The Pragmatic Programmers

TURING 图灵程序设计丛书

A Common-Sense Guide to
Data Structures and Algorithms

数据结构与算法图解

[美] 杰伊·温格罗◎著　袁志鹏◎译

- 摒弃复杂概念，非计算机专业读者也能看懂的专业书
- 只需了解简单加减乘除和图表分解便可升级核心编程技能

中国工信出版集团　人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

A Common-Sense Guide to Data Structures and Algorithms

数据结构与算法图解

[美] 杰伊·温格罗◎著　袁志鹏◎译

人民邮电出版社
北　京

图书在版编目（CIP）数据

数据结构与算法图解 /（美）杰伊·温格罗（Jay Wengrow）著；袁志鹏译. -- 北京：人民邮电出版社，2019.4
（图灵程序设计丛书）
ISBN 978-7-115-50900-0

Ⅰ. ①数… Ⅱ. ①杰… ②袁… Ⅲ. ①数据结构－图解②算法分析－图解 Ⅳ. ①TP311.12-64②TP301.6-64

中国版本图书馆CIP数据核字(2019)第037843号

内 容 提 要

本书是数据结构与算法的入门指南，不局限于某种特定语言，略过复杂的数学公式，用通俗易懂的方式针对编程初学者介绍数据结构与算法的基本概念，培养读者编程逻辑。主要内容包括：为什么要了解数据结构与算法，大 *O* 表示法及其代码优化利用，栈、队列等的合理使用，等等。

本书适合编程初学者、非计算机专业出身的程序员等阅读。

◆ 著 [美] 杰伊·温格罗
译 袁志鹏
责任编辑 张海艳
责任印制 周昇亮
◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷
◆ 开本：800×1000 1/16
印张：10.5
字数：248千字 2019年4月第1版
印数：1－4 000册 2019年4月北京第1次印刷
著作权合同登记号 图字：01-2017-9356号

定价：49.00元

读者服务热线：(010)51095186转600 印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

版权声明

前　言

数据结构与算法并不只是抽象的概念，掌握好的话可以写出更高效、运行得更快的代码，这对于如今盛行的网页和移动应用开发来说尤为重要。如果你最近一次使用算法是在大学课堂上或求职面试时，那你应该还没见识到它的真正威力。

这个主题的大多数资料都有一种通病——晦涩难懂。满纸的数学术语，搞得除非你是数学家，不然真不知道作者在说什么。即使是一些声称“简化”过的书，看起来也好像已经认定读者都掌握了高深的数学知识。这就导致了很多人对此主题望而生畏，以为自己的智商不足以理解这些概念。

但事实上，数据结构与算法都是能够从常识推导出来的。数学符号只是一种特定的语言，数学里的一切都是可以用常识去解释的。本书用到的数学知识就只有加减乘除和指数，所有的概念都可以用文字来解释。我还会采用大量的图表以便读者轻松地理解。

一旦掌握了这些知识，你就能写出高效、快速、优雅的代码。你还能权衡各种写法的优劣，并能合理判断适用于给定情况的最优方案。

一些读者可能是因为学校开设了这门课或者为准备技术面试而阅读本书的。本书对计算机科学基础的解释能有效地帮助你达到目的。此外，我还鼓励你正视这些概念在日常编程中的实用价值。为此，我将书中阐述的概念与实际结合，其中的用例都可供大家使用。

目标读者

本书适合以下读者。

- 有编程基础的初级开发者，想学习一些计算机科学的基本概念，以优化代码，提高编程技能。
- 自学编程的开发者，没学过正规的计算机科学课程（或者学过但忘光了），现在想利用数据结构与算法使代码更灵活、更具扩展性。
- 计算机科学专业的学生，希望找到用简洁语言阐述数据结构与算法的资料。这本书很适合作为“经典”教材的补充参考。
- 开发人员，平时也许没怎么利用过数据结构与算法的知识，希望复习这些概念为下次技术面试做准备。

为了使本书不特定于某种语言，我们的示例代码会用到多种语言，包括 Ruby、Python 和 JavaScript，了解这些语言的话可能会学得更快。不过，这些示例代码都没有严格按照惯用语法来写，避免读者因看不懂某种语言的特有语法而困惑。所以即使不太熟悉某种语言，也还是能跟得上的。

本书内容

本书的主旨就是数据结构与算法，具体内容如下。

第 1 章和第 2 章，解释数据结构和算法是什么，并探索时间复杂度这一判断算法效率的概念。此过程中还会经常提及数组、集合和二分查找。

第 3 章，以老奶奶都听得懂的方式去揭示大 *O* 记法的本质。因为大 *O* 记法全书都会用到，所以对这一章的理解非常重要。

第 4 章、第 5 章和第 6 章，进一步探索大 *O* 记法，并以实例来演示如何利用它来加快代码运行速度。这一路上，我们还会提到各种排序算法，包括冒泡排序、选择排序和插入排序。

第 7 章和第 8 章会再探讨几种数据结构，包括散列表、栈和队列，展示它们对代码速度和可读性的影响，并学会用其解决实际问题。

第 9 章会介绍递归，计算机科学中的核心概念。我们会对其进行分解，考察它在某些问题上的利用价值。第 10 章会运用递归来实现一些飞快的算法，例如快速排序和快速选择，提升读者的算法开发能力。

第 11 章、第 12 章和第 13 章会探索基于结点的数据结构，包括链表、二叉树和图，并展示它们在各种应用中的完美表现。

最后一章，第 14 章，介绍空间复杂度。当程序运行环境的内存空间不多，或处理的数据量很大时，理解空间复杂度便显得特别重要。

如何阅读本书

你得按顺序从第 1 章开始读起。虽然有些书允许读者单独翻阅某些章节，或跳过某些章节，**但这本不是**。本书的每一章都假定你已经读过其之前的内容，而且全书内容也确实是精心安排的，使得你在按序阅读的过程中逐步提高认知水平。

此外还有很重要的一点：为了易于大家理解，当介绍一个概念时，我可能不会把它一下子全部展露出来。有时候，理解一个复杂概念的最好方法就是把它拆分成小块，并且在完全明白某一块以后才去着手其他部分。要是我在描述某个术语时说得比较模糊，千万别把它当成一个完整的定义，想看清该术语的全貌，你得读完关于它的所有内容才行。

这其实是一种权衡：为了便于理解，我只能把一个概念先极度简化，然后再一步步去完善。当然这就导致了有些句子写得不够彻底、不够学术，或不够精确。但无须担心，因为到最后你一定能对它有一个完整的印象。

在线资源

本书网址是 https://pragprog.com/book/jwdsal。读者可从中获取更多关于本书的信息，或以下面的方式互动：

- 在论坛跟其他读者和笔者交流；
- 提交勘误[1]，改进本书。

各章习题以及示例代码下载见 http://commonsensecomputerscience.com[2]。

电子书

扫描如下二维码，即可购买本书电子版。

致谢

虽然写书好像只是个人的事情，但这个过程如果没有大家的支持，是不可能完成的。我想感谢所有帮助过我的人。

感谢我的妻子 Rena，谢谢你一直陪伴我，给予我情感上的支持。当我像隐士般久坐写书时，你帮我料理好了一切事情。感谢我可爱的孩子们——Tuvi、Leah 和 Shaya，谢谢你们耐心地等候我完成这本“算发”书。是的，我终于写完了。

感谢我的父母——Howard Wengrow 先生和 Debbie Wengrow 夫人，谢谢你们当初激发了我对计算机编程的兴趣，并鼓励我追逐梦想。你们可能不知道，正是你们作为 9 岁生日礼物送给我的计算机，为我奠定了职业生涯的基础，也为如今这本书埋下了种子。

① 本书中文版勘误请到 http://ituring.cn/book/2538 查看和提交。——编者注

② 读者也可以到图灵社区本书页面下载示例代码，网址是 http://ituring.cn/book/2538。——编者注

当初我给 Pragmatic 出版公司提交草稿时，我认为自己写得挺不错。但融合了诸位优秀编辑的各种专业意见和要求之后，我发现它比初稿好太多了。感谢总编 Brian MacDonald，你教会了我如何写书，你的见解使得本书的每一章都更为清晰，可以说本书随处可见你的思想印记。感谢我的责任编辑 Susannah Pfalzer，你让我心中显现了成书的模样，把只有理论的草稿变成一本真正能给每一位程序员阅读的书。感谢出版商 Andy Hunt 和 Dave Thomas，谢谢你们信任我的作品，谢谢你们把 Pragmatic 打造成世界上最值得投稿的出版社。

感谢 Colleen McGuckin 这位天才软件开发者以及艺术家，谢谢你把我简陋的图画变成了精美的数码图片。没有这些你以才华和技巧细心描绘的图片，这本书可能一文不值。

让我感到很幸运的是，这本书还经过了不少专家的评审。各位的反馈都非常有益，令本书内容之精确达到了极致。非常感谢你们的贡献：Aaron Kalair、Alberto Boschetti、Alessandro Bahgat、Arun S. Kumar、Brian Schau、Daivid Morgan、Derek Graham、Frank Ruiz、Ivo Balbaert、Jasdeep Narang、Jason Pike、Javier Collado、Jeff Holland、Jessica Janiuk、Joy McCaffrey、Kenneth Parekh、Matteo Vaccari、Mohamed Fouad、Neil Hainer、Nigel Lowry、Peter Hampton、Peter Wood、Rod Hilton、Sam Rose、Sean Lindsay、Stephan Kämper、Stephen Orr、Stephen Wolff 和 Tibor Simic。

我还要感谢所有 Actualize 的同事、同学和校友。感谢大家通过各种方式为这个本属于 Actualize 内部的项目做出了贡献。特别感谢 Luke Evans，是他让我萌生了写书的想法。

感谢大家让此书成真。

Jay Wengrow
jay@actualize.co
2017 年 8 月

目　录

第 1 章 数据结构为何重要

哪怕只写过几行代码的人都会发现，编程基本上就是在跟**数据**打交道。计算机程序总是在接收数据、操作数据或返回数据。不管是求两数之和的小程序，还是管理公司的企业级软件，都运行在数据之上。

数据是一个广义的术语，可以指代各种类型的信息，包括最基本的数字和字符串。在经典的“Hello World!”这个简单程序中，字符串`"Hello World!"`就是一条数据。事实上，无论是多么复杂的数据，我们都可以将其拆成一堆数字和字符串来看待。

数据结构则是指数据的组织形式。看看以下代码。

```
x = "Hello!"
y = "How are you"
z = "today?"

print x + y + z
```

这个非常简单的程序把 3 条数据串成了一句连贯的话。如果要描述该程序中的数据结构，我们会说，这里有 3 个独立的变量，分别引用着 3 个独立的字符串。

但在本书中你将会学到，数据结构不只是用于组织数据，它还极大地影响着代码的**运行速度**。因为数据结构不同，程序的运行速度可能相差多个数量级。如果你写的程序要处理大量的数据，或者要让数千人同时使用，那么你采用何种数据结构，将决定它是能够运行，还是会因为不堪重负而崩溃。

一旦对各种数据结构有了深刻的理解，并明白它们对程序性能方面的影响，你就能写出快速而优雅的代码，从而使软件运行得快速且流畅。当然，你的编程技能也会更上一层楼。

本章接下来将会分析两种数据结构：数组和集合。它们从表面上看好像差不多，但通过即将介绍的分析工具，你将会观察到它们在性能上的差异。

1.1 基础数据结构：数组

数组是计算机科学中最基本的数据结构之一。如果你用过数组，那么应该知道它就是一个含

有数据的列表。它有多种用途，适用于各种场景，下面就举个简单的例子。

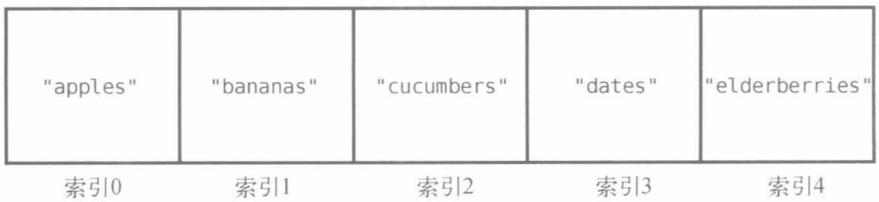
一个允许用户创建和使用购物清单的食杂店应用软件，其源代码可能会包含以下的片段。

```
array = ["apples", "bananas", "cucumbers", "dates", "elderberries"]
```

这就是一个数组，它刚好包含 5 个字符串，每个代表我会从超市买的食物。

此外，我们会用一些名为**索引**的数字来标识每项数据在数组中的位置。

在大多数的编程语言中，索引是从 0 算起的，因此在这个例子中，`"apples"`的索引为 0，`"elderberries"`的索引为 4，如下所示。



若想了解某个数据结构（例如数组）的性能，得分析程序怎样操作这一数据结构。

一般数据结构都有以下 4 种**操作**（或者说用法）。

- **读取**：查看数据结构中某一位置上的数据。对于数组来说，这意味着查看某个索引所指的数据值。例如，查看索引 2 上有什么食品，就是一种读取。
- **查找**：从数据结构中找出某个数据值的所在。对于数组来说，这意味着检查其是否包含某个值，如果包含，那么还得给出其索引。例如，检查`"dates"`是否存在于食品清单之中，给出其对应的索引，就是一种查找。
- **插入**：给数据结构增加一个数据值。对于数组来说，这意味着多加一个格子并填入一个值。例如，往购物清单中多加一项`"figs"`，就是一种插入。
- **删除**：从数据结构中移走一个数据值。对于数组来说，这意味着把数组中的某个数据项移走。例如，把购物清单中的`"bananas"`移走，就是一种删除。

本章我们将会研究这些操作在数组上的运行速度。

同时，我们也将学到本书的第一个重要理论：**操作的速度，并不按时间计算，而是按步数计算**。

为什么呢?

因为，你不可能很绝对地说，某项操作要花 5 秒。它在某台机器上要跑 5 秒，但换到一台旧一点的机器，可能就要多于 5 秒，而换到一台未来的超级计算机，运行时间又将显著缩短。所以，受硬件影响的计时方法，非常不可靠。

然而，若按**步数**来算，则确切得多。如果 A 操作要 5 步，B 操作要 500 步，那么我们可以很

肯定地说，无论是在什么样的硬件上对比，A 都快过 B。因此，衡量步数是分析速度的关键。

此外，操作的速度，也常被称为**时间复杂度**。在本书中，我们会提到**速度**、**时间复杂度**、**效率**、**性能**，但它们其实指的都是步数。

事不宜迟，我们现在就来探索上述 4 种操作方式在数组上要花多少步。

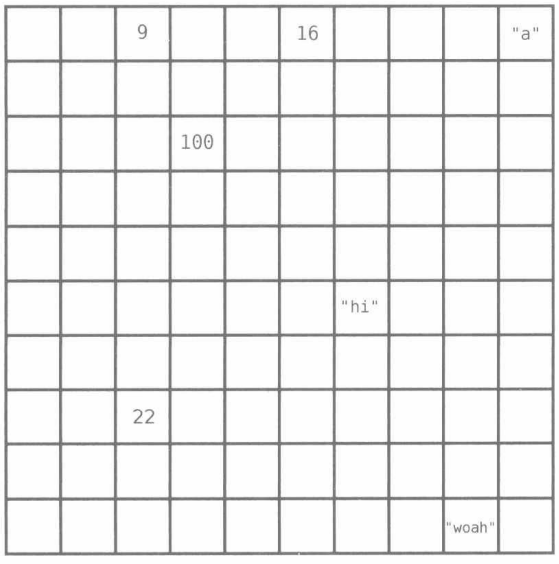
1.1.1 读取

首先看看**读取**，即查看数组中某个索引所指的数据值。

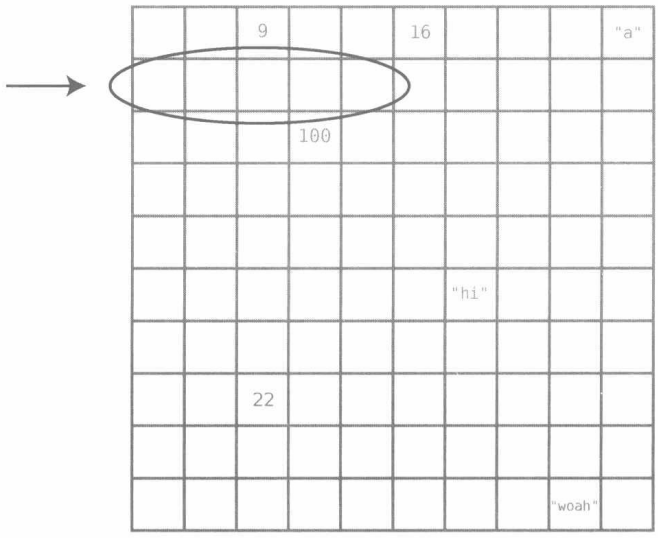
这只要一步就够了，因为计算机本身就有跳到任一索引位置的能力。在`["apples", "bananas", "cucumbers", "dates", "elderberries"]`的例子中，如果要查看索引 2 的值，那么计算机就会直接跳到索引 2，并告诉你那里有`"cucumbers"`。

计算机为什么能一步到位呢？原因如下。

计算机的内存可以被看成一堆格子。下图是一片网格，其中有些格子有数据，有些则是空白。



当程序声明一个数组时，它会先划分出一些连续的空格子以备使用。换句话说，如果你想创建一个包含 5 个元素的数组，计算机就会找出 5 个排成一行的空格子，将其当成数组。



内存中的每个格子都有各自的地址，就像街道地址，例如大街 123 号。不过内存地址就只用一个普通的数字来表示。而且，每个格子的内存地址都比前一个大 1，如下图所示。

1000	1001	1002	1003	1004	1005	1006	1007	1008	1009
1010	1011	1012	1013	1014	1015	1016	1017	1018	1019
1020	1021	1022	1023	1024	1025	1026	1027	1028	1029
1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
1040	1041	1042	1043	1044	1045	1046	1047	1048	1049
1050	1051	1052	1053	1054	1055	1056	1057	1058	1059
1060	1061	1062	1063	1064	1065	1066	1067	1068	1069
1070	1071	1072	1073	1074	1075	1076	1077	1078	1079
1080	1081	1082	1083	1084	1085	1086	1087	1088	1089
1090	1091	1092	1093	1094	1095	1096	1097	1098	1099

购物清单数组的索引和内存地址，如下图所示。

"apples"	"bananas"	"cucumbers"	"dates"	"elderberries"
内存地址：1010	1011	1012	1013	1014
索引：0	1	2	3	4

计算机之所以在读取数组中某个索引所指的值时，能直接跳到那个位置上，是因为它具备以下条件。

(1) 计算机可以一步就跳到任意一个内存地址上。（就好比，要是你知道大街 123 号在哪儿，那么就可以直奔过去。）

(2) 数组本身会记有第一个格子的内存地址，因此，计算机知道这个数组的开头在哪里。

(3) 数组的索引从 0 算起。

回到刚才的例子，当我们叫计算机读取索引 3 的值时，它会做以下演算。

(1) 该数组的索引从 0 算起，其开头的内存地址为 1010。

(2) 索引 3 在索引 0 后的第 3 个格子上。

(3) 于是索引 3 的内存地址为 1013，因为 1010 + 3 = 1013。

当计算机一步跳到 1013 时，我们就能获取到"dates"这个值了。

所以，数组的读取是一种非常高效的操作，因为它只要一步就好。一步自然也是最快的速度。这种一步读取任意索引的能力，也是数组好用的原因之一。

如果我们问的不是“索引 3 有什么值”，而是“"dates"在不在数组里”，那么这就需要进行查找操作了。下面我们就来看看。

1.1.2 查找

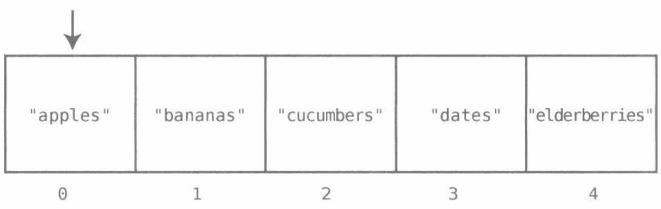
如前所述，对于数组来说，**查找**就是检查它是否包含某个值，如果包含，还得给出其索引。那么，我们就试试在数组中查找"dates"要用多少步。

对于我们人来说，可以一眼就看到这个购物清单上的"dates"，并数出它的索引为 3。但是，计算机并没有眼睛，它只能一步一步地检查整个数组。

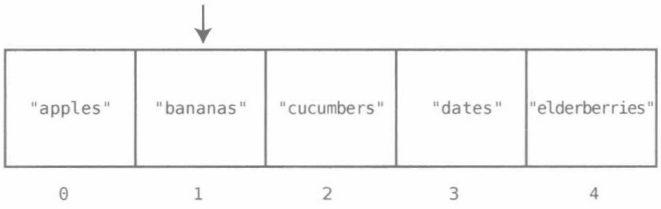
想要查找数组中是否存在某个值，计算机会先从索引 0 开始，检查其值，如果不匹配，则继续下一个索引，以此类推，直至找到为止。

我们用以下图来演示计算机如何从购物清单中查找"dates"。

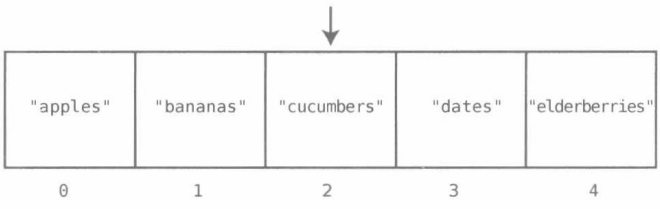
首先，计算机检查索引 0。



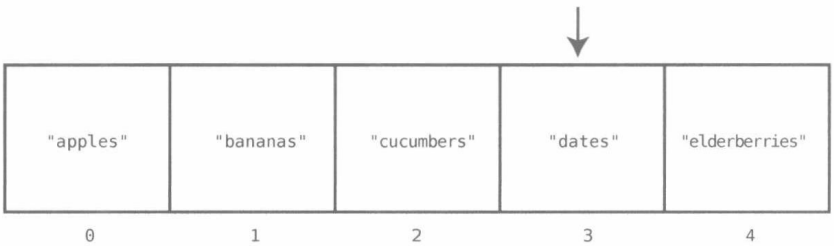
因为索引 0 的值是"apples"，并非我们所要的"dates"，所以计算机跳到下一个索引上。



索引 1 也不是"dates"，于是计算机再跳到索引 2。



但索引 2 的值仍不匹配，计算机只好再跳到下一格。



啊，真是千辛万苦，我们找到"dates"了，它就在索引 3 那里。自此，计算机不用再往后跳了，因为结果已经得到。

在这个例子中，因为我们检查了 4 个格子才找到想要的值，所以这次操作总计是 4 步。

这种逐个格子去检查的做法，就是最基本的查找方法——**线性查找**。第 2 章我们还会学习另一种查找方法。

但在那之前，我们再思考一下，在数组上进行线性查找**最多**要多少步呢？

如果我们要找的值刚好在数组的最后一个格子里（如本例的 elderberries），那么计算机从头到尾检查**每个**格子，会在最后才找到。同样，如果我们要找的值并不存在于数组中，那么计算机也还是得查遍每个格子，才能确定这个值不在数组中。

于是，一个 5 格的数组，其线性查找的步数最大值是 5，而对于一个 500 格的数组，则是 500。

以此类推，一个 N 格的数组，其线性查找的最多步数是 N（N 可以是任何自然数）。

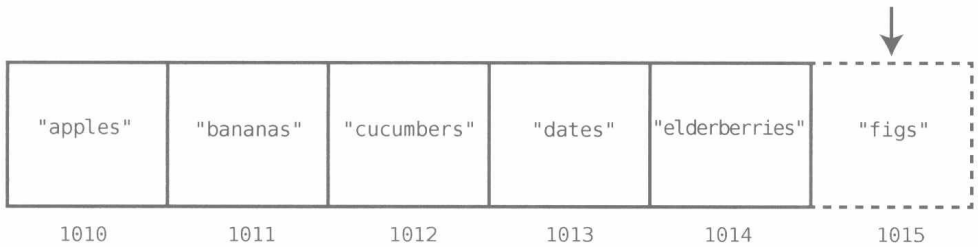
可见，无论是多长的数组，查找都比读取要慢，因为读取永远都只需要一步，而查找却可能需要多步。

接下来，我们再研究一下插入，准确地说，是插入一个新值到数组之中。

1.1.3 插入

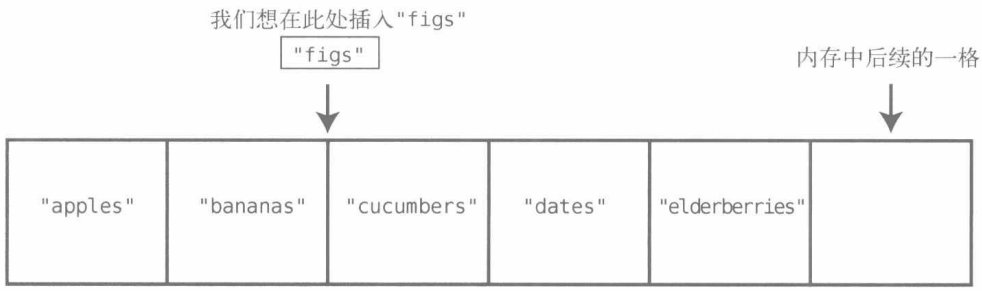
往数组里插入一个新元素的速度，取决于你想把它插入到**哪个**位置上。

假设我们想要在购物清单的末尾插入"figs"。那么只需一步。因为之前说过了，计算机知道数组开头的内存地址，也知道数组包含多少个元素，所以可以算出要插入的内存地址，然后一步跳到那里插入就行了。图示如下。



但在数组**开头或中间**插入，就另当别论了。这种情况下，我们需要移动其他元素以腾出空间，于是得花费额外的步数。

例如往索引 2 处插入"figs"，如下所示。



为了达到目的，我们必须先把"cucumbers"、"dates"和"elderberries"往右移，以便空出索引 2。而这也不是一步就能移好，因为我们首先要将"elderberries"右移一格，以空出位置给"dates"，然后再将"dates"右移，以空出位置给"cucumbers"，下面来演示这个过程。

第 1 步："elderberries"右移。