淘宝网Java技术专家，CSDN超人气博主作品

Broadview
www.broadview.com.cn

完全突破Java图书从环境搭建到语法点罗列再到案例总结的写作惯例，
直逼底层，懂原理，看源码，奠定Java老A的坚实基础！

全面提升Java单兵作战能力！

Java特种兵 上册

没有数页代码的简单堆砌，有的是新颖的思考方法
没有各类语法的无聊罗列，更多的是在探索技术背后的思路
没有难懂术语的枯燥晦涩，用的是对话和探讨
轻松，愉快，读来不忍释卷，要学Java，这本书不容错过

谢宇 编著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

内 容 简 介

本书分上、下册，上册强调个人修为的提升，也是本书主旨所在，希望能帮助各位读者朋友提升“功力”；下册将基于上册的内容融入设计、实现的细节。

本书上册共 10 章，主要内容包括：从简单的角度来验证功底，通过一些简单的例子来说明我们应当如何去掌握 Java 的基础；关于计算机的工作原理和 Java 虚拟机的基础知识；Java 通信；Java 并发；数据库知识；源码基础，说明 Java 常见的框架基础知识，比如反射、AOP、ORM、Annotation 和配置文件的原理；JDBC、Spring 的源码讲解，通过几种不同类型的框架源码，希望读者能体会源码之中的思维方式、设计、架构，以及了解到不同源码的区别所在；最后是知识总结。

本书既适合有一定 Java 基础，并希望能在 Java 技术上有所成长的人阅读，也适合能静心看书的初学者，以及以自我提升为主要目的的读者阅读，还适合工作一段时间，对知识和发展的方向很迷茫，甚至对某些观念也比较迷茫，但是又渴望去解决这些问题，渴望自己成长，渴望自己能找到道路的人阅读。

图书在版编目（CIP）数据

Java 特种兵. 上册 / 谢宇编著. —北京：电子工业出版社，2014.9
ISBN 978-7-121-23935-9

Ⅰ. ①J… Ⅱ. ①谢… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2014）第 173062 号

策划编辑：孙学瑛
责任编辑：葛 娜
印 刷：北京中新伟业印刷有限公司
装 订：三河市皇庄路通装订厂
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
开 本：787×1092 1/16 印张：30.75 字数：786 千字
版 次：2014 年 9 月第 1 版
印 次：2014 年 9 月第 1 次印刷
印 数：3000 册 定价：79.00 元（含光盘 1 张）

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

第 1 章
扎马：看看功底如何

本章主要内容

- String 的例子，见证下我们的功底
- 一此简单算法你会如何理解
- 简单数字游戏玩一玩
- 功底概述
- 原生态类型其他例子
- 集合类的那些破事儿
- 常见的工具包
- 面对技术，我们纠结的那些事儿
- 老 A 是在逆境中迎难而上者

1.1 String 的例子，见证下我们的功底

哇塞，第 1 节就开始讲代码例子，受不了啦，胖哥，你坏死了！所有的书第 1 节都是写这个领域有什么东西的。

哈哈，小胖哥天生就是个逆天之人哦，希望你能先实践有了感性认识后，再进行理论了解内在。

下面的代码改编于网络牛人的一段程序，先看代码清单 1-1。

代码清单 1-1　一段 String 的比较程序

```
private static void test1() {
    String a = "a" + "b" + 1;
    String b = "ab1";
    System.out.println(a == b);
}
```

胖哥，你是不是考我智商呀？我们平时对比两个对象不是用 equals()吗？老师告诉我：两个字符串用等号是匹配不了的，结果应该是 false 吧。那么结果到底是多少呢？

运行结果：

```
true
```

什么？竟然是 true？为什么是 true？这是要逆天吗？这小段程序彻底颠覆了我的经验和老师教我的真理！“我和我的小伙伴们惊呆了……”

胖哥告诉你这不能怪老师，老师带进门，修行靠个人！

也许有朋友做出了 true 或猜出了 true，那么可否知道原因呢？如果你知道，那么本节跳过，无须再看；如果还不知道，就听听胖哥给你说说他所理解的原因。

要理解这个问题，你需要了解些什么？

◎ 关于“==”是做什么的？

◎ equals 呢？

◎ a 和 b 在内存中是什么样的？

◎ 编译时优化方案。

下面的内容会很多，现在我们可以站起来简单运动一下，端一杯咖啡，慢慢解读下面的内容。

1.1.1　关于“==”

首先要知道“==”用于匹配内存单元上的内容，其实就是一个数字，计算机内部也只有数字，而在 Java 语言中，当“==”匹配的时候，其实就是对比两个内存单元的内容是否一样。

如果是原始类型 byte、boolean、short、char、int、long、float、double，就是直接比较它们的值。这个大家应该都清楚，这里不再详谈。

如果是引用（Reference），比较的就是引用的值，“引用的值”可以被认为是对象的逻辑地址。如果两个引用发生“==”操作，就是比较相应的两个对象的地址值是否一样。换一句话说，如果两个引用所保存的对象是同一个对象，则返回 true，否则返回 false（如果引用指向的是 null，其实这也是一个 JVM 赋予给它的某个指定的值）。

理解寓意：大家各自拿到了一个公司的 offer，现在我们看看哪些人拿到的 offer 是同一个公司的。

1.1.2　关于“equals()”

equals()方法，首先是在 Object 类中被定义的，它的定义中就是使用“==”方式来匹配的（这一点大家可以参看 Object 类的源码）。也就是说，如果不去重写 equals()方法，并且对应的类其父类列表中都没有重写过 equals()方法，那么默认的 equals()操作就是对比对象的地址。

equals()方法之所以存在，是希望子类去重写这个方法，实现对比值的功能，类似的，String 就自己实现了 equals()方法。为什么要自己去实现呢？因为两个对象只要根据具体业务的关键属性值来对比，确定它们是否是“一致的或相似的”，返回 true|false 即可。

迷惑：equals()不就是对比值的吗？为何说相似？

答曰：一日偶遇怪侠“蜗牛大师”一枚，赐予神剑于数人，猎人将其用于打猎；农夫将它用于劈柴；将军用它保家卫国；侠客用它行侠仗义、惩奸除恶，等等。

例如：在对比一些工程的图纸尺寸的时候，由于尺寸都会存在细节误差，可以认为宽度和高度比较接近，就可以返回 true，而不一定非要精确匹配。另外，图纸纸张的属性除了长度、宽度外，还有如名称、存放位置、卷号等属性，但是我们可能只需要对比它的长度与宽度，在这个范围内其余的属性不会考虑。也就是说，两个对象的值是否相同是自己

的业务决定的，而不是 Java 语言来决定的。

感悟：变通，让标准变为价值，给你一种思想和标准，你可以有不同的使用，不能死扣定理，我们要解决问题！

迷惑：equals()重写后，一般会重写 hashCode()方法吗？

要说明这个问题，我们先要补充一些概念。

Java 中的 hashCode 是什么——hashCode()方法提供了对象的 hashCode 值，它与 equals()一样在 Object 类中提供，不过它是一个 native（本地）方法，它的返回值默认与 System.identityHashCode(object)一致。在通常情况下，这个值是对象头部的一部分二进制位组成的数字，这个数字具有一定的标识对象的意义存在，但绝不等价于地址。

hashCode 的作用——它为了产生一个可以标识对象的数字，不论如何复杂的一个对象都可以用一个数字来标识。为什么需要用一个数字来标识对象呢？因为想将对象用在算法中，如果不这样，许多算法还得自己去组装数字，因为算法的基础是建立在数字基础之上的。那么对象如何用在算法中呢？

例如，在 HashMap、HashSet 等类似的集合类中，如果用某个对象本身作为 Key，也就是要基于这个对象实现 Hash 的写入和查找，那么对象本身如何能实现这个呢？就是基于这样一个数字来完成的，只有数字才能真正完成计算和对比操作。

hashCode 只能说是标识对象，因此在 Hash 算法中可以将对象相对离散开，这样就可以在查找数据的时候根据这个 key 快速地缩小数据的范围。但不能说 hashCode 值一定是唯一的，所以在 Hash 算法中定位到具体的链表后，需要进一步循环链表，然后通过 equals()来对比 Key 的值是否是一样的。这时 hashCode()与 equals()似乎就成为“天生一对”。换句话说，一个是为了算法快速定位数据而存在的，一个是为了对比真实值而存在的。

与 equasls()类似，hashCode()方法也可以重写，重写后的方法将会决定它在 Hash 相关数据结构中的分布情况，所以这个返回值最好是能够将对象相对离散的数据。如果发生一个极端的情况，即 hashCode()始终返回一个值，那么它们将存在于 HashMap 的同一个链表中，将会比链表查询本身还要慢。

在 JDK 1.7 中，Hash 相关的集合类对使用 String 作为 Key 的情况，不再使用 hashCode 方式，而是有了一个 hash32 属性，其余的类型保持不变。

换个思路，hashCode()与 equals()并不是必须强制在一起，如果不需要用到这样的算法，也未必要重写对应的方法，完全由你自己决定，没有语法上强制的规约。

寓意：此好比，宝剑是否需要有剑鞘？宝贝是否需要有宝箱？雄性是否必须需要雌性？地球是否需要有月亮？

而并非是，树是否需要土壤？生命是否需要食物？鱼儿是否必须需要水？世界是否需要阳光？

感悟：一切在于场景与需求，十分需要，但也可以在某些情况下放弃。

有人说：对比两个对象是否一致，可以先对比 hashCode 值再对比 equals()。

这似乎听上去挺有道理的，但是胖哥本人可并不这么认为！为什么呢？胖哥认为 hashCode 值根本不是为了对比两个对象是否一致而存在的，可以说它与两个对象是否一致“一点关系都没有”。

假如你希望对比两个对象是否是同一个对象，则完全可以直接用“==”来对比，而不需要用 hashCode()，因为直接对比地址值说明两个对象是否为同一个对象才是最靠谱的。另外，默认的 hashCode()方法还会发起 native 调用，并且两个对象都会分别发起 native 调用（native 调用的开销也是不小的）。

假如不是对比地址，而是对比值，自然就需要对象中的某些属性来对比。拿 String 类型的对象来讲，如果调用某个 String 对象的 hashCode()方法，它至少会在第 1 次调用这个方法时遍历所有 char[]数组相关元素来计算 hashCode 值（这里之所以说至少，是因为这个 String 如果并发调用 hashCode()方法，则可能被遍历多次），遍历过程中还需要外加一些运算。如果对比的两个对象分别获取 hashCode，自然两个 String 对象都会分别遍历一次 char[]数组。

即使 hashCode 一样了，也同样证明不了这两个 String 是一样的（这个 hashCode 是根据 char[]数组的值算出来的，不同的 String 完全可以算出一样的值），还得再次循环两个 String 中所有的字符来对比一次才能证明两个对象是一样的。其实遍历了两个 char[]数组已经是最坏的情况了，equals()还未必会这样做（在后文的图 1-2 中会详细说明）。

换一个角度来讲，如果是两个其他的自定义类型的对象（不是 String 类型的对象）之间判定出来 hashCode 不一样，也不能说它们的值不一样（有可能 equals()匹配的是一个综合值，与 hashCode 一点关系都没有），还是要进行 equals()，这样绕来绕去往往是把简单问题复杂化了。

equals()内部要怎么做就去怎么做嘛，想要优化，完全可以像 JDK 自带的许多类那样，先对比一些简单的属性值，再对比复杂的属性值，或者先对比业务上最快能区分对象的值，再对比其他的值，或者先对比地址、长度等处理方式，将那些不匹配的情况尽快排出。

有人说重写后的 hashCode()内部操作确实比 equals()简单许多倍，其实这个动作判定也是可以放在 equals()方法的第 1 步中来完成的，无须外部程序关注。

补充：String 的 equals()方法中默认就要先对比传入的对象与当前的 this 是不是同一个对象。在 HashMap 等集合类的查找过程中，也不是单纯的 equals()，也会先对比两个对象是不是同一个对象。

好累，休息休息！左三圈、右三圈，再来看看胖哥为你做解读！

a 和 b 的内存情况是什么样的？

回到“代码清单 1-1”的例子中，其中的等号说明 a 和 b 是指向同一块内存空间的，就像两个人拿到同一个公司的 Offer 一样，他们像什么呢？见图 1-1，“死冤家，又在一起了！”

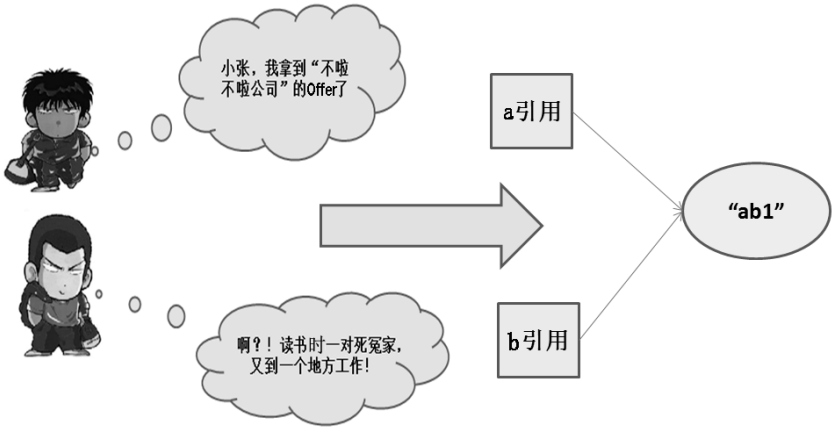

图 1-1　两个冤家又拿到同一个公司的 Offer

为什么 a、b 两个引用都引用到同一块空间了呢？请看 1.1.3 节的内容解释，不过在这一节中我们先感性认识下 JVM 的一些“东东”，在第 3 章中会有更详细的介绍。小伙伴们不要着急，我们一步步来学习。

1.1.3　编译时优化方案

a 引用是直接赋值的，b 引用是通过“+”赋值的，a 和 b 两个引用为什么会指向同一个内存单元？这就是 JVM 的“编译时优化”。如此神奇！小伙伴们惊呆了吧！

当编译器在编译代码：String a = "a" + "b" + 1;时，会将其编译为：String a = "ab1";。

为何？因为都是“常量”，编译器认为这 3 个常量叠加会得到固定的值，无须运行时再进行计算，所以就会这样优化。

疑惑：编译器为何要做此优化？

寓意：“小胖”说我的报销单写好了并盖章了，“小明”说我的也 OK 了，那么就合并一起邮寄报销单吧。“小锐”说我的快写好了，不过还没盖章，那你写好后再说吧。

寓意：为提升整体工作效率和节约资源，能提前做的事情就提前做。我们自己设计一种平台或语言的时候是否会考虑这些呢？

补充：编译器类似的优化还有许多（在后文中会有介绍），例如，当程序中出现 int i = 3 * 4 + 120 时，并不是在实际运行时再计算 i 的值，而是在编译时直接变成了 i = 132。

容易出错：JVM 只会优化它可以帮你优化的部分，它并不是对所有的内容都可以优化。例如，就拿上面叠加字符串的例子来说，如果几个字符串叠加中出现了“变量”，即在编译时，还不确定具体的值是多少，那么 JVM 是不会去做这样的编译时合并的。而 JVM 具体会做什么样的优化，不做什么样的优化，需要我们不断去学习，才能把工作做得更好。

同理证明的道理：String 的“+”操作并不一定比 StringBuilder.append()慢，如果是编译时合并就会更快，因为在运行时是直接获取的，根本不需要再去运算。同理，千万不要坚定地认为什么方式快、什么方式慢，一定要讲究场景。而为什么在很多例子中 StringBuilder.append()比 String 的“+”操作快呢？在后文中，胖哥会继续介绍原因。

1.1.4　补充一个例子

胖哥，我开始有点兴趣了，不过感觉在 String 方面还没过瘾！

OK，胖哥就再补充一个例子。

代码清单 1-2　String 测试 1

```
private static String getA() {return"a";}
public static void test2() {
    String a = "a";
    final String c = "a";

    String b = a + "b";
    String d = c + "b";
    String e = getA() + "b";

    String compare = "ab";
    System.out.println(b == compare);
    System.out.println(d == compare);
    System.out.println(e == compare);
    }
```

这段代码是说编译优化，看看输出是什么？

```
false
true
false
```

看到这个结果，如果你没有“抓狂”，一种可能是你真的懂了，还有一种可能就是你真的不懂这段代码在说什么。在这里，胖哥就给大家阐述一下这个输出的基本原因。

第 1 个输出 false。

“b”与“compare”对比，根据代码清单 1-2 中的解释，compare 是一个常量，那么 b 为什么不是呢？因为 b = a + "b"，a 并不是一个常量，虽然 a 作为一个局部变量，它也指向一个常量，但是其引用上并未“强制约束”是不可以被改变的。虽然知道它在这段代码中是不会改变的，但运行时任何事情都会发生，尤其是在“字节码增强”技术面前，当代码发生切入后，就可能发生改变。所以编译器是不会做这样优化的，所以此时在进行“+”运算时会被编译为下面类似的结果：

```
StringBuilder temp = new StringBuilder();
temp.append(a).append("b");
String b = temp.toString();
```

注：这个编译结果以及编译时合并的优化，并非胖哥凭空捏造的，在后文中探讨 javap 命令时，这些内容将得到实际的印证。

第 2 个输出 true。

与第 1 个输出 false 做了一个鲜明对比，区别在于对叠加的变量 c 有一个 final 修饰符。从定义上强制约束了 c 是不允许被改变的，由于 final 不可变，所以编译器自然认为结果是不可变的。

final 还有更多的特性用于并发编程中，我们将在第 5 章中再次与它亲密接触。

第 3 个输出 false。

它的叠加内容来源于一个方法，虽然方法内返回一个常量的引用，但是编译器并不会去看方法内部做了什么，因为这样的优化会使编译器困惑，编译器可能需要递归才能知道到底返回什么，而递归的深度是不可预测的，递归过后它也并不确保一定返回某一个指定的常量。另外，即使返回的是一个常量，但是它是对常量的引用实现一份拷贝返回的，这份拷贝并不是 final 的。

也许在 JIT 的优化中会实现动态 inline()，但是编译器是肯定不会去做这个动作的。

疑惑：我怎么知道编译器有没有做这样的优化？

答曰：懂得站在他人角度和场景看待事物的不同侧面，加以推导，结合别人的意见和建议，就有机会知道真相。

寓意：如果我是语言的作者，我会如何设计？再结合提供验证、本质、文档，共同引导我们了解真相。

理解方式：编译器优化一定是在编译阶段能确定优化后不会影响整体功能，类似于 final 引用，这个引用只能被赋值一次，但是它无法确定赋值的内容是什么。只有在编译阶段能确定这个 final 引用赋值的内容，编译器才有可能进行编译时优化（请不要和运行时的操作扯到一起，那样你可能理解不清楚），而在编译阶段能确定的内容只能来自于常量池中，例如 int、long、String 等常量，也就是不包含 new String()、new Integer()这样的操作，因为这是运行时决定的，也不包含方法的返回值。因为运行时它可能返回不同的值，带着这个基本思想，对于编译时的优化理解就基本不会出错了。

1.1.5 跟 String 较上劲了

看到这里你是否觉得自己对天天使用的 String 还不是特别了解，而胖哥之所以不断谈到 String 的一些细节，不仅仅是要说 String 本身有什么样的特征，在 Java 语言中还有许许多多的“类”值得我们去研究。

如果你觉得还没过瘾，和 String 较劲上了，那么关于 String 我们再来举个例子，然后胖哥做个小小总结，这里的故事就结局了。

代码清单 1-3　String 测试 2

```
public static void test3() {
    String a = "a";
    String b = a + "b";
    String c = "ab";
    String d = new String(b);
    println(b == c);
    println(c == d);
    println(c == d.intern());
    println(b.intern() == d.intern());
}
```

这段代码与上一个例子类似，只是增加了 intern()的调用。同样的，我们可以先看看输出是什么。

```
false
false
true
true
```

从输出上看规律，胖哥相信有不少小伙伴们应该能猜到是 intern()方法在起作用。是的，没错，就是它。

HotSpot VM 7 及以前的版本中都是有 Perm Gen（永久代）这个板块的（第 3 章中详述），在前面例子中所谈到的 String 引用所指向的对象，它们存储的空间正是这个板块中的一个“常量池”区域，它对于同一个值的字符串保证全局唯一。

如何保证全局唯一呢？当调用 intern()方法时，JVM 会在这个常量池中通过 equals()方法查找是否存在等值的 String，如果存在，则直接返回常量池中这个 String 对象的地址；若

没有找到，则会创建等值的字符串（即等值的 char[]数组字符串，但是 char[]是新开辟的一份拷贝空间），然后再返回这个新创建空间的地址。只要是同样的字符串，当调用 intern()方法时，都会得到常量池中对应 String 的引用，所以两个字符串通过 intern()操作后用等号是可以匹配的。

回到代码清单 1-3 的例子中，字符串"ab"本身就在常量池中，当发生 new String(b)操作时，仅仅是进行了 char[]数组的拷贝，创建了一个 String 实例。当这个新创建的实例发生 intern()操作时，在常量池中是能找到这个对象的，其余的内容，依此类推，可以得到为什么会这样的答案。

此时，能否有所感悟，没有的话没关系，这是第 1 节，胖哥带你一起感悟：

◎ String 到底还有多少方法我没见过？

◎ intern()有何用途？有什么坑？它是否会在常量池中被注销？

◎ 是否开始觉得这些东西复杂，又像是回到最简单的道理上来了？

◎ 是否认为自己的功底需要细化？

➤➤ 1.1.6　intern()/equals()

String 可能有很多的方法还没见过，在研究清楚 String 之前，还得看不少其他的代码。String 并没有我们想的那么简单，而且它在我们的工作中无处不在，所以需要去学习它。

胖哥在本节中也只是给大家讲个大概，String 中的许多奥秘，会在本书后面的章节中不时出现，大家留意哦。

如果仔细看了对 intern()的文档描述，就应该知道 intern()本身的效率并不高，它虽然在常量池中，但它也需要通过 equals()方法去对比，在常量池中找是否存在相同的字符串才能返回结果。显然，这个过程中对比的通常不止一个字符串，而是多个字符串，效率自然要更低一些。另外，它需要“保证唯一”，所以需要有锁的介入，效率自然再打折扣。因此，直接使用它循环对比得出的效率通常会比循环 equals()的效率低。但这并不是说对比地址比 equals()要慢一些，它是输在了对比地址之前需要先找到地址上。

它的效率低是相对 equals()来讲的，如果要细化认识这两者的效率并应用在实践中，那么我们要先看看 String.equals()方法的实现细节，请看图 1-2。

JDK 1.6的equals()

```
public boolean equals(Object anObject) {
if (this == anObject) {
    return true;
}
if (anObject instanceof String) {
    String anotherString = (String)anObject;
    int n = count;
    if (n == anotherString.count) {
    char v1[] = value;
    char v2[] = anotherString.value;
    int i = offset;
    int j = anotherString.offset;
    while (n-- != 0) {
        if (v1[i++] != v2[j++])
        return false;
    }
    return true;
    }
}
return false;
}
```

JDK 1.7的equals()

```
public boolean equals(Object anObject) {
    if (this == anObject) {
        return true;
    }
    if (anObject instanceof String) {
        String anotherString = (String) anObject;
        int n = value.length;
        if (n == anotherString.value.length) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = 0;
            while (n-- != 0) {
                if (v1[i] != v2[i])
                        return false;
                i++;
            }
            return true;
        }
    }
    return false;
}
```

图 1-2　String.equals()源码

不论是 JDK 1.6 还是 1.7，String.equals()的代码逻辑大致归纳如下。

（1）判定传入的对象和当前对象是否为同一个对象，如果是就直接返回 true（就像在代码清单 1-1 中一样，两个人拿到同一个公司的 offer）。

（2）判定传入对象的类型是否为 String，若不是则返回 false（如果是 null 也不会成立）。

（3）判定传入的 String 与当前 String 的长度是否一致，若不一致就返回 false。

（4）循环对比两个字符串的 char[]数组，逐个对比字符是否一致，若存在不一致的情况，则直接返回 false。

（5）循环结束都没有找到不匹配的，所以最后返回 true。

由此，我们至少得出两个最基本的结论。

（1）要匹配上，就要遍历两个数组，逐个下标匹配，这是最坏的情况。没想到吧，匹配上了还是最坏的情况。那么对于大字符串来讲，这件事情是悲剧的。

（2）字符串首先匹配了长度，如果长度不一致就返回 false。这对于我们来讲也是乐观的事情，如果真的遇上大字符串匹配问题，也许就有其他的技巧性方法哦。

好了，我们回过头来看看 intern()比 equals()还慢的问题。这个理论结果，可以用循环多次 equals()，然后循环多次 intern()，最后做“==”匹配来测试，得到的直接结论是：使用 equals()效率会更高一些。一些曾经认为 intern()对比地址效率高的“技术控”同学们，有点傻眼了。

而 intern()也并非无用武之地，它也可以用在一些类似于“枚举”的功能中，只是它是通过字符串来表达而已（其实枚举在底层的实现也是字符串）。

例如，在某种设计中会涉及很多数据类型（如 int、double、float 等），此时需要管理这些数据类型，因此通常是以字符串方式来存储的。当需要检索数据时，通过字典会得到它们的类型，然后根据不同的类型做不同的数据转换。这个动作最简单的方法可能就是用一个类型列表 for 循环 equals()匹配传入的类型参数，匹配到对应的类型后就跳转到对应的方法中。如果类型有上百种之多，自然 equals()的次数会非常多，而 equals()的效率很多时候

其实不好说，或许有些时候可以换个思路来做。

在加载数据字典的类型时就直接 intern()到内存中（因为这些类型是固定的），在真正匹配数据类型的时候就是“常量比常量”，对比地址，这样比较就快速多了，因为它不会使用 equals()，也不是匹配的时候再去调用 intern()。

举个例子：假如记录的是数据库的数据类型。这个中间平台在执行某些操作时需要检测每个表的元数据的每个列的类型，在很多时候这些元数据可能会被事先加载到内存中（前提是元数据不是特别多），并且以 intern()方式直接注入到常量池中，那么在后面逐个列的类型匹配过程中就不用 equals()了，而是直接用“==”，因为这些内容已经在常量池中了。

在这样的情况下，不同的表和属性如果类型是相同的，那么它们所包含的字符串对象也会是同一个，同时也可以在一定程度上节约空间。

这个方法比枚举要“土”一些，但是它也是一种方案，我们完全可以用枚举来实现，而枚举内在的实现也是类似的方式，只是它是“虚拟机”级别自带的而已。

intern()注入到常量池中的对象和普通的对象一样占用着相应的内存空间，唯一的区别是它存储的位置是在 Perm Gen（永久代）的常量池中。永久代会注销吗？会的，它也会被注销，在 FULL GC 的时候，如果没有任何引用指向它则会被注销。关于永久代的一些故事，我们还是放到第 3 章来讨论细节吧。

本章探讨的 intern()仅仅针对 JDK 1.6 环境，如果将环境更换为 JDK 1.7，String pool 将不会存在 Perm Gen 当中，而是存在堆当中，部分代码的测试结果将有可能更加诡异，请大家悉知。（在 JDK 1.7 环境下运行本章给出的例子，结果会有少数不一致，胖哥在本书光盘相应的测试类中增加了 testForJDK17()方法，大家可以单独看看，在 JDK 1.7 下可能还会有意想不到的效果。）

1.1.7　StringBuilder.append()与 String“+”的 PK

在很多的书籍和博客中，会经常看到某些小伙伴们用一个 for 循环几千万次，来证明 StringBuilder.append()操作要比 String“+”操作的效率高出许多倍。其实胖哥并不这么认为，为何呢？因为胖哥知道这是怎么发生的，下面就跟着胖哥的思路来看看吧。

胖哥认为：如果一件事物失去了价值，它就没有存在的必要，所以它存在必有价值。

在前面的例子中，String 通过“+”拼接的时候，如果拼接的对象是常量，则会在被编译器编译时合并优化，这个合并操作是在编译阶段就完成的，不需要运行时再去分配一个空间，自然效率是最高的，StringBuilder.append()也不可能做到。如果是运行时拼接呢？在后面内容的学习中，我们会发现一个 Java 字符串进行“+”操作时，在编译前后大概是如下这样的。

原始代码为：

```
String a = "a";
String b = "b";
String c = a + b + "f";
```

编译器会将它编译为：

```
String a = "a";
String b = "b";
StringBuilder temp = new StringBuilder();
temp.append(a).append(b).append("f");
String c = temp.toString();
```

注：为了说明实际问题，胖哥才用 Java 代码来说明字符串拼接中“+”的真正道理，在实际的场景中这是 class 字节码中的内容，而这个 temp 是胖哥虚拟出来的。

如果将 String 的“+”操作放在循环中，那么自然在循环体内部就会创建出来无穷多的 StringBuilder 对象，并且执行 append()后再调用 toString()来生成一个新的 String 对象。这些临时对象将会占用大量的内存空间，导致频繁的 GC。我们用图 1-3 来描述循环叠加字符串编译前后的代码对比。

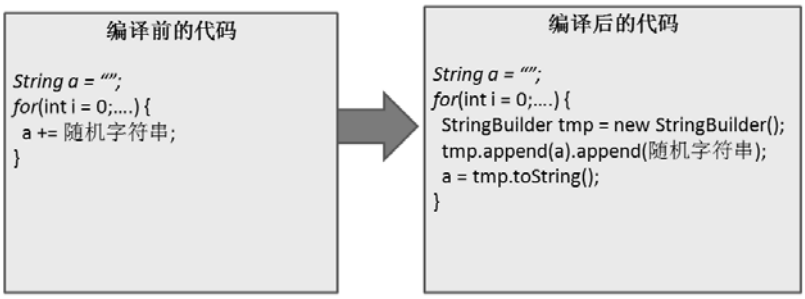

图 1-3　循环叠加字符串编译前后的代码对比

在循环过程中，a 引用所指向的字符串会越来越大，这就意味着垃圾空间也会越来越大。当这个 a 所指向的字符串达到一定程度后必然会进入 Old 区域，若它所占用的空间达到 Old 空间的 1/4，需要再次分配空间的时候，就有可能发生 OOM，而最多达到 1/4 的时候就肯定会发生 OOM。为何是 1/4？我们得先看看 StringBuilder 做了些什么。

上面的代码在循环内部会先分配一个 StringBuilder 对象，这个对象内部会分配 16 个长度的 char[]数组，当发生 append()操作时，如果空间够用，就继续向后添加元素；若空间不够用，则会尝试扩展空间。扩展空间的规则是：使用当前“StringBuilder 的 count 值 + 传入字符串的长度”作为新的 char[]数组的参考值，这个参考值将会与 StringBuilder 的 char[]数组总长度的 2 倍来取最大值，也就是最少会扩展为原来的 2 倍。

> count 值并不是 char[]数组的总长度，而是当前 StringBuilder 中有效元素的个数，或者说是通过 StringBuilder.length()方法得到的值。char[]数组的总长度可以通过 capacity()方法获取到。

关于 StringBuilder.append()功能的说明，我们来看看它的源码实现就更加清楚了，如图 1-4 所示。

```
public StringBuilder append(String str) {
super.append(str);
    return this;
}
public AbstractStringBuilder append(String str) {
if (str == null) str = "null";
    int len = str.length();
if (len == 0) return this;
int newCount = count + len;
if (newCount > value.length)
    expandCapacity(newCount);
str.getChars(0, len, value, count);
count = newCount;
return this;
}
void expandCapacity(int minimumCapacity) {
int newCapacity = (value.length + 1) * 2;
    if (newCapacity < 0) {
        newCapacity = Integer.MAX_VALUE;
    } else if (minimumCapacity > newCapacity) {
    newCapacity = minimumCapacity;
}
    value = Arrays.copyOf(value, newCapacity);
}
```

图 1-4　StringBuilder.append()的内存扩展

回到前面提到的“循环叠加字符串”的例子中，每次扩容会最少扩容 2 倍，而在每次扩容前，原来的对象还需要暂时保留，那么至少在某个时间点需要 3 倍的内存空间，自然 JVM 的 Young 空间的 Survivor 区域很快装不下，要让它进入 Old 区域。

当 a 引用的对象空间达到了 Old 区域的 1/4 大小时，同样会先分配一个 StringBuilder 对象，初始化的长度也是 16 个长度的 char[]数组。这个 StringBuilder 首先会进行 append(a)操作，在这次 append 操作时发现空间不够大，需要分配一个更大的空间来存放数据，但是这个字符串本身还不能被释放掉。

此时，字符串已经很大，那么肯定它不会只有 16 个长度，根据上面对 append()源码的描述，新分配的空间应当是 a 引用当前所引用的字符串的长度（因为当前的 count 值应当是 0，而同时也可以发现，其实 StringBuilder 内部已经没有空闲区了），所以 Old 区域最少占用了 1/4 + 1/4 = 1/2 的大小。

还没有完，还需要 append(随机字符串)来操作，此时 StringBuilder 是没有空闲区域的，那么自然空间是不够用的，又会涉及扩容的操作（只要随机字符串有 1 个字符），扩容最少是原来的 2 倍，就需要 1/4 *2=1/2 的空间，那么剩余的一半 Old 空间就又被用掉了。

分配一个 2 倍大小的空间，需要将数据先拷贝过来，原来的空间才能释放掉。但是这个时候还没分配出来，原来的空间自然释放不掉，那么 Old * 1/4 大小的 String 空间再一次叠加的时候就有可能将整个内存“撑死”。

假如 append()的随机字符串是空字符串" "，就不会发生扩容，这样是不是就不会发生 OOM 了呢？这还没有完哦，最后还需要 toString()方式来创建一个新的 String 对象，这个过程会开辟一个同样大小的 char[]数组将内容拷贝过去，也就是说，如果拼接前字符串所占用的空间已经达到了 Old 区域的 1/3 大小，就肯定会发生 OOM 了。如果你非要想想，这个时候的空间是不是就差“几个字节”就不会发生 OOM 呢？但是要考虑到下一轮循环是 100%会发生 OOM 的。

不过，大家不要因为后面还有一次 toString()需要分配空间，就认为前一种情况在内存空间 1/5 的时候就可以导致 OOM，要知道前面在发生扩容的时候，如果扩容成功，原来的数组就可以被当成垃圾释放掉了（这是在 StringBuilder 内部替换 char[]的时候发生的）。也就是说，原来的数组生命周期不会到 toString()这一步。

这里的例子是假设 JVM 所有的内存给一个 String 来使用，在实际的场景中肯定不是这样的，肯定会有其他的开销，因此内存溢出的概率会更高。

再回过头来看看，用 StringBuilder 来写测试代码时是怎么写的。

```
StringBuilder builder = new StringBuilder();
for(…) {
  builder.append(随机字符串);
}
```

这段代码中的 Stringbuilder 对象只有一个，虽然内部也只是先分配了 16 个长度的 char[]数组，但是在扩容的时候始终是 2 倍扩容。更加重要的是，它每次扩容后都会有一半的空间是空闲的，这一半的空间正好是扩容前数组的大小。

由于多出来的一半空间，通常不会在每次 append()的时候发生扩容，而且对象空间越大，越不容易发生扩容。更不会发生的是，每次操作都会申请一个大的 StringBuilder 对象并将它很快当成垃圾，并且在操作过程中还对这个很大的 StringBuilder 做扩容操作。

这种写法，最坏的情况是它所占用的内存空间接近 Old 区域 1/3 的时候发生扩容会导致 OOM（这里和 String 的拼接有点区别，如果遇到类似的差几个字节则导致 OOM，此时 String 的拼接在下一轮循环时必然会发生 OOM，但是 StringBuilder 扩容后会经历很长的一个过程不再发生扩容）。

另外，在 StringBuilder.append()中间过程中产生的垃圾内存大多数都是小块的内存，它所产生的垃圾就是拼接的对象以及扩容时原来的空间（下一次再发生 String 的“+”操作时，前一次“+”操作的结果就成了垃圾内存，自然垃圾会越来越多，而且内在扩容还会产生更多的垃圾），在这种场景下，通常使用 StringBuilder 来拼接字符串效率会高出许多倍。

所以，首先确认一点，不是 String 的“+”操作本身慢，而是大循环中大量的内存使用使得它的内存开销变大，导致了系统频繁 GC（在很多时候程序慢都是因为 GC 多造成的），而且是更多的 FULL GC，效率才会急剧下降。

但是回过头来想一想，在实际的工作场景中很少会遇到 for 循环几百万次去叠加字符串的情况，虽然许多不同的线程叠加字符串的次数加在一起可能会有几百万次，但是它们本身就会开辟出不同的空间（使用 StringBuilder 也是每个线程单独使用自己的空间，如果

是共享的就存在并发问题）。总而言之，如果是少量的小字符串叠加，那么采用 append()带来的效率提升其实并不会太明显；但是如果遇到大量的字符串叠加或大字符串叠加的时候（这些字符串不是常量字符串），那么使用 append()来拼接字符串效率的确会更高一些。

在 JVM 中，提倡的重点是让这个“线程内所使用的内存”尽快结束，以便让 JVM 认为它是垃圾，在 Young 空间就尽量释放掉，尽量不要让它进入 Old 区域。一个很重要的因素是代码是否跑得够快，其次是分配的空间要足够小（这些内容将在第 3 章中探讨）。

StringBuilder 的使用也是有“坑”的。上面已经看到 StringBuilder 内部也可能会创建新的数组，并不断将原来的数组数据拷贝到新的数组中，虽然不会像“+”那样每次操作都会出现拷贝，但是同样会出现很多的内存碎片。

如果你要优化到极致，则需要深知业务细节，申请的空间尽量确保有很少的拷贝和碎片，甚至于没有，例如：new StringBuilder(1024)。如果你是一个高手，对代码有洁癖，那么扩展 StringBuilder 实现的方法也是同样可以的。但是如果你不是很清楚具体的业务，则最好直接用默认的方式，其实它很多时候未必真的那么节约内存。为什么呢？

如果拼接的字符串不足 1024 个或差距很大，那么肯定是浪费空间的。换个角度，如果是 append(a).append(b)操作，a 的长度是 2，b 的长度是 1023，扩容是必然的，只是扩容前需要释放掉的 char[]数组长度是 1024（2KB 以上的空间），而且扩容后的长度是 2048（分配 4KB 以上的空间）。如果使用默认的 new StringBuilder()，也会发生扩容，扩容前需要释放掉的 char[]数组空间长度是 16（抛开头部的 32 字节），而新分配的 char[]数组长度是 1025 而不是 2048。

这也再次说明世事无绝对，一定要看场景，没有什么写法是绝对好的，使用这种方式，通常是希望在字符串拼接的过程中，将中间扩容降低到最少。

在类似的问题上，也有人问过胖哥：多个字符串拼接时，有的字符串很大，有的字符串很小，是先 append 大字符串还是小字符串呢？胖哥也不能给出明确的答案，只能说有一些不同的情况。

先添加小字符串，再添加大字符串，在什么时候比较好用呢？小字符串不是特别多，大字符串就一两个。在这种情况下，如果先 append 大字符串，在扩容的时候，可能不会进行 2 倍扩容，而是选择当前的 count 值+大字符串的长度来扩容，而且扩容后是没有剩余空间的，此时再来 append 小字符串，也会发生 2 倍扩容，自然浪费的空间会很多。如果先 append 小字符串，可能就扩展到几十个字符或几百个字符，几次扩容就可以搞定，而且扩容过程中的垃圾内存都是很小块的。由于板块小扩容的过程也是迅速的，后面 append 大字符串该有的扩容开销依然有，只是通常不会选择 2 倍扩容而已。

如果要添加许多小字符串（例如上千个小字符串），大字符串也不是太多，此时如果先 append 大字符串，在发生第 1 个小字符串 append 时，就会进行 2 倍扩容，那么就会有一半的剩余空间，这个剩余空间可以容纳非常多的小字符串，自然扩容开销就要小很多。反过来，如果先 append 许许多多的小字符串，由于初始化只有 16 个长度的 char[]数组，那么上千个小字符串叠加起来可能会发生 10 次左右的扩容，而扩容的空间也会越变越大。

关于 String 的“+”的补充说明

上面提到 String 的“+”会创建 StringBuilder 对象，然后再操作，那么粒度是多大呢？并不是“+”相关的两个 String 为粒度，而是一行代码为粒度。什么叫作一行代码为粒度

呢？例如：

```
String str = a + b + c + d + e;
```

这就是一行，它只会申请一个 StringBuilder 执行多次 append()操作，然后将其赋值给 str 引用，如果换成了两条代码就会申请多个。

但是这个“一行”并不是指 Java 代码中的一行，因为我们可以把 Java 代码中的很多行一层层嵌套到一条代码中，其实实际运行的代码还是多行，而是指可以连续在一起的代码。

关于内存拷贝和碎片，在本书 3.5 节中会有非常详尽的解释和说明。

本节胖哥用了十分简单的 3 个代码例子来说明有许多基础内容是值得我们去研究的。不知你是否能体会到其中的变化都是使用许多简单的基础知识加以变通的结果，这种变化是十分多的，但“万变不离其宗”。

本节从某种意义上印证了功底的重要性，即明白基本是明白内在的一个基础，上层都是由此演变而来的。

同样的，大家可以自己去研究 String 的其他方法，例如 startWith()、endWith()是如何实现的；如果代码中出现了反反复复这样的操作，是否有解决的思路。比如说要对每个请求的后缀进行 endWith()处理，自然就需要与许多的后缀循环进行匹配，而每个匹配都可能会去对比里面的许多字符，那么也许可以用其他的方法来处理。

本节胖哥是否对 String 做了一个诠释呢？没有，肯定没有，也不会！

本节所提到的内容，仅仅是想让你客观认识一下自己是否需要补充一些基础知识，希望你以此为例，学会看内在、看源码，活学活用。如果你认为自己“需要补充功底”了，胖哥的目的就达到了，此书分享过去的我的那些事，献给还在成长的你。

本节的故事到这里就结束了，胖哥不知道你是否满意，如果满意则祝愿你做一个美梦，以更好的心情和心态学习下一节的内容。

1.2 一些简单算法，你会如何理解

终于迎来第二次聚会的机会，本节内容会轻松许多，也许一盏茶的工夫就可以听完这个小故事。

注：其实本节并不是讨论算法，例子也会很简单，如果你对算法很熟悉，请跳过此节。

想要从一堆数据中找出一个 max、min。

想要从 100 万个数字中找出最大的 10 个数字。

你的想法是什么？你会如何找？先排序，再找，或者摸不到头脑。

胖哥的一些方法也许会帮到你：“想想学校里面排队、找人是怎么做的”。

假如一个学校有几千人，你要找一个人，胖哥给你提供几种方法，你选哪一种？

◎ 方法 1：几千个人我逐个去问，你是不是我要找的人，如果回答是，那么就是我要找的；如果不是，则继续问下一个。

◎ 方法 2：我知道学生的年级，然后在年级中每个班级按照方法 1 继续做同样的事情。

◎ 方法 3：我知道具体的班级，然后在班级里面像“方法 1”一样逐个问。
◎ 方法 4：我知道姓名，通过姓名得到有几个班级同名的人，这几个人我逐个问。
◎ 方法 5：我知道班级和姓名，也许就 1 个或 2 个学生同名，然后逐个问。
◎ 方法 6：我知道“学号”，直接找到人。

上面 6 种方法属于思维上的方法学，在不同的场景下使用不同的手段，但你会毫不犹豫地说：第 1 种方法肯定是最傻的方法。没错，但是我们的程序往往就是这样写的，一个长长的 for 循环加匹配，就是逐个问的道理，对于人数不多的情况它其实也是实用的，但人数多了自然会有问题。后面几种方法都是知道某些前提内容后，然后通过不同的索引手段找到了人。好了，下面我们回到正题，探讨几种计算方法的场景。

➤➤1.2.1　从一堆数据中找 max 和 min

在一堆数据中找到一个最大的数据，如果它是无序的，就不存在“班级、年级”的维度概念，也就是说，没有数据分组的概念，在这种场景下不得不去做一次数组遍历，和学校的学生不按照班级站队的道理一样。但是需要做一次排序吗？当然不需要，因为我们只要找出最大值和最小值就可以了，排序的代价太大了。如果要求最大值，则可以先给一个临时变量赋值为 Integer.MIN_VALUE 或数组中的第 1 个元素值，然后循环数组的每个下标，只要比当前的这个值大，就给它赋予最新的值。这个时间复杂度是循环一次就可以搞定的，自然就不需要排序那么大的时间复杂度了。而排序会是什么样的呢？

在学校排队的时候，学生有自己的主见，每人都有大脑（CPU），当“校长”（号令者）下达命令后，每个学生自己知道去找位置，可以将自己的大致位置找好，细节的可以再由班主任来排序，但计算机中的 CPU 是有限的，不能这样干，这种方式的时间复杂度减少不了，因此排序的代价是巨大的，除非有一天计算机的内存不通过 CPU 或很少通过 CPU 就可以自己排序。

寓意：“所有的算法来自生活，而且没有生活那么复杂。”

进一步来看这个问题，100 个人进行羽毛球比赛，最终需要挑选出水平最高的那一个人，最少会比赛 99 场。用这种方式就像擂台比武一样，倒下的人下台，胜利者继续在台上，在实际生活中，水平高的可能会累死，但在计算机中不会。

比赛的次数 99 次是最少的，有没有其他的方法呢？或许可以让两两之间进行较量，高手再两两之间继续较量，最后决出胜负，但是比赛的次数我们发现还是 *N*-1 次（例如 8 个人，首先 4 场胜出 4 人，然后 2 场胜出 2 人，最后 1 场加起来就是 7 场）。这种方式有一个重要的特征——在两两比赛的时候，只要有足够多的场地，就可以一起比赛，那么总体时间就节约下来了，在计算机中就是“多线程”技术。前者的设计方法由于简单，无法实现这种并行，但是它足够简单，只要 for 循环里面需要处理的内容足够简单，效率依然是很高的。

寓意：即使是多线程技术，在生活中也能找到活生生的例子。

将这个问题再做一点“发散性”扩展：在一堆杂乱无章的数据中，需要求出某个数据从大到小的“排名”情况，你会将所有的数据全部重排序一次吗？当然不会，因为完全重排序通常是 n^2 的时间复杂度。

不过，聪明的同学会给自己找一个理由：完全重排序后的数据是可以反复使用的，如果排序导致时间开销过多，则可以定时做完全排序，外部的请求直接从排序结果中

获取即可。这样我们需要忍受数据上的一点小延迟，因为这个排名可能随时在变化，由于是定时排序，所以变化后的数据并不会立即更新到排序后的结果中，或许在没有面临大数据的时候，换个思路会更快捷——在很多时候（只要数据量不是特别大），只需要找出比这个数据大的数据个数就自然知道排名了。这种思维其实是相通的，将上面的思路放在这里使用就是一种“变通”。

1.2.2 从 100 万个数字中找最大的 10 个数字

通过 1.2.1 节的介绍，胖哥认为你应该可以领悟到一些什么——既然在寻找最大数字的时候是保留一个最大值，那么在寻找 10 个最大数字的时候，就可以记录 10 个数字的最小值，或者将这 10 个数字排序也是很轻松的。

循环大数组的每个元素，如果当前数字大于这 10 个数字的最小值，就剔除最小值，将当前数字写进去。如果 10 个数字是有序的，那么可以快速定位数字要写入的位置；如果是无序的，则只需要重新找出最小值即可。如果数字比较随机，那么随着不断的叠加，这个最小值也会越变越大，这 10 个数字需要再次插入的概率就会变小很多，即使是在最坏的情况下，每个数字都要插入到 10 个数字中剔除最小的，也最多是 10×100 万的运算量，而不会是 100 万×100 万的运算量。

1.2.3 关于排序，实际场景很重要

胖哥算法学得不太好，在胖哥思维中的很多排序算法，没有什么排序算法是特别好的，或者在最坏的情况下，它们都半斤八两。在前面的例子中，一个学校要排队，每个学生都有自己的大脑（CPU），计算机要整体排序应如何排呢？

实际的场景很重要，这里假如有“杂乱无章”的数据（200 万个数据），两层 for 循环得到一个 n^2 的复杂度（200 万×200 万=4 万亿）。胖哥不想用，我们开始分析业务场景。如果我们发现这些数据中有 95%以上是相对均匀地分布在 1～200 万之间的，而且重复数据极少，那么在这种场景下会如何排序？还会用两层 for 循环去排序吗？

胖哥的第一感觉就是它是有技巧的，胖哥希望先分堆再排序。由于数据较为均匀，所以将它们分成 2002 份甚至于更多，其中 2000 份为 1～200 万之间（每个堆都是一个连续的区间，例如[1-1000]、[1001-2000]、[2001-3000]…），自然每个区间的数据范围是一个小堆，而且小堆之间的数据是有序的，“<1”的数据和“>200 万”的数据各自再分一个小堆，2002 个堆之间的数据是完全有序的，因此只要它们各自内部有序，就是全局有序。这个过程需要扫描 100 万数据一次，然后在 2002 个堆中找到自己的位置，由于它本身有序，所以采用二分查找算法是可以快速找到位置的（每次匹配范围会缩小 1 倍，在 2002 个范围中寻找，每个数字最坏匹配 11 次（因为 2 的 11 次方是 2048），总体上 11×100 万=1100 万）。

此时平均每个堆大概是 1000 条数据（200 万数据放在 2002 个堆中），那么每个堆自己排序的最坏时间复杂度将是 1000×1000=100 万，有 2002 个堆，因此是 100 万×2002=20.02 亿，加上分堆的开销是 20.135 亿，计算次数依然很高，但是可以看到，已经比 1 万亿这个数字降低了 500 倍。这就是根据业务场景来做逻辑优化的魅力，读者可以在这个基础上继续深度挖掘业务细节。

回顾：是否觉得这也是生活中的一些场景——先排好班级，每个班级之间就不用排序

了，而不是全校上千人一起来排队？它采用的是我们熟悉的分块排序，只是唯一的区别是，分块排序完成后就不需要再进行板块之间的排序了。这再次说明，我们实践中运用的还是“基础的算法”，在实际场景中需要变化，“变化后的招数才是最有攻击力的招数”。

内容扩展：当拆分成多个块以后，板块内部的排序是隔离的，因此各个板块是可以并行排序的，而且无须加锁，如果面对的是超大数据，还可以利用多线程来降低处理时间。将这个问题细化，就是分布式计算的基本原理了。

场景扩展：其实排序还有很多实际场景，远远不止这些，而且有些的确是很复杂的算法，不过无论多么复杂，总能在实际的生活中找到逻辑背景。在这种前提下，只要深度挖掘业务背景，就可以在基础算法之上进行灵活扩展。例如，有很多个已经排序后的分块，块内都是有序的，从第 1 个块到最后一个块的特征是后一个块的最小值大于前一个块的最小值，你会如何排序呢？

由于算法不是本书的重点，这里胖哥就简单说明一下。题目说明了每个块内部是有序的，且后一个块的最小值大于前一个块的最小值，第一想法自然是用后一个块的最小值，找出前一个块小于等于当前值的列表。由于本身是有序的，所以这样排序下来肯定是有序的，且剩下的数据肯定比当前取出的数据要大。

但是有个问题，剩下的数据就无法保证后一个块的最小值大于前一个块的最小值了，但是我们觉得这样“很爽”，还想要这样的情况继续发生。

在取出第 1 个块的数据后（不论取出几条数据），记录下现在的最小值，并认为是所有的最小值。进一步，下一个块取完后，得到最小值，如果它仍然大于前一个块的最小值，就用链表方式写在前一个块的后面；若小于前一个块的最小值，就放在前一个块的前面。我们也可以用一个单独的数组来存放每个块当前的最小值（这个长度与块个数一致，因此不会占用太多的空间），数组本身是有序的，相继递推的时候还可以用“二分查找算法快速定位”板块的位置。这样取完一次数据后，就得到了一个新的链表或者在分配的数组中记录了新的板块顺序。总之，现在是一个崭新的顺序，那么在进一步取数的过程中我们又可以轻松完成了，这也是一种分块排序的特殊场景。

说了这么多，胖哥还是在讲这句话：实际场景千变万化，切不可死记硬背。

➤➤1.2.4　数据库是怎么找数据的

按照 1.2.3 节中的例子，将一些数据划分为板块，但是由于数据太多，就会造成板块太多或板块内部的数据太多，此时就会有管理板块的板块。数据库通常承载的数据量都上亿（单表大小），甚至十亿以上，它是怎么做的呢？

其实数据库也离不开这些原理，大多数数据库的索引都会用换汤不换药的“B+树”（当然也有使用 Hash 来做索引的），此思想源自对平衡二叉树的一种扩展，而同时也来自基本算法——二分查找算法，只是在场景中也会有些变化，数据库中的分块、索引、排序等概念在这些理论中体现得非常明显。

例如，SQL 操作表与表之间的 join 的时候，如果是已经排序好的两个列表，join 就无须完全的两层 for 循环来操作，那是 $N \times N$ 的时间复杂度，此时有点类似于 1.2.3 节中的扩展例子，介绍如下。

假如 A 表与 B 表 join，到 B 表中查找 A 表的一个数据 100，它从 B 表最小的数据开始找，找到了许多小于 100 的数据，不过都排除在外，不论是否找到 100 这个数据，当去 B

表查找 A 表的下一个数据 101 的时候，就不会再从最小值开始找了，而是从 100 开始，因为数据库知道双方是有序的。随着这个规律的推移，扫描的数据范围会越来越小，而不是像两层 for 循环那样，每一个外部的 for 循环都需要在内存中从第 1 个下标开始扫描。

通常，如果 SQL 没有问题，join 表的个数往往不会太多，我们可以进一步做简单化优化。

1.2.5 Hash 算法的形象概念

其实 Hash 有很多种，有简单 Hash，还有一致性 Hash 等，这里我们就谈谈简单 Hash 算法。关于 Hash 算法，经常会在教材中看到：Hash 桶，快速定位，先计算再定位，并且号称复杂度为 *O*(1)。

胖哥有自己的看法，这样的概念容易让人迷惑 Hash 到底是“神马”东西，而容易认为 *O*(1)就是没有时间开销的，Hash 就是绝对定位的，Hash 算法就是无限快等思维都会出现。但实际场景也许并不是这样的。

胖哥不想搞得那么复杂，胖哥认为 Hash 算法有点像图 1-5 所示的那样（这里给大家一个简单的感性认识，在后文相关的专业内容介绍中，胖哥会再次讲到 Hash）。

简单地看，就是如果知道队名，就能知道在哪个范围，然后在范围内查找名称。在前面的一些介绍中已经有所体现，在找班级的时候就是这样找的，只是各自有自己的排序方法——这里按照“队伍”来分布，而前面的例子是按照“班级”来分布的。

在 Java 语言里 Hash 算法通常是按照 hashCode 来分布的，前面提到过，在不同的场景下这个 hashCode 会计算出不同的值——可用对象头部的一个标识符，可用某些对象的属性通过一定计算得到，同一个 hashCode 肯定在同一个分组上，这个分组也就是专业说法上的“桶”，“桶”内部的数据通常是无序的，通常内部用一个链表存放（有的小伙伴会问链表不是有序的吗？其实这里所谓的无序是指链表上存放的数据并没有要求外部写入的数据以先进先出或后进先出的顺序存放，外部程序也不能依赖于遍历的顺序）。

在查找数据时，首先根据 hashCode 找到桶，然后在这个桶内部需要逐个使用 equals() 方法来对比内部的值，而在极度冲突的情况下，许多数据将被放在同一个桶中（通常叫作“热点”），导致一个桶的数据链表会非常长，这样一来，分布在该桶上的查找、遍历等操作都会变得很慢。因此，千万不要因为 Hash 算法的复杂度是 *O*(1)就认为它天下无敌了，其实这只是一个相对量级别的说法，实际场景中的开销我们要用数字来说话。

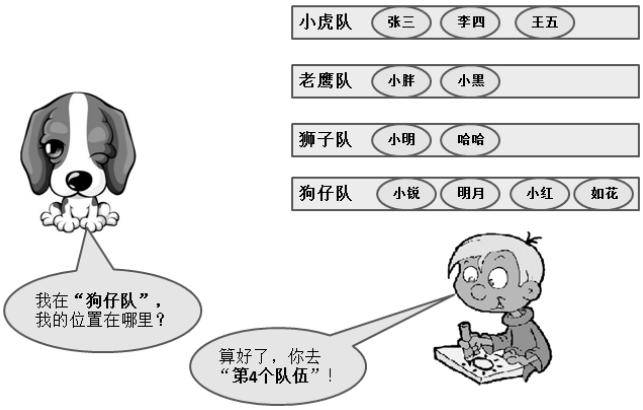

图 1-5　狗狗要找队伍和位置

反过来说，如果要使用 Hash 来做大量数据的查找，那么需要设计合适多的 Hash 桶。

另外，在设计 hashCode、hashCode 与桶的下标转换的时候要有足够的离散能力，否则会做“赔了夫人又折兵”的事情。

有人曾经用一种固定 Hash 的方式来存储网络上的 URL，喜欢钻空子的小伙伴利用了这种方式，形成一种网络攻击手段。他们在知道算法的基础上，通过模拟同样 hashCode 的不同 URL，使得某些“桶”内部的数据变得特别多，每一个分布到这个“桶”的请求都会在这个“桶”内部查找，最坏的情况就是一个也没有找到，那么就需要遍历整个链表，这样就可能使得系统性能极度下降。

看到这里，你若有所领悟，胖哥就心满意足了。虽然本节知识会相对枯燥些，但仍然在阐述一些和我们工作密切相关的基础思想。接下来我们再来看看基础知识“运算符”。

1.3　简单数字游戏玩一玩

数字游戏？没错，就是玩数字游戏！

Java 怎么玩？马上见证下！

玩数字有什么用途呢？我们不是虚拟数据给别人看，而是通过玩数字转换，让我们更了解计算机的数字运算，也许数字运算可以有一些神奇的地方，有些变态的问题也不是我们想的那么简单。

这里不讲基本的“四则运算”，胖哥会讲一些运算符，然后再讲讲“大数字”是如何处理的。

1.3.1　变量 A、B 交换有几种方式

胖哥认为有 3 种方法来实现变量交换，其中一种最简单的方法就是定义一个变量 C 作为中间量来实现，代码例子如下：

```
int C = A;
A = B;
B = C;
```

有人开始不想用“定义一个变量 C 作为中间量”来实现，而是尝试用“数据叠加后再减回来”的方法：

```
A = A + B;
B = A – B;
A = A – B;
```

这样也可以实现，A 首先变成 A 与 B 的和，然后减掉 B 自然就得到 A 的值再赋值给 B，此时 B 存储了 A 原先的值，再用 A 减掉 B 就得到原来 B 的值了。不过，这个方法有一个问题，就是 A + B 这个操作有可能会越界，越界后就会出现问题。

既然可能越界，聪明的小伙伴们又想到了另一个办法来解决这个问题，那就是“异或”运算：

```
A = A ^ B;
B = A ^ B;
A = A ^ B;
```

异或运算是按照二进制位进行“异或”的，对于 A、B 两个数字，如果某个二进制位

上的 A、B 都是 0 或都是 1，则返回 0；如果一个是 0，一个是 1，则返回 1，这样可以得到一个新的数字。这有什么用呢？当这个数字再与 A 发生“异或”的时候，就能得到 B（这个大家可以自行反推）。这样的技巧其实是二进制位上的一个加法但不进位的做法（由于只有 0、1，所以 1+1=2 后还是为 0），但是这个运算是最低级的 CPU 位运算，所以它的效率是极高的，而且不会越界。

“异或”运算在计算机系统中大量存在，它的一个很大的优势就是可以按照反向的规则还原数据本身。

1.3.2 将无序数据 Hash 到指定的板块

假如有一个长度为 5000 的数组，每个数组下标就像一个 Hash 桶那样存放一个链表，此时有一个数据 A 需要写入，随机吗？肯定不行，因为我们还要查询数据。那么将它放在 Hash 桶里面可以吗？似乎可以，下面一起来探讨一下。

我们希望写入的数据能按照某种规则读取出来，那么数据应当与数组的下标产生某种关系，这样写入和读取只要按照相同的规则就可以达到目的。此时假设一个“非负数”要写入，或许可以这样做，将这个数据%5000 就可以得到下标了，这样在数字不断变化时，得到的下标也会不断变化，数据自然会相对均匀分布到 5000 个数组中（当然有特殊情况，这里不做探讨）。如果是负数，则需要采用取绝对值 Math.abs(A) % 5000 来得到下标。

还有没有其他的方法呢？单纯要将数据写入到数组中，可以使用 A &4999 来完成，这个操作得到的数据将会永远<=4999，因为它自身二进制位为 0 的部分&操作后都会被清除为 0。理论上可以实现我们的要求，而且这种位操作也是最低级的系统运算，效率也是极高的。

只是这样的方法会有一个问题：数字 4999 是一个例子，此时我们用数字 11 来说明会更好一些，自然是要用 10 来做&操作，10 的二进制位是“1010”，那么意味着 0001、0100、0101 这些数字将永远无法获取到。宏观上讲，如果数组长度是 11，那么 1、8、9 这几个下标永远也没数据。

1.3.3 大量判定“是|否”的操作

如果发现一个业务系统中的许多操作都是在判定是、否，很多时候大家为了节约空间不会为每一个是、否的值用一个单独 int 来存储，因为那样比较浪费空间。而一个 int 是 4 个字节，所以它可以存放 32bit 二进制位，一个二进制位就可以代表一个“是|否”。

这样的设计是没有问题的，而且很多小伙伴们都知道这样的设计，但是胖哥发现有些可爱的同学在设计过后，拿出来判定对应的位置是与否的时候，是先用 Integer.toBinary String(int)转换为二进制字符串，然后从这个字符串中取出对应位置的 char，再判定这个 char 是'0'还是'1'的操作。

这种操作显然有点过了，而且操作的时间复杂度很高，或许我们可以学一学 Java 提供的一些类中是如何处理的。例如“java.lang.reflect.Modifier”，它对类型的修饰符有各种各样的判定（例如，字节码中存储 public final static 这样的一长串内容只用一个数字来表达），那么它是如何做到的呢？请听胖哥慢慢道来：Java 将这些符号进行了固定的规格编码，就像商品类型的名称与编号一样（这个编码会在生成 class 文件时存在，JVM 运行时也会使用）。

◎ 是否为 public，使用十六进制数据（0x00000001）表示。

◎ 是否为 static，使用十六进制数据（0x00000008）表示。十六进制的 8 代表二进制的最后 3 位是 100。

◎ 是否为 final，使用十六进制数据（0x00000010）表示。十六进制的 10 代表二进制最后 4 位为 1000。

在 Modifier 类中大家可以找到 private、protected、synchronized、volatile、transient、native、interface、abstract、strict 等各种类型的表达数字。

Java 程序在读取字节码时，读取到一个数字，该数字只需要与对应的编码进行“&”（按位求与）操作，根据结果是否为 0 即可得到答案。设计有点小技巧吧！

此时有的小伙伴就会说：“很多时候，需要判定它是不是 public final static 的，这么多类型要一个个判定好麻烦。”胖哥告诉你，这种方式其实还可以打组合拳。组合拳如何打呢？用“或”运算，具体操作就是：

```
final static int PUBLIC_FINAL_STATIC = PUBLIC | FINAL | STATIC;
```

这个值将会事先保存在程序中，由于这三项内容的二进制并不冲突，所以做或运算就代表三者都成立的意思。当传入参数后，只需要与这个值做“按位求与”就能得到结果了。不过，现在“按位求与”的结果不是与 0 比较，而是与 PUBLIC_FINAL_STATIC 比较是否等值（因为需要每个二进制位都要对应上，才能证明它的 3 个特征都成立）。所有的二进制位都必须要匹配上，即使一个不匹配也不行，示例如下：

```
public static boolean isPublicFinalStatic(int mod) {
    return (mod & PUBLIC_FINAL_STATIC) == PUBLIC_FINAL_STATIC;
}
```

1.3.4　简单的数据转换

数据转换有两种，其中一种是进制转换，另一种是数字与 byte[]的等值转换。我们先来看看进制转换。

◎ 将十进制数据转换为“二进制的字符串”，使用 Integer.toBinaryString()、Long.toBinaryString()来操作。大家注意了，这里说的是二进制的字符串，数字本身就是以二进制形式存储的，在 UI 上要看到这个数字的二进制位，所以才转换成字符串来显示。

◎ 将十进制数据转换为“十六进制的字符串”，使用 Integer.toHexString(int)来操作。类似的，在 float、double、long 中也有相应的方法存在。

◎ 将其他进制的数据转换为十进制数据，使用 Integer.parseInt(String,int)、Integer.valueOf(String,int)来完成，其中第二个参数传入的就是字符串数据。例如：Integer.valueOf("10",16)返回的值就是 16，Integer.valueOf("10",16)返回的值是 2，这样就可以实现任意进制之间的转换了。long 也提供了类似的方法。

数字与 byte[]之间的转换是什么意思呢？大家都知道，在 Java 语言中，int 是由 4 个字节（byte）组成的。在网络上发送数据的时候，都是通过 byte 流来处理的，所以会发送 4 个 byte 的内容，4 个 byte 会由高到低顺序排列发送，接收方反向解析。在 Java 中可以基于 DataOutputStream、DataInputStream 的 writeInt(int)和 readInt()来得到正确的数据，代码如图

1-6 所示。

DataOutputStream.writeInt(int)方法

```
public final void writeInt(int v) throws IOException {
    out.write((v >>> 24) & 0xFF);
    out.write((v >>> 16) & 0xFF);
    out.write((v >>>  8) & 0xFF);
    out.write((v >>>  0) & 0xFF);
    incCount(4);
}
```

DataInputStream.readInt()方法

```
public final int readInt() throws IOException {
    int ch1 = in.read();
    int ch2 = in.read();
    int ch3 = in.read();
    int ch4 = in.read();
    if ((ch1 | ch2 | ch3 | ch4) < 0)
        throw new EOFException();
    return ((ch1 << 24) + (ch2 << 16) + (ch3 << 8) + (ch4 << 0));
}
```

图 1-6　Data*XXX*Stream 对于数字的处理

如果使用了通道相关的技术，ByteBuffer 则由相关的 API 来实现。这是原始的实现方式，如果其他语言提供的 API 希望将这些 byte 位交换位置（例如要求低位先发送），那么你就要自己来处理了，而处理方式就可以参看 Data*XXX*Stream 的处理方式。

在 Data*XXX*Stream 的类中，不仅仅有对 int 类型的处理，还有对 boolean、byte、unsigned byte、short、unsigned short、char、int、long、float、unsigned float、double、UTF 的处理，而且还有一个 readFull()方法，这个方法要求读满一个缓冲区后才返回数据，它会在内部区循环处理。

在 ByteBuffer 的实现类中也提供了各种数据类型的 put、get 操作，内部也是通过 byte 转换来完成的。

在 java.io.Bits、java.nio.Bits 类中也有大量的数字与 byte[]的转换操作（这两个类我们的程序无法直接使用，但是可以看源码），大家可以参考它们的代码来实现自己的特定需求。

1.3.5　数字太大，long 都存放不下

long 采用 8 个字节来存放数字，它连时间的毫秒值、微秒值甚至纳秒值都可以存放下，它存放不下的数字很少见，但是它毕竟只有 8 个字节，如果真的遇到了某些变态的设计怎么办呢？此时需要使用 BigDecimal 来操作，它不是以固定长度的字节来存储数据，而是以对象的方式来管理位的。我们用一个例子来看看使用 BigDecimal 对一个大数字操作包含几个步骤。

1. 首先将大数字加上 10，输出结果。
2. 然后将结果的 byte 位取出来以二进制形式展现。
3. 最后根据二进制字符串反转为数字对象。

代码清单 1-4　BigDecimal 测试

```
import java.math.BigDecimal;
import java.math.BigInteger;

public class BigNumberTest {

    private static String lPad(String now ,
                               int expectLength ,
                               char paddingChar) {
        if(now == null || now.length() >= expectLength) {
            return now;
```

```
            }
            StringBuilder buf = new StringBuilder(expectLength);
            for(int i = 0 , paddingLength = expectLength - now.length();
                                    i < paddingLength ; i++) {
               buf.append(paddingChar);
            }
            return buf.append(now).toString();
      }

      public static void main(String []args) {
            //这个数字 long 是放不下的
            BigDecimal bigDecimal
             = new BigDecimal("123324324324324324324324324324324324241432423432");
            System.out.println("数字的原始值是：" + bigDecimal);

            //bigDecimal = bigDecimal.add(BigDecimal.TEN);
            //System.out.println("添加 10 以后：" + bigDecimal);

            //二进制数字
            byte[] bytes = bigDecimal.toBigInteger().toByteArray();
            for(byte b : bytes) {
                  String bitString = lPad(Integer.toBinaryString(b & 0xff) ,
                                                8 , '0');
                  System.out.println(bitString);
            }
            //还原结果
            BigInteger bigInteger = new BigInteger(bytes);
            System.out.println("还原结果为：" + bigInteger);
      }
}
```

在这段代码中，有一个 substring()、lPad()操作。这是因为 Integer.toBinaryString(b)传入的虽然是 byte，但是会转型为 int，如果 byte 的第 1 个 bit 位是 1，则代表是负数，那么转换为 int 的高 24 位将会填充 1。其实我们不需要这 24 位，所以用了 substring()。如果是正数，那么输出的字符串会将前面的 0 去掉，为了在显示上使用 8 位二进制对齐方式，所以在代码中用了 lPad()。我们再来看看输出结果。

```
数字的原始值是：123324324324324324324324324324324324241432423432
添加 10 以后：123324324324324324324324324324324324241432423442
00110111
01001100
11110000
00101001
11111111
10001010
00010101
00001101
00011100
01111111
00101001
00000001
01000111
01000110
10110011
01111000
01100100
00011100
00001000
还原结果为：123324324324324324324324324324324324241432423442
```

数字能转换为二进制数据，又能还原成数字，理论上应该没有问题，大家也可以用一

些比较小的数字做测试来印证这个结论（例如 2、15、16、1024 等）。不过，在印证的过程中，有的同学发现高位出现整个字节的 8 位全是 0 的情况，例如 128，输出的结果是两个字节的二进制字符串：0000000011111111。大家很疑惑：既然高 8 位全是 0，为何还要单独拿一个 byte 来存放呢？有一点要注意了，因为 128 是正数，而一个字节中的最高位变成了 1，如果直接用 11111111 来表达就表示它是一个负数（具体值是-1），所以需要多一个字节来表达自己是正数。

long 的存储能力到底有多大？

许多同学设计一些 PK 列的时候认为数字存储不下，用字符串来存放数字（当然不排除某种编码规则），但是用数字往往计算快速得多，而且空间也少很多。为了了解 long 的存储能力有多大，我们先看看 int 的存储能力有多大。Integer.MAX_VALUE 的值为“2147483647”，也就是 2G-1 的大小。如何得来？自然是 4 字节对应 32 位，然后去掉负数和 0 得来的。这个值换算成十进制数据就是 21 亿，理论上很多表达不到那么大，或者说绝大部分表不会有这么多 ID，但是若某些跳跃和历史数据超过这个数字怎么办？那么就用 long 吧，它的宽度是 int 的 2 倍，即 4G*4G/2–1（为什么？自己想想）。十六进制数据 0x7fffffffffffffffL，换算为十进制数据是“9223372036854775807”，这个数字看不出到底有多大，我们可以和时间进行比较。

Long.MAX_VALUE / (365l * 24 * 60 * 60 * 1000) = 292471208

Long.MAX_VALUE / (365l * 24 * 60 * 60 * 1000 * 1000) = 292471

Long.MAX_VALUE / (365l * 24 * 60 * 60 * 1000 * 1000000) = 292

如果存放毫秒值，则可以存放 2.92 亿年，微妙值可以存放 29.2 万年，纳秒值可以存放 292 年的数据。换句话说，一个表的 ID，即使加上跳跃每秒有 10E9 个 ID 自动增长，也可以增加 292 年的数据不重复，胖哥认为这种并发在现在的计算机时代还不存在。小伙伴们，通过这种量化后，自然应该理解某些设计应该如何做了吧。

数字游戏玩到这里就结束了，在本书的后文中，胖哥还会提到一些和数字相关的小玩意，例如 i++与++i 的问题，或许胖哥的说法与在教材中看到的解释不一样哦。

1.4 功底概述

本节是本章的“道”第一次总结，胖哥会尽量简单说明。

1.4.1 什么是功底

古人有句话：“心有灵犀一点通”，形容相互之间的交流十分默契，而融洽的基础在于彼此十分了解对方。在专业领域上要做到这一点，就是要深深地了解技术以及它的内在。

武侠中有一种武学叫“九阳神功”，也有一种武学叫“易筋经”等，它们之所以很神奇，是因为它们都有一种不变的特征，就是学习了这些武功后，再学习任何武学都很快，而且可以比别人学得更好。同时，这些“武学秘籍”也有一个共同的思想，就是以内功修为为主，而不是以招式为主。

胖哥写的书自然不敢与这些“神”一样的“秘籍”相提并论，胖哥只是借此告诉大家内在修为将会决定你在技术这条路上能走多远。

胖哥无法帮助你成为“张无忌”一样的绝世高手，但是一个真正的老 A，也同样需要有很深厚的内功修养；否则，当新的事物出现时，你会跟不上潮流，“out”了，正所谓：练武不练功，到老一场空。

在功底方面，我们还需要做到知其然并知其所以然，形成一种由知识引导思维，由思维引导答案，由结果印证理论，不断迭代的过程。在这个过程中，对问题的认识会越来越清晰，自然在见解上也会越来越有道理，同时也会对自己所在的领域充满自信。

1.4.2　功底有何用途

第 1 节讲解了 String 的故事，第 2 节讲解了关于一些算法的故事，第 3 节我们开始玩数字游戏，是否觉得自己用了很久的 Java，好多东西还不知道？当然，如果你认为胖哥说得过于小儿科，那么就跳过去吧。

在前面的这些知识中，很多技术本质源于生活，很多意想不到的事情发生了，这一切在本书中才刚刚开始，或许后续章节中的一些内容会更加让你感觉出乎意料。这些原本认为是真理的内容，被彻底颠覆，在实际的场景中也许就在不经意之间，你会留下一个很难琢磨的 Bug，或者根本不会认为是 Bug 的 Bug，然后让别人来解决这个 Bug。正所谓：我们不怕犯错，怕的是不知道什么是错，更怕的是我们一直坚信正确的事竟然是错的。

很多时候，印证了一个通过测试得到的经验并不那么靠谱，胖哥的例子也许已经令你感受到测试场景可能会决定许多不同的结果。在不知道本质的时候，仅仅通过某些特定的测试场景是不能作为真理来指导开发的，最多只能是指导同样场景下的开发。测试通常可以帮助我们去理解本质，以及在知道本质后它是一个印证的过程。

知道了内在，就像知道了 String 的编译优化、String 的常量池、String 的内存结构，知道了常规的算法，知道了生活中的对应方法，我们发现它并不是那么难，关键是你是否愿意去看它的本质。其实看本质对于许多程序员来讲是一件拥有快感的事情，因为生活的现实世界在自己的工作中找到了灵感，但是它又不像人性那么复杂，它比生活更加简单。

只是我们过于看中技术本身，过分仰望技术本身，“容易陷入深渊，而难以自拔”，甚至有人喜欢钻牛角尖，对于这样的同学来讲，胖哥只能说“苦海无涯”，你需放下才能找到新的方向，需解脱和放下才能超越自我。

总的来讲，胖哥认为：只有不断颠覆自我，找出本质，才能将一些问题连根拔起。而不是过分相信自己曾经做过的实验和别人说过的所谓真理，这些都仅仅作为参考而已。

1.4.3　如何磨练功底

功底并非天高，任何人只要愿意，能静心，就都行！

大多数初学者，对能做出一个例子，或做非常多的例子引以为傲，其实这并不是坏事，只是学习后一定要“落实于根本，回顾与总结”，切勿让“猴子掰玉米”的故事发生在你的身上。换句话说，浮躁的心态是“猴急”，什么都想要去学习，结果学了就丢了，没有自己知识的凝固，没有自己的总结和笔记。

首先，要能静心，静心才能落实，立足于当下，要知道学海

图 1-7 “一休”同学已静下心，在思考

无涯、浩瀚无边，并且社会在发展，知识在进步，我们的确需要不断学习，但是要从当下做起。

当我们静下心来，就可以开始总结了，总结啥呢？感想吗？也可以，不过老这么总结就不像 IT 从业者了，而像一个感情丰富的编辑。我们刚开始做总结有点像记笔记，就是将老师所讲，或自己所学记录下来，按照条目一条条地陈列。但是这不算总结，只能算是笔记，这些内容在许多的书籍和网站上可以找到更好的内容，而总结应当包含自己的理解和感受，应当能阐述学习和理解过程中的痛苦。

在技术领域你可以总结思路，总结所学所用，总结所遇到的问题，总结为什么，总结解决问题的手段和方法，分析问题和定位问题的思路，总结在解决问题时所用的知识，总结技术点上是否还有相关的方案，以及对比相关知识的优缺点。

在业务上总结自己最近做过些什么，那些是大事还是小事，自己所做的事情和几个月前比提高没有，效率上提升没有，时间安排如何，如果加班太多能否不加班，自己的工作效率是否可以进一步提升，以及如何提升，通过技术完成某些自动化还是基于时间管理来更合理地安排自己，未来几个月准备或希望去做什么样的事情，是否有能力去参与更重要的职责。

当然，我们不是每天都要去做总结，那样会很浪费时间。

即使每天去做总结，也是一个小总结而已，如果我们非常忙，那就没必要这样做。但是当我们看到一篇比较好的文章时，可以先初步看看内容，把它记录下来，写到总结中表示想要去学习的内容，等到闲暇之余去看看，再进一步来总结或许会更好。

本节我们用一个简单 String 等值对，就能说出一大堆基础知识，也能说出类似的方法，相应的算法也能联想出来，你不觉得这样的由小的点入手引导相关知识的方法也是一种不错的学习方法吗？

工作与学习本身并不矛盾，工作本身就是一种学习，只是它更偏重于实践，基于工作驱动学习一个新东西会更快一些，不过大多在使用层面。当遇上了稀奇古怪的问题时，就想要去看看内在的原理了，此时去看看源码，你会更加有动力。这样的碎片化学习方式，其实也是一个积土成山的过程，达到一定程度自然是“量变发生质变”，那个时候再去看看“牛人的秘籍”，就是一种交流，一种知识梳理的过程。

也许刚开始我们无法做到很深入的细化与知识联想，尤其是某些相关技术点需要很大的知识面才能把它讲清楚，技术需要理论，理论又需要技术，这是一个死循环，我们便开始产生了“鸡与蛋”的纠结问题，许多人在长期的纠结中选择了原地踏步，也就是相当于放弃了。其实我们需要的还是立足于当下业务，尽自己所能去挖掘，通过业务细化驱动技术的发展，尽我们所能广泛联系知识，整个学习过程是一个迭代的过程，功底是逐步提升的，不要纠结于死循环的问题。

如果你希望自己从事技术这个领域，就要学会相信自己，拥有一定的自信，有信心不是说自负，而是有信心去面对困难。当你有一定成就的时候，不可狂傲，要知道天外有天，人外有人，世界上没有最高，只有更高，即使是今天的最高，明天也会有更高出现。而我们的功底将支撑自己的这座山峰，我们更加愿意去追求自己的极限，追求自己理想的巅峰。

1.5 功底补充

看完 1.4 节，发现胖哥废话很多，貌似没啥干货了！

为了不让大家认为功底只有 String 那么一点点东西，胖哥就再增加对原生态类型、集合类的说明，这两方面的内容相信所有的 Java 开发者都必然会用到。

1.5.1　原生态类型

原生态类型是“神马”？

原生态类型就是 Java 中不属于对象的那 5%部分。

那到底是哪些东西呢？

包含：boolean、byte、short、char、int、float、long、double 这 8 种常见的数据类型（Primitive）。

好麻烦，为啥会用它们呢？用对象不可以吗？

计算机中的运算基础都来源于简单数字，包括 Java，即使是包装后的对象（Wrapper），在真正计算的时候也是通过内在的数字来完成的，Java 失去了它们，就好比鱼儿失去了水一样，失去了生命力。

它与包装后的对象有什么区别呢？

包装后的对象会按照对象的规则存储在堆中（例如，int 所对应的就是 Integer 类的对象），而“线程栈”上只存储引用地址。对象自然会占用相对较大的空间存放在堆中，在原生态类型中，“栈”上直接保存了它们的值，而不是引用（Reference）。

下面看个例子。

代码清单 1-5　一个 Integer 的简单测试

```
public static void main(String []args) {
    Integer a = 1;
    Integer b = 1;
    Integer c = 200;
    Integer d = 200;
    System.out.println(a == b);
    System.out.println(c == d);
}
```

输出结果：

```
true
false
```

这个结果在较低版本的 JDK 当中不会出现。

现在胖哥来解释一下这个结果。

在编译阶段，若将原始类型 int 赋值给 Integer 类型，就会将原始类型自动编译为 Integer.valueOf(int)；如果将 Integer 类型赋值给 int 类型，则会自动转换调用 intValue()方法，如果 Integer 对象为空，这时会在自动拆箱的时候抛空指针（这个自动转换可以通过后文中介绍 javap 命令的方法来证明）。

这些赋值操作可能不是那么明显，例如一些集合类的写入、一些对比操作，这就需要我们知道什么时候会自动拆装箱。换句话说，它简化了代码，但是并不是让我们一无所知。

即使是这样，两个结果也应该一样，要么都是 true，要么都是 false，但为何不一样呢？这算是一个 Java API 的坑，如果我们不知道这些坑，稍微不留神就会掉进去。知道了装箱是 Integer.valueOf(int)方法，那么就来看看 Integer.valueOf(int)方法的源码，如图 1-8 所示。

```
public static Integer valueOf(int i) {
    if(i >= -128 && i <= IntegerCache.high)
        return IntegerCache.cache[i + 128];
    else
        return new Integer(i);
}
```

图 1-8　Integer.valueOf(int)方法的源码截图

根据代码可以看出，当传入 i 的值在[-128, IntegerCache.high]区间的时候，会直接读取 IntegerCache.cache 这个数组中的值。

在代码中为什么使用 i + 128 作为数组的下标呢？

因为数组下标是从 0 开始的，而表示的数字范围是从-128 开始的，加上 128 就正好对上了。

继续跟踪源码可以得到，在默认情况下 IntegerCache.high 是 127。也就是说，如果传入的 int 值是-128～127 之间的数字，那么通过 Integer.valueOf(int)得到的对象是被 cache 的，自然的，对于同一个数字 cache 的对象是同一块内存地址，所以第 1 个输出结果是 true。第 2 个输出已经不在这个范围，因此会重新 new Integer(int)（创建一个新对象），所以得到的结果是 false。

我们可以通过设置 JVM 启动参数-Djava.lang.Integer.IntegerCache.high=200 来间接设置 IntegerCache.high 值，也可以通过设置参数-XX:AutoBoxCacheMax 来达到目的（这个不用查官方资料，看看源码以及源码周边的注释就懂了）。如果要将这个值变得更大来满足自己的需求，则可以在启动参数中增加该值（缩小也是一样的道理）。

这好像是做好事，将我们的数据 cache 起来，更加节约空间了，但是有的同学开始认为 Integer 可以用“==”匹配了，因为大家自己“测试”的时候发现 1、2、3、4 等数据都是没有问题的，但是程序发布后出现了诡异的问题，而这个最不容易被认为是问题的地方却真的成了问题。而 Java API 中没有明确地说明这一点，但开发人员不会将官方文档都学习一遍再来做开发吧，所以我们说它是“坑”。

有人问：真正的场景中会这样吗？

胖哥认为：肯定会，而且你遇不到的场景并不代表不会发生，今天遇不到的事情并不代表明天不会发生。例如，在某些工程设计中，某些状态值有特殊的意义，如果它们是非连续排布的，那么不在-128～127 范围内的可能性是肯定存在的。

这个例子很简单，我们学到的应该不止这些，因为坑无处不在，我们要学会看源码和本质；否则，即使是 Java 本身提供的 API 出现了“坑”，也会让我们“防不胜防”，在技术面前变得十分“可怜”。

我们可以说这个 API 写得不好，没有说明详细的使用情况，但是一个老 A 不应当被“武器”所玩弄，而是要驾驭武器，老 A 即使拿一把普通步枪也同样能战胜拿着“狙击步枪”的普通士兵，因为他们除了拥有极强的战斗素质外，还深知武器的脾气与秉性，这是人与

武器之间的驾驭和被驾驭关系。

也许自动拆装箱还有另一个很大的“坑”，就是如果大家不知道自动拆装箱是怎么完成的，可能就会有更多的问题发生，在程序中传递参数可能一会用 Integer 类型，一会用 int 类型，自然的就一会在做拆箱操作，一会在做装箱操作，这貌似没有太大的问题，但每次装箱的时候都有可能会创建一个对象（因为很多时候数字不一定在 cache 范围内，较低版本的 JDK 是没有 cache 的）。另外，这种装箱操作是很隐藏的。例如，我们想要用一个 int 类型的数字来作为 HashMap 的 Key，那么在 put()操作的时候就会自动发生装箱操作（因为 Key 被认为是 Object 的，HashMap 需要获取这个对象的 hashCode()方法来做离散规则，所以它会自动转型为 Integer）。同样的，如果想将许多基本类型的数据放在 List 里面，在 add()操作的时候也会自动发生装箱操作。此时，如果数据取出来后变成了基本类型，再用这个基本类型放入另一个集合类，就又会发生装箱操作，在这个过程中就会隐藏地浪费大量的空间，而自己却什么也不知道。

关于对象空间的大小，请参看第 3 章的内容。

□ 横向扩展

通过对 Integer 的一些了解，想到了 Boolean、Byte、Short、Long、Float、Double，它们是否有同样的情况，胖哥不想写重复的东西，直接给出结果，大家可以自己去看看代码，看看这些类型中的 valueOf()方法是如何操作的，或者说自动装箱是如何操作的。

◎ Boolean 的两个值 true 和 false 都是 cache 在内存中的，无须做任何改造，自己 new Boolean 是另外一块空间。

◎ Byte 的 256 个值全部 cache 在内存中，和自己 new Byte 操作得到的对象不是一块空间。

◎ Short、Long 两种类型的 cache 范围为-128～127，无法调整最大尺寸，即没有设置，代码中完全写死，如果要扩展，需自己来做。

◎ Float、Double 没有 cache，要在实际场景中 cache 需自己操作，例如，在做图纸尺寸时可以将每种常见尺寸记录在内存中。

□ 思维发散扩展

如果上面的操作变成 Integer 与 int 类型比较会是什么样的结果呢？如果是两个 Integer 数据做“>”、“>=”、“<”、“<=”比较，做 switch case 操作，会得到什么结果？反射的时候是否有特殊性？

这个结果大家可以去论证，且测试结果可以就认为是当前虚拟机的设计规范。下面胖哥直接给出结果。

◎ 当 Integer 与 int 类型进行比较的时候，会将 Integer 转化为 int 类型来比较（也就是通过调用 intValue()方法返回数字），直接比较数字，在这种情况下是不会出现例子中的问题的。

◎ Integer 做“>”、“>=”、“<”、“<=”比较的时候，Integer 会自动拆箱，就是比较它们的数字值。

◎ switch case 为选择语句，匹配的时候不会用 equals()，而是直接用“==”。而在 switch case 语句中，语法层面 case 部分是不能写 Integer 对象的，只能是普通的数字，如果是普通的数字就会将传入的 Integer 自动拆箱，所以它也不会出现例子中的情况。

在 JDK 1.7 中，支持对 String 对象的 switch case 操作，这其实是语法糖，在编译后的字节码中依然是 if else 语句，并且是通过 equals()实现的。

◎ 在反射当中，对于 Integer 属性不能使用 field.setInt()和 field.getInt()操作。在本书的 src/chapter01/AutoBoxReflect.java 中用例子来说明。

1.5.2 集合类

如果读了上一节后你有所体悟，那么胖哥认为你可以跳过此节，因为此节知识为上一节的一个平行扩展，我们不重视知识点本身，而在于让大家了解到许多秘密。

集合类非常多，从早期的 java.utils 的普通集合类，到现在增加的 java.util.concurrent 包下面的许多并发集合类，其实我们有些时候只是知道它们是很好用的东西，但在遇到某些问题的时候是否会想到是它们造成的（就像 String 一样），它们的使用技巧有哪些？它们的设计思想是什么？

本节不讨论并发包，就简单说说集合类的故事。

疑惑：集合类中包含了 List、Map、Set 几大类基本接口，而我们最常用、最简单的集合类是什么呢？

答曰：ArrayList、HashMap。

那么，当你用 ArrayList 的时候是否想起了 LinkedList、Vector；当你用 HashMap 的时候是否想起了 TreeMap、HashSet、HashTable；当你要排序的时候是否想起了 SortedSet 等。

它们有何区别？在什么情况下使用？

在讨论 String 后面的部分内容中，我们提到了 StringBuilder，它内在的数组的实现有大量的拷贝，这在集合类的内存拷贝方面的体现更加明显，并且占用空间更大。

占用更大空间的原因是集合类都是存储对象的引用的，在 32bit 及 64bit 压缩模式下，一个引用会占用 4 个字节，在 64bit 非压缩模式下会占用 8 个字节，而 StringBuilder 只是存储 char 字符的数组，每个位只占用 2 个字节。

此时以 ArrayList 为例，我们看看它的 add(E e)方法源码，如图 1-9 所示。

```
public boolean add(E e) {
ensureCapacity(size + 1);
elementData[size++] = e;
return true;
}
public void ensureCapacity(int minCapacity) {
modCount++;
int oldCapacity = elementData.length;
if (minCapacity > oldCapacity) {
    Object oldData[] = elementData;
    int newCapacity = (oldCapacity * 3)/2 + 1;
        if (newCapacity < minCapacity)
    newCapacity = minCapacity;
        // minCapacity is usually close to size, so this is a win:
        elementData = Arrays.copyOf(elementData, newCapacity);
}
}
```

图 1-9 ArrayList 的 add(E e)方法源码截图

通过源码我们发现，如果空间不够，会通过 Arrays.copyOf 创建一个新的内存空间，新空间的大小最小为原始空间的 3/2 倍+1，并将原始空间的内容拷贝进去。

这里所提到的新空间的大小为原始空间的 3/2 倍+1，是最小的，在 add(E)方法中不会发生，而在 addAll()方法中会发生。addAll()允许同时写入多个数据，如果写入的数据较多，每次按照 1.5 倍数扩容，可能发生多次扩容，这样就会有多余的垃圾空间产生，addAll()操作就会对比写入的量与 1.5 倍的大小，谁大就用谁，这个道理在 StringBuilder 中我们就知道，因此文中提到的是“最小”。

我们知道，ArrayList 是基于“数组”来实现的（本书 3.5 节会详细介绍内存结构），如果遇到 remove()操作，add(int index)指定的位置写入操作，我们有没有虑过 ArrayList 内部其实会移动相关的数据，而且随着数组越长，移动的数据会越多。如果要替换一个数据，我们会不会先 remove 再 add 一个数据，或者是通过 set(int index , E e)将对应下标的数据替换掉。

基于数组的 ArrayList 是非常适合于基于下标访问的，这是它擅长的地方（又回到基本的数据结构与算法了）。下面胖哥给出几个简单扩展，希望大家去思考。

◎ 在经常做修改操作的列表中，或者在数组通过下标检索并不是那么多的情况下，你是否考虑过使用 LinkedList 呢？因为 ArrayList 通常始终有些数组元素是空着的。

◎ 在知道 List 长度范围的情况下，你是否在实例化 ArrayList 的时候带上长度？例如 new ArrayList(128); 这样就降低了内存碎片和内存拷贝的次数。

◎ 当 List 太大的时候是否考虑过对它做分段处理，而不要一次加载到内存中？其实很多 OOM 都会在集合类中找到问题。

◎ 常见的框架中用了什么集合类？在什么情况下也会出现问题？

大家熟知的 HashMap 浪费空间更加严重，它的代码里面有一个 0.75 因子，当写入 HashMap 的数据个数（不是说所使用的数组下标个数，而是所有元素个数，也就是说，包含了同一个下标的链表中的所有元素个数）达到数组长度的 0.75 后，数组会自动扩展 1 倍，并且还需要做一个 rehash 操作，其实这个时候也许很多桶上的节点都是空的。

胖哥不想扯太多的集合类出来，把读者“读晕”，大家在理解这两个基本的集合类基础上，再去看其他的集合类也许会简单一点。胖哥只想让你知道这些东西是可选择的，在什么时候去选择，如何去选择完全要看你自己的功底，不论是做基础程序、做功底还是去做优化，都需要深知它的细节，才能做到心中有数。

1.6　常见的目录与工具包

很多做 Java 开发的同学们，在达到一定程度后，开始“身手不凡”，成为大侠，在了解了底层后，开始自己写东西。这个阶段容易纠结的就是重复制造，在了解了底层后我们需要提升知识面，知道哪些是别人提供的，哪些是需要我们自己写的。

Java 的三方包无穷无尽，无法一一列举这些工具包，但是我们应当知道有许多别人写好的工具包可以直接拿来使用，以及如何来寻找这些工具包，基础知识可以帮助我们快速学习新的工具包，以及掌握它的本质。同时，这里的内容也算是为“源码篇”做一个简单铺垫。

首先，要知道 Java 本身提供的一些源码放在哪里。一般来讲，安装完 JDK 后，在 JDK

的根目录下会有一个叫“src.zip”的压缩包，解压后是一个目录，其中包含了常见的 Java 源码，但是以 sun 开头的文章是找不到的，你可以通过一些反编译手段得到，也可以在 OpenJDK 上找到一些源码，主要目的是看懂意思和知道它的坑。

有工具后，从哪里开始看起？从我们用到的 API 看起，当你需要用到某个 API 的时候，就可以查看是否有相关的 API，简单记录下来。看看是否有更好的方式，要有个大致的笔记。例如，对于 List 的简单使用，小胖哥问几个小问题。

◎ 将 ArrayList 转换为 LinkedList 用什么方法？各自的好坏？

◎ 将集合类、数组做一次浅拷贝用什么方法？用 for 循环还是别的方法？

◎ 对 List、数组做排序用什么方法？

其实在对集合类、数组操作上，有一些 Java 本身提供的工具类（静态工具方法），分别位于 java.util.Collections、java.util.Arrays 类中，它们拥有非常多的对集合类和数组的操作动作，足以满足绝大部分需求。下面通过两个场景给大家一点感性认识。

场景 1：通过常量构造一个 ArrayList 返回。

最直接的写法是：

```
List<String>list = new ArrayList<String>();
list.add("a");
list.add("b");
list.add("c");
```

用工具就可以这样写：

```
List<String> list = Arrays.asList("a" , "b" , "c");
```

或许你认为就是语法糖而已，而 Java 本身要的就是简单，我们希望将更多的精力花在实际创造上，这样的代码可能会在系统中反反复复出现，其实很多时候是枯燥无味的。

场景 2：中文拼音排序。

代码片段 1-6　一个简单的中文拼音排序

```
public class SampleChineseSort {

    @SuppressWarnings("rawtypes")
    private final static Comparator CHINA_COMPARE
              = Collator.getInstance(java.util.Locale.CHINA);

    public static void main(String []args) {
         sortArray();
         sortList();
    }


    private static void sortList() {
         List<String>list = Arrays.asList("张三", "李四", "王五");
         Collections.sort(list , CHINA_COMPARE);
         for(String str : list) {
              System.out.println(str);
         }
    }

    private static void sortArray() {
         String[] arr = {"张三", "李四", "王五"};
         Arrays.sort(arr, CHINA_COMPARE);
         for(String str : arr) {
              System.out.println(str);
```

```
            }
        }
    }
```

关于排序不仅仅如此，有些时候我们还需要用对象排序，对象怎么排序呢？当然是基于对象内部某些自定义的属性来排序了。而不论用什么来做 Java 排序，都需要使用数字来排序，也就是要有大小关系。Java 提供了 Comparable 和 Comparator 两种接口（两种接口是分开用的），Comparable 接口需要让列表中的对象来实现接口中的方法 compareTo(E)，返回正数表示当前对象比传入对象大，当前对象会排序靠后；返回 0 表示等值，返回负数表示当前对象比传入对象小，排序靠前。也就是你告诉 Java 对象之间的大小关系（而谁大谁小是你自己决定的），Java 就会从小到大排列；如果想要反向排序，那么就将返回值“取反”；如果想要按照多个字段排序，在这个方法内部也是可以完成的。

大家可能发现这样的方式不是太灵活，因为一个对象实现接口后，这个方法就固定了。也就是说，它的排序算法已经固定了，如果它的排序算法不是固定的，是可以动态调整的，那么就用 Comparator 接口来扩展，它独立于被排序的对象单独存在，当需要排序的时候，以参数的形式传递，上面例子中的“CHINA_COMPARE”就是一个 Comparator 实例。你也可以自己实现一个自定义对象的排序方式来满足特定对象的要求，如果同样的对象一会想这样排序，一会想那样排序，就只需要使用不同的 Comparator 实例即可。

这是一种基本的封装思路，Java 就是希望让开发者关注更少的事情，它帮你去完成排序。这种思路也衍生到 SortedSet 中的排序，也在 Timer、ScheduleThreadPool 调度任务中使用（确切说是内部包装的，在 PriorityQueue 中使用，5.6 节会详细阐述）。关于这些内容，可以参考胖哥的个人博客：http://blog.csdn.net/xieyuooo/article/details/8611198。

例子举不完，Java 本身提供的工具非常多，而且每个版本都会有新的工具包出现。除此之外，还有许多的二方包、三方包都提供了大量的工具类，这里就不再逐个举例了。Java 是希望将一些复杂的逻辑细节封装在工具中，让业务代码尽量干净、整洁，容易维护。反之，如果业务代码中出现了大量的排序、字符串处理、日期处理、类型转换、数组或集合类循环拷贝与组装、文件处理、网络 IO 等片段，我们就会感觉代码“不干净”，这样的代码会在许多地方出现，这也意味着如果需要修改则要到许多地方去完成，而很多时候我们不知道到哪里去修改。这些代码会让我们的思路不断在技术和业务之间切换，没法专心做好业务细节，更没法深度挖掘业务。技术本身是在业务驱动下才能发挥作用的，而人在业务驱动下去学习技术是能得到最佳实践的。

例如，Apache 就提供了许多有用的三方包给我们使用，很多人都应该熟知 StringUtils 这个工具包吧，类似的还有很多，如 upload、连接池、log4j、字符集处理等都可以算是工具包，这些工具包提供了大量 API，大多数情况下无须自己去实现处理细节，因为它们的正确性和性能是经过很多公司验证的，如果有问题大家都会知道。所以，我们在注重功底的基础上也需要大量的学习，铺开知识面，才能有选择，但“深知内在细节是我们量化选择的条件”。

当然，不是所有的工具包都能完全满足我们的需求，而往往“最烦人”的就是这个工具包满足一部分需求，那个工具包满足一部分需求，而将这些工具包组合起来使用可能比自己写一个还麻烦，翻译为基础代码还会有一大堆的冗余，此时就可以考虑扩展了。

每个场景都会有一些变化，开源的工具仅仅是提供一些通用的处理，同时提供一种 Java 封装思想，许多代码也值得我们去参考。因此当遇到个性化问题的时候，并非别人没提供自己就不写了，个性化包装就是一个优秀程序员需要去承担的责任，所以老 A 程序员除了拥有非凡的功力外，也同时需要深知所涉及领域的业务和个性化，这样才能针对业务做出一个“很爽”的架构体系。

胖哥说话好像在“绕圈”，怎么又绕回功底来了？其实胖哥此处再谈功底，是要说明在这个基础上根据实际的场景应当“因地制宜”才能“事半功倍”，同时要以大量的知识面为基础，充分利用现有的资源。

1.7 面对技术，我们纠结的那些事儿

人生需要面对很多纠结，我们都是在纠结中磨练自我意志的。

纠结容易让人浮躁，容易让人犯错；但是纠结同样会让人成长，纠结是黎明前的黑暗。

学会将纠结化作成长的力量，在逆境中能生存的强者才是真正的“老 A 级程序员”。

1.7.1 为什么我这里好用，哪里不好用

“为什么我这里好用？哪里不好用？”

小鬼，你是不是经常说这句话？如果是，那你“中标”了。

你怎么知道我经常这么说呀？

这句话要么是老鸟说，要么是小鸟说，同一句话，老鸟说的意思和小鸟的得意思完全不一样哦！

有啥不一样呢？

小鸟通常是碰到问题就说这句话，而老鸟通常是解决问题后才说这句话；小鸟说这句话是纠结，继续跟自己过不去；老鸟说这句话是为了搞懂本质和原理，整合事实与理论，证明自己的结论，总结自己的知识。

小鸟还是不懂这是为什么，好可怜。

小鸟啊小鸟，你想想，你还不会飞，还没有走出你的窝，怎么能看清楚这个世界，这个世界上万物生灵的成长？你提出的为什么只是虚幻，没碰过的。老鸟已经在天空中翱翔数年，它们拥有自己的生存与自然界经验，深知自然界的生命循环与弱肉强食的道理，知道什么是好与坏，知道什么是美与丑，知道什么是天敌与猎物。你若想要了解这个世界，“努力锻炼自己，拥有一双可以翱翔的翅膀，去感悟这个世界吧”。

可能从一只小鸟成长起来会经历沉默，然而不在沉默中爆发，就会在沉默中灭亡。当你成长为老 A 的时候，你应该知道的是在解决问题后才会提出相关性问题，而不是在解决问题前，所以真正的一个老 A、一只老鸟，可以说这样的话。

反过来讲，要提出问题，也需要有点水平才能真正切入主题将问题说清楚，有的小鸟喜欢占主动权，问问题的时候想要牵着老鸟的鼻子走，老鸟会问小鸟想知道的问题细节，但小鸟继续说自己想说的，结果是老鸟不知道小鸟要什么，也搞不清楚问题是什么。老鸟逼于无奈而且自己还有事情，最后就装着什么也不知道，小鸟就经常会说老鸟有什么用啊，什么问题都搞不定，当自己发现低级错误的时候，还可能得意地想：“不就是一个低级错误吗，老鸟竟然解决不了。”

无论如何我们离不开的是实际场景，要根据实际的工作背景来解决问题，这样才能达到较好的效果，同时在一些代码 bug 的修复上，可能还会分析实际场景中代码修改的相关影响。在运维层面可能是保留现场后直接重启，再通过 copy 的现场做模拟；在一些长远问题上，可能我们需要慢慢来分析。

这就是胖哥讲的老鸟与小鸟的故事，如果你是小鸟，你懂了吗？

➤➤1.7.2　你的程序不好用，你会不会用，环境有问题

“你的程序不好用？你会不会用？”

能说出这句话也许你是一只老鸟了，但是不一定是一个老 A。

这是一个对自己充满自信的程序员，在面对别人提出质疑时的第一反应，可能作为老鸟，你会说自己不是这样的人，那么你真是一朵“奇葩”，或者你真的升华到了“大师”级别，修身养性的境界非同一般。在众多老鸟之中，我们都有一股“程序员的野性”，这种野性通常来自于自己研究了许多东西，以及在相关领域待的时间很长所赋予自己的强烈自信感。

学海浩瀚无边，任何人（即使是大师）也有犯错的时候，所谓“老虎也有打盹的时候”，尤其是在工期要求很紧的时候，“一流高手”和“二流高手”体现的水平不会有多大的区别，因为他们都是“老鸟级”的人物，代码编写速度和问题解决能力都是非常强的，而更高境

界上的区别，在这种情况下是体现不出来的。

在面对别人提出质疑的时候，第一反应其实是看你的内心世界是否足够强大。别人质疑你的第一句话，有可能是顶住你心窝的话，有可能是将你苦思冥想的方案瞬间彻底否决的话，有可能是将你很长时间的辛苦付出瞬间说得毫无价值的话，有可能是将你的成果瞬间打入无底深渊的话，也许是对你和大家都认可的事情简单敷衍的话。这些话相信比质疑你的代码有 Bug 会更加重一点吧，这个时候你会如何想呢？

或许一颗强大的内心需要去磨练，小胖哥也在不断磨练中成长自我。也许你会觉得很累，因为这是在挑战自己的个性，改变自我是最痛苦的事情。也许这个时候我们可以看看别人是如何修身养性的，别人能做到，我们同样可以做到。

代码有问题，我们就看原因，不论是不是自己的问题都要告诉真正的原因。有问题没关系，我们不怕犯错误，而是怕错了不改，反反复复犯同样的错误，可能还是同样低级的错误。

如果反反复复有人提出同样的问题，而绝大多数情况下都知道不是自己的问题，那么就将它以文档的方式呈现出来，因为代码是我们写的，使用者通常不知道所有的细节。

如果没有任何规约，就意味着可以随便“乱搞”，事实上逻辑的东西暂时是不可能做到这一点的（就算是写代码也得遵循基本的语法规则，用数据库也得遵循它的基本规约而不能乱删除东西，用接口需要遵循接口规约），“乱搞”的结果就是东西越多、问题就越多，做一件事情来解决问题，但是带来了更多的问题。

这个“乱搞”的底线是什么？完全是看系统的设计思路，也就是软件的伸缩性，你需要让用户知道不能去做或不建议去做的事情，我们可能会提供，但是不完全保证它的正确性或安全性，为什么不保证也许某些用户也想要知道。

回过头来，对于自己来讲最重要是冷静面对事情的真相，客观面对我们所面临的事情，加上对业务本身的熟悉程度、技术功底的配合、冷静思考、量化的判断、全局性影响的分析，逐步突破层层技术和业务难关，相信我们可以很“酷”地解决绝大部分的问题。解决问题的手法也会多种多样，而并非等闲之辈。

1.7.3 经验是否能当饭吃

通过对前面几节内容的介绍，胖哥就是想让你知道，也许某些经过试验论证或曾经解决问题得到的结论不是很靠谱，或者说并不是“永远靠谱”的。

前面也提到了，其实在技术的发展过程中，我们都是站在前人的肩膀上做进一步的事情，它们虽然以“平台、插件、思想”的方式提供给我们作为基础，但是通常也都是代码，可以先不说“只要是代码，肯定就会存在 Bug”，但是软件程序在不断更新换代的过程中肯定会存在各种逻辑的复杂性来满足更多的需求，就像一个系统做复杂了的道理一样。

随着实际代码的复杂性增强，就会产生各种各样的运行逻辑，此时如果 API 写得过分简单，那么许多程序员可能就不知道这个 API 下面的许多细节，也没有那么多精力来关注每个细节，因此就可能会导致很多不可预见的问题。在某种场景下我们可能将问题解决了，

但是也许并不是因为看到了问题的本质，而有可能是因为某种巧合走到了一条正确的执行路径上。

除了上面的例子，曾经有不少人在“乱码”上经常出现怪异的问题，小胖哥也曾经为此纠结过很久，很多人在遇到乱码后都会像小鸟一样说：“为什么我这里好用，另外一个地方却不好用呢？”这类问题在网上通常有很多方法，大多要求你将数据库、应用服务器、接口、环境变量、页面等输入/输出的字符串全部统一。其实这种方法只能解决某些局部问题，并未看到问题的根本。

为什么说只能解决某些局部问题呢？

因为在某些场景下，系统会连接不同的数据库、缓存、分布式存储、应用方、中间件等，它们都有可能会提供不同的字符集输入输出，我们也无法控制别人提供的服务必须用什么字符集。换句话说，你或许在某种程度上可以保证自己的应用系统使用什么字符集来避开许多问题，但是当面对复杂的分布式环境时，你不得不去知道字符集的根本所在。只有搞清楚这类问题的本质，才能在遇到类似的问题时去定位分析。

关于乱码更多的介绍：

乱码问题不是本节讨论重点，关于乱码方面的问题在本书后文中第 2.6 节、第 17.2 节中有更多的介绍，同时会在第 4 章的通信里面提到一些相关的基础知识。

再换位思考一下，软件在不断革命的过程中，内在的逻辑在不断变化，也变得更加复杂，它们提供的一些参数可能发生改变，即使是同样参数的 API 的内在实现也有可能与以前大不相同。此时，随着技术的发展一些经验开始变得“不太靠谱”，包括我们曾经记录下来的经验值、默认值，甚至是官方标准，它们都可能在一个小的版本变化中变得不靠谱。

因此，我们需要新的学习，是否会觉得很累呢？我们在不断学习别人提供的东西，在不断记忆别人的东西，前一样东西还没掌握清楚，新的又来了，肯定很累。但这需要方法，也需要功底，当你懂得它的根本与发展的方向和模式，基本就知道它要做什么了，也基本知道会怎么做了，自己也可以实现，只是没必要而已。一些开源软件中常见的参数不会太多，而更多的参数当你知道它的根本后，通过某些命令和文档查阅是非常快速的，我们学习的代价不应该太大。

回过头来总结一下，我们应当“站在前人的肩膀上看问题，不仅仅是要沿用前人写好的东西，更需要看到他们所遇到的那些坑”。在理论指导的基础上，比前人看得更高、更远，社会和技术才可能向前发展和进步，新一代程序员不必都经历上一代程序员所经历的同样的坑，那样只是延续，而没有任何发展。新一代程序员即使要经历坑，也是经历一些更新的坑，去挖掘更新的东西，而不是走老路。

小胖哥说了这么多话有什么用呢？就是想要告诉读者朋友，经验可以用以借鉴和参考，但经验不能用来当饭吃，同时别人告诉你的是他的经验，也同样是一种参考和借鉴，我们需要学会去看问题本质的方法与习惯。也许你不太认同小胖哥的观点，可能你和小胖哥看待问题的角度不一样，胖哥也很希望你有不同的自我见解，但也相信我们会有一些共同的认识。

接下来我们用轻松的方式来论道，谈论老 A 的那些事，老 A 在面对逆境时是如何面

对的。

1.8 老 A 是在逆境中迎难而上者

小胖哥虽然不是一个“传道者”，但是喜欢小小论道，因为在人生的道路上，很多时候你我都会面临许多纠结的事情，而这个时候我们的态度会决定命运，而道就是道理和方法。

◎ 作为“程序员”，每天被要求修改代码是否会烦躁不堪，甚至觉得失去工作的价值？当有人说你的代码有很多 Bug 时，你会像小鸟或老鸟一样反应吗？

◎ 作为“架构师”，成天被程序员说这里（那里）不好用、性能不好、伸缩性差、太重等，而程序员可能并没有理解框架的真正意义，你是否会纠结？你应该如何去面对和解决？

◎ 作为“经理”，要被客户骂，又无处发泄，还要听下面人的抱怨，你有一颗强大的内心世界吗？

有很多事情只可意会不可言传，对于大多数事情都需要你我修炼内心世界。大家可能都看过一些特种兵的电影或故事，老 A 们通常会面对常规兵种没有遇到过的困难和绝境，但他们总能找到绝处逢生的道路，这就是老 A 的本色，老 A 除了要练好基本功外，拥有面对逆境的心态和战胜困难的决心是十分重要的。

我们在解决问题时，不仅要学会面对成功，而且要学会面对失败与困难，这样才会进步，如果有人说他没有失败过，只能说他不愿意去承认自己的过去而已。我们都会在成功、失败、困难的过程中不断交替轮询，对于许多深入问题的看法，也不断在模糊与明了中轮询（就像图 1-10 所示的太极图那样）。在这个过程中只要愿意去坚持和思考，都会有自己对人生的理解和丰富的经验，并且能跟上社会的潮流，经历从量变到质变的过程，以达到境界的提升。

图 1-10 太极图

在逆境中敢于迎难而上的老 A 们，除了拥有信心战胜困难以外，也拥有极强的功底和快速学习的能力。提到学习能力，在不同的博客、网站、论文、通信群里面你能看到不少人提出的学习方法，而且每个人总结出来的学习方法都会有些不一样的地方，即使同样的学习方法，你也未必全部理解它们的意思。胖哥绞尽脑汁写出下文，希望你能看懂真正意义，能得到一些帮助。

每个人在不同的阶段应当有不同的学习方法，不能一概而论。提到学习方法，还是一句老话：多看、多练、多思考、多练习、多总结而对于这句老话，胖哥的理解如下。

☐ 多看

看什么？看别人如何写代码，如何分析和解决问题，用到了什么技术，如何面对工作的压力，如何为人处事，如何协调资源等。

怎么看？周围都是你的同事，同事中肯定有高手。网上有你的朋友，朋友中肯定有佼佼者。你会用到许多开源框架，其中肯定包含了许多经典的代码和思想。

☐ 多练

多练，但是很多人找不到方向，应该练什么？这是问题的关键，也是初学者陷入迷茫

的一个重要原因（很多已经有一定工作经验的人也会陷入这种迷茫）。

有些时候看到别人写的程序，自己什么都想练习，什么都想学习。其实要知道，无论别人写出多么优秀的程序，干出多么大的事情，这并非一朝一夕之事。

需要做的是立足当下，找到自己的进步点，而不要好高骛远。什么是立足当下呢？

立足当下就是找到自己的兴趣，而非别人的兴趣；找到自己的业务挖掘点，而并非别人的业务（并不是说不去关注别人所做的事情，而是一种经验的交流，不要老觉得别人碗里面的饭要香一些）。

兴趣建立在你对所解决的问题的价值是否理解上，兴趣可以给你带来不错的收入，兴趣是你可以处理一些周围人处理不了的问题，兴趣是你擅长的点是否能够得到发挥等，其实兴趣是自己挖掘的。

初学者一定要多练习，即使是练习代码，也是很好的，达到一定程度再思考都可以。即使是水平很高的人，也会多练习，只是练习的代码不一样罢了。练习可以让我们对问题有更多的感性认识，许多问题在练习后自然会逐步变得清晰起来。在练习的基础上，才会进一步有理性认识。

□ 多思考、多练习

经过多练习，很多人会走偏方向，进入一个“技术控”的死角里，因为“对技术的控制”会成为一种乐趣。“技术控”如果“深陷泥潭、难以自拔”，最终可能是被技术控制，会被技术牵着鼻子走。

这个阶段我们需要更多的思考——这种技术有什么缺点？我们用它能解决什么核心问题？相关的技术有什么？它们有什么区别？

当我们深入地去学习一样东西的时候，会自然而然地需要去学习另外一样东西，这就是知识之间的串联，碎片化的学习并非完全碎片化，随着学习不断深入，知识自然会串联在一起。

在练习、思考的基础上还要如何深入？各类网络博客、官方文档、源代码、书籍都是积累的关键要素，当资料很多的时候要学会挑关键点。

什么是关键点？我们关注的点就是关键点。其实关键点就是你最想要知道的一个点，这种关键点对不同层次的人会有不同的划分概念，它可以细到一条代码怎么写，一个二进制位在机器内部如何处理，也可以粗到总体架构是怎么回事，这完全和你的工作相关，同时也与你对工作的理解程度相关。这是一个积累的过程，知识和业务体系越“通”，想要找到关键点就越容易，所以学习、思考、时间是一套不断迭代的“组合拳”。

思维上我们要去放纵扩展、体系化扩展（但不要钻牛角尖），在扩展的过程中会迫使自己去学习相关的许多知识，而且这些知识会刻骨铭心，随着知识不断地串联化，也会逐步形成体系。如果有一天你发自内心地认为“一切源于基础思想和生活方法”，这时你再看高手们的书，或许就是一种知识的梳理过程和经验的交流。

□ 多总结

一个人不论学习多么努力，多么会思考，但如果他的知识没有沉淀，终究会丢掉一些东西。

何时沉淀？沉淀什么？这是我们不断摸索的东西，胖哥也认为没有定论，如果你认为它对你未来的发展是有帮助的，或者说你认为它可以帮助你提高水平或扩展知识面，那么

你就将它沉淀下来。

每天我们有了一个小沉淀，每周就会有点小总结，每月就有一些感悟，每季度就会发现几个月前的自己是那么的“幼稚”，那么自己就真的成长了。经过 1 年、2 年的坚持，你可能会比别人知道得更多，也深入了很多。

也许有人说无论你怎么沉淀都没有用，始终在原地踏步，几年后还是一个样。那么你就要想想：在总结的过程中是否考虑过方法有没有问题？或许自己始终在沉淀那些很简单、很容易的东西，仅仅是为了“记录”而缺乏自己的见解；或许在总结的过程中没有去考虑如何做得比以前更好。当然如何总结是个人的自由，胖哥无法干预，每一种总结都有它的价值所在，即使是“面”上的扩展总结，只要是没有曲解基本的道理，仍可以“引导他人”成长。

什么时候需要去沉淀知识呢？当你解决了问题时，当你学习到问题的本质时，当你看到了以前看不到的东西时，当你觉得值得去总结时，长期坚持总结是厚积薄发的基础，这是亘古不变的道理。从量变到质变取决于在总结的过程中你能否从某个点去突破和深入，也许就差那么一点点儿，当发生质变后在对待问题的态度和方法上会有一个更加广阔的空间，这就是我们通常所说的“茅塞顿开”。

沉淀什么？我们不仅要总结知识点，也要总结方法、手段、心态，以及身边可以深度挖掘的业务。

接下来的几章将会真正介绍 Java 的一些基础知识。