3.5 小结

1. 重点回顾

- 数据结构可以从逻辑结构和物理结构两个角度进行分类。逻辑结构描述了数据元素之间的逻辑关系，而物理结构描述了数据在计算机内存中的存储方式。
- 常见的逻辑结构包括线性、树状和网状等。通常我们根据逻辑结构将数据结构分为线性（数组、链表、栈、队列）和非线性（树、图、堆）两种。哈希表的实现可能同时包含线性数据结构和非线性数据结构。
- 当程序运行时，数据被存储在计算机内存中。每个内存空间都拥有对应的内存地址，程序通过这些内存地址访问数据。
- 物理结构主要分为连续空间存储（数组）和分散空间存储（链表）。所有数据结构都是由数组、链表或两者的组合实现的。
- 计算机中的基本数据类型包括整数 `byte` 、 `short` 、 `int` 、 `long` ，浮点数 `float` 、 `double` ，字符 `char` 和布尔 `bool` 。它们的取值范围取决于占用空间大小和表示方式。
- 原码、反码和补码是在计算机中编码数字的三种方法，它们之间可以相互转换。整数的原码的最高位是符号位，其余位是数字的值。
- 整数在计算机中是以补码的形式存储的。在补码表示下，计算机可以对正数和负数的加法一视同仁，不需要为减法操作单独设计特殊的硬件电路，并且不存在正负零歧义的问题。
- 浮点数的编码由 1 位符号位、8 位指数位和 23 位分数位构成。由于存在指数位，因此浮点数的取值范围远大于整数，代价是牺牲了精度。
- ASCII 码是最早出现的英文字符集，长度为 1 字节，共收录 127 个字符。GBK 字符集是常用的中文字符集，共收录两万多个汉字。Unicode 致力于提供一个完整的字符集标准，收录世界上各种语言的字符，从而解决由于字符编码方法不一致而导致的乱码问题。
- UTF-8 是最受欢迎的 Unicode 编码方法，通用性非常好。它是一种变长的编码方法，具有很好的扩展性，有效提升了存储空间的使用效率。UTF-16 和 UTF-32 是等长的编码方法。在编码中文时，UTF-16 占用的空间比 UTF-8 更小。Java 和 C# 等编程语言默认使用 UTF-16 编码。

2. Q & A

Q：为什么哈希表同时包含线性数据结构和非线性数据结构？

哈希表底层是数组，而为了解决哈希冲突，我们可能会使用“链式地址”（后续“哈希冲突”章节会讲）：数组中每个桶指向一个链表，当链表长度超过一定阈值时，又可能被转化为树（通常为红黑树）。

从存储的角度来看，哈希表的底层是数组，其中每一个桶槽位可能包含一个值，也可能包含一个链表或一棵树。因此，哈希表可能同时包含线性数据结构（数组、链表）和非线性数据结构（树）。

Q：`char` 类型的长度是 1 字节吗？

`char` 类型的长度由编程语言采用的编码方法决定。例如，Java、JavaScript、TypeScript、C# 都采用 UTF-16 编码（保存 Unicode 码点），因此 `char` 类型的长度为 2 字节。

Q：基于数组实现的数据结构也称“静态数据结构” 是否有歧义？栈也可以进行出栈和入栈等操作，这些操作都是“动态”的。

栈确实可以实现动态的数据操作，但数据结构仍然是“静态”（长度不可变）的。尽管基于数组的数据结构可以动态地添加或删除元素，但它们的容量是固定的。如果数据量超出了预分配的大小，就需要创建一个新的更大的数组，并将旧数组的内容复制到新数组中。

Q：在构建栈（队列）的时候，未指定它的大小，为什么它们是“静态数据结构”呢？

在高级编程语言中，我们无须人工指定栈（队列）的初始容量，这个工作由类内部自动完成。例如，Java 的 `ArrayList` 的初始容量通常为 10。另外，扩容操作也是自动实现的。详见后续的“列表”章节。

Q：原码转补码的方法是“先取反后加 1”，那么补码转原码应该是逆运算“先减 1 后取反”，而补码转原码也一样可以通过“先取反后加 1”得到，这是为什么呢？

A：这是因为原码和补码的相互转换实际上是计算“补数”的过程。我们先给出补数的定义：假设 $a + b = c$ ，那么我们称 a 是 b 到 c 的补数，反之也称 b 是 a 到 c 的补数。

给定一个 $n = 4$ 位长度的二进制数 0010 ，如果将这个数字看作原码（不考虑符号位），那么它的补码需通过“先取反后加 1”得到：

$$0010 \to 1101 \to 1110$$

我们会发现，原码和补码的和是 $0010 + 1110 = 10000$ ，也就是说，补码 1110 是原码 0010 到 10000 的“补数”。**这意味着上述“先取反后加 1”实际上是计算到 10000 的补数的过程**。

那么，补码 1110 到 10000 的“补数”是多少呢？我们依然可以用“先取反后加 1”得到它：

$$1110 \to 0001 \to 0010$$

换句话说，原码和补码互为对方到 10000 的“补数”，因此“原码转补码”和“补码转原码”可以用相同的操作（先取反后加 1 ）实现。

当然，我们也可以用逆运算来求补码 1110 的原码，即“先减 1 后取反”：

$$1110 \to 1101 \to 0010$$

总结来看，“先取反后加 1”和“先减 1 后取反”这两种运算都是在计算到 10000 的补数，它们是等价的。

本质上看，“取反”操作实际上是求到 1111 的补数（因为恒有 `原码 + 反码 = 1111` ）；而在反码基础上再加 1 得到的补码，就是到 10000 的补数。

上述 $n = 4$ 为例，其可推广至任意位数的二进制数。

上一页 3.4 字符编码 *

下一页 第 4 章 数组与链表

欢迎在评论区留下你的见解、问题或建议