发工作, 擅长Java/C++等主流编程语言, 具有丰富的项目经验。酷爱篮球运动, 喜欢户外活动。编著有《程序员求职之道》。



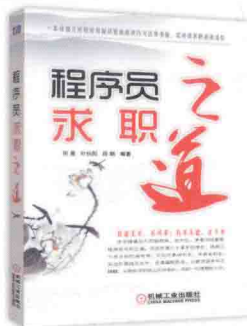
叶向阳

叶向阳, 湖北英山人, 工学硕士, 空军工程大学高级工程师。主要研究方向: 计算机网络, 无线通信。编著有《程序员面试笔试宝典》、《程序员求职之道》等书。



程序员面试笔试宝典

ISBN 978-7-111-39879-0
定价: 59.80 元



程序员求职之道

ISBN 978-7-111-45746-6
定价: 29.80 元

地址: 北京市百万庄大街22号
邮政编码: 100037
电话服务
社服务中心: 010-88361066
销售一部: 010-68326294
销售二部: 010-88379649
读者购书热线: 010-88379203
网络服务
教材网: <http://www.cmpedu.com>
机工官网: <http://www.cmpbook.com>
机工官博: <http://weibo.com/cmp1952>
封面无防伪标均为盗版

上架指导 计算机/程序设计

ISBN 978-7-111-47746-4

策划编辑◎时 静 / 封面设计◎



扫一扫, 或搜索微信号
cmpbook-IT, 关注即可
获得更多机工IT图书服务。

ISBN 978-7-111-47746-4



9 787111 477464

定价: 48.80 元

Java 程序员面试笔试宝典

何昊 薛鹏 叶向阳 编著

机械工业出版社

本书引入了一批来自于名牌高校、就职于明星企业的职场达人的真实求职案例，通过他们的求职经验与教训，将整个求职过程生动形象地展示在读者面前，进而对求职者起到一定的指引作用。同时，为了更具说服力，本书特邀多位 IT 名企的面试官现身说法，以独特的视角对面试过程中求职者存在的各类问题进行了深度剖析。为了能够让读者对即将投身的工作有一些更加清楚的认识，能够更加有针对性地进行求职准备，本书对各种类型的 IT 企业的招聘环节进行了庖丁解牛式的分析。

技术性知识的考查是程序员求职中最重要的内容，鉴于此，本书除了对传统的计算机相关知识（Java 语言基础知识、Web 基础知识、数据结构与算法、数据库、设计模式等）以及面试笔试真题进行分析与解答外，还根据当前计算机技术的发展潮流，对面试笔试中常见的海量数据处理问题进行了详细的分析。

本书是一本适合计算机相关专业毕业生阅读的求职指导用书，同时也适合期望在计算机软硬件行业大显身手的计算机爱好者阅读。

图书在版编目（CIP）数据

Java 程序员面试笔试宝典/何昊，薛鹏，叶向阳编著. —北京：机械工业出版社，2014.6

ISBN 978-7-111-47746-4

I. ①J… II. ①何…②薛…③叶… III. ①JAVA 语言—程序设计
IV. ①TP312

中国版本图书馆 CIP 数据核字（2014）第 192692 号

机械工业出版社（北京市百万庄大街 22 号 邮政编码 100037）

策划编辑：时 静 责任编辑：时 静 吴晋瑜

责任校对：张艳霞 责任印制：乔 宇

北京机工印刷厂印刷（三河市南杨庄国丰装订厂装订）

2014 年 9 月第 1 版·第 1 次印刷

184mm × 260mm · 22 印张 · 544 千字

0 001—3 500 册

标准书号：ISBN 978-7-111-47746-4

定价：48.80 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务

网络服务

社服务中心：(010)88361066 教材网：<http://www.cmpedu.com>

销售一部：(010)68326294 机工官网：<http://www.cmpbook.com>

销售二部：(010)88379649 机工官博：<http://weibo.com/cmp1952>

读者购书热线：(010)88379203 封面无防伪标均为盗版

前 言

很多朋友问我为什么编写《Java 程序员面试笔试宝典》，它与我先前编写的《程序员面试笔试宝典》究竟有何不同？其实，我之所以创作本书，就是希望通过本书能够为主攻 Java 方向的求职者提供一些帮助。《程序员面试笔试宝典》一书面市后，在学生群中反响不错。很多读者反映，通过阅读本书，他（她）们找到了自己比较满意的工作，但不足之处在于该书偏重于 C/C++ 语言，而很多招聘岗位考查的却是 Java 语言的相关内容。此外，根据前程无忧、智联招聘和中华英才三大专业招聘网站的就业数据调查分析可以看出，在目前的就业市场上，Java 工程师的市场需求量非常大，丝毫不亚于 C/C++ 工程师，所以，他（她）们希望能够看到一本专门针对 Java 语言面试笔试的书。

于是，我萌发了编写一本有关 Java 程序员面试笔试之书的想法。近些年来，无论是传统的互联网应用，还是当前发展迅猛的云计算、海量数据处理以及移动互联网，都离不开 Java 语言。Java 语言始终在信息技术浪潮中扮演着极其重要的角色。从历次编程语言排行榜不难看出，Java 语言的应用与 C 语言不分伯仲，它们都可以称得上是计算机历史上重要的编程语言。鉴于此，很多 IT 企业（例如门户网站、即时通信、电子商务、搜索引擎等）、手机应用开发企业等都使用 Java 语言作为开发语言，因此自然希望求职者熟悉 Java 语言的基本原理，并能够熟练使用 Java 语言从事实际的项目研发，所以，也会将 Java 作为面试笔试环节的主要考查内容。

作为《程序员面试笔试宝典》的姊妹篇，本书一方面延续并继承了其中诸如面试官箴言、面试心得交流、企业面试笔试攻略、海量数据处理等深受读者好评的经典内容；另一方面，编者结合当前的实际情况，对面试笔试中常涉及的 Java 语言基础知识进行了庖丁解牛式的深度剖析。此外，本书还针对各大 IT 名企的笔试考题，总结出了适用于应试的方法与思路，可以帮助读者轻松应对 Java 面试笔试中的各类问题。

本书由何昊、薛鹏、叶向阳共同编著。在本书的编写过程中，董西成、邵帅、王震、伍文明、李超、曹润涛、郭晶晶、阎贝、林方超、廖兰新、李志强、厉孙德、褚艳利、丁志浩、卢山、梁敏、回永利等为本书的编写提供了非常宝贵的材料。软件工程中心武方方主任、张向虎主任、鲁昊鹏研究员、张剑研究员、张玉博高工、屈华敏研究员、苏媚高工、杨黎高工、李靖高工、张敏高工、赵亮高工、辛航高工、徐建军高工、郑小宁高工等领导对我的工作给予了无微不至的帮助。何四为律师为本书提供了一些有关版权的法律援助。机械工业出版社计算机分社的时静编辑给了我大力的支持与通力的配合。除此之外，我的父母、亲人、同事、朋友、同学，无论我遇到了多大的挫折与困难，他们都一如既往地支持与帮助我，使我能够开开心心地度过每一天。在此对以上所有人一并致以最衷心的感谢。

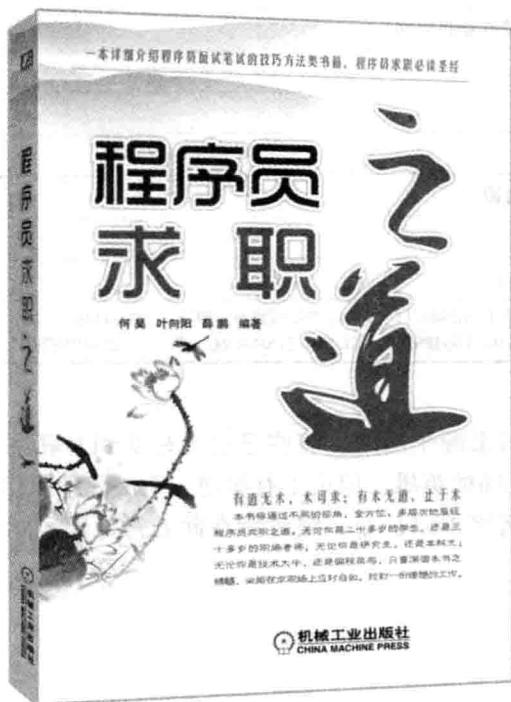
创作的过程是一个自我斗争、自我救赎的过程。无数个节假日，无数个深夜，当其他人沉浸在欢歌笑语中时，我需要安静地坐在计算机前，对所编写的内容仔细推敲，力求简单明了；将实例代码一一验证，力求准确无误。尽管孤独，但我觉得只要自己所做的事情、所付出的辛苦能为读者们增添几分求职成功的把握，我就心满意足了。

由于本人水平有限，书中不足之处在所难免，还望读者见谅。读者如果发现问题或是有此方面的困惑，可以通过邮箱 xdhehao@foxmail.com 或者 xdxuepeng@foxmail.com 与我们联系。

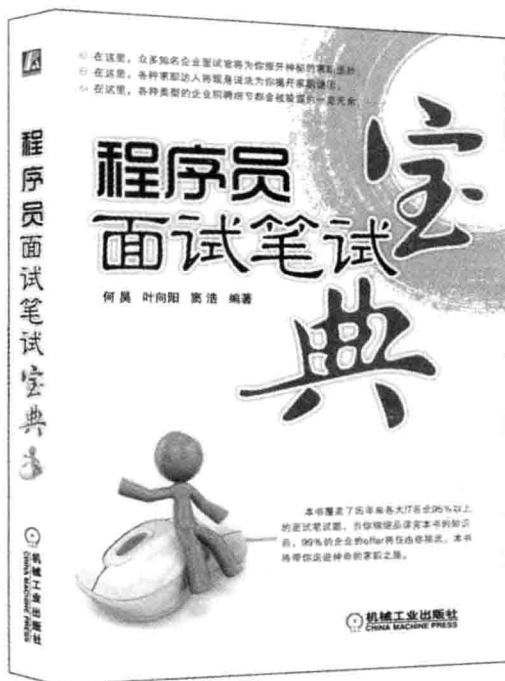
何 昊

于古都西安

优秀图书推荐



作者：何昊等编著
出版社：机械工业出版社
出版时间：2014-3-1
ISBN：9787111457466
定价：29.80 元



作者：何昊等编著
出版社：机械工业出版社
出版时间：2012-10-1
ISBN：9787111398790
定价：59.80 元

目 录

前言	main (String[] args) 这个方法	48
上篇 面试笔试经验技巧篇	4.1.4 如何实现在 main() 方法执行前输出 “Hello World”	49
第1章 面试官箴言	4.1.5 Java 程序初始化的顺序是怎样的	50
1.1 有道无术，术可求；有术无道， 止于术	4.1.6 Java 中的作用域有哪些	51
1.2 求精不求全	4.1.7 一个 Java 文件中是否可以定义 多个类	52
1.3 脚踏实地，培养多种技能	4.1.8 什么是构造函数	53
1.4 保持空杯心态	4.1.9 为什么 Java 中有些接口没有 任何方法	54
1.5 职场是能者的舞台	4.1.10 Java 中的 clone 方法有什么作用	56
1.6 学会“纸上谈兵”	4.1.11 什么是反射机制	60
1.7 小结	4.1.12 package 有什么作用	61
第2章 面试心得交流	4.1.13 如何实现类似于 C 语言中函数 指针的功能	62
2.1 心态决定一切	4.2 面向对象技术	64
2.2 假话全不说，真话不全说	4.2.1 面向对象与面向过程有什么区别	64
2.3 走自己的路，让别人去说吧	4.2.2 面向对象有哪些特征	65
2.4 夯实基础谋出路	4.2.3 面向对象的开发方式有什么优点	65
2.5 书中自有编程法	4.2.4 什么是继承	66
2.6 笔试成绩好，不会被鄙视	4.2.5 组合和继承有什么区别	66
2.7 不要一厢情愿做公司的 “备胎”	4.2.6 多态的实现机制是什么	67
2.8 小结	4.2.7 重载和覆盖有什么区别	69
第3章 企业面试笔试攻略	4.2.8 抽象类 (abstract class) 与接口 (interface) 有什么异同	70
3.1 互联网企业	4.2.9 内部类有哪些	71
3.2 网络设备提供商	4.2.10 如何获取父类的类名	73
3.3 外企	4.2.11 this 与 super 有什么区别	74
3.4 国企	4.3 关键字	76
3.5 研究所	4.3.1 变量命名有哪些规则	76
3.6 创业型企业	4.3.2 break、continue 以及 return 有 什么区别	76
3.7 如何抉择	4.3.3 final、finally 和 finalize 有什么 区别	77
下篇 面试笔试技术攻克篇	4.3.4 assert 有什么作用	79
第4章 Java 基础知识	4.3.5 static 关键字有哪些作用	80
4.1 基本概念	4.3.6 使用 switch 时有哪些注意事项	83
4.1.1 Java 语言有哪些优点		
4.1.2 Java 与 C/C++ 有什么异同		
4.1.3 为什么需要 public static void		

4.3.7	volatile 有什么作用	85	4.8.1	为什么说 Java 是平台独立性语言	125
4.3.8	instanceof 有什么作用	86	4.8.2	Java 平台与其他语言平台有哪些区别	126
4.3.9	strictfp 有什么作用	86	4.8.3	JVM 加载 class 文件的原理机制是什么	126
4.4	基本类型与运算	87	4.8.4	什么是 GC	127
4.4.1	Java 提供了哪些基本数据类型	87	4.8.5	Java 是否存在内存泄露问题	130
4.4.2	什么是不可变类	89	4.8.6	Java 中的堆和栈有什么区别	131
4.4.3	值传递与引用传递有哪些区别	91	4.9	容器	133
4.4.4	不同数据类型的转换有哪些规则	94	4.9.1	Java Collections 框架是什么	133
4.4.5	强制类型转换的注意事项有哪些	95	4.9.2	什么是迭代器	134
4.4.6	运算符优先级是什么?	95	4.9.3	ArrayList、Vector 和 LinkedList 有什么区别	135
4.4.7	Math 类中 round、ceil 和 floor 方法的功能各是什么	96	4.9.4	HashMap、HashTable、TreeMap 和 WeakHashMap 有哪些区别	136
4.4.8	++i 与 i++ 有什么区别	97	4.9.5	用自定义作为 HashMap 或 HashTable 的 key 需要注意哪些问题	137
4.4.9	如何实现无符号数的右移操作	98	4.9.6	Collection 和 Collections 有什么区别	142
4.4.10	char 型变量中是否可以存储一个中文汉字	100	4.10	多线程	142
4.5	字符串与数组	102	4.10.1	什么是线程? 它与进程有什么区别? 为什么要使用多线程	142
4.5.1	字符串创建与存储的机制是什么	102	4.10.2	同步和异步有什么区别	143
4.5.2	"=="、equals 和 hashCode 有什么区别	103	4.10.3	如何实现 Java 多线程	144
4.5.3	String、StringBuffer、StringBuilder 和 StringTokenizer 有什么区别	105	4.10.4	run()方法与 start()方法有什么区别	146
4.5.4	Java 中数组是不是对象	107	4.10.5	多线程同步的实现方法有哪些	148
4.5.5	数组的初始化方式有哪几种	107	4.10.6	sleep()方法与 wait()方法有什么区别	149
4.5.6	length 属性与 length()方法有什么区别	109	4.10.7	终止线程的方法有哪些	150
4.6	异常处理	110	4.10.8	synchronized 与 Lock 有什么异同	152
4.6.1	finally 块中的代码什么时候被执行	110	4.10.9	什么是守护线程	156
4.6.2	异常处理的原理是什么	113	4.10.10	join()方法的作用是什么	157
4.6.3	运行时异常和普通异常有什么区别	114	4.11	Java 数据库操作	158
4.7	输入输出流	116	4.11.1	如何通过 JDBC 访问数据库	158
4.7.1	Java IO 流的实现机制是什么	116	4.11.2	JDBC 处理事务采用什么方法	160
4.7.2	管理文件和目录的类是什么	118	4.11.3	Class.forName 的作用是什么	161
4.7.3	Java Socket 是什么	119	4.11.4	Statement、PreparedStatement 和 CallableStatement 有什么区别	161
4.7.4	Java NIO 是什么	120			
4.7.5	什么是 Java 序列化	121			
4.7.6	System.out.println()方法使用需要注意哪些问题	124			
4.8	Java 平台与内存管理	125			

4.11.5	getString()方法与 getObject() 方法有什么区别	163	5.2.10	EJB 规范规定 EJB 中禁止的 操作有哪些	187	
4.11.6	使用 JDBC 时需要注意 哪些问题	164	5.2.11	Web 服务器与 Web 应用服务器 有什么区别	188	
4.11.7	什么是 JDO	164	5.2.12	什么是 Web Service	188	
4.11.8	JDBC 与 Hibernate 有什么区别	164	5.2.13	SOAP 与 REST 有什么区别	189	
第 5 章 Java Web			165	5.2.14	什么是 XML	190
5.1	Servlet 与 JSP	165	5.2.15	数据库连接池的工作机制 是怎样的	193	
5.1.1	页面请求的工作流程是怎样的	165	5.2.16	J2EE 开发有哪些调优的方法	193	
5.1.2	HTTP 中 GET 与 POST 方法 有什么区别	166	5.3	框架	194	
5.1.3	什么是 Servlet	166	5.3.1	什么是 Struts 框架	194	
5.1.4	doPost()方法与 doGet() 方法怎么选择	168	5.3.2	Struts 框架响应客户请求的 工作流程是什么	196	
5.1.5	什么是 Servlet 的生命周期	168	5.3.3	Struts 框架的数据验证可分为 几种类型	197	
5.1.6	JSP 有哪些优点	169	5.3.4	Form Bean 的表单验证流程 是什么	197	
5.1.7	JSP 与 Servlet 有何异同	169	5.3.5	在 Struts 配置文件中, <action > 元素包含哪些属性和子元素	197	
5.1.8	如何使用 JSP 与 Servlet 实现 MVC 模型	170	5.3.6	ActionForm Bean 的作用有哪些	198	
5.1.9	Servlet 中 forward 和 redirect 有什么区别	172	5.3.7	ActionForm 的执行步骤有哪些	198	
5.1.10	JSP 的内置对象有哪些	173	5.3.8	forward 与 global-forward 有什么区别	198	
5.1.11	request 对象主要有哪些方法	174	5.3.9	Struts 如何实现国际化	199	
5.1.12	JSP 有哪些动作	175	5.3.10	Struts 1 与 Struts 2 有哪些区别	199	
5.1.13	JSP 中 include 指令和 include 动作 有什么区别	176	5.3.11	什么是 IoC	201	
5.1.14	会话跟踪技术有哪些	176	5.3.12	什么是 AOP	203	
5.1.15	Web 开发中如何指定 字符串的编码	177	5.3.13	什么是 Spring 框架	205	
5.1.16	什么是 Ajax	178	5.3.14	什么是 Hibernate	206	
5.1.17	cookie 和 session 有什么区别	178	5.3.15	什么是 Hibernate 的二级缓存	211	
5.2	J2EE 与 EJB	179	5.3.16	Hibernate 中 session 的 update() 和 saveOrUpdate()、load() 和 get() 有什么区别	212	
5.2.1	什么是 J2EE	179	5.3.17	Hibernate 有哪些主键 生成策略	213	
5.2.2	J2EE 中常用的术语有哪些	179	5.3.18	如何实现分页机制	214	
5.2.3	EJB 有哪些不同的类别	181	5.3.19	什么是 SSH	214	
5.2.4	EJB 与 JavaBean 有什么异同	182	第 6 章 数据库原理		216	
5.2.5	EJB 有哪些生命周期	182	6.1	SQL 语言的功能有哪些	216	
5.2.6	EJB 的角色有哪几种	184	6.2	内连接与外连接有什么区别	218	
5.2.7	EJB 的开发流程是怎样的	185	6.3	什么是事务	219	
5.2.8	EJB 3.0 与 EJB 2.0 有哪些 不同之处	187				
5.2.9	EJB 容器有哪些作用	187				

6.4 什么是存储过程? 它与函数 有什么区别与联系	220	8.3.7 如何进行堆排序	255
6.5 各种范式有什么区别	221	8.3.8 各种排序算法有什么优劣	257
6.6 什么是触发器	222	8.4 位运算	257
6.7 什么是游标	223	8.4.1 如何用移位操作实现乘法运算	257
6.8 如果数据库日志满了, 会出现 什么情况	224	8.4.2 如何判断一个数是否 为 2 的 n 次方	258
6.9 union 和 union all 有什么区别	224	8.4.3 如何求二进制数中 1 的个数	259
6.10 什么是视图	225	8.5 数组	260
第 7 章 设计模式	226	8.5.1 如何寻找数组中的最小值与 最大值	260
7.1 什么是单例模式	226	8.5.2 如何找出数组中第二大的数	261
7.2 什么是工厂模式	227	8.5.3 如何求最大子数组之和	262
7.3 什么是适配器模式	229	8.5.4 如何找出数组中重复元素 最多的数	266
7.4 什么是观察者模式	229	8.5.5 如何求数组中两两相加等于 20 的 组合种数	267
第 8 章 数据结构与算法	231	8.5.6 如何把一个数组循环右移 k 位	268
8.1 链表	231	8.5.7 如何找出数组中第 k 个 最小的数	269
8.1.1 如何实现单链表的增删操作	231	8.5.8 如何找出数组中只出现 一次的数字	270
8.1.2 如何从链表中删除重复数据	234	8.5.9 如何找出数组中唯一的 重复元素	272
8.1.3 如何找出单链表中的倒数 第 k 个元素	235	8.5.10 如何用递归方法求一个整数 数组的最大元素	276
8.1.4 如何实现链表的反转	236	8.5.11 如何求数对之差的 最大值	276
8.1.5 如何从尾到头输出单链表	237	8.5.12 如何求绝对值最小的数	280
8.1.6 如何寻找单链表的中间结点	237	8.5.13 如何求数组中两个元素的 最小距离	282
8.1.7 如何检测一个链表是否有环	238	8.5.14 如何求指定数字在数组中第一次 出现的位置	283
8.1.8 如何在不知道头指针的情况下 删除指定结点	239	8.5.15 如何对数组的两个子有序段 进行合并	284
8.1.9 如何判断两个链表是否相交	239	8.5.16 如何计算两个有序整型 数组的交集	285
8.2 栈与队列	241	8.5.17 如何判断一个数组中数值 是否连续相邻	287
8.2.1 栈与队列有哪些区别	241	8.5.18 如何求解数组中反序对的 个数	288
8.2.2 如何实现栈	241	8.5.19 如何求解最小三元组距离	289
8.2.3 如何用 $O(1)$ 的时间复杂度 求栈中最小元素	243	8.6 字符串	292
8.2.4 如何实现队列	244	8.6.1 如何实现字符串的 反转	292
8.2.5 如何用两个栈模拟队列操作	246		
8.3 排序	247		
8.3.1 如何进行选择排序	247		
8.3.2 如何进行插入排序	248		
8.3.3 如何进行冒泡排序	249		
8.3.4 如何进行归并排序	250		
8.3.5 如何进行快速排序	252		
8.3.6 如何进行希尔排序	254		

8.6.2	如何判断两个字符串是否由相同的字符组成	293	8.8.1	如何消除嵌套的括号	309
8.6.3	如何删除字符串中重复的字符	295	8.8.2	如何不使用比较运算就可以求出两个数的最大值与最小值	311
8.6.4	如何统计一行字符中有多少个单词	297	第9章 海量数据处理		312
8.6.5	如何按要求打印数组的排列情况	298	9.1	问题分析	312
8.6.6	如何输出字符串的所有组合	300	9.2	基本方法	312
8.7	二叉树	301	9.3	经典实例分析	325
8.7.1	二叉树基本概念	301	9.3.1	top K 问题	325
8.7.2	如何实现二叉排序树	303	9.3.2	重复问题	328
8.7.3	如何层序遍历二叉树	305	9.3.3	排序问题	330
8.7.4	已知先序遍历和中序遍历, 如何求后序遍历	306	附录		331
8.7.5	如何求二叉树中结点的最大距离	308	附录 A	软件企业 Java 笔试真题 1	331
8.8	其他	309	附录 B	软件企业 Java 笔试真题 2	336
			附录 C	软件企业 Java 笔试真题 3	339
			附录 D	求职有用网站及 QQ 群一览表	342



上篇

面试笔试经验技巧篇

第1章 面试官箴言

第2章 面试心得交流

第3章 企业面试笔试攻略

第1章

面试官箴言

什么样的求职者能够获得面试官的青睐？求职者需要准备哪些内容来面对形形色色的面试官？什么样的企业适合自己发展？在新的工作岗位上，如何努力才能从人才济济的企业中脱颖而出？本章中，几位资深软件工程师将现身说法，为您一一解答上述问题。

1.1 有道无术，术可求；有术无道，止于术

丁志浩，男，硕士，某知名芯片公司软件工程师。

以下这些内容是写给即将成为职业人的在校学生的，希望能够对他们的求职与以后的工作有一定的参考作用。

在介绍求职之前，我想先说一些与具体技术无关但却比技术更加重要的东西，主要有以下两个方面的内容：第一点，认清自我；第二点，保持强烈的求知欲。之所以提及这两点，并且认为它们是最重要的东西，是结合我的亲身经历，我认为一个人最重要的是认清自我，只有认清了自我，你才会知道自己想要做什么、适合做什么、能做什么。在某种程度上来说，这比所学的知识、技术更加重要。只有方向正确了，才会有前进的动力；只有有了前进的动力，才会为目标不断努力；只有朝着正确方向不断努力了，才可能会有收获。其次，要有强烈的求知欲，随着年龄的增大、个人阅历的增长，生活、家庭、工作会慢慢消磨掉你的雄心壮志，而能保持强烈的求知欲实属难能可贵。世界上很少有学不会的东西，就看你是否用心去做了，是否愿意花时间、动脑筋、投入精力去做，万事就怕认真，只要你认真做了，通常是可以学会的。

切入正题，作为一名以程序员为职业目标的求职者，关注的领域主要还是以技术为主，IT企业在面试的时候主要关注求职者什么方面的内容呢？以我这些年的工作经历来看，大企业看道，小企业看术。有道无术，术可求；有术无道，止于术。具体来说，大企业更加看重的是求职者的基础知识以及解决问题的能力。一般而言，大企业都会有比较完备的培训机制，它可以在较短时间内把一个什么都不会的员工塑造成一个它想要的人；而小企业则不然，他们更加注重求职者的实用性，求职者当前会什么，能给企业带来什么。这种思维方式的不同其实也是由企业的性质决定的，其本身没有对错之分。当然这也无可厚非，所以个人建议求职者最好夯实计算机基础知识，操作系统、编译原理、算法等这些基础知识就是重中之重了，需要重点掌握。万变不离其宗，当你到达了一定程度，对你而言只是个形式上的差异而已。

对于求职者需要如何准备才能更好地获得面试官的青睐，我觉得，IT企业一般需要的大

多数都是技术型人才，所以具有以下3个优点的人，一般更能受到面试官的亲睐：①基本功扎实的人，基础扎实了，后劲就足，发展前景就更好；②具有强烈的求知欲、对未知领域比较感兴趣、能够接受新事物的人；③在某个领域有比较深入的研究的人。因为如果求职者已经在某个方面有了比较深入的研究，有了良好的基础，对于将来的发展肯定会更好，例如，当前好多企业都在搞云计算，如果求职者对 Hadoop 这种架构有比较深入的理解，当然就比不懂 Hadoop 的求职者成功率更高。

有了录用通知（offer）以后，在挑选 offer 时，求职者往往也很纠结，其实我在这里也不是告诉求职者是该选择互联网还是芯片公司，或是其他类型企业，因为对这个问题，仁者见仁智者见智，每个人考虑的侧重点也不一样，所以在此我不说到底该选什么企业，以免误导大家，但我可以给求职者一个建议：往大的方面讲，首先是选择行业，然后选择企业，最后是选择职业。最好能够综合自己的兴趣爱好，因为兴趣是最好的老师。

入职之后，如何才能适应新的工作岗位，完成从学生到职业人的华丽转变呢？一般而言，刚毕业时，新人都是雄心壮志、意气风发，想在新的工作岗位上大展拳脚、有所作为，这虽然是一件非常好的事情，但是现代企业分工很明确，尤其是对于企业的新员工，刚工作时，很有可能接触的东西都是些缺乏技术含量或是相对边缘化的东西，只是充当企业的一颗“小螺丝钉”而已。所以在此，我建议求职者在刚入职时，最好能够放低姿态，当将军的人，都是从小兵一步步做起的。刚毕业态度最重要，切忌整天怨天尤人，否则会给人一种浮躁的感觉，对将来的发展肯定是不利的。

1.2 求精不求全

褚艳利，女，硕士，某知名电子商务公司软件工程师。

时光荏苒，我已经成为 IT 业一名所谓的“老鸟”了，但我也曾是一名普通的求职者，也曾在求职的路上历经风雨。希望我的一些经历和感悟，能为朋友们提供些许帮助。

对于应届生求职，我觉得每一场面试都是从“闻味儿”开始的。看似是一场简单的聊天，但其实求职者的各方面已经在被面试官考查了，例如在沟通过程中，从求职者的谈吐、穿着、眼神、沟通过程中，或多或少就“闻”出很多层“味道”了（求职者的性格、处事态度、表达能力、沟通能力、团队合作能力）。经常会听到求职者说：“面试官今天一道技术题都没问我。”这多是面试官对求职者综合素质的一种肯定（前提是成绩单不能太难看）。如果是应聘技术类职位，那么求职者的技术水平还是要积累的。

对于技术的积累，我觉得是“求精不求全”，如今的高等院校通常都会开设“C 语言”、“C++”、“Java”、“网络”、“数据库”、“编译原理”、“软件工程”等课程，但由于精力有限，毕竟不是每个人都可以做到门门精、样样通，所以我建议从兴趣出发，深入学习几门课程（当然，其他课程也要学，毕竟是在技术领域，一些概念和基本原理不知晓是不行的），例如我个人比较钟爱数据结构、算法、C 语言、操作系统等专业知识，对这些下足功夫做足功课，也正是这些基础，让我打赢了很多场艰难“战役”。当然，在面试他人的过程中，我也会问到一些可能他们不太擅长的知识，例如设计模式。其实我并不是为难他，只要他能讲出自己的理解，并直言自己这方面知识的欠缺，我也觉得无可厚非，这种坦白比不懂装懂来得更真实、更有力量。所以，作为一名“过来人”，我觉得大部分面试官在面试时，会更加侧重于考查求职者擅长的方面，从这点能看到求职者未来发展和潜力。

作为一名职场新手，在准备过程中，求职者应该根据职位要求略作筹备。虽然说万变不离其宗，但根据职位要求，有针对性地准备一下，效果会更好。例如面试数据库开发的，DB（数据库）知识就需要好好准备一下，这样不至于因什么也答不出来而弄得气氛太尴尬，也可获得后续的面试机会。对于普通的软件开发类职位，我认为求职者应该必备以下知识：数据结构、某类编程语言、操作系统和基本 DB 知识。

我觉得要想获得面试官们的青睐，求职者需要注意以下几个方面的问题：

- 1) 衣着妆扮。对于技术类职位，衣着妆扮虽然不做要求，但也不能过于邈邈。女性求职者画一点淡妆更好。
- 2) 眼神交流。记住，你对面坐着的是面试官，不是墙壁，你需要跟他有眼神交流。不要怕，试着抬起头来，面试官的笑容多半可以缓解求职者的紧张情绪，以及答不上题的尴尬气氛。之所以害怕，其实是自己吓倒了自己。
- 3) 气氛把握。语速不要太快，太快就容易将自己置于紧张的状态之中，回答问题无论会与不会，都要放慢节奏，松缓身心，因为你的状态会直接影响面试官的感受以及判断。
- 4) 背景了解。如果你参加一家公司的面试，最好是你真心喜欢的，并且对公司多少应该有些了解。例如公司理念、制度、规划，谈谈你喜欢的、你认为可以改善的（这一点上要注意“度”），如果你用了心，面试官往往会给予更多机会的。
- 5) 轻松话题。如果谈得比较愉快，求职者可以自己制造些轻松话题，例如旅游、业界话题等。

很多师弟、师妹们问我，如何挑选 offer，需要权衡哪些内容。我不是一名职业规划师，所以不能告诉他们如何做选择，我只能告诉他们，当初我在选择 offer 的时候，考虑了哪些内容，以供他们参考。但总的来说，我觉得应该参考以下 5 点内容：

- 1) 兴趣点。兴趣是最好的老师，如果没有兴趣，你很难在工作岗位上有所作为。
- 2) 公司未来发展空间和路线。很多时候不能只盯住眼前的利益，要从长远看，一个企业的发展空间和路线、对未来市场的认知与把握都会决定你未来的发展方向，所以，最好能够对企业的未来发展空间与路线有一个较清醒的认识。
- 3) 薪酬福利。“钱不是万能的，没有钱是万万不能的”。一个企业再好，如果不给工资，同样没人会去，因为人要吃饭穿衣，所以必须仔细考虑薪酬福利。
- 4) 个人成长点。每个企业对人才的定位都不一样，所以在选择 offer 时，尽量选择一些企业的核心研发部门，在这样的部门里面，个人成长、个人机会都会非常好。
- 5) 城市。什么样的城市是自己希望的，是政治中心北京，还是东方明珠上海；是人间天堂杭州，还是千年古都西安；是天府之国成都，还是千湖之城武汉。各个城市有各个城市的优劣，所以没有谁能够告诉你哪个城市好哪个城市不好，关键需要你自己拿主意。

其实，选完了 offer 之后，就面临着一个从学生到职业人身份的转换了，如何转换角色，我个人觉得新人入职之初，最重要的就是练就基本功，这个阶段犹如蚕蜕，是痛苦但也美丽的变身。例如，我们做的是线上一级系统，承载着每秒数万笔交易的创建及支付，那么系统的架构、稳定性、容量、可扩展性以及各种底层技术实现，方方面面要学的太多，任务紧、压力大、面对着无数个不可能，这个过程看似痛苦但却会让我们成长得非常快。尤其是当项目真正上线运转起来时，那些你原先认为不可能做的事情都做到了，还做得非常漂亮，那种成就感真的是无以言表。而且做每件事情的时候，一定要把姿态放下来、心态静下来、自信提上去，与你的团队一起合作，把不可能当作历史，把可能写在今天。经历一段时间的锻炼之后，你应该仔细思考一下，问一问自己是否可以独当一面，是否在业界，至少在公司部门内，可以听到你的“声音”、可以看到你的建议，如果可以，那么恭喜你，你应该可以升职了。

1.3 脚踏实地，培养多种技能

廖兰新，男，硕士，某创新型企业高级研发工程师、开发经理

作为一名一线的技术研发人员，我结合自己这么多年来在工作中的经历，分享一些经验给即将走入职场的毕业生，希望能帮助他们在求职的路上少走一些弯路。

(1) 行业选择

我个人觉得应届毕业生择业时，选择适合自己的行业是非常重要的。对于计算机类专业的毕业生，可供选择的行业很多，例如商业银行类、国企、央企、传统的软件公司、新兴的互联网公司等等。而这些行业又各有各的特点，对求职者能力的要求迥异。例如国企的工作相对轻松、薪资一般（体制内）、福利很好，对技术要求不是太高，对项目进度的要求一般不紧迫；互联网公司工作一般比较辛苦，对项目进度要求非常紧迫，技术研发能力也要求高，而企业文化一般较为自由，其薪资待遇也相对较高。所以，求职者应该根据自己的兴趣爱好以及能力特点选择合适的行业。

(2) 技术领域选择

随着现代化管理技术的不断发展，IT企业中的技术分工也越来越明显。俗话说：“隔行如隔山”同样是计算机科学技术，不同技术领域的人在技术上也是非常迥异的，例如互联网企业与芯片企业关注的重心就不一样。毕业生一般也很难做到“通才”。所以，在求职时，求职者应尽量选择自己喜欢或擅长的专业领域，这些会决定你今后职业生涯的主要工作内容，而且一般也不会轻易更换。

(3) 雇主选择

不同的雇主对求职者要求也不一样。以大型科技公司与创业型科技公司为例加以比较。创业型公司一般研发人员相对较少，每个研发人员都需要能够独当一面，对整个产品的核心代码都了如指掌，上至前端开发、Web界面，下至后台底层实现、操作系统，所以这对于个人成长是非常好的锻炼机会，但同样，创业公司也有其自身的局限性，由于工作的需要，员工一般身兼数职，经常加班，而且在专业技能上都不够规范，相比大型科技公司完善的团队、严格的规章制度等，相对欠缺。

但总的来说，创业型公司更能全方位地激发个人潜能，多角度地发展个人能力，而大型科技公司可以集中锻炼某项专业技能。当然，上述说法也不是绝对的，比如某些小型高科技公司也聚集了业内的人才，完全具备大企业的“高精尖”特点，而一些大公司的某些部门在初创阶段可能也会像创业公司一样艰苦。如果难于抉择，那你就尽量去一家已步入正轨的大公司吧。

(4) 求职建议

因为企业需要，我曾经担任过一段时间的面试官，帮助企业招聘新人。我们确实非常希望招聘到优秀的人才，但在招聘过程遇到了很多令人遗憾的事情。例如有的人在面试时因为紧张或是其他原因，发挥不出真实水平；有的人水平一般，却夸夸其谈，不脚踏实地，真正让他设计算法时，一头雾水。在此，我想说明一点，企业在招聘新人时，需要这样的人才：对人对事有信心、掌握多项技能、基础扎实、有冲劲、愿拼搏。所以，我建议毕业生在平时的学习中，一定要脚踏实地地学好专业知识，适当地扩展专业技能。

(5) 能力培养

进入工作岗位之后，很多毕业生颇感迷茫，很难快速从学生的角色向职业人的角色转变。我觉得IT行业的职业人应注重培养自身的3种能力：技术能力、管理能力和领导能力。职场

新人往往要靠技术能力进入职场，最初的晋升也主要依靠技术能力，它可以让你成为一名优秀的单兵或一名称职的经理，但很难让你成为优秀的经理人，因为它的杠杆效应非常有限，这时就需要第二种能力：管理能力。管理其实是对资源的管理和利用，以有效、可靠地生产产品或提供服务。管理能力一般可以通过学习得到，教育、经验、培训都是提高管理能力的手段。当然，个人的悟性也很重要，能够从表面现象中分析出规律，对管理能力来说很重要。管理能力主要是释放物的能力，它可给你一定的杠杆力量，能让你在小范围内有所贡献，但不会让你“走”很远。这时就需要第三种能力：领导能力。领导能力是释放别人的能力，再通过别人来释放个人或物的能力。领导能力作用可谓巨大，因为它有二级杠杆的效用。就领导而言，技术能力的重要性非常有限，管理能力次之，领导能力最为重要。职业道路不是单行道，而是可以从技术职位向管理职位过渡，再由管理向领导职位过渡的。

1.4 保持空杯心态

王震，男，硕士，某知名互联网企业研发工程师。

好友何昊拜托我一件事情，就是给当前程序员写一些关于求职的意见与建议，这着实有些为难我，并非我不愿意去做这样一件事情，而是因为本人入行虽然比较早，但入职却不太久，与一些资深的 IT 精英们相比，只能算是初出茅庐，所以不敢妄自尊大。不过，我非常愿意分享一下个人这些年来的几点粗浅体会，以起到抛砖引玉的效果。

对于个人的发展，扎实的基本功将更有利于你在行业里站稳脚跟，走得更远。“术业有专攻”，所谓专业，在于求深而不在于求广。当然，话无绝对，更广的知识面可以帮助你对整个大行业背景有一个比较清晰的认识，知道自己在产业链中处于一个什么样的位置，能够做出多大的成就，有多大的发展空间。结合我自己的经历，以软件类研发为例，具体而言，后台开发方向，系统、网络的底层，比如操作系统事件机制（例如 Windows 消息机制、Linux epoll 等），TCP/IP 协议栈，C/C++ STL 等，这些是服务器开发的主战场，对这里每项技术需要了解的程度应如同战场上士兵对手中所握兵器需要熟悉的程度一样。也许对小规模服务器程序开发而言，谈论这些内容可能有些夸大其辞、危言耸听，但确实存在很多需要如此考虑的情况，例如当前很多网上订票系统很难满足实际应用的需要，引起了用户的极大反感。而在前端方面，由于技术更迭较快，快速学习能力就显得尤为重要，程序员应紧跟时代潮流就要看准当前的形式，了解站在时代前沿的人有哪些，他们做了什么以及他们的研究成果有哪些。

至于经典的数据结构、算法，无论是前端研发还是后台研发都会有所涉及，即便是更深入的掌握一般也只在较专业的算法密集型领域，比如搜索、GIS 等。而对于你、对于面试官更注重什么，则看你们更侧重哪方面的内容了。

如果是已经入行的程序员应聘新的企业，经验及能力通常是面试官考查的重头戏。说得更直白一点，作为利益链条上的一环，你具备什么资本，能为公司创造什么价值，才是面试官关注的焦点所在，这也是你需要真正搞清楚并且为之准备的内容。做过什么项目，取得了什么样的成就，既说明了你的过往表现，也能将你的潜在价值表露一二。

进入工作岗位，我相信，不管是刚入职的毕业生还是已打拼多年的程序员，以“空杯心态”去融入当前企业文化，绝对不是件坏事。只有认可了企业的文化，工作时，你才能积极主动，才能上进、才能得到提升。就职业发展而言，一般公司都会有量化的绩效指标，完成这个指标即是一种自我提升，而在任务指标之外，结合自身情况制定出半年或全年个人发展规划，可以说是

自己短期能力提升的督促和目标实现的指引，有助于自己向着更明确的方向发展。

以上愚见称不上是成功的经验，只是我对程序员这个行业一点浅薄的理解而已。

1.5 职场是能者的舞台

林方超，学士，北京某上市公司软件工程师。

关于毕业生如何求职这个问题，老实说，我的“经验”并不是很多，若干年以前，因为应聘前准备得比较充分，所以命中率比较高，虽然也拿到了几个不错的 offer，但最终还是选择了现在这家企业。这么多年过去了，回过头来想想，也是感悟颇深。

我认为一个非常有针对性的准备工作，包括心理准备与知识准备，这对计算机相关专业的毕业生求职非常有用。

首先，求职者应当找准自己的位置，即通常所说的职位。一个对职位有着准确预期、对自己有着准确定位的人，在个人简历、面试中都能够表达出更准确、更吸引人的信息，而不至于投递完简历之后就没了下文。而找准一个方向、找准一个行业或是锁定一个企业，不仅可以缩小求职的范围，而且还可以让你在有限的精力、有限的时间内将准备的内容进一步深入、细化。如果做到了这一点，不管是大企业的招聘还是小企业的招聘，也不管是在笔试还是面试，你很快就能发现，真正能够与你竞争的人、能够把你 PK 掉的人真的是屈指可数，此时你就成了求职大军中“笑到最后的人”（插入一个感悟：时下往往被人普遍提及的流行技术，反而是陈词滥调，只有真正理解其中思想者才能脱颖而出，如果没有十成的把握，我宁可绝口不提）。

通过一些有针对性的准备工作后，笔试一般就不会存在问题了。而紧接着需要面对的就是面试这一关，每一次求职机会都很宝贵，每一次面试机会也很难得，而成功随时会降临。作为求职者，不应当将机会随意浪费掉，将成功拒之门外。所以，不要总以为自己运气好，可以“裸装上阵”赌一把。因为作为求职者，在与企业的博弈中，我们是弱勢的，因此，你需要对所应聘的企业以及岗位有一定的认识与了解，当然，你通常在此之前对其可能一无所知，如果此时稀里糊涂地去了，自然也是稀里糊涂地回来。其实，只要提前做好功课，这些都不是问题，因为如今的企业一般都会有自己的宣传网站，里面会详细地介绍企业的发展历程和现状，此外，不少网站在校园招聘时也会列出详细的招聘信息，这些内容都可以好好看看。至于对这些内容需要了解到什么程度，就要看这家企业在你心目中的地位了。想象一下，在面试时，当你谈及企业的一些信息时，面试官会想要给你介绍更多，甚至愿意带你去实地参观一下，那么接下来基本就可以直接谈待遇、谈签约了。

介绍再多的理论和方法，也只是“纸上谈兵”，是否可行还需要用行动来验证，只有行动了才能体会到其中的价值。如果成功拿到 offer，那是最理想的；如果没成功，最好要让面试官给你些建议，遇到他说不出来或闪烁其词的情况，说明面试官是凭个人喜好作出的判断，大可不必理会；而一针见血的评价以及善意的建议都会对你未来的求职、成长有很大的帮助，所以，不能被“一根绳子绊倒两次”，无论是成功了还是失败了，都会有所启发，成功可以收获经验，失败同样可以得到教训。

挑选 offer 也是一件比较艰难的事情。个人建议求职者最好按照自己的职业规划进行比较，但如果确实没有很明确的职业规划，或是从来没有想过职业规划这个问题，你可以优先挑选有发展潜力的工作，这样的工作会给你带来许多意外的收获，最终帮助你确定自己的职业路线，构建你的职业规划。

最后，我想说的是，职场是能者的舞台，真正比拼的是各种能力。技术是一种能力、交际

也是一种能力，发挥好任何一种能力都会使你的工作如鱼得水、锦上添花。因此，进入工作岗位后该如何发展，并非是一两句话能够回答的，关键还是要看求职者自己。

1.6 学会“纸上谈兵”

庐山，硕士，某知名搜索类公司软件工程师。

我于 2009 年硕士毕业于中科院计算技术研究所，到目前为止换过两次工作，最终选择了现在所在的这家企业。作为一个职场过来人，我也经历了很多事情，有初出茅庐时的意气风发，也有历经沧桑后的冷静思索，在这里我谈谈技术类职位面试应该怎样准备。其中有一些建议也是与产品类面试相通的。

在谈论面试笔试如何准备前，首先我想说说一些求职者在应聘过程中常见的误区：一是认为 GPA（成绩）越高，则面试成绩越好；二是认为编程的技术越好，面试成绩越好；三是认为在纸上写代码与在计算机上编程是一样的，不用准备或是不用特殊准备。我个人觉得，这些理解都是片面的。事实上，虽然说面试是一种主观行为，但它也是一种考试，准备的因素占了 50% 以上。但它又不同于高校中的考试，因此与 GPA 关系不大。

既然准备如此重要，那么求职者就要做好读技术面试书的准备了。此类书籍非常多，每本又都很厚，怎样在有限的时间内，从众多考点中识别出面试官常问的那些问题呢？规律是有的，因为面试官们精力有限，很少去凭空想象一些题目，很多都是套用现成的知识点，所以不论你申请什么职位，考点总会以这样那样的规律出现，复习中遇到就要记住。一般情况下，求职者需要注意以下方面的内容：

- 1) 列举处常考。在复习时看到一个知识点分成几个项目列出来的，就很可能是要考的，例如“在网页中使用 CSS 有 3 种方式，inline, internal 和 external”。
- 2) 比较处常考。例如“C 中的 auto, static, register 和 extern 有什么区别？”“const 与 define 有什么区别？”“C++ 中 struct 与 class 有什么区别”等。
- 3) 性能优化常考。例如“怎样提高网页加载速度”“如何提高数据库查询效率”“内存泄漏的原因、识别及防范”等，在 C 语言、Java 语言和算法方面也会经常考到类似的问题。
- 4) 算法设计与实现常考。经常会针对某些特定的算法对求职者进行考查，同时时间复杂度也很容易考，所以求职者要在掌握好算法原理、代码实现的同时，记住它们的复杂度。

除掌握常考的考点外，求职者还要练习在纸上编写程序。脱离了功能强大的 IDE（Integrated Development Environment，集成开发环境），在纸上编写程序就与在计算机上非常不一样了。这里没有自动提示，没有语法高亮，没有拼写纠正，没有自动编译、链接与运行，全凭求职者平时的积累。但是在笔试和面试中，常常要当场“纸上谈兵”，如果不熟练就要吃亏，所以这一关必须要过。

1.7 小结

尽管每一个面试官的工作背景不一样，个人能力也不一样，而且面试套路也可能各不相同，但是，他们的目的只有一个，那就是发掘最适合企业的优秀人才。对于求职者而言，面试官的判定往往决定了求职者的去留，所以，求职者应该好好斟酌面试官的意见与建议，认真体会，进而不断地提升自己，努力让自己成为企业青睐的“千里马”。

第2章

面试心得交流

“前车之鉴，后事之师”，本章以各大名牌高校、研究所的毕业生的亲身求职经历与体会为蓝本，对当前程序员面试、笔试相关准备工作、时间计划、书籍阅读、面试技巧、offer（录取通知）选择等多个方面的内容进行了独到的分析，对于即将踏入职场的毕业生有着很好的指引作用。

2.1 心态决定一切

董哥，男，中国科学院计算技术研究所2012届硕士研究生，现就职于北京腾讯搜搜。

1. 抛砖引玉

找工作的过程是较量综合实力的过程，一个好的 offer 凝聚着无数辛勤的汗水，需要勤奋、坚持、积累和付出。我在这里介绍一下我自己找工作的经验，希望对师弟师妹们有所启发。需要注意的是，完全做到了这里提到的几点并不意味着一定可以拿到一流的 offer，这仅是抛砖引玉，如果想在找工作时得心应手，你还需要平时不断积累和总结，领悟其中的真谛。

2. 心态决定一切

对于找工作，心态很重要，找工作之前，求职者一定要端正心态。20年寒窗苦读，最重要的一个目的就是找一份理想的工作，从而实现自身的价值，因而我觉得，求职者至少应该像准备高考那样，全身心地投入到找工作的准备中，将之前所学知识重新温习整理，以便将自己的所有能力最大限度地发挥出来，向面试官充分展示自己。

3. 冰冻三尺非一日之寒

关于找工作前的准备，有两个因素直接决定求职者是否能最终被录用：一个是项目；另一个是基础知识。这两者中的任何一个被面试官所认可，求职者均可能拿到 offer。

对于项目，不在多而在精。一般的项目，如普通的管理系统、网站等，面试官几乎不用耗费脑力，一眼就能看到底，没有什么好讲的。最切合也最能引起面试官兴趣的项目往往是与他现在所从事的领域相同或相近，解决的问题的确具有一定的难度，且提出的解决方案具有一定的创新点。但遗憾的是，大部分毕业生所做过的项目的深度往往不够，毕竟想在短短两三年时间里成为这方面的专家，还是比较有难度的，所以这个时候就全靠求职者的基础知识了。

基础知识大致可分为以下几个部分：编程语言、数据结构与算法、操作系统和其他小知识点。就编程语言而言，个人认为 C 语言是必须掌握的，很多公司把 C 语言作为必考项。另外，要在 C++ 和 Java 两种面向对象的编程语言中选一个，主要知识点是面向对象编程中的一些基

本概念，如虚函数、构造函数、析构函数、拷贝构造函数等。有一些题目已经成为经典，是必须、一定要掌握的，例如（C++ 语言）虚函数是怎么实现的？构造函数可以是虚函数吗？为什么鼓励将析构函数设计成虚函数？数据结构和算法是面试的重点，很多公司基本上只考数据结构与算法，这就求职者平时多积累、多练习。尤其对一些基本数据结构和算法，要非常清楚，例如单链表反转、Trie 树、两个数组交并差集等。就操作系统而言，求职者应主要掌握 Linux 里的一些基本概念，如线程、进程、内存管理、文件管理等（这些也会在面试中出现，求职者一定要好好复习）。最后是一些其他知识点，如设计模式（单例、工厂模式等）、编译原理（程序从编译到运行要经历的几个过程）等。

4. 修炼程序员之“葵花宝典”

找工作过程中，求职者一定要反复推敲一些经典的题目。这些题目大多来自固定的几本参考书，求职者应该好好琢磨一下这几本书中的题目。

(1) 《编程之美》

这是一本实战书，从事程序员一职的人都知道，很多笔试、面试题直接来自该书，值得各位认真阅读。该书中有些题目对于初入职场的求职者难度过大，从找工作角度考虑，可暂时不看。

(2) 《编程珠玑》

该书主要介绍软件设计思想，书中的例子已经成为百考不厌的经典题目，如数组循环移位、随机采样算法等。

(3) 《算法导论》

该书对各种常见算法进行了深入的讲解和详尽的证明，并对每个算法的起源、动机和求解过程有较多的涉及。

(4) 《深入理解计算机系统》

该书从程序员的视角介绍了计算机系统，几乎囊括了计算机的各个技术，包括数据表示、C 程序的机器级表示、处理器结构、程序优化、存储器层次结构、链接、异常控制流、虚拟存储器和存储器管理、系统级 I/O、网络编程和并发编程等。该书中提到的一些知识点，常作为面试题目出现，比如 Linux 信号量、虚拟内存管理等。

5. 八面玲珑

关于找工作的技巧，这里主要介绍两点：一是回答问题的技巧，对于项目，主要回答点应该是遇到的挑战和解决问题的思路；对于算法问题，要从复杂度高的算法逐步向复杂度低的算法过渡，第一眼见到题目，求职者可先将自己想到的思路说出来（比如 $O(n^2)$ 复杂度），然后不断优化（比如 $O(n \log n)$ 复杂度），最后尽量得到一个最优的算法（比如 $O(n)$ 复杂度），这时候可能要在纸上写出来，一旦没有了思路，应该主动要求面试官加以提示；二是交流技巧，这里指的是求职者之间的交流，这一点非常重要，当前一位求职者面试完后，你应该主动跟他交流，主要询问一些个人收获、心得以及失误，因为面试官一天之中要面试众多的求职者，很可能对不同的求职者提出相同的问题。

6. 多多益善

最后是 offer 的选择。求职者应尽量多拿一些 offer，以便给自己留一些选择的余地，至于怎么选择 offer，这是个人的问题，每个人侧重点不一样，因人而异，但我觉得适合自己的就是最好的，没必要和别人进行比较。

2.2 假话全不说，真话不全说

萧叶，中山大学2012届硕士研究生，现就职于睿初科技（深圳）有限公司。

1. 万事趁早

我大概是研究生三年级新学期开学后开始准备找工作的，从后来的情况来看，我已经准备得很晚了，因为校园招聘的时间好像提前了半个多月。这也给了我一个教训——万事趁早。因为我们不能预知公司什么时候来招聘，只能自己提前做准备。准备太晚的结果就是，我没能赶上9月中下旬的阿里巴巴、淘宝等公司的招聘。

2. 不经一事，不长一智

虽然我很早就确定了找工作的两条原则——去外企和搞技术，但是当校园招聘开始时，我几乎还是逢公司必投简历（当然得是软件研发类的），一来是因为自己手头无 offer，心里总是有些没底，不知道自己是否能够找到满意的工作，特别是看到周围同学暑假实习回来就拿到了 offer，心里不免更加担心，紧迫感更加强烈；二来是因为本科毕业就直接读研了，没有真正找过工作，对找工作还是很陌生。

所以我认为，“海投”也没有什么错，虽然“海投”的这些公司并不都是自己非常想去的，但是如果不能趁早积累和总结属于自己的找工作心得，等到心仪的公司来时胜算有多大就很难说了。

3. 读书破万卷，面试如有神

因为准备得比较晚（个人觉得从暑假开始准备算比较合适的），所以我基本上是一边找工作一边准备面试笔试，而准备的方式主要就是看书。对于大多数没有项目经验或项目经验较少的研究生和本科生而言，看书是投入产出比最高的准备方式。因为笔试面试最常见的内容不外乎语言、数据结构与算法、操作系统、软件工程等内容。语言类靠编程指南之类的书籍即可，对于其他专业知识点，我认为比较有帮助的书籍有《(more) Effective C++》《(more) Exceptional C++》《C++ Common Knowledge》《算法导论》等。语言类书籍给出的都是语言规范等确定性的知识（告诉你是什么），非常适合应对笔试；而后一类书则好比内功心法（给出一个场景，分析各种方案的优缺点，告诉你为什么是这样），看这类书的收获，与编写的 C++ 代码量正相关，面试时专业能力较强的面试官喜欢问这类问题。这个系列的书，无论读者水平是高是低（当然基本语言知识得懂），总能从中领悟到一些东西，而且每次再读，又会有新的体会，不仅适合找工作时读，平时读也有助于提高自己的业务水平。至于算法方面，我认为这不是看看书突击一下就能显著提高的，就算把那些常被问到的排序算法硬背下来，面试时也不太管用，这个还是要靠平时的积累和领悟。

4. 人性化的简历

在简历的制作上，排版可以讲究些，这样做的目的是让筛选者快速、准确地找到他所关注的内容（如技能、项目经验、成绩等），以两页为宜（有人说最好一页，但个人感觉一页根本写不下，也容易让筛选者觉得材料有点单薄）。至于打印，我觉得最好选稍厚一些的纸，至少不能很清晰地看到背面。总之，要让简历的筛选者拿着、看着觉得舒服，彩打就不必了（明确要求的除外）。

5. 假话全不说，真话不全说

面试到了尾声时，面试官（通常是技术主管、人力资源或经理）有时会问求职者有关职业规划、家庭背景、已经拿到了哪些 offer 等情况。尽管在此之前，很多师兄师姐给我传授了

相关技巧，但是我还是按照自己的真实想法做了回答，也许正是因为我太“老实”，最终与几个公司擦肩而过。华为、爱立信都问了我拿了哪些公司的 offer，我如实回答了，还有一家公司问到，如果给我 offer 我是否签约，我说要考虑一下。我觉得实话实说并没有什么不当，一个对自己负责的毕业生找工作时“货比三家”，最终选择自己最满意的工作是无可厚非的，企业应该能够理解这一点。但实话实说也并非一定要回答面试官的所有问题，例如有一位面试官对我家里的情况问得过于详细，还有两位面试官问到了其他公司给的待遇问题，我都没有正面回答。当然，拒绝回答问题就要靠技巧了，求职者要尽量委婉地拒绝，不要太过直接。

6. 豆腐白菜，各有所爱

对于 offer 的选择，这是一个见仁见智的问题，我个人觉得，自己最满意的就是对自己来说最好的。我找工作主要遵循两条原则：第一，以外企为重点，希望将来能有机会到国外工作，但也并不是非外企不去；第二，非技术类的工作不做，因为我知道自己不适合也不太喜欢做售后、策划等工作。结果，拿到的几个 offer 中，爱立信和深圳睿初科技都是符合这两条要求的。这两家公司中，爱立信给予的是带附加条件的 offer，要求现在能够过去实习至少两个月，人力资源和项目经理先后打电话问了两次，看得出来对方确实急缺人手，但是我的导师不同意实习，所以我只好作罢。而深圳睿初科技是我找工作以来遇到的所有公司中流程最严格（1 轮笔试，1 轮电话面试，4 轮现场面试，两轮总部的电话面试）却又最人性化的一家公司，我对它的期望和好感就是在新一轮又新一轮的面试和沟通中不断提升的，以至于它最终给我 offer 时，我毫不犹豫地签了约。

其实，我觉得求职者先得确定自己找工作的原则，明白什么是自己最为看重的，然后重点准备符合自己原则的那些公司的笔试面试。

2.3 走自己的路，让别人去说吧

小郭，女，西安电子科技大学 2012 届硕士研究生，现在计算机网络与信息安全教育部重点实验室攻读博士学位。

时间过得飞快，转眼间，找工作大潮已经过去了一段时间，这是我第一次找工作，现在把自己找工作的一些情况以及心得整理出来，一来对自己的经历是一个总结，二来可以为以后找工作的师弟师妹们提供一些信息。我本科期间读的是计算机科学与技术，毕业后被直接保送到本校的计算机软件与理论专业读研，研究生阶段从事的基本都是软件类研发工作。

1. 无悔的选择

在研二时，我就开始在找工作还是继续念博士之间犹豫不决，但紧迫感不够。到了研三，不能再犹豫了，我决定先找工作。最后，我真正拿到了 4 个 offer：华为的软件研发、阿里云的无线平台开发、百度的客户端研发和腾讯的后台研发。尽管我最终选择了攻读博士学位，但这一阶段的经历还是让我积累了一定的经验。

2. 出师未捷身先死

我是从研二放暑假回学校后开始着手找工作的，应该算比较晚的，复习的内容其实就是《面试指南》《编程之美》和各种专业课书（比如数据结构、操作系统、计算机网络等）。近年来，校招的时间越来越早，当第一批公司来的时候，我还很多内容没有复习。

最早来的是联发科。毕竟是第一次找工作，当时我心里还是很紧张的，笔试题不算难，我顺利过关了。接着就是一面了，大概半个小时的样子，主要问的就是实验室做的项目；一面结束后等待二面消息，可是当身边很多同学都收到二面通知时，我却并没有收到。第一次找工作就

碰壁，当时对我打击还是挺大的。后来我静下心来总结了这次面试失败的原因，其实联发科的面试问题并不是特别高深，都是一些基础知识，失败的主要原因我觉得在于两点：第一，面试太紧张；第二，准备不充分，尤其是项目部分，与面试官的沟通不是很好，面试官对我做的项目应该没有什么了解，而我所说的又没有提起面试官的兴趣，因此我说的话面试官不懂，面试官提的问题我也没有清楚明了地回答。

3. 过五关斩六将

九月下旬，华为、中兴等公司陆续开始了校园招聘。华为面试的场面非常壮观，每天参加面试的学生数以千计，4轮面试不停，我是从下午1点开始面试的，第一天直到晚上9点才面试了3轮，而第4轮面试要等到第二天，于是我拖着疲惫的身体返回了学校。华为的面试一共分为4轮，分别是技术面试、上机测试（上机编程）、性格测试和HR（人力资源）面试。技术面试的面试官就问了一下实验室项目，然后让我写了个简单的程序，接着是上机测试与性格测试，上机测试并不是要求编写的程序完全正确，而是面试官根据写的程序进行打分，然后参照同一批人的水平来决定是否通过。而最关键的就是性格测试了，很多人都在性格测试这一关止步了。我一个同学就因为性格测试的时候仔细斟酌，害怕回答得不好，最后没有通过性格测试。对于性格测试，我的心得就是不要太紧张，放轻松点，做题前后要保持一致，尽量不要前后矛盾，按自己的真实想法耐心回答即可。第二天进行的第4轮面试其实也只是随便聊聊天，面试官询问了我的家庭背景以及与一些与技术无关的问题，就直接发给我口头 offer 了。

之后是百度、腾讯和阿里云3家互联网公司，我感觉百度最注重算法，面试时间也最长。其实能拿到这3个 offer，我个人觉得很重要的一点就是心态，我去面试这3家公司的时候心里很放松，没有一丝紧张，权当是去锻炼锻炼，这样效果反而不错。当然也不是只要不紧张就可以了，面试成功的因素是多方面的，与求职者所遇到的面试官、当年的就业形势都有一定的关系。但是在自身方面，除了心态好，还要有充分的准备，尽量把自己会的、面试官也感兴趣的东西表达出来。在项目方面，因为有了之前面试的经验，我与面试官讨论项目越来越熟练，对项目的理解与总结也越来越好，这无疑为我求职的成功打下了良好的基础。此外，我个人认为不仅要对自己做过的每一个项目做充分准备，而且一定要实话实说，因为每家公司注重与感兴趣的内容不同，或许他们会对你没有准备的项目很有兴趣，如果这时候你显得很生疏，那么就很不利了。例如面试期间，阿里云对于我曾经参与过的编译器有关的项目感兴趣，而百度则对网络安全中的身份认证感兴趣。当然，实话实说的意思是不能说假话，但是并不意味着要把所有实话都说出来。如果说假话被面试官拆穿了，那么求职就彻底没戏了。有时候可能有人会抱着侥幸心理，不过我碰到的这3家公司的面试官对我简历上写的项目总有一个会很熟悉，有的甚至不止熟悉一个，因此我建议求职者还是踏踏实实、实话实说。这3家公司的面试题与华为、中兴的区别甚大，他们更注重的是求职者的能力和反应，面试官可能会就一个问题与你讨论很长时间，如果很顺利地回答好了，那么面试官会将这个问题延伸；如果回答不出来，面试官会给你提示并与你讨论。总之，和面试官交流的过程就是把自己的能力展示给面试官看的过程，就算回答不出来或者答得不完美其实也没什么关系。

4. 成绩第一

除了心态好、对项目熟悉之外，最关键的就是技术了。在面试过程中，我虽然没有把面试官问的问题全部回答出来，但是也回答得八九不离十，因为有很多面试题所涉及的知识都是我以前在实践中或者在技术书籍中看到过的。在研究生阶段，我利用课余时间看了不少专业书籍，如《编程之美》《编程珠玑》《计算机程序设计艺术》《Windows 程序设计》《C 陷阱与缺陷》《C 专家编程》和《深度探索 C++ 对象模型》等。这些书籍对我找工作的帮助非常大，不仅足以应对笔试，面试，还让我从中学到不少考虑问题的思路和方法。

在面试过程中，笔试成绩高还是很有优势的，我在阿里云面试时，因为笔试成绩很高（后来面试的时候看到的，接近满分），面试官也对我很有信心，尤其是第3轮面试的时候，部门负责人直接说已经可以确定我通过了，整个过程中都说我的笔试成绩很好，因此没有问什么技术问题，而是给我介绍他们的工作情况。

拒绝我的 Marvell（美满）上海研发中心是一家全球领先的半导体厂商，因为是外企，因此他们对英语有较高的要求。我很早就向这家公司投了简历，过了很久之后才接到了他们的面试通知。一面结束后我才了解到，让我去面试是因为我有参加 ACM 竞赛的经历，所以在这里插一句，有机会的话一定要尽量多参加一些竞赛，一来可以锻炼自己的能力，二来可以增加一些经验，而且有可能会让你拥有比别人更多的机会。Marvell 的面试一共有3轮，3个面试官全都问技术，3面下来花了四五个小时，面试官不同于上面提到的那3家互联网公司那种随和的感觉，每个人都很严肃。第一个面试官主要问我算法，让我设计一个两部电梯的调度算法，主要从人性化的角度去考虑，我设计了几个方案之后面试官都不太满意，算法题结束之后又用英语交谈了一下；第二个面试官主要问的是与项目有关的内容，还有一个与专业无关的测试，问项目的时候问得非常细致，幸亏我面试之前有所准备。这些结束之后他让我说说如果让我测试一款手机我会怎么测试，越完整越好，由于在此之前我曾去中兴西安研究所参观过手机测试部门，因此就说了一些自己见到的，面试官对我的回答应该还算满意。第三个面试官的问题包罗万象，软硬件都有所涉及，软件我还能应付，硬件就有些力不从心了，因为研究生阶段我没有接触过硬件。当天面试完毕之后，我感觉希望不大，不料过了一段时间，我收到了 Marvell 美国总部的邮件，叫我把 GPA 和英语简历发给他们，当时我已经决定攻读博士，还在准备英语考试，所以就没在意，随随便便发了一下，之后就没回音了，我感觉后来没能收到 Marvell 的 offer 很有可能是因为英语简历不过关，那份英语简历是我在暑假的时候草草做的，没有修改，很多地方都不完善（甚至有语句不通的可能）。虽然没能收到 Marvell 的 offer，不过我的收获还是很大的，这次面试让我知道了自己的知识和水平还有很大的提升空间，进而为我后来的学习提供了动力。

5. 走自己的路，让别人去说吧

最后我拒绝了所有 offer，继续攻读博士学位，这里有一些主观原因，也有客观原因。总之，选择了就要走下去。其实每个人都会在生活中遇到很多选择，我觉得不管你选择了什么，只要是自己选择的，就不要后悔，踏踏实实地走下去，坚持是最重要的。

2.4 夯实基础谋出路

Jololee，男，浙江大学2012届硕士研究生，现就职于杭州网易游戏。

我目前就职于网易游戏（杭州），以下是我对求职的一些感悟。求职时，我申请的职位是服务端研发工程师，现在从事的是在游戏部门做后端研发工作。

1. 万事不备

我是从7月份开始准备找工作的，刚开始并不算太努力，断断续续，自己也比较松懈，而且中间还得给导师做项目，所以只是零零散散地进行复习，对知识点的掌握也并不是非常精深。直到9月份，重心才完全投入到找工作中，我开始看一些专业书籍，如《算法导论》《C专家编程》等。

2. 夯实基础谋出路

对于面试笔试的准备，我觉得基础是根本，所以需要多学习一些基础知识，参考的图书有

《算法导论》《数据结构》《深入 Java 虚拟机》《Java 多线程模式》等，其他内容由于时间紧迫，看得比较匆忙，如《编程之美》《编程珠玑》，另外，编程指南类速成书籍我也看过，不过感觉一般，仅可以应付小公司的面试。所以我的经验是如果时间允许，“多读书，读好书，夯实基础”才是“王道”。

3. 字字珠玑

我找工作的过程真是几多坎坷，现在分析关键原因还是自己准备得太晚。9月份第一波招聘潮到来的时候，我还没有看过《编程之美》等书籍，这也导致我与一些好公司失之交臂。

简历制作要区分国企、私企、外企，国企考查求职者的综合素质，他们一般更注重综合素养，而不仅仅是技术细节；而私企一般会深入考查，需要把求职者简历上的每个项目都弄清楚，要求技术扎实、深入；外企需要求职者能够用英语讲述自己的经历、说清楚一个项目的工作以及具备良好的表达能力。

不准备算法，错过一半公司；不准备项目经验和技能，错过另一半公司。如果想去外企，英语好是必需的。

4. 多方询问

应届毕业生可以从师兄师姐那里得到一些关于企业的详细资料，也可以从学校 BBS（水木清华、饮水思源、飘渺水云间、西电好网、北邮人等）上的帖子获取相关信息，还可以广泛征求同学或朋友的意见和建议。一般实验室应届生毕业每年去的公司都差不多，要善于与毕业的前辈联系，多询问他们的建议，他们一般也会毫无保留地给予非常善意的回答。

5. 忠言也顺耳

找工作过程中的坎坷让我身心疲惫，但同时也受益匪浅。最后得到的结论就是应届生的水平一般不会差距太大，如果想把工作找好，就要下真功夫、下苦功夫，就跟高考冲刺一样，方能水到渠成。

9月份第一波招聘会来时，你就必须要把基础知识、算法、智力题、英语准备好了。否则你只能惨淡地接受教训，并在国庆后第二波高潮之前发奋努力、加紧追赶了。不过那样时间会比较紧张，效果往往不是太好。

最后，签约要慎重，如果觉得没有找到好工作，一定要坚持，不要以为后面没有机会就盲目签约。进入招聘后半段，大多数公司都会补招，这是坚持到最后的人才有的机会。

2.5 书中自有编程法

涛哥，男，西安电子科技大学 2012 届硕士研究生，现就职于华为技术有限公司西安研究所。

我虽然找到了一份不错的工作，但很难说有什么成功的经验。这里，我总结了一些求职过程中的经验教训，以期对后来的求职者起到些许警示与借鉴作用。

1. 选择因人而异

经过近一个月的努力，最终我真正拿到两个 offer：一个是华为技术有限公司西安研究所的云计算研发的职位，另一个是腾讯深圳的无线终端开发的职位，最后我选择了华为。放弃薪水更可观的腾讯而选择华为，一方面是由于我做的项目都是用 Java 语言开发的，自身对 C++ 不太熟悉，腾讯给的 offer 是终端开发，而我对此不是很感兴趣；另一方面是个人感觉深圳的生活压力大，我不想在工作压力大的同时，生活压力也这么大。这两个原因使我最终在华为“落户”了。

2. 有所不为才能有所作为

联发科当时在全西安进行招聘，没投简历的也可以参加笔试。由于联发科是最早来招聘的大公司，因此参加笔试的人特别多。其实笔试题目不难，考的都是一些基本的数据结构、操作系统、计算机组成原理和 C 语言的知识，有三四道《编程之美》上的算法题，做完后我感觉良好，之后顺理成章地收到了后续的面试通知。后续一共经过两轮面试，第一轮面试一共 10 分钟，问题只涉及了项目；第二轮面试我的是两个部门经理，也没问技术，时间大概有 20 分钟，提了提项目和性格方面的问题，最后他们决定给我 offer，然后我婉拒了，因为我想做后台开发，而他们提供的职位是终端开发。

3. 落花有意流水无情

我一直想去的是阿里系的公司，原因有两点：第一点是我觉得整个阿里系的公司技术和氛围比较好；第二点是我非常佩服马云。

阿里云是 9 月中旬开始校园招聘的，在西北工业大学笔试，1 个小时要做十几道算法题，因为太想进阿里云了，所以我非常紧张，最后没有发挥好，笔试都没通过。所以，在此提醒以后找工作的师弟师妹，找工作时心态一定要放平，相信自己。后面的笔试、面试我就非常淡定，但还是被百度和淘宝两家公司淘汰了。百度是 3 个小时做 10 道左右的算法题，应该是我参加的所有笔试中题目技术含量最高的，也是最难的，感觉只有 3 道题目肯定答对了。后来收到了百度的面试通知，一面的时候问了 3 个技术题，一道是数学题，一道是问 LRU 页面调度算法用程序怎么实现，还有一道是文件分布式存储方面的，感觉回答得不太好，果然后面没有收到二面的消息。而淘宝跟阿里云的笔试很像，时间也是 1 个小时，题量比较大，题目比较难，我笔试也没通过，淘宝和阿里云的失利对我打击颇大，因为它们是最想去的公司。

国庆之后没再找工作，一是因为没信心了，二是通过应聘这几家互联网公司，我发现自己的实力确实不行，首先是基础不扎实，对专业课中的知识点仅知皮毛，理解不深，其次是算法部分太过薄弱。

4. 书中自有编程法

在这里，我给大家推荐几本对找工作和以后从事软件技术工作有帮助的书籍，不过大家还要根据自己的喜好来进行选择。

专业基础：《深入理解计算机系统》《操作系统》《数据结构》。

算法：《算法导论》《编程之美》《编程珠玑》《编程珠玑 2》《计算机程序设计艺术》系列，算法的提高还要平常多做些题。

C：《C 语言程序设计》《C 陷阱与缺陷》《C 专家编程》《C 和指针》。

C++：《C++ 程序设计语言》《Effective C++》。

Linux：《UNIX 环境高级编程》《Linux 设备驱动程序》《深入理解 Linux 内核》《UNIX 网络编程》卷一、卷二。

Java：《Java 编程思想》《Java 虚拟机》《Java 与模式》。

5. 充电和实践非常重要

如果立志做软件研发工作，那么求职时最重要的还是技术实力，而实力的练就需平时的积累，现在还在上本科的同学要抓紧时间了，不管是工作还是读研，都要坚持每天给自己充电，如果有机会读研，尽量选准一个自己喜欢的方向，把大量的时间放在上面，而且要跟导师、师兄师姐以及企业里的人多交流。对于从事技术工作的人而言，实践是非常重要的，除了实验室的项目，大家还可以参加一些竞赛，如果在技术含量比较高的竞赛（ACM、腾讯创新大赛、华为创新设计大赛、“中兴捧月”程序设计大赛、百度之星等）中拿过奖，对找工作会有很大帮助。所以，我的建议是，

如果实验室有比较好的项目，那就做实验室的项目；如果没有，那就多参加一些竞赛。

2.6 笔试成绩好，不会被鄙视

小白，女，电子科技大学2012届硕士研究生，现就职于中国电子科技集团某研究所。

要说给学弟学妹们提点建议，我想从找工作前的一些方面说起。毕竟找工作也就那么短短几个月，真正决定应聘结果的是最初的一些准备。当然，找工作也不排除运气成分，但是运气并非我们所能掌控的，所以做些我们能够做到的事情才是最重要的。

1. 知己方能百战不殆

首先我要说的是，求职者一定要想清楚自己要什么。有不少人读研究生，其实并没有想清楚自己以后到底想要从事什么样的工作。和大部分人一样，我从一开始就是完全听老师的话，没有任何自己的规划与计划，老师让做什么就做什么，而不会去想自己为什么要做，怎么做更好。所以我建议大家从一开始，至少从研一修完学分开始，就想想自己想要怎样的工作。想去外企的，尽早做好英语的准备，毕竟英语好，对于进外企还是很有帮助的。其次要牢固掌握专业基础知识。专业基础扎实，成绩优秀也是外企较为看重的。对于进私企或其他单位而言，当然也需要为以后从事的工作做较多的准备。我是从事算法研究的，找工作的时候就比较苦恼，因为现在大多数IT企业，无论是招聘硬件工程师还是招聘软件工程师，都看重求职者的编程能力，他们很少将重心放在算法研究上，而我研究的算法面比较窄，他们也并不是很了解，所以在招聘过程中还是比较被动的。鉴于此，希望大家还是尽量多完善自己这方面的能力，不要等到找工作时，才手忙脚乱地开始准备。当然很多时候，作为学生，我们没有选择的权利，研究方向都是导师指定的，必须要做一些科研方面的东西，这时，就要合理地安排好自己的工作了。倘若想进研究所，算法方面的研究还是必要的，有比较出色的文章发表也是一个加分项。倘若想读博，一门心思搞学术才是硬道理。所以，大家提早衡量好自己的发展方向，有的放矢，绝对是有益无害的。

确定好了工作类型，下一个问题就是工作地点的问题了。在正式开始找工作之前，我建议大家结合自己的实际情况，和父母好好商量一下，如果有男/女朋友，也可以和男/女朋友好好商量一下，自己也多做些思考。毕竟全国各地的工作岗位那么多，海投的效果犹如大海捞针，更何况个人也没有那么多精力。毕业生应确定好几个工作地点，有针对性地选择准备，才能事半功倍。

2. 笔试成绩好，不会被鄙视

正式找工作开始前，求职者一定要看看找工作的相关书籍。程序员的笔试面试题目还是和做的实际项目不太一样，笔试面试一般侧重细节，更加注重基础知识的考核，所以进行这方面的准备还是很有必要的。

自己感兴趣的公司来招聘之前，求职者还是要做好提前准备工作，例如对公司的初步了解、往年的笔试面试题目等。公司面试时喜欢问“为什么选择我们公司”之类的问题，若求职者能回答得比较得体，印象分也会不错。

3. 诚者，天之道也；思诚者，人之道也

关于面试，本着相互尊重的态度，求职者应该穿戴整洁，必要的场合还应该穿正装。第一次参加面试，求职者心理上可能会有些紧张，其实面试笔试多了，也就习惯了，就会从容很多，所以把起初的几次面试作为锻炼自己的机会也未尝不可。

在面试期间，求职者应做到礼貌、大方，对于对方的问题，做短暂的认真思考后有条理地回答即可。同时我想强调一个问题，诚信还是很重要的，我自己就在这块差点栽跟头。面试前

听说面试官是宣讲会的主讲人，很在意有没有去听他的宣讲，所以当面试官问我是否有去听过他的宣讲会时，多了个心眼儿（之前也了解了宣讲的大概内容），我就回答说去了。结果没想到他突然问我啥时候去的，我完全不记得宣讲时间了，只记得是下午，大概说了个时间，结果差了1个小时，所以在此做个反面教材，给大家做个警示。

群面（集体面试）是一个面试中经常遇到的事情，像华为、华赛、腾讯产品一般都有群面。关于应对群面的方法和技巧，网上有很多，大家可以了解一下。我个人觉得，在群面中不一定要保持中立，不能多说话。而是要不该说的时候不要乱说，该说的时候一定要当仁不让。同时，注意语气态度，很多人摆着一副唯我独尊的架子，不给其他人说话的机会，其实因此而失败的原因不是话太多，而是心态没摆正。

4. offer 不在多，在于精

生活中的痛苦大多不是没有选择造成的，而是选项太多造成的，所以我个人认为 offer 不在于多而在于精，一两个保底，然后为自己最中意的 offer 再认真一搏，这些就够了。关于如何选择最后在手里的 offer，其实做好工作类型和地点的考虑后，基本也就能确定了。除此之外，在同等条件下，求职者不仅需要衡量基本工资、绩效、奖金、福利等诸多因素，还要考虑所在地的生活成本等因素。

与我同在一个教研室的某同学，从研二上学期开始，就认定了一家研究所，他详细了解该研究所的研究方向，在做实验室项目的同时，也参与了该研究所的相关项目。虽然研究得不够深入，但也做到了基本的了解，通过参与研究所的项目，不断弥补自己在这方面的欠缺，最后在找工作的时候，可谓是一击即中，所以希望大家以此为榜样。

5. 谋事在人，成事在天

我找工作的经历说难也不难，说顺利也不算顺利。确定了要回家并且进研究所的方向后，却发现大部分研究所更愿意招聘男生，而我“厚着脸皮”面试了一家并且耐心地等到了最后，幸好最后还是顺利签约了。虽然没有面试几家单位，但是心里承受的压力也不小，所以对于自己想去单位，一定要尽可能地表达自己强烈的意愿和真诚，天道酬勤，不轻易放弃，最后一丝机会也应该尽力抓住。“谋事在人，成事在天”，做出自己最大的努力，也就无悔了。

应届毕业生找工作，确实是一件大事，但其实也并没有想象中的那么重要。人的一生还有那么长，现在的认知未必和以后相同，机遇和发展都是不定的因素，所以良好的心态绝对是至关重要的，一次选择并不能确定你的一生。人的一生取决于他/她一直以来的认真和坚持，对于不可控的运气问题，应保持一个正确的态度，切不可急躁。真心不愿意的也不要屈就；感觉比较满意的，一定要把握好机会。其实就我的经验而言，只要有实力、有耐心，并保持好的心态，一般都会有较为满意的结果。

2.7 不要一厢情愿做公司的“备胎”

追风少侠，男，西安电子科技大学2012届硕士研究生，现就职于杭州支付宝网络技术有限公司。

以下是我个人的经验与教训，网络上有关大公司的笔试面试经验比比皆是，此处不再赘述，我只说一些求职过程中需要注意的地方。

1. 好学校不如好成绩

笔试成绩的好坏直接能决定求职者在面试官心目中的初期印象，而且很多面试顺序都

是按照笔试成绩的顺序排列的。淘宝的那次面试就把我安排在下午 3:30 进行，可是那天我参加了支付宝的笔试耽误了，赶到面试地点已将近 6:30 了，一轮面试刚开始就被面试官反问试卷及格了没，我说及格了，结果没答对，然后就一直处于她问我答的状态，而她问得很广也很细，面试大概一共持续了将近 1 小时 10 分钟，尽管如此，我最终还是被淘汰了。

2. 不要一厢情愿做公司的“备胎”

在和公司签署三方协议之前，求职者千万不要“在一棵树上吊死”，每个 HR 都会说自己的公司有多好多好，其实也不尽然。所以，求职者最好能接触一下这家公司的员工，大体知道个概况，不要一味地相信口头 offer。

口头 offer 本身没有法律效力。求职者要尽量多找几家公司，拿到多一点 offer，这样比较保险，同时在与用人单位谈工资时，也会比较有底气。所以，不要一厢情愿地做公司的“备胎”，要让自己多几个选择，以免处于被动局面。

3. 选择、权衡

如果就业有总部与分部之分，最好选择去企业总部，因为企业总部和分部差别很大。在企业总部，大部分的资源都会汇集在那儿，机会也相对较多，职员能够很快得到支持和帮助，学习的机会也更大，个人发展空间也更大。而在分部就会有很多局限性，很多公司都不会把核心业务放在分部，顶多就是设置一个办事处，晋升机会一般也少，接触核心的东西也少，对个人的成长空间也相对较小。

4. 论“持久战”

找工作是个艰苦的拉锯战、体力战、消耗战。有时候拿到了一个 offer，求职者还希望有更好的 offer；有时候拿到了多个 offer，求职者还需要考虑一段时间，不断地比较 offer 或者等待更好的 offer 的出现。所以一定要做好“持久战”的准备，并非人人都是“千里马”，也并非每个面试官都是“伯乐”。在短时间内得到面试官的认可也不是一件容易的事情，所以就算被这些“伯乐”拒绝也是再正常不过的事情，“此处不留人自有留人处”，求职者只要做好“持久战”的心理准备，并具备相应的专业知识，必定会找到适合自己的工作。

5. 实习是捷径

好公司人人都想去，可是好公司招聘的人数有限，并非人人都能进好公司，必然有很大一部分人最终与好公司擦肩而过。能进入好公司除了运气，更多的还是依靠实力。对于实力有点欠缺的求职者来说，千万要抓住该公司的实习机会，能去一定要去，以便为自己增加留下来的机会。而且和你一起竞争去实习的同学的数量与实力远远多于校招时，校招的人数很有可能会锐减。不仅如此，不错的实习经历也会给求职者的简历增加分量，在应聘其他企业时也会大有益处。

6. 做研究还是做项目

若不考虑读博士，同学们在读研期间就不要把精力只放在做研究上面，可以多做点工程性的项目。当然不可否认，研究生阶段做科研锻炼了我的思维能力，但是公司青睐的大多数还是工程技术性人才。若能将这两者较为完美地结合起来，无疑是如虎添翼的。

2.8 小结

成功不可复制，所以切忌盲目照搬别人的成功，因为每个人都是唯一的，都是不一样的（性格、环境、能力、智商、情商、机遇、身份都不一样）。但是“他山之石，可以攻玉”，成功的方法、失败的教训却可以借鉴，通过借鉴，求职者从中认识自我、创造自我、成就自我，最终一样能够站在前人的肩膀上，用自己勤劳的双手、聪明的头脑取得成功，开创自己的美好明天。

第3章

企业面试笔试攻略

面对无数 IT 企业，到底是应该“广撒网”还是应该集中精力重点突击某一个或是某几个，这一直是困扰应届毕业生的问题。其实，即便是有工作经验的人，也会为此问题烦恼。对于这个问题，“仁者见仁，智者见智”，但无论选择哪种方法，在找工作时，求职者都需要了解自己应聘企业的相关招聘信息，找准“攻击点”，才能事半功倍，取得意想不到的效果。

本章以当前主流 IT 企业为对象，如互联网企业、网络设备提供商（包括电信运营商以及银行等）、创业型企业等，对其面试笔试进行一对一的强力分析，包括招聘流程、面试笔试内容，面试笔试真题、面试需要注意的事项以及推荐知识点学习等，拨开这些企业面试笔试的神秘面纱，将其最直观的一面展现给求职者，以帮助求职者顺利找到适合自己的工作。

3.1 互联网企业

互联网正以人类无法想象的速度向前发展，正如十年前人们很难想到互联网会对我们的生活产生如此深远的影响一样，我们也很难想象未来十年互联网会是什么样子。但毋庸置疑的是，未来互联网的高速发展仍然不会停止，一系列新的技术，例如云计算、物联网、移动互联网等，将会继续蓬勃发展，进而让人们的生活产生巨大的变革，促进人类社会的飞速发展与进步。

伴随着互联网的发展，一大批优秀的互联网企业应运而生，有做门户网站的，有做搜索的，有做网络安全的，有做网络游戏的，有做电子商务的，有做即时通信的，门类繁多。互联网的发展成就了一些行业巨头，而反过来，它们的存在也极大地推动了整个互联网产业的发展。

互联网行业作为当前的高薪行业，动辄十几万，甚至几十万的年薪，吸引了无数青年才俊、IT 英才。而要想敲开这些名企的大门，也并非一件易事，求职者需要做很多准备，否则最终的结果只能是“落花有意随流水，流水无心恋落花”。

1. 招聘流程

随着全球经济的回暖，互联网企业的招聘规模也日趋扩大，很多互联网企业也由以前的零散招聘，变为现在的大肆招兵买马，动辄招聘上千人。所以，作为求职者，挑战虽然存在，但机会依然很多。

互联网企业的招聘一般从每年的 9 月份开始，一直持续到 11 月份，他们会选择国内一些计算机专业实力比较强的大学作为招聘点，如清华大学、中国科学技术大学、上海交通大学、

东南大学、浙江大学、华南理工大学、西安电子科技大学、武汉大学、西安交通大学、哈尔滨工业大学等名牌高校。

互联网企业的招聘流程一般也比较严格，主要包括以下几个步骤：网上注册简历→宣讲会→筛选简历→笔试→专业面试一→专业面试二→HR（人力资源）面试→综合面试→最终录用。需要注意的是，由于企业每年的招聘信息都可能会有变动，因此求职者应该更多地关注企业的招聘流程，做到实时了解。

2. 面试笔试注意事项

互联网是一个发展迅速的行业，所以在求职过程中，求职者应保持一颗平常心，相信自己，同时自己平时要多积累，多看与自己专业、职业相关的东西，比如浏览一些比较专业的技术网站，扩展自己的知识面，从而开阔自己的视野。

对于互联网企业的面试而言，首先求职者要好好准备面试。因为互联网企业一般都比较“年轻”，他们比较注重对求职者归属感的培养，所以在求职之前，求职者需要了解该企业的企业文化，了解自己应聘的职位，只有知己知彼，才能百战百胜。同时，还应分析各大企业历年的笔试面试题。往往能够发现很多一模一样的题，而且很多问题都是反复被问及，所以一定要对一些经常被问到的问题事先做好相关的准备。例如，个人优缺点、个人兴趣爱好、如何自我介绍等，做到有备无患。对于简历的内容一定要做到严谨、仔细、认真，面试官通常会针对简历或材料提出问题，所以简历最好突出重点，以吸引面试官的注意力，进而争取到比较大的发挥空间。同时，自己需要事先准备好几个最后提问的问题，一般面试官在最后会问求职者对公司有没有其他问题需要进一步了解的，最好可以问上一两个，一方面可以对企业了解深入一些，另一方面也可以表现求职者的积极态度。

其次，不要不懂装懂，尤其是互联网企业的招聘。技术型面试中，面试官个个都是身经百战的老手，他们也是从求职者过来的，对求职者的心态了如指掌，所以在他们面前，不会就是不会，不要抱着侥幸的心理以为可以蒙混过关。其实企业对应届毕业生在技术上的要求不会太高，掌握好基础知识就行了，弄虚作假的人是得不到企业青睐的。不该说的话绝对不要多说，尤其是人力资源类的面试，多说一句不合适的话就很可能搞砸了整个面试。

最后，就是调整好心态，充满信心，保持淡定。看着那么多人行色匆匆以及面试前的那种压抑的气氛，求职者很容易紧张，其实大可不必，应聘的目的不是为了求职者出丑，而是为了最大限度地发现人才。面试中，每位求职者被问到的问题区别很大，技术类面试一般针对简历或者其他面试材料来问，除了技术问题，也涉及一些工作能力的考查，比如效率观念等。人力资源类的面试则会问到学习成绩、性格、沟通能力等问题，但问题数量不算太多。

除了需要注意常见的面试笔试技巧与细节外，求职者还要针对互联网企业招聘的特点进行一些必要的准备，以避免一些不应该的错误，主要有以下一些方面的内容需要注意：

1) 互联网企业一般对求职者的在校成绩没有硬性要求，但是会把成绩当作一个重要的衡量标准，所以成绩好是一个很大的优势。对于专业技术一流，但成绩不够理想的求职者来说，应在简历中突显自己的技术优势，这样可以为自己增加“筹码”，从而减小“不战而败”的可能。

2) 由于企业实际业务需求以及岗位本身的发展机遇，企业可能需要对求职者的工作地点做出相应的安排，求职者需要在面试中与面试官进行及时的沟通与协调，因为有些业务的实际工作地点可能与招聘宣讲的不一样，因此一定要注意工作地点的问题。

3) 一段知名企业的实习经历,不仅可以为自己的简历锦上添花,还可以为求职者找工作增加“筹码”,尤其是当你要进入某一个互联网企业时,通过在企业实习实现留在企业工作还是比较容易的,例如,某些互联网企业会在每年四五月份进行实习生招聘,提前在应届毕业生中发掘并笼络人才,所以对于希望进入该企业的应届毕业生而言,这不失为一种捷径。而且即使未能通过它的实习生甄选,求职者仍然可以继续申请校园招聘,一般也不会受到什么影响。

4) 在校期间有机会多参与互联网企业组织的各种活动,很多互联网企业都会提供一些科技竞赛平台,发掘科技人才,如一些互联网企业组织的创新设计大赛、程序设计大赛等。除此之外,很多互联网企业会在一些高校设置俱乐部。一般而言,创新设计大赛获奖者以及企业俱乐部的主要负责人都有进入该互联网企业的“绿色通道”,相比其他求职者机会更多。

5) 从事研发的程序员一般都比较随意,除非是销售或是其他特殊场合(如银行、外企等),在面试过程中,一般都不用穿正装,只需衣冠整齐、干净得体即可。

6) 在对参与过的项目进行介绍时,求职者不能一味地按照事前准备好的模板照本宣科,而应该根据所申请的工作的性质,多说一些与自己申请的工作内容相近的东西。例如如果是搜索类企业,求职者就可以多提及一些与搜索有关的项目;如果是安全类企业,求职者就可以多提及一些有关网络安全的项目。

7) 有些在北京、上海设置工作岗位的互联网企业,很难为求职者办理当地户口。由于每个公司得到的指标数量都是由政府调控的,而且对户口指标控制得越来越严格,很多互联网企业在这一问题上不能给予绝对承诺。求职者应在面试过程中向企业了解清楚这一问题。

8) 很多互联网企业为求职者提供的 offer 并非都完全一样,所以一定要区分顶级 offer 与普通 offer。顶级 offer 是企业给予面试笔试非常优秀者的绿色通道。一般而言,拿到顶级 offer 的求职者各方面的待遇都较普通 offer 的要好,如薪资、户口、发展空间等。所以求职者一定要擦亮自己的眼睛,若有可能拿到企业的顶级 offer 或是有资格与企业谈条件时,一定不要放过机会。

9) 很多互联网企业,实行内推制,即通过内部员工推荐校友、师弟师妹、朋友等来此工作,如果被推荐的人最终被该企业录取了,推荐者也有机会获得该企业提供的奖励,这对推荐者与被推荐者来说都是一个莫大的荣耀,所以如果有机会,求职者一定要通过各种渠道“求内推”。

10) 互联网企业一般不鼓励违约,但是也不反对违约,所以如果提出违约,一般需要支付违约金。

11) 互联网企业的面试看起来有点随意,其实给每个人的机会都是均等的,它会给予求职者足够的机会来证明自己的能力。无论是名牌高校的毕业生还是普通高校的毕业生,无论是本科生还是研究生,只要你足够优秀,所面对的机会必定是均等的。

12) “不要被同一根绳子绊倒两次”。因为面试有时候可能有跨度,每一轮面试的面试官都不一样,但同一个问题可能会被不同的面试官提问。所以有些问题在面试的时候没回答好,面试结束后,求职者一定要仔细思考,以免在下次或是下一个公司面试中遇到同样的情况。最好能够将自己的面试内容做好记录,面试结束回去后好好想想。

13) 学会纸上写程序。大多求职者学习编写程序时,一般都是在计算机上输入代码,不习惯在纸上写程序,但是在进行技术面试时,一般都需要在纸上写代码,在纸上写代码一般都容

易出错，思路也比较紊乱，所以最好在平时就多加练习。

14) 建议准备一个日程本，记录每一次宣讲会、笔试和面试的时间，这样一旦企业打电话来预约面试，可以马上查找日程本上的空闲时间，以免发生时间上的冲突。每投一份简历，求职者应记录下企业的职位和要求，如果一段时间以后（1个月或更长）有面试机会，可以翻出来看看，有所准备，以免出现面试笔试张冠李戴的情况。

15) 互联网企业的行业特性导致在互联网企业工作的强度、压力都比较大，工作也比较辛苦。高薪意味着高付出，但高付出也意味着高回报，高薪不是“叫”出来的，是踏踏实实干出来的。

16) 大型互联网企业的用户群广泛，他们对海量数据处理很感兴趣，尤其是面试笔试过程中的压轴大题都是海量数据处理，所以求职者在应聘前一定要研究海量数据的处理问题，做到有备无患。

3. 真题分析

以下摘选一些著名互联网企业的部分面试笔试真题以及考查知识点，供读者参考。

- 1) static 的作用。
- 2) final 的作用。
- 3) overload 与 override 的区别。
- 4) 组合与继承的区别。
- 5) clone 的作用。
- 6) 前置 ++ 与后置 ++。
- 7) 内部类。
- 8) 二维数组的表示。
- 9) 接口与抽象类的对比。
- 10) 反射机制。
- 11) 函数调用方式。
- 12) 重载函数。
- 13) 构造函数。
- 14) 合并两个有序链表。
- 15) 逻辑推理——智力题。
- 16) 从 100 亿条记录的文本文件中取出重复数最多的前 10 条。
- 17) 判断单列表是否又环。
- 18) 二叉树的多种遍历算法实现。
- 19) 有读和写两个线程和一个队列，读线程从队列中读数据，写线程往队列中写数据。
- 20) stack 和 heap。
- 21) TCP 的流量控制和拥塞控制机制。
- 22) 写一个函数，返回一个字符串中只出现一次的第一个字符。
- 23) 求一个数组中第 k 大的数的位置。
- 24) 面向对象继承、多态问题，例如多态的实现机制。
- 25) 值传递与引用传递。
- 26) 什么是不变量？
- 27) == 与 equal 的区别。
- 28) 创建空类时，哪些成员函数是系统默认的？

- 29) 有 10 万个 IP 段, 这些 IP 段之间都不重合, 随便给定一个 IP, 求出属于哪个 IP 段。
- 30) 网络编程 (网络编程范式, 非阻塞 connect)。
- 31) TCP/IP。
- 32) Linux 的命令、原理以及底层实现。
- 33) Linux 编程, 包括所有互斥的方法、多线程编程, 进程间通信。
- 34) 一个一维数轴上有不同的线段, 求重复最长的两个线段, 例如, a: 1 ~ 3, b: 2 ~ 7, c: 2 ~ 8, 最长重复是 b 和 c。
- 35) Java 入口函数的特点。
- 36) 内存溢出与内存泄露有什么区别?
- 37) 利用互斥量和条件变量设计一个消息队列, 具有以下功能: ①创建消息队列 (消息中所含的元素); ②消息队列中插入消息; ③取出一个消息 (阻塞方式); ④取出第一消息 (非阻塞方式)。注意: 互斥量、条件变量和队列由系统给定。
- 38) 用非递归方法完成二叉树的遍历。
- 39) 如何实现类似函数指针的功能。
- 40) 设计模式。
- 41) 排列组合问题。
- 42) 若有序表的关键字序列为 (b, c, d, e, f, g, q, r, s, t), 则在二分查找关键字 b 的过程中, 先后进行比较的关键字依次是什么?
- 43) 有一个虚拟存储系统, 若进程在内存中占 3 页 (开始时内存为空), 若采用先进出 (FIFO) 页面淘汰算法, 当执行如下访问序列后, 1, 2, 3, 4, 5, 1, 2, 5, 1, 2, 3, 4, 5, 会发生多少缺页?
- 44) 有一个顺序栈 S, 元素 s1, s2, s3, s4, s5, s6 依次进栈, 若 6 个元素的出栈顺序为 s2, s3, s4, s6, s5, s1, 则顺序栈的容量至少应该有多少?
- 45) [0, 2, 1, 4, 3, 9, 5, 8, 6, 7] 是以数组形式存储的最小堆, 删除堆顶元素 0 后的结果是多少?
- 46) 某页式存储管理系统中, 地址寄存器长度为 24 位, 其中号占 14 位, 则主存的分块大小是多少字节?
- 47) 内存泄露。
- 48) 各种排序算法使用与比较。
- 49) 默认初始化问题。
- 50) 字符串的存储方式。
- 51) 面向对象与面向过程编程的区别?
- 52) 异常处理。
- 53) 垃圾回收器。
- 54) 多线程同步。
- 55) 数据库内连接与外连接的区别。
- 56) 设计模式。

互联网企业的面试中, 除了一些常见的技术面试问题外, 还有一些与项目、性格有关的问题, 例如:

- 1) 自我介绍。
- 2) 项目相关问题。

- 3) 了解我们企业吗?
- 4) 家乡是哪里的? 为什么要来这个城市工作?
- 5) 为什么会选择我们企业?
- 6) 为什么选择这个职位?
- 7) 竞赛获奖以及论文。
- 8) 自己的职业规划是什么?
- 9) 谈谈自己的优势与劣势。
- 10) 你是怎么在团队合作中发挥作用的?
- 11) 结合简历中的实习经历问一些细节。
- 12) 对我们企业的理解。喜欢我们吗?
- 13) 讲讲个人优缺点。
- 14) 个人对薪资问题。
- 15) 可以实习吗?
- 16) 你的同学为什么不选择我们企业?
- 17) 如果你没有被录用, 你觉得可能是什么问题?
- 18) 你有什么问题吗?

4. 推荐知识点学习

通过真题发现, 知名的互联网企业一般知识面考查得比较广, 从基本的语言知识, 到面向对象技术, 从排序到二叉树, 从逻辑推理到海量数据处理, 从英语题到智力题, 都有涉及, 所以最好的准备是从平时积累开始, 拓宽自己的知识面。

同时由于互联网企业侧重点往往不同, 针对这一特性, 各企业面试的重点也不尽相同。例如, 以搜索为核心的互联网企业更加侧重于算法、操作系统、数据库等相关知识; 电子商务企业则除了基础知识以外, 还需要一些 Java 方面的知识; 网络安全企业则侧重于有关软件安全、网络安全的专业知识。因此, 求职者应针对所应聘企业的特点做相应的准备。

3.2 网络设备提供商

互联网的巨大发展, 网络设备功不可没, 网络设备已经成为互联网发展的基石。随着 IT 业的发展, 很多网络设备提供商已经不再将目光只是锁定在这一块“蛋糕”上, 而是纷纷将“触角”伸展开来, 以至于其业务范围也变得越来越广泛; 例如云计算、智能手机、物联网, 正因如此, 他们对人才的渴望日益迫切, 招聘规模也有所扩大, 待遇自然也就较好。

1. 招聘流程

由于该类企业招聘规模一般比较大, 因此其校园招聘开始得也比较早, 有的从每年的七八月份就开始了, 有的甚至在四五月份就开始了校园宣讲。该类企业的招聘流程一般为: 注册简历→笔试→技术面试→上机测试或性格测试或群面→主管面试。校园宣讲会的举行地点一般会设在北京航空航天大学、西安电子科技大学、南开大学、武汉大学、湖南大学、北京邮电大学等高校。

2. 面试笔试注意事项

在整个应聘过程中, 面试是最具有决定性意义的一个环节, 事关求职成败。同时, 面试也是求职者全面展示自身素质、能力、品质的最好时机。若面试发挥出色, 可以弥补先前笔试或

是其他条件（如学历、专业）上的一些不足。除了常见的面试注意事项外，在该类企业的面试笔试过程中，求职者还应该注意以下几个方面的问题：

1) 该类企业的招聘主要以考查综合素质和技术能力为主，综合素质主要考查以下方面内容：责任心、沟通能力、团队精神、主动性、学习新知识的能力、意愿等。企业通过招聘主要考查人员以下3个方面：①言谈举止、仪容、仪表；②心态（心理状态）；③专业知识。

2) 面试要低调，待人诚恳，可以答不上题，但是一定要让面试官觉得你这个人踏实可靠。

3) 第一轮面试一般是技术面，只要态度够谦虚，又参与过实际的项目研发，一般都得到二面机会，特别是需求量比较大的岗位，诸如软件开发、云计算等。遇到会回答的问题应保持淡定；遇到不会回答的问题，也要保持淡定。该类企业通过技术面淘汰人并不多。该类企业的面试问题都是从求职者的简历出发，一点一点地问，问题会一个比一个深入，直到求职者回答不上或者面试官满意为止。

4) 由于该类企业的规模比较庞大，在全国很多大城市都设有研发基地，可能会根据岗位需要对求职者进行岗位调整，有时会进行异地研发，所以求职者一定要做好前往异地工作的心理准备。如果无法接受，一定要将自己的意愿表达清楚。

5) 该类企业一般都有性格测试这个环节，性格测试反映求职者是否能够适应岗位要求，性格测试是淘汰人的一个重要环节。一般而言，在进行性格测试时，求职者最好能够保证答题的一致性。同时，如果该类企业取消性格测试，很有可能会组织群面，群面也是该类企业淘汰人的一个重要环节，所以求职者应提前准备有关群面的技巧。

6) 该类企业在近年开始增加了针对研发类岗位的上机测试，用以考查求职者的实际编程能力。机试题目一般都非常简单（当然也不排除有一些非常难的题目），都是最常见的编程题，不涉及高深的算法，求职者可以选择 C/C++ 或 Java 语言编写源代码，所以在机试前认真仔细地自己的计算机上多练习编写代码，否则，在紧张的气氛里，很难发挥出自己的真实水平。

7) 该类企业一般每年都会在五六月组织一些在校学生参加的程序设计竞赛，获奖的学生除了获得奖品与证书外，一般还能享受到招聘“绿色通道”的优待。所以，建议在在校生如果有机会，在求职前多参加一下这些科技竞赛，对于个人水平的提高大有益处。

8) 该类企业的面试一般很集中：技术面、群面、机试、性格测试、主管面试，几乎安排在一两天时间内完成，对个人的体力与精力是一个极大的考验。

9) 该类企业的待遇一般比较好，虽然可能稍逊于互联网企业，但是该类企业对工作年限较长、业绩比较突出的优秀员工可能会提供股票、过渡房，所以，总体福利是不错的。

10) 该类企业面试有时会有英语口语测试。对于求职者而言，能说尽量开口说。一定要明白一个道理，那就是说得不好是能力问题，不说就是态度问题了。

11) 很多企业在与求职者签订协议时，都明确要求不允许求职者未来的多少年内跳槽到同类型的竞争企业中去。这一点应引起面试者的注意。

3. 真题分析

某知名公司笔试题。

1) 下列运算符中，优先级别最高的是（ ）。

- A. & B. && C. != D. ?:

2) 若用数组 S [0, ..., n] 作为两个栈 S1 和 S2 的存储结构，对任何一个栈只有当 S 全满时才不能做人栈操作。为这两个栈分配空间的最佳方案是（ ）。

- A. S1 的栈底位置为 0, S2 的栈底位置为 n + 1

- B. S1 的栈底位置为 0, S2 的栈底位置为 $n/2$
 C. S1 的栈底位置为 1, S2 的栈底位置为 $n/2$
 3) 经过强制类型转换以后, 变量 a, b 分别为 ()。

```
Short a = 128; byte b = (byte)a;
```

- A. 128 127 B. 128 -128 C. 128 128 D. 编译错误
 4) Java 的 Daemon 线程, setDaemon 设置必须要 ()。
 A. 在 Start 之前 B. 在 Start 之后 C. 前后都可以
 5) 下列不属于 Java 语言鲁棒性特点的是 ()。
 A. Java 能程序在编译和运行时的错误
 B. Java 能运行虚拟机实现跨平台
 C. Java 自己操纵内存减少了内存出错的可能性
 D. Java 还实现了真数组, 避免了覆盖数据类型的可能
 6) 有以下几个对象:

```
public class DataObject implements Serializable{
    private static int i = 0;
    private String word = "";
    public void setWord(String word) {
        this.word = word;
    }
    public void setI(int i) {
        DataObject.i = i;
    }
}
```

创建一个如下方式的语句:

```
DataObject dataObject = new DataObject();
dataObject.setWord("123"); dataObject.setI(2);
```

将此对象序列化为文件, 并在另外一个 JVM 中读取文件, 进行反序列化, 请问此时读出的 DataObject 对象中的 word 和 i 的值分别为 ()。

- A. "", 0 B. "", 2 C. "123", 2 D. "123", 0
 7) 基于 Servlet API, 如何实现转向时不在地址栏中显示转向后的地址? ()。
 A. redirect() B. sendRedirect() C. forward() D. transform()
 8) 假设有以下代码

```
String s = "hello";
String t = "hello";
Char c[] = {'h','e','l','l','o'};
```

下列选项中返回 false 的语句是 ()。

- A. s.equals(t); B. t.equals(c);
 C. s==t; D. t.equals(new String("hello"));
 9) 下面代码的运行结果是 ()。

```
class B extends Object{
```



```

static{
    System.out.println(" Load B");
}
public B(){
    System.out.println(" Create B");
}
}
class A extends B{
    static{
        System.out.println(" Load A");
    }
    public A(){
        System.out.println(" Create A");
    }
}
public class Testclass{
    public static void main(String[] args) {
        new A();
    }
}

```

- A. Load B Create B Load A Create A
 B. Load B Load A Create B Create A
 C. Load B Create B Create A Load A
 D. Create B Create A Load B Load A

10) 为了让浏览器以 UTF-8 编码显示 JSP 页面, 请问下列 JSP 代码正确的是 ()。

- A. <% page contentType =
 B. <meta http-equiv =
 C. 把所有输出内容重新编码:new String (content.getBytes())
 D. response.setContentType()

11) 下列数组定义及赋值中, 错误的是 ()。

- A. int intArray[];
 B. intArray = new int[3];intArray[1] = 1; intArray[2] = 2; intArray[3] = 3;
 C. int a[] = {1,2,3,4,5};
 D. int[][] a = new int[2][];a[0] = new int[3];a[1] = new int[3];

12) 关于守护线程的说法, 正确的是 ()。

- A. 所有非守护线程终止, 即使存在守护线程, 进程运行终止
 B. 所有守护线程终止, 即使存在非守护线程, 进程运行终止
 C. 只要有守护线程或者非守护进程其中之一存在, 进程就不会终止
 D. 只要所有的守护线程和非守护线程终止运行之后, 进程才会终止

13) 在 Java 语言中, 下列关于字符集编码 (Character set Encoding) 和国际化 (i18n) 的问题, 哪些是正确的? ()。

- A. 每个中文字符占用 2 B, 每个英文字符占用 1 B
 B. 假设数据库中的字符是以 GBK 编码的, 那么现实数据库数据的网页也必须是 GBK 编码的

C. Java 的 char 类型, 以 UTF - 16 Big Endian 的方式保存一个字符

D. 实现国际化应用常用的手段是利用 ResourceBundle 类

14) JNDI 可用于如下哪些应用场景? ()。

A. 配置信息存储 B. 异步信息发送 C. 数据库连接池查找 D. 远程对象查找

15) 当想创建一个具体的对象而又不希望指定具体的类时, 我们可以使用 () 模式。

A. Factory B. Adapter C. Command D. Singleton

16) 下面哪个不是标准的 Statement 类? ()。

A. Statement B. PreparedStatement C. CallableStatement D. BatchedStatement

17) 关于 Spring 的 PROPAGATION_REQUIRES_NEW 事务, 下面哪些说法是正确的? ()。

A. 内部事务回滚会导致外部事务回滚

B. 内部事务回滚了, 外部事务仍然可以提交

C. 外部事务回滚了, 内部事务也跟着回滚

D. 外部事务回滚了, 内部事务仍然可以提交

18) 利用 Thread. wait() 同步线程, 可以设置超时时间吗? ()。

A. 可以 B. 不可以

19) 若线性表最常用的操作是存取第 i 个元素及其前趋的值, 则采用 () 存储方式节省时间。

A. 单链表 B. 双链表 C. 单循环链表 D. 顺序表

20) 线程调用了 sleep() 方法后, 该线程将进入 () 状态。

A. 可运行状态 B. 运行状态 C. 阻塞状态 D. 终止状态

21) JDBC 的主要功能包括 ()。

A. 创建与数据库的连接 B. 发送 SQL 语句到数据库中

C. 处理数据并查询结果 D. 以上都是

22) Springmvc 的中心控制 Servlet 是哪个类? ()。

A. ActionServlet B. Dispatcherservlet C. AbstractController D. FacesServlet

23) 若下列所用变量均已经正确定义, 则其中不合法的是 ()。

A. x >> 3 B. +++j C. a = x > y? x: y D. x% = 4

24) 下面有关 forward 核 redirect 的描述, 正确的是 ()。

A. forward 是服务器将控制权转交给另外一个内部服务器对象, 由新的对象来全权负责响应用户的请求

B. 执行 forward 时, 浏览器不知道服务器发送的内容是从何处来, 浏览器地址栏中还是原来的地址

C. 执行 redirect 时, 服务器端告诉浏览器重新去请求地址

D. forward 是内部重定向, redirect 是外部重定向

E. redirect 默认将产生 301 Permanently moved 的 HTTP 响应

25) 下列说法中, 正确的是 ()。

A. Java 中包的主要作用是实现跨平台功能

B. package 语句只能放在 import 语句后面

C. 包 (package) 由一组类 (class) 和接口 (interface) 组成

D. 可以用 #include 关键字来表明来自其他包中的类

面试中部分非技术问题。

- 1) 自我介绍。
- 2) 家乡是哪里的? 为什么选择留在这个城市?
- 3) 是否喜爱运动? 喜爱什么运动项目?
- 4) 性格内向还是外向?
- 5) 用英语进行简短的自我介绍。
- 6) 对于工作地点有什么要求? 是否能够服从公司的分配?
- 7) 项目有关。
- 8) 自己的优缺点。
- 9) 为什么要离职?
- 10) 说说你的个人发展计划。
- 11) 对软件外包的认识。
- 12) 对经常加班的态度。
- 13) 对长期出差的认识。
- 14) 对工作责任心、沟通能力、团队精神、主动性的认识。
- 15) 群面。
- 16) 性格测试。

面试中部分技术问题。

- 1) struct 与 class 的区别。
- 2) error 与 exception 的区别。
- 3) 什么是 SSH?
- 4) 什么是 Ioc 和 AOP?
- 5) 多线程的实现方法以及同步。
- 6) 将字符串右移 N 位。
- 7) 序列化。
- 8) ArrayList、Vector、LinkedList 的区别。
- 9) final 的作用。
- 10) 简述 Java 如何读写文件。
- 11) 堆与栈的区别。
- 12) JVM 的工作原理。
- 13) Java 与 C++ 的区别。
- 14) 内存泄露。
- 15) NIO 的优点。
- 16) 静态变量与非静态变量的区别。
- 17) 对设计模式的了解。
- 18) 链表的后续遍历实现。
- 19) 有序单项链表的插入函数。
- 20) 根据简历上的项目提问。
- 21) 实时操作系统与非实时操作系统的区别。

某企业部分机试题。

- 1) 求一个数组里面能被 3 整除的个数。给了题目框架, 但框架不允许修改。

2) 计算一个数组中的奇数值和偶数之和。

3) 手机号码合法性判断,我国内地运营商的手机号码标准格式为“国家码+手机号码”,例如8613888888888,特点为:长度为13位,以86的国家码打头,手机号码的每一位都是数字。请实现手机号码合法性判断的函数要求:若手机号码合法,则返回0;若手机号码长度不合法,则返回1;若手机号码中包含非数字的字符,则返回2;若手机号码不是以86打头的,返回3。

4) 计算两个字符串中匹配项的字符串,并将匹配的字符串存储在c[]中。要求:字符串*可以匹配任意一个字符串,直到下一个匹配字母为止,其中字符串2中允许有*;输出相匹配的字符串;只要一个字符不匹配,匹配过程就结束,例如,字符串1为abcdefg,字符串2为a*f,输出为abcdef。

5) 从两个数组的最后一个元素比较两个数组中不同元素的个数,如有array1[5]={77,21,1,3,5},array2[3]={1,3,5},从array1[4]与array2[2]比较开始,到array1[2]与array[0]比较结束。这样得出它们不同的元素个数为0,若array1[6]={77,21,1,3,5,7},那么它们不同的元素为3。函数原型为int compare_array(int len1,int array1[],int len2,int array2[]);其中,len1与len2分别为数组array1[]和array2[]的长度,函数返回值为两个数组不同元素的个数。

6) 实现约瑟夫环。

7) 有字符串表示的一个四则运算表达式,要求计算出该表达式的正确数值。四则运算即:加、减、乘、除(+、-、*、/),另外该表达式中的数字只能是1位(数值范围为0~9)。另若有不能整除的情况,按向下取整处理,如8/3,得出值为2。若有字符串“8+7*2-9/3”,计算出其值为19。主要考点:①字的字符形式变换为数字形式的方法;②数字的数字形式变换为数字的字符串形式的方法。

4. 推荐知识点学习

该类企业笔试涉及的知识面比较广、比较细,计算机系统、数据结构、面向对象编程、C/C++、软件工程、操作系统、数据库系统、计算机网络、无线通信无一不涉及,重点是C/C++、数据结构与算法,而且对简历上的内容问得比较细。该类企业的招聘有时会包括群面与性格测试,而且一般都是通过这两个步骤淘汰求职者,所以求职者应该在招聘前加强这两个方面知识的训练。同时该类企业的面试会有少量的英语口语交流,英语基础薄弱的求职者最好能够做一些必要的准备工作。

3.3 外企

改革开放以来,中国向世界敞开了胸怀,无数外资企业抓住机会来到中国落地生根、开枝散叶,他们在带来先进技术的同时,也带来了完善的管理模式,自然也受到了国人的青睐。相比其他类型的企业,外企薪酬待遇优厚,出国旅游、社会保险、年假、失业保险和住房公积金都比较齐全,而且外企在管理上一般都有一套完善的规范,不存在本土企业自身的局限性。在这种模式下,员工的工作能力往往能够得到快速提高,所以进入外企工作成了很多人的梦想。

1. 招聘流程

外企的招聘流程通常为:网申(网络在线申请)→笔试→技术面试一→技术面试二→直属部门经理面试→更高级别经理面试→HR面试。

外企对人才的考核非常认真仔细,因为他们不愿意随意招聘到一个不适合的人,然后还得

花大气力来培养，而愿意在人才的发掘上花大力气，大投入也在所不惜，所以外企的招聘流程非常复杂烦琐。在笔试题目上，他们出的题目很有技术含量，能相对客观地反映出求职者的专业技能、英语水平、智力、表达能力等；在面试这个问题上，他们做得也同样很优秀。外企的面试少则两三轮，多则五六轮、七八轮，不仅考查求职者的专业技能，还会通过各种面试官的面试，来考查求职者的综合素养，所以整个求职过程所需要的时间短则半个月到1个月，长则3个月，有时甚至半年。

不同外企的校招时间各不相同，主要是根据企业自身的情况来定，所以在招聘季来临时，求职者应提前做好各方面的准备。

2. 面试笔试注意事项

外企的招聘过程不同于其他类型的企业，所以求职者在进行面试笔试时需要给予“特别照顾”。一般需要注意以下事项：

1) 注意仪表。在外企面试时一定要穿比较正式的职业装，男生应穿西装，女生应穿套装，但不一定非名牌不可。同时，女生最好不要携带一些晶光闪闪、叮叮当当的饰物，也不要化浓妆或穿太时髦、太暴露的服饰，最好化淡妆，发型简单整洁即可。

2) 注意礼仪。不要嚼口香糖或抽烟。喝水最忌讳的有两点：一是喝水出声；二是把水弄洒。求职者一定要把水杯放远一点，喝不喝都没关系；记住打喷嚏之前或之后一定要对面试官说“Excuse me”；当面试结束时，求职者不要忘记向面试官表达希望被录用的强烈愿望，在握手告辞之前，也可以问一句招聘的下一步内容是什么。

3) 切忌谈论政治。在外企招聘中，一般不要涉及与政治有关的内容，即使谈到也不要带有太浓的主观色彩，以客观的态度回答即可。

4) 在外企的初次面试中，除非能确认面试官对你很感兴趣，否则不应该询问有关薪水、假期、奖金、退休等问题。当然，如果面试官询问你希望的薪水时，应表示你对工作机会的兴趣要大于对具体的薪水，此时可以说明你的期望薪水。

5) 谈吐要清晰，尽量少用语气词。语气词或口头禅太多会让面试官误以为求职者自信心和准备不足，从而影响求职结果。

6) 不要过多解释或道歉。如果面试迟到了，一句抱歉就行了，或者补充一下真实的原因。不要以为编故事可以忽悠面试官，其实面试官都很精明，求职者切勿因小失大。

7) 不要当面询问面试结果。一般在面试结束后，客气地对面试官说声谢谢就行了。有些求职者可能为了体现上进心，在面试结束时，会向面试官套近乎，询问面试官对自己感觉怎样，有什么需要改进的地方，这完全没有必要，因为当天不可能知道结果，问了还有可能适得其反，如果真的想知道结果，可以在面试后3~5天后，通过电话或邮件的方式询问。

8) 不要谈论薪资。一般而言，外企是不会在招聘会上说出具体薪水的，因为在这一面试阶段还没有到谈论薪资细节的地步。而且，外企也不太喜欢完全冲着工资去的人。求职者最好不要主动提问薪资问题。如果想知道薪资，可以通过已毕业的师兄师姐了解他们所在行业的大致工资范围。

9) 在面试过程中，不要请求面试官帮忙。即便面试官是自己的校友或者朋友，也不要套近乎说“多谢您帮忙了”，这是很不妥当的做法，不仅起不到正面的效果，还可能适得其反。

10) 在回答问题时，一定要给出明确态度，不要模棱两可。例如，当面试官询问求职者性格是外向还是内向时，有些求职者可能会回答，“和朋友在一起时我比较外向，而在家时我比较内向。”这种回答，表面上看，两种性格都沾边，其实就等于没有回答。所以，在回答面试官问题的时候，一定要选择一个明确的方向，并给出理由作为支持。

11) 外企的笔试面试题一般会有一些开放性问题,如“为什么你要选择计算机专业”,大部分题目都没有固定答案,主要是考查求职者的生活经历和工作态度等,看是否和企业文化相契合,只要表达流畅就可以了。

12) 外企需要的人才应该胆大、心细、“脸皮厚”。胆大——不害怕;心细——认真、不拖沓;“脸皮厚”——技术功底不是最主要的,而“执着精神”是最关键的。

13) 英文很重要。外企笔试题都是英文,而且会涉及与企业内其他国家的工程师的面试环节,所以英语的准备非常重要,除非你备考过 GT 的高手。一般国内的英语水平都比较一般,所以如果英语不是太好的话,求职者应尽量在考前 1 个月多做英文题,如 GRE 的推理题、英文的数据结构题等。

14) 信面经(面试经验),但不全信。外企的门槛一般比较高,网上的一些有关知名外企的面经一般都是由一些“牛人”写的,他们站在自己的角度看问题,一些小的细节障碍、重点、难点对他们而言,可能太“小儿科”了,所以他们可能都不提,轻松地就“跨”过去了,而对于与之水平差距比较大的人而言,可能就是这些小细节就决定了成败,所以,求职者对此要抱着消化吸收的心态来学习,取其精华,去其糟粕,不要完全照搬。

15) 外企的笔试题目一般比较多,题量比较大,求职者答题时的速度一定要快。如果不是“牛人”,那么还是做好足够的心理准备,尽快做会做的,把会做的做全、做好,不太会的也要尽可能把会的部分写出来。

16) 在笔试前,求职者应尽量总结历年的考题。面试中,客观题必考,数据结构与算法设计能力则需要培养。主观题不同,一般都是发挥题。还会考测试用例的题目,所以求职者一定要多找些资料,归纳总结。编程类题目一般都有与树相关的数据结构,而且算法多样,所以一定要认真准备。

17) 笔试面试前,求职者一定要调整心态,从战略上藐视它,在战术上重视它,发挥出自己的真实水平就好,不要因为太在意、太认真而发挥失常。

18) 平时多练习数据结构与算法的题,发散思维,也可尝试在纸上手写程序,积累纸上写程序的经验。

19) 外企的待遇虽然丰厚,但也有一些劣势,主要包括工作压力会比较大、职业发展会存在“天花板”问题,而且失业率比较高,尤其是老员工,一旦上了年纪,如果还未能升至一定级别,很有可能会被炒鱿鱼。经济不景气时,失业的可能性更大。此外,员工在外企容易变成“螺丝钉”,这是因为外企职责分明,工作分得很细致,所以员工最终可能只能干本职工作,而其他职位的工作能力得不到发展,对于未来跳槽,可能会有一定的影响。

3. 真题分析

2008 年某知名咨询公司笔试真题。

1) 123456789 的火车经过如下轨道从左边入口处移到右边出口处(每车只能进临时轨道 M 1 次)按照从左向右的顺序,下面的结果不可能是_____。

A. 123876549 B. 321987654 C. 321456798 D. 789651234

2) 对于上一题,如果 M 只能容纳 4 列车,应选上面的_____选项。

3) 3 3 8 8 用四则运算符如何得出 24。

4) 编程实现:可变长有序数组的插入(无重复数据结点)。

5) 数 a 和 b,如何空间消耗最小交换 a b 中的数。

6) For the following description about OOP, which is right?

① An object can inherit the feature of another object.

- ② A sub class can contain additional attribute or behaviors.
- ③ Encapsulation is used to hide as much as possible about the inner working of the interface.
- ④ Encapsulation prevents the program from becoming independent.
- ⑤ Polymorphism allows the methods have different signature but with same name.
- A. ①, ② B. ①, ④ C. ②, ③ D. ③, ⑤ E. ④, ⑤

7) Function club is used to simulate guest in a club. With 0 guests initially and 50 as max occupancy, when guests beyond limitation, they need to wait outside; when some guests leave, the waiting list will decrease. The function will print out number of guests in the club and waiting outside. The function declaration as follows: void club (int x); positive x stands for guests arrived, negative x stands for guests left from within the club. For example, club (40) prints 40, 0; and then club (20) prints 50, 10; and then club (~ 5) prints 50, 5; and then club (~ 30) prints 25, 0; and then club (~ 30) prints N/A; since it is impossible input. To make sure this function works as defined, we have following set of data to pass into the function and check the result are correct.

a 60

b 20 50 ~ 10

c 40 ~ 30

d 60 ~ 5 ~ 10 ~ 10 10

e 10 ~ 20

f 30 10 10 10 ~ 60

g 10 10 10

h 10 ~ 10 10

A a d e g

B c d f g

C a c d h

D b d g h

E c d e f

4. 推荐知识点学习

在非技术问题上, 外企比较注重英文水平、学习能力、表达能力、团队合作能力和沟通能力; 技术问题, 外企侧重于数据结构与算法、C/C++ 等基础知识。需要特别强调的是, 数据结构与算法需要平时积累, 很难通过突击取得巨大成效, 所以求职者平时应多练习算法题。

3.4 国企

2008 年金融危机, 当民企、外企都在困难中艰难前行, 大幅度裁员、降薪时, 国企却依然岿然不动, 几乎没有受到巨大的影响, 反而继续保持发展态势。经过这次风暴后, 越来越多的人开始意识到, 尽管国企有它自身的局限性, 但仍然是一个非常不错的选择, 所以国企在求职者心中的地位自然也大大提升。而且, 随着国企改革的进一步深化, 国企在人才引进上也逐步与市场接轨, 人事制度的进一步完善使其招聘人才的方法也日趋科学合理, 所有这些都使得国企招聘变得炙手可热, 成了求职者心中的“香饽饽”。

1. 招聘流程

由于国企自身的特点，使其在招聘时，面试风格和其他类型的企业有所不同，带有明显的国企特色。国企一般对应届毕业生比较感兴趣，他们的校园招聘时间一般也比其他类型的企业晚，所以有志于进入国企工作的人一定要有心理准备，是忽略其他企业的存在、一直等到国企的到来，还是先选定一家企业保底然后继续等待更好的出现？求职者一定要明确自己的目标，切忌摇摆不定。

国企的招聘流程为：投递简历→人事面→主任面→录用 offer

2. 面试笔试注意事项

国企的面试笔试一般比较保守，但求职者自由发挥的余地也相对比较狭窄。所以，对于国企，求职者只要认真准备，在面试笔试的过程中，不犯一些典型的低级错误，最后过关还是有很大希望的。当然，希望要变为现实，并非“脑门一热、金口一开”就可实现，还需要天时、地利与人和。天时、地利一般都能具备，主要还是需要个人不断地努力，除了需要掌握基本的面试技巧外，同时还要清楚以下一些方面的内容：

1) 区别各种聘用制度。国企中并非所有人都是体制内的员工，有的是事业编制，而有的是劳务派遣，还有的是合同工，各种聘用制度下的员工的待遇、福利、晋升机会等都会存在很大的区别，如公积金、年终奖以及各种补贴等，所以在与国企签订三方协议时，求职者一定要弄清楚聘用制度。

2) 不用担心会在国企内埋没自己，只要有能力，在国企里，个人还是有很大发展空间的。是金子终究可以发光的，尤其是在大型的国企，例如银行、能源等单位，人事制度、管理都是非常完善的，对人才也是非常重视的，只要有实力，而且乐于奉献，一样可以有所作为。

3) 除了技术性特别强的职位，一般国企招聘笔试面试都不会重点考查求职者的专业知识，他们更注重的是求职者是否具有较高的政治素质、是否具有踏实肯干的良好品质、是否遵纪守法、是否认同企业文化、是否脚踏实地等。所以，国企一般对学生党员、学生干部比较感兴趣，因此在面试过程中，求职者一般需要主动地向面试官表现出自己这方面的优势，如果本身的政治素质过硬，就更容易获得面试官的青睐。

4) 国企一般来说不太喜欢面试中个性张扬的人，中规中矩、举止稳重、行为朴实的求职者更容易得到面试官的青睐。

5) 多才多艺可为面试加分。许多求职者在应聘国企时并不是因为专业能力出众被录用，而是以一技之长而得到面试官的青睐。例如棋琴书画皆会、球技出众、爱好广泛等。

6) 着装要正式。对于特定的岗位，有时一定要穿正装，一般来讲，保守一些的颜色更易被接受，深色西装、白衬衫、黑皮带、黑皮鞋都是商务着装的首选。毕业生在应聘银行的职位时，需要特别注意形象，但也不用太过关注品牌。对于刚毕业的学生而言，气质才是真正能够吸引面试官的地方。同时，应届毕业生最好不要用香水，国企更多看中的是应届毕业生的可塑性和发展性，喷香水会给人太过老成的感觉，反而不好。另外，男生不要染发或是做发型；女生也一定要打扮得体，不要太妖娆，不要佩戴过多的首饰，可以表现得有朝气，但不能表现得不成熟。

7) 国企比较重视学历与学习背景，较高的学历不但可以在求职时占有优势，入职后工资待遇也会有差别（但差别并不大）。但学历越高，晋升空间一般越大，发展前景一般更好。

8) 同等条件的外地人与本地人，国企一般更喜欢招收本地人。例如，某些运营商、银行在其所属省级、市级、县级的分公司都愿意招聘在外地求学的本地人充实自己的队伍，因为这些本地人更熟悉当地的方言、文化等，对于未来开拓市场更有优势，所以对有意回家乡发展的求职者来讲，这不失为一个很好的机会。

9) 国企很少对求职者进行英语面试, 但许多国企也特别在意求职者的“门面”问题, 虽然不会对求职者的英语水平进行直接考核, 但是还是希望求职者具备一定的英语水平, 例如求职者是否拥有大学英语四级或六级证书, 是否通过了托福或者 GRE 考试等。

10) 由于评判求职者个人能力强弱具有一定的主观性, 国企一般比较看重求职者手中是否有证书, 如国家级奖学金 (包括国家奖学金、国家励志奖学金等)、学校奖学金、社会奖学金等各类奖学金, ACM 竞赛获奖、数学建模获奖、高等数学竞赛等各类学科竞赛获奖, 英语竞赛获奖、辩论赛获奖等, 因此求职者如果能够在简历或面试中, 让面试官了解到自己有哪些证书, 会对求职者非常有用。

11) 国企需要的是稳重踏实、为人低调、耐得住寂寞、扛得住诱惑、守得住繁华的人, 那些面试时显得比较强势、夸夸其谈、激进急躁的人很难被录用。你可以不比别人聪明, 但一定要比别人努力。所以面试过程中, 求职者应该在展现自己的能力时, 应尽量保持低调。

12) 国企面试的问题常常会夹杂一些个人家庭背景等问题, 比如是否为独生子女, 父母工作情况等, 求职者只要如实回答就行, 不用过多地揣摩其中所谓的“深意”, 也不要撒谎, 撒谎始终都是被人厌恶的行为。

13) 对于网上的面经, 要取其精华、去其糟粕。在这些“老国企”面前, 求职者不要畅谈对国企的认识, 以免影响最终的面试结果。

3. 真题分析

2010 年某银行计算机类考试笔试题。

第 1 大题 判断题 20 道

略。

第 2 大题 单项选择题 40 道

略。

第 3 大题 简答题 2 道

1) 构成死锁的必要条件是什么, 如何检测死锁, 解除死锁?

2) 画出星形、树形、总线型、环形网络拓扑结构, 并写出星形、总线型网络拓扑结构的特点。

第 4 大题目 综合应用题

1) 多表查询: 从 S (学号, 姓名, 年龄, 生日) 表和 SC (学号, 课程号, 成绩) 中查询出没有选择课程号为 1001 的课程的所有学生的学号和姓名。

2) 根据程序写出其输出结果。

```
void main()
```

```
static char arr[5] = {'*', '*', '*', '*', '*'};
```

```
int i, j, k;
```

```
for(i=0; i < 5; i ++)
```

```
{
    printf("\n");
```

```
    for(j=0; j < i; j ++ ) printf(" ");
```

```
    for(k=0; k < 5; k ++ ) printf("%c", arr[k]);
```

3) 写出以下程序实现的功能。

```

void main()
{
    int a, b, c, *pa, *pb, *pc, *p;
    pa = &a;
    pb = &b;
    pc = &c;
    scanf("%d,%d,%d", pa, pb, pc);
    if(*pa > *pb) { *p = *pa; *pa = *pb; *pb = *p; }
    if(*pa > *pc) { *p = *pa; *pa = *pc; *pc = *p; }
    if(*pb > *pc) { *p = *pb; *pb = *pc; *pc = *p; }
    printf("%d,%d,%d", *pa, *pb, *pc);
}

```

4) 写出表达式的后缀形式。

5) 给出 A ~ H 8 个字母各自出现的概率，写出它的最优二进制编码，并画出最优二叉树和计算出平均码长。

面试中的技术题目。

1) 需求分析中，需要确定项目哪些方面的可行性？

2) 是否熟悉 Java？Java 有什么特点？

3) 构造函数与初始化列表的区别。

面试中的非技术真题。

1) 自我介绍。

2) 谈谈你的家庭情况。

3) 说说你的业余爱好。

4) 结合你实际情况（教育、背景、经历、性格特点、优缺点、兴趣），谈谈你对未来 3 年或 5 年的生活和工作的规划。

5) 你如何看待一个人以往的工作经验和他今后工作绩效之间的关系？

6) 请用英文陈述你对企业的认识和了解到的企业文化。

7) 简历内容相关问题。

8) 你为什么要选择我们？

9) 你认为我们为什么要选你，而不去选其他人？

10) 你对我们的了解有多少？

11) 你能为企业做些什么吗？

12) 如何看待职业生涯中“骑驴找马”的现象？

13) 周围的同学是怎样看你的？

14) 面试了哪些公司？

15) 是否签约了？

4. 推荐知识点学习

从考试内容上看，国企的笔试对技术的要求一般都不是很高，虽然囊括了计算机专业的所有课程（C/C++ 语言、面向对象、数据库、数据结构、操作系统、计算机组成原理、编译原理、多媒体技术、计算机网络、离散数学、设计模式等），但考试内容都是基础知识。虽然知识面涉及得非常广，但也并非什么都考，考试内容都是最常见的知识，同时知识点并不是很深，除此之外，还涉及了少量的业务知识。

针对这些问题，求职者要认真复习计算机专业知识，特别是常见的问题。另外，国企更多

的是考查求职者的综合能力，包括为人处世能力、表达能力、学习能力、反应能力等，求职者还需要提前准备一些企业文化相关内容。

3.5 研究所

近年来国家对研究所的投入与日俱增，研究所员工的待遇、社会地位较以往大幅提高，因此越来越多的应届毕业生都把进入研究所作为了自己实现个人价值的途径。

1. 招聘流程

与民企、外企相比，研究所因为单位性质的不同，其招聘结束方式、招聘流程也与民企、外企不太相同。研究所的招聘一般都比较晚，很多都是等在其他企业校园招聘结束后才开始进行招聘，一般在10月底、11月初，有很多研究所甚至将招聘放到了第二年才进行。

一般研究所的招聘流程为：投递简历→人事面→主任面（集体面试）→录用 offer。研究所的宣讲招聘形式也很有特色，除了进行校园宣讲外，还会组团到高校进行招聘，如“军工集团进校园”等。

2. 面试笔试注意事项

由于研究所不同于其他类型企业（虽然为科研设计单位，但也从事产品生产活动），且随着事业单位的改革，也开始向企业转型。在面试笔试时，既保留了部分国企的形式风格，也具备现代企业的招聘形式。因此，求职者需要弄清楚与研究所相关的信息，才能更好地准备面试笔试，除了注意一些常规的面试技巧外，还需要注意以下一些事项：

1) 研究所比较注重求职者的毕业学校、学历，越是效益好、职工福利好的研究所，对求职者的毕业学校、学历要求越高，一般都要求双“211”或者双“985”的硕士毕业生（即本科与研究生所在学校都是“211”、“985”高校）。少量非核心专业可能会降低一些要求，但也至少要求“211”或“985”高校的本科。

2) 研究所的待遇一般比企业要低，但是福利会好，基本工资不高，奖金在收入中占了很大比例，除此之外，还有岗位工资、年终奖、住房补贴、交通补贴、餐补、单身宿舍、食堂等，而且研究所的住房公积金的比例也较高。同时，由于研究所属于国家科研设计单位，新入职的员工一般有住房补贴，而其他类型企业可能没有。

3) 研究所一般没有笔试，即使有笔试，也只会侧重计算机基础知识的考核，而且笔试题目涉及的范围虽然比较广，但不会太深，都是最常见的问题。研究所的面试也与其他类型企业面试不同，研究所组织的面试一般分为两轮，第一轮是由人力资源部组织的人力资源面试，主要是对求职者的简历有一个基本的了解与确认，例如家庭、学历、成绩、奖励等内容。通过第一轮面试的求职者一般能够进入到第二轮面试中，第二轮面试一般都是各个部门的主管进行多对一的集中面试。

4) 在面试的过程中，研究所不会很关注于知识细节，对于专业知识，他们更加注重求职者的专业、项目是否对口。除此之外，因为研究所有一定的国企特色，所以更加注重求职者的综合实力，包括语言表达能力、运动能力、个人才艺等。

5) 在与HR谈待遇时，求职者应注意区别事业编制与企业编制，是体制内的事业编制人员还是体制外的合同工。事业编制的好处是不用交养老保险，退休以后领取退休金，更加稳定。

6) 每个研究所都有自己的专攻方向，也就是核心部门和配套部门，所以研究所各个部门、科室、岗位待遇、工作强度、发展前景一般并不一样，有时相差很大，所以作为刚入职的应届

生，还是应该“擦亮眼睛”看清楚，除了了解整个研究所的效益以外，还要看部门的效益、发展前景等。

7) 很多研究所在招聘时会说有福利分房的可能性，作为应届毕业生，也一定要自己心里清楚，福利分房只针对体制内的人，如果连事业编制都没有，是没有分房可能的。而且即使具有事业编制，已经成为“体制内”的人了，但如果要分房，也是需要论资排辈的，应届毕业生无论按什么进行排队都很难在短时间内分到房，所以对分房子这一福利一定要持谨慎态度。

8) 虽然研究所一般不独立培养本科生，但是很多研究所都有硕士学位点或是博士学位点，如果希望进某一个研究所，求职者可以通过保研或考研进入研究所，毕业后极有可能留在研究所工作。

9) 研究所一般与高校的教师有一些项目往来，如果有机会参与到这种项目中，最后通过导师推荐或是项目匹配一般也有可能拿到研究所的 offer。需要注意的是，大多数研究所的项目一般以底层开发为主，即 C/C++，对于上层的 Web 应用开发、云计算涉及的比较少，所以以 Java、.NET 为强项的求职者可能会“吃亏”。在学习期间，如果有机会，求职者应尽量多参与一些以 C/C++ 为开发语言的嵌入式软硬件开发项目。

10) 如果手头没有三方协议，而研究所又急于签约时，应届毕业生应该尝试与研究所签订一份双方认定的协议。

11) 国家对研究所有政策照顾。例如，对于北京户口，研究所一般可以解决，而在其他类型企业是很难解决的，对于想去北京的求职者而言，进入研究所工作是个非常不错选择，当然，并非全部研究所都是这样，求职者须在面试中询问清楚。

12) 早些年，研究所一般只招收应届毕业生，近些年，随着事业单位改革，也会招收少量的非应届人员，但数量也不多，所以对于希望进研究所的人，以应届毕业生的形式进入的可能性较之非应届会更大一些。相反，从事科研技术的人员，如果在研究所工作若干年后离职的员工，一般也可以跳槽到高校、企业，不会受到大的影响。

13) 研究所的文化氛围较外企、民企更加浓厚，对员工的人文关怀也更多，工作环境也比较好。

14) 不同于外企，研究所作为国家单位，面试官更多地把目光聚焦在求职者对企业忠诚、政治上上进、遵守规章制度、乐于付出等方面。

15) 随着国家对事业单位的改革，很多研究所都在逐步向企业转型，所以，很多研究所都改名为公司名字或是成立了很多下属子公司，所以在求职的过程中，求职者一定不要因为研究所改名而错过了投递简历或是投递到了下属子公司，一定要及时地关注并弄清楚研究所的名称以及研究所与下属子公司招聘的相关情况，以免出现理解错误而造成失之交臂或张冠李戴的后果。

3. 真题分析

“基础不牢，地动山摇”。技术类笔试面试题都是一些最常见的基础知识。以下是一些常见的研究所的非技术类面试题。

- 1) 自我介绍。
- 2) 你家是外地的，为什么要留在这里工作？
- 3) 你有男/女朋友吗？在哪里工作？
- 4) 做过什么项目？
- 5) 个人特长有哪些？有什么兴趣爱好吗？
- 6) 你的优点是什么？你的缺点是什么？
- 7) 如果现在录用你，你能立刻来实习吗？

- 8) 高考成绩如何? 重点线是多少? 为什么不选择更好的学校或是就近读书?
- 9) 在学校期间, 有什么事情是你觉得做得最好的?
- 10) 你的成绩不是很好, 为什么没有取得非常好的成绩?
- 11) 除了学习以外, 业余你还有别的兴趣爱好吗?
- 12) 你为什么愿意到我们所来工作?
- 13) 有拿到其他企业的 offer 吗? 为什么不去那里而选择我们?
- 14) 平时有什么爱好吗? 篮球水平怎么样? 乒乓球呢?

推荐知识点学习

通过研究所的面试笔试真题不难发现, 研究所一般不设置笔试, 即使有也是程序设计基础的最常考内容。对于面试, 涉及技术层面的也仅仅只是个人项目相关问题, 所以在应聘研究所前, 求职者应仔细研究个人项目, 同时好好分析并研究该研究所做的主要项目, 然后进行有针对性的准备。

3.6 创业型企业

当拉里·佩奇·谢尔盖·布林在斯坦福大学宿舍里面研究搜索算法时, 当马克·扎克伯格在哈佛大学为方便同学交流研究社交网络时, 没有谁能够想得到, 他们有一天能够改变整个世界。然而最终他们做到了, 只要有梦想、激情、能力、毅力, 在这样一个开放的时代, 创业不再是天方夜谭, 成功也不再遥远。

社会的进步离不开广大勤勤恳恳、脚踏实地、辛勤工作的人, 但真正推动社会进步的却是那些充满激情、将梦想变为现实的热血青年, “为自己打工”已经变得不再艰难, 无数有志青年都选择了将自己的前途寄托在自己身上, 进行了创业。创业让生活更加充满梦想与挑战。

本书以目前国内最大的图片购物搜索引擎公司——淘淘搜为例进行分析。

1. 招聘流程

成立于2010年4月的淘淘搜, 是全球领先的图片购物搜索引擎, 总部设在杭州, 并在北京设有运营团队。其中, 研发人员占员工总数的60%以上, 均在杭州工作。

淘淘搜的校园招聘一般于当年9月份开始, 并于10~11月会进行全国巡回宣讲与招聘。每年招聘的人数约为40人, 涉及的岗位包括算法研发工程师、C++开发工程师、前端开发工程师、Java开发工程师、视觉设计师、产品助理、运营助理等。近些年, 随着业务不断扩大, 招聘人数也在不断发展。

淘淘搜校园招聘采取定点培养、优中选优的精英策略, 定点高校包括浙江大学、华中科技大学、武汉大学、西安电子科技大学、四川大学、电子科技大学等。

淘淘搜的应聘流程一般比较严格, 包括以下几个步骤: 宣讲会→筛选简历→笔试→专业面试→HR面试→综合面试→最终录用。而简历的筛选环节将被统一安排在城市笔试环节开始前一周进行, 一般很少会通过简历淘汰求职者, 各个筛选环节的通过名单以及下一步安排一般都会通过淘淘搜HR的官方网站、短信、电话等形式予以通知, 所以求职者应及时关注相关信息。

2. 面试笔试注意事项

作为一家能够在IT浪潮中存活下来的创业型企业, 淘淘搜一方面不断学习一些大企业成功的经验, 同时也不断推陈出新, 积极发扬互联网开放的精神。

在求职创业型企业时, 除了注意常规的面试技巧和方法以外, 求职者还需要注意以下几个方面的内容:

1) 在面试中, 求职者尽可能不要与面试官谈及该企业与其他大型企业比较的劣势, 也不要过分关注眼前的利益, 创业型企业短期内的回报可能不如某些成熟型的大企业。但是, 如果在创业早期即成为企业骨干、核心, 未来将会大有作为。

2) 求职者应在面试前清楚一个事实, 即在创业型企业里, 每个员工所起的作用不仅仅局限在某一个单元或是某一个模块上, 而是可能同时交叉进行多项工作。

3) 创业型企业的招聘规模都不是很大, 所以招聘所能涉及的城市以及大学有限, 如果有志于进入创业型企业, 求职者最好能够提前做好必要的功课, 如寻求内推机会等, 否则很有可能失之交臂。

4) 在面试的过程中, 求职者尽可能地在面试官面前体现出坚强、创新、团结的品质, 因为创业型企业要在大企业间搏杀的“夹缝”中生存下来, 靠得就是这样一群有梦想、有追求、有激情、团结、博爱、创新、坚强的青年才俊。

5) 在挑选创业型企业时, 求职者应尽量挑选一些在沿海或是南方大城市的企业。相较于内地, 沿海城市或是南方大城市经济更加发达, 产业链更加完整, 市场更加开放, 机会也更多, 所以在此生根发芽的创业型企业, 生命力更强, 发展前景更好。

3. 真题分析

以下为淘淘搜 2011 年技术类笔试真题。

第一部分: 基础知识

1) 请用 C 或 Java 语言写出 BOOL 变量 flag 与“零值”比较的 if 语句_____。float 变量 x 与“零值”比较的 if 语句_____。char 指针变量 *p 与“零值”比较的 if 语句_____。

2) 下面第_____个 for 循环是无限循环。

① for (int i = 0; i == 10; i + = 0); ② for (int i = 10; (i ++ ^ - - i) == 0; i + = 0)。

3) C 语言参数的入栈顺序是_____。

4) 用 C 语言预处理指令 #define 声明一个常数, 用以表示一年有多少秒 (假设一年有 365 天) _____。

5) Linux 结束后台进程的命令是_____。

6) 以下 Linux 命令对中, 正确的是 ()

① ls 和 sl; ② cat 和 tac; ③ more 和 erom; ④ exit 和 tixe。

7) 下面_____条命令是在 vi 编辑器中执行存盘退出。

① : q; ② ZZ; ③ : q!; ④ : wq。

8) Linux 字符串查找命令是_____, nohup 命令的作用是_____。

9) 在 OSI 7 层模型中, 网络层的功能有_____。

① 确保数据的传送正确无误; ② 确定数据包如何转发与路由;

③ 在信道上传送比特流; ④ 纠错与流控。

简答:

10) TCP 和 UDP 有什么区别?

11) 简单描述一下 TCP/IP 建立连接的过程?

12) ping 命令是基于什么协议实现的, 这个协议处于哪一层?

13) 描述一下 Linux 的进程间通信方式。

14) 继承、多态、封装和抽象, 哪种面向对象的方法可以让你变得富有, 为什么?

15) 《公孙龙子》记载: “齐王之谓尹文曰: ‘寡人甚好士, 以齐国无士, 何也?’ 尹文曰: ‘愿闻大王之所谓士者。’ 齐王无以应。” 说明齐王_____。

①昏庸无道；②是个结巴；③不会下定义；④不会定义自己的需求。

16) 蒯相如，司马相如；魏无忌，长孙无忌。下列哪一组对应关系与此类似，请做出解释

①PHP, Python; ②JSP, servlet; ③java, javascript; ④C, C++。

解释：

第二部分：图像处理与分析

略。

第三部分：C/C++ 程序设计与数据结构

1) 请使用 C 语言给出下面变量 a 的定义。

a) An integer;

b) A pointer to an integer;

c) A pointer to a pointer to an integer;

d) An array of 10 integers;

e) An array of 10 pointers to integers;

f) A pointer to an array of 10 integers;

g) A pointer to a function that takes an integer as an argument and returns an integer;

h) An array of ten pointers to functions that take an integer argument and return an integer;

2) 阅读以下 C++ 程序，写出运行结果。

```
class A
{
public:
    void f1()
    {
        printf("A::f1\r\n");
    }
    virtual void f2()
    {
        printf("A::f2\r\n");
    }
    void callfunc()
    {
        printf("A::callfunc\r\n");
        f1();
        f2();
    }
};

class B : public A
{
public:
    void f1()
    {
        printf("B::f1\r\n");
    }
    void f2()
    {
        printf("B::f2\r\n");
    }
};
```



```

}
void callfunc()
{
    printf("B::callfunc\r\n");
    f1();
    f2();
}
};
int main()
{
    B * pB = new B;
    pB -> callfunc();
    A * pA = pB;
    pA -> callfunc();
    return 0;
}

```

程序输出:

3) 实现两个 $N * N$ 矩阵的乘法, 矩阵由一维数组表示。设矩阵 A_{NN} 和 B_{NN} 分别为:

$$A_{NN} = \begin{bmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & & \vdots \\ a_{N1} & \cdots & a_{NN} \end{bmatrix}, \quad B_{NN} = \begin{bmatrix} b_{11} & \cdots & b_{1N} \\ \vdots & & \vdots \\ b_{N1} & \cdots & b_{NN} \end{bmatrix}$$

4) 请用 C/C++ 编程实现将整数 12345 转换成字符串 (要求: ①C 或 C++ 编程语言可任选一种; ②在 Windows 和 Linux 环境下都可以编译通过并输出正确结果)。

5) 请用 C/C++ 编程实现单链表的逆置 (要求: ①C 或 C++ 编程语言可任选一种; ②在 Windows 和 Linux 环境下都可以编译通过并输出正确结果; ③撰写 gcc 工程管理文件 makefile) (请填写在答题纸上, 注明题号)。

第四部分: QA

1) 一套完整的测试应该由哪些阶段组成?

2) 简述黑盒测试、白盒测试、单元测试、集成测试、系统测试、验收测试的区别与联系。

3) 对于图 3-1 的程序流, 采用语句覆盖法设计测试案例, 至少需要设计几个测试案例, 并简述设计策略。

4) 为了验证程序是否实现单模块功能, 需要进行 (A); 为了验证单模块和其他模块按照规定方式工作, 需要进行 (B), 请简述理由。

(A) a. 单元测试 b. 集成测试 c. 确认测试
d. 功能测试

(B) a. 单元测试 b. 集成测试 c. 功能测试
d. 系统测试

5) 后台, 一个文本框, 要求输入 10 ~ 40 个长度的任意格式的字符串, 要求输入的字符可以在前台正常显示, 请据此设计测试用例及数据, 以完整把握功能的正常使用, 并阐述设计方法和思想。

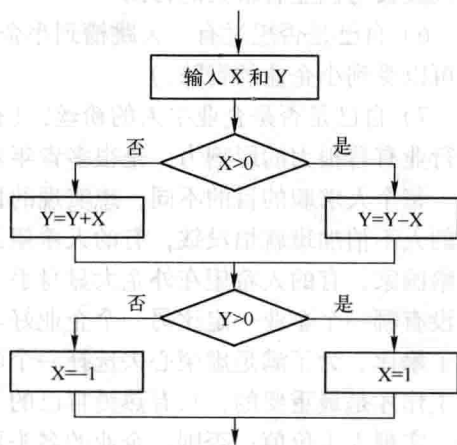


图 3-1 程序流程图

4. 推荐知识点学习

创业型公司对人才的考核与成熟型大型企业会不一样，在业务类岗位上，他们更偏重考查逻辑推理能力及创新能力，而技术类岗位则偏重考查专业知识的理解及自学能力，每位参加面试的求职者可以在一套笔试题里选择自己感兴趣并擅长的题目。

所以，在准备创新型企业的面试笔试时，求职者最好在夯实计算机基础专业知识的基础上，提高自身的综合能力。

3.7 如何抉择

求职如同择偶，好企业很多，素质很高的求职者也很多，但并非每个好员工都能成为每一个好的企业的“螺丝”，与每一个好企业实现无缝连接。所以，求职者在选择一个企业前，一定要想清楚，这个企业是否适合自己。是否是自己希望的工作类型。自己能否在这家企业安心工作。否则，如果去了一家尽管很好，甚至是世界五百强的大企业，但却不适合自己，就像谈恋爱没找到适合自己的对象一样。

不要以为自己很重要，对于企业而言，即使没有你的存在，企业的发展也不会受到多大影响，而对于你自己而言，却可能付出惨痛的代价，失去青春、激情甚至是前途。

一般情况下，求职者在选择 offer 时需要综合考虑很多方面，以下列举出一些常见的项目供求职者参考。

1) 是否因为感情问题，例如爱情、亲情问题选择一座城市？自己的兴趣爱好是什么？是喜欢开放性程度好的城市还是开放性相对低一些的城市？

2) 自己是否缺钱？（互联网企业、外企一般待遇丰厚，对于缺钱的求职者而言，非常具有诱惑力）。

3) 自己是喜欢具有挑战性的工作还是喜欢相对稳定的工作？自己是新技术“发烧友”还是普通的“技术控”？（一般来说，国企工作相对稳定，外企、民企的挑战性会更大一些。）

4) 自己是否对这个行业有着非常高的热情？虽然说“干一行，爱一行”，但更多时候是爱一行了才能干好一行，兴趣是最好的老师。否则，在自己不喜欢的行业、企业，很有可能荒废自己，所以，选择企业的时候一定要考虑清楚这个问题。

5) 自己是否有志于创业？（一般而言，在大公司磨练若干年，积累一些大公司的经验，对个人成长与创业有很大的帮助。）

6) 自己是否想过有一天跳槽到小企业当个小头目？（在一些知名的大企业的工作经历，一般可以受到小企业的青睐。）

7) 自己是否是企业牛人的粉丝？（很多大企业的 CEO 或董事局主席都是行业的名人，在本行业有着很大的影响力，是很多青年才俊崇拜的偶像。）

每个人求职的目的不同，想实现的目标也不尽相同，有的人想去外企，有的人想去国企，有的人不怕加班就怕没钱，有的人希望工作轻松稳定而不在意钱多钱少，有的人希望把青春奉献给国家，有的人希望在外企大显身手。其实，选择没有对错，求职也不是简单的买菜卖菜，更没有哪一个企业一定比另一个企业好。因此，最重要的是选择一个适合自己的企业，而不是为了攀比、为了满足虚荣心去选择一个可能名号很响亮，却并不一定适合自己的企业。开开心心工作才是最重要的，只有热爱自己的工作，才能把工作做好，进而让自己的生活更加丰富多彩，实现人生价值；否则，企业的名头再响亮，提供的平台再好，可是自己不喜欢，最终也很难在工作岗位上做出成绩来。

下篇

左图右说 1.5

面试笔试技术攻克篇

第 4 章 Java 基础知识

第 5 章 Java Web

第 6 章 数据库原理

第 7 章 设计模式

第 8 章 数据结构与算法

第 9 章 海量数据处理

附录

第4章

Java 基础知识

4.1 基本概念

4.1.1 Java 语言有哪些优点

Sun Microsystems 公司对 Java 语言的描述如下：“Java is a simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language”。具体而言，Java 语言具有以下几个方面的优点：

1) Java 为纯面向对象的语言。《Java 编程思想》提到 Java 语言是一种“Everything is object”的语言，它能够直接反应现实生活中的对象，例如火车、动物等，因此通过它，开发人员编写程序更为容易。

2) 平台无关性。Java 语言可以“一次编译，到处运行”。无论是在 Windows 平台还是在 Linux、MacOS 等其他平台上对 Java 程序进行编译，编译后的程序在其他平台上都可以运行。由于 Java 为解释型语言，编译器会把 Java 代码变成“中间代码”，然后在 Java 虚拟机（Java Virtual Machine, JVM）上解释执行。由于中间代码与平台无关，因此，Java 语言可以很好地跨平台执行，具有很好的可移植性。

3) Java 提供了很多内置的类库，通过这些类库，简化了开发人员的程序设计工作，同时缩短了项目的开发时间，例如，Java 语言提供了对多线程的支持，提供了对网络通信的支持，最重要的是提供了垃圾回收器，这使得开发人员从对内存的管理中解脱出来。

4) 提供了对 Web 应用开发的支持，例如，Applet、Servlet 和 JSP 可以用来开发 Web 应用程序；Socket、RMI 可以用来开发分布式应用程序的类库。

5) 具有较好的安全性和健壮性。Java 语言经常被用在网络环境中，为了增强程序的安全性，Java 语言提供了一个防止恶意代码攻击的安全机制（数组边界检测和 Bytecode 校验等）。Java 的强类型机制、垃圾回收器、异常处理和安全检查机制使得用 Java 语言编写的程序有很好的健壮性。

6) 去除了 C++ 语言中难以理解、容易混淆的特性，例如头文件、指针、结构、单元、运算符重载、虚拟基础类、多重继承等，使得程序更加严谨、简洁。

常见笔试题：

Java 语言是由（ ）语言改进并重新设计而来的。

A. Ada B. C++ C. Pascal D. BASIC

答案：B。Ada 语言是美国军方为了整合不同语言开发的系统而发明的一种语言，其最大

的特点是实时性，在 Ada 95 中已加入面向对象内容。Pascal 语言是为提倡结构化编程而发明的语言。BASIC 语言为了让大学生容易地控制计算机开发的语言，其特点是简单易懂，且可以用解释和编译两种方法执行。C++ 语言是一种静态数据类型检查的、支持多重编程范式的通用程序设计语言，它支持过程化程序设计、数据抽象、面向对象程序设计、泛型程序设计等多种程序设计风格。Java 语言是一种面向对象语言，从语法结构上看，与 C++ 类似。

4.1.2 Java 与 C/C++ 有什么异同

Java 与 C++ 都是面向对象语言，都使用了面向对象思想（例如封装、继承、多态等），由于面向对象有许多非常好的特性（继承、组合等），因此二者都有很好的可重用性。

需要注意的是，二者并非完全一样，下面主要介绍它们的不同点：

1) Java 为解释性语言，其运行过程为：程序源代码经过 Java 编译器编译成字节码，然后由 JVM 解释执行。而 C/C++ 为编译型语言，源代码经过编译和链接后生成可执行的二进制代码。因此，Java 的执行速度比 C/C++ 慢，但是 Java 能够跨平台执行，而 C/C++ 不能。

2) Java 为纯面向对象语言，所有代码（包括函数、变量等）必须在类中实现，除基本数据类型（包括 int、float 等）外，所有类型都是类。此外，Java 语言中不存在全局变量或全局函数，而 C++ 兼具面向过程和面向过程编程的特点，可以定义全局变量和全局函数。

3) 与 C/C++ 语言相比，Java 语言中没有指针的概念，这有效防止了 C/C++ 语言中操作指针可能引起的系统问题，从而使程序变得更加安全。

4) 与 C++ 语言相比，Java 语言不支持多重继承，但是 Java 语言引入了接口的概念，可以同时实现多个接口。由于接口也具有多态特性，因此在 Java 语言中可以通过实现多个接口来实现与 C++ 语言中多重继承类似的目的。

5) 在 C++ 语言中，需要开发人员去管理对内存的分配（包括申请与释放），而 Java 语言提供了垃圾回收器来实现垃圾的自动回收，不需要程序员显式地管理内存的分配。在 C++ 语言中，通常都会把释放资源的代码放到析构函数中，Java 语言中虽然没有析构函数，但却引入了一个 finalize() 方法，当垃圾回收器将要释放无用对象的内存时，会首先调用该对象的 finalize() 方法，因此，开发人员不需要关心也不需要知道对象所占的内存空间何时会被释放。

C++ 语言支持运算符重载，而 Java 语言不支持运算符重载。C++ 语言支持预处理，而 Java 语言没有预处理器，虽然不支持预处理功能（包括头文件、宏定义等），但它提供的 import 机制与 C++ 中的预处理器功能类似。C++ 支持默认函数参数，而 Java 不支持默认函数参数。C/C++ 支持 goto 语句，而 Java 不提供 goto 语句（但 Java 中 goto 是保留关键字）。C/C++ 支持自动强制类型转换，这会导致程序的不安全；而 Java 不支持自动强制类型转换，必须由开发人员进行显式地强制类型转换。C/C++ 中，结构和联合的所有成员均为公有，这往往会导致安全性问题的发生，而 Java 根本就不包含结构和联合，所有内容都封装在类里面。

Java 具有平台无关性，即对每种数据类型都分配固定长度，例如，int 类型总是占据 32 位，而 C/C++ 却不然，同一个数据类型在不同的平台上会分配不同的字节数。

Java 提供对注释文档的内建支持，所以源码文件也可以包含它们自己的文档。通过一个单独的程序，这些文档信息可以提取出来，并重新格式化成 HTML。

Java 包含了一些标准库，用于完成特定的任务，同时这些库简单易用，能够大大缩短开发周期，例如，Java 提供了用于访问数据库的 JDBC 库，用于实现分布式对象的 RMI 等标准库。C++ 则依靠一些非标准的、由其他厂商提供的库。

常见笔试题：

下列说法中，错误的有（ ）。

- A. Java 面向对象语言容许单独的过程与函数存在
- B. Java 面向对象语言容许单独的方法存在
- C. Java 语言中的方法属于类中的成员（member）
- D. Java 语言中的方法必定隶属于某一类（对象），调用方法与过程或函数相同

答案：A、B、C。见上面讲解。

4.1.3 为什么需要 `public static void main (String[] args)` 这个方法

`public static void main (String[] args)` 为 Java 程序的入口方法，JVM 在运行程序时，会首先查找 `main()` 方法。其中，`public` 是权限修饰符，表明任何类或对象都可以访问这个方法，`static` 表明 `main()` 方法是一个静态方法，即方法中的代码是存储在静态存储区的，只要类被加载后，就可以使用该方法而不需要通过实例化对象来访问，可以直接通过类名 `.main()` 直接访问，JVM 在启动时就是按照上述方法的签名（必须有 `public` 与 `static` 修饰，返回值为 `void`，且方法的参数为字符串数组）来查找方法的入口地址，若能找到，就执行；找不到，则会报错。`void` 表明方法没有返回值，`main` 是 JVM 识别的特殊方法名，是程序的入口方法。字符串数组参数 `args` 为开发人员在命令行状态下与程序交互提供了一种手段。

因为 `main` 是程序的入口方法，所以当程序运行时，第一个执行的方法就是 `main()` 方法。通常来讲，要执行一个类的方法，先必须实例化一个类的对象，然后通过对象来调用这个方法。但由于 `main` 是程序的入口方法，此时还没有实例化对象，因此在编写 `main()` 方法时就要求不需要实例化对象就可以调用这个方法，鉴于此，`main()` 方法需要被定义成 `public` 与 `static`。下例给出了在调用 `main()` 方法时传递参数的方法。

```
public class Test{
    public static void main(String[] args){
        for(int i=0;i < args.length;i++){
            System.out.println(args[i]);
        }
    }
}
```

在控制台，使用 `javac Test.java` 指令编译上述程序，使用 `java Test arg1 arg2 arg3` 指令运行程序，程序运行结果为：

```
arg1
arg2
arg3
```

引申：

1. `main()` 方法是否还有其他可用的定义格式？

1) 由于 `public` 与 `static` 没有先后顺序关系，因此下面的定义也是合理的。

```
static public void main (String[] args)
```

2) 也可以把 `main()` 方法定义为 `final`。

```
public static final void main (String[] args)
```

3) 也可以用 `synchronized` 来修饰 `main()` 方法。

```
static public synchronized void main (String[] args)
```

不管哪种定义方式，都必须保证 main() 方法的返回值为 void，并有 static 与 public 关键字修饰。同时由于 main() 方法为程序的入口方法，因此不能用 abstract 关键字来修饰。

2. 同一个 .java 文件中是否可以有多个 main() 方法？

虽然每个类中都可以定义 main() 方法，但是只有与文件名相同的用 public 修饰的类中的 main() 方法才能作为整个程序的入口方法。如下例所示，创建了一个名为 Test.java 的文件。

```
class T {
    public static void main(String[] args) {
        System.out.println("T main");
    }
}

public class Test {
    //程序入口函数
    public static void main(String[] args) {
        System.out.println("Test main");
    }
}
```

程序运行结果为：

```
Test main
```

常见笔试题：

Java 程序中程序运行入口方法 main 的签名正确的有 ()。

- A. public static void main (String[] args)
- B. public static final void main (String[] args)
- C. static public void main (String[] args)
- D. static public synchronized void main (String[] args)
- E. static public abstract void main (String[] args)

答案：A、B、C、D。见上面讲解。

4.1.4 如何实现在 main() 方法执行前输出 “Hello World”

众所周知，在 Java 语言中，main() 方法是程序的入口方法，在程序运行时，最先加载的就是 main() 方法，但这是否意味着 main() 方法就是程序运行时第一个被执行的模块呢？

答案是否定的。在 Java 语言中，由于静态块在类被加载时就会被调用，因此可以在 main() 方法执行前，利用静态块实现输出 “Hello World” 的功能，以如下代码为例。

```
public class Test {
    static {
        System.out.println("Hello World1");
    }

    public static void main(String args[]) {
        System.out.println("Hello World2");
    }
}
```

程序运行结果为：

```
Hello World1
Hello World2
```

由于静态块不管顺序如何，都会在 `main()` 方法执行之前执行，因此，以下代码会与上面的代码有同样的输出结果。

```
public class Test {
    public static void main(String args[]) {
        System.out.println("Hello World2");
    }
    static {
        System.out.println("Hello World1");
    }
}
```

4.1.5 Java 程序初始化的顺序是怎样的

在 Java 语言中，当实例化对象时，对象所在类的所有成员变量首先要进行初始化，只有当所有类成员完成初始化后，才会调用对象所在类的构造函数创建对象。

Java 程序的初始化一般遵循 3 个原则（优先级依次递减）：①静态对象（变量）优先于非静态对象（变量）初始化，其中，静态对象（变量）只初始化一次，而非静态对象（变量）可能会初始化多次。②父类优先于子类进行初始化。③按照成员变量的定义顺序进行初始化。即使变量定义散布于方法定义之中，它们依然在任何方法（包括构造函数）被调用之前先初始化。

Java 程序初始化工作可以在许多不同的代码块中来完成（例如静态代码块、构造函数等），它们执行的顺序如下：父类静态变量、父类静态代码块、子类静态变量、子类静态代码块、父类非静态变量、父类非静态代码块、父类构造函数、子类非静态变量、子类非静态代码块、子类构造函数。下面给出一个不同模块初始化时执行顺序的一个例子。

```
class Base {
    static {
        System.out.println("Base static block");
    }
    {
        System.out.println("Base block");
    }
    public Base() {
        System.out.println("Base constructor");
    }
}

public class Derived extends Base {
    static {
        System.out.println("Derived static block");
    }
    {
        System.out.println("Derived block");
    }
    public Derived() {
```

```

        System.out.println("Derived constructor");
    }

    public static void main(String args[]) {
        new Derived();
    }
}

```

程序运行结果为：

```

Base static block
Derived static block
Base block
Base constructor
Derived block
Derived constructor

```

常见笔试题：

下面代码的运行结果是什么？

```

class B extends Object{
    static{
        System.out.println("Load B");
    }
    public B(){
        System.out.println("Create B");
    }
    static{
        System.out.println("Load B");
    }
}

class A extends B{
    static{
        System.out.println("Load A");
    }
    public A(){
        System.out.println("Create A");
    }
}

public class Testclass{
    public static void main(String[] args) {
        new A();
    }
}

```

- A. Load B Create B Load A Create A B. Load B Load A Create B Create A
 C. Load B Create B Create A Load A D. Create B Create A Load B Load A

答案：B。见上面讲解。

4.1.6 Java 中的作用域有哪些

在计算机程序中，声明在不同地方的变量具有不同的作用域，例如局部变量、全局变量

等。在 Java 语言中，作用域是由花括号的位置决定的，它决定了其定义的变量名的可见性与生命周期。

在 Java 语言中，变量的类型主要有 3 种：成员变量、静态变量和局部变量。类的成员变量的作用范围与类的实例化对象的作用范围相同，当类被实例化时，成员变量就会在内存中分配空间并初始化，直到这个被实例化对象的生命周期结束时，成员变量的生命周期才结束。被 `static` 修饰的成员变量被称为静态变量或全局变量，与成员变量不同的是，静态变量不依赖于特定的实例，而是被所有实例所共享，也就是说，只要一个类被加载，JVM 就会给类的静态变量分配存储空间。因此，就可以通过类名和变量名来访问静态变量。局部变量的作用域与可见性为它所在的花括号内。

此外，成员变量也有 4 种作用域，它们的区别见表 4-1。

表 4-1 作用域的对比如

作用域与可见性	当前类	同一 package	子类	其他 package
public	√	√	√	√
private	√	×	×	×
protected	√	√	√	×
default	√	√	×	×

1) `public`。表明该成员变量或方法对所有类或对象都是可见的，所有类或对象都可以直接访问。

2) `private`。表明该成员变量或方法是私有的，只有当前类对其具有访问权限，除此之外的其他类或者对象都没有访问权限。

3) `protected`。表明该成员变量或方法对自己及其子类是可见的，即自己和子类具有权限访问。除此之外的其他类或对象都没有访问权限。

4) `default`。表明该成员变量或方法只有自己和与其位于同一包内的类可见。若父类与子类位于同一个包内，则子类对父类的 `default` 成员变量或方法都有访问权限；若父类与子类位于不同的 package（包）内，则没有访问权限。

需要注意的是，这些修饰符只能修饰成员变量，不能用来修饰局部变量。`private` 与 `protected` 不能用来修饰类（只有 `public`、`abstract` 或 `final` 能用来修饰类）。

常见笔试题：

下列说法中，正确的是（ ）

- A. 实例方法可直接调用超类的实例方法 B. 实例方法可直接调用超类的类方法
C. 实例方法可直接调用其他类的实例方法 D. 实例方法可直接调用本类的类方法

答案：D。当超类的实例方法或类方法为 `private` 时，是不能被子类调用的。同理，当其他类的实例方法为 `private` 时，也不能被直接调用。

4.1.7 一个 Java 文件中是否可以定义多个类

一个 Java 文件中可以定义多个类，但是最多只能有一个类被 `public` 修饰，并且这个类的类名与文件名必须相同，若这个文件中没有 `public` 的类，则文件名随便是一个类的名字即可。需要注意的是，当用 `javac` 指令编译这个 `.java` 文件时，它会给每一个类生成一个对应的 `.class` 文件，示例如下。


```

class Base {
    public void print() {
        System.out.println("Base");
    }
}

public class Derived extends Base {
    public static void main(String[] a) {
        Base c = new Derived();
        c.print();
    }
}

```

使用 `javac Derived.java` 指令编译上述代码，会生成两个字节码文件：`Base.class` 与 `Derived.class`，然后使用 `java Derived` 指令执行代码，此时，控制台的输出结果为：`Base`。

4.1.8 什么是构造函数

构造函数是一种特殊的函数，用来在对象实例化时初始化对象的成员变量。在 Java 语言中，构造函数具有以下特点。

- 1) 构造函数必须与类的名字相同，并且不能有返回值（返回值也不能为 `void`）。
- 2) 每个类可以有多个构造函数。当开发人员没有提供构造函数时，编译器在把源代码编译成字节码的过程中会提供一个没有参数默认的构造函数，但该构造函数不会执行任何代码。如果开发人员提供了构造函数，那么编译器就不会再创建默认的构造函数了。
- 3) 构造函数可以有 0 个、1 个或 1 个以上的参数。
- 4) 构造函数总是伴随着 `new` 操作一起调用，且不能由程序的编写者直接调用，必须要由系统调用。构造函数在对象实例化时会被自动调用，且只运行一次；而普通的方法是在程序执行到它时被调用，且可以被该对象调用多次。
- 5) 构造函数的主要作用是完成对象的初始化工作。
- 6) 构造函数不能被继承，因此，它不能被覆盖，但是构造函数能够被重载，可以使用不同的参数个数或参数类型来定义多个构造函数。
- 7) 子类可以通过 `super` 关键字来显式地调用父类的构造函数，当父类没有提供无参数的构造函数时，子类的构造函数中必须显式地调用父类的构造函数。如果父类提供了无参数的构造函数，此时子类的构造函数就可以不显式地调用父类的构造函数，在这种情况下编译器会默认调用父类提供的无参数的构造函数。当有父类时，在实例化对象时会先执行父类的构造函数，然后执行子类的构造函数。
- 8) 当父类和子类都没有定义构造函数时，编译器会为父类生成一个默认的无参数的构造函数，给子类也生成一个默认的无参数的构造函数。此外，默认构造器的修饰符只跟当前类的修饰符有关（例如，如果一个类被定义为 `public`，那么它的构造函数也是 `public`）。

引申：普通方法是否可以与构造函数有相同的方法名？
可以，示例如下。

```

public class Test {
    public Test() {
        System.out.println("construct");
    }
}

```

```

public void Test() {
    System.out.println("call Test");
}

public static void main(String[] args) {
    Test a = new Test(); //调用构造函数
    a.Test();           //调用 Test 方法
}
}

```

程序运行结果为：

```

construct
call Test

```

常见笔试题：

1. 下列关于构造方法的叙述中，错误的是（ ）。

- A. Java 语言规定构造方法名与类名必须相同
- B. Java 语言规定构造方法没有返回值，但不用 void 声明
- C. Java 语言规定构造方法不可以重载
- D. Java 语言规定构造方法只能通过 new 自动调用

答案：C。可以定义多个构造函数，只要不同的构造函数有不同的参数即可。

2. 下列说法中，正确的是（ ）。

- A. class 中的 constructor 不可省略
- B. constructor 必须与 class 同名，但方法不能与 class 同名
- C. constructor 在一个对象被 new 时执行
- D. 一个 class 只能定义一个 constructor

答案：C。见上面讲解。

4.1.9 为什么 Java 中有些接口没有任何方法

由于 Java 不支持多重继承，即一个类只能有一个父类，为了克服单继承的缺点，Java 语言引入了接口这一概念。接口是抽象方法定义的集合（接口中也可以定义一些常量值），是一种特殊的抽象类。接口中只包含方法的定义，没有方法的实现。接口中的所有方法都是抽象的。接口中成员的作用域修饰符都是 public，接口中的常量值默认使用 public static final 修饰。由于一个类可以实现多个接口，因此通常可以采用实现多个接口的方式来间接达到多重继承的目的。

在 Java 语言中，有些接口内部没有声明任何方法，也就是说，实现这些接口的类不需要重写任何方法，这些没有任何方法声明的接口又被叫做标识接口，标识接口对实现它的类没有任何语义上的要求，它仅仅充当一个标识的作用，用来表明实现它的类属于一个特定的类型。这个标签类似于汽车的标志图标，每当人们看到一个汽车的标志图标时，就能知道这款汽车的品牌。Java 类库中已存在的标识接口有 Cloneable 和 Serializable 等。在使用时会经常用 instanceof 来判断实例对象的类型是否实现了一个给定的标识接口。

下面通过一个例子来详细说明标识接口的作用。例如要开发一款游戏，游戏里面有一个人物专门负责出去寻找有用的材料，假设这个人物只收集矿石和武器，而不会收集垃圾。下面通过标识接口来实现这个功能。

```

import Java.util. ArrayList;
interface Stuff {}
//矿石
interface Ore extends Stuff {}
//武器
interface Weapon extends Stuff {}
//垃圾
interface Rubbish extends Stuff {}
//金矿
class Gold implements Ore {
    public String toString() {
        return "Gold";
    }
}
//铜矿
class Copper implements Ore {
    public String toString() {
        return "Copper";
    }
}
//枪
class Gun implements Weapon {
    public String toString() {
        return "Gun";
    }
}
//榴弹
class Grenade implements Weapon {
    public String toString() {
        return "Grenade";
    }
}
class Stone implements Rubbish {
    public String toString() {
        return "Stone";
    }
}
public class Test {
    public static ArrayList < Stuff > collectStuff( Stuff[] s ) {
        ArrayList < Stuff > al = new ArrayList < Stuff > ( );
        for( int i = 0; i < s. length; i ++ ) {
            if( ! ( s[ i ] instanceof Rubbish ) )
                al. add( s[ i ] );
        }
        return al;
    }
    public static void main( String[] args ) {
        Stuff[] s = { new Gold( ), new Copper( ), new Gun( ), new Grenade( ), new Stone( ) };
    }
}

```

```

ArrayList < Stuff > al = collectStuff(s);
System.out.println("The usefull Stuff collected is:");
for(int i=0;i < al.size();i++)
    System.out.println(al.get(i));
}
}

```

程序运行结果为:

```

The usefull Stuff collected is;
Gold
Copper
Gun
Grenade

```

在上例中, 设计了3个接口: Ore、Weapon 和 Rubbish 分别代表矿石、武器和垃圾, 只要是实现 Ore 或 Weapon 的类, 游戏中的角色都会认为这是有用的材料, 例如 Gold、Copper、Gun、Grenade, 因此会收集; 只要是实现 Rubbish 的类, 都会被认为是无用的东西, 例如 Stone, 因此不会被收集。

常见笔试题:

不能用来修饰 interface 的有 ()

A. private B. public C. protected D. static

答案: A、C、D。见上面讲解。

4.1.10 Java 中的 clone 方法有什么作用

由于指针不仅会给开发人员带来使用上的不便, 而且也是造成程序不稳定的根源, 为了消除 C/C++ 语言的这些缺点, Java 语言取消了指针的概念, 但这只是在 Java 语言中没有明确提供指针的概念与用法, 而实质上每个 new 语句返回的都是一个指针的引用, 只不过在大部分情况下开发人员不需要关心如何去操作这个指针而已。

由于 Java 取消了指针的概念, 因此开发人员在编程中往往忽略了对对象和引用的区别, 示例如下。

```

class Obj {
    public void setStr(String str) {
        this.str = str;
    }
    private String str = "default value";
    public String toString() {
        return str;
    }
}
public class TestRef {
    private Obj aObj = new Obj();
    private int aInt = 0;
    public Obj getAObj() {
        return aObj;
    }
}

```

```

public int getAInt() {
    return aInt;
}

public void changeObj( Obj inObj) {
    inObj. setStr( "changed value");
}

public void changeInt(int inInt) {
    inInt = 1;
}

public static void main(String[] args) {
    TestRef oRef = new TestRef();
    System. out. println( " ***** 引用类型 *****");
    System. out. println( "调用 changeObj()前: " + oRef. getAObj());
    oRef. changeObj(oRef. getAObj());
    System. out. println( "调用 changeObj()后: " + oRef. getAObj());
    System. out. println( " ***** 基本数据类型 *****
*");
    System. out. println( "调用 changeInt()前: " + oRef. getAInt());
    oRef. changeInt(oRef. getAInt());
    System. out. println( "调用 changeInt()后: " + oRef. getAInt());
}

```

程序运行结果为:

```

***** 引用类型 *****
调用 changeObj()前: default value
调用 changeObj()后: changed value
***** 基本数据类型 *****
调用 changeInt()前: 0
调用 changeInt()后: 0

```

上面两个看似类似的方法却有着不同的运行结果，主要原因是 Java 在处理基本数据类型（例如 int、char、double 等）时，都是采用按值传递（传递的是输入参数的复制）的方式执行，除此之外的其他类型都是按引用传递（传递的是对象的一个引用）的方式执行。对象除了在函数调用时是引用传递，在使用“=”赋值时也采用引用传递，示例代码如下。

```

class Obj {
    private int aInt = 0;
    public int getAInt() {
        return aInt;
    }
    public void setAInt(int int1) {
        aInt = int1;
    }
    public void changeInt() {
        this. aInt = 1;
    }
}

public class TestRef{

```



```
public static void main(String[] args) {  
    Obj a = new Obj();  
    b = a;  
    b.changeInt();  
    System.out.println("a:" + a.getAInt());  
    System.out.println("b:" + b.getAInt());  
}  
}
```

程序运行结果为:

```
a:1  
b:1
```

在实际编程中,经常会遇到从某个已有的对象 A 创建出另外一个与 A 具有相同状态的对象 B,并且对 B 的修改不会影响到 A 的情况,例如,Prototype (原型)模式中,就需要 clone 一个对象实例。在 Java 语言中,仅仅通过简单的赋值操作显然无法达到这个目的,而 Java 提供了一个简单有效的 clone()方法来满足这个需求。

Java 中的所有类默认都继承自 Object 类,而 Object 类中提供了一个 clone()方法。这个方法的作用是返回一个 Object 对象的复制。这个复制函数返回的是一个新的对象而不是一个引用。那么怎样使用这个方法呢?以下是使用 clone()方法的步骤。

1) 实现 clone 的类首先需要继承 Cloneable 接口。Cloneable 接口实质上是一个标识接口,没有任何接口方法。

2) 在类中重写 Object 类中的 clone()方法。

3) 在 clone 方法中调用 super.clone()。无论 clone 类的继承结构是什么,super.clone()都会直接或间接调用 java.lang.Object 类的 clone()方法。

4) 把浅复制的引用指向原型对象新的克隆体。

对上面的例子引入 clone 方法如下:

```
class Obj implements Cloneable {  
    private int aInt = 0;  
    public int getAInt() {  
        return aInt;  
    }  
    public void setAInt(int int1) {  
        aInt = int1;  
    }  
    public void changeInt() {  
        this.aInt = 1;  
    }  
    public Object clone() {  
        Object o = null;  
        try {  
            o = (Obj)super.clone();  
        } catch (CloneNotSupportedException e) {  
            e.printStackTrace();  
        }  
        return o;  
    }  
}
```

```

    }
}

public class TestRef{
    public static void main(String[] args) {
        Obj a = new Obj();
        Obj b = (Obj)a.clone();
        b.changeInt();
        System.out.println("a:" + a.getAInt());
        System.out.println("b:" + b.getAInt());
    }
}

```

程序运行结果为:

```

a:0
b:1

```

在 C++ 语言中,当开发人员自定义复制构造函数时,会存在浅复制与深复制之分。Java 语言在重载 clone()方法时也存在同样的问题,当类中只有一些基本的数据类型时,采用上述方法就可以了,但是当类中包含了一些对象时,就需要用到深复制了,实现方法是在对对象调用 clone()方法完成复制后,接着对对象中的非基本类型的属性也调用 clone()方法完成深复制,示例如下。

```

import java.util.Date;
class Obj implements Cloneable{
    private Date birth = new Date();
    public Date getBirth() {
        return birth;
    }
    public void setBirth(Date birth) {
        this.birth = birth;
    }
    public void changeDate() {
        this.birth.setMonth(4);
    }
    public Object clone() {
        Obj o = null;
        try {
            o = (Obj)super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        //实现深复制
        o.birth = (Date)this.getBirth().clone();
        return o;
    }
}

```

```

public class TestRef{
    public static void main(String[] args) {

```

```

Obj a = new Obj();
Obj b = (Obj)a.clone();
b.changeDate();
System.out.println("a = " + a.getBirth());
System.out.println("b = " + b.getBirth());
}
}

```

程序运行结果为：

```

a = Sat Jul 13 23:58:56 CST 2013
b = Mon May 13 23:58:56 CST 2013

```

那么在编程时，如何选择使用哪种复制方式呢？首先，检查类有无非基本类型（即对象）的数据成员。若没有，则返回 `super.clone()` 即可；若有，确保类中包含的所有非基本类型的成员变量都实现了深复制。

```
Object o = super.clone(); //先执行浅复制
```

对每一个对象 `attr` 执行以下语句：

```
o.attr = this.getAttr().clone();
```

最后返回 `o`。

需要注意的是，`clone()` 方法的保护机制在 `Object` 中 `clone()` 是被声明为 `protected` 的。以 `User` 类为例，通过声明为 `protected`，就可以保证只有 `User` 类里面才能“克隆” `User` 对象，原理可以参考前面关于 `public`、`protected`、`private` 的讲解。

引申：浅复制和深复制有什么区别？

浅复制 (Shallow Clone)：被复制对象的所有变量都含有与原来对象相同的值，而所有对其他对象的引用仍然指向原来的对象。换言之，浅复制仅仅复制所考虑的对象，而不复制它所引用的对象。

深复制 (Deep Clone)：被复制对象的所有变量都含有与原来对象相同的值，除去那些引用其他对象的变量。那些引用其他对象的变量将指向被复制的新对象，而不再是原有的那些被引用的对象。换言之，深复制把复制的对象所引用的对象都复制了一遍。

假如定义如下一个类。

```

class Test{
    public int i;
    public StringBuffer s;
}

```

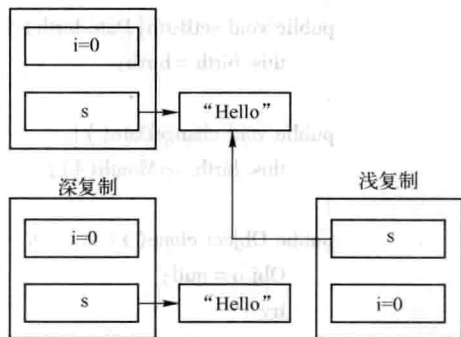


图 4-1 深复制与浅复制的区别

图 4-1 给出了对这个类的对象进行复制时，浅复制与深复制的区别。

4.1.11 什么是反射机制

反射机制是 Java 语言中一个非常重要的特性，它允许程序在运行时进行自我检查，同时也允许对其内部的成员进行操作。虽然这个特性在实际开发时使用得不多，但是像 Pascal、C 和 C++ 等语言根本就没有提供这样的特性。由于反射机制能够实现在运行时对类进行装载，

因此能够增加程序的灵活性，但是不恰当地使用反射机制，也会严重影响系统的性能。

具体而言，反射机制提供的功能主要有：得到一个对象所属的类；获取一个类的所有成员变量和方法；在运行时创建对象；在运行时调用对象的方法。

其实，反射机制非常重要的一个作用就是可以在运行时动态地创建类的对象，示例如下。

```
class Base {
    public void f() {
        System.out.println("Base");
    }
}

class Sub extends Base {
    public void f() {
        System.out.println("Sub");
    }
}

public class Test {
    public static void main(String[] args) {
        try{//使用反射机制加载类
            Class c = Class.forName("Sub");
            Base b = (Base)c.newInstance();
            b.f();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果为：

```
Sub
```

在反射机制中，class 是一个非常重要的类，那么如何才能获取 class 类呢？总共有如下 3 种方法可以获取到 class 类：

- 1) class.forName (“类的路径”)，如上例所示。
- 2) 类名.class。
- 3) 实例.getClass()。

常见笔试题：

Java 创建对象的方式有几种？

答案：共有 4 种创建对象的方法。

- 1) 通过 new 语句实例化一个对象。
- 2) 通过反射机制创建对象，见上述讲解。
- 3) 通过 clone() 方法创建一个对象，见 5.1.10 节。
- 4) 通过反序列化的方式创建对象，见 5.7.5 节。

4.1.12 package 有什么作用

package 的中文意思是“包”，它是一个比较抽象的逻辑概念，其宗旨是把 .java 文件（Java 源文件）、.class 文件（编译后的文件）以及其他 resource 文件（例如.xml 文件、.avi 文

件、.mp3 文件、.txt 文件等)有条理地进行一个组织,以供使用。它类似于 Linux 文件系统,有一个根,从根开始有目录和文件,然后目录中嵌套目录。

具体而言,package 主要有以下两个作用:第一,提供多层命名空间,解决命名冲突,通过使用 package,使得处于不同 package 中的类可以存在相同的名字。第二,对类按功能进行分类,使项目的组织更加清晰。当开发一个有非常多的类的项目时,如果不使用 package 对类进行分类,而是把所有类都放在一个 package 下,这样的代码不仅可读性差,而且可维护性也不好,会严重影响开发效率。

package 的用法一般如下(源文件所在目录为当前目录):

1) 在每个源文件的开头加上"package packagename;",然后源文件所在目录下创建一个新目录,名称为 packagename。

2) 用 javac 指令编译每个 sourcename.java 源文件,将生成的 sourcename.classname 文件复制到 packagename 目录。

3) 用 java 指令运行程序:java packagename.sourcename。

以下是一个简单的程序示例。

```
package com. pkg;
public class TestPackage {
    public static void main(String[] args) {
        System.out.println("Hello world");
    }
}
```

通过运行指令 javac -d . TestPackage.java 编译代码,会在当前目录下自动生成目录 com/pkg,然后通过运行指令 java com. pkg. TestPackage 执行程序,程序运行结果为:

```
Hello world
```

常见笔试题:

下列说法中,正确的是()。

- A. Java 中包的主要作用是实现跨平台功能
- B. package 语句只能放在 import 语句后面
- C. 包(package)由一组类(class)和接口(interface)组成
- D. 可以用#include 关键字来表明来自其他包中的类

答案:C。见上面讲解。

4.1.13 如何实现类似于 C 语言中函数指针的功能

在 C 语言中,有一个非常重要的概念——函数指针,其最重要的功能是实现回调函数。什么是回调函数呢?所谓回调函数,就是指函数先在某处注册,而它将在稍后某个需要的时候被调用。在 Windows 系统中,开发人员想让系统动态链接库(Dynamic Link Library, DLL)调用自己编写的一个方法,于是利用 DLL 当中回调函数的接口来编写程序,通过传递一个函数的指针来被调用,这个过程就称为回调。回调函数一般用于截获消息、获取系统信息或处理异步事件。举个简单例子,程序员何昊编写了一段程序 a,其中预留有回调函数接口,并封装好了该程序。程序员薛鹏要让 a 调用自己的程序 b 中的一个方法,于是,他通过 a 中的接口回调属于自己的程序 b 中的那个方法。

函数指针一般作为函数的参数来使用,开发人员在使用时可以根据自己的需求传递自定义的函数来实现指定的功能,例如,在实现排序算法时,可以通过传递一个函数指针来决定两个数的先后顺序,从而最终决定该算法是按升序还是降序排列。

在 Java 语言中没有指针的概念,那么如何才能能在 Java 语言中实现类似于函数指针的功能呢?可以利用接口与类来实现同样的效果。具体而言,应先定义一个接口,然后在接口中声明要调用的方法,接着实现这个接口,最后把这个实现类的一个对象作为参数传递给调用程序,调用程序通过这个参数来调用指定的函数,从而实现回调函数的功能,示例如下。

```
//接口中定义了一个用来比较大小的方法
```

```
interface IntCompare {
    public int cmp(int a,int b);
}
```

```
class Cmp1 implements IntCompare {
    public int cmp(int a, int b) {
        if(a > b)
            return 1;
        else if (a < b)
            return -1;
        else
            return 0;
    }
}
```

```
class Cmp2 implements IntCompare {
    public int cmp(int a, int b) {
        if(a > b)
            return -1;
        else if (a < b)
            return 1;
        else
            return 0;
    }
}
```

```
public class Test {
    public static void insertSort(int[] a, IntCompare cmp) {
```

```
        if (a != null) {
```

```
            for (int i = 1; i < a.length; i++) {
```

```
                int temp = a[i], j = i;
```

```
                if (cmp.cmp(a[j-1], temp) == 1) {
```

```
                    while (j >= 1 && cmp.cmp(a[j-1], temp) == 1) {
```

```
                        a[j] = a[j-1];
```

```
                        j--;
```

```
                    a[j] = temp;
```

```
        }
    }
    public static void main(String[] args) {
```

```

int[] array1 = {7,3,19,40,4,7,1};
insertSort(array1,new Cmp1());
System.out.println("升序排列:");
for(int i=0;i<array1.length;i++)
    System.out.print(array1[i]+" ");
System.out.println();
int[] array2 = {7,3,19,40,4,7,1};
insertSort(array2,new Cmp2());
System.out.println("降序排列:");
for(int i=0;i<array2.length;i++)
    System.out.print(array2[i]+" ");
}
}

```

程序运行结果为:

```

升序排列:1 3 4 7 7 19 40
降序排列:40 19 7 7 4 3 1

```

上例定义了一个用来比较大小的接口 `IntCompare`，这个接口实际上充当了 C 语言中函数指针的功能，在使用时，开发人员可以根据实际需求传入自定义的类。在上例中分别有两个类 `Cmp1` 和 `Cmp2` 都实现了这个接口，分别用来在实现升序排序和降序排序时使用。其实这也是策略设计模式所用到的思想。

4.2 面向对象技术

4.2.1 面向对象与面向过程有什么区别

面向对象中的对象不是指女朋友，它是一种编程术语。面向对象是当今软件开发方法的主流方法之一，它是把数据及对数据的操作方法放在一起，作为一个相互依存的整体，即对象。对同类对象抽象出其共性，即类，类中的大多数数据，只能被本类的方法进行处理。类通过一个简单的接口与外界发生关系，对象与对象之间通过消息进行通信。程序流程由用户在使用中决定，例如，站在抽象的角度，人具有身高、体重、年龄、血型等一些特征，人会劳动、人都会直立行走、人都会吃饭、人都会用自己的头脑去创造工具等这些方法，人仅仅只是一个抽象的概念，它是不存在的实体，但是所有具备人这个群体的属性与方法的对象都叫人，这个对象人是实际存在的实体，每个人都是人这个群体的一个对象。

而面向过程是一种以事件为中心的开发方法，就是自顶向下顺序执行，逐步求精，其程序结构是按功能划分为若干个基本模块，这些模块形成一个树状结构，各模块之间的关系也比较简单，在功能上相对独立，每一模块内部一般都是由顺序、选择和循环 3 种基本结构组成，其模块化实现的具体方法是使用子程序，而程序流程在写程序时就已经决定。以五子棋为例，面向过程的设计思路就是首先分析问题的步骤：第一步，开始游戏；第二步，黑子先走；第三步，绘制画面；第四步，判断输赢；第五步，轮到白子；第六步，绘制画面；第七步，判断输赢；第八步，返回第二步；第九步，输出最后结果。把上面每个步骤用分别的函数来实现，就是一个面向过程的开发方法。

具体而言，面向对象与面向过程主要有以下几个方面的不同之处。

1) 出发点不同。面向对象方法是用符合常规思维的方式来处理客观世界的问题, 强调把问题域的要领直接映射到对象及对象之间的接口上。而面向过程方法强调的则是过程的抽象化与模块化, 它是以过程为中心构造或处理客观世界问题的。

2) 层次逻辑关系不同。面向对象方法则是用计算机逻辑来模拟客观世界中的物理存在, 以对象的集合类作为处理问题的基本单位, 尽可能地使计算机世界向客观世界靠拢, 以使问题的处理更清晰直接, 面向对象方法是用类的层次结构来体现类之间的继承和发展。而面向过程方法处理问题的基本单位是能清晰准确地表达过程的模块, 用模块的层次结构概括模块或模块间的关系与功能, 把客观世界的问题抽象成计算机可以处理的过程。

3) 数据处理方式与控制程序方式不同。面向对象方法将数据与对应的代码封装成一个整体, 原则上其他对象不能直接修改其数据, 即对象的修改只能由自身的成员函数完成, 控制程序方式上是通过“事件驱动”来激活和运行程序。而面向过程方法是直接通过程序来处理数据, 处理完毕后即可显示处理结果, 在控制程序方式上是按照设计调用或返回程序, 不能自由导航, 各模块之间存在着控制与被控制、调用与被调用的关系。

4) 分析设计与编码转换方式不同。面向对象方法贯穿于软件生命周期的分析、设计及编码中, 是一种平滑过程, 从分析到设计再到编码是采用一致性的模型表示, 即实现的是一种无缝连接。而面向过程方法强调分析、设计及编码之间按规则进行转换, 贯穿于软件生命周期的分析、设计及编码中, 实现的是一种有缝的连接。

4.2.2 面向对象有哪些特征

面向对象的主要特征包括抽象、继承、封装和多态。

1) 抽象。抽象就是忽略一个主题中与当前目标无关的那些方面, 以便更充分地注意与当前目标有关的方面。抽象并不打算了解全部问题, 而只是选择其中的一部分, 暂时不用部分细节。抽象包括两个方面: 一是过程抽象; 二是数据抽象。

2) 继承。继承是一种联结类的层次模型, 并且允许和鼓励类的重用, 它提供了一种明确表述共性的方法。对象的一个新类可以从现有的类中派生, 这个过程称为类继承。新类继承了原始类的特性, 新类称为原始类的派生类(子类), 而原始类称为新类的基类(父类)。派生类可以从它的基类那里继承方法和实例变量, 并且派生类可以修改或增加新的方法使之更适合特殊的需要。

3) 封装。封装是指将客观事物抽象成类, 每个类对自身的数据和方法实行保护。类可以把自己的数据和方法只让可信的类或者对象操作, 对不可信的进行信息隐藏。

4) 多态。多态是指允许不同类的对象对同一消息作出响应。多态包括参数化多态和包含多态。多态性语言具有灵活、抽象、行为共享、代码共享等优势, 很好地解决了应用程序函数同名问题。

4.2.3 面向对象的开发方式有什么优点

采用面向对象的开发方式有诸多的优点, 下面主要介绍其中3个优点。

1) 较高的开发效率。采用面向对象的开发方式, 可以对现实的事物进行抽象, 可以把现实的事物直接映射为开发的对象, 与人类的思维过程相似, 例如可以设计一个 Car 类来表示现实中的汽车, 这种方式非常直观明了, 也非常接近人们的正常思维。同时, 由于面向对象的开发方式可以通过继承或者组合的方式来实现代码的重用, 因此可以大大地提高软件的开发效率。

2) 保证软件的鲁棒性。正是由于面向对象的开发方法有很高的重用性，在开发的过程中可以重用已有的而且在相关领域经过长期测试的代码，因此，自然而然地对软件的鲁棒性起到了良好的促进作用。

3) 保证软件的高可维护性。由于采用面向对象的开发方式，使得代码的可读性非常好，同时面向对象的设计模式也使得代码结构更加清晰明了。同时针对面向对象的开发方式，已有许多非常成熟的设计模式，这些设计模式可以使程序在面对需求的变更时，只需要修改部分的模块就可以满足需求，因此维护起来非常方便。

4.2.4 什么是继承

继承是面向对象中的一个非常重要的特性。通过继承，子类可以使用父类中的一些成员变量与方法，从而能够提高代码的复用性，提高开发效率。在 Java 语言中，被继承的类叫基类 (superclass) 或父类，继承基类或父类的类叫派生类或子类 (subclass)。继承是通过 extends 关键字来实现的，使用格式为：class 子类名 extends 父类名。

继承主要有如下几个特性：

1) Java 语言不支持多重继承，也就是说，子类至多只能有一个父类，但是可以通过实现多个接口来达到多重继承的目的。

2) 子类只能继承父类的非私有 (public 与 protected) 成员变量与方法。

3) 当子类中定义的成员变量和父类中定义的成员变量同名时，子类中的成员变量会覆盖父类的成员变量，而不会继承。

4) 当子类中的方法与父类中的方法有相同的函数签名 (相同的方法名，相同的参数个数与类型) 时，子类将会覆盖父类的方法，而不会继承。

常见笔试题：

下列有关继承的说法中，正确的是 ()。

A. 子类能继承父类的所有方法和状态

B. 子类能继承父类的非私有方法和状态

C. 子类只能继承父类 public 方法和状态

D. 子类能继承父类的方法，而不是状态

答案：B。见上面讲解。

4.2.5 组合和继承有什么区别

组合和继承是面向对象中两种代码复用的方式。组合是指在新类里面创建原有类的对象，重复利用已有类的功能。继承是面向对象的主要特性之一，它允许设计人员根据其他类的实现来定义一个类的实现。组合和继承都允许在新的类中设置子对象 (subobject)，只是组合是显式的，而继承则是隐式的。组合和继承存在着对应关系：组合中的整体类和继承中的子类对应，组合中的局部类和继承中的父类对应。

二者的区别在哪里呢？首先分析一个实例。Car 表示汽车对象，Vehicle 表示交通工具对象，Tire 表示轮胎对象。三者的类关系如图 4-2 所示。

从图 4-2 中可以看出，Car 是 Vehicle 的一种，因此是一种继承关系 (又被称为 “is-a” 关系)；而 Car 包含了多个 Tire，因此是一种组合关系 (又被称为 “has-a” 关系)。其实现方式如下：

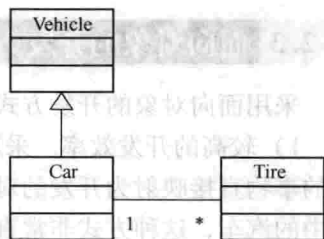


图 4-2 组合与继承对比

继 承	组 合
<pre>class Verhicle { } class Car extends Verhicle { } }</pre>	<pre>class Tire { } class Car extends Verhicle { private Tire t = new Tire (); } }</pre>

既然继承和组合都可以实现代码的重用，那么在实际使用时又该如何选择呢？一般情况下，遵循以下两点原则。

1) 除非两个类之间是“is - a”的关系，否则不要轻易地使用继承，不要单纯地为了实现代码的重用而使用继承，因为过多地使用继承会破坏代码的可维护性，当父类被修改时，会影响到所有继承自它的子类，从而增加程序的维护难度与成本。

2) 不要仅仅为了实现多态而使用继承，如果类之间没有“is - a”的关系，可以通过实现接口与组合的方式来达到相同的目的。设计模式中的策略模式可以很好地说明这一点，采用接口与组合的方式比采用继承的方式具有更好的可扩展性。

由于 Java 语言只支持单继承，如果想同时继承两个类或多个类，在 Java 中是无法直接实现的。同时，在 Java 语言中，如果继承使用太多，也会让一个 class 里面的内容变得臃肿不堪。所以，在 Java 语言中，能使用组合就尽量不要使用继承。

4.2.6 多态的实现机制是什么

多态是面向对象程序设计中代码重用的一个重要机制，它表示当同一个操作作用在不同对象时，会有不同的语义，从而会产生不同的结果，例如，同样是执行“+”操作，“3 + 4”用来实现整数相加，而“3” + “4”却实现了字符串的连接。在 Java 语言中，多态主要有以下两种表现方式：

1) 方法的重载 (overload)。重载是指同一个类中有多个同名的方法，但这些方法有着不同的参数，因此在编译时就可以确定到底调用哪个方法，它是一种编译时多态。重载可以被看作一个类中的方法多态性。

2) 方法的覆盖 (override)。子类可以覆盖父类的方法，因此同样的方法会在父类与子类中有着不同的表现形式。在 Java 语言中，基类的引用变量不仅可以指向基类的实例对象，也可以指向其子类的实例对象。同样，接口的引用变量也可以指向其实现类的实例对象。而程序调用的方法在运行期才动态绑定（绑定指的是将一个方法调用和一个方法主体连接到一起），就是引用变量所指向的具体实例对象的方法，也就是内存里正在运行的那个对象的方法，而不是引用变量的类型中定义的方法。通过这种动态绑定的方法实现了多态。由于只有在运行时才能确定调用哪个方法，因此通过方法覆盖实现的多态也可以被称为运行时多态，示例如下。

```
class Base{
    public Base(){
        g();
    }
    public void f(){
        System.out.println(" Base f()");
    }
    public void g(){
```



```

        System.out.println("Base g()");
    }
}

class Derived extends Base {
    public void f() {
        System.out.println("Derived f()");
    }

    public void g() {
        System.out.println("Derived g()");
    }
}

```

```

public class Test {
    public static void main(String[] args) {
        Base b = new Derived();
        b.f();
        b.g();
    }
}

```

程序运行结果为：

```

Derived g()
Derived f()
Derived g()

```

上例中，由于子类 `Derived` 的 `f()` 方法和 `g()` 方法与父类 `Base` 的方法同名，因此 `Derived` 的方法会覆盖 `Base` 的方法。在执行 `Base b = new Derived()` 语句时，会调用 `Base` 类的构造函数，而在 `Base` 的构造函数中，执行了 `g()` 方法，由于 Java 语言的多态特性，此时会调用子类 `Derived` 的 `g()` 方法，而非父类 `Base` 的 `g()` 方法，因此会输出 `Derived g()`。由于实际创建的是 `Derived` 类的对象，后面的方法调用都会调用子类 `Derived` 的方法。

此外，只有类中的方法才有多态的概念，类中成员变量没有多态的概念，示例如下。

```

class Base {
    public int i = 1;
}

class Derived extends Base {
    public int i = 2;
}

public class Test {
    public static void main(String[] args) {
        Base b = new Derived();
        System.out.println(b.i);
    }
}

```

程序运行结果为：

```
1
```

由此可见，成员变量是无法实现多态的，成员变量的值取父类还是子类并不取决于创建对

象的类型，而是取决于所定义变量的类型，这是在编译期间确定的。在上例中，由于 b 所属的类型为 Base，b.i 指的是 Base 类中定义的 i，因此程序输出结果为 1。

常见笔试题：

Java 中提供了哪两种用于多态的机制？

答案：编译时多态和运行时多态。编译时多态是通过方法的重载实现的，运行时多态是通过方法的覆盖（子类覆盖父类方法）实现的。

4.2.7 重载和覆盖有什么区别

重载 (overload) 和覆盖 (override) 是 Java 多态性的不同表现方式。其中，重载是在一个类中多态性的一种表现，是指在一个类中定义了多个同名的方法，它们或有不同的参数个数或有不同的参数类型。在使用重载时，需要注意以下几点：

1) 重载是通过不同的方法参数来区分的，例如不同的参数个数、不同的参数类型或不同的参数顺序。

2) 不能通过方法的访问权限、返回值类型和抛出的异常类型来进行重载。

3) 对于继承来说，如果基类方法的访问权限为 private，那么就不能在派生类对其重载；如果派生类也定义了一个同名的函数，这只是一个新的方法，不会达到重载的效果。

覆盖是指派生类函数覆盖基类函数。覆盖一个方法并对其重写，以达到不同的作用。在使用覆盖时需要注意以下几点：

1) 派生类中的覆盖方法必须要和基类中被覆盖的方法有相同的函数名和参数。

2) 派生类中的覆盖方法的返回值必须和基类中被覆盖方法的返回值相同。

3) 派生类中的覆盖方法所抛出的异常必须和基类（或是其子类）中被覆盖的方法所抛出的异常一致。

4) 基类中被覆盖的方法不能为 private，否则其子类只是定义了一个方法，并没有对其覆盖。

重载与覆盖的区别主要有以下几个方面：

1) 覆盖是子类 and 父类之间的关系，是垂直关系；重载是同一个类中方法之间的关系，是水平关系。

2) 覆盖只能由一个方法或只能由一对方法产生关系；重载是多个方法之间的关系。

3) 覆盖要求参数列表相同；重载要求参数列表不同。

4) 覆盖关系中，调用方法体是根据对象的类型（对象对应存储空间类型）来决定；而重载关系是根据调用时的实参表与形参表来选择方法体的。

常见笔试题：

如下代码的运行结果是什么？

```
class Super {
    public int f() {
        return 1;
    }
}
public class SubClass extends Super {
    public float f() {
        return 2f;
    }
}
```

```
public static void main(String[] args) {  
    Super s = new SubClass();  
    System.out.println(s.f());  
}
```

答案：编译错误。因为函数是不能以返回值来区分的，虽然父类与子类中的函数有着不同的返回值，但是它们有着相同的函数名，因此，编译器无法区分。

4.2.8 抽象类 (abstract class) 与接口 (interface) 有什么异同

如果一个类中包含抽象方法，那么这个类就是抽象类。在 Java 语言中，可以通过把类或者类中的某些方法声明为 abstract (abstract 只能用来修饰类或者方法，不能用来修饰属性) 来表示一个类是抽象类。接口就是指一个方法的集合，接口中的所有方法都没有方法体，在 Java 语言中，接口是通过关键字 interface 来实现的。

抽象类 (abstract class) 和接口 (interface) 都是支持抽象类定义的两机制 (注意：此句中的前后两个抽象类的意义不一样，前者表示的是一个实体，后者表示的是一个概念)。二者具有很大的相似性，甚至有时候是可以互换的。但同时，二者也存在很大的区别。

只要包含一个抽象方法的类就必须被声明为抽象类，抽象类可以声明方法的存在而不去实现它，被声明为抽象的方法不能包含方法体。在实现时，必须包含相同的或者更低的访问级别 (public→protected→private)。抽象类在使用的过程中不能被实例化，但是可以创建一个对象使其指向具体子类的实例。抽象类的子类为父类中的所有抽象方法提供具体的实现，否则它们也是抽象类。接口可以被看作抽象类的变体。接口中的所有方法都是抽象的，可以通过接口来间接地实现多重继承。接口中的成员变量都是 static final 类型。由于抽象类可以包含部分方法的实现，因此，在一些场合下抽象类比接口存在更多的优势。

接口与抽象类的相同点如下：

- 1) 都不能被实例化。
- 2) 接口的实现类或抽象类的子类都只有实现了接口或抽象类中的方法后才能被实例化。

接口与抽象类的不同点如下：

- 1) 接口只有定义，其方法不能在接口中实现，只有实现接口的类才能实现接口中定义的方法，而抽象类可以有定义与实现，即其方法可以在抽象类中被实现。
- 2) 接口需要实现 (用 implements)，但抽象类只能被继承 (用 extends)。一个类可以实现多个接口，但一个类只能继承一个抽象类，因此使用接口可以间接地达到多重继承的目的。

3) 接口强调特定功能的实现，其设计理念是“has - a”关系；而抽象类强调所属关系，其设计理念为“is - a”关系。

4) 接口中定义的成员变量默认为 public static final，只能够有静态的不能被修改的数据成员，而且，必须给其赋初值，其所有成员方法都是 public、abstract 的，而且只能被这两个关键字修饰。而抽象类可以有自己的数据成员变量，也可以有非抽象的成员方法，而且，抽象类中的成员变量默认为 default (本包可见)，当然也可以被定义为 private、protected 和 public，这些成员变量可以在子类中被重新定义，也可以被重新赋值，抽象类中的抽象方法 (其前有 abstract 修饰) 不能用 private、static、synchronized、native 等访问修饰符修饰，同时方法必须以分号结尾，并且不带花括号。所以，当功能需要累积时，用抽象类；不需要累积时，用接口。

5) 接口被运用于实现比较常用的功能, 便于日后维护或者添加删除方法; 而抽象类更倾向于充当公共类的角色, 不适用于日后重新对里面的代码进行修改。

简单点说, 接口是一种特殊形式的抽象类, 使用接口完全有可能实现与抽象类相同的操作, 但一般而言, 抽象类多用于在同类事物中有无法具体描述的方法的场景, 所以当子类 and 父类之间存在有逻辑上的层次结构时, 推荐使用抽象类; 而接口多用于不同类之间, 定义不同类之间的通信规则, 所以当希望支持差别较大的两个或者更多对象之间的特定交互行为时, 应该使用接口。

此外, 接口可以继承接口, 抽象类可以实现接口, 抽象类也可以继承具体类。抽象类也可以有静态的 main 方法。

常见笔试题:

1. 下列关于接口的定义中, 正确的是 ()。

- A. void methoda(); B. public double methoda();
 C. public final double methoda(); D. static void methoda (double d1);
 E. protected void methoda (double d1); F. int a;
 G. int b = 1;

答案: A、B、G。从上面的分析可知, 接口中的方法只能用关键字 public 和 abstract 来修饰, 因此选项 C、D、E 都是错误的。接口中的属性默认都为 public static final, 由于属性被 final 修饰, 因此它是常量, 常量在定义时必须初始化, 因此 F 是错误的。

2. 下列说法中, 正确的是 ()。

- A. 声明抽象方法大括号可有可无 B. 声明抽象方法不可写出大括号
 C. 抽象方法有方法体 D. abstract 可修饰属性、方法和类

答案: B。抽象方法不能有方法体, 同理也就不能有括号。abstract 只能用来修饰类与方法, 不能用来修饰属性。

4.2.9 内部类有哪些

在 Java 语言中, 可以把一个类定义到另外一个类的内部, 在类里面的这个类就叫做内部类, 外面的类叫做外部类。在这种情况下, 这个内部类可以被看作外部类的一个成员 (与类的属性和方法类似)。还有一种类被称为顶层 (top-level) 类, 指的是类定义代码不嵌套在其他类定义中的类。

需要注意的是, 嵌套类 (Nested Class) 与内部类 (Inner Class) 类似, 只是嵌套类是 C++ 的说法, 而内部类是 Java 的说法而已。内部类可以分为很多种, 主要有以下 4 种: 静态内部类 (static inner class)、成员内部类 (member inner class)、局部内部类 (local inner class) 和匿名内部类 (anonymous inner class)。它们的定义方法如下。

```
class outerClass {
    static class innerClass {}           //静态内部类
class outerClass {
    class innerClass {}                 //成员内部类(普通内部类)
class outerClass {
    public void memberFunction() {
        class innerClass {}           //局部内部类
    }
}
```

```

public class MyFrame extends Frame { //外部类
    public MyFrame() {
        addWindowListener(new WindowAdapter() { //匿名内部类
            public void windowClosing(WindowEvent e) {
                dispose();
                System.exit(0);
            }
        });
    }
}

```

静态内部类是指被声明为 `static` 的内部类，它可以不依赖于外部类实例而被实例化，而通常的内部类需要在外部类实例化后才能实例化。静态内部类不能与外部类有相同的名字，不能访问外部类的普通成员变量，只能访问外部类中的静态成员和静态方法（包括私有类型）。

一个静态内部类，如果去掉“`static`”关键字，就成为成员内部类。成员内部类为非静态内部类，它可以自由地引用外部类的属性和方法，无论这些属性和方法是静态的还是非静态的。但是它与一个实例绑定在了一起，不可以定义静态的属性和方法。只有在外部的类被实例化后，这个内部类才能被实例化。需要注意的是，非静态内部类中不能有静态成员。

局部内部类指的是定义在一个代码块内的类，它的作用范围为其所在的代码块，是内部类中最少使用到的一种类型。局部内部类像局部变量一样，不能被 `public`、`protected`、`private` 以及 `static` 修饰，只能访问方法中定义为 `final` 类型的局部变量。对一个静态内部类，去掉其声明中的“`static`”关键字，将其定义移入其外部类的静态方法或静态初始化代码段中就成为了局部静态内部类。对一个成员类，将其定义移入其外部类的实例方法或实例初始化代码中就成为了局部内部类。局部静态内部类与静态内部类的基本特性相同。局部内部类与内部类的基本特性相同。

匿名内部类是一种没有类名的内部类，不使用关键字 `class`、`extends`、`implements`，没有构造函数，它必须继承（`extends`）其他类或实现其他接口。匿名内部类的好处是代码更加简洁、紧凑，但带来的问题是易读性下降。它一般应用于 GUI（Graphical User Interface，图形用户界面）编程中实现事件处理等。在使用匿名内部类时，需要牢记以下几个原则：

- 1) 匿名内部类不能有构造函数。
- 2) 匿名内部类不能定义静态成员、方法和类。
- 3) 匿名内部类不能是 `public`、`protected`、`private`、`static`。
- 4) 只能创建匿名内部类的一个实例。
- 5) 一个匿名内部类一定是在 `new` 的后面，这个匿名类必须继承一个父类或实现一个接口。
- 6) 因为匿名内部类为局部内部类，所以局部内部类的所有限制都对其生效。

常见面试题：

定义如下一个外部类。

```

public class OuterClass {
    private int d1 = 1;
    //编写内部类
}

```

先需要在这个外部类中定义一个内部类，下面哪个定义是正确的？（ ）

- A. `class InnerClass {
public static int methoda () {return d1;}`
- B. `public class InnerClass {
static int methoda () {return d1;}`
- C. `private class InnerClass {
int methoda () {return d1;}`
- D. `static class InnerClass {
protected int methoda () {return d1;}`
- E. `abstract class InnerClass {
public abstract int methoda ();`

答案：C、E。由于在非静态内部类中不能定义静态成员，因此 A 和 B 是错误的。由于静态内部类不能访问外部类的非静态成员，因此 D 是错误的。

4.2.10 如何获取父类的类名

Java 语言提供了获取类名的方法：`getClass().getName()`，开发人员可以调用这个方法来获取类名，代码如下（示例 1）所示：

```
public class Test {
    public void test() {
        System.out.println(this.getClass().getName());
    }
    public static void main(String[] args) {
        new Test().test();
    }
}
```

程序运行结果为：

```
Test
```

通过以上这个例子的运行结果是否可以得出一个结论：通过调用父类的 `getClass().getName()` 方法来获取父类的类名是可行的呢？为了解答这个问题，首先来做一个实验，给出下面的程序（示例 2）。

```
class A {}
public class Test extends A {
    public void test() {
        System.out.println(super.getClass().getName());
    }
    public static void main(String[] args) {
        new Test().test();
    }
}
```

程序运行结果为：

Test

为什么输出的结果不是“A”而是“Test”呢？主要原因在于Java语言中任何类都继承自Object类，getClass()方法在Object类中被定义为final与native，子类不能覆盖该方法。因此this.getClass()和super.getClass()最终都调用的是Object中的getClass()方法。而Object的getClass()方法的释义是：返回此Object的运行类。由于在示例2中实际运行的类是Test而不是A，因此程序输出结果为Test。那么如何才能子类中得到父类的名字呢？可以通过Java的反射机制，使用getClass().getSuperclass().getName()，代码如下（示例3）所示：

```
class A {}
public class Test extends A {
    public void test() {
        System.out.println(this.getClass().getSuperclass().getName());
    }
    public static void main(String[] args) {
        new Test().test();
    }
}
```

程序运行结果为：

A

4.2.11 this 与 super 有什么区别

在Java语言中，this用来指向当前实例对象，它的一个非常重要的作用就是用来区分对象的成员变量与方法的形参（当一个方法的形参与成员变量的名字相同时，就会覆盖成员变量）。为了能够对this有一个更好的认识，首先创建一个类People，示例如下：

```
class People {
    String name;
    //正确的写法
    public People(String name) {
        this.name = name;
    }
    //错误的写法
    public People(String name) {
        name = name;
    }
}
```

上例中，第一个构造函数使用this.name来表示左边的值为成员变量，而不是这个构造函数的形式参数。对于第二个构造函数，由于在这个函数中形参与成员变量有着相同的名字，因此对于语句name = name，等号左边和右边的两个name都代表的是形式参数。在这种情况下，只有通过this才能访问到成员变量。

super可以用来访问父类的方法或成员变量。当子类的方法或成员变量与父类有相同名字时也会覆盖父类的方法或成员变量，要想访问父类的方法或成员变量只能通过super关键字来

访问，示例如下：

```
class Base {
    public void f() {
        System.out.println("Base:f()");
    }
}

class Sub extends Base {
    public void f() {
        System.out.println("Sub:f()");
    }

    public void subf() {
        f();
    }

    public void basef() {
        super.f();
    }
}

public class Test {
    public static void main(String[] args) {
        Sub s = new Sub();
        s.subf();
        s.basef();
    }
}
```

程序运行结果为：

```
Sub:f()
Base:f()
```

常见笔试题：

下面程序的运行结果是什么？

```
class Base {
    public Base() {
        System.out.println("Base");
    }
}

class Sub extends Base {
    public Sub() {
        System.out.println("Sub");
        super();
    }
}

public class Test {
    public static void main(String[] args) {
        Base s = new Sub();
    }
}
```

答案：编译错误。当子类构造函数需要显示调用父类构造函数时，`super()`必须为构造函数中的第一条语句，因此正确的写法如下：

```
public Sub() {
    super();
    System.out.println("Sub");
}
```

4.3 关键字

4.3.1 变量命名有哪些规则

在 Java 语言中，变量名、函数名、数组名统称为标识符，Java 语言规定标识符只能由字母（`a ~ z`，`A ~ Z`）、数字（`0 ~ 9`）、下划线（`_`）和 `$` 组成，并且标识符的第一个字符必须是字母、下划线或 `$`。此外，标识符也不能包含空白字符（换行符、空格和制表符）。

以下标识符都是非法的。

`char`：`char` 是 Java 语言的一个数据类型，是保留字，不能作为标识符，其他如 `int`、`float` 等与之类似。

`number of book`：标识符中不能有空格。

`3com`：标识符不能以数字开头。

`a * b`：`*` 不能作为标识符的字符。

值得注意的是，在 Java 语言中，变量名是区分大小写的，例如 `Count` 与 `count` 被认为是两个不同的标识符，而非相同的标识符。

常见笔试题：

1. 下列不属于 Java 标识符的是（ ）。

- A. `_HelloWorld`
- B. `3HelloWorld`
- C. `$ HelloWorld`
- D. `HelloWorld3`

答案：B。见上面讲解。

2. 下列标识符不合法的有（ ）。

- A. `new`
- B. `$ usdollars`
- C. `1234`
- D. `car. taxi`

答案：A、C、D。见上面讲解。

4.3.2 `break`、`continue` 以及 `return` 有什么区别

`break`、`continue` 以及 `return` 的区别如下：

1) `break` 用于直接强行跳出当前循环，不再执行剩余代码。当循环中遇到 `break` 语句时，忽略循环体中任何其他语句和循环条件测试，程序控制在循环体后面的语句重新开始。所以，当多层循环嵌套，并且 `break` 语句出现在嵌套循环中的内层循环时，它将仅仅是终止了内层循环的执行，而不影响外层循环的执行。

2) `continue` 用于停止当次循环，回到循环起始处，进入下一次循环操作。`continue` 语句之后的语句将不再执行，用于跳过循环体中的一部分语句，也就是不执行这部分语句，而不是跳出整个循环执行下一条语句，这就是 `continue` 与 `break` 的主要区别。简单来说，`continue` 只是

中断一次循环的执行而已。

3) return 语句是一个跳转语句,用来表示从一个方法返回(返回一个值或其他复杂类型),可以使程序控制返回到调用该方法的地方。当执行 main 方法时,return 语句可以使程序执行返回到 Java 运行系统。

break 只能跳出当前的循环,那么如何才能跳出多重循环呢?可以在多重循环的外面定义一个标识,然后在循环体里使用带有标识的 break 语句,这样即可跳出多重循环,示例如下:

```
public class Break {
    public static void main(String[] args) {
        out:
        for(int i=0;i<5;i++){
            for(int j=0;j<5;j++){
                if(j == 2)
                    break out;
                System.out.println(j);
            }
            System.out.println("break");
        }
    }
}
```

程序运行结果为:

0

1

break

上例中,当内部循环执行到 j 等于 2 时,程序跳出双重循环,执行 System.out.println("break") 语句。

引申:Java 语言中是否存在 goto 关键字?

虽然关键字 goto 作为 Java 的保留字,但目前并没有在 Java 中使用。在 C/C++ 中,goto 常被用于跳出多重循环,而在 Java 语言中,可以使用 break 和 continue 来达到同样的效果。那么,既然 goto 没有在 Java 语言中使用,为什么还要把它作为保留字呢?其中一个原因就是在这个关键字有可能会在将来被使用。如果现在不把 goto 作为保留字,开发人员就有可能用 goto 作为变量名来使用。一旦有一天 Java 支持 goto 关键字了,这会导致以前的程序无法正常运行,因此把 goto 作为保留字是非常有必要的。

这里需要注意的是,在 Java 语言中,虽然没有 goto 语句,但是却能使用标识符加冒号(:)的形式定义标签,如“mylabel:”,这主要是为了在多重循环中方便使用 break 和 continue 而设计的。

4.3.3 final、finally 和 finalize 有什么区别

final、finally 和 finalize 的区别如下:

1) final 用于声明属性、方法和类,分别表示属性不可变、方法不可覆盖和类不可被继承(不能再派生出新的子类)。

final 属性:被 final 修饰的变量不可变。由于不可变有两重含义:一是引用不可变;二是

对象不可变。那么，final 到底指的是哪种含义呢？下面通过一个例子来进行说明。

<pre>public class Test { public static void main (String [] arg) { final StringBuffer s = new StringBuffer (" Hello"); s.append (" world"); System.out.println (s); } }</pre>	<pre>public class Test { public static void main (String [] arg) { final StringBuffer s = new StringBuffer (" Hel- lo"); s = new StringBuffer (" Hello world"); } }</pre>
运行结果为：Hello world	运行结果为：编译期间错误

从以上的例子中可以看出，final 指的是引用的不可变性，即它只能指向初始时指向的那个对象，而不关心指向对象内容的变化。所以，被 final 修饰的变量必须被初始化。一般可以通过以下几种方式对其进行初始化：①在定义的时候初始化。②final 成员变量可以在初始化块中初始化，但不可在静态初始化块中初始化。③静态 final 成员变量可以在静态初始化块中初始化，但不可在初始化块中初始化。④在类的构造器中初始化，但静态 final 成员变量不可在构造函数中初始化。

final 方法：当一个方法声明为 final 时，该方法不允许任何子类重写这个方法，但子类仍然可以使用这个方法。另外，还有一种被称为 inline（内联）的机制，当调用一个被声明为 final 的方法时，直接将方法主体插入到调用处，而不是进行方法调用（类似于 C++ 中的 inline），这样做能提高程序的效率。

final 参数：用来表示这个参数在这个函数内部不允许被修改。

final 类：当一个类被声明为 final 时，此类不能被继承，所有方法都不能被重写。但这并不表示 final 类的成员变量也是不可改变的，要想做到 final 类的成员变量不可改变，必须给成员变量增加 final 修饰。值得注意的是，一个类不能既被声明为 abstract，又被声明为 final。

2) finally 作为异常处理的一部分，它只能用在 try/catch 语句中，并且附带一个语句块，表示这段语句最终一定被执行，经常被用在需要释放资源的情况下。

示例 1：不使用 finally 的代码如下所示：

```
Connection conn;
Statement stmt;
try {
    conn = DriverManager.getConnection(url1, userName, password);
    stmt = con.createStatement();
    stmt.executeUpdate(update); //执行一条 update 语句,此时出现异常
    stmt.close();
    conn.close();
} catch (Exception e) {
}
```

在上面的程序片段中，如果程序在运行过程中没有发生异常，那么数据库的连接能够得到释放，程序运行没有问题。如果在执行 update 语句时出现异常，后面的 close() 方法将不会被调用，数据库的连接将得不到释放。如果这样的程序长期运行，将会耗光数据库的连接资源。通过使用 finally 可以保证任何情况下数据库的连接资源都能够被释放。

示例 2：使用 finally 的代码如下所示。

```

Connection conn = null;
Statement stmt = null;
try {
    conn = DriverManager.getConnection(url1, userName, password);
    stmt = con. createStatement();
    stmt. executeUpdate(update); //执行一条 update 语句,此时出现异常
    stmt. close();
    conn. close();
} catch (Exception e) finally{
    if(stmt != NULL)
        stmt. close();
    if(conn != NULL)
        conn. close();
}

```

在示例 2 中，不管程序运行是否会出现异常，finally 中的代码一定会执行，这样能够保证在任何情况下数据库的连接资源都能被释放。

3) finalize 是 Object 类的一个方法，在垃圾回收器执行时会调用被回收对象的 finalize() 方法，可以覆盖此方法来实现对其他资源的回收，例如关闭文件等。需要注意的是，一旦垃圾回收器准备好释放对象占用的空间，将首先调用其 finalize() 方法，并且在下一次垃圾回收动作发生时，才会真正回收对象占用的内存。

常见笔试题：

JDK 中哪些类是不能继承的？

答案：从上面的介绍可以知道，不能继承的类是那些用 final 关键字修饰的类。一般比较基本的类型或防止扩展类无意间破坏原来方法的实现的类型都应该是 final 的，在 JDK 中，String、StringBuffer 等都是基本类型，所以，String、StringBuffer 等类是不能继承的。

4.3.4 assert 有什么作用

断言 (assert) 作为一种软件调试的方法，提供了一种在代码中进行正确性检查的机制，目前很多开发语言都支持这种机制。它的主要作用是对一个 boolean 表达式进行检查，一个正确运行的程序必须保证这个 boolean 表达式的值为 true，若 boolean 表达式的值为 false，则说明程序已经处于一种不正确的状态下，系统需要提供告警信息并且退出程序。在实际的开发中，assert 主要用来保证程序的正确性，通常在程序开发和测试时使用。为了提高程序运行的效率，在软件发布后，assert 检查默认是被关闭的。

assert 包括两种表达式，分别为 assert expression1 与 assert expression1 : expression2，其中，expression1 表示一个 boolean 表达式，expression2 表示一个基本类型或者是一个对象，基本类型包括 boolean、char、double、float、int 和 long。以下是对这两个表达式的应用。

```

public class Test {
    public static void main(String[] args) {
        assert 1 + 1 == 2;
        System. out. println(" assert1 ok");
        assert 1 + 1 == 3 : " assert faild ,exit";
        System. out. println(" assert2 ok");
    }
}

```


对于上述代码，当执行指令 `javac Test.java` 与 `java Test` 时，程序的输出结果为：

```
assert1 ok
assert2 ok
```

对于上述代码，当执行指令 `javac Test.java` 和 `java -ea Test` 时（注意：`Java -ea Test` 的意思是打开 `-ea` 开关），程序的输出结果为：

```
assert1 ok
Exception in thread "main" Java.lang.AssertionError: assert failed,exit
at Test.main(Test.java:5)
```

`assert` 的应用范围很多，主要包括① 检查控制流；② 检查输入参数是否有效；③ 检查函数结果是否有效；④ 检查程序不变量。虽然 `assert` 的功能与 `if` 判断类似，但二者存在着本质的区别：`assert` 一般在调试程序时使用，但如果不小心用 `assert` 来控制了程序的业务流程，那在调试结束后去掉 `assert` 就意味着修改了程序的正常逻辑，这样的做法是非常危险的；而 `if` 判断是逻辑判断，本身就是用以控制程序流程的。

需要注意的是，在 Java 语言中，`assert` 与 C 语言中的 `assert` 尽管功能类似，但也不完全一样，具体表现为两个方面的不同：① Java 语言中是使用 `assert` 关键字去实现其功能，而 C 语言中使用的是库函数；② C 语言中的 `assert` 是在编译时开启，而 Java 语言中则是在运行时开启。

4.3.5 static 关键字有哪些作用

`static` 关键字主要有两种作用：第一，为某特定数据类型或对象分配单一的存储空间，而与创建对象的个数无关。第二，实现某个方法或属性与类而不是对象关联在一起，也就是说，在不创建对象的情况下就可以通过类来直接调用方法或使用类的属性。具体而言，在 Java 语言中，`static` 主要有 4 种使用情况：成员变量、成员方法、代码块和内部类。

(1) static 成员变量

虽然 Java 语言中没有全局的概念，但可以通过 `static` 关键字来达到全局的效果。Java 类提供了两种类型的变量：用 `static` 关键字修饰的静态变量和不用 `static` 关键字修饰的实例变量。静态变量属于类，在内存中只有一个复制（所有实例都指向同一个内存地址），只要静态变量所在的类被加载，这个静态变量就会被分配空间，因此就可以被使用了。对静态变量的引用有两种方式，分别为“类.静态变量”和“对象.静态变量”。

实例变量属于对象，只有对象被创建后，实例变量才会被分配空间，才能被使用，它在内存中存在多个复制。只能用“对象.实例变量”的方式来引用。以下是静态变量与实例变量的使用示例。

```
public class TestAttribute {
    public static int staticInt = 0;
    public int nonStaticInt = 0;
    public static void main(String[] args) {
        TestAttribute t = new TestAttribute();
        System.out.println("t.staticInt = " + t.staticInt);
        System.out.println("TestAttribute.staticInt = " + TestAttribute.staticInt);
        System.out.println("t.nonStaticInt = " + t.nonStaticInt);
    }
}
```

```

System.out.println("对静态变量和实例变量分别 +1");
t.staticInt++;
t.nonStaticInt++;
TestAttribute t1 = new TestAttribute();
System.out.println("t1.staticInt = " + t1.staticInt);
System.out.println("TestAttribute.staticInt = " + TestAttribute.staticInt);
System.out.println("t1.nonStaticInt = " + t1.nonStaticInt);
}

```

程序运行结果为:

```

t.staticInt = 0
TestAttribute.staticInt = 0
t.nonStaticInt = 0
对静态变量和实例变量分别 + 1
t1.staticInt = 1
TestAttribute.staticInt = 1
t1.nonStaticInt = 0

```

从上例可以看出,静态变量只有一个,被类拥有,所有对象都共享这个静态变量,而实例对象是与具体对象相关的。需要注意的是,与 C++ 语言不同的是,在 Java 语言中,不能在方法体中定义 static 变量。

(2) static 成员方法

与变量类似,Java 类同时也提供了 static 方法与非 static 方法。static 方法是类的方法,不需要创建对象就可以被调用,而非 static 方法是对象的方法,只有对象被创建出来后才可以被使用。

static 方法中不能使用 this 和 super 关键字,不能调用非 static 方法,只能访问所属类的静态成员变量和成员方法,因为当 static 方法被调用时,这个类的对象可能还没被创建,即使已经被创建了,也无法确定调用哪个对象的方法。同理,static 方法也不能访问非 static 类型的变量。

static 一个很重要的用途是实现单例模式。单例模式的特点是该类只能有一个实例,为了实现这一功能,必须隐藏类的构造函数,即把构造函数声明为 private,并提供一个创建对象的方法,由于构造对象被声明为 private,外界无法直接创建这个类型的对象,只能通过该类提供的方法来获取类的对象,要达到这样的目的只能把创建对象的方法声明为 static,程序示例如下:

```

class Singleton {
    private static Singleton instance = null;
    private Singleton () {}
    public static Singleton getInstance() {
        if( instance == null ) {
            instance = new Singleton ();
        }
        return instance;
    }
}

```

用 `public` 修饰的 `static` 变量和方法本质上都是全局的，若在 `static` 变量前用 `private` 修饰，则表示这个变量可以在类的静态代码块或者类的其他静态成员方法中使用，但是不能在其他类中通过类名来直接引用。

(3) `static` 代码块

`static` 代码块（静态代码块）在类中是独立于成员变量和成员函数的代码块的。它不在任何一个方法体内，JVM 在加载类时会执行 `static` 代码块，如果有多个 `static` 代码块，JVM 将会按顺序来执行。`static` 代码块经常被用来初始化静态变量。需要注意的是，这些 `static` 代码块只会被执行一次，示例如下：

```
public class Test {
    private static int a;
    static {
        Test.a = 4;
        System.out.println(a);
        System.out.println("static block is called");
    }
    public static void main(String[] args) {
    }
}
```

程序运行结果为：

```
4
static block is called
```

(4) `static` 内部类

`static` 内部类是指被声明为 `static` 的内部类，它可以不依赖于外部类实例对象而被实例化，而通常的内部类需要在外部类实例化后才能实例化。静态内部类不能与外部类有相同的名字，不能访问外部类的普通成员变量，只能访问外部类中的静态成员和静态方法（包括私有类型），示例如下：

```
public class Outer {
    static int n = 5;
    static class Inner {
        void accessAttrFromOuter() {
            System.out.println("Inner:Outer. n = " + n);
        }
    }
    public static void main(String[] args) {
        Outer.Inner nest = new Outer.Inner();
        nest.accessAttrFromOuter();
    }
}
```

程序运行结果为：

```
Inner:Outer. n = 5
```

需要注意的是，只有内部类才能被定义为 `static`。

引申:

1. 什么是实例变量? 什么是局部变量? 什么是类变量? 什么是 final 变量?

实例变量: 变量归对象所有 (只有在实例化对象后才可以)。每当实例化一个对象时, 会创建一个副本并初始化, 如果没有显示初始化, 那么会初始化一个默认值。各个对象中的实例变量互不影响。

局部变量: 在方法中定义的变量, 在使用前必须初始化。

类变量: 用 static 可修饰的属性、变量归类所有, 只要类被加载, 这个变量就可以被使用 (类名. 变量名)。所有实例化的对象共享类变量。

final 变量: 表示这个变量为常量, 不能被修改。

2. static 与 final 结合使用表示什么意思?

在 Java 语言中, static 关键字常与 final 关键字结合使用, 用来修饰成员变量与成员方法, 有点类似于 C/C++ 语言中的“全局常量”。对于变量, 若使用 static final 修饰, 则表示一旦赋值, 就不可修改, 并且通过类名可以访问。对于方法, 若使用 static final 修饰, 则表示该方法不可覆盖, 并且可以通过类名直接访问。

常见笔试题:

```
public class Test {
    public static int testStatic() {
        static final int i = 0;
        System.out.println(i++);
    }

    public static void main(String args[]) {
        Test test = new Test();
        test.testStatic();
    }
}
```

上述程序的运行结果是什么? ()

A. 0 B. 1 C. 2 D. 编译失败

答案: D。在 Java 语言中, 不能在成员函数内部定义 static 变量。

4.3.6 使用 switch 时有哪些注意事项

switch 语句用于多分支选择, 在使用 switch (expr) 时, expr 只能是一个枚举常量 (内部也是由整型或字符类型实现) 或一个整数表达式, 其中整数表达式可以是基本类型 int 或其对应的包装类 Integer, 当然也包括不同的长度整型, 例如 short。由于 byte、short 和 char 类型的值都能够被隐式地转换为 int 类型, 因此这些类型以及它们对应的包装类型都可以作为 switch 的表达式。但是, long、float、double、String 类型不能够隐式地转换为 int 类型, 因此它们不能被用作 switch 的表达式。如果一定要使用 long、float 或 double 作为 switch 的参数, 必须将其强制转换为 int 型才可以, 例如, 以下对 switch 中参数的使用就是非法的。

```
float a = 0.123;
switch(a) // 错误! a 不是整型或字符类型变量。
```

另外, 与 switch 对应的是 case 语句, case 语句之后可以是直接的常量数值, 例如 1、2、

也可以是一个常量计算式，例如 $1 + 2$ 等，还可以是 final 型的变量（final 变量必须是编译时的常量），例如 `final int a = 0`，但不能是变量或带有变量的表达式，例如 $i * 2$ 等。当然更不能是浮点型数，例如 1.1 或者 $1.2/2$ 等。

```
switch(formWay)
{
    case 2 - 1 : //正确
    case a - 2 : //错误
    ...
    case 2.0 : //错误
    ...
}
```

随着 Java 语言的发展，在 Java 7 中，switch 开始支持 String 类型了。以下是一段支持 String 类型的示例代码：

```
public class Test {
    public void test(String str) {
        switch(str) {
            case "hehao":
                System.out.println("hehao");
                break;
            case "xuepeng":
                System.out.println("xuepeng");
                break;
            case "yexiangyang":
                System.out.println("yexiangyang");
                break;
            default:
                System.out.println("default");
        }
    }
}
```

从本质上来讲，switch 对字符串的支持，其实是 int 类型值的匹配。它的实现原理如下：通过对 case 后面的 String 对象调用 hashCode() 方法，得到一个 int 类型的 hash 值，然后用这个 hash 值来唯一标识这个 case。那么当匹配时，首先调用这个字符串 hashCode() 函数，获取一个 hash 值（int 类型），用这个 hash 值来匹配所有 case，如果没有匹配成功，说明不存在；如果匹配成功了，接着会调用字符串的 String.equals() 方法进行匹配（至于为什么需要调用 equals() 方法，请参照 5.5.2 节内容）。由此可以看出，String 变量不能为 null，同时，switch 的 case 子句中使用的字符串也不能为 null。

在使用 switch 时，需要注意另外一个问题：一般必须在 case 语句结尾添加 break 语句。因为一旦通过 switch 语句确定了入口点，就会顺序执行后面的代码，直到遇到关键字 break。否则，会执行满足这个 case 之后的其他 case 的语句而不管 case 是否匹配，直到 switch 结束或者遇到 break 为止。如果在 switch 中省略了 break 语句，那么匹配的 case 值后的所有情况（包括 default 情况）都会被执行，示例如下：

```

public class Test {
    public static void main(String[] args) {
        int x = 4;
        switch (x) {
            case 1: System.out.println(x);
            case 2: System.out.println(x);
            case 3: System.out.println(x);
            case 4: System.out.println(x);
            case 5: System.out.println(x);
            default: System.out.println(x);
        }
    }
}

```

程序运行结果为：

```

4
4
4

```

4.3.7 volatile 有什么作用

在用 Java 语言编写的程序中，有时为了提高程序的运行效率，编译器会自动对其进行优化，把经常被访问的变量缓存起来，程序在读取这个变量时有可能会直接从缓存（例如寄存器）中来读取这个值，而不会去内存中读取。这样做的一个好处是提高了程序的运行效率，但当遇到多线程编程时，变量的值可能因为别的线程而改变了，而该缓存的值不会相应改变，从而造成应用程序读取的值和实际的变量值不一致，例如，在本次线程内，当读取一个变量时，为提高存取速度，会先把变量读取到一个缓存中，当以后再取变量值时，就直接从缓存中取值，当变量值在本线程里改变时，会同时把变量的新值复制到该缓存中，以便保持一致。

`volatile` 是一个类型修饰符（type specifier），它是被设计用来修饰被不同线程访问和修改的变量。被 `volatile` 类型定义的变量，系统每次用到它时都是直接从对应的内存当中提取，而不会利用缓存。在使用了 `volatile` 修饰成员变量后，所有线程在任何时候所看到变量的值都是相同的。下面给出一个使用 `volatile` 的示例。

```

public class MyThread implements Runnable {
    private volatile Boolean flag;
    public void stop() {
        flag = false;
    }
    public void run() {
        while(flag)
            ;//do something
    }
}

```

以上代码示例是用来停止线程最常用的一种方法，如果 `boolean` 类型的变量 `flag` 没有被声明为 `volatile`，那么，当这个线程的 `run` 方法在判断 `flag` 值时，使用的有可能是缓存中的值，此时就不能及时地获取其他线程对 `flag` 所做的操作，因此会导致线程不能及时地停止。

需要注意的是，由于 `volatile` 不能保证操作的原子性，因此，一般情况下 `volatile` 不能代替 `synchronized`。此外，使用 `volatile` 会阻止编译器对代码的优化，因此会降低程序的执行效率。所以，除非迫不得已，否则，能不使用 `volatile` 就尽量不要使用 `volatile`。

4.3.8 instanceof 有什么作用

`instanceof` 是 Java 语言中的一个二元运算符，它的作用是判断一个引用类型的变量所指向的对象是否是一个类（或接口、抽象类、父类）的实例，即它左边的对象是否是它右边的类的实例该运算符返回 `boolean` 类型的数据。

常见的用法为：`result = object instanceof class`。如果 `object` 是 `class` 的一个实例，那么 `instanceof` 运算符返回 `true`；如果 `object` 不是 `class` 的一个实例，或者 `object` 是 `null`，那么 `instanceof` 运算符返回 `false`。

以如下程序为例：

```
public class Test {
    public static void main(String args[]){
        String s = "Hello";
        int[] a = {1,2};
        if(s instanceof String)
            System.out.println("true");
        if(s instanceof Object)
            System.out.println("true");
        if(a instanceof int[])
            System.out.println("true");
    }
}
```

程序运行结果为：

```
true
true
true
```

4.3.9 strictfp 有什么作用

关键字 `strictfp` 是 `strict float point` 的缩写，指的是精确浮点，它用来确保浮点数运算的准确性。JVM 在执行浮点数运算时，如果没有指定 `strictfp` 关键字，此时计算结果可能会不精确，而且计算结果在不同平台或厂商的虚拟机上会有不同的结果，导致意想不到的错误。而一旦使用了 `strictfp` 来声明一个类、接口或者方法，那么在所声明的范围内，Java 编译器以及运行环境会完全依照 IEEE 二进制浮点数算术标准（IEEE 754）来执行，在这个关键字声明的范围内所有浮点数的计算都是精确的。需要注意的是，当一个类被 `strictfp` 修饰时，所有方法都会自动被 `strictfp` 修饰。因此，`strictfp` 可以保证浮点数运算的精确性，而且在不同的硬件平台上会有一致的运行结果。下例给出了 `strictfp` 修饰类的使用方法：

```
public strictfp class Test{
    public static void testStrictfp(){
        float f=0.12365f;
        double d=0.03496421d;
    }
}
```

```

double sum = f + d;
System.out.println(sum);
}
public static void main(String[] args) {
testStrictfp();
}

```

程序运行结果为：

```
0.15861420949932098
```

4.4 基本类型与运算

4.4.1 Java 提供了哪些基本数据类型

Java 语言一共提供了 8 种原始的数据类型 (byte, short, int, long, float, double, char, boolean)，这些数据类型不是对象，而是 Java 语言中不同于类的特殊类型，这些基本类型的数据变量在声明之后就会立刻在栈上被分配内存空间。除了这 8 种基本的数据类型外，其他类型都是引用类型（例如类、接口、数组等），引用类型类似于 C++ 中的引用或指针的概念，它以特殊的方式指向对象实体，这类变量在声明时不会被分配内存空间，只是存储了一个内存地址而已。

表 4-2 为 Java 中的基本数据类型及其描述。

表 4-2 不同数据类型对比

数据类型	字节长度	范 围	默 认 值	包 装 类
int	4	$[-2147483648, 2147483647]$ ($-2^{31} \sim 2^{31} - 1$)	0	Integer
short	2	$[-32768, 32767]$	0	Short
long	8	$[-9223372036854775808, 9223372036854775807]$ ($-2^{63} \sim 2^{63} - 1$)	0L 或 0l	Long
byte	1	$[-128, 127]$	0	Byte
float	4	32 位 IEEE754 单精度范围	0.0f 或 0.0F	Float
double	8	64 位 IEEE754 双精度范围	0.0	Double
char	2	Unicode [0, 65535]	u0000	Character
boolean	1	true 和 false	false	Boolean

以上这些基本类型可以分为如下 4 种类型：

- 1) int 长度数据类型：byte (8 bit)、short (16 bit)、int (32 bit)、long (64 bit)。
- 2) float 长度数据类型：单精度 (32 bit float)、双精度 (64 bit double)。
- 3) boolean 类型变量的取值：ture、false。
- 4) char 数据类型：unicode 字符 (16 bit)。

此外, Java 语言还提供了对这些原始数据类型的封装类 (字符类型 `Character`, 布尔类型 `Boolean`, 数值类型 `Byte`、`Short`、`Integer`、`Long`、`Float`、`Double`)。需要注意的是, Java 中的数值类型都是有符号的, 不存在无符号的数, 它们的取值范围也是固定的, 不会随着硬件环境或者操作系统的改变而改变。除了以上提到的 8 种基本数据类型以外, 在 Java 语言中, 还存在另外一种基本类型 `void`, 它也有对应的封装类 `java.lang.void`, 只是无法直接对它进行操作而已。封装类型和原始类型有许多不同点: 首先, 原始数据类型在传递参数时都是按值传递, 而封装类型是按引用传递的。其次, 当封装类型和原始类型用作某个类的实例数据时, 它们所指定的默认值不同。对象引用实例变量的默认值为 `null`, 而原始类型实例变量的默认值与它们的类型有关 (例如 `int` 默认初始化为 0), 示例如下:

```
public class Test {
    String s;
    int i;
    float f;
    public static void main(String args[]) {
        Test t = new Test();
        System.out.println(t.s == null);
        System.out.println(t.i);
        System.out.println(t.f);
    }
}
```

程序运行结果为:

```
true
0
0.0
```

除了以上需要注意的内容外, 在 Java 语言中, 默认声明的小数是 `double` 类型的, 因此在对 `float` 类型的变量进行初始化时需要进行类型转换。`float` 类型的变量有两种初始化方法: `float f = 1.0f` 或 `float f = (float)1.0`。与此类似的是, 在 Java 语言中, 直接写的整型数字是 `int` 类型的, 如果在给数据类型为 `long` 的变量直接赋值时, `int` 类型的值无法表示一个非常大的数字, 因此, 在赋值时可以通过如下的方法来赋值: `long l = 26012402244L`。

引申:

1. 在 Java 语言中 `null` 值是什么? 在内存中 `null` 是什么?

`null` 不是一个合法的 `Object` 实例, 所以编译器并没有为其分配内存, 它仅仅用于表明该引用目前没有指向任何对象。其实, 与 C 语言类似, `null` 是将引用变量的值全部置 0。

2. 如何理解赋值语句 `String x = null`?

在 Java 语言中, 变量被分为两大类型: 原始值 (`primitive`) 与引用值 (`reference`)。声明为原始类型的变量, 其存储的是实际的值。声明为引用类型的变量, 存储的是实际对象的地址 (指针, 引用)。对于赋值语句 `String x = null`, 它定义了一个变量 “x”, x 中存放的是 `String` 引用, 此处为 `null`。

常见笔试题:

1. 下列表达式中, 正确的是 ()。

A. `byte b = 128;`

B. `boolean flag = null;`

C. float f = 0.9239;

D. long a = 2147483648L;

答案：D。A 中 byte 能表示的取值范围为 $[-128, 127]$ ，因此不能表示 128。B 中 boolean 的取值只能是 true 或 false，不能为 null。C 中 0.9239 为 double 类型，需要进行数据类型转换。

2. String 是最基本的数据类型吗？

答案：不是。基本数据类型包括 byte、int、char、long、float、double、boolean 和 short。

3. int 和 Integer 有什么区别？

答案：Java 语言提供两种不同的类型，即引用类型和原始类型（或内置类型）。int 是 Java 语言的原始数据类型，Integer 是 Java 语言为 int 提供的封装类。Java 为每个原始类型提供了封装类。

引用类型与原始类型的行为完全不同，并且它们具有不同的语义。而且，引用类型与原始类型具有不同的特征和用法。

4. 赋值语句 float f = 3.4 是否正确？

答案：不正确。数据 3.4 默认情况下是 double 类型，即双精度浮点数，将 double 类型数值赋值给 float 类型的变量，会造成精度损失，因此需要对其进行强制类型转换，即将 3.4 转换成 float 类型或者将 3.4 强制写成 float 类型。所以，float f = (float)3.4 或者 float f = 3.4F 写法都是可以的。

4.4.2 什么是不可变类

不可变类（immutable class）是指当创建了这个类的实例后，就不允许修改它的值了，也就是说，一个对象一旦被创建出来，在其整个生命周期中，它的成员变量就不能被修改了。它有点类似于常量（const），即只允许别的程序读，不允许别的程序进行修改。

在 Java 类库中，所有基本类型的包装类都是不可变类，例如 Integer、Float 等。此外，String 也是不可变类。可能有人会有疑问，既然 String 是不可变类，为什么还可以写出如下代码来修改 String 类型的值呢？

```
public class Test {
    public static void main(String[] args) {
        String s = "Hello";
        s += " world";
        System.out.println(s);
    }
}
```

程序运行结果为：

```
Hello world
```

表面上看，好像是修改 String 类型对象 s 的值。其实不是，String s = “Hello” 语句声明了一个可以指向 String 类型对象的引用，这个引用的名字为 s，它指向了一个字符串常量 “Hello”。s += “ world” 并没有改变 s 所指向的对象（由于 “Hello” 是 String 类型的对象，而 String 又是不可变量），这句代码运行后，s 指向了另外一个 String 类型的对象，该对象的内容为 “Hello world”。原来的那个字符串常量 “Hello” 还存在于内存中，并没有被改变。

在介绍完不可变类的基本概念后，下面主要介绍如何创建一个不可变类。通常来讲，要创建一个不可变类需要遵循下面 4 条基本原则：

- 1) 类中所有成员变量被 `private` 所修饰。
- 2) 类中没有写或者修改成员变量的方法，例如 `setxxx`，只提供构造函数，一次生成，永不改变。
- 3) 确保类中所有方法不会被子类覆盖，可以通过把类定义为 `final` 或者把类中的方法定义为 `final` 来达到这个目的。
- 4) 如果一个类成员不是不可变量，那么在成员初始化或者使用 `get` 方法获取该成员变量时，需要通过 `clone` 方法来确保类的不可变性。
- 5) 如果有必要，可使用覆盖 `Object` 类的 `equals()` 方法和 `hashCode()` 方法。在 `equals()` 方法中，根据对象的属性值来比较两个对象是否相等，并且保证用 `equals()` 方法判断为相等的两个对象的 `hashCode()` 方法的返回值也相等，这可以保证这些对象能被正确地放到 `HashMap` 或 `HashSet` 集合中。

除此之外，还有一些小的注意事项：由于类的不可变性，在创建对象时就需要初始化所有成员变量，因此最好提供一个带参数的构造函数来初始化这些成员变量。

下面通过给出一个错误的实现方法与一个正确的实现方法来说明在实现这种类时需要特别注意的问题。首先给出一个错误的实现方法：

```
import java.util. Date;
class ImmutableClass {
    private Date d;
    public ImmutableClass( Date d) {
        this. d = d;
    }
    public void printState() {
        System. out. println( d );
    }
}

public class TestImmutable {
    public static void main( String[ ] args ) {
        Date d = new Date();
        ImmutableClass immuC = new ImmutableClass( d );
        immuC. printState();
        d. setMonth( 5 );
        immuC. printState();
    }
}
```

程序运行结果为：

```
Sun Aug 04 17:41:47 CST 2013
Tue Jun 04 17:41:47 CST 2013
```

需要说明的是，由于 `Date` 对象的状态是可以被改变的，而 `ImmutableClass` 保存了 `Date` 类型对象的引用，当被引用的对象的状态改变时会导致 `ImmutableClass` 对象状态的改变。

其实，正确的实现方法如下所示：

```
import java.util. ArrayList;
import java.util. Date;
```

```

class ImmutableClass {
    private Date d;
    public ImmutableClass(Date d) {
        this.d = (Date)d.clone(); //解除了引用关系
    }
    public void printState() {
        System.out.println(d);
    }
    public Date getDate() {
        return (Date)d.clone();
    }
}

public class Test {
    public static void main(String[] args) {
        Date d = new Date();
        ImmutableClass immuC = new ImmutableClass(d);
        immuC.printState();
        d.setMonth(5);
        immuC.printState();
    }
}

```

程序运行结果为：

```

Sun Aug 04 17:47:03 CST 2013
Sun Aug 04 17:47:03 CST 2013

```

Java 语言中之所以设计有很多不可变类，主要是不可变类具有使用简单、线程安全、节省内存等优点，但凡事有利就有弊，不可变类自然也有其缺点，例如，不可变类的对象会因为值的不同而产生新的对象，从而导致无法预料的问题，所以，切不可滥用这种模式。

4.4.3 值传递与引用传递有哪些区别

方法调用是编程语言中非常重要的一个特性，在方法调用时，通常需要传递一些参数来完成特定的功能。Java 语言提供了两种参数传递的方式：值传递和引用传递。

(1) 值传递

在方法调用中，实参会把它的值传递给形参，形参只是用实参的值初始化一个临时的存储单元，因此形参与实参虽然有着相同的值，但是却有着不同的存储单元，因此对形参的改变不会影响实参的值。

(2) 引用传递

在方法调用中，传递的是对象（也可以看作是对象的地址），这时形参与实参的对象指向同一块存储单元，因此对形参的修改就会影响实参的值。

在 Java 语言中，原始数据类型在传递参数时都是按值传递，而包装类型在传递参数时是按引用传递的。

下面通过一个例子来介绍按值传递和按引用传递的区别：

```

public class Test {
    public static void testPassParameter(StringBuffer ssl, int n) {

```



```

        ssl.append(" World"); //引用
        n = 8; //值
    }
    public static void main(String[] args) {
        int i = 1;
        StringBuffer s1 = new StringBuffer(" Hello" );
        testPassParameter( s1,i);
        System.out.println(s1);
        System.out.println(i);
    }
}

```

程序运行结果为：

```

Hello World
1

```

按引用传递其实与传递指针类似，是把对象的地址作为参数的，如图 4-3 所示。

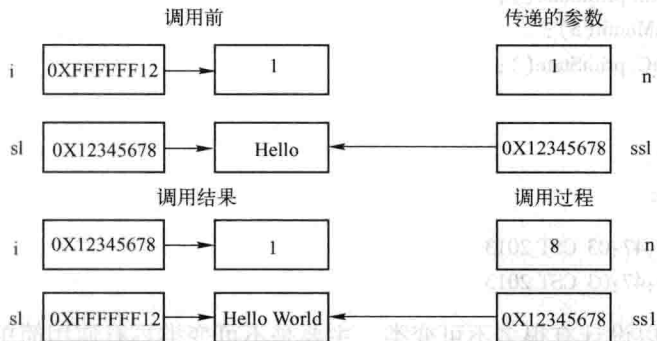


图 4-3 值传递与引用传递的区别

为了便于理解，假设 1 和“Hello”存储的地址分别为 0X12345678 和 0XFFFFFF12。在调用方法 `testPassParameter` 时，由于 `i` 为基本类型，因此参数是按值传递的，此时会创建一个 `i` 的副本，该副本与 `i` 有相同的值，把这个副本作为参数赋值给 `n`，作为传递的参数。而 `StringBuffer` 由于是一个类，因此按引用传递，传递的是它的引用（传递的是存储“Hello”的地址），如图 4-3 所示，在 `testPassParameter` 内部修改的是 `n` 的值，这个值与 `i` 是没关系的。但是在修改 `ssl` 时，修改的是 `ssl` 这个地址指向的字符串，由于形参 `ssl` 与实参 `s1` 指向的是同一块存储空间，因此修改 `ssl` 后，`s1` 指向的字符串也被修改了。

Java 中处理 8 种基本的数据类型用的是值传递，其他所有类型都用的是引用传递，由于这 8 种基本数据类型的包装类型都是不可变量，因此增加了对“按引用传递”的理解难度。下面给出一个示例来说明：

```

public class Test {
    public static void changeStringBuffer(StringBuffer ssl, StringBuffer ssl2) {
        ssl.append(" World");
        ssl2 = ssl;
    }
    public static void main(String[] args) {

```

```

Integer a = 1;
Integer b = a;
b++;
System.out.println(a);
System.out.println(b);
StringBuffer s1 = new StringBuffer("Hello");
StringBuffer s2 = new StringBuffer("Hello");
changeStringBuffer(s1, s2);
System.out.println(s1);
System.out.println(s2);

```

程序运行结果为：

```

1
2
Hello World
Hello

```

对于上述程序的前两个输出“1”和“2”，不少读者可能会认为，Integer 是按值传递的而不是按引用传递的，其实这是一个理解上的误区，上述代码还是按引用传递的，只是由于 Integer 是不可变类，因此没有提供改变它值的方法，在上例中，在执行完语句 `b++` 后，由于 Integer 是不可变类，因此此时会创建一个新值为 2 的 Integer 赋值给 `b`，此时 `b` 与 `a` 其实已经没有任何关系了。

下面通过程序的后两个输出来加深对“按引用传递”的理解。为了理解后两个输出结果，首先必须理解“引用也是按值传递的”这一要点。为了便于理解，假设 `s1` 和 `s2` 指向字符串的地址分别为 `0X12345678` 和 `0XFFFFFF12`，那么在调用函数 `changeStringBuffer` 时，传递 `s1` 与 `s2` 的引用就可以理解为传递了两个地址 `0X12345678` 和 `0XFFFFFF12`，而且这两个地址是按值传递的（即传递了两个值，`ss1` 为 `0X12345678`，`ss2` 为 `0XFFFFFF12`），在调用方法 `ss1.append("World")` 时，会修改 `ss1` 所指向的字符串的值，因此会修改调用者的 `s1` 的值，得到的输出结果为“Hello World”。但是在执行 `ss2 = ss1` 时，只会修改 `ss2` 的值而对 `s2` 毫无影响，因此 `s2` 的值在调用前后保持不变。为了便于理解，图 4-4 给出了函数调用的处理过程。

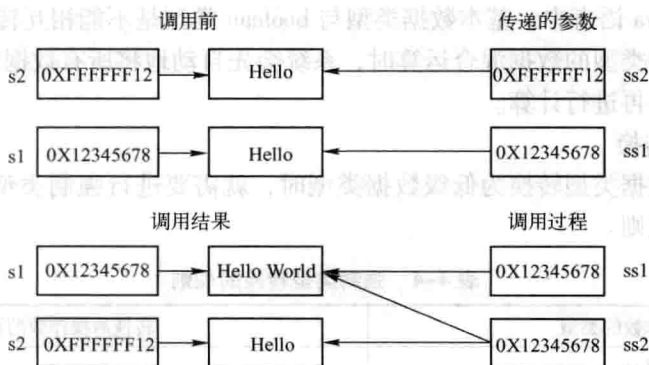


图 4-4 不变量的引用传递

从图 4-4 中可以看出，在传递参数时相当于传递了两个地址，在调用 `ss1.append("World")` 时，修改了这个地址所指向字符串的值，而在调用 `ss2 = ss1` 时，相当于修改了函数 `changeStringBuffer` 内部的局部变量 `ss2`，这个修改与 `ss1` 没关系。

常见笔试题：

下列说法中，正确的是（ ）。

- A. call by value 不会改变实际参数的值 B. call by reference 能改变实际参数
C. call by reference 不能改变实际参数的地址 D. call by reference 能改变实际参数的内容

答案：A、C、D。见上面讲解。

4.4.4 不同数据类型的转换有哪些规则

在 Java 语言中，当参与运算的两个变量的数据类型不同时，就需要进行隐式的数据类型转换，转换的规则为：从低精度向高精度转换，即优先级满足 `byte < short < char < int < long < float < double`，例如，不同数据类型的值在进行运算时，`short` 类型数据能够自动转为 `int` 类型，`int` 类型数据能够自动转换为 `float` 类型等。反之，则需要通过强制类型转换来实现。

在 Java 语言中，类型转换可以分为以下几种类型：

(1) 类型自动转换

低级数据类型可以自动转换为高级数据类型，表 4-3 给出了常见的自动类型转换的规则。

表 4-3 自动类型转换的规则

操作数 1 的类型	操作数 2 的类型	转换后的类型
long	byte short char int	long
int	byte short char	int
float	byte short int char long	float
double	byte short int long char float	double

当类型自动转换时，需要注意以下几点：

- 1) `char` 类型的数据转换为高级类型（如 `int`，`long` 等），会转换为其对应的 ASCII 码。
- 2) `byte`、`char`、`short` 类型的数据在参与运算时会自动转换为 `int` 型，但当使用“`+=`”运算时，就不会产生类型的转换（将在下一节中详细介绍）。
- 3) 另外，在 Java 语言中，基本数据类型与 `boolean` 类型是不能相互转换的。

总之，当有多种类型的数据混合运算时，系统会先自动地将所有数据转换成容量最大的那一种数据类型，然后再进行计算。

(2) 强制类型转换

当需要从高级数据类型转换为低级数据类型时，就需要进行强制类型转换，表 4-4 给出了强制类型转换的规则。

表 4-4 强制类型转换的规则

原操作数的类型	转换后操作数的类型
byte	char
char	byte char

(续)

原操作数的类型	转换后操作数的类型
short	byte char
int	byte short char
long	byte short char int
float	byte short char int long
double	byte short char int long double

需要注意的是，在进行强制类型转换时可能会损失精度。

常见笔试题：

1. 下面程序的运行结果是什么？

```
int i = 1;
if(i)
    System.out.println("true");
else
    System.out.println("false");
```

答案：编译错误。因为 if 条件只接受 boolean 类型的值（true 或 false），而 i 的类型为 int，int 类型不能被隐式地转换为 boolean 类型。

2. 对于下述代码结果强制类型转换后，变量 a 和 b 的值分别为（ ）。

```
short a = 128;
byte b = (byte)a
```

答案：a = 128，b = -1。short 类型变量占两个字节，a 对应的二进制为：00000000 10000000，由于 byte 只占一个字节，在强制转换为 byte 时只截取低字节：10000000，因此 b 的值为 -1。

4.4.5 强制类型转换的注意事项有哪些

Java 语言在涉及 byte、short 和 char 类型的运算时，首先会把这些类型的变量值强制转换为 int 类型，然后对 int 类型的值进行计算，最后得到的值也是 int 类型。因此，如果把两个 short 类型的值相加，最后得到的结果是 int 类型；如果把两个 byte 类型的值相加，最后也会得到一个 int 类型的值。如果需要得到 short 类型的结果，就必须显式地把运算结果转换为 short 类型，例如对于语句 short s1 = 1; s1 = s1 + 1，由于在运行时会首先将 s1 转换成 int 类型，因此 s1 + 1 的结果为 int 类型，这样编译器会报错，所以，正确的写法应该 short s1 = 1; s1 = (short) (s1 + 1)。

有一种例外情况。“+=”为 Java 语言规定的运算符，Java 编译器会对其进行特殊处理，因此，语句 short s1 = 1; s1 += 1 能够编译通过。

4.4.6 运算符优先级是什么

Java 语言中有很多运算符，由于运算符优先级的问题经常会导致程序出现意想不到的结果，表 4-5 详细介绍了运算符的优先级。

表 4-5 运算符的优先级

优先级	运算符	结合性
1	. () []	从左向右
2	+(正) -(负) ++ -- ~ !	
3	* / %	
4	+(加) -(减)	
5	<< >> (无符号右移) >>> (有符号右移)	
6	< <= > >= instanceof	
7	== !=	
8	&	
9		
10	^	
11	&&	
12		
13	?:	
14	= += -= *= /= %= &= = ^= ~= <<= >>= >>>=	

在实际使用时，如果不确定运算符的优先级，最好运用括号运算符来控制运算顺序。

常见笔试题：

下面程序的运行结果是什么？

```
public class Test {
    public static void main(String[] args) {
        byte a = 5;
        int b = 10;
        int c = a >> 2 + b >> 2;
        System.out.println(c);
    }
}
```

答案：0。由于“+”的优先级比“>>”高，因此程序中的表达式等价于 $a \gg (2 + b) \gg 2$ ，相当于 $a \gg 12 \gg 2$ ，因此运行结果为 0。

4.4.7 Math 类中 round、ceil 和 floor 方法的功能各是什么

round、ceil 和 floor 方法位于 Math 类中，Math 是一个包含了很多数学常量与计算方法的类，位于 java.lang 包下，能自动导入，而且 Math 类里边的方法全是静态方法。下面重点介绍这 3 个方法代表的含义。

1) round 方法表示四舍五入。round，意为“环绕”，其实现原理是在原来数字的基础上先增加 0.5 然后再向下取整，等同于 $(\text{int}) \text{Math.floor}(x + 0.5f)$ 。它的返回值类型为 int 型，例如， $\text{Math.round}(11.5)$ 的结果为 12， $\text{Math.round}(-11.5)$ 的结果为 -11。

2) ceil 方法的功能是向上取整。ceil，意为“天花板”，顾名思义是对操作数取顶， $\text{Math.ceil}(a)$ ，就是取大于 a 的最小的整数值。需要注意的是，它的返回值类型并不是 int 型，而是 double 型。若 a 是正数，则把小数“入”，若 a 是负数，则把小数“舍”。

3) floor 方法的功能是向下取整。floor, 意为“地板”, 顾名思义是对操作数取底。Math.floor(a), 就是取小于 a 的最大的整数值。它的返回值类型与 ceil 方法一样, 也是 double 型。若 a 是正数, 则把小数“舍”; 若 a 是负数, 则把小数“入”。

表 4-6 是一个实例分析。

表 4-6 floor、round 与 ceil 的区别

数 字	Math.floor 方法	Math.round 方法	Math.ceil 方法
1.4	1.0	1	2.0
1.5	1.0	2	2.0
1.6	1.0	2	2.0
-1.4	-2.0	-1	-1.0
-1.5	-2.0	-1	-1.0
-1.6	-2.0	-2	-1.0

以下是一段测试代码:

```
class MathTest {
    public static void main(String[] args) {
        float m = 6.4f;
        float n = -6.4f;
        System.out.println("Math.round(" + m + ") = " + Math.round(m));
        System.out.println("Math.round(" + n + ") = " + Math.round(n));
        System.out.println("Math.ceil(" + m + ") = " + Math.ceil(m));
        System.out.println("Math.ceil(" + n + ") = " + Math.ceil(n));
        System.out.println("Math.floor(" + m + ") = " + Math.floor(m));
        System.out.println("Math.floor(" + n + ") = " + Math.floor(n));
    }
}
```

上例的运行结果为:

```
Math.round(6.4) = 6
Math.round(-6.4) = -6
Math.ceil(6.4) = 7.0
Math.ceil(-6.4) = -6.0
Math.floor(6.4) = 6.0
Math.floor(-6.4) = -7.0
```

常见笔试题:

Math.round(11.5) 等于多少? Math.round(-11.5) 等于多少?

答案: 12, -11。见上面讲解。

4.4.8 ++i 与 i++ 有什么区别

在编程时, 经常会用到变量的自增或自减操作, 尤其在循环中用得最多。以自增为例, 有两种自增方式: 前置与后置, 即 ++i 和 i++, 它们的不同点在于 i++ 是在程序执行完毕后进行自增, 而 ++i 是在程序开始执行前进行自增, 示例如下:

```
public class Test {
```



```

public static void main(String[] a) {
    int i = 1;
    System.out.println(i+++i++);
    System.out.println("i = " + i);
    System.out.println(i+++++i);
    System.out.println("i = " + i);
    System.out.println(i+++i+++i+++);
    System.out.println("i = " + i);
}

```

程序运行结果为：

```

3
i = 3
8
i = 5
18
i = 8

```

上例中的程序运行结果让很多读者感到困惑不解，其实稍作分析，问题便迎刃而解了。表达式 $i+++i++$ 首先执行第一个 $i++$ 操作，由于自增操作会稍后执行，因此，运算时 i 的值还是 1，但自增操作后， i 的值变为了 2，接着执行第二个 $i++$ ，运算时， i 的值已经为 2 了，而执行了一个自增操作后， i 的值变为了 3，所以 $i+++i++=1+2=3$ ，而运算完成后， i 的值变为 3。

表达式 $i+++++i$ 首先执行第一个 $i++$ ，但是自增操作会稍后执行。因此，此时 i 的值还是 3，接着执行 $++i$ ，此时 i 的值变为 4，同时还要补执行 $i++$ 的自增操作，因此此时 i 的值变为 5，所以 $i+++++i=3+5=8$ 。

同理， $i+++i+++i+++=5+6+7=18$ 。

常见笔试题：

假设 $x=1$ ， $y=2$ ， $z=3$ ，则表达式 $y+=z--/++x$ 的值是 ()。

A. 3 B. 3.5 C. 4 D. 5

答案：A。见上面讲解。

4.4.9 如何实现无符号数的右移操作

Java 提供了两种右移运算符：“ \gg ”和“ \ggg ”。其中，“ \gg ”被称为有符号右移运算符，“ \ggg ”被称为无符号右移运算符，它们的功能是将参与运算的对象对应的二进制数右移指定的位数。二者的不同点在于“ \gg ”在执行右移操作时，若参与运算的数字为正数，则在高位补 0；若为负数，则在高位补 1。而“ \ggg ”则不同，无论参与运算的数字为正数或为负数，在执行运算时，都会在高位补 0。

此外，需要特别注意的是，在对 `char`、`byte`、`short` 等类型的数进行移位操作前，编译器都会自动地将数值转化为 `int` 类型，然后才进行移位操作。由于 `int` 型变量只占 4Byte (32 bit)，因此当右移的位数超过 32 bit 时，移位运算没有任何意义。所以，在 Java 语言中，为了保证移动位数的有效性，以使右移的位数不超过 32 bit，采用了取余的操作，即使 $a \gg n$ 等价于 $a \gg (n\%32)$ ，示例如下：

```

public class Test {
    public static void main(String[] a) {
        int i = -4;
        System.out.println("-----int >> : " + i);
        System.out.println("移位前二进制:" + Integer.toBinaryString(i));
        i >>= 1;
        System.out.println("移位后二进制:" + Integer.toBinaryString(i));
        System.out.println("-----int >> : " + i);
        i = -4;
        System.out.println("-----int >>> : " + i);
        System.out.println("移位前二进制:" + Integer.toBinaryString(i));
        i >>>= 1;
        System.out.println("移位后二进制:" + Integer.toBinaryString(i));
        System.out.println("-----int >>> : " + i);
        short j = -4;
        System.out.println("-----short >>> : " + j);
        System.out.println("移位前二进制:" + Integer.toBinaryString(j));
        j >>>= 1;
        System.out.println("移位后二进制:" + Integer.toBinaryString(j));
        System.out.println("-----short >>> : " + j);
        i = 5;
        System.out.println("-----int >> : " + i);
        System.out.println("移位前二进制:" + Integer.toBinaryString(i));
        i >>= 32;
        System.out.println("移位后二进制:" + Integer.toBinaryString(i));
        System.out.println("-----int >> : " + i);
    }
}

```

程序运行结果为:

```

-----int >> : -4
移位前二进制:11111111111111111111111111111110
移位后二进制:11111111111111111111111111111110
-----int >> : -2
-----int >>> : -4
移位前二进制:11111111111111111111111111111110
移位后二进制:11111111111111111111111111111110
-----int >>> :2147483646
-----short >>> : -4
移位前二进制:11111111111111111111111111111110
移位后二进制:11111111111111111111111111111110
-----short >>> : -2
-----int >> :5
移位前二进制:101
移位后二进制:101
-----int >> :5

```

需要特别说明的是,对于 short 类型来说,由于 short 只占 2Byte,在移位操作时会先转换为 int 类型,虽然在进进行无符号右移时会在高位补 1,但当把运算结果再赋值给 short 类型变量

时，只会取其中低位的两个字节，因此，高位无论补 0 还是补 1 对运算结果无影响。在上例中，-4 的二进制表示为 11111111 11111100（负数以补码格式存储的），在转换为二进制时会以 4Byte 的方式输出，高位会补 1，因此输出为 11111111111111111111111111111100，在执行无符号数右移后其二进制变为 01111111111111111111111111111110，当把运算结果再复制给 i 时只会取低位的两个字节，因此，运算结果的二进制表示为：11111111 11111110，对应的十进制值为 -2，当把 -2 以二进制形式输出时，同理会以 4Byte 的方式输出，高位会补 1，因此输出为 11111111111111111111111111111110。

引申：“<<”运算符与“>>”运算符有什么异同？

“<<”运算符表示左移，左移 n 位表示原来的值乘 2 的 n 次方，经常用来代替乘法操作，例如，一个数 m 乘以 16 可以表示为将这个数左移 4 位 ($m \ll 4$)，由于 CPU 直接支持位运算，因此位运算比乘法运算的效率高。

与右移运算不同的是，左移运算没有有符号与无符号左移，在左移时，移除高位的同时在低位补 0。以 $4 \ll 3$ （4 为 int 型）为例，其运算步骤如下所示。

- 1) 把 4 转换为二进制数字 0000 0000 0000 0000 0000 0000 0000 0100。
- 2) 把该数字的高三位移走，同时其他位向左移动 3 为。
- 3) 在最低位补 3 个零。最终结果为 0000 0000 0000 0000 0000 0000 0010 0000，对应的十进制数为 32。

与右移运算符相同的是，当进行左移运算时，如果移动的位数超过了该类型的最大位数，那么编译器会对移动的位数取模，例如对 int 型移动 33 位，实际上只移动了 $33 \% 32 = 1$ 位。

4.4.10 char 型变量中是否可以存储一个中文汉字

在 Java 语言中，默认使用的 Unicode 编码方式，即每个字符占用两个字节，因此可以用来存储中文。虽然 String 是由 char 所组成的，但是它采用了一种更加灵活的方式来存储，即英文占用一个字符，中文占用两个字符，采用这种存储方式的一个重要作用就是可以减少所需的存储空间，提高存储效率。

下例是一个打印中文字符字节数的例子。

```
public class Test {
    public static void getLen(String str) {
        System.out.println(str + " 的长度:" + str.length() + " 所占字节数:" + str.getBytes(
        .length);
    }
    public static void main(String[] args) {
        String s1 = "Hello";
        String s2 = "你好";
        getLen(s1);
        getLen(s2);
    }
}
```

程序运行结果为：

```
“Hello”的长度:5 所占字节数:5
“你好”的长度:2 所占字节数:4
```

在上例中：“Hello”是英文字符，因此所占字节数和字符串长度相同。“你好”字符串长度为2，由于每个字符都占用两个字节，因此总的字节数为4。此方法可以用来判断一个字符串中是否包含中文字符，示例如下：

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Test {
    public static void judgeChineseCharacter(String str) {
        String regex = "[\\u4e00-\\u9fa5]";
        //判断是否存在中文字符
        if (str.getBytes().length == str.length()) {
            System.out.println("无汉字");
        } else { //如果存在中文字符,找出字符串中的中文字符
            Pattern p = Pattern.compile(regex);
            Matcher m = p.matcher(str);
            while (m.find()) {
                System.out.print(m.group(0) + " ");
            }
        }
    }

    public static void main(String[] args) {
        judgeChineseCharacter("Hello World");
        judgeChineseCharacter("Hello 你好");
    }
}
```

程序运行结果为：

```
无汉字
你好
```

在上例中，首先通过字节长度和字符串长度判断字符串是否包含中文字符，若包含，则用正则表达式匹配的方式找出字符串中的所有中文字符。

常见笔试题：

在 Java 语言中，下列关于字符集编码（Character Set Encoding）和国际化（i18n）的叙述，哪些是正确的？（ ）

- A. 每个中文字符占用 2Byte，每个英文字符占用 1Byte
- B. 假设数据库中的字符是以 GBK 编码的，那么现实数据库数据的网页也必须是 GBK 编码的
- C. Java 的 char 类型，以 UTF-16 Big Endian 的方式保存一个字符
- D. 实现国际化应用常用的手段是利用 ResourceBundle 类

答案：A、D。从上面的介绍可以得出 A 是正确的，C 是错误的。对于 B，数据库与 Web 页面可以有各自的编码，二者没有必然的关系。对于 D，ResourceBundle 是一个资源处理类，可以经常在国际化应用中使用。

4.5 字符串与数组

4.5.1 字符串创建与存储的机制是什么

在 Java 语言中，字符串起着非常重要的作用，字符串的声明与初始化主要有如下两种情况：

1) 对于 `String s1 = new String("abc")` 语句与 `String s2 = new String("abc")` 语句，存在两个引用对象 `s1`、`s2`，两个内容相同的字符串对象 "abc"，它们在内存中的地址是不同的。只要用到 `new` 总会生成新的对象。

2) 对于 `String s1 = "abc"` 语句与 `String s2 = "abc"` 语句，在 JVM 中存在着一个字符串池，其中保存着很多 `String` 对象，并且可以被共享使用，`s1`、`s2` 引用的是同一个常量池中的对象。由于 `String` 的实现采用了 Flyweight 的设计模式，当创建一个字符串常量时，例如 `String s = "abc"`，会首先在字符串常量池中查找是否已经有相同的字符串被定义，其判断依据是 `String` 类 `equals(Object obj)` 方法的返回值。若已经定义，则直接获取对其的引用，此时不需要创建新的对象；若没有定义，则首先创建这个对象，然后把它加入到字符串池中，再将它的引用返回。由于 `String` 是不可变类，一旦创建好了就不能被修改，因此 `String` 对象可以被共享而且不会导致程序的混乱。

具体而言：

```
String s = "abc";           // 把"abc"放到常量区中,在编译时产生
String s = "ab" + "c";     // 把"ab" + "c"转换为字符串常量"abc"放到常量区中
String s = new String("abc"); // 在运行时把"abc"放到堆里面
```

例如，

```
String s1 = "abc";         //在常量区里面存放了一个"abc"字符串对象
String s2 = "abc";         //s2 引用常量区中的对象,因此不会创建新的对象
String s3 = new String("abc"); //在堆中创建新的对象
String s4 = new String("abc"); //在堆中又创建一个新的对象
```

为了便于理解，可以把 `String s = new String("abc")` 语句的执行人为地分解成两个过程：第一个过程是新建对象的过程，即 `new String("abc")`；第二个过程是赋值的过程，即 `String s =`。由于第二个过程只是定义了一个名为 `s` 的 `String` 类型的变量，将一个 `String` 类型对象的引用赋值给 `s`，因此在这个过程中不会创建新的对象。第一个过程中 `new String("abc")` 会调用 `String` 类的构造函数：

```
public String(String original) {
    //body
}
```

在调用这个构造函数时，传入了一个字符串常量，因此语句 `new String("abc")` 也就等价于 "abc" 和 `new String()` 两个操作了。若在字符串池中不存在 "abc"，则会创建一个字符串常量 "abc"，并将其添加到字符串池中；若存在，则不创建，然后 `new String()` 会在堆中创建一个新的对象，所以 `s3` 与 `s4` 指向的是堆中不同的 `String` 对象，地址自然也不相同了，如图 4-5 所示。

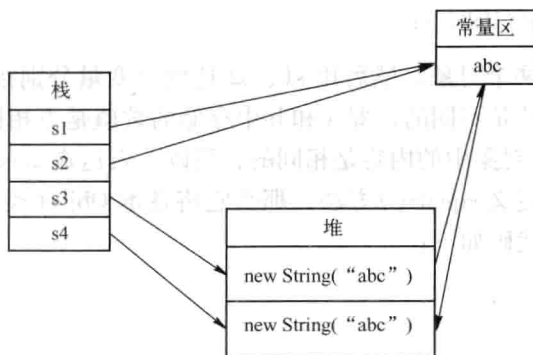


图 4-5 两种字符串存储方式

引申：对于 String 类型的变量 s，赋值语句 `s = null` 与 `s = ""` 是否相同？

对于赋值语句 `s = null`，其中 s 是一个字符串类型的引用，它不指向任何一个字符串。而赋值语句 `s = ""` 中的 s 是一个字符串类型的引用，它指向另外一个字符串（这个字符串的值为“”，即空字符串），因此，这两者是不同的。

常见笔试题：

`new String("abc")` 创建了几个对象？

答案：一个或两个。如果常量池中原来有“abc”，那么只创建一个对象；如果常量池中原来没有字符串“abc”，那么就会创建两个对象。

4.5.2 “==”、equals 和 hashCode 有什么区别

1) “==” 运算符用来比较两个变量的值是否相等。也就是说，该运算符用于比较变量对应的内存中所存储的数值是否相同，要比较两个基本类型的数据或两个引用变量是否相等，只能使用“==”运算符。

具体而言，如果两个变量是基本数据类型，可以直接使用“==”运算符来比较其对应的值是否相等。如果一个变量指向的数据是对象（引用类型），那么，此时涉及了两块内存，对象本身占用一块内存（堆内存），变量也占用一块内存，例如，对于赋值语句 `String s = new String()`，变量 s 占用一块存储空间，而 `new String()` 则存储在另外一块存储空间里，此时，变量 s 所对应内存中存储的数值就是对象占用的那块内存的首地址。对于指向对象类型的变量，如果要比较两个变量是否指向同一个对象，即要看这两个变量所对应内存中的数值是否相等（这两个对象是否指向同一块存储空间），这时候就可以用“==”运算符进行比较。但是，如果要比较这两个对象的内容是否相等，那么用“==”运算符就无法实现了。

2) equals 是 Object 类提供的方法之一。每一个 Java 类都继承自 Object 类，所以每一个对象都具有 equals 这个方法。Object 类中定义的 equals(Object) 方法是直接使用“==”运算符比较的两个对象，所以在没有覆盖 equals(Object) 方法的情况下，equals(Object) 与“==”运算符一样，比较的是引用。

相比“==”运算符，equals(Object) 方法的特殊之处就在于它可以被覆盖，所以可以通过覆盖的方法让它不是比较引用而是比较数据内容，例如 String 类的 equals 方法是用于比较两个独立对象的内容是否相同，即堆中的内容是否相同，以下面的代码为例：

```
String s1 = new String("Hello");
```



```
String s2 = new String("Hello");
```

两条 new 语句创建了两个对象，然后用 s1、s2 这两个变量分别指向一个对象，这是两个不同的对象，它们的首地址是不同的，即 a 和 b 中存储的数值是不相同的，所以，表达式 a = b 将返回 false，而这两个对象中的内容是相同的，所以，表达式 a.equals(b) 将返回 true。

如果一个类没有自己定义 equals() 方法，那么它将继承 Object 类的 equals() 方法，Object 类的 equals() 方法的实现代码如下：

```
boolean equals(Object o) {
    return this == o;
}
```

通过以上例子可以说明，如果一个类没有自己定义 equals() 方法，它默认的 equals() 方法（从 Object 类继承的）就是使用 “==” 运算符，也是在比较两个变量指向的对象是否是同一对象，此时使用 equals() 方法和使用 “==” 运算符会得到同样的结果。若比较的是两个独立的对象，则总返回 false。如果编写的类希望能够比较该类创建的两个实例对象的内容是否相同，那么必须覆盖 equals() 方法，由开发人员自己编写代码来决定在什么情况下即可认为两个对象的内容是相同的。

3) hashCode() 方法是从 Object 类中继承过来的，它也用来鉴定两个对象是否相等。Object 类中的 hashCode() 方法返回对象在内存中地址转换成的一个 int 值，所以如果没有重写 hashCode() 方法，任何对象的 hashCode() 方法都是不相等的。

虽然 equals() 方法也是用来判断两个对象是否相等的，但是它与 hashCode() 方法是有区别的。一般来讲，equals() 方法是给用户调用的，如果需要判断两个对象是否相等，可以重写 equals() 方法，然后在代码中调用，这样就可以判断它们是否相等了。对于 hashCode() 方法，用户一般不会去调用它，例如在 hashmap 中，由于 key 是不可以重复的，它在判断 key 是否重复时就判断了 hashCode() 这个方法，而且也用到了 equals() 方法。此处“不可以重复”指的是 equals() 和 hashCode() 只要有一个不等就可以了。所以，hashCode() 方法相当于是一个对象的编码，就好像文件中的 md5，它与 equals() 方法的不同之处就在于它返回的是 int 型，比较起来不直观。

一般在覆盖 equals() 方法的同时也要覆盖 hashCode() 方法，否则，就会违反 Object.hashCode 的通用约定，从而导致该类无法与所有基于散列值 (hash) 的集合类 (HashMap、HashSet 和 Hashtable) 结合在一起正常运行。

hashCode() 方法的返回值和 equals() 方法的关系如下：如果 x.equals(y) 返回 true，即两个对象根据 equals 方法比较是相等的，那么调用这两个对象中任意一个对象的 hashCode() 方法都必须产生同样的整数结果。如果 x.equals(y) 返回 false，即两个对象根据 equals() 方法比较是不相等的，那么 x 和 y 的 hashCode() 方法的返回值有可能相等，也有可能不相等。反之，hashCode() 方法的返回值不相等，一定能推出 equals() 方法的返回值也不相等，而 hashCode() 方法的返回值相等，equals 方法的返回值则可能相等，也可能不相等。

常见笔试题：

1. 假设有以下代码 String s = “hello”; String t = “hello”; char c [] = { 'h', 'e', 'l', 'l', 'o' }, 下列选项中返回 false 语句的是：()

A. s.equals(t) B. t.equals(c) C. s==t D. t.equals(new String(“hello”))

答案：B。从上面的介绍可以看出 A 与 D 显然会返回 true，从上一节的介绍中可以得出选

项 C 的返回值也为 true。对于 B，由于 t 与 c 分别为字符串类型和数组类型，因此返回值为 false。

2. 下面程序的输出结果是什么？

```
String s = "abc";
String s1 = "ab" + "c";
System.out.println(s == s1);
```

答案：true。在上一节中已经介绍过，“ab” + “c”在编译器就被转换为“abc”，存放在常量区，因此输出结果为 true。

3. Set 里的元素是不能重复的，那么用什么方法来区分是否重复呢？是用“==”还是 equals()？它们有什么差别？

答案：用 iterator() 方法来区分是否重复。equals() 方法与“==”运算符的区别见上面讲解。

4.5.3 String、StringBuffer、StringBuilder 和 StringTokenizer 有什么区别

Java 语言有 4 个类可以对字符或字符串进行操作，它们分别是 Character、String、StringBuffer 和 StringTokenizer，其中 Character 用于单个字符操作，String 用于字符串操作，属于不可变类，而 StringBuffer 也是用于字符串操作，不同之处是 StringBuffer 属于可变类。

String 是不可变类，也就是说，String 对象一旦被创建，其值将不能被改变，而 StringBuffer 是可变类，当对象被创建后仍然可以对其值进行修改。由于 String 是不可变类，因此适合在需要被共享的场合中使用，而当一个字符串经常需要被修改时，最好使用 StringBuffer 来实现。如果用 String 来保存一个经常被修改的字符串时，在字符串被修改时会比 StringBuffer 多很多附加的操作，同时生成很多无用的对象，由于这些无用的对象会被垃圾回收器来回收，因此会影响程序的性能。在规模小的项目里面这个影响很小，但是在一个规模大的项目里面，这会对程序的运行效率带来很大的影响。

String 与 StringBuffer 的另外一个区别在于当实例化 String 时，可以利用构造函数 (String s1 = new String("world")) 的方式来对其进行初始化，也可以用赋值 (String s = "Hello") 的方式来初始化，而 StringBuffer 只能使用构造函数 (StringBuffer s = new StringBuffer("Hello")) 的方式来初始化。

String 字符串修改实现的原理如下：当用 String 类型来对字符串进行修改时，其实现方法是首先创建一个 StringBuffer，其次调用 StringBuffer 的 append() 方法，最后调用 StringBuffer 的 toString() 方法把结果返回，示例如下：

```
String s = "Hello";
s += "World";
s += "World"
```

以上代码等价于下述代码：

```
StringBuffer sb = new StringBuffer(s);
s.append("World"); s = sb.toString();
```

由此可以看出，上述过程比使用 StringBuffer 多了一些附加的操作，同时也生成了一些临时的对象，从而导致程序的执行效率降低。为了更好地说明这一问题，下面分析一个示例：

```
public class Test {
```

```

public static void testString() {
    String s = "Hello";
    String s1 = "world";
    long start = System.currentTimeMillis();
    for (int i = 0; i < 10000; i++) {
        s += s1;
    }
    long end = System.currentTimeMillis();
    long runTime = (end - start);
    System.out.println("testString:" + runTime);
}

public static void testStringBuffer() {
    StringBuffer s = new StringBuffer("Hello");
    String s1 = "world";
    long start = System.currentTimeMillis();
    for (int i = 0; i < 10000; i++) {
        s.append(s1);
    }
    long end = System.currentTimeMillis();
    long runTime = (end - start);
    System.out.println("testStringBuffer:" + runTime);
}

public static void main(String[] args) {
    testString();
    testStringBuffer();
}

```

程序运行结果为：

```
testString:1760
```

```
testStringBuffer:3
```

从程序的运行结果可以看出，当一个字符串需要经常被修改时，使用 `StringBuffer` 比使用 `String` 要好很多。

`StringBuilder` 也可以被修饰的字符串，它与 `StringBuffer` 类似，都是字符串缓冲区，但 `StringBuilder` 不是线程安全的，如果只是在单线程中使用字符串缓冲区，那么 `StringBuilder` 的效率会更高些。因此在只有单线程访问时可以使用 `StringBuilder`，当有多个线程访问时，最好使用线程安全的 `StringBuffer`。因为 `StringBuffer` 必要时可以对这些方法进行同步，所以任意特定实例上的所有操作就好像是以串行顺序发生的，该顺序与所涉及的每个线程进行的方法调用顺序一致。

在执行效率方面，`StringBuilder` 最高，`StringBuffer` 次之，`String` 最低，鉴于这一情况，一般而言，如果要操作的数据量比较小，应优先使用 `String` 类；如果是在单线程下操作大量数据，应优先使用 `StringBuilder` 类；如果是在多线程下操作大量数据，应优先考虑 `StringBuffer` 类。

`StringTokenizer` 是用来分割字符串的工具类，示例如下：

```

import java.util.StringTokenizer;

public class Test {

```

```

public static void main(String args[]) {
    StringTokenizer st = new StringTokenizer("Welcome to our country");
    while (st.hasMoreTokens()) {
        System.out.println(st.nextToken());
    }
}

```

程序运行结果为:

```

Welcome
to
our
country

```

4.5.4 Java 中数组是不是对象

数组是指具有相同类型的数据的集合,它们一般具有固定的长度,并且在内存中占据连续的空间。在 C/C++ 语言中,数组名只是一个指针,这个指针指向了数组的首元素,既没有属性也没有方法可以调用,而在 Java 语言中,数组不仅有其自己的属性(例如 length 属性),也有一些方法可以被调用(例如 clone 方法)。由于对象的特点是封装了一些数据,同时提供了一些属性和方法,从这个角度来讲,数组是对象。每个数组类型都有其对应的类型,可以通过 instanceof 来判断数据的类型,示例如下:

```

public class SubClass {
    public static void main(String[] args) {
        int [] a = {1,2};
        int [][] b = new int[2][4];
        String [] s = {"a","b"};
        if(a instanceof int[])
            System.out.println("the type for a is int[]");
        if(b instanceof int[][])
            System.out.println("the type for b is int[][]");
        if(s instanceof String[])
            System.out.println("the type for s is String[]");
    }
}

```

程序运行结果为:

```

the type for a is int[]
the type for b is int[][]
the type for s is String[]

```

4.5.5 数组的初始化方式有哪几种

在 Java 语言中,一维数组的声明方式为

```
type arrayName[] 或 type[] arrayName
```

其中, type 既可以是基本的数据类型,也可以是类, arrayName 表示数组的名字, [] 用来

表示这个变量的类型为一维数组。与 C/C++ 语言不同的是，在 Java 语言中，数组被创建后会根据数组存放的数据类型初始化成对应的初始值（例如，int 类型会初始化为 0，对象会初始化为 null）。另外一个不同之处是 Java 数组在定义时，并不会给数组元素分配存储空间，因此 [] 中不需要指定数组的长度，对于使用上面方式定义的数组在使用时还必须为之分配空间，分配方法为：

```
arrayName = new type[arraySize]; // arraySize 表示数组的长度
```

在完成数组的声明后，需要对其进行初始化，下面介绍两种初始化方法：

- 1) `int[] a = new int[5];` // 动态创建了一个包含 5 个整型值的数组，默认初始化为 0
- 2) `int[] a = {1,2,3,4,5};` // 声明一个数组类型变量并初始化

当然，在使用时也可以把数组的声明和初始化分开来写，例如：

- 1) `int[] a;` // 声明一个数组类型的对象 a
`a = new int[5];` // 给数组 a 申请可以存放 5 个 int 类型大小的空间，默认值为 0
- 2) `int[] a;` // 声明一个数组类型的对象 a
`a = new int[] {1,2,3,4,5};` // 给数组申请存储空间，并初始化为默认值

以上主要介绍了一维数组的声明与初始化的方式，下面介绍二维数组的声明与初始化的方式，二维数组有 3 种声明的方法：

- 1) `type arrayName[][];`
- 2) `type[][] arrayName;`
- 3) `type[] arrayName[];`

需要注意的是，在声明二维数组时，其中 [] 必须为空。

二维数组也可以用初始化列表的方式进行初始化，其一般形式为

```
type arrayName[][] = { {c11,c12,c13...}, {c21,c22,c23...}, {c31,c32,c33...} ...};
```

除了以上介绍的方法以外，也可以通过 new 关键字来给数组申请存储空间，形式如下：

```
type arrayname[][] = new type[行数][列数]
```

与 C/C++ 语言不同的是，在 Java 语言中，二维数组的第二维的长度可以不同。假如要定义一个有两行的二维数组，第一行有两列，第二行有三列，定义方法如下：

- 1) `int [][] arr = { {1, 2}, {3, 4, 5} };`
- 2) `int [][] a = new int[2][];`
`a[0] = new int[] {1,2};`
`a[1] = new int[] {3, 4, 5};`

对二维数组的访问也是通过下标来完成，一般形式为 `arrayName[行号][列号]`，下例介绍二维数组的遍历方法：

```
public class SubClass {
    public static void main(String[] args) {
        int a[][] = new int[2][];
        a[0] = new int[] {1,2};
        a[1] = new int[] {3,4,5};
        for(int i=0; i<a.length; i++){
            for(int j=0; j<a[i].length; j++){
                System.out.print(a[i][j] + " ");
            }
        }
    }
}
```

程序运行结果为：

1 2 3 4 5

常见笔试题：

1. 下列数组的定义中，哪3条是正确的？（ ）

- A. `public int a []` B. `static int [] a` C. `public [] int a`
 D. `private int a [3]` E. `private int [3] a []` F. `public final int [] a`

答案：A、B、F。见上面讲解。

2. 下列数组定义及赋值中，错误的是（ ）。

- A. `int intArray[];`
 B. `intArray = new int[3];intArray[1] = 1; intArray[2] = 2; intArray[3] = 3;`
 C. `int a[] = {1,2,3,4,5};`
 D. `int[][] a = new int[2][];a[0] = new int[3];a[1] = new int[3];`

答案：B。B中对数组的访问越界了。数组大小为3，数组第一个元素为 `intArray[0]`，最后一个元素为 `intArray[2]`。

3. 下列说法中，错误的有（ ）。

- A. 数组是一种对象 B. 数组属于一种原生类
 C. `int number[] = {31, 23, 33, 43, 35, 63}` D. 数组的大小可以任意改变

答案：B、C、D。原生类指未被实例化的类，数组一般指实例化、被分配空间的类，所以不属于原生类。

4. 下列语句中，创建了一个数组实例的是（ ）

- A. `int[] ia = new int [15];` B. `float fa = new float [20];`
 C. `char[] ca = "Some String";` D. `int ia [][] = {4, 5, 6} {1, 2, 3};`

答案：A。见上面讲解。

4.5.6 length 属性与 length() 方法有什么区别

在 C/C++ 语言中，每当调用一个方法需要传递数组时，就必须同时传递数组的长度，因为在方法调用时传递的参数为数组的首地址，而对数组的实际长度却无法获知，这样会导致在对数组进行访问时可能产生越界。而在 Java 语言中，数组提供了 `length` 属性来获取数组的长度。

在 Java 语言中，`length()` 方法是针对字符串而言的，`String` 提供了 `length()` 方法来计算字符串的长度，示例如下例：

```
public class Test {
    public static void testArray(int[] arr) {
        System.out.println("数组长度为:" + arr.length);
    }
    public static void testString(String s) {
        System.out.println("字符串长度为:" + s.length());
    }
}
```



```

public static void main(String[] args) {
    int[] arr = {1,3,5,7};
    String s = "1357";
    testArray(arr);
    testString(s);
}
}

```

程序运行结果为：

数组长度为:4

字符串长度为:4

除了 `length` 属性与 `length()` 方法外，Java 中还有一个计算对象大小的方法——`size()` 方法，该方法是针对泛型集合而言的，用于查看泛型中有多少个元素。（备注：泛型是对 Java 语言的类型系统的一种扩展，以支持创建可以按类型进行参数化的类，可以把类型参数看作是使用参数化类型时指定的类型的一个占位符，就像方法的形式参数是运行时传递的值的占位符一样）

4.6 异常处理

4.6.1 finally 块中的代码什么时候被执行

问题描述：`try {}` 里有一个 `return` 语句，那么紧跟在这个 `try` 后的 `finally {}` 中的代码是否会被执行？如果会的话，什么时候被执行，在 `return` 之前还是 `return` 之后？

在 Java 语言的异常处理中，`finally` 块的作用就是为了保证无论出现什么情况，`finally` 块里的代码一定会被执行。由于程序执行 `return` 就意味着结束对当前函数的调用并跳出这个函数体，因此任何语句要执行都只能在 `return` 前执行（除非碰到 `exit` 函数），因此 `finally` 块里的代码也是在 `return` 前执行的。此外，如果 `try - finally` 或者 `catch - finally` 中都有 `return`，那么 `finally` 块中的 `return` 语句将会覆盖别处的 `return` 语句，最终返回到调用者那里的是 `finally` 中 `return` 的值。下面通过一个例子（示例 1）来说明这个问题：

```

public class Test {
    public static int testFinally() {
        try {
            return 1;
        } catch (Exception e) {
            return 0;
        } finally {
            System.out.println("execute finally");
        }
    }

    public static void main(String[] args) {
        int result = testFinally();
        System.out.println(result);
    }
}

```

程序运行结果为：

```
execute finally
1
```

从上面这个例子中可以看出，在执行 return 语句前确实执行了 finally 块中的代码。紧接着，在 finally 块里面放置个 return 语句，例子（示例 2）如下所示：

```
public class Test {
    public static int testFinally() {
        try {
            return 1;
        } catch (Exception e) {
            return 0;
        } finally {
            System.out.println("execute finally");
            return 3;
        }
    }

    public static void main(String[] args) {
        int result = testFinally();
        System.out.println(result);
    }
}
```

程序运行结果为：

```
execute finally
3
```

从以上运行结果可以看出，当 finally 块中有 return 语句时，将会覆盖函数中其他 return 语句。此外，由于在一个方法内部定义的变量都存储在栈中，当这个函数结束后，其对应的栈就会被回收，此时在其方法体中定义的变量将不存在了，因此 return 在返回时不是直接返回变量的值，而是复制一份，然后返回。因此，对于基本类型的数据，在 finally 块中改变 return 的值对返回值没有任何影响，而对引用类型的数据会有影响。下面通过一个例子（示例 3）来说明这个问题：

```
public class Test {
    public static int testFinally1() {
        int result = 1;
        try {
            result = 2;
            return result;
        } catch (Exception e) {
            return 0;
        } finally {
            result = 3;
            System.out.println("execute finally2");
        }
    }

    public static StringBuffer testFinally2() {
        StringBuffer s = new StringBuffer("Hello");
    }
}
```

```

        try {
            return s;
        } catch (Exception e) {
            return null;
        } finally {
            s.append(" World");
            System.out.println("execute finally2");
        }
    }

    public static void main(String[] args) {
        int resultVal = testFinally1();
        System.out.println(resultVal);
        StringBuffer resultRef = testFinally2();
        System.out.println(resultRef);
    }
}

```

程序运行结果为：

```

execute finally1
2
execute finally2
Hello World

```

程序在执行到 `return` 时会首先将返回值存储在一个指定的位置，其次去执行 `finally` 块，之后再返回。在方法 `testFinally1` 中调用 `return` 前，先把 `result` 的值 1 存储在一个指定的位置，然后再去执行 `finally` 块中的代码，此时修改 `result` 的值将不会影响到程序的返回结果。`testFinally2` 中，在调用 `return` 前首先把 `s` 存储到一个指定的位置，由于 `s` 为引用类型，因此在 `finally` 块中修改 `s` 将会修改程序的返回结果。

引申：出现在 Java 程序中的 `finally` 块是不是一定会被执行？

答案：不一定会被执行。下面给出两个 `finally` 代码块不会被执行的例子。

1) 当程序在进入 `try` 语句块之前就出现异常时，会直接结束，不会执行 `finally` 块中的代码，示例如下：

```

public class Test {
    public static void testFinally() {
        int i = 5/0;
        try {
            System.out.println("try block");
        } catch (Exception e) {
            System.out.println("catch block");
        } finally {
            System.out.println("finally block");
        }
    }

    public static void main(String[] args) {
        testFinally();
    }
}

```

程序运行结果为：

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Test.testFinally(Test.java:3)
    at Test.main(Test.java:13)
```

程序在执行 `int i = 5/0` 时会抛出异常，导致没有执行 `try` 块，因此 `finally` 块也就不会被执行。

2) 当程序在 `try` 块中强制退出时也不会去执行 `finally` 块中的代码，示例如下：

```
public class Test {
    public static void testFinally() {
        try {
            System.out.println("try block");
            System.exit(0);
        } catch (Exception e) {
            System.out.println("catch block");
        } finally {
            System.out.println("finally block");
        }
    }

    public static void main(String[] args) {
        testFinally();
    }
}
```

程序运行结果为：

```
try block
```

上例在 `try` 块中通过调用 `System.exit(0)` 强制退出了程序，因此导致 `finally` 块中的代码没有被执行。

常见笔试题：

下面程序的运行结果是什么？

```
public class Foo {
    public static void main(String[] args) {
        try {
            return;
        } finally {
            System.out.println("Finally");
        }
    }
}
```

A. Finally B. 编译失败 C. 代码正常运行但没有任何输出 D. 运行时抛出异常

答案：A。见上面讲解。

4.6.2 异常处理的原理是什么

异常是指程序运行时（非编译时）所发生的非正常情况或错误，当程序违反了语义规则

时, JVM 就会将出现的错误表示为一个异常并抛出。这个异常可以在 catch 程序块中进行捕获, 然后进行处理。而异常处理的目的是为了程序的安全性与鲁棒性。

Java 语言把异常当作对象来处理, 并定义了一个基类 (java.lang.Throwable) 作为所有异常的父亲。在 Java API 中, 已经定义了许多异常类, 这些异常类分为 Error (错误) 和 Exception (异常) 两大类。

违反语法规则包括两种情况: 一种是 Java 类库内置的语义检查, 例如当数组下标越界时, 会引发 IndexOutOfBoundsException, 当访问 null 的对象时, 会引发 NullPointerException; 另一种情况是 Java 允许开发人员扩展这种语义检查, 开发人员可以创建自己的异常类 (所有异常都是 Java.lang.Throwable 的子类), 并自由选择何时用 throw 关键字抛出异常。

常见笔试题:

下列异常中, 能使用 throw 抛出的是 ()。

- A. Error B. Event C. Object D. Throwable E. Exception
F. RuntimeException

答案: A、D、E、F。其中 Throwable 为异常处理的基类, Error、Exception 和 RuntimeException 都是 Throwable 的子类, 因此都能使用 throw 抛出。

4.6.3 运行时异常和普通异常有什么区别

Java 提供了两种错误的异常类, 分别为 Error 和 Exception, 且它们拥有共同的父类——Throwable。

Error 表示程序在运行期间出现了非常严重的错误, 并且该错误是不可恢复的, 由于这属于 JVM 层次的严重错误, 因此这种错误是会导致程序终止执行的。此外, 编译器不会检查 Error 是否被处理, 因此在程序中不推荐去捕获 Error 类型的异常, 主要原因是运行时异常多是由于逻辑错误导致的, 属于应该解决的错误, 也就是说, 一个正确的程序中是不应该存在 Error 的。OutOfMemoryError、ThreadDeath 等都属于错误。当这些异常发生时, JVM 一般会选择将线程终止。

Exception 表示可恢复的异常, 是编译器可以捕捉到的。它包含两种类型: 检查异常 (checked exception) 和运行时异常 (runtime exception)。

(1) 检查异常

检查异常是在程序中最经常碰到的异常。所有继承自 Exception 并且不是运行时异常的异常都是检查异常, 比如最常见的 IO 异常和 SQL 异常。这种异常都发生在编译阶段, Java 编译器强制程序去捕获此类型的异常, 即把可能会出现这些异常的代码放到 try 块中, 把对异常的处理的代码放到 catch 块中。这种异常一般在如下几种情况中使用:

1) 异常的发生并不会导致程序出错, 进行处理后可以继续执行后续的操作, 例如, 当连接数据库失败后, 可以重新连接后进行后续操作。

2) 程序依赖于不可靠的外部条件, 例如系统 IO。

(2) 运行时异常对于运行时异常, 编译器没有强制对其进行捕获并处理。如果不对这种异常进行处理, 当出现这种异常时, 会由 JVM 来处理, 例如 NullPointerException 异常, 它就是运行时异常。在 Java 语言中, 最常见的运行时异常包括 NullPointerException (空指针异常)、ClassCastException (类型转换异常)、ArrayIndexOutOfBoundsException (数组越界异常)、ArrayStoreException (数组存储异常)、BufferOverflowException (缓冲区溢出异常)、ArithmeticException

tion (算术异常) 等。

出现运行时异常后, 系统会把异常一直往上层抛出, 直到遇到处理代码为止。若没有处理块, 则抛到最上层; 如果是多线程就用 `Thread.run()` 方法抛出, 如果是单线程, 就用 `main()` 方法抛出。抛出之后, 如果是线程, 那么这个线程也就退出了。如果是主程序抛出的异常, 那么整个程序也就退出了。所以, 如果不对运行时的异常进行处理, 后果是非常严重的, 一旦发生, 要么是线程中止, 要么是主程序终止。

在使用异常处理时, 还需要注意以下几个问题:

1) Java 异常处理用到了多态的概念, 如果在异常处理过程中, 先捕获了基类, 然后再捕获子类, 那么捕获子类的代码块将永远不会被执行。因此, 在进行异常捕获时, 正确的写法是: 先捕获子类, 再捕获基类的异常信息, 示例如下:

正确的写法	错误的写法
<pre>try //access db code catch (SQLException e1) //deal with this exception catch (Exception e2) </pre>	<pre>try //access db code catch (Exception e1) //deal with this exception catch (SQLException e2) </pre>

2) 尽早抛出异常, 同时对捕获的异常进行处理, 或者从错误中恢复, 或者让程序继续执行。对捕获的异常不进行任何处理是一个非常不好的习惯, 这样将非常不利于调试。但也不是抛出异常越多越好, 对于有些异常类型, 例如运行时异常, 实际上根本不必处理。

3) 可以根据实际的需求自定义异常类, 这些自定义的异常类只要继承自 `Exception` 类即可。

4) 异常能处理就处理, 不能处理就抛出。对于一般异常, 如果不能进行行之有效的处理, 最好转换为运行时异常抛出。对于最终没有处理的异常, JVM 会进行处理。

常见笔试题:

1. 下面程序能否编译通过? 如果把 `ArithmeticException` 换成 `IOException` 呢?

```
public class ExceptionTypeTest {
    public void doSomething() throws ArithmeticException {
        System.out.println();
    }

    public static void main() {
        ExceptionTypeTest ett = new ExceptionTypeTest();
        ett.doSomething();
    }
}
```

答案: 能。由于 `ArithmeticException` 属于运行时异常, 编译器没有强制对其进行捕获并处理, 因此编译可以通过。但是如果换成 `IOException` 后, 由于 `IOException` 属于检查异常, 编译器强制去捕获此类型的异常, 因此如果不对异常进行捕获将会有编译错误。

2. 异常包含下列哪些内容? ()

- A. 程序中的语法错误
- B. 程序的编译错误
- C. 程序执行过程中遇到的事先没有预料到的情况
- D. 程序事先定义好的可能出现的意外情况

答案：C。见上面讲解。

3. 下列关于异常的说法中，正确的是（ ）。

A. 一旦出现异常，程序运行就终止了

B. 如果一个方法申明将抛出某个异常，它就必须真的抛出那个异常

C. 在 catch 子句中匹配异常是一种精确匹配

D. 可能抛出系统异常的方法是不需要申明异常的

答案：D。见上面讲解。

4.7 输入输出流

4.7.1 Java IO 流的实现机制是什么

在 Java 语言中，输入和输出都被称为抽象的流，流可以被看作一组有序的字节集合，即数据在两设备之间的传输。

流的本质是数据传输，根据处理数据类型不同，流可以分为两大类：字节流和字符流。字节流以字节（8 bit）为单位，包含两个抽象类：InputStream（输入流）和 OutputStream（输出流）。字符流以字符（16 bit）为单位，根据码表映射字符，一次可以读多个字节，它包含两个抽象类：Reader（输入流）和 Writer（输出流）。字节流和字符流最主要的区别为：字节流在处理输入输出时不会用到缓存，而字符流用到了缓存。每个抽象类都有很多具体的实现类，在这里就不详细介绍了。图 4-6 主要介绍 Java 中 IO 的设计理念。Java IO 类在设计时采用了 Decorator（装饰者）设计模式，以 InputStream 为例，介绍 Decorator 设计模式在 IO 类中的使用如下。

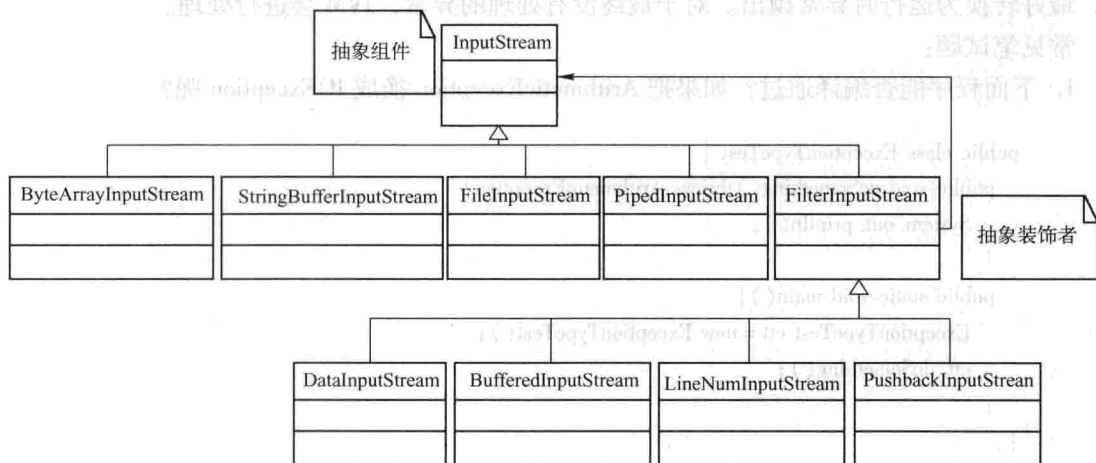


图 4-6 IO 设计类图

其中，ByteArrayInputStream、StringBufferInputStream、FileInputStream 和 PipedInputStream 是 Java 提供的最基本的对流进行处理类，FilterInputStream 为一个封装类的基类，可以对基本的 IO 类进行封装，通过调用这些类提供的基本的流操作方法来实现在更复杂的流操作。

使用这种设计模式的好处是可以在运行时动态地给对象添加一些额外的职责，与使用继承的设计方法相比，该方法具有很好的灵活性。

假如现在要设计一个输入流的类，该类的作用为在读文件时把文件中的大写字母转换成小写字母，把小写字母转换为大写字母。在设计时，可以通过继承抽象装饰者类（FilterInputStream）来实现一个装饰类，通过调用 InputStream 类或其子类提供的一些方法再加上逻辑判断代码从而可以很简单地实现这个功能，示例如下：

```
class MyOwnInputStream extends FilterInputStream {
    public MyOwnInputStream(InputStream in) {
        super(in);
    }

    public int read() throws IOException {
        int c = 0;
        if ( (c = super.read()) != -1) {
            //把小写转换为大写
            if(Character.isLowerCase((char)c))
                return Character.toUpperCase((char)c);
            //把大写转换为小写
            else if(Character.isUpperCase((char)c))
                return Character.toLowerCase((char)c);
            //如果不是字母,保持不变
            else
                return c;
        }
        else {
            return -1;
        }
    }
}

public class Test {
    public static void main(String[] args) {
        int c;
        try {
            InputStream is = new MyOwnInputStream( new BufferedInputStream( new FileInputStream("
test.txt" )));
            while((c = is.read()) >= 0) {
                System.out.print((char)c);
            }
            is.close();
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

当文件 test.txt 中的内容为：aaaBBBcccDDD123 时，程序输出为：AAAbbbCCCddd123。

常见笔试题：Java 中有几种类型的流？

答案：常见的流有两种，分别为字节流与字符流。其中，字节流继承于 InputStream 与 OutputStream，字符流继承于 Reader 与 Writer。在 java.io 包中还有许多其他的流，流的作用主要是为了改善程序性能并且使用方便。

4.7.2 管理文件和目录的类是什么

对文件或目录进行管理与操作在编程中有着非常重要的作用，Java 提供了一个非常重要的类（File）来管理文件和文件夹，通过类不仅能够查看文件或目录的属性，而且还可以实现对文件或目录的创建、删除与重命名等操作。下面主要介绍 File 类中常用的几个方法，见表 4-7。

表 4-7 File 类常用的方法

方 法	作 用
File(String pathname)	根据指定的路径创建一个 File 对象
createNewFile()	若目录或文件存在，则返回 false，否则创建文件或文件夹
delete()	删除文件或文件夹
isFile()	判断这个对象表示的是否是文件
isDirectory()	判断这个对象表示的是否是文件夹
listFiles()	若对象代表目录，则返回目录中所有文件的 File 对象
mkdir()	根据当前对象指定的路径创建目录
exists()	判断对象对应的文件是否存在

常见笔试题：

如何列出某个目录下的所有目录和文件？

答案：假设目录“C:\testDir1”下有两个文件夹（dir1 和 dir2）和一个文件 file1.txt，实现代码如下：

```
import java.io. File;
public class Test {
    public static void main(String[] args) {
        File file = new File("C:\testDir");
        // 判断目录是否存在
        if (! file. exists()) {
            System. out. println(" directory is empty");
            return;
        }
        File[] fileList = file. listFiles();
        for (int i = 0; i < fileList. length; i++) {
            //判断是否为目录
            if (fileList[i]. isDirectory()) {
                System. out. println(" directory is: " + fileList[i]. getName());
            } else {
                System. out. println(" file is: " + fileList[i]. getName());
            }
        }
    }
}
```

程序运行结果为：

directory is: dir1

```
directory is: dir2
file is:file1.txt
```

4.7.3 Java Socket 是什么

网络上的两个程序通过一个双向的通信连接实现数据的交换，这个双向链路的一端称为一个 Socket。Socket 也称为套接字，可以用来实现不同虚拟机或不同计算机之间的通信。在 Java 语言中，Socket 可以分为两种类型：面向连接的 Socket 通信协议（TCP，Transmission Control Protocol，传输控制协议）和面向无连接的 Socket 通信协议（UDP，User Datagram Protocol，用户数据报协议）。任何一个 Socket 都是由 IP 地址和端口号唯一确定的，如图 4-7 所示。

基于 TCP 的通信过程如下：首先，Server（服务器）端 Listen（监听）指定的某个端口（建议使用大于 1024 的端口）是否有连接请求；其次，Client（客户）端向 Server 端发出 Connect（连接）请求；最后，Server 端向 Client 端发回 Accept（接受）消息。一个连接就建立起来了，会话随即产生。Server 端和 Client 端都可以通过 Send、Write 等方法与对方通信。

Socket 的生命周期可以分为 3 个阶段：打开 Socket、使用 Socket 收发数据和关闭 Socket。在 Java 语言中，可以使用 ServerSocket 来作为服务器端，Socket 作为客户端来实现网络通信。

常见笔试题：

用 Socket 实现客户端和服务端通信，要求客户发送数据后能够回显相同的数据。

答案：首先，创建一个名为 Server.java 的服务器端代码，如下所示。

```
import java.net.*;
import java.io.*;
class Server {
    public static void main(String[] args) {
        BufferedReader br = null;
        PrintWriter pw = null;
        try {
            ServerSocket server = new ServerSocket(2000);
            Socket socket = server.accept();
            // 获取输入流
            br = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            // 获取输出流
            pw = new PrintWriter(socket.getOutputStream(), true);
            String s = br.readLine(); // 获取接收的数据
            pw.println(s);           // 发送相同的数据给客户端
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                br.close();
                pw.close();
            }
        }
    }
}
```

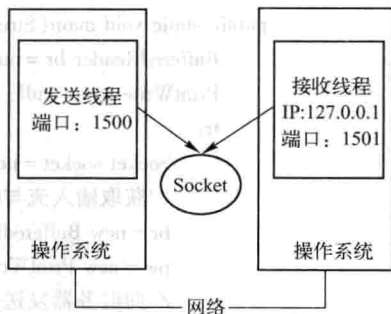


图 4-7 Socket 原理图

```
    } catch (Exception e) {
    }
}
```

其次，创建一个 Client.java 的客户端程序，如下所示：

```
import java.net.*;
import java.io.*;
class Client {
    public static void main( String[] args) {
        BufferedReader br = null;
        PrintWriter pw = null;
        try {
            Socket socket = new Socket("localhost", 2000);
            //获取输入流与输出流
            br = new BufferedReader( new InputStreamReader( socket.getInputStream()));
            pw = new PrintWriter( socket.getOutputStream(), true);
            //向服务器发送数据
            pw.println("Hello");
            String s = null;
            while ( true) {
                s = br.readLine();
                if ( s != null)
                    break;
            }
            System.out.println(s);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                br.close();
                pw.close();
            } catch (Exception e) {
            }
        }
    }
}
```

最后启动服务器端程序，然后运行客户端程序，客户端将会把从服务器端转发过来的“Hello”打印出来。

4.7.4 Java NIO 是什么

在非阻塞 IO (Nonblocking IO, NIO) 出现之前，Java 是通过传统的 Socket 来实现基本的网络通信功能的。以服务器端为例，其实现基本流程如图 4-8 所示。

如果客户端还没有对服务器端发起连接请求，那么 accept



图 4-8 Socket 使用流程

就会阻塞（阻塞指的是暂停一个线程的执行以等待某个条件发生，例如某资源就绪）。如果连接成功，当数据还没有准备好时，对 read 的调用同样会阻塞。当要处理多个连接时，就需要采用多线程的方式，由于每个线程都拥有自己的栈空间，而且由于阻塞会导致大量线程进行上下文切换，使得程序的运行效率非常低下，因此在 J2SE 1.4 中引入了 NIO 来解决这个问题。

NIO 通过 Selector、Channel 和 Buffer 来实现非阻塞的 IO 操作，其实现原理如图 4-9 所示。

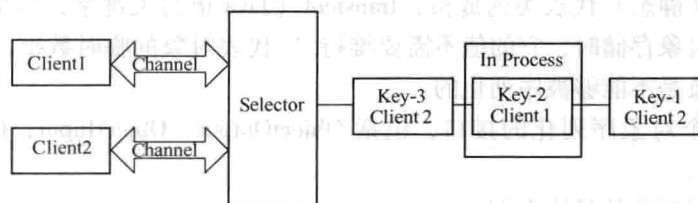


图 4-9 NIO 实现原理图

NIO 非阻塞的实现主要采用了 Reactor（反应器）设计模式，这个设计模式与 Observer（观察者）设计模式类似，只不过 Observer 设计模式只能处理一个事件源，而 Reactor 设计模式可以用来处理多个事件源。

在上图中，Channel 可以被看作一个双向的非阻塞的通道，在通道的两边都可以进行数据的读写操作。Selector 实现了用一个线程来管理多个通道（采用了复用与解复用的方式使得一个线程能够管理多个通道，即可以把多个流合并成为一个流，或者把一个流分成多个流的方式），它类似于一个观察者。在实现时，把需要处理的 Channel 的 IO 事件（例如 connect、read 或 write 等）注册给 Selector。Selector 内部的实现原理为：对所有注册的 Channel 进行轮询访问，一旦轮询到一个 Channel 有注册的事件发生，例如有数据来了，它就通过传回 SelectionKey 的方式来通知开发人员对 Channel 1 进行数据的读或写操作。Key（由 SelectionKey 类表示）封装一个特定 Channel 和一个特定的 selector 之间的关系。这种通过轮询的方式在处理多线程请求时不需要上下文的切换，而采用多线程的实现方式在线程之间切换时需要上下文的切换，同时也需要进行压栈与弹栈操作。因此，NIO 有较高的执行效率。

Buffer 用来保存数据，可以用来存放从 Channel 1 读取的数据，也可以存放使用 Channel 1 进行发送的数据。Java 提供了多种不同类型的 Buffer，例如 ByteBuffer、CharBuffer 等，通过 Buffer，大大简化了开发人员对流数据的管理。

NIO 在网络编程中有着非常重要的作用，与传统的 Socket 方式相比，由于 NIO 采用了非阻塞的方式，在处理大量并发请求时，使用 NIO 要比使用 Socket 效率高很多。

4.7.5 什么是 Java 序列化

Java 提供了两种对象持久化的方式，分别为序列化和外部序列化。

(1) 序列化 (Serialization)

在分布式环境下，当进行远程通信时，无论是何种类型的数据，都会以二进制序列的形式在网络上传送。序列化是一种将对象以一连串的字节的描述的过程，用于解决在对对象流进行读写操作时所引发的问题。序列化可以将对象的状态写在流里进行网络传输，或者保存到文件、数据库等系统里，并在需要时把该流读取出来重新构造一个相同的对象。

如何实现序列化呢？其实，所有要实现序列化的类都必须实现 Serializable 接口，Serializ-

able 接口位于 java.lang 包中，它里面没有包含任何方法。使用一个输出流（例如 FileOutputStream）来构造一个 ObjectOutputStream（对象流）对象，紧接着，使用该对象的 writeObject（Object obj）方法就可以将 obj 对象写出（即保存其状态），要恢复时可以使用其对应的输入流。

序列化有以下两个特点：

- 1) 如果一个类能被序列化，那么它的子类也能够被序列化。
- 2) 由于 static（静态）代表类的成员，transient（Java 语言关键字，如果用 transient 声明一个实例变量，当对象存储时，它的值不需要维持。）代表对象的临时数据，因此被声明为这两种类型的数据成员是不能够被序列化的。

Java 提供了多个对象序列化的接口，包括 ObjectOutputStream、ObjectInputStream 和 ObjectInputStream。

下面给出一个序列化的具体实例：

```
import java.io. FileInputStream;
import java.io. FileOutputStream;
import java.io. ObjectInputStream;
import java.io. ObjectOutputStream;
import java.io. Serializable;

public class People implements Serializable {
    private String name;
    private int age;
    public People () {
        this.name = "lili";
        this.age = 20;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getName() {
        return this.name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public static void main(String[] args) {
        People p = new People();
        ObjectOutputStream oos = null;
        ObjectInputStream ois = null;
        try {
            FileOutputStream fos = new FileOutputStream("perple.out");
            oos = new ObjectOutputStream(fos);
            oos.writeObject(p);
            oos.close();
        } catch (Exception ex) {
```

```

    People p1;
    try {
        FileInputStream fis = new FileInputStream("perple.out");
        ois = new ObjectInputStream(fis);
        p1 = (People) ois.readObject();
        System.out.println("name:" + p1.getName());
        System.out.println("age:" + p1.getAge());
        ois.close();
    } catch (Exception ex) {
    }
}

```

程序运行结果为:

```

name:lili
age:20

```

由于序列化的使用会影响系统的性能,因此如果不是必须要使用序列化,应尽可能不要使用序列化。那么在什么情况下需要使用该序列化呢?

- 1) 需要通过网络来发送对象,或对象的状态需要被持久化到数据库或文件中。
- 2) 序列化能实现深复制,即可以复制引用的对象。

与序列化相对的是反序列化,它将流转换为对象。在序列化与反序列化的过程中,serialVersionUID 起着非常重要的作用,每个类都有一个特定的 serialVersionUID,在反序列化的过程中,通过 serialVersionUID 来判定类的兼容性。如果待序列化的对象与目标对象的 serialVersionUID 不同,那么在反序列化时就会抛出 InvalidClassException 异常。作为一个好的编程习惯,最好在序列化的类中显式地声明 serialVersionUID (该字段必须定义为 static final)。自定义 serialVersionUID 主要有如下 3 个优点。

1) 提高程序的运行效率。如果在类中未显式声明 serialVersionUID,那么在序列化时会通过计算得到一个 serialVersionUID 值。通过显式声明 serialVersionUID 的方式省去了计算的过程,因此提高了程序的运行效率。

2) 提高程序不同平台上的兼容性。由于各个平台的编译器在计算 serialVersionUID 时完全有可能会采用不同的计算方式,这就会导致在一个平台上序列化的对象在另外一个平台上将无法实现反序列化的操作。通过显式声明 serialVersionUID 的方法完全可以避免该问题的发生。

3) 增强程序各个版本的兼容性。在默认情况下,每个类都有唯一的 serialVersionUID,因此,当后期对类进行修改时(例如加入新的属性),类的 serialVersionUID 值将会发生变化,这将会导致类在修改前对象序列化的文件在修改后将无法进行反序列化操作。同样,通过显式声明 serialVersionUID 也会解决这个问题。

(2) 外部序列化

Java 语言还提供了另外一种方式来实现对象持久化,即外部序列化。其接口如下:

```

public interface Externalizable extends Serializable {
    void readExternal(ObjectInput in);?
    void writeExternal(ObjectOutput out);?
}

```

外部序列化与序列化主要的区别在于序列化是内置的 API，只需要实现 Serializable 接口，开发人员不需要编写任何代码就可以实现对象的序列化，而使用外部序列化时，Externalizable 接口中的读写方法必须由开发人员来实现。因此与实现 Serializable 接口的方法相比，使用 Externalizable 编写程序的难度更大，但是由于把控制权交给了开发人员，在编程时有更多的灵活性，对需要持久化的那些属性可以进行控制，可能会提高性能。

引申：在用接口 Serializable 实现序列化时，这个类中的所有属性都会被序列化，那么怎样才能实现只序列化部分属性呢？

一种方法为实现 Externalizable 接口，开发人员可以根据实际需求来实现 readExternal 与 writeExternal 方法来控制序列化与反序列化所使用的属性，这种方法的缺点为增加了编程的难度。另一种方法为使用关键字 transient 来控制序列化的属性。被 transient 修饰的属性是临时的，不会被序列化。因此，可以通过把不需要被序列化的属性用 transient 来修饰。

常见笔试题：

```
import java.io.Serializable;
public class DataObject implements Serializable{
    private static int i=0;
    private String word = "";
    public static void setI(int i) {
        DataObject.i = i;
    }
    public void setWord(String word) {
        this.word = word;
    }
}
```

创建一个如下方式的 DataObject：DataObject object = new DataObject(); object.setWord("123");object.setI(2); 将此对象序列化文件，并在另一个 JVM 中读取文件，进行反序列化，请问此时读出的 DataObject 对象中的 word 和 i 的值分别是（ ）。
A. “”，0 B. “”，2 C. “123”，2 D. “123”，0

答案：D。Java 在序列化时不会实例化 static 变量，因此上述代码只实例化了 word，而没有实例化 i。在反序列化时只能读取到 word 的值，i 为默认值。

4.7.6 System.out.println() 方法使用需要注意哪些问题

Java 中的 System.out.println() 方法提供了一种非常有效简单的方法来实现控制台的输出，该方法默认接收一个字符串类型的变量作为参数。当然，在使用时可以传递任意能够转换为 String 类型的变量作为参数（例如基本类型 int，或者一个实现 toString 方法的自定义类等），示例如下：

```
class People {
    private String name;
    private int age;
    public People() {
        this.name = "何昊";
        this.age = 26;
    }
    public String toString() {
```

```

        return "name:" + this.name + " age:" + this.age;
    }
}

class Test {
    public static void main(String[] args) {
        System.out.println(new Poeples());
        System.out.println(1 + 2 + "");
        System.out.println("" + 1 + 2);
    }
}

```

程序运行结果为：

```

name:何昊 age:26
3
12

```

对于第一个输出语句来说，由于传入的参数是一个对象，因此会调用这个对象的 `toString()` 方法，把返回的字符串打印出来。对于第二个输出语句来说，参数中的 `+` 会由左到右顺序计算。首先计算 `1 + 2`，由于它们都是整型变量，因此计算结果为 `3`，接着计算 `3 + ""`，由于 `""` 是字符串，因此首先会把 `3` 转换为字符串，其次执行加操作，计算结果为 `"3"`，因此输出结果为 `3`。对于最后一个输出语句来说，首先计算 `"" + 1`，会把 `1` 转换为字符串，其次执行加操作，计算结果为 `"1"`，同理，接着计算 `"1" + 2` 结果为 `"12"`，因此输出结果为 `12`。

4.8 Java 平台与内存管理

4.8.1 为什么说 Java 是平台独立性语言

平台独立性是指可以在一个平台上编写和编译程序，而在其他平台上运行。保证 Java 具有平台独立性的机制为“中间码”和“Java 虚拟机 (Java Virtual Machine, JVM)”。Java 程序被编译后不是生成能在硬件平台上可执行的代码，而是生成了一个“中间码”。不同的硬件平台上会安装有不同的 JVM，由 JVM 来负责把“中间码”翻译成硬件平台能执行的代码。由此可以看出 JVM 不具有平台独立性，而是与硬件平台相关的。

解释执行过程分三步进行：代码的装入、代码的校验和代码的执行。装入代码的工作由“类装载机”完成。被装入的代码由字节码校验器进行检查。

Java 字节码的执行也分为两种方式：即时编译方式与解释执行方式，即时编译方式指的是解释器先将字节码编译成机器码，然后再执行该机器码。解释执行方式指的是解释器通过每次解释并执行一小段代码来完成 Java 字节码程序的所有操作。通常采用的是解释执行方式。

而在 C/C++ 语言中，编译后的代码只能在特定的硬件上执行，换个硬件平台这些代码就无法执行了，从而也导致了 C/C++ 没有跨平台的特性。但 C/C++ 有更高的执行效率。

常见笔试题：

1. 一个 Java 程序运行从上到下的环境次序是 ()。
 - A. 操作系统、Java 程序、JRE/JVM、硬件
 - B. JRE/JVM、Java 程序、硬件、操作系统
 - C. Java 程序、JRE/JVM、操作系统、硬件
 - D. Java 程序、操作系统、JRE/JVM、硬件

答案：C。见上面讲解。

2. 下列说法中, 正确的是 ()。

A. Java 程序经编译后会产生机器码

B. Java 程序经编译后会产生字节码

C. Java 程序经编译后会产生 DLL

D. 以上都不正确

答案: B。Java 文件被 javac 指令编译为 .class 后缀的字节码文件, 再由 JVM 执行。

4.8.2 Java 平台与其他语言平台有哪些区别

Java 平台是一个纯软件的平台, 这个平台可以运行在一些基于硬件的平台 (例如 Linux、Windows 等) 之上。Java 平台主要包含两个模块: JVM 与 Java API (Application Program Interface, 应用程序接口)。

JVM 是一个虚构出来的计算机, 用来把 Java 编译生成的中间代码转换为机器可以识别的编码并运行。它有自己完善的硬件架构, 例如处理器、堆栈、寄存器等, 还具有相应的指令系统, 它屏蔽了与具体操作系统平台相关的信息, 使得 Java 程序只需生成在 JVM 上运行的目标代码 (即字节码), 就可以在多种平台上不加修改地顺利运行。每当一个 Java 程序运行时, 都会有一个对应的 JVM 实例, 只有当程序运行结束后, 这个 JVM 才会退出。JVM 实例通过调用类的 main() 方法来启动一个 Java 程序, 而这个 main() 方法必须是公有的、静态的且返回值为 void 的方法, 该方法接受一个字符串数组的参数, 只有同时满足这些条件才可以作为程序的入口方法。

Java API 是 Java 为了方便开发人员进行开发而设计的, 它提供了许多非常有用的接口, 这些接口也是用 Java 语言编写的, 并且运行在 JVM 上。

4.8.3 JVM 加载 class 文件的原理机制是什么

Java 语言是一种具有动态性的解释型语言, 类 (class) 只有被加载到 JVM 中后才能运行。当运行指定程序时, JVM 会将编译生成的 .class 文件按照需求和一定的规则加载到内存中, 并组织成为一个完整的 Java 应用程序。这个加载过程是由类加载器来完成的, 具体来说, 就是由 ClassLoader 和它的子类来实现的。类加载器本身也是一个类, 其实质是把类文件从硬盘读取到内存中。

类的加载方式分为隐式加载与显式加载两种。隐式加载指的是程序在使用 new 等方式创建对象时, 会隐式地调用类的加载器把对应的类加载到 JVM 中。显式加载指的是通过直接调用 class.forName() 方法来把所需的类加载到 JVM 中。

任何一个工程项目都是由许多个类组成的, 当程序启动时, 只把需要的类加载到 JVM 中, 其他类只有被使用到的时候才会被加载, 采用这种方法, 一方面可以加快加载速度, 另外一方面可以节约程序运行过程中对内存的开销。此外, 在 Java 语言中, 每个类或接口都对应一个 .class 文件, 这些文件可以被看成一个一个可以被动态加载的单元, 因此当只有部分类被修改时, 只需要重新编译变化的类即可, 而不需要重新编译所有文件, 因此加快了编译速度。

在 Java 语言中, 类的加载是动态的, 它并不会一次性将所有类全部加载后再运行, 而是保证程序运行的基础类 (例如基类) 完全加载到 JVM 中, 至于其他类, 则在需要时才加载。在 Java 语言中, 可以把类分为 3 类: 系统类、扩展类和自定义类。Java 针对这 3 种不同的类提供了 3 种类型的加载器, 这 3 种加载器的关系如下:

Bootstrap Loader - 负责加载系统类(jre/lib/rt.jar 的类)

-- ExtClassLoader - 负责加载扩展类(jar/lib/ext/*.jar 的类)

```

|
-- AppClassLoader - 负责加载应用类
(classpath 指定的目录或 jar 中的类)

```

以上这 3 个类是如何协调工作来完成类的加载呢？其实，它们是通过委托的方式实现的。具体而言，就是当有类需要被加载时，类加载器会请求父类来完成这个载入工作，父类会使用其自己的搜索路径来搜索需要被载入的类，如果搜索不到，才会由子类按照其搜索路径来搜索待加载的类。下例可以充分说明类加载器的工作原理：

```

public class TestLoader {
    public static void main(String[] args) throws Exception {
        //调用 class 加载器
        ClassLoader clApp = TestLoader.class.getClassLoader();
        System.out.println(clApp);
        //调用上一层 Class 加载器
        ClassLoader clExt = clApp.getParent();
        System.out.println(clExt);
        //调用根部 Class 加载器
        ClassLoader clBoot = clExt.getParent();
        System.out.println(clBoot);
    }
}

```

程序运行结果为：

```

sun.misc.Launcher $AppClassLoader@19821f
sun.misc.Launcher $ExtClassLoader@addbfl
null

```

从上例可以看出，TestLoader 类是由 AppClassLoader 来加载的。另外需要说明的一点是，由于 Bootstrap Loader 是用 C++ 语言来实现的，因此，在 Java 语言中是看不到它的，所以此时程序会输出 null。

类加载的主要步骤分为以下 3 步：

- 1) 装载。根据查找路径找到相对应的 class 文件，然后导入。
- 2) 链接。链接又可以分为 3 个小的步骤，具体如下。
 - ① 检查。检查待加载的 class 文件的正确性。
 - ② 准备。给类中的静态变量分配存储空间。
 - ③ 解析。将符号引用转换成直接引用（这一步是可选的）。
- 3) 初始化。对静态变量和静态代码块执行初始化工作。

4.8.4 什么是 GC

在 Java 语言中，垃圾回收（Garbage Collection, GC）是一个非常重要的概念，它的主要作用是回收程序中不再使用的内存。在使用 C/C++ 语言进行程序开发时，开发人员必须非常仔细地管理好内存的分配与释放，如果忘记或者错误地释放内存往往会导致程序运行不正常甚至是程序崩溃。为了减轻开发人员的工作，同时增加系统的安全性与稳定性，Java 语言提供了垃圾回收器来自动检测对象的作用域，可自动地把不再被使用的存储空间释放掉。具体而言，垃圾回收器要负责完成 3 项任务：分配内存、确保被引用对象的内存不被错误地回收以及回收

不再被引用的对象的内存空间。

垃圾回收器的存在一方面把开发人员从释放内存的复杂工作中解脱出来，提高了开发人员的生产效率；另一方面，对开发人员屏蔽了释放内存的方法，可以避免因开发人员错误地操作内存而导致应用程序的崩溃，保证了程序的稳定性。但是，垃圾回收也带来了问题，为了实现垃圾回收，垃圾回收器必须跟踪内存的使用情况，释放没用的对象，在完成内存的释放后还需要处理堆中的碎片，这些操作必定会增加 JVM 的负担，从而降低程序的执行效率。

对对象而言，如果没有任何变量去引用它，那么该对象将不可能被程序访问，因此可以认为它是垃圾信息，可以被回收。只要有一个以上的变量引用该对象，该对象就不会被垃圾回收。

对于垃圾回收器来说，它使用有向图来记录和管理堆内存中的所有对象，通过这个有向图就可以识别哪些对象是“可达的”（有引用变量引用它就是“可达的”），哪些对象是“不可达的”（没有引用变量引用它就是不可达的），所有“不可达”对象都是可被垃圾回收的，示例如下：

```
public class Test {
    public static void main(String[] a) {
        Integer i1 = new Integer(1);
        Integer i2 = new Integer(2);
        i2 = i1;
        //some other code
    }
}
```

上述代码在执行到 `i2 = i1` 后，内存的引用关系如图 4-10 所示。

此时，如果垃圾回收器正在进行垃圾回收操作，在遍历上述有向图时，资源 2 所占的内存是不可达的，垃圾回收器就会认为这块内存已经不会再被使用了，因此就会回收该块内存空间。

垃圾回收都是依据一定的算法进行的，下面介绍其中几种常用的垃圾回收算法。

(1) 引用计数算法 (Reference Counting Collector)

引用计数作为一种简单但是效率较低的方法，其主要原理如下：在堆中对每个对象都有一个引用计数器；当对象被引用时，引用计数器加 1；当引用被置为空或离开作用域的时，引用计数减 1，由于这种方法无法解决相互引用的问题，因此 JVM 没有采用这个算法。

(2) 追踪回收算法 (Tracing Collector)

追踪回收算法利用 JVM 维护的对象引用图，从根结点开始遍历对象的应用图，同时标记遍历到的对象。当遍历结束后，未被标记的对象就是目前已不被使用的对象，可以被回收了。

(3) 压缩回收算法 (Compacting Collector)

压缩回收算法的主要思路如下：把堆中活动的对象移动到堆中一端，这样就会在堆中另外一端留出很大的一块空闲区域，相当于对堆中的碎片进行了处理。虽然这种方法可以大大简化消除堆碎片的工作，但是每次处理都会带来性能的损失。

(4) 复制回收算法 (Copying Collector)

复制回收算法的主要思路如下：把堆分成两个大小相同的区域，在任何时刻，只有其中的

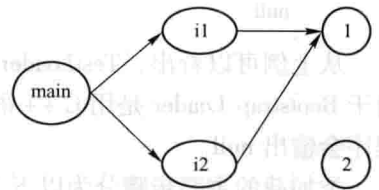


图 4-10 对象引用关系

一个区域被使用，直到这个区域的被消耗完为止，此时垃圾回收器会中断程序的执行，通过遍历的方式把所有活动的对象复制到另外一个区域中，在复制的过程中它们是紧挨着布置的，从而可以消除内存碎片。当复制过程结束后程序会接着运行，直到这块区域被使用完，然后再采用上面的方法继续进行垃圾回收。

这个算法的优点是在进行垃圾回收的同时对对象的布置也进行了安排，从而消除了内存碎片。但是这也付出了很高的代价：对于指定大小的堆来说，需要两倍大小的内存空间；同时由于在内存调整的过程中要中断当前执行的程序，从而降低了程序的执行效率。

(5) 按代回收算法 (Generational Collector)

复制回收算法主要的缺点如下：每次算法执行时，所有处于活动状态的对象都要被复制，这样效率很低。由于程序有“程序创建的大部分对象的生命周期都很短，只有一部分对象有较长的生命周期”的特点，因此可以根据这个特点对算法进行优化。按代回收算法的主要思路如下：把堆分成两个或者多个子堆，每一个子堆被视为一代。算法在运行的过程中优先收集那些“年幼”的对象，如果一个对象经过多次收集仍然“存活”，那么就可以把这个对象转移到高一级的堆里，减少对其的扫描次数。

常见笔试题：

1. 现有如下代码。

```
1. public Object m() {
2.     Object o = new Float(3.14F);
3.     Object [] oa = new Object[1];
4.     oa[0] = o;
5.     o = null;
6.     oa[0] = null;
7.     print 'return 0';
8. }
```

当 Float 对象在第 2 行被创建后，什么时候能够被垃圾回收？()

A. 4 行以后 B. 5 行以后 C. 6 行以后 D. 7 行以后

答案：C。在第 6 行后不再有对象引用 Float 对象了，因此能够被垃圾回收。

2. 下列关于垃圾回收的说法中，正确的是 ()。

A. 一旦一个对象成为垃圾，就立刻被回收掉

B. 对象空间被回收掉之后，会执行该对象的 finalize 方法

C. finalize 方法和 C++ 的析构函数完全是一回事

D. 一个对象成为垃圾是因为不再有引用指着它，但是线程并非如此

答案：D。成为垃圾的对象，只有在下次垃圾回收器运行时才会被回收，而不是马上被清理，因此选项 A 错误。finalize 方法是在对象空间被回收前调用的，因此选项 B 错误。在 C++ 语言中，调用了析构函数后，对象一定会被销毁，而 Java 语言调用了 finalize 方法，垃圾却不一定被回收，因此 finalize 方法与 C++ 的析构函数是不同的，所以选项 C 也不正确。对于 D，当一个对象不再被引用后就成为垃圾可以被回收，但是线程就算没有被引用也可以独立运行的，因此与对象不同。所以正确答案为 D。

3. 是否可以主动通知 JVM 进行垃圾回收？

答案：由于垃圾回收器的存在，Java 语言本身没有给开发人员提供显式释放已分配内存的方法，也就是说，开发人员不能实时地调用垃圾回收器对某个对象或所有对象进行垃圾回收。

但开发人员却可以通过调用 `System.gc()` 方法来“通知”垃圾回收器运行，当然，JVM 也并不会保证垃圾回收器马上就会运行。由于 `System.gc()` 方法的执行会停止所有响应，去检查内存中是否有可回收的对象，这会对程序的正常运行以及性能造成极大的威胁，因此实际编程时，不推荐频繁使用这一方法。

4.8.5 Java 是否存在内存泄露问题

内存泄露是指一个不再被程序使用的对象或变量还在内存中占有存储空间。在 C/C++ 语言中，内存的分配与释放是由开发人员来负责的，如果开发人员忘记释放已分配的内存就会造成内存泄露。而在 Java 语言中引进了垃圾回收机制，由垃圾回收器负责回收不再使用的对象，既然有垃圾回收器来负责回收垃圾，那么是否还会存在内存泄露的问题呢？

其实，在 Java 语言中，判断一个内存空间是否符合垃圾回收的标准有两个：第一，给对象赋予了空值 `null`，以后再没有被使用过；第二，给对象赋予了新值，重新分配了内存空间。一般来讲，内存泄露主要有两种情况：一是在堆中申请的空间没有被释放；二是对象已不再被使用，但还仍然在内存中保留着。垃圾回收机制的引入可以有效地解决第一种情况；而对于第二种情况，垃圾回收机制则无法保证不再使用的对象会被释放。因此，Java 语言中的内存泄露主要指的是第二种情况。

下面通过一个示例来介绍 Java 语言中的内存泄露：

```
Vector v = new Vector(10);
for (int i = 1; i < 10; i++) {
    Object o = new Object();
    v.add(o);
}
```

在上述例子的循环中，不断创建新的对象加到 `Vector` 对象中，当退出循环后，`o` 的作用域将会结束，但是由于 `v` 在使用这些对象，因此垃圾回收器无法将其回收，此时就造成了内存泄露。只有将这些对象从 `Vector` 中删除才能释放创建的这些对象。

在 Java 语言中，容易引起内存泄露的原因很多，主要有以下几个方面的内容：

1) 静态集合类，例如 `HashMap` 和 `Vector`。如果这些容器为静态的，由于它们的生命周期与程序一致，那么容器中的对象在程序结束之前将不能被释放，从而造成内存泄露，如上例所示。

2) 各种连接，例如数据库连接、网络联接以及 IO 连接等。在对数据库进行操作的过程中，首先需要建立与数据库的连接，当不再使用时，需要调用 `close` 方法来释放与数据库的连接。只有连接被关闭后，垃圾回收器才会回收对应的对象。否则，如果在访问数据库的过程中，对 `Connection`、`Statement` 或 `ResultSet` 不显式地关闭，将会造成大量的对象无法被回收，从而引起内存泄露。

3) 监听器。在 Java 语言中，往往会使用到监听器。通常一个应用中会用到多个监听器，但在释放对象的同时往往没有相应地删除监听器，这也可能导致内存泄露。

4) 变量不合理的作用域。一般而言，如果一个变量定义的作用范围大于其使用范围，很有可能会造成内存泄露，另一方面如果没有及时地把对象设置为 `null`，很有可能会导致内存泄露的发生，示例如下：

```
class Server {
    private String msg;
```

```
public void recieveMsg() {
    readFromNet(); //从网络接收数据保存到 msg 中
    saveDB();      //把 msg 保存到数据库中
}
```

在上述伪代码中，通过 `readFromNet()` 方法接收的消息保存在变量 `msg` 中，然后调用 `saveDB()` 方法把 `msg` 的内容保存到数据库中，此时 `msg` 已经没用了，但是由于 `msg` 的生命周期与对象的生命周期相同，此时 `msg` 还不能被回收，因此造成了内存泄露。对于这个问题，有如下两种解决方法：第一种方法，由于 `msg` 的作用范围只在 `recieveMsg()` 方法内，因此可以把 `msg` 定义为这个方法的局部变量，当方法结束后，`msg` 的生命周期就会结束，此时垃圾回收器就可以回收 `msg` 的内容了；第二种方法，在使用完 `msg` 后就把 `msg` 设置为 `null`，这样垃圾回收器也会自动回收 `msg` 内容所占的内存空间。

5) 单例模式可能会造成内存泄露。单例模式的实现方法有很多种，下例中所使用的单例模式就可能会造成内存泄露：

```
class BigClass {
    //class body
}
class Singleton {
    private BigClass bc;
    private static Singleton instance = new Singleton(new BigClass());
    private Singleton(BigClass bc) { this.bc = bc; }
    public Singleton getInstance() {
        return instance;
    }
}
```

在上述实现的单例模式中，`Singleton` 存在一个对对象 `BigClass` 的引用，由于单例对象以静态变量的方式存储，因此它在 JVM 的整个生命周期中都存在，同时由于它有一个对对象 `BigClass` 的引用，这样会导致 `BigClass` 类的对象不能够被回收。

4.8.6 Java 中的堆和栈有什么区别

在 Java 语言中，堆与栈都是内存中存放数据的地方。变量分为基本数据类型和引用类型，基本数据类型的变量（例如 `int`、`short`、`long`、`byte`、`float`、`double`、`boolean` 以及 `char` 等）以及对象的引用变量，其内存都分配在栈上，变量出了作用域就会自动释放，而引用类型的变量，其内存分配在堆上或者常量池（例如字符串常量和基本数据类型常量）中，需要通过 `new` 等方式进行创建。

具体而言，栈内存主要用来存放基本数据类型与引用变量。栈内存的管理是通过压栈和弹栈操作来完成的，以栈帧为基本单位来管理程序的调用关系，每当有函数调用时，都会通过压栈方式创建新的栈帧，每当函数调用结束后都会通过弹栈的方式释放栈帧。

堆内存用来存放运行时创建的对象。一般来讲，通过 `new` 关键字创建出来的对象都存放在堆内存中。由于 JVM 是基于堆栈的虚拟机，而每个 Java 程序都运行在一个单独的 JVM 实例上，每一个实例唯一对应一个堆，一个 Java 程序内的多个线程也就运行在同一个 JVM 实例上，因此这些线程之间会共享堆内存，鉴于此，多线程在访问堆中的数据时需要加锁。

据进行同步。

在 C++ 中，堆内存的管理都是由开发人员来负责的，也就是说，开发人员在堆中申请的内存，当不再使用时，必须由开发人员来完成堆内存释放的工作。而在 Java 语言中，这个内存释放的工作由垃圾回收器来负责执行，开发人员只需要申请所需的堆空间而不需要考虑释放的问题。

在堆中产生了一个数组或对象后，还可以在栈中定义一个特殊的变量，让栈中这个变量的取值等于数组或对象在堆内存中的首地址，栈中的这个变量就成了数组或对象的引用变量。引用变量就相当于为数组或对象起的一个名称，以后就可以在程序中使用栈中的引用变量来访问堆中的数组或对象。这就是 Java 中引用的用法。

从堆和栈的功能以及作用来比较，堆主要用来存放对象的，栈主要是用来执行程序。相较于堆，栈的存取速度更快，但栈的大小和生存期必须是确定的，因此缺乏一定的灵活性。而堆却可以在运行时动态地分配内存，生存期不用提前告诉编译器，但这也导致了其存取速度的缓慢。

堆和栈的存储如下例所示：

```
class Rectangle{
    private int width;
    private int length;
    public Rectangle(int width,int length){
        this.width = width;
        this.length = length;
    }
}

public class Test {
    public static void main(String[] a) {
        int i = 1;
        Rectangle r = new Rectangle(3,5);
    }
}
```

在上述程序进入 main() 方法后，数据的存储关系如图 4-11 所示。

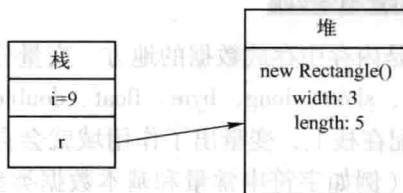


图 4-11 栈与堆的区别

由于 i 为基本数据类型的局部变量，因此它存储在栈空间中，而 r 为对象的引用变量，因此也被存储在栈空间中；实际的对象存储在堆空间中，当 main() 方法退出后，存储在栈中的 i 和 r 通过压栈和弹栈操作将会在栈中被回收，而存储在堆中的对象将会由垃圾回收器来自动回收。

4.9 容器

4.9.1 Java Collections 框架是什么

Java Collections 框架中包含了大量集合接口以及这些接口的实现类和操作它们的算法（例如排序、查找、反转、替换、复制、取最小元素、取最大元素等），具体而言，主要提供了 List（列表）、Queue（队列）、Set（集合）、Stack（栈）和 Map（映射表，用于存放键值对）等数据结构。其中，List、Queue、Set、Stack 都继承自 Collection 接口。

Collection 是整个集合框架的基础，它里面储存一组对象，表示不同类型的 Collections，它的作用只是提供维护一组对象的基本接口而已。

下面分别介绍 Set、List 和 Map 3 个接口。

1) Set 表示数学意义上的集合概念。其最主要的特点是集合中的元素不能重复，因此存入 Set 的每个元素都必须定义 equals() 方法来确保对象的唯一性。该接口有两个实现类：HashSet 和 TreeSet。其中 TreeSet 实现了 SortedSet 接口，因此 TreeSet 容器中的元素是有序的。

2) List 又称为有序的 Collection。它按对象进入的顺序保存对象，所以它能对列表中的每个元素的插入和删除位置进行精确的控制。同时，它可以保存重复的对象。LinkedList、ArrayList 和 Vector 都实现了 List 接口。

3) Map 提供了一个从键映射到值的数据结构。它用于保存键值对，其中值可以重复，但键是唯一的，不能重复。Java 类库中有多个实现该接口的类：HashMap、TreeMap、LinkedHashMap、WeakHashMap 和 IdentityHashMap。虽然它们都实现了相同的接口，但执行效率却不是完全相同的。具体而言，HashMap 是基于散列表实现的，采用对象的 hashCode 可以进行快速查询。LinkedHashMap 采用列表来维护内部的顺序。TreeMap 基于红黑树的数据结构来实现的，内部元素是按需排列的。

Collection 的框架类图如图 4-12 所示。

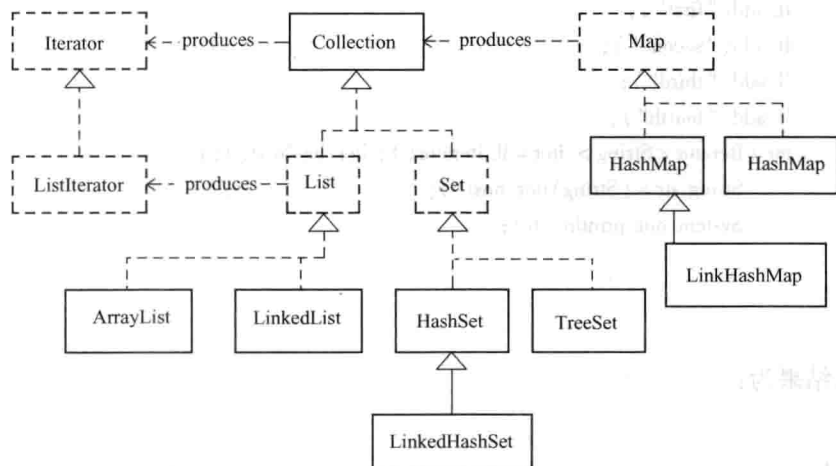


图 4-12 Collection 框架类图

常见笔试题：

下面哪种创建 Map 集合的方式是正确的？（ ）

- A. `Map m = new Map()`
- B. `Map m = new Map(init capacity, increment capacity)`
- C. `Map m = new Map(new Collection())`
- D. `Map` 是接口，所以不能实例化

答案：D。由于 `Map` 是一个接口，因此不能直接实例化 `Map` 的对象，但是可以实例化实现 `Map` 接口的类的对象，例如 `Map m = new HashMap()`。

4.9.2 什么是迭代器

迭代器 (Iterator) 是一个对象，它的工作是遍历并选择序列中的对象，它提供了一种访问一个容器 (container) 对象中的各个元素，而又不必暴露该对象内部细节的方法。通过迭代器，开发人员不需要了解容器底层的结构，就可以实现对容器的遍历。由于创建迭代器的代价小，因此迭代器通常被称为轻量级的容器。

迭代器的使用主要有以下 3 个方面的注意事项：

- 1) 使用容器的 `iterator()` 方法返回一个 `Iterator`，然后通过 `Iterator` 的 `next()` 方法返回第一个元素。
- 2) 使用 `Iterator` 的 `hasNext()` 方法判断容器中是否还有元素，如果有，可以使用 `next()` 方法获取下一个元素。
- 3) 可以通过 `remove()` 方法删除迭代器返回的元素。

`Iterator` 支持派生的兄弟成员。`ListIterator` 只存在于 `List` 中，支持在迭代期间向 `List` 中添加或删除元素，并且可以在 `List` 中双向滚动。

`Iterator` 的使用方法如下例所示：

```
import java.util.*;

public class IteratorTest {

    public static void main(String[] args) {

        List<String> ll = new LinkedList<String>();
        ll.add("first");
        ll.add("second");
        ll.add("third");
        ll.add("fourth");

        for (Iterator<String> iter = ll.iterator(); iter.hasNext();) {
            String str = (String)iter.next();
            System.out.println(str);
        }
    }
}
```

程序运行结果为：

```
first
second
third
fourth
```

在使用 `iterator()` 方法时经常会遇到 `ConcurrentModificationException` 异常，这通常是由于在使用 `Iterator` 遍历容器的同时又对容器做增加或删除操作所导致的，或者由于多线程操作导致，

当一个线程使用迭代器遍历容器的同时，另外一个线程对这个容器进行增加或删除操作。下例主要介绍单线程抛出 `ConcurrentModificationException` 的情况：

```
import java.util.*;

public class IteratorTest {
    public static void main(String[] args) {
        List<String> ll = new LinkedList<String>();
        ll.add("first");
        ll.add("second");
        ll.add("third");
        ll.add("fourth");
        for (Iterator<String> iter = ll.iterator(); iter.hasNext(); ) {
            String str = (String) iter.next();
            System.out.println(str);
            if (str.equals("second"))
                ll.add("five");
        }
    }
}
```

程序运行结果为：

```
first
second
Exception in thread "main" java.util.ConcurrentModificationException
    at java.util.LinkedList $ListItr.checkForComodification(Unknown Source)
    at java.util.LinkedList $ListItr.next(Unknown Source)
    at IteratorTest.main(IteratorTest.java:11)
```

抛出上述异常的主要原因是当调用容器的 `iterator()` 方法返回 `Iterator` 对象时，把容器中包含对象的个数赋值给了一个变量 `expectedModCount`，在调用 `next()` 方法时会比较变量 `expectedModCount` 与容器中实际对象的个数 `modCount` 的值是否相等，若二者不相等，则会抛出 `ConcurrentModificationException` 异常，因此在使用 `Iterator` 遍历容器的过程中，如果对容器进行增加或删除操作，就会改变容器中对象的数量，从而导致抛出异常。解决方法如下：在遍历的过程中把需要删除的对象保存到一个集合中，等遍历结束后在调用 `removeAll()` 方法来删除，或者使用 `iter.remove()` 方法。

以上主要介绍了单线程的解决方案，那么多线程访问容器的过程中抛出 `ConcurrentModificationException` 异常又该怎么解决呢？

- 1) 在 JDK 1.5 版本引入了线程安全的容器，比如 `ConcurrentHashMap` 和 `CopyOnWriteArrayList` 等。可以使用这些线程安全的容器来代替非线程安全的容器。
- 2) 在使用迭代器遍历容器时对容器的操作放到 `synchronized` 代码块中，但是当引用程序并发程度比较高时，这会严重影响程序的性能。

引申：`Iterator` 与 `ListIterator` 有什么区别？

`Iterator` 只能正向遍历集合，适用于获取移除元素。`ListIterator` 继承自 `Iterator`，专门针对 `List`，可以从两个方向来遍历 `List`，同时支持元素的修改。

4.9.3 ArrayList、Vector 和 LinkedList 有什么区别

`ArrayList`、`Vector`、`LinkedList` 类均在 `java.util` 包中，均为可伸缩数组，即可以动态改变长度的数组。

ArrayList 和 Vector 都是基于存储元素的 Object[] array 来实现的，它们会在内存中开辟一块连续的空间来存储，由于数据存储是连续的，因此，它们支持用序号（下标）来访问元素，同时索引数据的速度比较快。但是在插入元素时需要移动容器中的元素，所以对数据的插入操作执行得比较慢。ArrayList 和 Vector 都有一个初始化的容量的大小，当里面存储的元素超过这个大小时就需要动态地扩充它们的存储空间。为了提高程序的效率，每次扩充容量，不是简单地扩充一个存储单元，而是一次增加多个存储单元。Vector 默认扩充为原来的 2 倍（每次扩充空间的大小是可以设置的），而 ArrayList 默认扩充为原来的 1.5 倍（没有提供方法来设置空间扩充的方法）。

ArrayList 与 Vector 最大的区别就是 synchronization（同步）的使用，没有一个 ArrayList 的方法是同步的，而 Vector 的绝大多数方法（例如 add、insert、remove、set、equals、hashCode 等）都是直接或者间接同步的，所以 Vector 是线程安全的，ArrayList 不是线程安全的。正是由于 Vector 提供了线程安全的机制，其性能上也要略逊于 ArrayList。

LinkedList 是采用双向列表来实现的，对数据的索引需要从列表头开始遍历，因此用于随机访问则效率比较低，但是插入元素时不需要对数据进行移动，因此插入效率较高。同时，LinkedList 是非线程安全的容器。

那么，在实际使用时，如何从这几种容器中选择合适的使用呢？当对数据的主要操作为索引或只在集合的末端增加、删除元素时，使用 ArrayList 或 Vector 效率比较高；当对数据的操作主要为指定位置的插入或删除操作时，使用 LinkedList 效率比较高；当在多线程中使用容器时（即多个线程会同时访问该容器），选用 Vector 较为安全。

常见笔试题：

1. 若线性表最常用的操作是存取第 i 个元素及其前趋的值，则采用（ ）存储方式节省时间。

- A. 单链表 B. 双链表 C. 单循环链表 D. 顺序表

答案：D。顺序适合在随机访问的场合使用，访问时间复杂度为 $O(1)$ ，而列表的随机访问操作的时间复杂度为 $O(n)$ 。

2. 对于 import java.util 包，下列说法中，错误的是（ ）。

- A. Vector 类属于 java.util 包 B. Vector 类放在.../java/util/目录下
C. Vector 类放在 java.util 文件中 D. Vector 类是 Sun 公司的产品

答案：C。见上面讲解。

4.9.4 HashMap、HashTable、TreeMap 和 WeakHashMap 有哪些区别

Java 为数据结构中的映射定义了一个接口 java.util.Map，它包括 3 个实现类：HashMap、HashTable 和 TreeMap。Map 是用来存储键值对的数据结构，在数组中通过数组下标来对其内容索引的，而在 Map 中，则是通过对象来进行索引，用来索引的对象叫做 key，其对应的对象叫做 value。

HashMap 是一个最常用的 Map，它根据键的 hashCode 值存储数据，根据键可以直接获取它的值，具有很快的访问速度。由于 HashMap 与 HashTable 都采用了 hash 法进行索引，因此二者具有许多相似之处，它们主要有如下的一些区别：

1) HashMap 是 HashTable 的轻量级实现（非线程安全的实现），它们都完成了 Map 接口，主要区别在于 HashMap 允许空（null）键值（key）（但需要注意，最多只允许一条记录的键为 null，不允许多条记录的值为 null），而 HashTable 不允许。

2) HashMap 把 Hashtable 的 contains 方法去掉了, 改成 containsValue 和 containsKey, 因为 contains 方法容易让人引起误解。Hashtable 继承自 Dictionary 类, 而 HashMap 是 Java 1.2 引进的 Map interface 的一个实现。

3) Hashtable 的方法是线程安全的, 而 HashMap 不支持线程的同步, 所以它不是线程安全的。在多个线程访问 Hashtable 时, 不需要开发人员对它进行同步, 而对于 HashMap, 开发人员必须提供额外的同步机制。所以, 就效率而言, HashMap 可能高于 Hashtable。

4) Hashtable 使用 Enumeration, HashMap 使用 Iterator。

5) Hashtable 和 HashMap 采用的 hash/rehash 算法都几乎一样, 所以性能不会有很大的差异。

6) 在 Hashtable 中, hash 数组默认大小是 11, 增加的方式是 $old \times 2 + 1$ 。在 HashMap 中, hash 数组的默认大小是 16, 而且一定是 2 的指数。

7) hash 值的使用不同, Hashtable 直接使用对象的 hashCode。

以上 3 种类型中, 使用最多的是 HashMap。HashMap 里面存入的键值对在取出时没有固定的顺序, 是随机的。一般而言, 在 Map 中插入、删除和定位元素, HashMap 是最好的选择。由于 TreeMap 实现了 SortMap 接口, 能够把它保存的记录根据键排序, 因此, 取出来的是排序后的键值对, 如果需要按自然顺序或自定义顺序遍历键, 那么 TreeMap 会更好。LinkedHashMap 是 HashMap 的一个子类, 如果需要输出的顺序和输入的相同, 那么用 LinkedHashMap 可以实现, 它还可以按读取顺序来排列。

WeakHashMap 与 HashMap 类似, 二者的不同之处在于 WeakHashMap 中 key 采用的是“弱引用”的方式, 只要 WeakHashMap 中的 key 不再被外部引用, 它就可以被垃圾回收器回收。而 HashMap 中 key 采用的是“强引用的方式”, 当 HashMap 中的 key 没有被外部引用时, 只有在这个 key 从 HashMap 中删除后, 才可以被垃圾回收器回收。

常见笔试题:

1. 在 Hashtable 上下文中, 同步指的是什么?

答案: 同步意味着在一个时间点只能有一个线程可以修改 hash 表, 任何线程在执行 Hashtable 的更新操作前都需要获取对象锁, 其他线程则等待锁的释放。

2. 如何实现 HashMap 的同步?

答案: HashMap 可以通过 `Map m = Collections.synchronizedMap(new HashMap())` 来达到同步的效果。具体而言, 该方法返回一个同步的 Map, 该 Map 封装了底层的 HashMap 的所有方法, 使得底层的 HashMap 即使是在多线程的环境中也是安全的。

4.9.5 用自定义作为 HashMap 或 Hashtable 的 key 需要注意哪些问题

HashMap 与 Hashtable 是用来存放键值对的一种容器, 在使用这两个容器时有一个限制: 不能用来存储重复的键。也就是说, 每个键只能唯一映射一个值, 当有重复的键时, 不会创建新的映射关系, 而会使用先前的 value 值。为了更好地说明这个问题, 我们首先来看一段示例代码:

```
import java.util.*;
class Person {
    String id;
    String name;
    public Person(String id, String name) {
```

```

        this.id = id;
        this.name = name;
    }

    public String toString() {
        return "id = " + id + ", name = " + name;
    }
}

public class Test {
    public static void test1() {
        System.out.println("Use user defined class as key:");
        HashMap<String, String> hm = new HashMap<String, String>();
        hm.put("aaa", "bbb");
        hm.put("aaa", "ccc");

        Iterator iter = hm.entrySet().iterator();
        while (iter.hasNext()) {
            Map.Entry entry = (Map.Entry) iter.next();
            String key = (String) entry.getKey();
            String val = (String) entry.getValue();
            System.out.println(key + " " + val);
        }
    }

    public static void main(String args[]) {
        test1();
    }
}

```

程序运行结果为：

```

Use user defined class as key:
aaa    ccc

```

从上面的例子可以看出，首先向 HashMap 中添加 <"aaa", "bbb">，接着添加 <"aaa", "ccc"> 的时候由于与前面已经添加的数据有相同的 key:"aaa"，因此会用新的值"ccc"替换"bbb"。

但当用自定义的类的对象作为 HashMap 的 key 时，有时候会给人造成一种假象——key 是可以重复的，示例如下：

```

import java.util.*;

class Person {
    String id;
    String name;
    public Person(String id, String name) {
        this.id = id;
        this.name = name;
    }

    public String toString() {
        return "id = " + id + ", name = " + name;
    }
}

```

```

public class Test {
    public static void test2() {
        System.out.println("Use String as key:");
        HashMap <Person,String > hm = new HashMap <Person,String > ();
        Person p1 = new Person("111", "name1");
        Person p2 = new Person("111", "name1");
        hm.put(p1, "address1");
        hm.put(p2, "address1");

        Iterator iter = hm.entrySet().iterator();
        while (iter.hasNext()) {
            Map.Entry entry = (Map.Entry) iter.next();
            Person key = (Person) entry.getKey();
            String val = (String)entry.getValue();
            System.out.println("key = " + key + "    value = " + val);
        }
    }

    public static void main(String args[]) {
        test2();
    }
}

```

程序运行结果为：

```

Use String as key:
key = id = 111 ,name = name1    value = address1
key = id = 111 ,name = name1    value = address1

```

从表面上看，向 HashMap 中添加的两个键值对的 key 值是相同的，可是为什么在后面添加的键值对没有覆盖前面的 value 呢？为了说明这个问题，下面首先介绍 HashMap 添加元素的操作过程。具体而言，在向 HashMap 中添加键值对 <key, value > 时，需要经过如下几个步骤：首先，调用 key 的 hashCode() 方法生成一个 hash 值 h1，如果这个 h1 在 HashMap 中不存在，那么直接将 <key, value > 添加到 HashMap 中；如果这个 h1 已经存在，那么找出 HashMap 中所有 hash 值为 h1 的 key，然后分别调用 key 的 equals() 方法判断当前添加的 key 值是否与已经存在的 key 值相同。如果 equals() 方法返回 true，说明当前需要添加的 key 已经存在，那么 HashMap 会使用新的 value 值来覆盖掉旧的 value 值；如果 equals() 方法返回 false，说明新增加的 key 在 HashMap 中不存在，因此会在 HashMap 中创建新的映射关系。当新增加的 key 的 hash 值已经在 HashMap 中存在时，就会产生冲突。一般而言，对于不同的 key 值可能会得到相同的 hash 值，因此就需要对冲突进行处理。一般而言，处理冲突的方法有开放地址法、再 hash 法、链地址法等。HashMap 使用的是链地址法来解决冲突，具体操作方法如图 4-13 所示。

向 HashMap 中添加元素时，若有冲突产生，其实现方式如图 4-14 所示。

从 HashMap 中通过 key 查找 value 时，首先调用的是 key 的 hashCode() 方法来获取到 key 对应的 hash 值 h，这样就可以确定键为 key 的所有值存储的首地址。如果 h 对应的 key 值有多个，那么程序接着会遍历所有 key，通过调用 key 的 equals() 方法来判断 key 的内容是否相等。只有当 equals() 方法的返回值为 true 时，对应的 value 才是正确的结果。

在上例中，由于使用自定义的类作为 HashMap 的 key，而没有重写 hashCode() 方法和 e-

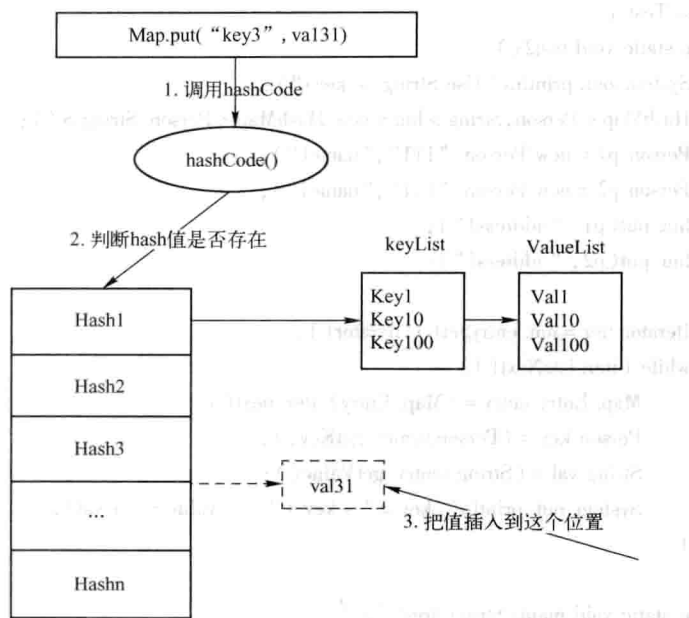


图 4-13 Map 工作原理 1

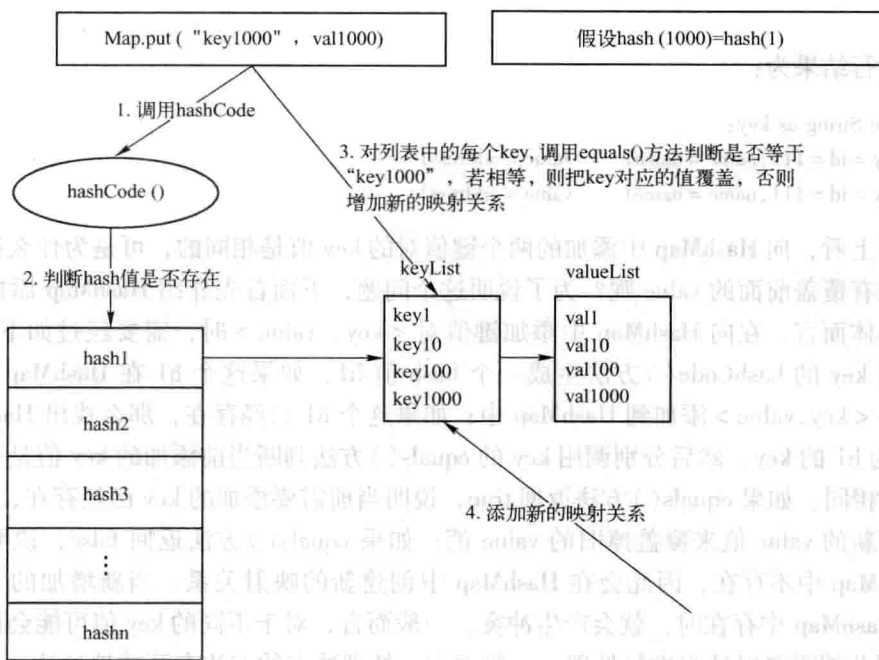


图 4-14 Map 工作原理 2

`equals()`方法，默认使用的是 `Object` 类的 `hashCode()`方法和 `equals()`方法。`Object` 类的 `equals()`方法的比较规则如下：当参数 `obj` 引用的对象与当前对象为同一个对象时，就返回 `true`，否则返回 `false`。`hashCode()`方法会返回对象存储的内存地址。由于在上例中创建了两个对象，虽然它们拥有相同的内容，但是存储在内存中不同的地址，因此在向 `HashMap` 中添加对象时，调用 `equals()`方法的返回值为 `false`，`HashMap` 会认为它们是两个不同的对象，就会分别创建不

同的映射关系，因此为了实现在向 `HashMap` 中添加键值对，可以根据对象的内容来判断两个对象是否相等，这就需要重写 `hashCode()` 方法和 `equals()` 方法，示例如下：

```
import java.util.*;

class Person {
    String id;
    String name;
    public int hashCode() {
        return id.hashCode();
    }
    public Person(String id, String name) {
        this.id = id;
        this.name = name;
    }
    public String toString() {
        return "id = " + id + ", name = " + name;
    }
    public boolean equals(Object obj) {
        Person p = (Person) obj;
        if (p.id.equals(this.id))
            return true;
        else
            return false;
    }
}

public class Test {
    public static void test2() {
        System.out.println("Use String as key:");
        HashMap<Person, String> hm = new HashMap<Person, String>();
        Person p1 = new Person("111", "name1");
        Person p2 = new Person("111", "name2");
        hm.put(p1, "address1");
        hm.put(p2, "address2");
        Iterator iter = hm.entrySet().iterator();
        while (iter.hasNext()) {
            Map.Entry entry = (Map.Entry) iter.next();
            Person key = (Person) entry.getKey();
            String val = (String) entry.getValue();
            System.out.println("key = " + key + "    value = " + val);
        }
    }

    public static void main(String args[]) {
        test2();
    }
}
```

程序运行结果为：

```
Use String as key:
```

```
key = id = 111, name = name1    value = address2
```

由此可以看出，开发者在使用自定义类作为 HashMap 的 key 时，需要注意以下几个问题：

- 1) 如果想根据对象的相关属性来自定义对象是否相等的逻辑，此时就需要重写 equals() 方法，一旦重写了 equals() 方法，那么就必须重写 hashCode() 方法。
- 2) 当自定义类的多项作为 HashMap (HashTable) 的 key 时，最好把这个类设计为不可变类。
- 3) 从 HashMap 的工作原理可以看出，如果两个对象相等，那么这两个对象有着相同的 hashCode，反之则不成立。

4.9.6 Collection 和 Collections 有什么区别

Collection 是一个集合接口。它提供了对集合对象进行基本操作的通用接口方法。实现该接口的类主要有 List 和 Set，该接口的设计目标是为各种具体的集合提供最大化的统一的操作方式。

Collections 是针对集合类的一个包装类，它提供一系列静态方法以实现对各种集合的搜索、排序、线程安全化等操作，其中大多数方法都是用来处理线性表。Collections 类不能实例化，如同一个工具类，服务于 Collection 框架。若在使用 Collections 类的方法时，对应的 collection 的对象为 null，则这些方法都会抛出 NullPointerException。

使用 Collections 的示例如下：

```
import java.util.*;  
public class Test {  
    public static void main(String args[]) {  
        List<Integer> list = new LinkedList<Integer>();  
        int array[] = { 1, 7, 3, 2 };  
        for (int i=0; i<array.length; i++) {  
            list.add(new Integer(array[i]));  
        }  
        Collections.sort(list);  
        for (int i=0; i<array.length; i++) {  
            System.out.println(list.get(i));  
        }  
    }  
}
```

程序运行结果为：

```
1  
2  
3  
7
```

4.10 多线程

4.10.1 什么是线程？它与进程有什么区别？为什么要使用多线程

线程是指程序在执行过程中，能够执行程序代码的一个执行单元。在 Java 语言中，线程

有4种状态：运行、就绪、挂起和结束。

进程是指一段正在执行的程序。而线程有时也被称为轻量级进程，它是程序执行的最小单元，一个进程可以拥有多个线程，各个线程之间共享程序的内存空间（代码段、数据段和堆空间）及一些进程级的资源（例如打开的文件），但是各个线程拥有自己的栈空间，进程与线程的对比关系如图4-15所示。

在操作系统级别上，程序的执行都是以进程为单位的，而每个进程中通常都会有多个线程互不影响地并发执行，那么为什么要使用多线程呢？其实，多线程的使用为程序研发带来了巨大的便利，具体而言，有以下几个方面的内容：

1) 使用多线程可以减少程序的响应时间。在单线程（单线程指的是程序执行过程中只有一个有效操作的序列，不同操作之间都有明确的执行先后顺序）的情况下，如果某个操作很耗时，或者陷入长时间的等待（如等待网络响应），此时程序将不会响应鼠标和键盘等操作，使用多线程后，可以把这个耗时的线程分配到一个单独的线程去执行，从而使程序具备了更好的交互性。

2) 与进程相比，线程的创建和切换开销更小。由于启动一个新的线程必须给这个线程分配独立的地址空间，建立许多数据结构来维护线程代码段、数据段等信息，而运行于同一进程内的线程共享代码段、数据段，线程的启动或切换的开销比进程要少很多。同时多线程在数据共享方面效率非常高。

3) 多CPU或多核计算机本身就具有执行多线程的能力，如果使用单个线程，将无法重复利用计算机资源，造成资源的巨大浪费。因此在多CPU计算机上使用多线程能提高CPU的利用率。

4) 使用多线程能简化程序的结构，使程序便于理解和维护。一个非常复杂的进程可以分成多个线程来执行。

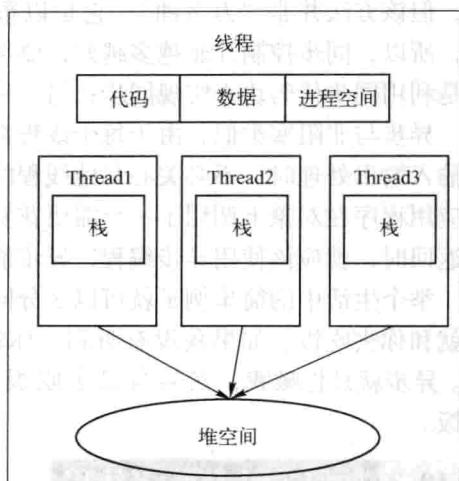


图4-15 进程与线程的对比关系

4.10.2 同步和异步有什么区别

在多线程的环境中，经常会碰到数据的共享问题，即当多个线程需要访问同一个资源时，它们需要以某种顺序来确保该资源在某一时刻只能被一个线程使用，否则，程序的运行结果将会是不可预料的，在这种情况下就必须对数据进行同步，例如多个线程同时对同一数据进行写操作，即当线程A需要使用某个资源时，如果这个资源正在被线程B使用，同步机制就会让线程A一直等待下去，直到线程B结束对该资源的使用后，线程A才能使用这个资源，由此可见，同步机制能够保证资源的安全。

要想实现同步操作，必须要获得每一个线程对象的锁。获得它可以保证在同一时刻只有一个线程能够进入临界区（访问互斥资源的代码块），并且在这个锁被释放之前，其他线程就不能再进入这个临界区。如果还有其他线程想要获得该对象的锁，只能进入等待队列等待。只有当拥有该对象锁的线程退出临界区时，锁才会被释放，等待队列中优先级最高的线程才能获得该锁，从而进入共享代码区。

Java 语言在同步机制中提供了语言级的支持，可以通过使用 `synchronized` 关键字来实现同步，但该方法并非“万金油”，它是以很大的系统开销作为代价的，有时候甚至可能造成死锁，所以，同步控制并非越多越好，要尽量避免无谓的同步控制。实现同步的方式有两种：一种是利用同步代码块来实现同步；另一种是利用同步方法来实现同步。

异步与非阻塞类似，由于每个线程都包含了运行时自身所需要的数据或方法，因此，在进行输入输出处理时，不必关心其他线程的状态或行为，也不必等到输入输出处理完毕才返回。当应用程序在对象上调用了—一个需要花费很长时间来执行的方法，并且不希望让程序等待方法的返回时，就应该使用异步编程，异步能够提高程序的效率。

举个生活中的简单例子就可以区分同步与异步了。同步就是你喊我去吃饭，如果听到了，我就和你去吃饭；如果我没有听到，你就不停地喊，直到我告诉你听到了，我们才一起去吃饭。异步就是你喊我，然后自己去吃饭，我得到消息后可能立即走，也可能等到下班才去吃饭。

4.10.3 如何实现 Java 多线程

Java 虚拟机允许应用程序并发地运行多个线程。在 Java 语言中，多线程的实现一般有以下 3 种方法，其中前两种为最常用的方法。

(1) 继承 `Thread` 类，重写 `run()` 方法

`Thread` 本质上也是实现了 `Runnable` 接口的一个实例，它代表一个线程的实例，并且，启动线程的唯一方法就是通过 `Thread` 类的 `start()` 方法。`start()` 方法是一个 `native`（本地）方法，它将启动一个新线程，并执行 `run()` 方法（`Thread` 中提供的 `run()` 方法是一个空方法）。这种方式通过自定义直接 `extend Thread`，并重写 `run()` 方法，就可以启动新线程并执行自己定义的 `run()` 方法。需要注意的是，调用 `start()` 方法后并不是立即执行多线程代码，而是使得该线程变为可运行态（`Runnable`），什么时候运行多线程代码是由操作系统决定的。下例给出了 `Thread` 的使用方法：

```
public void run() {
    System.out.println("Thread body"); //线程的函数体
}

public class Test {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start(); //开启线程
    }
}
```

(2) 实现 `Runnable` 接口，并实现该接口的 `run()` 方法

以下是主要步骤：

- 1) 自定义类并实现 `Runnable` 接口，实现 `run()` 方法。
- 2) 创建 `Thread` 对象，用实现 `Runnable` 接口的对象作为参数实例化该 `Thread` 对象。
- 3) 调用 `Thread` 的 `start()` 方法。

```
class MyThread implements Runnable { //创建线程类
    public void run() {
        System.out.println("Thread body");
    }
}
```

```

    }
}
public class Test {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        Thread t = new Thread(thread);
        t.start(); //开启线程
    }
}

```

其实，不管是通过继承 Thread 类还是通过使用 Runnable 接口来实现多线程的方法，最终还是通过 Thread 的对象的 API 来控制线程的。

(3) 实现 Callable 接口，重写 call()方法

Callable 接口实际是属于 Executor 框架中的功能类，Callable 接口与 Runnable 接口的功能类似，但提供了比 Runnable 更强大的功能，主要表现为以下 3 点：

- 1) Callable 可以在任务结束后提供一个返回值，Runnable 无法提供这个功能。
- 2) Callable 中的 call()方法可以抛出异常，而 Runnable 的 run()方法不能抛出异常。
- 3) 运行 Callable 可以拿到一个 Future 对象，Future 对象表示异步计算的结果，它提供了检查计算是否完成的方法。由于线程属于异步计算模型，因此无法从别的线程中得到函数的返回值，在这种情况下，就可以使用 Future 来监视目标线程调用 call()方法的情况，当调用 Future 的 get()方法以获取结果时，当前线程就会阻塞，直到 call()方法结束返回结果。

```

import java.util.concurrent.*;
public class CallableAndFuture {
    //创建线程类
    public static class CallableTest implements Callable<String> {
        public String call() throws Exception {
            return "Hello World!";
        }
    }
    public static void main(String[] args) {
        ExecutorService threadPool = Executors.newSingleThreadExecutor();
        //启动线程
        Future<String> future = threadPool.submit(new CallableTest());
        try {
            System.out.println("waiting thread to finish");
            System.out.println(future.get()); //等待线程结束,并获取返回结果
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

上述程序的输出结果为：

```
waiting thread to finishHello World!
```

以上 3 种方式中，前两种方式线程执行完后都没有返回值，只有最后一种是带返回值的。当需要实现多线程时，一般推荐实现 Runnable 接口的方式，其原因是：首先，Thread 类定义

了多种方法可以被派生类使用或重写。但是只有 `run()` 方法是必须被重写的，在 `run()` 方法中实现这个线程的主要功能。这当然是实现 `Runnable` 接口所需的方法。其次，很多 Java 开发人员认为，一个类仅在他们需要被加强或修改时才会被继承。因此，如果没有必要重写 `Thread` 类中的其他方法，那么通过继承 `Thread` 的实现方式与实现 `Runnable` 接口的效果相同，在这种情况下最好通过实现 `Runnable` 接口的方式来创建线程。

引申：一个类是否可以同时继承 `Thread` 与实现 `Runnable` 接口？

答案：可以。为了说明这个问题，首先给出如下示例：

```
public class Test extends Thread implements Runnable {
    public static void main(String args[]) {
        Thread t = new Thread(new Test());
        t.start();
    }
}
```

从上例中可以看出，`Test` 类实现了 `Runnable` 接口，但是并没有实现接口的 `run()` 方法，可能有些读者会认为这会导致编译错误，但实际上它是能够编译通过并运行的，因为 `Test` 类从 `Thread` 类中继承了 `run()` 方法，这个继承的 `run()` 方法可以被当作对 `Runnable` 接口的实现，因此这段代码能够编译通过。当然也可以不使用继承的 `run()` 方法，而是需要通过在 `Test` 类中重写 `run()` 方法来实现 `Runnable` 接口中的 `run()` 方法，示例如下：

```
public class Test extends Thread implements Runnable {
    public void run() {
        System.out.println("this is run()");
    }
    public static void main(String args[]) {
        Thread t = new Thread(new Test());
        t.start();
    }
}
```

程序运行结果为：

```
this is run()
```

4.10.4 `run()` 方法与 `start()` 方法有什么区别

通常，系统通过调用线程类的 `start()` 方法来启动一个线程，此时该线程处于就绪状态，而非运行状态，也就意味着这个线程可以被 JVM 来调度执行。在调度过程中，JVM 通过调用线程类的 `run()` 方法来完成实际的操作，当 `run()` 方法结束后，此线程就会终止。

如果直接调用线程类的 `run()` 方法，这会被当作一个普通的函数调用，程序中仍然只有主线程这一个线程，也就是说，`start` 方法()能够异步地调用 `run()` 方法，但是直接调用 `run()` 方法却是同步的，因此也就无法达到多线程的目的。

由此可知，只有通过调用线程类的 `start()` 方法才能真正达到多线程的目的。下面通过一个例子来说明 `run()` 方法与 `start()` 方法的区别。

```
class ThreadDemo extends Thread {
    @Override
```

```

        public void run() {
            System.out.println("ThreadDemo:begin");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("ThreadDemo:end");
        }
    }

    public class Test {
        public static void test1() {
            System.out.println("test1:begin");
            Thread t1 = new ThreadDemo();
            t1.start();
            System.out.println("test1:end");
        }

        public static void test2() {
            System.out.println("test2:begin");
            Thread t1 = new ThreadDemo();
            t1.run();
            System.out.println("test2:end");
        }

        public static void main(String[] args) {
            test1();
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            System.out.println();
            test2();
        }
    }

```

程序运行结果为：

```

test1:begin
test1:end
ThreadDemo:begin
ThreadDemo:end

```

```

test2:begin
ThreadDemo:begin
ThreadDemo:end
test2:end

```

从 test1 的运行结果可以看出，线程 t1 是在 test1 方法结束后才执行的（System.out.println

("test1:end")语句不需要等待 t1.start() 运行结果就可以执行), 因此, 在 test1 中调用 start() 方法是异步的, 所以 main 线程与 t1 线程是异步执行的。从 test2 的运行结果可以看出, 调用 t1.run() 是同步的调用方法, 因为 System.out.println("test2:end") 只有等 t1.run() 调用结束后才能执行。

4.10.5 多线程同步的实现方法有哪些

当使用多线程访问同一个资源时, 非常容易出现线程安全的问题 (例如, 当多个线程同时对一个数据进行修改时, 会导致某些线程对数据的修改丢失)。因此, 需要采用同步机制来解决这种问题。Java 主要提供了 3 种实现同步机制的方法:

(1) synchronized 关键字

在 Java 语言中, 每个对象都有一个对象锁与之相关联, 该锁表明对象在任何时候只允许被一个线程所拥有, 当一个线程调用对象的一段 synchronized 代码时, 需要先获取这个锁, 然后去执行相应的代码, 执行结束后, 释放锁。

synchronized 关键字主要有两种用法 (synchronized 方法和 synchronized 块), 此外该关键字还可以作用于静态方法、类或某个实例, 但这都对程序的效率有很大的影响。

1) synchronized 方法。在方法的声明前加入 synchronized 关键字, 示例如下:

```
public synchronized void mutiThreadAccess();
```

只要把多个线程对类需要被同步的资源的操作放到 mutiThreadAccess() 方法中, 就能保证这个方法在同一时刻只能被一个线程访问, 从而保证了多线程访问的安全性。然而, 当一个方法的方法体规模非常大时, 把该方法声明为 synchronized 会大大影响程序的执行效率。为了提高程序的效率, Java 提供了 synchronized 块。

2) synchronized 块。synchronized 块既可以把任意的代码段声明为 synchronized, 也可以指定上锁的对象, 有非常高的灵活性。其用法如下:

```
synchronized(syncObject) {  
    //访问 syncObject 的代码  
}
```

(2) wait() 方法与 notify() 方法

当使用 synchronized 来修饰某个共享资源时, 如果线程 A1 在执行 synchronized 代码, 另外一个线程 A2 也要同时执行同一对象的同一 synchronized 代码时, 线程 A2 将要等到线程 A1 执行完成后, 才能继续执行。在这种情况下可以使用 wait() 方法和 notify() 方法。

在 synchronized 代码被执行期间, 线程可以调用对象的 wait() 方法, 释放对象锁, 进入等待状态, 并且可以调用 notify() 方法或 notifyAll() 方法通知正在等待的其他线程。notify() 方法仅唤醒一个线程 (等待队列中的第一个线程) 并允许它去获得锁, notifyAll() 方法唤醒所有等待这个对象的线程并允许它们去获得锁 (并不是让所有唤醒线程都获取到锁, 而是让它们去竞争)。

(3) Lock

JDK 5 新增加了 Lock 接口以及它的一个实现类 ReentrantLock (重入锁), Lock 也可以用来实现多线程的同步, 具体而言, 它提供了如下一些方法来实现多线程的同步:

1) lock()。以阻塞的方式获取锁, 也就是说, 如果获取到了锁, 立即返回; 如果别的线程持有锁, 当前线程等待, 直到获取锁后返回。

2) `tryLock()`。以非阻塞的方式获取锁。只是尝试性地去获取一下锁，如果获取到锁，立即返回 `true`，否则，立即返回 `false`。

3) `tryLock(long timeout, TimeUnit unit)`。如果获取了锁，立即返回 `true`，否则会等待参数给定的时间单元，在等待的过程中，如果获取了锁，就返回 `true`，如果等待超时，返回 `false`。

4) `lockInterruptibly()`。如果获取了锁，立即返回；如果没有获取锁，当前线程处于休眠状态，直到获得锁，或者当前线程被别的线程中断（会收到 `InterruptedException` 异常）。它与 `lock()` 方法最大的区别在于如果 `lock()` 方法获取不到锁，会一直处于阻塞状态，且会忽略 `interrupt()` 方法，示例如下：

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class Test {
    public static void main(String[] args) throws InterruptedException {
        final Lock lock = new ReentrantLock();
        lock.lock();
        Thread t1 = new Thread(new Runnable() {
            public void run() {
                try {
                    lock.lockInterruptibly();
                    // lock.lock(); 编译器报错
                } catch (InterruptedException e) {
                    System.out.println("interrupted.");
                }
            }
        });
        t1.start();
        t1.interrupt();
        Thread.sleep(1);
    }
}
```

程序运行结果如下：

```
interrupted.
```

如果把 `lock.lockInterruptibly()` 替换为 `lock.lock()`，编译器将会提示 `lock.lock() catch` 代码块无效，这是因为 `lock.lock()` 不会抛出异常，由此可见 `lock()` 方法会忽略 `interrupt()` 引发的异常。

4.10.6 sleep() 方法与 wait() 方法有什么区别

`sleep()` 是使线程暂停执行一段时间的方法。`wait()` 也是一种使线程暂停执行的方法，例如，当线程交互时，如果线程对一个同步对象 `x` 发出一个 `wait()` 调用请求，那么该线程会暂停执行，被调对象进入等待状态，直到被唤醒或等待时间超时。

具体而言，`sleep()` 方法与 `wait()` 方法的区别主要表现在以下几个方面：

1) 原理不同。`sleep()` 方法是 `Thread` 类的静态方法，是线程用来控制自身流程的，它会使此线程暂停执行一段时间，而把执行机会让给其他线程，等到计时时间一到，此线程会自动“苏醒”，例如，当线程执行报时功能时，每一秒钟打印出一个时间，那么此时就需要在打印

锁（死锁指的是两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，如果无外力作用，它们都将无法推进）。由于调用 `suspend()` 方法不会释放锁，这就导致一个问题：如果用一个 `suspend` 挂起一个有锁的线程，那么在锁恢复之前将不会被释放。如果调用 `suspend()` 方法，线程将试图取得相同的锁，程序就会发生死锁，例如，线程 A 已经获取到了互斥资源 M 的锁，此时线程 A 通过 `suspend()` 方法挂起线程 A 的执行，接着线程 B 也去访问互斥资源 M，这时候就造成了死锁。鉴于以上两种方法的不安全性，Java 语言已经不建议使用以上两种方法来终止线程了。

那么，如何才能终止线程呢？一般建议采用的方法是让线程自行结束进入 Dead 状态。一个线程进入 Dead 状态，即执行完 `run()` 方法，也就是说，如果想要停止一个线程的执行，就要提供某种方式让线程能够自动结束 `run()` 方法的执行。在实现时，可以通过设置一个 flag 标志来控制循环是否执行，通过这种方法来让线程离开 `run()` 方法从而终止线程。下例给出了结束线程的方法：

```
public class MyThread implements Runnable {
    private volatile Boolean flag;
    public void stop() {
        flag = false;
    }
    public void run() {
        while(flag)
            ;//do something
    }
}
```

上例中，通过调用 `MyThread` 的 `stop()` 方法虽然能够终止线程，但同样也存在问题：当线程处于非运行状态时（当 `sleep()` 方法被调用或当 `wait()` 方法被调用或当被 I/O 阻塞时），上面介绍的方法就不可用了。此时可以使用 `interrupt()` 方法来打破阻塞的情况，当 `interrupt()` 方法被调用时，会抛出 `InterruptedException` 异常，可以通过在 `run()` 方法中捕获这个异常来让线程安全退出，具体实现方式如下：

```
public class MyThread {
    public static void main(String[] args) {
        Thread thread = new Thread(new Runnable() {
            public void run() {
                System.out.println("thread go to sleep");
                try {
                    //用休眠来模拟线程被阻塞
                    Thread.sleep(5000);
                    System.out.println("thread finish");
                } catch (InterruptedException e) {
                    System.out.println("thread is interrupted!");
                }
            }
        });
        thread.start();
        thread.interrupt();
    }
}
```


程序运行结果为：

```
thread go to sleep
thread is interrupted!
```

如果程序因为 I/O 而停滞，进入非运行状态，基本上要等到 I/O 完成才能离开这个状态，在这种情况下，无法使用 `interrupt()` 方法来使程序离开 `run()` 方法。这就需要使用一个替代的方法，基本思路也是触发一个异常，而这个异常与所使用的 I/O 相关，例如，如果使用 `readLine()` 方法在等待网络上的一个信息，此时线程处于阻塞状态。让程序离开 `run()` 方法就是使用 `close()` 方法来关闭流，在这种情况下会引发 `IOException` 异常，`run()` 方法可以通过捕获这个异常来安全地结束线程。

4.10.8 synchronized 与 Lock 有什么异同

Java 语言提供了两种锁机制来实现对某个共享资源的同步：`synchronized` 和 `Lock`。其中，`synchronized` 使用 `Object` 对象本身的 `notify`、`wait`、`notifyAll` 调度机制，而 `Lock` 可以使用 `Condition` 进行线程之间的调度，完成 `synchronized` 实现的所有功能。

具体而言，二者的主要区别主要表现在以下几个方面的内容：

1) 用法不一样。在需要同步的对象中加入 `synchronized` 控制，`synchronized` 既可以加在方法上，也可以加在特定代码块中，括号中表示需要锁的对象。而 `Lock` 需要显式地指定起始位置和终止位置。`synchronized` 是托管给 JVM 执行的，而 `Lock` 的锁定是通过代码实现的，它有比 `synchronized` 更精确的线程语义。

2) 性能不一样。在 JDK 5 中增加了一个 `Lock` 接口的实现类 `ReentrantLock`。它不仅拥有和 `synchronized` 相同的并发性和内存语义，还多了锁投票、定时锁、等候和中断锁等。它们的性能在不同的情况下会有所不同：在资源竞争不是很激烈的情况下，`synchronized` 的性能要优于 `ReentrantLock`，但是在资源竞争很激烈的情况下，`synchronized` 的性能会下降得非常快，而 `ReentrantLock` 的性能基本保持不变。

3) 锁机制不一样。`synchronized` 获得锁和释放的方式都是在块结构中，当获取多个锁时，必须以相反的顺序释放，并且是自动解锁，不会因为出了异常而导致锁没有被释放从而引发死锁。而 `Lock` 则需要开发人员手动去释放，并且必须在 `finally` 块中释放，否则会引起死锁问题的发生。此外，`Lock` 还提供了更强大的功能，它的 `tryLock()` 方法可以采用非阻塞的方式去获取锁。

虽然 `synchronized` 与 `Lock` 都可以用来实现多线程的同步，但是，最好不要同时使用这两种同步机制，因为 `ReentrantLock` 与 `synchronized` 所使用的机制不同，所以它们的运行是独立的，相当于两种类型的锁，在使用时互不影响，示例如下：

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
class SyncTest {
    private int value = 0;
    Lock lock = new ReentrantLock();
    public synchronized void addValueSync() {
```

```

        this.value++;
        System.out.println(Thread.currentThread().getName() + ":" + value);
    }
    public void addValueLock() {
        try {
            lock.lock();
            value++;
            System.out.println(Thread.currentThread().getName() + ":" + value);
        } finally {
            lock.unlock();
        }
    }
}

public class Test {
    public static void main(String[] args) {
        final SyncTest st = new SyncTest(); //测试 synchronized
        //final LockTest st = new LockTest(); //测试 Lock

        Thread t1 = new Thread(
            new Runnable() {
                public void run() {
                    for(int i=0;i<5;i++){
                        st.addValueSync();
                    }
                }
            }
        );
        Thread t2 = new Thread(
            new Runnable() {
                public void run() {
                    for(int i=0;i<5;i++){
                        st.addValueLock();
                    }
                }
            }
        );
        t1.start();
        t2.start();
    }
}

```

程序运行结果为

```
Thread-0:1
Thread-1:2
Thread-0:4
Thread-1:4
Thread-0:5
Thread-1:6
Thread-0:8
Thread-1:8
Thread-0:10
Thread-1:10
```

当然，上例中，并不是每次运行的结果都是相同的，与前一个例子对比可以发现，上例中的输出结果 value 的值并不是连续的，这就是因两种上锁方法采用了不同的机制而造成的，因此在实际使用时，最好不要同时使用两种上锁机制。

常考笔试题：

1. 当一个线程进入一个对象的一个 synchronized() 方法后，其他线程是否可进入此对象的其他方法？

答案：当一个线程进入一个对象的一个 synchronized() 方法后，其他线程是否可进入此对象的其他方法取决于方法本身，如果该方法是非 synchronized() 方法，那么是可以访问的，示例如下。

```
class Test {
    public synchronized void synchronizedMethod() {
        System.out.println("begin calling synchronizedMethod");
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
        System.out.println("finish calling synchronizedMethod");
    }
    public void generalMethod() {
        System.out.println("call generalMethod...");
    }
}

public class MultiThread {
    static final Test t = new Test();
    public static void main(String[] args) {
        Thread t1 = new Thread() {
            public void run() {
                t.synchronizedMethod();
            }
        };
        Thread t2 = new Thread() {
            public void run() {
                t.generalMethod();
            }
        };
    }
}
```

```

    }
    };
    t1.start();
    t2.start();
}
}

```

程序运行结果为：

```

begin calling synchronizedMethod
call generalMethod...
finish calling synchronizedMethod

```

从上例可以看出，线程 t1 在调用 synchronized() 方法的过程中，线程 t2 仍然可以访问同一对象的非 synchronized() 方法。

如果其他方法是静态方法（使用 static 修饰的方法），它用的同步锁是当前类的字节码，与非静态的方法不能同步（因为非静态的方法用的是 this），因此，静态方法可以被调用，示例如下。

```

class Test {
    public synchronized void synchronizedMethod() {
        System.out.println("begin calling synchronizedMethod");
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
        System.out.println("finish calling synchronizedMethod");
    }
    public synchronized static void generalMethod() {
        System.out.println("call generalMethod...");
    }
}

```

```

public class MutiThread {
    static final Test t = new Test();
    public static void main(String[] args) {
        Thread t1 = new Thread() {
            public void run() {
                t.synchronizedMethod();
            }
        };
        Thread t2 = new Thread() {
            public void run() {
                t.generalMethod();
            }
        };
        t1.start();
        t2.start();
    }
}

```

```
}
```

程序运行结果为：

```
begin calling synchronizedMethod
call generalMethod ...
finish calling synchronizedMethod
```

从上例可以看出，当线程 t1 在调用对象 t 的 synchronized() 方法时，线程 t2 仍然可以调用这个对象的静态 synchronized() 方法。

2. 答案：

如果这个方法内部调用了 wait() 方法，那么其他线程就可以访问同一对象的其他 synchronized() 方法。如果这个方法内部没有调用 wait() 方法，并且其他方法都为 synchronized() 方法，那么其他线程将无法访问这个对象的其他方法。

4.10.9 什么是守护线程

Java 提供了两种线程：守护线程与用户线程。守护线程又被称为“服务进程”“精灵线程”或“后台线程”，是指在程序运行时在后台提供一种通用服务的线程，这种线程并不属于程序中不可或缺的部分。通俗点讲，任何一个守护线程都是整个 JVM 中所有非守护线程的“保姆”。

用户线程和守护线程几乎一样，唯一的不同之处就在于如果用户线程已经全部退出运行，只剩下守护线程存在了，JVM 也就退出了。因为当所有非守护线程结束时，没有了被守护者，守护线程也就没有工作可做了，也就没有继续运行程序的必要了，程序也就终止了，同时会“杀死”所有守护线程。也就是说，只要有任何非守护线程还在运行，程序就不会终止。

在 Java 语言中，守护线程一般具有较低的优先级，它并非只由 JVM 内部提供，用户在编写程序时也可以自己设置守护线程，例如，将一个用户线程设置为守护线程的方法就是在调用 start() 方法启动线程之前调用对象的 setDaemon(true) 方法，若将以上参数设置为 false，则表示的是用户进程模式。需要注意的是，当在一个守护线程中产生了其他线程，那么这些新产生的线程默认还是守护线程，用户线程也是如此，示例如下：

```
class ThreadDemo extends Thread {
    public void run() {
        System.out.println(Thread.currentThread().getName() + ":begin");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + ":end");
    }
}

public class Test {
    public static void main(String[] args) {
        System.out.println("test3:begin");
        Thread t1 = new ThreadDemo();
        t1.setDaemon(true);
    }
}
```

```
t1.start();
System.out.println("test3:end");
```

程序运行结果为：

```
test3:begin
test3:end
Thread-0:begin
```

从运行结果中可以发现，没有输出 Thread-0: end。之所以结果是这样的，是在启动线程前将其设置为守护线程了，当程序中只有守护线程存在时，JVM 是可以退出的，也就是说，当 JVM 中只有守护线程运行时，JVM 会自动关闭。因此，当 test3 方法调用结束后，main 线程将退出，此时线程 t1 还处于休眠状态没有运行结束，但是由于此时只有这个守护线程在运行，JVM 将会关闭，因此不会输出“Thread-0: end”。

守护线程的一个典型的例子就是垃圾回收器。只要 JVM 启动，它始终在运行，实时监控和管理系统中可以被回收的资源。

常见笔试题：

1. Java 的 Daemon 线程，setDaemon 设置必须要 ()。

A. 在调用 start() 方法之前 B. 调用在 start() 方法之后 C. 前后都可以

答案：A。见上面讲解。

2. 关于守护线程的说法，正确的是 ()。

- A. 所有非守护线程终止，即使存在守护线程，进程运行终止
- B. 所有守护线程终止，即使存在非守护线程，进程运行终止
- C. 只要有守护线程或者非守护进程其中之一存在，进程就不会终止
- D. 只要所有守护线程和非守护线程终止运行之后，进程才会终止

答案：A。见上面讲解。

4.10.10 join() 方法的作用是什么

在 Java 语言中，join() 方法的作用是让调用该方法的线程在执行完 run() 方法后，再执行 join 方法后面的代码。简单点说，就是将两个线程合并，用于实现同步功能。具体而言，可以通过线程 A 的 join() 方法来等待线程 A 的结束，或者使用线程 A 的 join(2000) 方法来等待线程 A 的结束，但最多只等待 2s，示例如下：

```
public class JoinTest {
    public static void main(String[] args) {
        Thread t = new Thread( new ThreadImp());
        t.start();
        try {
            t.join(1000); //主线程等待 t 结束,只等 1 秒
            if(t.isAlive()) //t 已经结束
                System.out.println("t has not finished");
            else
                System.out.println("t has finished");
            System.out.println("joinFinish");
        }
    }
}
```



```

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class ThreadImp implements Runnable {
    public void run() {
        try {
            System.out.println("Begin ThreadImp");
            Thread.sleep(5000);
            System.out.println("End ThreadImp");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

程序运行结果为：

```

Begin ThreadImp
t has not finished
joinFinish
End ThreadImp

```

4.11 Java 数据库操作

4.11.1 如何通过 JDBC 访问数据库

Java 数据库连接 (Java Data Base Connectivity, JDBC) 用于在 Java 程序中实现数据库操作功能, 它提供了执行 SQL 语句、访问各种数据库的方法, 并为各种不同的数据库提供统一的操作接口, java.sql 包中包含了 JDBC 操作数据库的所有类。通过 JDBC 访问数据库一般有如下几个步骤:

- 1) 加载 JDBC 驱动器。将数据库的 JDBC 驱动加载到 classpath 中, 在基于 JavaEE 的 Web 应用开发过程中, 通常要把目标数据库产品的 JDBC 驱动复制到 WEB-INF/lib 下。

- 2) 加载 JDBC 驱动, 并将其注册到 DriverManager 中。一般使用反射 Class.forName(String driveName)。

- 3) 建立数据库连接, 取得 Connection 对象。一般通过 DriverManager.getConnection(url, username, passwd) 方法实现, 其中, url 表示连接数据库的字符串, username 表示连接数据库的用户名, passwd 表示连接数据库的密码。

- 4) 建立 Statement 对象或是 PreparedStatement 对象。

- 5) 执行 SQL 语句。

- 6) 访问结果集 ResultSet 对象。

- 7) 依次将 ResultSet、Statement、PreparedStatement、Connection 对象关闭, 释放掉所占用的资源, 例如 rs.close(), con.close() 等。为什么要这么做呢? 原因在于 JDBC 驱动在底层通常都是通过网络 IO 实现 SQL 命令与数据传输的。

常见笔试题:

1. 举出一个用 JDBC 访问 MySQL 的例子。

答案: 首先, 创建 Employee 表。

```
create table Employee(
    id int primary key,
    name varchar(20),
    age int
);
```

其次, 创建一个示例程序, 如下所示。

```
import java.sql.*;

public class Test {
    public static void main(String[] args) throws Exception {
        String user = "user1";
        String password = "pwd1";
        String url = "jdbc:mysql://localhost:3306/Test";
        String driver = "com.mysql.jdbc.Driver";
        Connection con = null;
        Statement stmt = null;
        ResultSet rs = null;
        try {
            Class.forName(driver);
            con = DriverManager.getConnection(url, user, password);
            stmt = con.createStatement();
            stmt.execute("insert into Employee values(1,'James1',25)");
            stmt.execute("insert into Employee values(2,'James2',26)");
            rs = stmt.executeQuery("select * from Employee");
            while(rs.next()) {
                System.out.println(rs.getInt(1) + " " + rs.getString(2) + " " + rs.getInt(3));
            }
        } catch (SQLException e1) {
            e1.printStackTrace();
        } finally {
            try {
                if (rs != null) rs.close();
                if (stmt != null) stmt.close();
                if (con != null) con.close();
            } catch (SQLException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

程序运行结果为:

```
1 James1 25
```

2. JDBC 的主要功能有 ()。

- A. 创建与数据库的连接
C. 处理数据并查询结果

- B. 发送 SQL 语句到数据库中
D. 以上都是

答案: D。见上面讲解。

3. 提供 Java 存取数据库能力的包是 ()。

- A. java. sql B. java. awt C. java. lang D. java. swing

答案: A。对数据库操作的所有类都在 java. sql 包中。

4.11.2 JDBC 处理事务采用什么方法

一个事务是由一条或多条对数据库操作的 SQL 语句所组成的一个不可分割的工作单元, 只有当事务中的所有操作都正常执行完了, 整个事务才会被提交给数据库。在 JDBC 中, 一般是通过 `commit()` 方法或 `rollback()` 方法来结束事务的操作。其中 `commit()` 方法表示完成对事务的提交, `rollback()` 方法表示完成事务回滚, 多用于在处理事务的过程中出现了异常的情况, 这两种方法都位于 `java. sql. Connection` 类中。一般而言, 事务默认操作是自动提交, 即操作成功后, 系统将自动调用 `commit()` 方法, 否则将调用 `rollback()` 方法。

当然, 在 JDBC 中, 也可以通过调用 `setAutoCommit (false)` 方法来禁止自动提交, 然后就可以把多个数据库操作的表达式作为一个事务, 在操作完成后调用 `commit()` 方法实现整体提交, 如果其中一个表达式操作失败, 就会抛出异常而不会调用 `commit()` 方法。在这种情况下, 就可以在异常捕获的代码块中调用 `rollback()` 方法进行事务回滚。通过此种方法可以保持对数据库的多次操作后, 数据仍然保持一致性。

引申: JDBC 有哪些事务隔离级别?

为了解决与“多个线程请求相同数据”相关的问题, 事务之间通常会用锁相互隔离开。如今, 大多数主流的数据库支持不同类型的锁。因此, JDBC API 支持不同类型的事务, 它们由 `Connection` 对象指派或确定。在 JDBC 中, 定义了以下 5 种事务隔离级别:

1) TRANSACTION_NONE JDB。不支持事务。

2) TRANSACTION_READ_UNCOMMITTED。未提交读。说明在提交前一个事务可以看到另一个事务的变化。这样读“脏”数据、不可重复读和虚读都是允许的。

3) TRANSACTION_READ_COMMITTED。已提交读。说明读取未提交的数据是不允许的。这个级别仍然允许不可重复读和虚读产生。

4) TRANSACTION_REPEATABLE_READ。可重复读。说明事务保证能够再次读取相同的数据而不会失败, 但虚读仍然会出现。

5) TRANSACTION_SERIALIZABLE。可序列化。是最高的事务级别, 它防止读“脏”数据、不可重复读和虚读。

(备注: ①读“脏”数据。一个事务读取了另一个事务尚未提交的数据, 例如, 当事务 A 与事务 B 并发执行时, 当事务 A 更新后, 事务 B 查询读取到 A 尚未提交的数据, 此时事务 A 回滚, 则事务 B 读到的数据是无效的“脏”数据。②不可重复读。一个事务的操作导致另一个事务前后两次读取到不同的数据, 例如, 当事务 A 与事务 B 并发执行时, 当事务 B 查询读取数据后, 事务 A 更新操作更改事务 B 查询到的数据, 此时事务 B 再次读去该数据, 发现前后两次的数据不一样。③虚读。一个事务的操作导致另一个事务前后两次查询的结果数据量不同, 例如, 当事务 A 与事务 B 并发执行时, 当事务 B 查询读取数据后, 事务 A 新增或删除了

一条满足事务 A 的查询条件的记录，此时，事务 B 再次查询，发现查询到前次不存在的记录，或者前次的某个记录不见了。）

事务隔离级别越高，为避免冲突所花的精力也就越多。可以通过 Connection 对象的 `conn.setTransactionLevel()` 方法来设置隔离级别，通过 `conn.getTransactionIsolation()` 方法来确定当前事务的级别。

4.11.3 Class.forName 的作用是什么

在 Java 语言中，任何类只有被装载到 JVM 上才能运行。Class.forName() 方法的作用就是把类加载到 JVM 中，它会返回一个与带有给定字符串名的类或接口相关联的 Class 对象，并且 JVM 会加载这个类，同时 JVM 会执行该类的静态代码段。

在使用 JDBC 连接数据库前，一般都会调用 Class.forName("com.mysql.jdbc.Driver") 方法来加载 JDBC 驱动，那么是否一定需要调用这个方法呢？如果是，那为什么要调用这个方法呢？其实，并不一定非要调用这种方法，例如 `Test t = (Test) Class.forName("Test").newInstance()` 语句和 `Test t = new Test()` 语句就具有相同的效果，所以使用 new 也可以，但二者的区别也非常明显：创建对象的方式不同。前者使用类加载机制，后者是创建了一个新的类。使用第一种方法往往能提高软件的可扩展性，例如，一个软件项目开发后会被多家公司来使用，每家公司的处理流程大致相同，只有个别公司的业务逻辑不同，在开发过程中，完全可以把不通用的地方抽取出来定义成一个接口 `BussinessInterface`，针对每个公司不同的业务流程定义不同的实现类 `sub1`、`sub2`、`sub3` 等，通过创建不同的子类来完成不同公司的业务需求。为了达到良好的可扩展性，可以把子类采用配置文件的方式放到 XML 文件中。在程序部署时，只需要从读配置文件中读取类名 `className`，然后采用 `BussinessInterface b = (BussinessInterface) Class.forName(className).newInstance()` 创建实例即可提高开发人员的开发效率。当以后再有新的需求时，即使开发了新的子类，也不需要修改创建实例的代码，只需要修改配置文件即可，从而使得程序具有很好的可扩展性。

JDBC 规范中要求 Driver 类在使用前必须向 DriverManager 注册自己，所以，当执行 Class.forName("com.mysql.jdbc.Driver") 时，JVM 会加载名字为 "com.mysql.jdbc.Driver" 对应的 Driver 类，而 com.mysql.Driver 类的实现如下例所示：

```
public class Driver extends NonRegisteringDriver implements java.sql.Driver {
    static {
        try {
            java.sql.DriverManager.registerDriver(new Driver());
        } catch (SQLException E) {
            throw new RuntimeException("Can't register driver!");
        }
    }
}
```

在调用 Class.forName() 方法时，这个 Driver 类被加载了，由于静态部分被执行，因此 Driver 也被注册到了 DriverManager 中。

4.11.4 Statement、PreparedStatement 和 CallableStatement 有什么区别

Statement 用于执行不带参数的简单 SQL 语句，并返回它所生成结果的对象，每次执行

SQL 语句时，数据库都要编译该 SQL 语句。以下是一个最简单的 SQL 语句：

```
Statement stmt = conn. getStatement();
stmt. executeUpdate("insert into client values('aa', 'aaaa')");
```

PreparedStatement 表示预编译的 SQL 语句的对象，用于执行带参数的预编译 SQL 语句。

CallableStatement 则提供了用来调用数据库中存储过程的接口，如果有输出参数要注册，说明是输出参数。下面给出一个使用 PreparedStatement 的例子：

```
import java. sql. * ;
public class Test {
    public static void main(String[] args) throws Exception {
        String user = "user1" ;
        String password = "pwd1" ;
        String url = "jdbc:mysql://localhost:3306/Test" ;
        String driver = "com. mysql. jdbc. Driver" ;
        Connection con = null ;
        PreparedStatement stmt = null ;
        ResultSet rs = null ;
        try {
            Class. forName(driver) ;
            con = DriverManager. getConnection(url, user, password) ;
            stmt = con. prepareStatement("select * from Employee where id = ?") ;
            stmt. setInt(1, 1) ; //传递参数(第一个问号,传递的值)
            rs = stmt. executeQuery() ;
            while(rs. next()) {
                System. out. println(rs. getInt(1) + " " + rs. getString(2) + " " + rs. getInt(3)) ;
            }
        } catch(SQLException e1) {
            e1. printStackTrace() ;
        } finally {
        }
        try {
            if(rs != null)
                rs. close() ;
            if(stmt != null)
                stmt. close() ;
            if(con != null)
                con. close() ;
        } catch(SQLException e) {
            System. out. println(e. getMessage()) ;
        }
    }
}
```

程序运行结果为：

虽然 Statement 对象与 PreparedStatement 对象能够完成相同的功能，但相比之下，PreparedStatement 具有以下优点：

1) 效率更高。在使用 PreparedStatement 对象执行 SQL 命令时，命令会被数据库进行编译和解析，并放到命令缓冲区。然后，每当执行同一个 PreparedStatement 对象时，由于在缓冲区中可以发现预编译的命令，虽然它会被再解析一次，但不会被再次编译，是可以重复使用的，能够有效提高系统性能，因此，如果要执行插入、更新、删除等操作，最好使用 PreparedStatement。鉴于此，PreparedStatement 适用于存在大量用户的企业级应用软件中。

2) 代码可读性和可维护性更好。以下两种方法分别使用 Statement 与 PreparedStatement 来执行 SQL 语句，显然方法 2 具有更好的可读性。

方法 1：

```
stmt.executeUpdate("insert into t(coll,col2) values('"+var1+"','"+var2+"')");
```

方法 2：

```
perstmt = con.prepareStatement("insert into tb_name (coll,col2) values (?,?)");
```

```
perstmt.setString(1,var1);
```

```
perstmt.setString(2,var2);
```

3) 安全性更好。使用 PreparedStatement 能够预防 SQL 注入攻击，所谓 SQL 注入，指的是通过把 SQL 命令插入到 Web 表单递交或输入域名或页面请求的查询字符串，最终达到欺骗服务器，达到执行恶意 SQL 命令的目的。注入只对 SQL 语句的编译过程有破坏作用，而执行阶段只是把输入串作为数据处理，不再需要对 SQL 语句进行解析，因此也就避免了类似 `select * from user where name = 'aa' and password = 'bb' or 1 = 1` 的 SQL 注入问题的发生。

CallableStatement 由 prepareCall() 方法所创建，它为所有 DBMS (Database Management System, 数据库管理系统) 提供了一种以标准形式调用已储存过程的方法。它从 PreparedStatement 中继承了用于处理输入参数的方法，而且还增加了调用数据库中的存储过程和函数以及设置输出类型参数的功能。

常见笔试题：

用于调用存储过程的对象是 ()。

A. ResultSet

B. DriverManager

C. CallableStatement

D. PreparedStatement

答案：C。JDBC 中的 CallableStatement 对象为所有 RDBMS (Relational Database Management System, 关系数据库管理系统) 提供了一种标准形式调用存储过程的方法。其对存储过程的调用存在两种形式：带结果参数和不带结果参数。结果参数是一种输出参数，是存储过程的返回值。两种形式都可带有数量可变的输入 (IN 参数)、输出 (OUT 参数) 或输入和输出 (IN-OUT 参数) 的参数。

4.11.5 getString() 方法与 getObject() 方法有什么区别

JDBC 提供了 getString()、getInt() 和 getData() 等方法从 ResultSet 中获取数据，当查询结果集中的数据量较小时，不用考虑性能，使用这些方法完全能够满足需求，但是当查询结果集中的数据量非常大时，则会抛出异常：OracleException 未处理：ORA - 01000: maximum open cursors exceeded (以访问 Oracle 数据库为例)。而通常情况下，使用 getObject() 方法就可以解决这个问题。

getString() 或 getInt() 等方法在被调用时，程序会一次性地把数据都放到内存中，然后通

过调用 `ResultSet` 的 `next()` 和 `getString()` 等方法来获取数据。当数据量大到内存中放不下时就会抛出异常，而使用 `getObject()` 方法就不会这种问题，因为数据不会一次性被读到内存中，每次调用时会直接从数据库中去获取数据，因此使用这种方法不会因为数据量过大而出错。

4.11.6 使用 JDBC 时需要注意哪些问题

在使用 JDBC 编程时，首先需要建立于数据库的连接，才能完成对数据库的访问，由于与数据库的连接是非常重要的资源。JDBC 连接池提供了 JDBC 连接定义和数目有限的连接，如果连接数量不够，就需要长时间的等待。不正常关闭 JDBC 连接会导致等待回收无效的 JDBC 连接。只有正常的关闭和释放 JDBC 连接，JDBC 资源才可以被快速地重用，从而使得系统性能得到改善。因此在编程时，一定要保证释放不再使用的连接。

一般来讲，在使用 JDBC 访问数据库时，`createStatement` 和 `prepareStatement` 最好放在循环外面，而且使用了这些 `Statement` 后，需要及时关闭。最好是在执行了一次 `executeQuery`、`executeUpdate` 等之后，如果不需要使用结果集 (`ResultSet`) 的数据，就马上将 `Statement` 关闭。因为每次执行 `conn.createStatement()` 或 `conn.prepareStatement()`，实际上都相当于在数据库中打开了一个 `cursor` (游标)，如果把对这两个方法的调用放到循环内，会一直不停地打开 `cursor`。如果不能及时地关闭，会导致程序抛出异常。

4.11.7 什么是 JDO

Java 数据对象 (Java Data Object, JDO) 是一个用于存取某种数据仓库中的对象的标准 API，它使开发人员能够间接地访问数据库。

JDO 是 JDBC 的一个补充，它提供了透明的对象存储，因此对开发人员来说，存储数据对象完全不需要额外的代码 (例如 JDBC API 的使用)。这些烦琐的工作已经转移到 JDO 产品提供商身上，使开发人员解脱出来，从而集中时间和精力在业务逻辑上。另外，相较于 JDBC，JDO 更灵活、更通用，它提供了到任何数据底层的存储功能，例如关系数据库、文件、XML 以及对象数据库管理系统 (Object Database Management System, ODBMS) 等，使得应用可移植性更强。

4.11.8 JDBC 与 Hibernate 有什么区别

Hibernate 是 JDBC 的封装，采用配置文件的形式将数据库的连接参数写到 XML 文件中，至于对数据库的访问还是通过 JDBC 来完成的。

Hibernate 是一个持久层框架，它将表的信息映射到 XML 文件中，再从 XML 文件映射到相应的持久化类中，这样可以使用 Hibernate 独特的查询语言 Hibernate 查询语言 (Hibernate Query Language, HQL) 了。Hibernate 的 HQL 查询语句返回的是 `List < Object[.] >` 类，而 JDBC 通过 `statement` 返回的查询结果是 `ResultSet` 并且有时候需要自己封装到 `List` 中。另外一个重要区别在于，Hibernate 具有访问层 (DAO 类层，DAO 全称为 Data Access Object 数据访问接口，意为数据访问接口)，该层是 HQL 查询语句唯一出现的位置，再往上层则不会出现查询语句，而 JDBC 可以随时连接随时访问，例如有 100 个类都有 SQL 查询语句，如果表名改变了，那么要使用 JDBC 的方式，就必须重写所有查询语句，而采用 Hibernate 的方式只需修改 DAO 层的类即可，因此 Hibernate 具有很好的维护性和扩展性。

第 5 章

Java Web

5.1 Servlet 与 JSP

5.1.1 页面请求的工作流程是怎样的

一个 Web 应用程序一般都是由客户端程序与服务器端程序两部分组成。其中，客户端主要是指用户和浏览器，用户可以通过浏览器查找所需的资源，而这些资源则位于服务器上。浏览器是一个工具软件，它主要有两个作用：一是完成与服务器端的交互；二是完成 HTML (Hyper-Text Markup Language, 超文本标记语言, 用来告诉浏览器怎样给用户展示内容) 的解析, 从而实现把用户需要查看的资源信息以直观的形式展现出来。服务器端用来接收客户端发来的请求, 并对该请求进行处理, 找到客户端请求的资源, 最后把查找到的资源返回给客户端, 这些资源主要包括 HTML 页面、图片、音频、视频、PDF 文件等内容。

图 5-1 给出了最基本的页面访问的处理流程。



图 5-1 页面访问的处理流程

- 1) 用户通过浏览器输入链接地址来请求所需的资源。
- 2) 浏览器接受用户的请求, 并把该请求组装成指定的格式发送给服务器端, 客户端与服务器端之间通过 HTTP 来完成具体的交互。其中请求的数据流中主要包含 HTTP (HyperText Transfer Protocol, 超文本传输协议, 建立在 TCP/IP 基础上的一个协议, 主要用来实现客户端与服务器端之间的通信) 请求方法 (GET 或 POST)、请求的网址 (URL, Uniform Resource Locator, 统一资源定位符) 以及请求的一些参数信息。
- 3) 服务器接收到客户端发来的请求, 并查找用户所需要的资源。
- 4) 服务器查找到用户请求的资源后, 将该资源返回给客户端。
- 5) 服务器通过把响应消息组装成特定的消息格式后返回给客户端, 这个过程通过 HTTP 来完成。响应的数据流主要包含状态编码 (代表请求成功或失败), Content-type (例如 text、picture、HTML 等), 响应消息的内容 (图片或 HTML 格式的内容)。

6) 浏览器对 HTML 进行解析, 并把响应结果展现给用户。

5.1.2 HTTP 中 GET 与 POST 方法有什么区别

HTTP 请求的方法有很多种类, 例如 GET、POST、HEAD、TRACE、OPTIONS 等, 但是 GET 与 POST 是两个最常用的方法。其中, GET 是最简单的一种请求方法, 其主要功能是从服务器端获取用户所需资源, 并将其作为响应返回给客户端, 这些资源可以是 HTML 页面、图片、文档等内容中的任何一种, 但需要注意的是, GET 方法的作用主要用来获取服务器端资源信息, 就如同数据库中查询操作一样, 不会影响到资源自身的状态, 例如删除、修改或新增资源都是不允许的。而 POST 方法提供了比 GET 方法更强大的功能, 它除了能够从服务器端获取资源外, 同时还可以向服务器上传数据。

虽然 GET 方法主要用来从服务器上获取数据, 也可以向服务器上传数据, 但是一般不建议采用 GET 方法来向服务器上传数据, 而是推荐使用 POST 方法实现该功能。具体而言, 主要有以下两个方面原因:

1) 采用 GET 方法向服务器上传数据时, 一般将数据添加到 URL 后面, 并且二者用“?”连接, 各个变量之间用“&”连接。由于对 URL 的长度存在限制, 因此采用这种方法能上传的数据量非常小, 通常在 1024Byte 左右。而 POST 方法传递数据是通过 HTTP 请求的附件进行的, 传送的数据量更大一些, 一般默认为不受限制的。

2) 由于 GET 方法上传的数据是添加在 URL 中的, 因此上传的数据被彻底“暴露”出来了, 本身存在安全隐患, 尤其是当用户需要向服务器提交一些敏感信息时。而 POST 方法向服务器提交的内容在 URL 中并没有明文显示, 对用户都是不可见的, 所以, 安全性更好一些。

常见笔试题:

在 HTTP 中, 用于发送大量数据的方法是 ()。

A. GET B. POST C. PUT D. OPTIONS

答案: B。见上面讲解。

5.1.3 什么是 Servlet

HTML 只能用来保存静态内容, 而通常情况下, 静态页面很难满足实际应用的需要, 鉴于此, 动态页面的概念被引入。所谓动态页面, 指的是能够根据不同时间、不同用户而显示不同内容的页面, 例如常见的论坛、留言板、电子商务网站等都是通过动态页面来实现的。那么如何才能生成动态页面呢? 其中一种方法是采用公共网关接口 (Common Gateway Interface, CGI)。CGI 是一种用 Perl 脚本编写的程序, 可以用来生成动态页面, 而另一种方式则是采用 Servlet 技术。

什么是 Servlet 呢? Servlet 是采用 Java 语言编写的服务器端程序, 它运行于 Web 服务器中的 Servlet 容器中, 其主要功能是提供请求/响应的 Web 服务模式, 可以生成动态的 Web 内容, 而这正是 HTML 所不具备的功能。

与其他生成动态页面的技术相比, Servlet 有诸多优点, 具体而言, 主要表现在如下几个方面:

1) 较好的可移植性。由于 Java 语言具有跨平台和可移植性强的特点, 使得 Servlet 也有较好的可移植性, 即无需修改代码就可以部署到多种不同类型的 Web 服务器上。

2) 执行效率高。由于 CGI 针对每个请求都会创建一个进程来处理, 而 Servlet 针对每个请

求创建一个线程来执行，而创建线程比创建进程的开销要小，因此与 CGI 相比，Servlet 在交互过程中有更短的响应时间，响应效率更高。

3) 功能强大。Servlet 可以与 Web 服务器进行交互，而 CGI 却无法与 Web 服务器直接交互。

4) 使用方便。Servlet 提供了许多非常有用的接口以用来读取或设置 HTTP 头消息，处理 Cookie 和跟踪会话状态等。

5) 可扩展性强。由于 Servlet 是用 Java 语言编写的，因此它具备了 Java 语言的所有优点。Java 语言是健壮的、面向对象的编程语言，它很容易扩展，Servlet 自然也具备这样的优点。

为了更好地说明 Servlet，在介绍 Servlet 处理请求的过程前，首先引入 Servlet 程序的基本结构，如下所示：

```
public class MyServlet extends HttpServlet {
    public void doPost (HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    }
    public void doGet (HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        Out.println("some html formate string");
    }
}
```

从以上 Servlet 的程序结构中可以看出，在 Servlet 中，并没有 main() 方法，连所谓的入口方法都没有，那么它到底是如何执行的呢？其实，它是在容器的控制下执行的，最常被使用的容器为 Tomcat。当 Web 服务器获取到一个对 Servlet 的请求时，该服务器将会把这个请求交给对应的容器来处理，容器通过调用 Servlet 的方法 (doGet() 或 doPost()) 来响应客户端的请求。

具体而言，Servlet 处理客户端请求有如下几个步骤：

1) 用户通过单击一个链接来向 Servlet 发起请求。

2) Web 服务器接收到该请求后，会把该请求交给相应的容器来处理，当容器发现这是对 Servlet 发起的请求后，容器此时会创建两个对象：HttpServletResponse 和 HttpServletRequest。

3) 容器可以根据请求消息中的 URL 消息找到对应的 Servlet，然后针对该请求创建一个单独的线程，同时把第 2) 步中创建的两个对象以参数的形式传递到新创建的线程中。

4) 容器调用 Servlet 的 service() 方法来完成对用户请求的响应，service() 方法会调用 doPost() 或 doGet() 方法来完成具体的响应任务，同时把生成的动态页面返回给容器。

5) 容器把响应消息组装成 HTTP 格式返回给客户端。此时，这个线程运行结束，同时删除第 2) 步创建的两个对象。

以上处理流程如图 5-2 所示。

容器会针对每次请求创建一个新的线程进行处理，同时会针对每次请求创建 HttpServletResponse 和 HttpServletRequest 两个对象。处理完成后线程也就退出了。

常见笔试题：

1. Servlet 处理请求的方式为 ()。

A. 以进程的方式 B. 以程序的方式 C. 以线程的方式 D. 以响应的方式

答案：C。见上面讲解。

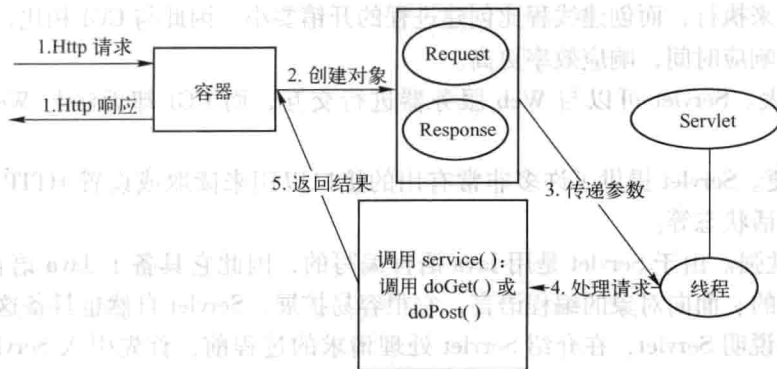


图 5-2 Servlet 的处理流程

2. Servlet 与 CGI 有什么区别?

答案: 相较于 CGI, Servlet 处于服务器进程之中, 它通过多线程方式运行其 `service()` 方法, 一个实例可以服务于多个请求, 并且其实例一般不会被销毁, 而 CGI 对每个请求都产生新的进程, 服务完成后就销毁, 所以效率不如 Servlet。

5.1.4 doPost() 方法与 doGet() 方法怎么选择

从 Web 容器处理 HTTP 请求的流程中可以看出, 最终请求都会交给 Servlet 来处理, 而 Servlet 是通过调用 `service()` 方法来处理请求的, `service()` 方法会根据不同的请求类型分别调用 `doPost()` 方法 (用于处理 POST 请求) 或 `doGet()` 方法 (用于处理 GET 请求) 来处理用户请求, 实现对客户的响应。如果请求是 GET, 就调用 `doGet()` 方法; 如果请求是 POST, 就调用 `doPost()` 方法。在 Servlet 接口和 `GenericServlet` (一个通用的、不特定于任何协议的 Servlet, 它实现了 Servlet 接口) 中是没有 `doGet()` 方法与 `doPost()` 方法的, 而 `HttpServlet` 中定义了这些方法, 但是都是返回 error 信息, 所以, 当定义一个 Servlet 时, 都必须实现 `doGet()` 方法或 `doPost()` 方法。`doGet` 和 `doPost` 都接受请求 (`HttpServletRequest`) 和响应 (`HttpServletResponse`)。

具体而言, 当 HTTP 请求中的 `method` 属性为 `get` 时, 调用 `doGet()` 方法; 当 `method` 属性为 `post` 时, 则调用 `doPost()` 方法。

5.1.5 什么是 Servlet 的生命周期

由于 Servlet 运行在容器中, 没有 `main()` 方法, 因此, 整个生命周期都是由容器来控制的。简单而言, Servlet 的生命周期只有两个状态: 未创建状态与初始化状态。这两种状态的转换主要是由 3 个重要的方法来进行控制: `init()`、`service()` 和 `destroy()`。其中, `init()` 方法是 Servlet 生命的起点, 用于创建或打开任何与 Servlet 相关的资源以及执行初始化工作。`service()` 方法是 Servlet 中真正处理客户端传过来的请求的方法, 它根据 HTTP 请求方法 (GET、POST 等) 将请求分发到 `doGet()`、`doPost()` 等方法。`destroy()` 方法释放任何在 `init()` 方法中打开的与 Servlet 相关的资源。

Servlet 的状态图如图 5-3 所示。

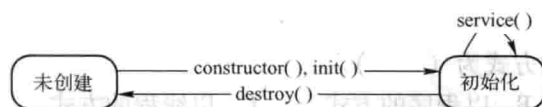


图 5-3 Servlet 的状态图

具体而言,Servlet 的生命周期可以分为加载、创建、初始化、处理客户请求和卸载 5 个阶段。

- 1) 加载。容器通过类加载器使用 Servlet 类对应的文件来加载 Servlet。
- 2) 创建。通过调用 Servlet 的构造函数来创建一个 Servlet 实例。
- 3) 初始化。通过调用 Servlet 的 `init()` 方法来完成初始化工作,这个方法是在 Servlet 已被创建但向客户端提供服务之前调用的,需要注意的是, `init()` 方法只会被调用一次。
- 4) 处理客户请求。Servlet 一旦被创建后,它就可以为客户端提供服务了。每当有新的客户请求到来时,容器都会创建一个新的线程来处理该请求,接着会调用 Servlet 的 `service()` 方法来完成客户端的请求,当然, `service()` 方法会根据请求的 `method` 属性值的不同调用决定是调用 `doGet()` 方法还是调用 `doPost()` 方法来完成具体的响应。
- 5) 卸载。容器在卸载 Servlet 之前需要调用 `destroy()` 方法,让 Servlet 自己释放其占用的系统资源,一旦 `destroy` 方法被调用,容器就不会再向这个 Servlet 发送任何请求消息了。如果容器需要这个 Servlet,那么就必须重新创建并初始化一个实例。需要注意的是, `destroy()` 方法只会被调用一次。

常见笔试题:

在 Servlet 的生命周期中,容器只调用一次的方法是 ()。

A. `service` B. `getServletConfig` C. `init` D. `destroy`

答案: C、D。见上面讲解。

5.1.6 JSP 有哪些优点

JSP (Java Server Pages) 是由 Sun 公司倡导、许多企业参与建立起来的一种动态技术标准,从本质上来讲,就是嵌入了 Java 代码的 HTML 文件(但需要注意的是, JSP 页面最好少写 Java 代码)。在引入 JSP 之前,在 Web 应用程序中,所有业务逻辑和 HTML 的响应都是在 Servlet 中实现的,但在使用这种方式时,存在一个比较大的缺陷:必须把给用户响应的视图组装成一个很长的 HTML 格式的字符串写入 `println()` 方法中。所以,使用这种方式编程,在编写一段很长的 HTML 字符串时非常容易出错,而且代码的可读性也非常差,同时由于业务逻辑与视图没有分离,系统的可扩展性、可维护性都较差。

JSP 的引入在一定程度上解决了 Servlet 存在的缺点。其实现理念是让每个 Servlet 只负责其对应的业务逻辑的处理,让 JSP 来负责用户的 HTML 显示,因此实现了业务逻辑与视图实现的分离,从而极大地提高了系统的可扩展性。

在引入 JSP 后,Servlet 的实现结构如下:

```
public class MyServlet extends HttpServlet {
    public void doGet (HttpServletRequest request, HttpServletResponse response) throws ServletException,
        IOException {
        //business logic code
        //forward the request to a JSP page
    }
}
```

5.1.7 JSP 与 Servlet 有何异同

JSP 与 Servlet 的相同点为: JSP 可以被看作一个特殊的 Servlet,它只不过是 Servlet 的扩

展，只要是 JSP 可以完成的工作，使用 Servlet 都可以完成，例如生成动态页面。由于 JSP 页面最终要被转换成 Servlet 来运行，因此处理请求实际上是编译后的 Servlet。

JSP 与 Servlet 的不同点为：① Servlet 的实现方式是在 Java 中嵌入 HTML 代码，编写和修改 HTML 非常不方便，所以它比较适合做流程控制、业务处理；而 JSP 的实现方式为在 HTML 中嵌入 Java 代码，比较适合页面的显示，例如，在 Struts 框架中，Servlet 位于 MVC 设计模式的控制层，而 JSP 位于视图层。② Servlet 中没有内置对象，JSP 中的内置对象都是必须通过 HttpServletRequest 对象、HttpServletResponse 对象以及 HttpSession 对象得到。

5.1.8 如何使用 JSP 与 Servlet 实现 MVC 模型

MVC 是 Model（模型）、View（视图）和 Controller（控制器）3 个单词的首字母组合。MVC 是一种目前广泛流行的应用模型，其目的是实现 Web 系统的职能分工。图 5-4 为 MVC 模型关系图，其中模型层实现系统中的业务逻辑，通常可以用 JavaBean 或 EJB 来实现；视图层则用于与用户的交互，通常用 JSP 来实现；控制层则是模型与视图之间沟通的桥梁，它可以把用户的请求分派并选择恰当的视图来显示它们，同时它也可以解释用户的输入并将其映射为模型层能够执行的操作。

MVC 强制性地分离 Web 应用的输入、处理和输出，使得 MVC 应用程序被分成 3 个核心部件：模型、视图和控制器。它们各自处理自己的任务。

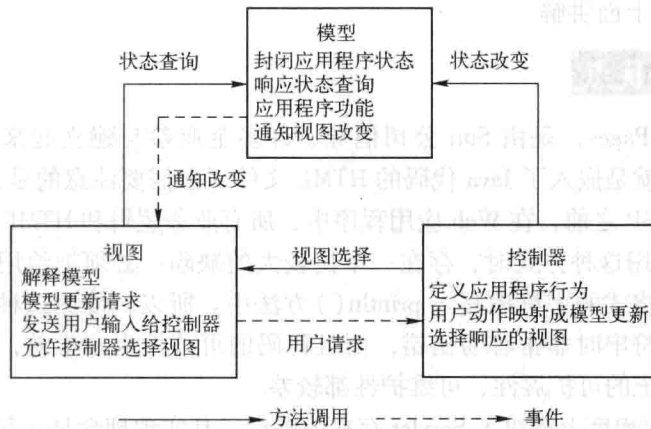


图 5-4 MVC 模型关系图

(1) 模型（业务逻辑层）

模型表示企业数据和业务逻辑，它是应用程序的主体部分。业务流程的处理过程对其他层来说是黑箱操作，模型接收视图请求数据，并返回最终的处理结果。业务模型的设计可以说是 MVC 最主要的核心。目前流行的 EJB 模型就是一个典型的应用例子，它从应用技术实现的角度对模型做了进一步的划分，以便充分利用现有的组件，但它不能作为应用设计模型的框架。它仅仅告诉设计人员按这种模型设计就可以利用某些技术组件，从而减少了技术上的困难，可以使设计人员专注于业务模型的设计。

MVC 把应用的模型按一定的规则抽象出来。抽象的层次很重要，这也是判断设计人员是否优秀的主要依据。抽象与具体不能隔得太远，也不能太近。MVC 并没有提供模型的设计方法，而只告诉设计人员应该如何组织管理这些模型，以便于模型的重构和提高重用性。

业务模型还有一个很重要的模型那就是数据模型。数据模型主要指实体对象的数据持续

化, 比如将一张订单保存到数据库, 从数据库获取订单, 将这个模型单独列出, 所有相关数据库的操作只限制在该模型中。

(2) 视图 (表示层)

视图是用户看到的并与之交互的界面。对早期的 Web 应用来说, 视图就是由 HTML 元素组成的界面, 在新式的 Web 应用中, HTML 依旧在视图中扮演着重要的角色, 但一些新的技术已层出不穷, 它们包括 Adobe Flash 以及诸如 XHTML、XML/XSL 等一些标识语言和 Web 服务等。

随着 Web 应用开发技术的发展, 用户要求的日益提高, 如何处理应用程序的界面已经变得越来越有挑战性。MVC 架构一个大的好处是它能为 Web 应用处理很多不同的视图。在视图中其实没有真正的业务处理发生, 不管这些数据是联机存储的还是一个雇员列表, 作为视图来讲, 它只是作为一种输出数据并允许用户操纵的方式。

视图功能强大, 主要表现在以下两个方面:

- 1) 根据客户类型显示信息。
- 2) 显示商业逻辑 (模型) 的结构, 而不关心信息如何获得何时获得。

(3) 控制器

控制器接收用户的输入并调用模型和视图去完成用户的需求。所以当用户单击 Web 页面中的超链接和发送 HTML 表单时, 控制器 (例如 Servlet) 本身不输出任何东西, 也不执行任何处理, 它只是接收请求并决定调用哪个模型构件去处理请求, 然后确定用哪个视图来显示模型处理返回的数据。

MVC 的处理过程是这样的: 对于每一个用户输入的请求, 先被控制器接收, 并决定由哪个模型来进行处理, 然后模型通过业务逻辑层处理用户的请求并返回数据, 最后控制器用相应的视图格式化模型返回的数据, 并通过显示页面呈现给用户。

MVC 这种特殊的设计结构, 给应用开发带来了很多便利, 通过使用 MVC 架构, 大大提高了 Web 应用的开发效率, 具体来说, MVC 设计结构主要有以下几个方面的优点:

1) 低耦合性。由于视图层和业务层分离, 这样就使得修改视图层代码时不需要重新编译模型和控制器的代码, 同样, 一个应用的业务流程或者业务规则的改变只需要改动 MVC 的模型层即可。因为模型与控制器和视图相分离, 所以很容易改变应用程序的数据层和业务规则。

2) 高重用性和可适用性。由于技术的不断进步, 现在访问应用程序可以有越来越多的方式。MVC 模式允许使用各种不同样式的视图来访问同一个服务器端的代码。它包括任何 Web (HTTP) 浏览器或者无线浏览器 (WAP), 例如, 用户可以通过计算机或手机来订购某样产品, 虽然订购的方式不一样, 但处理订购产品的方式是一样的。由于模型返回的数据没有进行格式化, 因此同样的构件能被不同的界面使用, 例如, 很多数据可能用 HTML 来表示, 但是也有可能用 WAP 来表示, 而这些表示所需要的命令仅是改变视图层的实现方式, 而控制层和模型层无需做任何改变。

3) 较低的生命周期成本。MVC 使得开发和维护用户接口的技术难度降低。

4) 部署快速。使用 MVC 模式可以大大缩减开发时间, 这使得后台开发人员集中精力于业务逻辑上, 使界面开发人员 (包括 HTML 和 JSP 开发人员) 集中精力于表现形式上。

5) 可维护性。分离视图层和业务逻辑层也使得 Web 应用更易于维护和修改。

6) 有利于软件工程化管理。由于采用了分层思想, 每一层不同的应用具有某些相同的特征, 有利于通过工程化、工具化管理程序代码。

用 JSP 与 Servlet 实现的 MVC 模型如图 5-5 所示。

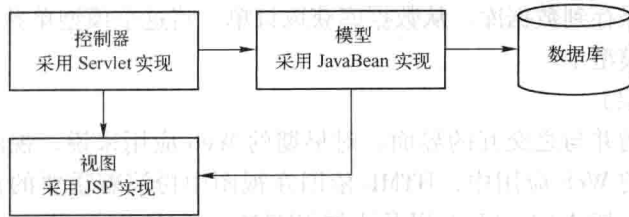


图 5-5 用 JSP 与 Servlet 实现的 MVC 模型

在这个 MVC 模型中，视图模块采用 JSP 来实现，主要负责数据的展现。视图可以从控制器上获取模型的状态，当然不是直接从控制器上获取，而是控制器把模型的数据放到一个视图可以访问的地方，通过间接方式来访问模型的数据。控制器采用 Servlet 来实现，客户端的所有请求都发送给 Servlet，它接收请求，并根据请求消息把它们分发给对应的 JSP 页面来响应，同时根据需求生成 JavaBean 实例供 JSP 来使用。模型采用 JavaBean 来实现，这个模块实现了实际的业务逻辑。

常见笔试题：

按照 MVC 设计模式，JSP 用于实现（ ）。

A. Model B. View C. Controller D. 容器

答案：B。见上面讲解。

5.1.9 Servlet 中 forward 和 redirect 有什么区别

在设计 Web 应用程序时，经常需要把一个系统进行结构化设计，即按照模块进行划分，让不同的 Servlet 来实现不同的功能，例如可以让其中一个 Servlet 接收用户的请求，另外一个 Servlet 来处理用户的请求。为了实现这种程序的模块化，就需要保证在不同的 Servlet 之间可以相互跳转，而 Servlet 中主要有两种实现跳转的方式：forward 方式与 redirect 方式。

forward 是服务器内部的重定向，服务器直接访问目标地址的 URL，把那个 URL 的响应内容读取过来，而客户端并不知道，因此在客户端浏览器的地址栏中不会显示转向后的地址，还是原来的地址。由于在整个定向的过程中用的是同一个 Request，因此 forward 会将 Request 的信息带到被定向的 JSP 或 Servlet 中使用。

redirect 则是客户端的重定向，是完全的跳转，即客户端浏览器会获取到跳转后的地址，然后重新发送请求，因此浏览器中会显示跳转后的地址。同时，由于这种方式比 forward 方式多了一次网络请求，因此其效率要低于 forward 方式。需要注意的是，客户端的重定向可以通过设置特定的 HTTP 头或写 JavaScript 脚本实现。

图 5-6 可以更好地说明 forward 与 redirect 的区别。

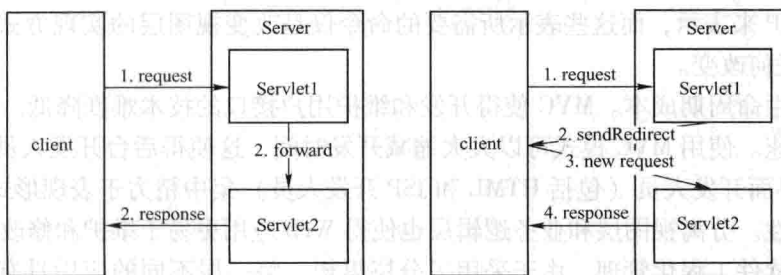


图 5-6 forward 与 redirect 的区别

鉴于以上区别，一般当 forward 方式可以满足需求时，尽可能地使用 forward 方式。但在有些情况下，例如，需要跳转到一个其他服务器上的资源，则必须使用 redirect 方式。

引申：filter 的作用是什么？主要实现什么方法？

filter 使用户可以改变一个 request 并且修改一个 response。filter 不是一个 Servlet，它不能产生一个 response，但它能够在 request 到达 Servlet 之前预处理 request，也可以在离开 Servlet 时处理 response。filter 其实是一个“Servlet Chaining”（Servlet 链）。

一个 filter 的作用包括以下几个方面：

- 1) 在 Servlet 被调用之前截获。
- 2) 在 Servlet 被调用之前检查 Servlet Request。
- 3) 根据需要修改 Request 头和 Request 数据。
- 4) 根据需要修改 Response 头和 Response 数据。
- 5) 在 Servlet 被调用之后截获。

常见笔试题：

下列有关 forward 和 redirect 的描述中，正确的是（ ）。

- A. forward 是服务器将控制权转交给另外一个内部服务器对象，由新的对象来全权负责响应用户的请求
- B. 执行 forward 时，浏览器不知道服务器发送的内容是从何处来，浏览器地址栏中还是原来的地址
- C. 执行 redirect 时，服务器端告诉浏览器重新去请求地址
- D. forward 是内部重定向，redirect 是外部重定向

答案：B、C、D。见上面讲解。

5.1.10 JSP 的内置对象有哪些

在 JSP 中，内置对象又称为隐含对象，是指在不声明和不创建的情况下就可以被使用的一些成员变量。JSP 一共提供有 9 个内置对象：request（请求对象）、response（响应对象）、pageContext（页面上下文对象）、session（会话对象）、application（应用程序对象）、out（输出对象）、config（配置对象）、page（页面对象）与 exception（例外对象）。JSP 内置对象的具体描述见表 5-1。

表 5-1 JSP 内置对象

名称	描述
request	客户端请求，此请求包含来自 GET/POST 请求的参数。客户端的请求信息被封装在 request 对象中，通过它才能了解到客户的需求，然后做出响应，因此 request 对象是用来获取请求参数的非常重要的途径。它是 HttpServletRequest 类的实例
response	用来表示服务器端对客户端的响应，将 Web 服务器处理后的结果返回给客户端。但在 JSP 中很少直接使用到它。它是 HttpServletResponse 类的实例
pageContext	提供了对 JSP 页面的所有对象及命名空间的访问，也就是说，用它可以访问到本页面中的所有其他对象，例如前面已经描述的 request、response 以及后面要介绍的 session 和 application 对象等。它的本类名也叫 pageContext
session	用来表示客户端与服务器的一次会话。从客户端与 Web 服务器建立连接的时候会话开始，直到关闭浏览器时结束会话。它是 HttpSession 类的实例
application	代表 JSP 所属的 Web 应用本身。application 对象可存放全局变量，因此可以实现用户间的数据共享。它的生命周期与服务器的生命周期一致，也就是说，服务器启动后这个对象即被创建出来，直到服务器停止后这个对象的生命周期才结束。在任何地方，对此对象属性的操作都将影响到其他用户对此的访问。它是 ServletContext 类的实例

名 称	描 述
out	用于在客户端浏览器内输出信息。它是 JspWriter 类的实例
config	主要作用是取得服务器的配置信息。当一个 Servlet 初始化时，容器把某些信息通过 config 对象传递给这个 Servlet，Servlet 可以使用这个对象获取所需的配置信息
page	表示当前 JSP 页面，类似于 Java 中的 this 指针。它是 java.lang.Object 类的实例
exception	用来表示异常。当一个页面在运行过程中发生了例外，就会产生这个对象。如果 JSP 需要使用这个对象，就必须把 isErrorPage 设为 true，否则将无法编译。它是 java.lang.Throwable 的对象

根据以上 9 个内置对象的作用的不同，可以将它们分为 4 类：第一类，与 Servlet 有关的 page 和 config；第二类，与 Input/Output 有关的 out，request 和 response；第三类，与 Context 有关的 application，session 和 pageContext；第四类，与 Error 有关的 exception。

常见笔试题：

JSP 主要内置对象有 _____、_____、_____、_____、_____、_____、_____、out、config 和 page。

答案：见上面讲解。

5.1.11 request 对象主要有哪些方法

当使用 JSP 与 Servlet 开发 Web 应用程序时，如何获取用户提交的请求信息是非常重要的内容之一。request 对象就是用来封装用户请求数据的，每当有请求到达服务器时，系统都会创建一个 request 对象。在服务器进行处理时可以通过获取 request 对象的属性来获取用户的请求数据。此外，还可以通过对 request 对象设置新的一些属性来实现在 Servlet 与 JSP 之间跳转时传递一些参数的功能。具体来讲，request 对象的方法见表 5-2。

表 5-2 request 对象的方法

名 称	描 述
setAttribute(String name, Object)	用来设置名字为 name 所对应的属性值。在对请求进行转发处理时，也可以通过该方法设置一些属性，从而将数据传递到转发后的页面中
getAttribute(String name)	用来获取名字为 name 所对应的属性值，可以用来获取通过 setAttribute 方法设置的一些属性
getAttributeNames()	返回 request 对象所有属性的名字集合。其返回值为枚举的实例
getCookies()	用来返回客户端的所有 Cookie 对象，其结果是一个 Cookie 数组
getCharacterEncoding()	用来返回请求的消息中字符的编码方式
getContentLength()	用来获取请求消息的 Body 的长度
getInputStream()	返回请求的输入流，这个输入流可以被用于获得请求中的数据
getMethod()	用来获取 HTTP 的请求方式，如 get 或 post
getParameter(String name)	用来获取用户提交的数据，其中 name 与表单中的 name 属性一一对应
getParameterNames()	用来获取客户端发送给服务器端的所有参数的名字，其结果是一个枚举的实例
getParameterValues(String name)	获得有 name 指定的参数的所有值
getProtocol()	用来获取客户端向服务器端传送数据所使用的协议名称
getQueryString()	用来获得查询字符串，返回值为 URL 后面的参数串
getRequestURI()	用来获取发出请求字符串的客户端地址
getRemoteAddr()	获取客户端的 IP 地址
getRemoteHost()	获取客户端的名字
getSession(boolean create)	用来获取与请求相关 session

(续)

名 称	描 述
getServerName()	获取服务器的名字
getServletPath()	用来获取客户端所请求的脚本文件 (JSP 或 Servlet) 的路径
getServerPort()	用来获取服务器的端口号
removeAttribute(String name)	用来删除名字为 name 对应的属性

常见笔试题:

HttpServletRequest 对象的 () 方法能够获取一个表单参数的值。

A. getQueryString() B. getPathInfo() C. getParameter() D. getAttribute()

答案: C。见上面讲解。

5.1.12 JSP 有哪些动作

JSP 使用动作来实现动态地插入文件、实现重定向和对 JavaBean 的引用等功能。它共有 6 个基本动作: `jsp:include`、`jsp:useBean`、`jsp:setProperty`、`jsp:getProperty`、`jsp:forward` 和 `jsp:plugin`。下面将分别对这些动作进行具体介绍。

1) `jsp:include`。用来在页面被请求时引入一个文件。`include` 指令是在 JSP 文件被转换成 Servlet 时引入文件, 而 `jsp:include` 插入文件的时间是在页面被请求时, 而且被引用文件不能包含某些 JSP 代码 (例如不能设置 HTTP 头), 示例如下:

```
<jsp:include page = "test.jsp" flush = "true" >
    <jsp:param name = "name" value = "value" />
</jsp:include >
```

以上代码表示在当前文件中可以引入 `test.jsp` 文件。

2) `jsp:useBean`。用来寻找或者实例化一个 JavaBean。它使得开发人员既可以发挥 Java 组件重用的优势, 同时也避免了损失 JSP 区别于 Servlet 的方便性, 示例如下:

```
<jsp:useBean id = "car" scope = "session" class = "com.Car" >
```

以上代码表示实例化了一个 `com.Car` 类的实例。

3) `jsp:setProperty`。用来设置已经实例化的 Bean 对象的属性, 示例如下:

```
<jsp:setProperty name = "car" property = "colour" value = "red" />
```

以上代码用来将名字为 `car` 的实例的 `colour` 属性设置为 `red`。

4) `jsp:getProperty`。用来获取某个 JavaBean 的属性, 示例如下:

```
Colour = <jsp:getProperty name = "car" property = "colour" > </jsp:getProperty >
```

以上代码用来获取名字为 `car` 的实例的 `colour` 属性。

5) `jsp:forward`。用来把请求转到一个新页面, 示例如下:

```
<jsp:forward page = "/Servlet/login" />
```

以上代码用来把当前页面重定向到 `/Servlet/login` 来处理。

6) `jsp:plugin`。用于在浏览器中播放或显示一个对象。使用这个动作能插入所需的特定的浏览器的 OBJECT 或 EMBED 元素来指定浏览器运行一个 JAVA Applet 所需的插件, 示例如下:


```
<jsp:plugin type = "applet" codebase = "/ch5" code = "Hello.class" height = "40" width = "320" >
```

以上代码用来在浏览器中运行一个 applet 插件。

5.1.13 JSP 中 include 指令和 include 动作有什么区别

include 的主要作用是用来在当前文件中引入另外一个文件，以便在当前文件中使用，例如，当应用程序中的所有页面的某些部分（例如标题、页脚、导航栏等）都一模一样时，就可以考虑把相同的部分提取出来写入一个单独的文件中，然后通过 include 方式引入。

include 有两种使用方法：include 指令和 include 动作。其中，include 指令的使用方式为：`<%@ include file = "test.jsp" %>`，include 动作的使用方式为：`<jsp:include page = "test.jsp" flush = "true" ><jsp:param name = "name" value = "value"/></jsp:include >`。

include 指令与 include 动作之间的根本性差异在于二者被调用的时间。include 指令是编译阶段的指令，即在编译时，编译器会把指令所指向目标文件的内容复制到指令所在的位置，替换指令，最终形成一个文件，在运行时只有一个文件。也就是说，include 指令所包含文件的内容是在编译时插入到 JSP 文件中的，当文件内容有变化时就需要重写编译，因此适合于包含静态页面的情况，例如可以包含一个 Servlet。而 include 动作是运行时的语法，在主页面被请求时，才将用到的页面包含进来，涉及两个文件，类似于方法调用，因此更适用于包含动态页面的情况。除此之外，二者的差别还有以下 3 点：

1) 当使用 include 动作时，在页面中声明的变量不可用于另一文件，除非将变量放置在 request、session、application 作用域中；而在使用 include 指令时，当前页面和被包含页面可以共享变量。

2) 当使用 include 指令时，新生成的 JSP 页面要符合 JSP 语法要求，应该避免变量名的冲突；而在使用 include 动作时，不存在变量名冲突问题。

3) include 指令会修改被包含文件，但不会立即生效，除非修改主页面或删除主页面的类；而 include 动作修改了被包含的文件，会立即生效。

考虑到 include 动作在维护上的优势，当这两种方法都适用时，优先考虑使用 include 动作。仅在所包含文件中定义了主页面要用到的字段或方法，或所包含文件设置了主页面的响应报头时，才应该使用 include 指令，例如，很多站点的导航部分和版权信息部分都是相同的，在每个文件中都会出现，所以可以考虑把这些内容放在单独的文件中，然后使用 `<%@ include% >` 指令引用即可。

常见笔试题：

在 JSP 中，可动态导入其他页面的标签是（ ）。

A. `<% include/ ></textarea >`

B. `<%@ include% >`

C. `<jsp:importPage/ >`

D. `<jsp:include/ >`

答案：D。见上面讲解。

5.1.14 会话跟踪技术有哪些

在开发 Web 应用程序时，经常需要能够做到数据共享或者是在不同页面之间可以传递参数，而且，一个会话中的数据可能会在不同的地方使用，因此就需要有专用的机制来传递和保存这些数据。

所谓会话，指的是从客户端打开与服务器的连接并发出请求到服务器响应客户端请求的全过程。会话跟踪则是对同一个用户对服务器的连续请求和接受响应的监视，由于客户端与服务器端

之间是通过 HTTP 进行通信的,而 HTTP 本身是无状态协议,它不能保存客户的信息,即一次响应完成之后连接就断开了,在下次请求时,需要重新建立连接,等到建立完连接后还需要判断是否是同一个用户,因此,要想对会话的过程进行监控,最好的方法就是使用会话跟踪技术。

具体而言,会话跟踪技术主要有如下 4 种:

1) page。代表与一个页面相关的对象和属性。一个页面由一个编译好的 Java Servlet 类(可以带有任何的 include 指令,但是没有 include 动作)表示。这既包括 Servlet 又包括被编译成 Servlet 的 JSP 页面。

2) request。代表与 Web 客户端发送的一个请求相关的对象和属性。一个请求可能跨越多个页面,涉及多个 Web 组件。

3) session。代表与用于某个 Web 客户端的一个用户体验相关的对象和属性,一个 Web 会话可以也经常会跨越多个客户端请求。

4) application。代表与整个 Web 应用程序相关的对象和属性,这实质上是跨越多个 Web 应用程序,包括多个页面、请求和会话的一个全局作用域。

常见笔试题:

如果只希望在多个页面间共享数据,可以使用()作用域。

A. request, session

B. application session

C. request, application

D. pageContext request

答案:A。见上面讲解。

5.1.15 Web 开发中如何指定字符串的编码

在 Java 语言中,常用的字符编码方式有 ISO-8859-1、GB2312、GBK、UTF-8/UTF-16/UTF-32 等。其中,ISO-8859-1 用来编码拉丁文,它由单字节(0~255)组成。GB2312、GBK 用来编码简体中文,由单字节和双字节混合组成,最高位为 1 的字节和下一个字节构成一个汉字,最高位为 0 的字节是 ASCII 码。UTF-8/UTF-16/UTF-32 是国际标准 UNICODE 的编码方式,UTF 的全称为 Unicode Translation Format,即把 Unicode 转作某种格式的意思。其中,使用最多的编码方式是 UTF-8,因为该方式在对拉丁文编码时节约空间,它的特点是对不同范围的字符使用不同长度的编码。

String 序列化成 byte 数组或反序列化时需要选择正确的编码方式,如果编码方式使用不正确,就会得到一些 0x3F 的值。所以,在 Web 应用开发中,经常会遇到需要指定字符串的编码格式的情况,为了防止出现乱码,最好的方法就是指定编码格式。下面的例子可以把字符串以 ISO-8859-1 的编码格式输出:

```
public String translate (String str) {
    String result = "";
    try {
        result = new String( str.getBytes("ISO-8859-1"), "GBK");
        result = result.trim();
    }
    catch (Exception e) {
        System.err.println(e.getMessage());
    }
    return result;
}
```

需要注意的是，在 Web 开发时也可以通过 `response.setContentType()` 方法来指定 JSP 页面显示的编码格式。

常见笔试题：

为了让浏览器以 UTF-8 编码显示 JSP 页面，请问，下列 JSP 代码中，正确的是（ ）。

A. `<% page contentType =`

B. `<meta http-equiv =`

C. 把所有输出内容重新编码：`new String(content.getBytes())`

D. `response.setContentType()`

答案：D。选项 A 设置的是服务器端编码，选项 B 设置的是客户端编码，选项 C 设置的是某个字符串的编码。只有选项 D 满足题意。

5.1.16 什么是 Ajax

Ajax (Asynchronous JavaScript and XML, 异步 JavaScript 与 XML) 是一个结合了 Java 技术、XML 以及 JavaScript 的编程技术，其主要目的是在不刷新页面的情况下通过与服务器进行少量数据的交互来提高页面的交互性，减少响应时间，从而改善用户体验。使用 Ajax 技术后，页面就不需要在每次用户提交修改时重新加载了。

在使用传统软件架构开发的应用程序中，当客户端需要与服务器端频繁交互时，用户只有等整个页面重新加载后才能看到从服务器中获取到的资源信息，页面会被重新加载很多次。当前后两个页面中的大部分 HTML 代码相同时，这种做法就会非常浪费网络带宽，毕竟很多资源信息的获取都是重复无用的。

在这种情况下，如果使用 Ajax 技术会带来许多好处：首先，由于 Ajax 技术可以只向服务器发送并取回必需的数据内容，使得数据交互量大幅降低，从而降低了服务器的网络负载；其次，由于它通过使用 SOAP (Simple Object Access Protocol, 简单对象访问协议，一种交换数据的协议规范) 或其他一些基于 XML 的 Web Service 接口，在客户端采用 JavaScript 处理来自服务器的响应，也降低了 Web 服务器的处理时间；最后，由于不需要重新加载整个页面，因此系统有更短的响应时间，而这有利于提高系统的稳定性与可用性，从而增强用户的满意度。

需要注意的是，Ajax 技术是客户端技术，其核心是 JavaScript 对象 `XmlHttpRequest`，该对象是一种支持异步请求的技术，它使得开发人员可以使用 JavaScript 向服务器提出请求并处理响应，而不阻塞用户。

5.1.17 cookie 和 session 有什么区别

cookie 是在 HTTP 下，服务器或脚本可以维护客户工作站上信息的一种方式。它是由 Web 服务器保存在用户浏览器上的小文件，可以包含有关用户的信息（如身份识别号码、密码等信息）。session 是指用来在客户端与服务器端之间保持状态的解决方案以及存储结构。

尽管二者都能够进行信息存储，但是也存在着区别。具体而言，二者有以下几个方面的区别：

1) cookie 机制采用的是在客户端保持状态的方案，即数据存放在客户的浏览器上；而 session 机制采用的是在服务器端保持状态的方案，即数据放在服务器上。

2) cookie 安全性不够。由于 cookie 信息存放在客户端，其他人可以很容易地得到存放在本地的 cookie，并进行 cookie 欺骗；而 session 信息存放在服务器端，因此较为安全。

3) cookie 性能更高一些。由于 session 会在一定时间内保存在服务器上，因此当访问量增

多时，会降低服务器的性能。

4) 单个 cookie 保存的数据不能超过 4 KB，很多浏览器都限制一个站点最多保存 20 个 cookie；而 session 不存在此问题。

鉴于以上几点区别，一般情况下，将用户登录信息等重要信息存放至 session 中，而其他需要保留的信息可以放在 cookie 中。

5.2 J2EE 与 EJB

5.2.1 什么是 J2EE

J2EE (Java2 Platform, Enterprise Edition) 是 Java 平台企业版的简称，是用来开发与部署企业级应用的一个架构，它提供了一种统一的、开放标准的多层平台，该平台主要由构件、服务和通信 3 个模块构成。

构件包含客户端构件和服务器端构件两种类型，其中，客户端构件主要包含两类 Applets 和 Application Clients，服务器端构件分为两类 Web 构件 (Servlet 与 JSP) 和 EJBs (Enterprise Java Beans) 两种。服务由 J2EE 平台提供商实现，分为 Service API (开发时使用) 和运行时服务。通信由容器提供的支持协作构件之间的通信。

从本质上来讲，J2EE 只是一个行业标准，主要用来通过 Java 开发服务器端应用提供一个独立的、可移植的、多用户的企业级平台，从而能够简化应用程序的开发和部署。正是由于 J2EE 只是一个标准而不是一个成熟的产品，因此目前有很多不同类型的 J2EE 服务器。只要开发的应用程序符合 J2EE 的标准，就都可以部署在遵循了 J2EE 开发标准的 J2EE 服务器上。这种标准使得开发人员只需要专注于各种应用系统的商业逻辑与架构设计，而不用过多地考虑底层繁琐的程序编写工作，系统的开发与部署效率大幅提升。

常见笔试题：

在 J2EE 中，属于 Web 层的组件有 ()。

A. Servlet B. EJB C. Applet D. HTML

答案：A。见上面讲解。

5.2.2 J2EE 中常用的术语有哪些

J2EE 中常用的术语有 Web 服务器、Web 容器、EJB 容器、Applet 容器、Application Client 容器、JNDI、JMS、JTA、JAF、RMI 等。下面将分别对它们进行解释。

(1) Web 服务器

Web 服务器是指驻留在 Internet 上的计算机程序。它是一种服务程序，其主要工作是接收来自于客户端的请求，然后把对请求的处理结果返回给客户端。用户可以通过浏览器来请求所需资源，这些资源可以是 HTML 页面、图片、音频、视频或者 PDF 文档等，Web 服务器接收到请求后会去查找用户请求的资源，然后把找到的资源返回给用户。需要注意的是，它是一个被动的程序，只有当收到来自客户端的请求后，才会发送响应。

目前最流行的两大 Web 服务器是：IIS (Internet Information Services) 和 Apache Http Server (简称 Apache)。

(2) Web 容器

Web 容器也被叫作 Web 应用服务器，它是一种服务程序，用来给运行在其中的程序 (例

如 Servlet、JSP 等) 提供一个运行的环境。由于 Web 容器的存在, Servlet 只需要关注业务逻辑的处理而不用关注与客户端的交互, 因为这些交互都已经由 Web 容器来协助完成了。常见的 Web 容器有 Tomcat、JBoss、WebLogic 和 WebSphere 等。

下面以 Tomcat 为例介绍一下 Web 容器主要完成了哪些工作: 当 Web 服务器(例如 Apache) 接收到一个客户端对 Servlet 的请求后, 不是直接把这些请求交给 Servlet 来处理, 而是交给 Web 容器来处理, 由 Web 容器负责给 Servlet 提供 HTTP 请求与响应对象, 同时容器调用 Servlet 的 doGet() 或 doPost() 方法来处理用户的请求。

(3) EJB 容器

EJB 容器是一个服务器端容器。EJB (Enterprise JavaBean) 是 J2EE 应用的业务层的技术标准, 只要满足 J2EE 规范的组件就能在 EJB 容器中运行, 这个组件就会被 EJB 容器高效地管理。同时, EJB 容器还给运行在其中的组件提供了安全而优越的服务环境, 例如事务管理、邮件服务等。

需要注意的是, EJB 组件不能显式地使用 EJB 容器的 API (Application Programming Interface, 应用程序接口) 来请求容器提供的中间件服务, 但可以隐式地让 EJB 容器知道它们的需求, 例如在基于 XML 的部署描述符中制定所需的配置信息, 在 Bean 类中使用部署注释请求中间件服务。EJB 容器为 EJB 组件隐式地提供中间件服务。

(4) Applet 容器

Applet 容器是一个客户端容器, 包含的组件为 Applet。Applet 是一种嵌入在浏览器中的轻量级客户端, 一般而言, 只有当使用 Web 页面无法充分地表现数据或应用界面时, 才会使用它。Applet 是一种替代 Web 页面的手段, 开发人员只能使用 J2SE 开发 Applet, 出于安全性的考虑, Applet 无法使用 J2EE 的各种 Service 和 API。

(5) Application Client 容器

Application Client 容器是一个客户端容器, 包含的组件为 Application Client。相对 Applet 而言, Application Client 是一种较重量级的客户端, 它能够使用 J2EE 的大多数 Service 和 API。

(6) JNDI

JNDI 全称为 Java Naming and Directory Interface, 译为 Java 命名和目录接口。它提供了一个目录系统, 并将服务名称与对象关联起来, 从而使得开发人员在开发过程中可使用名称来访问对象。

通过使用 JNDI, 一方面实现了快速查找和定位分布式应用程序的功能, 另一方面使得程序有了更好的可扩展性。由于 JNDI 是独立于目录协议的, 因此应用还可以使用 JNDI 访问各种特定的目录服务, 例如轻量级目录访问协议 (Lightweight Directory Access Protocol, LDAP)、网络数据服务 (Network Data Service, NDS) 和域名系统 (Domain Name System, DNS) 等。

(7) JMS

JMS 全称为 Java Message Service, 译为 Java 消息服务。它是一个 Java 平台中面向消息中间件的 API, 主要实现各个应用程序之间进行异步通信, 包括创建、发送、接收、读取消息等。通过使用 JMS, 能够最大限度地提升消息应用的可移植性。JMS 既支持点对点的消息通信, 也支持发布/订阅式的消息通信。

(8) JTA

JTA 全称为 Java Transaction API, 译为 Java 事务服务。它提供各种分布式事务服务 (在多个网络计算机上进行数据访问和更新), 为 J2EE 实现分布式事务处理提供了重要的支撑。

(9) JAF

JAF 的全称为 JavaBeans Activation Framework, 译为 JavaBeans 激活框架。它是一个专用的

数据处理框架，提供了一种统一处理不同数据格式的方法。

(10) RMI

RMI 全称为 Remote Method Invocation，译为远程方法调用，它主要用于远程调用服务，通过它，可以像调用本地函数一样调用另外一台计算机上运行的程序，而不需要关心远程计算机所使用的系统或所使用的语言，也就是说，只要满足一定的规范，就可以实现在不同的计算机上进行函数调用。

常见笔试题：

JNDI 可用于如下哪些应用场景？（多选题）

- A. 配置信息存储
- B. 异步信息发送
- C. 数据库连接池查找
- D. 远程对象查找

答案：D。见上面讲解。

5.2.3 EJB 有哪些不同的类别

EJB 是 Enterprise JavaBean 的简称，相当于分布式组件对象模型（Distributed Component Object Model, DCOM），它是一种服务器端组件体系结构，用于开发和部署多层的、分布式的以及面向对象的应用系统的跨平台体系结构。EJB 简化了 Java 开发企业级的分布式组件应用程序的过程，它定义了一组可重用的组件——Enterprise Beans，开发人员可以利用这些组件，像搭积木一样建立各种分布式应用。所以，使用 EJB 可以写出可扩展的、健壮的、安全的应用程序，而不需要开发人员自己去编写复杂的分布式组件框架。

依据特性的不同，EJB 可以分为以下 3 种类型：Session Bean（会话 Bean），Entity Bean（实体 Bean）和 Message Driven Bean（消息驱动 Bean）。下面将分别对这 3 种 Bean 进行介绍。

(1) Session Bean

Session Bean 用来实现服务器端的业务逻辑，同时协调 Bean 之间的交互。Session Bean 仅存在于客户应用与服务器交互的时间段内，Session Bean 中的数据是不保存在数据库中的。根据 Session Bean 是否有状态又可以分为两种类型：Stateless Session Bean（无状态的 Session Bean）和 Stateful Session Bean（有状态的 Session Bean）。

Stateless Session Bean 在方法调用期间不维护任何状态，所有事务处理都是在一个方法中处理完成，因此一个 Stateless Session Bean 可以被多个客户共享，即一个 Stateless Session Bean 可以同时处理多个客户应用的请求；而 Stateful Session Bean 则不同，它可以记录客户应用请求的状态，例如在线购物系统中，每个客户都拥有购物车（Stateful Bean），而装载的货物和货物的数量在 Session Bean 中通过方法来操作，因此每个 Stateful Session Bean 只能处理一个客户的请求。与 Stateless Session Bean 相比，Stateful Session Bean 的优点是能够记录客户应用的状态，其缺点是它不能被共享，开销较大，性能逊于 Stateless Bean，因此当请求用户数量比较多时会消耗更多的内存。

(2) Entity Bean

Entity Bean 主要是资料组件，代表数据库中的记录，因此它与数据库中的数据有着相同的生存周期，也就是说，只要数据库中的数据存在，Entity bean 就一直存在。此外，Entity Bean 还可以被多个客户应用共享。Entity 有两种对数据持久化的处理方式：① CMP（Container Managed Persistence，容器管理的持续性）。构件的相关数据库操作由容器自动完成，不需要在 bean 中编写数据库操作的代码。② BMP（Bean Managed Persistence，Bean 管理的持续性）。构件的相关数据库操作由开发人员在代码中通过 JDBC 编程来实现。这两种 Entity Bean 的形态不

同，但目的相同，二者的主要区别在于维护资料的角色，CMP 是由 EJB Container 来维护，而 BMP 则是由 Bean 自行维护资料的一致性。

Entity Bean 一共有 3 种状态：no - state、pooled 和 ready。其中，no - state 状态表示 Bean 实例还没有被创建；pooled 状态表示 Bean 实例已经被创建，但还没有和一个 EJB Object 关联起来；ready 状态表示与 EJB Object 关联起来，若断开关联，则回到 pooled 状态。

(3) Message Driver Bean

Message Driver Bean 用来处理异步消息，一般不是由用户来调用的。它的调用方式为：当有异步消息发送到某个 Message Driver Bean 后，容器会负责调用 Message Driver Bean 的 OnMessage() 方法来处理这个异步请求。

5.2.4 EJB 与 JavaBean 有什么异同

EJB 是基于 Java 的远程方法调用 (Remote Method Invocation, RMI) 技术，可以被远程访问 (跨进程、跨计算机)，但它必须被部署在 Webspere、WebLogic 等容器中。EJB 客户从不直接访问真正的 EJB 组件，而是通过其容器访问。EJB 容器是 EJB 组件的代理，EJB 组件由容器所创建和管理。

Java Bean 是可复用的组件，理论上讲，任何一个 Java 类都可以是一个 Bean。但通常情况下，由于 Java Bean 是被容器 (例如 Tomcat) 所创建的，因此 Java Bean 应具有一个无参的构造器，另外，通常 Java Bean 还要实现 Serializable 接口用于实现 Bean 的持久性。Java Bean 实际上相当于微软 COM 模型中的本地进程内 COM 组件，是不能被跨进程访问的。

EJB 与 JavaBean 都是基于 Java 语言的构件模型。在开发应用时，既可以选择使用 EJB 模型，也可以选择使用 JavaBean 模型。尽管如此，二者也并非完全通用，其主要区别见表 5-3。

表 5-3 EJB 与 JavaBean 对比

EJB	JavaBean
主要用在服务器端的开发	主要用在客户端的开发
可以作为独立的单元被部署到 EJB 容器中	不可部署
支持使用部署描述符对 EJB 应用进行定制化	定制化只能在开发阶段
EJB 构件分布式对象，可以被远程的对象访问	JavaBean 构件不是分布式对象，只能在应用程序内部被访问
EJB 构件对终端用户不可见	部分 JavaBean 对终端用户可见

5.2.5 EJB 有哪些生命周期

Stateless Session Bean 的生命周期是由容器决定的，当客户端发出请求要建立一个 Bean 的实例时，EJB 容器不一定要创建一个新的 Bean 实例来供客户端调用，如果当前有 Stateless Session Bean 的实例能够满足客户需求，就不会去创建新的实例。对于 Stateless Session Bean 来说，实例的创建和删除都是由 EJB 容器来控制的。Stateless Session Bean 的生命周期共有两个状态：No State (无状态) 和 Method Ready (准备方法)，图 5-7 是两种状态的状态转移图。

其中，No State 表示容器中没有 Stateless Session Bean 的实例。只有 EJB 容器认为实例池中需要更多的实例为客户提供服务时，才会去创建新的实例，此时新创建的实

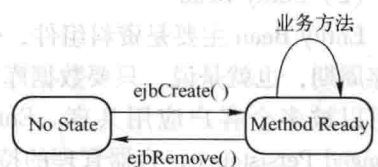


图 5-7 No State 和 Method Ready 的状态转移图

例就由 No State 转换为 Method Ready 状态；相反，如果 EJB 容器认为当前实例池中已经不需要这么多的实例来给客户端提供服务，就会根据某些策略从实例池中删除一些实例，这些被删除的实例的状态就转换为 No State。

Stateful Session Bean 的生命周期是与用户的操作相关联的，由于 Stateful Session Bean 不能被共享，因此当客户端第一次调用一个 Stateful Session Bean 时，容器必须立即在服务器中创建一个新的 Bean 实例，并关联到客户端上，以后当此客户端调用 Stateful Session Bean 的方法时，容器会把调用分派到与此客户机相关联的 Bean 实例。Stateful Session Bean 的实例在生命周期中共有 3 个状态：No State（无状态）、Method Ready（准备方法）和 Passivated（钝化）。状态间的转换如图 5-8 所示。

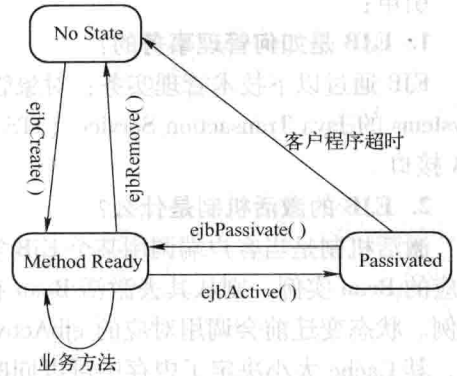


图 5-8 Stateful Session Bean 的状态转换图

其中，No State 表示容器中没有 Stateful Session Bean 的实例。只要有新的用户请求到来，EJB 容器就会创建新的 Stateful Session Bean 实例，此时被创建的实例的状态就从 No State 转换为 Mechod Ready。当这个实例不再被使用时（客户端不再使用或客户端超时），EJB 容器就会删除该实例，此时这个实例的状态就会转换为 No State。与 Stateless Session Bean 不同的是，Stateful Session Bean 不能被共享，因此在某一时刻 EJB 容器中可能会有大量的 Stateful Session Bean 实例，从而导致内存开销过大，为了解决这个问题，在 Stateful Session Bean 的生命周期中引入了另外一个状态——Passivated。当 Stateful Session Bean 实例过多时，EJB 容器只保留那些正在被使用的实例，从而把当前不被使用的实例从内存移到持久存储介质上（例如硬盘），此时这些实例的状态就转换为 Passivated。只有当客户再次请求使用处于 Passivated 状态的实例时，这些实例才会被激活并恢复到内存中，同时其状态也转换为 Method Ready。当实例不再被使用或超时后，其状态都会转换为 No State。

Method Ready 状态与 Passivated 状态之间通过根据 MRU 或 NRU 算法进行迁移。状态迁移前会调用对应的 ejbActive() 和 ejbPassivate() 方法。

Entity Bean 能存活相对较长的时间，并且状态是持续的。只要数据库中的数据存在，Entity Bean 就一直存活，而不是按照应用程序或者服务进程来说的，即使 EJB 容器崩溃了，Entity Bean 也是存活的。Entity Bean 的生命周期能够被容器或者 Bean 自己管理。为了很好地管理 Entity Bean，它的生命周期被划分为 3 个状态：No State、Pooled State 和 Ready State，这 3 个状态之间的转换如图 5-9 所示。

其中，No State 同样表明实例不存在。当 EJB 容器创建新的实例后，被创建的实例会进入 Pooled State，处于这个状态的实例不与任何 EJB 对象关联，与数据库记录对应的字段也未初始化；当客户端调用 Home 接口中的方法来创建或找到某个实体 bean 时，该实体的状态就会从 Polled State 转换到 Ready State，处于 Ready State 的实例已经建立

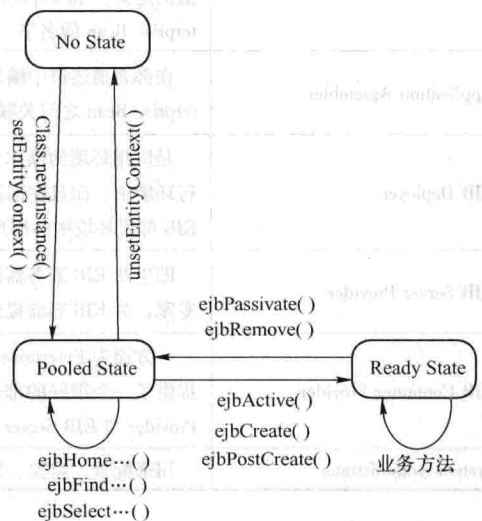


图 5-9 Entity Bean 的状态转换图

了与 EJB 对象的关联，同时也已经和数据库中的记录对应起来了。与 Session Bean 一样，当 Entity Bean 的对象不再使用时其状态就会转换为 No State。

引申：

1. EJB 是如何管理事务的？

EJB 通过以下技术管理事务：对象管理组织（OMG）的对象事务服务（OTS），Sun Microsystems 的 Java Transaction Service（JTS）、Java Transaction API（JTA），开发组（X/Open）的 XA 接口。

2. EJB 的激活机制是什么？

激活机制是当客户端调用某个 EJB 实例业务方法时，若对应 EJB Object 发现自己没有绑定对应的 Bean 实例，则从其去激活 Bean 存储中（通过序列化机制存储实例）恢复（激活）此实例。状态变迁前会调用对应的 `ejbActive()` 和 `ejbPassivate()` 方法。以 Stateful Session Bean 为例，其 Cache 大小决定了内存中可以同时存在的 Bean 实例的数量，根据 MRU（Most Recently Used，最近最常使用）算法或 NRU（Not Recently Used，最近未使用）算法，实例在激活和去激活状态之间迁移。

5.2.6 EJB 的角色有哪几种

EJB 将开发部署 EJB 应用的任务划分到了 6 个不同的角色，它们分别为 Enterprise Bean Provider（企业级组件开发者）、Application Assembler（应用组合者）、EJB Deployer（EJB 部署者）、EJB Server Provider（EJB 服务器提供者）、EJB Container Provider（EJB 容器提供者）和 System Administrator（系统管理员）。每个角色都可以由不同的开发商提供，每个角色所做的工作都必须严格遵循 EJB 规范，并保证彼此之间的兼容性。

表 5-4 为 EJB 中 6 个角色及其功能描述。

表 5-4 EJB 中 6 个角色及其功能描述介绍

角 色	功 能 描 述
Enterprise Bean Provider	用来编写 EJB 应用所需的构件。主要包含 Enterprise Bean（包含应用的商业逻辑）、Home 接口（包含 Enterprise Bean 生命周期管理的相关方法）、Remote 接口（包含商业方法的定义）和 Deployment Descriptor（在部署描述符中，Enterprise Bean Provider 指明 Enterprise Bean 的名字、事务属性以及安全性角色等）
Application Assembler	在部署描述符中编写组装 Enterprise Bean 的代码。这些代码中包含构成应用的多个 Enterprise Bean 之间关联的定义
EJB Deployer	是应用环境的操作专家，负责与 EJB 部署相关的工作。它们将 Enterprise Bean 安装到运行环境中，而且在部署时可以进行定制化，例如可以在部署时把定义的安全角色和 EJB 部署环境中的用户组及账号对应起来，实现它们的映射
EJB Server Provider	即生产 EJB 服务器的厂家，它们是分布式事务、分布式对象和其他低级系统级服务的专家，为 EJB 容器提供运行时环境和服务框架
EJB Container Provider	一方面为 Enterprise Bean 的执行提供系统级服务，如 Home 接口的自动注册；另一方面提供了一个很好的部署工具，可以将 Enterprise Bean 部署到 EJB 容器中。EJB Container Provider 与 EJB Server Provider 一般都由同一厂商来提供
System Administrator	用来配置、监控、维护 EJB Server 与部署的 Enterprise Bean 的日常运行

需要注意的是，以上 6 个角色可以由不同的开发商提供，但有一个前提，即它们都必须遵循 EJB 规范并且保证相互之间的兼容性。

5.2.7 EJB 的开发流程是怎样的

在 EJB 的开发流程中有非常重要的 3 个对象：Remote 接口、Home 接口和 Bean 类。其中 Remote 接口定义了业务方法，用于 EJB 客户端调用业务方法，需要注意的是，开发 EJB 时，只需要对其进行定义即可，并不需要去实现它，因为容器会在编译时会根据接口定义和 Enterprise Bean 类的内容自动创建一个实现了该接口的类。Home 接口提供了产生和定位 Remote 接口实例的方法，用于 EJB 工厂创建、移除、查找 EJB 实例，它又分为远程 Home 接口与本地 Home 接口两种。Bean 类指的是实现了 `javax.ejb.EnterpriseBean` 接口的类，它定义了应用的商业逻辑。

具体而言，EJB 开发步骤如下所示。

1) 定义 Remote 接口。编写一个接口继承自 `javax.ejb.EJBObject`。在这个接口中定义需要实现业务逻辑的方法对应的接口。需要注意的是，这个接口中定义的类都必须抛出 `RemoteException` 异常，示例如下。

```
import javax.ejb.*;
import java.rmi.*;
public interface HelloInterface extends EJBObject {
    String hello() throws RemoteException;
}
```

2) 编写一个接口继承 `javax.ejb.EJBHome`。这个接口中实现 `create()` 方法，这个方法的返回值必须为 Remote 接口类型（可以被远程访问的接口），同时这个方法还需要抛出两个异常：`CreateException` 和 `RemoteException`。`create()` 方法取代传统 OO（Object Oriented，面向对象）中的 constructor（构造方法）来初始化一个对象。EJB 容器可以维护一定数量的对象实例，示例如下。

```
import javax.ejb.*;
public interface HelloHome extends EJBHome {
    public HelloInterface create() throws
        java.rmi.RemoteException, javax.ejb.CreateException;
}
```

以上这个例子中的 `create()` 方法返回值为上面创建的 `HelloInterface` 接口。

3) 编写 bean 类的实现类（`SessionBean`）。实现类必须实现下面 5 个方法：

- ① `public void ejbCreate()` 方法。EJB 容器会调用这个方法创建 enterprise bean 的对象。
- ② `public void ejbRemove()` 方法。
- ③ `public void ejbActivate()` 方法。
- ④ `public void ejbPassivate()` 方法。
- ⑤ `public void setSessionContext(SessionContext ctx)`

```
this.ctx = ctx;
```

以上 4 个方法（`ejbRemove()`、`ejbActivate()`、`ejbPassivate()`、`setSessionContext()`）主要用来控制 `SessionBean` 的生命周期。具体而言，容器在删除 enterprise bean 的实例之前会调用其 `ejbRemove()` 方法。容器在将 enterprise bean 的实例从临时持久存储介质加载到内存中后会调用其 `ejbActivate()` 方法。容器在将 enterprise bean 的实例转移到临时持久存储介质之前会调用其 `ejbPassivate()` 方法。每次创建一个 `SessionBean` 时，容器会调用 `setSessionContext()` 方法，用来

初始化 enterprise bean 使用的 SessionContext 变量。具体使用方法如下所示：

```
import javax.ejb.*;

public class HelloBean implements SessionBean {
    SessionContext context;

    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}
    public void setSessionContext(SessionContext context) {
        this.context = context;
    }

    public String hello() {
        return "Hello world!";
    }
}
```

4) 在项目的 META-INF 目录中创建两个文件：ejb-jar.xml 和 jboss.xml。其中，ejb-jar.xml 文件的内容如下所示：

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>HelloBean </ejb-name>
      <home>HelloHome </home>
      <remote>HelloInterface </remote>
      <ejb-class>HelloBean </ejb-class>
      <session-type>Stateless </session-type>
      <transaction-type>Container </transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

jboss.xml 文件的内容如下所示：

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>HelloBean </ejb-name>
      <jndi-name>Hello </jndi-name>
    </session>
  </enterprise-beans>
</jboss>
```

5) 把上述 EJB 项目发布到 JBoss 容器上以后，客户端就可以调用该 EJB 对象了。

客户端调用 EJB 的步骤如下：设置 JNDI 服务工厂以及 JNDI 服务地址系统属性，查找 Home 接口，从 Home 接口调用 create() 方法创建 Remote 接口，通过 Remote 接口调用其业务方法，示例如下：

```
import javax.naming.*;

public class Client {
    public static void main(String[] args) {
```



```

try {
    Context initial = new InitialContext();
    //从 JNDI 中以 Hello 名字来定位到对象
    HelloHome home = (HelloHome)initial.lookup("Hello");
    HelloInterface h = home.create();
    System.out.println(h.hello());
} catch (Exception e) {
    System.out.println(e.getMessage());
}
}

```

5.2.8 EJB 3.0 与 EJB 2.0 有哪些不同之处

EJB 3.0 在 EJB 2.0 的基础上, 引入了更多的概念, 而且它对开发过程实现了进一步的简化, 使得开发过程变得更加方便。具体而言, 就是在系统开发时, 不用再像 EJB 2.0 那样需要两个接口和一个 Bean 的实现类, 它主要通过 annotations 元注释来标明 Session Bean 的类型, 例如, 在实现上例中的 Stateless Bean 时就可以用下面的方式来实现:

```

import javax.ejb.*;
@Stateless
@Remote
public class HelloBean {
    public String hello() {
        return "Hello world!";
    }
}

```

从这个例子可以看出, EJB 3.0 在开发时省去了许多方法 (例如 `ejbCreate()`、`ejbRemove()` 等), 大大简化了开发工作。

EJB 3.0 与 EJB 2.0 最大的不同之处就在于 EJB 3.0 通过元数据、去掉部署文件、省去对 Home 接口的使用等方法简化了开发的工作。此外, 在访问数据库的方法上也有所不同: EJB 2.0 在访问数据库时使用的是 Entity Bean, 而 EJB 3.0 用的是 JPA (Java Persistence API)。

5.2.9 EJB 容器有哪些作用

所有 EJB 都必须运行在 EJB 容器中, EJB 容器是 Enterprise JavaBean 的拥有者, 它负责管理运行在其中的 bean。EJB 容器提供了许多服务, 使得开发人员只需要把开发重点放在业务逻辑上。具体来讲, EJB 容器提供了一些非常重要的服务: 事务管理、持久性管理、安全管理、并发访问控制管理、生命周期管理和代码生成等。

5.2.10 EJB 规范规定 EJB 中禁止的操作有哪些

具体而言, EJB 规范规定 EJB 中禁止的操作有以下 8 种:

- 1) 不能操作线程和线程 API (线程 API 指非线程对象的方法, 例如 `notify`、`wait` 等)。
- 2) 不能操作 AWT (Abstract Windowing Toolkit, 抽象窗口工具包)。
- 3) 不能实现服务器功能。

- 4) 不能对静态属性存取。
- 5) 不能使用 IO 操作直接存取文件系统。
- 6) 不能加载本地库。
- 7) 不能将 this 作为变量, 不能将 this 返回。
- 8) 不能循环调用。

5.2.11 Web 服务器与 Web 应用服务器有什么区别

Web 服务器是可以向发出请求的浏览器提供文档的程序。其主要功能是提供网上信息浏览服务; 即接收浏览器的请求并把处理结果传回浏览器进行显示。Web 服务器支持以 HTTP 的方式来访问, 当 Web 服务器接收到一个 HTTP 请求时, 它同样会以 HTTP 格式返回一个响应, 这个响应可以是一个静态的 HTML 页面, 也可以是结果处理的一个动态的页面, 还可以是音频、视频等信息。为了处理一个请求, Web 服务器可以做出一个响应, 并进行页面跳转, 或者把动态响应的产生委托给一些其他程序, 例如 CGI 脚本、JSP、Servlet 或者一些其他的服务器端程序。Web 服务器一般都使用了一些特有的机制 (例如容错机制) 来保证 Web 服务器有较好的扩展性, 并能提供不间断的服务。常见的 Web 服务器有 IIS 和 Apache。

应用服务器提供访问业务逻辑的途径以供客户端应用程序使用。具体而言, 它通过 HTTP、TCP/IP、IIOP (Internet Inter - ORB Protocol, 互联网内部对象请求代理协议) 或 JRMP (Java Remote Method Protocol, Java 远程方法协议) 等协议来提供业务逻辑接口。为了系统的可靠性, 同样使用了一些可扩展性和容错机制。除此之外, 它还为应用的开发提供了许多服务, 例如事务管理、安全管理、对象生命周期管理等。常见的应用服务器有 BEA WebLogic Server, IBM WebSphere Application Server, IPlanet Application Server, Oracle9i Application Server, JBoss 和 Tomcat 等。

Web 服务器一般是通用的, 而应用服务器一般是专用的, 例如 Tomcat 只处理 Java 应用程序而不能处理 ASPX 或 PHP。需要注意的是, Web 服务器与应用服务器是并列关系, 二者不存在相互包容关系。在使用时, 如果访问的页面只有 HTML, 用 Web 服务器就足够了, 但是如果是 JSP, 此时就需要应用服务器了, 因为只有应用服务器才能解析 JSP 里的 Java 代码, 并将解析结果以 HTML 的格式返回给用户。

5.2.12 什么是 Web Service

Web Service 是一种基于网络的分布式模块化组件, 它可以将可调用的功能发布到 Web 上以供应用程序访问 (应用程序可以使用标准的 Web 协议和数据格式来访问它)。由于 Web Service 遵循一定的技术规范, 使得它能够与其他组件或系统有很好的兼容性。

具体而言, Web Service 是基于下面的一些协议来实现的。

- 1) 可扩展可标记语言 (eXtensible Markup Language, XML)。它是实现 Web Service 的基础, 非常适用于在网络上传输数据时使用。

- 2) Web 服务描述语言 (Web Service Description Language, WSDL)。它是采用 XML 语言来描述 Web Service 属性的语言。它将 Web Service 描述为能够进行消息交换的服务访问点的集合, 具体定义了 Web Service 可以做什么、在哪里以及怎样去调用。

- 3) 通用描述、发现与集成服务 (Universal Description, Discovery and Integration, UDDI)。它是一种由 OASIS (Organization for the Advancement of Structured Information Standards, 结构化

信息标准促进组织) 制定的规范, 主要提供基于 Web 服务的注册和发现机制, 为 Web 服务提供 3 个重要的技术支持: ①标准、透明、专门描述 Web 服务的机制; ②调用 Web 服务的机制; ③可以访问的 Web 服务注册中心。它维护了一个 Web Service 的全球目录, 其中的信息描述格式也是基于 XML 格式的。UDDI 的核心组件是 UDDI 商业注册, 它使用 XML 文档来描述企业及其提供的 Web Service。

4) 简单对象存取协议 (Simple Object Access Protocol, SOAP)。它是 Web Service 的通信协议。当用户通过 UDDI 找到对应的 WSDL 描述符后, 就可以通过 SOAP 调用 Web 服务中的操作。SOAP 也基于 XML 描述的方法调用规范。

调用 Web Service 时, 服务提供者把所提供的服务发布到服务代理的一个目录上, 然后服务请求者使用 UDDI 首先到服务代理提供的目录上搜索服务, 得到如何调用该服务的信息 (WSDL), 然后根据得到的调用信息使用 SOAP 调用服务提供者提供的服务, 如图 5-10 所示。

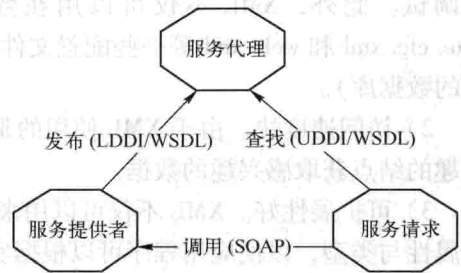


图 5-10 Web Service 规范

Web Service 的这种实现方式使其拥有很多优点, 具体表现为如下几个方面:

- 1) 完好的封装性。服务使用者只需要知道 Web Service 提供的功能列表, 而不需要关心具体的实现。
- 2) 松耦合。Web Service 在不改变接口的情况下可以随意改变实现方式, 而且不会影响到服务的使用者。也就是说, 服务的提供者与使用者互不影响。
- 3) 高度客户操作性。可以跨平台、跨语言进行调用。
- 4) 动态性。可以自动发现服务并进行调用。

5.2.13 SOAP 与 REST 有什么区别

SOAP 是一个严格定义的信息交换协议, 用于在 Web Service 中把远程调用和返回封装成机器可读的格式化数据, SOAP 数据使用 XML 数据格式, 定义了一整套复杂的标签, 来描述调用的远程过程、参数、返回值和出错信息等内容。

REST (Representational State Transfer, 表述性状态转移) 形式上为客户端通过申请资源来实现状态的转换, 可以被看作一台虚拟的状态机。需要注意的是, 它只是一种软件架构风格, 而不是一个具体的协议或标准, 而且, 它是面向资源的, 甚至连服务也会被抽象成资源。

表 5-5 为 SOAP 与 REST 的对比。

表 5-5 SOAP 与 REST 的对比

	SOAP	REST
寻址模型	URI 只用来定位 SOAP 端点; 资源与 URI 是一一对应; 一端点对应多个资源	标准化的 URI、DNS; URI 与资源 (包括服务) 一一对应
接口	不提供通用操作, 每个服务定义自己的方法 (操作)	提供通用操作, 即 HTTP 的 GET、PUT、POST 和 DELETE
中间媒介	不兼容传统的 Web 中间媒介	兼容传统的 Web 中间媒介, 包括代理、缓存服务器、网管等
安全性	十分复杂, 不能使用现有防火墙控制	简单有效, 可用现有防火墙控制

5.2.14 什么是 XML

可扩展标记语言 (eXtensible Markup Language, XML) 是一套定义语义标记规则的语言, 可以被用来描述业务数据、数学数据等。XML 一个非常重要的用途就是实现了系统的解耦机制 (例如, Web Service 和 JMS 都大量使用了 XML 文件, 实现了系统不同模块的解耦)。那么 XML 有哪些优点呢? 相较于 HTML, XML 将用户界面与结构化数据分隔开来, 这种数据与显示的分离使得集成来自不同源的数据成为可能。除此之外, XML 还有以下 4 个主要优点。

1) 实用性强。由于 XML 是以文本而不是二进制的方式存储的, 因此很容易对其进行修改和调试。此外, XML 不仅可以用在数据量较小的场合 (例如 `struts - config.xml`、`hibernate.cfg.xml` 和 `web.xml` 等一些配置文件), 同时也适用于存储大量的数据 (例如基于 XML 存储的数据库)。

2) 访问速度快。由于 XML 使用的是层次结构, 因此可以非常方便、快速地通过深入到感兴趣的结点获取感兴趣的数据。

3) 可扩展性好。XML 不仅可以用来存储和显示数据, 而且还能通过相关的属性标记数据的属性与类型, 以便应用程序可以根据实际情况选取不同的数据。同时, 由于 XML 中的标记可以由使用者自己定义, 因此它具有很好的可扩展性。

4) 跨平台性好。XML 具有统一的标准语法, 因此, 大部分系统支持的 XML 文件都具有相同的语法结构, 这就使得它具有很好的跨平台特性。

尽管 XML 优点出众, 但其也存在其不足之处。XML 的主要缺点就是当数据量过大时, 它的存储效率会变得较低, 往往会比其他存储方式 (例如 CSV) 占用更大的存储空间。因此, 如果要在网络中通信的场合、带宽受限的情况下, 一般不适合使用 XML。

XML 文档定义有两种形式: 文档类型定义 (Document Type Define, DTD) 与 Schema。它们一方面用于定义 XML 文档的结构, 另一方面用于验证 XML 文档是否满足指定的结构。具体而言, DTD 是一套标记的语法规则, 一个 XML DTD 定义了 XML 文档的元素架构、元素标记和属性, 规定了用户在 DTD 关联的 XML 文档中可以使用什么标记、各个标记出现的顺序以及标记的层次关系, 并定义了实体。当建立 XML 文档时, 通常需要按照 DTD 规范来进行, 反过来也可以通过对文档进行 DTD 验证, 检验 XML 文档建立的正确性。DTD 不能定义一些必要的限制条件, 例如元素出现的次数、数据类型 (例如整型、浮点型、布尔型) 等, 因此 DTD 更适于以文档为中心的 XML 内容。而 Schema 与 DTD 不同, 它本身基于 XML, 是用一套预先规定的 XML 元素的属性创建的, 这些元素的属性定义了 XML 文档的结构和内容模式。XML Schema 同时还支持命名空间, 能够定义比 DTD 更复杂的数据类型和结构。XML Schema 内置支持一系列的简单数据类型, 例如字符串、小数和整数等, 还可以定义元素出现的次数。除此之外, XML Schema 利用命名空间将文档中的特殊节点与 Schema 说明相联系, 一个 XML 文件可以有多个 Schema, 而对于 DTD, 一个 XML 只能有一个相对应的 DTD, 因此 XML Schema 更适合以数据为中心的文档。所以, 越来越多的应用偏向于采用 XML Schema 来定义和验证 XML 文档。

目前, 对 XML 的解析最主要的方式有两种: DOM 和 SAX (Simple API for XML, XML 简单 API)。DOM 方式会根据给定的 XML 文件在内存中创建一个树形结构, 因此, 这种处理方法会占用较多的内存, 在处理大文件时效率会急剧下降。而且 DOM 必须在解析文件之前把整个文档装入内存, 所以该方式主要适用于对 XML 的随机访问与频繁对 XML 中的内容进行修改的场合。而 SAX 是事件驱动型的 XML 解析方式, 它不会在内存中存储 XML 文件的内容, 只

是把每次对数据的请求看作一个事件，通过遍历文件来获取用户所需的数据。当遇到像文件开头、文档结束或者标签开头与标签结束时，它会触发一个事件，用户通过在其回调事件中写入处理代码来处理 XML 文件。所以，它的使用场合一般为对 XML 的顺序访问、XML 文件太大以至于在内存中放不下等情况。

下面分别给出它们操作 XML 的示例。首先创建一个 test.xml，代码如下：

```
<?xml version = "1.0" encoding = "UTF - 8" ? >
<people >
<p >
    <name > name1 </name >
    <age > 23 </age >
</p >
<p >
    <name > name2 </name >
    <age > 24 </age >
</p > </people >
```

采用 DOM 方式进行解析，示例代码如下：

```
import java. io. File;
import org. w3c. dom. * ;
import javax. xml. parsers. * ;
public class Test {
    public static void main( String arge[ ] ) {
        try {
            File f = new File( " test. xml" ) ;
            DocumentBuilderFactory factory = DocumentBuilderFactory. newInstance( ) ;
            DocumentBuilder builder = factory. newDocumentBuilder( ) ;
            Document doc = builder. parse( f ) ;
            NodeList nl = doc. getElementsByTagName( " p" ) ;
            for ( int i = 0 ; i < nl. getLength( ) ; i ++ ) {
                System. out. print( " 姓名:" + doc. getElementsByTagName( " name" ) . item ( i )
                    . getFirstChild( ) . getNodeValue( ) ) ;
                System. out. println( " 年龄:" + doc. getElementsByTagName( " age" ) . item ( i )
                    . getFirstChild( ) . getNodeValue( ) ) ;
            }
        } catch ( Exception e ) {
            System. out. println( e. getMessage( ) ) ;
        }
    }
}
```

程序运行结果为：

```
姓名: name1 年龄: 23
姓名: name2 年龄: 24
```

采用 SAX 方式进行解析，示例代码如下：

```
import java. io. File;
import javax. xml. parsers. SAXParser;
```

```

import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class Test extends DefaultHandler {

    public void characters(char[] ch,int start,int length) throws SAXException {
        System.out.print(new String(ch,start,length));
        super.characters(ch,start,length);
    }

    public void endDocument() throws SAXException {
        super.endDocument();
    }

    public void endElement(String url,String localName,String qName) throws SAXException {
        System.out.print(" </" + qName + " >");
        super.endElement(url,localName,qName);
    }

    public void startDocument() throws SAXException {
        System.out.println("<?xml version = \"1.0\" encoding = \"UTF - 8\"? >");
        super.startDocument();
    }

    public void startElement(String uri,String localName,String qName,Attributes attrs) throws SAXException {
        System.out.print(" <" + qName);
        if (attrs != null) {
            for (int i=0; i < attrs.getLength(); i++) {
                System.out.print(" " + attrs.getQName(i) + "=\"" + attrs.getValue(i) + "\"");
            }
        }
        System.out.print(" >");
        super.startElement(uri,localName,qName,attrs);
    }

    public static void main(String[] args) throws Exception {
        SAXParserFactory factory = SAXParserFactory.newInstance();
        SAXParser parser = factory.newSAXParser();
        File f = new File("test.xml");
        Test dh = new Test();
        parser.parse(f,dh);
    }
}

```

程序运行结果为：

```

<?xml version = "1.0" encoding = "UTF - 8" ? >
<people >
<p >
    <name >name1 </name >
    <age >23 </age >
</p >
<p >

```



```

<name>name2</name>
<age>24</age>
</p></people>

```

5.2.15 数据库连接池的工作机制是怎样的

数据库连接是一种非常珍贵且有限的资源，尤其在多用户的网络应用环境中更是如此。对数据库连接管理的好坏会直接影响整个系统的性能：一是建立与数据库的连接是一个耗时的操作，在页面应用中，如果每次用户的请求都需要建立新的数据库连接，那么响应时间就会很长，会严重影响用户的体验；二是数据库的连接个数是有限制的，如果管理不好，用户经常建立与数据库的连接却忘记释放，运行时间久了，数据库的连接资源就会耗尽，当再有新的用户需要访问数据库时，就需要等待很长一段时间，直到有用户释放连接资源才能访问数据，这对系统的可用性有着严重的影响。因此，管理好数据库的连接资源对应用系统尤其是页面应用系统是非常重要的。

数据库连接池负责分配、管理并释放数据库连接，它允许应用程序重复使用一个现有的数据库连接，而不再是重新建立一个新的数据库连接，同时，它还负责释放空闲时间超过最大空闲时间的数据库连接，避免因为没有释放数据库连接而引起的数据库连接遗漏。

在 J2EE 中，服务器在启动时会创建一定数量的池连接，并一直维持不少于此数目的连接池。当客户程序需要访问数据库时，就可以直接从池中获取与数据库的连接（获取一个空闲的连接），而不用去创建一个新的连接，同时将该连接标记为忙状态。当使用完毕后再把该连接标记为空闲状态，这样其他用户就可以使用这个连接了。如果当前没有空闲的连接，那么服务器就会根据配置参数在池中创建一定数量的连接。采用这种方法对数据库连接进行管理后可以大幅缩短用户的响应时间，提高运行效率。另一方面，为了提高数据库操作的性能，数据库连接池会释放空闲时间超过最大空闲时间的数据库连接来避免因为没有释放数据库连接而引起的数据库连接遗漏。

5.2.16 J2EE 开发有哪些调优的方法

当使用 J2EE 开发 Web 应用程序时，若只是考虑功能实现，一般都不存在什么问题，但由于 Web 应用的分布式特性，在很多特定的场合下，并发数据量会变得非常大，如何能够保证此时系统还能够高效地运转，而且能够快速响应用户的请求呢？

下面介绍一些常用的方法。

1) 优化设计，例如，小心使用继承。继承会导致递归，同时由于父类的构造器将会被遍历，继承嵌套得过深，会产生巨大的创建开销，因此推荐尽量使用组合方式来代替继承方式。除此之外，还有很多其他的方法：封装和重用常用的业务方法以及相关工具，避免在其他类中重复编写；简化类结构；面向接口编程；尽量使用主要类型；避免对象的过度使用，例如，不要为了使用一个方法就创建对象，应尽可能地使用静态方法或是使用将业务组件实例化后共享的方法等。

2) 尽可能使用数据库连接池。建立数据库的连接是一项必须但又非常耗时的工作，如果频繁地建立与数据库的连接将会导致系统性能的急剧下降。因此，在实际开发系统时，经常使用数据库连接池来提高系统性能。具体而言，就是使应用服务器维护着数据库连接池，这样就不需要在每个事务处理开始时就创建一个连接了。此外，由于 PreparedStatement 有缓存功能，这使其具有较高的效率，因此，在使用 JDBC 的方式访问数据库时，应尽可能使用 PreparedStatement。

3) 给 Web 容器配置合理的线程数量来处理客户端的 HTTP 请求。一般而言，最小的线程

数量设置为容器处理请求的平均数（平均负载），最大值设置为系统在高峰期处理的请求数，同时 Web 容器中线程的个数最好不要多于 Web 服务器中线程的个数。

4) 根据实际情况设置 Java 虚拟机中堆空间的大小。合理地设置堆空间的大小能够使得垃圾回收器运行的时间间隔被控制在一个合理的范围内，从而减少许多不必要的系统开销。具体设置堆空间大小的方法可参考专用资料。

5) 使用框架（例如 Hibernate）来提高系统的效率。

6) 把一些经常被访问的 Servlet 或 JSP 缓存起来，能够减少响应时间和提高系统的性能。但需要注意的是，缓存并非越多越好，无规则地使用缓存也可能会导致系统的崩溃。

7) 当在系统中使用 EJB 时，由于对 EJB 的调用是采用对象请求代理（Object Request Broker, ORB）的方式来完成，而 ORB 使用了线程来处理对 EJB 的请求，因此，应当合理配置线程池的大小以便它能够在平均负载和高峰期都能够很好地处理 EJB 的请求。

8) 优化 I/O 性能。I/O 使用不当将会造成资源竞争，降低系统性能。所以，在实际应用时，尽可能少使用 System.out 打印调试信息，推荐使用缓冲。

9) 优化查询。在模型设计时就应考虑冗余相对不会变化的数据，如果设计大数据量，最好能够考虑分区设计。

10) 对 session 进行合理管理与设置。根据实际情况对内存中可能存在的 session 的个数设置一个合适的值。注意：尽量减少 session 的大小，以降低其对内存的使用。而且，除非是在必须的情况下，否则尽量不要启用对 session 的持久化。同时，通过对 session 设置一个合理的超时时间来回收较长时间不使用的 session，也可以有效地提高系统的效率。

5.3 框架

5.3.1 什么是 Struts 框架

Struts 名字来源于在建筑与旧式飞机中使用的支持金属架，它是由自定义标签、信息资源（message resources）、Servlet 和 JSP 组成的一个可重用的 MVC 2 模式的框架。以 Struts 1.0 为例，它的结构图如图 5-11 所示。

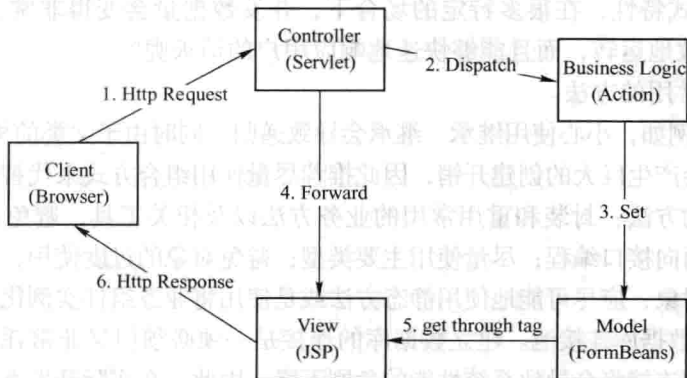


图 5-11 Struts 的结构图

从图 5-11 可以看出，Struts 的体系结构采用了 MVC 设计模式，同时包含客户端（Client）请求以及业务逻辑处理（Business Logic），而 MVC 设计模式主要由模型（Model）、视图（View）和控制器（Controller）三部分组成。

下面将分别对这些模块进行介绍。

(1) 客户端 (Client)

一方面可以通过浏览器发送 HTTP 请求, 另一方面可以把接收到的 HTTP 响应消息在浏览器上展现出来。

(2) 控制器 (Controller)

控制器主要包括 `ActionServlet` 类和 `RequestProcessor` 类。其中, `ActionServlet` 类是 MVC 实现的控制器部分, 是整个框架的核心部分, 它用来接收用户的请求, 并根据用户的请求从模型模块中获取用户所需的数据, 然后选择合适的视图来响应用户的请求。它采用了命令设计模式来实现这个功能: 通过 `struts-config.xml` 配置文件来确定处理请求的 `Action` 类。在处理用户请求时, 关于请求的处理大部分已交由 `RequestProcessor.process()` 方法来处理。`RequestProcessor` 类的 `process()` 方法采用了模板的设计模式 (按照处理的步骤与流程顺序的调用了一系列的方法)。处理的主要流程为:

- 1) `processPath (request, response)`。根据 URI (Uniform Resource Identifier, 统一资源标识符, 用来唯一的标识一个资源) 来得到 `ActionMapping` 元素的路径。

- 2) `processMapping (request, response)`。根据路径信息找到 `ActionMapping` 对象。

- 3) `processRoles (request, response, mapping)`。`Struts` 为 Web 应用提供了一种认证机制, 当用户登录时, 会通过 `processRoles` 方法调用 `request.isUserInRole()` 方法来检查这个用户是否有权限来执行给定的 `ActionMapping`。

- 4) `processValidate (request, response, form, mapping)`。调用 `ActionForm` 的 `validate()` 方法。

- 5) `processActionCreate (request, response, mapping)`。这个方法从 `<action>` 的 `type` 属性得到 `Action` 类名, 并创建返回它的实例。

- 6) `processActionPerform (req, res, action, form, mapping)`。这个方法调用 `Action` 类的 `execute()` 方法, 其中 `execute()` 方法中包含了业务逻辑的实现。需要注意的是, `Action` 类并不是线程安全的。

(3) 业务逻辑 (Business Logic)

`Servlet` 在接收到请求后会根据配置文件中的对应关系, 把请求转给指定的 `Action` 类来处理, `Action` 类采用适配器设计模式, 它只是对业务逻辑进行了包装 (真正的业务逻辑是由 EJB 的 `session bean` 或普通的 `Java` 类来实现)。

(4) 模型 (Model)

在 `Struts` 体系结构中, 模型分为两个部分: 系统的内部状态和可以改变状态的操作 (业务逻辑)。内部状态通常由一组 `ActionForm Bean` 表示, `ActionForm` 封装了 HTTP 请求的数据的类或对象。`ActionForm` 是一个抽象类, 每一个输入表单都对应着它的一个子类。配置文件 `struts-config.xml` 中保存了 HTTP 请求表单与具体 `ActionForm` 类的映射关系。

(5) 视图 (View)

视图就是一个 JSP 文件, 该 JSP 文件中没有业务逻辑的处理, 也不保存系统的状态信息, 它通过一些标签把数据以浏览器能识别的方式展现出来。目前, 标签库主要有 `Bean Tags`、`HTML tags`、`Logic Tags`、`Nested Tags` 以及 `Template Tags` 等。

`Struts` 框架作为一项开放源码项目, 优点众多, 具体而言, 主要有如下几点:

- 1) 由于采用了 MVC 模式, 因此它实现了表现与逻辑的分离, 使得系统有较好的可扩展性。同时 `Struts` 的标记库 (`Taglib`) 包含了大量的 `tag`, 有助于提高系统的开发效率。

- 2) 提供了页面导航功能, 使系统的脉络更加清晰。通过一个配置文件建立整个系统各部

分之间的联系，使有系统结构变得更加清晰，从而增强了系统的可扩展性与可维护性。

- 3) 提供了表单的验证功能，进一步增强了系统的健壮性。
- 4) 提供了数据库连接池管理。
- 5) 提供了 Exception 处理机制。
- 6) 支持国际化。

当然，Struts 也有它的不足之处，主要有以下几点：

- 1) Taglib 中包含了大量的 tag，对于初学者而言，开发难度比较大。
- 2) Struts 开发中包含了许多的 XML 格式的配置文件。一方面，这些配置文件不易调试；另一方面，大量的 XML 文件也不便于管理。
- 3) Struts 只能支持 Web 应用程序的开发。
- 4) Struts 的 Action 不是线程安全的，因此 Action 类用到的所有资源都必须进行同步。
- 5) 单元测试不方便。由于 Action 与 Web 层的紧耦合，导致其非常依赖于 Web 容器，给单元测试带来了不便。
- 6) 部署烦琐。当转到表示层时，需要配置 forward，例如，如果有 10 个表示层的 JSP 文件，则需要配置 10 个 Struts。此外，当目录、文件变更后，需要重新修改 forward，而且每次修改配置之后，还需要重新部署整个项目，对于 Tomcat 等服务器，还必须重启服务器。
- 7) 对 Servlet 的依赖性过强。Struts 处理 Action 时必须需要依赖 ServletRequest 和 ServletResponse，摆脱不了对 Servlet 容器的依赖。

5.3.2 Struts 框架响应客户请求的工作流程是什么

在 Struts 框架中，控制器主要是 ActionServlet，但是对业务逻辑的操作则主要由 Action、ActionMapping、ActionForward 等组件协调完成。其中，Action 扮演了真正的控制逻辑实现者的角色，而 ActionMapping 和 ActionForward 则指定了不同业务逻辑或流程的运行方向。

对于采用 Struts 框架的 Web 应用而言，在 Web 应用启动时，会加载并初始化 ActionServlet，ActionServlet 从 struts - config.xml 文件中读取配置信息，并把它们存放到 ActionMappings 对象中。具体而言，当 ActionServlet 接收到一个客户请求时，执行如下流程：

- 1) 检索和用户请求匹配的 ActionMapping 实例，如果不存在，就返回用户请求路径无效的信息。
- 2) 如果 ActionForm 实例不存在，就创建一个 ActionForm 对象，把客户提交的表单数据保存到 ActionForm 对象中。
- 3) 根据配置信息决定是否需要进行表单验证。如果需要验证，就调用 ActionForm 的 validate() 方法。
- 4) 如果 ActionForm 的 validate() 方法返回 null 或返回一个不包含 ActionMessage 的 ActionErrors 对象，就表示表单验证成功。
- 5) ActionServlet 根据 ActionMapping 实例包含的映射信息决定将请求转发给哪个 Action。如果对应的 Action 实例不存在，就先创建一个实例，然后调用 Action 的 execute() 方法。
- 6) Action 的 execute() 方法返回一个 ActionForward 对象，ActionServlet 再把客户请求转发给 ActionForward 对象指向的 JSP 组件。
- 7) ActionForward 对象指向的 JSP 组件生成动态页面，返回给客户。

对于以上流程中的步骤 4)，如果 ActionForm 的 validate() 方法返回一个（包含一个）或多个 ActionMessage 的 ActionErrors 对象，就表示表单验证失败，此时，ActionServlet 将直接把请

求转发给包含客户提交表单的 JSP 组件。在这种情况下，不会再创建 Action 对象并调用 Action 的 execute 方法了。

5.3.3 Struts 框架的数据验证可分为几种类型

数据验证也称为输入校验，用于指导对用户的输入进行基本的过滤，包括必填的字段（字段必须为数字）以及两次输入的密码必须一致等。Struts 框架提供了现成的、易于使用的的数据验证功能。

具体而言，数据验证可以分为两种类型：表单验证与业务逻辑验证。其中，表单验证由 ActionForm Bean 处理，例如，如果用户没有在表单中输入姓名就提交表单，将生成表单验证错误。该方式重写 ActionForm 的 validate() 方法，在该方法内对所有字段进行基本的校验。若出现不符合要求的输出，则将错误提示封装在 ActionError 对象里，最后将多个 ActionError 组合成 ActionErrors 对象，因此 ActionErrors 对象中封装了所有出错信息。

业务逻辑验证由 Action 处理，如果用户在表单中输入的姓名为“Hehao”，那么按照本应用的业务规则，不允许输入“Hehao”，因此将生成业务逻辑错误。需要注意的是，在 Action 里面完成数据验证，实际上就是在 execute() 方法前面增加数据验证的部分代码。

5.3.4 Form Bean 的表单验证流程是什么

Form Bean 的表单验证主要有以下 4 个步骤：

- 1) 用户提交 HTML 表单后，Struts 框架会自动把表单数据组装到 ActionForm Bean 中。
- 2) Struts 框架调用 ActionForm Bean 的 validate() 方法进行表单验证。
- 3) 如果 validate() 方法返回的 ActionErrors 对象为 null，或者不包含任何 ActionMessage 对象，就表示没有错误，数据验证通过。
- 4) 如果 ActionErrors 中包含 ActionMessage 对象，就表示发生了验证错误，Struts 框架会把 ActionErrors 对象保存到 request 范围内，然后把请求转发到恰当的视图组件中，视图组件通过 <html: errors> 标签把 request 范围内的 ActionErrors 对象中包含的错误信息显示出来，提示用户修改错误。

5.3.5 在 Struts 配置文件中，<action> 元素包含哪些属性和子元素

<action> 元素的属性见表 5-6。

表 5-6 <action> 元素的属性

属 性	描 述
attribute	设置和 Action 关联的 ActionForm Bean 在 request 和 session 范围内的 key
className	和 Action 元素对应的配置元素，默认为 org.apache.struts.action.ActionMapping
forward	定义了一个请求转发路径
include	指定包含的 URL 路径
path	指定请求访问 Action 的路径
parameter	指定 Action 的配置参数，在 Action 类的 execute() 方法中，可以调用 ActionMapping 对象的 getParameter() 方法来读取该配置参数
roles	指定允许调用 Action 的安全角色，多个角色之间用“,” 隔开。在处理请求时，RequestProcessor 会根据该配置项来决定用户是否有权限调用 Action
type	指定 Action 的完整类名，该类必须是扩展了 Struts 的 Action 类

属 性	描 述
name	指定需要传递给 Action 的 ActionForm Bean
scope	指定 ActionForm Bean 的存放范围, 其值为 Request 或 Session
unknown	设置为 true, 该操作将被作为所有没有定义的 ActionMapping 的 URL 的默认操作。当设置为 true 时, 表示可以处理用户发出的所有无效的 Action URL, 默认为 false
validate	指定是否执行表单验证, 默认为 true
input	指定当表单验证失败时的转发路径

5.3.6 ActionForm Bean 的作用有哪些

Action 一般用于控制业务逻辑的处理, 例如增加、删除、修改、查询等, ActionForm Bean 用于封装用户请求的参数。当接收到页面输入的数据后, 会首先保存在 ActionForm Bean 中, 然后在 Action 里面调用逻辑层的代码来处理这些数据。

ActionForm Bean 的作用有如下 3 点:

1) ActionForm Bean 本质上也是一种 JavaBean, 是专门用来传递表单数据的数据传递对象 (DATA Transfer Object, DTO)。除了具有一些 JavaBean 的常规方法外, ActionForm Bean 还包含一些特殊的方法, 例如用于验证 HTML 表单的数据以及将其属性重新设置为默认值。

2) Struts 框架利用 ActionForm Bean 来进行 View 组件和 Controller 组件之间表单数据的传递。

3) Struts 框架把 View 组件接收到的用户输入的表单数据保存在 ActionForm Bean 中, 然后把它传递给 Controller 组件, Controller 组件可以对 ActionForm Bean 中的数据进行修改, JSP 文件使用 Struts 标签读取修改后的 ActionForm Bean 的信息, 重新设置 HTML 表单。

5.3.7 ActionForm 的执行步骤有哪些

ActionForm 的执行步骤有以下几点:

1) 检查 Action 的映射, 确定 Action 中是否已经配置了对 ActionForm 的映射。

2) 根据 name 属性查找 Form Bean 的配置信息。

3) 检查 Action 的 Form Bean 的使用范围, 确定在此范围下是否已经有此 Form Bean 的实例。假如当前范围内已经存在此 Form Bean 的实例, 而且对当前请求来说, 是同一种类型的话, 那么就重用。否则, 就重新构建一个 Form Bean 的实例。

4) Form Bean 的 reset() 方法被调用。

5) 调用对应的 setter() 方法, 对状态属性赋值。

6) 根据 validate 的属性选择调用方法。如果 validate 的属性被设置为 true, 那么就调用 Form Bean 的 validate() 方法。如果 validate() 方法没有返回任何错误, 控制器将 ActionForm 作为参数传递给 Action 实例的 execute() 方法并执行。

5.3.8 forward 与 global - forward 有什么区别

forward 的主要作用是根据 Action 返回的值找到对应的 JSP 页面。global - forward 是全局的 forward, 当多个 Action 返回同一个值时, 例如在分页时或者得到数据列表时可将这个 forward 元素写在 global - forward 中, 这样就不用每次都在 Action 里面配置 forward 了。

以下示例给出了二者的使用区别:

forward:

```
<forward name = "login" path = "/login.jsp" redirect = "true" />
```

global - forward:

```
<global - forwards >
```

```
    <forward name = "login" path = "/login.jsp" redirect = "true" />
```

```
</global - forwards >
```

5.3.9 Struts 如何实现国际化

国际化 (internationalization, 简称 i18n, 其意为 internationalization 的首末字符 i 和 n 之间有 18 个字符) 与本地化 (localization, 简称 l10n, 其意为 localization 的首末字符 l 和 n 之间有 10 个字符) 是指让产品 (例如出版物、软件、硬件等) 能够适应非本地环境, 特别是其他语言与文化。具体而言, 就是要求程序在不修改内部代码的情况下, 能根据不同语言以及地区显示相应的界面。国际化资源文件是指用不同国家的语言描述相同的信息, 并放在各自对应的 .properties 属性文件中, 程序根据运行时的环境决定到底加载哪个文件。

Struts 提供了对国际化的支持。使用国际化功能也非常简单, 主要是先准备各语言的资源文件, 在资源文件中定义键和对应的字符串, 然后在显示的地方指定键就可以了, 例如, 在实现国际化时需要在 JSP 文件中加入 Struts 的 bean 标记库 `<%@ taglib uri = "/WEB-INF/struts-bean.tld" prefix = "bean" %>`, 如下例所示。

```
<%@ taglib uri = "/WEB-INF/struts-bean.tld" prefix = "bean" %>
```

```
<%@ taglib uri = "/WEB-INF/struts-html.tld" prefix = "html" %>
```

```
<%@ taglib uri = "/WEB-INF/struts-logic.tld" prefix = "logic" %>
```

```
<html:html locale = "true" >
```

```
<head >
```

```
    <title > welcome </title >
```

```
</head >
```

```
<body >
```

```
    <h2 > <bean:message key = "page.wel" /> </h2 >
```

```
</body >
```

```
</html:html >
```

接下来需要创建一个资源文件 ApplicationResource.properties, 文件中的内容如下:

```
page.wel = Welcome
```

如果想把英文显示转换为中文, 那么需要先创建一个临时的中文资源文件 ApplicationResource_temp.properties, 例如, page.wel = 欢迎, 接着对临时中文资源文件进行编码转换。可以使用 myeclipse 的插件, 也可以在命令行模式下执行如下命令:

```
native2ascii -encoding gb2312 ApplicationResource_temp.properties ApplicationResource_zh_CN.properties
```

最后, 把资源文件 ApplicationResource_zh_CN.properties 添加到相应的 struts-config.xml 配置文件中即可。

5.3.10 Struts 1 与 Struts 2 有哪些区别

Strut 1 框架由 ActionForm 和 JavaBean 组成, 其中 ActionForm 用于封装用户的请求参数, 封装成 ActionForm 对象, 该对象被 ActionServlet 转发给 Action, Action 根据 ActionForm 里面的请求参数处理用户的请求。Struts 2 框架的基础是核心控制器 FilterDispatcher, 它包含了框架内

部的控制流程和处理机制。业务控制器 Action 和业务逻辑组件是需要用户自己来实现的。开发人员在开发 Action 和业务逻辑组件的同时，还需要编写相关的配置文件，以供核心控制器 FilterDispatcher 来使用。

Struts 1 与 Struts 2 都是 MVC 的 Web 框架，尽管二者都叫 Struts，但也存在着很多不一致的地方。具体而言，主要表现在以下方面：

1) 风险控制方面。Struts 1 是老牌框架，应用广泛，有很好的群众基础，开发风险很小，成本更低。Struts 2 虽然基于 Struts 1，但是相对不成熟，而且未知的风险和变化很多，受众并不多。所以，使用 Struts 2 开发项目的风险更大，成本更高。

2) Action 实现类方面。Struts 1 要求 Action 类继承一个抽象基类，而 Struts 2 中 Action 类既可以实现一个 Action 接口，也可以实现其他接口，使可选和定制的服务成为可能。同时，Struts 2 提供了一个 ActionSupport 基类去实现常用的接口。

3) 线程模式方面。Struts 1 Action 是单例模式并且必须是线程安全的，因为仅有 Action 的一个实例来处理所有请求。而 Struts 2 Action 对象为每一个请求产生一个实例，因此不存在线程安全的问题。

4) Servlet 依赖方面。Struts 1 Action 依赖于 Servlet API，因为 Struts 1 Action 的 execute() 方法中有 HttpServletRequest 和 HttpServletResponse 方法。Struts 2 Action 不再依赖于 Servlet API，因为 Struts 2 的 Action 是由 POJO (Plain Old Java Objects, 简单的 Java 对象) 组成，在 Struts 2 中，Servlet 上下文以简单 Map 的形式表现出来，这使得 Action 可以进行独立的测试，当然，如果 Action 需要直接访问 HttpServletRequest 和 HttpServletResponse 参数，Struts 2 Action 仍然可以访问它们。但是，大部分时候，Action 都无需直接访问 HttpServletRequest 和 HttpServletResponse，从而给了开发人员更多选择。

5) 可测性方面。测试 Struts 1 Action 的一个主要问题是 execute() 方法依赖于 Servlet API，这使得 Action 的测试要依赖于 Web 容器。为了脱离 Web 容器测试 Struts 1 的 Action，必须借助于第三方扩展：Struts TestCase，该扩展包包含了系列的 Mock 对象（模拟了 HttpServletRequest 和 HttpServletResponse 对象），从而可以脱离 Web 容器测试 Struts 1 的 Action 类。Struts 2 Action 可以通过初始化、设置属性、调用方法来测试。

6) 封装请求参数。Struts 1 使用 ActionForm 对象封装用户的请求参数，所有 ActionForm 必须继承一个基类——ActionForm。普通的 JavaBean 不能用作 ActionForm，因此，开发人员必须创建大量的 ActionForm 类封装用户请求参数。虽然 Struts 1 提供了动态 ActionForm 来简化 ActionForm 的开发，但依然需要在配置文件中定义 ActionForm。而 Struts 2 直接使用 Action 属性来封装用户请求属性，避免了开发人员需要大量开发 ActionForm 类的烦琐。实际上，这些属性还可以是包含子属性的 Rich 对象类型。如果开发人员依然“怀念” Struts 1 ActionForm 的模式，Struts 2 提供了 ModelDriven 模式，可以让开发人员使用单独的 Model 对象来封装用户请求参数。但该 Model 对象无需继承任何 Struts 2 基类，是一个 POJO，从而降低了代码污染。

7) 表达式语言方面。Struts 1 整合了 JSP 标准标签库 (JSP Standard Tag Library, JSTL)，因此可以使用 JSTL 表达式语言。Struts 2 可以使用 JSTL，但它整合了一种更强大和灵活的表达式语言——对象图的符号语言 (Object Graph Notation Language, OGNL)，因此，Struts 2 下的表达式语言功能更加强大。

8) 绑定值到视图。Struts 1 使用标准 JSP 机制把对象绑定到视图页面，而 Struts 2 使用“ValueStack”技术，使标签库能够访问值，而不需要把对象和视图页面绑定在一起。

9) 类型转换。Struts 1 ActionForm 属性通常都是 String 类型。Struts 1 使用 Commons - Bea-

nutils 进行类型转换，每个类一个转换器，转换器是不可配置的；Struts 2 使用 OGNL 进行类型转换，支持基本数据类型和常用对象之间的转换。

10) 数据校验。Struts 1 与 Struts 2 都支持通过 `validate()` 方法的手动验证，不同的是，Struts 1 支持在 `ActionForm` 重写 `validate()` 方法中手动校验，或者通过整合 Commons Validator 框架来完成数据校验，而 Struts 2 支持通过重写 `validate()` 方法进行校验，也支持整合 XWork 校验框架进行校验。

11) Action 执行控制。Struts 1 支持每一个模块对应一个请求处理（即生命周期的概念），但是模块中的所有 Action 必须共享相同的生命周期。Struts 2 支持通过拦截器堆栈（Interceptor Stacks）为每一个 Action 创建不同的生命周期。开发人员可以根据需要创建相应堆栈，以便和不同的 Action 一起使用。

12) 捕获输入。Struts 1 使用 `ActionForm` 对象捕获输入，所有 `ActionForm` 必须继承自一个框架依赖的基类。因为其他 `JavaBean` 不能用作 `ActionForm`，开发人员经常创建多余的类捕获输入。而 Struts 2 直接使用 Action 属性作为输入属性，消除了对第二个输入对象的需求。

5.3.11 什么是 IoC

控制反转（Inverse of Control, IoC）有时也被称为依赖注入，是一种降低对象之间耦合关系的设计思想。一般而言，在分层体系结构中，都是上层调用下层的接口，上层依赖于下层的执行，即调用者依赖于被调用者。而通过 IoC 方式，使得上层不再依赖于下层的接口，即通过采用一定的机制来选择不同的下层实现，完成控制反转，使得由调用者来决定被调用者。IoC 通过注入一个实例化的对象来达到解耦的目的。使用这种方法后，对象不会被显式地调用，而是根据需求通过 IoC 容器（例如 Spring）来提供。

采用 IoC 机制能够提高系统的可扩展性，如果对象之间通过显式调用进行交互会导致调用者与调用者存在着非常紧密的联系，其中一方的改动将会导致程序出现很大的改动，例如，要为一家卖茶的商店提供一套管理系统，在这家商店刚开业时只卖绿茶（GreenTea），随着规模的扩大或者根据具体销售量，未来可能会随时改变茶的类型，例如红茶（BlackTea）等，传统的实现方法会针对茶抽象化一个基类，绿茶类只需要继承自该基类即可，如图 5-12 所示。

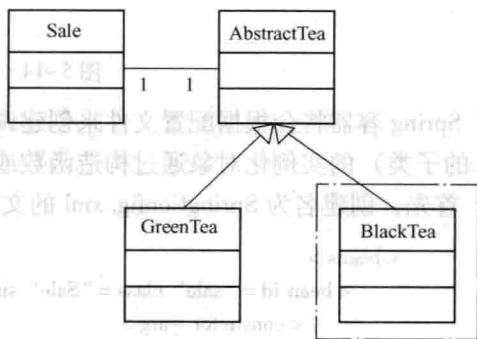


图 5-12 卖茶系统类图（一）

采用该实现方法后，在需要使用 `GreenTea` 时只需要执行以下代码即可：`AbstractTea t = new GreenTea()`，当然，这种方法是满足当前设计要求的。但是该方法的可扩展性不好，存在着不恰当的地方，例如，商家发现绿茶的销售并不好，决定开始销售红茶（`BlackTea`）时，那么只需要实现一个 `BlackTea` 类，并且让这个类继承自 `AbstractTea` 即可。但是，系统中所有用到 `AbstractTea t = new GreenTea()` 的地方都需要被改为 `AbstractTea t = new BlackTea()`，而这种创建对象实例的方法往往会导致程序的改动量非常大。

那么怎样才能增强系统的可扩展性呢？——设计模式，此时可以使用设计模式中的工厂模式来把创建对象的行为包装起来，实现方法如图 5-13 所示。

通过以上方法，可以把创建对象的过程委托给 `TeaFactory` 来完成，在需要使用 `Tea` 对象时只需要调用 `Factory` 类的 `getTea` 方法即可，具体创建对象的逻辑在 `TeaFactory` 中来实现，那么当商家需要把绿茶替换为红茶时，系统中只需要改动 `TeaFactory` 中创建对象的逻辑即可，采用

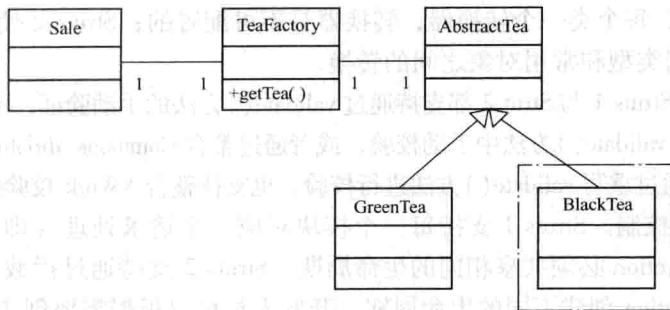


图 5-13 卖茶系统类图（二）

了工厂模式后，只需要在一个地方做改动就可以满足要求，这样就增强了系统的可扩展性。

虽然说采用工厂设计模式后增强了系统的可扩展性，但是从本质上来讲，工厂模式只不过是把程序中会变动的逻辑移动到工厂类里面了，当系统中的类较多时，在系统扩展时需要经常改动工厂类中的代码。而采用 IoC 设计思想后，程序将会有更好的可扩展性，下面主要介绍 Spring 框架在采用 IoC 后的实现方法，如图 5-14 所示。

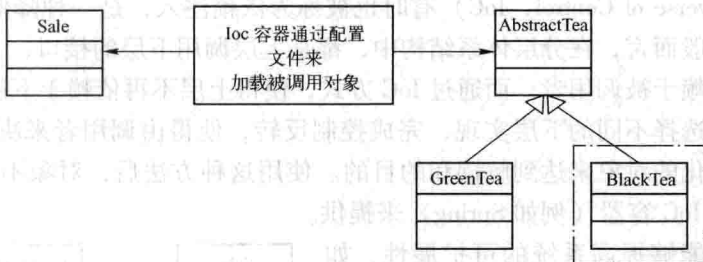


图 5-14 卖茶系统类图（三）

Spring 容器将会根据配置文件来创建调用者对象（Sale），同时把被调用的对象（AbstractTea 的子类）的实例化对象通过构造函数或 set() 方法的形式注入到调用者对象中。

首先，创建名为 SpringConfig.xml 的文件。

```

<beans>
  <bean id="sale" class="Sale" singleton="false">
    <constructor-arg>
      <ref bean="tea"/>
    </constructor-arg>
  </bean>
  <bean id="tea" class="BlueTea" singleton="false">
  </beans>
  
```

在实现 Sale 类时，需要按照如下方式实现。

```

class Sale {
  private AbstractTea t;
  public Sale(AbstractTea t) {
    this.t = t;
  }
  //其他方法就可以使用 t 了
}
  
```

当 Spring 容器创建 Sale 对象时，根据配置文件 SpringConfig.xml 就会创建一个 BlueTea 的

对象，作为 Sale 构造函数的参数。当需要把 BlueTea 改为 BlackTea 时，只需要修改上述配置文件，而不需要修改代码。

在需要 Sale 时，可以通过如下方式来创建 Sale 对象：

```
ApplicationContext ctx = new FileSystemXmlApplicationContext("SpringConfig.xml");
Sale s = (Sale)ctx.getBean("sale");
```

上例中，Spring 采用 IoC 的方式来实现把实例化的对象注入到开发人员自定义的对象中，具有较强的可扩展性。

具体而言，IoC 主要有以下两个方面的优点：

1) 通过 IoC 容器，开发人员不需要关注对象如何被创建的，同时增加新类也非常方便，只需要修改配置文件即可实现对象的“热插拔”。

2) IoC 容器可以通过配置文件来确定需要注入的实例化对象，因此非常便于进行单元测试。

尽管如此，IoC 也有自身的缺点，具体表现为以下两点：

1) 对象是通过反射机制实例化出来的，因此会对系统的性能有一定的影响。

2) 创建对象的流程变得比较复杂。

5.3.12 什么是 AOP

面向切面编程 (Aspect - Oriented Programming, AOP) 是对面向对象开发的一种补充，它允许开发人员在不改变原来模型的基础上动态地修改模型以满足新的需求，例如，开发人员可以在不改变原来业务逻辑模型的基础上可以动态地增加日志、安全或异常处理的功能。

下面介绍一个在 Spring 中使用 AOP 编程的简单例子。

1) 创建一个接口以及实现这个接口的类。TestAOPIn.java 文件的内容如下所示。

```
public interface TestAOPIn {
    public void doSomething();
}
```

TestAOPImpl.java 文件的内容如下所示。

```
public class TestAOPImpl implements TestAOPIn {
    public void doSomething() {
        System.out.println("TestAOPImpl:doSomething");
    }
}
```

2) 配置 SpringConfig.xml，使得这个类的实例化对象可以被注入到使用这个对象的 Test 类中。

```
<?xml version = "1.0" encoding = "UTF - 8" ? >
<!DOCTYPE beans PUBLIC " -//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/
spring - beans. dtd" >

<beans >
    <bean id = "testAOPBean" class = "org.springframework.aop.framework.ProxyFactoryBean" >
        <property name = "target" >
            <bean class = "testAOPIn" singleton = "false" / >
        </property >
    </bean >
</beans >
```

3) 在完成配置文件后, 编写如下测试代码。

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
public class Test {
    public static void main(String[] args) {
        ApplicationContext ctx = new FileSystemXmlApplicationContext("SpringConfig.xml");
        TestAOPIn t = (TestAOPIn) ctx.getBean("testAOPBean");
        t.doSomething();
    }
}
```

程序运行结果为:

```
TestAOPImpl:doSomething
```

编写完这个模块后, 开发人员需要增加对 doSomething() 方法调用的跟踪, 也就是说, 要跟踪该方法何时被调用以及调用何时结束等内容。当然, 使用传统的方法也可以实现该功能, 但却会产生额外的开销, 即需要修改已存在的模块。此时可以采用 AOP 的方式来实现这个功能。它在不修改原有模块的前提下可以完成相同的功能。

```
public class TestAOPImpl implements TestAOPIn {
    public void doSomething() {
        System.out.println("beginCall doSomething");
        System.out.println("TestAOPImpl:doSomething");
        System.out.println("endCall doSomething");
    }
}
```

其实现原理如图 5-15 所示。

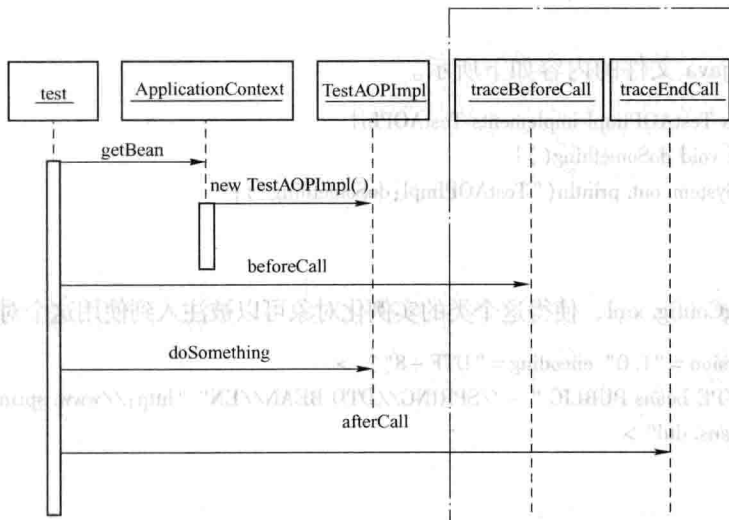


图 5-15 AOP 实现原理图

为此需要提供用来跟踪函数调用的类。traceBeforeCall.java 文件的内容如下所示:

```
public class traceBeforeCall implements MethodBeforeAdvice {
    public void beforeCall (Method arg0, Object[] arg1, Object arg2) throws Throwable {
```

```
System.out.println("beginCall doSomething");
```

traceEndCall.java 文件的内容如下所示：

```
import java.lang.reflect.Method;
import org.springframework.aop.AfterReturningAdvice;
public class traceEndCall implements AfterReturningAdvice {
    public void afterCall(Object arg0, Method arg1, Object[] arg2, Object arg3) throws Throwable {
        System.out.println("endCall doSomething");
    }
}
```

只需在配置文件中配置在调用 doSomething() 方法之前需要调用 traceBeforeCall 类的 beforeCall() 方法以及在调用 doSomething() 方法之后需要调用 traceEndCall 类的 afterCall() 方法，Spring 容器就会根据配置文件在调用 doSomething() 方法前后自动调用相应的方法，通过在 beforeCall() 方法和 afterCall() 方法中添加跟踪的代码就可以满足对 doSomething() 方法调用的跟踪要求，同时还不需要更改原来已实现的代码模块。

5.3.13 什么是 Spring 框架

Spring 是一个 J2EE 的框架，这个框架提供了对轻量级 IoC 的良好支持，同时也提供了对 AOP 技术非常好的封装。相比其他框架，Spring 框架的设计更加模块化，框架内的每个模块都能完成特定的工作，而且各个模块可以独立地运行，不会相互牵制。因此，在使用 Spring 框架时，开发人员可以使用整个框架，也可以只使用框架内的一部分模块，例如可以只使用 Spring AOP 模块来实现日志管理功能，而不需要使用其他模块。

Spring 框架主要由 7 个模块组成，它们分别是 Spring AOP、Spring ORM、Spring DAO、Spring Web、Spring Context、Spring Web MVC、Spring Core 等。Spring 框架图如图 5-16 所示。

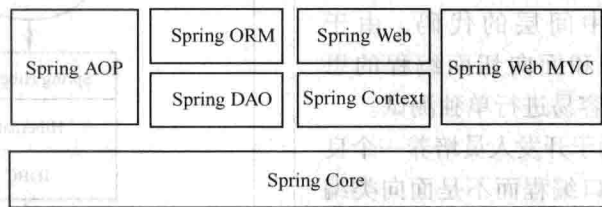


图 5-16 Spring 框架图

表 5-7 详细介绍了各个模块的作用。

表 5-7 Spring 框架的 7 个模块的作用

模 块	描 述
Spring AOP	采用了面向切面编程的思想，使 Spring 框架管理的对象支持 AOP，同时这个模块也提供了事务管理，可以不依赖具体的 EJB 组件，就可以将事务管理集成到应用程序中
Spring ORM	提供了对现有 ORM 框架的支持，例如 Hibernate、JDO 等
Spring DAO	提供了对数据访问对象 (Data Access Object, DAO) 模式和 JDBC 的支持。DAO 可以实现把业务逻辑与数据库访问的代码实现分离，从而降低代码的耦合度。通过对 JDBC 的抽象，简化了开发工作，同时简化了对异常的处理 (可以很好地处理不同数据库厂商抛出的异常)

模块	描述
Spring Web	提供了 Servlet 监听器的 Context 和 Web 应用的上下文。同时还集成了一些现有的 Web 框架，例如 Struts
Spring Context	扩展核心容器，提供了 Spring 上下文环境，给开发人员提供了很多非常有用的服务，例如国际化、E-mail 和 JNDI 访问等
Spring Web MVC	提供了一个构建 Web 应用程序的 MVC 的实现
Spring Core	Spring 框架的核心容器，它提供了 Spring 框架的基本功能。这个模块中最主要的一个组件为 BeanFactory，它使用工厂模式来创建所需的对象。同时 BeanFactory 使用 IOC 思想，通过读取 XML 文件的方式来实例化对象，可以说 BeanFactory 提供了组件生命周期的管理，组件的创建、装配、销毁等功能

Spring 在 J2EE 中到底扮演着怎样的角色？在哪些地方可以使用 Spring？

Spring 的工作原理如图 5-17 所示，可以看出，Spring 有着非常广泛的用途，不仅可以在 Web 容器中用来管理 Web 服务器端的模块，例如 Servlet，还可以用来管理用于访问数据库的 Hibernate。由于 Spring 在管理 Business Object（业务对象）和 DAO 时使用了 IoC 和 AOP 的思想，因此这些被 Spring 管理的对象都可以脱离 EJB 容器单独运行和测试。在需要被 Spring 容器管理时，只需要增加配置文件，Spring 框架就会根据配置文件与相应的机制实现对这些对象的管理。

除此之外，使用 Spring 还有如下好处：

1) 在使用 J2EE 开发多层应用程序时，Spring 有效地管理了中间层的代码，由于 Spring 采用了控制反转和面向切面编程的思想，因此这些代码非常容易进行单独测试。

2) 使用 Spring 有助于开发人员培养一个好的编程习惯：面向接口编程而不是面向类编程。面向接口编程使得程序有更好的可扩展性。

3) Spring 对数据的存取提供了一个一致的框架（不论是使用 JDBC 还是 O/R 映射的框架，例如 Hibernate 或 JDO）。

4) Spring 通过支持不同的事务处理 API（如 JTA、JDBC、Hibernate 等）的方法对事务的管理提供了一致的抽象方法。

5) 使用 Spring 框架编写的大部分业务对象不需要依赖 Spring。

5.3.14 什么是 Hibernate

Hibernate 是一个开放源代码的对象关系映射（Object Relation Mapping, ORM，一种用来完成对象模型到关系模型的映射技术）框架，它不仅可以在 J2EE 容器中，也可以在 J2EE

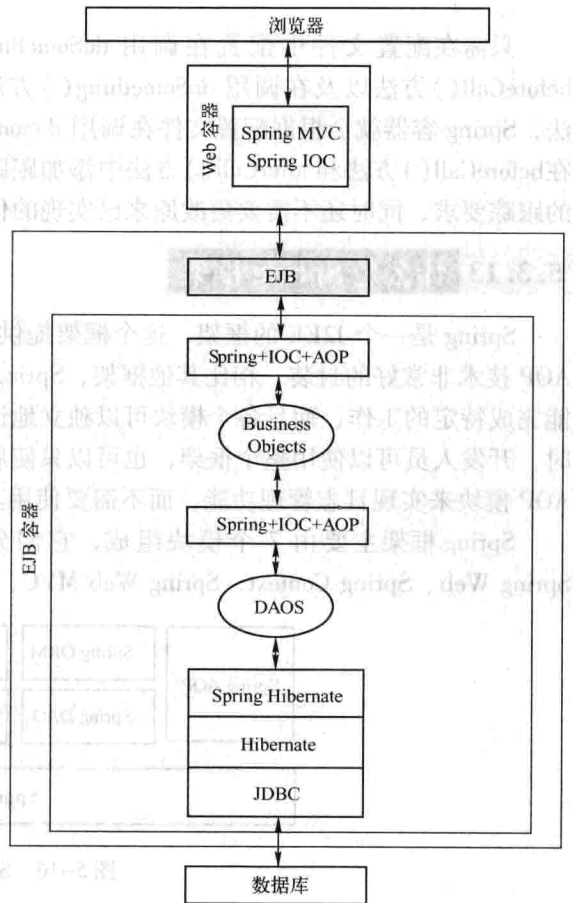


图 5-17 Spring 的工作原理

容器外运行。它对 JDBC 进行了非常轻量级的对象封装，所以任何可以使用 JDBC 的地方都可以用 Hibernate 来替代。Hibernate 实现了 Java 对象与关系数据库记录的映射关系，简化了开发人员访问数据库的流程，极大地提高了软件的开发效率。

Hibernate 主要提供了 5 个核心接口，分别为 Session、SessionFactory、Transaction、Query 和 Configuration。通过使用这些接口不仅可以完成对数据库的访问（例如查询、插入、更新与删除等），而且还可以实现对事务的控制。Hibernate 的结构图如图 5-18 所示。

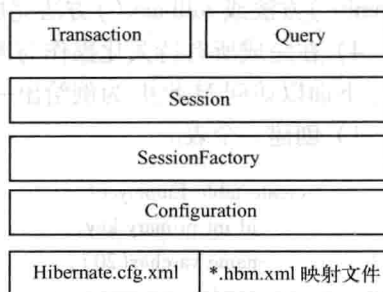


图 5-18 Hibernate 结构图

表 5-8 详细介绍了各个模块的作用。

表 5-8 Hibernate 模块介绍

接口名	描述
Session	一个轻量级的非线性安全的对象，主要负责被持久化对象与数据库的操作。可以使用 SessionFactory 来创建一个 Session，当对数据库的所有操作都执行完成后，就可以关闭 Session。Session 在访问数据库时会建立与数据库的连接，这个连接只有在需要时才会被建立
SessionFactory	负责初始化 Hibernate。它可以被看作数据源的代理，可以用来创建 Session 对象。此外，SessionFactory 是线程安全的，因此可以被多个线程同时访问。一般而言，SessionFactory 会在 Hibernate 启动时创建一次，因此，为了便于使用，SessionFactory 应该用一个单例模式来实现
Transaction	负责事务相关的操作。它的主要方法有 commit() 和 rollback()，其中 commit() 方法负责事务的提交，rollback() 方法负责事务的回滚，可以通过 Session 的 beginTransaction() 方法来创建
Query	负责执行各种数据库查询。可以使用 Hibernate 查询语言（Hibernate Query Language, HQL）或 SQL 语句两种方式进行查询（这两种查询方式非常类似，与 SQL 不同的是，HQL 语言使用类和属性而不是表与列名进行查询）。可以通过 Session 的 createQuery() 方法来创建 Query。此外，Hibernate 还提供了另外一种查询方式 QBC（Query By Criteria），其使用方法为：先使用 Session 实例的 createCriteria() 方法创建 Criteria 对象，接着使用工具类 Restrictions 的方法为 Criteria 对象设置查询条件，同时还可以用 Order 工具类的方法设置排序方式，最后用 Projections 工具类的方法进行统计和分组，使用 Criteria 对象的 list() 方法进行查询并返回结果。需要注意的是，QBC 是一种类型安全的面向对象的查询方式
Configuration	用于读取 Hibernate 配置文件，并生成 SessionFactory 对象。其中配置文件主要有两类：一类是 hibernate.cfg.xml 或 hibernate.properties；另一类是映射文件，例如 *.hbm.xml。其中 hibernate.cfg.xml 或 hibernate.properties 用来配置 Hibernate 服务的信息（例如连接数据库使用的驱动类、数据库连接的 URL、数据库的用户名和密码等信息）。如果同时提供了 hibernate.cfg.xml 和 hibernate.properties 文件，那么 hibernate.cfg.xml 会覆盖 hibernate.properties 中的配置信息。映射文件（*.hbm.xml）用来配置 java 对象与关系数据库记录的映射关系。为了便于管理与维护，通常会给每个对象创建一个单独的映射文件

Hibernate 的使用过程如下：

1) 应用程序通过 Configuration 类读取配置文件并创建 SessionFactory 对象。

```
SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
```

2) 通过 SessionFactory 生成一个 Session 对象。

```
Session session = sessionFactory.openSession();
```

3) 通过 Session 对象的 beginTransaction() 方法创建一个事务。

Transaction t = session.beginTransaction(); 接着可以通过 Session 对象的 get()、load()、save()、update()、delete() 和 saveOrUpdate() 等方法实现数据的加载、保存、更新和删除等

操作；也可以通过 session 生成一个 Query 对象，然后利用 Query 对象执行查询操作；最后通过 commit() 方法或 rollback() 方法完成事务操作。

4) 在完成所有持久化操作与事务操作后需要关闭 Session 与 SessionFactory。

下面以访问 MySQL 为例给出一个使用 Hibernate 的示例。

1) 创建一个表。

```
create table Employee(
    id int primary key,
    name varchar(20),
    age int
);
```

2) 在 Eclipse 中创建一个工程，接着导入相关的类库，例如 Hibernate 需要用到的类 (hibernate3.jar)，访问数据库的驱动包 (mysql - connector - java - 5.0.8 - bin.jar) 和操作 XML 文件用到的 jar 包等。接着创建一个持久化类。

```
public class Employee {
    private Integer id;
    private String name;
    private Integer age;
    public Employee() {}
    public Employee(Integer id, String name, Integer age) {
        this.id = id;
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Integer getAge() {
        return age;
    }
    public void setAge(Integer age) {
        this.age = age;
    }
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
}
```

3) 创建一个目录 hibernate，在这个目录下创建两个配置文件：hibernate.cfg.xml 与 employee.hdm.xml。其中，hibernate.cfg.xml 文件的内容如下所示：

```

<?xml version = '1.0' encoding = 'UTF - 8' ? >
<!DOCTYPE hibernate - configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate - configuration - 3.0.dtd" >
<!-- Generated by MyEclipse Hibernate Tools. -->
<hibernate - configuration >
<session - factory >
    <property name = "connection.username" > user1 </property >
    <property name = "connection.url" >
        jdbc:mysql://localhost:3306/Test
    </property >
    <property name = "dialect" >
        org.hibernate.dialect.MySQLDialect
    </property >
    <property name = "myeclipse.connection.profile" > MySQL5.1 </property >
    <property name = "connection.password" > pwd1 </property >
    <property name = "connection.driver_class" >
        com.mysql.jdbc.Driver
    </property >
    <property name = "show_sql" > true </property >
    <mapping resource = "hibernate/employee.hdm.xml" / >
</session - factory >
</hibernate - configuration >

```

配置文件 employee.hdm.xml 中存放了访问数据库的基本信息，具体内容如下所示：

```

<?xml version = "1.0" encoding = "utf - 8" ? >
<!DOCTYPE hibernate - mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate - mapping - 3.0.dtd" >
<!--
Mapping file autogenerated by MyEclipse - Hibernate Tools
-->
<hibernate - mapping >
    <class name = "Employee" table = "Employee" >
        <id name = "id" type = "java.lang.Integer" >
            <column name = "id" / >
            <!-- Hibernate 可以实现自增 -->
            <generator class = "increment" / >
        </id >
        <property name = "name" type = "java.lang.String" >
            <column name = "name" length = "20" / >
        </property >
        <property name = "age" type = "java.lang.Integer" >
            <column name = "age" / >
        </property >
    </class >
</hibernate - mapping >

```

以上文件中存放了 Java 类与数据库表的对应关系。

4) 编写访问数据库的代码。

```

import java.sql.SQLException;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
public class Test {
    public static void main(String[] args) throws ClassNotFoundException, SQLException {
        Configuration config = new Configuration().configure("/hibernate/hibernate.cfg.xml");
        SessionFactory sessionFactory = config.buildSessionFactory();
        Session s = sessionFactory.openSession();
        Transaction tx = s.beginTransaction();
        Employee e = new Employee();
        e.setName("James");
        e.setAge(25);
        try {
            s.save(e);           //保存持久类对象
            tx.commit();        //提交到数据库
            //从数据库中查找刚才保存的数据
            Query q = s.createQuery("select * from Employee").addEntity(Employee.class);
            List<Employee> rs = q.list();
            for(int i=0;i<rs.size();i++){
                Employee e1 = rs.get(i);
                System.out.println("id:" + e1.getId() + " name:" + e1.getName() + " age:" +
                    e1.getAge());
            }
            s.close();
        } catch (Exception e1) {
            e1.printStackTrace();
            tx.rollback();
        }
    }
}

```

程序运行结果为：

```

Hibernate: select max(id) from Employee
Hibernate: insert into Employee (name,age,id) values (?,?,?)
Hibernate: select * from Employee
id:1 name:James age:25

```

需要注意的是，输出结果的前三行是由 Hibernate 框架打印出的信息，最后一行数据为从数据库中查找出的数据。

具体而言，使用 Hibernate 框架有诸多好处，主要表现为以下几个方面。

- 1) 提高开发效率。
- 2) 使得开发可以完全采用面向对象的思想，不需要关心数据库的关系模型。

3) 使用 Hibernate 开发的系统有很好的可移植性, 可以很容易地实现不同数据库之间的移植而不需要关系不同数据库 SQL 语句的差异。

4) 支持透明持久化, Hibernate 的 API 没有侵入性, 当保存一个对象时, 这个对象不需要继承 Hibernate 中的任何类和实现任何接口。

虽然如此, 但 Hibernate 只适用于针对单一对象简单的增、删、查、改, 而对于批量的修改/删除的场合, 则不适用, 这也是 OR 框架的弱点, 所以, 当要使用数据库的特定优化机制时, 不适合使用 Hibernate。

引申:

1. 在使用 Hibernate 时如何提高性能?

使用 Hibernate 时, 有多种方法可以用来提高性能, 具体内容如下所示。

1) 延迟加载。当 Hibernate 从数据库获取某一个对象数据、获取某一个对象的集合属性值时或者获取某一个对象所关联的另一个对象时, 并不会立即从数据库中把数据加载到对象中, 而是通过建立一个代理对象, 把这个对象的属性都设置为默认值, 只有当这些数据在被使用时才会从数据库中去加载对应的数据, 使用这种方法有助于提高 Hibernate 的性能。

2) 缓存技术。Hibernate 中提供了一级缓存与二级缓存, 合理的利用缓存也有助于提高系统的性能, 为了避免不合理的利用缓存导致内存的过度消耗降低系统的性能, 可以通过合理配置缓存的参数 (例如配置缓存可以加载数据的个数) 来避免这个问题。

3) 优化查询语句。通过优化查询语句来提高系统的性能。

2. Hibernate 中怎样实现类之间的关系? (例如一对多、多对多关系)

类之间的关系主要体现在表之间的关系, 它们都是对对象进行操作, 程序中把所有表与类都映射在一起, 它们通过配置文件中的 many-to-one、one-to-many、many-to-many 来进行配置。

5.3.15 什么是 Hibernate 的二级缓存

缓存的目的是为了通过减少应用程序对物理数据源访问的次数来提高程序运行的效率, 原理则是把当前或接下来一段时间可能会用到的数据保存到内存中, 在使用时直接从内存中读取, 而不是从硬盘中去读取, 简单来说, 缓存就是数据库中数据在内存中的“临时容器”。

在 Hibernate 中, 缓存用来把从数据库中查询出来的和使用过的对象保存在内存中, 以便在后期需要用到这个对象时可以直接从缓存中来获取这个对象 (只有当该对象在缓存中不存在时才会去数据库中查询)。显然, 由于避免了因大量发送 SQL 语句到数据库查询导致的性能损耗, 缓存机制可以显著提高程序的运行效率。

在 Hibernate 中有一级缓存与二级缓存的概念, 一级缓存由 Session 来管理, 二级缓存由 SessionFactory 来管理。在使用时, 二级缓存是可有可无的, 但一级缓存是必不可少的。

一级缓存使用的场合如下: 当使用 Session 查询数据时, 首先会在该 Session 内部查找该对象是否存在, 若存在, 则直接返回, 否则, 就到数据库中去查询, 并将查询的结果缓存起来以便后期使用。一级缓存的缺点就是当使用 Session 来表示一次会话时, 它的生命周期较短, 而且它是线程不安全的, 不能被多个线程共享, 因此, 在实际使用时, 对效率的提升不是非常明显。

鉴于以上原因, 二级缓存的概念被引入了。二级缓存用来为 Hibernate 配置一种全局的缓存, 以便实现多个线程与事务共享。在使用了二级缓存机制后, 当查询数据时, 会首先在内部缓存中去查找, 如果不存在, 接着在二级缓存中查找, 最后才去数据库中查找。与一级缓存相比, 二级缓存是独立于 Hibernate 的软件部件, 属于第三方的产品, 常见的产品有 EhCache、

OSCache 和 JbossCache 等，Hibernate 3 以后默认使用的产品为 EhCache。在使用时，可以根据需求通过配置二级缓存插件来实现二级缓存功能，Hibernate 为了集成这些插件，提供了 org.hibernate.cache.CacheProvider 接口来充当缓存插件与 Hibernate 之间的适配器。当然，二级缓存除了以内存作为存储介质外，还可以选用硬盘等外部存储设备。

合理地使用 Hibernate 的二级缓存机制有助于提高系统的运行效率，但如果使用得不合理，不仅不会提高效率，反而有可能会降低系统的性能。

二级缓存一般适用于以下几种情况：

1) 数据量较小。如果数据量太大，缓存太多，会消耗大量内存，造成内存资源短缺，从而降低系统的性能。

2) 对数据的修改较少。如果进行大量的修改，就需要频繁地对缓存中数据与数据库中的数据进行同步，而这也会影响系统的性能。

3) 不会被大量的应用共享的数据。如果数据被大量线程或事务共享，多线程访问时的同步机制也会影响系统的性能。

4) 不是很重要的数据。如果查询的数据非常重要（例如财务数据），对数据的正确性要求非常高，最好不要使用二级缓存。

5.3.16 Hibernate 中 session 的 update() 和 saveOrUpdate()、load() 和 get() 有什么区别

Hibernate 的对象有3种状态，分别为：瞬时态 (Transient)、持久态 (Persistent) 和脱管态 (Detached)。处于持久态的对象也被称为 PO (Persistence Object)，瞬时对象和脱管对象也被称为 VO (Value Object)。

saveOrUpdate() 方法同时包含了 save() 和 update() 方法的功能。Hibernate 会根据对象的状态来确定是调用 save() 方法还是调用 update() 方法：若对象是持久化对象，则不进行任何操作，直接返回；若传入的对象与 session 中的另一个对象有相同的标识符，则抛出一个异常；若对象的标识符属性（用来唯一确定一个对象）在数据库中不存在或者是一个临时值，则调用 save() 方法把它保存到数据库中，否则，调用 update() 方法更新对象的值到数据库中。鉴于此，在使用时，若能确定对象的状态，则最好不要调用 saveOrUpdate() 方法，这样有助于提高效率，例如，如果能够确定这个对象所对应的值在数据库中肯定不存在，那么就可以直接调用 save() 方法。

get() 方法与 load() 方法都是用来通过从数据库中加载所需的数据来创建一个持久化的对象，它们主要有以下几个不同点：

1) 如果数据库中不存在该对象，load() 方法会抛出一个 ObjectNotFoundException 异常，而 get() 方法则会返回 null。

2) get() 方法首先查询 Session 内部缓存，若不存在，则接着查询二级缓存，最后查询数据库；而 load() 方法在创建时会首先查询 Session 内部缓存，如果不存在，就创建代理对象，实际使用数据时才查询二级缓存和数据库，因此 load() 方法支持延迟加载（对象中的属性在使用时才会加载，而不是在创建对象时就加载所有属性）。

3) get() 方法永远只返回实体类，而 load() 方法可以返回实体类的代理类实例。

4) get() 方法和 find() 方法都是直接从数据库中检索，而 load() 方法的执行则比较复杂：首先查找 Session 的 persistent Context 中是否有缓存，若有，则直接返回；若没有，则判断是否是 lazy。如果不是，直接访问数据库检索，查到记录返回，查不到抛出异常；若是 lazy，则需要建立代理对象，对象的 initialized 属性为 false，target 属性为 null。在访问获得的代理对象的

属性时检索数据库，若找到记录，则把该记录的对象复制到代理对象的 target 上，并将 initialized 置为 true；若找不到，就抛出异常。

5.3.17 Hibernate 有哪些主键生成策略

Hibernate 作为一种优秀的持久层框架，采用 ORM 方式，大大地简化了对数据库的操作。同时，Hibernate 框架提供的主键生成策略，使开发人员可以通过在实体类的映射文件中设定关键字来告诉 Hibernate 要使用的主键生成方式，然后 Hibernate 会根据设定完成数据库的主键控制。Hibernate 中的主键生成策略主要有如下几种：

1) Assigned。使用该方法时，主键不是由 Hibernate 生成的，而是由外部程序负责生成，所以无需 Hibernate 参与，但需要开发人员在调用 save() 方法之前来指定，否则调用 save() 方法会抛出异常。其缺点是在执行新增操作时需查询数据库判断生成的主键是否已经存在，否则很容易产生主键冲突。

2) Hilo。该方法使用一个高/低位算法 (High/Low Algorithm) 生成 long、short 或 int 类型的标识符。给定一个表和字段作为高位值的来源 (默认的表是 hibernate_unique_key，默认的字段是 next_hi)。它将 id 的产生源分成两部分：DB + 内存，然后，按照算法结合在一起产生 id 值，从而可以在很少的连接次数内产生多条记录，提高效率。

需要注意的是，该方法需要额外的数据库表保存主键生成历史状态。Hilo 能保证同一个数据库中主键的唯一性，但不能保证多个数据库之间主键的唯一性。

3) Seqhilo。与 Hilo 类似，Seqhilo 是一种通过高/低位算法实现的主键生成机制，只是主键历史状态保存在 Sequence 中，适用于支持 Sequence 的数据库，例如 Oracle。

4) Increment。这种方式采用对主键自增的方式来生成新的主键。实现机制为：在当前应用实例中维持一个变量，以保存当前的最大值，之后每当需要生成主键时，便会将此值加 1 作为主键。该方式要求数据库支持 Sequence。

尽管该方式优点众多，但问题也不少。首先，新增数据前需要先查询一遍，这会影响系统的性能；其次，主键的类型只能是数值的 int 或 long 型；最后，会产生并发问题，即如果当前有多个实例访问同一个数据库，那么由于各个实例各自维护主键状态，不同实例可能生成同样的主键，从而造成主键重复异常。所以，该方法只适合单线程对数据库的访问方式，不适合在多进程并发更新数据库的场合使用。因此，如果同一数据库有多个实例访问，最好不要使用这种方法。需要注意的是，该主键递增的方式是由 Hibernate 来维护的。

5) Identity。这种方式采用数据库提供的自增方式来生成新的主键，例如 DB2、SQL Server、MySQL 中的主键生成机制。该方式的特点是不需要 Hibernate 与开发人员的干涉，使用起来非常方便，但会给程序在不同数据库的移植带来严重不便。

6) Sequence。这种方式采用数据库提供的 Sequence (序列) 机制生成主键。这就要求数据库必须提供 Sequence 机制，例如 Oracle 就提供 Sequence 机制。其主要缺点是当程序在不同数据库之间移植时，特别是从支持序列的数据库移植到不支持序列的数据库时，使用该方式会非常麻烦。

7) Native。在该方式中，由 Hibernate 根据底层数据库自行选取 Identity、Hilo、Sequence 中的一种作为主键生成方式，例如，对于 Oracle 采用 Sequence 方式，对于 MySQL 和 SQL Server 采用 Identity 方式。该方式的一个主要优点就是灵活性更强，便于程序的移植。

8) UUID。uuid.hex 由 Hibernate 基于 128 位唯一值产生算法生成 16 进制数值 (编码后以长度 32 位的字符串表示) 作为主键。这种方式能够保证在不同的环境下主键的一致性。

uuid.string 与 uuid.hex 类似，只是生成的主键未进行编码（长度 16 位）。在某些数据库（例如 PostgreSQL）中可能出现问题。

该方式能够保证数据库中的主键唯一性，但是在生成的主键占用比较大的存储空间。

9) Foreign GUID。这种方式用于在一对一的关系中采用一种特殊的算法来生成主键，从而保证了主键的唯一性。

10) select。这种方式使用触发器生成主键，主要用于早期的数据库主键生成机制，现在使用得比较少。

5.3.18 如何实现分页机制

在交互式的应用程序中，当数据量很大时，如果一次性把所需数据全部从数据库中查询出来，不仅非常耗费时间，而且还会消耗大量的内存，导致用户操作的延时，严重影响系统的可用性。因此，为了降低系统的响应时间，提高系统的性能，往往会使用分页机制，即不是把用户所需数据一次性全部查找出来，而是把数据分成很多的页，每一页只包含指定的记录数，在查询时根据需求每次只查找一页或多页的数据而不是所有数据。由于采用分页机制使得查询的结果集中数据量减少了，同时也降低了内存的消耗，因此可以显著降低响应时间，有助于提高系统的可用性，增强用户体验。

下面主要介绍两种分页的实现方法。

(1) Hibernate 自带的分页机制

Hibernate 提供了一个支持跨系统的分页机制，该机制保证无论底层是什么样的数据库都能使用统一的接口进行分页操作。其用法如下：首先，通过 Session 对象获取 Query 对象；其次，使用 Query 对象的 setFirstResult() 方法来设置要查询的第一行数据；最后，用 setMaxResults() 方法来设置要查询结果集的大小。

下列代码的功能就是从第 100 条开始取出 50 条记录：

```
Query q = session.createQuery("from Student as s");
q.setFirstResult(100);
q.setMaxResults(50);
List l = q.list();
```

(2) 用 SQL 语句来实现分页

以 MySQL 为例，可以使用 limit 关键字来实现分页。其用法如下：

```
select * from tableName where 条件 limit begin,count
```

第一个参数（begin）代表查询开始的地方，第二个参数（count）代表每页显示多少条数据。需要注意的是，第一页用 0 表示，例如查询语句 select * from employee limit 0, 2 表示从表 employee 中第一行开始查找两行数据。

5.3.19 什么是 SSH

SSH 是 Struts、Spring 和 Hibernate 的首字母组合，它是一种比较流行的用来开发 Web 应用程序的开源框架，用于构建灵活、易于扩展的多层 Web 应用。

使用 SSH 框架开发的系统从职责上可以分为 4 层：表示层、业务逻辑层、数据持久化层和域模块层，如图 5-19 所示。

其中，Struts 实现了 MVC 模式，它对 Model、View 和 Controller 的各个模块都提供了支持。

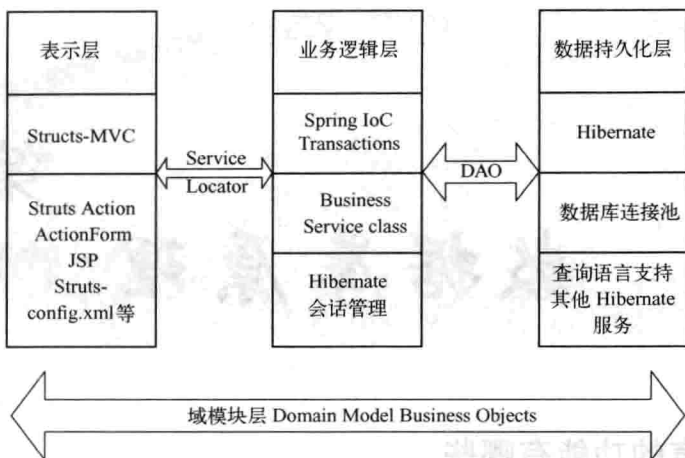


图 5-19 SSH 模块图

Struts 框架使用 JSP 实现了视图部分，模型部分则通过 Hibernate 框架提供的支持来实现数据的持久化，业务层则使用了 Spring 作为支持来管理对象。

接着，Spring 把抽象出的模型用 Java 类来实现，同时为这些模型编写对应的 DAO 接口，同时给出基于 Hibernate 的 DAO 的实现，即实现 Java 对象与关系数据库之间数据的转换。然后使用 Spring 来完成业务逻辑，来管理 Hibernate 与 Struts 对象。

采用 SSH 框架，不仅能实现视图、控制器与模型的彻底分离，而且还能实现业务逻辑层与数据持久层的分离。无论前端如何变化，模型层只需很少的改动即可满足要求；此外，数据库的变化也不会对前端有所影响，从而大大提高了系统的可复用性、可扩展性与易维护性。而且由于不同层之间的低耦合度，使得团队成员能够并行开发，从而大大节省了时间，提高了应用的开发效率。

由于 SSH 提供了很多有用的框架，因此能显著提高项目的开发效率。其主要优点如下：

1) Struts 实现了 MVC 模式，这种模式的特点是对应用程序进行了很好的分层，使得开发人员只需要把开发重点放在业务逻辑的开发即可。不仅如此，Struts 还提供了许多非常有用的标签库，这些标签能够显著地提高开发人员的开发效率。

2) Spring 可以用来管理对象。使用 Spring 的 IoC 容器，把对象之间的依赖关系交给 Spring，降低组件之间的耦合性，让开发人员更专注于业务逻辑的开发。Spring 采用了 IoC 和 AOP 的思想，使得它具有很好的模块化，程序与可以根据需求使用其中的一个模块。Spring 还提供了一些非常有用的功能，例如事务管理等。

3) Hibernate 作为一个轻量级的持久性框架，实现了高效的对象关系映射，使得开发可以完全采用面向对象的思想，而不需要关心数据库的关系模型。使用 Hibernate 开发的系统有很好的可移植性，可以很容易地实现不同数据库之间的移植，而不需要关心不同数据库 SQL 语句的差异。

第6章

数据库原理

6.1 SQL 语言的功能有哪些

SQL 是结构化查询语言 (Structured Query Language) 的缩写, 其功能包括数据查询、数据操纵、数据定义和数据控制 4 个部分。

数据查询是数据库中最常见的操作, 通过 select 语句可以得到所需的信息。SQL 语言的数据操纵语句 (Data Manipulation Language, DML) 主要包括插入数据、修改数据以及删除数据 3 种语句。SQL 语言使用数据定义语言 (Data Definition Language, DDL) 实现数据定义功能, 可对数据库用户、基本表、视图、索引进行定义与撤销。数据控制语句 (Data Control Language, DCL) 用于对数据库进行统一的控制管理, 保证数据在多用户共享的情况下能够安全。

基本的 SQL 语句有 select、insert、update、delete、create、drop、grant、revoke 等。其具体使用方式见表 6-1。

表 6-1 基本 SQL 语句的使用方式

	关键字	描述	语法格式
数据查询	select	选择符合条件的记录	select * from table where 条件语句
	insert	插入一条记录	insert into table (字段 1, 字段 2...) values (值 1, 值 2...)
数据操纵	update	更新语句	update table set 字段名 = 字段值 where 条件表达式
	delete	删除记录	Delete from table where 条件表达式
数据定义	create	数据表的建立	create table tablename (字段 1, 字段 2...)
	drop	数据表的删除	drop table tablename
数据控制	grant	为用户授予系统权限	grant <系统权限> <角色> [, <系统权限> <角色>] ...to <用户名> <角色> public [, <用户名> <角色>] ... [with admin option]
	revoke	收回系统权限	revoke <系统权限> <角色> [, <系统权限> <角色>] ... from <用户名> <角色> public [, <用户名> <角色>] ...

例如, 设教务管理系统中有 3 个基本表:

学生信息表 S(SNO, SNAME, AGE, SEX), 其属性分别表示学号、学生姓名、年龄和性别。

选课信息表 SC(SNO, CNO, SCGRADE), 其属性分别表示学号、课程号和成绩。

课程信息表 C(CNO, CNAME, CTEACHER), 其属性分别表示课程号、课程名称和任课老师姓名。

下面运用 SQL 语句进行相关操作。

1) 把 SC 表中每门课程的平均成绩插入到另外一个已经存在的表 SC_C(CNO, CNAME, AVG_GRADE)中, 其中 AVG_GRADE 表示每门课程的平均成绩。


```
INSERT INTO SC_C(CNO,CNAME,AVG_GRADE)
SELECT SC.CNO,C.NAME,AVG(SCGRADE) FROM SC,C WHERE SC.CNO = C.CNO
```

- 2) 从 SC 表中把选何昊老师所授课程的女生的选课记录删除。

```
DELETE FROM SC,S,C WHERE SC.SNO = S.SNO AND SC.CNO = C.CNO AND C.CTEACHER = '何昊'
```

- 3) 规定女生所选修何昊老师的课程的成绩都应该在 80 分以上 (含 80 分)。

```
ALTER TABLE SC,S,C
ADD CONSTRAINT GRADE CHECK( GRADE > =80)
WHERE SC.CNO = C.CNO and SC.SNO = S.SNO AND C.CTEACHER = '何昊'
```

- 4) 找出没有选修过何昊老师的课程的所有学生的姓名。

```
SELECT SNAME FROM S
WHERE NOT EXISTS(
SELECT * FROM SC,C WHERE SC.CNO = C.CNO AND CNAME = '何昊' AND SC.SNO = S.SNO)
```

- 5) 列出有两门以上 (含两门) 不及格课程 (成绩小于 60) 的学生的姓名及其平均成绩。

```
SELECT S.SNO,S.SNAME,AVG_SCGRADE = AVG(SC.SCGRADE)
FROM S,SC,(
SELECT SNO FROM SC WHERE SCGRADE < 60 GROUP BY SNO
HAVING COUNT(DISTINCT CNO) > =2)A WHERE S.SNO = A.SNO AND SC.SNO = A.SNO
GROUP BY S.SNO,S.SNAME
```

- 6) 列出既学过“1”号课程,又学过“2”号课程的所有学生的姓名。

```
SELECT S.SNO,S.SNAME
FROM S,(SELECT SC.SNO FROM SC,C
WHERE SC.CNO = C.CNO AND C.CNAME IN('1','2'))
GROUP BY SNO
HAVING COUNT(DISTINCT CNO) =2
)SC WHERE S.SNO = SC.SNO
```

- 7) 列出“1”号课成绩比“2”号同学该门课成绩高的所有学生的学号。

```
SELECT S.SNO,S.SNAME
FROM S,(
SELECT SC1.SNO
FROM SC SC1,C C1,SC SC2,C C2
WHERE SC1.CNO = C1.CNO AND C1.NAME = '1'
AND SC2.CNO = C2.CNO AND C2.NAME = '2'
AND SC1.SCGRADE > SC2.SCGRADE
)SC WHERE S.SNO = SC.SNO
```

- 8) 列出“1”号课成绩比“2”号课成绩高的所有学生的学号及其“1”号课和“2”号课的成绩。

```
SELECT S.SNO,S.SNAME,SC.[1号课成绩],SC.[2号课成绩]
FROM S,(
SELECT SC1.SNO,[1号课成绩] = SC1.SCGRADE,[2号课成绩] = SC2.SCGRADE
FROM SC SC1,C C1,SC SC2,C C2
WHERE SC1.CNO = C1.CNO AND C1.NAME = '1'
AND SC2.CNO = C2.CNO AND C2.NAME = '2'
```



```
AND SC1.SCGRADE > SC2.SCGRADE
)SC WHERE S.SNO = SC.SNO
```

引申: delete 与 truncate 命令有哪些区别?

相同点: 都可以用来删除一个表中的数据。

不同点:

1) truncate 是一个数据定义语言 (Data Definition Language, DDL), 它会被隐式地提交, 一旦执行后将不能回滚。delete 执行的过程是每次从表中删除一行数据, 同时将删除的操作以日志的形式进行保存, 以便将来进行回滚操作。

2) 用 delete 操作后, 被删除的数据占用的存储空间还在, 还可以恢复。而用 truncate 操作删除数据后, 被删除的数据会立即释放占用的存储空间, 被删除的数据是不能被恢复的。

3) truncate 的执行速度比 delete 快。

常见笔试题:

Oracle 数据库的一个表中有若干条数据, 其占用的存储空间为 10MB, 如果用 delete 语句删除表中的所有数据, 此时该表所占存储空间为多大?

答案: 10MB。数据库中 delete 操作类似于在 Windows 系统中把数据放到回收站, 还可以恢复, 因此它不会立即释放所占的存储空间。如果想在删除数据后立即释放存储空间, 可以使用 truncate。

6.2 内连接与外连接有什么区别

内连接, 也被称为自然连接, 只有两个表相匹配的行才能在结果集中出现。返回的结果集选取了两个表中所有相匹配的数据, 舍弃了不匹配的数据。由于内连接是从结果表中删除与其他连接表中没有匹配的所有行, 所以内连接可能会造成信息的丢失。内连接的语法如下:

```
select fieldlist from table1 [inner] join table2 on table1.column = table2.column
```

内连接是保证两个表中的所有行都满足连接条件, 而外连接则不然。外连接不仅包含符合连接条件的行, 而且还包括左表 (左外连接时)、右表 (右外连接时) 或两个边接表 (全外连接) 中的所有数据行。SQL 的外连接共有 3 种类型: 左外连接 (关键字为 LEFT OUTER JOIN)、右外连接 (关键字为 RIGHT OUTER JOIN) 和全外连接 (关键字为 FULL OUTER JOIN)。外连接的使用法和内连接一样, 只是将 INNER JOIN 关键字替换为相应的外连接关键字即可。

内连接只显示符合连接条件的记录, 外连接除了显示符合连接条件的记录外, 还显示表中的记录, 例如, 如果使用左外连接, 还显示左表中的记录。

下面为学生表 A 和学生表 B。表 6-2 为学生表 A, 表 6-3 为学生表 B。

表 6-2 学生表 A

学 号	姓 名
0001	张三
0002	李四
0003	王五

表 6-3 学生表 B

学 号	课 程 名
0001	数学
0002	英语
0003	数学
0004	计算机

对表 A 和表 B 进行内连接后的结果见表 6-4。

对表 A 和表 B 进行左外连接后的结果见表 6-5。

表 6-4 内连接

学 号	姓 名	课 程 名
0001	张三	数学
0002	李四	英语
0003	王五	数学

表 6-5 左外连接

学 号	姓 名	课 程 名
0001	张三	数学
0002	李四	英语
0003	王五	数学
0004		计算机

6.3 什么是事务

事务是数据库中一个单独的执行单元 (Unit)，它通常由高级数据库操作语言 (例如 SQL) 或编程语言 (例如 C++、Java 等) 编写的用户程序的执行所引起。当在数据库中更改数据成功时，在事务中更改的数据便会提交，不再改变。否则，事务就取消或者回滚，更改无效。

以网上购物为例，其交易过程至少包括以下几个步骤：

- 1) 更新客户所购商品的库存信息。
- 2) 保存客户付款信息。
- 3) 生成订单并且保存到数据库中。
- 4) 更新用户相关信息，例如购物数量等。

在正常情况下，这些操作都将顺利进行，最终交易成功，与交易相关的所有数据库信息也成功地更新。但是，如果遇到突然断电或是其他意外情况，导致这一系列过程中任何一个环节出了差错 (例如在更新商品库存信息时发生异常、顾客银行账户余额不足等)，都将导致整个交易过程失败。而一旦交易失败，数据库中所有信息都必须保持交易前的状态不变，例如最后一步更新用户信息时失败而导致交易失败，那么必须保证这笔失败的交易不影响数据库的状态，即原有的库存信息没有被更新、用户也没有付款、订单也没有生成。否则，数据库的信息将会不一致，或者出现更为严重的不可预测的后果，数据库事务正是用来保证这种情况下交易的平稳性和可预测性的技术。

事务必须满足 4 个属性，即原子性 (atomicity)、一致性 (consistency)、隔离性 (isolation)、持久性 (durability)，即 ACID 4 种属性。

(1) 原子性

事务是一个不可分割的整体，为了保证事务的总体目标，事务必须具有原子性，即当数据修改时，要么全执行，要么全不执行，即不允许事务部分地完成，避免了只执行这些操作的一部分而带来的错误。原子性要求事务必须被完整执行。

(2) 一致性

一个事务执行之前和执行之后，数据库数据必须保持一致性状态。数据库的一致性状态应该满足模式锁指定的约束，那么在完整执行该事务后数据库仍然处于一致性状态。为了维护所有数据的完整性，在关系型数据库中，所有规则必须应用到事务的修改上。数据库的一致性状态由用户来负责，由并发控制机制实现，例如银行转账，转账前后两个账户金额之和应保持不变。由于并发操作带来的数据不一致性包括丢失数据修改、读“脏”数据、不可重复读和产生“幽灵”数据。

(3) 隔离性

隔离性也被称为独立性，当两个或多个事务并发执行时，为了保证数据的安全性，将一个事物内部的操作与事务的操作隔离起来，不被其他正在进行的事务看到，例如，对任何一对事

事务 T1 和 T2, 对 T1 而言, T2 要么在 T1 开始之前已经结束, 要么在 T1 完成之后再开始执行。数据库有 4 种类型的事务隔离级别: 不提交的读、提交的读、可重复的读和串行化。因为隔离性使得每个事务的更新在它被提交之前, 对其他事务都是不可见的, 所以, 实现隔离性是解决临时更新与消除级联回滚问题的一种方式。

(4) 持久性

持久性也被称为永久性, 事务完成以后, DBMS 保证它对数据库中的数据修改是永久性的, 当系统或介质发生故障时, 该修改也永久保持。持久性一般通过数据库备份与恢复来保证。

严格来说, 数据库事务属性 (ACID) 都是由数据库管理系统来进行保证的, 在整个应用程序运行过程中, 应用无需去考虑数据库的 ACID 实现。

一般情况下, 通过执行 COMMIT 或 ROLLBACK 语句来终止事务, 当执行 COMMIT 语句时, 自事务启动以来对数据库所做的一切更改就成为永久性的了, 即被写入磁盘; 而当执行 ROLLBACK 语句时, 自事务启动以来对数据库所做的一切更改都会被撤销, 并且数据库中内容返回到事务开始之前所处的状态。无论什么情况, 在事务完成时, 都能保证回到一致状态。

6.4 什么是存储过程? 它与函数有什么区别与联系

SQL 语句在执行时要先编译, 然后再被执行。在大型数据库系统中, 为了提高效率, 将为了完成特定功能的 SQL 语句集进行编译优化后, 存储在数据库服务器中, 用户通过指定存储过程的名字来调用执行。

下面为一个创建存储过程的常用语法。

```
create procedure sp_name @[参数名][类型]
as
begin
.....
end
```

调用存储过程语法: `exec sp_name [参数名]`。

删除存储过程语法: `drop procedure sp_name`。

使用存储过程可以增强 SQL 语言的功能和灵活性, 由于可以用流程控制语句编写存储过程, 有很强的灵活性, 因此可以完成复杂的判断和运算, 并且可以保证数据的安全性和完整性。同时, 存储过程可以使没有权限的用户在控制之下间接地存取数据库, 也保证了数据的安全。

需要注意的是, 存储过程不等于函数, 二者虽然本质上没有区别, 但还是有如下几个方面的不同:

1) 存储过程一般是作为一个独立的部分来执行, 而函数可以作为查询语句的一个部分来调用。由于函数可以返回一个对象, 因此它可以在查询语句中位于 From 关键字的后面。

2) 一般而言, 存储过程实现的功能较复杂, 而函数实现的功能针对性较强。

3) 函数需要用括号包住输入的参数, 且只能返回一个值或表对象, 而存储过程可以返回多个参数。

4) 函数可以嵌入在 SQL 中使用, 可以在 select 中调用, 存储过程则不行。

5) 函数不能直接操作实体表, 只能操作内建表。

6) 存储过程在创建时即在服务器上进行了编译, 其执行速度比函数快。

6.5 各种范式有什么区别

在设计与操作维护数据库时，最关键的问题就是要确保数据能够正确地分布到数据库的表中。使用正确的数据结构，不仅有助于对数据库进行相应的存取操作，还可以极大地简化应用程序中的其他内容（查询、窗体、报表、代码等），按照“数据库规范化”对表进行设计，其目的就是减少数据库中的数据冗余，以增加数据的一致性。

规范化是在识别数据库中的数据元素、关系以及定义所需的表和表中的项目这些初始工作之后的一个细化的过程。常见的范式有1NF、2NF、3NF、BCNF以及4NF。

1) 1NF（第一范式）。第一范式是指数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。如果出现重复的属性，就可能需要定义一个新的实体，新的实体由重复的属性构成，新实体与原实体之间为一对多关系。第一范式的模式要求属性值不可再分裂成更小部分，即属性项不能是属性组合或由组属性组成。简而言之，第一范式就是无重复的列，例如，由“职工号”“姓名”“电话号码”组成的表（一个人可能有一部办公电话和一部移动电话），这时将其规范化化为1NF可以将电话号码分为“办公电话”和移动电话两个属性，即职工（职工号，姓名，办公电话，移动电话）。

2) 2NF（第二范式）。第二范式（2NF）是在第一范式（1NF）的基础上建立起来的，即满足第二范式（2NF）必须先满足第一范式（1NF）。第二范式（2NF）要求数据库表中的每个实例或行必须可以被唯一地区分。为实现区分通常需要为表加上一个列，以存储各个实例的唯一标识。如果关系模式R为第一范式，并且R中的每一个非主属性完全函数依赖于R的某个候选键，则称R为第二范式模式（如果A是关系模式R的候选键的一个属性，则称A是R的主属性，否则称A是R的非主属性），例如，在选课关系表（学号，课程号，成绩，学分），关键字为组合关键字（学号，课程号），但由于非主属性学分仅依赖于课程号，对关键字（学号，课程号）只是部分依赖，而不是完全依赖，因此此种方式会导致数据冗余以及更新异常等问题，解决办法是将其分为两个关系模式：学生表（学号，课程号，分数）和课程表（课程号，学分），新关系通过学生表中的外关键字课程号联系，在需要时进行连接。

3) 3NF（第三范式）。如果关系模式R是第二范式，且每个非主属性都不传递依赖于R的候选键，则称R是第三范式的模式，以学生表（学号，姓名，课程号，成绩）为例，其中学生姓名无重名，所以该表有两个候选码（学号，课程号）和（姓名，课程号），故存在函数依赖：学号→姓名，（学号，课程号）→成绩，（姓名，课程号）→成绩，唯一的非主属性成绩对码不存在部分依赖，也不存在传递依赖，所以属于第三范式。

4) BCNF。它构建在第三范式的基础上，如果关系模式R是第一范式，且每个属性都不传递依赖于R的候选键，那么称R为BCNF的模式。假设仓库管理关系表（仓库号，存储物品号，管理员号，数量），满足一个管理员只在一个仓库工作；一个仓库可以存储多种物品，则存在如下关系：

（仓库号，存储物品号）→（管理员号，数量）

（管理员号，存储物品号）→（仓库号，数量）

所以，（仓库号，存储物品号）和（管理员号，存储物品号）都是仓库管理关系表的候选码，表中的唯一非关键字段为数量，它是符合第三范式的。但是，由于存在如下决定关系：

（仓库号）→（管理员号）

（管理员号）→（仓库号）

即存在关键字段决定关键字段的情况，因此其不符合 BCNF。把仓库管理关系表分解为两个关系表：仓库管理表（仓库号，管理员号）和仓库表（仓库号，存储物品号，数量），这样的数据库表是符合 BCNF 的，并消除了删除异常、插入异常和更新异常。

5) 4NF (第四范式)。设 R 是一个关系模式，D 是 R 上的多值依赖集合。如果 D 中存在凡多值依赖 $X \twoheadrightarrow Y$ 时，X 必是 R 的超键，那么称 R 是第四范式的模式，例如，职工表（职工编号，职工孩子姓名，职工选修课程），在这个表中同一个职工可能会有多个职工孩子姓名，同样，同一个职工也可能会有多个职工选修课程，即这里存在着多值事实，不符合第四范式。如果要符合第四范式，只需要将上表分为两个表，使它们只有一个多值事实，例如职工表一（职工编号，职工孩子姓名），职工表二（职工编号，职工选修课程），两个表都只有一个多值事实，所以符合第四范式。

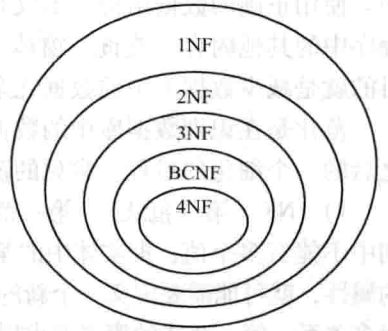


图 6-1 为各范式的关系图。

图 6-1 各范式的关系图

6.6 什么是触发器

触发器是一种特殊类型的存储过程，它由事件触发，而不是程序调用或手工启动，当数据库有特殊的操作时，这些操作由数据库中的事件来触发，自动完成这些 SQL 语句。使用触发器可以用来保证数据的有效性和完整性，完成比约束更复杂的数据约束。

触发器与存储过程的区别见表 6-6。

表 6-6 触发器与存储过程区别

触发器	存储过程
当某类数据操纵 DML 语句发生时隐式地调用	从一个应用或过程中显式地调用
在触发器体内禁止使用 COMMIT 和 ROLLBACK 语句	在过程体内可以使用所有 PL/SQL 块中都能使用的 SQL 语句，包括 COMMIT 和 ROLLBACK 语句
不能接受参数输入	可以接受参数输入

根据 SQL 语句的不同，触发器可分为 DML 触发器和 DLL 触发器。

DML 触发器是当数据库服务器发生数据操作语言事件时执行的存储过程，有 After 和 Instead Of 这两种触发器。After 触发器被激活触发是在记录改变之后进行的一种触发器。Instead Of 触发器是在记录变更之前，去执行触发器本身所定义的操作，而不是执行原来 SQL 语句里的操作。DLL 触发器是在响应数据定义语言事件时执行的存储过程。

触发器的主要作用表现为以下几个方面：

- 1) 增加安全性。
- 2) 利用触发器记录所进行的修改以及相关信息，跟踪用户对数据库的操作，实现审计。
- 3) 维护那些通过创建表时的声明约束不可能实现的复杂的完整性约束以及对数据库中特定事件进行监控与响应。
- 4) 实现复杂的非标准的数据库相关完整性规则、同步实时地复制表中的数据。
- 5) 触发器是自动的，它们在对表的数据做了任何修改之后就会被激活，例如可以自动计算数据值，若数据的值达到了一定的要求，则进行特定的处理。以某企业财务管理为例，如果

企业的资金出现短缺，并且达到某种程度时，则会发送警告信息。

下面是一个触发器的例子，该触发器的功能是在每周末进行数据表更新，若当前用户没有访问 WEEKEND_UPDATE_OK 表的权限，则需要重新赋予权限。

```
CREATE OR REPLACE TRIGGER update_on_weekends_check
BEFORE UPDATE OF sal ON EMP
FOR EACH ROW
DECLARE
my_count number(4);
BEGIN
SELECT COUNT(u_name)
FROM WEEKEND_UPDATE_OK INTO my_count
WHERE u_name = user_name;
IF my_count = 0 THEN
RAISE_APPLICATION_ERROR(20508, 'Update not allowed ');
END IF;
END;
```

引申：触发器分为事前触发和事后触发，这两者有什么区别？语句级触发和行级触发有什么区别？

事前触发发生在事件发生之前，用于验证一些条件或进行一些准备工作；事后触发发生在事件发生之后，做收尾工作。事前触发可以获得之前和新的字段值，而事后触发可以保证事务的完整性。语句级触发可以在语句执行之前或之后执行，而行级触发在触发器所影响的每一行触发一次。

6.7 什么是游标

数据库中，游标提供了一种对从表中检索出的数据进行操作的灵活手段，它实际上是一种能从包含多条数据记录的结果集中每次提取一条记录的机制。

游标总是与一条 SQL 选择语句相关联，因为游标是由结果集（可以是零条、一条或由相关的选择语句检索出的多条记录）和结果集中指向特定记录的游标位置组成的。当决定对结果集进行处理时，必须声明一个指向该结果集的游标。

游标允许应用程序对查询语句 select 返回的行结果集中的每一行进行相同或不同的操作，而不是一次对整个结果集进行同一种操作；它还提供对基于游标位置而对表中数据进行删除或更新的功能；游标还把作为面向集合的数据库管理系统和面向行的程序设计连接了起来，使两种数据处理方式能够进行“沟通”。

例如，声明一个游标 student_cursor，用于访问数据库 SCHOOL 中的“学生基本信息表”，代码如下：

```
USE SCHOOL
GO
DECLARE student_cursor CURSOR
FROM SELECT * FROM 学生基本信息表
```

上述代码中，声明游标时，在 SELECT 语句中未使用 WHERE 子句，故此游标返回的结果集是由“学生基本信息表”中的所有记录构成的。

在 select 返回的行集合中，游标不允许程序对整个行集合执行相同的操作，但对每一行数据的操作不作要求。游标有以下两个优点：

- 1) 在使用游标的表中，对行提供删除和更新的功能。
- 2) 游标将面向集合的数据库管理系统和面向行的程序设计连接了起来。

6.8 如果数据库日志满了，会出现什么情况

日志文件 (Log File) 记录所有对数据库数据的修改，主要是保护数据库以防故障发生，以及恢复数据时使用。其特点如下：

- 1) 每一个数据库至少包含两个日志文件组。每个日志文件组至少包含两个日志文件成员。
- 2) 日志文件组以循环方式进行写操作。
- 3) 每一个日志文件成员对应一个物理文件。

通过日志文件来记录数据库事务可以最大限度地保证数据的一致性与安全性，但一旦数据库中日志满了，就只能执行查询等读操作，不能执行更改、备份等操作，原因是任何写操作都要记录日志，也就是说，基本上处于不能使用的状态。

6.9 union 和 union all 有什么区别

union 在进行表求并集后会去掉重复的元素，所以会对所产生的结果集进行排序运算，删除重复的记录再返回结果。

union all 则只是简单地将两个结果集合并后就返回结果。因此，如果返回的两个结果集中有重复的数据，那么返回的结果集就会包含重复的数据。

从上面的对比可以看出，在执行查询操作时，union all 要比 union 快很多，所以，如果可以确认合并的两个结果集中不包含重复的数据，那么最好使用 union all，例如，现有两个学生表 Table 1 (见表 6-7) 和 Table 2 (见表 6-8)。

表 6-7 Table 1

Table 1	
C1	C2
1	1
2	2
3	3

表 6-8 Table 2

Table 2	
C1	C2
3	3
4	4
1	1

select * from Table1 union select * from Table2 的查询结果见表 6-9：

表 6-9 使用 union 得到的查询结果

C1	C2
1	1
2	2
3	3
4	4

select * from Table1 union all select * from Table2 的查询结果见表 6-10：

表 6-10 使用 union all 得到的查询结果

C1	C2
1	1
2	2
3	3
3	3
4	4
1	1

6.10 什么是视图

视图是由从数据库的基本表中选取出来的数据组成的逻辑窗口，与基本表不同，它是一个虚表。在数据库中，存放的只是视图的定义，而不存放视图包含的数据项，这些项目仍然存放在原来的基本表结构中。

视图的作用主要有以下几点：首先，可以简化数据查询语句；其次，可以使用户能从多角度看待同一数据；然后，通过引入视图可以提高数据的安全性；最后，视图提供了一定程度的逻辑独立性等。

通过引入视图机制，用户可以将注意力集中在其关心的数据上（而非全部数据），这样就大大提高了用户效率与用户满意度，而且如果这些数据来源于多个基本表结构，或者数据不仅来自于基本表结构，还有一部分数据来源于其他视图，并且搜索条件又比较复杂时，需要编写的查询语句就会比较烦琐，此时定义视图就可以使数据的查询语句变得简单可行。定义视图可以将表与表之间的复杂的操作连接和搜索条件对用户不可见，用户只需要简单地对一个视图进行查询即可，故增加了数据的安全性，但不能提高查询的效率。

第7章

设计模式

设计模式 (Design Pattern) 是一套被反复使用、为多数人知晓、经过分类编目的、代码设计经验的总结。使用设计模式的目的是为了代码重用, 避免程序大量修改, 同时使代码更易于理解, 并且保证代码可靠性。显然, 设计模式不管是对自己还是对他人还是对系统都是有益的, 设计模式使得代码编制真正实现工程化, 设计模式可以说是软件工程的“基石”。

GoF (Gang of Four) 23 种经典设计模式见表 7-1。

表 7-1 23 种经典设计模式

	创建型	结构型	行为型
类	Factory Method (工厂方法)	Adapter_Class (适配器类)	Interpreter (解释器) Template Method (模板方法)
对象	Abstract Factory (抽象工厂) Builder (生成器) Prototype (原型) Singleton (单例)	Adapter_Object (适配器对象) Bridge (桥接) Composite (组合) Decorator (装饰) Façade (外观) Flyweight (享元) Proxy (代理)	Chain of Responsibility (职责链) Command (命令) Iterator (迭代器) Mediator (中介者) Memento (备忘录) Observer (观察者) State (状态) Strategy (策略) Visitor (访问者模式)

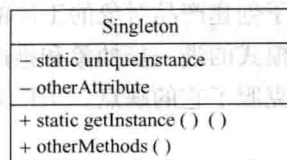
常见的设计模式有工厂模式 (Factory Pattern)、单例模式 (Singleton Pattern)、适配器模式 (Adapter Pattern)、享元模式 (Flyweight Pattern) 以及观察者模式 (Observer Pattern) 等。

7.1 什么是单例模式

在某些情况下, 有些对象只需要一个就可以了, 即每个类只需要一个实例, 例如, 一台计算机上可以连接多台打印机, 但是这个计算机上的打印程序只能有一个, 这里就可以通过单例模式来避免两个打印作业同时输出到打印机中, 即在整个的打印过程中只有一个打印程序的实例。

简单说来, 单例模式 (也叫单件模式) 的作用就是保证在整个应用程序的生命周期中, 任何一个时刻, 单例类的实例都只存在一个 (当然也可以不存在)。

单例模式确保某一个类只有一个实例，而且自行实例化，并向整个系统提供这个实例单例模式。单例模式只应在有真正的“单一实例”需求时才可使用。单例模式的设计类图如图 7-1 所示。



其中，类变量 `uniqueInstance` 持有唯一的单例实例，类方法 `getInstance()` 用来获取唯一的实例化对象。

图 7-1 单例模式的设计类图

全局变量和单例模式的区别为：首先，全局变量是对一个对象的静态引用，全局变量确实可以提供单例模式实现的全局访问这个功能，但是它并不能保证应用程序中只有一个实例；其次，编码规范也明确指出应该要少用全局变量，因为过多使用全局变量，会造成代码难读；最后全局变量并不能实现继承（虽然单例模式在继承上也不能很好地处理，但是还是可以实现继承的）。而单例模式在类中保存了它的唯一实例——这个类，它可以保证只能创建一个实例，同时它还提供了一个访问该唯一实例的全局访问点。

使用单例模式，需要注意的是，单例模式用来保证系统中一个类只有一个实例。单例类的构造函数必须为私有，同时单例类必须提供一个全局访问点。

常见笔试题：

请实现一个单例模式。

答案：代码如下所示。

```
public class Test {
    private Test() {}
    private static Test uniqueInstance = new Test();
    public static Test getInstance() {
        return uniqueInstance;
    }
}
```

使用这种方法实现的单例模式，在类被加载时就会实例化这个类的一个对象，由于在使用之前对象已经创建好，因此，可以在多线程环境下使用这种方法。如果采用按需实例化的方法（在实例化对象在使用的时候才实例化），就需要考虑多线程的同步。

7.2 什么是工厂模式

工厂模式专门负责实例化有大量公共接口的类。工厂模式可以动态地决定将哪一个类实例化，而不必事先知道每次要实例化哪一个类。客户类和工厂类是分开的。消费者无论什么时候需要某种产品，需要做的只是向工厂提出请求即可。消费者无须修改就可以接纳新产品。当然也存在缺点，就是当产品修改时，工厂类也要做相应的修改。

工厂模式包含以下几种形态：

1) 简单工厂 (Simple Factory) 模式。简单工厂模式的工厂类是根据提供给它的参数，返回的是几个可能产品中的一个类的实例，通常情况下，它返回的类都有一个公共的父类和公共的方法。简单工厂模式的设计类图如图 7-2 所示。

其中，`Product` 为待实例化类的基类，它可以有多个子类；`Simple Factory` 类中提供了实例化 `Product` 的方法，这个方法可以根据传入的参数动态地创建出某一类型产品的对象。

2) 工厂方法 (Factory Method) 模式。工厂方法模式是类的创建模式，其用意是定义一个

用于创建产品对象的工厂的接口，而将实际创建工作推迟到工厂接口的子类中。它属于简单工厂模式的进一步抽象和推广。多态的使用，使得工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点。工厂方法模式的设计类图如图 7-3 所示。

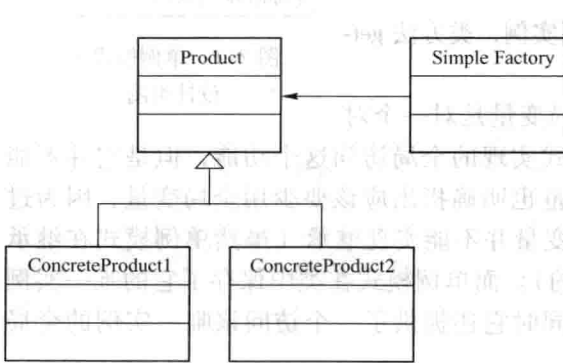


图 7-2 简单工厂模式的设计类图

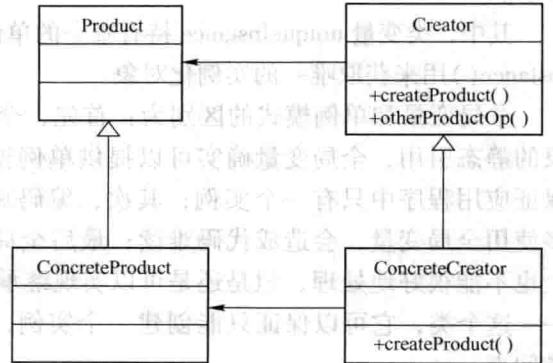


图 7-3 工厂方法模式的设计类图

Product 为产品的接口或基类，所有产品都实现这个接口或抽象类（例如 ConcreteProduct），这样就可以在运行时根据需求创建对应的产品类。Creator 实现了对产品的所有操作方法，而不实现产品对象的实例化。产品的实例化由 Creator 的子类来完成。

3) 抽象工厂 (Abstract Factory) 模式。抽象工厂模式是所有形态的工厂模式中最为抽象和最具一般性的一种形态。抽象工厂模式是指当有多个抽象角色时使用的一种工厂模式，抽象工厂模式可以向客户端提供一个接口，使客户端在不必指定产品的具体的情况下，创建多个产品族中的产品对象。根据 LSP 原则（即 Liskov 替换原则），任何接受父类型的地方，都应当能够接受子类型。因此，实际上系统所需要的，仅仅是类型与这些抽象产品角色相同的一些实例，而不是这些抽象产品的实例。换句话说，也就是这些抽象产品的具体子类的实例。工厂类负责创建抽象产品的具体子类的实例。抽象工厂模式的设计类图如图 7-4 所示。

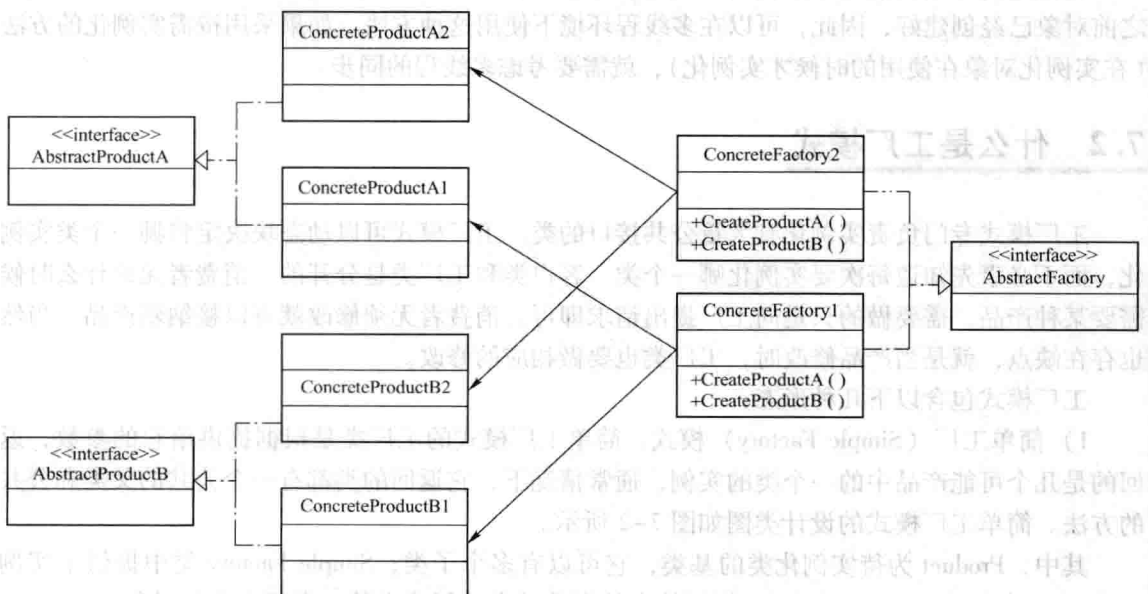


图 7-4 抽象工厂模式的设计类图

AbstractProductA 和 AbstractProductB 代表一个产品家族，实现这些接口的类代表具体的产品。AbstractFactory 为创建产品的接口，能够创建这个产品家族的所有类型的产品，它的子类可以根据具体情况创建对应的产品。

7.3 什么是适配器模式

适配器模式也称为变压器模式，它是把一个类的接口转换成客户端所期望的另一种接口，从而使原本因接口不匹配而无法一起工作的两个类能够一起工作。适配类可以根据所传递的参数返回一个合适的实例给客户端。

适配器模式主要应用于“复用一些现存的类，但是接口又与复用环境要求不一致”的情况，在遗留代码复用、类库迁移等方面非常有用。同时，适配器模式有对象适配器和类适配器两种形式的实现结构，但是类适配器采用“多继承”的实现方式，会引起程序的高耦合，所以一般不推荐使用；而对象适配器采用“对象组合”的方式，耦合度低，应用范围更广。

例如，现在系统里已经实现了点、线、正方形，而现在客户要求实现一个圆形，一般的做法是建立一个 Circle 类来继承以后的 Shape 类，然后去实现对应的 display、fill、undisplay 方法，此时如果发现项目组其他人已经实现了一个画圆的类，但是他的方法名却和自己的不一样（为 displayhh、fillhh、undisplayhh），则不能直接使用这个类，因为那样无法保证多态，而有的时候，也不能要求组件类改写方法名，在这样的情况下，可以采用适配器模式。适配器模式的设计类图如图 7-5 所示。

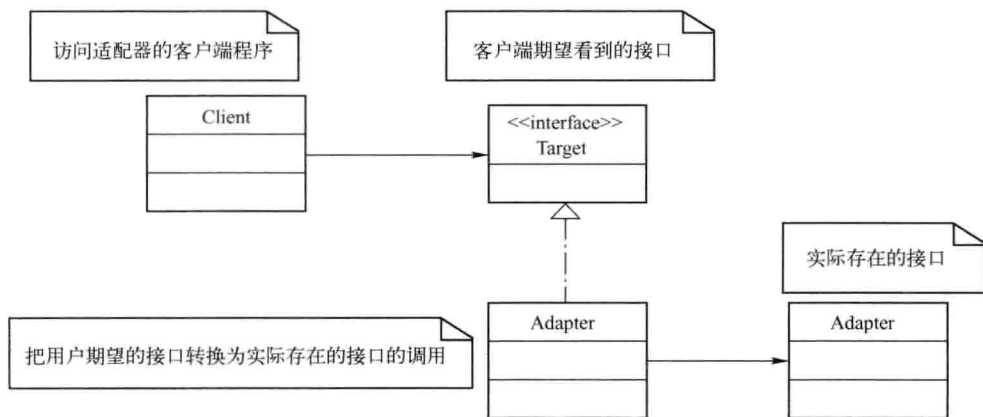


图 7-5 适配器模式的设计类图

7.4 什么是观察者模式

观察者模式（也被称为发布/订阅模式）提供了避免组件之间紧密耦合的另一种方法，它将观察者和被观察的对象分离开来。在该模式中，一个对象通过添加一个方法（该方法允许另一个对象，即观察者注册自己）使本身变得可观察。当可观察的对象更改时，它会将消息发送到已注册的观察者。这些观察者使用该信息执行的操作与可观察的对象无关，结果是对象可以相互对话，而不必了解原因。Java 与 C# 的事件处理机制就是采用的此种设计模式。

例如，用户界面可以作为一个观察者，业务数据是被观察者，用户界面观察业务数据的变化，发现数据变化后，就显示在界面上。面向对象设计的一个原则是：系统中的每个类将重点放在某一个功能上，而不是其他方面。一个对象只做一件事情，并且将它做好。观察者模式在模块之间划定了清晰的界限，提高了应用程序的可维护性和重用性。观察者模式的设计类图如图 7-6 所示。

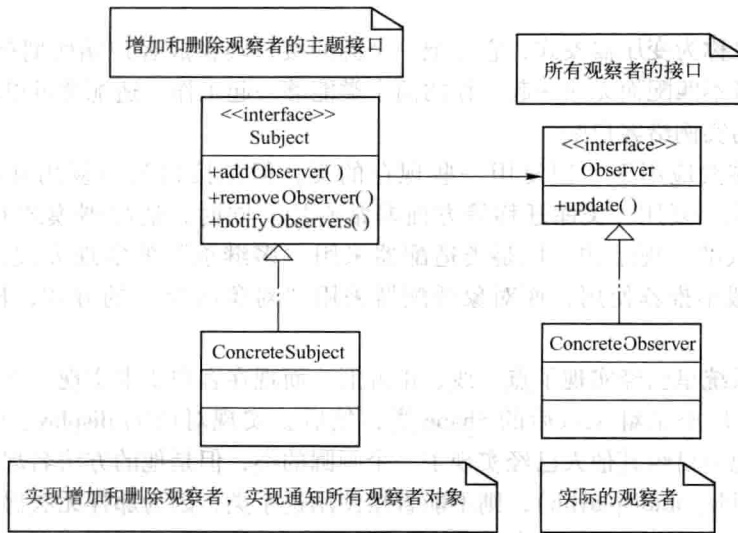


图 7-6 观察者模式的设计类图

第 8 章

数据结构与算法

8.1 链表

8.1.1 如何实现单链表的增删操作

链表作为最基本的数据结构，在程序设计中有着非常重要的作用，其存储特点如下：可以用任意一组存储单元来存储单链表中的数据元素（存储单元可以是不连续的），而且，除了存储每个数据元素 a_i 的值以外，还必须存储指示其直接后继元素的信息。这两部分信息组成的数据元素 a_i 的存储映像称为结点。N 个结点链在一块被称为链表，当结点只包含其后继结点的信息的链表就被称为单链表，在内存中存储的方式如图 8-1 所示。

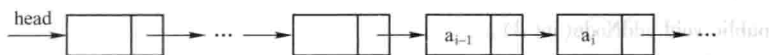


图 8-1 单链表存储的方式

在 Java 语言中，可以定义如下的数据类来存储结点信息。

```
class Node {
    Node next = null;
    int data;
    public Node(int data) { this.data = data; }
}
```

链表最重要的操作就是向链表中插入元素和从链表中删除元素。

单链表的插入操作是将值为 x 的新结点插入到单链表的第 i 个结点的位置上，即插入到数据元素 a_{i-1} 与 a_i 之间。其具体步骤如下：

- 1) 找到 a_{i-1} 的引用（存储位置） p 。
- 2) 生成一个数据域为 x 的新结点 s 。
- 3) 设置 $p.next = s$ 。
- 4) 设置 $s.next = a_i$ 。

图 8-2 为单链表插入结点示意图。

单链表的删除操作是将单链表的第 i 个结点删去。其具体步骤如下：

- 1) 找到 a_{i-1} 的存储位置 p 。

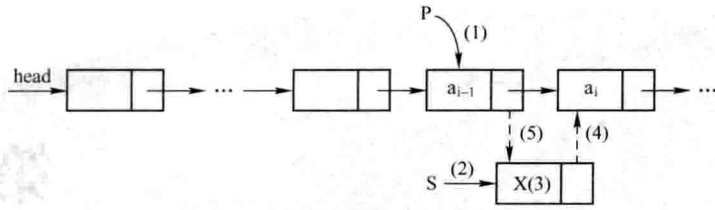


图 8-2 单链表插入结点示意图

2) 令 $p.next$ 指向 a_i 的直接后继结点 (即把 a_i 从链上摘下) a_{i+1} 。

图 8-3 为单链表删除结点示意图。

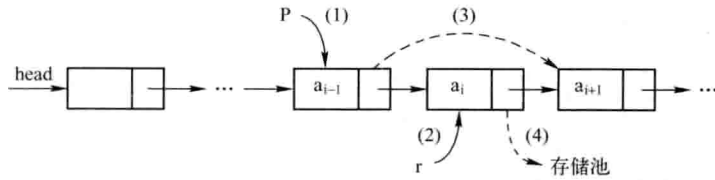


图 8-3 单链表删除结点示意图

下面链表操作的示例给出了链表的基本操作。

```

public class MyLinkedList {
    Node head = null; // 链表头的引用
    /** 向链表中插入数据
     * @param d: 插入数据的内容
     */
    public void addNode(int d) {
        Node newNode = new Node(d);
        if (head == null) {
            head = newNode;
            return;
        }
        Node tmp = head;
        while (tmp.next != null) {
            tmp = tmp.next;
        }
        // add node to end
        tmp.next = newNode;
    }
    /**
     * @param index: 删除第 index 个结点
     * @return 成功返回 true, 失败返回 false
     */
    public Boolean deleteNode(int index) { // 删除元素的位置不合理
        if (index < 1 || index > length()) {
            return false;
        }
        // 删除链表第一个元素
        if (index == 1) {

```

```

        head = head. next;
        return true;
    }
    int i = 1;
    Node preNode = head;
    Node curNode = preNode. next;
    while( curNode != null ) {
        if( i == index ) {
            preNode. next = curNode. next;
            return true;
        }
        preNode = curNode;
        curNode = curNode. next;
        i ++;
    }
    return true;
}
/**
 * @return 返回结点的长度
 */
public int length() {
    int length = 0;
    Node tmp = head;
    while( tmp != null ) {
        length ++;
        tmp = tmp. next;
    }
    return length;
}
/**
 * 对链表进行排序
 * 返回排序后的头结点
 */
public Node orderList() {
    Node nextNode = null;
    int temp = 0;
    Node curNode = head;
    while( curNode. next != null ) {
        nextNode = curNode. next;
        while( nextNode != null ) {
            if( curNode. data > nextNode. data ) {
                temp = curNode. data;
                curNode. data = nextNode. data;
                nextNode. data = temp;
            }
            nextNode = nextNode. next;
        }
        curNode = curNode. next;
    }
}

```

```

        return head;
    }

    public void printList() {
        Node tmp = head;
        while (tmp != null) {
            System.out.println(tmp.data);
            tmp = tmp.next;
        }
    }

    public static void main(String[] args) {
        MyLinkedList list = new MyLinkedList();
        list.addNode(5);
        list.addNode(3);
        list.addNode(1);
        list.addNode(3);
        System.out.println("listLen = " + list.length());
        System.out.println("before order:");
        list.printList();
        list.orderList();
        System.out.println("after order:");
        list.printList();
    }
}

```

上述程序运行结果为：

```

listLen = 3
before order:
1
5
3
after order:
1
3
5

```

以上这个例子主要实现了链表的最基本的操作，这些操作包括给链表增加结点（每次都把新增加的结点加到链表尾部）和删除链表中的结点和计算链表的长度。此外还通过插入排序算法实现了对链表的排序。

8.1.2 如何从链表中删除重复数据

如何从链表中删除重复数据，最容易想到的方法就是遍历链表，把遍历到的值存储到一个 Hashtable 中，在遍历过程中，若当前访问的值在 Hashtable 中已经存在，则说明这个数据是重复的，因此就可以删除。具体实现代码如下：

```

public void deleteDuplecate(Node head) {
    Hashtable < Integer, Integer > table = new Hashtable < Integer, Integer > ();
    Node tmp = head;
    Node pre = null;

```

```

while(tmp != null) {
    if(table.containsKey(tmp.data))
        pre.next = tmp.next;
    else {
        table.put(tmp.data,1);
        pre = tmp;
    }
    tmp = tmp.next;
}

```

以上这种方法的优点是时间复杂度较低，但也有一个很明显的缺点，就是在遍历过程中需要额外的存储空间来保存已遍历过的值。是否还有更加高效的算法呢？下面介绍另外一种不需要额外存储空间的算法。

这种方法的主要思路为对链表进行双重循环遍历，外循环正常遍历链表，假设外循环当前遍历的结点为 *cur*，内循环从 *cur* 开始遍历，若碰到与 *cur* 所指向结点值相同，则删除这个重复结点。算法的实现方法如下：

```

public void deleteDupliccate(Node head) {
    Node p = head;
    while(p != null) {
        Node q = p;
        while(q.next != null) {
            if(p.data == q.next.data) {
                q.next = q.next.next;
            } else
                q = q.next;
        }
        p = p.next;
    }
}

```

以上方法的优点是不需要额外的存储空间，缺点也很明显，时间复杂度比上面介绍的算法的时间复杂度高。假设外循环当前遍历的结点为 *cur*，内循环在遍历的过程中会删除掉与 *cur* 结点值相同的所有结点。在实现时还可以采用另外一种实现方法：外循环当前遍历的结点为 *cur*，内循环从链表头开始遍历至 *cur*，只要碰到与 *cur* 值相同的结点就删除该结点，同时内循环结束，因为与 *cur* 相同的结点只可能存在一个（如果存在多个，在前面的遍历过程中已经被删除了）。采用这种方法在特定的数据发布的情况下会提高算法的时间复杂度。

8.1.3 如何找出单链表中的倒数第 *k* 个元素

为了找出单链表中的倒数第 *k* 个元素，最容易想到的方法是首先遍历一遍单链表，求出整个单链表的长度 *n*，然后将倒数第 *k* 个，转换为正数第 *n - k* 个，接下去遍历一次就可以得到结果。但是该方法存在一个问题，即需要对链表进行两次遍历，第一次遍历用于求解单链表的长度，第二次遍历用于查找正数第 *n - k* 个元素。

显然，以上这种方法还可以进行优化。于是想到了第二种方法，如果沿从头至尾的方向从链表中的某个元素开始，遍历 *k* 个元素后刚好达到链表尾，那么该元素就是要找的倒数第 *k* 个

元素，根据这一性质，可以设计如下算法：从头结点开始，依次对链表的每一个结点元素进行这样的测试，遍历 k 个元素，查看是否到达链表尾，直到找到那个倒数第 k 个元素。此种方法将对同一批元素进行反复多次的遍历，对于链表中的大部分元素而言，都要遍历 k 个元素，如果链表长度为 n ，该算法时间复杂度为 $O(kn)$ 级，效率太低。

存在另外一种更高效的方式，只需要一次遍历即可查找到倒数第 k 个元素。由于单链表只能从头到尾依次访问链表的各个结点，因此，如果要找出链表的倒数第 k 个元素的话，也只能从头到尾进行遍历查找，在查找过程中，设置两个指针，让其中一个指针比另一个指针（虽然 Java 语言没有指针的概念，但是引用与指针有着非常相似的性质。为了便于理解，在后续的介绍中都采用指针的概念来介绍）先前移 $k-1$ 步，然后两个指针同时往前移动。循环直到先行的指针值为 NULL 时，另一个指针所指的位置就是所要找的位置。程序代码如下：

```
public Node findElem(Node head,int k){
    if(k < 1 || k > this.length())
        return null;
    Node p1 = head;
    Node p2 = head;
    for(int i=0;i < k-1;i++)//前移 k-1 步
        p1 = p1.next;
    while(p1 != null){
        p1 = p1.next;
        p2 = p2.next;
    }
    return p2;
}
```

8.1.4 如何实现链表的反转

为了正确地反转一个链表，需要调整指针的指向，而与指针操作相关代码总是非常容易出错的。先举个例子看一下具体的反转过程，例如， i ， m ， n 是 3 个相邻的结点，假设经过若干步操作，已经把结点 i 之前的指针调整完毕，这些结点的 $next$ 指针都指向前面一个结点。现在遍历到结点 m ，当然，需要调整结点的 $next$ 指针，让它指向结点 i ，但需要注意的是，一旦调整了指针的指向，链表就断开了，因为已经没有指针指向结点 n ，没有办法再遍历到结点 n 了，所以为了避免链表断开，需要在调整 m 的 $next$ 之前要把 n 保存下来。接下来试着找到反转后链表的头结点，不难分析出反转后链表的头结点是原始链表的尾结点，即 $next$ 为空指针的结点。下面给出非递归方法实现链表的反转的实现代码。

```
public void ReverseIteratively(Node head){
    Node pReversedHead = head;
    Node pNode = head;
    Node pPrev = null;
    while(pNode != null){
        Node pNext = pNode.next;
        if(pNext == null)
            pReversedHead = pNode;
        pNode.next = pPrev;
        pPrev = pNode;
    }
}
```

```

        pNode = pNode.next;
    }
    this.head = pReversedHead;
}

```

8.1.5 如何从尾到头输出单链表

从头到尾输出单链表比较简单，通过借鉴的想法，要想解决本问题，很自然地想把链表中链接结点的指针反转过来，改变链表的方向，然后就可以从尾到头输出了，但该方法需要额外的操作，是否还有更好的方法呢？答案是有。

接下来的想法是从头到尾遍历链表，每经过一个结点，把该结点放到一个栈中。当遍历完整个链表后，再从栈顶开始输出结点的值，此时输出的结点的顺序已经反转过来了。该方法虽然没有只需要遍历一遍链表，但是需要维护一个额外的栈空间，实现起来会比较麻烦。

是否还能有更高效的方法？既然想到了栈来实现这个函数，而递归本质上就是一个栈结构，于是很自然地又想到了递归来实现。要实现反过来输出链表，每访问到一个结点，先递归输出它后面的结点，再输出该结点自身，这样链表的输出结果就反过来了。

具体实现代码如下：

```

public void printListReversely(Node pListHead) {
    if(pListHead != null) {
        printListReversely(pListHead.next);
        System.out.println(pListHead.data);
    }
}

```

8.1.6 如何寻找单链表的中间结点

如何寻找单链表的中间结点？最容易想到的思路是先求解单链表的长度 $length$ ，然后遍历 $length/2$ 的距离即可查找到单链表的中间结点，但是此种方法需要遍历两次链表，即第一次遍历求解单链表的长度，第二次遍历根据索引获取中间结点。

如果是双向链表，可以首尾并行，利用两个指针一个从头到尾遍历，一个从尾到头遍历，当两个指针相遇时，就找到了中间元素。以此思想为基础，如果是单链表，也可以采用双指针的方式来实现中间结点的快速查找。

具体而言，第一步，有两个指针同时从头开始遍历；第二步，一个快指针一次走两步，一个慢指针一次走一步；第三步，快指针先到链表尾部，而慢指针则恰好到达链表中部（快指针到链表尾部时，当链表长度为奇数时，慢指针指向的即是链表中间指针，当链表长度为偶数时，慢指针指向的结点和慢指针指向结点的下一个结点都是链表的中间结点）。

具体实现代码如下：

```

public Node SearchMid(Node head) {
    Node p = this.head;
    Node q = this.head;
    while(p != null && p.next != null && p.next.next != null) {
        p = p.next.next;
        q = q.next;
    }
}

```

```

        return q;
    }
}

```

8.1.7 如何检测一个链表是否有环

定义两个指针 fast 与 slow，其中，fast 是快指针，slow 是慢指针，二者的初始值都指向链表头，slow 每次前进一步，fast 每次前进两步，两个指针同时向前移动，快指针每移动一次都要跟慢指针比较，直到当快指针等于慢指针为止，就证明这个链表是带环的单向链表，否则，证明这个链表是不带环的循环链表（fast 先行到达尾部为 NULL，则为无环链表）。具体实现代码如下：

```

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

```

上述方法只能用来判断链表是否有环，那么如何找到环的入口点呢？如果单链表有环，按照上述思路，当走得快的指针 fast 与走得慢的指针 slow 相遇时，slow 指针肯定没有遍历完链表，而 fast 指针已经在环内循环了 n 圈（ $n \geq 1$ ）。假设 slow 指针走了 s 步，则 fast 指针走了 2s 步（fast 步数还等于 s 加上在环上多转的 n 圈），设环长为 r，则满足如下关系表达式：

$$2s = s + nr$$

$$s = nr$$

设整个链表长 L，入口环与相遇点距离为 x，起点到环入口点的距离为 a，则满足如下关系表达式：

$$a + x = nr$$

$$a + x = (n - 1)r + r = (n - 1)r + L - a$$

$$a = (n - 1)r + (L - a - x)$$

($L - a - x$) 为相遇点到环入口点的距离，从链表头到环入口点等于 $(n - 1)$ 循环内环 + 相遇点到环入口点，于是在链表头与相遇点分别设一个指针，每次各走一步，两个指针必定相遇，且相遇第一点为环入口点。

具体实现代码如下：

```

public Node FindLoopPort(Node head) {
    Node slow = head, fast = head;
    while(fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }
}

```

```

        if( slow == fast) break;
    }
    if( fast == null || fast. next == null)
        return null;
    slow = head;
    while( slow != fast) {
        slow = slow. next;
        fast = fast. next;
    }
    return slow;
}

```

8.1.8 如何在不知道头指针的情况下删除指定结点

可以分为两种情况来讨论：

① 若待删除的结点为链表尾结点，则无法删除，因为删除后无法使其前驱结点的 `next` 指针置为空；

② 若待删除的结点不是尾结点，则可以通过交换这个结点与其后继结点的值，然后删除后继结点。

具体实现代码如下：

```

public boolean deleteNode(Node n) {
    if(n == null || n. next == null)
        return false;
    int tmp = n. data;
    n. data = n. next. data;
    n. next. data = tmp;
    n. next = n. next. next;
    return true;
}

```

8.1.9 如何判断两个链表是否相交

如果两个链表相交，那么它们一定有着相同的尾结点，因此实现思路为：分别遍历两个链表，记录它们的尾结点，如果它们的尾结点相同，那么这两个链表相交，否则不相交。具体实现代码如下：

```

public boolean isIntersect(Node h1, Node h2) {
    if(h1 == null || h2 == null)
        return false;
    Node tail1 = h1;
    //找到链表 h1 的最后一个结点
    while(tail1. next != null)
        tail1 = tail1. next;
    Node tail2 = h2;
    //找到链表 h2 的最后一个结点
    while(tail2. next != null) {
        tail2 = tail2. next;
    }
}

```

```

    }
    return tail1 == tail2;
}

```

由于这个算法只需要分别遍历一次两个链表，因此算法的时间复杂度为 $O(\text{len1} + \text{len2})$ ，其中 len1 和 len2 分别代表两个链表的长度。

引申：如果两个链表相交，如何找到它们相交的第一个结点？

首先分别计算两个链表 head1 、 head2 的长度 len1 和 len2 （假设 $\text{len1} > \text{len2}$ ），接着先对链表 head1 遍历 $(\text{len1} - \text{len2})$ 个结点到结点 p ，此时结点 p 与 head2 到它们相交的结点的距离相同，此时同时遍历两个链表，直到遇到相同的结点为止，这个结点就是它们相交的结点。需要注意的是，在查找相交的第一个结点前，需要先判断两个链表是否相交，只有在相交的情况下才有必要去找它们的交点，否则根本就没有必要了。

程序代码如下：

```

public static Node getFirstMeetNode(Node h1, Node h2) {
    if (h1 == null || h2 == null)
        return null;
    Node tail1 = h1;
    int len1 = 1;
    //找到链表 h1 的最后一个结点
    while (tail1.next != null) {
        tail1 = tail1.next;
        len1++;
    }
    Node tail2 = h2;
    int len2 = 1;
    //找到链表 h2 的最后一个结点
    while (tail2.next != null) {
        tail2 = tail2.next;
        len2++;
    }
    //两链表不相交
    if (tail1 != tail2) {
        return null;
    }
    Node t1 = h1;
    Node t2 = h2;
    //找出较长的链表,先遍历
    if (len1 > len2) {
        int d = len1 - len2;
        while (d != 0) {
            t1 = t1.next;
            d--;
        }
    }
    else {
        int d = len2 - len1;
        while (d != 0) {

```

```

        t2 = t2. next;
        d -- ;
    }
}
while( t1 != t2 ) {
    t1 = t1. next;
    t2 = t2. next;
}
return t1;
}

```

同理，由于这个算法也只需要分别遍历一次两个链表，因此算法的时间复杂度为 $O(\text{len1} + \text{len2})$ ，其中 len1 和 len2 分别代表两个链表的长度。当然，在具体实现时可以使用前面已经实现的方法来判断两个链表是否相交，也可以利用前面已经实现的方法来分别计算两个链表的长度。但这种方法也存在着一个缺点：需要对每个链表遍历两遍。第一遍用来判断链表是否相交，第二遍计算链表的长度，因此效率会比上例中的实现方式低。其优点是代码简洁，可用性强。

8.2 栈与队列

8.2.1 栈与队列有哪些区别

栈与队列是在程序设计中被广泛使用的两种重要的线性数据结构，都是在一个特定范围的存储单元中存储的数据。这些数据都可以重新被取出使用，与线性表相比，它们的插入和删除操作受到更多的约束和限定，故又称为限定性的线性表结构。不同的是，栈就像一个很窄的桶，先存进去的数据只能最后被取出来，是 LIFO (Last In First Out, 后进先出)，它将进出顺序逆序，即先进后出，后进先出，栈结构示意图如图 8-4 所示。队列则像人们日常排队买东西的“队列”，先排队的人先买，后排队的人后买，是 FIFO (First In First Out, 先进先出) 的，它保持进出顺序一致，即先进先出，后进后出，队列结构示意图如图 8-5 所示。

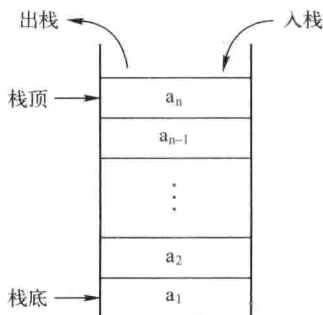


图 8-4 栈结构示意图

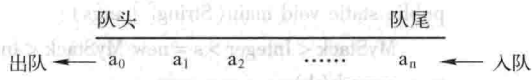


图 8-5 队列结构示意图

需要注意的是，有时在数据结构中还有可能出现按照大小排队或按照一定条件排队的队列，这时的队列属于特殊队列，就不一定按照“先进先出”的原则读取数据了。

8.2.2 如何实现栈

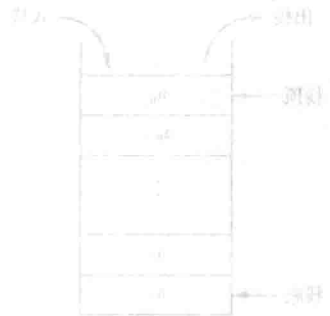
可以采用数组与链表这两种方法来实现栈。

下面给出用数组实现栈的代码：

```

import java.util. Arrays;
public class MyStack < E > {
    private Object[] stack;
    private int size;//数组中存储元素的个数
    public MyStack() {
        stack = new Object[10]; //初始长度为 10
    }
    //判断堆栈是否为空
    public boolean isEmpty() {
        return size == 0;
    }
    public E peek() {
        if (isEmpty()) {
            return null;
        }
        return (E) stack[ size - 1 ];
    }
    public E pop() {
        E e = peek();
        stack[ size - 1 ] = null;
        size --;
        return e;
    }
    public E push(E item) {
        ensureCapacity( size + 1 ); //检查容量
        stack[ size ++ ] = item;
        return item;
    }
    //判断数组器是否已满,若已满,则扩充数组空间
    private void ensureCapacity(int size) {
        int len = stack. length;
        if (size > len) { //数组已满
            int newLen = 10; //每次数组扩充的容量
            stack = Arrays. copyOf( stack, newLen );
        }
    }
    public static void main( String[] args ) {
        MyStack < Integer > s = new MyStack < Integer > ();
        s. push( 1 );
        s. push( 2 );
        System. out. println( " 栈中元素个数:" + s. size );
        System. out. println( " 栈顶元素为:" + s. pop();
        System. out. println( " 栈顶元素为:" + s. pop();
    }
}

```



下面给出采用链表的方式实现栈的代码。

```

class Node < E > {
    Node < E > next = null;
    E data;
    public Node(E data) { this. data = data; }
}

public class Stack < E > {
    Node < E > top = null;
    public boolean isEmpty() {
        return top == null;
    }
    public void push(E data) {
        Node < E > newNode = new Node < E > ( data );
        newNode. next = top;
        top = newNode;
    }
    public E pop() {
        if( this. isEmpty() )
            return null;
        E data = top. data;
        top = top. next;
        return data;
    }
    public E peek() {
        if( isEmpty() ) {
            return null;
        }
        return top. data;
    }
}

```

需要注意的是，上述的实现不是线程安全的。若要实现线程安全的栈，则需要对入栈和出栈等操作进行同步，在此就不详细介绍了。

8.2.3 如何用 $O(1)$ 的时间复杂度求栈中最小元素

由于栈具有后进先出的特点，因此 push 和 pop 只需要对栈顶元素进行操作。如果使用上述的实现方式，只能访问到栈顶的元素，而无法得到栈中最小的元素。当然，可以用另外一个变量来记录栈底的位置，通过遍历栈中的所有元素找出最小值，但是这种方法的时间复杂度为 $O(n)$ ，那么如何才能用 $O(1)$ 的时间复杂度求出栈中最小的元素呢？在算法设计中，经常会采用空间来换取时间的方式来提高时间复杂度，也就是说，采用额外的存储空间来降低操作的时间复杂度。具体来讲就是在实现时使用两个栈结构，一个栈用来存储数据，另一个栈用来存储栈的最小元素。其实现思路如下：如果当前入栈的元素比原来栈中的最小值还小，则把这个值压入保存最小元素的栈中；在出栈时，如果当前出栈的元素恰好为当前栈中的最小值，保存最小值的栈顶元素也出栈，使得当前最小值变为其入栈之前的那个最小值。为了简单起见，可以在栈中保存 Integer 类型，采用前面用链表方式实现的栈，实现代码如下：

```

public class MyStack1 {
    MyStack < Integer > elem;

```

```

MyStack < Integer > min;
public MyStack1 () {
    elem = new MyStack < Integer > ();
    min = new MyStack < Integer > ();
}
public void push(int data) {
    elem.push(data);
    if (min.isEmpty())
        min.push(data);
    else {
        if (data < min.peek())
            min.push(data);
    }
}
public int pop() {
    int topData = elem.peek();
    elem.pop();
    if (topData == this.min())
        min.pop();
    return topData;
}
public int min() {
    if (min.isEmpty())
        return Integer.MAX_VALUE;
    else
        return min.peek();
}
}

```

8.2.4 如何实现队列

与栈类似，队列也可以采用数组和链表两种方式来实现。下面给出采用链表方式实现队列的代码：

```

class Node < E > {
    Node < E > next = null;
    E data;
    public Node(E data) { this.data = data; }
}
public class MyQueue < E > {
    private Node < E > head = null;
    private Node < E > tail = null;
    public boolean isEmpty() {
        return head == null;
    }
    public void put(E data) {
        Node < E > newNode = new Node < E > (data);
        if (head == null && tail == null) //队列为空
            head = tail = newNode;
    }
}

```

```

else {
    tail.next = newNode;
    tail = newNode;
}
}
public E pop() {
    if(this.isEmpty())
        return null;
    E data = head.data;
    head = head.next;
    return data;
}
public int size() {
    Node < E > tmp = head;
    int n = 0;
    while( tmp != null) {
        n ++;
        tmp = tmp.next;
    }
    return n;
}
public static void main(String[] args) {
    MyQueue < Integer > q = new MyQueue < Integer > ();
    q.put(1);
    q.put(2);
    q.put(3);
    System.out.println(" 队列长度为:" + q.size());
    System.out.println(" 队列首元素:" + q.pop());
    System.out.println(" 队列首元素:" + q.pop());
}
}

```

运行结果为:

```

队列长度为:3
队列首元素:1
队列首元素:2

```

下面介绍数组实现队列的方式，为了实现多线程安全，增加了对队列操作的同步，实现代码如下:

```

public class MyQueue < E > {
    private LinkedList < E > list = new LinkedList < E > ();
    private int size = 0;
    public synchronized void put(E e) {
        list.addLast(e);
        size ++;
    }
    public synchronized E pop() {
        size --;

```

```

        return list.removeFirst();
    }

    public synchronized boolean empty() {
        return size == 0;
    }

    public synchronized int size() {
        return size;
    }
}

```

8.2.5 如何用两个栈模拟队列操作

假设使用栈 A 与栈 B 模拟队列 Q，A 为插入栈，B 为弹出栈，以实现队列 Q。再假设 A 和 B 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。要入队列，入栈 A 即可，而出队列则需要分两种情况考虑：

- 1) 若栈 B 不为空，则直接弹出栈 B 的数据。
- 2) 若栈 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。

以上情况可以利用前面介绍的栈来实现，也可以采用 Java 类库提供的 Stack 来实现，下面代码是采用 Java 内置的 Stack 来实现的：

```

public class MyQueue < E > {
    private Stack < E > s1 = new Stack < E > ();
    private Stack < E > s2 = new Stack < E > ();
    public synchronized void put(E e) {
        s1.push(e);
    }
    public synchronized E pop() {
        if(s2.isEmpty())
            while(!s1.isEmpty())
                s2.push(s1.pop());
        return s2.pop();
    }
    public synchronized boolean empty() {
        return s1.isEmpty() && s2.isEmpty();
    }
    public static void main(String[] args) {
        MyQueue < Integer > q = new MyQueue < Integer > ();
        q.put(1);
        q.put(2);
        System.out.println("队列首元素:" + q.pop());
        System.out.println("队列首元素:" + q.pop());
    }
}

```

程序运行结果为：

```

队列首元素:1
队列首元素:2

```

引申：如何使用两个队列实现栈？

假设使用队列 q1 与队列 q2 模拟栈 S，q1 为入队列，q2 为出队列。

实现思路：可以认为队列 q1 提供压栈的功能，队列 q2 提供弹栈的功能。

要压栈，入队列 q1 即可，而当弹栈时，出队列则需要分两种情况考虑：

1) 若队列 q1 中只有一个元素，则让 q1 中的元素出队列并输出即可。

2) 若队列 q1 中不只一个元素，则队列 q1 中的所有元素出队列，入队列 q2，最后一个元素不入队列 B，输出该元素，然后将队列 q2 所有元素入队列 q1。

8.3 排序

排序问题一直是计算机技术研究的重要问题，排序算法的好坏直接影响程序的执行速度和辅助存储空间的占有量，所以，各大 IT 企业在笔试面试中也经常出现有关排序的题目，本节将详细分析常见的各种排序算法，并从时间复杂度、空间复杂度、适用情况等多个方面对它们进行综合比较。

8.3.1 如何进行选择排序

选择排序是一种简单直观的排序算法，其基本原理如下：对于给定的一组记录，经过第一轮比较后得到最小的记录，然后将该记录与第一个记录的位置进行交换；接着对不包括第一个记录以外的其他记录进行第二轮比较，得到最小的记录并与第二个记录进行位置交换；重复该过程，直到进行比较的记录只有一个时为止。以数组 {38, 65, 97, 76, 13, 27, 49} 为例，选择排序的具体步骤如下：

第一趟排序后：13[65 97 76 38 27 49]

第二趟排序后：13 27[97 76 38 65 49]

第三趟排序后：13 27 38[76 97 65 49]

第四趟排序后：13 27 38 49[97 65 76]

第五趟排序后：13 27 38 49 65[97 76]

第六趟排序后：13 27 38 49 65 76[97]

最后排序结果：13 27 38 49 65 76 97

程序示例如下：

```
public class TestSort {
    public static void selectSort(int[] a) {
        int i;
        int j;
        int temp = 0;
        int flag = 0;
        int n = a.length;
        for (i = 0; i < n; i++) {
            temp = a[i];
            flag = i;
            for (j = i + 1; j < n; j++) {
                if (a[j] < temp) {
                    temp = a[j];
                    flag = j;
                }
            }
            a[i] = a[flag];
            a[flag] = temp;
        }
    }
}
```



```

    }
    if(flag != i) {
        a[flag] = a[i];
        a[i] = temp;
    }
}

public static void main(String[] args) {
    int i=0;
    int a[] = {5,4,9,8,7,6,0,1,3,2};
    selectSort(a);
    for(i=0;i < a.length;i++)
        System.out.print(a[i] + " ");
    System.out.println("\n");
}
}

```

程序运行结果为：

```
0 1 2 3 4 5 6 7 8 9
```

8.3.2 如何进行插入排序

对于给定的一组记录，初始时假设第一个记录自成一个有序序列，其余记录为无序序列。接着从第二个记录开始，按照记录的大小依次将当前处理的记录插入到其之前的有序序列中，直至最后一个记录插入到有序序列中为止。仍以数组 {38, 65, 97, 76, 13, 27, 49} 为例，直接插入排序的具体步骤如下。

第一步插入 38 以后：[38]65 97 76 13 27 49

第二步插入 65 以后：[38 65]97 76 13 27 49

第三步插入 97 以后：[38 65 97]76 13 27 49

第四步插入 76 以后：[38 65 76 97]13 27 49

第五步插入 13 以后：[13 38 65 76 97]27 49

第六步插入 27 以后：[13 27 38 65 76 97]49

第七步插入 49 以后：[13 27 38 49 65 76 97]

程序示例如下：

```

public class TestSort {
    public static void insertSort(int[] a) {
        if(a != null) {
            for(int i = 1; i < a.length; i++) {
                int temp = a[i], j = i;
                if(a[j-1] > temp) {
                    while(j >= 1 && a[j-1] > temp) {
                        a[j] = a[j-1];
                        j--;
                    }
                }
            }
        }
    }
}

```

```

int i,j;
int len = array.length;
int tmp;
for(i=0;i < len - 1; ++i)
    for(j = len - 1; j > i; --j)
        if(array[j] < array[j - 1]) {
            tmp = array[j];
            array[j] = array[j - 1];
            array[j - 1] = tmp;
        }
}

public static void main(String[] args) {
    int a[] = {5,4,9,8,7,6,0,1,3,2};
    BubbleSort(a);
    for(int i=0;i < a.length;i++){
        System.out.print(a[i] + " ");
    }
}

```

程序输出为:

0 1 2 3 4 5 6 7 8 9

8.3.4 如何进行归并排序

归并排序是利用递归与分治技术将数据序列划分成为越来越小的半子表，再对半子表排序，最后再用递归方法将排好序的半子表合并成为越来越大的有序序列。归并排序中，“归”代表的是递归的意思，即递归的将数组折半的分离为单个数组，例如数组： $[2,6,1,0]$ ，会先折半，分为 $[2,6]$ 和 $[1,0]$ 两个子数组，然后再折半将数组分离，分为 $[2]$ 、 $[6]$ 和 $[1]$ 、 $[0]$ 。“并”就是将分开的数据按照从小到大或者从大到小的顺序在放到一个数组中。如上面的 $[2]$ 、 $[6]$ 合并到一个数组中是 $[2,6]$ ， $[1]$ 、 $[0]$ 合并到一个数组中是 $[0,1]$ ，然后再将 $[2,6]$ 和 $[0,1]$ 合并到一个数组中即为 $[0,1,2,6]$ 。

归并排序算法的原理如下：对于给定的一组记录（假设共有 n 个记录），首先将每两个相邻的长度为 1 的子序列进行归并，得到 $n/2$ （向上取整）个长度为 2 或 1 的有序子序列，再将其两两归并，反复执行此过程，直到得到一个有序序列。

所以，归并排序的关键就是两步：第一步，划分半子表；第二步，合并半子表。以数组 $\{49,38,65,97,76,13,27\}$ 为例，归并排序的具体步骤如下：

初始关键字： $[49] [38] [65] [97] [76] [13] [27]$

一趟归并后： $[38 49] [65 97] [13 76] [27]$

二趟归并后： $[38 49 65 97] [13 27 76]$

三趟归并后： $[13 27 38 49 65 76 97]$

程序示例如下：

```

public class Test {
    public static void Merge(int array[], int p, int q, int r) {

```

```

int i,j,k,n1,n2;
n1 = q - p + 1;
n2 = r - q;
int[] L = new int[n1];
int[] R = new int[n2];
for(i=0,k=p;i<n1;i++,k++)
    L[i] = array[k];
for(i=0,k=q+1;i<n2;i++,k++)
    R[i] = array[k];
for(k=p,i=0,j=0;i<n1 && j<n2;k++) {
    if(L[i] > R[j]) {
        array[k] = L[i];
        i++;
    }
    else {
        array[k] = R[j];
        j++;
    }
}
if(i < n1) {
    for(j=i;j<n1;j++,k++)
        array[k] = L[j];
}
if(j < n2) {
    for(i=j;i<n2;i++,k++)
        array[k] = R[i];
}
}
public static void MergeSort(int array[],int p,int r) {
    if(p < r) {
        int q = (p+r)/2;
        MergeSort(array,p,q);
        MergeSort(array,q+1,r);
        Merge(array,p,q,r);
    }
}
public static void main(String[] args) {
    int i=0;
    int a[] = {5,4,9,8,7,6,0,1,3,2};
    int len = a.length;
    MergeSort(a,0,len-1);
    for(i=0;i<len;i++)
        System.out.print(a[i] + " ");
}
}

```

程序运行结果为:

9 8 7 6 5 4 3 2 1 0

二路归并排序的过程需要进行 $\log n$ 趟。每一趟归并排序的操作，就是将两个有序子序列进行归并，而每一对有序子序列归并时，记录的比较次数均小于等于记录的移动次数，记录移动的个数均等于文件中记录的个数 n ，即每一趟归并的时间复杂度为 $O(n)$ 。因此，二路归并排序的时间复杂度为 $O(n \log n)$ 。

8.3.5 如何进行快速排序

快速排序是一种非常高效的排序算法，它采用“分而治之”的思想，把大的拆分为小的，小的再拆分为更小的。其原理如下：对于一组给定的记录，通过一趟排序后，将原序列分为两部分，其中前一部分的所有记录均比后一部分的所有记录小，然后再依次对前后两部分的记录进行快速排序，递归该过程，直到序列中的所有记录均有序为止。

具体而言，其算法步骤如下：

1) 分解。将输入的序列 $\text{array}[m \cdots n]$ 划分成两个非空子序列 $\text{array}[m \cdots k]$ 和 $\text{array}[k + 1 \cdots n]$ ，使 $\text{array}[m \cdots k]$ 中任一元素的值不大于 $\text{array}[k + 1 \cdots n]$ 中任一元素的值。

2) 递归求解。通过递归调用快速排序算法分别对 $\text{array}[m \cdots k]$ 和 $\text{array}[k + 1 \cdots n]$ 进行排序。

3) 合并。由于对分解出的两个子序列的排序是就地进行的，所以在 $\text{array}[m \cdots k]$ 和 $\text{array}[k + 1 \cdots n]$ 都排好序后不需要执行任何计算 $\text{array}[m \cdots n]$ 就已排好序。

以数组 $\{38, 65, 97, 76, 13, 27, 49\}$ 为例。

第一趟排序过程如下：

初始化关键字 $[49 \ 38 \ 65 \ 97 \ 76 \ 13 \ 27 \ 49]$

第一次交换后： $[27 \ 38 \ 65 \ 97 \ 76 \ 13 \ 49 \ 49]$

第二次交换后： $[27 \ 38 \ 49 \ 97 \ 76 \ 13 \ 65 \ 49]$

j 向左扫描，位置不变，第三次交换后： $[27 \ 38 \ 13 \ 97 \ 76 \ 49 \ 65 \ 49]$

i 向右扫描，位置不变，第四次交换后： $[27 \ 38 \ 13 \ 49 \ 76 \ 97 \ 65 \ 49]$

j 向左扫描 $[27 \ 38 \ 13 \ 49 \ 76 \ 97 \ 65 \ 49]$

整个排序过程如下所示。

初始化关键字 $[49 \ 38 \ 65 \ 97 \ 76 \ 13 \ 27 \ 49]$

一趟排序之后： $[27 \ 38 \ 13] \ 49 \ [76 \ 97 \ 65 \ 49]$

二趟排序之后： $[13] \ 27 \ [38] \ 49 \ [49 \ 65] \ 76 \ [97]$

三趟排序之后： $13 \ 27 \ 38 \ 49 \ 49 \ [65] \ 76 \ 97$

最后的排序结果： $13 \ 27 \ 38 \ 49 \ 49 \ 65 \ 76 \ 97$

程序示例如下：

```
public class Test {
    public static void sort(int array[], int low, int high) {
        int i, j;
        int index;
        if (low >= high)
            return;
        i = low;
        j = high;
        index = array[i];
        while (i < j) {
```

```

while(i < j && array[j] >= index)
    j--;
if(i < j)
    array[i++] = array[j];
while(i < j && array[i] < index)
    i++;
if(i < j)
    array[j--] = array[i];
}
array[i] = index;
sort(array, low, i - 1);
sort(array, i + 1, high);
}

public static void quickSort(int array[]) {
    sort(array, 0, array.length - 1);
}

public static void main(String[] args) {
    int i = 0;
    int a[] = {5, 4, 9, 8, 7, 6, 0, 1, 3, 2};
    int len = a.length;
    quickSort(a);
    for(i = 0; i < len; i++)
        System.out.print(a[i] + " ");
}

```

程序输出为:

0 1 2 3 4 5 6 7 8 9

当初始的序列整体或局部有序时,快速排序的性能将会下降,此时,快速排序将退化为冒泡排序。

快速排序的相关特点如下:

1) 最坏时间复杂度。最坏情况是指每次区间划分的结果都是基准关键字的左边(或右边)序列为空,而另一边区间中的记录项仅比排序前少了一项,即选择的基准关键字是待排序的所有记录中最小或者最大的,例如,如果选取第一个记录为基准关键字,当初始序列按递增顺序排列时,每次选择的基准关键字都是所有记录中的最小者,这时记录与基准关键字的比较次数会增多。因此,在这种情况下,需要进行 $(n-1)$ 次区间划分。对于第 k ($0 < k < n$)次区间划分,划分前的序列长度为 $(n-k+1)$,需要进行 $(n-k)$ 次记录的比较。因此,当 k 从1到 $(n-1)$ 时,进行的比较次数总共为 $n(n-1)/2$,所以,在最坏情况下快速排序的时间复杂度为 $O(n^2)$ 。

2) 最好时间复杂度。最好情况是指每次区间划分的结果都是基准关键字左右两边的序列长度相等或者相差为1,即选择的基准关键字为待排序的记录中的中间值。此时,进行的比较次数总共为 $n \log n$,所以,在最好情况下快速排序的时间复杂度为 $O(n \log n)$ 。

3) 平均时间复杂度。快速排序的平均时间复杂度为 $O(n \log n)$ 。虽然快速排序在最坏情况下的时间复杂度为 $O(n^2)$,但是在所有平均时间复杂度为 $O(n \log n)$ 的算法中,快速排序的平均性能是最好的。

4) 空间复杂度。快速排序的过程中需要一个栈空间来实现递归。当每次对区间的划分都比较均匀时（即最好情况），递归树的最大深度为 $\lceil \log n \rceil + 1$ （ $\log n$ 为向上取整）；当每次区间划分都使得有一边的序列长度为 0 时（即最差情况），递归树的最大深度为 n 。在每轮排序结束后比较基准关键字左右的记录个数，对记录多的一边先进行排序，此时，栈的最大深度可降为 $\log n$ 。因此，快速排序的平均空间复杂度为 $O(\log n)$ 。

5) 基准关键字的选取。基准关键字的选取是决定快速排序算法性能的关键。常用基准关键字的选取方式如下：

① 三者取中。三者取中是指在当前序列中，将其首、尾和中间位置上的记录进行比较，选择三者的中值作为基准关键字，在划分开始前交换序列中的第一个记录与基准关键字的位置。

② 取随机数。取 $left$ （左边）和 $right$ （右边）之间的一个随机数 m ($left \leq m \leq right$)，用 $n[m]$ 作为基准关键字。这种方法使得 $n[left]$ 和 $n[right]$ 之间的记录是随机分布的，采用此方法得到的快速排序一般称为随机的快速排序。

需要注意一个问题，就是快速排序与归并排序的区别与联系。快速排序与归并排序的原理都是基于“分而治之”思想，即首先把待排序的元素分成两组，然后分别对这两组排序，最后把两组结果合并起来。

而它们的不同点在于，进行的分组策略不同，后面的合并策略也不同。归并排序的分组策略是假设待排序的元素存放在数组中，那么其把数组前面一半元素作为一组，后面一半作为另外一组。而快速排序则是根据元素的值来分组，即大于某个值的元素放在一组，而小于的放在另外一组，该值称为基准。所以，对整个排序过程而言，基准值的挑选非常重要，如果选择不合适，太大或太小，那么所有元素都分在一组了。总的来说，快速和归并排序，如果分组策略越简单，则后面的合并策略就越复杂，因为快速排序在分组时，已经根据元素大小来分组了，而合并时，只需把两个分组合并起来就行了，归并排序则需要对两个有序的数组根据大小合并。

8.3.6 如何进行希尔排序

希尔排序也被称为“缩小增量排序”，其基本原理如下：先将待排序的数组元素分成多个子序列，使得每个子序列的元素个数相对较少，然后对各个子序列分别进行直接插入排序，待整个待排序序列“基本有序后”，最后再对所有元素进行一次直接插入排序。

希尔排序的具体步骤如下：

1) 选择一个步长序列 t_1, t_2, \dots, t_k ，满足 $t_i > t_j (i < j), t_k = 1$ 。

2) 按步长序列个数 k ，对待排序序列进行 k 趟排序。

3) 每趟排序，根据对应的步长 t_i ，将待排序列分割成 t_i 个子序列，分别对各个子序列进行直接插入排序。

注意：当步长因子为 1 时，所有元素作为一个序列来处理，其长度为 n 。以数组 {26, 53, 67, 48, 57, 13, 48, 32, 60, 50}，步长序列为 {5, 3, 1} 为例。具体步骤如下：

初始关键字：26 53 67 48 57 13 48 32 60 50



第 1 趟：26 48 32 48 50 26 53 67 60 57

第 2 趟：13 48 26 48 50 32 53 67 60 57

第 3 趟：13 26 32 48 48 50 53 57 60 67

程序示例如下：

```

public class Test {
    public static void shellSort(int array[]) {
        int length = array.length;
        int i, j;
        int h;
        int temp;
        for (h = length/2; h > 0; h = h/2) {
            for (i = h; i < length; i++) {
                temp = array[i];
                for (j = i - h; j >= 0; j -= h) {
                    if (temp < array[j]) {
                        array[j + h] = array[j];
                    }
                    else
                        break;
                }
                array[j + h] = temp;
            }
        }
    }

    public static void main(String[] args) {
        int i = 0;
        int a[] = {5, 4, 9, 8, 7, 6, 0, 1, 3, 2};
        int len = a.length;
        shellSort(a);
        for (i = 0; i < len; i++)
            System.out.print(a[i] + " ");
    }
}

```

程序运行结果为：

0 1 2 3 4 5 6 7 8 9

希尔排序的关键并不是随便地分组后各自排序，而是将相隔某个“增量”的记录组成一个子序列，实现跳跃式移动，使得排序的效率提高。

8.3.7 如何进行堆排序

堆是一种特殊的树形数据结构，其每个结点都有一个值，通常提到的堆都是指一棵完全二叉树，根结点的值小于（或大于）两个子结点的值，同时，根结点的两个子树也分别是一个堆。

堆排序是一树形选择排序，在排序过程中，将 $R[1 \cdots n]$ 看作一颗完全二叉树的顺序存储结构，利用完全二叉树中父结点和子结点之间的内在关系来选择最小的元素。

堆一般分为大顶堆和小顶堆两种不同的类型。对于给定 n 个记录的序列 $(r(1), r(2), \dots, r(n))$ ，当且仅当满足条件 $(r(i) \geq r(2i), i = 1, 2, \dots, n)$ 时称之为大顶堆，此时，堆顶元素比为最大值。对于给定 n 个记录的序列 $(r(1), r(2), \dots, r(n))$ ，当且仅当满足条件 $(r(i) \leq$

这个特点时，就可以用移位操作来代替乘法操作，从而提高效率。

示例如下：

```
public class Muti {
    public static int powerN(int m,int n) { //m 乘以 2 的 n 次方
        for(int i=0;i<n;i++)
            m = m << 1;
        return m;
    }
    public static void main(String[] args) {
        System.out.println("3 乘以 8 = " + powerN(3,3));
        System.out.println("3 乘以 16 = " + powerN(3,4));
    }
}
```

程序运行结果为：

```
3 乘以 8 = 24
3 乘以 16 = 48
```

8.4.2 如何判断一个数是否为 2 的 n 次方

2 的 n 次方可以表示为： 2^0 ， 2^1 ， 2^2 ， \dots ， 2^n 。最直观的思想是用 1 做移位操作，然后判断移位后的值是否与给定的数相等，具体实现代码如下：

```
public class Test {
    public static boolean isPower(int n) {
        if (n < 1) return false;
        int i = 1;
        while (i <= n) {
            if (i == n)
                return true;
            i <<= 1;
        }
        return false;
    }
    public static void main(String[] args) {
        System.out.println(isPower(4));
        System.out.println(isPower(6));
    }
}
```

程序运行结果为：

```
true
false
```

上述算法的时间复杂度为 $O(\log n)$ 。那么是否存在效率更高的算法呢？通过对 2^0 ， 2^1 ， 2^2 ， \dots ， 2^n 进行分析，发现这些数字的二进制形式分别为：1，10，100， \dots 。从二进制的表示可以看出，如果一个数是 2 的 n 次方，那么这个数对应的二进制表示中只有一位是 1，其余位都为 0。因此，判断一个数是否为 2 的 n 次方可以转换为这个数对应的二进制表示中是否只

有一位为1。如果一个数的二进制表示只有一位是1，例如 $\text{num} = 00010000$ ，那么 $\text{num} - 1$ 的二进制表示为 $\text{num} - 1 = 00001111$ ，由于 num 与 $\text{num} - 1$ 二进制表示中每一位都不相同，因此 $\text{num} \& (\text{num} - 1)$ 的运算结果为0，可以利用这种方法来判断一个数是否为2的n次方。具体实现代码如下：

```
public class Test {
    public static boolean isPower(int n) {
        if (n < 1) return false;
        int m = n & (n - 1);
        return m == 0;
    }
    public static void main(String[] args) {
        System.out.println(isPower(4));
        System.out.println(isPower(6));
    }
}
```

8.4.3 如何求二进制数中1的个数

问题描述：给定一个整数，输出这个整数二进制表示中1的个数，例如，给定整数7，其二进制表示为111，因此输出结果为3。

该问题可以采用位操作来完成。具体思路如下：首先，判断这个数的最后一位是否为1，如果为1，则计数器加1，然后，通过右移丢弃掉最后一位。循环执行该操作直到这个数等于0为止。在判断二进制表示的最后一位是否为1时，可以采用与运算来达到这个目的。具体实现代码如下：

```
public class Test {
    public static int countOne(int n) {
        int count = 0; // 用来计数
        while (n > 0) {
            if ((n & 1) == 1) // 判断最后一位是否为1
                count++;
            n >>= 1; // 移位
        }
        return count;
    }
    public static void main(String[] args) {
        System.out.println(countOne(7));
        System.out.println(countOne(8));
    }
}
```

程序运行结果为：

```
3
1
```

以上这个算法的时间复杂度为 $O(n)$ ，其中 n 代表二进制数的位数。由于题目要求求出1的个数，此时可以只考虑1的个数，把二进制表示中的每个1看作独立的个体，利用上一节中

复杂度为 $O(n \log n)$ 。

有没有更好的方法能降低时间复杂度了？答案是有，可以只通过一遍扫描数组即可找出数组中第二大的数，即通过设置两个变量来进行判断。先定义两个变量：一个变量用来存储数组的最大数，初始值为数组首元素，另一个变量用来存储数组元素的第二大数，初始值为最小负整数，然后遍历数组元素，如果数组元素的值比最大数变量的值大，则将第二大变量的值更新为最大数变量的值，最大数变量的值更新为该数组元素的值，若数组元素的值比最大数的值小，则判断该数组元素的值是否比第二大数的值大；若数组元素的值比最大数的值大，则更新第二大数的值为该数组元素的值。

示例如下：

```
public class SecondMax {
    public static int FindSecMax(int data[]) {
        int count = data.length;
        int maxnumber = data[0];
        int sec_max = Integer.MIN_VALUE;
        for(int i = 1; i < count; i++) {
            if(data[i] > maxnumber) {
                sec_max = maxnumber;
                maxnumber = data[i];
            }
            else {
                if(data[i] > sec_max)
                    sec_max = data[i];
            }
        }
        return sec_max;
    }
    public static void main(String[] args) {
        int[] array = {7,3,19,40,4,7,1};
        System.out.println(FindSecMax(array));
    }
}
```

8.5.3 如何求最大子数组之和

问题描述：一个有 n 个元素的数组，这 n 个元素可以是正数也可以是负数，数组中连续的一个或多个元素可以组成一个连续子数组，一个数组可能有多个这种连续子数组，求子数组和的最大值，例如：对于数组 $\{1, -2, 4, 8, -4, 7, -1, -5\}$ 而言，其最大和的子数组为 $\{4, 8, -4, 7\}$ ，最大值为 15。

方法一：“蛮力”法

最简单也是最容易想到的方法就是找出所有子数组，然后求出子数组的和，在所有子数组的和取最大值。

```
public class Test {
    public static int maxSubArray(int arr[]) {
        int n = arr.length;
        int ThisSum = 0, MaxSum = 0, i, j, k;
```

```

        for(i=0;i<n;i++)
            for(j=i;j<n;j++)
                ThisSum = 0;
                for(k=i;k<j;k++)
                    ThisSum += arr[k];
                if(ThisSum > MaxSum)
                    MaxSum = ThisSum;
            }
        return MaxSum;
    }

    public static void main(String[] args) {
        int[] array = {1, -2, 4, 8, -4, 7, -1, -5};
        System.out.println(maxSubArray(array));
    }
}

```

程序运行结果为：

15

以上这个算法的时间复杂度为 $O(n^3)$ ，显然效率太低，通过对该算法进行分析发现，许多子数组都重复计算了，鉴于此，下面给出一种优化的方法。

方法二：重复利用已经计算的子数组和

例如 $Sum[i, j] = Sum[i, j - 1] + arr[j]$ ，采用这种方法可以省去计算 $Sum[i, j - 1]$ 的时间，因此可以提高程序的效率。示例如下：

```

public class Test {
    public static int maxSubArray(int arr[]) {
        int size = arr.length;
        int maxSum = Integer.MIN_VALUE;
        for(int i=0;i<size;i++) {
            int sum = 0;
            for(int j=i;j<size;j++) {
                sum += arr[j];
                if(sum > maxSum) {
                    maxSum = sum;
                }
            }
        }
        return maxSum;
    }

    public static void main(String[] args) {
        int[] array = {1, -2, 4, 8, -4, 7, -1, -5};
        System.out.println(maxSubArray(array));
    }
}

```

以上这个算法的时间复杂度为 $O(n^2)$ 。

方法三：动态规划方法

可以采用动态规划的方法来降低算法的时间复杂度，实现思路如下。

首先可以根据数组的最后一个元素 $arr[n-1]$ 与最大子数组的关系分为以下 3 种情况：

1) 最大子数组的包含 $arr[n-1]$ ，即以 $arr[n-1]$ 结尾。

2) $arr[n-1]$ 单独构成最大子数组。

3) 最大子数组不包含 $arr[n-1]$ ，那么求 $arr[1, \dots, n-1]$ 的最大子数组可以转换为求 $arr[1, \dots, n-2]$ 的最大子数组。

通过上述分析可以得出如下结论：假设已经计算出 $(arr[0], \dots, arr[i-1])$ 最大的一段数组和为 $All[i-1]$ ，同时也计算出 $(arr[0], \dots, arr[i-1])$ 中包含 $arr[i-1]$ 的最大的一段数组和为 $End[i-1]$ ，则可以得出如下关系： $All[i-1] = \max\{arr[i-1], End[i-1], All[i-2]\}$ 。利用这个公式和动态规划的思想可以得到如下代码：

```
public class Test {
    public static int max(int m, int n) {
        return m > n ? m : n;
    }

    public static int maxSubArray(int arr[]) {
        int n = arr.length;
        int End[] = new int[n];
        int All[] = new int[n];
        End[n-1] = arr[n-1];
        All[n-1] = arr[n-1];
        End[0] = All[0] = arr[0];
        for(int i = 1; i < n; ++i) {
            End[i] = max(End[i-1] + arr[i], arr[i]);
            All[i] = max(End[i], All[i-1]);
        }
        return All[n-1];
    }

    public static void main(String[] args) {
        int[] array = {1, -2, 4, 8, -4, 7, -1, -5};
        System.out.println(maxSubArray(array));
    }
}
```

与前面几种方法相比，这种方法的时间复杂度为 $O(n)$ ，显然效率更高，但是由于在计算的过程中额外申请了两个数组空间，因此该算法的空间复杂度也为 $O(n)$ 。

方法四：优化的动态规划方法

方法三中每次只用到 $End[i-1]$ 与 $All[i-1]$ ，而不是整个数组中的值，因此可以定义两个变量来保存 $End[i-1]$ 与 $All[i-1]$ 的值，并且可以反复利用，这样就可以在保证时间复杂度为 $O(n)$ 的同时降低空间复杂度。示例如下：

```
public class Test {
    public static int max(int m, int n) {
        return m > n ? m : n;
    }

    public static int maxSubArray(int arr[]) {
        int n = arr.length;
        int nAll = arr[0]; // 有 n 个数字数组的最大子数组和
```



```

int nEnd = arr[0]; //有 n 个数字数组包含最后一个元素的子数组的最大和
for(int i = 1; i < n; ++i) {
    nEnd = max(nEnd + arr[i], arr[i]);
    nAll = max(nEnd, nAll);
}
return nAll;

public static void main(String[] args) {
    int[] array = {1, -2, 4, 8, -4, 7, -1, -5};
    System.out.println(maxSubArray(array));
}

```

在知道子数组的最大和之后，如何才能确定最大子数组的位置呢？为了得到最大子数组的位置，首先介绍另外一种计算最大子数组和的方法。在方法三中，通过对公式 $End[i] = \max(End[i-1] + arr[i], arr[i])$ 的分析可以看出，当 $End[i-1] < 0$ 时， $End[i] = array[i]$ ，其中， $End[i]$ 表示包含 $array[i]$ 的子数组和，如果某一个值使得 $End[i-1] < 0$ ，那么就重新从 $arr[i]$ 重新开始。示例如下：

```

public class Test {
    private static int begin = 0; //记录最大子数组的起始位置
    private static int end = 0; //记录最大子数组的结束位置
    public static int maxSubArray(int arr[]) {
        int maxSum = Integer.MIN_VALUE; //子数组最大值
        int nSum = 0; //包含子数组最后一位的最大值
        int nStart = 0;
        for(int i = 0; i < arr.length; i++) {
            if(nSum < 0) {
                nSum = arr[i];
                nStart = i;
            }
            else {
                nSum += arr[i];
            }
            if(nSum > maxSum) {
                maxSum = nSum;
                begin = nStart;
                end = i;
            }
        }
        return maxSum;
    }

    public static void main(String[] args) {
        int[] array = {1, -2, 4, 8, -4, 7, -1, -5};
        System.out.println("max = " + maxSubArray(array));
        System.out.println("begin = " + begin + ", end = " + end);
    }
}

```

程序运行结果为：

```
max = 15
begin = 2, end = 5
```

8.5.4 如何找出数组中重复元素最多的数

问题描述：对于数组 {1,1,2,2,4,4,4,4,5,5,6,6,6}，元素 1 出现的次数为 2 次，元素 2 出现的次数为 2 次，元素 4 出现的次数为 4 次，元素 5 出现的次数为 2 次，元素 6 出现的次数为 3 次，找出数组中出现重复次数最多的数。

上述问题中，程序的输出应该为元素 4。

可以采取如下两种方法来计算数组中重复次数最多的数。

方法一：空间换时间。可以定义一个数组 `int count[MAX]`，并将其数组元素都初始化为 0，然后执行 `for(int i=0;i<100;i++)count[A[i]]++` 操作，在 `count` 数组中找最大的数，即为重复次数最多的数。这是一种典型的空间换时间的算法。一般情况下，除非内存空间足够大，一般不采用这种方法。

方法二：使用 Map 映射表。通过引入 Map 映射表（Map 提供一对一的数据处理能力，其中第一个为关键字，每个关键字只能在 Map 中出现一次，第二个称为该关键字的值）来记录每一个元素出现的次数，然后判断次数大小，进而找出重复次数最多的元素。示例如下：

```
import java.util.*;

public class Test {

    public static int findMostFrequentInArray(int[] a) {
        int result = 0;
        int size = a.length;
        if (size == 0)
            return Integer.MAX_VALUE;
        //记录每个元素出现的次数
        Map<Integer,Integer> m = new HashMap<Integer,Integer>();
        for(int i=0;i<size;i++){
            if(m.containsKey(a[i])){
                m.put(a[i],m.get(a[i])+1);
            }
            else{
                m.put(a[i],1);
            }
        }
        //找出出现次数最多的元素
        int most = 0;
        Iterator iter = m.entrySet().iterator();
        while(iter.hasNext()){
            Map.Entry entry = (Map.Entry) iter.next();
            int key = (Integer) entry.getKey();
            int val = (Integer) entry.getValue();
            if(val > most){
                result = key;
                most = val;
            }
        }
    }
}
```

```

    }
    return result;
}

public static void main(String[] args) {
    int array[] = {1,5,4,3,4,4,5,4,5,5,6,6,6,6,6};
    int maxFrequencyNum = findMostFrequentInArray(array);
    System.out.println(maxFrequencyNum);
}
}

```

程序运行结果为：

6

8.5.5 如何求数组中两两相加等于20的组合种数

问题描述：给定一个数组{1,7,17,2,6,3,14}，这个数组中满足条件的有两对组合—— $17 + 3 = 20$ 和 $6 + 14 = 20$ 。

方法一：“蛮力”法

最容易想到的方法就是采用两重循环遍历数组来判断两个数的和是否为20。实现代码如下：

```

public class Test {
    public static void findSum(int[] a,int sum){
        int len = a.length;
        for(int i=0;i<len;i++){
            for(int j=i+1;j<len;j++){
                if(a[i] + a[j] == sum)
                    System.out.println(a[i] + "," + a[j]);
            }
        }
    }

    public static void main(String[] args) {
        int array[] = {1,7,17,2,6,3,14};
        findSum(array,20);
    }
}

```

程序运行结果为：

17,3
6,14

由于采用了双重循环，因此这个算法的时间复杂度为 $O(n^2)$ 。

方法二：排序法

先对数组元素进行排序，可以选用堆排序或快速排序，此时算法的时间复杂度为 $O(n \log n)$ ，然后对排序后的数组分别从前到后和从后到前遍历，假设从前往后遍历的下标为 $begin$ ，从后往前遍历的下标为 end ，那么当满足 $arr[begin] + arr[end] < 20$ 时，如果存在两个数的和为20，那么这两个数一定在 $[begin + 1, end]$ 之间；当满足 $arr[begin] + arr[end] > 20$ 时，如果存在两个数

的和为 20, 那么这两个数一定在 $[\text{begin}, \text{end} + 1]$ 之间。这个过程的时间复杂度为 $O(n)$, 因此整个算法的时间复杂度为 $O(n \log n)$ 。实现代码如下:

```
import java.util. Arrays;
public class Test {
    public static void findSum(int[] a,int sum) {
        Arrays. sort( a );
        int begin =0;
        int end = a. length - 1;
        while( begin < end ) {
            if( a[ begin ] + a[ end ] < sum)
                begin ++;
            else if( a[ begin ] + a[ end ] > sum)
                end --;
            else {
                System. out. println( a[ begin ] + ", " + a[ end ] );
                begin ++;
                end --;
            }
        }
    }
    public static void main( String[] args ) {
        int array[] = { 1,7,17,2,6,3,14 };
        findSum( array,20 );
    }
}
```

程序运行结果为:

```
3,17
6,14
```

这个算法的时间复杂度主要由排序算法的时间复杂度来决定。因此, 选择时间复杂度较低的排序算法能显著提高该算法的效率。

8.5.6 如何把一个数组循环右移 k 位

假设要把数组序列 12345678 右移 2 位变为 78123456, 比较移位前后数组序列的形式, 不难看出, 其中有两段序列的顺序是不变的, 即 78 和 123456, 可以把这两段看作两个整体, 右移 k 位就是把数组的两部分交换一下。鉴于此, 可以设计这样一种算法, 步骤如下 (以数组序列 12345678 为例):

- 1) 逆序数组子序列 123456, 数组序列的形式变为 65432178。
- 2) 逆序数组子序列 78, 数组序列的形式变为 65432187。
- 3) 全部逆序, 数组序列的形式变为 78123456。

程序代码如下:

```
public class Test {
    public static void reverse(int a[],int b,int e) {
        for(; b < e; b ++, e --) {
            int temp = a[ b ];

```

```

        a[e] = a[b];
        a[b] = temp;
    }
}

public static void shift_k(int a[],int k) {
    int n = a.length;
    k = k % n;// 为了防止 k 比 n 大,右移 k 位跟右移 k%n 位的结果是一样的
    reverse(a,n-k,n-1);
    reverse(a,0,n-k-1);
    reverse(a,0,n-1);
}

public static void main(String[] args) {
    int array[] = {1,2,3,4,5,6,7,8};
    shift_k(array,2);
    for(int i=0;i < array.length;i++) {
        System.out.print(array[i] + " ");
    }
}
}

```

程序运行结果如下:

```
7 8 1 2 3 4 5 6
```

从上例中可以看出,该算法只进行了3次逆序操作,因此时间复杂度为 $O(n)$ 。

8.5.7 如何找出数组中第 k 个最小的数

问题描述: 给定一个无序的数组, 从一个数组中找出第 k 个最小的数, 例如, 对于给定数组序列 $\{1,5,2,6,8,0,6\}$, 其中第 4 小的数为 5。

方法一: 排序法

最容易想到的方法就是对数组进行排序, 排序后的数组中第 $k-1$ 个位置上的数字即为数组的第 k 个最小的数 (原因是数组下标从 0 开始计数), 这种方法最好的时间复杂度为 $O(n\log n)$ 。

方法二: “剪枝”法

采用快速排序的思想来实现。主要思路如下: 选一个数 $tmp = a[n-1]$ 作为枢纽, 把比它小的数都放在它的左边, 比它大的数都放在它的右边, 然后判断 tmp 的位置, 如果它的位置为 $k-1$, 那么它就是第 k 个最小的数; 如果它的位置小于 $k-1$, 那么说明第 k 个小的元素一定在数组的右半部分, 采用递归的方法在数组的右半部分继续查找; 否则第 k 个小的元素在数组的左半部分, 采用递归的方法在左半部分数组中继续查找。示例如下:

```

public class Test {
    public static int quickSort(int array[],int low,int high,int k) {
        int i,j;
        int tmp;
        if(low > high)
            return Integer.MIN_VALUE;
        i = low + 1;
        j = high;
    }
}

```

```

tmp = array[i];
while(i < j) {
    while(i < j && array[j] >= tmp)
        j--;
    if(i < j)
        array[i++] = array[j];
    while(i < j && array[i] < tmp)
        i++;
    if(i < j)
        array[j--] = array[i];
}
array[i] = tmp;
if(i + 1 == k)
    return tmp;
else if(i + 1 > k)
    return quickSort(array, low, i - 1, k);
else
    return quickSort(array, i + 1, high, k);
}

public static int getKMin(int array[], int k) {
    if(array == null)
        return Integer.MIN_VALUE;
    if(array.length < k)
        return Integer.MIN_VALUE;
    return quickSort(array, 0, array.length - 1, k);
}

public static void main(String[] args) {
    int a[] = { 1, 5, 2, 6, 8, 0, 6 };
    int kMin = getKMin(a, 4);
    System.out.println(kMin);
}

```

程序运行结果为：

5

表面上看起来这种方法还是在对数组进行排序，但是它比排序法的效率高；主要原因是当在数组右半部分递归查找时，完全不需要关注左半部分数组的顺序，因此省略了对左半部分数组的排序。因此，这种方法可以被看作一种“剪枝”方法，不断缩小问题的规模，直到找到第 k 个小的元素。

8.5.8 如何找出数组中只出现一次的数字

问题描述：一个整型数组里除了一个数字之外，其他数字都出现了两次。找出这个只出现 1 次的数字。要求时间复杂度是 $O(n)$ ，空间复杂度是 $O(1)$ 。

如果本题对时间复杂度没有要求，最容易想到的方法就是先对这个整型数组排序，然后从第一个数字开始遍历，比较相邻的两个数，从而找出这个只出现 1 次的数字，这种方法的时间复杂度最快为 $O(n \log n)$ 。

由于时间复杂度与空间复杂度的限制，该方法不可取，因此需要一种更高效的方式。题目强调只有一个数字出现1次，其他数字出现了两次，首先想到的是异或运算，根据异或运算的定义可知，任何一个数字异或它自己都等于0，所以，如果从头到尾依次异或数组中的每一个数字，那些出现两次的数字全部在异或中会被抵消掉，最终的结果刚好是这个只出现1次的数字。示例如下：

```
public class Test {
    public static int findNotDouble(int a[]) {
        int n = a.length;
        int result = a[0];
        int i;
        for(i = 1; i < n; ++i)
            result ^= a[i];
        return result;
    }

    public static void main(String[] args) {
        int array[] = { 1,2,3,2,4,3,5,4,1 };
        int num = findNotDouble(array);
        System.out.println(num);
    }
}
```

程序运行结果为：

5

引申：如果题目改为数组 A 中，一个整型数组里除了一个数字之外，其他数字都出现了 3 次，那么如何找出这个数？

上述异或运算的方法只适用于其他数字出现的次数为偶数的情况，如果其他数字出现的次数为奇数，上述介绍的方法则不再适用。如果数组中的所有数都出现 n 次，那么这个数组中的所有数对应的二进制数中，各个位上的 1 出现的个数均可以被 n 整除。以 $n=3$ 为例，假如数组中有如下元素： $\{1,1,1,2,2,2\}$ ，它们对应的二进制表示为 01, 01, 01, 10, 10, 10。显然，这个数组中的所有数字对应的二进制数中第 0 位有 3 个 1，第 1 位有 3 个 1。对于本题而言，假设出现一次的这个数为 a ，那么去掉 a 后其他所有数字对应的二进制数的每个位置出现 1 的个数为 3 的倍数。因此可以对数组中的所有数字对应的二进制数中各个位置上 1 的个数对 3 取余数，就可以得到出现 1 次的这个数的二进制表示，从而可以找出这个数。示例如下：

```
public class Test {
    public static int findOnce(int a[], int appearTimes) {
        int n = a.length;
        int[] bitCount = new int[32];
        // 计算数组中所有数组对应的二进制数各个位置上出现 1 的次数
        for(int i = 0; i < n; i++)
            for(int j = 0; j < 32; j++)
                bitCount[j] += ((a[i] >> j) & 1);
        // 若某位上的结果不能被整除，则肯定目标数字在这一位上
        int appearOne = 0;
        for(int i = 0; i < 32; i++)
```

```

        if (bitCount[i] % appearTimes != 0)
            appearOne += (1 << i);
        return appearOne;
    }

    public static void main(String[] args) {
        int array[] = {1,2,1,2,4,2,4,4,1,3};
        int num = findOnce(array,3);
        System.out.println(num);
    }
}

```

程序运行结果为：

3

此外，这种方法不仅适用于求解其他数字出现个数为奇数的情况，也适用于求解出现次数为偶数的情况，具有更好的通用性。

8.5.9 如何找出数组中唯一的重复元素

问题描述：数组 $a[N]$ ， $1 \sim N-1$ 这 $N-1$ 个数存放在 $a[N]$ 中，其中某个数重复 1 次。写一个函数，找出被重复的数字。要求每个数组元素只能访问 1 次，并且不用辅助存储空间。

由于题目要求每个数组元素只能访问 1 次，且不用辅助存储空间，因此可以从原理上入手，采用数学求和法，因为只有一个数字重复 1 次，而又是连续的，根据累加和原理，对数组的所有项求和，然后减去 $1 \sim N-1$ 的和，即为所求的重复数。示例如下：

```

public class Test {
    public static int xor_findDup(int[] a) {
        int n = a.length;
        int tmp1 = 0;
        int tmp2 = 0;
        for (int i = 0; i < n - 1; ++i) {
            tmp1 += (i + 1);
            tmp2 += a[i];
        }
        tmp2 += a[n - 1];
        int result = tmp2 - tmp1;
        return result;
    }

    public static void main(String[] args) {
        int a[] = {1,2,1,3,4};
        int missingNum = xor_findDup(a);
        System.out.println(missingNum);
    }
}

```

程序运行结果为：

1

如果题目没有要求每个数组元素只能访问1次,且不允许使用辅助存储空间,还可以用异或法和位图法来求解。

(1) 异或法

根据异或法的计算方式,每两个相异的数执行异或运算之后,结果为1;每两个相同的数执行异或运算之后,结果为0,所以,数组 $a[N]$ 中的 N 个数异或结果与 $1 \sim N-1$ 异或的结果再做异或运算,得到的值即为所求。

设重复数为 A ,其余 $N-2$ 个数异或结果为 B , N 个数异或结果为 $A \wedge A \wedge B$, $1 \sim N-1$ 异或结果为 $A \wedge B$,由于异或满足交换律和结合律,且 $X \wedge X = 0$, $0 \wedge X = X$,则有 $(A \wedge B) \wedge (A \wedge A \wedge B) = A \wedge B \wedge B = A$ 。示例如下:

```
public class Test {
    public static int xor_findDup(int[] a) {
        int n = a.length;
        int i;
        int result = 0;
        for(i = 0; i < n; i++) {
            result ^= a[i];
        }
        for(i = 1; i < n; i++) {
            result ^= i;
        }
        return result;
    }
    public static void main(String[] args) {
        int a[] = {1,2,1,3,4}; int missingNum = xor_findDup(a);
        System.out.println(missingNum);
    }
}
```

程序运行结果为:

1

(2) 空间换时间法

申请长度为 $N-1$ 的整型数组 $flag$ 并初始化为0,然后从头开始遍历数组 a ,取每个数组元素 $a[i]$ 的值,将其对应的数组 $flag$ 中的元素赋值为1,如果已经置过1,那么该数就是重复的数。示例如下:

```
public class Test {
    public static int findInteger(int[] a) {
        int n = a.length;
        boolean[] arrayflag = new boolean[n];
        int i = 1;
        int result = Integer.MAX_VALUE;
        while (i < n) {
            arrayflag[i] = false;
            i++;
        }
    }
}
```

```

        for(i=0;i<n;i++){
            if(arrayflag[ a[i] ] == false)
                arrayflag[ a[i] ] = true;
            else {
                result = a[i];
            }
        }
        return result;
    }
}

public static void main(String[] args) {
    int a[] = {1,2,1,3,4};
    int missingNum = findInteger(a);
    System.out.println(missingNum);
}

```

程序运行结果为：

1

这种方法的空间复杂度比较大，需要申请长度为 N 的整数数组。当然也可以通过使用位图的方法来降低空间复杂度，即不是用一个整型数字来表示元素是否出现过（0 表示未出现，1 表示出现过），而是使用 1bit 来表示，因此需要申请数组的长度为 $N/32$ 取上整。

此题可以进行一个变形：取值为 $[1, n-1]$ 含 n 个元素的整数数组，至少存在一个重复数，即可能存在多个重复数， $O(n)$ 时间内找出其中任意一个重复数，例如， $array[] = \{1, 2, 2, 4, 5, 4\}$ ，则 2 和 4 均是重复元素。

方案一：位图法。使用大小为 n 的位图，记录每个元素是否已经出现过，一旦遇到一个已经出现过的元素，则直接将其输出。该方法的时间复杂度是 $O(n)$ ，空间复杂度为 $O(n)$ 。

方案二：数组排序法。先对数组进行计数排序，然后顺序扫描整个数组，一旦遇到一个已出现的元素，则直接将其输出。该方法的时间复杂度为 $O(n)$ ，空间复杂度为 $O(n)$ 。

以上提出的两种方案都需要额外的存储空间，能否不使用额外存储空间呢？答案是可以。于是想到了方案三：取反法。取反法的基本思路如下：如果遍历到数组中的元素为 i ，那么把 $a[i]$ 的值取反，如果 i 在数组中出现两次，那么 $a[i]$ 会经过两次取反操作， $a[i]$ 的值跟原始的值相等，且为正数；如果 i 出现了 1 次，那么 $a[i]$ 的值为原始值的相反数，且为负数，可以根据这个原理来实现。实现方法如下：将数组元素值作为索引，对于元素 $array[i]$ ，如果 $array[array[i]]$ 大于 0，那么设置 $array[array[i]] = -array[array[i]]$ ；如果 $array[array[i]]$ 小于 0，那么设置 $array[-array[i]] = -array[-array[i]]$ ，最后从数组第二个元素开始遍历数组，如果 $array[i] > 0$ ，那么这个数就是重复的。由于在进行遍历后对数组中的数据进行了修改，因此需要对数据进行还原（对数组中的负数取反）。示例如下：

```

public class Test {
    public static int xor_findDup(int[] a) {
        int n = a.length;
        int result = Integer.MAX_VALUE;
        for(int i=0;i<n;i++){
            if(a[i]>0){
                a[a[i]] = -a[a[i]];
            }
        }
    }
}

```

```

    } else {
        a[ -a[i] ] = -a[ -a[i] ];
    }
}
for(int i=1;i<n;i++){
    if(a[i]>0)
        result=i;
    else
        a[i] = -a[i];
}
return result;
}
public static void main(String[] args) {
    int a[] = {4,2,1,3,4};
    int missingNum = xor_findDup(a);
    System.out.println(missingNum);
}
}

```

方法四是一种非常“诡异”的算法，就是采用类似于单链表是否存在环的问题。“判断单链表是否存在环”是一个非常经典的问题，同时单链表可以采用数组实现，此时每个元素值作为 next 指针指向下一个元素。本题可以转化为“已知一个单链表中存在环，找出环的入口点”这种想法。具体思路如下：将 $array[i]$ 看作第 i 个元素的索引，即 $array[i] \rightarrow array[array[i]] \rightarrow array[array[array[i]]] \rightarrow array[array[array[array[i]]]] \rightarrow \dots$ ，最终形成一个单链表，由于数组 a 中存在重复元素，因此一定存在一个环，且环的入口元素即为重复元素。

该题的关键在于，数组 $array$ 的长度是 n ，而元素的范围是 $[1, n-1]$ ，所以 $array[0]$ 不会指向自己，进而不会陷入错误的自循环。如果元素的范围中包含 0，那么该题不可直接采用该方法。示例如下：

```

public class Test1 {
    public static int findInteger(int a[]) {
        int x,y;
        x=y=0;
        do {
            x=a[a[x]]; // x 一次走两步
            y=a[y]; // y 一次走一步
        } while(x!=y); // 找到环中的一个点
        x=0;
        do {
            x=a[x];
            y=a[y];
        } while(x!=y); // 找到入口点
        return x;
    }
    public static void main(String[] args) {
        int a[] = {1,2,1,3,4};
        int missingNum = findInteger(a);
        System.out.println(missingNum);
    }
}

```

程序运行结果为：

1

8.5.10 如何用递归方法求一个整数数组的最大元素

对于本题而言，最容易实现的方法为对数组进行遍历，定义一个变量 `max` 为数组的第一个元素，然后从第二个元素开始遍历，在遍历过程中，每个元素都与 `max` 的值进行比较，若该元素的值比 `max` 的值大，则把该元素的值赋给 `max`。当遍历完数组后，最大值也就求出来了。而使用递归方法求解的主要思路为：递归的求解“数组第一个元素”与“数组中其他元素组成的子数组的最大值”的最大值。示例如下：

```
public class Test {
    private int max(int a,int b){
        return a > b? a:b;p
    }
    public int maxnum(int a[],int begin){
        int length = a.length - begin;
        if(length == 1)
            return a[begin];
        else {
            return max(a[begin],maxnum(a,begin + 1));
        }
    }
    public static void main(String[] args){
        Test t = new Test();
        int[] num = {0,16,2,3,4,5,10,7,8,9};
        System.out.println(t.maxnum(num,0));
    }
}
```

程序运行结果为：

16

8.5.11 如何求数对之差的最大值

问题描述：数组中的一个数字减去它右边子数组中的一个数字可以得到一个差值，求所有可能的差值中的最大值，例如，数组 `{1,4,17,3,2,9}` 中，最大的差值为 $17 - 2 = 15$ 。

方法一：“蛮力”法。“蛮力”法也是最容易想到的方法，其原理如下：首先，遍历数组，找到所有可能的差值；其次，从所有差值中找出最大值。具体实现方法为：针对数组 `a` 中的每个元素 `a[i]` ($0 < i < n - 1$)，求所有 `a[i] - a[j]` ($i < j < n$) 的值中的最大值。示例如下：

```
public class Test {
    public static int getMax(int[] a){
        if(a == null)
            return Integer.MIN_VALUE;
    }
}
```



```

int len = a.length;
if (len <= 1)
    return Integer.MIN_VALUE;
int max = Integer.MIN_VALUE;
for (int i = 0; i < len - 1; i++) {
    for (int j = i + 1; j < len; j++)
        if (a[i] - a[j] > max)
            max = a[i] - a[j];
}
return max;
}
public static void main(String[] args) {
    int[] a = {1, 4, 17, 3, 2, 9};
    System.out.println(getMax(a));
}
}

```

程序运行结果为：

15

采用这种方法虽然也能求出最大值，但是它的时间复杂度为 $O(n)$ 。

方法二：二分法。通过二分法可以减少计算的次数。思路如下：把数组分为两个子数组，那么最大的差值只能有 3 种可能：①最大的差值对应的被减数和减数都在左子数组中，假设最大差值为 leftMax；②被减数和减数都在右子数组中，假设最大差值为 rightMax；③被减数是左子数组的最大值，减数是右子数组中的最小值，假设差值为 mixMax。那么就可以得到这个数组的最大差值为这 3 个差值的最大值，即 $\max(\text{leftMax}, \text{rightMax}, \text{mixMax})$ 。实现代码如下：

```

import java.util.concurrent.atomic.AtomicInteger;
public class Test {
    public static int getMax(int a[]) {
        if (a == null)
            return Integer.MIN_VALUE;
        int len = a.length;
        if (len <= 1)
            return Integer.MIN_VALUE;
        AtomicInteger max = new AtomicInteger(0);
        AtomicInteger min = new AtomicInteger(0);
        return getMaxDiff(a, 0, len - 1, max, min);
    }
    public static int getMaxDiff(int a[], int begin, int end, AtomicInteger max, AtomicInteger min) {
        if (begin == end) {
            max.set(a[begin]);
            min.set(a[begin]);
            return Integer.MIN_VALUE;
        }
        int middle = begin + (end - begin) / 2;
        // 数组前半部分的最小值与最大值
        AtomicInteger leftMax = new AtomicInteger(0);

```

```

    AtomicInteger lMin = new AtomicInteger(0);
    //数组前半部分的最大差值(第一种情况)
    int leftMax = getMaxDiff(a, begin, middle, lMax, lMin);
    //数组后半部分的最小值与最大值
    AtomicInteger rMax = new AtomicInteger(0);
    AtomicInteger rMin = new AtomicInteger(0);
    //数组后半部分的最大差值(第二种情况)
    int rightMax = getMaxDiff(a, middle + 1, end, rMax, rMin);
    //对应第三种情况
    int mixMax = lMax.get() - rMin.get();
    //求数组的最大值与最小值
    if (lMax.get() > rMax.get())
        max.set(lMax.get());
    else
        max.set(rMax.get());
    if (lMin.get() < rMin.get())
        min.set(lMin.get());
    else
        min.set(rMin.get());
    //求最大的差值
    int allMax = (leftMax > rightMax) ? leftMax : rightMax;
    allMax = (allMax > mixMax) ? allMax : mixMax;
    return allMax;
}
public static void main(String[] args) {
    int[] a = {1, 4, 17, 3, 2, 9};
    System.out.println(getMax(a));
}
}

```

程序运行结果为:

15

显然, 以上这种方法对数组只经过一次遍历, 一次时间复杂度为 $O(n)$, 但是由于采用了递归的实现方式, 在递归调用时要进行压栈与弹栈操作, 因此有额外的开销, 会导致算法性能有所下降。另外, 在实现时, 为了通过传递引用的方式获取数组的最大值与最小值, 使用了 `AtomicInteger` 而不是 `Integer` 类。主要原因为 `Integer` 类虽然也可以传递引用, 但是它是不可变量, 在方法内部不能对其进行修改。

方法三: 动态规划。通过对题目进行分析, 发现这是一个非常典型的动态规划问题, 可以用动态规划的方法来求解。实现思路如下: 给定数组 `a`, 申请额外的数组 `diff` 和 `max`, 其中 `diff[i]` 是以数组中第 `i` 个数字为减数的所有数对之差的极大值 (前 `i+1` 个数组成的子数组中最大的差值), `max[i]` 为前 `i+1` 个数的极大值。假设已经求得了 `diff[i]`, `diff[i+1]` 的值有两种可能性: ①等于 `diff[i]`; ②等于 `max[i] - a[i]`。通过上面的分析, 可以得到动态规划方法的计算表达式为: `diff[i+1] = max(diff[i], max[i-1] - a[i])`, `max[i+1] = max(max[i], a[i+1])`。数组最大的差值为 `diff[n-1]` (`n` 为数组的长度)。示例如下:

```
public class Test {
```

```

public static int max(int m,int n) {
    return(m > n)? m:n;
}

public static int getMax(int[] a) {
    if(a == null)
        return Integer. MIN_VALUE;
    int len = a. length;
    if(len <= 1)
        return Integer. MIN_VALUE;
    int[] diff = new int[ len];
    int[] max = new int[ len];
    diff[0] = 0;
    max[0] = a[0];
    for(int i = 1; i < len; i++) {
        diff[i] = max(diff[i - 1], max[i - 1] + a[i]);
        max[i] = max(max[i - 1], a[i]);
    }
    return diff[ len - 1];
}

public static void main(String[] args) {
    int[] a = {1,4,17,3,2,9};
    System. out. println( getMax(a));
}

```

程序运行结果为:

15

以上这种方法也是对数据进行了一次遍历,因此时间复杂度为 $O(n)$,由于没有采用递归的方式,因此相比方法二,在性能上有所提升。由于引入了两个额外的数组,因此这个算法的空间复杂度也为 $O(n)$ 。

从动态规划方法的计算公式中可以看出,在求解 $\text{diff}[i+1]$ 时,只用到了 $\text{diff}[i]$ 与 $\text{max}[i]$,而与数组 diff 和 max 中其他数字无关,因此可以通过两个变量而不是数组来记录 $\text{diff}[i]$ 与 $\text{max}[i]$ 的值,从而降低了算法的空间复杂度。示例如下:

```

public class Test {
    public static int max(int m,int n) {
        return(m > n)? m:n;
    }

    public static int getMax(int[] a) {
        if(a == null)
            return Integer. MIN_VALUE;
        int len = a. length;
        if(len <= 1)
            return Integer. MIN_VALUE;
        int diff = 0;
        int max = a[0];
        for(int i = 1; i < len; i++) {

```

```

        diff = max(diff, max - a[i]);
        max = max(max, a[i]);
    }
    return diff;
}

public static void main(String[] args) {
    int[] a = {1,4,17,3,2,9};
    System.out.println(getMax(a));
}

```

引申：这道题还可以用求最大子数组之和的方法来解决吗？

答案：可以。实现思路如下：给定一个数组 a （数组长度为 n ），额外申请一个长度为 $n - 1$ 的数组 $diff$ ，数组 $diff$ 中的值满足 $diff[i] = a[i] - a[i + 1]$ ，那么 $a[i] - a[j]$ ($0 < i < j < n$) 就等价于 $diff[i] + diff[i + 1] + \dots + diff[j]$ 。因此，求所有 $a[i] - a[j]$ 组合的最大值就可以转换为求解所有 $diff[i] + diff[i + 1] + \dots + diff[j]$ 组合的最大值。由于 $diff[i] + diff[i + 1] + \dots + diff[j]$ 代表 $diff$ 的一个子数组，因此可以用 11.5.3 节中求最大子数组之和的方法来解决。

8.5.12 如何求绝对值最小的数

问题描述：有一个升序排列的数组，数组中可能有正数、负数或 0，求数组中元素的绝对值最小的数，例如，数组 $\{-10, -5, -2, 7, 15, 50\}$ ，绝对值最小的是 -2 。

求绝对值最小的数可以分为 3 种情况：①如果数组第一个元素为非负数，那么绝对值最小的数肯定为数组的第一个元素；②如果数组最后一个元素为负数，那么绝对值最小的数肯定是数组的最后一个元素；③数组中既有正数又有负数时，首先找到正数与负数的分界点，如果分界点恰好为 0，那么 0 就是绝对值最小的数，否则通过比较分界点左右的正数与负数的绝对值来确定最小的数。对于上面的例子来说，正数与负数的分界点为 -2 和 7 。通过比较它们的绝对值从而确定 -2 的绝对值更小，因此 -2 就是要查找的数。

那么如何来查找正数与负数的分界点呢？最简单的方法仍然是顺序遍历数组，找出第一个非负数（前提是数组中既有正数又有负数），接着通过比较分界点两个数的值来找出绝对值最小的数。这种方法在最坏的情况下时间复杂度为 $O(n)$ 。下面主要介绍采用二分法来查找正数与负数的分界点的方法。其主要思路为：取数组中间位置的值 $a[mid]$ 。① $a[mid] = 0$ ，那么这个数就是绝对值最小的数；② $a[mid] > 0$ ，如果 $a[mid - 1] < 0$ ，那么就找到了分界点，通过比较 $a[mid]$ 与 $a[mid - 1]$ 的绝对值就可以找到数组中绝对值最小的数，如果 $a[mid - 1] = 0$ ，那么 $a[mid - 1]$ 就是要找的数，否则接着在数组的左半部分查找；③ $a[mid] < 0$ ，如果 $a[mid + 1] > 0$ ，那么通过比较 $a[mid]$ 与 $a[mid + 1]$ 的绝对值即可，如果 $a[mid + 1] = 0$ ，那么 $a[mid + 1]$ 就是要查找的数，否则接着在数组的右半部分继续查找。实现代码如下：

```

public class Test {
    public static int getMinAbsoluteValue(int[] a) {
        if (a == null)
            return Integer.MIN_VALUE;
        int len = a.length;
        if (len < 1)
            return Integer.MIN_VALUE;
        //数组中没有负数
    }
}

```

```

if( a[0] >=0)
    return a[0];
//数组中没有正数
if( a[ len - 1] <=0)
return a[ len - 1];
int mid =0;
int begin =0;
int end = len - 1;
int absMin =0;
//数组中既有正数又有负数
while ( true) {
    mid = begin + ( end - begin)/2;
    //如果值等于0,那么就是绝对值最小的数
    if( a[ mid] ==0) {
        return 0;
    }
    // 如果值大于0,那么正负数的分界点在左半部分
    else if( a[ mid] >0) {
        //继续在数组的左半部分查找
        if( a [ mid - 1] >0)
            end = mid - 1;
        else if( a[ mid - 1] ==0)
            return 0;
        //找到正负数的分界点
        else
            break;
    }
    // 如果值小于0,在数组右半部分查找
    else {
        //在数组右半部分继续查找
        if( a [ mid + 1] <0)
            begin = mid + 1;
        else if( a[ mid + 1] ==0)
            return 0;
        //找到正负数的分界点
        else
            break;
    }
}
// 获取正负数分界点出绝对值最小的值
if( a [ mid] >0) {
    if( a [ mid] < Math. abs( a[ mid - 1]))
        absMin = a[ mid];
    else
        absMin = a[ mid - 1];
}
else {
    if( Math. abs( a[ mid] ) < a[ mid + 1])
        absMin = a[ mid];
    else
        absMin = a[ mid + 1];
}
}

```

```

        return absMin;
    }

    public static void main(String[] args) throws Exception {
        int[] a1 = { -10, -5, -2, 7, 15, 50 };
        int[] a2 = { 2, 4, 6, 8, 27 };
        int[] a3 = { -13, -9, -7, -3 };
        int value = getMinAbsoluteValue(a1);
        System.out.println(value);
        value = getMinAbsoluteValue(a2);
        System.out.println(value);
        value = getMinAbsoluteValue(a3);
        System.out.println(value);
    }
}

```

程序运行结果为：

```

-2
2
-3

```

8.5.13 如何求数组中两个元素的最小距离

问题描述：给定一个数组，数组中含有重复元素，给出两个数 $n1$ 和 $n2$ ，求这两个数字在数组中所出现位置的最小距离，例如，数组 $\{4, 5, 6, 4, 7, 4, 6, 4, 7, 8, 5, 6, 4, 3, 10, 8\}$ 中，4 和 8 的最小距离为 2。

实现思路如下：遍历数组，会遇到以下两种情况。

1) 当遇到 $n1$ 时，记录下 $n1$ 值对应的数组下标的位置 $n1_index$ ，通过求 $n1_index$ 与上次遍历到 $n2$ 的下标值 $n2_index$ 的差，可以求出最近一次遍历到的 $n1$ 与 $n2$ 的距离。

2) 当遇到 $n2$ 时，同样记录下它在数组中下标的位置 $n2_index$ ，然后通过求 $n2_index$ 与上次遍历到 $n1$ 的下标值 $n1_index$ 的差，求出最近一次遍历到的 $n1$ 与 $n2$ 的距离。

定义一个变量 min_dist 记录 $n1$ 与 $n2$ 的最小距离，在以上两种情况下，每次求出 $n1$ 与 $n2$ 的距离后与 min_dist 相比，求最小值。这样只需对数组进行一次遍历就可以求出最小距离，因此时间复杂度为 $O(n)$ 。实现代码如下：

```

public class Test {
    private static int min(int a, int b) {
        return a > b ? b : a;
    }

    public static int minDistance(int a[], int n1, int n2) {
        if (a == null)
            return Integer.MIN_VALUE;

        int len = a.length;
        int n1_index = -1;
        int n2_index = -1;
        int min_dist = Integer.MIN_VALUE + 1;
        for (int i = 0; i < len; ++i) {
            if (a[i] == n1) {

```



```

        n1_index = i;
        if( n2_index >=0)
            min_dist = min( Math. abs( min_dist ), Math. abs( n1_index - n2_index ));
    if( a [ i ] == n2) {
        n2_index = i;
        if( n1_index >=0)
            min_dist = min( Math. abs( min_dist ), Math. abs( n2_index - n1_index ));
    }
}
return min_dist;
}

public static void main( String [ ] args ) {
    int a [ ] = { 4, 5, 6, 4, 7, 4, 6, 4, 7, 8, 5, 6, 4, 3, 10, 8 };
    System. out. println( minDistance( a, 4, 8 ));
}
}

```

程序运行结果为：

2

8.5.14 如何求指定数字在数组中第一次出现的位置

问题描述：给定数组 $a = \{3, 4, 5, 6, 5, 6, 7, 8, 9, 8\}$ ，这个数组中相邻元素之差都为1，给定数字9，它在数组中第一次出现的位置的下标为8（数组下标从0开始）。

方法一：“蛮力”法。假设指定数字为 t 顺序遍历数组中每一个元素，并且将数组中的元素与 t 进行比较，判断两个值是否相等，若相等，则返回下标位置；若遍历完数组还没找到 t ，则说明 t 在数组中不存在，返回 -1 。该方法的时间复杂度为 $O(n)$ 。

这种方法显然没有用到题目中“这个数组中相邻元素之差的绝对值为1”这一条件，下面介绍一种更高效的方法。

方法二：跳跃搜索法。通过对数组中元素的特点进行分析发现如下规律：假设先从数组 a 中查找9出现的位置，首先用数组中第一个元素（数组下标为0）3与9进行比较，它们的差值为6，由于数组中相邻两个元素的差值为1，因此9在数组中出现的最早的位置必定为：第 $1+6=7$ 个位置（数组下标为6）。这是因为：如果数组是递增的，那么数组第7个元素的值才为9；如果数组不是递增的，那么9出现的位置肯定在数组中第7个元素后面；上面的示例中待查找的数比数组中第一个元素的值大，对于待查找的数比数组中第一个元素小的情况，可以用相同的方法。根据这个特点可以得出算法的思路为：从数组第一个元素开始（ $i=0$ ），把数组当前位置的值与 t 进行比较，如果相等，则返回数组下标，否则，从数组下标为 $i + |t - a[i]|$ 处继续查找。实现代码如下：

```

public class Test {
    public static int findIndex( int a [ ], int t ) {
        if( a == null )
            return ;
        int len = a. length ;
        int i = 0 ;
        while ( i < len ) {

```

```

        if(a[i] == t) {
            return i;
        } else {
            i += Math.abs(t - a[i]);
        }
    }
    return -1;
}

public static void main(String[] args) {
    int a[] = {3,4,5,6,5,6,7,8,9,8};
    System.out.println(findIndex(a,9));
}
}

```

程序运行结果为：

8

显然，采用以上跳跃搜索法减少了对数组中元素的访问个数，从而提高了算法的效率。

8.5.15 如何对数组的两个子有序段进行合并

问题描述：数组 $a[0, mid - 1]$ 和 $a[mid, n - 1]$ 是各自有序的，对数组 $a[0, n - 1]$ 的两个子有序段进行合并，得到 $a[0, n - 1]$ 整体有序。要求空间复杂度为 $O(1)$ （注： $a[i]$ 元素是支持 ' $<$ ' 运算符的）。假设给定数组 $a = \{1, 5, 6, 7, 9, 2, 4, 8, 10, 13, 14\}$ ， $mid = 5$ ， $a[0] \sim a[4]$ 是有序的， $a[5] \sim a[10]$ 是有序的，合并后的数组为 $\{1, 2, 4, 5, 6, 7, 8, 9, 10, 13, 14\}$ 。

假设数组中的两个子有序段都按升序排列，如上例所示。下面给出在这种情况下实现思路。

由于限定空间复杂度为 $O(1)$ ，因此不能使用归并方法。最容易想到的就是插入排序方法，这种算法的时间复杂度为 $O(n^2)$ ，空间复杂度为 $O(1)$ ，能满足题目要求，但是由于插入排序方法没有用到“数组 $a[0, mid - 1]$ 和 $a[mid, num - 1]$ 是各自有序的”这个条件，因此这种算法肯定不是最好的算法，下面给出另外一种方法。

实现思路：首先，遍历数组中下标为 $0 \sim mid - 1$ 的元素，将遍历到的元素的值与 $a[mid]$ 进行比较，当遍历到 $a[i]$ ($0 \leq i \leq mid - 1$) 时，如果满足 $a[mid] < a[i]$ ，那么交换 $a[i]$ 与 $a[mid]$ 的值。接着找到交换后的 $a[mid]$ 在 $a[mid, num - 1]$ 中的具体位置（在 $a[mid, num - 1]$ 中进行插入排序），实现方法为：遍历 $a[mid \sim num - 2]$ ，如果 $a[mid + 1] < a[mid]$ ，那么交换 $a[mid]$ 与 $a[mid + 1]$ 的位置。实现代码如下：

```

public class Test {
    public static void findRightPlaceForMid(int a[], int mid) {
        int len = a.length;
        int tmp;
        for(int i = mid; i < len - 1; i++) {
            if(a[i + 1] < a[i]) {
                tmp = a[i];
                a[i] = a[i + 1];
                a[i + 1] = tmp;
            }
        }
    }
}

```

```

    }
}

public static void sort(int a[], int mid) {
    int tmp;
    for (int i = 0; i <= mid - 1; i++) {
        if (a[mid] < a[i]) {
            tmp = a[i];
            a[i] = a[mid];
            a[mid] = tmp;
            findRightPlaceForMid(a, mid);
        }
    }
}

public static void main(String[] args) {
    int a[] = {1, 5, 6, 7, 9, 2, 4, 8, 10, 13, 14};
    sort(a, 5);
    for (int i = 0; i < 11; i++)
        System.out.print(a[i] + " ");
}

```

程序运行结果为：

1 2 4 5 6 7 8 9 10 13 14

在实现时需要注意的问题是：题目中给出了“ $a[i]$ 元素是支持‘<’运算符的”这一限制条件，因此，在程序中只能出现形如“ $a[i] < a[j]$ ”的表达式，最好不要出现“ $a[j] > a[i]$ ”这种写法。

引申：

- 1) 如果数组中两个子有序段都按降序排列，可以用类似的方法来解决。
- 2) 如果其中一个子序段按升序排列，另外一个子序段按降序排列，可以首先对其中一个子序段进行逆序，然后采用上面介绍的方法进行排序。

8.5.16 如何计算两个有序整型数组的交集

假设两个含有 n 个元素的有序（非降序）整型数组 a 和 b ，其中 $a = \{0, 1, 2, 3, 4\}$ ， $b = \{1, 3, 5, 7, 9\}$ ，那么它们的交集为 $\{1, 3\}$ 。

计算数组交集可以采用很多种方法，但数组的相对大小一般会影响到算法的效率，所以需要根据两个数组的相对大小来确定采用的方法：

- 1) 对于两个数组长度相当的情况，一般可以采取如下 3 种方法。

方法一：二路归并法。设两个数组分别为 $array1[n1]$ 和 $array2[n2]$ 。分别以 i, j 从头开始遍历两个数组。在遍历过程中，若当前遍历位置的 $array1[i]$ 与 $array2[j]$ 相等，则此数为两个数组的交集，记录下来，并继续向后遍历 $array1$ 和 $array2$ 。若 $array1[i]$ 大于 $array2[j]$ ，则须继续向后遍历 $array2$ 。若 $array1[i]$ 小于 $array2[j]$ ，则需要继续向后遍历 $array1$ ，直到有一个数组结束遍历即停止。实现代码如下：

```

import java.util.*;

public class Test {

```

```

public static ArrayList < Integer > mixed(int array1[] ,int array2[] ) {
    ArrayList < Integer > mix = new ArrayList < Integer > ();
    int i =0 ,j =0;
    int n1 = array1. length;
    int n2 = array2. length;
    while ( i < n1 && j < n2) {
        if( array1[i] == array2[j] ) {
            mix. add( array1[i] );
            i ++;
            j ++;
        } else if( array1[i] > array2[j] ) {
            j ++;
        } else if( array1[i] < array2[j] ) {
            i ++;
        }
    }
    return mix;
}

public static void main( String[] args ) {
    int[] a = {0,1,2,3,4};
    int[] b = {1,3,5,7,9};
    ArrayList < Integer > mix = mixed( a, b );
    for( int i =0; i < mix. size(); i ++ )
        System. out. print( mix. get( i ) + " " );
}

```

程序运行结果为：

1 3

方法二：顺序遍历法。顺序遍历两个数组，将数组元素存放到哈希表中，同时对统计的数组元素进行计数，若为 2，则为二者的交集元素。

方法三：散列法。遍历两个数组中任意一个数组，将遍历得到的元素存放到散列表，然后遍历另外一个数组，同时对建立的散列表进行查询，若存在，则为交集元素。

2) 对于两个数组长度相差悬殊的情况，例如，数组 a 的长度远远大于数组 b 的长度，则可以采用下面几种方法。

方法一：依次遍历长度短的数组，将遍历得到的数组元素在长数组中进行二分查找。具体而言，设两个指向两个数组末尾元素的指针，取较小的那个数在另一个数组中二分查找，找到，则存在一个交集，并且将该目标数组的指针指向该位置的前一个位置。如果没有找到，同样可以找到一个位置，使得目标数组中在该位置后的数肯定不在另一个数组中存在，直接移动该目标数组的指针指向该位置的前一个位置，再循环找，直到一个数组为空为止。由于两个数组中都可能出现重复的数，因此二分查找时，当找到一个相同的数 x 时，其下标为 i，那么下一个二分查找的下界变为 i+1，避免 x 重复使用。

方法二：采用与方法一类似的方法，但是每次查找在前一次查找的基础上进行，这样可以大大缩小查找表的长度。

方法三：采用与方法二类似的方法，但是遍历长度小的数组的方式有所不同，即从数组头

部和尾部同时开始遍历，这样可以进一步缩小查找表的长度。

8.5.17 如何判断一个数组中数值是否连续相邻

一个数组序列，元素取值可能是 $0 \sim 65535$ 中的任意一个数，相同数值不会重复出现。 0 是例外，可以反复出现。设计一种算法，当从该数组序列中随意选取 5 个数值，判断这 5 个数值是否连续相邻。需要注意以下 4 点：

- 1) 5 个数值允许是乱序的，例如 $\{8,7,5,0,6\}$ 。
- 2) 0 可以通配任意数值，例如 $\{8,7,5,0,6\}$ 中的 0 可以通配成 9 或者 4 。
- 3) 0 可以多次出现。
- 4) 全 0 算连续，只有一个非 0 算连续。

如果没有 0 的存在，要组成连续的数列，最大值和最小值的差距必须是 4 ，存在 0 的情况下，只要最大值和最小值的差距小于 4 就可以了，所以应找出数列中非 0 的最大值和非 0 的最小值，时间复杂度为 $O(n)$ ，如果非 0 最大 - 非 0 最小 + $1 \leq 5$ （即非 0 最大 - 非 0 最小 ≤ 4 ），那么这 5 个数值连续相邻，否则，不连续相邻。因此，该算法的时间复杂度为 $O(n)$ 。

程序代码如下：

```
public class Test {
    public static Boolean IsContinuous(int[] a) {
        int n = a.length;
        int min = -1, max = -1;
        for(int i=0; i<n; i++) {
            if(a[i] != 0) {
                if(min > a[i] || -1 == min)
                    min = a[i];
                if(max < a[i] || -1 == max)
                    max = a[i];
            }
        }
        if(max - min > n - 1)
            return false;
        else
            return true;
    }

    public static void main(String[] args) {
        int array[] = {8,7,5,0,6};
        if(IsContinuous(array))
            System.out.println("数组{8,7,5,0,6}连续相邻\n");
        else
            System.out.println("数组{8,7,5,0,6}不连续相邻");
    }
}
```

程序运行结果为：

8,7,5,0,6 连续相邻

8.5.18 如何求解数组中反序对的个数

问题描述：给定一个数组 a ，如果 $a[i] > a[j]$ ($i < j$)，那么 $a[i]$ 与 $a[j]$ 被称为一个反序，例如，给定数组 $\{1,5,3,2,6\}$ ，共有 $(5,3)$ 、 $(5,2)$ 和 $(3,2)$ 三个反序对。

方法一：“蛮力”法。最容易想到的方法是对数组中的每一个数字，遍历它后面的所有数字，如果后面的数字比它小，那么就找到一个逆序对，实现代码如下：

```
public class Test {
    public static int reverseCount( int a[] ) {
        int count = 0;
        int len = a.length;
        for( int i = 0; i < len; i++ ) {
            for( int j = i + 1; j < len; j++ ) {
                if( a[ i ] > a[ j ] ) {
                    count ++;
                }
            }
        }
        return count;
    }
    public static void main( String[] args ) {
        int array[] = { 1, 5, 3, 2, 6 };
        int count = reverseCount( array );
        System.out.println( count );
    }
}
```

程序运行结果为：

3

这种方法是采用二重遍历实现的，因此时间复杂度为 $O(n^2)$ 。

方法二：分治归并法。可以参考归并排序的方法，在归并排序的基础上额外使用一个计数器来记录逆序对的个数。下面以数组序列 $\{5,8,3,6\}$ 为例，说明计数的方法，归并的过程如下所示。

[5 8] [3 6]
└──────────┘

归并第一步 : [3]	逆序对 +2
归并第二步 : [3 5]	逆序对 +0
归并第三步 : [3 5 6]	逆序对 +1
归并第四步 : [3 5 6 7]	逆序对 +1

示例如下：

```
public class Test {
    public static int reverseCount = 0;
    public static void merge( int array[] , int begin, int mid, int end ) {
        int i, j, k, n1, n2;
        n1 = mid - begin + 1;
```



```

n2 = end - mid;
int[] L = new int[n1];
int[] R = new int[n2];
for(i=0, k=begin; i < n1; i++, k++)
    L[i] = array[k];
for(i=0, k=mid+1; i < n2; i++, k++)
    R[i] = array[k];
for(k=begin, i=0, j=0; i < n1 && j < n2; k++) {
    if(L[i] < R[j])
        array[k] = L[i++];
    } else {
        reverseCount += mid - i + 1;
        array[k] = R[j++];
    }
}
if(i < n1)
    for(j=i; j < n1; j++, k++)
        array[k] = L[j];
if(j < n2)
    for(i=j; i < n2; i++, k++)
        array[k] = R[i];
}

public static void merge_sort(int a[], int begin, int end) {
    if(begin < end) {
        int mid = (end + begin) / 2;
        merge_sort(a, begin, mid);
        merge_sort(a, mid + 1, end);
        merge(a, begin, mid, end);
    }
}

public static void main(String[] args) {
    int array[] = {1, 5, 3, 2, 6};
    merge_sort(array, 0, array.length - 1);
    System.out.println(reverseCount);
}

```

程序运行结果为：

3

这种方法与归并排序有着相同的时间复杂度 $O(n \log n)$ 。

8.5.19 如何求解最小三元组距离

问题描述：已知 3 个升序整数数组 $a[1]$ 、 $b[m]$ 和 $c[n]$ 。请在 3 个数组中各找一个元素，使得组成的三元组距离最小。三元组的距离定义是：假设 $a[i]$ 、 $b[j]$ 和 $c[k]$ 是一个三元组，那么距离为 $\text{Distance} = \max(|a[i] - b[j]|, |a[i] - c[k]|, |b[j] - c[k]|)$ ，请设

计一个求最小三元组距离的最优算法。

方法一：“蛮力”法。最容易想到的方法就是分别遍历3个数组中的元素，分别求出它们的距离，然后从这些值里面查找最小值，示例如下：

```
public class Test {
    public static int max(int a,int b,int c){
        int max = a < b? b:a;
        max = max < c? c:max;
        return max;
    }
    public static int Solvingviolence(int a[],int b[],int c[]){
        int aLen = a.length;
        int bLen = b.length;
        int cLen = c.length;
        int minDist = max(Math.abs(a[0] - b[0]), Math.abs(a[0] - c[0]), Math.abs(b[0] - c[0]));
        int dist = 0;
        for(int i = 0; i < aLen; i++){
            for(int j = 0; j < bLen; j++){
                for(int k = 0; k < cLen; k++){
                    //求距离
                    dist = max(Math.abs(a[i] - b[j]), Math.abs(a[i] - c[k]), Math.abs(b[j] - c[k]));
                    //找出最小距离
                    if(minDist > dist){
                        minDist = dist;
                    }
                }
            }
        }
        return minDist;
    }
    public static void main(String[] args){
        int a[] = {3,4,5,7};
        int b[] = {10,12,14,16,17};
        int c[] = {20,21,23,24,37,30};
        System.out.println(Solvingviolence(a,b,c));
    }
}
```

这个算法的时间复杂度为 $O(1 * m * n)$ ，显然这个方法没有用到数组升序这一特性，因此不是最好的方法。

方法二：最小距离法。假设当前遍历到这3个数组中的元素分别为 a_i 、 b_i 、 c_i ，并且 $a_i \leq b_i \leq c_i$ ，此时它们的距离肯定为 $D_i = c_i - a_i$ ，那么可以分如下3种情况讨论：

1) 如果接下来求 a_i 、 b_i 、 c_{i+1} 的距离，那么由于 $c_{i+1} \geq c_i$ ，此时它们的距离必定为 $D_{i+1} = c_{i+1} - a_i$ ，显然 $D_{i+1} \geq D_i$ ，因此 D_{i+1} 不可能为最小距离。

2) 如果接下来求 a_i 、 b_{i+1} 、 c_i 的距离，由于 $b_{i+1} \geq b_i$ ，如果 $b_{i+1} \leq c_i$ ，此时它们的距离仍然为 $D_{i+1} = c_i - a_i$ ；如果 $b_{i+1} > c_i$ ，那么此时它们的距离为 $D_{i+1} = b_{i+1} - a_i$ ，显然 $D_{i+1} \geq D_i$ ，

因此 D_{i+1} 不可能为最小距离。

3) 如果接下来求 a_{i+1} , b_i , c_i 的距离, 如果 $a_{i+1} < |c_i - a_i| + c_i$, 此时它们的距离 $D_{i+1} = c_i - a_{i+1}$, 显然 $D_{i+1} < D_i$, 因此 D_{i+1} 有可能是最小距离。

综上所述, 在求最小距离时只需要考虑第3种情况。具体实现思路为: 从3个数组的第一个元素开始, 先求出它们的距离 minDist , 接着找出这3个数中最小数对应的数组, 只对这个数组的下标往后移一个位置, 接着求3个数组中当前元素的距离, 若比 minDist 小, 则把当前距离赋值给 minDist , 以此类推, 直到遍历完其中一个数组。实现代码如下:

```
public class Test {
    public static int min(int a,int b,int c){
        int min = a < b? a:b;
        min = min < c? min:c;
        return min;
    }
    public static int max(int a,int b,int c){
        int max = a < b? b:a;
        max = max < c? c:max;
        return max;
    }
    public static int minDistance(int a[],int b[],int c[]){
        int aLen = a.length;
        int bLen = b.length;
        int cLen = c.length;
        int curDist = 0;
        int min = 0;
        int minDist = Integer.MAX_VALUE;
        int i = 0; //数组 a 的下标
        int j = 0; //数组 b 的下标
        int k = 0; //数组 c 的下标
        while(true){
            curDist = max(Math.abs(a[i] - b[j]),Math.abs(a[i] - c[k]),Math.abs(b[j] - c[k]));
            if(curDist < minDist)
                minDist = curDist;
            //找出当前遍历到3个数组中的最小值
            min = min(a[i],b[j],c[k]);
            if(min == a[i]){
                if(++i >= aLen)
                    break;
            }
            else if(min == b[j]){
                if(++j >= bLen)
                    break;
            }
            else{
                if(++k >= cLen)
                    break;
            }
        }
    }
}
```

```

return minDist;

public static void main(String[] args) {
    int a[] = {3,4,5,7};
    int b[] = {10,12,14,16,17};
    int c[] = {20,21,23,24,37,30};
    System.out.println(minDistance(a,b,c));
}
}

```

采用这种算法最多只需对 3 个数组分别遍历一遍，因此时间复杂度为 $O(1+m+n)$ 。

8.6 字符串

8.6.1 如何实现字符串的反转

问题描述：把一个句子中的单词进行反转，例如，“how are you”，进行反转后为“you are how”。

这道题的解决方法比较简单，只需要进行两次字符反转的操作即可，第一次对整个字符串中的字符进行反转，反转结果为：“uoy era woh”，通过这一次的反转已经实现了单词顺序的反转，只不过每个单词中字符的顺序反了，接下来只需要对每个单词进行字符反转即可得到想要的结果：“you are how”。实现代码如下：

```

public class Test {
    public void swap(char[] cArr,int front,int end){
        while(front < end){
            char tmp = cArr[end];
            cArr[end] = cArr[front];
            cArr[front] = tmp;
            front++;
            end--;
        }
    }

    public String swapWords(String s){
        char[] cArr = s.toCharArray();
        //对整个字符串进行字符反转操作
        swap(cArr,0,cArr.length-1);
        int begin=0;
        //对每个单词进行字符反转操作
        for(int i=1;i<cArr.length;i++){
            if(cArr[i]==" "){
                swap(cArr,begin,i-1);
                begin=i+1;
            }
        }
        swap(cArr,begin,cArr.length-1);
        return new String(cArr);
    }
}

```

```

    }
    public static void main(String[] args) {
        String str = "how are you";
        System.out.println(new Test().swapWords(str));
    }
}

```

程序运行结果为：

```
you are how
```

当然，在 Java 语言中，许多内置的类库可以使字符反转操作更加简单，有兴趣的读者可以自己研究一下。

8.6.2 如何判断两个字符串是否由相同的字符组成

问题描述：由相同的字符组成是指组成两个字符串的字母以及各个字母的个数是一样的，只是排列顺序不同而已，例如，“aaaabbc”与“abcbaaa”就由相同的字符组成的。下面讲述判断给定的两个字符串是否由相同的字符组成的方法。

方法一：排序法。最容易想到的方法就是对两个字符串中的字符进行排序，比较两个排序后的字符串是否相等。若相等，则表明它们是由相同的字符组成的，否则，则表明它们是由不同的字符组成的。实现代码如下：

```

import java.util. Arrays;

public class Test {

    public static void compare(String s1,String s2) {
        byte[] b1 = s1.getBytes();
        byte[] b2 = s2.getBytes();
        Arrays.sort(b1);
        Arrays.sort(b2);
        s1 = new String(b1);
        s2 = new String(b2);
        if(s1.equals(s2))
            System.out.println("equal");
        else
            System.out.println("not equal");
    }

    public static void main(String[] args) {
        String s1 = "aaaabbc";
        String s2 = "abcbaaa";
        compare(s1,s2);
        s1 = "aaaabbc";
        s2 = "abcbaab";
        compare(s1,s2);
    }
}

```

程序运行结果为：

```
equal
```

not equal

以上方法的时间复杂度取决于排序算法的时间复杂度，由于最快的排序算法的时间复杂度为 $O(n \log n)$ ，因此，该方法的时间复杂度也为 $O(n \log n)$ 。

方法二：空间换时间。在算法设计中，经常会采用空间换时间的方法以降低时间复杂度，即通过增加额外的存储空间来达到优化算法的效果。就本题而言，假设字符串中只使用 ASCII 字符，由于 ASCII 字符共有 266 个（对应的编码为 0 ~ 255），在实现时可以通过申请大小为 266 的数组来记录各个字符出现的个数，并初始化为 0，然后遍历第一个字符串，将字符串中字符对应的 ASCII 码值作为数组下标，把对应数组的元素加 1，然后遍历第二个字符串，把数组中对应的元素值 - 1。如果最后数组中各个元素的值都为 0，说明这两个字符串是由相同的字符组成的；否则，说明这两个字符串是由不同的字符组成的。示例代码如下：

```
public class Test {
    public static void compare( String s1, String s2 ) {
        byte[] b1 = s1.getBytes();
        byte[] b2 = s2.getBytes();
        int[] bCount = new int[256];
        for( int i = 0; i < 256; i ++ ) {
            bCount[i] = 0;
        }
        for( int i = 0; i < b1.length; i ++ )
            bCount[ b1[i] - '0' ] ++;
        for( int i = 0; i < b2.length; i ++ )
            bCount[ b2[i] - '0' ] --;
        for( int i = 0; i < 256; i ++ )
            if( bCount[i] != 0 ) {
                System.out.println( " not equal" );
                return;
            }
        System.out.println( " equal" );
    }
    public static void main( String[] args ) {
        String s1 = "aaaabbc";
        String s2 = "abcbaaa";
        compare( s1, s2 );
        s1 = "aaaabbc";
        s2 = "abcbaab";
        compare( s1, s2 );
    }
}
```

程序运行结果为：

```
equal
not equal
```

与方法一相比，方法二多申请了额外的存储空间，但是提高了算法的效率，该算法的时间复杂度为 $O(n)$ 。

8.6.3 如何删除字符串中重复的字符

问题描述：删除字符串中重复的字符，例如，“good”去掉重复的字符后就变为“god”。

方法一：“蛮力”法。最简单的方法就是把这个字符串看作一个字符数组，对该数组使用双重循环进行遍历，如果发现有重复的字符，就把该字符置为‘\0’，最后再把这个字符数组中的所有‘\0’去掉，此时得到的字符串就是删除重复字符后的目标字符串。示例如下：

```
public class Test {
    public static String removeDuplicate(String str) {
        char[] c = str.toCharArray();
        int len = c.length;
        for (int i = 0; i < len; i++) {
            if (c[i] == '\0')
                continue;
            for (int j = i + 1; j < len; j++) {
                if (c[j] == '\0')
                    continue;
                //把重复的字符置为'\0'
                if (c[i] == c[j])
                    c[j] = '\0';
            }
        }
        int l = 0;
        //去掉'\0'
        for (int i = 0; i < len; i++) {
            if (c[i] != '\0')
                c[l++] = c[i];
        }
        return new String(c, 0, l);
    }

    public static void main(String[] a) {
        String str = "abcaabcd";
        str = removeDuplicate(str);
        System.out.println(str);
    }
}
```

程序运行结果为：

```
abcd
```

由于这个方法使用了双重循环对字符数组进行了遍历，因此算法的时间复杂度为 $O(n^2)$ ，其中 n 是指字符串的长度。那么是否还有其他效率更高的方法？

方法二：空间换时间。在算法中经常会采用空间换时间的方法。对于这个问题，也可以采取这种方法。其主要思路如下：由于常见的字符只有 256 个，可以假设这道题字符串中不同的字符个数最多为 256 个，那么可以申请一个大小为 256 的 `int` 类型的数组来记录每个字符出现的次数，初始化都为 0，把这个字符的编码作为数组的下标，在遍历字符数组时，如果这个字符出现的次数为 0，那么把它置为 1；如果这个字符出现的次数为 1，说明这个字符在前面已

经出现过了，就可以把这个字符置为 ‘\0’，最后去掉所有 ‘\0’，就实现了去重的目的。采用这种方法只需要对字符数组进行一次遍历即可，因此时间复杂度为 $O(n)$ ，但是需要额外申请 256 大小的空间。由于申请的数组用来记录一个字符是否出现，只需要 1bit 就能实现这个功能，因此作为更好的一种方案，可以只申请大小为 8 的 int 类型的数组，由于每个 int 类型占 32bit，因此大小为 8 的数组总共为 256bit，用 1bit 来表示一个字符是否已经出现过可以达到同样的目的，示例如下：

```
public class Test {
    public static String removeDuplicate(String str) {
        char[] c = str.toCharArray();
        int len = c.length;
        int[] flags = new int[8]; // 只需要 8 个 32bit 的 int, 8 * 32bit = 256bit
        int i;
        for(i = 0; i < 8; i++)
            flags[i] = 0;
        for(i = 0; i < len; i++) {
            int index = (int)c[i]/32;
            int shift = (int)c[i] % 32;
            if((flags[index] & (1 << shift)) != 0)
                c[i] = '\0';
            flags[index] |= (1 << shift);
        }
        int l = 0;
        for(i = 0; i < len; i++) {
            if(c[i] != '\0')
                c[l++] = c[i];
        }
        return new String(c, 0, l);
    }
    public static void main(String[] a) {
        String str = "abcaabcd";
        str = removeDuplicate(str);
        System.out.println(str);
    }
}
```

程序运行结果为：

```
abcd
```

方法三：正则表达式。在 Java 语言中，利用正则表达式也可以达到同样的目的：(?.s)(.)(?. =. * \1)。

关于正则表达式可以参考专门的书籍，本书不详细叙述。

由于这种方法与方法一的思想类似，只不过方法一中去掉了字符串中后面出现的重复字符，而这种方法去掉了前面出现的重复字符，因此通过正则表达式进行全局替换后，字符串中字符的出现顺序将会和原字符串中出现的顺序不一致，可以通过对字符串进行反转，然后使用正则表达式替换，最后对字符串再进行反转，从而实现字符串的去重，示例如下：

```

public class Test {
    public static String reverse(String str) {
        StringBuffer sb = new StringBuffer(str);
        sb = sb.reverse();
        return sb.toString();
    }
    public static String removeDuplicate(String str) {
        str = reverse(str);
        str = str.replaceAll("(? s)(.)(? =. * \\1)", "");
        str = reverse(str);
        return str;
    }
    public static void main(String[] a) {
        String str = "abcaabcd";
        str = removeDuplicate(str);
        System.out.println(str);
    }
}

```

程序运行结果为：

```
abcd
```

8.6.4 如何统计一行字符中有多少个单词

单词的数目可以由空格出现的次数决定（连续的若干个空格作为出现一次空格；一行开头的空格不统计在内）。若测出某一个字符为非空格，而它的前面的字符是空格，则表示“新的单词开始了”，此时使单词计数器 count 值加 1；若当前字符为非空格而其前面的字符也是非空格，则意味着仍然是原来那个单词的继续，count 值不应再累加 1。前面一个字符是否空格可以从 word 的值看出来，若 word 等于 0，则表示前一个字符是空格；若 word 等于 1，意味着前一个字符为非空格。

示例如下：

```

public class Test {
    public static int wordCount(String s) {
        int word = 0;
        int count = 0;
        for(int i = 0; i < s.length(); i++) {
            if(s.charAt(i) == " ") {
                word = 0;
            } else if(word == 0) {
                word = 1;
                count++;
            }
        }
        return count;
    }
    public static void main(String[] args) {
        String s = "i am hehao";
    }
}

```

```

        System.out.println(s);
        System.out.println("单词个数为:" + wordCount(s));
    }
}

```

程序运行结果为:

```
i am hehao 单词个数为:3
```

8.6.5 如何按要求打印数组的排列情况

问题描述: 针对 1、2、2、3、4、5 这 6 个数字, 写一个函数, 打印出所有不同的排列, 例如 512234、215432 等, 要求“4”不能在第三位, “3”与“5”不能相连。

打印数组的排列组合方式最简单的方法就是递归, 但本题存在两个难点: 第一, 数字中存在重复数字; 第二, 明确规定了某些位的特性。采用常规的求解方法似乎不能完全适用了。

其实, 可以换一种思维, 把求解这 6 个数字的排列组合问题转换为大家都熟悉的图的遍历问题, 解答起来就容易多了。可以把 1、2、2、3、4、5 这 6 个点看作图的 6 个结点, 对 6 个结点两两相连可以组成一个无向连通图, 这 6 个数字对应的全排列等价于从这个图中各个结点出发深度遍历这个图所有可能路径所组成的数字集合, 例如, 从结点 1 出发的所有遍历路径组成了以 1 开头的所有数字的组合。由于“3”与“5”不能相连, 因此在构造图时使图中 3 和 5 对应的结点不连通就可以满足这个条件。对于“4”不能在第三位, 可以在遍历结束后判断是否满足这个条件。

具体而言, 实现步骤如下所示。

1) 用 1、2、2、3、4、5 这 6 个数字作为 6 个结点, 构造一个无向连通图。除了“3”与“5”不连通外, 其他所有结点都两两相连。

2) 分别从这 6 个结点出发对图做深度优先遍历。每次遍历完所有结点, 把遍历的路径对应数字的组合记录下来, 若这个数字的第三位不是“4”, 则把这个数字存放到集合 Set 中 (由于这 6 个数中有重复的数, 因此最终的组合肯定也会有重复的。由于集合 Set 的特点为集合中的元素是唯一的, 不能有重复的元素, 因此通过把组合的结果放到 Set 中可以过滤掉重复的组合)。

3) 遍历 Set 集合, 打印出集合中的所有结果, 这些结果就是本问题的答案。

实现代码如下:

```

import java.util.*;

public class Test {
    private int[] numbers = new int[] {1,2,2,3,4,5};
    private int n = numbers.length;
    //用来标记图中结点是否被遍历过
    private boolean[] visited = new boolean[n];
    //图的二维数组表示
    private int[][] graph = new int[n][n];
    //数字的组合
    private String combination = "";
    public Set<String> getAllCombinations() {
        // 构造图
        buildGraph();
    }
}

```

```

// 用来存放所有组合
Set <String> set = new HashSet <String> ();
// 分别从不同的结点出发深度遍历图
for(int i=0;i < n;i ++ ) {
    this. depthFirstSearch(i, set);
}
return set;
}

private void buildGraph() {
    for(int i=0;i < n;i ++ ) {
        for(int j=0;j < n;j ++ ) {
            if(i == j) {
                graph[i][j] = 0;
            } else {
                graph[i][j] = 1;
            }
        }
    }

    // 确保在遍历时 3 与 5 是不可达的
    graph[3][5] = 0;
    graph[5][3] = 0;
}

//对树从结点 start 位置开始进行深度遍历
private void depthFirstSearch(int start, Set <String> set) {
    visited[ start ] = true;
    combination = combination + numbers[ start ];
    if(combination. length() == n) {
        // 4 不出现在第三个位置
        if( combination. indexOf("4") != 2)
            set. add(combination);
    }
    for(int j=0;j < n;j ++ ) {
        if(graph[ start ][j] == 1 && visited[j] == false)
            depthFirstSearch(j, set);
    }
    combination = combination. substring(0, combination. length() - 1);
    visited[ start ] = false;
}

public static void main(String[] args) {
    Test t = new Test();
    Set <String> set = t. getAllCombinations();
    Iterator <String> it = set. iterator();
    while(it. hasNext()) {
        String string = (String) it. next();
        System. out. println(string);
    }
}
}

```

由于结果过多，这里就不列出详细的运行结果了。

8.6.6 如何输出字符串的所有组合

问题描述：假设字符串中的所有字符都不重复，如何输出字符串的所有组合？例如，输入字符串“abc”，则输出 a、b、c、ab、ac、bc、abc，共 7 种组合。

根据题意，如果字符串中有 n 个字符，根据排列组合的性质，此时一共需要输出 $2^n - 1$ 种组合。

最容易想到的方式是递归法，遍历字符串，每个字符只能取或不取。若取该字符，就把它放到结果字符串中，遍历完毕后，输出结果字符串。

程序代码如下：

```
public class Test {
    public static void CombineRecursiveImpl(char[] c,int begin,int len,StringBuffer sb) {
        if(len == 0) {
            System.out.print(sb + " ");
            return;
        }
        if(begin == c.length) {
            return;
        }
        sb.append(c[begin]);
        CombineRecursiveImpl(c,begin + 1,len - 1,sb);
        sb.deleteCharAt(sb.length() - 1);
        CombineRecursiveImpl(c,begin + 1,len,sb);
    }
    public static void main(String[] args) {
        String s = "abc";
        char[] c = s.toCharArray();
        StringBuffer sb = new StringBuffer("");
        int len = c.length;
        for(int i = 1;i <= len;i++) {
            CombineRecursiveImpl(c,0,i,sb);
        }
    }
}
```

程序输出为：

```
a b c ab ac bc abc
```

采用递归法求解，当 n 的值不是很大时，不存在效率低下的问题，但当 n 比较大时，效率会变得很差，因为栈调用次数约为 2^n ，尾递归优化后也有 $2^{(n-1)}$ 。为了提高效率，考虑到本题的特性，可以构造一个长度为 n 的 01 字符串（或二进制数）表示输出结果中是否包含某个字符，例如，“001”表示输出结果中不含字符 a、b，只含 c，即输出结果为 c，而“101”，表示输出结果为 ac。原题就是要求输出“001”到“111”这 $2^n - 1$ 个组合对应的字符串。

程序代码如下：

```
public class Test {
```



```

public static void Combine(char[] c) {
    if (c == null)
        return;
    int len = c.length;
    boolean used[] = new boolean[len];
    char cache[] = new char[len];
    int result = len;
    while (true) {
        int index = 0;
        while (used[index]) {
            used[index] = false;
            ++result;
        }
        if (++index == len)
            return;
        used[index] = true;
        cache[--result] = c[index];
        System.out.print(new String(cache).substring(result) + " ");
    }
}

public static void main(String[] args) {
    String s = "abc";
    char[] c = s.toCharArray();
    Combine(c);
}

```

程序运行结果为:

```
a b ab c ac bc abc
```

8.7 二叉树

二叉树是一种非常常见并实用的数据结构，它结合了有序数组与链表的优点，在二叉树中查找数据与在数组中查找数据一样快，在二叉树中添加删除数据的速度也与在链表中一样高效，所以，有关二叉树的相关技术一直是程序员面试笔试中必考的知识点。

8.7.1 二叉树基本概念

二叉树 (Binary Tree) 也称为二分树、二元树、对分树等，它是 $n(n \geq 0)$ 个有限元素的集合，该集合或者为空、或者由一个称为根 (root) 的元素及两个不相交的、被分别称为左子树和右子树的二叉树组成。当集合为空时，该二叉树被称为空二叉树。

在二叉树中，一个元素也称作一个结点。二叉树的递归定义为：二叉树或者是一棵空树，或者是一棵由一个根结点和两棵互不相交的分别称作根结点的左子树和右子树所组成的非空树，左子树和右子树又同样是一棵二叉树。

以下是一些常见的二叉树的基本概念：

1) 结点的度。结点所拥有子树的个数称为该结点的度。

2) 叶结点。度为 0 的结点称为叶结点，或者称为终端结点。

3) 分枝结点。度不为 0 的结点称为分枝结点，或者称为非终端结点。一棵树的结点除叶结点外，其余的都是分枝结点。

4) 左孩子、右孩子、双亲。树中一个结点的子树的根结点称为这个结点的孩子。这个结点称为它孩子结点的双亲。具有同一个双亲的孩子结点互称为兄弟。

5) 路径、路径长度。如果一棵树的一串结点 n_1, n_2, \dots, n_k 有如下关系：结点 n_i 是 $n_i + 1$ 的父结点 ($1 \leq i < k$)，就把 n_1, n_2, \dots, n_k 称为一条由 n_1 至 n_k 的路径。这条路径的长度是 $k - 1$ 。

6) 祖先、子孙。在树中，如果有一条路径从结点 M 到结点 N，那么 M 就称为 N 的祖先，而 N 称为 M 的子孙。

7) 结点的层数。规定树的根结点的层数为 1，其余结点的层数等于它的双亲结点的层数加 1。

8) 树的深度。树中所有结点的最大层数称为树的深度。

9) 树的度。树中各结点度的最大值称为该树的度，叶子结点的度为 0。

10) 满二叉树。在一棵二叉树中，如果所有分枝结点都存在左子树和右子树，并且所有叶子结点都在同一层上，这样的一棵二叉树称作满二叉树。

11) 完全二叉树。一棵深度为 k 的有 n 个结点的二叉树，对树中的结点按从上至下、从左到右的顺序进行编号，如果编号为 i ($1 \leq i \leq n$) 的结点与满二叉树中编号为 i 的结点在二叉树中的位置相同，那么这棵二叉树被称为完全二叉树。完全二叉树的特点是：叶子结点只能出现在最下层和次下层，且最下层的叶子结点集中在树的左部。需要注意的是，满二叉树肯定是完全二叉树，而完全二叉树不一定是满二叉树。

二叉树的基本性质如下：

性质 1：一棵非空二叉树的第 i 层上最多有 2^{i-1} 个结点 ($i \geq 1$)。

性质 2：一棵深度为 k 的二叉树中，最多具有 $2^k - 1$ 个结点，最少有 k 个结点。

性质 3：对于一棵非空的二叉树，度为 0 的结点（即叶子结点）总是比度为 2 的结点多一个，即如果叶子结点数为 n_0 ，度数为 2 的结点数为 n_2 ，则有 $n_0 = n_2 + 1$ 。

证明：用 n_0 表示度为 0（叶子结点）的结点总数，用 n_1 表示度为 1 的结点总数， n_2 表示度为 2 的结点总数， n 表示整个完全二叉树的结点总数，则 $n = n_0 + n_1 + n_2$ ，根据二叉树和树的性质，可知 $n = n_1 + 2 * n_2 + 1$ （所有结点的度数之和 + 1 = 结点总数），根据两个等式可知 $n_0 + n_1 + n_2 = n_1 + 2 * n_2 + 1$ ，所以 $n_2 = n_0 - 1$ ，即 $n_0 = n_2 + 1$ ，所以答案为 1。

性质 4：具有 n 个结点的完全二叉树的深度为 $\lceil \log_2 n \rceil + 1$ 。

证明：根据性质 2，深度为 k 的二叉树最多只有 $2^k - 1$ 个结点，且完全二叉树的定义是与同深度的满二叉树前面编号相同，即它的总结点数 n 位于 k 层和 $k - 1$ 层满二叉树容量之间，即 $2^{k-1} - 1 < n \leq 2^k - 1$ 或 $2^{k-1} \leq n < 2^k$ ，三遍同时取对数，于是有 $k - 1 \leq \log_2 n < k$ ，因为 k 是整数，所以 $k = \lceil \log_2 n \rceil + 1$ 。

性质 5：对于具有 n 个结点的完全二叉树，如果按照从上至下和从左到右的顺序对二叉树中的所有结点从 1 开始顺序编号，则对于任意的序号为 i 的结点，有：①如果 $i > 1$ ，那么序号为 i 的结点的双亲结点的序号为 $i/2$ （其中“/”表示整除）；如果 $i = 1$ ，那么序号为 i 的结点是根结点，无双亲结点。②如果 $2i \leq n$ ，那么序号为 i 的结点的左孩子结点的序号为 $2i$ ；如果 $2i > n$ ，那么序号为 i 的结点无左孩子。③如果 $2i + 1 \leq n$ ，那么序号为 i 的结点的右孩子结点的序号为 $2i + 1$ ；如果 $2i + 1 > n$ ，那么序号为 i 的结点无右孩子。

此外,如果对二叉树的根结点从0开始编号,那么相应的*i*号结点的双亲结点的编号为 $(i-1)/2$,左孩子的编号为 $2i+1$,右孩子的编号为 $2i+2$ 。

例题1:一棵完全二叉树上有1001个结点,其中叶子结点的个数是多少?

分析:二叉树的公式: $n = n_0 + n_1 + n_2 = n_0 + n_1 + (n_0 - 1) = 2 * n_0 + n_1 - 1$ 。而在完全二叉树中, n_1 只能取0或1。如果 $n_1 = 1$,那么 $2 * n_0 = 1001$,可推出 n_0 为小数,不符合题意;如果 $n_1 = 0$,那么 $2 * n_0 - 1 = 1001$,则 $n_0 = 501$,所以答案为501。

例题2:如果根的层次为1,具有61个结点的完全二叉树的高度为多少?

分析:根据二叉树的性质,具有*n*个结点的完全二叉树的深度为 $\lceil \log_2(n+1) \rceil$,因此含有61个结点的完全二叉树的高度为 $\lceil \log_2(61+1) \rceil = 6$,即应该为6层,所以答案为6。

例题3:在具有100个结点的树中,其边的数目为多少?

分析:在一棵树中,除了根结点之外,每一个结点都有一条入边,因此总边数应该是 $100 - 1$,即99条,所以答案为99。

8.7.2 如何实现二叉排序树

二叉排序树又称二叉查找树。它或者是一棵空树,或者是具有下列性质的二叉树:①如果左子树不空,那么左子树上所有结点的值均小于它的根结点的值;②如果右子树不空,那么右子树上所有结点的值均大于它的根结点的值;③左、右子树也分别为二叉排序树。

由于二叉树具有有序的特定,因此在笔试或面试过程中经常会出现二叉排序树相关的题目。下面给出二叉排序树的实现代码:

```
class Node {
public int data;
public Node left;
public Node right;
public Node(int data) {
    this.data = data;
    this.left = null;
    this.right = null;
}
}

public class BinaryTree {
private Node root;
public BinaryTree() {
    root = null;
}
//将 data 插入到排序二叉树中
public void insert(int data) {
    Node newNode = new Node(data);
    if (root == null)
        root = newNode;
    else {
        Node current = root;
        Node parent;
        while (true) // 寻找插入的位置
        {
```

```

        parent = current;
        if (data < current.data) {
            current = current.left;
            if (current == null) {
                parent.left = newNode;
                return;
            }
        } else {
            current = current.right;
            if (current == null) {
                parent.right = newNode;
                return;
            }
        }
    }
}

//将数值输入构建二叉树
public void buildTree(int[] data) {
    for (int i = 0; i < data.length; i++) {
        insert(data[i]);
    }
}

//中序遍历方法递归实现
public void inOrder(Node localRoot) {
    if (localRoot != null) {
        inOrder(localRoot.left);
        System.out.print(localRoot.data + " ");
        inOrder(localRoot.right);
    }
}

public void inOrder() {
    this.inOrder(this.root);
}

//先序遍历方法递归实现
public void preOrder(Node localRoot) {
    if (localRoot != null) {
        System.out.print(localRoot.data + " ");
        preOrder(localRoot.left);
        preOrder(localRoot.right);
    }
}

public void preOrder() {
    this.preOrder(this.root);
}

//后序遍历方法递归实现
public void postOrder(Node localRoot) {
    if (localRoot != null) {
        postOrder(localRoot.left);
    }
}

```

```

        postOrder(localRoot.right);
        System.out.print(localRoot.data + " ");
    }
}

public void postOrder() {
    this.postOrder(this.root);
}

public static void main(String[] args) {
    BinaryTreeNode biTree = new BinaryTreeNode();
    int[] data = {2,8,7,4,9,3,1,6,7,5};
    biTree.buildTree(data);
    System.out.print("二叉树的中序遍历:");
    biTree.inOrder();
    System.out.println();
    System.out.print("二叉树的先序遍历:");
    biTree.preOrder();
    System.out.println();
    System.out.print("二叉树的后序遍历:");
    biTree.postOrder();
    System.out.println();
}

```

程序运行结果为:

```

二叉树的中序遍历:1 2 3 4 5 6 7 7 8 9
二叉树的先序遍历:2 1 8 7 4 3 6 5 7 9
二叉树的后序遍历:1 3 5 6 4 7 7 9 8 2

```

上述算法主要实现了二叉树的构建、插入新的结点和遍历。

8.7.3 如何层序遍历二叉树

可以使用队列来实现二叉树的层序遍历。其主要思路如下:先将根结点放入队列中,然后每次都从队列中取出一个结点打印该结点的值,若这个结点有子结点,则将其子结点放入队列尾,直到队列为空。实现代码如下:

```

public void layerTraverse() {
    if (this.root == null)
        return;
    Queue<Node> q = new LinkedList<Node>();
    q.add(this.root);
    while (!q.isEmpty()) {
        Node n = q.poll();
        System.out.print(n.data);
        System.out.print(" ");
        if (n.left != null)
            q.add(n.left);
        if (n.right != null)
            q.add(n.right);
    }
}

```



8.7.4 已知先序遍历和中序遍历，如何求后序遍历

一般数据结构都有遍历操作，根据需求的不同，二叉树一般有以下几种遍历方式：先序遍历、中序遍历、后序遍历和层序遍历。

(1) 先序遍历

如果二叉树为空，遍历结束。否则，第一步，访问根结点；第二步，先序遍历根结点的左子树；第三步，先序遍历根结点的右子树。

(2) 中序遍历

如果二叉树为空，遍历结束。否则，第一步，中序遍历根结点的左子树；第二步，访问根结点；第三步，中序遍历根结点的右子树。

(3) 后序遍历

如果二叉树为空，遍历结束。否则，第一步，后序遍历根结点的左子树；第二步，后序遍历根结点的右子树；第三步，访问根结点。

(4) 层次遍历

从二叉树的第一层（根结点）开始，从上至下逐层遍历，在同一层中，则按从左到右的顺序对结点逐个访问。

图 8-6 所示的二叉树的先序遍历产生的序列是 ABDHIEJCFG，中序遍历产生的序列是 HDIBJEAFCG，后序遍历产生的序列是 HIDJEBFGCA，层次遍历产生的序列是 ABCDEFGHIJ。

如果已知先序序列为 ABDECF，中序序列为 DBE AFC，如何求解后序序列呢？首先，由于先序遍历树的规则为根左右，因此可以得出先序遍历序列的第一个元素必为树的根结点，则 A 就为根结点。再看中序遍历为左根右，再根据根结点 A，可知左子树包含元素为 DBE，右子树包含元素为 FC。其次，递归求解左子树（左子树的先序为 BDE，中序为 DBE），递归求解右子树（即右子树的先序为 CF，中序为 FC）。如此递归到没有左右子树为止，所以树结构如图 8-7 所示。

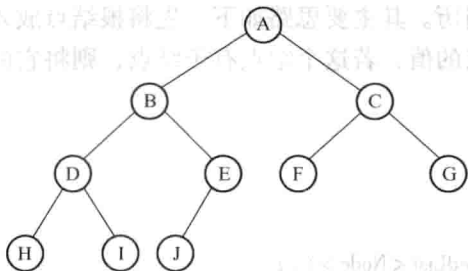


图 8-6 二叉树 (一)

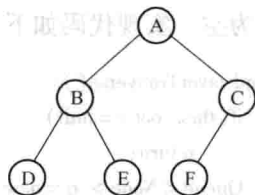


图 8-7 二叉树 (二)

通过上面的例子可以总结出用先序遍历和中序遍历来求解二叉树的过程，步骤如下：

1) 确定树的根结点。树根是当前树中所有元素在先序遍历中最先出现的元素，即先序遍历的第一个结点就是二叉树的根。

2) 求解树的子树。找到根在中序遍历的位置，位置左边是二叉树的左孩子，位置右边是二叉树的右孩子，如果根结点左边或右边为空，那么该方向子树为空；如果根结点左边和右边都为空，那么根结点已经为叶子结点。

3) 对二叉树的左、右孩子分别进行步骤1)、2), 直到求出二叉树结构为止。

具体实现代码如下:

```

class Node {
    public int data;
    public Node left;
    public Node right;
    public Node() {
    }
    public Node(int data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}

public class BinaryTree {
    private Node root;
    public BinaryTree() {
        root = null;
    }
    //后序遍历方法递归实现
    public void postOrder(Node localRoot) {
        if(localRoot != null) {
            postOrder(localRoot.left);
            postOrder(localRoot.right);
            System.out.print(localRoot.data + " ");
        }
    }
    public void postOrder() {
        this.postOrder(this.root);
    }
    public void initTree(int[] preOrder, int[] inOrder) {
        this.root = this.initTree(preOrder, 0, preOrder.length - 1, inOrder, 0,
            inOrder.length - 1);
    }
    public Node initTree(int[] preOrder, int start1, int end1, int[] inOrder,
        int start2, int end2) {
        if(start1 > end1 || start2 > end2) {
            return null;
        }
        int rootData = preOrder[start1];
        Node head = new Node(rootData);
        // 找到根结点所在的位置
        int rootIndex = findIndexInArray(inOrder, rootData, start2, end2);
        int offSet = rootIndex - start2 - 1;
        // 构建左子树
        Node left = initTree(preOrder, start1 + 1, start1 + 1 + offSet,
            inOrder, start2, start2 + offSet);
        // 构建右子树

```

```

        Node right = initTree(preOrder, start1 + offset + 2, end1, inOrder,
            rootIndex + 1, end2);
        head.left = left;
        head.right = right;
        return head;
    }

    public int findIndexInArray(int[] a, int x, int begin, int end) {
        for(int i = begin; i <= end; i++) {
            if(a[i] == x)
                return i;
        }
        return -1;
    }

    public static void main(String[] args) {
        BinaryTree biTree = new BinaryTree();
        int[] preOrder = {1,2,4,8,9,5,10,3,6,7};
        int[] inOrder = {8,4,9,2,10,5,1,6,3,7};
        biTree.initTree(preOrder, inOrder);
        System.out.print("二叉树的后序遍历:");
        biTree.postOrder();
    }
}

```

程序运行结果为:

二叉树的后序遍历:8 9 4 10 5 2 6 7 3 1

8.7.5 如何求二叉树中结点的最大距离

问题描述: 结点的距离是指这两个结点之间边的个数。写一个程序求一棵二叉树中相距最远的两个结点之间的距离。

一般而言, 对二叉树的操作通过递归方法实现比较容易。求最大距离的主要思想如下: 首先, 求左子树距根结点的最大距离, 记为 `leftMaxDistance`; 其次, 求右子树距根结点的最大距离, 记为 `rightMaxDistance`, 那么二叉树中结点的最大距离 `maxDistance` 满足 $\text{maxDistance} = \text{leftMaxDistance} + \text{rightMaxDistance}$ 。

实现代码如下:

```

class Node {
    public int data;
    public Node left;
    public Node right;
    public int leftMaxDistance; //左子树距根结点的最大距离
    public int rightMaxDistance; //右子树距根结点的最大距离
    public Node(int data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}

```

```

public class BinaryTree {
    private int maxLen = 0;
    private int max(int a, int b) {
        return a > b ? a : b;
    }
    public void FindMaxDistance(Node root) {
        if (root == null)
            return;
        if (root.left == null)
            root.leftMaxDistance = 0;
        if (root.right == null)
            root.rightMaxDistance = 0;
        if (root.left != null)
            FindMaxDistance(root.left);
        if (root.right != null)
            FindMaxDistance(root.right);
        // 计算左子树中距离根节点的最大距离
        if (root.left != null)
            root.leftMaxDistance = max(root.left.leftMaxDistance, root.left.rightMaxDistance) + 1;
        // 计算右子树中距离根节点的最大距离
        if (root.right != null)
            root.rightMaxDistance = max(root.right.leftMaxDistance, root.right.rightMaxDistance) + 1;
        // 获取二叉树所有节点的最大距离
        if (root.leftMaxDistance + root.rightMaxDistance > maxLen) {
            maxLen = root.leftMaxDistance + root.rightMaxDistance;
        }
    }
}

```

8.8 其他

8.8.1 如何消除嵌套的括号

问题描述：给定一个如下格式的字符串 (1,(2,3),(4,(5,6),7))，括号内的元素可以是数字，也可以是另一个括号，实现一个算法以消除嵌套的括号，例如，把上面的表达式变成 (1,2,3,4,5,6,7)，若表达式有误，则报错。

从问题描述可以看出，这道题要求实现两个功能：一是判断表达式是否正确；二是消除表达式中嵌套的括号。对于判定表达式是否正确这个问题，可以从如下几个方面来入手：首先，表达式中只有数字、逗号和括号这几种字符，如果有其他字符出现则是非法表达式；其次，判断括号是否匹配，若碰到“(”，则把括号的计数器值加1；如果碰到“)”，此时再判断计数器值是否大于1，若是，则把计数器减1，否则为非法表达式。当遍历完表达式后，若括号计数器值为0，则说明括号是配对出现的，否则说明括号不配对，则表达式为非法表达式。

实现代码如下：

```

public class Test {
    public static String change_str( String s) {

```

```

String result = "(" ;
char [] ch = s.toCharArray();
int bracket_num = 0;
int i = 0;
while( i < ch.length ) {
    if( ch[i] == '(' ) {
        bracket_num ++ ;
    }
    else if( ch[i] == ')' )
        if( bracket_num > 0 )
            bracket_num -- ;
        else {
            System.out.println("Expression wrong!\n");
            return null;
        }
    else if( ch[i] == ',' ) {
        result + = ch[i ++ ];
        continue;
    }
    else if( ch[i] >= '0' && ch[i] <= '9' ) {
        result + = ch[i];
    }
    else {
        System.out.println("Expression wrong!\n");
        return null;
    }
    i ++ ;
}
if( bracket_num > 0 ) {
    System.out.println("Expression wrong!\n");
    return null;
}
result + = ')';
return result;
}

public static void main(String[] args) {
    String s = "(1,(2,3),(4,(5,6),7))";
    String result = change_str(s);
    if(result != null)
        System.out.println(result);
    s = "((1,(2,3),(4,(5,6),7))";
    result = change_str(s);
    if(result != null)
        System.out.println(result);
}
}

```

程序运行结果为：

(1,2,3,4,5,6,7)

Expression wrong!

8.8.2 如何不使用比较运算就可以求出两个数的最大值与最小值

通常来讲,在求两个数中的最大值或最小值时,最常用的方法就是比较大小。下面给出一种不需比较大小就可以求出两个数中的最大值与最小值的方法,该方法用到了一种比较巧妙的数学方法,即最大值 $\text{Max}(a,b) = (a+b + |a-b|)/2$, 最小值 $\text{Min}(a,b) = (a+b - |a-b|)/2$ 。当然,这种方法存在着一个问题,即当 a 与 b 的值非常大时,会出现数据溢出的情况,解决的办法就是在计算时把 a 与 b 的值转换为长整型,从而可以避免溢出的发生。示例如下:

```
public class Test {
    //可能会溢出
    public static int max(int a,int b){
        return(a+b+Math.abs(a-b))/2;
    }
    public static int min(int a,int b){
        return(a+b-Math.abs(a-b))/2;
    }
    //不会溢出
    public static int max1(int a,int b){
        return(a+b+Math.abs(a-b))/2;
    }
    public static int min1(int a,int b){
        return(a+b-Math.abs(a-b))/2;
    }
    public static void main(String[] args){
        System.out.println("max(3,5)="+max(3,5));
        System.out.println("min(3,5)="+min(3,5));
        System.out.println("max1(3,5)="+max1(3,5));
        System.out.println("min1(3,5)="+min1(3,5));
    }
}
```

程序运行结果为:

max(3,5) = 5

min(3,5) = 3

max1(3,5) = 5

min1(3,5) = 3

第9章

海量数据处理

计算机硬件的扩容确实可以极大地提高程序的处理速度，但考虑到其技术、成本等方面的因素，它并非一条“放之四海而皆准”的途径。而随着互联网技术的发展，云计算、物联网、移动通信技术的兴起，每时每刻，数以亿计的用户产生着数量巨大的信息，海量数据时代已经来临。由于通过对海量数据的挖掘能有效地揭示用户的行为模式，加深对用户需求的理解，提取用户的集体智慧，从而为研发人员决策提供依据，提升产品用户体验，进而占领市场，因此当前各大互联网公司研究都将重点放在了海量数据处理上，但是，只寄希望于硬件扩容是很难满足海量数据处理需要的，如何利用现有条件进行海量信息处理已经成为各大互联网公司亟待解决的问题。所以，海量信息处理日益成为当前程序员笔试面试中一个新的亮点。

9.1 问题分析

海量信息，即大规模数据。随着互联网技术的发展，互联网上的信息越来越多，如何从海量信息中提取有用信息成为当前互联网技术发展必须面对的问题。

从海量数据中提取信息，不同于从常规量级数据中提取信息，在海量信息中提取有用数据，会存在以下几个方面的问题：首先，数据量过大，数据中什么情况都可能存在，如果信息数量只有 20 条，人工可以逐条进行查找、比对，可是当数据规模扩展到上百条、数千条、数亿条，甚至更多时，仅仅通过手工已经无法解决，必须借助工具或者程序进行处理。其次，处理海量数据信息，除了要有良好的软硬件配置，还需要合理使用工具，合理分配系统资源，通常情况下，如果需要处理的数据量非常大，超过了 TB (1 TB = 1024 GB) 级，小型机、大型工作站是要考虑的，普通的计算机如果有好的方法也可以考虑，例如通过联机做成工作集群。最后，信息处理海量数据时，要求很高的处理方法和技巧，如何进行数据挖掘算法的设计以及如何进行数据的存储访问等都是研究的难点。

本节的重点将放在如何运用好的方法和技巧来进行海量数据信息处理。

9.2 基本方法

针对海量数据的处理，可以使用的方法非常多，常见的方法有 Hash 法、Bit-map 法、Bloom filter 法、数据库优化法、倒排索引法、外排序法、Trie 树、堆、双层桶法以及 MapReduce 法。

1. Hash 法

Hash 一般被称为散列, 它是以一种映射关系, 即给定一个数据元素, 其关键字为 key, 按一个确定的散列函数计算出 $\text{hash}(\text{key})$, 把 $\text{hash}(\text{key})$ 作为关键字 key 对应元素的存储地址 (或称散列地址), 再进行数据元素的插入和检索操作。简而言之, 散列函数就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。

散列表是具有固定大小的数组, 其中, 表长 (即数组的大小) 应该为质数。散列函数是用于关键字与存储地址之间的一种映射关系, 但是, 不能保证每个元素的关键字与函数值是一一对应的, 因为极有可能出现对应于不同的元素, 却计算出了相同的函数值, 冲突指的就是两个关键字映射到同一个存储地址的情况, 即对不同的关键字可能得到同一散列地址, 即 $\text{key1} \neq \text{key2}$, 而 $f(\text{key1}) = f(\text{key2})$ 。

散列函数一般应具备以下几个特点:

- 1) 运算应该尽可能简单。
- 2) 函数的值域必须在散列表的范围内。
- 3) 尽可能地减少冲突。

针对散列函数的这些特点, 在构建散列表时, 不仅要设定一个号的散列函数, 而且还要设定一种处理冲突的方法。常用散列函数的构建方法一般有以下几种:

(1) 直接寻址法

取关键字或关键字的某个线性函数值为散列地址, 即 $h(\text{key}) = \text{key}$ 或 $h(\text{key}) = a \cdot \text{key} + b$, 其中 a 和 b 均为整型常数, 这种散列函数叫作自身函数, 例如, 有一个人口数字统计表, 人的年龄取值范围为 $1 \sim 100$, 其中, 年龄作为关键字, 散列函数取关键字自身, 但这种方法效率比较低, 时间复杂度为 $O(1)$, 空间复杂度为 $O(n)$, n 为关键字的个数。

直接寻址法不会产生冲突, 但由于它没有压缩映像, 因此, 当关键字集合很大时, 使用这种 Hash 函数是不可能实现地址编码的散列的。

(2) 取模法

选择一个合适的正整数 p, 令 $\text{hash}(\text{Key}) = \text{Key} \bmod p$ 。p 如果选择的是比较大的素数, 则效果比较好, 一般选取 p 为 TableSize, 即散列表的长度。

(3) 数字分析法

设关键字是 d 位的以 r 为基的数 (例如以 10 为基的十进制数), 且共有 n 个关键字, 则关键字的每个位可能有 r 个不同的数符出现 (即 0, 1, 2, ..., 9), 但这 r 个数符在各个位上出现的频率不一定相同, 可能在某些位上分布比较均匀, 即每个数符出现的次数接近于 n/r , 而在另一些位上分布不均匀。因此可选取其中分布比较均匀的那些位, 重新组成新的数, 用其作为散列地址。

这种方法比较简单、直观, 但是需要预先知道每个关键字的情况, 这就限制了它的使用范围。

(4) 折叠法

将关键字分成位数为 t 的几个部分 (最后一部分的位数可能小于 t), 然后把各部分按位对齐进行相加, 将所得的和舍弃进位, 留下 t 位作为散列地址。当关键字位数很多, 而且关键字中每位上数字分布比较均匀时, 采用折叠法比较合适。

(5) 平方取中法

这是一种较常用的方法，将关键字进行平方运算，然后从结果的中间取出若干位（位数与散列地址的位数相同），将其作为散列地址，具体取几位，由散列表的表长决定。

(6) 除留余数法

除留余数法是一种比较常用的散列函数，其主要原理是取关键字除以某个数 p (p 不大于散列表的长度 (TableSize)) 的余数作为散列地址，即

$$\text{Hash}(\text{key}) = \text{key} \% p$$

使用除留余数法时，选取合适的 p 值很重要，一般要求 $p \leq \text{TableSize}$ ，且接近 TableSize 或等于 TableSize， p 一般选取质数，也可以是不包含小于 20 质因子的合数。

(7) 随机数法

选择一个随机函数，然后用关键字 key 的随机函数值作为散列地址，即

$$\text{Hash}(\text{key}) = \text{random}(\text{key});$$

其中， $\text{random}()$ 为随机函数。当关键字的长度不相等时，采用这种方法比较合适。

在构造散列表的过程中，不管使用什么样的散列函数，冲突都是不可能完全避免的，所以解决冲突是构造散列表的一个必不可少的过程。解决冲突的主要途径是当一个关键字映射到散列表中的某一个地址，且该地址上已有关键字时，再为该关键字寻找新的存储地址。常用的解决冲突办法有以下几种：

(1) 开放地址法

开放地址法的基本思想是当发生地址冲突时，在散列表中再按照某种方法继续探测其他的存储地址，直到找到空闲的地址为止。该过程可描述为

$$H_i(\text{key}) = (H(\text{key}) + d_i) \bmod m \quad (i = 1, 2, \dots, k \ (k \leq m - 1))$$

其中， $H(\text{key})$ 为关键字 key 的直接散列地址， m 为散列表的长度， d_i 为每次再探测时的地址增量。

采用这种方法时，首先计算出关键字的直接散列地址，即 $H(\text{key})$ ，若该直接散列地址上已经有其他的关键字，则继续查看地址为 $[H(\text{key}) + d_i]$ 的存储地址，判断其是否为空。如此反复，直至找到空闲的存储地址为止，然后将关键字 key 存放该地址。

增量 d_i 可以有不同的取法，常用的有以下 3 种：

- 1) $d_i = 1, 2, 3, \dots, m - 1$ ，称为线性探测再散列。
- 2) $d_i = 12, -12, 22, -22, \dots, -k2 \ (k \leq m/2)$ ，称为二次探测再散列。
- 3) $d_i =$ 伪随机序列，称为伪随机再散列。

注意：对于利用开放地址法处理冲突所产生的散列表中，删除一个元素时不能直接删除，因为这样将会影响其他具有相同散列地址的元素的查找地址，所以，通常采用设定一个特殊的标志的方法表示该元素已经被删除。

(2) 链地址法

链地址法解决冲突的主要思想是：若散列表空间为 $[0, m - 1]$ ，则设置一个由 m 个指针组成的一维数组 $\text{CH}[m]$ ，然后在寻找关键字散列地址的过程中，所有散列地址为 i 的数据元素都插入到头指针为 $\text{CH}[i]$ 的链表中。

这种方法比较适合于冲突比较严重的情况下使用，例如，设有 8 个元素 $\{a, b, c, d, e, f, g, h\}$ ，采用某种散列函数得到的地址为： $\{0, 2, 4, 1, 0, 8, 7, 2\}$ ，当散列表长度为 10 时，采用链

地址法解决冲突的散列表如图 9-1 所示。

(3) 再散列法

当发生冲突时,使用第二个、第三个、散列函数计算地址,直到无冲突为止。但这种方法的缺点是计算时间会大幅增加。

(4) 建立一个公共溢出区

假设散列函数的值域为 $[0, m - 1]$,则设向量 HashTable $[0 \cdots m - 1]$ 为基本表,另外设立存储空间向量 OverTable $[0 \cdots v]$ 用以存储发生冲突的记录。

Hash 主要是用来进行“快速存取”,在 $O(1)$ 时间复杂度里就可以查找到目标元素,或者判断其是否存在。Hash 数据结构里的数据对外是杂乱无序的,因此其具体存储位置及各个存储元素位置之间的相互关系是无法得知的,但是却可以在常数时间里判断元素位置及存在与否。在处理海量数据的过程中,使用 Hash 方法一般可以快速存取、统计某些数据,将大量数据进行分类,例如提取某日访问网站次数最多的 IP 地址等。

2. Bit-map 法

Bit-map (位图)法的基本原理是使用位数组来表示某些元素是否存在,例如从 8 位电话号码中查重复号码,本法适用于海量数据的快速查找、判重、删除等。

具体而言, Bit-map 法的结果是生成一个 N 位长的串,每位上以“1”或“0”表示需要排序的集合(后简称“集合”)中的数,例如集合为 $\{2, 7, 4, 9, 1, 10\}$,则生成一个 10 位的串,将第 2、7、4、9、1、10 位置为“1”,其余位置为“0”,当把串中所有位都置完后,排序也自动完成了(因为字符串的下标是有序的),上例中用 Bit-map 法排序后的结果为 1101001011。

再如,要排序 $0 \sim 15$ 内的以下元素序列 $\{5, 8, 1, 12, 6, 2\}$,那么首先开辟 2 Byte 的空间,也就是 16 位,分别对应 $0 \sim 15$ 这 16 个数,并将这 16 位置为“0”。遍历序列,在出现的数字的对应位置上置“1”,也就是将每个元素对应到了位图的相应位置。再遍历这 16 位,就完成了对元素的排序。过程如图 9-2 所示。

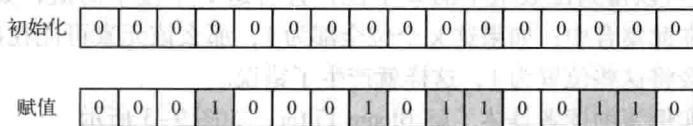


图 9-2 Bit-map 法初始化与赋值

Bit-map 法排序的时间复杂度是 $O(n)$,比一般的排序都快,但它是用空间换时间(需要一个 N 位的串)的,而且有一些限制,即数据状态不是很多,例如排序前集合大小最好已知,而且集合中元素的最大重复次数必须已知,最好是稠集数据(不然空间浪费很大)。

在程序设计中,经常会遇到判断集合中是否存在重复的问题,数据量比较小时,对时间复杂度可能要求并不高,但当集合中数据量比较大时,则希望能够少进行几次扫描,此时如果还采用双重循环法,效率就太低下了,显然不可取,而 Bit-map 法则比较适合于这种情况,首先扫描一遍集合,找出集合中的最大元素,然后按照集合中最大元素 max 创建一个长度为“max + 1”的新数组,接着再次扫描原数组,每遇到一个元素,就将新数组中下标为元素值的位置 1,例如,如果遇到元素 5,就将新数组的第 6 个元素置为 1,如此下去,当下次再遇到元

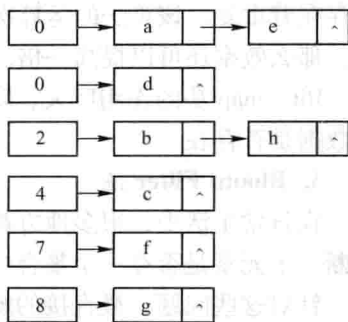


图 9-1 链地址法解决冲突法

素 5 想置位时，发现新数组的第 6 个元素已经被置为 1 了，则说明这次的数据肯定和以前的数据存在着重复。该算法的运算次数最坏的情况为 $2N$ ，但如果能够事先知道集合的最大元素的值，那么效率还可以提高一倍。

Bit-map 法的作用巨大，除了判断数据是否重复以外，也经常使用本法来判断集合中某个数据是否存在。

3. Bloom Filter 法

在日常生活中，很多地方都会遇到类似这样的问题：在设计计算机软件系统时，经常需要判断一个元素是否在一个集合中；在字处理软件中，需要检查一个英语单词是否拼写正确等。

针对这些问题，最直接的解决方法就是将集合中的全部元素都存储在计算机中，每当遇到一个新元素时，将它和集合中的元素直接进行比较即可。这种做法虽然能够准确无误地完成任任务，但存在一个问题，就是比较次数太多，效率低下，当数据量不大时，这种效率低的问题并不显著，但是当数据量巨大时，例如在海量数据信息处理中，效率低的问题就突显出来了，例如邮箱总是需要过滤垃圾邮件，一个办法就是记录下那些发垃圾邮件的 E-mail 地址，可是由于那些发送者还会不停地再注册新的地址，如果使用散列表，每存储一亿个 E-mail 地址，一般而言，每个 E-mail 地址需要占用 16B，所以一共需要 $1 \text{ 亿} * 16\text{B}$ ，大约 1.6GB 的内存，除非是超级计算机，一般服务器是无法存储如此海量信息的。

Bloom Filter 正是解决这一问题的有效方法，它是一种空间效率和时间效率都很高的随机数据结构，用来检测一个元素是否属于一个集合。但它同样带来一个问题——牺牲了正确率。Bloom Filter 以牺牲正确率为前提，来换取空间效率与时间效率的提高。当它判断某元素不属于这个集合时，该元素一定不属于这个集合；当它判断某元素属于这个集合时，该元素不一定属于这个集合。具体而言，查询结果有两种可能，即“不属于这个集合（绝对正确）”和“属于这个集合（可能错误）”。所以，Bloom Filter 法适用于对低错误率可以容忍的场合。

Bloom Filter 的基本原理是位数组与 Hash 函数的联合使用。首先，Bloom Filter 是一个包含了 m 位的位数组，数组的每一位都初始化为 0；其次，定义 k 个不同的 Hash 函数，每个函数都可以将集合中的元素映射到位数组的某一位。当向集合中插入一个元素时，根据 k 个 Hash 函数可以得到位数组中的 k 个位，将这些位设置为 1。如果查询某个元素是否属于集合，那么根据 k 个 Hash 函数可以得到位数组中的 k 个位，查看这 k 个位中的值，如果有的位不为 1，那么该元素肯定不在此集合中；如果这 k 个位全部为 1，那么该元素可能在此集合中。在插入其他元素时，可能会将这些位置为 1，这样就产生了错误。

下面通过一个实例帮助读者具体了解 Bloom Filter，如图 9-3 所示。

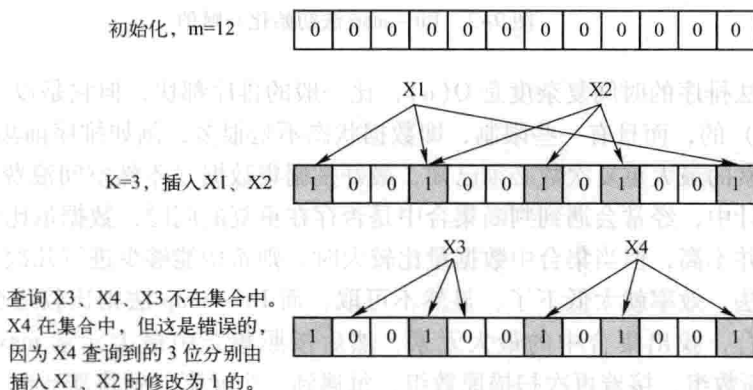


图 9-3 Bloom Filter 实例解析

所以,使用 Bloom Filter 法的难点是如何根据输入元素个数 n , 来确定位数组 m 的大小以及 Hash 函数。当 Hash 函数个数 $k = (\ln 2) * (m/n)$ 时错误率最小, 在错误率不大于 E 的情况下, m 至少要等于 $n * \lg(1/E)$ 才能表示任意 n 个元素的集合。但 m 还应该更大些, 因为还要保证位数组里至少一半为 0, 所以 m 应该 $\geq n \lg(1/E) * \lg e$, 大概就是 $n \lg(1/E)$ 的 1.44 倍 (\lg 表示以 2 为底的对数)。

假设 E 为 0.01, 即错误率为 0.01, 则此时 m 应该大约为 n 的 13 倍。这样 k 大约是 8 个 (注意: m 与 n 的单位不同, m 的单位是 bit, 而 n 则是以元素个数为单位)。通常单个元素的长度都是有几百 bit 的, 所以若使用 Bloom Filter 法, 内存上通常都是节省的。

Bloom Filter 的优点是具有很好的空间效率和时间效率。它的插入和查询时间都是常数, 另外, 它不保存元素本身, 具有良好的安全性。然而, 这些优点都是以牺牲正确率为代价的。当插入的元素越多, 错判“元素属于这个集合”的概率就越大。另外, Bloom Filter 只能插入元素, 却不能删除元素, 因为多个元素的散列结果可能共用了 Bloom Filter 结构中的同一个位, 如果删除元素, 就可能会影响多个元素的检测。所以 Bloom Filter 可以用来实现数据字典、进行数据的判重或者集合求交集。

CBF 与 SBF 是 Bloom Filter 的扩展, Counting Bloom Filter (CBF) 将位数组中的每一位扩展为一个 counter, 从而支持了元素的删除操作。Spectral Bloom Filter (SBF) 将其与集合元素的出现次数关联, SBF 采用 counter 中的最小值来近似表示元素的出现频率。

4. 数据库优化法

互联网上的数据一般都被存储在数据库中, 很多情况下, 人们并非对这些海量数据本身感兴趣, 而是需要从这些海量数据中提取出对自己有用的信息, 例如, 从数据中获取访问最多的页面信息等, 这就涉及了数据的查询技术等相关内容。

数据库管理软件的选择是否合理、表结构设计是否规范、索引创建是否恰当等都是影响数据库性能的重要因素, 所以, 对数据库进行优化, 是实现海量数据高效处理地有效方法之一。常见的数据库优化方法有如下几种:

(1) 优秀的数据库管理工具

选择一款优秀的数据库管理工具非常重要。现在的数据库工具厂家比较多, 对海量数据的处理对所使用的数据库工具要求比较高, 一般使用 Oracle、DB2、MySQL 等。

(2) 数据分区

进行海量数据的查询优化, 其中一种重要的方式就是如何有效地存储并降低需要处理的数据规模, 所以可以对海量数据进行分区操作提高效率, 例如, 针对按年份存取的数据, 可以按年进行分区, 不同的数据库有不同的分区方式, 不过处理机制却大体相同, 例如, SQL Server 的数据库分区是将不同的数据存于不同的文件组下, 而不同的文件组存于不同的磁盘分区下, 这样将数据分散开, 减小磁盘 I/O, 减小了系统负荷, 而且还可以将日志、索引等放于不同的分区下。

(3) 索引

索引一般可以加速数据的检索速度, 加速表与表之间的链接, 提高性能, 所以在对海量数据进行处理时, 考虑到信息量比较大, 应该对表建立索引, 包括在主键上建立聚簇索引, 将聚合索引建立在日期列上等。

索引的优点很多, 但是对于索引的建立, 还需要考虑到实际情况, 而不是对每一个列建立一个索引, 例如针对大表的分组、排序等字段, 都要建立相应索引, 同时还应该考虑建立复合索引。增加索引同时也有很多不利的方面: 首先, 创建索引和维护索引要耗费时间, 这种时间

随着数据量的增加而增加；其次，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大；最后，当对表中的数据进行增加、删除和修改时，索引也要动态地维护，这样就降低了数据的维护速度。

所以，索引要用到好的时机，索引的填充因子和聚集、非聚集索引都要考虑。

(4) 缓存机制

当数据量增加时，一般的处理工具都要考虑到缓存问题。缓存大小设置的好差也关系到数据处理的成败，例如，在处理 2 亿条数据聚合操作时，缓存设置为 100000 条/Buffer 可行。

(5) 加大虚存

由于系统资源有限，而需要处理的数据量非常大，因此当内存不足时，可以通过增加虚拟内存来解决。

(6) 分批处理

由于需要处理的信息量巨大，可以对海量数据进行分批处理（类似于云计算中的 MapReduce 思想），然后再对处理后的数据进行合并操作，分而治之，这样有利于小数据量的处理，不至于面对大数据量带来的问题。

(7) 使用临时表和中间表

数据量增加时，处理中要考虑提前汇总。这样做的目的是化整为零，大表变小表，分块处理完成后，再利用一定的规则进行合并，处理过程中临时表的使用和中间结果的保存都非常重要，如果对于超海量的数据，大表处理不了，只能拆分为多个小表。如果处理过程中需要多步汇总操作，可按汇总步骤一步一步来。

(8) 优化查询语句

查询语句的性能对查询效率的影响是非常大的，编写高效优良的 SQL 脚本和存储过程是数据库工作人员的职责，也是检验数据库工作人员水平的一个标准，在对 SQL 语句的编写过程中，例如减少关联，少用或不用游标，设计好高效的数据库表结构等都十分必要。

(9) 使用视图

视图中的数据来源于基本表，对海量数据的处理，可以将数据按一定的规则分散到各个基本表中，查询或处理过程可以基于视图进行。

(10) 使用存储过程

在存储过程中，尽量使用 SQL 自带的返回参数，而非自定义的返回参数，以减少不必要的参数，避免数据冗余。

(11) 用排序来取代非顺序存取

磁盘存取臂的来回移动使得非顺序磁盘存取变成了最慢的操作，但是在 SQL 语句中这个现象被隐藏了，这样就使得查询中进行了大量的非顺序页查询，降低了查询速度。

(12) 使用采样数据进行数据挖掘

基于海量数据的数据挖掘正在逐渐兴起，面对着超海量的数据，一般的挖掘软件或算法往往采用数据抽样的方式进行处理，这样的误差不会很高，大大提高了处理效率和处理的成功率。一般采样时要注意数据的完整性，防止过大的偏差。

5. 倒排索引法

倒排索引是目前搜索引擎公司对搜索引擎最常用的存储方式，也是搜索引擎的核心内容，在搜索引擎的实际应用中，有时需要按照关键字的某些值查找记录，所以是按照关键字建立索引，这个索引就被称为倒排索引。

倒排索引也常被称为反向索引、置入档案或反向档案，它本质上是一种索引方法，被用来

存储在全文搜索下某个单词在一个文档或者一组文档中的存储位置的映射，它是文档检索系统中最常用的数据结构，有两种不同的反向索引形式：第一种形式是一条记录的水平反向索引（或者反向档案索引）包含每个引用单词的文档的列表；第二种形式是一个单词的水平反向索引（或者完全反向索引）又包含每个单词在一个文档中的位置。第二种形式提供了更多的兼容性（例如短语搜索），但是需要更多的时间和空间来创建。

一般情况下可以采用矩阵的方式存储来存储，但会浪费大量的空间，例如，对于如下内容，

D1: The GDP increased.

D2: The text is this.

D3: My name is.

采用矩阵的方式存储的具体表示见表9-1，其中行表示关键词，列表示所有文件。

表9-1 矩阵方式存储表示

	D1	D2	D3
The	1	1	0
GDP	1	0	0
increased	1	0	0
text	0	1	0
is	0	1	1
name	0	0	1

而根据表9-1中的信息，就能得到下面的倒排索引：

The: { D1, D2 } ;

GDP: { D1 } ;

increased: { D1 } ;

Text: { D2 } ;

is: { D2, D3 } ;

Name: { D3 } .

通过比较发现，采用倒排索引比采用矩阵的方式节省很多的空间。

正向索引用来存储每个文档的单词的列表。正向索引的查询往往满足每个文档有序频繁的全文查询和每个单词在校验文档中的验证这样的查询。在正向索引中，文档占据了中心的位置，每个文档指向了一个它所包含索引项的序列。也就是说，文档指向了它包含的那些单词，而反向索引则是单词指向了包含它的文档，很容易看到这个反向的关系。而与正向索引相比，倒排索引的优点是在处理复杂的多关键字查询时，可在倒排表中先完成查询的并、交等逻辑运算，得到结果后再对记录进行存取，这样把对记录的查询转换为地址集合的运算，不必对每个记录随机存取，从而提高查找速度。所以倒排索引一般被应用于文档检索系统，查询那些文件包含了某单词，例如常见的学术论文的关键字搜索。

6. 外排序法

当待排序的对象数目特别多时，在内存中不能一次处理，必须把它们以文件的形式存放于外存，排序时再把它们一部分一部分地调入内存进行处理，这种方式就是外排序法。

外排序是相对内排序而言的，它是大文件的排序，待排序的记录存储在外存储器上，待排序的文件无法一次装入内存，需要在内存和外部存储器之间进行多次数据交换，以达到对整个

文件进行排序的目的。一般采用归并排序等方式实现外排序，主要分成两个步骤：第一步，生成若干初始归并段（顺串），也被称为文件预处理，把含有 n 个记录的文件，按内存大小划分为若干长度为 L 的子文件，然后分别将子文件调入内存，采用有效的内排序方法排序后送回外存；第二步，进行多路归并，即对这些初始归并段进行多遍归并，使得有序的归并段逐渐扩大，最后在外存上形成整个文件的单一归并段，也就是完成了文件的外排序。

外排序的适用于大数据的排序以及去重复。但外排序也存在着很大的缺陷，就是它会消耗大量的 IO，效率不会很高。

7. Trie 树

Trie 这个单词来自于“retrieve”，Trie 树又称字典树或键树，它是一种用于快速字符串检索的多叉树结构，其原理是利用字符串的公共前缀来减少时空开销，即以空间换时间，从而达到提高程序效率的目的。Trie 树的典型应用是用于统计和排序大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是：最大限度地减少无谓的字符串比较，查询效率比散列表高。

Trie 树一般具有 3 个基本特性：

- 1) 根节点不包含字符，除根节点外每一个结点都只包含一个字符。
- 2) 从根节点到某一结点，路径上经过的字符连接起来，为该结点对应的字符串。
- 3) 每个结点的所有子结点包含的字符都不相同。

Trie 树可以利用字符串的公共前缀来节约存储空间，如图 9-4 所示，该 Trie 树用 10 个结点保存了 5 个字符串 amy, ann, em, rob, rg。

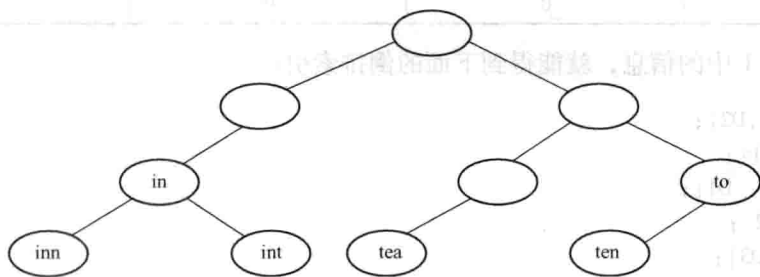


图 9-4 公共前缀图

在该 Trie 树中，字符串“amy”和“ann”有公共前缀“a”。当然，若系统中存在大量字符串且这些字符串基本没有公共前缀，则相应的 Trie 树将非常消耗内存，这也是 Trie 树的一个缺点。

例如，给一个单词 a，如果通过交换单词中字母的顺序可以得到另外的单词 b，那么称 b 是 a 的兄弟单词，例如单词 army 和 mary 互为兄弟单词。现在要给出一种解决方案，对于用户输入的单词，根据给定的字典找出输入单词有哪些兄弟单词。

一般情况下，Trie 树的结构都是采用 26 叉树进行组织的，每个结点对应一个字母，查找的时候，就是一个字母一个字母地进行匹配，算法的时间复杂度就是单词的长度 n ，效率很高。本例可以定义一个 Trie 树作为数据结构来查询，此时就转化为在一棵 Trie 树中查找兄弟单词，只要在 Trie 树中的前缀中再存储一个 vector 结构的容器，就可以大大降低时间复杂度。

具体求解兄弟单词的程序代码如下所示：

```
import java.util.*;
public class Test {
```



```

class TrieNode {
    Vector <String > bwords = new Vector <String > ();
    TrieNode next[] = new TrieNode[26];
    TrieNode() {
        for(int i=0;i<26;i++) {
            next[i] = null;
        }
    }
};
int CmpChar(char c1, char c2) {
    return(c1 - c2);
}
void InsertNode( TrieNode root, String wd) {
    if(wd. length() ==0) {
        return;
    }
    if(root == null) {
        root = new TrieNode( );
    }
    int i =0;
    char swd[] = wd. toCharArray( );
    Arrays. sort(swd);
    TrieNode next = root;
    while(i < wd. length()) {
        if(next. next[ swd[i] -'a' ] == null) {
            TrieNode nn = new TrieNode( );
            next. next[ swd[i] -'a' ] = nn;
        }
        next = next. next[ swd[i] -'a' ];
        i++;
    }
    next. bwords. add( wd);
}
boolean SearchNode( TrieNode root, String wd) {
    char swd[] = wd. toCharArray( );
    Arrays. sort(swd);
    int i =0;
    while(i < wd. length()) {
        if(root. next[ swd[i] -'a' ] != null) {
            root = root. next[ swd[i] -'a' ];
            i++;
        } else {
            break;
        }
    }
    if(i == wd. length()) {
        for(int j =0; j < root. bwords. size(); j++) {
            System. out. print( root. bwords. get(j) + " " );
        }
    }
}

```

```
        System.out.println();
        return true;
    }
    return false;
}

public void findBrother() {
    TrieNode root = new TrieNode();
    InsertNode(root, "hehao");
    InsertNode(root, "ehaoh");
    InsertNode(root, "haohe");
    InsertNode(root, "aohheh");
    InsertNode(root, "facri");
    InsertNode(root, "et");
    SearchNode(root, "oheha");
}

public static void main(String[] args) {
    new Test().findBrother();
}
}
```

程序运行结果为：

```
hehao ehaoh haohe aohheh
```

上例中，Trie 树的构建是在预处理阶段完成的，先根据字典中的单词来建立字典树，这样查询兄弟单词的效率就会提高很多，比 hash 法效率还要高。

Trie 树适用于数据量大、重复多，但是数据种类小可以放入内存的情况，例如已知 n (n 很大) 个由小写字母构成的平均长度为 10 的单词，判断其中是否存在某个字符串是另一个字符串的前缀子串。针对这种问题，一般可以采用如下 3 种方法。

1) 迭代法。对于每一个单词，都要去查找其前面的单词中是否包含它，看每个字符串是否为字符串集中某个字符串的前缀，由于需要不停地迭代比较，因此此时的时间复杂度为 $O(n^2)$ 。

2) Hash 法。使用 Hash 方法存储所有字符串的所有前缀子串。建立存有子串 Hash 的时间复杂度为 $O(n * len)$ ，查询的复杂度为 $O(n) * O(1) = O(n)$ 。

3) Trie 树。假设要查询的单词是 abcd，那么在它前面的单词中，以 b, c, d, f 之类开头的单词则不必考虑，而只要找以 a 开头的单词中是否存在 abcd 就可以了。同样在以 a 开头的单词中，只要考虑以 b 作为第二个字母的单词即可，所以建立 Trie 树的复杂度为 $O(n * len)$ ，而建立操作与查询操作在 Trie 树中是可以同时执行的。所以总的复杂度为 $O(n * len)$ ，实际查询的复杂度只是 $O(len)$ ，例如有串：911, 911456 输入，如果要同时执行建立与查询，过程如下：先查询 911，没有；存入 9、91、911，再查询 911456，没有；然后存入 9114、91145、911456，而程序没有记忆功能，并不知道 911 在输入数据中出现过，所以使用 hash 法必须先存入所有子串，然后进行 for 循环查询。而 Trie 树则可以，存入 911 后，已经将 911 记录为出现的字符串，在存入 911456 的过程中就能发现并输出答案，反过来也可以，先存入 911456，再存入 911 时，当指针指向最后一个 1 时，程序会发现这个 1 已经存在，说明 911 必定是某个字符串的前缀。

8. 堆

堆是一种树形数据结构，每个结点都有一个值，而通常所说的堆，一般是指二叉堆。在堆中，以大顶堆为例，堆的根结点的值最大，且根结点的两个子树也是一个大顶堆，基于以上特点，堆适用于海量数据求前 N 大（用小顶堆）或者前 N 小（用大顶堆）数问题，其中 N 一般比较小。例如当求海量数据前 N 小的数据时，使用大顶堆，比较当前元素与大顶堆的最大元素（即堆顶元素），若该元素小于最大元素，则应该替换该最大元素，并调整堆的结构（具体过程见 13.5.7 堆排序一节内容）。当求海量数据前 N 大的数据时，思路一样。由于采用堆，只需要扫描一遍即可得到所有前 n 元素，因此在海量信息处理中，效率非常高。

在海量数据处理中，堆的作用见表 9-2。

表 9-2 堆及其描述

	描 述	堆 类 型	作 用
堆	求海量数据中前 n 大/小的值中位数	最大堆	求前 n 小
		最小堆	求前 n 大
		双堆	中位数

9. 双层桶法

双层桶不是一种数据结构，而是一种算法思想，类似于分治思想。因为元素范围很大，不能利用直接寻址表，所以通过多次划分，逐步确定范围，然后最后在一个可以接受的范围内进行。

本文以桶排序进行分析，桶排序的基本思想是把 $[0,1)$ 划分为 n 个大小相同的子区间，每一子区间是一个桶，然后将 n 个记录分配到各个桶中。因为关键字序列是均匀分布在 $[0,1)$ 上的，所以一般不会有多个记录落入同一个桶中。由于同一桶中的记录其关键字不尽相同，因此必须采用关键字比较的排序方法（通常用插入排序）对各个桶进行排序，然后依次将各非空桶中的记录连接（收集）起来即可。这种排序思想的前提是假设输入的 n 个关键字序列是随机分布在区间 $[0,1)$ 之上，如果关键字序列的取值范围不是该区间，只要其取值均非负，总能将所有关键字除以某一合适的数，将关键字映射到该区间上，但要保证映射后的关键字是均匀分布在 $[0,1)$ 上的。

桶排序的平均时间复杂度是 $O(n)$ ，最坏情况仍有可能是 $O(n^2)$ ，一般只适用于关键字取值范围较小的情况，否则所需桶的数目 m 太多导致浪费存储空间和计算时间，例如， $n=10$ ，被排序的记录关键字 k_i 是 $0 \sim 99$ 的整数（36,5,16,98,95,47,32,36,48）时，要用 100 个箱子来做一趟排序。

下面是一个桶排序的示例：

```
public class Test {
    class Node {
        int key;
        Node next;
    };
    void IncSort(int keys,int bucketsize) {
        int size = keys.length;
        Node bucket_table = new Node[bucketsize];
        for (int i=0; i<bucketsize; i++) {
            bucket_table[i] = new Node();
        }
    }
}
```

```

        bucket_table[i].key = 0; // 记录当前桶中的数据量
        bucket_table[i].next = null;
    }
    for (int j = 0; j < size; j++) {
        Node node = new Node();
        node.key = keys[j];
        node.next = null;
        int index = keys[j] / 10;
        Node p = bucket_table[index];
        if (p.key == 0) {
            bucket_table[index].next = node;
            (bucket_table[index].key)++;
        } else {
            // 链表结构的插入排序
            while (p.next != null && p.next.key <= node.key)
                p = p.next;
            node.next = p.next;
            p.next = node;
            (bucket_table[index].key)++;
        }
    }
    // 打印结果
    for (int b = 0; b < bucketsize; b++)
        for (Node k = bucket_table[b].next; k != null; k = k.next)
            System.out.print(k.key + " ");
}

public static void main(String args) {
    int array = { 49, 38, 65, 97, 76, 13, 27, 49 };
    new Test().IncSort(array, 10);
}

```

程序运行结果为：

```
13 27 38 49 49 65 76 97
```

桶排序一般适用于寻找第 k 大的数、寻找中位数、寻找不重复或重复的数字等情况，例如：

- 1) 在一个文件中有 10 G 个整数，乱序排列，要求找出中位数，内存限制为 2 GB。
- 2) 现在有一个 0 ~ 30000 的随机数生成器。请根据这个随机数生成器，设计一个抽奖范围是 0 ~ 350000 彩票中奖号码列表，其中要包含 20000 个中奖号码。

10. MapReduce 法

MapReduce 是云计算的核心技术之一，是一种简化并行计算的分布式编程模型，是 Google 的一项重要技术，它为并行系统的数据处理提供了一个简单、高效的解决方案，其主要目的是为了大型集群的系统能在大数据集上进行并行工作，并用于大规模数据的并行运算。

MapReduce 适用于大规模数据集（通常大于 1TB）的并行运算，其核心操作是 Map 和 Reduce，即 MapReduce 拆开为“Map（映射）”和“Reduce（化简）”，其中，Map 函数独立地对每个元素进行操作，它用于把一组键值对映射成一组新的键值对，即先通过 Map 程序将数据

切割成不相关的区块，分配（调度）给大量计算机处理达到分布计算的效果，然后通过指定并发的 Reduce 函数来将结果汇总，保证所有映射键值对中的每一个共享相同的键组。

简而言之，一个映射函数就是对一些独立元素组成的概念上的列表（例如，一个测试成绩的列表）的每一个元素进行指定的操作（例如，有人发现所有学生的成绩都被低估了一分，它可以定义一个“加一”的映射函数，用来修正这个错误）。而 Map 操作与 Reduce 操作都可以高度并行运行，Map 是把一组数据一对一地映射为另外的一组数据，其映射的规则由一个函数来指定，例如对 [1,2,4,8] 进行乘 2 的映射就变为 [2,4,8,16]；Reduce 是对一组数据进行规约，这个规约的规则是由另外一个函数指定的，例如对 [1,2,4,8] 进行求和规约得到的结果是 15，而对它进行求积的规约是 64。

通过 MapReduce，不会分布式并行编程的程序员也可以很容易地将自己的程序运行在分布式系统上，同时，通过该模型，能够充分高效地利用集群中每个机器的资源，适合在集群中处理大规模数据的计算任务，这些优点使得 MapReduce 已经成为云计算平台的主流编程模型。

在 Google 发表 MapReduce 后，2004 年，开源社区用 Java 搭建出一套 Hadoop 框架，用于实现 MapReduce 算法，它能够把应用程序分割成许多很小的工作单元，每个单元可以在任何集群节点上执行或重复执行。同时，Hadoop 还提供一个分布式文件系统 GFS（Google File System，Google 文件系统），它是一个可扩展、结构化、具备日志的分布式文件系统，支持大型、分布式大数据量的读写操作，容错性较强，而分布式数据库是一个有序稀疏的多维度映射表，有良好的伸缩性和高可用性，用来将数据存储或部署到各个计算结点上。

在架构中，MapReduce API 提供 Map 和 Reduce 处理、GFS 分布式文件系统和 BigTable 分布式数据库提供数据存取。

面对海量数据的处理，分布式的计算方式会导致网络间大量频繁的数据交换，在这种情况下网络带宽相对属于稀缺资源。而输入的数据存储在集群中机器的本地磁盘上，这样对有限的带宽来说是有利的。系统按照一个的大小划分数据段，原始文件被划分到各个数据段中。再对每个数据段进行备份，分布在不同的机器上。管理机存储这些文件的位置信息，并安排处理这些文件或文件副本的映射任务。如果操作失败，管理机将重新安排映射任务给包含原始文件副本的工作执行。当在集群的工作站运行大型的 MapReduce 操作时，大部分输入数据都可以在本地读取，这样减小了对网络带宽的占用。

海量数据处理的最大难题在于数据规模巨大，使得传统处理方式面临计算能力不足和存储能力不足的瓶颈问题，而基于 Hadoop 可以非常轻松和方便完成处理海量数据的分布式并行程序，并运行于大规模集群上。目前，Hadoop 由 Apache 基金会维护，Yahoo、Facebook、淘宝等公司利用 Hadoop 构建数据处理平台，以满足海量数据分析处理需求。

9.3 经典实例分析

有关海量数据处理的问题一直以来都是互联网企业笔试面试的重点，此类题目也非常多，但归纳起来，主要有以下 3 类：top K 问题、重复问题和排序问题，下面将分别对这 3 类问题进行详细的分析。

9.3.1 top K 问题

在大规模数据处理中，经常会遇到的一类问题就是在海量数据中找出出现频率最高的前 K 个数，或者从海量数据中找出最大的前 K 个数，这类问题通常被称为 top K 问题，例如在搜索

引擎中，统计搜索最热门的 10 个查询词；在歌曲库中统计下载率最高的前 10 首歌等。

针对 top K 类问题，通常比较好的方案是分治 + Trie 树/hash + 小顶堆，即先将数据集按照 hash 方法分解成多个小数据集，然后使用 Trie 树或者 hash 统计每个小数据集中的 query 词频，之后用小顶堆求出每个数据集中出频率最高的前 K 个数，最后在所有 top K 中求出最终的 top K。

例如，有 1 亿个浮点数，如何找出其中最大的 10000 个？

第一种方法是将数据全部排序，然后在排序后的集合中进行查找，最快的排序算法的时间复杂度一般为 $O(n \log n)$ ，例如快速排序。而在 32 位机器上，每个 float 类型占 4B，1 亿个浮点数就要占用 400M 的存储空间，对于一些可用内存小于 400MB 的计算机而言，显然是不能一次将全部数据读入内存进行排序的。其实即使内存能够满足要求，该方法也并不高效，因为题目的目的是寻找出最大的 10000 个数即可，而排序却是将所有元素进行排序，做了很多无用功。

第二种方法为局部淘汰法，该方法与排序方法类似，用一个容器保存前 10000 个数，然后将剩余的所有数字一一与容器内的最小数字相比，如果所有后续的元素都比容器内的 1000 个数还小，那么容器内的这 10000 个数就是最大的 10000 个数。若某一后续元素比容器内的最小元素大，则删掉容器内的最小元素，并将该元素插入容器，最后遍历完这 1 亿个数，得到的结果容器中保存的数即为最终结果了。此时的时间复杂度为 $O(n + m^2)$ ，其中 m 为容器的大小，即 10000。

第三种方法是分治法，将 1 亿个数据分成 100 份，每份 100 万个数据，找出每份数据中最大的 10000 个，最后在剩下的 100×10000 个数据里面找出最大的 10000 个。如果 100 万数据选得足够理想，那么可以过滤掉 1 亿数据里面 99% 的数据。100 万个数据里面查找最大的 10000 个数据的方法如下：用快速排序的方法，将数据分为两堆，如果大的那堆个数 N 大于 10000 个，继续对大堆快速排序一次分成两堆；如果大堆个数 N 小于 10000，就在小的那堆里面快速排序一次，找第 $10000 - n$ 大的数字；递归以上过程，就可以找到第 10000 大的数。参考上述方法找出第 10000 大数字，就可以类似的方法找出前 10000 大的数字了。此种方法每次需要的内存空间为 $100 \text{ 万} * 4 = 4\text{M}$ ，一共需要 101 次这样的比较。

第四种方法是 hash 法。如果这 1 亿个数里面有很多重复的数，先通过 hash 法，把这 1 亿个数去重复，这样如果重复率很高的话，会减少很大的内存用量，从而缩小运算空间，然后通过分治法或最小堆法进行差早最大的 10000 个数。

第五种方法采用最小堆。先读入前 10000 个数来创建大小为 10000 的小顶堆，建堆的时间复杂度为 $O(m \log m)$ (m 为数组的大小即为 10000)，然后遍历后续的数字，并与堆顶（最小）数字进行比较，若比最小的数小，则继续读取后续数字；若比堆顶数字大，则替换堆顶元素并重新调整堆为小顶堆。整个过程直至 1 亿个数全都遍历完为止。然后按照中序遍历的方式输出当前堆中的所有 10000 个数字。该算法的时间复杂度为 $O(nm \log m)$ ，空间复杂度是 10000（常数）。

实际上，最优的解决方案应该是最符合实际设计需求的方案，在实际应用中，可能有足够大的内存，那么直接将数据扔到内存中一次性处理即可，也可能机器有多个核，这样可以采用多线程处理整个数据集。

下面针对不同的应用场景，分析了适合相应应用场景的解决方案。

(1) 单机 + 单核 + 足够大内存

例如如果需要查找 10 亿个查询词（每个占 8Byte）中出现频率最高的 10 个，考虑到每个

查询词占 8Byte, 则 10 亿个查询词所需的内存大约是 $10^9 \times 8\text{Byte} = 8\text{GB}$ 内存。如果有这么大的内存, 直接在内存中对查询词进行排序, 顺序遍历找出 10 个出现频率最大的 10 个即可。这种方法简单快速, 更加实用。当然, 也可以先用 HashMap 求出每个词出现的频率, 然后求出出现频率最大的 10 个词。

(2) 单机 + 多核 + 足够大内存

这时可以直接在内存中实用 hash 方法将数据划分成 n 个 partition, 每个 partition 交给一个线程处理, 线程的处理逻辑同 (1) 类似, 最后一个线程将结果归并。

该方法存在一个瓶颈——会明显影响效率, 即数据倾斜, 每个线程的处理速度可能不同, 快的线程需要等待慢的线程, 最终的处理速度取决于慢的线程。而针对此问题, 解决的方法是: 将数据划分成 $c \times n$ 个 partition ($c > 1$), 每个线程处理完当前 partition 后主动取下一个 partition 继续处理, 直到所有数据处理完毕, 最后由一个线程将结果归并。

(3) 单机 + 单核 + 受限内存

这种情况下, 需要将原数据文件切割成一个一个小文件, 如采用 $\text{hash}(x) \% M$, 将原文件中的数据切割成 M 小文件, 如果小文件仍大于内存大小, 继续采用 hash 的方法对数据文件进行切割, 直到每个小文件小于内存大小, 这样, 每个文件可放到内存中处理。采用 (1) 的方法依次处理每个小文件。

(4) 多机 + 受限内存

这种情况下, 为了合理利用多台机器的资源, 可将数据分发到多台机器上, 每台机器采用 (3) 节中的策略解决本地的数据。可采用 Hash + Socket 方法进行数据分发。

从实际应用的角度考虑, (1) (2) (3) (4) 方案并不可行, 因为在大规模数据处理环境下, 作业效率并不是首要考虑的问题, 算法的扩展性和容错性才是首要考虑的。算法应该具有良好的扩展性, 以便数据量进一步加大 (随着业务的发展, 数据量加大是必然的) 时, 在不修改算法框架的前提下, 可达到近似的线性比; 算法应该具有容错性, 即当前某个文件处理失败后, 能自动将其交给另外一个线程继续处理, 而不是从头开始处理。

top K 问题很适合采用 MapReduce 框架解决, 用户只需编写一个 Map 函数和两个 Reduce 函数, 然后提交到 Hadoop (采用 Mapchain 和 Reducechain) 上即可解决该问题。具体而言, 就是首先根据数据值或者把数据 $\text{hash}(\text{MD5})$ 后的值按照范围划分到不同的机器上, 最好可以让数据划分后可以一次读入内存, 这样不同的机器负责处理各种的数值范围, 实际上就是 Map。得到结果后, 各个机器只需拿出各自的出现次数最多的前 N 个数据, 然后汇总, 选出所有数据中出现次数最多的前 N 个数据, 这实际上就是 Reduce 过程。对于 Map 函数, 采用 hash 算法, 将 hash 值相同的数据交给同一个 Reduce task; 对于第一个 Reduce 函数, 采用 HashMap 统计出每个词出现的频率, 对于第二个 Reduce 函数, 统计所有 Reduce task 输出数据中的 top K 即可。

直接将数据均分到不同的机器上进行处理是无法得到正确结果的。因为一个数据可能被均分到不同的机器上, 而另一个则可能完全聚集到一个机器上, 同时还可能存在具有相同数目的数据。例如如果要找出现次数最多的前 100 个, 将 1000 万的数据分布到 10 台机器上, 找到每台出现次数最多的前 100 个, 归并之后这样不能保证找到真正的第 100 个, 因为比如出现次数最多的第 100 个可能有 1 万个, 但是它被分到了 10 台机器, 这样在每台上只有 1000 个, 假设这些机器排名在 1000 个之前的那些都是单独分布在一台机器上的, 比如有 1001 个, 这样本来具有 10000 个的这个就会被淘汰, 即使让每台机器选出出现次数最多的 1000 个再归并, 仍然会出错, 因为可能存在大量个数为 1001 个的发生聚集。因此不能将数据随便均分到不同机器

上,而是要根据 hash 后的值将它们映射到不同的机器上处理,让不同的机器处理一个数值范围。

top K 问题还有很多应用场景,尤其是在程序员面试笔试中有很多实例,它们都可以采用上述方法解决。下面是一些历年来经常被各大互联网公司提及的 top K 问题。

1) 有 1000 万个记录,这些查询串的重复度比较高,如果除去重复后,不超过 300 万个。一个查询串的重复度越高,说明查询它的用户越多,也就是越热门。请统计最热门的 10 个查询串,要求使用的内存不能超过 1GB。

2) 有 10 个文件,每个文件 1GB,每个文件的每一行都存放的是用户的 query,每个文件的 query 都可能重复。请按照 query 的频度排序。

3) 有一个 1GB 大小的一个文件,里面每一行是一个词,词的大小不超过 16B,内存限制大小是 1MB。返回频数最高的 100 个词。

4) 提取某日访问网站次数最多的那个 IP。

5) 从 10 亿个整数找出重复次数最多的 100 个整数。

6) 搜索的输入信息是一个字符串,统计 300 万条输入信息中的最热门的前 10 条,每次输入的一个字符串为不超过 255Byte,内存使用只有 1GB。

7) 有 1000 万个身份证号以及它们对应的数据,身份证号可能重复,找出出现次数最多的身份证号。

9.3.2 重复问题

在海量数据中查找出重复出现的元素或者去除重复出现的元素也是常考的问题。针对此类问题,一般可以通过位图法实现,例如,已知某个文件内包含一些电话号码,每个号码为 8 位数字,统计不同号码的个数。

本题最好的解决方法是通过使用位图法来实现。8 位整数可以表示的最大十进制数值为 99999999,如果每个数字对应于位图中一个 bit 位,那么存储八位整数大约需要 99Mbit,因为 1Byte = 8 bit,所以 99Mbit 折合成内存为 $99/8 = 12.375\text{MB}$ 的内存,即可以只用 12.375MB 的内存表示所有 8 位数电话号码的内容。

程序示例如下所示:

```
import java.util.Random;
public class Test {
    int ARRNUM = 100;
    int mmin = 10000000;
    int mmax = 99999999;
    int N = (mmax - mmin + 1);
    int BITS_PER_WORD = 32;
    int WORD_OFFSET(int b) {
        return b/BITS_PER_WORD;
    }
    int BIT_OFFSET(int b) {
        return b % BITS_PER_WORD;
    }
    void SetBit(int[] words,int n) {
        n -= mmin;
        words[WORD_OFFSET(n)] |= (1 << BIT_OFFSET(n));
    }
}
```

```

}
void ClearBit(int[] words,int n) {
    words[ WORD_OFFSET(n) ]&=~(1 << BIT_OFFSET(n));
}
boolean GetBit(int[] words,int n) {
    int bit = words[ WORD_OFFSET(n) ]&(1 << BIT_OFFSET(n));
    return bit !=0;
}
public void sort() {
    int i;
    int j;
    int arr[] = new int[ ARRNUM ];
    System.out.println("数组大小:" + ARRNUM);
    // 用来存放位图,每一位对应 mmin 到 mmax 范围内的一个数
    int[] words = new int[1 + N/BITS_PER_WORD];
    int count=0;
    Random r = new Random();
    // 生成 100 个随机数存放到数组 arr 中 for(j=0;
    j < ARRNUM;j++) {arr[j] = r.nextInt(N);
        arr[j] += mmin;
        System.out.print(arr[j] + " ");
    }
    System.out.println();
    for(j=0;j < ARRNUM;j++) {
        SetBit(words, arr[j]);
    }
    System.out.println("排序后 a 为:");
    for(i=0;i < N;i++) {
        if( GetBit(words, i) ) {
            System.out.print(i + mmin + " ");
            count++;
        }
    }
    System.out.println();
    System.out.println("总个数为:" + count);
}
public static void main(String[] args) {
    new Test(). sort();
}
}

```

上例中,采用时间作为种子,产生了100个随机数,对着100个数进行位图法排序,进而找出其中重复的数据。与此问题相似的面试题笔试题还有:

- 1) 10亿个正整数,只有1个数重复出现过,要求在 $O(n)$ 的时间里找出这个数。
- 2) 给定a、b两个文件,各存放50亿个URL,每个URL各占用64Byte,要求在 $O(n)$ 的时间里找出a、b文件共同的URL。
- 3) 给40亿个不重复的unsigned int的整数(没排过序的),然后再给一个数,如何快速判断这个数是否在那40亿个数当中?

9.3.3 排序问题

海量数据处理中一类常见的问题就是排序问题，即对海量数据中的数据进行排序，例如，一个文件中有 9 亿条不重复的 9 位整数，对这个文件中数字进行排序。

针对这个问题，最容易想到的方法是将所有数据导入到内存中，然后使用常规的排序方法，例如插入排序、快速排序、归并排序等各种排序方法对数据进行排序，最后将排序好的数据存入文件。但这些方法在此却并不适用，由于数据量巨大，在 32 位机器中，一个整数占用 4B，而 9 亿条数据以供占用 9 亿 \times 4B，大约需要占用 3.6GB 内存，对于 32 位机器而言，很难将这么多数据一次载入到内存，更不用说进行排序了，所以此种方法一般不可行，需要考虑其他方法。

方法一，数据库排序法。将文本文件导入到数据库中，让数据库进行索引排序操作后提取数据到文件。该方法虽然操作简单、方便，但是运算速度较慢，而且对数据库设备要求比较高。

方法二，分治法。通过 Hash 法将 9 亿条数据分为 20 段，每段大约 5000 万条，大约占用 500 万 \times 4B = 200MB 空间，在文件中依次搜索 0 ~ 5000 万，50000001 ~ 1 亿……将排序的结果存入文件。该方法要装满 9 位整数，一共需要 20 次，所以一共要进行 20 次排序，需要对文件进行 20 次读操作。该方法虽然缩小了每次使用的内存空间大小，但是编码复杂，速度也慢。

方案三，位图法。考虑到最大的 9 位整数为 999999999，由于 9 亿条数据是不重复的，可以把这些数据组成一个队列或数组，让它有 0 ~ 999999999（一共 10 亿个数）元素数组下标表示数值，结点中用 0 表示没有这个数，1 表示存在这个数，判断 0 或 1 只用一个 bit 存储就够了，而声明一个可以包含 9 位整数的 bit 数组，一共需要 10 亿/8，大约 120MB 内存，把内存中的数据全部初始化为 0，读取文件中的数据，并将数据放入内存。比如读到一个数据为 341245909 这个数据，那就先在内存中找到 341245909 这个 bit，并将 bit 值置为 1，遍历整个 bit 数组，将 bit 为 1 的数组下标存入文件，最终得到排序后的内容。

此类排序问题的求解方法一般都是采用上述方法。而海量数据处理中与此类似的问题还有以下两种：

1) 一年的全国高考考生人数为 500 万，分数使用标准分，最低 100 分，最高 900 分，不存在成绩为小数的情况，把这 500 万考生的分数排序。

2) 一个包含 n 个正整数的文件，每个正整数小于 n ， n 等于 1000 万，并且文件内的正整数没有重复和关联数据，输出整数的升序排列。

附录

附录 A 软件企业 Java 笔试真题 1

一、不定项选择题

1. 欲构造 ArrayList 类的一个实例，此类继承了 List 接口，下列方法中，正确的是 ()。
A. ArrayList myList = new Object(); B. List myList = new ArrayList();
C. ArrayList myList = new List(); D. List myList = new List();
2. paint() 方法使用哪种类型的参数? ()
A. Graphics B. Graphics2D C. String D. Color
3. 下列表达式中，正确的是 ()。
A. Byte = 128; B. Boolean = null; C. Long l = 0xffff; D. Double = 0.9239d;
4. 下列程序运行的结果为 ()。

```
public class Example {  
    String str = new String("good");  
    Char [] ch = {'a', 'b', 'c'};  
    public static void main(String args[]) {  
        Example ex = new Example();  
        Ex. change(ex. str, ex. ch);  
        System. out. print(ex. str + "and");  
        System. out. print(ex. ch);  
    }  
    public void change(String str, char ch []) {  
        str = "test ok";  
        ch [0] = '0';  
    }  
}
```

- A. good and abc B. good and gbc C. test ok and abc D. test ok and gbc
5. 下列程序运行的结果为 ()。

```
public class X extends Thread implements Runnable {  
    public void run() {
```

```

        System.out.println("this is run()");
    }
    public static void main(String args[]){
        Thread t = new Thread(new X());
        t.start();
    }
}

```

- A. 第 1 行会产生编译错误
 B. 第 6 行会产生编译错误
 C. 第 6 行会产生运行错误
 D. 程序会运行和启动
6. 要从文件“file.dat”文件中读出第 10 个字节到变量 C 中，下列哪个方法适合？（ ）
- A. `FileInputStream in = new FileInputStream("file.dat"); in.skip(9); int c = in.read();`
 B. `FileInputStream in = new FileInputStream("file.dat"); in.skip(10); int c = in.read();`
 C. `FileInputStream in = new FileInputStream("file.dat"); int c = in.read();`
 D. `RandomAccessFile in = new RandomAccessFile("file.dat"); in.skip(9); int c = in.readByte();`
7. 容器被重新设置大小后，哪种布局管理器的容器中的组件大小不随容器大小的变化而改变？（ ）
- A. `CardLayout` B. `FlowLayout` C. `BorderLayout` D. `GridLayout`
8. 给出下面代码：

```

public class Person {
    static int arr[] = new int[10];
    public static void main(String a[]){
        System.out.println(arr[1]);
    }
}

```

下列哪种说法是正确的？（ ）

- A. 编译时将产生错误
 B. 编译时正确，运行时将产生错误
 C. 输出零
 D. 输出空
9. 下列哪个关键字可以对对象加互斥锁？（ ）
- A. `transient` B. `synchronized` C. `serialize` D. `static`
10. 下列关于内存回收的说法正确的是（ ）。
- A. 程序员必须创建一个线程来释放内存
 B. 内存回收程序负责释放无用内存
 C. 内存回收程序允许程序员直接释放内存
 D. 内存回收程序可以在指定的时间释放内存对象
11. 下列代码中，会出错的是（ ）。

```

1) public void modify() {
2)     int i,j,k;
3)     i = 100
4)     while( i > 0 ) {
5)         i = j * 2;

```

- ```

6) System.out.println("The value of j is" + j);
7) k = k + 1;
8) i--;
9) }
10) }

```

A. line 4                      B. line 6                      C. line 7                      D. line 8

## 二、多项选择题

1. 执行下列代码后，哪个结论是正确的？（     ）

```
String[] s = new String[10];
```

A. s[10]为" "            B. s[9]为 null            C. s[0]为未定义    D. s.length 为 10

2. 下列表达式中，正确的是（     ）。

- A. String s = "你好"; int i = 3; s + = i;  
 B. String s = "你好"; int i = 3; if(i == s) { s + = i };  
 C. String s = "你好"; int i = 3; s = i + s;  
 D. String s = "你好"; int i = 3; s = i + ;  
 E. String s = "你好"; int i = (s != null) && (s.length > 0) ? s.length() : 0;

3. 下列标识符中，合理的是（     ）。

A. :\_sysl\_lll            B. 2mail                      C. \$ change                      D. class

4. 哪个布局管理器使用的是组件的最佳尺寸？（     ）

A. FlowLayout            B. BorderLayout            C. GridLayout                      D. CardLayout  
 E. GridBagLayout

5. 下列哪个方法可用于创建一个可运行的类？（     ）

- A. public class X implements Runnable { public void run() { ..... } }  
 B. public class X implements Thread { public void run() { ..... } }  
 C. public class X implements Runnable { public int run() { ..... } }  
 D. public class X implements Runnable { protected void run() { ..... } }  
 E. public class X implements Thread { public void run() { ..... } }

6. 下面哪个方法可以在任何时候被任何线程调用？（     ）

A. notify()                      B. wait()                      C. notifyall()  
 D. sleep()                      E. yield()                      F. synchronized(this)

7. 构造 BufferedInputStream 的合适参数是哪个？（     ）

A. BufferedInputStream                      B. BufferedOutputStream  
 C. FileInputStream                      D. FileOuterStreamE: File

8. 下列说法中，正确的是（     ）。

A. Java.lang.Cloneable 是类                      B. Java.lang.Runnable 是接口  
 C. Double 对象在 java.lang 包中                      D. Double a = 1.0 是正确的 Java 语句

9. 下列表达式中，正确的是（     ）。

A. double a = 1.0;  
 C. byte a = 340;  
 B. Double a = new Double (1.0);  
 D. Byte a = 120;

10. 定义一个类名为“MyClass.java”的类，并且该类可被一个工程中的所有类访问，那



么该类的正确声明应为 ( )。

- A. private class MyClass extends Object      B. class MyClass extends Object  
C. public class MyClass                              D. public class MyClass extends Object

11. 指出下列哪个方法与方法 public void add(int a) {} 为合理的重载方法 ( )。

- A. public int add(int a)                              B. public void add(long a)  
C. public void add(int a, int b)                      D. public void add(float a)

12. 如果下列方法能够正常运行,那么控制台上将显示什么? ( )

```
public void example() {
 try {
 unsafe();
 System.out.println("Test 1");
 }
 catch (SafeException e) {
 System.out.println("Test 2");
 }
 finally { System.out.println("Test 3"); }
 System.out.println("Test 4");
}
```

- A. Test 1                              B. Test 2                              C. Test 3                              D. Test 4

13. 下列哪些情况可以终止当前线程的运行? ( )

- A. 抛出一个例外时                              B. 当该线程调用 sleep() 方法时  
C. 当创建一个新线程时                              D. 当一个优先级高的线程进入就绪状态时

### 三、填空题

1. 执行下列代码后的结果是什么?

```
int x, a=2, b=3, c=4; x = ++a + b++ + c++;
```

2. 哪个包包含了 Collection 的接口和类的 API?  
3. main() 方法的声明格式包括什么?  
4. 下列程序中构造了一个 SET 并且调用其 add() 方法, 输出结果是什么?

```
public class A {
 public int hashCode() { return 1; }
 public Boolean equals(Object b) { return true; }
 public static void main(String args[]) {
 Set set = new HashSet();
 set.add(new A());
 set.add(new A());
 set.add(new A());
 System.out.println(set.size());
 }
}
```

5. 下列程序的运行结果是什么?

```
class A {
 class Dog {
 private String name;
```

```

private int age;
Dog(String s,int a) {
 name = s;
 age = a;
 step = 0;
}

public void run(Dog fast) {
 fast. step ++ ;
}

public static void main(String args[]) {
 A a = new A();
 Dog d = a. new Dog("Tom" ,3);
 d. step = 25;
 d. run(d);
 System. out. println(d. step);
}

```

#### 四、编程题

1. 编写一个输出“Hello World!”的程序，用两种方式实现（Application、Applet）。

2. 用输入/输出写一个程序，让用户输入一些姓名和电话号码。每一个姓名和电话号码将加在文件里。用户通过单击“Done”按钮来告诉系统整个列表已输入完毕。如果用户输入整个列表，程序将创建一个输出文件并显示或打印出来。格式如：555 - 1212, Tom 123 - 456 - 7890, Peggy L. 234 - 567, Marc 234 - 5678, Ron 876 - 4321, Beth&Brian 33.1.42.45.70, Jean - Marc。

## 附录 B 软件企业 Java 笔试真题 2

### 一、不定项选择题

- 下面哪些是合法的变量名? ( )  
A. 2variable      B. variable2      C. \_whatavariabale      D. \_3\_  
E. \$anothervar      F. #myvar
- 请问“abcd” instanceof Object 返回的值是? ( )  
A. “abcd”      B. true      C. false      D. String
- 下列说法中, 正确的是 ( )。  
A. 若源代码中有 package 语句, 则该语句必须放在代码的第一行 (不考虑注释和空格)  
B. 若源代码中有 import 语句, 则该语句必须放在代码的第一行 (不考虑注释和空格)  
C. 若源代码中有 main() 方法, 则该方法必须被放在代码的第一行  
D. 若某文件的源代码中定义了一个 public 的接口, 则接口名和文件名可以不同
- 下列有关方法覆盖说法中, 不正确的是 ( )。  
A. 方法覆盖要求覆盖和被覆盖的方法有相同的名字, 参数列以及返回值  
B. 方法覆盖要求覆盖和被覆盖的方法必须具有相同的访问权限  
C. 覆盖的方法不能比被覆盖的方法抛出更多的异常  
D. 覆盖的方法一定不能是 private 的
- 一个 Java 程序运行从上到下的环境次序是 ( )。  
A. 操作系统、Java 程序、JRE/JVM、硬件      B. JRE/JVM、Java 程序、硬件、操作系统  
C. Java 程序、JRE/JVM、操作系统、硬件      D. Java 程序、操作系统、JRE/JVM、硬件
- 下列选项中哪个关键字可以用来修饰接口中的变量和方法? ( )  
A. static      B. private      C. synchronized      D. protected
- 针对下述代码段的描述中, 正确的是 ( )。

```
String s = "abcde";
System.out.println(s.charAt(4));
```

- 输出字符 d      B. 输出字符 e
  - 什么都没有, 抛出 ArrayIndexOutOfBoundsException
  - 代码编译不成功, 因为 charAt() 方法不属于 String 类
- 下列哪种创建 Map 集合的方式是正确的? ( )  
A. Map m = new Map()  
B. Map m = new Map(initialCapacity, incrementCapacity)  
C. Map m = new Map(new Collection())  
D. Map 是接口, 所以不能实例化
- 关于被私有保护访问控制符 protected 修饰的成员变量, 以下说法正确的是? ( )  
A. 可以被三种类所引用: 该类自身、与它在同一个包中的其他类、在其他包中的该类的子类

- B. 可以被两种类访问和引用：该类本身、该类的所有子类  
 C. 只能被该类自身所访问和修改  
 D. 只能被同一个包中的类访问
10. 下列有关继承的说法中，正确的是（ ）。  
 A. 子类能继承父类的所有方法和状态  
 B. 子类能继承父类的非私有方法和状态  
 C. 子类只能继承父类 public 方法和状态  
 D. 子类能继承父类的方法，而不是状态
11. 对于构造方法，下列叙述中，正确的是（ ）。  
 A. 构造方法的方法名必须与类名相同  
 B. 构造方法必须用 void 申明返回类型  
 C. 构造方法可以被程序调用  
 D. 如果编程人员没再类中定义构造方法，程序将报错
12. 为了区分类中重载的同名的不同的方法，要求（ ）。  
 A. 采用不同的形式参数列表  
 B. 返回值的数据类型不同  
 C. 调用时用类名或者对象名做前缀  
 D. 参数名不同
13. 下列有关子类继承父类构造函数的描述中，正确的是（ ）。  
 A. 创建子类的对象时，先调用子类自己的构造函数，然后调用父类的构造函数  
 B. 子类无条件地继承父类不含参数的构造函数  
 C. 子类必须通过 super() 关键字调用父类的构造函数  
 D. 子类无法继承父类的构造函数
14. 下列说法中，正确的是（ ）。  
 A. 类是变量和方法的集合体  
 B. 数组是无序数据的集合  
 C. 抽象类可以实例化  
 D. 类成员数据必须是公有的
15. 现有程序代码如下：

```

1) public class Test {
2) public static void main(String args[]) {
3) class Foo {
4) public int i = 3;
5) }
6) Object o = (Object) new Foo();
7) Foo foo = (Foo) o;
8) System.out.println("i = " + foo.i);
9) }
10) }

```

上述程序运行后的结果为（ ）。

- A. i = 3  
 B. 编译失败  
 C. 第 6 行抛出 ClassCastException  
 D. 第 7 行抛出 ClassCastException

## 二、填空题

1. 请问构造函数、成员变量初始化以及静态成员变量初始化三者的先后顺序是\_\_\_\_\_。
2. 用于将一个类修饰为最终类的关键字是\_\_\_\_\_。用于声明一个类为抽象类的关键字是\_\_\_\_\_。
3. Java 基本数据类型中，整型、字符型占用字节数分别为\_\_\_\_\_和\_\_\_\_\_。

4. Java 中提供了两种用于多态的机制 \_\_\_\_\_ 和 \_\_\_\_\_。
5. 一般有两种用于创建线程对象的方法：一是 \_\_\_\_\_；二是 \_\_\_\_\_。

### 三、问答题

1. 面向对象的特征有哪些？
2. 接口是否可继接口？抽象类是否可实现（implements）接口？抽象类是否可继承实体类？
3. String and StringBuffer 的区别？
4. 说出 ArrayList、Vector、LinkedList 的存储性能和特性 HashMap 和 Hashtable 的区别？
5. 描述 final、finally、finalize 的区别？

### 四、附加题

1. 排序都有哪几种方法？请列举。并用 Java 语言实现一个插入排序。
2. 编程：编写一个截取字符串的函数，输入为一个字符串和字节数，输出为按字节截取的字符串。但是要保证汉字不被截半个，如“我 ABC”4，应该截为“我 AB”；输入“我 ABC 汉 DEF”，应该输出为“我 ABC”而不是“我 ABC + 汉的半个”。

## 附录 C 软件企业 Java 笔试真题 3

### 一、单项选择题

- Java 是从 ( ) 语言改进重新设计而来的。  
A. Ada      B. C++      C. Pascal      D. BASIC
- 下列语句中, 正确的是 ( )。  
A. Java 程序经编译后会产生 machine code    B. Java 程序经编译后会产生 byte code  
C. Java 程序经编译后会产生 DLL      D. 以上都不正确
- 下列说法中, 正确的是 ( )。  
A. class 中的 constructor 不可省略  
B. constructor 必须与 class 同名, 但方法不能与 class 同名  
C. constructor 在一个对象被 new 时执行  
D. 一个 class 只能定义一个 constructor
- 提供 Java 存取数据库能力的包是 ( )。  
A. java.sql      B. java.awt      C. java.lang      D. java.swing
- 下列运算符中, 合法的是 ( )。  
A. &&      B. < >      C. if      D. ; =
- 执行如下程序代码后, c 的值是 ( )。  

```
a=0;
c=0;
do{
 --c;
 a=a-1;
}while(a>0);
```

  
A. 0      B. 1      C. -1      D. 死循环
- 下列哪种叙述是正确的? ( )  
A. abstract 修饰符可修饰字段、方法和类  
B. 抽象方法的 body 部分必须用一对大括号 {} 包住  
C. 声明抽象方法, 大括号可有可无  
D. 声明抽象方法不可写出大括号
- 下列语句中, 正确的是 ( )。  
A. 形式参数可被视为 local variable  
B. 形式参数可被字段修饰符修饰  
C. 形式参数为方法被调用时, 真正被传递的参数  
D. 形式参数不可以是对象
- 下列哪种说法是正确的? ( )  
A. 实例方法可直接调用超类的实例方法    B. 实例方法可直接调用超类的类方法  
C. 实例方法可直接调用其他类的实例方法    D. 实例方法可直接调用本类的类方法



## 二、多项选择题

- 下列说法中，正确的有 ( )。
  - 环境变量可在编译 source code 时指定
  - 在编译程序时，所能指定的环境变量不包括 class path
  - javac 一次可同时编译数个 Java 源文件
  - javac.exe 能指定编译结果要置于哪个目录 (directory)
- 下列标识符中，不合法的有 ( )。
  - new
  - \$Usdollars
  - 1234
  - car.taxi
- 下列说法中，错误的有 ( )。
  - 数组是一种对象
  - 数组属于一种原生类
  - int number = [ ] = {31,23,33,43,35,63}
  - 数组的大小可以任意改变
- 不能用来修饰 interface 的有 ( )。
  - private
  - public
  - protected
  - static
- 下列说法中，正确的有 ( )。
  - call by value 不会改变实际参数的数值
  - call by reference 能改变实际参数的参考地址
  - call by reference 不能改变实际参数的参考地址
  - call by reference 能改变实际参数的内容
- 下列说法中，错误的有 ( )。
  - 在类方法中可用 this 来调用本类的类方法
  - 在类方法中可直接调用本类中的类方法
  - 在类方法中只能调用本类中的类方法
  - 在类方法中绝对不能调用实例方法
- 下列说法中，错误的有 ( )。
  - Java 面向对象语言容许单独的过程与函数存在
  - Java 面向对象语言容许单独的方法存在
  - Java 语言中的方法属于类中的成员 (member)
  - Java 语言中的方法必定隶属于某一类 (对象)，其调用方法与过程或函数相同
- 下列说法中，错误的有 ( )。
  - 能被 java.exe 成功运行的 java class 文件必须有 main() 方法
  - J2SDK 就是 Java API
  - Appletviewer.exe 可利用 jar 选项运行 .jar 文件
  - 能被 Appletviewer 成功运行的 java class 文件必须有 main() 方法

## 三、判断题

- Java 程序中的起始类名称必须与存放该类的文件名相同。( )
- Unicode 是用 16 位来表示一个字的。( )
- 原生类中的数据类型均可任意转换。( )

## 四、程序题

- 写出下列程序的运行结果。

```
public static void main(String[] args) {
 String a = "hello";
```

```

 change(a);
 System.out.println(a);
}
public static String change(String name){
 return "world";
}
}

```

## 2. 写出下列程序运行的结果。

```

public class TestFoo {
 static boolean foo(char c) {
 System.out.print(c);
 return true;
 }
 public static void main(String[] argv) {
 int i=0;
 for(foo('A');foo('B')&&(i<2);foo('C')){
 i++;
 foo('D');
 }
 }
}

```

## 五、简答题

1. & 和 && 的区别。
2. HashMap 和 Hashtable 的区别。
3. Colletion 和 Collections 的区别。
4. abstract class 和 interace 的区别。
5. final、finally 和 finalize 的区别。

## 六、加分题

1. 谈谈 SpringMVC、ICO、AOP 的理解。
2. 写出常用的设计模式，谈谈你对工厂模式的理解。

## 附录 D 求职有用网站及 QQ 群一览表

|                                                                         |                                                                                                                                                                    |
|-------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 技术学习网站                                                                  | <a href="http://www.iteye.com">http://www.iteye.com</a>                                                                                                            |
|                                                                         | <a href="http://www.51cto.com">http://www.51cto.com</a>                                                                                                            |
|                                                                         | <a href="http://www.cnblogs.com/">http://www.cnblogs.com/</a>                                                                                                      |
|                                                                         | <a href="http://leetcode.com/">http://leetcode.com/</a>                                                                                                            |
|                                                                         | <a href="https://github.com/soulmachine/leetcode">https://github.com/soulmachine/leetcode</a> (leetcode 题解供参考)                                                     |
|                                                                         | <a href="http://wikioi.com/">http://wikioi.com/</a>                                                                                                                |
|                                                                         | <a href="http://codility.com/">http://codility.com/</a>                                                                                                            |
|                                                                         | <a href="http://coolshell.cn/">http://coolshell.cn/</a> (酷壳)                                                                                                       |
|                                                                         | <a href="https://www.hackerrank.com/">https://www.hackerrank.com/</a>                                                                                              |
| 求职网站                                                                    | <a href="http://www.itmian4.com/">http://www.itmian4.com/</a> (IT 面试)                                                                                              |
|                                                                         | <a href="http://www.51projob.com">http://www.51projob.com</a> (程序员求职网)                                                                                             |
|                                                                         | <a href="http://www.jobcoding.com">http://www.jobcoding.com</a> (程序员面试笔试宝典)                                                                                        |
|                                                                         | <a href="http://s.sousb.com/">http://s.sousb.com/</a> (程序员面试之家)                                                                                                    |
|                                                                         | <a href="http://hawstein.com/posts/ctci-solutions-contents.html">http://hawstein.com/posts/ctci-solutions-contents.html</a> (Cracking the coding interview——问题与解答) |
| 求职论坛                                                                    | <a href="http://www.careercup.com/">http://www.careercup.com/</a> (careercup)                                                                                      |
|                                                                         | <a href="http://www.newsmth.net/">http://www.newsmth.net/</a> (水木社区)                                                                                               |
|                                                                         | <a href="http://www.xdnice.com">http://www.xdnice.com</a> (西电好网)                                                                                                   |
|                                                                         | <a href="http://bbs.pku.edu.cn/">http://bbs.pku.edu.cn/</a> (北大未名)                                                                                                 |
|                                                                         | <a href="http://bbs.nwpu.edu.cn/portal.php">http://bbs.nwpu.edu.cn/portal.php</a> (西工大三大航四方 BBS)                                                                   |
|                                                                         | <a href="http://bbs.xjtu.edu.cn/">http://bbs.xjtu.edu.cn/</a> (兵马俑 BBS)                                                                                            |
|                                                                         | <a href="http://bbs.byu.cn/index">http://bbs.byu.cn/index</a> (北邮人)                                                                                                |
|                                                                         | <a href="https://bbs.sjtu.edu.cn">https://bbs.sjtu.edu.cn</a> (饮水思源)                                                                                               |
|                                                                         | <a href="http://bbs.whnet.edu.cn/">http://bbs.whnet.edu.cn/</a> (白云黄鹤 BBS)                                                                                         |
|                                                                         | <a href="http://www.job592.com/zyq.html">http://www.job592.com/zyq.html</a> (592 职业圈)                                                                              |
| 就业信息网                                                                   | <a href="http://job.xidian.edu.cn/">http://job.xidian.edu.cn/</a> (西安电子科技大学就业信息网)                                                                                  |
|                                                                         | <a href="http://job.ucas.ac.cn/">http://job.ucas.ac.cn/</a> (中国科学院大学研究生就业信息服务网)                                                                                    |
|                                                                         | <a href="http://www.xsjy.whu.edu.cn/">http://www.xsjy.whu.edu.cn/</a> (武汉大学学生就业信息网)                                                                                |
|                                                                         | <a href="http://jy.seu.edu.cn/">http://jy.seu.edu.cn/</a> (东南大学就业信息网)                                                                                              |
|                                                                         | <a href="http://job.bupt.edu.cn/">http://job.bupt.edu.cn/</a> (北京邮电大学就业信息网)                                                                                        |
|                                                                         | <a href="http://career.tsinghua.edu.cn">http://career.tsinghua.edu.cn</a> (清华大学就业信息网)                                                                              |
|                                                                         | <a href="http://career.dlut.edu.cn">http://career.dlut.edu.cn</a> (大连理工大学就业信息网)                                                                                    |
|                                                                         | <a href="http://job.nwpu.edu.cn">http://job.nwpu.edu.cn</a> (西北工业大学就业信息网)                                                                                          |
| <a href="http://job.xjt.edu.cn">http://job.xjt.edu.cn</a> (西安交通大学就业信息网) |                                                                                                                                                                    |

(续)

|         |                                                                                                                                                                                  |
|---------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 招聘服务网   | <a href="http://www.zhaopin.com/">http://www.zhaopin.com/</a> (智联招聘)                                                                                                             |
|         | <a href="http://www.yingjiesheng.com/">http://www.yingjiesheng.com/</a> (应届生求职网)                                                                                                 |
|         | <a href="http://job.dajie.com/">http://job.dajie.com/</a> (大街网)                                                                                                                  |
|         | <a href="http://www.51job.com">http://www.51job.com</a> (前程无忧)                                                                                                                   |
|         | <a href="http://www.hiall.com.cn/">http://www.hiall.com.cn/</a> (hiall)                                                                                                          |
|         | <a href="http://www.job9151.com/">http://www.job9151.com/</a> (中国高校就业联盟网)                                                                                                        |
| 宣讲会查询系统 | <a href="http://xjh.haitou.cc/">http://xjh.haitou.cc/</a>                                                                                                                        |
|         | <a href="http://www.yjbys.com/xuanjianghui/">http://www.yjbys.com/xuanjianghui/</a>                                                                                              |
|         | <a href="http://campus.dajie.com/talk/index">http://campus.dajie.com/talk/index</a>                                                                                              |
| 知名博主    | <a href="http://zhedahht.blog.163.com/">http://zhedahht.blog.163.com/</a> (何海涛)                                                                                                  |
|         | <a href="http://blog.csdn.net/xdhehao">http://blog.csdn.net/xdhehao</a> (何昊)                                                                                                     |
|         | <a href="http://blog.csdn.net/v_JULY_v">http://blog.csdn.net/v_JULY_v</a> (July)                                                                                                 |
| QQ 讨论群  | 群 1: 279492438, 群 2: 262740805, 群 3: 61846711, 群 4: 262740919, 群 5: 262741149, 群 6: 237808825, 群 7: 275363584, 群 8: 170863804, 群 9: 193489317, 群 10: 275808460, 群 11: 279495828。 |

### 致谢

本书所涉及的内容,有一部分原始资料来源于网络,有些思想也是受到互联网上一些博主、分享者的启发,他们是本书背后默默无闻的英雄。但由于有些地方无法获悉原作者信息或联系不上原作者,所以无法在此对他们的名字予以公布,编者特在此处对他们表示衷心的感谢,并致以最崇高的敬意。