# 第一部分： 预备知识

这部分包括4章介绍性的内容。第一章介绍了数据结构和算法的概念并解释了使用它们的原因。这一部分也介绍了一些软件工程方面的主题，并将这些主题和思想贯穿运用到了全书的其他部分。第二章讲到了若干关于指针的主题。指针在本书中大量出现，因此这一章可作为对于指针的回顾。第三章介绍了一种在很多数据结构和算法中都经常见到的技术——递归。第四章介绍了如何去评估和分析算法。这一章中的内容可用来分析本书中涉及的算法。

# 介绍

12岁那年我和我的哥哥一起学习弹钢琴。每周我们都要到老师家去上课，当我们中的一个在上课时另一个人需要在客厅等着。幸运的是，老师每次都在咖啡桌上安排一些游戏帮我们打发等待的时间。我记得其中一个游戏里有一小片插上钉子的木板。当时我并不知道，但这个游戏的确可以用来对数据结构和算法做初步的介绍。

游戏规则是这样的：除了一个蓝色的夹子外，其他夹子都是白色的。刚开始游戏时拿掉其中一个白色夹子，这样就有了一个空位，然后每次跳过一些夹子就拿掉一个。直到最后只剩下一个夹子，或者剩下的夹子都无法再跳过（译者注：类似约瑟夫环问题）。游戏的目的是通过跳过夹子保证蓝色的夹子是最终剩下的那个。根据游戏的规则，成功达成目标的玩家被称为“天才”，这是完成此游戏所能获得的最高评价，而其他一些评价级别则根据游戏的不同结果来判定。对于我来说，只要我在玩游戏时老师家的小猫Clara能够安安静静的呆着，而不是试图从沙发的各个角落用爪子突袭我，从而让我能顺利的完成一盘游戏我就很满足了。我想，对这个结果感到满足表明我拥有着一种“常识”。

记得在玩这个游戏时我在思考一定有一种可确定的方法可以每次都让蓝色的夹子最后剩下。我在寻找的是一种算法。算法是定义良好的用来解决问题的步骤。直到若干年之后我真的实现了解决夹子问题的算法。在大学的人工智能课上我决定用LISP来解决这个问题。为了解决这个问题，我用多种数据结构来表示游戏中的信息。数据结构是组织信息的一种概念，它们和算法关系很紧密，因为很多算法为了获得较高的效率需要选择恰当的数据结构。

通常人们以松散的方式处理信息，比如板子上的钉子，笔记本中的便条，或者文件夹中的图片。然而用计算机处理信息需要将数据规则的组织起来。此外，周密地计划要做的事情对我们是绝对有帮助的。数据结构和算法帮助我们完成这些任务。简单的说，它们能帮助我们开发出这样的程序，用一个词来表达——优雅。作为软件工程师，很重要的一点是记住我们不仅仅只是对编程语言和开发工具熟悉，开发出优秀的软件是一项技艺。想要成为大师很重要的一点是需要对数据结构和算法有很好的理解。

## 数据结构简介

数据有各种形式和大小，但通常它们可以以相同的方式来组织。比如，设想有一张清单，上面列出了需要做的事情，配方中的成分列表，某门课程的阅读清单。尽管每种都包含不同类型的数据但它们包含的数据都以一种相似的方式进行组织：列表。列表是一种简单的数据结构。当然，同样还有许多其他类似的方法来组织数据。在计算机科学领域，一些最常用来组织数据的方式有：链表，堆栈，队列，集合，哈希表，树，堆，优先级队列，图。本书将讨论所有上述的数据结构。使用数据结构的三个原因是：效率，抽象，重用性。

*效率*

数据结构组织数据的方式使得算法变得更加高效。例如，考虑一下我们该如何组织数据以对其进行检索。最简单的方式是将数据存放到数组中，然后遍历其中每一个元素直到找到我们需要的元素为止。然而这种方式确是低效的，因为在很多情况下我们需要遍历数组中的每一个元素才行。通过使用另一种数据结构比如哈希表（见第8章）或者二叉树（见第9章）我们可以显著的提高检索的速度。

*抽象*

数据结构使我们以一种更加容易理解的方式去看待数据。也就是说它们为解决问题提供了一层抽象概念。比如，要把数据存入一个堆栈（见第6章），我们可以把精力集中在我们可以对堆栈做什么操作上，例如压栈和出栈，而不是实现每种操作的具体细节上。换句话说，数据结构使我们以不那么“程序化”的方式去看待程序。

*重用性*

数据结构是可重用的，因为它们应该是模块化且上下文无关的。它们是模块化的因为每种数据结构都有各自指定的接口，也就是说我们只能通过定义的接口来访问操作数据。数据结构是上下文无关的，因为它们能在任意环境或上下文中应用于任意一种类型的数据之上。在C语言里我们通过使用指向void类型的指针的方式来管理任意类型的数据，而不是在数据结构内部维护一份数据的私有拷贝。

当面对数据结构时，我们通常会想到特定的行为或者操作。例如，给定一个链表，我们会自然的想到插入，移除，遍历和计算元素个数等操作。数据结构加上这些基本操作就称为抽象数据类型（ADT）。一个抽象数据类型的操作就组成它的公共接口。抽象数据类型的公共接口精确地定义了我们可以对它做什么。建立并遵守抽象数据类型的接口是绝对必要的，因为这会使我们能更好的管理程序的数据使得程序变得更容易理解也更容易维护。

## 1.2 算法简介

算法是定义良好的用来解决问题的步骤。在计算机科学领域算法是必不可少的，因为它们正是计算机完成系统操作所需要的具体步骤。好的算法就如同好的工具一样，以合理的付出完成相应的工作。使用不当的或定义不清的算法就像用台锯去切割一张纸，或者用剪刀去切夹板一样：尽管工作也许能完成但你不得不考虑完成工作的效率。和数据结构一样，使用算法也有3个原因：效率，抽象，重用性。

*效率*

由于特定类型的问题经常在计算机领域出现，随着时间的推移人们已经找到了高效的方法来解决这类问题。比如，试想一下要对一本书中的索引号排序。因为排序是一项常见的任务因此对于有许多高效的算法可以完成排序任务你应该不会感到太惊讶。我们将在第12章探究一些排序算法。

*抽象*

在解决问题时，算法能够提供一定程度的抽象，因为很多看似复杂的问题都可以用已存在的算法来简化。一旦我们能用简化的眼光去看待复杂的问题，那么更为复杂的问题都可以被我们看作一个更为简单问题的抽象。例如，试想一下如何找到两个网关之间的最短路由。一旦我们意识到这个问题不过就是更具一般性的单对最短路径问题（见第16章）的变种时，我们就能够以这种泛化的方式来解决问题。

*重用性*

算法在很多不同场景下能够得到重用。因为很多著名的算法解决的问题都是由复杂的问题抽象而来的，这也是因为很多复杂的问题都能够简化为简单的问题。一些能够有效解决这类特定问题的方法使我们有可能解决更多其它的问题。

### 1.2.1 算法设计的一般方法

从广义上讲，很多算法解决问题的思路是相同的。因此，为了方便通常按照算法采用的方法和思路来给它们分类。这样给算法分类的一个原因是如果我们理解了它采用的一般思路我们常常就能够对该算法获得一些深入的了解。在解决一些没有现成算法求解，但与现有问题类似的问题时，我们从中可以得到一些启发和灵感。当然，有些算法有悖于分类原则，而另一些则是多种方法相结合的产物。这一节我们将介绍一些常用的方法。

#### 1.2.1.1 随机法

随机法依赖于随机数的统计特性。一个应用随机法的例子是快速排序（见第12章）。快速排序按如下方式工作：设想要对一堆作废的支票排序，我们首先将无序的一整堆分成两部分。其中一堆里我们使所有的支票号码都小于等于我们所设定的一个中间值，另一堆里我们保证所有的支票号码都大于这个中间值。一旦我们有了这样的两堆支票后，我们再以同样的方式对这两堆支票重复刚才的划分过程，直到每一堆里都只有一张支票为止。这个时候所有的支票就都排好序了。

为了获得较高的性能，快速排序依赖于每一次我们要如何去划分支票，我们需要让划分出的两堆支票数量几乎相同。为了实现这一步，理想的方法是我们在划分支票之前首先找到支票号码的中间值。可是为了确定这个中间值需要遍历所有的支票因此我们并不打算这么做。作为替代的方法，我们随机选择一个支票号码作为划分的依据。快速排序的平均性能很不错，因为随机数的正态分布使得划分的结果是相对平衡的。

#### 1.2.1.2 分治法

分治法包含3个步骤：分解，求解，合并。在分解阶段，我们将数据分解为更小，更容易管理的部分。在求解阶段，我们对每个分解出的部分进行处理。在合并阶段，我们将每部分处理的结果进行合并。一个分治法的例子是归并排序（见第12章）。

归并排序按照如下方式工作。如前所述，同样假设我们要排序一堆作废的支票。我们首先将无序的一整堆分成两半，下一步我们分别将两堆支票再各自分成两半，一直持续这个步骤直到每一堆中都只有一张支票为止。一旦所有堆中都只有一张支票时，我们将其两两合并且保证每一个合并后的新堆都是有序的。一直做两两合并直到我们重新得到一整个堆，此时所有的支票就都已经排好序了。

在所有的分治算法中都有相同的三个步骤。归并排序能以以下方式来描述，首先，在分解阶段将数据划分为两份。接下来按照递归的方式对分解出的两部分应用归并排序。最后，在合并阶段将两部分合并成一个排好序的集合。

#### 1.2.1.3 动态规划

动态规划同分治法类似，都是将较大的问题分解为子问题最后再将结果合并。然而，它们处理问题的方式与子问题之间的关系有关。在分治法中每一个子问题都是独立的。为此，我们以递归（见第3章）的方式解决每一个子问题，然后将结果与其他子问题的结果合并。在动态规划中，子问题之间并不是独立的。换句话说，子问题之间可能有关联。这类问题采用动态规划法比分治法更合适。因为若用分治法来解决这类问题会多做很多不必要的工作，有些子问题会被重复计算多次。尽管动态规划法是一种重要的思想且很多算法都利用了这种思想，但本书介绍的算法中都没有使用到它。

#### 1.2.1.4 贪心法

贪心法在求解问题时总能够做出在当前来看的最佳选择。换句话说，不是从整体最优上去考虑，而仅仅是在某种意义上的局部最优解。不幸的是当前的最优解长远来看却未必是最优的。因此，贪心法并不会一直产生最优结果。然而在某些方面来说，贪心法确是最佳选择。一个采用贪心法的例子是霍夫曼编码（见第14章），这是一个数据压缩算法。

霍夫曼编码中最重要的部分是构建一颗霍夫曼树（译者注：又称最优二叉树）。为了构建出一颗霍夫曼树，我们从它的叶子节点自底向上处理。我们将每个要处理的符号和它们在数据中出现的次数（频率）一起作为二叉树根节点保存。接下来，我们选择两颗拥有最小频率值的根节点作为左右子树，构造出一颗新的二叉树，且保证新的二叉树根节点的频率值为左右子树节点的频率值之和。然后我们重复这个步骤，直到我们得到唯一的一颗树，这就是最终的霍夫曼树。霍夫曼树的根节点包含了数据中符号的总数，叶子节点包含了原始的符号以及它们出现的频率。霍夫曼编码是一种贪心算法，因为每次它都会挑选出当前最合适的两棵树来合并。

#### 1.2.1.5 近似法

近似法并不计算出最优解，相反，它只计算出“足够好”的解。通常我们利用近似法去解决那些计算成本很高又因为其本身十分有价值而不愿放弃的问题。推销员问题（见第16章）是一个通常会用近似法去解决的问题。

设想一位推销员需要设计一条去往好几个城市工作的路线。推销员问题的目的是找到最短的可能路径，以便推销员能够在回到出发点前恰好每座城市都只去过一次。由于推销员问题可能存在一种最优的策略，但计算的代价很高，我们可以采用启发式的方法得到一个近似解。当最优策略行不通时，启发式是一种比最优策略稍逊一些的我们能够接受的策略。

推销员问题可以用图表的方式来描绘。我们把推销员必须前往的城市在格子中用点标记出来。然后我们按照如下启发式的方法来寻找这些点之间的最短路径。从推销员出发的位置开始只有一个点，将这个点涂黑。所有其他的点在加入路线图之前都是白色的，当它们加入时也同样涂黑。接下来，对于每一个还没有加入到路线图中的点v，计算最后一个加入到路线图中的点u和v之间的距离。通过这种方法选择出离u最近的那个点，将其涂黑并加入路线图中。重复这个过程直到所有的点都已经涂黑。最后再次将出发点加入路线图使其闭合，这样就完成了整个路线图。

## 小酌软件工程

本章开头曾经提到过，对数据结构和算法的理解在开发软件时是非常重要的。与其同等重要的是在我们的开发工作中应用一些软件工程中的良好准则。软件工程是个很广的主题，但从一些思想和概念中我们可以得到很多收获。这里就介绍一些相关的主题并将这些思想运用到全书中。

*模块化*

在软件设计中为了达到模块化的目的，一种方法是将精力集中在黑盒上。在软件开发中黑盒代表一个模块，它的内部实现并不希望被使用这个模块的用户看到。用户只能通过模块设计者预定义好的公共接口去和这个模块交互。这就是说设计者仅仅只公开用户需要使用这个模块的必要信息，而将其他所有实现细节都隐藏起来。其结果就是用户并不关心模块的实现细节，也避免了（至少理论上是这样，随编程语言而定）与模块的内部实现相耦合。这些就是数据隐藏和封装的基本思想。软件工程中的良好准则在面向对象语言中更是被强制施行。尽管非面向对象编程语言并没有在语言本身强制施行这些观点和准则，我们自己依然可以做到。本书中的例子是设计抽象数据类型（ADT）。基本上每种数据类型都是一个结构体。我们对结构体所能做的一切操作都被精确定义，并在头文件中以公共接口的形式公开。

*可读性*

通过一些方法我们能使程序变得更加可读。比如编写有意义的注释，使用贴切的标识符，编写自注释的代码，这都是一些范例。关于如何编写出好的注释，各种不同的观点之间相差很大。但一个好的基本准则是加上注释以便于其他的开发人员能够很容易的通过阅读这段注释来理解程序逻辑。另一方面，如果代码是自注释的则不需要添加注释，因为代码读起来几乎和注释一样清楚明了。本书中的自注释代码例子是在头文件中对给出的数据结构和算法的公共接口定义。

*简洁性*

不幸的是，在人类社会中我们常常倾向于将“复杂”和“精巧”联系在一起。事实上，精巧的解决方案通常都是最简单的。此外，最简单的解决方案常常也是最难找到的。本书中的大部分算法都能够展示出简洁的力量。尽管人们对一些算法做了大量的研究工作后才证明了其正确性，但它们最终呈现出的形态都是对问题本质而言简洁而清晰的解决方案。

*一致性*

在软件开发中，一件非常值得去做的事情是建立编码规范并一直遵守这个约定。当然，约定必须容易识别。毕竟，如果其他人无法确定什么是约定，则约定也就不成约定了。

这种约定可以体现在许多层面上。例如，它们可能只是代码形式上的修饰，或者它们可能和如何解决特定类型的问题有关联。无论是什么情况，一个好的约定的优秀之处在于一旦我们在一处看到，我们就可以在别处识别出来并理解它的作用。因此，一致性也促进了代码的可读性和简洁性。本书中的两个形式约定的例子是注释的写法以及和数据结构相关联的操作的命名方式。两个概念上的约定例子是在数据结构中管理数据的方式以及把静态函数当作私有函数使用，也就是，它们不属于公共接口部分。

## 如何使用本书

本书根据需要可当作教材或参考书使用。全书组织为3个部分。第1部分包括介绍性内容，也包含了关于指针操作、递归以及算法分析的章节。这些主题对阅读本书其余章节起帮助性的作用。第2部分引入了基本的数据结构，包括链表、堆栈、队列、集合、哈希表、树、堆、优先级队列以及图。第3部分引入了解决问题的常用算法，包括排序、查找、数值分析、数据压缩、数据加密、图论以及几何计算。

第2和第3部分中的每一章内容结构都保持统一，以方便读者阅读和参考。每一章以简介开始，跟着是一系列与主题相关的内容和一系列真实的应用。讲解每一种数据结构和算法时都以介绍开头，跟着是接口的定义，然后是具体的实现以及分析。对于许多数据结构和算法也都给出了它们在真实应用中的例子。每一章都以一系列的问题和回答作为结束，还提供有相关主题的提要列表以便读者进行进一步的探索。

每一种数据结构和算法的引入都会先介绍其基本概念，然后再逐步深入到代码实现细节。因此，读者能方便的根据需要找到自己感兴趣的部分。介绍性的内容涵盖了数据结构和算法是如何工作的。接口定义能够让读者快速了解如何在程序中使用该数据结构或算法。具体实现和分析对接口是如何实现的以及实现的性能给出了更细致的讲解。问题和回答以及相关主题的提要能够帮助那些把本书作为教材的读者获得更深入的理解。每一章开头的内容帮助读者浏览本章的主题以及它们在现实应用中的作用。