

# 目 录

致谢
数据结构思维中文版
前言
第一章 接口
第二章 算法分析
第三章 ArrayList
第四章 LinkedList
第五章 双链表
第六章 树的遍历
第七章 到达哲学
第八章 索引器
第九章 Map接口
第十章 哈希
第十一章 HashMap
第十二章 TreeMap
第十三章 二叉搜索树
第十四章 持久化
第十五章 爬取维基百科
第十六章 布尔搜索
第十七章 排序

# 致谢

当前文档 《数据结构思维中文版》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-05-03。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：<http://www.bookstack.cn/books/think-dast-zh>

书栈官网：<http://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

# 数据结构思维中文版

- [数据结构思维中文版](#)
  - [赞助我](#)
  - [协议](#)

# 数据结构思维中文版

---

原书: [Think Data Structures: Algorithms and Information Retrieval in Java](#)

译者: [飞龙](#)

版本: 1.0.0

自豪地采用[谷歌翻译](#)

- [在线阅读](#)
- [PDF格式](#)
- [EPUB格式](#)
- [MOBI格式](#)
- [代码仓库](#)

## 赞助我

---



## 协议

---

[CC BY-NC-SA 4.0](#)



# 前言

- [前言](#)
  - [本书背后的哲学](#)
    - [0.1 预备条件](#)
  - [处理代码](#)
  - [贡献者](#)

# 前言

原文: [Preface](#)

译者: [飞龙](#)

协议: [CC BY-NC-SA 4.0](#)

自豪地采用[谷歌翻译](#)

## 本书背后的哲学

数据结构和算法是过去 50 年来最重要的发明之一，它们是软件工程师需要了解的基础工具。但是在我看来，这些话题的大部分书籍都过于理论，过于庞大，也是“自底向上”的：

过于理论

算法的数学分析基于许多简化假设，它们限制了实践中的可用性。这个话题的许多描述都掩盖了简化，并专注于数学。在这本书中，我介绍了这个话题的最实际的子集，并省略或不强调其余的内容。

过于庞大

这些话题的大多数书籍至少有 500 页，有些超过 1000 页。通过关注我认为对软件工程师最有用的话题，我把这本书限制在 200 页以下。

过于“自底向上”

许多数据结构的书籍着重于数据结构如何工作（实现），而不是使用它们（接口）。在这本书中，我从接口开始，“自顶向下”。读者在学习如何使用 Java 集合框架中的结构之后，再了解它们的工作原理。

最后，有些书将这个材料展示在上下文之外，缺少动机：这只是另一个数据结构！我试图使之生动起来，通过围绕一个应用 - 网页搜索 - 来组织这些话题，它广泛使用数据结构，并且是一个有趣和重要的话题。

这个应用激发了一些话题，通常不会在介绍性数据结构的课中涵盖，包括 Redis 的持久化数据结

构。

我已经做出了一些艰难的决定，来进行取舍，但我也做了一些妥协。我包括了大多数读者永远不会使用的一些话题，但是可能在技术面试中，你需要知道这些话题。对于这些话题，我提出了传统的观点和我怀疑的理由。

本书还介绍了软件工程实践的基本方面，包括版本控制和单元测试。大多数章节都包括一个练习，允许读者应用他们学到的内容。每个练习都提供自动化测试，来检查解决方案。对于大多数练习，我在下一章的开头展示我的解决方案。

## 0.1 预备条件

本书面向计算机科学及相关领域的大学生，专业软件工程师，软件工程培训人员和技术面试准备人员。

在你开始读这本书之前，你应该很熟悉 Java，尤其应该知道如何定义一个扩展现有类的新类，或实现一个 `interface`。如果你不熟悉 Java 了，这里有两本书可以用于起步：

- Downey 和 Mayfield, 《Think Java》(O'Reilly Media, 2016)，它面向以前从未编程过的人。
- Sierra 和 Bates, 《Head First Java》(O'Reilly Media, 2005)，它适用于已经知道另一种编程语言的人。

如果你不熟悉 Java 中的接口，你可能需要在 <http://thinkdast.com/interface> 上完成一个名为“什么是接口”的教程。

一个词汇注解：“接口”这个词可能会令人困惑。在应用编程接口 (API) 的上下文中，它指代一组提供某些功能的类和方法。

在 Java 的上下文中，它还指代一个与类相似的语言特性，它规定了一组方法。为了避免混淆，我将使用正常字体中的“接口”来表示接口的一般思想，代码字体的 `interface` 用于 Java 语言特性。

你还应该熟悉类型参数和泛型类型。例如，你应该知道如何使用类型参数创建对象，如 `ArrayList<Integer>`。如果不是，你可以在 <http://thinkdast.com/types> 上了解类型参数。

你应该熟悉 Java 集合框架 (JCF)，你可以阅读 <http://thinkdast.com/collections>。特别是，你应该知道 `List interface`，以及 `ArrayList` 和 `LinkedList` 类。

理想情况下，你应该熟悉 Apache Ant，它是 Java 的自动化构建工具。你可以在 <http://thinkdast.com/anttut> 上阅读 Ant 的更多信息。

你应该熟悉 JUnit，它是 Java 的单元测试框架。你可以在 <http://thinkdast.com/junit> 上阅读更多信息。

# 处理代码

---

本书的代码位于 <http://thinkdast.com/repo> 上的 Git 仓库中。

Git 是一个“版本控制系统”，允许你跟踪构成项目的文件。Git 控制下的文件集合称为“仓库”。

GitHub 是一个托管服务，为 Git 仓库提供存储和方便的 Web 界面。它提供了几种使用代码的方法：

- 你可以通过按下 `Fork`（派生）按钮，在 GitHub 上创建仓库的副本。如果你还没有 GitHub 帐户，则需要创建一个。派生之后，你可以在 GitHub 上拥有你自己的仓库，你可以使用它们来跟踪你编写的代码。然后，你可以“克隆”仓库，它将文件的副本下载到你的计算机。
- 或者，你可以克隆仓库而不进行派生。如果你选择此选项，则不需要 GitHub 帐户，但你无法将更改保存在 GitHub 上。
- 如果你不想使用 Git，你可以使用 GitHub 页面上的 `Download`（下载）按钮或此链接 <http://thinkdast.com/zip>，以 ZIP 压缩包格式下载代码。

克隆仓库或解压 ZIP 文件后，你应该有一个名为 `ThinkDataStructures` 的目录，其中有一个名为 `code` 的子目录。

本书中的示例是使用 Java SE 7 开发和测试的。如果你使用的是较旧的版本，一些示例将无法正常工作。如果你使用的是更新版本，那么它们都应该能用。

# 贡献者

---

这本书是我为纽约市 Flatiron School 写的课程的一个改编版，它提供了编程和网页开发相关的各种在线课程。他们提供基于这个材料的课程，提供在线开发环境，来自教师和其他学生的帮助，以及结业证书。你可以在 <http://flatironschool.com> 上找到更多信息。

- 在 Flatiron School，Joe Burgess，Ann John 和 Charles Pletcher 通过实现和测试，提供了来自初始规范的指导，建议和更正。谢谢你们！
- 我非常感谢我的技术审校员 Barry Whitman，Patrick White 和 Chris Mayfield，他提出了许多有用的建议，并捕获了许多错误。当然，任何剩余的错误都是我的错，而不是他们的错！
- 感谢 Olin College 的数据结构和算法课程中的教师和学生，他们读了这本书并提供了有用的反馈。

如果你对文本有任何意见或建议，请发送至：[feedback@greenteapress.com](mailto:feedback@greenteapress.com)。





# 第一章 接口

- 第一章 接口
  - 1.1 为什么有两种 `List` ？
  - 1.2 Java 中的接口
  - 1.3 `List` 接口
  - 1.4 练习 1

# 第一章 接口

原文: *Chapter 1 Interfaces*

译者: 飞龙

协议: *CC BY-NC-SA 4.0*

自豪地采用谷歌翻译

本书展示了三个话题:

- 数据结构: 从 Java 集合框架 (JCF) 中的结构开始, 你将学习如何使用列表和映射等数据结构, 你将看到它们的工作原理。
- 算法分析: 我提供了技术, 来分析代码以及预测运行速度和需要多少空间 (内存)。
- 信息检索: 为了激发前两个主题, 并使练习更加有趣, 我们将使用数据结构和算法构建简单的 Web 搜索引擎。

以下是话题顺序的大纲:

- 我们将从 `List` 接口开始, 你将编写实现这个接口的两种不同的方式。然后我们将你的实现与 Java `ArrayList` 和 `LinkedList` 类进行比较。
- 接下来, 我将介绍树形数据结构, 你将处理第一个应用程序: 一个程序, 从维基百科页面读取页面, 解析内容, 并遍历生成的树来查找链接和其他特性。我们将使用这些工具来测试“到达哲学”的猜想 (你可以通过阅读 <http://thinkdast.com/getphil> 来了解)。
- 我们将了解 Java 的 `Map` 接口和 `HashMap` 实现。然后, 你将使用哈希表和二叉搜索树来编写实现此接口的类。
- 最后, 你将使用这些 (以及其他一些我之前介绍的) 类来实现一个 Web 搜索引擎, 其中包括: 一个查找和读取页面的爬虫程序, 一个存储网页内容的索引器, 以便有效地搜索, 以及一个从用户那里接受查询并返回相关结果的检索器。

让我们开始吧。

## 1.1 为什么有两种 `List` ？

当人们开始使用 Java 集合框架时，有时候会混淆 `ArrayList` 和 `LinkedList`。为什么 Java 提供两个 `List interface` 的实现呢？你应该如何选择使用哪一个？我们将在接下来的几章回答这些问题。

我将从回顾 `interface` 和实现它们的类开始，我将介绍“面向接口编程”的概念。

在最初的几个练习中，你将实现类似于 `ArrayList` 和 `LinkedList` 的类，这样你就会知道他们如何工作，我们会看到，他们每个类都有优点和缺点。对于 `ArrayList`，一些操作更快或占用更少的空间；但对于 `LinkedList` 其他操作更快或空间更少。哪一个更适合于特定的应用程序，取决于它最常执行的操作。

## 1.2 Java 中的接口

Java `interface` 规定了一组方法；任何实现这个 `interface` 的类都必须提供这些方法。例如，这里是 `Comparable` 的源代码，它是定义在 `java.lang` 包中的 `interface`：

```
1. public interface Comparable<T> {
2.     public int compareTo(T o);
3. }
```

这个 `interface` 的定义使用类型参数 `T`，这使得 `Comparable` 是个泛型类型。为了实现这个 `interface`，一个类必须：

- 规定类型 `T`，以及，
- 提供一个名为 `compareTo` 的方法，接受一个对象作为参数，并返回 `int`。

例如，以下是 `java.lang.Integer` 的源代码：

```
1. public final class Integer extends Number implements Comparable<Integer> {
2.
3.     public int compareTo(Integer anotherInteger) {
4.         int thisVal = this.value;
5.         int anotherVal = anotherInteger.value;
6.         return (thisVal < anotherVal ? -1 : (thisVal == anotherVal ? 0 : 1));
7.     }
8.
9.     // other methods omitted
10. }
```

译者注：根据 `Comparable<T>` 的文档，不必要这么复杂，直接返回 `this.value - that.value` 就足够了。

这个类扩展了 `Number`，所以它继承了 `Number` 的方法和实例变量；它实现 `Comparable<Integer>`，所以它提供了一个名为 `compareTo` 的方法，接受 `Integer` 并返回一个 `int`。

当一个类声明它实现一个 `interface`，编译器会检查，它提供了所有 `interface` 定义的方法。

除此之外，这个 `compareTo` 的实现使用“三元运算符”，有时写作 `?:`。如果你不熟悉，可以阅读 <http://thinkdast.com/ternary>。

## 1.3 `List` 接口

Java集合框架（JCF）定义了一个 `interface`，称为 `List`，并提供了两个实现方式，`ArrayList` 和 `LinkedList`。

这个 `interface` 定义了 `List` 是什么意思；实现它的任何类 `interface` 必须提供一组特定的方法，包括 `add`，`get`，`remove`，以及其它大约 20 个。

`ArrayList` 并 `LinkedList` 提供这些方法，因此可以互换使用。用于 `List` 也可用于 `ArrayList`，`LinkedList`，或实现 `List` 的其它任何对象。

这是一个人为的示例，展示了这一点：

```

1. public class ListClientExample {
2.     private List list;
3.
4.     public ListClientExample() {
5.         list = new LinkedList();
6.     }
7.
8.     private List getList() {
9.         return list;
10.    }
11.
12.    public static void main(String[] args) {
13.        ListClientExample lce = new ListClientExample();
14.        List list = lce.getList();
15.        System.out.println(list);
16.    }
17. }
```

`ListClientExample` 没有任何有用的东西，但它封装了 `List`，并具有一个类的基本要素。也就是说，它包含一个 `List` 实例变量。我会使用这个类来表达这个要点，然后你将在第一个练习中使用它。

通过实例化（也就是创建）新的 `LinkedList`，这个 `ListClientExample` 构造函数初始化 `list`；读取器方法叫做 `getList`，返回内部 `List` 对象的引用；并且 `main` 包含几行代码来测试这些方法。

这个例子的要点是，它尽可能地使用 `List`，避免指定 `LinkedList`，`ArrayList`，除非有必要。例如，实例变量被声明为 `List`，并且 `getList` 返回 `List`，但都不指定哪种类型的列表。

如果你改变主意并决定使用 `ArrayList`，你只需要改变构造函数；你不必进行任何其他更改。

这种风格被称为基于接口的编程，或者更随意，“面向接口编程”（见 <http://thinkdast.com/interbaseprog>）。这里我们谈论接口的一般思想，而不是 Java 接口。

当你使用库时，你的代码只依赖于类似“列表”的接口。它不应该依赖于一个特定的实现，像 `ArrayList`。这样，如果将来的实现发生变化，使用它的代码仍然可以工作。

另一方面，如果接口改变，依赖于它的代码也必须改变。这就是为什么库的开发人员避免更改接口，除非绝对有必要。

## 1.4 练习 1

因为这是第一个练习，我们会保持简单。你将从上一节获取代码并交换实现；也就是说，你会将 `LinkedList` 替换为 `ArrayList`。因为面向接口编写程序，你将能够通过更改一行并添加一个 `import` 语句来交换实现。

以建立你的开发环境来开始。对于所有的练习，你需要能够编译和运行 Java 代码。我使用 JDK7 来开发示例。如果你使用的是更新的版本，则所有内容都应该仍然可以正常工作。如果你使用的是旧版本，可能会发现某些东西不兼容。

我建议使用交互式开发环境（IDE）来获取语法检查，自动完成和源代码重构。这些功能可帮助你避免错误或快速找到它们。但是，如果你正在准备技术面试，请记住，在面试期间你不会拥有这些工具，因此你也可以在没有他们的情况下练习编写代码。

如果你尚未下载本书的代码，请参阅 0.1 节中的指南。

在名为 `code` 的目录中，你应该找到这些文件和目录：

- `build.xml` 是一个 Ant 文件，可以更容易地编译和运行代码。
- `lib` 包含你需要的库（对于这个练习，只是 JUnit）。
- `src` 包含源代码。

如果你浏览 `src/com/allendowney/thinkdast`，你将找到此练习的源代码：

- `ListClientExample.java` 包含上一节的代码。
- `ListClientExampleTest.java` 包含一个 JUnit 测试 `ListClientExample`。

查看 `ListClientExample` 并确保你了解它的作用。然后编译并运行它。如果你使用 Ant，你可以访问代码目录并运行 `ant ListClientExample`。

你可能会得到一个警告。

```
1. List is a raw type. References to generic type List<E>
```

```
2. should be parameterized.
```

为了使这个例子保持简单，我没有留意在列表中指定元素的类型。如果此警告让你烦恼，你可以通过将 `List` 或 `LinkedList` 替换为 `List<Integer>` 或 `LinkedList<Integer>` 来修复。

回顾 `ListClientExampleTest`。它运行一个测试，创建一个 `ListClientExample`，调用 `getList`，然后检查结果是否是一个 `ArrayList`。最初，这个测试会失败，因为结果是一个 `LinkedList`，而不是一个 `ArrayList`。运行这个测试并确认它失败。

注意：这个测试对于这个练习是有意义的，但它不是测试的一个很好的例子。良好的测试应该检查被测类是否满足接口的要求；他们不应该依赖于实现的细节。

在 `ListClientExample` 中，将 `LinkedList` 替换为 `ArrayList`。你可能需要添加一个 `import` 语句。编译并运行 `ListClientExample`。然后再次运行测试。修改了这个之后，测试现在应该通过了。

为了这个测试通过，你只需要在构造函数中更改 `LinkedList`；你不必更改任何 `List` 出现的地方。如果你这样做会发生什么？来吧，将一个或者多个 `List` 替换为 `ArrayList`。程序仍然可以正常工作，但现在是“过度指定”了。如果你将来改变主意，并希望再次交换接口，则必须更改代码。

在 `ListClientExample` 构造函数中，如果将 `ArrayList` 替换为 `List`，会发生什么？为什么不能实例化 `List`？

## 第二章 算法分析

- 第二章 算法分析
  - 2.1 选择排序
  - 2.2 大 O 表示法
  - 2.3 练习 2

## 第二章 算法分析

原文: [Chapter 2 Analysis of Algorithms](#)

译者: 飞龙

协议: [CC BY-NC-SA 4.0](#)

自豪地采用[谷歌翻译](#)

我们在前面的章节中看到, Java 提供了两种 `List` 接口的实现, `ArrayList` 和 `LinkedList`。对于一些应用, `LinkedList` 更快; 对于其他应用, `ArrayList` 更快。

要确定对于特定的应用, 哪一个更好, 一种方法是尝试它们, 并看看它们需要多长时间。这种称为“性能分析”的方法有一些问题:

- 在比较算法之前, 你必须实现这两个算法。
- 结果可能取决于你使用什么样的计算机。一种算法可能在一台机器上更好; 另一个可能在不同的机器上更好。
- 结果可能取决于问题规模或作为输入提供的数据。

我们可以使用算法分析来解决这些问题中的一些问题。当它有效时, 算法分析使我们可以比较算法而不必实现它们。但是我们必须做出一些假设:

- 为了避免处理计算机硬件的细节, 我们通常会识别构成算法的基本操作, 如加法, 乘法和数字比较, 并计算每个算法所需的操作次数。
- 为了避免处理输入数据的细节, 最好的选择是分析我们预期输入的平均性能。如果不可能, 一个常见的选择是分析最坏的情况。
- 最后, 我们必须处理一个可能性, 一种算法最适合小问题, 另一个算法适用于较大的问题。在这种情况下, 我们通常专注于较大的问题, 因为小问题的差异可能并不重要, 但对于大问题, 差异可能是巨大的。

这种分析适用于简单的算法分类。例如, 如果我们知道算法 `A` 的运行时间通常与输入规模成正比, 即 `n`, 并且算法 `B` 通常与 `n ** 2` 成比例, 我们预计 `A` 比 `B` 更快, 至少对于 `n` 的较大值。

大多数简单的算法只能分为几类。

- 常数时间：如果运行时间不依赖于输入的大小，算法是“常数时间”。例如，如果你有一个  $n$  个元素的数组，并且使用下标运算符 (`[]`) 来访问其中一个元素，则此操作将执行相同数量的操作，而不管数组有多大。
- 线性：如果运行时间与输入的大小成正比，则算法为“线性”的。例如，如果你计算数组的和，则必须访问  $n$  个元素并执行  $n - 1$  个添加。操作的总数（元素访问和加法）为  $2 * n - 1$ ，与  $n$  成正比。
- 平方：如果运行时间与  $n^2$  成正比，算法是“平方”的。例如，假设你要检查列表中的任何元素是否多次出现。一个简单的算法是将每个元素与其他元素进行比较。如果有  $n$  个元素，并且每个元素与  $n - 1$  个其他元素进行比较，则比较的总数是  $n^2 - n$ ，随着  $n$  增长它与  $n^2$  成正比。

## 2.1 选择排序

例如，这是一个简单算法的实现，叫做“选择排序”（请见 <http://thinkdast.com/selectsort>）：

```

1. public class SelectionSort {
2.
3.     /**
4.      * Swaps the elements at indexes i and j.
5.      */
6.     public static void swapElements(int[] array, int i, int j) {
7.         int temp = array[i];
8.         array[i] = array[j];
9.         array[j] = temp;
10.    }
11.
12.    /**
13.     * Finds the index of the lowest value
14.     * starting from the index at start (inclusive)
15.     * and going to the end of the array.
16.     */
17.    public static int indexLowest(int[] array, int start) {
18.        int lowIndex = start;
19.        for (int i = start; i < array.length; i++) {
20.            if (array[i] < array[lowIndex]) {
21.                lowIndex = i;
22.            }
23.        }
24.        return lowIndex;
25.    }
26.
27.    /**
28.     * Sorts the elements (in place) using selection sort.
29.     */

```

```

30.     public static void selectionSort(int[] array) {
31.         for (int i = 0; i < array.length; i++) {
32.             int j = indexLowest(array, i);
33.             swapElements(array, i, j);
34.         }
35.     }
36. }

```

第一个方法 `swapElements` 交换数组的两个元素。元素的是常数时间的操作，因为如果我们知道元素的大小和第一个元素的位置，我们可以使用一个乘法和一个加法来计算任何其他元素的位置，这都是常数时间的操作。由于 `swapElements` 中的一切都是恒定的时间，整个方法是恒定的时间。

第二个方法 `indexLowest` 从给定的索引 `start` 开始，找到数组中最小元素的索引。每次遍历循环的时候，它访问数组的两个元素并执行一次比较。由于这些都是常数时间的操作，因此我们计算什么并不重要。为了保持简单，我们来计算一下比较的数量。

- 如果 `start` 为 `0`，则 `indexLowest` 遍历整个数组，并且比较的总数是数组的长度，我称之为 `n`。
- 如果 `start` 为 `1`，则比较数为 `n - 1`。
- 一般情况下，比较的次数是 `n - start`，因此 `indexLowest` 是线性的。

第三个方法 `selectionSort` 对数组进行排序。它从 `0` 循环到 `n - 1`，所以循环执行了 `n` 次。每次调用 `indexLowest` 然后执行一个常数时间的操作 `swapElements`。

第一次 `indexLowest` 被调用的时候，它进行 `n` 次比较。第二次，它进行 `n - 1` 比较，依此类推。比较的总数是

```
1. n + n-1 + n-2 + ... + 1 + 0
```

这个数列的和是  $n(n+1)/2$ ，它（近似）与  $n^2$  成正比；这意味着 `selectionSort` 是平方的。

为了得到同样的结果，我们可以将 `indexLowest` 看作一个嵌套循环。每次调用 `indexLowest` 时，操作次数与 `n` 成正比。我们调用它 `n` 次，所以操作的总数与  $n^2$  成正比。

## 2.2 大 O 表示法

所有常数时间算法属于称为  $O(1)$  的集合。所以，说一个算法是常数时间的另一个方法就是，说它是  $O(1)$  的。与之类似，所有线性算法属于  $O(n)$ ，所有二次算法都属于  $O(n^2)$ 。这种分类算法的方式被称为“大 O 表示法”。

注意：我提供了一个大 O 符号的非专业定义。更多的数学处理请参见

<http://thinkdast.com/bigo>。

这个符号提供了一个方便的方式，来编写通用的规则，关于算法在我们构造它们时的行为。例如，如



果你执行线性时间算法，之后是常量算法，则总运行时间是线性的。 $\in$  表示“是...的成员”：

```
1.  $f \in O(n) \ \&\& \ g \in O(1) \Rightarrow f + g \in O(n)$ 
```

如果执行两个线性运算，则总数仍然是线性的：

```
1.  $f \in O(n) \ \&\& \ g \in O(n) \Rightarrow f + g \in O(n)$ 
```

事实上，如果你执行任何次数的线性运算， $k$ ，总数就是线性的，只要  $k$  是不依赖于  $n$  的常数。

```
1.  $f \in O(n) \ \&\& \ k \text{ 是常数} \Rightarrow kf \in O(n)$ 
```

但是，如果执行  $n$  次线性运算，则结果为平方：

```
1.  $f \in O(n) \Rightarrow nf \in O(n ** 2)$ 
```

一般来说，我们只关心  $n$  的最大指数。所以如果操作总数为  $2 * n + 1$ ，则属于  $O(n)$ 。主要常数  $2$  和附加项  $1$  对于这种分析并不重要。与之类似， $n ** 2 + 100 * n + 1000$  是  $O(n ** 2)$  的。不要被大的数值分心！

“增长级别”是同一概念的另一个名称。增长级别是一组算法，其运行时间在同一个大  $O$  分类中；例如，所有线性算法都属于相同的增长级别，因为它们的运行时间为  $O(n)$ 。

在这种情况下，“级别”是一个团体，像圆桌骑士的阶级，这是一群骑士，而不是一种排队方式。因此，你可以将线性算法的阶级设想为一组勇敢，仗义，特别有效的算法。

## 2.3 练习 2

本章的练习是实现一个 `List`，使用 Java 数组来存储元素。

在本书的代码库（请参阅 0.1 节）中，你将找到你需要的源文件：

- `MyArrayList.java` 包含 `List` 接口的部分实现。其中四个方法是不完整的；你的工作是填充他们。
- `MyArrayListTest.java` 包含 JUnit 测试，可用于检查你的工作。

你还会发现 Ant 构建文件 `build.xml`。你应该可以从代码目录运行 `ant MyArrayList`，来运行 `MyArrayList.java`，其中包含一些简单的测试。或者你可以运行 `ant MyArrayListTest` 运行 JUnit 测试。

当你运行测试时，其中几个应该失败。如果你检查源代码，你会发现四条 TODO 注释，表示你应该填

充的方法。

在开始填充缺少的方法之前，让我们来看看一些代码。这里是类定义，实例变量和构造函数。

```
1. public class MyArrayList<E> implements List<E> {
2.     int size;                // keeps track of the number of elements
3.     private E[] array;       // stores the elements
4.
5.     public MyArrayList() {
6.         array = (E[]) new Object[10];
7.         size = 0;
8.     }
9. }
```

正如注释所述，`size` 跟踪 `MyArrayList` 中由多少元素，而且 `array` 是实际包含的元素的数组。

构造函数创建一个 10 个元素的数组，这些元素最初为 `null`，并且 `size` 设为 `0`。大多数时候，数组的长度大于 `size`，所以数组中由未使用的槽。

Java 的一个细节：你不能使用类型参数实例化数组；例如，这样不起作用：

```
1. array = new E [10];
```

要解决此限制，你必须实例化一个 `Object` 数组，然后进行类型转换。你可以在 <http://thinkdast.com/generics> 上阅读此问题的更多信息。

接下来，我们将介绍添加元素到列表的方法：

```
1. public boolean add(E element) {
2.     if (size >= array.length) {
3.         // make a bigger array and copy over the elements
4.         E[] bigger = (E[]) new Object[array.length * 2];
5.         System.arraycopy(array, 0, bigger, 0, array.length);
6.         array = bigger;
7.     }
8.     array[size] = element;
9.     size++;
10.    return true;
11. }
```

如果数组中没有未使用的空间，我们必须创建一个更大的数组，并复制这些元素。然后我们可以将元素存储在数组中并递增 `size`。

为什么这个方法返回一个布尔值，这可能不明显，因为它似乎总是返回 `true`。像之前一样，你可以在文档中找到答案：<http://thinkdast.com/colladd>。如何分析这个方法的性能也不明显。在正

常情况下，它是常数时间的，但如果我们必须调整数组的大小，它是线性的。我将在 3.2 节中介绍如何处理这个问题。

最后，让我们来看看 `get`；之后你可以开始做这个练习了。

```
1. public T get(int index) {
2.     if (index < 0 || index >= size) {
3.         throw new IndexOutOfBoundsException();
4.     }
5.     return array[index];
6. }
```

其实 `get` 很简单：如果索引超出范围，它会抛出异常；否则读取并返回数组的元素。注意，它检查索引是否小于 `size`，大于等于 `array.length`，所以它不能访问数组的未使用的元素。

在 `MyArrayList.java` 中，你会找到 `set` 的桩，像这样：

```
1. public T set(int index, T element) {
2.     // TODO: fill in this method.
3.     return null;
4. }
```

阅读 `set` 的文档，在 <http://thinkdast.com/listset>，然后填充此方法的主体。如果再运行 `MyArrayListTest`，`testSet` 应该通过。

提示：尽量避免重复索引检查的代码。

你的下一个任务是填充 `indexOf`。像往常一样，你应该阅读 <http://thinkdast.com/listindof> 上的文档，以便你知道应该做什么。特别要注意它应该如何处理 `null`。

我提供了一个辅助方法 `equals`，它将数组中的元素与目标值进行比较，如果它们相等，返回 `true`（并且正确处理 `null`），则返回。请注意，此方法是私有的，因为它仅在此类中使用；它不是 `List` 接口的一部分。

完成后，再次运行 `MyArrayListTest`；`testIndexOf`，以及依赖于它的其他测试现在应该通过。

只剩下两个方法了，你需要完成这个练习。下一个是 `add` 的重载版本，它接受下标并将新值存储在给定的下标处，如果需要，移动其他元素来腾出空间。

再次阅读 <http://thinkdast.com/listadd> 上的文档，编写一个实现，并运行测试进行确认。

提示：避免重复扩充数组的代码。

最后一个：填充 `remove` 的主体。文档位于 <http://thinkdast.com/listrem>。当你完成它时，

所有的测试都应该通过。

一旦你的实现能够工作，将其与我的比较，你可以在 <http://thinkdast.com/myarraylist> 上找到它。

## 第三章 ArrayList

- 第三章 `ArrayList`
  - 3.1 划分 `MyArrayList` 的方法
  - 3.2 `add` 的划分
  - 3.3 问题规模
  - 3.4 链接数据结构
  - 3.5 练习 3
  - 3.6 垃圾回收的注解

## 第三章

`ArrayList`

原文: [Chapter 3 ArrayList](#)

译者: 飞龙

协议: [CC BY-NC-SA 4.0](#)

自豪地采用谷歌翻译

本章一举两得：我展示了上一个练习的解法，并展示了一种使用摊销分析来划分算法的方法。

### 3.1 划分 `MyArrayList` 的方法

对于许多方法，我们不能通过测试代码来确定增长级别。例如，这里是 `MyArrayList` 的 `get` 的实现：

```
1. public E get(int index) {
2.     if (index < 0 || index >= size) {
3.         throw new IndexOutOfBoundsException();
4.     }
5.     return array[index];
6. }
```

`get` 中的每个东西都是常数时间的。所以 `get` 是常数时间，没问题。

现在我们已经划分了 `get`，我们可以使用它来划分 `set`。这是我们以前的练习中的 `set`：

```
1. public E set(int index, E element) {
2.     E old = get(index);
3.     array[index] = element;
4.     return old;
5. }
```

该解决方案的一个有些机智的部分是，它不会显式检查数组的边界；它利用 `get`，如果索引无效则引发异常。

`set` 中的一切，包括 `get` 的调用都是常数时间，所以 `set` 也是常数时间。

接下来我们来看一些线性的方法。例如，以下是我的实现 `indexOf`：

```
1. public int indexOf(Object target) {
2.     for (int i = 0; i < size; i++) {
3.         if (equals(target, array[i])) {
4.             return i;
5.         }
6.     }
7.     return -1;
8. }
```

每次在循环中，`indexOf` 调用 `equals`，所以我们首先要划分 `equals`。这里就是：

```
1. private boolean equals(Object target, Object element) {
2.     if (target == null) {
3.         return element == null;
4.     } else {
5.         return target.equals(element);
6.     }
7. }
```

此方法调用 `target.equals`；这个方法的运行时间可能取决于 `target` 或 `element` 的大小，但它不依赖于该数组的大小，所以出于分析 `indexOf` 的目的，我们认为这是常数时间。

回到之前的 `indexOf`，循环中的一切都是常数时间，所以我们必须考虑的下一个问题是：循环执行多少次？

如果我们幸运，我们可能会立即找到目标对象，并在测试一个元素后返回。如果我们不幸，我们可能需要测试所有的元素。平均来说，我们预计测试一半的元素，所以这种方法被认为是线性的（除了在不太可能的情况下，我们知道目标元素在数组的开头）。

`remove` 的分析也类似。这里是我的时间。

```
1. public E remove(int index) {
2.     E element = get(index);
3.     for (int i=index; i<size-1; i++) {
4.         array[i] = array[i+1];
5.     }
6.     size--;
```

```

7.     return element;
8. }

```

它使用 `get`，这是常数时间，然后从 `index` 开始遍历数组。如果我们删除列表末尾的元素，循环永远不会运行，这个方法是常数时间。如果我们删除第一个元素，我们遍历所有剩下的元素，它们是线性的。因此，这种方法同样被认为是线性的（除了在特殊情况下，我们知道元素在末尾，或到末尾距离恒定）。

## 3.2 `add` 的划分

这里是 `add` 的一个版本，接受下标和元素作为参数：

```

1. public void add(int index, E element) {
2.     if (index < 0 || index > size) {
3.         throw new IndexOutOfBoundsException();
4.     }
5.     // add the element to get the resizing
6.     add(element);
7.
8.     // shift the other elements
9.     for (int i=size-1; i>index; i--) {
10.         array[i] = array[i-1];
11.     }
12.     // put the new one in the right place
13.     array[index] = element;
14. }

```

这个双参数的版本，叫做 `add(int, E)`，它使用了单参数的版本，称为 `add(E)`，它将新的元素放在最后。然后它将其他元素向右移动，并将新元素放在正确的位置。

在我们可以划分双参数 `add` 之前，我们必须划分单参数 `add`：

```

1. public boolean add(E element) {
2.     if (size >= array.length) {
3.         // make a bigger array and copy over the elements
4.         E[] bigger = (E[]) new Object[array.length * 2];
5.         System.arraycopy(array, 0, bigger, 0, array.length);
6.         array = bigger;
7.     }
8.     array[size] = element;
9.     size++;
10.    return true;
11. }

```

单参数版本很难分析。如果数组中存在未使用的空间，那么它是常数时间，但如果我们必须调整数组的大小，它是线性的，因为 `System.arraycopy` 所需的时间与数组的大小成正比。

那么 `add` 是常数还是线性时间的？我们可以通过考虑一系列  $n$  个添加中，每次添加的平均操作次数，来分类此方法。为了简单起见，假设我们以一个有 2 个元素的空间的数组开始。

- 我们第一次调用 `add` 时，它会在数组中找到未使用的空间，所以它存储 1 个元素。
- 第二次，它在数组中找到未使用的空间，所以它存储 1 个元素。
- 第三次，我们必须调整数组的大小，复制 2 个元素，并存储 1 个元素。现在数组的大小是 4。
- 第四次存储 1 个元素。
- 第五次调整数组的大小，复制 4 个元素，并存储 1 个元素。现在数组的大小是 8。
- 接下来的 3 个添加储存 3 个元素。
- 下一个添加复制 8 个并存储 1 个。现在的大小是 16。
- 接下来的 7 个添加复制了 7 个元素。

以此类推，总结一下：

- 4 次添加之后，我们储存了 4 个元素，并复制了两个。
- 8 次添加之后，我们储存了 8 个元素，并复制了 6 个。
- 16 次添加之后，我们储存了 16 个元素，并复制了 14 个。

现在你应该看到了规律：要执行  $n$  次添加，我们必须存储  $n$  个元素并复制  $n-2$  个。所以操作总数为  $n + n - 2$ ，为  $2 * n - 2$ 。

为了得到每个添加的平均操作次数，我们将总和除以  $n$ ；结果是  $2 - 2 / n$ 。随着  $n$  变大，第二项  $2 / n$  变小。参考我们只关心  $n$  的最大指数的原则，我们可以认为 `add` 是常数时间的。

有时线性的算法平均可能是常数时间，这似乎是奇怪的。关键是我们每次调整大小时都加倍了数组的长度。这限制了每个元素被复制的次数。否则 - 如果我们向数组的长度添加一个固定的数量，而不是乘以一个固定的数量 - 分析就不起作用。

这种划分算法的方式，通过计算一系列调用中的平均时间，称为摊销分析。你可以在 <http://thinkdast.com/amort> 上阅读更多信息。重要的想法是，复制数组的额外成本是通过一系列调用展开或“摊销”的。

现在，如果 `add(E)` 是常数时间，那么 `add(int, E)` 呢？调用 `add(E)` 后，它遍历数组的一部分并移动元素。这个循环是线性的，除了在列表末尾添加的特殊情况中。因此，`add(int, E)` 是线性的。

## 3.3 问题规模

最后一个例子中，我们将考虑 `removeAll`，这里是 `MyArrayList` 中的实现：

```
1. public boolean removeAll(Collection<?> collection) {
```



```

2.     boolean flag = true;
3.     for (Object obj: collection) {
4.         flag &= remove(obj);
5.     }
6.     return flag;
7. }

```

每次循环中，`removeAll` 都调用 `remove`，这是线性的。所以认为 `removeAll` 是二次的很诱人。但事实并非如此。

在这种方法中，循环对于每个 `collection` 中的元素运行一次。如果 `collection` 包含 `m` 个元素，并且我们从包含 `n` 个元素的列表中删除，则此方法是  $O(nm)$  的。如果 `collection` 的大小可以认为是常数，`removeAll` 相对于 `n` 是线性的。但是，如果集合的大小与 `n` 成正比，`removeAll` 则是平方的。例如，如果 `collection` 总是包含 100 个或更少的元素，`removeAll` 则是线性的。但是，如果 `collection` 通常包含的列表中的 1% 元素，`removeAll` 则是平方的。

当我们谈论问题规模时，我们必须小心我们正在讨论哪个大小。这个例子演示了算法分析的陷阱：对循环计数的诱人捷径。如果有一个循环，算法往往是线性的。如果有两个循环（一个嵌套在另一个内），则该算法通常是平方的。不过要小心！你必须考虑每个循环运行多少次。如果所有循环的迭代次数与 `n` 成正比，你可以仅仅对循环进行计数之后离开。但是，如在这个例子中，迭代次数并不总是与 `n` 成正比，所以你必须考虑更多。

## 3.4 链接数据结构

对于下一个练习，我提供了 `List` 接口的部分实现，使用链表来存储元素。如果你不熟悉链表，你可以阅读 <http://thinkdast.com/linkedlist>，但本部分会提供简要介绍。

如果数据结构由对象（通常称为“节点”）组成，其中包含其他节点的引用，则它是“链接”的。在链表中，每个节点包含列表中下一个节点的引用。其他链接结构包括树和图，其中节点可以包含多个其他节点的引用。

这是一个简单节点的定义：

```

1. public class ListNode {
2.
3.     public Object data;
4.     public ListNode next;
5.
6.     public ListNode() {
7.         this.data = null;
8.         this.next = null;
9.     }
10.
11.     public ListNode(Object data) {

```

```

12.         this.data = data;
13.         this.next = null;
14.     }
15.
16.     public ListNode(Object data, ListNode next) {
17.         this.data = data;
18.         this.next = next;
19.     }
20.
21.     public String toString() {
22.         return "ListNode(" + data.toString() + ")";
23.     }
24. }

```

该 `ListNode` 对象具有两个实例变量：`data` 是某种类型的 `Object` 的引用，并且 `next` 是列表中下一个节点的引用。在列表中的最后一个节点中，按照惯例，`next` 是 `null`。

`ListNode` 提供了几个构造函数，可以让你为 `data` 和 `next` 提供值，或将它们初始化为默认值，`null`。

你可以将每个 `ListNode` 看作具有单个元素的列表，但更通常，列表可以包含任意数量的节点。有几种方法可以制作新的列表。一个简单的选项是，创建一组 `ListNode` 对象，如下所示：

```

1. ListNode node1 = new ListNode(1);
2. ListNode node2 = new ListNode(2);
3. ListNode node3 = new ListNode(3);

```

之后将其链接到一起，像这样：

```

1. node1.next = node2;
2. node2.next = node3;
3. node3.next = null;

```

或者，你可以创建一个节点并将其链接在一起。例如，如果要在列表开头添加一个新节点，可以这样做：

```

1. ListNode node0 = new ListNode(0, node1);

```

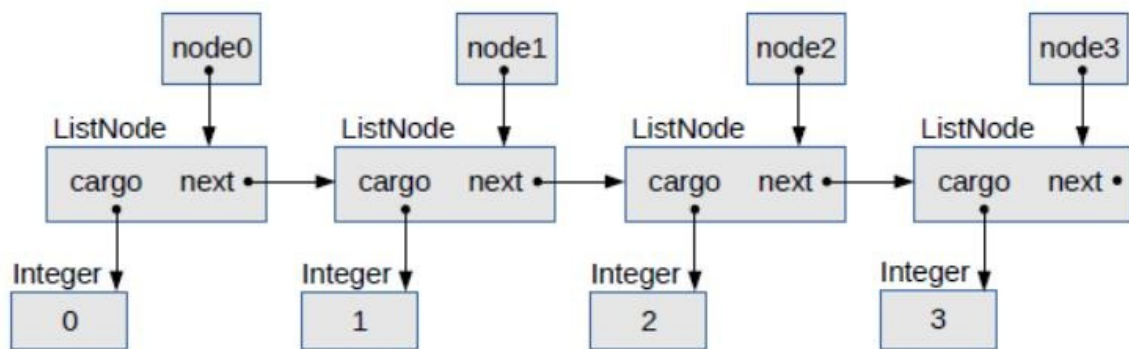


图 3.1 链表的对象图

图 3.1 是一个对象图，展示了这些变量及其引用的对象。在对象图中，变量的名称出现在框内，箭头显示它们所引用的内容。对象及其类型（如 `ListNode` 和 `Integer`）出现在框外面。

## 3.5 练习 3

这本书的仓库中，你会找到你需要用于这个练习的源代码：

- `MyLinkedList.java` 包含 `List` 接口的部分实现，使用链表存储元素。
- `MyLinkedListTest.java` 包含用于 `MyLinkedList` 的 JUnit 测试。

运行 `ant MyArrayList` 来运行 `MyArrayList.java`，其中包含几个简单的测试。

然后可以运行 `ant MyArrayListTest` 来运行 JUnit 测试。其中几个应该失败。如果你检查源代码，你会发现三条 `TODO` 注释，表示你应该填充的方法。

在开始之前，让我们来看看一些代码。以下是 `MyLinkedList` 的实例变量和构造函数：

```

1. public class MyLinkedList<E> implements List<E> {
2.
3.     private int size;           // keeps track of the number of elements
4.     private Node head;         // reference to the first node
5.
6.     public MyLinkedList() {
7.         head = null;
8.         size = 0;
9.     }
10. }
  
```

如注释所示，`size` 跟踪 `MyLinkedList` 有多少元素；`head` 是列表中第一个 `Node` 的引用，或者如果列表为空则为 `null`。

存储元素数量不是必需的，并且一般来说，保留冗余信息是有风险的，因为如果没有正确更新，就有机会产生错误。它还需要一点点额外的空间。

但是如果显式存储 `size`，我们可以实现常数时间的 `size` 方法；否则，我们必须遍历列表并对元素进行计数，这需要线性时间。

因为我们显式存储 `size` 明确地存储，每次添加或删除一个元素时，我们都要更新它，这样一来，这些方法就会减慢，但是它不会改变它们的增长级别，所以很值得。

构造函数将 `head` 设为 `null`，表示空列表，并将 `size` 设为 `0`。

这个类使用类型参数 `E` 作为元素的类型。如果你不熟悉类型参数，可能需要阅读本教程：<http://thinkdast.com/types>。

类型参数也出现在 `Node` 的定义中，嵌套在 `MyLinkedList` 里面：

```
1. private class Node {
2.     public E data;
3.     public Node next;
4.
5.     public Node(E data, Node next) {
6.         this.data = data;
7.         this.next = next;
8.     }
9. }
```

除了这个，`Node` 类似于上面的 `ListNode`。

最后，这是我的 `add` 的实现：

```
1. public boolean add(E element) {
2.     if (head == null) {
3.         head = new Node(element);
4.     } else {
5.         Node node = head;
6.         // loop until the last node
7.         for ( ; node.next != null; node = node.next) {}
8.         node.next = new Node(element);
9.     }
10.    size++;
11.    return true;
12. }
```

此示例演示了你需要的两种解决方案：

对于许多方法，作为特殊情况，我们必须处理列表的第一个元素。在这个例子中，如果我们向列表添

加列表第一个元素，我们必须修改 `head`。否则，我们遍历列表，找到末尾，并添加新节点。此方法展示了，如何使用 `for` 循环遍历列表中的节点。在你的解决方案中，你可能会在此循环中写出几个变体。注意，我们必须在循环之前声明 `node`，以便我们可以在循环之后访问它。

现在轮到你了。填充 `indexOf` 的主体。像往常一样，你应该阅读文档，位于 <http://thinkdast.com/listindexOf>，所以你知道应该做什么。特别要注意它应该如何处理 `null`。

与上一个练习一样，我提供了一个辅助方法 `equals`，它将数组中的一个元素与目标值进行比较，并检查它们是否相等，并正确处理 `null`。这个方法是私有的，因为它在这个类中使用，但它不是 `List` 接口的一部分。

完成后，再次运行测试；`testIndexOf`，以及依赖于它的其他测试现在应该通过。

接下来，你应该填充双参数版本的 `add`，它使用索引并将新值存储在给定索引处。再次阅读 <http://thinkdast.com/listadd> 上的文档，编写一个实现，并运行测试进行确认。

最后一个：填写 `remove` 的主体。文档在这里：<http://thinkdast.com/listrem>。当你完成它时，所有的测试都应该通过。

一旦你的实现能够工作，将它与仓库 `solution` 目录中的版本比较。

## 3.6 垃圾回收的注解

在 `MyArrayList` 以前的练习中，如果需要，数字会增长，但它不会缩小。该数组从不收集垃圾，并且在列表本身被销毁之前，元素不会收集垃圾。

链表实现的一个优点是，当元素被删除时它会缩小，并且未使用的节点可以立即被垃圾回收。

这是我的实现的 `clear` 方法：

```
1. public void clear() {
2.     head = null;
3.     size = 0;
4. }
```

当我们将 `head` 设为 `null` 时，我们删除第一个 `Node` 的引用。如果没有其他 `Node` 的引用（不应该有），它将被垃圾收集。这个时候，第二个 `Node` 引用被删除，所以它也被垃圾收集。此过程一直持续到所有节点都被收集。

那么我们应该如何划分 `clear`？该方法本身包含两个常数时间的操作，所以它看起来像是常数时间。但是当你调用它时，你将使垃圾收集器做一些工作，它与元素数成正比。所以也许我们应该将其认为是线性的！

这是一个有时被称为性能 bug 的例子：一个程序做了正确的事情，在这种意义上它是正确的，但它不属于我们预期的增长级别。在像 Java 这样的语言中，它在背后做了大量工作的，例如垃圾收集，这种 bug 可能很难找到。

## 第四章 LinkedList

- 第四章 `LinkedList`
  - 4.1 `MyLinkedList` 方法的划分
  - 4.2 `MyArrayList` 和 `MyLinkedList` 的对比
  - 4.3 性能分析
  - 4.4 解释结果
  - 4.5 练习 4

## 第四章

`LinkedList`原文: *Chapter 4 LinkedList*

译者: 飞龙

协议: *CC BY-NC-SA 4.0*

自豪地采用谷歌翻译

这一章展示了上一个练习的解法，并继续讨论算法分析。

## 4.1 `MyLinkedList` 方法的划分

我的 `indexOf` 实现在下面。在阅读说明之前，请阅读它，看看你是否可以确定其增长级别。

```
1. public int indexOf(Object target) {
2.     Node node = head;
3.     for (int i=0; i<size; i++) {
4.         if (equals(target, node.data)) {
5.             return i;
6.         }
7.         node = node.next;
8.     }
9.     return -1;
10. }
```

最初 `node` 为 `head` 的副本，所以他们都指向相同的 `Node`。循环变量 `i` 从 `0` 计数到 `size-1`。每次在循环中，我们都用 `equals` 来看看我们是否找到了目标。如果是这样，我们立即返回 `i`。否则我们移动到列表中的下一个 `Node`。

通常会检查以确保下一个 `Node` 不是 `null`，但在这里，它是安全的，因为当我们到达列表的末尾时循环结束（假设与列表中 `size` 与实际节点数量一致）。

如果我们走完了循环而没有找到目标，我们返回 `-1`。

那么这种方法的增长级别是什么？

- 每次在循环中，我们调用了 `equals`，这是一个常数时间（它可能取决于 `target` 或 `data` 大小，但不取决于列表的大小）。循环中的其他操作也是常数时间。
- 循环可能运行 `n` 次，因为在更糟的情况下，我们可能必须遍历整个列表。

所以这个方法的运行时间与列表的长度成正比。

接下来，这里是我的双参数 `add` 方法的实现。同样，你应该尝试对其进行划分，然后再阅读说明。

```
1. public void add(int index, E element) {
2.     if (index == 0) {
3.         head = new Node(element, head);
4.     } else {
5.         Node node = getNode(index-1);
6.         node.next = new Node(element, node.next);
7.     }
8.     size++;
9. }
```

如果 `index==0`，我们在开始添加新的 `Node`，所以我们把它当作特殊情况。否则，我们必须遍历列表来查找 `index-1` 处的元素。我们使用辅助方法 `getNode`：

```
1. private Node getNode(int index) {
2.     if (index < 0 || index >= size) {
3.         throw new IndexOutOfBoundsException();
4.     }
5.     Node node = head;
6.     for (int i=0; i<index; i++) {
7.         node = node.next;
8.     }
9.     return node;
10. }
```

`getNode` 检查 `index` 是否超出范围；如果是这样，它会抛出异常。否则，它遍历列表并返回所请求的节点。

我们回到 `add`，一旦我们找到合适的 `Node`，我创建新的 `Node`，并把它插到 `node` 和 `node.next` 之间。你可能会发现，绘制此操作的图表有助于确保你了解此操作。

那么，`add` 的增长级别什么呢？

- `getNode` 类似 `indexOf`，出于同样的原因也是线性的。
- 在 `add` 中，`getNode` 前后的一切都是常数时间。



所以放在一起，`add` 是线性的。

最后，我们来看看 `remove`：

```

1. public E remove(int index) {
2.     E element = get(index);
3.     if (index == 0) {
4.         head = head.next;
5.     } else {
6.         Node node = getNode(index-1);
7.         node.next = node.next.next;
8.     }
9.     size--;
10.    return element;
11. }

```

`remove` 使用了 `get` 查找和存储 `index` 处的元素。然后它删除包含它的 `Node`。

如果 `index==0`，我们再次处理这个特殊情况。否则我们找到节点 `index-1` 并进行修改，来跳过 `node.next` 并直接链接到 `node.next.next`。这有效地从列表中删除 `node.next`，它可以被垃圾回收。

最后，我们减少 `size` 并返回我们在开始时检索的元素。

那么，`remove` 的增长级别是什么呢？`remove` 中的一切是常数时间，除了 `get` 和 `getNode`，它们是线性的。因此，`remove` 是线性的。

当人们看到两个线性操作时，他们有时会认为结果是平方的，但是只有一个操作嵌套在另一个操作中才适用。如果你在一个操作之后调用另一个，运行时间会相加。如果它们都是  $O(n)$  的，则总和也是  $O(n)$  的。

## 4.2 `MyArrayList` 和 `MyLinkedList` 的对比

下表总结了 `MyArrayList` 和 `MyLinkedList` 之间的差异，其中 `1` 表示  $O(1)$  或常数时间，和 `n` 表示  $O(n)$  或线性。

	<code>MyArrayList</code>	<code>MyLinkedList</code>
<code>add</code> (末尾)	1	n
<code>add</code> (开头)	n	1
<code>add</code> (一般)	n	n
<code>get</code> / <code>set</code>	1	n
<code>indexOf</code> / <code>lastIndexOf</code>	n	n

<code>isEmpty</code> / <code>size</code>	1	1
<code>remove</code> (末尾)	1	n
<code>remove</code> (开头)	n	1
<code>remove</code> (一般)	n	n

- `MyArrayList` 的优势操作是，插入末尾，移除末尾，获取和设置。
- `MyLinkedList` 的优势操作是，插入开头，以及移动开头。

对于其他操作，这两个实现方式的增长级别相同。

哪个实现更好？这取决于你最有可能使用哪些操作。这就是为什么 Java 提供了多个实现，因为它取决于你。

## 4.3 性能分析

对于下一个练习，我提供了一个 `Profiler` 类，它包含代码，使用一系列问题规模运行方法，测量运行时间和绘制结果。

你将使用 `Profiler`，为 Java 的实现 `ArrayList` 和 `LinkedList`，划分 `add` 方法的性能。

以下是一个示例，展示了如何使用分析器：

```

1. public static void profileArrayListAddEnd() {
2.     Timeable timeable = new Timeable() {
3.         List<String> list;
4.
5.         public void setup(int n) {
6.             list = new ArrayList<String>();
7.         }
8.
9.         public void timeMe(int n) {
10.            for (int i=0; i<n; i++) {
11.                list.add("a string");
12.            }
13.        }
14.    };
15.
16.    String title = "ArrayList add end";
17.    Profiler profiler = new Profiler(title, timeable);
18.
19.    int startN = 4000;
20.    int endMillis = 1000;
21.    XYSeries series = profiler.timingLoop(startN, endMillis);
22.    profiler.plotResults(series);
23. }
```

此方法测量在 `ArrayList` 上运行 `add` 所需的时间，它向末尾添加新元素。我将解释代码，然后展示结果。

为了使用 `Profiler`，我们需要创建一个 `Timeable`，它提供两个方法：`setup` 和 `timeMe`。`setup` 方法执行在启动计时之前所需的任何工作；这里它会创建一个空列表。然后 `timeMe` 执行我们试图测量的任何操作；这里它将 `n` 个元素添加到列表中。

创建 `timeable` 的代码是一个匿名类，用于定义 `Timeable` 接口的新实现，并同时创建新类的实例。如果你不熟悉匿名类，你可以阅读这里：<http://thinkdast.com/anonclass>。

但是下一次练习不需要太多的知识；即使你不喜欢匿名类，也可以复制和修改示例代码。

下一步是创建 `Profiler` 对象，传递 `Timeable` 对象和标题作为参数。

`Profiler` 提供了 `timingLoop`，它使用存储为实例变量的 `Timeable`。它多次调用 `Timeable` 对象上的 `timeMe` 方法，使用一系列的 `n` 值。`timingLoop` 接受两个参数：

- `startN` 是 `n` 的值，计时循环应该从它开始。
- `endMillis` 是以毫秒为单位的阈值。随着 `timingLoop` 增加问题规模，运行时间增加；当运行时间超过此阈值时，`timingLoop` 停止。

当你运行实验时，你可能需要调整这些参数。如果 `startN` 太低，运行时间可能太短，无法准确测量。如果 `endMillis` 太低，你可能无法获得足够的数​​据，来查看问题规模和运行时间之间的明确关系。

这段代码位于 `ProfileListAdd.java`，你将在下一个练习中运行它。当我运行它时，我得到这个输出：

```
1. 4000, 3
2. 8000, 0
3. 16000, 1
4. 32000, 2
5. 64000, 3
6. 128000, 6
7. 256000, 18
8. 512000, 30
9. 1024000, 88
10. 2048000, 185
11. 4096000, 242
12. 8192000, 544
13. 16384000, 1325
```

第一列是问题规模，`n`；第二列是以毫秒为单位的运行时间。前几个测量非常嘈杂；最好将 `startN` 设置在 `64000` 左右。

`timingLoop` 的结果是包含此数据的 `XYSeries`。如果你将这个序列传给 `plotResults`，它会产生一个如图 4.1 所示的图形。

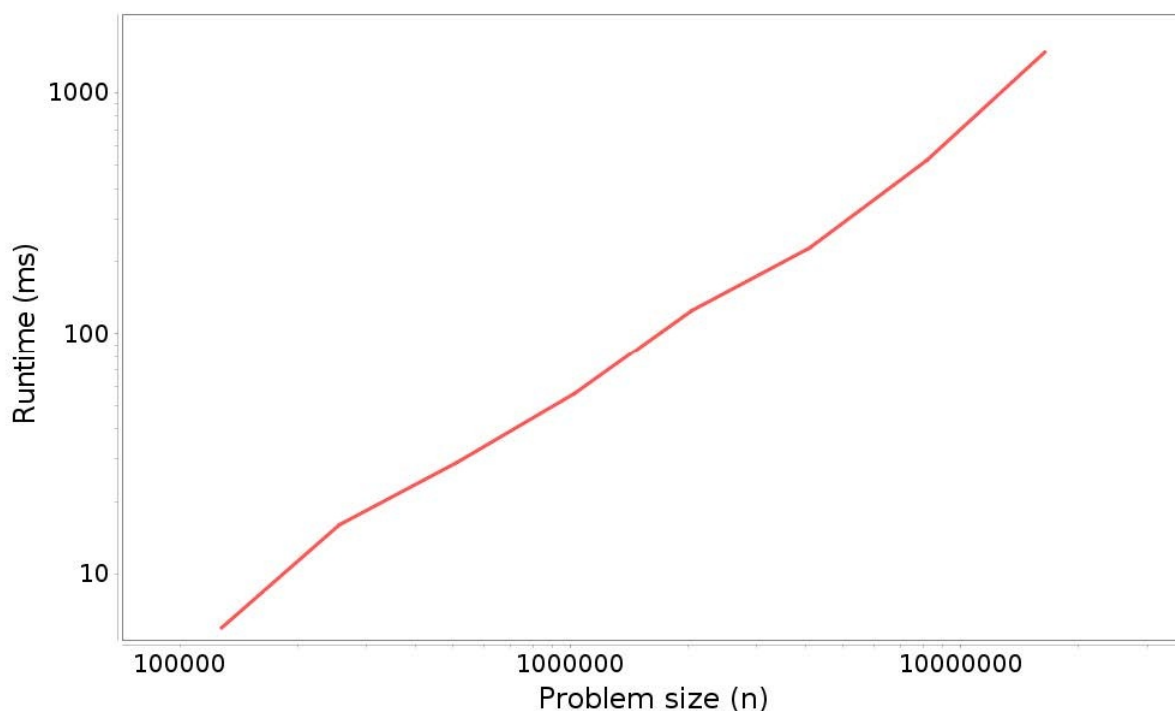


图 4.1 分析结果：将 `n` 个元素添加到 `ArrayList` 末尾的运行时间与问题规模。

下一节解释了如何解释它。

## 4.4 解释结果

基于我们对 `ArrayList` 工作方式的理解，我们期望，在添加元素到最后时，`add` 方法需要常数时间。所以添加 `n` 个元素的总时间应该是线性的。

为了测试这个理论，我们可以绘制总运行时间和问题规模，我们应该看到一条直线，至少对于大到足以准确测量的问题规模。在数学上，我们可以为这条直线编写一个函数：

```
1. runtime = a + b * n
```

其中 `a` 是线的截距，`b` 是斜率。

另一方面，如果 `add` 是线性的，则 `n` 次添加的总时间将是平方。如果我们绘制运行时间与问题规模，我们预计会看到抛物线。或者在数学上，像：

```
1. runtime = a + b * n + c * n ** 2
```

有了完美的数据，我们可能能够分辨直线和抛物线之间的区别，但如果测量结果很嘈杂，可能很难辨别。解释嘈杂的测量值的更好方法是，在重对数刻度上绘制的运行时间和问题规模。

为什么？我们假设运行时间与  $n^k$  成正比，但是我们不知道指数  $k$  是什么。我们可以将关系写成这样：

```
1. runtime = a + b * n + ... + c * n ** k
```

对于  $n$  的较大值，最大指数项是最重要的，因此：

```
1. runtime ≈ c * n ** k
```

其中  $\approx$  意思是“大致相等”。现在，如果我们对这个方程的两边取对数：

```
1. log(runtime) ≈ log(c) + k * log(n)
```

这个方程式意味着，如果我们在重对数合度上绘制运行时间与  $n$ ，我们预计看到一条直线，截距为  $\log(c)$ ，斜率为  $k$ 。我们不太在意截距，但斜率表示增长级别：如果  $k = 1$ ，算法是线性的；如果  $k = 2$ ，则为平方的。

看上一节中的数字，你可以通过眼睛来估计斜率。但是当你调用 `plotResults` 它时，会计算数据的最小二乘拟合并打印估计的斜率。在这个例子中：

```
1. Estimated slope = 1.06194352346708
```

它接近  $1$ ；并且这表明  $n$  次添加的总时间是线性的，所以每个添加是常数时间，像预期的那样。

其中重要的一点：如果你在图形看到这样的直线，这并不意味着该算法是线性的。如果对于任何指数  $k$ ，运行时间与  $n^k$  成正比，我们预计看到斜率为  $k$  的直线。如果斜率接近  $1$ ，则表明算法是线性的。如果接近  $2$ ，它可能是平方的。

## 4.5 练习 4

在本书的仓库中，你将找到此练习所需的源文件：

- `Profiler.java` 包含上述 `Profiler` 类的实现。你会使用这个类，但你不必知道它如何工作。但可以随时阅读源码。
- `ProfileListAdd.java` 包含此练习的起始代码，包括上面的示例，它测量了 `ArrayList.add`。你将修改此文件来测量其他一些方法。

此外，在 `code` 目录中，你将找到 Ant 构建文件 `build.xml`。

运行 `ant ProfileListAdd` 来运行 `ProfileListAdd.java`。你应该得到类似图 4.1 的结果，但是你可能需要调整 `startN` 或 `endMillis`。估计的斜率应该接近 `1`，表明执行 `n` 个添加操作的所需时间与 `n` 成正比；也就是说，它是  $O(n)$  的。

在 `ProfileListAdd.java` 中，你会发现一个空的方法 `profileArrayListAddBeginning`。用测试 `ArrayList.add` 的代码填充这个方法的主体，总是把新元素放在开头。如果你以 `profileArrayListAddEnd` 的副本开始，你只需要进行一些更改。在 `main` 中添加一行来调用这个方法。

再次运行 `ant ProfileListAdd` 并解释结果。基于我们对 `ArrayList` 工作方式的理解，我们期望，每个添加操作是线性的，所以 `n` 次添加的总时间应该是平方的。如果是这样，在重对数刻度中，直线的估计斜率应该接近 `2`。是吗？

现在我们来将其与 `LinkedList` 比较。当我们把新元素放在开头，填充 `profileLinkedListAddBeginning` 并使用它划分 `LinkedList.add`。你期望什么性能？结果是否符合你的期望？

最后，填充 `profileLinkedListAddEnd` 的主体，使用它来划分 `LinkedList.add`。你期望什么性能？结果是否符合你的期望？

我将在下一章中展示结果并回答这些问题。

## 第五章 双链表

- 第五章 双链表
  - 5.1 性能分析结果
  - 5.2 分析 `LinkedList` 方法的性能
  - 5.3 `LinkedList` 的尾部添加
  - 5.4 双链表
  - 5.5 结构的选择

## 第五章 双链表

原文: [Chapter 5 Doubly-linked list](#)

译者: 飞龙

协议: [CC BY-NC-SA 4.0](#)

自豪地采用谷歌翻译

本章回顾了上一个练习的结果，并介绍了 `List` 接口的另一个实现，即双链表。

### 5.1 性能分析结果

在之前的练习中，我们使用了 `Profiler.java`，运行 `ArrayList` 和 `LinkedList` 的各种操作，它们具有一系列的问题规模。我们将运行时间与问题规模绘制在重对数比例尺上，并估计所得曲线的斜率，它表示运行时间和问题规模之间的关系的主要指数。

例如，当我们使用 `add` 方法将元素添加到 `ArrayList` 的末尾，我们发现，执行  $n$  次添加的总时间正比于  $n$ 。也就是说，估计的斜率接近 1。我们得出结论，执行  $n$  次添加是  $O(n)$  的，所以平均来说，单个添加的时间是常数时间，或者  $O(1)$ ，基于算法分析，这是我们的预期。

这个练习要求你填充 `profileArrayListAddBeginning` 的主体，它测试了，在 `ArrayList` 头部添加一个新的元素的性能。根据我们的分析，我们预计每个添加都是线性的，因为它必须将其他元素向右移动；所以我们预计， $n$  次添加是平方复杂度。

这是一个解决方案，你可以在仓库的 `solution` 目录中找到它。

```
1. public static void profileArrayListAddBeginning() {
2.     Timeable timeable = new Timeable() {
3.         List<String> list;
4.
5.         public void setup(int n) {
6.             list = new ArrayList<String>();
```

```

7.      }
8.
9.      public void timeMe(int n) {
10.         for (int i=0; i<n; i++) {
11.             list.add(0, "a string");
12.         }
13.     }
14. };
15. int startN = 4000;
16. int endMillis = 10000;
17. runProfiler("ArrayList add beginning", timeable, startN, endMillis);
18. }

```

这个方法几乎和 `profileArrayListAddEnd` 相同。唯一的区别在于 `timeMe`，它使用 `add` 的双参数版本，将新元素置于下标 `0` 处。同样，我们增加了 `endMillis`，来获取一个额外的数据点。

以下是时间结果（左侧是问题规模，右侧是运行时间，单位为毫秒）：

```

1. 4000, 14
2. 8000, 35
3. 16000, 150
4. 32000, 604
5. 64000, 2518
6. 128000, 11555

```

图 5.1 展示了运行时间和问题规模的图形。

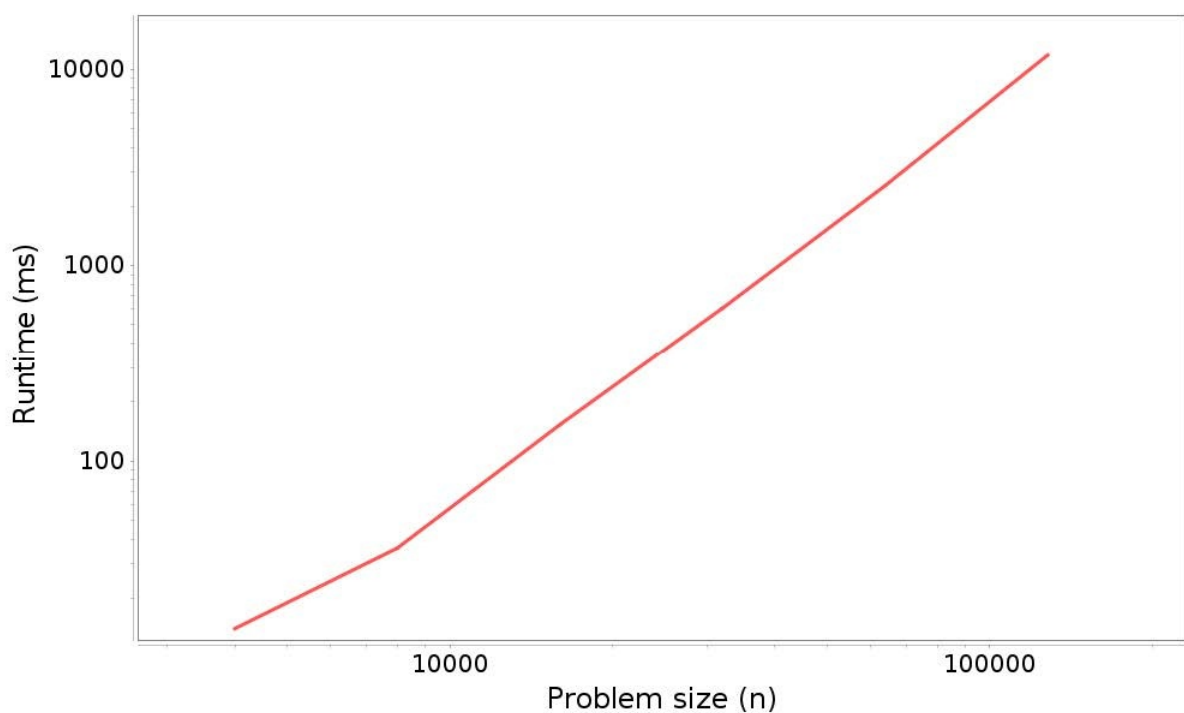




图 5.1: 分析结果: 在 `ArrayList` 开头添加 `n` 个元素的运行时间和问题规模

请记住, 该图上的直线并不意味着该算法是线性的。相反, 如果对于任何指数 `k`, 运行时间与 `n ** k` 成正比, 我们预计会看到斜率为 `k` 的直线。在这种情况下, 我们预计, `n` 次添加的总时间与 `n ** 2` 成正比, 所以我们预计会有一条斜率为 `2` 的直线。实际上, 估计的斜率是 `1.992`, 非常接近。恐怕假数据才能做得这么好。

## 5.2 分析 `LinkedList` 方法的性能

在以前的练习中, 你还分析了, 在 `LinkedList` 头部添加新元素的性能。根据我们的分析, 我们预计每个 `add` 都要花时间, 因为在一个链表中, 我们不必转移现有元素; 我们可以在头部添加一个新节点。所以我们预计 `n` 次添加的总时间是线性的。

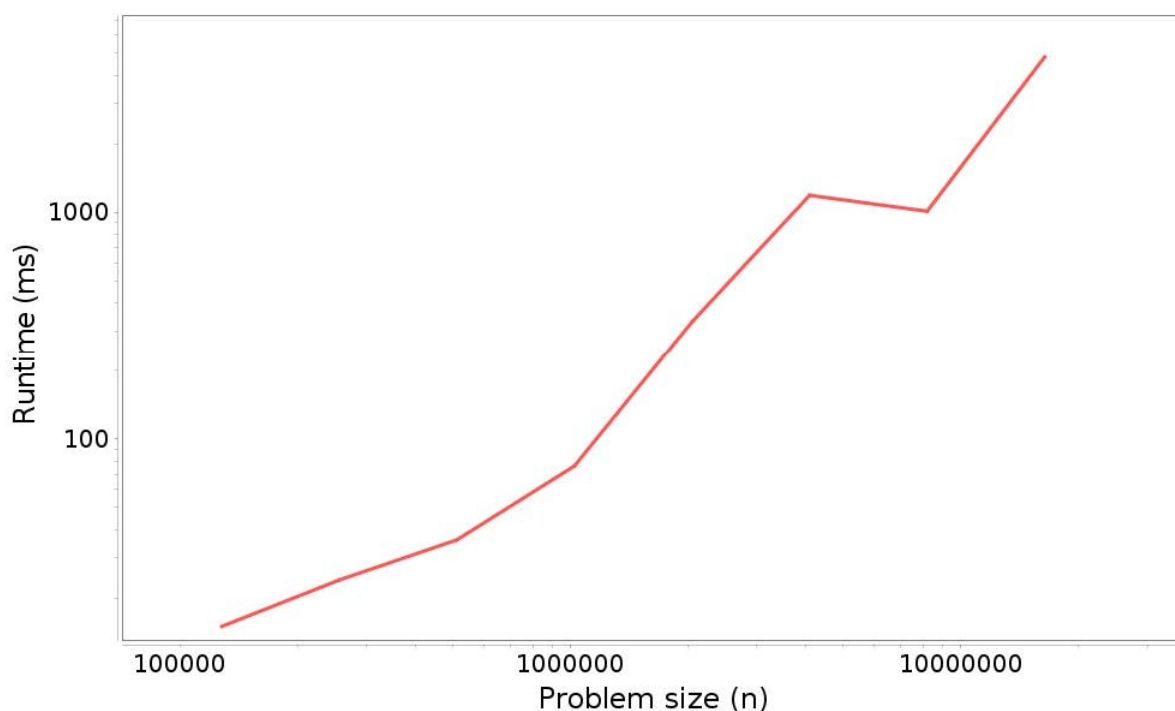
这是一个解决方案:

```
1. public static void profileLinkedListAddBeginning() {
2.     Timeable timeable = new Timeable() {
3.         List<String> list;
4.
5.         public void setup(int n) {
6.             list = new LinkedList<String>();
7.         }
8.
9.         public void timeMe(int n) {
10.            for (int i=0; i<n; i++) {
11.                list.add(0, "a string");
12.            }
13.        }
14.    };
15.    int startN = 128000;
16.    int endMillis = 2000;
17.    runProfiler("LinkedList add beginning", timeable, startN, endMillis);
18. }
```

我们只做了一些修改, 将 `ArrayList` 替换为 `LinkedList` 并调整 `startN` 和 `endMillis`, 来获得良好的数据范围。测量结果比上一批数据更加嘈杂; 结果如下:

```
1. 128000, 16
2. 256000, 19
3. 512000, 28
4. 1024000, 77
5. 2048000, 330
6. 4096000, 892
7. 8192000, 1047
8. 16384000, 4755
```

图 5.2 展示了这些结果的图形。

图 5.2：分析结果：在 `LinkedList` 开头添加 `n` 个元素的运行时间和问题规模

并不是一条很直的线，斜率也不是正好是 `1`，最小二乘拟合的斜率是 `1.23`。但是结果表明，`n` 次添加的总时间至少近似于 `O(n)`，所以每次添加都是常数时间。

## 5.3 `LinkedList` 的尾部添加

在开头添加元素是一种操作，我们期望 `LinkedList` 的速度快于 `ArrayList`。但是为了在末尾添加元素，我们预计 `LinkedList` 会变慢。在我的实现中，我们必须遍历整个列表来添加一个元素到最后，它是线性的。所以我们预计 `n` 次添加的总时间是二次的。

但是不是这样。以下是代码：

```

1. public static void profileLinkedListAddEnd() {
2.     Timeable timeable = new Timeable() {
3.         List<String> list;
4.
5.         public void setup(int n) {
6.             list = new LinkedList<String>();
7.         }
8.
9.         public void timeMe(int n) {

```

```
10.         for (int i=0; i<n; i++) {
11.             list.add("a string");
12.         }
13.     }
14. };
15.     int startN = 64000;
16.     int endMillis = 1000;
17.     runProfiler("LinkedList add end", timeable, startN, endMillis);
18. }
```

这里是结果：

```
1. 64000, 9
2. 128000, 9
3. 256000, 21
4. 512000, 24
5. 1024000, 78
6. 2048000, 235
7. 4096000, 851
8. 8192000, 950
9. 16384000, 6160
```

图 5.3 展示了这些结果的图形。

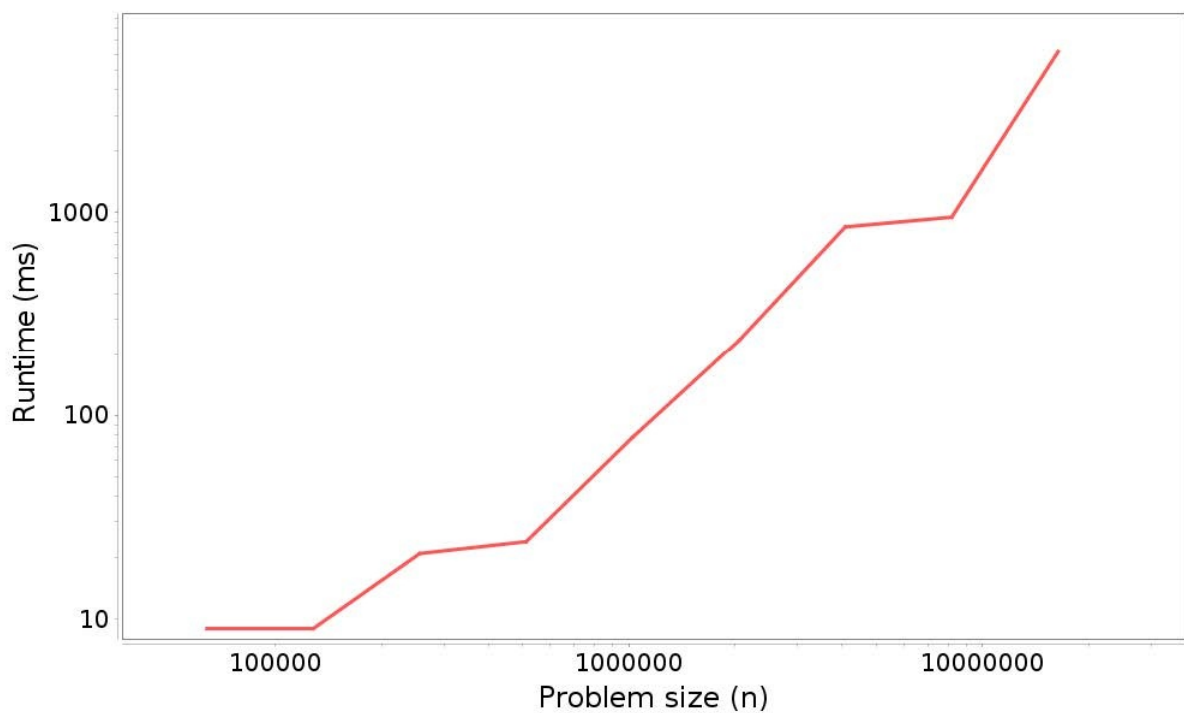


图 5.2：分析结果：在 `LinkedList` 末尾添加 `n` 个元素的运行时间和问题规模

同样，测量值很嘈杂，线不完全是直的，但估计的斜率为 `1.19`，接近于在头部添加元素，而并不非常接近 `2`，这是我们根据分析的预期。事实上，它接近 `1`，这表明在尾部添加元素是常数元素。这是怎么回事？

## 5.4 双链表

我的链表实现 `MyLinkedList`，使用单链表；也就是说，每个元素都包含下一个元素的链接，并且 `MyArrayList` 对象本身具有第一个节点的链接。

但是，如果你阅读 `LinkedList` 的文档，网址为 <http://thinkdast.com/linked>，它说：

`List` 和 `Deque` 接口的双链表实现。[...] 所有的操作都能像双向列表那样执行。索引该列表中的操作将从头或者尾遍历列表，使用更接近指定索引的那个。

如果你不熟悉双链表，你可以在 <http://thinkdast.com/doublelist> 上阅读更多相关信息，但简称为：

- 每个节点包含下一个节点的链接和上一个节点的链接。
- `LinkedList` 对象包含指向列表的第一个和最后一个元素的链接。

所以我们可以从列表的任意一端开始，并以任意方向遍历它。因此，我们可以在常数时间内，在列表的头部和末尾添加和删除元素！

下表总结了 `ArrayList`，`MyLinkedList`（单链表）和 `LinkedList`（双链表）的预期性能：

	<code>MyArrayList</code>	<code>MyLinkedList</code>	<code>LinkedList</code>
<code>add</code> （尾部）	1	n	1
<code>add</code> （头部）	n	1	1
<code>add</code> （一般）	n	n	n
<code>get</code> / <code>set</code>	1	n	n
<code>indexOf</code> / <code>lastIndexOf</code>	n	n	n
<code>isEmpty</code> / <code>size</code>	1	1	1
<code>remove</code> （尾部）	1	n	1
<code>remove</code> （头部）	n	1	1
<code>remove</code> （一般）	n	n	n

## 5.5 结构的选择

对于头部插入和删除，双链表的实现优于 `ArrayList`。对于尾部插入和删除，都是一样好。所以，`ArrayList` 唯一优势是 `get` 和 `set`，链表中它需要线性时间，即使是双链表。

如果你知道，你的应用程序的运行时间取决于 `get` 和 `set` 元素的所需时间，则 `ArrayList` 可能是更好的选择。如果运行时间取决于在开头或者末尾附加添加和删除元素，`LinkedList` 可能会更好。

但请记住，这些建议是基于大型问题的增长级别。还有其他因素要考虑：

- 如果这些操作不占用你应用的大部分运行时间 - 也就是说，如果你的应用程序花费大部分时间来执行其他操作 - 那么你对 `List` 实现的选择并不重要。
- 如果你正在处理的列表不是很大，你可能无法获得期望的性能。对于小型问题，二次算法可能比线性算法更快，或者线性可能比常数时间更快。而对于小型问题，差异可能并不重要。
- 另外，别忘了空间。到目前为止，我们专注于运行时间，但不同的实现需要不同的空间。在 `ArrayList` 中，这些元素并排存储在单个内存块中，所以浪费的空间很少，并且计算机硬件通常在连续的块上更快。在链表中，每个元素需要一个节点，带有一个或两个链接。链接占用空间（有时甚至超过数据！），并且节点分散在内存中，硬件效率可能不高。

总而言之，算法分析为数据结构的选择提供了一些指南，但只有：

- 你的应用的运行时间很重要，
- 你的应用的运行时间取决于你选择的数据结构，以及，
- 问题的规模足够大，增长级别实际上预测了哪个数据结构更好。

作为一名软件工程师，在较长的职业生涯中，你几乎不必考虑这种情况。

## 第六章 树的遍历

- 第六章 树的遍历
  - 6.1 搜索引擎
  - 6.2 解析 HTML
  - 6.3 使用 `jsoup`
  - 6.4 遍历 DOM
  - 6.5 深度优先搜索
  - 6.6 Java 中的栈
  - 6.7 迭代式 DFS

## 第六章 树的遍历

原文: [Chapter 6 Tree traversal](#)

译者: 飞龙

协议: [CC BY-NC-SA 4.0](#)

自豪地采用谷歌翻译

本章将介绍一个 Web 搜索引擎，我们将在本书其余部分开发它。我描述了搜索引擎的元素，并介绍了第一个应用程序，一个从维基百科下载和解析页面的 Web 爬行器。本章还介绍了深度优先搜索的递归实现，以及迭代实现，它使用 Java `Deque` 实现“后入先出”的栈。

### 6.1 搜索引擎

网络搜索引擎，像谷歌搜索或 Bing，接受一组“检索项”，并返回一个网页列表，它们和这些项相关（之后我将讨论“相关”是什么意思）。你可以在 <http://thinkdast.com/searcheng> 上阅读更多内容，但是我会解释你需要什么。

搜索引擎的基本组成部分是：

抓取：我们需要一个程序，可以下载网页，解析它，并提取文本和任何其他页面的链接。

索引：我们需要一个数据结构，可以查找一个检索项，并找到包含它的页面。

检索：我们需要一种方法，从索引中收集结果，并识别与检索项最相关的页面。

我们以爬虫开始。爬虫的目标是查找和下载一组网页。对于像 Google 和 Bing 这样的搜索引擎，目标是查找所有网页，但爬虫通常仅限于较小的域。在我们的例子中，我们只会读取维基百科的页面。

作为第一步，我们将构建一个读取维基百科页面的爬虫，找到第一个链接，并跟着链接来到另一个页面，然后重复。我们将使用这个爬虫来测试“到达哲学”的猜想，它是：

点击维基百科文章正文中的第一个小写的链接，然后对后续文章重复这个过程，通常最终会到达“哲学”的文章。

这个猜想在 <http://thinkdast.com/getphil> 中阐述，你可以阅读其历史。

测试这个猜想需要我们构建爬虫的基本部分，而不必爬取整个网络，甚至是所有维基百科。而且我觉得这个练习很有趣！

在几个章节之内，我们将处理索引器，然后我们将到达检索器。

## 6.2 解析 HTML

当你下载网页时，内容使用超文本标记语言（即 HTML）编写。例如，这里是一个最小的 HTML 文档：

```
1. <!DOCTYPE html>
2. <html>
3.   <head>
4.     <title>This is a title</title>
5.   </head>
6.   <body>
7.     <p>Hello world!</p>
8.   </body>
9. </html>
```

短语 `This is a title` 和 `Hello world!` 是实际出现在页面上的文字；其他元素是指示文本应如何显示的标签。

当我们的爬虫下载页面时，它需要解析 HTML，以便提取文本并找到链接。为此，我们将使用 `jsoup`，它是一个下载和解析 HTML 的开源 Java 库。

解析 HTML 的结果是文档对象模型（DOM）树，其中包含文档的元素，包括文本和标签。树是由节点组成的链接数据结构；节点表示文本，标签和其他文档元素。

节点之间的关系由文档的结构决定。在上面的例子中，第一个节点称为根，是 `<html>` 标签，它包含指向所包含两个节点的链接，`<head>` 和 `<body>`；这些节点是根节点的子节点。

`<head>` 节点有一个子节点，`<title>`，`<body>` 节点有一个子节点，`<p>`（代表“段落”）。图 6.1 以图形方式表示该树。

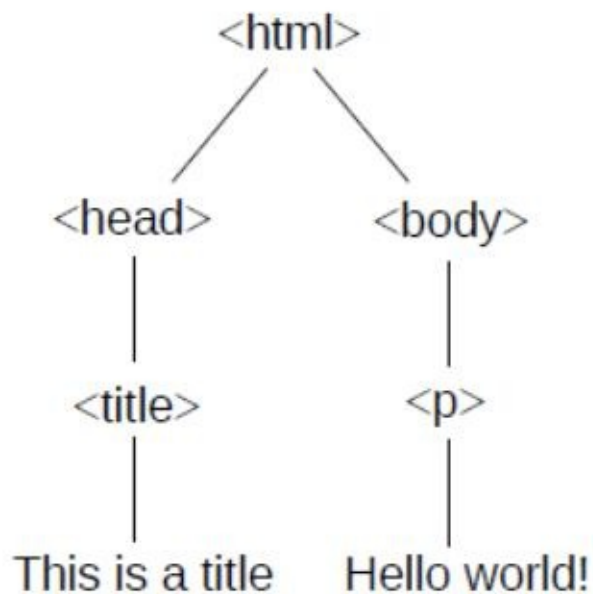


图 6.1 简单 HTML 页面的 DOM 树

每个节点包含其子节点的链接；此外，每个节点都包含其父节点的链接，所以任何节点都可以向上或向下浏览树。实际页面的 DOM 树通常比这个例子更复杂。

大多数网络浏览器提供了工具，用于检查你正在查看的页面的 DOM。在 Chrome 中，你可以右键单击网页的任何部分，然后从弹出的菜单中选择 `Inspect`（检查）。在 Firefox 中，你可以右键单击并从菜单中选择 `Inspect Element`（检查元素）。Safari 提供了一个名为 Web Inspector 的工具，你可以阅读 <http://thinkdast.com/safari>。对于 Internet Explorer，你可以阅读 <http://thinkdast.com/explorer> 上的说明。



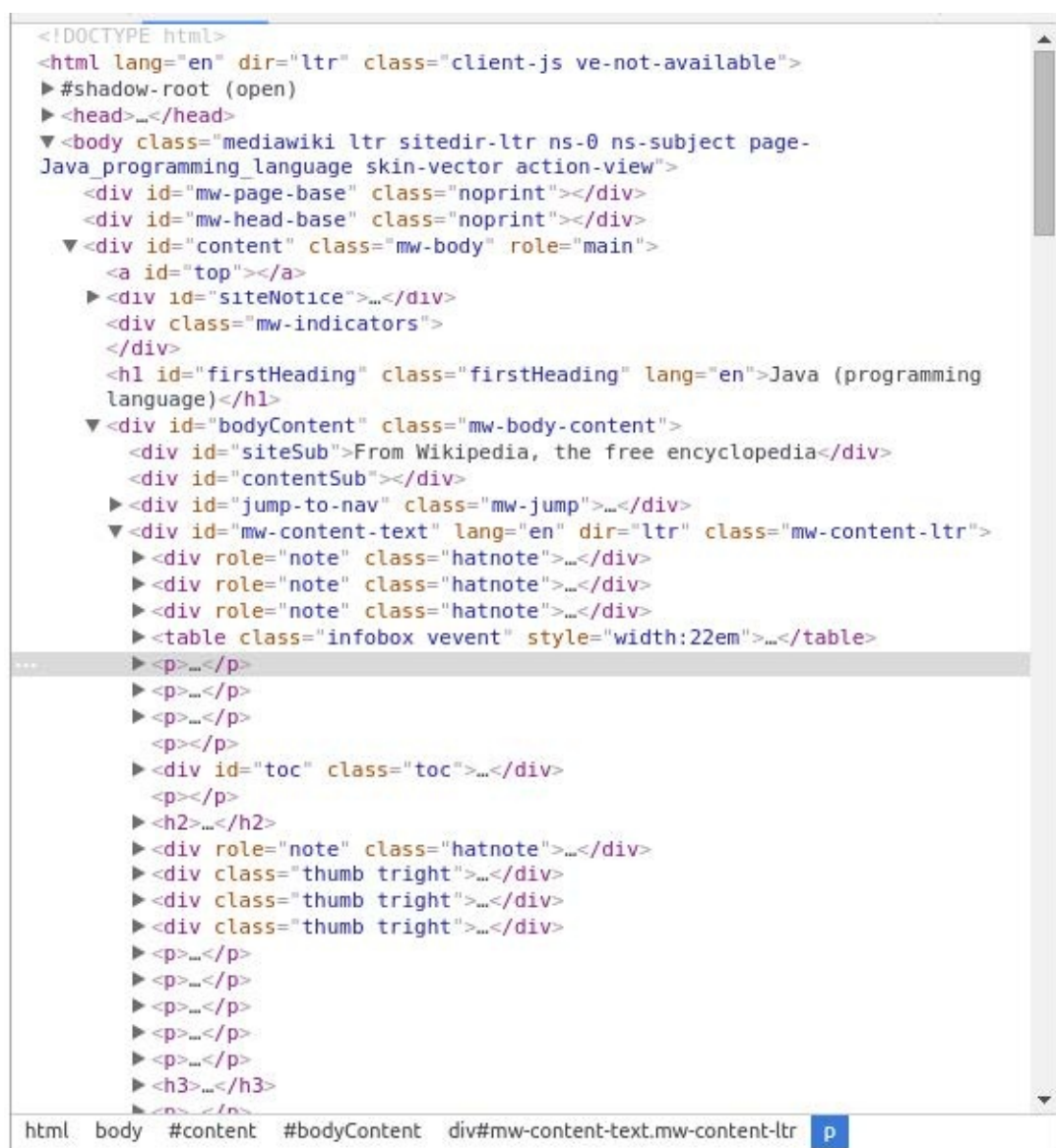


图 6.2: Chrome DOM 查看器的截图

图 6.2 展示了维基百科 Java 页面 (<http://thinkdast.com/java>) 的 DOM 截图。高亮的元素是文章正文的第一段，它包含在一个 `<div>` 元素中，带有 `id="mw-content-text"`。我们将使用这个元素 ID 来标识我们下载的每篇文章的正文。

## 6.3 使用 `jsoup`

`jsoup` 非常易于下载，和解析 Web 页面，以及访问 DOM 树。这里是一个例子：

```
1. String url = "http://en.wikipedia.org/wiki/Java_(programming_language)";
2.
3. // download and parse the document
4. Connection conn = Jsoup.connect(url);
```

```

5. Document doc = conn.get();
6.
7. // select the content text and pull out the paragraphs.
8. Element content = doc.getElementById("mw-content-text");
9. Elements paragraphs = content.select("p");

```

`Jsoup.connect` 接受 `String` 形式的 `url`，并连接 Web 服务器。`get` 方法下载 HTML，解析，并返回 `Document` 对象，他表示 DOM。

`Document` 提供了导航树和选择节点的方法。其实它提供了很多方法，可能会把人搞晕。此示例演示了两种选择节点的方式：

- `getElementById` 接受 `String` 并在树中搜索匹配 `id` 字段的元素。在这里，它选择节点 `<div id="mw-content-text" lang="en" dir="ltr" class="mw-content-ltr">`，它出现在每个维基页面上，来确定包含文章正文的 `<div>` 元素，而不是导航边栏和其他元素。`getElementById` 的返回值是一个 `Element` 对象，代表这个 `<div>`，并包含 `<div>` 中的元素作为后继节点。
- `select` 接受 `String`，遍历树，并返回与所有元素，它的标签与 `String` 匹配。在这个例子中，它返回所有 `content` 中的段落标签。返回值是一个 `Elements` 对象。

译者注：`select` 方法接受 CSS 选择器，不仅仅能按照标签选择。请见 <https://jsoup.org/apidocs/org/jsoup/select/Selector.html>。

在你继续之前，你应该仔细阅读这些类的文档，以便知道他们能做什么。最重要的类是 `Element`，`Elements` 和 `Node`，你可以阅读 <http://thinkdast.com/jsoupe1t>，<http://thinkdast.com/jsoupe1ts> 和 <http://thinkdast.com/jsoupn1de>。

`Node` 表示 DOM 树中的一个节点；有几个扩展 `Node` 的子类，其中包括 `Element`，`TextNode`，`DataNode`，和 `Comment`。`Elements` 是 `Element` 对象的 `Collection`。

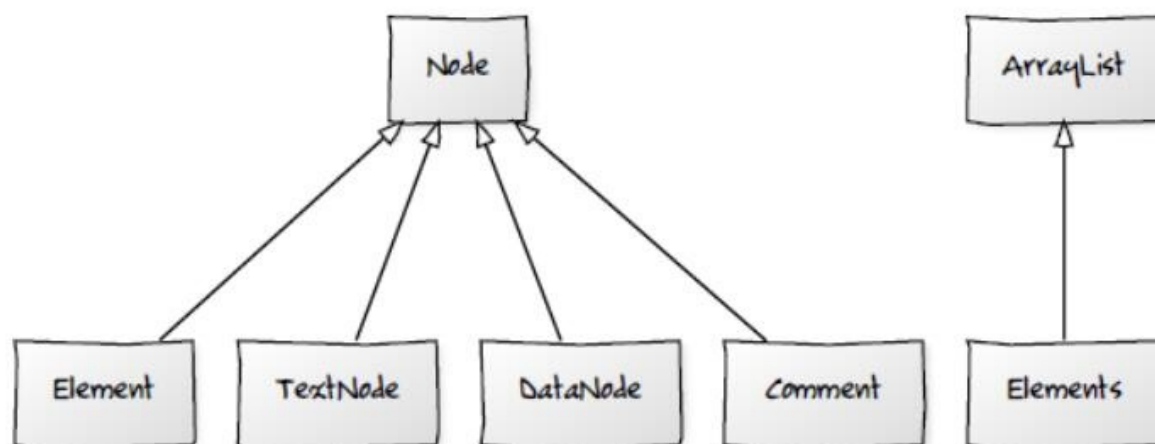


图 6.3：被选类的 UML 图，由 `jsoup` 提供。编辑：<http://yum1.me/edit/4bc1c919>

图 6.3 是一个 UML 图，展示了这些类之间的关系。在 UML 类图中，带有空心箭头的线表示一个类继承另一个类。例如，该图表示 `Elements` 继承 `ArrayList`。我们将在第 11.6 节中再次接触 UML 图。

## 6.4 遍历 DOM

为了使你变得更轻松，我提供了一个 `WikiNodeIterable` 类，可以让你遍历 DOM 树中的节点。以下是一个示例，展示如何使用它：

```
1. Elements paragraphs = content.select("p");
2. Element firstPara = paragraphs.get(0);
3.
4. Iterable<Node> iter = new WikiNodeIterable(firstPara);
5. for (Node node: iter) {
6.     if (node instanceof TextNode) {
7.         System.out.print(node);
8.     }
9. }
```

这个例子紧接着上一个例子。它选择 `paragraphs` 中的第一个段落，然后创建一个 `WikiNodeIterable`，它实现 `Iterable<Node>`。 `WikiNodeIterable` 执行“深度优先搜索”，它按照它们将出现在页面上的顺序产生节点。

在这个例子中，仅当 `Node` 是 `TextNode` 时，我们打印它，并忽略其他类型的 `Node`，特别是代表标签的 `Element` 对象。结果是没有任何标记的 HTML 段落的纯文本。输出为：

```
1. Java is a general-purpose computer programming language that is concurrent, class-based, object-
   oriented, [13] and specifically designed ...
```

Java 是一种通用的计算机编程语言，它是并发的，基于类的，面向对象的，[13] 和特地设计的...

## 6.5 深度优先搜索

有几种方式可以合理地遍历一个树，每个都有不同的应用。我们从“深度优先搜索”（DFS）开始。DFS 从树的根节点开始，并选择第一个子节点。如果子节点有子节点，则再次选择第一个子节点。当它到达没有子节点的节点时，它回溯，沿树向上移动到父节点，在那里它选择下一个子节点，如果有的话；否则它会再次回溯。当它探索了根节点的最后一个子节点，就完成了。

有两种常用的方式来实现 DFS，递归和迭代。递归实现简单优雅：

```
1. private static void recursiveDFS(Node node) {
2.     if (node instanceof TextNode) {
```

```

3.         System.out.print(node);
4.     }
5.     for (Node child: node.childNodes()) {
6.         recursiveDFS(child);
7.     }
8. }

```

这个方法对树中的每一个 `Node` 调用，从根节点开始。如果 `Node` 是一个 `TextNode`，它打印其内容。如果 `Node` 有任何子节点，它会按顺序在每一个子节点上调用 `recursiveDFS`。

在这个例子中，我们在遍历子节点之前打印每个 `TextNode` 的内容，所以这是一个“前序”遍历的例子。你可以在 <http://thinkdast.com/treetrav> 上了解“前序”，“后序”和“中序”遍历。对于此应用程序，遍历顺序并不重要。

通过进行递归调用，`recursiveDFS` 使用调用栈 (<http://thinkdast.com/callstack>) 来跟踪子节点并以正确的顺序处理它们。作为替代，我们可以使用栈数据结构自己跟踪节点；如果我们这样做，我们可以避免递归并迭代遍历树。

## 6.6 Java 中的栈

在我解释 DFS 的迭代版本之前，我将解释栈数据结构。我们将从栈的一般概念开始，我将使用小写 `s` 指代“栈”。然后我们将讨论两个 Java `interfaces`，它们定义了栈的方法：`Stack` 和 `Deque`。

栈是与列表类似的数据结构：它是维护元素顺序的集合。栈和列表之间的主要区别是栈提供的方法较少。在通常的惯例中，它提供：

- `push`：它将一个元素添加到栈顶。
- `pop`：它从栈中删除并返回最顶部的元素。
- `peek`：它返回最顶部的元素而不修改栈。
- `isEmpty`：表示栈是否为空。

因为 `pop` 总是返回最顶部的元素，栈也称为 LIFO，代表“后入先出”。栈的替代品是“队列”，它返回的元素顺序和添加顺序相同；即“先入先出 (FIFO)”。

为什么栈和队列是有用的，可能不是很明显：它们不提供任何列表没有的功能；实际上它们提供的功能更少。那么为什么不使用列表的一切？有两个原因：

- 如果你将自己限制于一小部分方法 - 也就是小型 API - 你的代码将更加易读，更不容易出错。例如，如果使用列表来表示栈，则可能会以错误的顺序删除元素。使用栈 API，这种错误在字面上是不可能的。避免错误的最佳方法是使它们不可能。
- 如果一个数据结构提供了小型 API，那么它更容易实现。例如，实现栈的简单方法是单链表。当我们压入一个元素时，我们将它添加到列表的开头；当我们弹出一个元素时，我们在开头删除它。对于链表，在开头添加和删除是常数时间的操作，因此这个实现是高效的。相反，大型 API

更难实现高效。

为了在 Java 中实现栈，你有三个选项：

- 继续使用 `ArrayList` 或 `LinkedList`。如果使用 `ArrayList`，请务必从最后添加和删除，这是一个常数时间的操作。并且小心不要在错误的地方添加元素，或以错误的顺序删除它们。
- Java 提供了一个 `Stack` 类，它提供了一组标准的栈方法。但是这个类是 Java 的一个旧部分：它与 Java 集合框架不兼容，后者之后才出现。
- 最好的选择可能是使用 `Deque` 接口的一个实现，如 `ArrayDeque`。

`Deque` 代表“双向队列”；它应该被发音为“deck”，但有些人叫它“deek”。在 Java 中，`Deque` 接口提供 `push`，`pop`，`peek` 和 `isEmpty`，因此你可以将 `Deque` 用作栈。它提供了其他方法，你可以阅读 <http://thinkdast.com/deque>，但现在我们不会使用它们。

## 6.7 迭代式 DFS

这里是 DFS 的迭代版本，它使用 `ArrayDeque` 来表示 `Node` 对象的栈。

```

1. private static void iterativeDFS(Node root) {
2.     Deque<Node> stack = new ArrayDeque<Node>();
3.     stack.push(root);
4.
5.     while (!stack.isEmpty()) {
6.         Node node = stack.pop();
7.         if (node instanceof TextNode) {
8.             System.out.print(node);
9.         }
10.
11.        List<Node> nodes = new ArrayList<Node>(node.childNodes());
12.        Collections.reverse(nodes);
13.
14.        for (Node child: nodes) {
15.            stack.push(child);
16.        }
17.    }
18. }
```

参数 `root` 是我们想要遍历的树的根节点，所以我们首先创建栈并将根节点压入它。

循环持续到栈为空。每次迭代，它会从栈中弹出 `Node`。如果它得到 `TextNode`，它打印内容。然后它把子节点们压栈。为了以正确的顺序处理子节点，我们必须以相反的顺序将它们压栈；我们通过将子节点复制成一个 `ArrayList`，原地反转元素，然后遍历反转的 `ArrayList`。

DFS 的迭代版本的一个优点是，更容易实现为 Java `Iterator`；你会在下一章看到如何实现。

但是首先，有一个 `Deque` 接口的最后的注意事项：除了 `ArrayDeque`，Java 提供另一个 `Deque` 的实现，我们的老朋友 `LinkedList`。`LinkedList` 实现两个接口，`List` 和 `Deque`（还有 `Queue`）。你得到哪个接口，取决于你如何使用它。例如，如果将 `LinkedList` 对象赋给 `Deque` 变量，如下所示：

```
1. Deque<Node> deque = new LinkedList<Node>();
```

你可以使用 `Deque` 接口中的方法，但不是所有 `List` 中的方法。如果你将其赋给 `List` 变量，像这样：

```
1. List<Node> deque = new LinkedList<Node>();
```

你可以使用 `List` 接口中的方法，但不是所有 `Deque` 中的方法。并且如果像这样赋值：

```
1. LinkedList<Node> deque = new LinkedList<Node>();
```

你可以使用所有方法，但是混合了来自不同接口的方法。你的代码会更不可读，并且更易于出错。

## 第七章 到达哲学

- 第七章 到达哲学
  - 7.1 起步
  - 7.2 可迭代对象和迭代器
  - 7.3 WikiFetcher
  - 7.4 练习 5

## 第七章 到达哲学

原文: *Chapter 7 Getting to Philosophy*

译者: 飞龙

协议: *CC BY-NC-SA 4.0*

自豪地采用谷歌翻译

本章的目标是开发一个 Web 爬虫，它测试了第 6.1 节中提到的“到达哲学”猜想。

### 7.1 起步

在本书的仓库中，你将找到一些帮助你起步的代码：

- WikiNodeExample.java 包含前一章的代码，展示了 DOM 树中深度优先搜索（DFS）的递归和迭代实现。
- WikiNodeIterable.java 包含 Iterable 类，用于遍历 DOM 树。我将在下一节中解释这段代码。
- WikiFetcher.java 包含一个工具类，使用 jsoup 从维基百科下载页面。为了帮助你遵守维基百科的服务条款，此类限制了你下载页面的速度；如果你每秒请求许多页，在下载下一页之前会休眠一段时间。
- WikiPhilosophy.java 包含你为此练习编写的代码的大纲。我们将在下面进行说明。

你还会发现 Ant 构建文件 build.xml。如果你运行 ant WikiPhilosophy，它将运行一些简单的启动代码。

### 7.2 可迭代对象和迭代器

在前一章中，我展示了迭代式深度优先搜索（DFS），并且认为与递归版本相比，迭代版本的优点在于，它更容易包装在 Iterator 对象中。在本节中，我们将看到如何实现它。

如果你不熟悉 Iterator 和 Iterable 接口，你可以阅读 <http://thinkdast.com/iterator> 和 <http://thinkdast.com/iterable>。

看看 `WikiNodeIterable.java` 的内容。外层的类 `WikiNodeIterable` 实现 `Iterable<Node>` 接口，所以我们可以使用它：

```
1. Node root = ...
2. Iterable<Node> iter = new WikiNodeIterable(root);
3. for (Node node: iter) {
4.     visit(node);
5. }
```

其中 `root` 是我们想要遍历的树的根节点，并且 `visit` 是一个方法，当我们“访问” `Node` 时，做任何我们想要的事情。

`WikiNodeIterable` 的实现遵循以下惯例：

- 构造函数接受并存储根 `Node` 的引用。
- `iterator` 方法创建一个返回一个 `Iterator` 对象。

这是它的样子：

```
1. public class WikiNodeIterable implements Iterable<Node> {
2.
3.     private Node root;
4.
5.     public WikiNodeIterable(Node root) {
6.         this.root = root;
7.     }
8.
9.     @Override
10.    public Iterator<Node> iterator() {
11.        return new WikiNodeIterator(root);
12.    }
13. }
```

内层的类 `WikiNodeIterator`，执行所有实际工作。

```
1. private class WikiNodeIterator implements Iterator<Node> {
2.
3.     Deque<Node> stack;
4.
5.     public WikiNodeIterator(Node node) {
6.         stack = new ArrayDeque<Node>();
7.         stack.push(root);
8.     }
9.
10.    @Override
11.    public boolean hasNext() {
```



```

12.         return !stack.isEmpty();
13.     }
14.
15.     @Override
16.     public Node next() {
17.         if (stack.isEmpty()) {
18.             throw new NoSuchElementException();
19.         }
20.
21.         Node node = stack.pop();
22.         List<Node> nodes = new ArrayList<Node>(node.childNodes());
23.         Collections.reverse(nodes);
24.         for (Node child: nodes) {
25.             stack.push(child);
26.         }
27.         return node;
28.     }
29. }

```

该代码与 DFS 的迭代版本几乎相同，但现在分为三个方法：

- 构造函数初始化栈（使用一个 `ArrayDeque` 实现）并将根节点压入这个栈。
- `isEmpty` 检查栈是否为空。
- `next` 从 `Node` 栈中弹出下一个节点，按相反的顺序压入子节点，并返回弹出的 `Node`。如果有人空 `Iterator` 上调用 `next`，则会抛出异常。

可能不明显的是，值得使用两个类和五个方法，来重写一个完美的方法。但是我们现在已经完成了，在需要 `Iterable` 的任何地方，我们可以使用 `WikiNodeIterable`，这使得它的语法整洁，易于将迭代逻辑（DFS）与我们对节点的处理分开。

## 7.3 `WikiFetcher`

编写 Web 爬虫时，很容易下载太多页面，这可能会违反你要下载的服务器的服务条款。为了帮助你避免这种情况，我提供了一个 `WikiFetcher` 类，它可以做两件事情：

- 它封装了我们在上一章中介绍的代码，用于从维基百科下载页面，解析 HTML 以及选择内容文本。
- 它测量请求之间的时间，如果我们在请求之间没有足够的时间，它将休眠直到经过了合理的间隔。默认情况下，间隔为 `1` 秒。

这里是 `WikiFetcher` 的定义：

```

1. public class WikiFetcher {
2.     private long lastRequestTime = -1;
3.     private long minInterval = 1000;

```

```

4.
5.     /**
6.      * Fetches and parses a URL string,
7.      * returning a list of paragraph elements.
8.      *
9.      * @param url
10.     * @return
11.     * @throws IOException
12.     */
13.     public Elements fetchWikipedia(String url) throws IOException {
14.         sleepIfNeeded();
15.
16.         Connection conn = Jsoup.connect(url);
17.         Document doc = conn.get();
18.         Element content = doc.getElementById("mw-content-text");
19.         Elements paragraphs = content.select("p");
20.         return paragraphs;
21.     }
22.
23.     private void sleepIfNeeded() {
24.         if (lastRequestTime != -1) {
25.             long currentTime = System.currentTimeMillis();
26.             long nextRequestTime = lastRequestTime + minInterval;
27.             if (currentTime < nextRequestTime) {
28.                 try {
29.                     Thread.sleep(nextRequestTime - currentTime);
30.                 } catch (InterruptedException e) {
31.                     System.err.println(
32.                         "Warning: sleep interrupted in fetchWikipedia.");
33.                 }
34.             }
35.         }
36.         lastRequestTime = System.currentTimeMillis();
37.     }
38. }

```

唯一的公共方法是 `fetchWikipedia`，接收 `String` 形式的 URL，并返回一个 `Elements` 集合，该集合包含的一个 DOM 元素表示内容文本中每个段落。这段代码应该很熟悉了。

新的代码是 `sleepIfNeeded`，它检查自上次请求以来的时间，如果经过的时间小于 `minInterval`（毫秒），则休眠。

这就是 `WikiFetcher` 全部。这是一个演示如何使用它的例子：

```

1. WikiFetcher wf = new WikiFetcher();
2.
3. for (String url: urlList) {

```

```

4.     Elements paragraphs = wf.fetchWikipedia(url);
5.     processParagraphs(paragraphs);
6. }

```

在这个例子中，我们假设 `urlList` 是一个 `String` 的集合，并且 `processParagraphs` 是一个方法，对 `Elements` 做一些事情，它由 `fetchWikipedia` 返回。

此示例展示了一些重要的东西：你应该创建一个 `WikiFetcher` 对象并使用它来处理所有请求。如果有多个 `WikiFetcher` 的实例，则它们不会确保请求之间的最小间隔。

注意：我的 `WikiFetcher` 实现很简单，但是通过创建多个实例，人们很容易误用它。你可以通过制作 `WikiFetcher` “单例”来避免这个问题，你可以阅读 <http://thinkdast.com/singleton>。

## 7.4 练习 5

在 `WikiPhilosophy.java` 中，你会发现一个简单的 `main` 方法，展示了如何使用这些部分。从这个代码开始，你的工作是写一个爬虫：

1. 获取维基百科页面的 URL，下载并分析。
2. 它应该遍历所得到的 DOM 树来找到第一个有效的链接。我会在下面解释“有效”的含义。
3. 如果页面没有链接，或者如果第一个链接是我们已经看到的页面，程序应该指示失败并退出。
4. 如果链接匹配维基百科页面上的哲学网址，程序应该提示成功并退出。
5. 否则应该回到步骤 1。

该程序应该为它访问的 URL 构建 `List`，并在结束时显示结果（无论成功还是失败）。

那么我们应该认为什么是“有效的”链接？你在这里有一些选择 各种版本的“到达哲学”推测使用略有不同的规则，但这里有一些选择：

- 这个链接应该在页面的内容文本中，而不是侧栏或弹出框。
- 它不应该是斜体或括号。
- 你应该跳过外部链接，当前页面的链接和红色链接。
- 在某些版本中，如果文本以大写字母开头，则应跳过链接。

你不必遵循所有这些规则，但我们建议你至少处理括号，斜体以及当前页面的链接。

如果你有足够的信息来起步，请继续。或者你可能想要阅读这些提示：

- 当你遍历树的时候，你将需要处理的两种 `Node` 是 `TextNode` 和 `Element`。如果你找到一个 `Element`，你可能需要转换它的类型，来访问标签和其他信息。
- 当你找到包含链接的 `Element` 时，通过向上跟踪父节点链，可以检查是否是斜体。如果父节点链中有一个 `<i>` 或 `<em>` 标签，链接为斜体。
- 为了检查链接是否在括号中，你必须在遍历树时扫描文本，并跟踪开启和闭合括号（理想情况下，你的解决方案应该能够处理嵌套括号（像这样））。

如果你从 Java 页面开始，你应该在跟随七个链接之后到达哲学，除非我运行代码后发生了改变。

好的，这就是你所得到的所有帮助。现在全靠你了。玩的开心！

## 第八章 索引器

- 第八章 索引器
  - 8.1 数据结构选取
  - 8.2 `TermCounter`
  - 8.3 练习 6

## 第八章 索引器

原文: [Chapter 8 Indexer](#)

译者: 飞龙

协议: [CC BY-NC-SA 4.0](#)

自豪地采用谷歌翻译

目前，我们构建了一个基本的 Web 爬虫；我们下一步将是索引。在网页搜索的上下文中，索引是一种数据结构，可以查找检索词并找到该词出现的页面。此外，我们想知道每个页面上显示检索词的次数，这将有助于确定与该词最相关的页面。

例如，如果用户提交检索词“Java”和“编程”，我们将查找两个检索词并获得两组页面。带有“Java”的页面将包括 Java 岛屿，咖啡昵称以及编程语言的网页。具有“编程”一词的页面将包括不同编程语言的页面，以及该单词的其他用途。通过选择具有两个检索词的页面，我们希望消除不相关的页面，并找到 Java 编程的页面。

现在我们了解索引是什么，它执行什么操作，我们可以设计一个数据结构来表示它。

### 8.1 数据结构选取

索引的基本操作是查找；具体来说，我们需要能够查找检索词并找到包含它的所有页面。最简单的实现将是页面的集合。给定一个检索词，我们可以遍历页面的内容，并选择包含检索词的内容。但运行时间与所有页面上的总字数成正比，这太慢了。

一个更好的选择是一个映射（字典），它是一个数据结构，表示键值对的集合，并提供了一种方法，快速查找键以及相应值。例如，我们将要构建的第一个映射是 `TermCounter`，它将每个检索词映射为页面中出现的次数。键是检索词，值是计数（也称为“频率”）。

Java 提供了 `Map` 的调用接口，它指定映射应该提供的方法；最重要的是：

- `get(key)`：此方法查找一个键并返回相应的值。
- `put(key, value)`：该方法向 `Map` 添加一个新的键值对，或者如果该键已经在映射中，它将替换与 `key` 关联的值。

Java 提供了几个 `Map` 实现，包括我们将关注的两个，`HashMap` 以及 `TreeMap`。在即将到来的章节中，我们将介绍这些实现并分析其性能。

除了检索词到计数的映射 `TermCounter` 之外，我们将定义一个被称为 `Index` 的类，它将检索词映射为出现的页面的集合。而这又引发了下一个问题，即如何表示页面集合。同样，如果我们考虑我们想要执行的操作，它们就指导了我们的决定。

在这种情况下，我们需要组合两个或多个集合，并找到所有这些集合中显示的页面。你可以将此操作看做集合的交集：两个集合的交集是出现在两者中的一组元素。

你可能猜到了，Java 提供了一个 `Set` 接口，来定义集合应该执行的操作。它实际上并不提供设置交集，但它提供了方法，使我们能够有效地实现交集和其他结合操作。核心的 `Set` 方法是：

- `add(element)`：该方法将一个元素添加到集合中；如果元素已经在集合中，则它不起作用。
- `contains(element)`：该方法检查给定元素是否在集合中。

Java 提供了几个 `Set` 实现，包括 `HashSet` 和 `TreeSet`。

现在我们自顶向下设计了我们的数据结构，我们将从内到外实现它们，从 `TermCounter` 开始。

## 8.2 `TermCounter`

`TermCounter` 是一个类，表示检索词到页面中出现次数的映射。这是类定义的第一部分：

```
1. public class TermCounter {
2.
3.     private Map<String, Integer> map;
4.     private String label;
5.
6.     public TermCounter(String label) {
7.         this.label = label;
8.         this.map = new HashMap<String, Integer>();
9.     }
10. }
```

实例变量 `map` 包含检索词到计数的映射，并且 `label` 标识检索词的来源文档；我们将使用它来存储 URL。

为了实现映射，我选择了 `HashMap`，它是最常用的 `Map`。在几章中，你将看到它是如何工作的，以及为什么它是一个常见的选择。

`TermCounter` 提供 `put` 和 `get`，定义如下：

```
1. public void put(String term, int count) {
2.     map.put(term, count);
```

```

3.     }
4.
5.     public Integer get(String term) {
6.         Integer count = map.get(term);
7.         return count == null ? 0 : count;
8.     }

```

`put` 只是一个包装方法；当你调用 `TermCounter` 的 `put` 时，它会调用内嵌映射的 `put`。

另一方面，`get` 做了一些实际工作。当你调用 `TermCounter` 的 `get` 时，它会在映射上调用 `get`，然后检查结果。如果该检索词没有出现在映射中，则 `TermCounter.get` 返回 `0`。`get` 的这种定义方式使 `incrementTermCount` 的写入更容易，它需要一个检索词，并增加关联该检索词的计数器。

```

1.     public void incrementTermCount(String term) {
2.         put(term, get(term) + 1);
3.     }

```

如果这个检索词未见过，则 `get` 返回 `0`；我们设为 `1`，然后使用 `put` 向映射添加一个新的键值对。如果该检索词已经在映射中，我们得到旧的计数，增加 `1`，然后存储新的计数，替换旧的值。

此外，`TermCounter` 还提供了这些其他方法，来帮助索引网页：

```

1.     public void processElements(Elements paragraphs) {
2.         for (Node node: paragraphs) {
3.             processTree(node);
4.         }
5.     }
6.
7.     public void processTree(Node root) {
8.         for (Node node: new WikiNodeIterable(root)) {
9.             if (node instanceof TextNode) {
10.                 processText(((TextNode) node).text());
11.             }
12.         }
13.     }
14.
15.     public void processText(String text) {
16.         String[] array = text.replaceAll("\\pP", " ").
17.             toLowerCase().
18.             split("\\s+");
19.
20.         for (int i=0; i<array.length; i++) {
21.             String term = array[i];
22.             incrementTermCount(term);

```

```

23.     }
24. }

```

最后，这里是一个例子，展示了如何使用 `TermCounter`：

```

1.  String url = "http://en.wikipedia.org/wiki/Java_(programming_language)";
2.  WikiFetcher wf = new WikiFetcher();
3.  Elements paragraphs = wf.fetchWikipedia(url);
4.
5.  TermCounter counter = new TermCounter(url);
6.  counter.processElements(paragraphs);
7.  counter.printCounts();

```

这个示例使用了 `WikiFetcher` 从维基百科下载页面，并解析正文。之后它创建了 `TermCounter` 并使用它来计数页面上的单词。

下一节中，你会拥有一个挑战，来运行这个代码，并通过填充缺失的方法来测试你的理解。

## 8.3 练习 6

在本书的存储库中，你将找到此练习的源文件：

- `TermCounter.java` 包含上一节中的代码。
- `TermCounterTest.java` 包含测试代码 `TermCounter.java`。
- `Index.java` 包含本练习下一部分的类定义。
- `WikiFetcher.java` 包含我们在上一个练习中使用的，用于下载和解析网页的类。
- `WikiNodeIterable.java` 包含我们用于遍历 DOM 树中的节点的类。

你还会发现 Ant 构建文件 `build.xml`。

运行 `ant build` 来编译源文件。然后运行 `ant TermCounter`；它应该运行上一节中的代码，并打印一个检索词列表及其计数。输出应该是这样的：

```

1. genericservlet, 2
2. configurations, 1
3. claimed, 1
4. servletresponse, 2
5. occur, 2
6. Total of all counts = -1

```

运行它时，检索词的顺序可能不同。

最后一行应该打印检索词计数的总和，但是由于方法 `size` 不完整而返回 `-1`。填充此方法并 `ant TermCounter` 重新运行。结果应该是 `4798`。



运行 `ant TermCounterTest` 来确认这部分练习是否完整和正确。

对于练习的第二部分，我将介绍 `Index` 对象的实现，你将填充一个缺失的方法。这是类定义的开始：

```

1. public class Index {
2.
3.     private Map<String, Set<TermCounter>> index =
4.         new HashMap<String, Set<TermCounter>>();
5.
6.     public void add(String term, TermCounter tc) {
7.         Set<TermCounter> set = get(term);
8.
9.         // if we're seeing a term for the first time, make a new Set
10.        if (set == null) {
11.            set = new HashSet<TermCounter>();
12.            index.put(term, set);
13.        }
14.        // otherwise we can modify an existing Set
15.        set.add(tc);
16.    }
17.
18.    public Set<TermCounter> get(String term) {
19.        return index.get(term);
20.    }

```

实例变量 `index` 是每个检索词到一组 `TermCounter` 对象的映射。每个 `TermCounter` 表示检索词出现的页面。

`add` 方法向集合添加新的 `TermCounter`，它与检索词关联。当我们索引一个尚未出现的检索词时，我们必须创建一个新的集合。否则我们可以添加一个新的元素到一个现有的集合。在这种情况下，`set.add` 修改位于 `index` 里面的集合，但不会修改 `index` 本身。我们唯一修改 `index` 的时候是添加一个新的检索词。

最后，`get` 方法接受检索词并返回相应的 `TermCounter` 对象集。

这种数据结构比较复杂。回顾一下，`Index` 包含 `Map`，将每个检索词映射到 `TermCounter` 对象的 `Set`，每个 `TermCounter` 包含一个 `Map`，将检索词映射到计数。

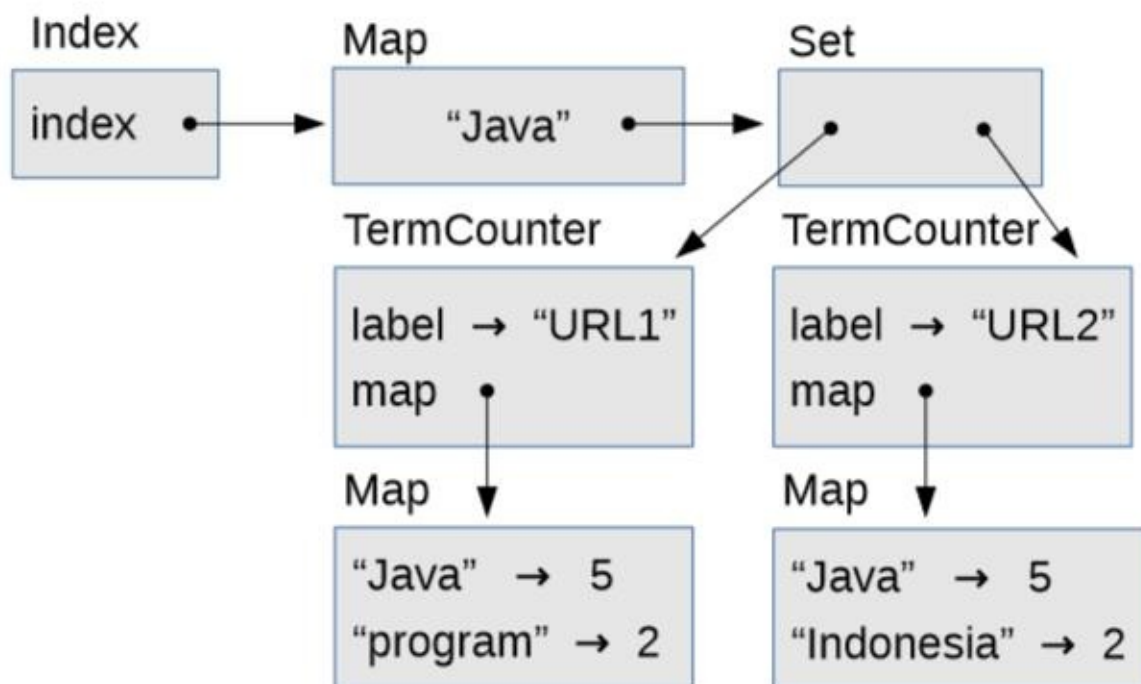


图 8.1 Index 的对象图

图 8.1 是展示这些对象的对象图。Index 对象具有一个名为 index 的 Map 实例变量。在这个例子中，Map 只包含一个字符串，"Java"，它映射到一个 Set，包含两个 TermCounter 对象的，代表每个出现单词"Java"的页面。

每个 TermCounter 包含 label，它是页面的 URL，以及 map，它是 Map，包含页面上的单词和每个单词出现的次数。

printIndex 方法展示了如何解压缩此数据结构：

```

1.  public void printIndex() {
2.      // loop through the search terms
3.      for (String term: keySet()) {
4.          System.out.println(term);
5.
6.          // for each term, print pages where it appears and frequencies
7.          Set<TermCounter> tcs = get(term);
8.          for (TermCounter tc: tcs) {
9.              Integer count = tc.get(term);
10.             System.out.println("    " + tc.getLabel() + " " + count);
11.         }
12.     }
13. }
  
```

外层循环遍历检索词。内层循环迭代 TermCounter 对象。

运行 `ant build` 来确保你的源代码已编译，然后运行 `ant Index`。它下载两个维基百科页面，对它们进行索引，并打印结果；但是当你运行它时，你将看不到任何输出，因为我们已经将其中一个方法留空。

你的工作是填写 `indexPage`，它需要一个 URL（一个 `String`）和一个 `Elements` 对象，并更新索引。下面的注释描述了应该做什么：

```
1. public void indexPage(String url, Elements paragraphs) {
2.     // 生成一个 TermCounter 并统计段落中的检索词
3.
4.     // 对于 TermCounter 中的每个检索词，将 TermCounter 添加到索引
5. }
```

它能工作之后，再次运行 `ant Index`，你应该看到如下输出：

```
1. ...
2. configurations
3.     http://en.wikipedia.org/wiki/Programming_language 1
4.     http://en.wikipedia.org/wiki/Java_(programming_language) 1
5. claimed
6.     http://en.wikipedia.org/wiki/Java_(programming_language) 1
7. servletresponse
8.     http://en.wikipedia.org/wiki/Java_(programming_language) 2
9. occur
10.    http://en.wikipedia.org/wiki/Java_(programming_language) 2
```

当你运行的时候，检索词的顺序可能有所不同。

同样，运行 `ant TestIndex` 来确定完成了这部分练习。

## 第九章 Map接口

- 第九章 Map 接口
  - 9.1 实现 MyLinearMap
  - 9.2 练习 7
  - 9.3 分析 MyLinearMap

## 第九章 Map 接口

原文: [Chapter 9 The Map interface](#)

译者: 飞龙

协议: [CC BY-NC-SA 4.0](#)

自豪地采用谷歌翻译

在接下来的几个练习中，我介绍了 Map 接口的几个实现。其中一个基于哈希表，这可以说是所发明的最神奇的数据结构。另一个是类似的 TreeMap，不是很神奇，但它有附加功能，它可以按顺序迭代元素。

你将有机会实现这些数据结构，然后我们将分析其性能。

但是在我们可以解释哈希表之前，我们将从一个 Map 开始，它使用键值对的 List 来简单实现。

### 9.1 实现 MyLinearMap

像往常一样，我提供启动代码，你将填写缺少的方法。这是 MyLinearMap 类定义的起始：

```
1. public class MyLinearMap<K, V> implements Map<K, V> {
2.
3.     private List<Entry> entries = new ArrayList<Entry>();
```

该类使用两个类型参数，K 是键的类型，V 是值的类型。MyLinearMap 实现 Map，这意味着它必须提供 Map 接口中的方法。

MyLinearMap 对象具有单个实例变量，entries，这是一个 Entry 的 ArrayList 对象。每个 Entry 都包含一个键值对。这里是定义：

```
1.     public class Entry implements Map.Entry<K, V> {
2.         private K key;
3.         private V value;
4.
```

```

5.     public Entry(K key, V value) {
6.         this.key = key;
7.         this.value = value;
8.     }
9.
10.    @Override
11.    public K getKey() {
12.        return key;
13.    }
14.    @Override
15.    public V getValue() {
16.        return value;
17.    }
18. }

```

`Entry` 没有什么，只是一个键和一个值的容器。该定义内嵌在 `MyLinearList` 中，因此它使用相同类型的参数，`K` 和 `V`。

这就是你做这个练习所需的所有东西，所以让我们开始吧。

## 9.2 练习 7

在本书的仓库中，你将找到此练习的源文件：

- `MyLinearMap.java` 包含练习的第一部分的起始代码。
- `MyLinearMapTest.java` 包含 `MyLinearMap` 的单元测试。

你还会找到 Ant 构建文件 `build.xml`。

运行 `ant build` 来编译源文件。然后运行 `ant MyLinearMapTest`；几个测试应该失败，因为你有一些任务要做。

首先，填写 `findEntry` 的主体。这是一个辅助方法，不是 `Map` 接口的一部分，但是一旦你让它工作，你可以在几种方法中使用它。给定一个目标键（Key），它应该搜索条目（Entry）并返回包含目标的条目（按照键，而不是值），或者如果不存在则返回 `null`。请注意，我提供了 `equals`，正确比较两个键并处理 `null`。

你可以再次运行 `ant MyLinearMapTest`，但即使你的 `findEntry` 是正确的，测试也不会通过，因为 `put` 不完整。

填充 `put`。你应该阅读 `Map.put` 的文档，<http://thinkdast.com/listput>，以便你知道应该做什么。你可能希望从一个版本开始，其中 `put` 始终添加新条目，并且不会修改现有条目；这样你可以先测试简单的情况。或者如果你更加自信，你可以一次写出整个东西。

一旦你 `put` 正常工作，测试 `containsKey` 应该通过。

阅读 `Map.get` 的文档，<http://thinkdast.com/listget>，然后填充方法。再次运行测试。

最后，阅读 `Map.remove` 的文档，<http://thinkdast.com/maprem> 并填充方法。

到了这里，所有的测试都应该通过。恭喜！

## 9.3 分析 `MyLinearMap`

这一节中，我展示了上一个练习的答案，并分析核心方法的性能。这里是 `findEntry` 和 `equals`。

```

1. private Entry findEntry(Object target) {
2.     for (Entry entry: entries) {
3.         if (equals(target, entry.getKey())) {
4.             return entry;
5.         }
6.     }
7.     return null;
8. }
9.
10. private boolean equals(Object target, Object obj) {
11.     if (target == null) {
12.         return obj == null;
13.     }
14.     return target.equals(obj);
15. }
```

`equals` 的运行时间可能取决于 `target` 键和键的大小，但通常不取决于条目的数量，`n`。那么 `equals` 是常数时间。

在 `findEntry` 中，我们可能会很幸运，并在一开始就找到我们要找的键，但是我们不能指望它。一般来说，我们要搜索的条目数量与 `n` 成正比，所以 `findEntry` 是线性的。

大部分的 `MyLinearMap` 核心方法使用 `findEntry`，包括 `put`，`get`，和 `remove`。这就是他们的样子：

```

1. public V put(K key, V value) {
2.     Entry entry = findEntry(key);
3.     if (entry == null) {
4.         entries.add(new Entry(key, value));
5.         return null;
6.     } else {
7.         V oldValue = entry.getValue();
8.         entry.setValue(value);
9.         return oldValue;
10.    }
```

```

11. }
12. public V get(Object key) {
13.     Entry entry = findEntry(key);
14.     if (entry == null) {
15.         return null;
16.     }
17.     return entry.getValue();
18. }
19. public V remove(Object key) {
20.     Entry entry = findEntry(key);
21.     if (entry == null) {
22.         return null;
23.     } else {
24.         V value = entry.getValue();
25.         entries.remove(entry);
26.         return value;
27.     }
28. }

```

`put` 调用 `findEntry` 之后，其他一切都是常数时间。记住这个 `entries` 是一个 `ArrayList`，所以降魔为添加元素平均是常数时间。如果键已经在映射中，我们不需要添加条目，但我们必须调用 `entry.getValue` 和 `entry.setValue`，而这些都是常数时间。把它们放在一起，`put` 是线性的。

同样，`get` 也是线性的。

`remove` 稍微复杂一些，因为 `entries.remove` 可能需从一开始或中间删除 `ArrayList` 的一个元素，并且需要线性时间。但是没关系：两个线性运算仍然是线性的。

总而言之，核心方法都是线性的，这就是为什么我们将这个实现称为 `MyLinearMap`（嗒嗒！）。

如果我们知道输入的数量很少，这个实现可能会很好，但是我们可以做得更好。实际上，`Map` 所有的核心方法都是常数时间的实现。当你第一次听到这个消息时，可能似乎觉得不可能。实际上我们所说的是，你可以在常数时间内大海捞针，不管海有多大。这是魔法。

我们不是将条目存储在一个大的 `List` 中，而是把它们分解成许多短的列表。对于每个键，我们将使用哈希码（在下一节中进行说明）来确定要使用的列表。

使用大量的简短列表比仅仅使用一个更快，但正如我将解释的，它不会改变增长级别；核心功能仍然是线性的。但还有一个技巧：如果我们增加列表的数量来限制每个列表的条目数，就会得到一个恒定时间的映射。你会在下一个练习中看到细节，但是首先要了解哈希！

在下一章中，我将介绍一种解决方案，分析 `Map` 核心方法的性能，并引入更有效的实现。

## 第十章 哈希

- 第十章 哈希
  - 10.1 哈希
  - 10.2 哈希如何工作？
  - 10.3 哈希和可变性
  - 10.4 练习 8

## 第十章 哈希

原文: [Chapter 10 Hashing](#)

译者: 飞龙

协议: [CC BY-NC-SA 4.0](#)

自豪地采用谷歌翻译

在本章中，我定义了一个比 `MyLinearMap` 更好的 `Map` 接口实现，`MyBetterMap`，并引入哈希，这使得 `MyBetterMap` 效率更高。

### 10.1 哈希

为了提高 `MyLinearMap` 的性能，我们将编写一个新的类，它被称为 `MyBetterMap`，它包含 `MyLinearMap` 对象的集合。它在内嵌的映射之间划分键，因此每个映射中的条目数量更小，这加快了 `findEntry`，以及依赖于它的方法的速度。

这是类定义的开始：

```
1. public class MyBetterMap<K, V> implements Map<K, V> {
2.
3.     protected List<MyLinearMap<K, V>> maps;
4.
5.     public MyBetterMap(int k) {
6.         makeMaps(k);
7.     }
8.
9.     protected void makeMaps(int k) {
10.        maps = new ArrayList<MyLinearMap<K, V>>(k);
11.        for (int i=0; i<k; i++) {
12.            maps.add(new MyLinearMap<K, V>());
13.        }
14.    }
15. }
```



实例变量 `maps` 是一组 `MyLinearMap` 对象。构造函数接受一个参数 `k`，决定至少最开始，要使用多少个映射。然后 `makeMaps` 创建内嵌的映射并将其存储在一个 `ArrayList` 中。

现在，完成这项工作的关键是，我们需要一些方法来查看一个键，并决定应该进入哪个映射。当我们 `put` 一个新的键时，我们选择一个映射；当我们 `get` 同样的键时，我们必须记住我们把它放在哪里。

一种可能性是随机选择一个子映射，并跟踪我们把每个键放在哪里。但我们应该如何跟踪？看起来我们可以用一个 `Map` 来查找键，并找到正确的子映射，但是练习的重点是编写一个有效的 `Map` 实现。我们不能假设我们已经有了。

一个更好的方法是使用一个哈希函数，它接受一个 `Object`，一个任意的 `Object`，并返回一个称为哈希码的整数。重要的是，如果它不止一次看到相同的 `Object`，它总是返回相同的哈希码。这样，如果我们使用哈希码来存储键，当我们查找时，我们将得到相同的哈希码。

在Java中，每个 `Object` 都提供了 `hashCode`，一种计算哈希函数的方法。这种方法的实现对于不同的对象是不同的；我们会很快看到一个例子。

这是一个辅助方法，为一个给定的键选择正确的子映射：

```
1. protected MyLinearMap<K, V> chooseMap(Object key) {
2.     int index = 0;
3.     if (key != null) {
4.         index = Math.abs(key.hashCode()) % maps.size();
5.     }
6.     return maps.get(index);
7. }
```

如果 `key` 是 `null`，我们任意选择索引为 `0` 的子映射。否则，我们使用 `hashCode` 获取一个整数，调用 `Math.abs` 来确保它是非负数，然后使用余数运算符 `%`，这保证结果在 `0` 和 `maps.size()-1` 之间。所以 `index` 总是一个有效的 `maps` 索引。然后 `chooseMap` 返回为其所选的映射的引用。

我们使用 `chooseMap` 的 `put` 和 `get`，所以当我们查询键的时候，我们得到添加时所选的不同映射，我们选择了相同的映射。至少应该是 - 稍后我会解释为什么这可能不起作用。

这是我的 `put` 和 `get` 的实现：

```
1. public V put(K key, V value) {
2.     MyLinearMap<K, V> map = chooseMap(key);
3.     return map.put(key, value);
4. }
5.
6. public V get(Object key) {
```

```

7.     MyLinearMap<K, V> map = chooseMap(key);
8.     return map.get(key);
9. }

```

很简单，对吧？在这两种方法中，我们使用 `chooseMap` 来找到正确的子映射，然后在子映射上调用一个方法。这就是它的工作原理。现在让我们考虑一下性能。

如果在  $k$  个子映射中分配了  $n$  个条目，则平均每个映射将有  $n/k$  个条目。当我们查找一个键时，我们必须计算其哈希码，这需要一些时间，然后我们搜索相应的子映射。

因为 `MyBetterMap` 中的条目列表，比 `MyLinearMap` 中的短  $k$  倍，我们的预期是  $k$  倍的搜索速度。但运行时间仍然与  $n$  成正比，所以 `MyBetterMap` 仍然是线性的。在下一个练习中，你将看到如何解决这个问题。

## 10.2 哈希如何工作？

哈希函数的基本要求是，每次相同的对象应该产生相同的哈希码。对于不变的对象，这是比较容易的。对于具有可变状态的对象，我们必须花费更多精力。

作为一个不可变对象的例子，我将定义一个 `SillyString` 类，它包含一个 `String`：

```

1. public class SillyString {
2.     private final String innerString;
3.
4.     public SillyString(String innerString) {
5.         this.innerString = innerString;
6.     }
7.
8.     public String toString() {
9.         return innerString;
10.    }

```

这个类不是很有用，所以它叫做 `SillyString`。但是我会使用它来展示，一个类如何定义它自己的哈希函数：

```

1.     @Override
2.     public boolean equals(Object other) {
3.         return this.toString().equals(other.toString());
4.     }
5.
6.     @Override
7.     public int hashCode() {
8.         int total = 0;
9.         for (int i=0; i<innerString.length(); i++) {

```

```

10.         total += innerString.charAt(i);
11.     }
12.     return total;
13. }

```

注意 `SillyString` 重写了 `equals` 和 `hashCode`。这个很重要。为了正常工作，`equals` 必须和 `hashCode` 一致，这意味着如果两个对象被认为是相等的 - 也就是说，`equals` 返回 `true` - 它们应该有相同的哈希码。但这个要求只是单向的；如果两个对象具有相同的哈希码，则它们不一定必须相等。

`equals` 通过调用 `toString` 来工作，返回 `innerString`。因此，如果两个 `SillyString` 对象的 `innerString` 实例变量相等，它们就相等。

`hashCode` 的原理是，迭代 `String` 中的字符并将它们相加。当你向 `int` 添加一个字符时，Java 将使用其 Unicode 代码点，将字符转换为整数。你不需要了解 Unicode 的任何信息来弄清此示例，但如果你好奇，可以在 <http://thinkdast.com/codepoint> 上阅读更多内容。

该哈希函数满足要求：如果两个 `SillyString` 对象包含相等的内嵌字符串，则它们将获得相同的哈希码。

这可以正常工作，但它可能不会产生良好的性能，因为它为许多不同的字符串返回相同的哈希码。如果两个字符串以任何顺序包含相同的字母，它们将具有相同的哈希码。即使它们不包含相同的字母，它们可能会产生相同的总量，例如 `"ac"` 和 `"bb"`。

如果许多对象具有相同的哈希码，它们将在同一个子映射中。如果一些子映射比其他映射有更多的条目，那么当我们将 `k` 个映射时，加速比可能远远小于 `k`。所以哈希函数的目的之一是统一；也就是说，以相等的可能性，在这个范围内产生任何值。你可以在 <http://thinkdast.com/hash> 上阅读更多设计完成的，散列函数的信息。

## 10.3 哈希和可变性

`String` 是不可变的，`SillyString` 也是不可变的，因为 `innerString` 定义为 `final`。一旦你创建了一个 `SillyString`，你不能使 `innerString` 引用不同的 `String`，你不能修改所指向的 `String`。因此，它将始终具有相同的哈希码。

但是让我们看看一个可变对象会发生什么。这是一个 `SillyArray` 定义，它与 `SillyString` 类似，除了它使用一个字符数组而不是一个 `String`：

```

1. public class SillyArray {
2.     private final char[] array;
3.
4.     public SillyArray(char[] array) {
5.         this.array = array;
6.     }

```

```

7.
8.     public String toString() {
9.         return Arrays.toString(array);
10.    }
11.
12.    @Override
13.    public boolean equals(Object other) {
14.        return this.toString().equals(other.toString());
15.    }
16.
17.    @Override
18.    public int hashCode() {
19.        int total = 0;
20.        for (int i=0; i<array.length; i++) {
21.            total += array[i];
22.        }
23.        System.out.println(total);
24.        return total;
25.    }

```

`SillyArray` 也提供 `setChar`，它能够修改修改数组内的字符。

```

1. public void setChar(int i, char c) {
2.     this.array[i] = c;
3. }

```

现在假设我们创建了一个 `SillyArray`，并将其添加到 `map`。

```

1. SillyArray array1 = new SillyArray("Word1".toCharArray());
2. map.put(array1, 1);

```

这个数组的哈希码是 `461`。现在如果我们修改了数组内容，之后尝试查询它，像这样：

```

1. array1.setChar(0, 'C');
2. Integer value = map.get(array1);

```

修改之后的哈希码是 `441`。使用不同的哈希码，我们就很可能进入了错误的子映射。这就很糟糕了。

一般来说，使用可变对象作为散列数据结构中的键是很危险的，这包括 `MyBetterMap` 和 `HashMap`。如果你可以保证映射中的键不被修改，或者任何更改都不会影响哈希码，那么这可能是正确的。但是避免这样做可能是一个好主意。

## 10.4 练习 8

在这个练习中，你将完成 `MyBetterMap` 的实现。在本书的仓库中，你将找到此练习的源文件：

- `MyLinearMap.java` 包含我们在以前的练习中的解决方案，我们将在此练习中加以利用。
- `MyBetterMap.java` 包含上一章的代码，你将填充一些方法。
- `MyHashMap.java` 包含按需增长的哈希表的概要，你将完成它。
- `MyLinearMapTest.java` 包含 `MyLinearMap` 的单元测试。
- `MyBetterMapTest.java` 包含 `MyBetterMap` 的单元测试。
- `MyHashMapTest.java` 包含 `MyHashMap` 的单元测试。
- `Profiler.java` 包含用于测量和绘制运行时间与问题大小的代码。
- `ProfileMapPut.java` 包含配置该 `Map.put` 方法的代码。

像往常一样，你应该运行 `ant build` 来编译源文件。然后运行 `ant MyBetterMapTest`。几个测试应该失败，因为你有一些工作要做！

从以前的章节回顾 `put` 和 `get` 的实现。然后填充 `containsKey` 的主体。提示：使用 `chooseMap`。再次运行 `ant MyBetterMapTest` 并确认通过了 `testContainsKey`。

填充 `containsValue` 的主体。提示：不要使用 `chooseMap`。`ant MyBetterMapTest` 再次运行并确认通过了 `testContainsValue`。请注意，比起找到一个键，我们必须做更多的操作才能找到一个值。

类似 `put` 和 `get`，这个实现的 `containsKey` 是线性的，因为它搜索了内嵌子映射之一。在下一章中，我们将看到如何进一步改进此实现。

# 第十一章 HashMap

- 第十一章 HashMap
  - 11.1 练习 9
  - 11.2 分析 MyHashMap
  - 11.3 权衡
  - 11.4 分析 MyHashMap
  - 11.5 修复 MyHashMap
  - 11.6 UML 类图

## 第十一章

HashMap

原文: [Chapter 11 HashMap](#)

译者: 飞龙

协议: [CC BY-NC-SA 4.0](#)自豪地采用[谷歌翻译](#)

上一章中，我们写了一个使用哈希的 Map 接口的实现。我们期望这个版本更快，因为它搜索的列表较短，但增长顺序仍然是线性的。

如果存在  $n$  个条目和  $k$  个子映射，则子映射的大小平均为  $n/k$ ，这仍然与  $n$  成正比。但是，如果我们与  $n$  一起增加  $k$ ，我们可以限制  $n/k$  的大小。

例如，假设每次  $n$  超过  $k$  的时候，我们都使  $k$  加倍；在这种情况下，每个映射的条目的平均数量将小于  $1$ ，并且几乎总是小于  $10$ ，只要散列函数能够很好地展开键。

如果每个子映射的条目数是不变的，我们可以在常数时间内搜索一个子映射。并且计算散列函数通常是常数时间（它可能取决于键的大小，但不取决于键的数量）。这使得 Map 的核心方法，put 和 get 时间不变。

在下一个练习中，你将看到细节。

## 11.1 练习 9

在 MyHashMap.java 中，我提供了哈希表的大纲，它会按需增长。这里是定义的起始：

```
1. public class MyHashMap<K, V> extends MyBetterMap<K, V> implements Map<K, V> {
2.
3.     // average number of entries per sub-map before we rehash
4.     private static final double FACTOR = 1.0;
```

```

5.
6.     @Override
7.     public V put(K key, V value) {
8.         V oldValue = super.put(key, value);
9.
10.        // check if the number of elements per sub-map exceeds the threshold
11.        if (size() > maps.size() * FACTOR) {
12.            rehash();
13.        }
14.        return oldValue;
15.    }
16. }

```

`MyHashMap` 扩展了 `MyBetterMap`，所以它继承了那里定义的方法。它覆盖的唯一方法是 `put`，它调用了超类中的 `put` — 也就是说，它调用了 `MyBetterMap` 中的 `put` 版本 — 然后它检查它是否必须 `rehash`。调用 `size` 返回总数量 `n`。调用 `maps.size` 返回内嵌映射的数量 `k`。

常数 `FACTOR`（称为负载因子）确定每个子映射的平均最大条目数。如果 `n > k * FACTOR`，这意味着 `n/k > FACTOR`，意味着每个子映射的条目数超过阈值，所以我们调用 `rehash`。

运行 `ant build` 来编译源文件。然后运行 `ant MyHashMapTest`。它应该失败，因为执行 `rehash` 会抛出异常。你的工作是填充它。

填充 `rehash` 的主体，来收集表中的条目，调整表的大小，然后重新放入条目。我提供了两种可能会派上用场的方法：`MyBetterMap.makeMaps` 和 `MyLinearMap.getEntries`。每次调用它时，你的解决方案应该使映射数量加倍。

## 11.2 分析 `MyHashMap`

如果最大子映射中的条目数与 `n/k` 成正比，并且 `k` 与 `n` 成正比，那么多个核心方法就是常数时间的：

```

1.     public boolean containsKey (Object target) {
2.         MyLinearMap <K, V> map = chooseMap (target);
3.         return map.containsKey (target);
4.     }
5.
6.     public V get (Object key) {
7.         MyLinearMap <K, V> map = chooseMap (key); return map.get (key);
8.     }
9.     public V remove (Object key) {
10.        MyLinearMap <K, V> map = chooseMap (key);
11.        return map.remove (key);
12.    }

```

每个方法都计算键的哈希，这是常数时间，然后在一个子映射上调用一个方法，这个方法是常数时间的。

到现在为止还挺好。但另一个核心方法，`put` 有点难分析。当我们不需要 `rehash` 时，它是不变的时间，但是当我们这样做时，它是线性的。这样，它与 3.2 节中我们分析的 `ArrayList.add` 类似。

出于同样的原因，如果我们平摊一系列的调用，结果是常数时间。同样，论证基于摊销分析（见 3.2 节）。

假设子映射的初始数量 `k` 为 2，负载因子为 1。现在来看看 `put` 一系列的键需要多少工作量。作为基本的“工作单位”，我们将计算对密钥哈希，并将其添加到子映射中的次数。

我们第一次调用 `put` 时，它需要 1 个工作单位。第二次也需要 1 个单位。第三次我们需要 `rehash`，所以需要 2 个单位重新填充现有的键，和 1 个单位来对新键哈希。

译者注：可以单独计算 `rehash` 中转移元素的数量，然后将元素转移的复杂度和计算哈希的复杂度相加。

现在哈希表的大小是 4，所以下次调用 `put` 时，需要 1 个工作单位。但是下一次我们必须 `rehash`，需要 4 个单位来 `rehash` 现有的键，和 1 个单位来对新键哈希。

图 11.1 展示了规律，对新键哈希的正常工作量在底部展示，额外工作量展示为塔楼。

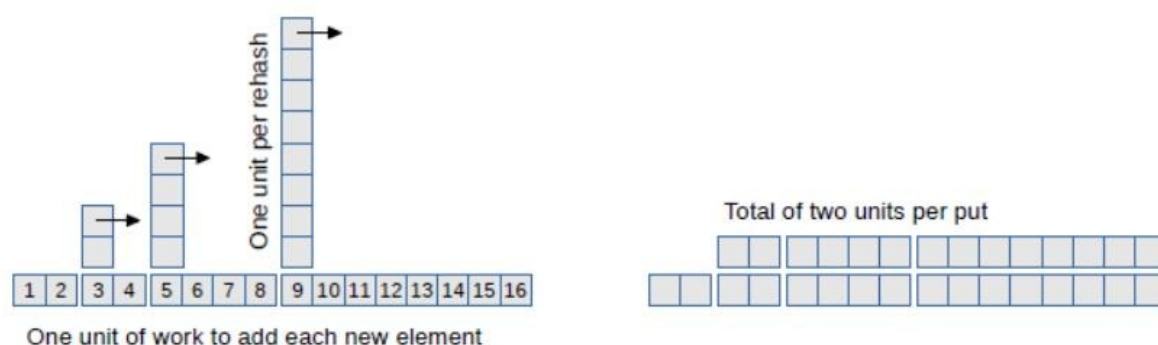


图 11.1：向哈希表添加元素的工作量展示

如箭头所示，如果我们把塔楼推倒，每个积木都会在下一个塔楼之前填满空间。结果似乎 2 个单位的均匀高度，这表明 `put` 的平均工作量约为 2 个单位。这意味着 `put` 平均是常数时间。

这个图还显示了，当我们 `rehash` 的时候，为什么加倍子映射数量 `k` 很重要。如果我们只是加上 `k` 而不是加倍，那么这些塔楼会靠的太近，他们会开始堆积。这样就不会是常数时间了。

## 11.3 权衡

我们已经表明，`containsKey`，`get` 和 `remove` 是常数时间，`put` 平均为常数时间。我们应该花一点时间来欣赏它有多么出色。无论哈希表有多大，这些操作的性能几乎相同。算是这样吧。



记住，我们的分析基于一个简单的计算模型，其中每个“工作单位”花费相同的时间量。真正的电脑比这更复杂。特别是，当处理足够小，适应高速缓存的数据结构时，它们通常最快；如果结构不适合高速缓存但仍适合内存，则稍慢一点；如果结构不适合在内存中，则非常慢。

这个实现的另一个限制是，如果我们得到了一个值而不是一个键时，那么散列是不会有帮助的：`containsValue` 是线性的，因为它必须搜索所有的子映射。查找一个值并找到相应的键（或可能的键），没有特别有效的方式。

还有一个限制：`MyLinearMap` 的一些常数时间的方法变成了线性的。例如：

```
1. public void clear() {
2.     for (int i=0; i<maps.size(); i++) {
3.         maps.get(i).clear();
4.     }
5. }
```

`clear` 必须清除所有的子映射，子映射的数量与 `n` 成正比，所以它是线性的。幸运的是，这个操作并不常用，所以在大多数应用中，这种权衡是可以接受的。

## 11.4 分析 `MyHashMap`

在我们继续之前，我们应该检查一下，`MyHashMap.put` 是否真的是常数时间。

运行 `ant build` 来编译源文件。然后运行 `ant ProfileMapPut`。它使用一系列问题规模，测量 `HashMap.put`（由 Java 提供）的运行时间，并在重对数比例尺上绘制运行时间与问题规模。如果这个操作是常数时间，`n` 个操作的总时间应该是线性的，所以结果应该是斜率为 `1` 的直线。当我运行这个代码时，估计的斜率接近 `1`，这与我们的分析一致。你应该得到类似的东西。

修改 `ProfileMapPut.java`，来测量你的 `MyHashMap` 实现，而不是 Java 的 `HashMap`。再次运行分析器，查看斜率是否接近 `1`。你可能需要调整 `startN` 和 `endMillis`，来找到一系列问题规模，其中运行时间多于几毫秒，但不超过几秒。

当我运行这个代码时，我感到惊讶：斜率大约为 `1.7`，这表明这个实现不是一直都是常数的。它包含一个“性能错误”。

在阅读下一节之前，你应该跟踪错误，修复错误，并确认现在 `put` 是常数时间，符合预期。

## 11.5 修复 `MyHashMap`

`MyHashMap` 的问题是 `size`，它继承自 `MyBetterMap`：

```
1. public int size() {
```

```

2.     int total = 0;
3.     for (MyLinearMap<K, V> map: maps) {
4.         total += map.size();
5.     }
6.     return total;
7. }

```

为了累计整个大小，它必须迭代子映射。由于我们增加了子映射的数量 `k`，随着条目数 `n` 增加，所以 `k` 与 `n` 成正比，所以 `size` 是线性的。

`put` 也是线性的，因为它使用 `size`：

```

1.     public V put(K key, V value) {
2.         V oldValue = super.put(key, value);
3.
4.         if (size() > maps.size() * FACTOR) {
5.             rehash();
6.         }
7.         return oldValue;
8.     }

```

如果 `size` 是线性的，我们做的一切都浪费了。

幸运的是，有一个简单的解决方案，我们以前看过：我们必须维护实例变量中的条目数，并且每当我们调用一个改变它的方法时更新它。

你会在这本书的仓库中找到我的解决方案 `MyFixedHashMap.java`。这是类定义的起始：

```

1. public class MyFixedHashMap<K, V> extends MyHashMap<K, V> implements Map<K, V> {
2.
3.     private int size = 0;
4.
5.     public void clear() {
6.         super.clear();
7.         size = 0;
8.     }

```

我们不修改 `MyHashMap`，我定义一个扩展它的新类。它添加一个新的实例变量 `size`，它被初始化为零。

更新 `clear` 很简单；我们在超类中调用 `clear`（清除子映射），然后更新 `size`。

更新 `remove` 和 `put` 有点困难，因为当我们调用超类的该方法，我们不能得知子映射的大小是否改变。这是我的解决方式：

```

1.     public V remove(Object key) {
2.         MyLinearMap<K, V> map = chooseMap(key);
3.         size -= map.size();
4.         V oldValue = map.remove(key);
5.         size += map.size();
6.         return oldValue;
7.     }

```

`remove` 使用 `chooseMap` 找到正确的子映射，然后减去子映射的大小。它会在子映射上调用 `remove`，根据是否找到了键，它可以改变子映射的大小，也可能不会改变它的大小。但是无论哪种方式，我们将子映射的新大小加到 `size`，所以最终的 `size` 值是正确的。

重写的 `put` 版本是类似的：

```

1.     public V put(K key, V value) {
2.         MyLinearMap<K, V> map = chooseMap(key);
3.         size -= map.size();
4.         V oldValue = map.put(key, value);
5.         size += map.size();
6.
7.         if (size() > maps.size() * FACTOR) {
8.             size = 0;
9.             rehash();
10.        }
11.        return oldValue;
12.    }

```

我们在这里也有同样的问题：当我们在子地图上调用 `put` 时，我们不知道是否添加了一个新的条目。所以我们使用相同的解决方案，减去旧的大小，然后加上新的大小。

现在 `size` 方法的实现很简单了：

```

1.     public int size() {
2.         return size;
3.     }

```

并且正好是常数时间。

当我测量这个解决方案时，我发现放入 `n` 个键的总时间正比于 `n`，也就是说，每个 `put` 是常数时间的，符合预期。

## 11.6 UML 类图

在本章中使用代码的一个挑战是，我们有几个互相依赖的类。以下是类之间的一些关系：

- `MyLinearMap` 包含一个 `LinkedList` 并实现了 `Map`。
- `MyBetterMap` 包含许多 `MyLinearMap` 对象并实现了 `Map`。
- `MyHashMap` 扩展了 `MyBetterMap`，所以它也包含 `MyLinearMap` 对象，并实现了 `Map`。
- `MyFixedHashMap` 扩展了 `MyHashMap` 并实现了 `Map`。

为了有助于跟踪这些关系，软件工程师经常使用 UML 类图。UML 代表统一建模语言（见 <http://thinkdast.com/uml>）。“类图”是由 UML 定义的几种图形标准之一。

在类图中，每个类由一个框表示，类之间的关系由箭头表示。图 11.2 显示了使用在线工具 yUML (<http://yuml.me/>) 生成的，上一个练习的 UML 类图。

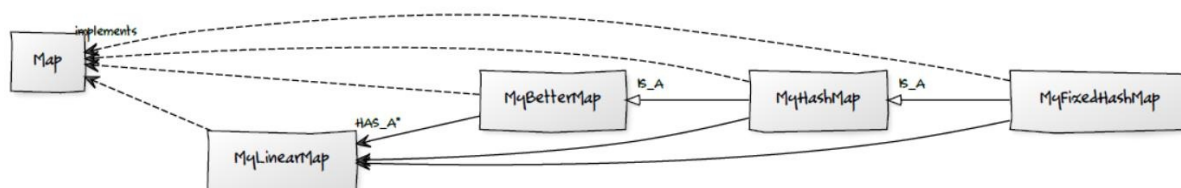


图11.2：本章中的 UML 类图

不同的关系由不同的箭头表示：

- 实心箭头表示 HAS-A 关系。例如，每个 `MyBetterMap` 实例包含多个 `MyLinearMap` 实例，因此它们通过实线箭头连接。
- 空心和实线箭头表示 IS-A 关系。例如，`MyHashMap` 扩展了 `MyBetterMap`，因此它们通过 IS-A 箭头连接。
- 空心和虚线箭头表示一个类实现了一个接口；在这个图中，每个类都实现 `Map`。

UML 类图提供了一种简洁的方式，来表示大量类集合的信息。在设计阶段中，它们用于交流备选设计，在实施阶段中，用于维护项目的共享思维导图，并在部署过程中记录设计。

## 第十二章 TreeMap

- 第十二章 `TreeMap`
  - 12.1 哈希哪里不对？
  - 12.2 二叉搜索树
  - 12.3 练习 10
  - 12.4 实现 `TreeMap`

## 第十二章 `TreeMap`

原文: [Chapter 12 TreeMap](#)

译者: 飞龙

协议: [CC BY-NC-SA 4.0](#)

自豪地采用谷歌翻译

这一章展示了二叉搜索树，它是个 `Map` 接口的高效实现。如果我们想让元素有序，它非常实用。

### 12.1 哈希哪里不对？

此时，你应该熟悉 Java 提供的 `Map` 接口和 `HashMap` 实现。通过使用哈希表来制作你自己的 `Map`，你应该了解 `HashMap` 的工作原理，以及为什么我们预计其核心方法是常数时间的。

由于这种表现，`HashMap` 被广泛使用，但并不是唯一的 `Map` 实现。有几个原因可能需要另一个实现：

哈希可能很慢，所以即使 `HashMap` 操作是常数时间，“常数”可能很大。

如果哈希函数将键均匀分配给子映射，效果很好。但设计良好的散列函数并不容易，如果太多的键在相同的子映射上，那么 `HashMap` 的性能可能会很差。

哈希表中的键不以任何特定顺序存储；实际上，当表增长并且键被重新排列时，顺序可能会改变。对于某些应用程序，必须或至少保持键的顺序，这很有用。

很难同时解决所有这些问题，但是 Java 提供了一个称为 `TreeMap` 的实现：

- 它不使用哈希函数，所以它避免了哈希的开销和选择哈希函数的困难。
- 在 `TreeMap` 之中，键被存储在二叉搜索树中，这使我们可以以线性时间顺序遍历键。
- 核心方法的运行时间与  $\log(n)$  成正比，并不像常数时间那样好，但仍然非常好。

在下一节中，我将解释二进制搜索树如何工作，然后你将使用它来实现 `Map`。另外，使用树实现时，我们将分析映射的核心方法的性能。

## 12.2 二叉搜索树

二叉搜索树 (BST) 是一个树，其中每个 `node` (节点) 包含一个键，并且每个都具有“BST 属性”：

- 如果 `node` 有一个左子树，左子树中的所有键都必须小于 `node` 的键。
- 如果 `node` 有一个右子树，右子树中的所有键都必须大于 `node` 的键。

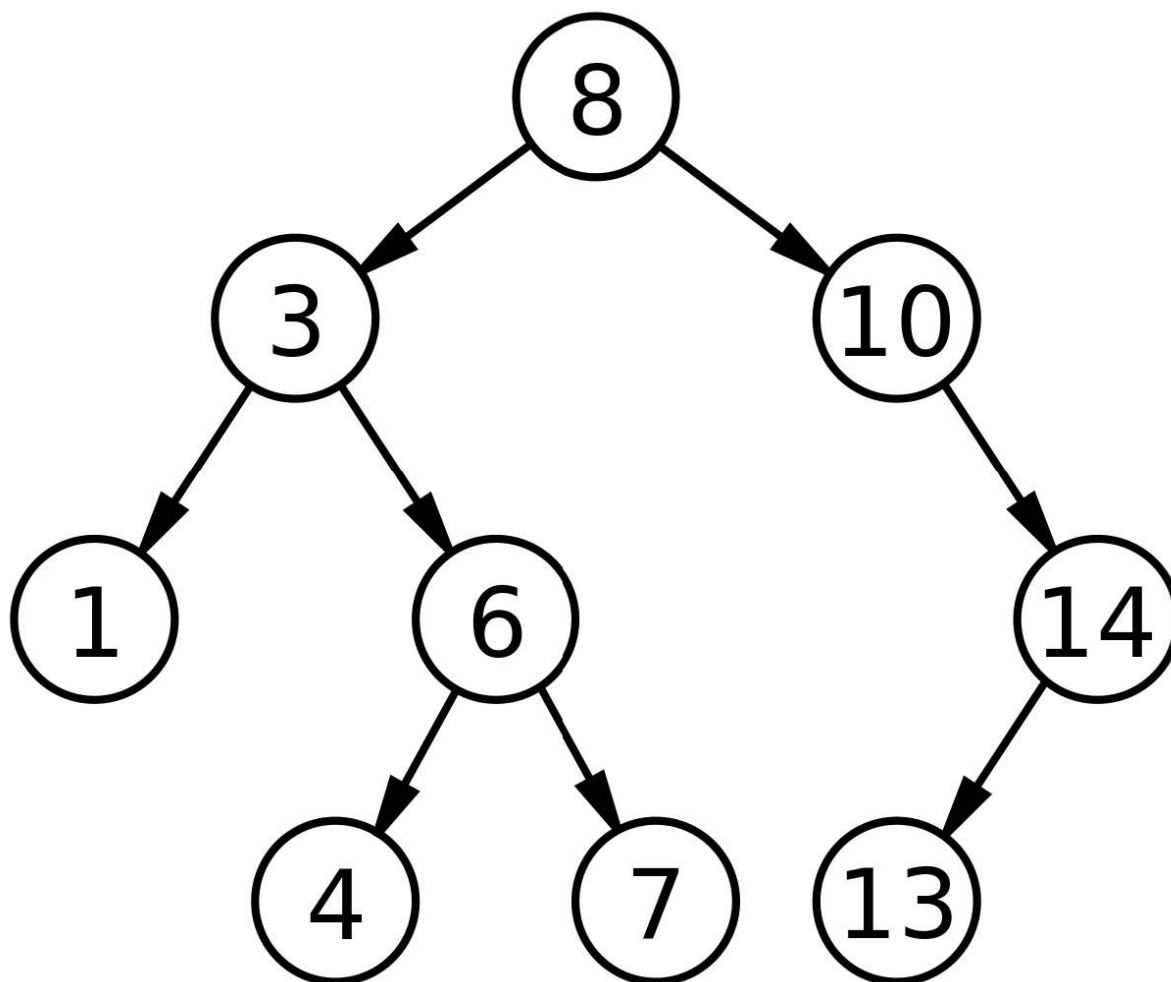


图 12.1：二叉搜索树示例

图 12.1 展示了一个具有此属性的整数的树。这个图片来自二叉搜索树的维基百科页面，位于 <http://thinkdast.com/bst>，当你做这个练习时，你会发现它很实用。

根节点中的键为 `8`，你可以确认根节点左边的所有键小于 `8`，右边的所有键都更大。你还可以检查其他节点是否具有此属性。

在二叉搜索树中查找一个键是很快，因为我们不必搜索整个树。从根节点开始，我们可以使用以下算法：

- 将你要查找的键 `target`，与当前节点的键进行比较。如果他们相等，你就完成了。
- 如果 `target` 小于当前键，搜索左子树。如果没有，`target` 不在树上。
- 如果 `target` 大于当前键，搜索右子树。如果没有，`target` 不在树上。

在树的每一层，你只需要搜索一个子树。例如，如果你在上图中查找 `target = 4`，则从根节点开始，它包含键 `8`。因为 `target` 小于 `8`，你走了左边。因为 `target` 大于 `3`，你走了右边。因为 `target` 小于 `6`，你走了左边。然后你找到你要找的键。

在这个例子中，即使树包含九个键，它需要四次比较来找到目标。一般来说，比较的数量与树的高度成正比，而不是树中的键的数量。

因此，我们可以计算树的高度 `h` 和节点个数 `n` 的关系。从小的数值开始，逐渐增加：

如果 `h=1`，树只包含一个节点，那么 `n=1`。

如果 `h=2`，我们可以添加两个节点，总共 `n=3`。

如果 `h=3`，我们可以添加多达四个节点，总共 `n=7`。

如果 `h=4`，我们可以添加多达八个节点，总共 `n=15`。

现在你可能会看到这个规律。如果我们将树的层数从 `1` 数到 `n`，第 `i` 层可以拥有多达  $2^{(i-1)}$  个节点。`h` 层的树共有  $2^{h-1}$  个节点。如果我们有：

```
1. n = 2^h - 1
```

我们可以对两边取以 `2` 为底的对数：

```
1. log2(n) ≈ h
```

意思是树的高度正比于  $\log n$ ，如果它是满的。也就是说，如果每一层包含最大数量的节点。

所以我们预计，我们可以以正比于  $\log n$  的时间，在二叉搜索树中查找节点。如果树是慢的，即使是部分满的，这是对的。但是并不总是对的，我们将会看到。

时间正比于  $\log n$  的算法是对数时间的，并且属于  $O(\log n)$  的增长级别。

## 12.3 练习 10

对于这个练习，你将要使用二叉搜索树编写 `Map` 接口的一个实现。

这里是实现的开头，叫做 `MyTreeMap`：

```
1. public class MyTreeMap<K, V> implements Map<K, V> {
2.
3.     private int size = 0;
```

```
4.     private Node root = null;
```

实例变量是 `size`，它跟踪了键的数量，以及 `root`，它是树中根节点的引用。树为空的时候，`root` 是 `null`，`size` 是 `0`。

这里是 `Node` 的定义，它在 `MyTreeMap` 之中定义。

```
1.     protected class Node {
2.         public K key;
3.         public V value;
4.         public Node left = null;
5.         public Node right = null;
6.
7.         public Node(K key, V value) {
8.             this.key = key;
9.             this.value = value;
10.        }
11.    }
```

每个节点包含一个键值对，以及两个子节点的引用，`left` 和 `right`。任意子节点都可以为 `null`。

一些 `Map` 方法易于实现，比如 `size` 和 `clear`：

```
1.     public int size() {
2.         return size;
3.     }
4.
5.     public void clear() {
6.         size = 0;
7.         root = null;
8.     }
```

`size` 显然是常数时间的。

`clear` 也是常数时间的，但是考虑这个：当 `root` 赋为 `null` 时，垃圾收集器回收了树中的节点，这是线性时间的。这个工作是否应该由垃圾收集器的计数来完成呢？我认为是的。

下一节中，你会填充一些其它方法，包括最重要的 `get` 和 `set`。

## 12.4 实现 `TreeMap`

这本书的仓库中，你将找到这些源文件：



- `MyTreeMap.java` 包含上一节的代码，其中包含缺失方法的大纲。
- `MyTreeMapTest.java` 包含单元 `MyTreeMap` 的测试。

运行 `ant build` 来编译源文件。然后运行 `ant MyTreeMapTest`。几个测试应该失败，因为你有一些工作要做！

我已经提供了 `get` 和 `containsKey` 的大纲。他们都使用 `findNode`，这是我定义的私有方法；它不是 `Map` 接口的一部分。以下是它的起始：

```
1.     private Node findNode(Object target) {
2.         if (target == null) {
3.             throw new IllegalArgumentException();
4.         }
5.
6.         @SuppressWarnings("unchecked")
7.         Comparable<? super K> k = (Comparable<? super K>) target;
8.
9.         // TODO: FILL THIS IN!
10.        return null;
11.    }
```

参数 `target` 是我们要查找的键。如果 `target` 是 `null`，`findNode` 抛出异常。一些 `Map` 实现可以将 `null` 处理为一个键，但是在二叉搜索树中，我们需要能够比较键，所以处理 `null` 是有问题的。为了保持简单，这个实现不将 `null` 视为键。

下一行显示如何将 `target` 与树中的键进行比较。按照 `get` 和 `containsKey` 的签名（名称和参数），编译器认为 `target` 是一个 `Object`。但是，我们需要能够对键进行比较，所以我们将 `target` 强制转换为 `Comparable<? super K>`，这意味着它可以与类型 `K`（或任何超类）的示例比较。如果你不熟悉“类型通配符”的用法，可以在 <http://thinkdast.com/gentut> 上阅读更多内容。

幸运的是，Java 的类型系统的处理不是这个练习的重点。你的工作是填写剩下的 `findNode`。如果它发现一个包含 `target` 键的节点，它应该返回该节点。否则应该返回 `null`。当你使其工作，`get` 和 `containsKey` 的测试应该通过。

请注意，你的解决方案应该只搜索通过树的一条路径，因此它应该与树的高度成正比。你不应该搜索整棵树！

你的下一个任务是填充 `containsValue`。为了让你起步，我提供了一个辅助方法 `equals`，比较 `target` 和给定的键。请注意，树中的值（与键相反）不一定是可比较的，所以我们不能使用 `compareTo`；我们必须在 `target` 上调用 `equals`。

不像你以前的 `findNode` 解决方案，你的 `containsValue` 解决方案应该搜索整个树，所以它的运行时间正比于键的数量 `n`，而不是树的高度 `h`。

译者注：这里你可能想使用之前讲过的 `DFS` 迭代器。

你应该填充的下一个方法是 `put`。我提供了处理简单情况的起始代码：

```

1.     public V put(K key, V value) {
2.         if (key == null) {
3.             throw new IllegalArgumentException();
4.         }
5.         if (root == null) {
6.             root = new Node(key, value);
7.             size++;
8.             return null;
9.         }
10.        return putHelper(root, key, value);
11.    }
12.
13.    private V putHelper(Node node, K key, V value) {
14.        // TODO: Fill this in.
15.    }

```

如果你尝试将 `null` 作为关键字，`put` 则会抛出异常。

如果树为空，则 `put` 创建一个新节点并初始化实例变量 `root`。

否则，它调用 `putHelper`，这是我定义的私有方法；它不是 `Map` 接口的一部分。

填写 `putHelper`，让它搜索树，以及：

- 如果 `key` 已经在树中，它将使用新值替换旧值，并返回旧值。
- 如果 `key` 不在树中，它将创建一个新节点，找到正确的添加位置，并返回 `null`。

你的 `put` 实现的是时间应该与树的高度 `h` 成正比，而不是元素的数量 `n`。理想情况下，你只需搜索一次树，但如果你发现两次更容易搜索，可以这样做：它会慢一些，但不会改变增长级别。

最后，你应该填充 `keySet`。根据 <http://thinkdast.com/mapkeyset> 的文档，该方法应该返回一个 `Set`，可以按顺序迭代键；也就是说，按照 `compareTo` 方法，升序迭代。我们在 8.3 节中使用的 `HashSet` 实现不会维护键的顺序，但 `LinkedHashSet` 实现可以。你可以阅读 <http://thinkdast.com/linkedhashset>。

我提供了一个 `keySet` 的大纲，创建并返回 `LinkedHashSet`：

```

1.     public Set<K> keySet() {
2.         Set<K> set = new LinkedHashSet<K>();
3.         return set;
4.     }

```

你应该完成此方法，使其以升序向 `set` 添加树中的键。提示：你可能想编写一个辅助程序；你可能

想让它递归；你也可能想要阅读 <http://thinkdast.com/inorder> 上的树的中序遍历。

当你完成时，所有测试都应该通过。下一章中，我会讲解我的解法，并测试核心方法的性能。

## 第十三章 二叉搜索树

- 第十三章 二叉搜索树
  - 13.1 简单的 `MyTreeMap`
  - 13.2 搜索值
  - 13.3 实现 `put`
  - 13.4 中序遍历
  - 13.5 对数时间的方法
  - 13.6 自平衡树
  - 13.7 更多练习

## 第十三章 二叉搜索树

原文: [Chapter 13 Binary search tree](#)

译者: 飞龙

协议: [CC BY-NC-SA 4.0](#)

自豪地采用[谷歌翻译](#)

本章介绍了上一个练习的解决方案，然后测试树形映射的性能。我展示了一个实现的问题，并解释了 Java 的 `TreeMap` 如何解决它。

### 13.1 简单的 `MyTreeMap`

上一个练习中，我给了你 `MyTreeMap` 的大纲，并让你填充缺失的方法。现在我会展示结果，从 `findNode` 开始：

```
1. private Node findNode(Object target) {
2.     // some implementations can handle null as a key, but not this one
3.     if (target == null) {
4.         throw new IllegalArgumentException();
5.     }
6.
7.     // something to make the compiler happy
8.     @SuppressWarnings("unchecked")
9.     Comparable<? super K> k = (Comparable<? super K>) target;
10.
11.    // the actual search
12.    Node node = root;
13.    while (node != null) {
14.        int cmp = k.compareTo(node.key);
```

```

15.         if (cmp < 0)
16.             node = node.left;
17.         else if (cmp > 0)
18.             node = node.right;
19.         else
20.             return node;
21.     }
22.     return null;
23. }

```

`findNode` 是 `containsKey` 和 `get` 所使用的一个私有方法；它不是 `Map` 接口的一部分。参数 `target` 是我们要查找的键。我在上一个练习中解释了这种方法的第一部分：

- 在这个实现中，`null` 不是键的合法值。
- 在我们可以 `target` 上调用 `compareTo` 之前，我们必须把它强制转换为某种形式的 `Comparable`。这里使用的“类型通配符”会尽可能允许；也就是说，它适用于任何实现 `Comparable` 类型，并且它的 `compareTo` 接受 `K` 或者任何 `K` 的超类。

之后，实际搜索比较简单。我们初始化一个循环变量 `node` 来引用根节点。每次循环中，我们将目标与 `node.key` 比较。如果目标小于当前键，我们移动到左子树。如果它更大，我们移动到右子树。如果相等，我们返回当前节点。

如果在没有找到目标的情况下，我们到达树的底部，我就认为，它不在树中并返回 `null`。

## 13.2 搜索值

我在前面的练习中解释了，`findNode` 运行时间与树的高度成正比，而不是节点的数量，因为我们不必搜索整个树。但是对于 `containsValue`，我们必须搜索值，而不是键；BST 的特性不适用于值，因此我们必须搜索整个树。

我的解法是递归的：

```

1. public boolean containsValue(Object target) {
2.     return containsValueHelper(root, target);
3. }
4.
5. private boolean containsValueHelper(Node node, Object target) {
6.     if (node == null) {
7.         return false;
8.     }
9.     if (equals(target, node.value)) {
10.        return true;
11.    }
12.    if (containsValueHelper(node.left, target)) {
13.        return true;

```

```

14.     }
15.     if (containsValueHelper(node.right, target)) {
16.         return true;
17.     }
18.     return false;
19. }

```

`containsValue` 将目标值作为参数，并立即调用 `containsValueHelper`，传递树的根节点作为附加参数。

这是 `containsValueHelper` 的工作原理：

- 第一个 `if` 语句检查递归的边界情况。如果 `node` 是 `null`，那意味着我们已经递归到树的底部，没有找到 `target`，所以我们应该返回 `false`。请注意，这只意味着目标没有出现在树的一条路径上；它仍然可能会在另一条路径上被发现。
- 第二种情况检查我们是否找到了我们正在寻找的东西。如果是这样，我们返回 `true`。否则，我们必须继续。
- 第三种情况是执行递归调用，在左子树中搜索 `target`。如果我们找到它，我们可以立即返回 `true`，而不搜索右子树。否则我们继续。
- 第四种情况是搜索右子树。同样，如果我们找到我们正在寻找的东西，我们返回 `true`。否则，我们搜索完了整棵树，返回 `false`。

该方法“访问”了树中的每个节点，所以它的所需时间与节点数成正比。

## 13.3 实现 `put`

`put` 方法比起 `get` 要复杂一些，因为要处理两种情况：（1）如果给定的键已经在树中，则替换并返回旧值；（2）否则必须在树中添加一个新的节点，在正确的地方。

在上一个练习中，我提供了这个起始代码：

```

1. public V put(K key, V value) {
2.     if (key == null) {
3.         throw new IllegalArgumentException();
4.     }
5.     if (root == null) {
6.         root = new Node(key, value);
7.         size++;
8.         return null;
9.     }
10.    return putHelper(root, key, value);
11. }

```

并且让你填充 `putHelper`。这里是我的答案：

```

1. private V putHelper(Node node, K key, V value) {
2.     Comparable<? super K> k = (Comparable<? super K>) key;
3.     int cmp = k.compareTo(node.key);
4.
5.     if (cmp < 0) {
6.         if (node.left == null) {
7.             node.left = new Node(key, value);
8.             size++;
9.             return null;
10.        } else {
11.            return putHelper(node.left, key, value);
12.        }
13.    }
14.    if (cmp > 0) {
15.        if (node.right == null) {
16.            node.right = new Node(key, value);
17.            size++;
18.            return null;
19.        } else {
20.            return putHelper(node.right, key, value);
21.        }
22.    }
23.    V oldValue = node.value;
24.    node.value = value;
25.    return oldValue;
26. }

```

第一个参数 `node` 最初是树的根，但是每次我们执行递归调用，它指向了不同的子树。就像 `get` 一样，我们用 `compareTo` 方法来弄清楚，跟随哪一条树的路径。如果 `cmp < 0`，我们添加的键小于 `node.key`，那么我们要走左子树。有两种情况：

- 如果左子树为空，那就是，如果 `node.left` 是 `null`，我们已经到达树的底部而没有找到 `key`。这个时候，我们知道 `key` 不在树上，我们知道它应该放在哪里。所以我们创建一个新节点，并将它添加为 `node` 的左子树。
- 否则我们进行递归调用来搜索左子树。

如果 `cmp > 0`，我们添加的键大于 `node.key`，那么我们要走右子树。我们处理的两个案例与上一个分支相同。最后，如果 `cmp == 0`，我们在树中找到了键，那么我们更改它并返回旧的值。

我使用递归编写了这个方法，使它更易于阅读，但它可以直接用迭代重写一遍，你可能想留作练习。

## 13.4 中序遍历

我要求你编写的最后一个方法是 `keySet`，它返回一个 `Set`，按升序包含树中的键。在其他 `Map` 实现中，`keySet` 返回的键没有特定的顺序，但是树形实现的一个功能是，对键进行简单而

有效的排序。所以我们应该利用它。

这是我的答案：

```

1. public Set<K> keySet() {
2.     Set<K> set = new LinkedHashSet<K>();
3.     addInOrder(root, set);
4.     return set;
5. }
6.
7. private void addInOrder(Node node, Set<K> set) {
8.     if (node == null) return;
9.     addInOrder(node.left, set);
10.    set.add(node.key);
11.    addInOrder(node.right, set);
12. }

```

在 `keySet` 中，我们创建一个 `LinkedHashSet`，这是一个 `Set` 实现，使元素保持有序（与大多数其他 `Set` 实现不同）。然后我们调用 `addInOrder` 来遍历树。

第一个参数 `node` 最初是树的根，但正如你的期望，我们用它来递归地遍历树。`addInOrder` 对树执行经典的“中序遍历”。

如果 `node` 是 `null`，这意味着子树是空的，所以我们返回，而不向 `set` 添加任何东西。否则我们：

1. 按顺序遍历左子树。
2. 添加 `node.key`。
3. 按顺序遍历右子树。

请记住，BST 的特性保证左子树中的所有节点都小于 `node.key`，并且右子树中的所有节点都更大。所以我们知道，`node.key` 已按正确的顺序添加。

递归地应用相同的参数，我们知道左子树中的元素是有序的，右子树中的元素也一样。并且边界情况是正确的：如果子树为空，则不添加任何键。所以我们可以认为，该方法以正确的顺序添加所有键。

因为 `containsValue` 方法访问树中的每个节点，所以所需时间与 `n` 成正比。

## 13.5 对数时间的方法

在 `MyTreeMap` 中，`get` 和 `put` 方法所需时间与树的高度 `h` 成正比。在上一个练习中，我们展示了如果树是满的 - 如果树的每一层都包含最大数量的节点 - 树的高度与 `log n` 成横臂。

我也说了，`get` 和 `put` 是对数时间的；也就是说，他们的所需时间与 `log n` 成正比。但是对于大多数应用程序，不能保证树是满的。一般来说，树的形状取决于键和添加顺序。



为了看看这在实践中是怎么回事，我们将用两个样本数据集来测试我们的实现：随机字符串的列表和升序的时间戳列表。

这是生成随机字符串的代码：

```
1. Map<String, Integer> map = new MyTreeMap<String, Integer>();
2.
3. for (int i=0; i<n; i++) {
4.     String uuid = UUID.randomUUID().toString();
5.     map.put(uuid, 0);
6. }
```

`UUID` 是 `java.util` 中的类，可以生成随机的“通用唯一标识符”。`UUID` 对于各种应用是有用的，但在这个例子中，我们利用一种简单的方法来生成随机字符串。

我使用 `n=16384` 来运行这个代码，并测量了最后的树的运行时间和高度。以下是输出：

```
1. Time in milliseconds = 151
2. Final size of MyTreeMap = 16384
3. log base 2 of size of MyTreeMap = 14.0
4. Final height of MyTreeMap = 33
```

我包含了“`MyTreeMap` 大小的 `2` 为底的对数”，看看如果它已满，树将是多高。结果表明，高度为 `14` 的完整树包含 `16384` 个节点。

随机字符串的树高度实际为33，这远大于理论上的最小值，但不是太差。要查找 `16,384` 个键中的一个，我们只需要进行 `33` 次比较。与线性搜索相比，速度快了近 `500` 倍。

这种性能通常是随机字符串，或其他不按照特定顺序添加的键。树的最终高度可能是理论最小值的 `2~3` 倍，但它仍然与 `log n` 成正比，这远小于 `n`。事实上，随着 `n` 的增加，`log n` 会慢慢增加，在实践中，可能很难将对数时间与常数时间区分开。

然而，二叉搜索树并不总是表现良好。让我们看看，当我们以升序添加键时会发生什么。下面是一个示例，以微秒为单位测量时间戳，并将其用作键：

```
1. MyTreeMap<String, Integer> map = new MyTreeMap<String, Integer>();
2.
3. for (int i=0; i<n; i++) {
4.     String timestamp = Long.toString(System.nanoTime());
5.     map.put(timestamp, 0);
6. }
```

`System.nanoTime` 返回一个 `long` 类型的整数，表示以微秒为单位的启动时间。每次我们调用它时，我们得到一个更大的数字。当我们将这些时间戳转换为字符串时，它们按字典序增加。

让我们看看当我们运行它时会发生什么：

```
1. Time in milliseconds = 1158
2. Final size of MyTreeMap = 16384
3. log base 2 of size of MyTreeMap = 14.0
4. Final height of MyTreeMap = 16384
```

运行时间是以前的时间的七倍多。时间更长。如果你想知道为什么，看看树的最后的高度： 16384 ！

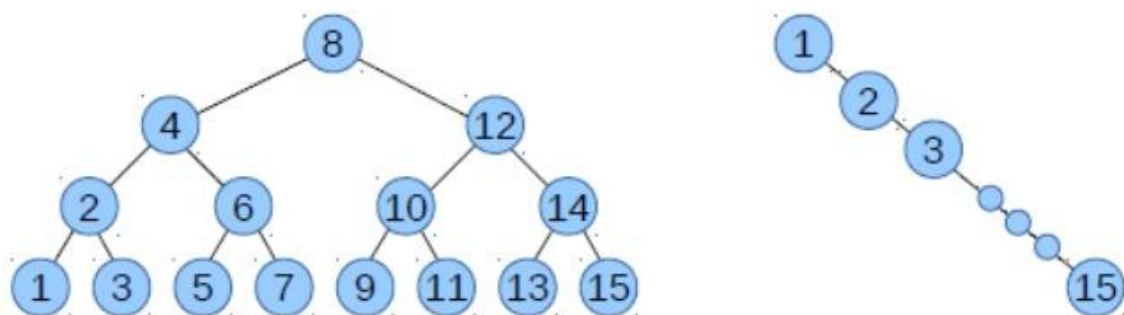


图 13.1：二叉搜索树，平衡（左边）和不平衡（右边）

如果你思考 `put` 如何工作，你可以弄清楚发生了什么。每次添加一个新的键时，它都大于树中的所有键，所以我们总是选择右子树，并且总是将新节点添加为，最右边的节点的右子节点。结果是一个“不平衡”的树，只包含右子节点。

这种树的高度正比于 `n`，不是 `log n`，所以 `get` 和 `put` 的性能是线性的，不是对数的。

图 13.1 显示了平衡和不平衡树的示例。在平衡树中，高度为 4，节点总数为  $2^4 - 1 = 15$ 。在节点数相同的不平衡树中，高度为 15。

## 13.6 自平衡树

这个问题有两种可能的解决方案：

你可以避免向 `Map` 按顺序添加键。但这并不总是可能的。

你可以制作一棵树，如果碰巧按顺序处理键，那么它会更好地处理键。

第二个解决方案是更好的，有几种方法可以做到。最常见的是修改 `put`，以便它检测树何时开始变得不平衡，如果是，则重新排列节点。具有这种能力的树被称为“自平衡树”。普通的自平衡树包括 AVL 树（“AVL”是发明者的缩写），以及红黑树，这是 Java `TreeMap` 所使用的。

在我们的示例代码中，如果我们用 Java 的 `MyTreeMap` 替换，随机字符串和时间戳的运行时间大致

相同。实际上，时间戳运行速度更快，即使它们有序，可能是因为它们花费的时间较少。

总而言之，二叉搜索树可以以对数时间实现 `get` 和 `put`，但是只能按照使得树足够平衡的顺序添加键。自平衡树通过每次添加新键时，进行一些额外的工作来避免这个问题。

你可以在 <http://thinkdast.com/balancing> 上阅读自平衡树的更多信息。

## 13.7 更多练习

---

在上一个练习中，你不必实现 `remove`，但你可能需要尝试。如果从树中央删除节点，则必须重新排列剩余的节点，来恢复 BST 的特性。你可以自己弄清楚如何实现，或者你可以阅读 <http://thinkdast.com/bstdel> 上的说明。

删除一个节点并重新平衡一个树是类似的操作：如果你做这个练习，你将更好地了解自平衡树如何工作。

## 第十四章 持久化

- [第十四章 持久化](#)
  - [14.1 Redis](#)
  - [14.2 Redis 客户端和服务端](#)
  - [14.3 制作基于 Redis 的索引](#)
  - [14.4 Redis 数据类型](#)
  - [14.5 练习 11](#)
  - [14.6 更多建议（如果你需要的话）](#)
  - [14.7 几个设计提示](#)

## 第十四章 持久化

原文: [Chapter 14 Persistence](#)

译者: [飞龙](#)

协议: [CC BY-NC-SA 4.0](#)

自豪地采用[谷歌翻译](#)

在接下来的几个练习中，我们将返回到网页搜索引擎的构建。为了回顾，搜索引擎的组件是：

- 抓取：我们需要一个程序，可以下载一个网页，解析它，并提取文本和任何其他页面的链接。
- 索引：我们需要一个索引，可以查找检索项并找到包含它的页面。
- 检索：我们需要一种方法，从索引中收集结果，并识别与检索项最相关的页面。

如果你做了练习 8.3，你使用 Java 映射实现了一个索引。在本练习中，我们将重新审视索引器，并创建一个新版本，将结果存储在数据库中。

如果你做了练习 7.4，你创建了一个爬虫，它跟踪它找到的第一个链接。在下一个练习中，我们将制作一个更通用的版本，将其查找到的每个链接存储在队列中，并对其进行排序。

然后，最后，你将处理检索问题。

在这些练习中，我提供较少的起始代码，你将做出更多的设计决策。这些练习也更加开放。我会提出一些最低限度的目标，你应该尝试实现它们，但如果你想挑战自己，有很多方法可以让你更深入。

现在，让我们开始编写一个新版本的索引器。

### 14.1 Redis

索引器的之前版本，将索引存储在两个数据结构中：`TermCounter` 将检索词映射为网页上显示的次

数，以及 `Index` 将检索词映射为出现的页面集合。

这些数据结构存储在正在运行的 Java 程序的内存中，这意味着当程序停止运行时，索引会丢失。仅在运行程序的内存中存储的数据称为“易失的”，因为程序结束时消失。

在创建它的程序结束后，仍然存在的数据称为“持久的”。通常，存储在文件系统中的文件，以及存储在数据库中的数据是持久的。

使数据持久化的一种简单方法是，将其存储在文件中。在程序结束之前，它可以将其数据结构转换为 JSON 格式 (<http://thinkdast.com/json>)，然后将它们写入文件。当它再次启动时，它可以读取文件并重建数据结构。

但这个解决方案有几个问题：

- 读取和写入大型数据结构（如 Web 索引）会很慢。
- 整个数据结构可能不适合单个运行程序的内存。
- 如果程序意外结束（例如，由于断电），则自程序上次启动以来所做的任何更改都将丢失。

一个更好的选择是提供持久存储的数据库，并且能够读取和写入数据库的部分，而无需读取和写入整个数据。

有多种数据库管理系统（DBMS）提供不同的功能。你可以在 <http://thinkdast.com/database> 阅读概述。

我为这个练习推荐的数据库是 Redis，它提供了类似于 Java 数据结构的持久数据结构。具体来说，它提供：

字符串列表，与 Java 的 `List` 类似。

哈希，类似于 Java 的 `Map`。

字符串集合，类似于 Java 的 `Set`。

译者注：另外还有类似于 Java 的 `LinkedHashSet` 的有序集合。

Redis 是一个“键值数据库”，这意味着它包含的数据结构（值）由唯一的字符串（键）标识。Redis 中的键与 Java 中的引用相同：它标识一个对象。我们稍后会看到一些例子。

## 14.2 Redis 客户端和服务端

Redis 通常运行于远程服务；其实它的名字代表“REmote DIctionary Server”（远程字典服务，字典其实就是映射）。为了使用 Redis，你必须在某处运行 Redis 服务器，然后使用 Redis 客户端连接到 Redis 服务器。有很多方法可用于设置服务器，也有许多你可以使用的客户端。对于这个练习，我建议：

不要自己安装和运行服务器，请考虑使用像

RedisToGo (<http://thinkdast.com/redistogo>) 这样的服务，它在云主机运行 Redis。他

们提供了一个免费的计划（配置），有足够的资源用于练习。

对于客户端，我推荐 Jedis，它是一个 Java 库，提供了使用 Redis 的类和方法。

以下是更详细的说明，以帮助你开始使用：

- 在 RedisToGo 上创建一个帐号，网址为 <http://thinkdast.com/redissign>，并选择所需的计划（可能是免费的起始计划）。
- 创建一个“实例”，它是运行 Redis 服务器的虚拟机。如果你单击“实例”选项卡，你将看到你的新实例，由主机名和端口号标识。例如，我有一个名为 `dory-10534` 的实例。
- 单击实例名称来访问配置页面。记下页面顶部附近的网址，如下所示：

```
1. redis://redistogo:1234567feedfacebeefale1234567@dory.redistogo.com:10534
```

这个 URL 包含服务器的主机名称 `dory.redistogo.com`，端口号 `10534` 和连接到服务器所需的密码，它是中间较长的字母数字的字符串。你将需要此信息进行下一步。

## 14.3 制作基于 Redis 的索引

在本书的仓库中，你将找到此练习的源文件：

- `JedisMaker.java` 包含连接到 Redis 服务器并运行几个 Jedis 方法的示例代码。
- `JedisIndex.java` 包含此练习的起始代码。
- `JedisIndexTest.java` 包含 `JedisIndex` 的测试代码。
- `WikiFetcher.java` 包含我们在以前的练习中看到的代码，用于阅读网页并使用 `jsoup` 进行解析。

你还将需要这些文件，你在以前的练习中碰到过：

`Index.java` 使用 Java 数据结构实现索引。

`TermCounter.java` 表示从检索项到其频率的映射。

`WikiNodeIterable.java` 迭代 `jsoup` 生成的 DOM 树中的节点。

如果你有这些文件的有效版本，你可以使用它们进行此练习。如果你没有进行以前的练习，或者你对你的解决方案毫无信心，则可以从 `solutions` 文件夹复制我的解决方案。

第一步是使用 Jedis 连接到你的 Redis 服务器。`JedisMaker.java` 展示了如何实现。它从文件读取你的 Redis 服务器的信息，连接到它并使用你的密码登录，然后返回一个可用于执行 Redis 操作的 Jedis 对象。

如果你打开 `JedisMaker.java`，你应该看到 `JedisMaker` 类，它是一个帮助类，它提供静态方法 `make`，它创建一个 Jedis 对象。一旦该对象认证完毕，你可以使用它与你的 Redis 数据库进行通信。

`JedisMaker` 从名为 `redis_url.txt` 的文件读取你的 Redis 服务器信息，你应该放在目录 `src/resources` 中：

- 使用文本编辑器创建并编辑 `ThinkDataStructures/code/src/resources/redis_url.txt`。
- 粘贴服务器的 URL。如果你使用的是 RedisToGo, 则 URL 将如下所示:

```
1. redis://redistogo:1234567feedfacebeefa1e1234567@dory.redistogo.com:10534
```

因为此文件包含你的 Redis 服务器的密码, 你不应将此文件放在公共仓库中。为了帮助你避免意外避免这种情况, 仓库包含 `.gitignore` 文件, 使文件难以 (但不是不可能) 放入你的仓库。

现在运行 `ant build` 来编译源文件, 以及 `ant JedisMaker` 来运行 `main` 中的示例代码:

```
1.  public static void main(String[] args) {
2.
3.     Jedis jedis = make();
4.
5.     // String
6.     jedis.set("mykey", "myvalue");
7.     String value = jedis.get("mykey");
8.     System.out.println("Got value: " + value);
9.
10.    // Set
11.    jedis.sadd("myset", "element1", "element2", "element3");
12.    System.out.println("element2 is member: " +
13.        jedis.sismember("myset", "element2"));
14.
15.    // List
16.    jedis.rpush("mylist", "element1", "element2", "element3");
17.    System.out.println("element at index 1: " +
18.        jedis.lindex("mylist", 1));
19.
20.    // Hash
21.    jedis.hset("myhash", "word1", Integer.toString(2));
22.    jedis.hincrBy("myhash", "word2", 1);
23.    System.out.println("frequency of word1: " +
24.        jedis.hget("myhash", "word1"));
25.    System.out.println("frequency of word1: " +
26.        jedis.hget("myhash", "word2"));
27.
28.    jedis.close();
29. }
```

这个示例展示了数据类型和方法, 你在这个练习中最可能使用它们。当你运行它时, 输出应该是:

```
1. Got value: myvalue
2. element2 is member: true
3. element at index 1: element2
4. frequency of word1: 2
5. frequency of word2: 1
```

下一节中我会解释代码的工作原理。

## 14.4 Redis 数据类型

Redis 基本上是一个从键到值的映射，键是字符串，值可以是字符串，也可以是几种数据类型之一。最基本的 Redis 数据类型是字符串。我将用斜体书写 Redis 类型，来区别于 Java 类型。

为了向数据库添加一个字符串，请使用 `jedis.set`，类似于 `Map.put`；参数是新的键和相应的值。为了查找一个键并获取其值，请使用 `jedis.get`：

```
1. jedis.set("mykey", "myvalue");
2. String value = jedis.get("mykey");
```

在这个例子中，键是 `"mykey"`，值是 `"myvalue"`。

Redis 提供了一个集合结构，类似于 Java 的 `Set<String>`。为了向 Redis 集合添加元素，你可以选择一个键来标识集合，然后使用 `jedis.sadd`：

```
1. jedis.sadd("myset", "element1", "element2", "element3");
2. boolean flag = jedis.sismember("myset", "element2");
```

你不必用单独的步骤来创建集合。如果不存在，Redis 会创建它。在这种情况下，它会创建一个名为 `myset` 的集合，包含三个元素。

`jedis.sismember` 方法检查元素是否在一个集合中。添加元素和检查成员是常数时间的操作。

Redis 还提供了一个列表结构，类似于 Java 的 `List<String>`。`jedis.rpush` 方法在末尾（右端）向列表添加元素：

```
1. jedis.rpush("mylist", "element1", "element2", "element3");
2. String element = jedis.lindex("mylist", 1);
```

同样，你不必在开始添加元素之前创建结构。此示例创建了一个名为 `mylist` 的列表，其中包含三个元素。

`jedis.lindex` 方法使用整数索引，并返回列表中指定的元素。添加和访问元素是常数时间的操作。

最后，Redis 提供了一个哈希结构，类似于 Java 的 `Map<String, String>`。`jedis.hset` 方法为哈希表添加新条目：

```
1. jedis.hset("myhash", "word1", Integer.toString(2));
```



```
2.      String value = jedis.hget("myhash", "word1");
```

此示例创建一个名为的 `myhash` 哈希表，其中包含一个条目，该条目从键 `word1` 映射到值 `"2"`。

键和值都是字符串，所以如果我们要存储 `Integer`，在我们调用 `hset` 之前，我们必须将它转换为 `String`。当我们使用 `hget` 查找值时，结果是 `String`，所以我们可能必须将其转换回 `Integer`。

使用 Redis 的哈希表可能会令人困惑，因为我们使用一个键来标识我们想要的哈希表，然后用另一个键标识哈希表中的值。在 Redis 的上下文中，第二个键被称为“字段”，这可能有助于保持清晰。所以类似 `myhash` 的“键”标志一个特定的哈希表，然后类似 `word1` 的“字段”标识一个哈希表中的值。

对于许多应用程序，Redis 哈希表中的值是整数，所以 Redis 提供了一些特殊的方法，比如 `hincrby` 将值作为数字来处理：

```
1.      jedis.hincrBy("myhash", "word2", 1);
```

这个方法访问 `myhash`，获取 `word2` 的当前值（如果不存在则为 `0`），将其递增 `1`，并将结果写回哈希表。

在哈希表中，设置，获取和递增条目是常数时间的操作。

你可以在 <http://thinkdast.com/redistypes> 上阅读 Redis 数据类型的更多信息。

## 14.5 练习 11

这个时候，你可以获取一些信息，你需要使用它们来创建搜索引擎的索引，它将结果储存在 Redis 数据库中。

现在运行 `ant JedisIndexTest`。它应该失败，因为你有一些工作要做！

`JedisIndexTest` 测试了这些方法：

- `JedisIndex`，这是构造器，它接受 `Jedis` 对象作为参数。
- `indexPage`，它将一个网页添加到索引中；它需要一个 `StringURL` 和一个 `Jsoup Elements` 对象，该对象包含应该建立索引的页面元素。
- `getCounts`，它接收检索词，并返回 `Map<String, Integer>`，包含检索词到它在页面上的出现次数的映射。

以下是如何使用这些方法的示例：

```

1.      WikiFetcher wf = new WikiFetcher();
2.      String url1 =
3.          "http://en.wikipedia.org/wiki/Java_(programming_language)";
4.      Elements paragraphs = wf.readWikipedia(url1);
5.
6.      Jedis jedis = JedisMaker.make();
7.      JedisIndex index = new JedisIndex(jedis);
8.      index.indexPage(url1, paragraphs);
9.      Map<String, Integer> map = index.getCounts("the");

```

如果我们在结果 `map` 中查看 `url1`，我们应该得到 `339`，这是 Java 维基百科页面（即我们保存的版本）中，`the` 出现的次数。

如果我们再次索引相同的页面，新的结果将替换旧的结果。

将数据结构从 Java 翻译成 Redis 的一个建议是：记住 Redis 数据库中的每个对象都以唯一的键标识，它是一个字符串。如果同一数据库中有两种对象，则可能需要向键添加前缀来区分它们。例如，在我们的解决方案中，我们有两种对象：

- 我们将 `URLSet` 定义为 Redis 集合，它包含 `URL`，`URL` 又包含给定检索词。每个 `URLSet` 的键的起始是 `"URLSet:"`，所以要获取包含单词 `the` 的 `URL`，我们使用键 `"URLSet:the"` 来访问该集合。
- 我们将 `TermCounter` 定义为 Redis 哈希表，将出现在页面上的每个检索词映射到它的出现次数。`TermCounter` 每个键的开头都以 `"TermCounter:"` 开头，以我们正在查找的页面的 `URL` 结尾。

在我的实现中，每个检索词都有一个 `URLSet`，每个索引页面都有一个 `TermCounter`。我提供两个辅助方法，`urlSetKey` 和 `termCounterKey` 来组装这些键。

## 14.6 更多建议（如果你需要的话）

到了这里，你拥有了完成练习所需的所有信息，所以如果准备好了就可以开始了。但是我有几个建议，你可能想先阅读它：

- 对于这个练习，我提供的指导比以前的练习少。你必须做出一些设计决策；特别是，你将必须弄清楚如何将问题分解成，你可以一次性测试的部分，然后将这些部分组合成一个完整的解决方案。如果你尝试一次写出整个项目，而不测试较小的部分，调试可能需要很长时间。
- 使用持久性数据的挑战之一是它是持久的。存储在数据库中的结构可能会在每次运行程序时发生更改。如果你弄乱了数据库，你将不得不修复它或重新开始，然后才能继续。为了帮助你控制住自己，我提供的方法叫 `deleteURLSets`，`deleteTermCounters` 和 `deleteAllKeys`，你可以用它来清理数据库，并重新开始。你也可以使用 `printIndex` 来打印索引的内容。
- 每次调用 `Jedis` 的方法时，你的客户端会向服务器发送一条消息，然后服务器执行你请求的操作并返回消息。如果执行许多小操作，可能需要很长时间。你可以通过将一系列操作分组为一

个 `Transaction`，来提高性能。

例如，这是一个简单的 `deleteAllKeys` 版本：

```
1. public void deleteAllKeys() {
2.     Set<String> keys = jedis.keys("*");
3.     for (String key: keys) {
4.         jedis.del(key);
5.     }
6. }
```

每次调用 `del` 时，都需要从客户端到服务器的双向通信。如果索引包含多个页面，则该方法需要很长时间来执行。我们可以使用 `Transaction` 对象来加速：

```
1. public void deleteAllKeys() {
2.     Set<String> keys = jedis.keys("*");
3.     Transaction t = jedis.multi();
4.     for (String key: keys) {
5.         t.del(key);
6.     }
7.     t.exec();
8. }
```

`jedis.multi` 返回一个 `Transaction` 对象，它提供 `Jedis` 对象的所有方法。但是当你调用 `Transaction` 的方法时，它不会立即执行该操作，并且不与服务器通信。在你调用 `exec` 之前，它会保存一批操作。然后它将所有保存的操作同时发送到服务器，这通常要快得多。

## 14.7 几个设计提示

现在你真的拥有了你需要的所有信息；你应该开始完成练习。但是如果你卡住了，或者如果你真的不知道如何开始，你可以再来一些提示。

在运行测试代码之前，不要阅读以下内容，尝试一些基本的 Redis 命令，并在 `JedisIndex.java` 中编写几个方法。

好的，如果你真的卡住了，这里有一些你可能想要处理的方法：

```
1. /**
2.  * 向检索词相关的集合中添加 URL
3.  */
4. public void add(String term, TermCounter tc) {}
5.
6. /**
7.  * 查找检索词并返回 URL 集合
```

```
8.      */
9.      public Set<String> getURLs(String term) {}
10.
11.     /**
12.      * 返回检索词出现在给定 URL 中的次数
13.      */
14.     public Integer getCount(String url, String term) {}
15.
16.     /**
17.      * 将 TermCounter 的内容存入 Redis
18.      */
19.     public List<Object> pushTermCounterToRedis(TermCounter tc) {}
```

这些是我在解决方案中使用的方法，但它们绝对不是将项目分解的唯一方法。所以如果他们有帮助，请接受这些建议，但是如果没有，请忽略它们。

对于每种方法，请考虑首先编写测试。当你弄清楚如何测试一个方法时，你经常会了解如何编写它。

祝你好运！

## 第十五章 爬取维基百科

- 第十五章 爬取维基百科
  - 15.1 基于 Redis 的索引器
  - 15.2 查找的分析
  - 15.3 索引的分析
  - 15.4 图的遍历
  - 15.5 练习 12

## 第十五章 爬取维基百科

原文: [Chapter 15 Crawling Wikipedia](#)

译者: 飞龙

协议: [CC BY-NC-SA 4.0](#)

自豪地采用谷歌翻译

在本章中，我展示了上一个练习的解决方案，并分析了 Web 索引算法的性能。然后我们构建一个简单的 Web 爬虫。

### 15.1 基于 Redis 的索引器

在我的解决方案中，我们在 Redis 中存储两种结构：

- 对于每个检索词，我们有一个 `URLSet`，它是一个 Redis 集合，包含检索词的 URL。
- 对于每个网址，我们有一个 `TermCounter`，这是一个 Redis 哈希表，将每个检索词映射到它出现的次数。

我们在上一章讨论了这些数据类型。你还可以在 <http://thinkdast.com/redistypes> 上阅读 Redis `Set` 和 `Hash` 的信息

在 `JedisIndex` 中，我提供了一个方法，它可以接受一个检索词并返回 Redis 中它的 `URLSet` 的键：

```
1. private String urlSetKey(String term) {
2.     return "URLSet:" + term;
3. }
```

以及一个方法，接受 URL 并返回 Redis 中它的 `TermCounter` 的键。

```
1. private String termCounterKey(String url) {
```

```

2.     return "TermCounter:" + url;
3. }

```

这里是 `indexPath` 的实现。

```

1. public void indexPath(String url, Elements paragraphs) {
2.     System.out.println("Indexing " + url);
3.
4.     // make a TermCounter and count the terms in the paragraphs
5.     TermCounter tc = new TermCounter(url);
6.     tc.processElements(paragraphs);
7.
8.     // push the contents of the TermCounter to Redis
9.     pushTermCounterToRedis(tc);
10. }

```

为了索引页面，我们：

- 为页面内容创建一个 Java 的 `TermCounter`，使用上一个练习中的代码。
- 将 `TermCounter` 的内容推送到 Redis。

以下是将 `TermCounter` 的内容推送到 Redis 的新代码：

```

1. public List<Object> pushTermCounterToRedis(TermCounter tc) {
2.     Transaction t = jedis.multi();
3.
4.     String url = tc.getLabel();
5.     String hashname = termCounterKey(url);
6.
7.     // if this page has already been indexed, delete the old hash
8.     t.del(hashname);
9.
10.    // for each term, add an entry in the TermCounter and a new
11.    // member of the index
12.    for (String term: tc.keySet()) {
13.        Integer count = tc.get(term);
14.        t.hset(hashname, term, count.toString());
15.        t.sadd(urlSetKey(term), url);
16.    }
17.    List<Object> res = t.exec();
18.    return res;
19. }

```

该方法使用 `Transaction` 来收集操作，并将它们一次性发送到服务器，这比发送一系列较小操作要快得多。

它遍历 `TermCounter` 中的检索词。对于每一个，它：

- 在 Redis 上寻找或者创建 `TermCounter`，然后为新的检索词添加字段。
- 在 Redis 上寻找或创建 `URLSet`，然后添加当前的 URL。

如果页面已被索引，则 `TermCounter` 在推送新内容之前删除旧页面。

新的页面的索引就是这样。

练习的第二部分要求你编写 `getCounts`，它需要一个检索词，并从该词出现的每个网址返回一个映射。这是我的解决方案：

```
1. public Map<String, Integer> getCounts(String term) {
2.     Map<String, Integer> map = new HashMap<String, Integer>();
3.     Set<String> urls = getURLs(term);
4.     for (String url: urls) {
5.         Integer count = getCount(url, term);
6.         map.put(url, count);
7.     }
8.     return map;
9. }
```

此方法使用两种辅助方法：

- `getURLs` 接受检索词并返回该字词出现的网址集合。
- `getCount` 接受 URL 和检索词，并返回该检索词在给定 URL 处显示的次数。

以下是实现：

```
1. public Set<String> getURLs(String term) {
2.     Set<String> set = jedis.smembers(urlSetKey(term));
3.     return set;
4. }
5.
6. public Integer getCount(String url, String term) {
7.     String redisKey = termCounterKey(url);
8.     String count = jedis.hget(redisKey, term);
9.     return new Integer(count);
10. }
```

由于我们设计索引的方式，这些方法简单而高效。

## 15.2 查找的分析

假设我们索引了 `N` 个页面，并发现了 `M` 个唯一的检索词。检索词的查询需要多长时间？在继续之

前，先考虑一下你的答案。

要查找一个检索词，我们调用 `getCounts`，其中：

- 创建映射。
- 调用 `getURLs` 来获取 URL 的集合。
- 对于集合中的每个 URL，调用 `getCount` 并将条目添加到 `HashMap`。

`getURLs` 所需时间与包含检索词的网址数成正比。对于罕见的检索词，这可能是一个很小的数字，但是对于常见检索词，它可能和 `N` 一样大。

在循环中，我们调用了 `getCount`，它在 Redis 上寻找 `TermCounter`，查找一个检索词，并向 `HashMap` 添加一个条目。那些都是常数时间的操作，所以在最坏的情况下，`getCounts` 的整体复杂度是  $O(N)$ 。然而实际上，运行时间正比于包含检索词的页面数量，通常比 `N` 小得多。

这个算法根据复杂性是有效的，但是它非常慢，因为它向 Redis 发送了许多较小的操作。你可以使用 `Transaction` 来加快速度。你可能留作一个练习，或者你可以在 `RedisIndex.java` 中查看我的解决方案。

## 15.3 索引的分析

使用我们设计的数据结构，页面的索引需要多长时间？再次考虑你的答案，然后再继续。

为了索引页面，我们遍历其 DOM 树，找到所有 `TextNode` 对象，并将字符串拆分成检索词。这一切都与页面上的单词数成正比。

对于每个检索词，我们在 `HashMap` 中增加一个计数器，这是一个常数时间的操作。所以创建 `TermCounter` 的所需时间与页面上的单词数成正比。

将 `TermCounter` 推送到 Redis，需要删除 `TermCounter`，对于唯一检索词的数量是线性的。那么对于每个检索词，我们必须：

- 向 `URLSet` 添加元素，并且
- 向 Redis `TermCounter` 添加元素。

这两个都是常数时间的操作，所以推送 `TermCounter` 的总时间对于唯一检索词的数量是线性的。

总之，`TermCounter` 的创建与页面上的单词数成正比。向 Redis 推送 `TermCounter` 与唯一检索词的数量成正比。

由于页面上的单词数量通常超过唯一检索词的数量，因此整体复杂度与页面上的单词数成正比。理论上，一个页面可能包含索引中的所有检索词，因此最坏的情况是  $O(M)$ ，但实际上我们并不期待看到更糟糕的情况。

这个分析提出了一种提高效率的方法：我们应该避免索引很常见的词语。首先，他们占用了大量的时



间和空间，因为它们出现在几乎每一个 `URLSet` 和 `TermCounter` 中。此外，它们不是很有用，因为它们不能帮助识别相关页面。

大多数搜索引擎避免索引常用单词，这在本文中称为停止词 (<http://thinkdast.com/stopword>)。

## 15.4 图的遍历

如果你在第七章中完成了“到达哲学”练习，你已经有了一个程序，它读取维基百科页面，找到第一个链接，使用链接加载下一页，然后重复。这个程序是一种专用的爬虫，但是当人们说“网络爬虫”时，他们通常意味着一个程序：

加载起始页面并对内容进行索引，  
查找页面上的所有链接，并将链接的 URL 添加到集合中  
通过收集，加载和索引页面，以及添加新的 URL，来按照它的方式工作。  
如果它找到已经被索引的 URL，会跳过它。

你可以将 Web 视为图，其中每个页面都是一个节点，每个链接都是从一个节点到另一个节点的有向边。如果你不熟悉图，可以阅读 <http://thinkdast.com/graph>。

从源节点开始，爬虫程序遍历该图，访问每个可达节点一次。

我们用于存储 URL 的集合决定了爬虫程序执行哪种遍历：

- 如果它是先进先出 (FIFO) 的队列，则爬虫程序将执行广度优先遍历。
- 如果它是后进先出 (LIFO) 的栈，则爬虫程序将执行深度优先遍历。
- 更通常来说，集合中的条目可能具有优先级。例如，我们可能希望对尚未编入索引的页面给予较高的优先级。

你可以在 <http://thinkdast.com/graphtrav> 上阅读图的遍历的更多信息。

## 15.5 练习 12

现在是时候写爬虫了。在本书的仓库中，你将找到此练习的源文件：

- `WikiCrawler.java`，包含你的爬虫的其实代码。
- `WikiCrawlerTest.java`，包含 `WikiCrawler` 的测试代码。
- `JedisIndex.java`，这是我以前的练习的解决方案。

你还需要一些我们以前练习中使用过的辅助类：

- `JedisMaker.java`
- `WikiFetcher.java`

- `TermCounter.java`
- `WikiNodeIterable.java`

在运行 `JedisMaker` 之前，你必须提供一个文件，关于你的 Redis 服务器信息。如果你在上一个练习中这样做，你应该全部配置好了。否则，你可以在 14.3 节中找到说明。

运行 `ant build` 来编译源文件，然后运行 `ant JedisMaker` 来确保它配置为连接到你的 Redis 服务器。

现在运行 `ant WikiCrawlerTest`。它应该失败，因为你有工作要做！

这是我提供的 `WikiCrawler` 类的起始：

```

1. public class WikiCrawler {
2.
3.     public final String source;
4.     private JedisIndex index;
5.     private Queue<String> queue = new LinkedList<String>();
6.     final static WikiFetcher wf = new WikiFetcher();
7.
8.     public WikiCrawler(String source, JedisIndex index) {
9.         this.source = source;
10.        this.index = index;
11.        queue.offer(source);
12.    }
13.
14.    public int queueSize() {
15.        return queue.size();
16.    }

```

实例变量是：

- `source` 是我们开始抓取的网址。
- `index` 是 `JedisIndex`，结果应该放进这里。
- `queue` 是 `LinkedList`，这里面我们跟踪已发现但尚未编入索引的网址。
- `wf` 是 `WikiFetcher`，我们用来读取和解析网页。

你的工作是填写 `crawl`。这是原型：

```

1. public String crawl(boolean testing) throws IOException {}

```

当这个方法在 `WikiCrawlerTest` 中调用时，`testing` 参数为 `true`，否则为 `false`。

如果 `testing` 是 `true`，`crawl` 方法应该：

- 以 FIFO 的顺序从队列中选择并移除一个 URL。

- 使用 `WikiFetcher.readWikipedia` 读取页面的内容，它读取仓库中包含的，页面的缓存副本来进行测试（如果维基百科的版本更改，则避免出现问题）。
- 它应该索引页面，而不管它们是否已经被编入索引。
- 它应该找到页面上的所有内部链接，并按他们出现的顺序将它们添加到队列中。“内部链接”是指其他维基百科页面的链接。
- 它应该返回其索引的页面的 URL。

如果 `testing` 是 `false`，这个方法应该：

- 以 FIFO 的顺序从队列中选择并移除一个 URL。
- 如果 URL 已经被编入索引，它不应该再次索引，并应该返回 `null`。
- 否则它应该使用 `WikiFetcher.fetchWikipedia` 读取页面内容，从 Web 中读取当前内容。
- 然后，它应该对页面进行索引，将链接添加到队列，并返回其索引的页面的 URL。

`WikiCrawlerTest` 加载具有大约 `200` 个链接的队列，然后调用 `crawl` 三次。每次调用后，它将检查队列的返回值和新长度。

当你的爬虫按规定工作时，此测试应通过。祝你好运！

## 第十六章 布尔搜索

- 第十六章 布尔搜索
  - 16.1 爬虫的答案
  - 16.2 信息检索
  - 16.3 布尔搜索
  - 16.4 练习 13
  - 16.5 Comparable 和 Comparator
  - 16.6 扩展

## 第十六章 布尔搜索

原文: [Chapter 16 Boolean search](#)

译者: 飞龙

协议: [CC BY-NC-SA 4.0](#)

自豪地采用谷歌翻译

在本章中，我展示了上一个练习的解决方案。然后，你将编写代码来组合多个搜索结果，并按照它与检索词的相关性进行排序。

### 16.1 爬虫的答案

首先，我们来解决上一个练习。我提供了一个 `WikiCrawler` 的大纲；你的工作是填写 `crawl`。作为一个提醒，这里是 `WikiCrawler` 类中的字段：

```
1. public class WikiCrawler {
2.     // keeps track of where we started
3.     private final String source;
4.
5.     // the index where the results go
6.     private JedisIndex index;
7.
8.     // queue of URLs to be indexed
9.     private Queue<String> queue = new LinkedList<String>();
10.
11.    // fetcher used to get pages from Wikipedia
12.    final static WikiFetcher wf = new WikiFetcher();
13. }
```

当我们创建 `WikiCrawler` 时，我们传入 `source` 和 `index`。最初 `queue` 只包含一个元

素， `source` 。

注意， `queue` 的实现是 `LinkedList`，所以我们可以从末尾添加元素，并从开头删除它们 - 以常数时间。通过将 `LinkedList` 对象赋给 `Queue` 变量，我们将使用的方法限制在 `Queue` 接口中；具体来说，我们将使用 `offer` 添加元素，以及 `poll` 来删除它们。

这是我的 `WikiCrawler.crawl` 的实现：

```

1.  public String crawl(boolean testing) throws IOException {
2.      if (queue.isEmpty()) {
3.          return null;
4.      }
5.      String url = queue.poll();
6.      System.out.println("Crawling " + url);
7.
8.      if (testing==false && index.isIndexed(url)) {
9.          System.out.println("Already indexed.");
10.         return null;
11.     }
12.
13.     Elements paragraphs;
14.     if (testing) {
15.         paragraphs = wf.readWikipedia(url);
16.     } else {
17.         paragraphs = wf.fetchWikipedia(url);
18.     }
19.     index.indexPage(url, paragraphs);
20.     queueInternalLinks(paragraphs);
21.     return url;
22. }

```

这个方法的大部分复杂性是使其易于测试。这是它的逻辑：

- 如果队列为空，则返回 `null` 来表明它没有索引页面。
- 否则，它将从队列中删除并存储下一个 URL。
- 如果 URL 已经被索引， `crawl` 不会再次对其进行索引，除非它处于测试模式。
- 接下来，它读取页面的内容：如果它处于测试模式，它从文件读取；否则它从 Web 读取。
- 它将页面索引。
- 它解析页面并向队列添加内部链接。
- 最后，它返回索引的页面的 URL。

我在 15.1 节展示了 `Index.indexPage` 的一个实现。所以唯一的新方法是 `WikiCrawler.queueInternalLinks`。

我用不同的参数编写了这个方法的两个版本：一个是 `Elements` 对象，包含每个段落的 DOM 树，另一个是 `Element` 对象，包含大部分段落。

第一个版本只是循环遍历段落。第二个版本是实际的逻辑。

```

1.     void queueInternalLinks(Elements paragraphs) {
2.         for (Element paragraph: paragraphs) {
3.             queueInternalLinks(paragraph);
4.         }
5.     }
6.
7.     private void queueInternalLinks(Element paragraph) {
8.         Elements elts = paragraph.select("a[href]");
9.         for (Element elt: elts) {
10.            String relURL = elt.attr("href");
11.
12.            if (relURL.startsWith("/wiki/")) {
13.                String absURL = elt.attr("abs:href");
14.                queue.offer(absURL);
15.            }
16.        }
17.    }

```

要确定链接是否为“内部”链接，我们检查 URL 是否以 `/wiki/` 开头。这可能包括我们不想索引的一些页面，如有关维基百科的元页面。它可能会排除我们想要的一些页面，例如非英语语言页面的链接。但是，这个简单的测试足以起步了。

这就是它的一切。这个练习没有很多新的材料；这主要是一个机会，把这些作品组装到一起。

## 16.2 信息检索

这个项目的下一个阶段是实现一个搜索工具。我们需要的部分包括：

- 一个界面，其中用户可以提供检索词并查看结果。
- 一种查找机制，它接收每个检索词并返回包含它的页面。
- 用于组合来自多个检索词的搜索结果的机制。
- 对搜索结果打分和排序的算法。

用于这样的过程的通用术语是“信息检索”，你可以在 <http://thinkdast.com/infret> 上阅读更多信息。

在本练习中，我们将重点介绍步骤 3 和 4。我们已经构建了一个 2 的简单的版本。如果你有兴趣构建 Web 应用程序，则可以考虑完成步骤 1。

## 16.3 布尔搜索

大多数搜索引擎可以执行“布尔搜索”，这意味着你可以使用布尔逻辑来组合来自多个检索词的结果。例如：

- 搜索“java + 编程”（加号可省略）可能只返回包含两个检索词：“java”和“编程”的页面。
- “java OR 编程”可能会返回包含任一检索词但不一定同时出现的页面。
- “java -印度尼西亚”可能返回包含“java”，不包含“印度尼西亚”的页面。

包含检索词和运算符的表达式称为“查询”。

当应用给搜索结果时，布尔操作符 `+`，`OR` 和 `-` 对应于集合操作 交，并和差。例如，假设

- `s1` 是包含“java”的页面集，
- `s2` 是包含“编程”的页面集，以及
- `s3` 是包含“印度尼西亚”的页面集。

在这种情况下：

- `s1` 和 `s2` 的交集是含有“java”和“编程”的页面集。
- `s1` 和 `s2` 的并集是含有“java”或“编程”的页面集。
- `s1` 与 `s2` 的差集是含有“java”而不含有“印度尼西亚”的页面集。

在下一节中，你将编写实现这些操作的方法。

## 16.4 练习 13

在本书的仓库中，你将找到此练习的源文件：

- `WikiSearch.java`，它定义了一个对象，包含搜索结果并对其执行操作。
- `WikiSearchTest.java`，它包含 `WikiSearch` 的测试代码。
- `Card.java`，它演示了如何使用 `java.util.Collections` 的 `sort` 方法。

你还将找到我们以前练习中使用过的一些辅助类。

这是 `WikiSearch` 类定义的起始：

```
1. public class WikiSearch {
2.
3.     // map from URLs that contain the term(s) to relevance score
4.     private Map<String, Integer> map;
5.
6.     public WikiSearch(Map<String, Integer> map) {
7.         this.map = map;
8.     }
9. }
```

```

10.     public Integer getRelevance(String url) {
11.         Integer relevance = map.get(url);
12.         return relevance==null ? 0: relevance;
13.     }
14. }

```

`WikiSearch` 对象包含 URL 到它们的相关性分数的映射。在信息检索的上下文中，“相关性分数”用于表示页面多么满足从查询推断出的用户需求。相关性分数的构建有很多种方法，但大部分都基于“检索词频率”，它是搜索词在页面上的显示次数。一种常见的相关性分数称为 TF-IDF，代表“检索词频率 - 逆向文档频率”。你可以在 <http://thinkdast.com/tfidf> 上阅读更多信息。

你可以选择稍后实现 TF-IDF，但是我们将从一些更简单的 TF 开始：

- 如果查询包含单个检索词，页面的相关性就是其词频；也就是说该词在页面上出现的次数。
- 对于具有多个检索词的查询，页面的相关性是检索词频率的总和；也就是说，任何检索词出现的总次数。

现在你准备开始练习了。运行 `ant build` 来编译源文件，然后运行 `ant WikiSearchTest`。像往常一样，它应该失败，因为你有工作要做。

在 `WikiSearch.java` 中，填充的 `and`，`or` 以及 `minus` 的主体，使相关测试通过。你不必担心 `testSort`。

你可以运行 `WikiSearchTest` 而不使用 `Jedis`，因为它不依赖于 Redis 数据库中的索引。但是，如果要对索引运行查询，则必须向文件提供有关 `Redis` 服务器的信息。详见 14.3 节。

运行 `ant JedisMaker` 来确保它配置为连接到你的 Redis 服务器。然后运行 `WikiSearch`，它打印来自三个查询的结果：

- “java”
- “programming”
- “java AND programming”

最初的结果不按照特定的顺序，因为 `WikiSearch.sort` 是不完整的。

填充 `sort` 的主体，使结果以递增的相关顺序返回。我建议你使用 `java.util.Collections` 提供的 `sort` 方法，它可以排序任何种类的 `List`。你可以阅读 <http://thinkdast.com/collections> 上的文档。

有两个 `sort` 版本：

- 单参数版本接受列表并使用它的 `compareTo` 方法对元素进行排序，因此元素必须是 `Comparable`。
- 双参数版本接受任何对象类型的列表和一个 `Comparator`，它是一个提供 `compare` 方法的对象，用于比较元素。

如果你不熟悉 `Comparable` 和 `Comparator` 接口，我将在下一节中解释它们。



## 16.5 Comparable 和 Comparator

本书的仓库包含了 `Card.java`，它演示了两个方式来排序 `Card` 对象的列表。这里是类定义的起始：

```
1. public class Card implements Comparable<Card> {
2.
3.     private final int rank;
4.     private final int suit;
5.
6.     public Card(int rank, int suit) {
7.         this.rank = rank;
8.         this.suit = suit;
9.     }
```

`Card` 对象拥有两个整形字段，`rank` 和 `suit`。`Card` 实现了 `Comparable<Card>`，也就是说它提供 `compareTo`：

```
1.     public int compareTo(Card that) {
2.         if (this.suit < that.suit) {
3.             return -1;
4.         }
5.         if (this.suit > that.suit) {
6.             return 1;
7.         }
8.         if (this.rank < that.rank) {
9.             return -1;
10.        }
11.        if (this.rank > that.rank) {
12.            return 1;
13.        }
14.        return 0;
15.    }
```

`compareTo` 规范表明，如果 `this` 小于 `that`，则应该返回一个负数，如果它更大，则为正数，如果它们相等则为 `0`。

如果使用单参数版本的 `Collections.sort`，它将使用元素提供的 `compareTo` 方法对它们进行排序。为了演示，我们可以列出 `52` 张卡，如下所示：

```
1.     public static List<Card> makeDeck() {
2.         List<Card> cards = new ArrayList<Card>();
3.         for (int suit = 0; suit <= 3; suit++) {
4.             for (int rank = 1; rank <= 13; rank++) {
```

```

5.         Card card = new Card(rank, suit);
6.         cards.add(card);
7.     }
8. }
9. return cards;
10. }

```

并这样排序它们：

```

1. Collections.sort(cards);

```

这个版本的 `sort` 将元素按照所谓的“自然秩序”放置，因为它由对象本身决定。

但是可以通过提供一个 `Comparator` 对象，来强制实现不同的排序。例如，`Card` 对象的自然顺序将 `Ace` 视为最小的牌，但在某些纸牌游戏中，它的排名最高。我们可以定义一个 `Comparator`，将 `Ace` 视为最大的牌，像这样：

```

1.     Comparator<Card> comparator = new Comparator<Card>() {
2.         @Override
3.         public int compare(Card card1, Card card2) {
4.             if (card1.getSuit() < card2.getSuit()) {
5.                 return -1;
6.             }
7.             if (card1.getSuit() > card2.getSuit()) {
8.                 return 1;
9.             }
10.            int rank1 = getRankAceHigh(card1);
11.            int rank2 = getRankAceHigh(card2);
12.
13.            if (rank1 < rank2) {
14.                return -1;
15.            }
16.            if (rank1 > rank2) {
17.                return 1;
18.            }
19.            return 0;
20.        }
21.
22.        private int getRankAceHigh(Card card) {
23.            int rank = card.getRank();
24.            if (rank == 1) {
25.                return 14;
26.            } else {
27.                return rank;
28.            }
29.        }

```

```
30.         };
```

该代码定义了一个匿名类，按需实现 `compare`。然后它创建一个新定义的匿名类的实例。如果你不熟悉 Java 中的匿名类，可以在 <http://thinkdast.com/anonclass> 上阅读它们。

使用这个 `Comparator`，我们可以这样调用 `sort`：

```
1. Collections.sort(cards, comparator);
```

在这个顺序中，黑桃的 `Ace` 是牌组上的最大的牌；梅花二是最小的。

如果你想试验这个部分的代码，它们在 `Card.java` 中。作为一个练习，你可能打算写一个比较器，先按照 `rank`，然后再按照 `suit`，所以所有的 `Ace` 都应该在一起，所有的二也是。以此类推。

## 16.6 扩展

如果你完成了此练习的基本版本，你可能需要处理这些可选练习：

- 请阅读 <http://thinkdast.com/tfidf> 上的 TF-IDF，并实现它。你可能需要修改 `JavaIndex` 来计算文档频率；也就是说，每个检索词在索引的所有页面上出现的总次数。
- 对于具有多个检索词的查询，每个页面的总体相关性目前是每个检索词的相关性的总和。想想这个简单版本什么时候可能无法正常运行，并尝试一些替代方案。
- 构建用户界面，允许用户输入带有布尔运算符的查询。解析查询，生成结果，然后按相关性排序，并显示评分最高的 URL。考虑生成“片段”，它显示了检索词出现在页面的哪里。如果要为用户界面制作 Web 应用程序，请考虑将 Heroku 作为简单选项，用于开发和部署 Java Web 应用程序。见 <http://thinkdast.com/heroku>。

## 第十七章 排序

- [第十七章 排序](#)
  - [17.1 插入排序](#)
  - [17.2 练习 14](#)
  - [17.3 归并排序的分析](#)
  - [基数排序](#)
  - [17.5 堆排序](#)
  - [17.6 有界堆排序](#)
  - [17.7 空间复杂性](#)

## 第十七章 排序

原文: [Chapter 17 Sorting](#)

译者: [飞龙](#)

协议: [CC BY-NC-SA 4.0](#)

自豪地采用[谷歌翻译](#)

计算机科学领域过度痴迷于排序算法。根据 CS 学生在这个主题上花费的时间，你会认为排序算法的选择是现代软件工程的基石。当然，现实是，软件开发人员可以在很多年中，或者整个职业生涯中，不必考虑排序如何工作。对于几乎所有的应用程序，它们都使用它们使用的语言或库提供的通用算法。通常这样就行了。

所以如果你跳过这一章，不了解排序算法，你仍然是一个优秀的开发人员。但是有一些原因你可能想要这样：

- 尽管有绝大多数应用程序都可以使用通用算法，但你可能需要了解两种专用算法：基数排序和有界堆排序。
- 一种排序算法，归并排序，是一个很好的教学示例，因为它演示了一个重要和实用的算法设计策略，称为“分治”。此外，当我们分析其表现时，你将了解到我们以前没有看到的生长级别，即线性对数。最后，一些最广泛使用的算法是包含归并排序的混合体。
- 了解排序算法的另一个原因是，技术面试官喜欢询问它们。如果你想要工作，如果你能展示 CS 文化素养，就有帮助。

因此，在本章中我们将分析插入排序，你将实现归并排序，我将给你讲解基数排序，你将编写有界堆排序的简单版本。

### 17.1 插入排序

我们将从插入排序开始，主要是因为它的描述和实现很简单。它不是很有效，但它有一些补救的特性，我们将看到它。

我们不在这里解释算法，建议你阅读 <http://thinkdast.com/insertsort> 中的插入排序的维基百科页面，其中包括伪代码和动画示例。当你理解了它的思路再回来。

这是 Java 中插入排序的实现：

```

1. public class ListSorter<T> {
2.
3.     public void insertionSort(List<T> list, Comparator<T> comparator) {
4.
5.         for (int i=1; i < list.size(); i++) {
6.             T elt_i = list.get(i);
7.             int j = i;
8.             while (j > 0) {
9.                 T elt_j = list.get(j-1);
10.                 if (comparator.compare(elt_i, elt_j) >= 0) {
11.                     break;
12.                 }
13.                 list.set(j, elt_j);
14.                 j--;
15.             }
16.             list.set(j, elt_i);
17.         }
18.     }
19. }
```

我定义了一个类，`ListSorter` 作为排序算法的容器。通过使用类型参数 `T`，我们可以编写一个方法，它在包含任何对象类型的列表上工作。

`insertionSort` 需要两个参数，一个是任何类型的 `List`，一个是 `Comparator`，它知道如何比较类型 `T` 的对象。它对列表“原地”排序，这意味着它修改现有列表，不必分配任何新空间。

下面的示例演示了，如何使用 `Integer` 的 `List` 对象，调用此方法：

```

1.     List<Integer> list = new ArrayList<Integer>()
2.         Arrays.asList(3, 5, 1, 4, 2));
3.
4.     Comparator<Integer> comparator = new Comparator<Integer>() {
5.         @Override
6.         public int compare(Integer elt1, Integer elt2) {
7.             return elt1.compareTo(elt2);
8.         }
9.     };
10.
```

```

11.         ListSorter<Integer> sorter = new ListSorter<Integer>();
12.         sorter.insertionSort(list, comparator);
13.         System.out.println(list);

```

`insertionSort` 有两个嵌套循环，所以你可能会猜到，它的运行时间是二次的。在这种情况下，一般是正确的，但你做出这个结论之前，你必须检查，每个循环的运行次数与 `n`，数组的大小成正比。

外部循环从 `1` 迭代到 `list.size()`，因此对于列表的大小 `n` 是线性的。内循环从 `i` 迭代到 `0`，所以在 `n` 中也是线性的。因此，两个循环运行的总次数是二次的。

如果你不确定，这里是证明：

第一次循环中，`i = 1`，内循环最多运行一次。

第二次，`i = 2`，内循环最多运行两次。

最后一次，`i = n - 1`，内循环最多运行 `n` 次。

因此，内循环运行的总次数是序列 `1, 2, ..., n - 1` 的和，即 `n(n - 1)/2`。该表达式的主项（拥有最高指数）为 `n^2`。

在最坏的情况下，插入排序是二次的。然而：

- 如果这些元素已经有序，或者几乎这样，插入排序是线性的。具体来说，如果每个元素距离它的有序位置不超过 `k` 个元素，则内部循环不会运行超过 `k` 次，并且总运行时间是 `O(kn)`。
- 由于实现简单，开销较低；也就是，尽管运行时间是 `an^2`，主项的系数 `a`，也可能是小的。

所以如果我们知道数组几乎是有序的，或者不是很大，插入排序可能是一个不错的选择。但是对于大数组，我们可以做得更好。其实要好很多。

## 17.2 练习 14

归并排序是运行时间优于二次的几种算法之一。同样，不在这里解释算法，我建议你阅读维基百科 <http://thinkdast.com/mergesort>。一旦你有了想法，反回来，你可以通过写一个实现来测试你的理解。

在本书的仓库中，你将找到此练习的源文件：

- `ListSorter.java`
- `ListSorterTest.java`

运行 `ant build` 来编译源文件，然后运行 `ant ListSorterTest`。像往常一样，它应该失败，因为你有工作要做。

在 `ListSorter.java` 中，我提供了两个方法的大纲，`mergeSortInPlace` 以及 `mergeSort`：

```

1.     public void mergeSortInPlace(List<T> list, Comparator<T> comparator) {

```

```

2.         List<T> sorted = mergeSortHelper(list, comparator);
3.         list.clear();
4.         list.addAll(sorted);
5.     }
6.
7.     private List<T> mergeSort(List<T> list, Comparator<T> comparator) {
8.         // TODO: fill this in!
9.         return null;
10.    }

```

这两种方法做同样的事情，但提供不同的接口。`mergeSort` 获取一个列表，并返回一个新列表，具有升序排列的相同元素。`mergeSortInPlace` 是修改现有列表的 `void` 方法。

你的工作是填充 `mergeSort`。在编写完全递归版本的合并排序之前，首先要这样：

- 将列表分成两半。
- 使用 `Collections.sort` 或 `insertionSort` 来排序这两部分。
- 将有序的两部分合并为一个完整的有序列表中。

这将给你一个机会来调试用于合并的代码，而无需处理递归方法的复杂性。

接下来，添加一个边界情况（请参阅 < <http://thinkdast.com/basecase> >）。如果你只提供一个列表，仅包含一个元素，则可以立即返回，因为它已经有序。或者如果列表的长度低于某个阈值，则可以使用 `Collections.sort` 或 `insertionSort`。在进行前测试边界情况。

最后，修改你的解决方案，使其进行两次递归调用来排序数组的两个部分。当你使其正常工作，`testMergeSort` 和 `testMergeSortInPlace` 应该通过。

## 17.3 归并排序的分析

为了对归并排序的运行时间进行划分，对递归层级和每个层级上完成多少工作方面进行思考，是很有帮助的。假设我们从包含 `n` 个元素的列表开始。以下是算法的步骤：

- 生成两个新数组，并将一半元素复制到每个数组中。
- 排序两个数组。
- 合并两个数组。

图 17.1 显示了这些步骤。

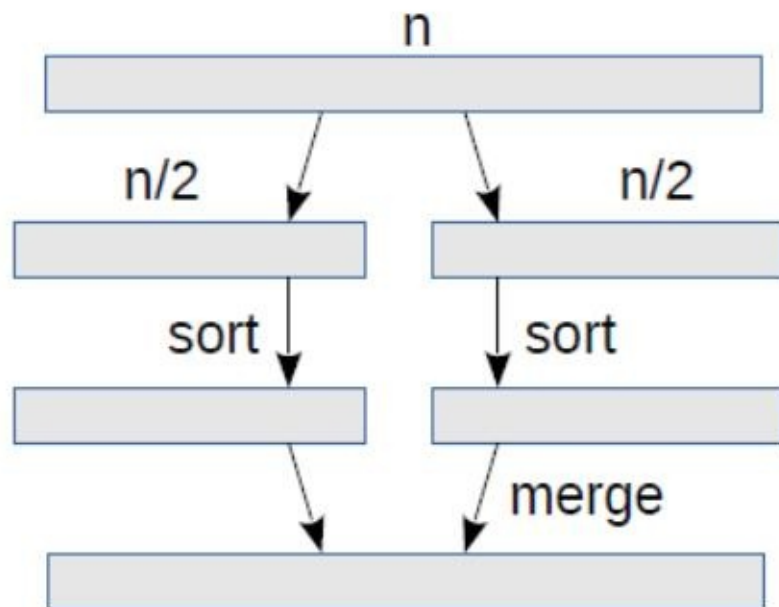


图 17.1: 归并排序的展示，它展示了递归的一个层级。

第一步复制每个元素一次，因此它是线性的。第三步也复制每个元素一次，因此它也是线性的。现在我们需要弄清楚步骤 2 的复杂性。为了做到这一点，查看不同的计算图片会有帮助，它展示了递归的层数，如图 17.2 所示。

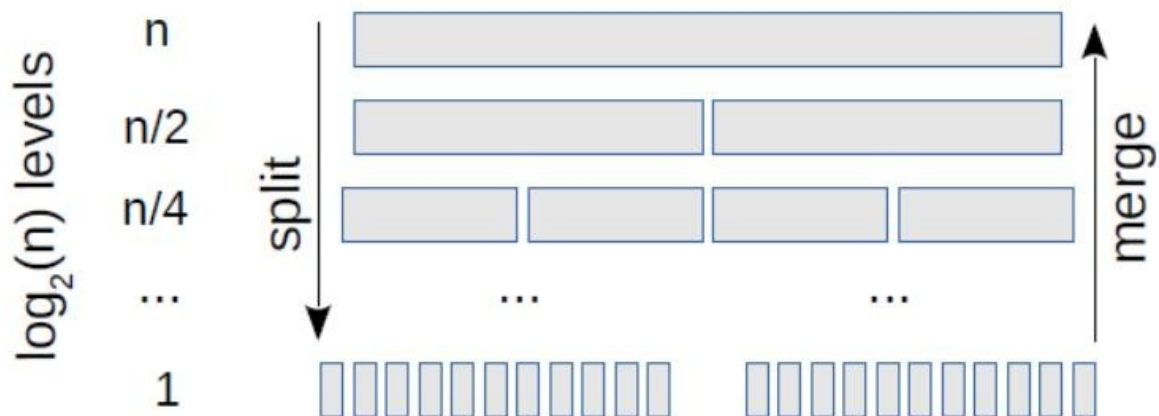


图 17.2: 归并排序的展示，它展示了递归的所有层级。

在顶层，我们有 1 个列表，其中包含  $n$  个元素。为了简单起见，我们假设  $n$  是 2 的幂。在下一层，有 2 个列表包含  $n/2$  个元素。然后是 4 个列表与  $n/4$  元素，以此类推，直到我们得到  $n$  个列表与 1 元素。

在每一层，我们共有  $n$  个元素。在下降的过程中，我们必须将数组分成两半，这在每一层上都需要



与  $n$  成正比的时间。在回来的路上，我们必须合并  $n$  个元素，这也是线性的。

如果层数为  $h$ ，算法的总工作量为  $O(nh)$ 。那么有多少层呢？有两种方法可以考虑：

- 我们用多少步，可以将  $n$  减半直到  $1$ ？
- 或者，我们用多少步，可以将  $1$  加倍直到  $n$ ？

第二个问题的另一种形式是“ $2$  的多少次方是  $n$ ”？

$$1. \quad 2^h = n$$

对两边取以  $2$  为底的对数：

$$1. \quad h = \log_2(n)$$

所以总时间是  $O(n \log n)$ 。我没有纠结于对数的底，因为底不同的对数差别在于一个常数，所以所有的对数都是相同的生长级别。

$O(n \log n)$  中的算法有时被称为“线性对数”的，但大多数人只是说  $n \log n$ 。

事实证明， $O(n \log n)$  是通过元素比较的排序算法的理论下限。这意味着没有任何“比较排序”的生长级别比  $n \log n$  好。请参见 <http://thinkdast.com/compsort>。

但是我们将在下一节中看到，存在线性时间的非比较排序！

## 基数排序

在 2008 年美国总统竞选期间，候选人巴拉克·奥巴马在访问 Google 时，被要求进行即兴算法分析。首席执行官埃里克·施密特开玩笑地问他，“排序一百万个 32 位整数的最有效的方法”。显然有人暗中告诉了奥巴马，因为他很快就回答说：“我认为冒泡排序是错误的。”你可以在 <http://thinkdast.com/obama> 观看视频。

奥巴马是对的：冒泡排序在概念上是简单的，但其运行时间是二次的；即使在二次排序算法中，其性能也不是很好。见 <http://thinkdast.com/bubble>。

施密特想要的答案可能是“基数排序”，这是一种非比较排序算法，如果元素的大小是有界的，例如 32 位整数或 20 个字符的字符串，它就可以工作。

为了看看它是如何工作的，想象你有一堆索引卡，每张卡片包含三个字母的单词。以下是一个方法，可以对卡进行排序：

- 根据第一个字母，将卡片放入桶中。所以以  $a$  开头的单词应该在一个桶中，其次是以  $b$  开头的单词，以此类推
- 根据第二个字母再次将卡片放入每个桶。所以以  $aa$  开头的应该在一起，其次是以  $ab$  开头的，

以此类推当然，并不是所有的桶都是满的，但是没关系。

- 根据第三个字母再次将卡片放入每个桶。

此时，每个桶包含一个元素，桶按升序排列。图 17.3 展示了三个字母的例子。

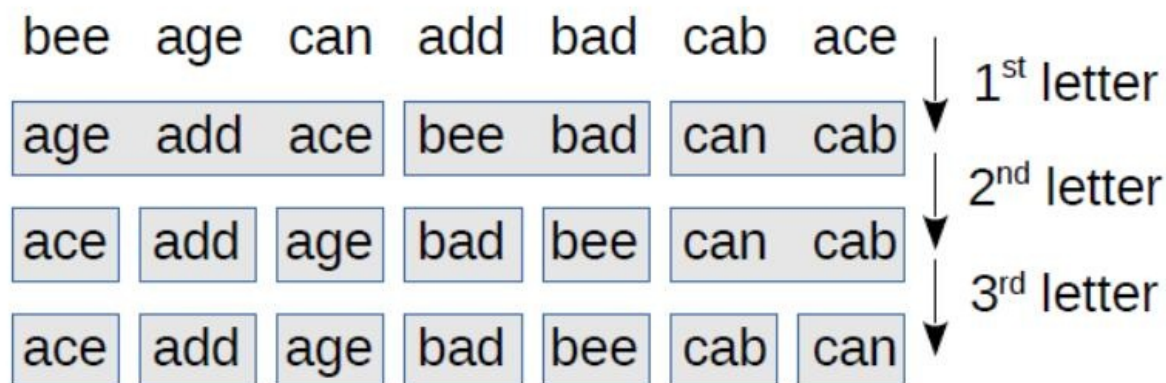


图 17.3：三个字母的基数排序的例子

最上面那行显示未排序的单词。第二行显示第一次遍历后的桶的样子。每个桶中的单词都以相同的字母开头。

第二遍之后，每个桶中的单词以相同的两个字母开头。在第三遍之后，每个桶中只能有一个单词，并且桶是有序的。

在每次遍历期间，我们遍历元素并将它们添加到桶中。只要桶允许在恒定时间内添加元素，每次遍历是线性的。

遍历数量，我会称之为  $w$ ，取决于单词的“宽度”，但不取决于单词的数量， $n$ 。所以增长级别是  $O(wn)$ ，对于  $n$  是线性的。

基数排序有许多变体，并有许多方法来实现每一个。你可以在 <http://thinkdast.com/radix> 上阅读他们的更多信息。作为一个可选的练习，请考虑编写基数排序的一个版本。

## 17.5 堆排序

基数排序适用于大小有界的东西，除了他之外，还有一种你可能遇到的其它专用排序算法：有界堆排序。如果你在处理非常大的数据集，你想要得到前 10 个或者前  $k$  个元素，其中  $k$  远小于  $n$ ，它是很有用的。

例如，假设你正在监视一个 Web 服务，它每天处理十亿次事务。在每一天结束时，你要汇报最大的  $k$  个事务（或最慢的，或者其它最 xx 的）。一个选项是存储所有事务，在一天结束时对它们进行排序，然后选择最大的  $k$  个。需要的时间与  $n \log n$  成正比，这非常慢，因为我们可能无法将十亿次交易记录在单个程序的内存中。我们必须使用“外部”排序算法。你可以在

<http://thinkdast.com/extsort> 上了解外部排序。

使用有界堆，我们可以做得更好！以下是我们的实现方式：

- 我会解释（无界）堆排序。
- 你会实现它
- 我将解释有界堆排序并进行分析。

要了解堆排序，你必须了解堆，这是一个类似于二叉搜索树（BST）的数据结构。有一些区别：

- 在 BST 中，每个节点 `x` 都有“BST 特性”：`x` 左子树中的所有节点都小于 `x`，右子树中的所有节点都大于 `x`。
- 在堆中，每个节点 `x` 都有“堆特性”：两个子树中的所有节点都大于 `x`。
- 堆就像平衡的 BST；当你添加或删除元素时，他们会做一些额外的工作来重新使树平衡。因此，可以使用元素的数组来有效地实现它们。

译者注：这里先讨论最小堆。如果子树中所有节点都小于 `x`，那么就是最大堆。

堆中最小的元素总是在根节点，所以我们可以常在常数时间内找到它。在堆中添加和删除元素需要的时间与树的高度 `h` 成正比。而且由于堆总是平衡的，所以 `h` 与 `log n` 成正比。你可以在 <http://thinkdast.com/heap> 上阅读更多堆的信息。

Java `PriorityQueue` 使用堆实现。`PriorityQueue` 提供 `Queue` 接口中指定的方法，包括 `offer` 和 `poll`：

- `offer`：将一个元素添加到队列中，更新堆，使每个节点都具有“堆特性”。需要 `log n` 的时间。
- `poll`：从根节点中删除队列中的最小元素，并更新堆。需要 `log n` 的时间。

给定一个 `PriorityQueue`，你可以像这样轻松地排序的 `n` 个元素的集合：

- 使用 `offer`，将集合的所有元素添加到 `PriorityQueue`。
- 使用 `poll` 从队列中删除元素并将其添加到 `List`。

因为 `poll` 返回队列中剩余的最小元素，所以元素按升序添加到 `List`。这种排序方式称为堆排序（请参阅 <http://thinkdast.com/heapsort>）。

向队列中添加 `n` 个元素需要 `n log n` 的时间。删除 `n` 个元素也是如此。所以堆排序的运行时间是 `O(n log n)`。

在本书的仓库中，你可以在 `ListSorter.java` 中找到 `heapSort` 方法的大纲。填充它，然后运行 `ant ListSorterTest` 来确认它可以工作。

## 17.6 有界堆排序

有界堆是一个限制为最多包含 `k` 个元素的堆。如果你有 `n` 个元素，你可以跟踪这个最大的 `k` 个

元素：

最初堆是空的。对于每个元素  $x$ ：

- 分支 1：如果堆不满，请添加  $x$  到堆中。
- 分支 2：如果堆满了，请与堆中  $x$  的最小元素进行比较。如果  $x$  较小，它不能是最大的  $k$  个元素之一，所以你可以丢弃它。
- 分支 3：如果堆满了，并且  $x$  大于堆中的最小元素，请从堆中删除最小的元素并添加  $x$ 。

使用顶部为最小元素的堆，我们可以跟踪最大的  $k$  个元素。我们来分析这个算法的性能。对于每个元素，我们执行以下操作之一：

- 分支 1：将元素添加到堆是  $O(\log k)$ 。
- 分支 2：找到堆中最小的元素是  $O(1)$ 。
- 分支 3：删除最小元素是  $O(\log k)$ 。添加  $x$  也是  $O(\log k)$ 。

在最坏的情况下，如果元素按升序出现，我们总是执行分支 3。在这种情况下，处理  $n$  个元素的总时间是  $O(n \log k)$ ，对于  $n$  是线性的。

在 `ListSorter.java` 中，你会发现一个叫做 `topK` 的方法的大纲，它接受一个 `List`、`Comparator` 和一个整数  $k$ 。它应该按升序返回 `List` 的  $k$  个最大的元素。填充它，然后运行 `ant ListSorterTest` 来确认它可以工作。

## 17.7 空间复杂性

到目前为止，我们已经谈到了很多运行时间的分析，但是对于许多算法，我们也关心空间。例如，归并排序的一个缺点是它会复制数据。在我们的实现中，它分配的空间总量是  $O(n \log n)$ 。通过更机智的实现，你可以将空间要求降至  $O(n)$ 。

相比之下，插入排序不会复制数据，因为它会原地排序元素。它使用临时变量来一次性比较两个元素，并使用一些其它局部变量。但它的空间使用不取决于  $n$ 。

我们的堆排序实现创建了新 `PriorityQueue`，来存储元素，所以空间是  $O(n)$ ；但是如果你能够原地对列表排序，则可以使用  $O(1)$  的空间执行堆排序。

刚刚实现的有界堆栈算法的一个好处是，它只需要与  $k$  成正比的空间（我们要保留的元素的数量），而  $k$  通常比  $n$  小得多。

软件开发人员往往比空间更加注重运行时间，对于许多应用程序来说，这是适当的。但是对于大型数据集，空间可能同等或更加重要。例如：

- 如果一个数据集不能放入一个程序的内存，那么运行时间通常会大大增加，或者根本不能运行。如果你选择一个需要较少空间的算法，并且这样可以将计算放入内存中，则可能会运行得更快。同样，使用较少空间的程序，可能会更好地利用 CPU 缓存并运行速度更快（请参阅

<http://thinkdast.com/cache>)。

- 在同时运行多个程序的服务器上，如果可以减少每个程序所需的空间，则可以在同一台服务器上运行更多程序，从而降低硬件和能源成本。

所以这些是一些原因，你应该至少了解一些算法的空间需求。