第 1 章

解决问题的策略

本书的主题是怎样解决问题，但“解决问题”的确切定义是什么呢？当人们在日常谈话中提到这个术语时，所表达的意思往往与我们这里所讨论的含义截然不同。如果一辆1997年生产的本田思域轿车的排气管直冒蓝烟，怠速时震颤明显，并且油耗明显上升，说明它出现了问题。为了解决这个问题，必须具备相关的汽车知识和诊断故障的能力，并配备相应的替换部件，另外还需要一些常用的维修工具。如果把这个问题告诉朋友，可能会得到这样的建议：“嗨！你应该卖掉这辆旧本田车，然后换辆新的，问题就解决了”。但是，这位朋友的建议并不是这个问题的真正解决方案，它只是逃避这个问题的一种方法而已。

我们所说的“问题”其实包含了约束条件。约束条件就是与问题本身或者它的解决方法相关的、不能被违反的规则。以这台出了故障的思域轿车为例，其中一个约束条件就是需要修理这辆汽车，而不是购买一辆新车。其他的约束条件可能还包括修理成本、修理时限或者在修理的时候不能购买新的修理工具。

在解决程序中的问题时，也存在约束条件。常见的约束条件包括编程语言、平台（它在 PC、iPhone 还是其他平台上运行）、性能（有些游戏程序要求图形每秒至少刷新 30 次，而商业应用程序可能要求对用户输入的响应时间设置上限）或内存需求。有时候，约束条件还涉及到可供参考的其他代码，例如程序中不能包含有开源代码，或者正好相反，它只能使用开源代码。

因此，作为程序员，我们可以把“解决问题”定义为编写一个原创程序，使它执行一组特定的任务，并满足所有预先声明的约束条件。

程序员新手常常能够非常热切地完成这个定义的第一部分，也就是编写一个程序，执行一个特定的任务。但是，他们常常受挫于这个定义的第二部分，也就是无法满足预先声明的约束条件。我把类似这样的程序（即看上去能够产生正确的结果，但是违背了一个或多个预先声明的规则）称为小林丸号（Kobayashi Maru）。如果读者对这个名称感到陌生，说明对作为极克文化试金石之一的电影《星际迷航 II：可汗怒吼》还不够熟悉。这部电影里面有一段情节，描述了星际学院中充满热情的学员们所经受的一场训练：学员们被放在一艘模拟的星舰桥上，以船长的身份接受一个不可能完成的任务。无辜的人们在一条受伤的船（小林丸号）上等待死亡。为了靠近他们，必须与克林贡战舰一战。这场战斗只可能导致船长的星舰被摧毁。这场训练是为了测试军校学生面临战火时的勇气。没有获胜的方法，所有的选择都会导致悲剧性的结果。当电影临近结束时，我们发现柯克船长更改了模拟训练装置，使它实际上可以获胜。柯克非常聪明，但他并没有解决小林丸号的困境，只是避开了它而已。

所幸的是，程序员所面临的问题通常是可以解决的，但许多程序员仍然采取了柯克船长的方法。在有些情况下，他们是不小心才这样的。（“噢！天哪，我的解决方案只有在数据项的数量不大于 100 的时候才行得通，但它必须能够处理数量不受限制的数据集，看来我必须重新加以考虑了。”）但在有些情况下，他们是有意去掉约束条件的，这是为了能够赶上老板或导师所施加的最后期限。还有一些情况，程序员只是不知道怎样满足所有的约束条件。在我所见到过的最坏的例子中，负责编程的学生花钱请人编写程序。不管出于什么动机，我们必须要想方设法避免小林丸号。

1.1 经典难题

当读者深入学习本书的时候，将会注意到尽管源代码的特定细节可能因不同的问题而异，但有些模式会一直在我们所采取的方法中出现。这是非常重要的，因为它使我们最终

能够充满自信地解决任何问题，而不管我们对于当前的问题领域是否拥有丰富的经验。专家级的问题解决者能够迅速发现类比关系，认识到一个已解决的问题和一个未解决的问题之间可供利用的相似之处。如果我们发现问题 A 的一个特性与已经解决的问题 B 的一个特性具有相似之处，就为解决问题 A 奠定了良好的基础。

在本节中，我们将讨论编程世界之外的一些经典问题，把其中的经验和教训应用于我们的编程问题。

1.1.1 狐狸、鹅和玉米

我们将要讨论的第一个经典问题是一位需要过河的农夫所面临的难题。读者以前可能已经遇到过类似性质的问题。

怎样过河

一位农夫带着一只狐狸、一只鹅和一袋玉米过河。农夫有一条划艇，只能容纳他自己加上其中一件物品。遗憾的是，狐狸和鹅都饥肠辘辘。狐狸和鹅不能单独待在一起，因为狐狸会吃了鹅。同样，也不能单独把鹅和那袋玉米放在一起，因为鹅会吃了玉米。农夫怎样才能把所有东西都送过河呢？

图 1.1 展示了这个问题的场景。如果读者以前从来没有遇到过这样的问题，可以花几分钟的时间认真研究一下怎样解决这个问题。如果以前曾经遇到过这样的问题，也可以回忆一下它的解决方案，看看自己有没有办法独立解决。

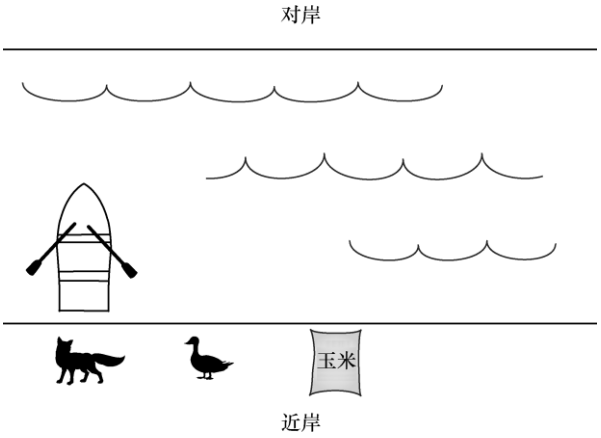

图 1.1 狐狸、鹅和一袋玉米。船一次只能载一样物品。狐狸不能单独和鹅待在同一岸边，鹅不能单独和玉米待在同一岸边

在没有任何提示的情况下，很少有人能够解出这个难题。作者也自认没有这个能耐。下面是人们解决这个问题时的通常思路。由于农夫一次只能携带一样物品，因此他需要往返多次才能把所有物品送到对岸。在第一趟过河的时候，如果农夫带走狐狸，鹅就会单独和那袋玉米待在一起，肯定会吃了玉米。类似地，如果农夫选择带走那袋玉米，狐狸就会和鹅单独呆在一起，鹅也难逃被吃的厄运。因此，农夫第一趟必须带鹅过河，从而出现如图 1.2 所示的场景。

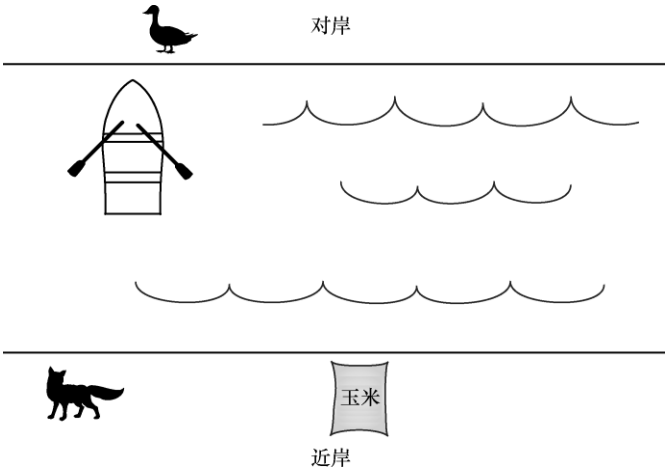

图 1.2 解决狐狸、鹅和玉米问题必然采取的第一步。但是，从这一步开始，所有后续的步骤看上去都将以失败告终

到目前为止，一切正常。但在第二趟过河时，农夫必须带走狐狸或玉米。但是，不管这次农夫带走什么，当农夫从对岸返回取最后一件物品时，第二趟所带去的物品必须在对岸与鹅单独待在一起。这意味着，要么是狐狸和鹅独处，要么鹅和玉米独处。由于这两种情况都是无法接受的，因此问题看上去似乎是无法解决的。

如果读者以前曾经看到过这个问题，很可能还记得解决方案的关键要素。如前所述，农夫在第一趟时必须带走鹅。在第二趟时，我们假设农夫带走了狐狸。但是，他并不是让狐狸和鹅一起待在对岸，而是在返回时把鹅带回近岸。然后，农夫把玉米带到对岸，将狐狸和玉米放在一起后返回。在第四趟过河时，农夫最后把鹅带走。图 1.3 展示了完整的解决方案。

这个问题很难，因此大多数人从来不会想到把物品再从对岸带回近岸。有些人甚至觉

得这个问题是不公平的，表示：“你并没有说我可以把物品带回来！”确实如此，但是在描述问题的时候，题目也没有说禁止从对岸把物品带回来。

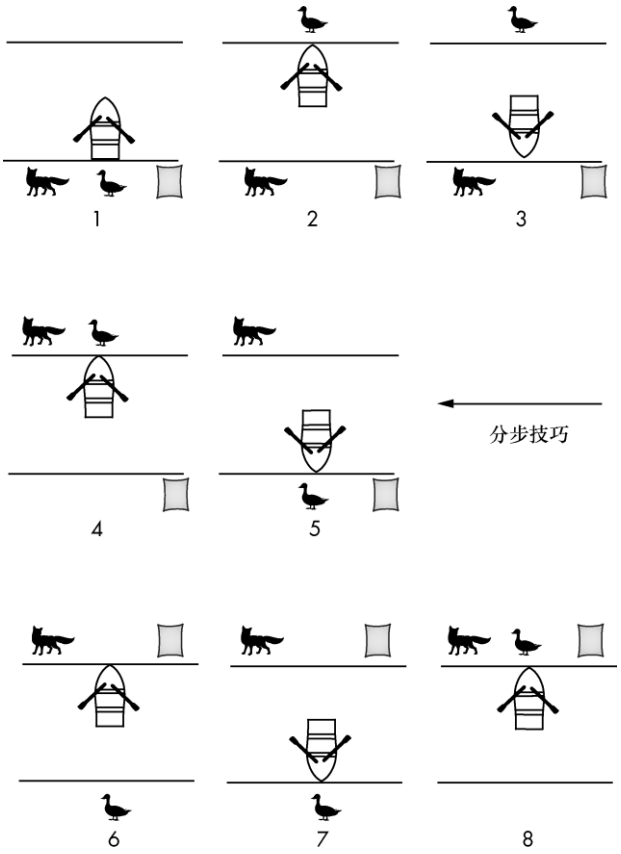

图 1.3 狐狸、鹅和玉米难题的一步步解决方案

可以想象一下，如果事先明确说明可以把物品从对岸带回来，这个问题就会相当容易解决：

农夫有一条划艇，可以来回搬运物品，但小艇每次只能容纳农夫自身再加上他的三件物品之一。

有了这个说明之后，很多人都能解决这个问题了。这说明了解决问题的一个重要原则：如果没有意识到所有可以采取的动作，很可能无法解决问题。我们可以把这些动作称为操作。通过列举所有可能的操作，可以解决许多问题。我们只要测试这些操作的每种组合，直到发现可行的方案。概括地说，就是用更加形式化的术语重新陈述一个问题，常常可以

发现此前被我们所忽略的解决方案。

我们先把这个问题的解决方案抛之脑后，然后用更形式化的方式陈述这个特定的难题。首先，我们列出了约束条件。这个问题的关键约束条件如下。

1．农夫在船中一次只能放一件物品。

2．狐狸和鹅不能单独放在同一岸边。

3．鹅和玉米不能单独放在同一岸边。

这个问题是说明约束条件重要性的一个很好的例子。如果我们去除所有的约束条件，这个问题就毫无难度。如果我们去除第一个约束条件，可以简单地一次携带全部 3 件物品过河。即使一次只能在船上放两件物品，也可以让狐狸和玉米先过河，然后再带鹅过河。如果我们去掉第二个约束条件（但保留另两个约束条件），就必须小心谨慎了。首先要带鹅过河，然后带狐狸过河，最后带玉米过河。因此，如果我们忘了或忽略了任何一个约束条件，就相当于遇到了小林丸号这样的情况。

接着，我们列出所有的操作。陈述这个问题的操作可以采用多种不同的方式。我们可以创建一个特定的列表，列出所有可以采取的操作。

1．操作：把狐狸带到河的对岸。

2．操作：把鹅带到河的对岸。

3．操作：把玉米带到河的对岸。

但是必须谨记，以形式化的方式重新陈述问题的目标是为了能够更好地发现解决方案。除非我们已经解决了问题并发现了“被隐藏的”可能操作（也就是把鹅带回到近岸），否则我们是无法通过上面所列出的这些操作方式发现这个操作的。因此，我们应该设法列出更基本（或参数化）的操作。

1．操作：把船从岸的一边划到另一边。

2．操作：如果船为空，从岸上装载一件物品。

3．操作：如果船不为空，把物品卸到岸上。

通过用最基本的术语来考虑问题，上面的操作列表可以让我们轻松地解决这个问题，就不会把“把鹅从对岸带回近岸”看成是什么奇思妙想了。如果我们枚举所有可能的移动

序列，当每个序列违反了其中一个约束条件或重复了此前已经看到过的一个配置时就终止该序列，最终能够得到图 1.3 所描述的序列并解决这个问题。通过形式化地重新陈述这个问题的约束条件和操作，就成功地避开了它内在的困难性。

经验和教训

我们可以从狐狸、鹅和玉米的问题中学到什么呢？

用更形式化的方式重新陈述问题是一种非常出色的技巧，可以让我们拥有对问题更好的洞察力。许多程序员设法与其他程序员一起讨论问题，并不仅仅因为对方可能已经有了答案，而是因为清晰地陈述问题常常会激发有用的新思路。重新陈述问题就相当于与其他程序员讨论问题，只不过现在是一人分饰两角。

更深远的意义在于，认识到思考问题很可能与思考解决方案具有相同的工作效率，甚至更胜一筹。在许多情况下，通往解决方案的正确道路本身就是解决方案。

1.1.2 瓷砖滑块问题

瓷砖滑块问题存在许多不同的规格，正如我们稍后所看到的那样，它提供了一种特定的解决机制。下面是对 3×3 版本的瓷砖滑块问题的描述。

滑动 8 块

一个 3×3 的网格中放了 8 块瓷砖（编号从 1～8），剩下一格为空。一开始，网格的配置很杂乱。瓷砖可以滑动到邻近的空格中，使它的原先位置为空。这个问题的目标是在网格中滑动瓷砖，使它们从左上角开始在网格中有序地排列。

图 1.4 展示了这个问题的目标。如果读者之前从来没有遇到过这样的问题，可以花些时间考虑怎么解决。网络上可以找到大量的滑动瓷砖问题模拟程序，但最好还是用扑克牌或索引卡在桌上试验。图 1.5 显示了推荐的初始配置。

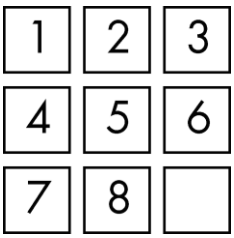

图 1.4　8 块瓷砖版本的瓷砖滑块问题的目标配置，空格表示邻近的瓷砖可以移动到的空间

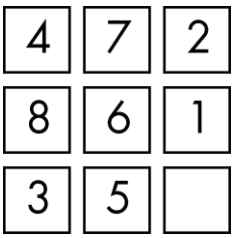

图 1.5　瓷砖滑块问题的一个特定初始配置

这个问题与前面所讨论的农夫与狐狸、鹅和玉米的问题截然不同。过河问题的难度在于可能会忽略其中一种可行的操作。在这个问题中，并不会发生这样的情况。对于任何特定的配置，空格周围最多有 4 块瓷砖，它们都可以移动到这个空格中。这样的描述已经枚举了所有可能的操作。

这个问题的难度在于解决方案所需的漫长的操作环节。一系列的滑动操作可能把某些瓷砖滑动到它们的目标位置，同时又把其他瓷砖移出正确位置，或者可能把某些瓷砖滑动到靠近目标位置，同时又把其他瓷砖滑动到远离目标位置。由于这个原因，很难认定任何特定的操作是否朝着最终的目标迈近了一步。因为没有办法衡量进度，所以很难形成一种策略。很多人通过随机的滑动来解决这个问题，希望恰好能够滑动到最终的目标配置。

但是，瓷砖滑块问题还是存在策略的。为了演示其中的一种方法，我们可以考虑一个更小的网络，它是长方形的，而不是正方形的。

滑动 5 块

有一个 2×3 的网格里放了 5 块瓷砖（编号从 4～8），另外剩下 1 个空格。一开始，这些瓷砖排列得很混乱。瓷砖可以滑动到邻近的空格中，使自己原来的位置成为空格。这个问题的目标是使这个网格中的瓷砖排列有序，4 号瓷砖出现在网格的左上角，接下来依次类推。

读者可能注意到这几块瓷砖是以 4 到 8 编号的，而不是从 1 到 5。读者很快就能知道这样安排的原因。

尽管它是与 8 块的瓷砖滑块相同的基本问题，但只有 5 块瓷砖显然要简单得多。尝试完成如图 1.6 所示的配置。

如果试上几分钟，很可能会找到一种解决方案。从较小数量的瓷砖滑块问题出发，我开发了一个特定的技巧。这个技巧加上我们稍后将要进行的讨论，可以用来解决所有的瓷砖滑块问题。

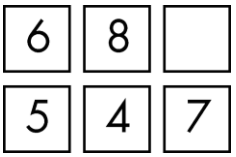

图 1.6 2×3 版本的瓷砖滑块问题的一个特定起始配置

我把这种技巧称为“串列”，它基于对一组瓷砖位置所形成环路的观察。这些位置加上空格就构成了一个瓷砖串列，可以在这个环路中旋转，同时保持这些瓷砖的相对顺序。图 1.7 演示了由 4 个位置所组成的最小可能串列。在一开始的配置中，1 可以滑动到空格中，2 可以滑动到 1 移走后所留下的空格中，最后 3 可以滑动到 2 移走后所留下的空格中。这样，空格就与 1 相邻，使这个串列可以继续旋转。因此，这些瓷砖可以在这条串列路径中有效地旋转到任何位置。

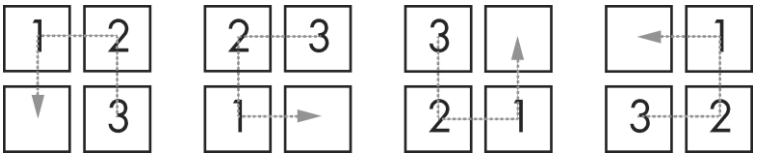

图 1.7 “串列”，与空格相邻的瓷砖开始形成了一条瓷砖路径，在游戏过程中可以像一列火车一样滑动

我们可以使用串列移动一系列的瓷砖，同时保持它们之间的相对关系。现在我们回到前面的 2×3 的网格配置。尽管这个网格中没有任何一个瓷砖位于它最终的正确位置，但有些瓷砖却靠近它们在最终配置中需要靠近的瓷砖。例如，在最终的配置中，4 将出现在 7 的上面，而这两块瓷砖当前是相邻的。如图 1.8 所示，我们可以用一个包含 6 个位置的串列把 4 和 7 移动到它们最终正确的位置。当我们完成这个操作时，剩余的瓷砖几乎都处于正确的

位置，只需要再移动一下 8 就可以了。

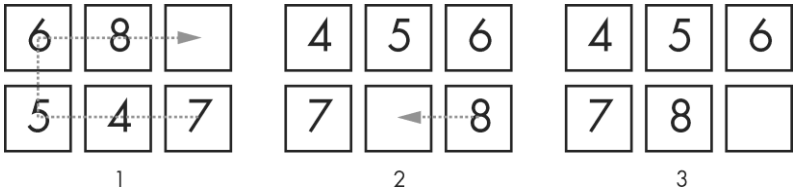

图 1.8 从配置 1 出发，经过 2 次沿规划的“串列”旋转之后来到配置 2，然后只要滑动 1 次就可以产生最终的配置 3

这种技巧是怎么解决所有瓷砖滑块问题的呢？考虑最初的 3×3 配置。我们可以用包含 6 个位置的串列移动相邻的 1 和 2，使 2 和 3 相邻，如图 1.9 所示。

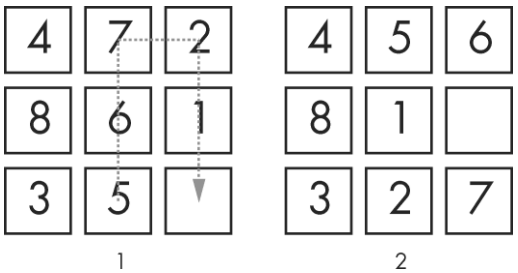

图 1.9 从配置 1 出发，瓷砖沿规定的“串列”旋转到达配置 2

这样 1、2 和 3 就相邻了。在 8 个位置的串列中，我们就可以旋转 1、2 和 3 了，使它们到达最终的正确位置，如图 1.10 所示。

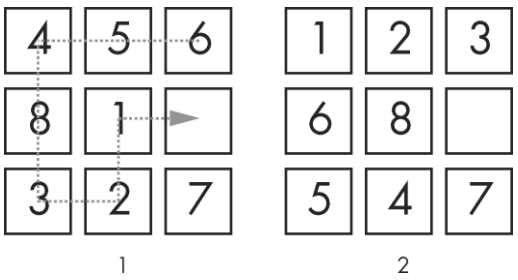

图 1.10 从配置 1 出发，瓷砖经过旋转之后到达配置 2，这样瓷砖 1、2 和 3 位于正确的最终位置

注意瓷砖 4～8 的位置。这些瓷砖的配置正好与前面的 2×3 网格的例子相同。这是一个至关重要的观察。把瓷砖 1～3 放在正确的位置之后，我们就可以把剩余的瓷砖看成是一个更小、更容易解决的独立问题。注意，为了使这种方法可行，必须要解决一整行或一整列的瓷砖。如果我们把瓷砖 1 和 2 放在正确的位置，但 3 仍然置于别处，那样就无法在不移

动左上角两块已经就绪的瓷砖的情况下，把任何瓷砖移动到右上角位置。

这个技巧也可以用于解决规模更大的瓷砖滑块问题。常见的最大尺寸是 15 块瓷砖，也就是 4×4 的网格。这样的问题也可以用分解法来解决：首先把瓷砖 1～4 移动到正确的位置，这样就只剩下一个 3×4 的网格，然后再完成最左列的瓷砖，这样就只剩下一个 3×3 的网格。此时，就只剩下 8 块瓷砖需要移动了。

经验和教训

我们可以从瓷砖滑块问题中学到什么呢？

由于瓷砖移动的次数相当之多，因此无法在初始配置时就规划一个完整的解决方案。但是，无法规划完整的解决方案并不意味着就无法采取策略或技巧系统性地解决问题。在解决编程问题时，有时候会出现无法看到通向解决方案的清晰道路的情况，但这绝不能成为跳过计划和采用系统性方法的借口。更好的办法是采用一种策略，而不是通过简单地反复尝试和失败来解决问题。

我是通过对较小的问题进行研究时发现这种“串列”技巧的。在编程中，我常常会使用这种类似的技巧。在面临一个复杂的问题时，我常常会对这个问题的削减版本进行试验。这些试验常常能够产生有价值的思路。

另一个经验是问题的细分通常并不是非常明显的解决之道。由于移动一块瓷砖不仅影响这块瓷砖本身，还会影响接下来可能发生的移动，人们可能觉得瓷砖滑块问题必须在一个步骤中完成，而不能分阶段解决。因此，花时间研究怎样对问题进行细分通常是非常合算的。即使无法找到一种清晰的细分，仍然有助于增强对这个问题的理解，可以促进这个问题的解决。在解决问题时，头脑里已经拥有一个特定的目标总比随机的尝试要好得多，无论最终是否能够实现这个目标。

1.1.3 数独

数独游戏作为一种在报纸和杂志上出现的益智游戏，其流行程度已经达到了惊人的地步，并且已经发展成一种基于网络和手机的游戏。数独拥有许多不同的版本，但这里只简单讨论传统的版本。

完成数独方块

一个 9×9 的网格，其中部分方格填有数字（1～9），玩家必须填满剩余的空格，并满足下面这个约束条件：对于每一行和每一列，每个数字恰

好只出现 1 次。而且，对于粗框内的每个 3×3 区域，每个数字也恰好只出现 1 次。

如果读者以前曾经玩过这个游戏，很可能已经知道应该采用什么策略在最短的时间内完成一个空格。我们首先观察如图 1.11 所示的方块，关注最关键的起始策略。

数独游戏的难度差异极大，它们的难度取决于需要填充的空格数量。按照这种衡量方法，这是一个相当容易的问题，因为已经有 36 个空格已经填好，只需要再填写 45 个空格就可以完成任务了。问题在于，我们应该首先填充哪个空格呢？

	9	1		6		7		
				8	2		3	9
5		3				2		
			9	1	3		6	2
		2	4		6	8		
1	4		8	2	5			
		9				5		7
6	7		1	5				
		5		4		6	9	

图 1.11 一个很简单的数独方块问题

记住，这个问题包含了约束条件。在每一行、每一列以及粗框内的每个 3×3 区域中，9 个数字必须都正好出现 1 次。这些规则决定了我们应该从哪里着手。最中间的 3×3 区域的 9 个方格已经有 8 个填好了数字。因此，正中心的那个空格只可能填入这个 3×3 区域的其他方格没有出现的那个数字。我们应该从这个空格开始解决这个问题。这个区域所缺少的数字是 7，因此我们在最中间的那个空格中填入 7。

填好了这个数字之后，注意最中间那列的 9 个值已经有 7 个已经填好，只剩下 2 个空格，必须填上这一列所缺少的两个值：3 和 9。与这个列有关的约束条件允许把这两个值放在任意一个位置，但是注意 3 已经在第三行出现过，9 已经在第七行出现过。因此，我们应该把 9 填在中间那列的第三行，把 3 填在中间那列的第七行。图 1.12 对这些步骤进行了概括。

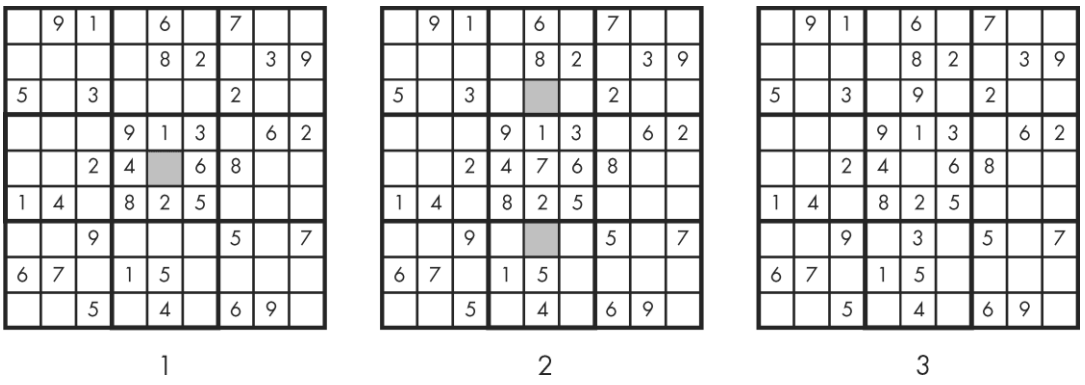

图 1.12 解决示例数独问题的开始步骤

我们不打算完成整个方块的填写，但前面这几个步骤已经说明了重要的一点，就是搜索那些可能出现的值最少的空格。在最理想的情况，就是只剩下一个空格。

经验和教训

我们在数独问题中所学到的主要经验就是应该寻找问题约束性最强的部分。虽然约束条件往往使问题难以着手（还记得狐狸、鹅和玉米问题吗），但它们也可以简化思路，因为它们消除了很多选择。

尽管我们不会在本书中详细讨论人工智能，但还是要简单地提一下，在人工智能中解决某些类型的问题时有一个称为“最大约束变量”的规则。它表示在一个问题中，当我们向一些不同的变量赋一些不同的值来满足约束条件时，应该从约束性最强的变量开始。换用更通俗的说法，就是那些可能采用的值具有最少的变量。

下面是这种思维方式的一个例子。假设有一组工友计划一起吃午饭，并且想要找一家每个人都喜欢的餐厅。问题在于，每个工人对于整个小组的决策都会施加某种程度的影响。例如，Pam 是个素食主义者，Todd 不喜欢中国菜等。如果目标是最大限度地减少寻找餐厅的时间，首先应该询问对餐厅最挑剔的那个工人。例如，如果 Bob 对很多食物都过敏，首先列出他可以进食的餐厅列表是非常合理的。像 Todd 不喜欢中国菜这样的癖好应该放在最后考虑，因为这个困难是很容易克服的。

这个技巧往往也可以用于编程问题。如果问题的某个部分具有很强的约束条件，很可能应该从这一部分开始着手，这样就不必担心把时间花在将来可能会返工的任务上了。一个相关的推论是：应该从最显而易见的那部分任务开始着手。如果可以解决这个部分的问题，就可以在此基础上继续执行其他可以完成的任务。通过审视自己的代码，可能会激发

自己的想象力，从而解决剩余部分的问题。

1.1.4 Quarrasi 锁

对于上面这几个问题，读者以前可能也看到过。但是对于本章的最后一个问题，除非以前曾经阅读过本书，否则绝不可能见过，因为这是我自己“发明”的。请认真阅读，因为这个问题的描述稍微有点复杂。

打开外星锁

一种敌对的外星生物 Quarrasi 登陆到地球上，你在战斗中被它们抓获并关押在飞船里。你设法打倒了看守，尽管它们体形庞大并且长有触角。但是，为了逃离飞船（仍然在地面上），必须打开巨大的舱门。开门的指令非常奇怪，它是用英语的形式显示的，但非常难弄。为了打开舱门，必须沿着轨道滑动 3 个条状的 Kratzz，从右侧的接收器滑动到位于门尽头的左侧接收器，距离大约 3m。

这个任务相当简单，但是必须避免触发警报。警报的工作原理如下：每个 Kratzz 就是一个或多个星形的水晶力量宝石，称为 Quinicrys。每个接收器具有 4 个传感器，如果一个纵列中 Quinicrys 的数量为偶，它们就会被点亮。如果被点亮的传感器的数量正好为 1，就会发出警报。好消息是每个警报都配备了一个抑制器，只要按下这个按钮，就可以防止警报发出声音。如果可以同时按下所有的抑制器，问题就非常简单了，但是没有办法做到这一点，因为人类的胳膊过于短小，不比长长的 Quarassi 触角。

根据上面的描述，怎么才能在不触发任一警报的前提下滑动 Kratzz 打开舱门呢？

图 1.13 展示了初始配置，3 个 Kratzz 都位于右侧的接收器。为了清晰起见，图 1.14 展示了一种不好的思路：把最上面那个 Kratzz 滑动到左侧接收器会导致右侧接收器处于报警状态。你可能想到用抑制器来避免报警，但是要记住，你刚刚把最上面的 Kratzz 移动到左侧接收器，够不着相距 3m 的右侧接收器上的抑制器。

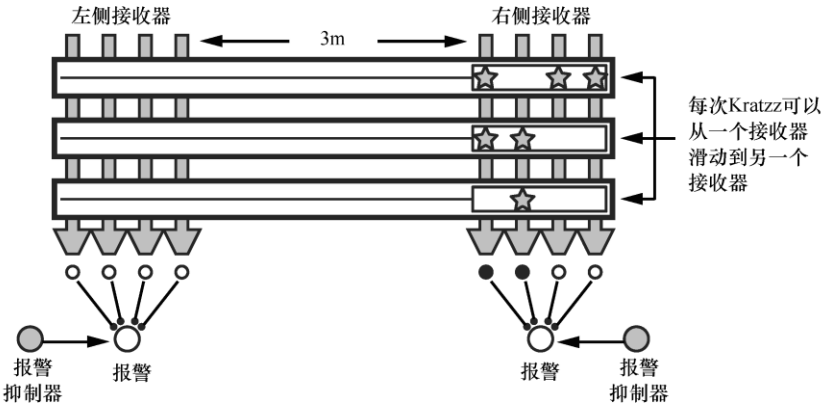

图 1.13 Quarrasi 锁问题的初始配置。必须滑动当前位于右侧接收器的 3 个 Kratzz 条，在不触发任何一个警报的情况下把它们滑动到左侧的接收器。当偶数个星形的 Quinicrys 出现在上面的纵列时，就会点亮一个传感器，如果正好有一个被连接的传感器被点亮，就会触发警报。抑制器可以防止警报发声，但你只能控制自己所站那一侧的抑制器

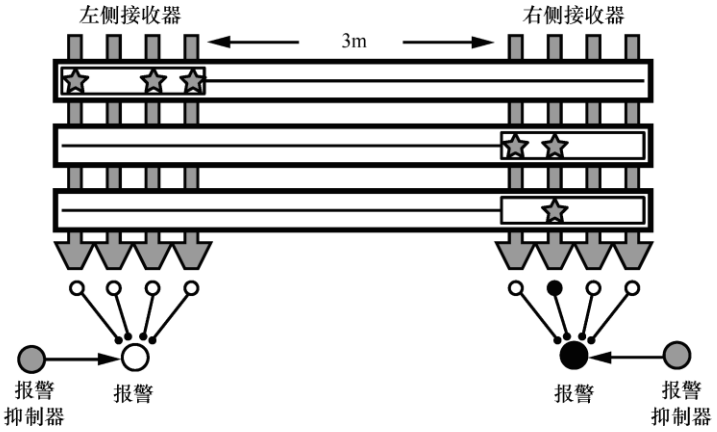

图 1.14 处于报警状态的 Quarrasi 锁。你刚刚把最上面的 Kratzz 滑动到左侧的接收器，因此够不到右侧的接收器。右侧警报的第 2 个传感器被点亮，因为出现在那个纵列的 Quinicrys 数量为偶，现在正好是一个传感器被点亮，所以就会触发报警

在继续尝试之前，先花点时间研究这个问题，设法确定一个解决方案。这取决于看问题的着眼点，此问题并没有看上去那么难。认真地说，要在尝试之前先对它进行思考！

考虑好了吗？现在是不是能够想出一个解决方案？

为了回答这个问题，可以选择两条可能的路径。第一条路径就是不断尝试，不过它是错误的做法：尝试用各种方式移动这几个 Kratzz，一旦达到警报状态时就返回到前一步骤，

直到最终通过一系列的移动，成功地打开锁。

第二条路径是认识到这个问题实际上是个机关。你从前可能没看到过这种机关，它实际上就是披着伪装外衣的狐狸、鹅和玉米问题。尽管警报的规则是以通用的方式描述的，但是与这种特殊的锁有关的组合却是有限的。如果只考虑 3 个 Kratzz，我们只需要知道接收器上的哪些 Kratzz 组合是可以接受的。如果我们把这 3 个 Kratzz 分别命名为 top、middle 和 bottom，那么会触发报警的组合是“top 和 middle”以及“middle 和 bottom”。

如果我们把 top 重新命名为狐狸，把 middle 重新命名为鹅，把 bottom 重新命名为玉米，这样所有的麻烦组合都与狐狸、鹅和玉米问题一样了，也就是“狐狸和鹅”和“鹅和玉米”。

因此，这个问题的解决方式与狐狸、鹅和玉米问题相同。我们把 middle（鹅）滑动到左侧的接收器，再把 top（狐狸）滑动到左侧，当我们移动 top（狐狸）时摁住左侧警报的抑制器；接着，我们把 middle（鹅）滑动回右侧的接收器；然后，我们把 bottom（玉米）滑动到左侧；最后，我们把 middle（鹅）再次滑动到左侧，这样就打开了锁。

经验和教训

这个问题向我们提供的主要经验就是认识到类比的重要性。我们可以看到 Quarrasi 锁问题实际上与狐狸、鹅和玉米问题非常相似。如果我们早早就发现了这种类比，就可以根据狐狸、鹅和玉米问题直接想到解决办法，而不需要从头创建一个新的解决方案，从而大大节省了精力。在解决问题时，大多数类比并不是这样直接，但它们的出现频率却非常高。

如果读者难以发现这个问题和狐狸、鹅和玉米问题之间的联系，这只是因为我故意增加了一些多余的细节。与 Quarrasi 锁问题相关的背景对于解决这个问题而言是无关紧要的，那些外星术语也是如此，它们唯一的作用就是让读者感觉生疏。而且，警报的奇/偶机制使这个问题看上去比实际更为复杂。如果观察 Quinicrys 的实际位置，就可以看到顶部的 Kratzz 和底部的 Kratzz 正好相对，因此它们并不会与警报系统交互。但是，中间的 Kratzz 却与另两个发生交互。

同样，如果没有发现类比，也不必担心。对它们有了足够的警觉之后，就很容易认识到它们。

1.2 基本的问题解决技巧

我们所讨论的这些例子说明了在解决问题时将要采用的许多关键技巧。在本书的剩余部分，我们将观察特定的编程问题，并想办法怎样解决它们。但是，我们首先需要了解一组基本的技巧和原则。有些问题领域需要使用特定的技巧，但下面这些规则几乎适用于所有的场合。如果把它们作为自己的问题解决方法的常规武器，就能想出办法解决任何特定问题。

1.2.1 总是要制订计划

这也许是最重要的规则。我们事先必须要制订计划，而不是直接进行漫无方向的尝试。

根据这个观点，我们应该理解事先制订计划总是可能的。如果在头脑里还不知道怎么解决问题，就不可以制订计划编写代码实现一个解决方案。这个任务是在以后完成的。但是，即使是在开始阶段，还是应该为怎样寻找解决方案制订一个计划。

平心而论，这样的计划可能需要在其他阶段进行修改，或者必须抛弃原先的计划并制

订一个新计划。既然如此，为什么这个规则仍然非常重要呢？艾森豪威尔将军有句名言：“我总是发现计划没什么用处，但计划仍然是必不可少的。”他的意思是战争是极为混乱的，事先预测将要发生的每件事情并为每种结果准备预定的方案是不可能的。从这个意义上说，计划在战场上是没有用处的（普鲁士军事领袖赫尔默特 冯 毛奇曾有名言“一旦与敌人交上火，所有计划都不再有效”）。但是，如果没有计划和组织，任何军队都不可能取得胜利。通过精心制订计划，将军能够了解敌军的战斗力、部队中不同部门的配合方式等重要信息。

同样，我们在解决一个问题时必须要制订一个计划。也许这个计划一旦与敌人交上火就不再有效，也许我们在源代码编辑器中开始输入代码时就会抛弃这个计划，但我们还是必须制订计划。

如果没有计划，我们只能简单地希望“摔出好运气”，相当于在键盘上随意输入却产生了莎士比亚的伟大作品。“摔出好运气”是极为罕见的，而且就算如此可能仍然需要计划。很多人听说过青霉素的发现过程：一位名叫亚历山大 弗莱明的研究人员在一个晚上忘了盖上一个培养皿，第二天早上他发现霉菌抑制了培养皿上细菌的生长。但是，弗莱明并不是干坐在那里等待摔出好运气，他已经按照一种精心可控的方式进行了试验，因此能够认识到他在培养皿上所看到的事实的重要性。（如果我发现前一天晚上所放置的某样东西上长了霉菌，肯定不会对科学产生重大的贡献。）

计划还允许我们设置中期目标并实现它们。如果没有计划，我们就只有一个目标：解决整个问题。在解决了整个问题之前，我们不会感觉自己实现了什么目标。我们很可能有过这样的经验，许多程序在接近完成前没有实现任何实用的功能。因此，只朝着主要目标努力会不可避免地遭受挫折，因为在工作结束之前，不会有正面的推动力激励自己。反之，如果我们创建了一个具有一系列分阶段目标的计划，虽然看上去与主要的问题不是非常密切，仍然可以朝着解决方案取得可衡量的进展，并感觉自己的时间被合理地使用了。在每个工作阶段完成时，我们就可以检查计划所制订的各个事项，更有信心找到解决方案，而不是被日益加重的挫折感所笼罩。

1.2.2 重新陈述问题

狐狸、鹅和玉米的问题鲜活地证明了重新陈述问题可能会产生非常有价值的结果。在某些情况下，一个看上去非常困难的问题如果用一种不同的方式或术语进行阐述，就会变得非常容易。重新陈述问题就像是在登山之前寻找一个不同的出发地点。在登山之前，为什么不从每个角度进行观察，确定最容易攀登的路线呢？

重新陈述问题有时候可以向我们展示此前没有想到的目标。我曾经看过一个故事，一位祖母一边编着毛衣一边照看孙女。为了在织毛衣时不受干扰，祖母把婴儿放在她旁边的一个可移动游戏围栏中，但她的孙女不喜欢待在里面，不停地哭泣。祖母尝试了把所有各种类型的玩具放在围栏里都没有收到效果。最终，她才意识到把孙女放在围栏里只是实现目标的一种方法，她的真正目标是能够安静地织毛衣。解决方案是：让孙女在地毯上自由自在地玩耍，祖母则坐在围栏里面专心织毛衣。重新陈述可以成为一种威力强大的技巧，但很多程序员会忽略它，因为它并不直接与编写代码有关，甚至和解决方案的设计也没什么关联。这是制订计划之所以重要的另一个原因。如果没有计划，我们的目标就是可运行的代码，而重新陈述问题则是将时间用在与编写代码没有关系的地方。有了计划之后，我们就可以把“形式化地重新陈述问题”作为自己的第一个步骤，完成重新陈述就意味着取得了进展。

即使重新陈述问题并没有直接让我们获得新思路，它仍然可能在其他方面提供帮助。例如，如果我们碰到了一个问题（由上级或指导老师所“指派”），我们可以把问题重新陈述给指派这个任务的人，以确认自己的理解无误。另外，重新陈述问题对于使用其他常用的技巧也可能是一个必要的先决步骤，例如削减或划分问题。

总而言之，重新陈述问题可以转换整个问题领域。我在第 6 章的递归解决方案中所使用的技巧就是一种重新陈述递归问题的方法，使我可以像处理迭代问题一样处理它们。

1.2.3 划分问题

找到一种方式把一个问题的解决方法划分为几个步骤或几个阶段，可以使问题更容易解决。如果我们把一个问题划分为两个片段，可以认为每个片段的难度相当于原先整个问题的一半，但通常还要容易得多。

如果读者了解常用的排序算法，对下面这个类比应该是非常熟悉了。假设我们需要把 100 个文件按照字母顺序放在一个箱子里，并且我们采用的基本字母排序方法是一种很有效的被称为插入排序的方法：随机取其中一个文件，把它放在箱子里，然后把第 2 个文件放在箱子中相对于第 1 个文件正确的位置，接着继续这个过程，总是把新文件放在箱子里相对于其他文件正确的位置。因此，在任一特定时刻，箱子中的文件就是按字母顺序排列的。假设有人一开始粗略地把这些文件分成数量大致相等的 4 组：A～F、G～M、N～S 和 T～Z，然后告诉我们分别对各组文件按字母顺序排列，最后依次把各组文件放在箱子里。

如果每组大约包括了 25 个文件，人们可能觉得分别对 4 组 25 个文件按字母排序所需要的工作量和对单组 100 个文件按字母排序的工作量是一样的。实际上，前者的工作量要小得多，因为插入一个文件所需要的工作量是随着已经排好序的文件数量的增加而增长的。我们必须检查箱子中的每个文件才能知道这个新文件应该放在哪个位置。（如果对此感到怀疑，可以考虑一个更极端的版本，比较一下对 50 组 2 个文件进行排序的方法，很可能不到一分钟就能完成全部 100 个文件的排序。）

同样，对问题进行划分常常可以使问题的难度大幅度降低。把各个编程技巧组合在一起使用要比单独使用每个技巧困难得多。例如，如果一段代码在一个 while 循环内部包含了一系列的 if 语句，而这个循环本身又位于一个 for 循环的内部，这样的代码是很难编写和理解的。相对而言，按照顺序编写这些相同的控制语句则要容易编写和理解很多。

我们将在后面的章节中讨论划分问题的具体方法。但是，我们终归应该意识到这种可能性。记住，像瓷砖滑块这样的问题常常隐藏着潜在的划分方法。有时候，寻找问题的划分方法就是削减问题的方法，正如我们稍后将要讨论的一样。

1.2.4 从自己所知的开始

作家在开始创作的时候，常常会收到这样的建议“写自己知道的东西”。这并不意味着作家只能描写他们在日常生活中直接观察到的人和事。如果这样，我们就无法看到奇幻小说、历史小说以及其他许多流派的小说了。但是，这个建议的意思是，作家所写的东西距离他自己的经历越远，写作的难度也就越大。

同样，在编程的时候，我们应该尽量从自己知道的部分开始着手。例如，一旦我们把问题划分为几个片段，应该寻找自己已经知道怎样编写代码的片段。完成了解决方案的一部分之后，可能会激发完成剩余工作的灵感。另外，正如我们可能预料到的那样，问题解决领域的一个常见主题就是取得实际的进展以构建自己的信心，相信自己能够最终完成整个任务。通过从自己所知的领域开始着手，我们就能够构建获得成功的信心和动力。

“从自己所知的开始”这句格言还适用于尚未对问题进行划分的时候。想象一下，有人创建了一个包含每个编程技巧的完整列表：编写一个 C++类、对一个数值列表进行排序、寻找一个链表的最大值等。作为程序员，在开发过程的每一刻，这个列表中可能有许多任务是我们所掌握的，有些是需要花费一些心思的，还有一些则是现在还不知道怎么解决的。一个特定的问题用自己已经掌握的技巧可能是完全可以解决的，也可能是无法解决的。但

是，在寻找其他方法之前，我们应该用已经掌握的技巧对问题进行完整的研究。如果我们把编程技巧看成是工具，把编程问题看成是家庭维护项目，首先应该用手头上已经有的工具进行修理，然后再考虑到五金店购买新工具。

这个技巧遵循了我们已经讨论的原则。它遵循了一个计划，指挥我们的工作。当我们用自己所掌握的技巧对一个问题进行研究时，可以更好地理解这个问题本身以及它的最终目标。

1.2.5 削减问题

当我们面临一个无法解决的问题时，通过这种技巧，可以削减问题的范围。我们可以添加或取消约束条件，产生一个自己知道如何解决的问题。在后面的章节中，我们将通过实际例子讨论这种技巧，不过下面先通过一个基本的例子阐述它的概念。假设一个三维空间中有一系列的坐标，我们的任务是找到空间距离最近的那对坐标。如果我们并不能立即知道怎样解决这个问题，可以采用一些不同的方式削减这个问题以寻求解决方案。例如，如果这些坐标是在一个二维空间而不是三维空间中如何是好？如果削减成这样还是无法解决，那么把二维空间再缩减为一条直线，这些坐标就成了各个不同的数（是否可以用 C++ 的 double 类型表示），这样不是就可以解决了吗？现在，这个问题在本质上就成了在一个数的列表中寻找两个具有最小绝对差的数。

或者，我们在削减问题时仍然让这些坐标位于三维空间中，但只处理 3 个值，而不是任意数量的坐标系列。因此，现在问题就不再是寻找一种算法，计算两个任何坐标之间的距离，而是变成了坐标 A 与坐标 B 比较、坐标 B 与坐标 C 比较，然后是坐标 A 与坐标 C 比较。

这两种削减方法通过不同的方式简化了原先的问题。第一种削减方法消除了计算三维点之间距离的需要。也许我们还不知道该怎样计算空间距离，但在知道怎么做之前，仍然可以取得一些进展。反之，第二种削减方法仍然要求我们计算三维点之间的距离，但是消除了在任意数量的三维点之间寻找最小值的问题。

当然，为了解决最初的问题，我们最终需要组合这两种削减方案所涉及的技巧。即使如此，削减问题允许我们对一个更简单的问题进行操作，即使我们无法找到一种方法把问题划分为几个阶段。在实际效果上，它相当于故意同时、又是临时的小林丸号。尽管我们知道并不是对整个问题进行处理，但是经过削减的问题与原先的问题仍然具有相当多的共

性，足以让我们向最终的解决方案又迈进一步。许多时候，程序员发现他们具备了解决问题所需要的所有单独技巧，通过为问题的每个单独片段编写代码，他们可以想到怎样把各个不同的代码片段组合为一个统一的整体。

削减问题还允许我们准确地理解剩余的难点位于何处。程序员新手常常需要向经验丰富的程序员寻求帮助，但是如果他无法准确地描述自己所需要的帮助，那么对于双方都很可能是一种饱受挫折的体验。我们绝对不可能把问题削减为“这是我的程序，它无法工作。为什么？”使用问题削减技巧，我们可以准确地描述自己所需要的帮助，表示“这是我编写的一些代码。如您所见，我知道怎样计算两个三维坐标之间的距离，并且知道一个距离是否小于另一个距离。但是，我无法找到一种通用的解决方案，在众多坐标中寻找那对具有最小距离的坐标。”

1.2.6 寻找类比

对于我们而言，类比就是一个当前问题和一个已经解决的问题之间的相似性。我们可以利用这种相似性来解决当前问题，而这种相似性可能以多种形式存在。有时候，它意味着两个问题实际上是同一个问题。狐狸、鹅、玉米问题和 Quarrasi 锁问题就属于这种情况。

大多数类比并没有这么直接。有时候，相似性只涉及到问题的一部分。例如，两个数值处理问题可能在其他方面都不一样，唯一的共同点就是都需要比内置的浮点数据类型更精确的数值。我们无法使用这种类比来解决整个问题，但是如果我们已经想出了办法处理额外的精度问题，那就可以按照同样的方式再次处理相同的问题。

尽管认识类比是提高自己的问题解决速度和技能的最重要方式，但它也是最难培养的一种技巧。原因是在一开始很难发现类似的关系，除非有大量以前的解决方案可供参考。

这是成长中的程序员经常寻求捷径的地方。他们常常寻找那些与所需要的代码相似的代码，并在后者的基础上进行修改。但是，出于某些原因，这是错误的做法。首先，如果我们没有自己完成一个解决方案，就不能彻底理解并吸收它。简单地说，要想正确地修改一个自己并没有完全理解的程序是非常困难的。我们并不需要通过编写代码获得完全的理解，但是如果我们无法编写代码，我们对它的理解肯定是有限的。其次，我们所编写的每个成功的程序并不仅仅是一个当前问题的解决方案，它还是一种潜在的类比资源，可以供解决未来的问题所用。我们现在对其他程序员的代码的依赖程度越深，在未来仍然需要这种依赖的可能性也就越大。我们将在第 7 章深入讨论“良好的复用”和“不良的复用”。

1.2.7 试验

有时候，取得进展的最好方法是对事物进行试验并观察其结果。注意，试验与猜测并

不相同。当我们进行猜测时，自己输入一些代码并希望它能够完成任务，但对于能否达到目的自己并没有很强的信心。试验则是一种可控的过程。我们假设当某些代码执行时将会发生什么，然后对它进行试验，观察自己的假设是否正确。根据这些观察，我们可以获得一些信息，帮助自己解决原先的问题。

在处理应用程序编程接口或类库时，试验尤其能够提供帮助。假设我们编写了一个程序，使用了一个表示向量的库类（当元素被添加时能够自动增长的一维数组），但以前从来没有使用过这个类，并且不确定从这个向量中删除一个元素时会出现什么情况。在还没有掌握这个类的详细机制时，不要匆匆用它来解决原先的问题，而是可以先创建一个简短的单独程序，专门对这个向量类进行试验，尤其要在自己关心的场景下对它进行试验。如果在这个“向量演示程序”中花费一点时间，那么在以后的工作中它可以作为这个类的参考使用。

另一种形式的试验与调试相似。例如，一个特定的程序所产生的输出与预期的正好相反。如果输出的是数值并且所输出的数正如预想的一样，但顺序正好相反。如果在检查了代码之后还不明白为什么会发生这种情况，可以进行试验，修改代码，故意产生反向的输出（可能是运行一个反方向的循环）。如果输出结果发生了变化或者没有发生变化，都可能揭示出自己原先的源代码所存在的问题，为自己的思路打开一个缺口。不管怎样，我们都朝着解决方案迈进了一步。

1.2.8 避免陷入挫折感

最后一个技巧其实谈不上技巧，而是一句格言：避免陷入挫折感。当我们陷入挫折感时，自己的思维不再那么清晰、工作不再那么高效，所有的任务需要花费更长的时间并且看上去更为困难。更糟的是，挫折感会不断恶化，很可能从一开始轻度的焦虑变成最终难以遏制的烦躁。

当我向程序员新手提出这个忠告时，他们常常会反驳说，虽然他们在原则上同意我的观点，但他们对于是否会遭遇挫折是无法控制的。要求一位缺乏成功感的程序员避免陷入挫折感岂不是相当于要求小孩子在踩到钉子时不要叫喊吗？除非能够预料到自己将踩到钉子上，否则不可能及时抑制大脑的本能反应。因此，我们能够做的就是让小孩子避免踩到钉子。

程序员的处境并不相同。程序员不会像自我激励的大师一样大声叫喊，他们在遭受挫

折时并不会对外部刺激做出强烈的反应。陷入挫折感的程序员并不是对显示器上的源代码生气，尽管他可能会对着屏幕表达挫折感。反之，陷入挫折感的程序员是对自己生气。挫折的来源同时也是挫折的目标，也就是程序员的思维。

如果我们允许自己陷入挫折感时（我故意使用了“允许”这个词），实际上就为自己继续失败找到了借口。假设我们正在处理一个难题，并且挫折感不断上升。几个小时后，我们咬牙切齿，愤怒地把铅笔折成两截，并告诉自己如果能冷静下来，一定能取得实质性的进展。事实上，我们已经决定向自己的怒气屈服，觉得这要比勇敢面对这个难题容易得多。

最终，避免陷入挫折感是我们必须做出的决定。不过，我们可以采用一些思路，也许相助于实现这一点。首先，不要忘记第一条规则，也就是说始终要制订计划。虽然编写解决原先问题的代码是这个计划的目标，但这并不是这个计划的唯一步骤。因此，如果制订了一个计划并遵循了它，我们就能够取得进展并对此坚信不疑。如果完成了最初计划的所有步骤，但还是无法开始编写代码，这个时候就应该制订另一个计划。

另外，如果读者觉得继续干下去可能会陷入挫折感时，可以休息一会。一个诀窍是让手头上所处理的问题不止一个。这样，如果读者对一个问题感到无可奈何，可以把精力转向另一个问题。注意，如果成功地划分了问题，就可以对单个问题应用这种技巧，只要把陷入僵局的那部分问题扔在一边，转而解决其他部分的问题就可以了。如果没有可以处理的其他问题，也可以离开椅子做些其他事情。可以热热身，放松一下脑子，例如散步、洗衣服、做伸展运动（如果读者是一个整天坐在计算机前的程序员，我强烈建议养成做伸展运动的习惯）。在休息结束之前，不要再考虑那个问题。

1.3 习题

记住，为了真正学到东西，只有通过实践才有可能。因此，要尽可能多地完成习题。当然在第 1 章中，我们还没有讨论编程。但即使是这样，我还是鼓励读者尝试完成一些习题。读者可以把下面这些习题看成是演奏真正音乐之前的指法练习。

1.1 完成一个中等难度的数独题（可以从网络或当地报纸上寻找），用不同的策略进行试验，并对结果加以说明。能不能为解决数独问题编写一个通用计划呢？

1.2 考虑瓷砖滑块问题的一种变型，每块瓷砖上显示的是图片而不是数字。这种变化使难度增加了多少？为什么？

1.3 为瓷砖滑块问题寻找一种与我不同的解决策略。

1.4 搜索旧式的狐狸、鹅和玉米问题的变型并尝试解决它们。很多著名的难题来源于 Sam Loyd 或者是由他推广的。因此，可以通过他的名字进行搜索。而且，一旦发现（或者觉得太难而放弃思考，只是看看）了解决方案，思考怎样据此创建难题的简化版本。有哪些东西必须修改？仅仅修改约束条件还是描述方式？

1.5 尝试为其他传统的铅笔和纸游戏（例如纵横填字谜）编写一种显式的策略。应该从什么地方开始呢？在陷入僵局时应该怎么做呢？即使是“Jumble（类似七巧板的益智游戏）”这样的简单报纸游戏，也非常适合思考它的解决策略。